ISSN: 2394-7659
IMPACT FACTOR- 3.775

IJERMS
International Journal of Engineering Researches and Management Studies

UNDERSTANDING GAME THEORETIC APPROACH TO SOFTWARE TESTING

Avinaash Anand K., Harshit Raj & Shaurya Joshi

ABSTRACT

Software development cycle (also known as SDLC) is the process of software development in small modules which are later integrated to form a standalone software. Testing is on of the steps which are a part of the process whereby a module of code is tested before being incorporated into the final program.

1. INTRODUCTION

Software development cycle (also known as SDLC) is the process of software development in small modules which are later integrated to form a standalone software. It consists of various steps such as software design, coding, implementation, etc. Testing is one of the steps which are a part of the process whereby a module of code is tested before being incorporated into the final program. For ensuring a healthy code, it is imperative that the code which is sent out to the customer is error free, thereby for every code generated by a developer, there is a set of attackers of attackers, who entail a certain effort to discover the flaws in the code. Depending on the tolerance level of an organization and prioritization parameters, a code can be deemed to be either fit to use. Unlike other products, software per se has a low manufacturing cost as the main expense is in formulation of that design and ensuring a smooth execution. There can be multiple kinds of testing on a software like unit testing, integration testing or system testing Boris Be (Beizer,2009).

Game theory has traditionally been used as a mathematical modelling technique based on the strategic interactions of cooperation and competition. (Eduardo Mattos, 2014). There has been some literature wherein the incentive mechanism for a worker is determined using an application of game theory. Through ourgame, using a tester and developer model, weintend to test the application of game theory in a software testing framework, in order to better understand the motivation behind either the developer to create an efficient code or that of the tester to ensure that the test code run by him/her is free of errors. This game has been modelled on this precise phenomenon wherein we try and estimate the utility of each of these players to perform the task allotted to them to the best of their abilities. Therefore, game theory was used as a platform to understand this mentality of the players and understand how they would react given different situations.

2. THE TESTING GAME

The game between the tester and the developer can be compared to a security game, akin to the one which is played between security staff (defenders) and system attackers in various situations. In this case, the testers assume the role of defenders and developers play attackers. The testers try to protect the software from bugs and inefficiencies in the code, while the attackers try to breach the system by checking in inefficient chunks of code which might contain bugs. The company has limited security resources (testers) and they try to deploy them as efficiently as possible to prevent the code from bugs. The testers gain a positive pay off from defending the software against bugs, thus gaining from every bug that they catch. The developers gain utility from saving their time in correcting the inefficiencies, whilst devoting the time that they save in the process on working on another task: thereby expediting their work. Both set of players have their respective penalties also. Testers are penalized for any piece of code that contains bug and goes untested, as this deteriorates the software quality and can damage the reputation of the software company, thereby leading to heavy losses.

In case the bug goes undetected, the client might be dissatisfied with the software quality and the bug would have to be removed in future software update which would be costlier. This process explains the penalty behind bug misses by testers. On the contrary, the developers are also penalized but in somewhat different conditions. In case the testers catch the bug/inefficient code that the developer has checked in, the developer must rectify the code which takes even more effort than it would've taken in the initial iteration. Besides, the developer also faces a social cost for each bug that is caught, as it results in a loss of quality perception in the eyes of his clientele and fellow employees. Thus, testers and defenders both face some constraints and are forced with making some choices pertaining to their concurrent states. Although, both defenders and attackers have a singular goal: maximizing their individual payoffs. Both try to rationalize their choices given the action of the adversary which results in a game being played between the two.

3. MODELLING THE GAME

This game between the tester and the developer can be modelled in three steps. The first step consists of determining the types of bugs that can creep in a software module, the associated utilities/rewards the tester would get on catching a specific type of bug, the penalties that he would have to incur if he fails to catch that bug, besides the set of utilities and penalties for the developer as described in the above paragraph. The next step would involve devising the utility, penalty (or effort), and the pay off equations for the tester and developer. The final step would be solving the two set of equations individually for maximizing the profit for the tester and the developer and have a look at the conditions in which they would deliver optimal performance.

A. Classification of bugs and definition of notations

Bugs are inefficiencies or errors in a piece of code which can cause the software to malfunction or stop. Bugs are mostly encountered on corner cases (the extreme ends of data sets that the software program is capable of processing), besides being found in many other peculiar forms. Some of these can be more severe than the others. Based on severity, bugs have been classified into three parts for the purpose of this study:

- High severity(HS):
- Medium severity(MS)
- Low severity(LS)

We assume that each type of bug is found in some amount in a software module, and tester tries to run the designated test scripts in order to catch as many bugs as possible (or optimal). The developer tries to check-in the bugs at the rate of *n(X)* where X denotes the severity quotient of the bug.
Total number of high severity (HS) bugs in the software module = n(HS) Total number of medium severity (HS) bugs in the software module = n(MS) Total number of low severity (HS) bugs in the software module = n(LS)

Further, we denote the number of bugs caught by tester with $n_t\ (X)$ where X represents the severity quotient.
Number of high severity (HS) bugs caught = $n_t(HS)$

Number of medium severity (MS) bugs caught = $n_t(MS)$ *Number of low severity (LS) bugs caught =* $n_t(LS)$
Also, the tester is bound to miss some of the bugs in the module, which are denoted by n_m *(X).*

Number of high severity (HS) bugs missed = $n_m(HS)$ *Number of medium severity (MS) bugs missed =* $n_m(MS)$ *Number of low severity (LS) bugs missed =* $n_m(MS)$

We define a variable *q* (0 < q < 1) which denotes the proportion of bugs caught by the testers out of the total bugs present in the module. Therefore,
$n_t(HS) + n_m(HS) = n(HS)$ (where $n_t(HS) = q * n(HS)$) $n_t(MS) + n_m(MS) = n(MS)$ (where $n_t(MS) = q * n(MS)$) $n_t(LS) + n_m(LS) = n(LS)$ (where $n_t(LS) = q * n(LS)$)

After defining the variables for number and types of bugs, we move to define the coefficients for utilities and penalties in case of a tester for catching different types of bugs. The utility coefficients are denoted by α_X and the penalties by P_X where X represents the severity coefficient. The utility coefficients vary on a scale of 0 to 10 (with 10 being the highest utility coefficient and 0 the lowest), while the penalty coefficients range from 0 to 10 (with 0 being the lowest penalty and 10 being the highest).

a) *Tester utility coefficients:*
Utility co-efficient for catching a high severity (HS) bug = $\boldsymbol{\alpha}_{HS}$ *Utility co-efficient for catching a medium severity (MS) bug =* $\boldsymbol{\alpha}_{MS}$ *Utility co-efficient for catching a low severity (LS) bug =* $\boldsymbol{\alpha}_{LS}$

b) *Tester penalty coefficients:*
Penalty for missing a high severity (HS) bug = P_{HS} *Penalty for missing a medium severity (MS) bug =* P_{MS}
Penalty for missing a low severity (LS) bug = P_{LS}

A similar exercise is undertaken for the developers with utility coefficients denoted by $\boldsymbol{\beta}_x$ and penalty coefficients denoted by P'_X where X again denotes the severity quotient. These coefficients are also on a scale of 0 to 10 in the case of utility coefficients and 0 to 10 in case of penalty coefficients.

c) *Developer utility coefficients:*
Utility co-efficient for not rectifying a high severity (HS) bug = $\boldsymbol{\beta}_{HS}$*Utility co-efficient for not rectifying a medium severity (MS) bug =* $\boldsymbol{\beta}_{MS}$ *Utility co-efficient for not rectifying a low severity (LS) bug =* $\boldsymbol{\beta}_{LS}$

d) Developer penalty coefficients:
Penalty for high severity (HS) bug correction in second iteration = P'_{HS} *Penalty for medium severity (MS) bug correction in second iteration =* P'_{MS} *Penalty for low severity (LS) bug corrections in second iteration =* P'_{LS}

Now, as mentioned previously, the testers are not perfect, and they make some errors which lead to bugs creeping in the software even after a suitable test script being run. The probabilities with which such errors can creep in are denoted by $\boldsymbol{Y}_X$as depicted below:
Probability that tester will catch a high severity (HS) bug = $\boldsymbol{Y}_{HS}$
Probability that tester will catch a medium severity (MS) bug = $\boldsymbol{Y}_{MS}$
Probability that tester will catch a low severity (LS) bug = $\boldsymbol{Y}_{LS}$

In this case, the number of rectifications that a developer would have to make is a function of two factors: the number of bugs tested of a specific category, $n_t(X)$ and the probability of the tester catching a bug of that category, $\boldsymbol{Y}_{X.}$ These are denoted by $n_r(X)$.
Number of high severity (HS) bugs resent to developer for rectification = $n_r(HS)$ *Number of medium severity (MS) bugs resent to developer for rectification =* $n_r(MS)$ *Number of low severity (LS) bugs resent to developer for rectification =* $n_r(LS)$

Also, the developer would incur a social cost, *s* with every bug that he has to retest. This social cost captures the loss in the quality perception of his work.
Social cost incurred by developer per bug (of any severity) = s

Now that all the notations and definitions are done with, we move on to formulating the utility function, penalty function and thus, the resultant payoff equations for the developer and the tester.

B. Formulation of payoff functions

First, we'll look at the construction of payoff function of a tester, post which we'll navigate to the developers.

a) *Payoff for testers:*
The utility of tester is the sum of utilities gained by catching all types of bugs as shown below:
Utility of tester $(U_T) = \boldsymbol{\alpha}_{HS}{*}n_t(HS) + \boldsymbol{\alpha}_{MS}{*}\ n_t(MS) + \boldsymbol{\alpha}_{LS}{*}\ n_t(LS)$
Also, penalty that he would incur in case of missed bugs would be given by:

Penalty for missing $(P_T) = P_{HS}*n_m^2(HS) + P_{MS}*n_m^2(MS) + P_{LS}*n_m^2(LS)$

In this case, the penalty is directly proportional to the square of number of bugs found reflects the fact that a client is likely to get increasingly frustrated with a software as the number of bugs increase, as he might tolerate some mistakes but a lot of them would reflect very poorly on the company. Thus, this helps in modelling the incremental (above normal) penalty with each increasing bug.

Now, the payoff of a tester would be given by the difference between his total utility and total penalty.
Payoff (Tester) $= U_T – P_T$
$= [\alpha_{HS}* n_t(HS) + \alpha_{MS}* n_t(MS) + \alpha_{LS}* n_t(LS)] – [P_{HS}*n_m^2(HS) + P_{MS}*n_m^2(MS) + P_{LS}*n_m^2(LS)]$

$= [\alpha_{HS}* n_t(HS) - P_{HS}*\{n(HS) – n_t(HS)\}]^2 + [\alpha_{MS}* n_t(MS) – P_{MS}*\{n(MS) – n_t(MS)\}]^2 + [\alpha_{LS}* n_t(LS) – P_{LS}*\{n(LS) – n_t(HS)\}^2]$

$= [\alpha_{HS}* n_t(HS) - P_{HS}*\{n^2(HS) + n_t^2(HS) - 2* n(HS)* n_t(HS)\}] + [\alpha_{MS}* n_t(MS) – P_{MS}*\{n^2(MS) + n_t^2(MS) - 2* n(MS)* n_t(MS)\}]+ [\alpha_{LS}* n_t(LS) – P_{LS}*\{n^2(LS) + n_t^2(LS) - 2* n(LS)* n_t(LS)\}]$

$= \{n_t(HS)[\alpha_{HS} – 2* n(HS)] - P_{HS}[n(HS)^2 - n_t(HS)^2]\} +\{n_t(MS)[\alpha_{MS} – 2* n(MS)] – P_{MS}[n(MS)^2 - n_t(MS)^2]\} + \{n_t(LS)[\alpha_{LS} – 2* n(LS)] – P_{LS}[n(LS)^2 - n_t(LS)^2]\}$

b) Payoff for developers:
A developer's utility is equal to the sum of time he saves from skipping the bugs that he passes on with the code to the tester, as shown below:
Utility of developer $(U_D) = t_{HS} + t_{MS} + t_{LS},$
where t_X *=time saved by skipping a bug belonging to a specific bug category X.*

Now, the time utility for these bugs can be quantified using the utility coefficient per bug and number of bugs belonging to each category.

Time saved by skipping high severity (HS) bugs $= t_{HS} = \beta_{HS} * n(HS)$
Time saved by skipping medium severity (MS) bugs $= t_{MS} = \beta_{MS} * n(MS)$

Time saved by skipping low severity (LS) bugs $= t_{LS} = \beta_{LS} * n(LS)$

Therefore,
Utility of developer $(U_D) = \beta_{HS} * n(HS) + \beta_{MS} * n(MS) + \beta_{LS} * n(LS)$
Similarly, the penalties that the developer will incur can also be modelled as shown below:
Penalty of developer $(P_D) = (s + P'_{HS}) * n_r (HS)^2 + (s + P'_{MS}) * n_r (MS)^2 + (s + P'_{LS}) * n_r (LS)^2$

Here, the penalties include a uniform social cost for any kind of bug that is caught by the tester, and the penalty function is again directly proportional to the square of bug count reflecting the extra effort that would be required in rectifying additional bugs as the count increases (analogous to the case of testers).

As already mentioned, the bug counts can be rewritten as:
$n_r(HS) = \mathbf{Y}_{HS} * n_t(HS)$
$n_r(MS) = \mathbf{Y}_{MS} * n_t(MS)$
$n_r(LS) = \mathbf{Y}_{LS} * n_t(LS)$

Figuring these values in the above equation, the penalty can also be written as:
$P_D =(s + P'_{HS}) *(\mathbf{Y}_{HS} * n_t(HS))^2 + (s + P'_{MS}) * (\mathbf{Y}_{MS} * n_t(MS))^2 + (s + P'_{LS}) * (\mathbf{Y}_{LS} * n_t(LS))^2$

Payoff (developer) $= U(d) – P(d)$
$= \{\beta_{HS} * n(HS) – [(s + P'_{HS}) * \mathbf{Y}_{HS}^2 * n_t(HS)^2]\} + \{\beta_{MS} * n(MS) – [(s + P'_{MS}) * \mathbf{Y}_{MS}^2 * n_t(MS)^2]\} + \{\beta_{LS} * n(LS) – [(s + P'_{LS}) * \mathbf{Y}_{LS}^2 * n_t(LS)^2]\}$

Now that the payoff equations are done with, the individual optimal states for developers and testers can be calculated.

C. Calculating the optimal solution

The testers would like to increase their individual payoffs, and in turn minimize the inefficiencies present in the code. This would also be the optimal solution for a software firm focussed on quality of its software.

Tester Strategy

To maximize the tester payoff P_t, we put break the equation into 3 parts and then differentiate in terms of the severity of errors.

$$\frac{\partial \text{ Payoff (tester)}}{\partial \, n_t(HS)} = 0;$$
$$\frac{\partial \text{ Payoff (tester)}}{\partial \, n_t(MS)} = 0;\&$$
$$\frac{\partial \text{ Payoff (tester)}}{\partial \, n_t(LS)} = 0;$$

For High Severity bugs:

$$\frac{\partial \left\{n_t(HS)\left[\alpha_{HS} - 2 * n(HS)\right] - P_{HS}\left[n(HS)^2 - n_t(HS)^2\right]\right\}}{\partial \, n_t(HS)} = 0;$$

As $n_t(HS) = q * n(HS)$,

Thus, $n(HS) = \frac{n_t(HS)}{q}$

$$\Rightarrow \frac{\partial \left\{n_t(HS)\left[\alpha_{HS} - \frac{2*n_t(HS)}{q}^2\right] - P_{HS}\left[\frac{n_t(HS)^2}{q^2} + n_t(HS)^2\right]\right\}}{\partial \, n_t(HS)} = 0$$

$$\Rightarrow 2n_t(HS) * P_{HS}\frac{q^2+1}{q^2} = \alpha_{HS} - \frac{4*n_t(HS)}{q}$$

$$\Rightarrow \alpha_{HS} = [2 * P_{HS}\frac{q^2+1}{q^2} * n_t(HS)] + \frac{4*n_t(HS)}{q}$$

$$\Rightarrow \alpha_{HS} = n_t(HS) * \frac{2*P_{HS}(q^2+1)+4q}{q^2}$$

Thus, $n_t(HS) = \frac{\alpha_{HS} * q^2}{2*P_{HS}(q^2+1)+4q}$

For Medium Severity bugs:

$$\frac{\partial \left\{n_t(MS)\left[\alpha_{MS} - 2 * n(MS)\right] - P_{MS}\left[n(MS)^2 - n_t(MS)^2\right]\right\}}{\partial \, n_t(MS)} = 0;$$

As $n_t(MS) = q * n(MS)$,

Thus, $n(MS) = \frac{n_t(MS)}{q}$

$$\frac{\partial \left\{ n_t(MS)\left[\alpha_{MS} - \frac{2*n_t(MS)}{q}^2\right] - P_{MS}\left[\frac{n_t(MS)^2}{q^2} + n_t(MS)^2\right]\right\}}{\partial \, n_t(MS)} = 0$$

$$\Rightarrow 2n_t(MS) * P_{MS}\frac{q^2+1}{q^2} = \alpha_{MS} - \frac{4*n_t(MS)}{q}$$

$$\Rightarrow \alpha_{MS} = [2 * P_{MS}\frac{q^2+1}{q^2} * n_t(MS)] + \frac{4* n_t(MS)}{q}$$

$$\Rightarrow \alpha_{MS} = n_t(MS) * \frac{2*P_{MS}(q^2+1)+4q]}{q^2}$$

Thus, $n_t(MS) = \frac{\alpha_{MS} * q^2}{2*P_{MS}(q^2+1)+4q}$

For Low Severity bugs:

$$\frac{\partial \{ n_t(LS)[\alpha_{LS} - 2* n(LS)] - P_{LS}[n(LS)^2 - n_t(LS)^2]}{\partial n_t(LS)} = 0;$$

As $n_t(LS) = q * n(LS)$,

Thus, $n(LS) = \frac{n_t(LS)}{q}$

$$\Rightarrow \frac{\partial \{ n_t(LS)\left[\alpha_{LS} - \frac{2*n_t(LS)^2}{q}\right] - P_{LS} * \left[\frac{n_t(LS)^2}{q^2} + n_t(LS)^2\right]\}}{\partial n_t(LS)} = 0$$

$$\Rightarrow 2n_t(LS) * P_{LS}\frac{q^2+1}{q^2} = \alpha_{LS} - \frac{4*n_t(LS)}{q}$$

$$\Rightarrow \alpha_{LS} = [2 * P_{LS}\frac{q^2+1}{q^2} * n_t(LS)] + \frac{4* n_t(LS)}{q}$$

$$\Rightarrow \alpha_{LS} = n_t(LS) * \frac{2*P_{LS}(q^2+1)+4q}{q^2}$$

Thus, $n_t(LS) = \frac{\alpha_{LS} * q^2}{2*P_{LS}(q^2+1)+4q}$

$$\boxed{\text{Optimal number of bugs caught} = \frac{q^2 * \alpha_X}{2*P_X(1+q^2)+4q}}$$

Thus, for a tester, the optimal number of anybug of type X to be caught is given by-

Developer Strategy

Similarly, we can go ahead and calculate the point where the developers can maximize their payoffs.

To maximize the payoff P_d, we put break the equation into 3 parts and then differentiate in terms of the severity of errors.

$$\frac{\partial \text{ Payoff (tester)}}{\partial n_t(HS)} = 0;$$

$$\frac{\partial \text{ Payoff (tester)}}{\partial n_t(MS)} = 0;$$

$$\frac{\partial \text{ Payoff (tester)}}{\partial n_t(LS)} = 0;$$

For High Severity bugs:

$$\frac{\partial \{\beta_{HS} * n(HS) - [(s + P'_{HS}) * \Upsilon_{HS}^2 * n_t(HS)^2\}}{\partial(n(HS))} = 0$$

On using $n_t(HS) = q * n(HS)$

$$\Rightarrow \frac{\partial\,\{\beta_{HS} * n(HS) - [(s + P'_{HS}) * Y_{HS}^2 * q^2 * n(HS)^2]\}}{\partial(n(HS))} = 0$$

$$\Rightarrow \beta_{HS} - 2*(s + P'_{HS}) * Y_{HS}^2 * q^2 * n(HS) = 0$$

$$\Rightarrow \beta_{HS} = 2*(s + P'_{HS}) * Y_{HS}^2 * q^2 * n(HS)$$

Thus, $n(HS) = \frac{\beta_{HS}}{2*(s + P'_{HS}) * Y_{HS}^2 * q^2}$

For Medium Severity bugs:

$$\frac{\partial\,\{\beta_{MS} * n(MS) - [(s + P'_{MS}) * Y_{MS}^2 * n_t(MS)^2\}}{\partial(n(MS))} = 0$$

On using $n_t(MS) = q * n(MS)$

$$\Rightarrow \frac{\partial\,\{\beta_{MS} * n(MS) - [(s + P'_{MS}) * Y_{MS}^2 * q^2 * n(MS)^2]\}}{\partial(n(MS))} = 0$$

$$\Rightarrow \beta_{MS} - 2*(s + P'_{MS}) * Y_{MS}^2 * q^2 * n(MS) = 0$$

$$\Rightarrow \beta_{MS} = 2*(s + P'_{MS}) * Y_{MS}^2 * q^2 * n(MS)$$

Thus, $n(MS) = \frac{\beta_{MS}}{2*(s + P'_{MS}) * Y_{MS}^2 * q^2}$

For Low Severity bugs:

$$\frac{\partial\,\{\beta_{LS} * n(LS) - [(s + P'_{LS}) * Y_{LS}^2 * n_t(LS)^2\}}{\partial(n(LS))} = 0$$

On using $n_t(LS) = q * n(LS)$

$$\Rightarrow \frac{\partial\,\{\beta_{LS} * n(LS) - [(s + P'_{LS}) * Y_{LS}^2 * q^2 * n(LS)^2]\}}{\partial(n(LS))} = 0$$

$$\Rightarrow \beta_{LS} - 2*(s + P'_{LS}) * Y_{LS}^2 * q^2 * n(LS) = 0$$

$$\Rightarrow \beta_{LS} = 2*(s + P'_{LS}) * Y_{LS}^2 * q^2 * n(LS)$$

Thus, $n(LS) = \frac{\beta_{LS}}{2*(s + P'_{LS}) * Y_{LS}^2 * q^2}$

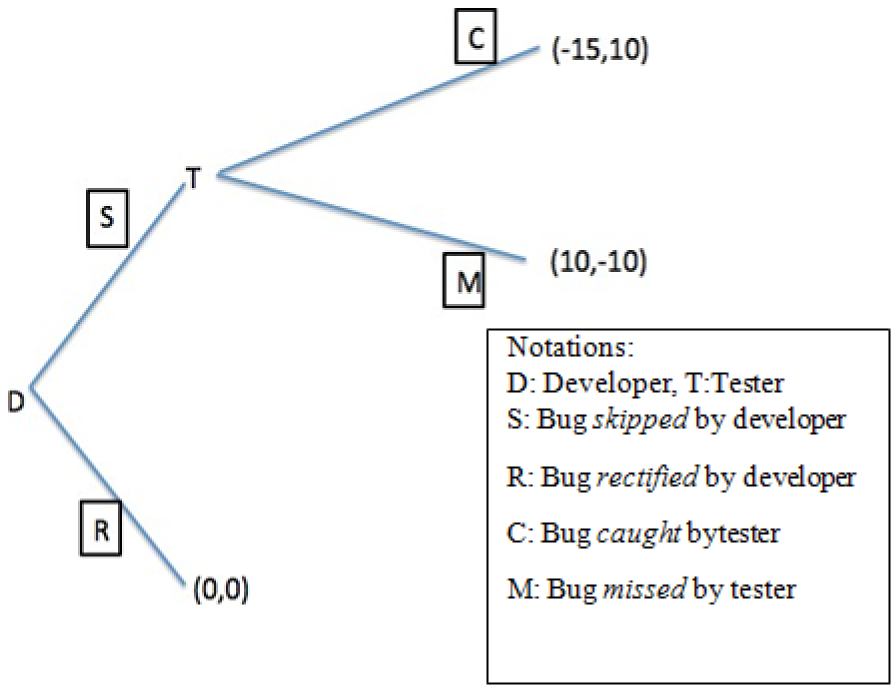
Thus, for a tester, the optimal number of any bug of type X to be caught is given by-

$$\boxed{\text{Optimal number of bugs corrected} = \frac{\beta_X}{2*(s + P'_X) * Y_X^2 * q^2}}$$

4. THE REAL-LIFE DILEMMA

In real-life, software firms can incur heavy loss in terms of monetary value and market perception if the software provided by them to the client contains many bugs. Therefore, testing exists: to identify and rectify the issues beforehand, and to provide client with a near- perfect software. The game which was depicted in Section 3 was not plugged in with numbers. Instead, we talked about payoffs in terms of different variables. In real-life, utility/penalty coefficients such as αx, βx, Px, Px', and s, have values which deter the programmers and testers from checking in inefficient codes and missing bugs

respectively. The company consists of both: testers and developers, and thus it would be optimal for a company to maximize the combined benefit that it can extract from both, instead of maximizing the payoff of a single entity. But this comes only after minimizing the bug count as it has even worse consequences for the firm as discussed above. Let us design a game similar to the one played in section 2 and 3, albeit with only a single bug left in the code at hand with the developer. This one-shot game can help us in gauging the state of maximum benefit for the company, besides having a look at the Nash equilibria that might occur. We keep rest of the assumptions same, and the penalty versus bug count curve remains a parabola (depicting exponential marginal loss with increasing bug count). In this example, we takeαx
= βx = Px = Px' =10; and s=5

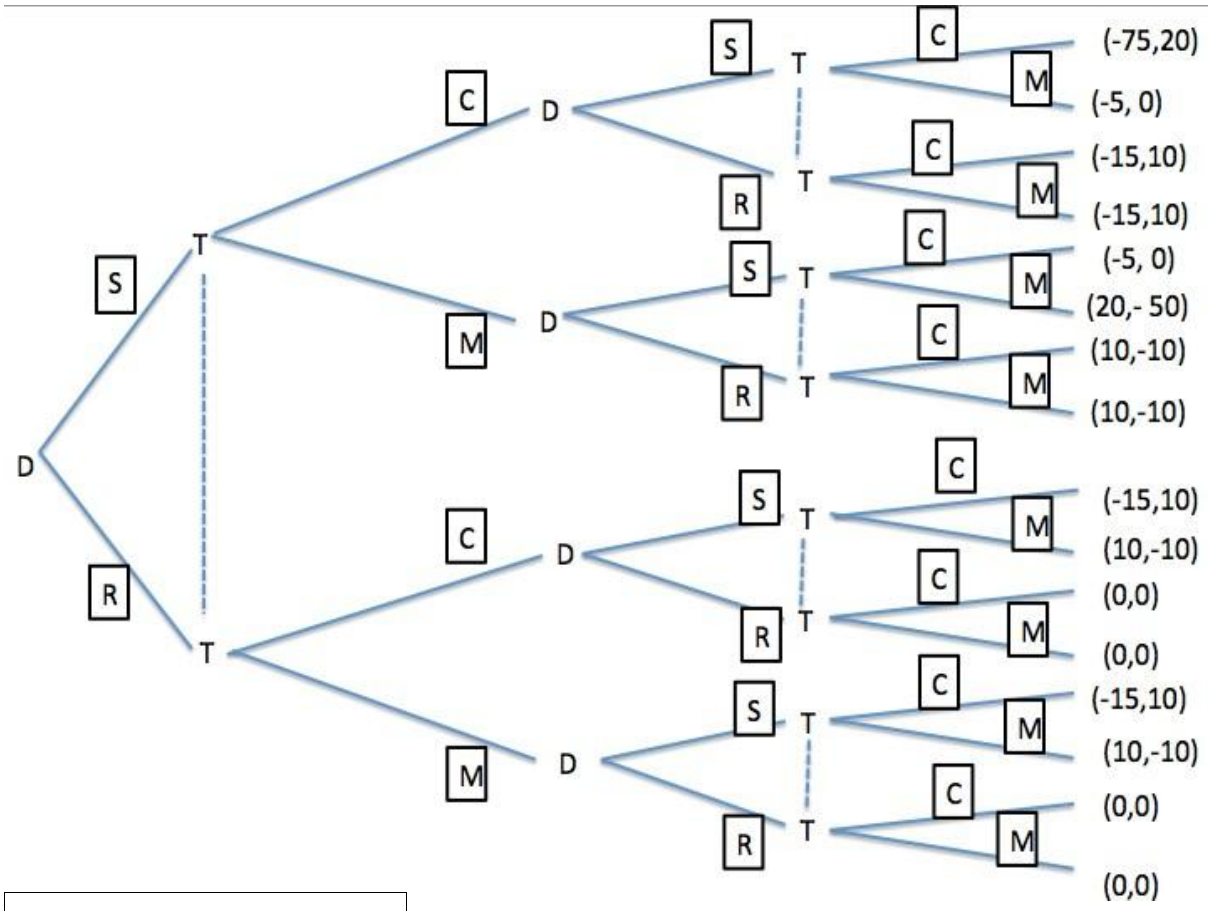

C
(-15,10)
T
S
(10,-10)
M
D
R
(0,0)
Notations:
D: Developer, T:Tester
S: Bug *skipped* by developer
R: Bug *rectified* by developer
C: Bug *caught* bytester
M: Bug *missed* by tester

(Figure 1) A one-shot tester-developer game

(Table 1) Normal form representation of extensive form game in Figure 1

Tester (/Developer ( )	S	R
C	(-15,10)	(0,0)
M	(10, -10)	(0,0)

As evident from the normal form, the Nash equilibria of this game is in the state (R, C) (as shown in Table 1). It is also the state in which the maximum combined payoff occurs for a firm. Let us see what happens if we extend this game to one more bug (one more period).



Notations:

D: Developer, T:Tester

S: Bug *skipped* by developer

R: Bug *rectified* by developer

C: Bug *caught* bytester

M: Bug *missed* by tester

(Figure 2) A two-shot tester-developer game

(Table 2) Normal form representation of extensive form game in Figure 2

	SS	RS	SR	RR
CC	(-75,20)	(-15,10)	(-15,10)	(0,0)
CM	(-5,0)	(-15,10)	(10,10)	(0,0)
MC	(-5,0)	(10, -10)	(-15,10)	(0,0)
MM	(20, -50)	(10, -10)	(10, -10)	(0,0)

Similar to the previous game, this two-shot game also has its N ash equilibriain the state (RR, CC) which happens to be the group payoff maximizing state (refer Table 2). On extending this game further, the Nash equilibria is (R_n, C_n) where the subscript *'n'* denotes the number of iterations of the attached alphabet. This shows that the best strategy for a company is to try and minimize the bugs left by developer while operating the testers at full vigil to maximize its profit. How certain situations can result in changes to this or reinforce this is discussed in further sections

5. SECTION 5: INSIGHTS

INSIGHT 1: For a firm which is more focused on the quality of software it delivers to its clients; the optimal strategy is to reduce the time burden on the developers.
REASON: From the equation depicting optimal number of bugs corrected by developer in the first iteration itself, we get-

$$\textit{Optimal number of bugs passed on by the developer} \propto \beta_x$$

As we relax the time constraint, the developer would gain lesser benefits from time savings as β_x reduces and thus, he would check in lesser errors into the module, thereby increasing software quality.

INSIGHT 2: A firm which has better quality of testers faces lesser inefficiencies in code not only because the testers are good, but also since developers are more apprehensive of passing on bugs due to their beliefs about the testers.
REASON: From the equation depicting optimal number of bugs corrected by developer in the first iteration itself, we get-

$$\textit{Optimal number of bugs passed on by the developer} \propto \frac{1}{Y_X^2}$$

As Y_x increases, the code in efficiencies checked in by developers decreases.

INSIGHT 3: The number of inefficiencies checked in by developers decreases as the testing schedule gets more stringent.
REASON: From the equation depicting optimal number of bugs corrected by developer in the first iteration itself, we get-

$$\textit{Optimal number of bugs passed on by the developer} \propto \frac{1}{q^2}$$

As the number of test cases that are executed increases, the inefficiencies checked in by developers falls

6. RANDOMIZATION LEVELS AND PAYOFFS

Software companies often deploy standard test suites which consist of several test scripts which test the code against the most crucial and vulnerable errors which can be present. In such a case, the developer has full knowledge of the testing conditions which can be evaluated. As discussed in the previous section, a stringent testing schedule means lesser errors in the software. But there lies the problem: the more stringent a testing schedule gets; the more cost is incurred by the firm. The testing resources are limited, and there are a lot of modules to be tested. This means that a judicious distribution of resources is of paramount importance. A schedule can be randomized to mitigate this problem. This means that instead of using a fixed script, a tester can randomize the test cases to mitigate the pre-existing beliefs of developers about the test scripts, thus catching more bugs.

Now the question arises that what should be the optimal randomization level? A randomization level of 0 means a fully known script to the developer, which is not something desirable. A randomization level of 1 means that the test schedule is completely randomized, which in turn means that the developer can't breach the system ashe has no prior belief about the test cases. But it also means that the test cases will miss out on some very important cases due to complete randomization, and this can wreak havoc on the program. Thus, an optimal balance needs to be maintained during the selection of randomization schedule in order to sufficiently mitigate the beliefs of developers while also taking into account the important cases which must be tested.

References

1. *Beizer, B. (2009). Software Testing Techniques (2 ed.). Delhi, Delhi, India: dreamtech.*
2. *Eduardo Mattos, M. V. (2014). Applying Game Theory to the Incremental Funding Method in Software Products. Journal of Software, 9, 1435-1443.*