Journal of Cases on Information Technology
Volume 22 • Issue 3 • July-September 2020

A Game Theoretic Approach for Quality Assurance in Software Systems Using Antifragility-Based Learning Hooks

Vimaladevi M., Pondicherry Engineering College, India
Zayaraz G., Pondicherry Engineering College, India

ABSTRACT

The use of software in mission critical applications poses greater quality needs. Quality assurance activities are aimed at ensuring such quality requirements of the software system. Antifragility is a property of software that increases its quality as a result of errors, faults, and attacks. Such antifragile software systems proactively accepts the errors and learns from these errors and relies on test-driven development methodology. In this article, an innovative approach is proposed which uses a fault injection methodology to perform the task of quality assurance. Such a fault injection mechanism makes the software antifragile and it gets better with the increase in the intensity of such errors up to a point. A software quality game is designed as a two-player game model with stressor and backer entities. The stressor is an error model which injects errors into the software system. The software system acts as a backer, and tries to recover from the errors. The backer uses a cheating mechanism by implementing software Learning Hooks (SLH) which learn from the injected errors. This makes the software antifragile and leads to improvement of the code. Moreover, the SLH uses a Q-Learning reinforcement algorithm with a hybrid reward function to learn from the incoming defects. The game is played for a maximum of K errors. This approach is introduced to incorporate the anti-fragility aspects into the software system within the existing framework of object-oriented development. The game is run at the end of every increment during the construction of object-oriented systems. A detailed report of the injected errors and the actions taken is output at the end of each increment so that necessary actions are incorporated into the actual software during the next iteration. This ensures at the end of all the iterations, the software is immune to majority of the so-called Black Swans. The experiment is conducted with an open source Java sample and the results are studied selected two categories of evaluation parameters. The defect related performance parameters considered are the defect density, defect distribution over different iterations, and number of hooks inserted. These parameters show much reduction in adopting the proposed approach. The quality parameters such as abstraction, inheritance, and coupling are studied for various iterations and this approach ensures considerable increases in these parameters.

KEYWORDS

Antifragility, Defect Metrics, Fault Injection, Game Theory, Graph Metrics, Hooking Methods, Q-Learning, Quality Attributes

DOI: 10.4018/JCIT.2020070101

1. INTRODUCTION

Software applications are becoming more complex day by day and it is difficult to maintain code quality and manage the cost of the software development. Some of the factors that make this quality-cost balance a challenging task needs further discussion. They are the growing pressure on the software organizations, rise of the developmental costs, need to get the product to market quickly and accelerated development schedules. The most effective way to keep the development cost down is the minimization and the introduction of defects. The software bug cost of United States economy has increased from $59.5 billion to $1.1 trillion from 2002 to 2016. This increase in cost is due to the loss in revenue due to the software being unusable, payments to developers for bug fixing, loss in shareholder value, etc. Also, there are some indirect financial costs arising due to the problem of brand reputation and customer loyalty. The bug fixing process even interferes with other developments and enhancements for new functionality addition that ultimately affect the project schedule. It is critical to catch the defects early since, the cost of fixing the defects increases exponentially as the software progresses through the life cycle phases. From the report of National Institute of Standards and Technology (NIST), the increase in the bug fix follows the trend as shown in Table 1 (National Institute of Standards and Technology, 2002). Here, X is the normalized unit of cost and can be expressed in terms of person-hours.

Table 1. Cost of defect fixing

Design	1X
Implementation	5 X
Integration Testing	10 X
Customer Beta Testing	15 X
Post Product Release	30 X

Hence, there is an important need for proactive approaches to improve the overall quality and decrease the software development cost. This research work discusses such an arrangement to proactively detect defects by building antifragile characteristics into an object-oriented software within the existing software development framework. But this defect prevention is a challenging task. The operating environment and the kinds of failure and recovery of a software system are highly uncertain and are open ended. For example, an information report states that the Eclipse development environment runs on at least 5 million different machines. The developer foreseeing all possible failures is nearly impossible, so as the hard coding of all possible failure conditions, recovery conditions and recovery strategies. The programming style needs to be changed in order to overcome these unforeseen errors or surprises, called as Black swans (Taleb, 2008). The software systems are constantly exposed to randomness, shocks, disorders and stressors. There are lots of works that have come over the decade to ensure quality in critical software systems that make majority of the systems resilient and robust. But with much of the evolution and advancements in the software programming and the development methodologies, it is time to take the art of software development to the next level. The property called *Antifragility*, explained by Taleb in (Taleb, 2012), is explored in this work, which takes the software to one more step ahead in addition to resilience and robustness. A resilient resist the shock and stays the same; whereas the antifragile resists and gets better with such disorders. The antifragile loves disorder and learns from it to improve itself. This requirement poses a lot of challenges in programming such a system, which can learn from failures. Also, it is much better to incorporate this property into the software within the existing successful developmental model such

as the Agile methods. This is preferred instead of designing a new process model, since these models are almost mature with its processes and standards in place. Thus, the research question RQ1 can be formulated as:

RQ1: How the "Antifragility" aspect can be introduced in software development within the existing Object-Oriented framework?

A game is a structured form of play played for an achievement or reward or sometimes purely for entertainment. A game can be used as a design tool in research methods to achieve certain goals. Game Theory applies mathematical models to study the strategies the players can make in a gaming environment. It is popularly used in fields such as economics, politics, and biology and in computer science. In computer science, game theory is widely used in Cloud Computing, network security, recommendation systems, and machine learning. Even though it has already been successfully applied in fields such as economics, the application of Game Theory to Software engineering problems is relatively new. The games are of different types such as two-player and N-player games, cooperative and non-cooperative games, simultaneous and sequential games, constant sum, zero sum, and non-zero sum games, etc. Considering the aim of making software antifragile, a two-player game theoretic approach is proposed which makes the software to learn from errors that are injected into the software system. Thus, the research question RQ2 is put forth as:

RQ2: How successfully the concepts of "Game Theory" can be used to build an Antifragile Object Oriented Software?

In order to upgrade to antifragility and achieve the strengths of game theory, a two-player game model is designed with a Stressor and Backer entities, which act as the players of the game. The stressor is interested in introducing errors into the system, whereas the backer defends those and learns from the introduced errors. The Stressor uses the architectural information of the software and the graph metrics to generate crucial defects. The Backer accomplishes its tasks by executing software Learning Hooks (SLH) which implements a Q-Learning algorithm. Even though, this type of approach can be considered one of the test-driven methodologies, it is different from the traditional testing phase, in which the testing is performed to validate specific, well-formed and small failures. Usually these types of antifragile stressors are designed to be incorporated in the production environment. But, in the proposed approach, they are introduced during development phase of object-oriented software, before the release of every increment. The game is run for all the iterations and the report of the errors introduced and actions taken are output for every run. This is helpful to plan for the next iteration. In case of any modification required in the code the developer reviews and makes decision before any changes is applied to the code. This approach is followed foreseeing the risk whether the system can fully recover from such disturbances and this method can outperform the losses. This proposal thus aims in introducing Antifragility within the existing framework of object-oriented software development. The Antifragile Software Manifesto (Russoa & Ciancarinia, 2016), is still in the inception, invites discussion from the software engineering community for its successful application.

To test the constructive benefits of the research questions RQ1 and RQ2, the proposal need to be validated for a sample project for increase in quality and reduction in the defects. Hence, the parameters need to be validated is twofold. The increase in quality need to be tested using major quality attributes namely abstraction, inheritance, and coupling. The important defect related parameters are defect density, defect distribution and number of hooks inserted per iteration. A considerable decrease in the defect related parameters is favored.

The rest of the article is organized as follows: Section 2 presents the motivation and related work. Section 3 gives the details of the proposed game structure. Section 4 furnishes the details

of the simulation, results obtained and discussion on the results. Section 5 lists the contributions of this research work and Section 6 lists various threats to the validity of the work. Section 7 concludes the paper.

2. MOTIVATION

There are numerous techniques that are available for quality improvement in various stages of the software life cycle. Any quality assurance technique strives to achieve zero errors post release. In spite of all these constant and effective techniques, there are still some failures in the software that makes the software difficult to survive. There are various factors that make the software obsolete such as changes in the requirements and the operating environment, inefficient calculations and data, difficulty in using, understanding and maintaining the software, ageing of the software, market conditions, etc. (Leach, 2016). There are countless research techniques and ideas that are proposed to overcome these issues to some extent. Every technique has its own advantages and disadvantages. Irrespective of all these efforts, failures still exist in software products which are difficult to predict and avoid completely. Hence, there is a need to find an intelligent solution to counter and cope with these errors. In order to enable the software live with these uncertainties, an innovative methodology needs to be designed, which reinvents the process of software development and its usage.

Antifragility is such an emerging issue in software engineering which is beyond building robust and resilient systems. A system which is robust is resilient to errors to a certain threshold but will remain the same. An antifragile system is one, in addition to being robust, tries to improve when exposed to errors. The Antifragile Software Manifesto proposed in (Russoa & Ciancarinia, 2016) discusses some of the principles and the processes of an antifragile system. This research work is a step towards incorporating the concepts of antifragility into the existing framework of Object-Oriented Software Development. In this work, Anti-Fragility is also proposed as an Anti-Ageing solution to software systems. Fragile, Robust and Antifragile are defined as a triad in explaining the desirable properties of software. A fragile is one which is easily breakable to any disturbance. A robust system resists and withstands such shocks to some extent, but it remains the same. Antifragility is beyond resilience or robustness. The concepts of an Antifragile System are explained meticulously in (Taleb, 2012).

Antifragility is the negative of fragility. An antifragile system gets better by exposure to disorder, shock or uncertainties. An antifragile system is able to evolve its identity by learning from the disorder and by improving itself.

Game theory is becoming popular in the decision-making process in a cooperative or a competitive environment. It has been successfully applied in computer science in network security, scheduling problems, robotics, and interactions in cloud environment and in project management process to arrive at an optimal decision making. However, so far, its application is less in the field of software engineering and it is being widely studied and applied in solving optimization problems. In (Huang, Peled, schewe, & Wang, 2016), a game theoretic approach is proposed in order to validate the resilience of a software system against k dense errors. The authors have designed a two-player concurrent game model with the application of alternating-time μ-calculus (AMC) with an extension. An AMC is a special type of labeled transition systems, called concurrent game structures, where each transition is a result of set of decisions a player takes. The analysis has been modeled as a model checking problem for the software to be resilient to utmost k dense errors.

In (Kumar & Prabhakar, 2009), the authors applied game theory to find an optimal solution for software architecture design with conflicting quality attributes by applying the concepts of Nash equilibrium and Pareto optimality. Mehdi, Raza, & Hussain (2017), presents a game theory based trust model for VANETs by designing a two player attacker and defender model. The outcome of the game is presented using a payoff matrix and Nash equilibrium (NE) is applied to arrive at an optimal strategy for the attacker-defender scenario. Nash Equilibrium is a solution concept in game theory. Each player in a non-cooperative game settles on a strategy; such that no player

has anything to gain by merely changing his own strategy constitutes a Nash Equilibrium (Jiang & Brown, 2009). Game Theory is a well studied area of mathematical modeling which finds its origin back in 1944 by John Van Neumann. The modern game theory was developed actively by John Nash during 1950 (Nash, 1951), who introduced the criterion for players' strategies popularly known as Nash Equilibrium. Since then, this topic has been regarded as an important tool and extensively studied in the field of economics.

In (Deng et al., 2014), a game theory framework is designed for a multi-criteria decision-making process in a competitive environment. This work employs Dempster – Shafer Theory (DST) to deal with uncertain information. DST is an approach of combining the degrees of belief from each independent evidence and provides a reasoning to calculate the overall probability of an event. A prisoner's dilemma situation in software development using extreme programming has been analyzed by Hazzan & Dubinsky (2005). Prisoner's dilemma is a popular example in game theory to illustrate the non-cooperative nature of two rational players.

As mentioned earlier, Game Theory models are being applied to identify solutions for decision making and optimization processes in computer science. Among which, the notable few works that applied Game Theoretic concepts to Software Engineering are discussed in this section. Software Refactoring is an important activity in software maintenance which helps in the improvement of the internal structure of the code, without modifying the external visible properties. In (Bavota et al., 2010), the authors apply the game theory solution for the Extract Class refactoring, by modeling as a two-player non-cooperative game. They proved the applicability of the game theory concepts for a refactoring problem.

In this research work, the characteristics of antifragility are introduced into the software using fault injection mechanism. This idea of fault injection in production has been used for a long time to ensure the reliability and thus to trust the system. In (Basiri et al., 2016), the authors explain a practice called "Chaos engineering" to verify the reliability of a distributed system with complex behavior and failure modes. Traditional software testing approaches are not sufficient to identify the potential failures of such complex distributed systems. The server and the clients are overloaded to test the different failure modes. Allspaw (2012) discusses a GameDay exercise that injects faults in production to anticipate similar behavior and understand the effects in future. This exercise forces failures to happen and even designing systems to fail to uncover the risks in the business.

Monperrus (2017) discusses the novel concept of called "software antifragility". The foundations of software antifragility, from classical fault tolerance to the most recent advances on automatic software repair and fault injection in production are explained. The concept of introducing antifragility into the software is viewed as the dream of building reliable system from unreliable components. In this paper also, the test-driven development is insisted in order to uncover the impact of the errors.

The summary of the literature so far is only the beginning of antifragile software engineering. It urges to devise a technique to build effective antifragile software within the existing developmental models. This would make the proposed method to be easily adopted for improved quality and reduced risk. Hence, the application of game theory for building antifragile software would be the most appropriate choice worth exploring.

3. SOFTWARE QUALITY GAME MODEL

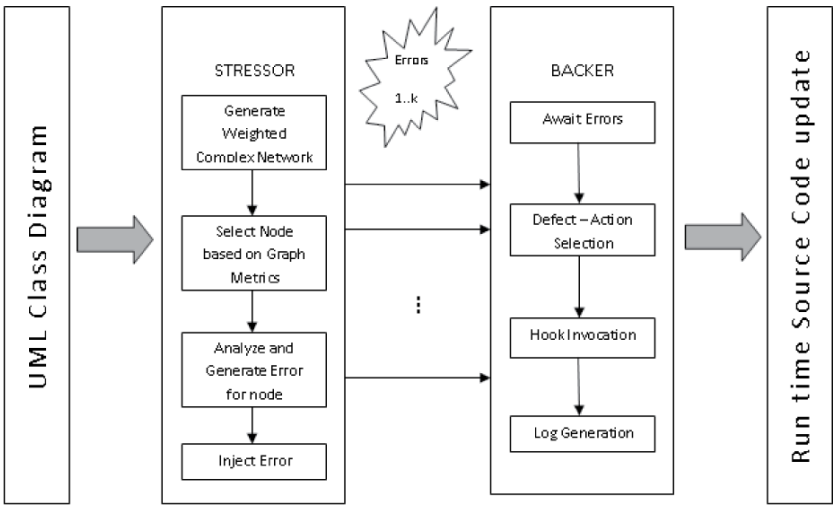
To facilitate the explanation of the antifragility-based game theoretic approach, we start by reviewing the concepts of game theory. A software game can be played between two or more players. The game model designed in this research work is a two-player concurrent game. In a concurrent game, all the involved players make their moves simultaneously. The outcome of the game is jointly determined by the moves taken by all the players. Such a game model is significant in expressing a software system. The two players involved in the game designed in this work are the stressor and the backer entities. The stressor is responsible for injecting errors into the software system. The strategy of the stressor

is to crash the system by injecting a greater number of errors by selecting critical modules of the software system. The software system acts as the backer, which tries to defend by catching the errors injected by the stressor and invoking appropriate error handlers. The game ends whenever the system crashes or a maximum of *K* errors are reached. When the system crashes, the backer fails and the stressor wins. In contrast to existing software, in which the error handlers are hard coded, the backer designed in this work executes a special function, called the Software Learning Hooks (SLH). For any uncaught errors, the backer executes these software Learning Hooks by cheating the stressor and improvises the software by making it immune to such type of errors in the future. Since the backer invokes SLHs for the unknown errors encountered, the software learns and improves continuously. Hence, the probability of the system crashing due to a greater number of different types of unknown errors is extremely less. This probability also depends on flooding a greater number of errors arriving at frequent intervals. This game model is run for *k* number of errors at the end of every iteration, before release to the customer. The SLH modifies the software at runtime temporarily in order to counter the effects of the injected errors. The SLH takes certain actions depending upon the type and nature of the incoming errors, by learning using Q-learning algorithm. The Q-leaning algorithm is a Reinforcement Learning (RL) algorithm, which learns by receiving a reinforcement or reward from the interacting environment. These RL algorithms are predominantly applied in game playing and control problems. This Q-learning algorithm used here is explained in detail in Section 3.2. Once the game is successfully run, a detailed report of the injected errors and the actions taken are output for the developers to perform the quality analysis and make decisions on whether any change in the code is required. This approach is followed instead of actually modifying the code, foreseeing the risk whether the system can completely handle the injected errors, any degradation that may occur due to changes done by the SLH, whether the changes over perform the cost of the software, etc. Hence, the decision is left to the developer of the software to make any changes, if required by analyzing the detailed report. The overall design and working of the game components are depicted in Figure 1.

3.1. Stressor Design

In general, the component of stress is important for any system to operate and improve. From the perspective of biology, certain types of stress improve the state of the organism. All stressors are not treated equally, and some stresses certainly make an organism stronger. “Hormesis” as explained in biology is a condition where, a low dosage of toxin leads to beneficial effect of immunization to that

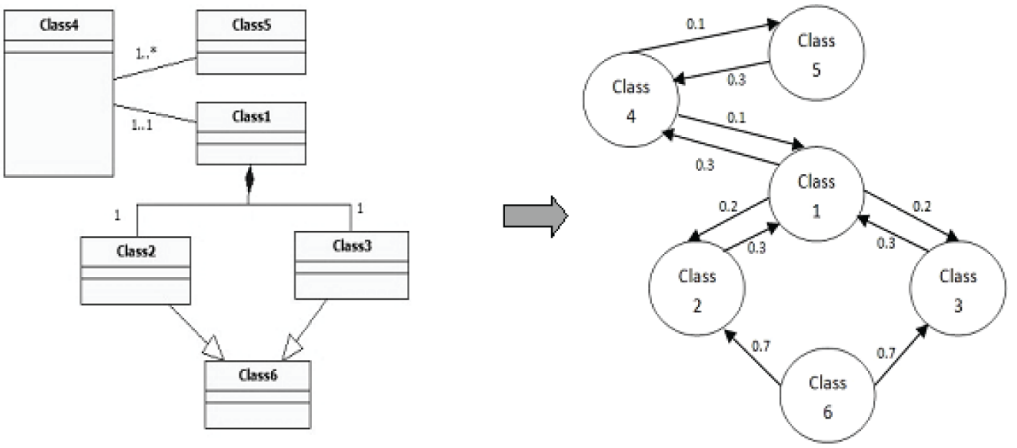
Figure 1. Flow chart of the software game model



toxin. As elucidated in (Taleb, 2012), Hormesis is close to Antifragility. The software is exposed to a series of stressors in order to make them immune against these types of perturbations. In order to achieve this task, a versatile design for the stressor is necessary. An UML class diagram is an important artifact in any software development where, various classes and their relationships are picturized. These UML class diagrams are converted into a weighted complex network. A weighted approach is chosen in order to preserve the information of the classes and the semantics of the relationships that exist between them. The weights are calculated based on the QMOOD (Bansiya & Davis, 2002) metrics of the class using LCOM4 and WMC and the relationships that exist between the classes are given a weightage based of an ordinal scale from 1 to 10 depending on the type of relationship between the participating classes (Chong & Lee, 2015; Pan, 2011). The approach used in (Chong & Lee, 2015) is adopted here for modeling a complex Object-Oriented Software System.

A sample complex network that consists of five classes with the weights assigned as per the relationships between the classes is shown in Figure 2.

Figure 2. A sample conversion of UML class diagram to complex network



Once the software is modeled as a complex network, the application of Graph level metrics is much beneficial in exploiting the structural characteristics of the software system. These structural characteristics of a software network thus generated are closely related to the quality of the software system (Chong & Lee, 2015). The stressor is designed to select the classes based on its importance in the structure of the software and to introduce errors that cause defects. The classes are chosen based on the graph level metrics and its importance in fulfilling the requirements of the overall software. Metrics such as Average Weighted Degree, Average Shortest Path length and Betweenness Centrality are chosen to identify the most significant classes in the software and the error injection among these classes is given higher weightage.

A graph consists of a set of nodes and edges and represented as G(V,E). The degree of node is the total number of edges connected to the node. The in-degree is the number of incoming edges for a node, whereas, out-degree is the number of outgoing edges emanating from a node. The average degree of a network represents the average degree of the entire nodes in the network. For a weighted complex network, it is denoted as average weighted degree. The average shortest path length is the distance measure of a source to all other reachable destinations of the network. The betweenness centrality of a node measures the number of shortest paths that passes through a given node (Albert & Barabasi, 2002).

The operating states of the selected classes are analyzed in a fine-grained level for the injection of the error. The error generation rate of the stressor follows Poisson distribution. Poisson's process is one of the most widely used processes for modeling the probability of occurrence of random events at certain rate (Ross, 2009). The Poisson process is a random process which counts the number of random events that have occurred up to some point t in time. The random events must be independent of each other and occur with a fixed average rate. The probability mass function of a Poisson distribution with a random variable X and rate λ is given below:

$$P\left(X = k\right) = \frac{e^{-\lambda}\lambda^{k}}{k!}, where, k = 0,1,2,\ldots$$

The algorithm for the Stressor module is presented below.

Stressor Algorithm

```
Step  Computation Details
1     Start Stressor
2     Input the source code zip format
      Input UML files XML format
      Initialize max error count k
3     Generate Graph parsing the UML file
4     Calculate graph metrics for every node in the graph
5     Prioritize the node list based on graph metric values
6     If error count reaches k; Goto Step 11
7     Select the prioritized class from list
        Analyze class members and relationships
        Generate defect data
8     Generate Poisson Defect event with defect data
        Instantiate class with defect data
9     Log the error generated details
10    Increment error count; Goto Step 6
11    Exit Stressor
```

The prioritization of the nodes is done based on the graph metrics such as average weighted degree, average shortest path length and Betweenness centrality. These three-graph metrics are chosen since these provide the importance of a node in a network. The higher value of these metrics explains that the nodes are vital in the operation of a network. From the software perspective, erroneous operation of a node with high values of these metrics poses greater chances of failure of the software. Hence, it is important to validate the efficiency of operation and error handling in these priority nodes. The Stressor design ensures more types of errors are injected involving these nodes.

3.2. Backer Design

The function of the Backer entity is to act upon the errors injected by the Stressor. These errors have to be captured by the error handlers provided in the software. This game play ensures frequent invocation of these error routines, which prevent code rusting, making the software more robust against these categories of errors. However, the error handlers provided in the software will not be sufficient to handle all the errors generated by a versatile stressor as explained earlier since foreseeing and hard coding of all these error cases is highly impossible. Hence, the Backer designed here is able to capture the unknown and unexpected errors and record the fine-grained information about

the incoming errors. This fine-grained information is required to analyze and learn from the errors to aid in the process of making the software antifragile in the successive increments.

This functionality of the Backer is achieved by a cheating mechanism by implementing hooks, called as Software Learning Hooks (SLH). These hooks incorporate a reinforcement learning algorithm to learn from the incoming errors. The Q-learning algorithm used here is helpful in selecting the most efficient action for the defects encountered. This builds antifragility into the software when the code is updated in successive iterations with knowledge gained from the game run.

Reinforcement learning is a type of unsupervised learning, where a software entity learns by receiving a reward from the environment in return of executing an action. Based on the reward, the software dynamically learns to operate perfectly in a particular environment (Singh, Lewis, Barto, & Sorg, 2010). The hooks designed in this model follow a "cause and effect" idea. The purpose of the SLH is to select an action for the defects injected by the Stressor. For selection of this action, the Backer uses this SLH, which implements a Q-Learning algorithm. Q-Learning is a Temporal Difference (TD) learning approach of machine learning to predict a quantity based on differences in predictions over some time steps (Poole & Mackworth, 2017). This algorithm learns to arrive at an optimal policy for selecting the actions based on reinforcement called reward. The Q-Learning algorithm usually receives and learns from the reward received from the environment upon which the action is taken. However, this reward function need not be extrinsic always. For example, in a biological environment, certain animals perform some action merely for enjoyment of the task or out of curiosity. In these situations, the reward function can also be a function of the internal state of the agent, in which case it is considered as an intrinsic reward (Singh, Lewis, Barto, & Sorg, 2010). The Q-Learning algorithm designed here uses a combination of both intrinsic and extrinsic reward functions. This hybrid reward model is used because, the extrinsic reward, which is calculated from the changes in the quality attributes may not be always sufficient in calculating the reward. The intrinsic reward is calculated based on the defect salience. Defect salience denotes the severity and the priority of the defects. The extrinsic reward is calculated based on the quality functions such as Abstraction, Inheritance and Coupling measured from the source where the hooks are inserted. A weighted sum of software metrics is used in calculating the extrinsic reward. The source code metrics used for calculation of the quality attributes are adopted from Mohan, Greer and McMullan (2016).

The operational detail of the Backer module is explained in the following algorithm.

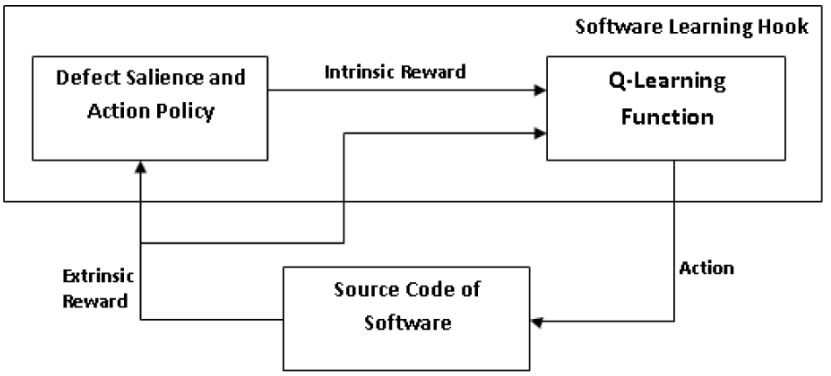
Backer Algorithm

Step	Computation Details
1	Initialize reward functions // for quality attribute calculation Load Defect-Action policy lookup file Initialize τ, α, γ
2	For each incoming unhandled defect D^i; Goto Step 3
3	Analyze the defect data Select Action A^i from Defect-Action policy Add a runtime hook
4	Invoke the hook
5	Log the defect-action details Capture the core and memory dump
6	Calculate intrinsic reward $r^i = \tau\ [1 - P(D^i)]$
7	Calculate reward functions Calculate extrinsic reward r^e = cumulative reward function values
8	Calculate the Q-Value using $Q\left(D,A\right) \leftarrow Q\left(D,A\right) + \alpha[r^i + r^e + \gamma \cdot max\ Q\left(D',A'\right) - Q\left(D,A\right)]$
9	Update the Defect-Action Policy; Goto Step 2

Here,
α →The learning rate, $(0 < \alpha \leq 1)$
γ → The discount factor, $(0 < \gamma \leq 1)$
r^i → Intrinsic reward based on defect salience
r^e → Extrinsic reward based on quality attributes
τ → A constant multiplier
$P(D^i)$ → Probability model for defect D^i
$Q(D,A)$ → Q-value of a Defect – Action pair

The Q-value of a Defect – Action pair estimates the goodness of the action taken or the extent to which the results of the action taken is expected to be. These value functions are represented using notation $Q(D,A)$ –which represents the value of taking action *A* for the defect *D* under certain policy. This policy ensures that the expected value of the reward return of the actions taken is to the maximum achieved. The functional design of a Backer module representing the hybrid reward model is diagrammatically represented using Figure 3.

Figure 3. Functional diagram of SLH



By performing the above method, the hook learns over a period of time about the actions to be taken for the incoming types of defects. Hence, operating upon a considerable number of defects, the hooks help in the betterment of the performance of the software to certain type of repeated defects.

3.3. Evaluation Parameters

The effectiveness of the proposed technique is analyzed using two categories of performance parameters, namely the defect related and quality related attributes. These parameters are evaluated at every iteration and at the end of the all the iterations to facilitate the comparison of improvements. The details of these game evaluation parameters are given in Table 2.

Defect Density is an important and fundamental metric in Software Engineering. The metric is calculated from the total number of defects discovered divided by the size of the software. Size can be expressed in terms of Function Points (FP) or Lines of Code (LOC). This metric gives a picture of the overall quality of the software and decides whether the product is ready for release. Many research works in literature uses defect density as a vital software metric for quality investigation and defect prediction (Cartwright & Shepperd, 2000; Koru, Zhang, Emam, & Liu, 2009; Yadav & D. Yadav, 2015). Defect Distribution gives the categorization of the defects based on type, root cause priority, test type, environment, etc. Studying the defect distribution chart is helpful in understanding and

Table 2. Details of the performance parameters

Category	Parameter	Context	Desired Value
Defect related	Defect Density	The number of defects per thousand lines of code	Low
	Defect Distribution	The number of different types of defects in each iteration	Low
	Hooks per Increment	The number of hooks that are executed in each iteration	Low
Quality related	Abstraction	Easiness with which the system can be extended, suppressing the more complex details	High
	Inheritance	The mechanism of creating new classes from existing classes	High
	Coupling	The measure of dependency between classes	Low

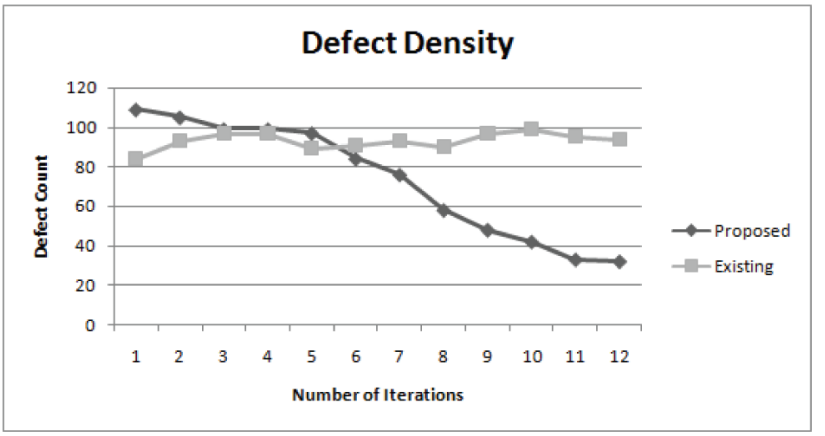
identifying the areas to target for defect removal and the components that requires the application of quality improvement techniques. Researchers use this defect distribution analysis to gain understanding on the technique they applied and its outcome (Cartwright & Shepperd, 2000; Li, Li, & Sun, 2010; Beller, Bholanath, McIntosh, & Zaidman, 2016). The Hooks per Increment metric gives the trend of the number of unknown defects and hence, it is important to arrive at a logical conclusion for the reduction in the number defects in the source code. Apart from these defect related parameters, this study uses quality related parameters, namely the quality attributes such as Abstraction, Inheritance and Coupling. These attributes are used in most of the quality analysis literature and applied in majority of the software engineering problems such as architectural analysis, code quality analysis and improvement, refactoring, etc. These quality attributes are measured using standard well-defined software metrics. As mentioned earlier, QMOOD metrics proposed by Bansiya and Davis (2002) is used in this research.

4. RESULTS AND DISCUSSION

The experiment is conducted using a sample open source Java project with 236 classes, namely the OpenFAST. OpenFAST is an open source Java implementation of the FAST protocol (OpenFAST, 2013). The classes are prioritized based on the graph metrics and the experiment is conducted with 12 iterations. The initial iterations are planned with the important classes and the other classes are integrated incrementally with the number of iterations. The software quality game model designed here is executed for every iteration and the game parameters are studied. The detailed report of the defects and the hooks that are inserted are analyzed and the changes in the code, if required, are performed in the next successive iterations. This is done after the review confirmation from the developer in order to avoid any adverse effect that may be caused by the execution of hooks. Moreover, the hooks are only temporary alterations to the code in runtime. Hence, the changes have to be studied before making any actual changes to the code. This process as explained earlier pushes the software towards antifragility. The following graphs explain the details of the results achieved.

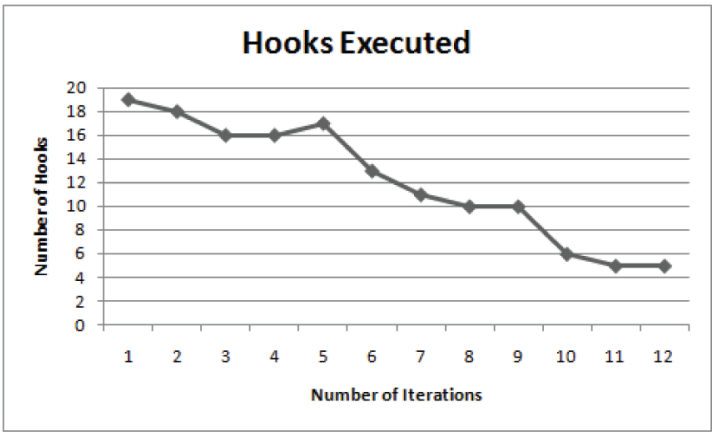
Figure 4 shows the defect density values in various iterations with and without the proposed game theoretic approach. From the graph, there is reduction in the number of defects when the proposed technique is applied. This is due to the fact that the defects that are injected in the iterations are learned by the SLHs and the changes are made in the code to capture these categories of defects. In the existing method, these defects still exists which leads to the increased number of defects. Hence,

Figure 4. Defect density over number of iterations



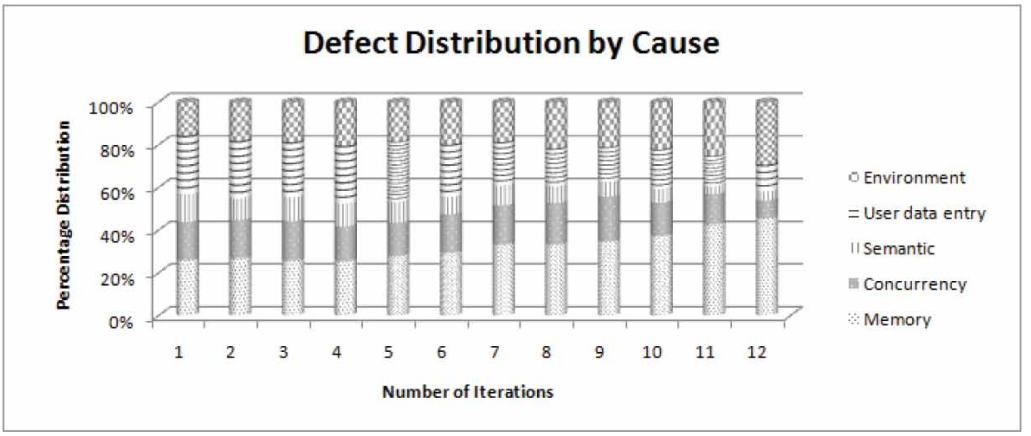
with the successive iterations, there is a reduced defect count by using the proposed technique, whereas the defect count is still high in existing method. A similar inference can be made for the number of hooks inserted over iterations (Figure 5), which reduces over iterations. There are only few hooks required to handle the injected defects and the majority of the defects are captured by the code itself,

Figure 5. Number of hooks executed over number of iterations



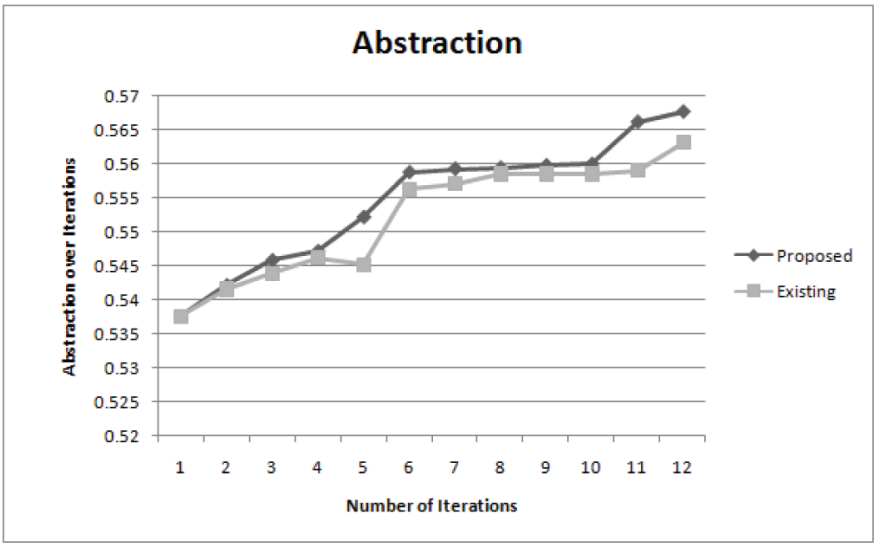
which is modified in earlier iterations based on the learning information gained by the SLHs. The defect distribution chart is given in Figure 6, which gives the categories of defects occurred over different iterations. From this chart, it can be inferred that the majority of the defects during the initial phases of iteration were the defects related to memory, user data entry and environment. Even though there is reduction in the defects related to user data entry over iterations, memory and environment related defects contribute more, which needs special attention in the software design process. Much of the semantics and concurrency related defects are uncovered during earlier iterations and there is reduction in these categories of defects. It has to be noted that here the error injection rate by the Stressor is unaltered and only the number of uncaught or unforeseen errors are reduced.

Figure 6. Defect distribution by cause chart over number of iterations



Figures 7, 8 and 9 shows the improvement in the quality attributes, namely the Abstraction, Inheritance and coupling respectively. The values of these quality attributes are compared with and without the proposed game theoretic approach. The improvement shown here is due to the modification done in the code by the development team based on the details of the defect-action report output by the Backer. As such the execution of SLH does not contribute to the improvement of the code; it is

Figure 7. Improvement in abstraction over number of iterations



the effectiveness of the learning gained by the SLHs. From the Figures 7, 8, and 9, even though there is improvement in the quality attributes, the proposed technique does not increase them to a greater extent. But from Figure 4, there is drastic decrease in the defect density by using the proposed game model. Hence the proposed technique can be very well applied to reduce the defect density of any object-oriented software.

Figure 8. Improvement in inheritance over number of iterations

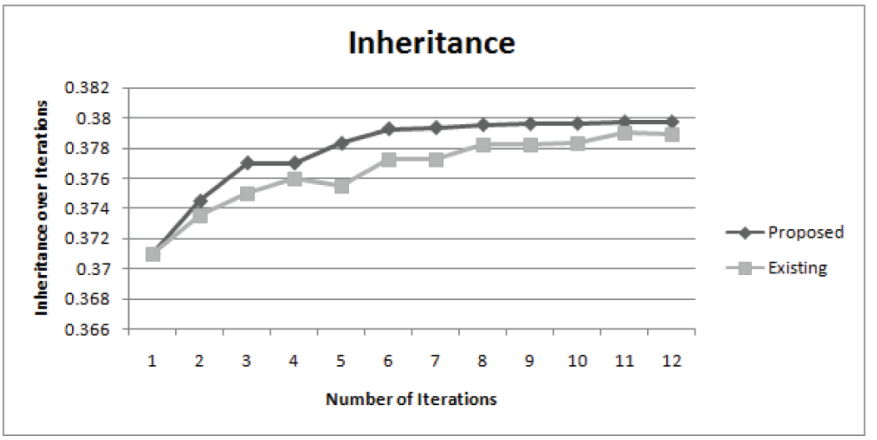
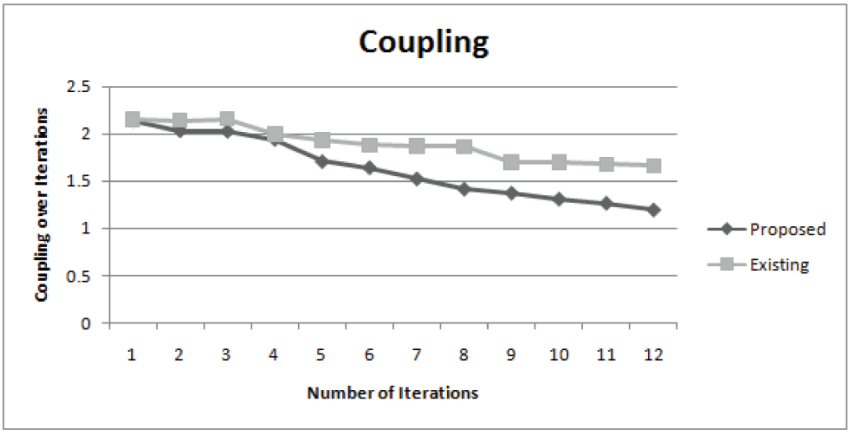


Figure 9. Improvement in coupling over number of iterations



5. CONTRIBUTIONS

The proposed model enhances the quality of operation of the software system by following a novel approach for the development process. Within the existing traditional way of development and quality assurance, an innovative lateral approach has been introduced in this work. The summary of our contributions are as follows:

1. Introduction of game theoretic approach for quality enhancement of the software system by using the stressor and backer entities;
2. Following an innovative lateral approach by using cheating methods (via, software leaning hooks) to achieve positive benefits;
3. Antifragility is a rising issue in Software Engineering. This work provides a good start on achieving antifragility within the existing framework of software development methodology;
4. Apart from the existing error models, which inject errors at random, our model uses an intelligent and more realistic error model. This is achieved by modeling the software system as weighted

complex networks and making use of the graph level metrics such as Betweenness Centrality, Average weighted degree of the nodes and Average Shortest Path Length;
5. Application of Q-Learning reinforcement function with a hybrid reward function that predicts the actions to be taken for the injected errors;
6. A better result in terms of reduced defect density and improved quality attributes.

6. EMPIRICAL VALIDATION

The threats to the validity of the study are discussed with respect to the design viewpoint of the Stressor and the Backer modules. Firstly, even though the Stressor uses a versatile design using graph metrics, the generation of defects varies for different class and highly depends on the design of the class. This means that the type of attributes, the type of relationship of the class and role of the class in the overall structure of the software plays a vital role and differ from project to project. Hence, the technique has to be applied for wide range of real time projects and experience need to be gained in order to arrive at a mature logic for defect generation. Secondly, the software learning hooks used in the backer design needs more attention which poses security concerns. The hooks are only the temporary alterations to the code at runtime and hence require caution while using and also, the execution time of the proposed method increases with the number of hooks running at a time. The next viewpoint is based on the defect-action policy used by the backer. This again requires testing on a greater number of projects. The last viewpoint is the logging of fine-grained information about the incoming defects and the action taken by the hooks. The log must provide required information for the developer to carry out the analysis and hence the level of details logged must be satisfying. The core dump and the memory dump need to be captured for each generated defect that put forth storage constraints of the output files.

7. CONCLUSION

The increase in the complexity of software systems mandates the use of innovative technologies to cope with the increased number of software defects. Antifragility is a most desired property of software in current trends. This property makes the software immune to unknown and unexpected defects, the called Black swans. It prepares the software to learn from the errors. Antifragility is a breakthrough and the binding property of the future software development. This research work is motivated in incorporating antifragility into the existing framework of object oriented software development, thereby increasing the quality of operation of the software. Hence, to build this property in any software, the system must be exposed to a number of disorders; in the software perspective, the defects that occur during the operation of the software.

Usually, this type of technique in applied in a production environment as a routine of defect driven approach. In contrast to this method, in our work, the proposal is applied during the development phase of the software life cycle. This approach is combined with the concepts of Game Theory, by designing a two-player quality game. Player 1 called the Stressor injects errors into the software and the Software acting as Player 2 resists and learns from these errors. The Stressor uses the intelligence gained through the graph metrics measured from the architecture of the software for defect generation. The defects are introduced at intervals following a Poisson's distribution up to a maximum of 'k' defects. The Backer acts upon these errors and its primary concern is to avoid crashing of the system. To achieve this, the Backer uses a cheating mechanism using hooking techniques to defend against the errors. These software learning hooks incorporate reinforced learning algorithms, to progressively gain insight into the type of defect and selecting appropriate action to take to handle these errors. The hooks are introduced at run time into the software to make temporary changes into the software. At the end of the game run, a detailed report of defect-action information is output for the developer to analyze and make actual modifications in the code to handle the possible errors. Since hooks

are only temporary modifications to the code at runtime, these changes have to be validated by the development team before incorporating into the actual source code. This step is used as a precaution to avoid any adverse effect of the code changes. This process is repeated for various iterations of the project, making the software develop its antifragility asset.

The proposed method is validated using an open source Java project with 236 classes. The results are studied with a two-fold category of parameters, namely, the defect related and quality related sort. The defect related parameters give the insight of the number and the category of the defects that can be uncovered from the system. By following the proposed approach leads to reduced defect density compared to the existing method of developments. The results of the quality related parameters, namely, abstraction, inheritance and coupling measured using QMOOD metrics shows stable improvement.

Even though there are certain challenges that need further insights and improvements as explained in the Empirical validation section, this approach is a step towards building antifragility into the software development methods. Thus, the research questions RQ1 and RQ2 set in Section 1 are tested and asserted, proving the applicability of Game Theory in building Antifragile Software Systems. Some future research directions include, (i) Examining the cases where the continuous improvement of the system leads to changes in the operational profiles of the software, (ii) Security issues when hooking techniques are used and analysis of alternate techniques instead of hooks, (iii) Selection of better learning algorithms for the learning hooks, and (iv) Analyzing the usage of other techniques such as Meta Programming and Self Modifying code instead of hooks to overcome performance issues.

ACKNOWLEDGMENT

This research work is funded by the Quality improvement Programme (QIP) launched by All India Council for Technical Education (AICTE), Government of India.

REFERENCES

Albert, R., & Barabasi, A. L. (2002). Statistical mechanics of complex networks. *Reviews of Modern Physics*, *74*(1), 47–97. doi:10.1103/RevModPhys.74.47

Allspaw, J. (2012). Fault injection in production. *Communications of the ACM*, *55*(10), 48–52. doi:10.1145/2347736.2347751

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17. doi:10.1109/32.979986

Basiri, A., Behnam, N., Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos engineering. *IEEE Software*, *33*(3), 35–41. doi:10.1109/MS.2016.60

Bavota, G., Rocco, O., Lucia, A. D., Antoniol, G., & Guéhéneuc, Y. G. (2010). Playing with refactoring: Identifying extract class opportunities through game theory. In *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE Press. doi:10.1109/ICSM.2010.5609739

Beller, M., Bholanath, R., McIntosh, S., & Zaidman, A. (2016). Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (pp. 470-481). IEEE Press. doi:10.1109/SANER.2016.105

Cartwright, M., & Shepperd, M. (2000). An Empirical Investigation of an Object-Oriented Software System. *IEEE Transactions on Software Engineering*, *26*(8), 786–796. doi:10.1109/32.879814

Chong, C. Y., & Lee, S. P. (2015). Analyzing maintainability and reliability of object-oriented software using weighted complex network. *The Journal of Systems and Software, Elsevier*, *110*, 28–53. doi:10.1016/j.jss.2015.08.014

Deng, X., Zheng, X., Su, X., Chand, F. T. S., Hu, Y., Sadiq, R., & Deng, Y. (2014). An evidential game theory framework in multi-criteria decision making process. *Applied Mathematics and Computation, Elsevier*, *244*, 783–793. doi:10.1016/j.amc.2014.07.065

Hazzan, O., & Dubinsky, Y. (2005). Social Perspective of Software Development Methods: The Case of the Prisoner Dilemma and Extreme Programming. In *Proceedings of the International Conference on Extreme Programming and Agile Processes in Software Engineering* (pp. 74–81). Springer. doi:10.1007/11499053_9

Huang, C. H., Peled, D., Schewe, S., & Wang, F. (2016). A Game-Theoretic Foundation for the Maximum Software Resilience against Dense Errors. *IEEE Transactions on Software Engineering*, *42*(7), 1–1. doi:10.1109/TSE.2015.2510001

Jiang, A. X., & Leyton-Brown, K. (2009). A tutorial on the proof of the existence of Nash equilibria. Univ. British Columbia.

Koru, A. G., Zhang, D., Emam, K. E., & Liu, H. (2009). An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. *IEEE Transactions on Software Engineering*, *35*(2), 293–304. doi:10.1109/TSE.2008.90

Kumar, K., & Prabhakar, T. V. (2009). Quality Attribute Game: A Game Theory Based Technique for Software Architecture Design. In *Proceeding of the 2nd annual conference on India software engineering conference* (pp. 133–134). ACM. doi:10.1145/1506216.1506244

Leach, R. J. (2016). *Introduction to Software Engineering* (2nd ed.). CRC Press.

Li, N., Li, Z., & Sun, X. (2010). Classification of Software Defect Detected by Black-Box Testing: An Empirical Study. In *Proceeding of the 2010 Second World Congress on Software Engineering* (pp. 234-240). Academic Press. doi:10.1109/WCSE.2010.28

Mehdi, M. M., Raza, I., & Hussain, S. A. (2017). A game theory based trust model for Vehicular Ad hoc Networks (VANETs). *Computer Networks: The International Journal of Computer and Telecommunications Networking*, *121*, 152–172. doi:10.1016/j.comnet.2017.04.024

Mohan, M., Greer, D., & McMullan, P. (2016). Technical debt reduction using search based automated refactoring. *Journal of Systems and Software, Elsevier*, *120*, 183–194. doi:10.1016/j.jss.2016.05.019

Monperrus, M. (2017). Principles of Antifragile Software. In Programming '17. ACM Press.

Nash, J. (1951). Non-Cooperative Games. *The Annals of Mathematics*. *Second Series*, *54*(2), 286–295. doi:10.2307/1969529

National Institute of Standards and Technology. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing.

OpenFAST reference. (2013). Retrieved from http://openfast.sourceforge.net/

Pan, W. (2011). Applying Complex Network Theory to Software Structure Analysis. *World Academy of Science, Engineering and Technology*, *60*, 1636–1642.

Poole, D., & Mackworth, A. (2017). *Artificial Intelligence: Foundations of Computational Agents* (2nd ed.). Cambridge University Press. doi:10.1017/9781108164085

Ross, S. (2009). *A First Course in Probability* (8th ed.). Prentice Hall.

Russoa, D., & Ciancarinia, P. (2016). A Proposal for an Antifragile Software Manifesto. *Procedia Computer Science*, *83*, 982–987. doi:10.1016/j.procs.2016.04.196

Singh, S., Lewis, R. L., Barto, A. G., & Sorg, J. (2010). Intrinsically Motivated Reinforcement Learning: An Evolutionary Perspective. *IEEE Transactions on Autonomous Mental Development*, *2*(2), 70–82. doi:10.1109/TAMD.2010.2051031

Taleb, N. N. (2008). *The black swan: the impact of the highly improbable* (2nd ed.). London: Penguin.

Taleb, N. N. (2012). *Antifragile: Things That Gain From Disorder*. USA: Random House Publishing Group.

Yadav, H. B., & Yadav, D. K. (2015). A fuzzy logic based approach for phase-wise software defects prediction using software metrics. *Information and Software Technology*, *63*, 44–57. doi:10.1016/j.infsof.2015.03.001

Vimaladevi M. is currently a research scholar in the Department of Computer Science and Engineering, Pondicherry Engineering College. She completed her B. Tech. and M. Tech. in computer Science and engineering from Pondicherry University. She has 12 years of work experience in both industry and academics. She had been working as Project Lead in L&T Infotech for Hitachi client. She had executed various projects and involved in all the phases of software life cycle including planning and estimation, design, development and testing. She is experienced in executing projects in various process models like Waterfall, V-model and Agile. She was involved in the quality analysis and assessment of the software, client communication and received customer appreciation. Her areas of interest include software engineering, project estimation, project management, and process models.

G. Zayaraz (PhD) is currently working as a Professor in the Department of Computer Science and Engineering, Pondicherry Engineering College. His areas of interests include software engineering and information security. He completed his B. Tech., M. Tech. and PhD in computer Science and engineering from Pondicherry University. He has 28 years of teaching experience at all levels namely industry, diploma, undergraduate, post graduate and research. He has officiated as the Head of MCA department and currently is the Associate Dean (Student Affairs). To his credit he has published more than hundred research papers in reputed International Journals and Conferences. He has authored a book titled Quantitative Evaluation of Software Architectures sold by leading book sellers. He has been the advisory member, and reviewer for several International Conferences. He has been the guest editor of Inderscience special issue journal on software engineering. Under his guidance, 7 scholars have successfully completed their PhD and 4 students are pursuing their PhD.