Towards Effective Adoption of Security Practices

ABSTRACT

Security tools guide software developers to identify potential vulnerabilities in their codes. However, the use of security tools is not common among developers. The goal of this research is to develop a framework for modeling the adoption of security practices in software development, and explore various sanctioning mechanisms in promoting greater adoption of these practices among developers. We propose a multiagent simulation framework that incorporates developers and manager roles, where developers try to maximize task completion and security compliance and the manager enforces sanctions based on functionality and security of the task outcome. The adoption of security practices emerge through the interaction of manager and developer agents in time-critical projects. Using the framework, we evaluated the adoption of security practices for developers with different preferences and strategies under individual and group sanctions. The model will improve understanding of the adoption of security tools and provide guidance to identify appropriate sanctioning mechanisms for increasing the adoption in software development.

ACM Reference sanctioning format:


DOI: 10.1145/nmnmnn.nnmmnn

1 INTRODUCTION

Secure software development tools, or security tools, are programs that analyze software to help developers find and fix vulnerabilities[1]. Using security tools, software developers can identify and fix potential vulnerabilities in their codes. However, the use of security tools is not common among developers [1], and sanctions are a way to enforce adoption of security practices [2]. This research develops a framework to (1) simulate sanctions in software development environments [3], and (2) assess the use of sanctions to promote learning and use of security practices among developers.

Adoption of security practices. Previous research that looked into the adoption of security practices are based on quantitative data collected through surveys. Witschey et al.[1] collected quantitative data on the relative importance of factors through an online survey of software developers who were recruited from 14 companies and five mailing lists. They identified 39 factors: social system factors such as security concern and awareness, policies and standards, structures, education and training, cultures; innovative factors such as relative advantage, observability, complexity, and trialability; communication channel factors such as trust and exposure; and potential adopter factors. They built a combined logistic regression model using all 39 factors and found seven were statically significant including the intercept. The seven are observability, advantages, policies, inquisitiveness, education and exposure. While Surveys give us an idea of the factors influencing the adoption, it is logistically challenging to observe how each developer works in real time and simulation could help us foresee adoption in different scenarios. From a decision theory perspective, the factors can be viewed as a developer’s individual preferences and perceived utility from using security practices. While a developer will only prefer using security tools if it leads to a rationally rewarding outcome for his or her individual utility, a manager who oversees the adoption and the overall quality of the outcome may offer rewards or punishments to promote the adoption, because the security of the end product depends on it. We propose a rational decision making framework to explain the adoption of security practices by developers. The model simulates individual developers’ preferences for inquisitiveness and decision making based on their perceptions of the advantages of security tools. The model also includes a manager who has full observability of the adoption practices and follows a sanction mechanism that underlines the policy to use security practices to met a specified standard.

Norms and Sanction. We recognize the term, norms, to describe “directed normative relationships between participants in the context of an organization” [2, 4]. Norms are a powerful means for regulating interactions among autonomous agents[5–7]. Failure to comply with normative expectations is met with a sanction (consequence for norm violation applied to a principal or group of principals) by a sanctioning agent. A sanction may be positive or negative and manifested in reprimand or reward, respectively [8]. In the context of the developers’ adoption environment, a norm violation would be met with negative sanction. When an individual principal is singled out and censured for defecting against a norm, we recognize this as individual sanction. Alternatively, when a sanction is applied to a group of individuals for the actions of some subset of that group, we recognize this as group sanction, also known as collective sanction [9].

Example 1.1. Consider Alex, Bob, Charlie and Dave, who are software developers working as a team to deliver a product. Erin is their manager. Erin divides the project into multiple tasks and assigns those to the developers. Erin wants to make sure the product is delivered in time and meets functionality and security requirements. Erin and her team use certain security tools to identify security bugs. Alex and Bob are experienced developers who use security tools. Charlie and Dave are new to the team and not aware of the security practices.

Consider a scenario where, Charlie and Dave learn to use security tools. Everyone follows the standard security practices and delivers the product on time. Erin evaluates the product’s functionality and
security and finds those as satisfactory. She rewards her team and encourages everyone to continue following security practices.

Now consider a second scenario: the project is time-critical, and Charlie skips executing security tools to deliver the product with functionality. This can lead to multiple alternatives: (1) Erin identifies that Charlie did not follow the standard security practices and scolds him. Charlie realizes the importance of using security tools. (2) Erin evaluates the product and finds the security unsatisfactory. She scolds her team to use security tools in future. (3) Alex finds out that Charlie is not following the security practices and prompts him to use security tools. The illustrative example can be used to distinguish among individual, group and peer sanctions. Erin sanctioning Charlie is an example of individual sanction where the manager monitors and sanctions individuals who does not follow a norm. Erin scolding everyone can be considered as a group sanction where the sanctions are imposed on a whole group regardless of who in the group violated or satisfied a norm. Alex prompting Charlie, is an example of peer sanction where a peer sanctions another peer.

Contributions. The research addresses the following research question “Which sanctioning mechanism promotes greater adoption of security practices?” We provide a model that will improve understanding of the adoption of security tools among developers. The model can be applied to identify appropriate sanctioning mechanisms for increasing the use of security tools among a group of developers with heterogeneous skill levels. Model output demonstrates the emergence of the adoption and use of security tools by simulating the system dynamics as the interactions among developers and a manager in the completion of project tasks.

Organization. Section 2 describe our security practices adoption model. Section 3 details the simulation experiments and its results. Section 5 concludes with important future directions.

2 SECURITY PRACTICES ADOPTION MODEL

We develop a framework that simulates the dynamic interactions between developers and a manager in a time critical project. The multiagent system model contains three types of entities:

- A lab, or an organization, wherein agents perform tasks.
- A is a set of developer agents who perform tasks pertaining to a time-critical project. Each task corresponds to an artifact of a product \( P \).
- \( M \) is a manager who monitors the coding and security practices of the agents and sanctions agents based on the functionality and security of the product.

The following sections describe the attributes and actions of the Manager and developer agents.

2.1 Manager

Manager agent is in charge of assigning tasks and sanctions. Thus its responsibilities are to:

(1) Assign Tasks

A task can be in one of the three states: NotCoded, Coded, Tested. NotCoded indicates the task has not yet been adequately coded, Coded represents an agent performed Code action on the task but not tested for bugs, and Tested indicates that an agent performed Security action on the task and no bugs are found (Figure 1). A task has the following attributes:

- Minimum skill required to code: Each task has an associated minimum skill required to code. If a developer’s coding skill is higher than that required skill, it can perform Code action on that task. A developer can perform LearnCode action to increase skill of coding.
- Minimum skill required to run security tools: Each task has an associated minimum skill required to run security tools. If a developer’s security skill is higher than that required skill, it can perform Security action on that task. A developer can perform LearnSecurity action to increase skill of using security tools.
- Time required to code: Each task has an associated time required to code. The time refers to time taken for a coding skill equal to the average skill of coding of all developers and is inversely proportional to the coding skill of a developer.
- Time required to run security tools: Each task has an associated time required to run security tools. The time refers to the time taken for security skill equal to the average skill of security of all developers and is inversely proportional to the security skill of a developer.

The tasks are assumed to be independent, meaning the coding and testing of one task are not dependent on the completion of other tasks. Each task corresponds to an artifact. An artifact, \( \text{Art}_j \), corresponding to task \( T_j \), has two attributes. Functionality of the artifact is simulated as a function of the skill of coding of the corresponding agent. Security of the artifact is a function of the skill of using security tools of the agent, whether or not Security action is performed, and whether bugs are found in the artifact.

Figure 1: Change in task’s states under developer’s actions

(2) Assign Sanctions

Sanctions can be Individual and group sanctions. In individual sanctioning, developers are subjected to sanctions individually, if they fail to deliver artifacts with the standard set by manager. In a group sanction, the manager only monitors the overall product and sanctions every developer if the product does not meet the standard. Sanctions are applied in the following two cases:

- For functionality: The manager monitors functionality of the artifacts periodically and imposes sanctions for individual agents produced artifacts with less functionality than the standards set by manager (individual sanction) or all
developers if the overall product functionality is less than project goal (group sanction).

- For security: Similar to functionality, the manager monitors security of the artifacts periodically and imposes individual or group sanctions.

### 2.2 Developer Agents

Each developer agent has five attributes:

1. A set of tasks assigned by the manager agent.
2. A deadline, or the amount of time a developer is allowed to complete the tasks. The deadline also marks the end of a project.
3. Skill of coding and skill of running security tools. Each task has a corresponding minimum skill. The time taken to code or run security tools decreases with an increase in skill, and developers can increase their skill by performing LearnCode or LearnSecurity actions.
4. Agent Health represents a developer agent’s health and corresponds to the completion of the tasks assigned to the agent. The initial value is 100, which is the maximum health value. Agent health is subject to change in the simulation for the following reason: if an agent is unable to meet the functionality and security standards within the deadline, health is reduced proportionally. We define 60 as the threshold that triggers an agent’s death. If an agent leaves, he or she is no longer able to perform any action and will never return to the simulation [4].
5. Preference is the probability that an agent will choose to do an action [4]. For example, if an agent’s preference is 40% for coding, 20% for using security tools, 20% for learning coding and 20% for learning security tools, he or she will have an 40% of chance of considering the coding task and 20% of chance of considering the security, learn-code, and learn-security tasks. The sum of the preferences is unity.

An agent can choose to perform one of the actions (Code ∪ Security ∪ LearnCode ∪ LearnSecurity ∪ DoNothing) at each time step.

- **Action Code**: A task changes from NotCoded to Coded state when the assigned agent performs Code action on the task.
- **Action Security**: When the corresponding agent performs Security action, the task state changes from Coded to Tested if no bugs are found and to NotCoded if bugs are found.
- **Action Learn Code**: Action LearnCode increases the skill of coding of an agent, which decreases the time for Code action and increases the functionality of the corresponding artifact.
- **Action Learn Security**: LearnSecurity increases the skill of testing of the agent which decreases time for Security action and increases the security of the corresponding artifact.
- **Action Do Nothing**: A rational agent may DoNothing if there is any reward for that action or all the tasks in a project are Coded and Tested.

### 3 SIMULATION AND EVALUATION

#### 3.1 Assumptions and Settings

We have made several assumptions in our experiment. We do not trivialize the significance of these variables, nor do we recognize them as arbitrary. In future work, we intend to determine valid replacements through further research, experimentation, and data collection. However, as the nature of our simulation is exploratory, we have intuitively assigned values to these variables, which are held constant throughout all treatments and runs of the simulation. These assumptions and other related parameters are as follows:

- The manager has a perfect knowledge of the system. A developer has knowledge only about tasks assigned to it and its own attributes.
- Developers perform tasks in the order they are assigned. A developer considers coding a task only if its prior task is Coded.
- The skill of coding and the skill of running security tasks of a developer are independent. Each skill varies in the range of [0, 100].
- The expected reward of running security tools on a coded task is higher than the expected reward of coding a task.
- The manager only assigns negative sanctions based on functionality and security compliances.

#### 3.2 Runtime Actions

The total tasks are distributed equally among the active developers. The deadline for the completion of tasks is equal to the time required for coding and running security tasks for all the tasks for a developer with average skill of coding and running security tools. At each tick, the following actions occur:

1. Each developer identifies the available actions to perform at that time tick.

- **Action Code** is available, if there is any task in the NotCoded state and the corresponding developer’s skill of coding is equal to or higher than the required skill for the task.
- **Action Security** is available, if there is any task in Coded state and the corresponding developer’s skill of running security tools is equal or higher than the required skill for the task.
- **Action Learn Code** is available, if the corresponding developer’s coding skill is less than 100.

---

**Algorithm 1: Developers decision and task completions**

```python
1: Function DevDecision (developers, Tasks, fDecision, fChangeState)
2: for all developers do
3: if decision is to learn then
4:     skill+= IncreaseSkills (Skills, skill, fChangeState);
5: else if decision is to code then
6:     Tasks ← CodeTasks (Tasks, fChangeState);
7: else if decision is to test then
8:     Tasks ← TestTasks (Tasks, Bugs, fChangeState);
9: end for

def IncreaseSkills(Skills, skill, fChangeState):
    skill+= fChangeState * skill
    return skill

def CodeTasks(Tasks, fChangeState):
    # Code tasks...
    return Tasks

def TestTasks(Tasks, Bugs, fChangeState):
    # Test tasks...
    return Tasks
```

---
• Action Learn Security is available, if the corresponding developer’s skill of running security tool is less than 100.

(2) Each developer compares the expected reward of each action
• Expected reward of Action Code is calculated by multiplying units of task that can be completed with current coding skill within time remaining, by the reward for coding a unit task.
• Expected reward of Action Security is calculated by multiplying units of task that can be tested with the current security skill within the time remaining, by the reward for testing a unit task.
• Expected reward of Action Learn Code is calculated by multiplying the units of task that can be coded if the current time step is spent on learning coding, i.e., units of tasks that can be coded with an increased skill within (time remaining -1) time steps by the reward for coding unit task.
• Expected reward of Action Learn Security is calculated by multiplying the units of task that can be tested if the current time step is spent on learning security tools, i.e., units of tasks that can be tested with an increased skill within (time remaining -1) time steps by the reward for testing a unit task.

(3) Each developer identifies its action. The action that has the highest product of preference and expected reward, is identified by the developer (function $f_{\text{decision}}$ in algorithm 1). The state of each developer is updated (function $f_{\text{change state}}$) as follows:
• If Action Code is chosen, developer remains busy for time steps equal to the time required for coding that task. After that, the state of the task changes from NotCoded to Coded. Once a task state changes from NotCoded to Coded state, the functionality of the corresponding artifact is generated as a random number in the range of [skill of coding, 100] and the security of the artifact in the range of [skill of security, 100].
• If Action Security is chosen, developer remains busy for time steps equal to the time required for testing that task. After that, the state of the task changes from Coded to Tested if no bugs are found and to NotCoded state if bugs are found. The probability of finding a bug is a random number in the range [0, (100- security skill)]. Once a task state changes from Coded to Tested, the security of the corresponding artifact is increased by a percentage of the skill of security of the corresponding developer.
• If Action Learn Code or Learn Security is chosen, the corresponding skill of the developer increases.

(4) When time step is equal to deadline, a project ends and the manager sanctions based on compliance with timeliness, functionality and security (Algorithm 2). When a developer is sanctioned for timeliness, its preference for Code action increases, when a developer is sanctioned for functionality, its preference for Code action and Learn-Code action increases, and when a developer is sanctioned for security, its preference for Security action and Learn-Security action increases. A new project begins at the end of a project.

Algorithm 2: Managers sanction

1: Function $f$ (developers, Artifacts, threshold, Stype, $f_{\text{sanction}}$)
2: violations $\leftarrow$ SanctionCheck (artifacts, threshold, $f_{\text{sanction}}$);
3: for all artifacts in violations do
4: if developer is sanctioned then
5: preferences $\leftarrow$ UpPref (preferences, Stype, $f_{\text{sanction}}$);

3.3 Scenarios and Metrics

First, we compare the performances of three different types of developers. The first type of developer always select code if coding is an option. It will learn coding only if learning is required. Once all the tasks are coded, it will perform security actions and learn security tools only if learning is required. The second type of developer codes a task and runs security tests immediately when the coding of the task is complete. It will learn coding or running security tools as required. The developer will start coding the next task when the previous task is coded and tested. The third type of developers will learn first. The developer will learn coding to reach the maximum coding skill and then learn security to reach the maximum security skill. Once their skills are at maximum levels, they will code and run security tests. We compared the sanction, sanction efficacy and overall group health for these three types of developers. Because they have predefined strategies, the developers don’t learn or change actions in response to sanctions.

Next we considered a group of developers who learn and update their preferences in response to sanctions. We compared three group of developers. The first group of developers has no preferences, i.e., the preferences for Code, Security, LearnCode and LearnSecurity actions are equal (equal to 25). The second group of developers have higher preference for coding, i.e., the preferences for Code action (55) is higher than the actions Security, LearnCode and LearnSecurity (15). The third group of developers have higher preference for running security tools, i.e., the preferences for Security action (55) is higher than the actions Code, LearnCode and LearnSecurity (15). The following four metrics are measured over the course of the simulation:

• Tasks tested: Tasks tested (%) is the percentage of tasks in a project in which a developer performed Security action and no bugs are found. In other words, tasks tested (%) is the ratio of tasks in tested state and total tasks, measured at deadline.
• Time spent on security tasks: Time spent on security tasks (%) is the total time steps spent by all developers on Security and LearnSecurity actions, as a percentage of the total time steps of the project.
• Sanctions: Sanctions (%) is the number of sanctions applied by the manager as a percentage of total number of sanctions in the simulation.
• Sanction efficacy: Sanction efficacy (%) is the measure that if an agent is sanctioned once, how often it is sanctioned
again. sanction efficacy is calculated as the the ratio of the number of developers that are sanctioned once and the number of developers sanctioned in the simulation.

4 EVALUATION

4.1 Fixed vs Adaptive Developers

We compared the performances of developers with different fixed strategies and preferences. Among the fixed strategies, developers who learn coding or run security tests first and then perform code and security actions has the highest percentage of tasks (60%) tested. This group of developers also has the highest percentage (44.5%) of security activities among the three strategies compared. The group of developers who code or learn coding first and then performs security or learn security actions has the highest (87%) of tasks coded but also the lowest (7%) of tasks tested. This group also has the lowest security activities (7.6%).

Among the adaptive developers, i.e., developers that update their preferences when sanctioned; developers with no preferences or preferences for testing have a higher percentage of tasks tested (19 and 20%) and security activities (37%) compared to developers with preference for coding (4% tasks tested and 8% security activities)

4.2 Preferences

Among the three developer groups with different preferences, developers with preferences for testing had the highest security practices and tasks tested. Developers with preferences for coding had the lowest security practices and tasks tested. This group of developers also had the highest sanctions with lowest sanction efficacies across the different sanction types.

While simulations for developers with no preferences and preferences for testing retains all the developers for the complete simulation, only 70(%) developers remain active within the developers with security preferences at the end of the simulation. The remaining developers become inactive due to repetitive sanctions and non-compliance.

Table 1: Task completions and security activities in different strategies

<table>
<thead>
<tr>
<th>Sanctions</th>
<th>For Functionality</th>
<th>For Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Individual</td>
<td>Group</td>
</tr>
<tr>
<td></td>
<td>Sanction (Efficacy)</td>
<td>Health</td>
</tr>
<tr>
<td>Code→LearnCode→Test→LearnTest</td>
<td>52 (30)</td>
<td>88</td>
</tr>
<tr>
<td>Test→LearnTest→Code→LearnCode</td>
<td>98 (00)</td>
<td>80</td>
</tr>
<tr>
<td>LearnCode→LearnTest→Code→Test</td>
<td>40 (00)</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 2: Sanctions for different fixed strategies

<table>
<thead>
<tr>
<th>Sanctions</th>
<th>For Functionality</th>
<th>For Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Individual</td>
<td>Group</td>
</tr>
<tr>
<td></td>
<td>Sanction (Efficacy)</td>
<td>Health</td>
</tr>
<tr>
<td>Code→LearnCode→Test→LearnTest</td>
<td>68 (20)</td>
<td>86</td>
</tr>
<tr>
<td>Test→LearnTest→Code→LearnCode</td>
<td>56 (40)</td>
<td>86</td>
</tr>
<tr>
<td>LearnCode→LearnTest→Code→Test</td>
<td>40 (00)</td>
<td>86</td>
</tr>
</tbody>
</table>

for developers with no preferences, 2% for preferences for coding and by 1% for preferences for testing. While sanctioning individually for functionality compliances improved tasks tested for developers with no preferences, there was not a positive effect on developers with preferences for coding or testing. Comparing the different sanctioning, group sanctioning for security has the lowest sanctions with higher sanction efficacy whereas, individual sanctioning for security has the highest number of sanctions with the lowest efficacy.

For the fixed strategies considered, developers who would code or learn code first and then perform security or learn security actions, are sanctioned highest (68%) for individual sanctioning for security condition, followed by individual sanctioning for functionality (52%). For developers who would perform security or learn security actions first and then code and learn code actions, all four sanctioning conditions reaches maximum or nearly maximum (98-100%), except individual sanctioning for security, which has lower (56%) sanctions with higher health (86). Developers who perform learn actions first and then code and security actions have the lowest sanctions (40%), as they are sanctioned initially while learning, followed by full compliance with security and functionality in later projects.

5 DISCUSSION

Our framework aims to improve the understanding of adoption of security practices among developers. It compares the security
Table 3: Task completions and security practices for different preferences

<table>
<thead>
<tr>
<th>Performance</th>
<th>No sanction</th>
<th>For functionality</th>
<th>For security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Indiv Group</td>
<td>Indiv Group</td>
<td></td>
</tr>
<tr>
<td>Developers with no preferences</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks tested (%)</td>
<td>19 22 20 19</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Time spent on security tasks (%)</td>
<td>37 37 37 35</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>Developers with preference for coding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks tested (%)</td>
<td>4 1 4 0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Time spent on security tasks (%)</td>
<td>8 3 7 0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Developers with preference for testing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks tested (%)</td>
<td>20 18 20 16</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Time spent on security tasks (%)</td>
<td>37 37 37 33</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Sanctions for different preferences

<table>
<thead>
<tr>
<th>Performance</th>
<th>No sanction</th>
<th>For functionality</th>
<th>For security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Indiv Group</td>
<td>Indiv Group</td>
<td></td>
</tr>
<tr>
<td>Developers with no preferences</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanctions (%)</td>
<td>- 20 20 50 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanction efficacy (%)</td>
<td>- 100 100 40 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Developers with preference for coding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanctions (%)</td>
<td>- 44 20 66 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanction efficacy (%)</td>
<td>- 70 100 30 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Developers with preference for testing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanctions (%)</td>
<td>- 20 20 46 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanction efficacy (%)</td>
<td>- 100 100 30 100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

practices under different sanctioning mechanism at different level by comparing tasks that are tested using security tools, time spent on security practices and overall changes under sanctions. It also assists in comparing functionality and security compliance in different projects for different sanctions, by (1) analyzing the recurrence of norm violation after sanctioning, referred to as sanction efficacy, and (2) monitoring developers’ health under different sanctioning mechanism. Our exploration of system level performance through variable sanction type has yielded some interesting conjectures. For example, group sanctioning for security practices, yields better performances, whereas, individual sanctioning results in lower retention of developers. In the future work, we plan to further develop our scenarios and the accompanying simulation in order to conduct more in-depth research on norm-governance in multiagent systems. Incorporation of multidisciplinary concepts from decision theories and measuring resilience and liveliness of the system in connection to the sanctions, among others, will help to get a holistic view of the system to guide effective sanctions towards security practices.

Acknowledgment
We thank the US Department of Defense for support through the Science of Security Lablet grant to North Carolina State University.

APPENDIX

Table 5: Experiment parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No of projects</td>
<td>5</td>
</tr>
<tr>
<td>No. of Developers</td>
<td>10</td>
</tr>
<tr>
<td>Tasks per project</td>
<td>50</td>
</tr>
<tr>
<td>Project duration</td>
<td>55</td>
</tr>
<tr>
<td>Increase in skill by learning</td>
<td>6</td>
</tr>
<tr>
<td>Threshold for active developers</td>
<td>60</td>
</tr>
<tr>
<td>Preference for coding preferred</td>
<td>55</td>
</tr>
<tr>
<td>Preference for other action for coding preferred</td>
<td>15</td>
</tr>
<tr>
<td>Preference for testing preferred</td>
<td>55</td>
</tr>
<tr>
<td>Preference for other action for testing preferred</td>
<td>15</td>
</tr>
<tr>
<td>Preference for all actions for no preferences</td>
<td>25</td>
</tr>
<tr>
<td>Maximum skill</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6: Variables with normal distribution

<table>
<thead>
<tr>
<th>Variables</th>
<th>µ</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time required to code a task</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Time required to test a task</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Skill of developers (initialization)</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>Skill required for a task (initialization)</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>Health of developers (initialization)</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
REFERENCES


