Tool-supported Iterative Learning of Component-based Software Architecture for Games[*]

David Llansó
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
llanso@fdi.ucm.es

Marco A. Gómez-Martín
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
marcoa@fdi.ucm.es

Pedro P. Gómez-Martín
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
pedrop@fdi.ucm.es

Pedro A. González-Calero
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
pedro@fdi.ucm.es

Magy Seif El-Nasr
College of Arts, Media and Design and Computer and Information Sciences
Northeastern University, Boston, MA 02115, United States
m.seifel-nasr@neu.edu

ABSTRACT

Students of game development at the master level usually have a hard time becoming comfortable and proficient in applying component-based software architecture design, used in most professional games, to their own projects. In this paper we describe a teaching methodology that allows students to very rapidly iterate through versions of simple games, and, with the help of *Rosette*, a tool that facilitates the identification of components, evaluate a large number of component distributions, accelerating the learning process.

1. INTRODUCTION

The master program on game development at Complutense University of Madrid in Spain, established in 2004, focuses on delivering a degree on Computer Science targeted for game programing and development. For the last 5 years, we taught the component-based software architecture [1] used in most professional games nowadays, and using such an architecture has been a requirement for our students' capstone projects. Nevertheless the learning of the component-based architecture is hard and complex for even programmers [4] and, in our experience, students at the master level usually have a hard time becoming comfortable and proficient in the application of the component-based software architecture.

[*]Spanish Ministry of Economy and Competitiveness under grant TIN2009-13692-C03-03

Component-based software architecture deconstructs object-oriented mechanisms in order to promote reusability by generating game entities through the composition of fine-grained components. Although the main ideas of this approach are usually easily grasped by a Computer Science graduate. It often takes time and several errors to developed the skills required to come up with a good component-based design in terms of reusability, extensibility and lack of functionality duplication.

For that reason, we designed a new teaching methodology that allows students to very rapidly iterate through increasingly complex versions of simple games. To test our method, we allowed them to iterate over two simple games, one developed in Unity and the other in C++. This process allows them to try and evaluate a large number of component distributions, thus accelerating the learning process. This approach is made possible thanks to *Rosette*, an authoring tool we have developed to facilitate the collaboration between programmers and game designers by maintaining an ontological view of the entities in a game and connecting it to aggregations of components in the source code [2, 3].

2. THE METHODOLOGY

As teachers in a master degree of videogames in the Complutense University of Madrid, we face some challenges while teaching component-based architectures [1] in the last few years. Thus, we adopted a method that introduces *Rosette* and uses it as a pedagogical tool to allow students to assimilate component-based design and methodology.

We use it to put into practice a methodology where students implement a 3D game using a component-based architecture in a guided way. After some basic explanation about the theoretical aspects of the architecture, they were told what components they should implement for developing the small test-bed game, and why. We provide them with sample components, code sketches and many placeholders where students write specific code for implementing the game.

Although students seemed to assimilate the inner workings of CBA, when they were confronted with their master project where they must implement their own videogame nearly from scratch, they suffered a lot of difficulties trying to distribute the functionality among components. The first prototypes used to have both very big components with low cohesion or very specific non configurable components. None of them are reusable, so each new feature, kind of enemy or change in the game design supposed a new component, usually very similar to an existing one, causing duplicity of code (and bugs).

Our conclusion was that, although CBA fundamentals are easy to understand, practically identifying and designing a good component distribution is not so simple. After all, CBA breaks in some way the object oriented programming paradigm, so students must change their minds when designing components [4].

In order to overcome the previous difficulties, we have changed our teaching methodology. We confront students with two small videogames where they must not just fill in the gaps, but completely design the GO layer, identifying entities and distributing their functionality into components.

To emulate a complete videogame development, where game design is a moving target, students are provided with a partial definition of each game, that is enriched in a progressive way after some iterations. Our intention is to force the students to reevaluate their previous components distributions, trying to reuse as much components as possible or deciding when they are forced to refactor. In this way, we mimic the challenges students will afford during their projects, but in a more controlled environment.

Instead of leaving students completely on their own when developing their game, they will use *Rosette*, a game authoring tool developed by us that facilitates the game design process and the implementation of these game designs. Each iteration consists of the next stages:

- Students are provided with a *running example* of the game they must implement at the end of the current iteration. We could have used a prototypical game design document explaining the requested functionality. However, those documents require some time to be analyzed, and are prone to misinterpretations. A running game accelerates the comprehension phase and avoids problems. With the executable, students receive also all the graphical resources that they need to implement their own clones of the game.

- Once the requirements are assimilated, students must focus in distributing the functionality among components and defining the GO of the game. For this task the students use *Rosette* which helps them to create a collection of components in the first iteration or to update this collection in subsequent iterations. In this stage the students do not have to worry about implementation details, but rather focus on creating a better semantic distribution.

- Finally, students use *Rosette* to (re)create the code sketches, and implement the new functionalities they added in the previous stage. As *Rosette* is an iterative development tool, they can generate the code fragments at any moment without losing the changes and additions done in the source code files of previous iterations.

We have defined the process as three stages but, during the same iteration, students could move from the semantic distribution designed in *Rosette* to the implementation stage more than once in the case that they did not correctly specify the domain model and component decomposition in the first try.

As teachers, our aim with those practice sessions is teaching how to distribute components, but the motivation of the students is to create videogames. In that sense, *Rosette* allows us to achieve the two goals because it greatly speeds up the development of the iterations and eases the CBA refactorization. This means that students can very quickly test different component distributions without being worried about the subsequent changes in the source code, because the tedious work of rearranging it is facilitated by *Rosette*. We will show later, the use of *Rosette* and how it give us this possibility that has become a great mechanism for learning (and teaching) CBA and definitely .

We have put into practice the ideas presented above in two separate experiences with the same group of people. All of them were students of our post-graduate masters in game development at the Complutense University of Madrid, and most of them have completed a BS in computer science before enrolling in the masters program. Of the 19 students signed up for the study, only a few of them have prior experience developing games and none having ever created a component distribution.

Before the study, we asked them to create a small game using components. Specifically, in the first weeks of our studies, we introduced the game development process using Unity, a component-based game development platform. Instructors guided students to build, step by step, some small games. After that, they attended theoretical lectures about game architecture in general, the old inheritance based implementations and component based architectures.

Our experiment took place in this context, and was split into two different experiences. The first one consists on using Unity itself to build a small game. Unity has a built-in library with common components, so students had a collection of components to use and extend. *Rosette*, which has an integrated semantic representation of all those components, is able to generate Unity compatible C# code. Therefore, students could use it as an external tool to model the game ontology and components before implementing the details in Unity.

After that, we switched to a game development from scratch, where they had to code the entire game using C++.

3. EVALUATION

Before teaching these classes, our students knew the theory of the videogame component-based architecture and they have worked with Unity creating some prototypes in a guided way. However, when they started working on the first videogame they had serious difficulties dividing the features among components. This led us to believe that the previous theoretical and practical classes were not enough. In this section, we will show learning results once students used *Rosette* in the games mentioned in the previous section.

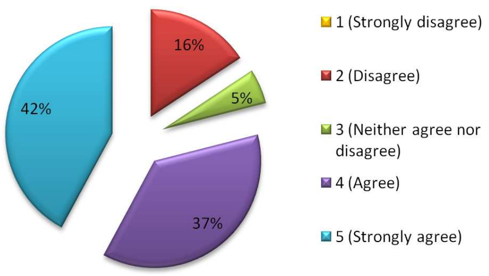

Figure 1: Statement: *Rosette* and the methodology, to teach (or learn) what a component-based architecture is, is very useful

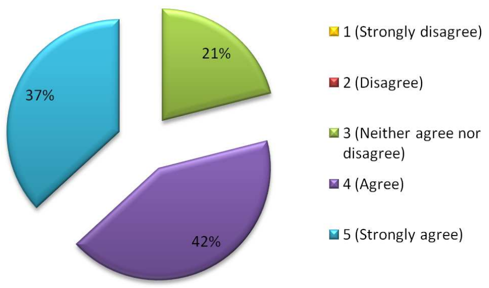

Figure 2: Statement: The iterative code generation of *Rosette* is very useful when the domain is modified because it preserves all the code implementation

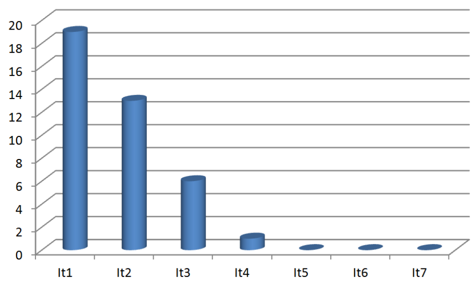

Figure 3: Number of students that finished the different iterations of the first game

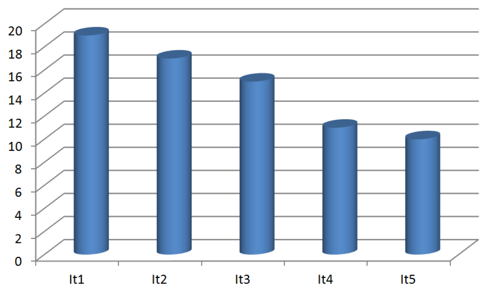

Figure 4: Number of students that finished the different iterations of the second game

First of all, we must confess that one of the things that positively affected the teaching was the high motivation of the students. The motivation was high due to the fact that students were invested in developing their own games, as opposed to previous years were they had to develop a game based on other people's work based on a given assignment. We really think that this is one of the key aspects and if we had made students work only in a semantic way, without implementing the game, the result would have not been the same.

To evaluate our teaching approach, we wanted to measure the progression of the students in dividing responsibilities among components. To do this we adopted several methods:

1. We gave students a questionnaire at the end of the classes to gauge the usefulness of the tool as they perceive it.

2. We also collected data from each iteration, including code, the semantic domain modelled in *Rosette*, traces from *Rosette*, etc.

The questionnaire was composed of several quantitative and qualitative questions (in a likert scale). The Figure 1 shows that almost the 80% of the students consider that the methodology of the classes was useful, whilst only the 16% of them considered that working in a high level point of view (using *Rosette*) does not provide anything to them and preferred to directly work with Unity scripts or C++ (they told us so in the qualitative question). Figure 2 reflect that the majority of them (again the 80%) found it useful to have the possibility to modify or refactorize the component distribution from a semantic point of view without paying a price in re-coding the functionality previously programmed that is moved to a new component. In this case, no student considered this feature as something undesirable.

On the other hand, we have then used the collected data to observe the results of our methodology focusing in different aspects. Our premise is that if the component distribution created by students along the iterations are good enough, they have properly learnt how the component-based architecture works. A form to measure the quality of the created components is seeing their reusability and flexibility so, if the components created in one iteration have been reused to implement new entities in subsequent iterations we can infer that they are good components. In the same way, if there are code replicated in different components we will assume that the distribution is not good enough due to there is a good candidate to be a new component. We can also understand that if the mistakes in early iterations are solved in later iterations, then the student has identified the mistakes and has learnt about them.

The first part of the study using Unity was hard for the students, nobody was able to finish all the iterations of the first game (Figure 3). In fact, a lot of them did not pass the second one. 31% of the students only finished the first iteration whilst 42% only reached the second one, 21% the third and only 5% of them finished the fourth iteration. Even worse is the fact that the component distributions were not very good in general and they dedicated a lot of time to design the components. The little good news was that some of them improved their distributions in the middle of an iteration or when a new feature was added (but barely the 30%). The most difficult thing for them was to find similarities between abilities of different entities to reuse components. Almost every student created several components that includes very similar code and functionality.

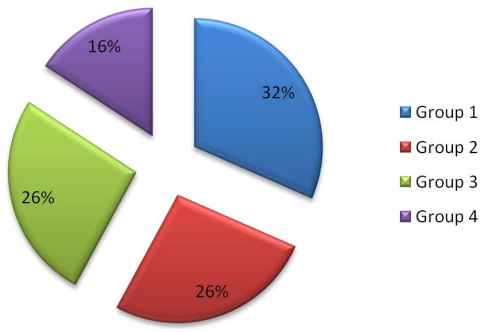

Figure 5: Groups of students with different progression and quality of their component distributions

However, when students started with the second game it seemed that they had learnt more than we initially thought. Figure 4 shows that there were many who finished all the game iterations (more than the 50%) and the component distribution in this second game, although not perfect, they were substantially better. Students spent less time in *Rosette* distributing the components and more time dealing with C++ and the game architecture.

Besides the previous data, we have analyzed the data collected from the iterations in order to infer the quality of the component distributions the students did. To this end, we have analyzed in each iteration the semantic domain, the source code and the traces of *Rosette*. Having a glance at the domain of an iteration, we can suppose if the GO and component distributions seem to be good or not because we can detect functionality that is duplicated from a semantic point of view (i.e. two components have the same attributes and messages to do the same functionality in different GO). However, although the view of *Rosette* is enough to detect bad distributions, it is not to assure that we have a good component collection. Consequently, we have also to inspect the source code of the components looking for duplicated pieces of code that reveal possible candidates to be new components. Finally, in the traces of *Rosette* we can see distributions in the middle of an iteration and we can see if the student refactorized its domain.

According to our data analysis, we distributed the students in four groups (Figure 5) depending on their progressions in the development and the quality of the component distributions they did in the iterations. The students of each group are characterized themselves by:

1. They were able to identify from the beginning the reusable components the game design should used. These students had no mistakes splitting the functionality in components and, over the iterations, they just needed to make very little improvements, adding parameters to adjust the component behaviour.

2. They did not create a perfect component distribution in its first attempt but as the iterations advanced, they refactor them, finishing with a component distribution as good as the one reached by the first group.

3. They were not able to clearly classify all the features of the game; they identified some good components that were later reused in subsequent iterations, but they also missed important ones such as the movement component that includes the code to set the position of the entity according to its speed (a component reusable in player, enemy ship and asteroid).

4. The distribution of these small group of students did not pass the minimum threshold to be considered a good distribution because of the use of big non-reusable components or because of not having understood the component based architecture at all.

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented the tool and the method we used for teaching component-based architectures (CBA) for game development in a Computer Science post-graduate master. Students are usually fluent with object-oriented design, using hierarchies, but they need to adapt their habits to the new architecture philosophy.

Our method consists of giving students a couple of small games that they must implement on their own, instead of in a step-by-step way. Although the proposed games are simple enough, it is unrealistic to expect students will be able to experiment with different component distributions and, at the same time, to write all the code in a short time.

To solve this problem, we used *Rosette*, a graphical tool that let the user define the semantic game domain, and provides iterative consistence checking and code generation. It boosts the game development cycle, something very useful for the first stages of CBA learning where component refactor is quite common. Students are also motivated during the practice sessions, because they have the opportunity not just to learn CBA, but also build and play their evolving game.

Although students do not, obviously, become CBA experts in just 12 or 15 hours, results show that they get the fundamentals of CBA and internalize their mechanisms in a better way than previous students (who did not employ *Rosette* and this methodology). Additionally, *Rosette* has shown itself as a useful tool not just for teaching but for game development in general; some of the students are still using it for their capstone projects, even when CBA introduction lectures ended more than a month ago.

5. REFERENCES

[1] M. Chady. Theory and practice of game object component architecture. In *Game Developers Conference*, 2009.

[2] D. Llansó, M. A. Gómez-Martín, P. P. Gómez-Martín, and P. A. González-Calero. Explicit domain modelling in video games. In *International Conference on the Foundations of Digital Games (FDG)*, Bordeaux, France, June 2011. ACM.

[3] D. Llansó, P. P. Gómez-Martín, M. A. Gómez-Martín, and P. A. González-Calero. Iterative software design of computer games through FCA. In *Procs. of the 8th International Conference on Concept Lattices and their Applications (CLA)*, Nancy, France, October 2011. INRIA Nancy.

[4] M. West. Evolve your hiearchy. *Game Developer*, 13(3):51–54, Mar. 2006.