Turning Real-World Software Development into a Game

Erick B. Passos, IFPI Lims
Danilo B. Medeiros, Infoway Sol. Inf.
Pedro A. S. Neto, UFPI
Esteban W. G. Clua, UFF Media Lab

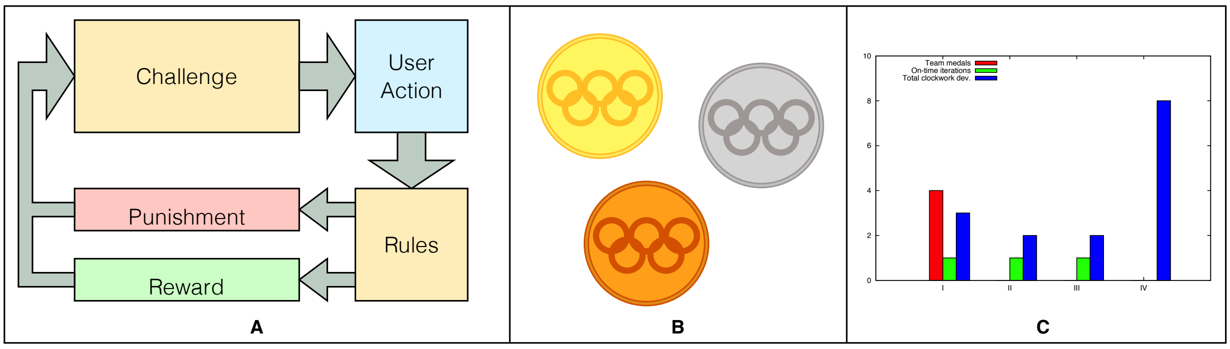

Figure 1: *Game design elements applied to software engineering processes and results: a) game mechanics explained in a challenge-punishment-reward loop; b) achievement medals awarded to software developers; c) comparison of team achievements earned by the projects in our case study.*

Abstract

Software development is a challenging, but seldom amusing activity. At the same time, gamification, a recent trend that brings game mechanics to websites and interactive media, together with many past works that propose the use of serious games to teach software engineering in a fun way, show evidence that this type of real-world activity can also incorporate game design elements. In this work, we propose a novel approach: incorporating game mechanics directly into a software development process, effectively turning it into a live game. We show interesting results from a case study with a production team of a software house, and firmly believe it is important that the game academic community spreads this type of knowledge to influence other research areas.

Keywords:: Serious Games, Software Engineering, Game Design

Author's Contact:

{epassos,esteban}@ic.uff.br
danilomedeiros@infoway-pi.com.br
pasn@ufpi.edu.br

1 Introduction

Developing software is a challenging activity that is seldom regarded as fun. Just like many other types of activities, it can be organized as a set of hierarchical and partially ordered challenges that must be overcome, often requiring several different skills from developers, and lots of teamwork effort. Surprisingly, this is very similar to an abstract definition for games: activities in which a player must learn new skills, use and combine them to overcome challenges, getting rewards or punishments, depending on success or failure, respectively. Furthermore, key concepts of games such as goals, rules, challenge, and interaction are also present in several real-world activities, for example a structured software development process.

Many previous works have proposed simulations and serious games as learning or training tools for software engineering [Navarro 2006; Baker et al. 2005; Claypool and Claypool 2005; Sweedyk and Keller 2005]. These edutainment applications resemble a game for their use of virtual environments and animated characters, but many times they lack the fun factor. One reason for this is that they are not built as games from scratch, rather being governed by complicated rules that have nothing to do with time-tested game design mechanics.

At the same time, incorporating game mechanics into websites and other types of software that include human interaction, an approach known as gamification [Takahashi 2010; Corcoran 2010], has become a trend. We believe that one can generalize the gamification concept to other activities apart from software interaction, which is something that we haven't found in the literature.

In this work, we propose a novel approach which is exactly the opposite of a serious game. Instead of creating a virtual simulation of a real-world human activity to help people learn, we propose the inclusion of game design concepts into the software engineering process itself. We show that software development is closely related to a game when the governing rules of both activities are concerned.

We also present results from a case study with an actual software development team, and a prototype of a tool to incorporate immediate feedback and game design features to task management in software engineering. We understand that knowledge from the game academic community can contribute to other areas of applied research, and expect this work can inspire others to try this approach with different types of activities.

1.1 Contributions of the Paper

The main contribution of the paper is to show that game design theory can be applied to other important real-world problems, such as software engineering. After a small survey about the important concepts related to this work, we present the original contributions of this work, which are summarized bellow:

- A gamification approach to software engineering: going beyond websites and interactive software tools, applying game design to software development activity;
- A mapping of game mechanics to software development concepts: modeling the iterative software development cycle as a hierarchical challenge graph;
- Results from a case study: we designed and evaluated individual and group achievements against historical data of a real-world development team from a software house;
- A proposal for a tool that incorporates game design elements,

such as immediate feedback and achievements, to task management.

1.2 Related Work

According to [Takahashi 2010], gamification is a trend nowadays. The technique can encourage people to perform chores that they ordinarily consider boring, such as completing surveys, shopping, or reading web sites. Corcoran [Corcoran 2010] considers that the educational systems of all activities will somehow be strongly influenced and conceptualized following video-game paradigms. This trend is already spawning product developments, such as the CloudCaptive framework [Captive 2011]. In this paper, we propose going beyond with the gamification concept, applying it not only to interactive media, but to real-world activities, in this particular case software engineering processes.

In parallel to the gamification approach, creating a serious game to simulate and teach a real-world activity is the most straightforward approach to include game mechanics in other contexts. Baker et. al. [Baker et al. 2003; Baker et al. 2005] designed a card game to augment a traditional software engineering course. We remark that our approach is opposite to this, since we are not trying to simulate or teach an activity, but rather incorporating game design elements and the fun factor into the real world.

Other interesting works [Navarro 2006; Claypool and Claypool 2005; Sweedyk and Keller 2005] also follow the teaching path by incorporating game design elements as subject of a software engineering syllabus. Students are challenged to design games, learning software engineering concepts with this hands-on approach, hopefully in a more fun way. Our proposal is not intended for classroom use, and we actually experimented our ideas with a production software development team.

Some recent works surveyed the potential use of games in software engineering courses [Fernandes and Werner 2009], or analyzed how the actual impact of this use is evaluated in the classroom [von Wangenheim et al. 2009].

An example of gamification in the software development context is the RedCritter Tracker[1] system. It is a software development task management tool that applies some mechanisms, like rewards and skill badges, after task accomplishments are achieved. Our proposal goes beyond the gamification of a tool, aiming to transforming the actual software development process into a game.

To summarize this section, we reinforce that previous work either propose the creation of simulations/serious games, or the gamification of other kinds of media, which are already interactive anyway. Compared to these, our work consists of the opposite of the first, and a novel approach for the second, applying a similar concept to a real-world activity instead of an interactive software.

2 Software Engineering Primer

The main goal of Software Engineering is to produce software products with quality, respecting time and budget constraints. The software development activity usually follows a Software Process, that defines a set of tasks to be executed, indicating what must be done, when, how, by whom, what must be used as input and produced as result. All of these tasks must be planned considering the available resources [Humphrey 1995].

The development is usually performed through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during development of earlier parts or versions of the system. This kind of development is based on the iterative and incremental life cycle model. At each iteration, design modifications can be made and new functional capabilities are added.

Each iteration results in an increment, which is a release of the system that contains added or improved functionality compared with the previous release. A software release is the distribution, whether public or private, of an initial or new and upgraded version of a computer software product during one or more iterations [Jacobson et al. 1999].

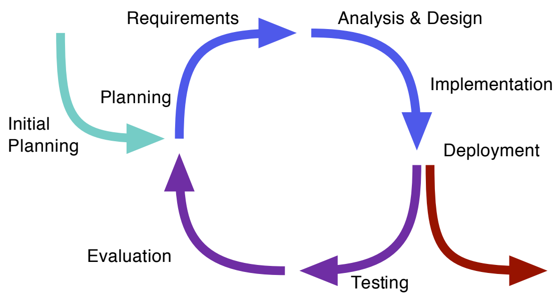

Figure 2: *Iterative development.*

There are several disciplines that can be used in an iteration, since it is possible to elicit requirements, refactor the architecture, manage the quality, and validate the software behavior by using automated tests. This is shown in Figure 2. In summary, the software development can be thought as a cycle, similar to PDCA (plan, do, check, act), where many of the tasks are repetitive.

Developers normally do not consider software development as a funny activity, since they do not have chance to choose a project to work: they have to develop what is relevant for the organization, and current software processes tend not to include leisure disciplines.

3 Game Design Concepts

The ultimate goal while transforming an activity into a game is to make it challenging and fun at the same time. Well designed game mechanics are the atoms that differentiate a fun game from other leisure activities such as reading. There are many definitions for game mechanics, but we'll use the one proposed by game designer Daniel Cook [Cook 2006], which is in turn based on Raph Koster's Theory of Fun [Koster and Wright 2004], for the scope of this work:

”Game mechanics are rule based systems / simulations that facilitate and encourage a user to explore and learn the properties of their possibility space through the use of feedback mechanisms.”

Many real-world activities, when governed by gameplay rules or mechanics, may become fun. Kicking a ball while running in a field may become boring quite fast, but when there are rules, adversaries and rewards, this activity is rather enjoyable and we call it soccer. When designing gameplay rules, one has to consider that the challenges must be well balanced, since the brain stops enjoying challenges that are either too difficult or too easy to surpass.

In this context, another element of game design that we wanted to bring to software engineering is the challenge-punishment-reward loop, which is closely related to the aforementioned definition of game mechanics. Games are constantly challenging the user to pass obstacles (or solve puzzles), which he tries to do by using known/learned skills, which in turn models his possibility space. When he doesn't succeed in the challenge, he faces the punishment defined by the game rules, while when he does succeed, a reward (either direct or indirect) is earned. In both cases, however, is mandatory that the system gives the player immediate feedback of the outcome, which increases the chances that the user fixes the learned rules. Figure 3 illustrates how these elements are put together.

3.1 An Example Game Framework

Fundamentally, games are based on the fact that humans are hardwired to like challenges, and chemically rewarded when they use skills (or learn new ones) to surpass these challenges. The following

[1] http://www.redcrittertracker.com

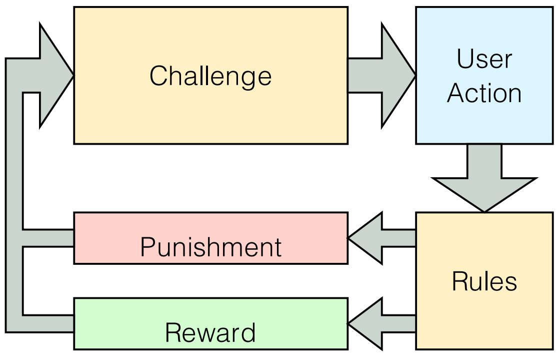

Figure 3: *Elements of game design arranged in the challenge-punishment-reward loop*

assumptions about a game design skill-set are desirable (adapted from [Cook 2008]):

- Decomposable - to make the process of learning and using a skill fun, complex skills must be decomposable into less complex ones;
- Chain-able - this decomposition process should create challenges that progressively rely on more and more complex skills, forming an increasingly difficult learning curve;
- Combinable - furthermore, challenges should depend on the combination of mastered and new skills, avoiding the creation of a loose set of isolated challenges.

Given these properties, a structure for organizing a set of challenges into a game design can be a hierarchical directed graph, where nodes represent challenges, and edges indicate the precedence relation. Each node/challenge can be either immediate, or decomposed into a more detailed graph. For instance, in the highest layer, nodes represent the set of levels on a game, while in the second-highest layer nodes represent quests/missions in a level, and in the lowest one nodes are immediate challenges that the player must surpass using his current set of skills, or by combining them with a new one. Figure 4 illustrates this hierarchical structure.

Many times in games, a challenge-graph does not need to use any edge, since the set of sub-challenges can be accomplished in any order. However, most of the times, a hybrid approach is used, with both connected and disconnected sub-graphs composing a higher level challenge. We define challenge succeed as a binary function that outputs 1 (true) for challenge complete, and 0 (false) challenge incomplete.

For immediate challenges, this function takes as input the game state corresponding to that particular game setting, and its implementation is specific to each game. For decomposable challenges, the result is given by either a logical or mathematical expression on the recursive set of values of the succeed function for all component sub-challenges. Eq. 1 and Eq. 2 formalize the domain for these two functions respectively. S denotes the space of game state variables, whereas F is the value of the succeed function for each component sub-challenges, and n represents the set of sub-challenges. For both equations, a result of 1 means the challenge as successfully passed, whereas 0 means the opposite.

$$f : S \to (1,0) \tag{1}$$

$$f : F^n \to (1,0) \tag{2}$$

Game rules are the elements that implement these functions in any game, and are independent of the type of referee used to compute them: human referees, game masters, interactive software or the players themselves.

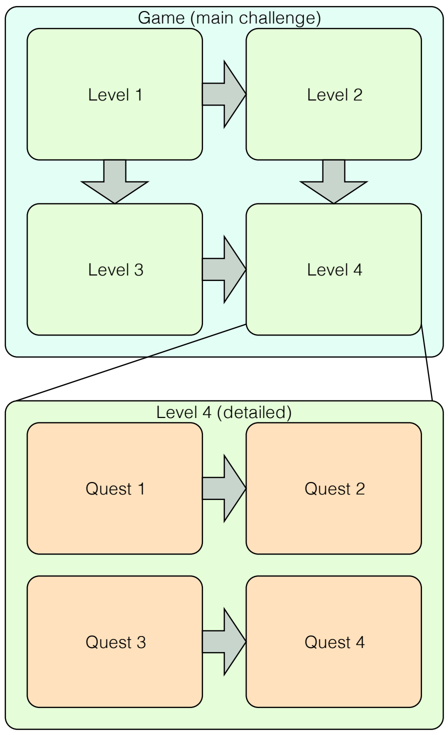

Figure 4: *Hierarchical challenge graph*

3.2 Achievements

Achievements are a concept that is orthogonal to the challenge graphs, and works as a secondary scoring mechanism that measures the use of skills to solve the same type of challenge (normally a low level one). Often, achievements are computed using two approaches: repetition or rate.

Repetition achievements are defined as the number of times the player uses a certain skill to solve the same type of challenge such as killing a certain type of enemy. Eq 3 shows a formula for this type of achievement, where n denotes the total number of times challenge c was tried, and $f(c)$ is the value for its succeed function. Rate achievements are constrained to a scope, either the container challenge or a defined time slice, and normally measure the rate between succeeded against total attempts for a particular challenge. Eq 4 defines a formula for computing rate achievements.

$$rep = \sum_{i=1}^{n} f(c) \tag{3}$$

$$rate = \frac{\sum_{i=1}^{n} f(c)}{n} \tag{4}$$

Repetition achievements are often awarded based on an approximate logarithmic scale, rendering each subsequent level increasingly more difficult to earn. For rate achievements, there are multiple levels as well, normally following a linear scale of thresholds, often rewarding the player with a medal: either a bronze, silver or gold, depending on the threshold reached. They are also cumulative, meaning that the player can collect more medals for subsequent scopes.

3.3 Immediate Feedback

A key aspect of (specially electronic) games is the use of immediate feedback to keep the player aware of his progress (or failures) through the challenges. It is desirable that this feedback is given

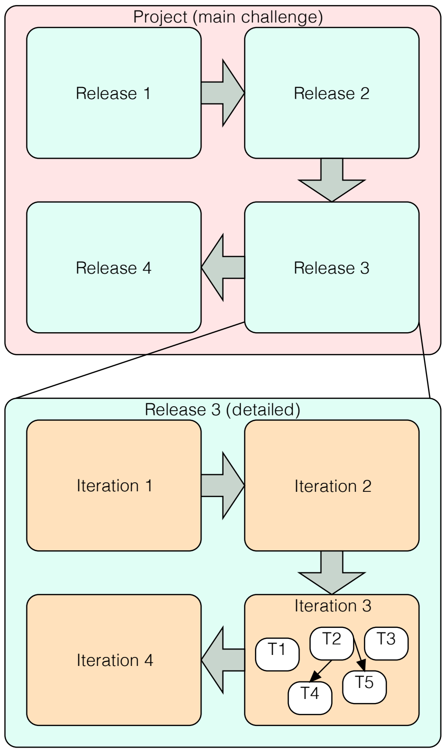

Figure 5: *Mapping of software engineering build blocks into a hierarchical challenge graph*

in real-time, increasing the feeling of immersion and also the perception that the player is the sole responsible for the outcome. Ideally, any activity that is suitable to using game design ideas should consider incorporating immediate feedback, as it is mandatory to maintain the sense of urge.

4 Mapping Game Mechanics to Software Development

When trying to design game-like mechanics/rules to real-world human activities, the first concept one needs to map is the challenge-graph. The fundamental goals of the activity will be adapted as challenges to compose the nodes in the graph. It is important to remember that these challenges/goals must be decomposable, combinable and chained to create a compelling game-like activity.

In software engineering, the main goal is to create production-grade software, normally an iterative process that is structured in a project composed of several subsequent releases, which are in turn made of a series of iterations. Mapping these concepts into a challenge-graph is straightforward, as illustrated in Figure 5.

The leaf-nodes in this graph are the immediate tasks appointed to developers, who must use their skills in systems analysis, design, programming and testing to succeed. The evaluation of the completion for these tasks is normally made by a project-manager, and often logs for this are kept stored in task-management database software. The evaluation of all higher-level challenges (iteration, release and project), however, is well defined and may be formalized by a single completion function, described in Eq. 5, where C is the challenge to be evaluated, and c are its component sub-challenges;

$$f(C) = \begin{cases} 1, & \text{if } g(c) = 1,\ \forall c \in C \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

This means that complex challenges in software development are only considered complete when all sub-challenges are also evaluated as complete. Sometimes, these sub-challenges have a precedence ordering, such as dependent programming tasks or the iterations sequence of a release, or can be completed in any order, such as independent tasks inside an iteration, as illustrated by *Tasks 1* and *3* of *Iteration 3* in Figure 5.

4.1 Metrics to Achievements

Besides the straightforward mapping of a software development project hierarchy into a challenge-graph, many companies also define and use a fairly large set of numeric metrics to either measure or even reward the best developers and teams. These metrics are commonly based on the developers performance in task planning and execution, programming, testing or other activities. Examples of software engineering numeric metrics commonly available in task-management and source code analysis software:

- Number of tasks completed;
- Average time to complete a task (estimated and actual);
- Duration of each iteration (planned and actual);
- Test code coverage;
- Code complexity;

Each of these metrics have specific meanings in the software engineering context, but simply measuring and exposing them is not compelling because they lack reference goals and sense of competition. In order to convert metrics into desirable challenge, we propose the design of solo and collective achievements. Solo achievements can be based in any individual metric, even if it includes tasks from more than one project. Collective achievements are bounded to projects, and ideally should be based in team-grade metrics.

Any of these metrics can be converted in achievements, either repetition or rate based. However, it is important to carefully design and balance the levels (for repetition achievements) and thresholds (rate ones), in order to make them interesting. The first level and lowest grade medal (bronze) must be very easy to achieve, while the others must be rendered increasingly challenging. In Section 5 we show a case study of both solo and collective achievements.

4.2 Burnout

Software development relies on many disciplines, each one having its own learning curve. When one is learning how to program, every new homework, such as ordering an array, is challenging, and this skill-learning phase may be amusing by its own. However, differently than in a game, an experienced developer already has the set of skills needed to complete the assigned tasks. Many times, this fact may render the underlying challenges rather boring. In games, when this occurs, we refer to it as *burnout*, meaning that the skill is mastered to the point that it becomes annoying to use it.

In order to avoid this burnout effect, game designers normally rely on storytelling, setting and other distractions to keep the player immersed and interested in the game. We propose to map the software development process as a real-life RPG, where the skills and challenges are real, but the red herrings are still present as popular concepts such as:

- Experience points - the completion of tasks/challenges should give experience points to the developers who solve them;
- Character attributes - since the skills are real, and the tasks given to a developer will normally follow his skill-set, character attributes must evolve according to the type of tasks the developer completes (attribute levels will be inferred, instead of chosen);
- Classes - similar to character attributes, classes must be automatically inferred, based on the overall attribute balance of a developer.

One may argue that in RPGs the player normally have the choice of how to evolve his characters, something that is basically inferred in our proposal. However, this choice still exists, it just happens in the real-world when the team plans the type of tasks each developer will do, and many times this is actually his own choice. In Section 6, we propose the integration of a complete set of these RPG features into existing task-management software.

5 Case Study

We evaluated the applicability of our approach with a case study involving a real-world software development team. Company Infoway [2] is a medium-sized software house with 28 developers, which has been successfully using agile processes for 10 years, and keeping a fine-grained data base for all project-related tasks. Their task-tracking system includes information from planning and execution phases, also recording estimated and actual time spent to accomplish tasks, grouped by iterations and project. We used their extensive log of source-control transactions to augment the experiment as well.

The case study consisted of the design and evaluation of 4 game achievements based on measured metrics: 2 achievements for individual developers (one based on repetition and one based on rate); and 2 team-oriented ones, also repetition and rate-based respectively. For this case study, we used historical data to compute the aforementioned achievements and award a selection of 4 teams, comprising a total of 13 developers, during a time frame of 4 months, which encompasses 7 iterations from these 4 projects. Firstly, we'll describe the achievements designed for the case study, and then present and analyze the results. Later on, in section 6, we discuss the integration of these achievements and other game design elements into a task management tool and the use of real-time feedback.

5.1 Achievements

We tried to design achievements that cover different aspects of task execution, such as amount of work done and rewarding developers and teams who keep up-to-date with deadlines. We also included an achievement to reward excellence in automated testing. This section describes first the individual achievements, and then the team-based ones.

5.1.1 Individual Achievements

Clockwork developer: its a rate achievement that accounts for the number of tasks a developer finished inside the planed time frame, compared to his total number of tasks, during an iteration. For this achievement, three thresholds/medals were defined:

- 🥉 Bronze : 50%
- 🥈 Silver : 75%
- 🥇 Gold : 100%

Marathonist: its a repetition achievement that accounts for the total number of tasks a developer successfully completed during his time inside the company. For this achievement, three levels were defined:

- level 1 : 5 tasks
- level 2 : 50 tasks
- level 3 : 500 tasks

5.1.2 Team Achievements

Tester team: its a rate achievement that awards the team based on test code coverage. The team earns medals for the percentage of the project's code that is covered by automated tests at the end of each iteration, according to the following thresholds:

- 🥉 Bronze : 50%
- 🥈 Silver : 75%
- 🥇 Gold : 100%

[2]http://www.infoway-pi.com.br

Clockwork team: its a repetition achievement that accounts for the number of iterations a team finished inside the planed time frame. For this achievement, again three levels were defined:

- level 1: 1 iteration
- level 2: 3 iterations
- level 3: 10 iterations

5.2 Results and Analysis

The result of the clockwork developer achievement award, as showed in Table 1, shows the number of medals earned by each developer grouped by project. Summing the seven iterations considered for each project, developers could have achieved up to 7 gold medals, however, no developer had more than two medals in total. Despite the low number of medals won, only three developers did not earn any medal, and all developers from Project IV earned two medals.

Clockwork Developer		
Project	**Developer**	**Medals**
Project I	Developer A	🥇
Project I	Developer B	-
Project I	Developer C	🥇🥉
Project II	Developer D	🥇
Project II	Developer E	-
Project II	Developer F	🥇
Project III	Developer G	🥉
Project III	Developer H	🥉
Project III	Developer I	-
Project IV	Developer J	🥇🥉
Project IV	Developer K	🥇🥉
Project IV	Developer L	🥈🥉
Project IV	Developer M	🥇🥉

Table 1: *Individual results for the clockwork developer achievement, grouped by project: total medals earned by each developer during the iterations considered in the case study.*

For the marathonist achievement, no developer failed to achieve at least the first level, as shown in Table 2. Unfortunately, only one of them (Developer M from Project IV), achieved the second level. However, the developers, A, H and I, are very close to reach the 50 tasks mark and earn the next level. The progress of tasks completeness of developers from project IV is showed in the Figure 6. As we see, the *level 2* of Developer M was earned only at the sixth iteration.

Marathonist			
Project	**Developer**	**Level**	**Tasks Completed**
Project I	Developer A	Level 1	44
Project I	Developer B	Level 1	29
Project I	Developer C	Level 1	28
Project II	Developer D	Level 1	21
Project II	Developer E	Level 1	25
Project II	Developer F	Level 1	13
Project III	Developer G	Level 1	37
Project III	Developer H	Level 1	43
Project III	Developer I	Level 1	43
Project IV	Developer J	Level 1	25
Project IV	Developer K	Level 1	20
Project IV	Developer L	Level 1	13
Project IV	Developer M	Level 2	53

Table 2: *Individual results for the marathonist achievement, grouped by project: level achieved for each developer, followed by the total number of tasks completed during the time-frame considered for the case study.*

As shown in Table 3, only Project IV failed to reach *level 1* at the Clockwork Team achievement, meaning its team never finished an iteration in time. Projects II and III reached the first level by having one iteration finished in time, and only Project I is close to reaching *level 2*. As for the Tester Team achievement, Project I earned 4 medals, being 2 silver and 2 bronze, based on its test code coverage metric. No one other team earned any medal during the time-frame considered, as shown in Table 4.

Clockwork Team	
Project	**Level**
Project I	Level 1 (2 Iterations)
Project II	Level 1 (1 Iteration)
Project III	Level 1 (1 Iteration)
Project IV	-

Table 3: *Team results for the clockwork team achievement: levels achieved for each project and total number of iterations finished in time.*

Tester Team	
Project	**Medals**
Project I	⚪⚪🟠🟠
Project II	-
Project III	-
Project IV	-

Table 4: *Team results for the Tester Team achievement: medals achieved for each project team during the iterations considered in the case study.*

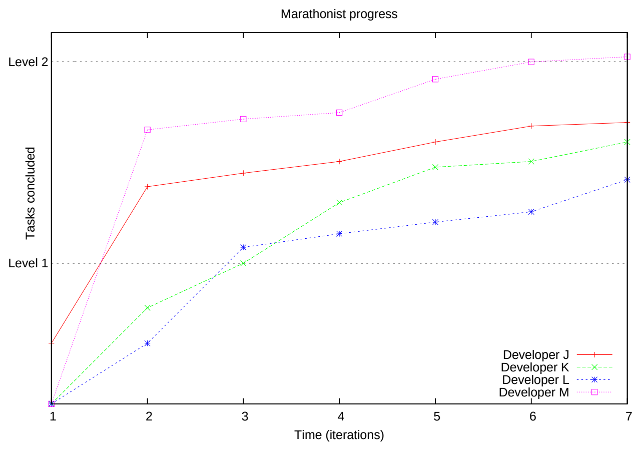

Figure 6: *Marathonist achievement progress chart for developers of Project I, showing different rhythms, probably due to disparate levels of experience and other factors.*

Figure 7 shows a comparison of projects, considering the team achievements and also the total number of individual medals earned by all developers of the team for the Clockwork Developer achievement. As one can see, Project I seems to be more consistent then the others, while Project IV hasn't earned a single team achievement, but its developers shane at the Clockwork Developer medals.

Interestingly, due to this apparent inconsistency, the company decided to do further investigations to understand the reasons behind this behavior. This investigation lead to the conclusion that the use of achievements not only may help engage developers into doing their work, but can also help monitor and control the development process as a whole. Given the results obtained, all team managers and supervisors that we talked to became interested in further developments of this work.

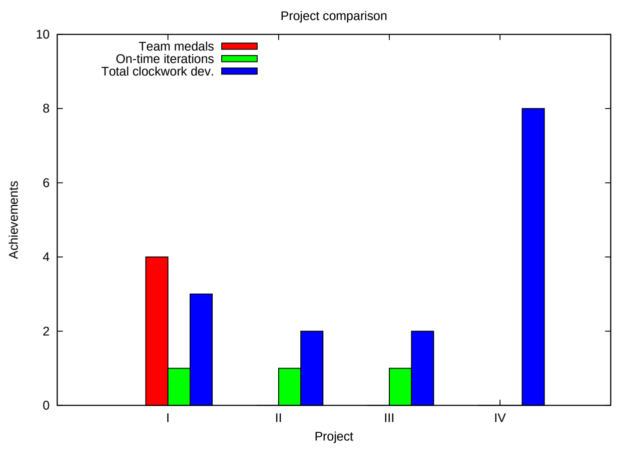

Figure 7: *Comparison of projects using both collective and total individual achievements for the team.*

6 DevRPG: Integration with a Task-management Tool

Having mapped game design concepts to software engineering and evaluated achievements in a case study, we are convinced that our approach can also be useful in everyday software development processes. For this to happen, we propose a set of RPG-like features to be integrated into a task-management tool. The standard requirements for this tool are fairly basic, which makes our approach applicable to many existing software, as long as there is source code available.

The goal is to include the game design features as an add-on, implementing as many feedback mechanisms as possible, making the developers always aware of these mechanics. Firstly, we will list the standard features required from a task-management software to make it eligible to incorporate the DevRPG elements. In Subsection 6.2, we suggest how the RPG concepts and features may be implemented and added to such software.

6.1 Standard Task-management Features Required

In this section we present only requirements related to a task management tool commonly used in software houses. The main goal of this tool is normally to record the time spent in the several software development tasks, in order to keep a historical database. Besides tracking progress and reporting productivity, the goal is also to facilitate the estimation of the resources needed by subsequent projects and iterations, based on the resources that have already needed in past projects.

Requirements are presented as a simple list of loose expressions, illustrating the concepts required, followed by an explanation that details the requirement and its use.

- **Project, release, iteration, and task containers**. A software house must keep in touch all the information about the projects in execution. A project has several iterations, each one composed of several tasks. At the end of one or more iterations, it is possible to deliver a release to the client. All these relationships among the project, release, iteration and tasks must be recorded, since the achievements proposed in this work require the analysis of them in their own scope and branches to give immediate feedback.
- **Fine-grained logs for tasks (developer, time, requirement, discipline)**. In a software development project, a task represents the registry of a work performed by one developer in order to reach a specific goal. A task is always related to some software engineering discipline, e.g. requirements, design, testing, project management or quality assurance. In order to allow the analysis of the achievements proposed here, it is fundamental to record the developer assigned to each task, the

time spent for this execution, associated requirements, and the nature of the task (discipline). Notice that logging in to record work-time is considered a very boring action, so it is important to create mechanisms to simplify this task. It must be easy to start, stop, pause and finish a task. Additionally, it is also important to create functions to help the developers remember doing this actions;

- **Source-code metrics**. The main artifact of a software project is the software source code. Due to this, it is fundamental to register metrics related to source code evolution at the version control level. There are several measures that can mine the source code and report summaries such as amount of lines, test coverage, complexity, package tangle, duplications, comments. Normally, the goal of a task management tool does not include directly gathering these software metrics, but this information is required to introduce game mechanics in a software project. Thus, a task-management tool must track this information, directly or indirectly, by using other tools. Nowadays, there are free development tools that can do this work. An useful example is Sonar[3], which is an open platform to manage code quality. Several achievements can be generated using the code quality attributes generated by such tool.

6.2 DevRPG Proposed Features

DevRPG is a proposal to incorporate RPG-like mechanics to the everyday software development activity. The goal is to make the developer more aware of his *programming character*, creating emotional bonds with this virtual persona, and thus better engaging in his daily duties. Differently than a proper game, this character reflects the player very own skills and actions, whose consequences are real. In this section we list the desired features for DevRPG.

- **Character Attributes Engine**. In DevRPG, the skills used to solve challenges are the software engineering disciplines, and since each task is related to one of these, the goal is to have the system compute the amount of experience the developer gains after finishing his assigned task. Ideally, the character attributes should be directly named after the disciplines used in the software development process, plus a master one, to control the character leveling. It is also interesting to have the software allow for the team managers to give different experience points to tasks. Another important feature of this attribute engine is to implement a decay algorithm to reduce the amount of experience points earned when the task is not completed in time.

- **Class Engine**. Given that attributes are not chosen, but rather inferred by the attribute engine, the **character class** must also be automatically inferred based on the most used disciplines for each developer. For instance, a developer who have many test tasks assigned to will end up escalating the levels of a **master-tester** class. Similar to repetition achievements, each class must have a threshold scale of minimum experience point to reach different levels.

- **Achievements Engine**. In our case study, we mined the achievements directly onto the task database. For the DevRPG tool, this feature must be automated, in order to give immediate feedback whenever an achievement is earned. However, achievements may be based in different aspects/data from the task-management tool, which asks for a flexible way to define and model the formulas to compute them. Ideally, the tool should use a production rule engine to implement this.

- **Immediate Feedback**. Developers update the state of their tasks all the time. Sometimes, an update generate a status change in the related task, or even project. An example of this is the conclusion of a task, which can finish an iteration, a release, or even the project, since all of these concepts are linked together. Even when the status change is only for the task, it is possible that this change renders a new achievement or makes the character leveling up, based on the experience points earned. The system should give immediate feedback, both visual and audible, of this changes in the character attributes, level and achievements. Thus, every action executed must invoke the attribute, class and achievement engines in order to update the character status.

- **Character Profile Screen**. The aforementioned character data must be available to the developer in a character profile screen, with historical information in graphical form. This character profile must be kept in the database, since achievements earned will linger even in case the developer change projects. This avoids the continuos execution of a snoop function, which would need to analyze too much data from all the project.

This is a simple list of desired features for an augmented task-management tool that includes the game mechanics we devised for an RPG-like software engineering process. Some of these features are straightforward to implement, while some may need further design an the use of state of the art frameworks such as a production rule engine. We are implementing these DevPRG features for the in-house task-management tool at company XPTO, and plan to run a long term study on the impact its use will have in their software development process.

7 Conclusion

In this paper, we fundamentally showed that game mechanics can be applied to any activity that includes challenges and the use of skills to succeed in them. We proposed this approach to turn software development processes into a game. For this, we presented a mapping for the challenge-graph from games to software engineering concepts, and suggested a method to incorporate achievements.

We also presented a case study in which we designed a set of achievements that were evaluated against historical data from a real-world, medium-sized software development team. The results from this experiments showed that achievements can be an interesting metric to measure performance in this context, also helping to stimulate competition among developers. The feedback obtained from the team was encouraging, as everybody was impressed with the results and enthusiastic about the idea of incorporate even more mechanics into their activity.

The inclusion of such ideas into a task-management tool is the next natural step, so we already proposed how this should be done for an example RPG-like game. We expect to have it implemented soon, and run a long term experiment with even more development teams. The main goal of the tool is to give developers real-time feedback of the game elements and general progress, helping to engage them in the underlying challenge, also introducing bits of competition among different teams.

Compared to previous research, we position our work as similar to the concept of gamification, but instead of going the straightforward path by incorporating game mechanics into other typer of interactive media (websites, for example), we showed that this concept can be incorporated into real-world activities. There are also many proposals of simulations and serious games to help teach/train software engineering, but we wanted to go the opposite direction, making an already trained team get more engaged into software development because of the use of game mechanics.

This work was also a mind-opener for both the researchers, and the team that took part in the experiment. Many of us were skeptical at the beginning, but now are convinced that game design, a centuries-old craft, can be effectively applied to other activities. We remark that the goal of the games academic community should also be to spread knowledge to other research areas, spawning novel applications such as this.

Acknowledgements

The authors would like to acknowledge the Infoway company by the important contributions to the research.

[3]http://www.sonarsource.org

References

BAKER, A., NAVARRO, E. O., AND HOEK, A. V. D. 2003. Problems and programmers: An educational software engineering card game. In *In ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, 614–619.

BAKER, A., NAVARRO, E. O., AND VAN DER HOEK, A. 2005. An experimental card game for teaching software engineering processes. *Journal of Systems and Software 75*, 1-2, 3 – 16. Software Engineering Education and Training.

CAPTIVE, C., 2011. Cloud captive: gamification made easy. `http://www.cloudcaptive.com/`, July.

CLAYPOOL, K., AND CLAYPOOL, M. 2005. Teaching software engineering through game design. *SIGCSE Bull. 37* (June), 123–127.

COOK, D., 2006. What are game mechanics? `http://www.lostgarden.com/2006/10/what-are-game-mechanics.html`, October.

COOK, D., 2008. What activities can be turned into games? `http://www.lostgarden.com/2008/06/what-actitivies-that-can-be-turned-into.html`, June.

CORCORAN, E., 2010. Gaming education. `http://radar.oreilly.com/2010/10/gaming-education.html`, October.

FERNANDES, L., AND WERNER, C. M. L. 2009. Sobre o uso de jogos digitais para o ensino de engenharia de software. In *In FEES '09: Proceedings of the Software Engineering Education Forum*, XXIII SBES, vol. 1, SBC: Brazilian Computing Society, 17–24.

HUMPHREY, W. 1995. *A Discipline for Software Engineering*. Addison-Wesley.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. 1999. *The Unified Software Development Process*. Addison-Wesley.

KOSTER, R., AND WRIGHT, W. 2004. *A Theory of Fun for Game Design*. Paraglyph Press.

NAVARRO, E. 2006. *SimSE: a software engineering simulation environment for software process education*. PhD thesis, California State University at Long Beach, Long Beach, CA, USA. AAI3243955.

SWEEDYK, E., AND KELLER, R. M. 2005. Fun and games: a new software engineering course. *SIGCSE Bull. 37* (June), 138–142.

TAKAHASHI, D., 2010. Gamification gets its own conference. `http://venturebeat.com/2010/09/30/gamification-gets-its-own-conference/`, September.

VON WANGENHEIM, C. G., KOCHANSKI, D., AND SAVI, R. 2009. Revisão sistemática sobre avaliação de jogos voltados para aprendizagem de engenharia de software no brasil. In *In FEES '09: Proceedings of the Software Engineering Education Forum*, XXIII SBES, vol. 1, SBC: Brazilian Computing Society, 1–8.