# – Implementation

## Introduction

In this work, the provenance in games framework is instantiated in the SDM game (KOHWALTER; CLUA; MURTA, 2011) as a proof of concept. The SDM game focuses on introducing Software Engineering concepts and skills to undergraduate students. The new and improved version of SDM presented in this paper includes provenance gathering and analysis, allowing students to visualize their actions and identify steps that lead to successful or unsuccessful outcomes. While the main application of provenance in this paper is over a serious game, we believe that the concepts discussed in this paper are applicable to other kinds of games and useful to support advanced analysis, such as gameplay balancing, events and behaviors data mining, and even storytelling enhancements.

The proposed framework was instantiated in a Software Engineering educational game named *Software Development Manager* (SDM) (KOHWALTER; CLUA; MURTA, 2011). The goal of SDM is to allow undergraduate students to understand the existing cause-effect relationships in software development. As so, the adoption of provenance becomes an important instrument to better support knowledge acquisition, allowing the possibility of tracking mistakes made during a game session.

## SDM

In SDM, which was developed using the game engine Unity3D (HIGGINS, 2010), the player has a team of employees that are used to develop software according to contracts made with customers. The gameplay and game mechanics are modeled presenting possibilities to the player to decide strategies for development and define the roles for each staff member. As in any contract, the software has requirements that must be followed during development. From a gameplay point of view, these requirements help to balance the mechanics and rules. When the software is completed and delivered to the customer, there is a quality assessment of the software and a project completion payment accordingly to the product quality.

Since SDM focuses in people management, the main elements of the game are the employees, which represent the player’s labor force. Employees can perform different roles (analyst, architect, manager, marketing, programmer, and tester.), which uses the employee’s attributes to calculate his performance depending on the role performed. Another element present in the game is specialization, used to define the employee working competence. With the specialization system, it is possible for employees to undergo training to learn new sets of skills. Also the concepts of working hours, morale, and stamina are used to modify the employee’s productivity. Figure 1 shows a simplified version of SDM’s class diagram focusing on the employee, showing his human attributes, types of specializations, the possibility of training to acquire specializations, and that the employee is affect by other employees that belong to the staff team. It also illustrates the project, its characteristics and requirement.

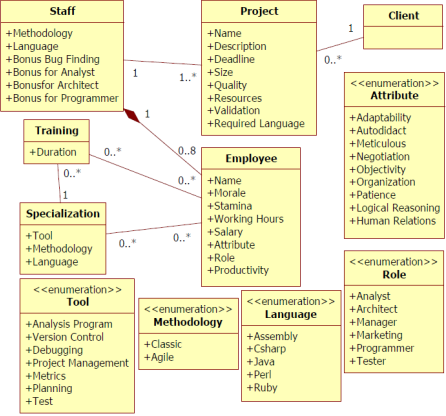


Figure 1: SDM simplified class diagram

### Provenance Gathering

The information structure used in SDM is similar to the one explained at chapter 4. Each project contains a list of the employees involved in its development. In turn, each employee has a list of his actions executed throughout the development. If any action had an external influence during its execution, then that action will also have a pointer to the action that influenced it. Throughout the game, when actions are executed or new employees are hired, the information of the event is collected and stored for later usage. Actions will go to their respective lists while new employees are added to the project list, creating their own list of actions. Each day passed in the game also records the state of the software development.

Since the information collected is used for the generation of the provenance graph, it contents are mapped to the three possible types of provenance vertex (*activities*, *agents*, and *entities*). This mapping is done according to the data model explained at chapter 4. In the following paragraphs is described the information details extracted from the game and their respective roles in the provenance graph.

Each action executed during the game is represented by an *activity* vertex. The information collected during its execution includes: who executed it, which task and role the employee was occupying, and the current morale and stamina status of the employee that executed the action. The worked hours in the day the action was generated and credits spent to execute the action are also stored. Lastly, the progress made in his task, if any. Besides those, if the action had any external influences, used or altered an artifact (prototypes or test cases), it is stored a link to the action or artifact that affected its execution.

The employees that participated in the development of the software are mapped to *agent* vertex in the provenance graph. The information collected includes the employee’s name, his current staff grade (junior, mid-level, or senior), his level, attributes, and specializations. Lastly, the *entity* vertex in the provenance graph represents one of the three possible artifacts in SDM: Prototypes produced by architects and consumed by analysts; Test Cases produced by analysts, architects, and programmers and consumed by testers; and Project, which represents the instances of the software development progress recorded each day.

The daily project information collected includes the day of its instance, the project’s deadline, how much coding was produced and the code overall quality. It also stores the clients requirements identified and modeled by analysts, how many credits the player had by the end of that day, and the state of each type of bugs in the software. For prototypes and test cases only the day they were created and their names are stored, since actions will contain the information of when they were used. Figure 2 illustrates the information collected in SDM and shown in Proof Viewer according to the vertex’s type.

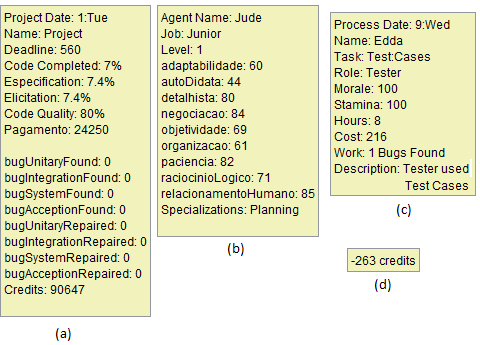


Figure 2: Information data in the provenance graph.

The *entity* vertex representation for the project’s data (a), the employee’s *agent* vertex data (b), and the action’s *activity* vertex data (c).

At the end of a gaming session, the collected data during the game session, also known as *game flow log*, is exported to an external visualization and analysis tool, the *Proof Viewer*, for provenance analysis. *Proof Viewer* in turn analyzes the data and generates the corresponding provenance graph that represents the game session. The next section describes a game scenario in SDM that is used as an example for describing the *Proof Viewer*.

## Guiding Example

Use experiment 02 as an example.

## Proof Viewer

The *game flow log* is used by *Proof Viewer* to generate a provenance graph corresponding to the game session. In order to do this, *Proof Viewer* analyses the information and interpret it to generate the vertices and edges of the graph. First, the data is processed, classifying the information to their corresponding vertex types (*activity*, *entity*, or *agent*), and then generates the edges that link each vertex.

To simplify this procedure, the information extracted from the game is arranged in pairs, where the pair represents two vertices followed by the edge that links them. As such, in a pre-processing step, *Proof Viewer* generates the vertices and edge every time it processes a pair of vertices. Each time *Proof Viewer* process a vertex, it searches the database to check if the vertex was already processed. If the vertex was previously processed, then *Proof Viewer* will use the processed vertex instead of creating a new one. Otherwise it will create the vertex. This avoids duplicates, since a single vertex can appear multiple times in the file due to the nature of how the file is structured: a vertex, another vertex, and the edge that links them.

After processing both vertices, *Proof Viewer* creates the edge and stores it in a list of edges that are later on used to generate the graph. In *Proof Viewer*, an edge is contains pointers to the vertex source, the vertex target, and the edge’s information (value and type). The source and target are the vertices previously processed from the pair. This is done until the entire *game flow log* was processed and all edges were placed in a list of edges that is used to generate the graph. All the information from the *game flow log* is processed in this stage, even if they don’t initially appear in the graph.

As mentioned earlier, vertices can belong to three types: *activities*, *entities*, and *agents*. When generating the file that contains the information extracted from the game, an additional tag is added to distinguish the vertex type. This distinction is used by *Proof Viewer* when generating the vertex in order to generate the correct vertex type. Note however that the input format can be customized, as long as it generates a list of edges, where each element in the list has the vertex source, the vertex target, and the edge.

To generate the graph, *Proof Viewer* uses the generated list of edges, creating each edge in the graph, and consequentially, the vertices from the edges. It is done this way because *Proof Viewer* was based on the JUNG framework (JOSHUA O’MADADHAIN; DANYEL FISHER; TOM NELSON, 2010), where an edge is created by the method *addEdge(source, target, edge)*, which adds the edge in the graph from *source* to *target* with the information *edge*. If *source* and/or *target* are not in the graph, then JUNG automatically generates the vertex. This avoids the need of creating each vertex before creating the edge in the graph, while at the same time checking for duplicates.

FIGURE illustrates the process of reading the *game flow log* and creating the graph. Firstly, *Proof Viewer* reads the *game flow log* and gets the first pair of vertices and the edge information, if any. Then it classifies the vertices according to their types and generates them. The next stage it generates the edge, with the source and target being the vertices previously generated and the information contained that came along with the pair. Then it adds the generated edge to the list and checks if there are any more vertices pairs to be processed. Repeat until the entire *game flow log* is processed. After creating all edges and storing them in the list, it proceeds to the next stage, which is the graph generation. For each edge in the list, create the edge in the graph. After all edges, and consequentially vertices, were created in the graph, *Proof Viewer* draws the graph.

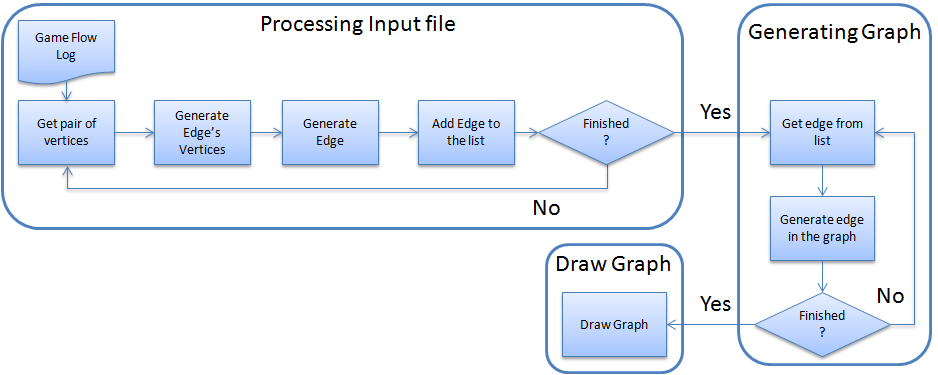


Figure 3: *Proof Viewer* processing the *game flow log* and generating the graph.

### Representations

How the data is represented in Proof Viewer (vertices and edges details). Also, how collapsing works.

The provenance graph representation from the guiding example is shown by FIGURE.

### Filters

How the filters are used in Proof Viewer.

## Final Considerations

This paper introduces new perspectives on software engineering learning, leveraging the current state of the art, based on game, to a level where the game provenance can produce and consolidate knowledge. This knowledge can help on (1) confirming the hypotheses formulated by students, (2) supporting tutors for a better guidance, (3) motivating group dynamics around some case studies, and (4) extracting behavior patterns from individual sessions or groups of sessions.

The provenance visualization can occur both on-the-fly or in post-mortem sessions. It allows the discovery of issues that contributed to specific game flows and results achieved throughout the gaming session. This analysis can be used on games to improve understanding of the game flow and identifying actions that influenced the outcome, aiding the player to understand why they happened the way they did. It can also be used to analyze a game story development, how it was generated, and which events affected it.

Currently, we do not make inferences to the user, but let the user decide what he wants to infer. Studies in this area can be made in order to identify information that can be omitted from the user without affecting the overall analysis. Another interesting research is to automatically identify patterns in the game flow. Lastly, we are working on different graph visualization layouts and running experimental studies on the usage of provenance in educational games to evaluate the aspects of learnability.

# References

HIGGINS, T. *Unity - 3D Game Engine*. Disponível em: <http://unity3d.com/>. Acesso em: 5 maio 2011.

JOSHUA O’MADADHAIN; DANYEL FISHER; TOM NELSON. *JUNG: Java Universal Network/Graph Framework*. sourceforge: Open-source, 2010. Disponível em: <http://jung.sourceforge.net/>.

KOHWALTER, Troy; CLUA, Esteban; MURTA, Leonardo. SDM – An Educational Game for Software Engineering. In: 2011 X BRAZILIAN SYMPOSIUM ON GAMES AND DIGITAL ENTERTAINMENT (SBGAMES), nov. 2011, Salvador. *Anais*... Salvador: In: X SBGames, nov. 2011.