Empirical Evaluation of the Automatic Generation of a Component-Based Software Architecture for Games[*]

David Llansó, Pedro Pablo Gómez-Martín,
Marco Antonio Gómez-Martín and Pedro A. González-Calero
Dep. Ingeniería del Software e Inteligencia Artificial Universidad Complutense de Madrid, Spain
{llanso,pedrop,marcoa,pedro}@fdi.ucm.es

Abstract

Component-based architecture is the software design of choice for most modern games and game engines, allowing for extensible and reusable software that easily adapts to changes in requirements. Nevertheless, designing a component-based architecture is not an easy achievement and some techniques have been previously proposed for the automatic generation of a component distribution to implement a given declarative model of the entities in the game. The goal of the work presented here is to empirically compare the quality of the component distributions designed by programmers, with those others automatically obtained with such techniques.

Introduction

The state-of-the-art in software engineering for games recommends the use of a component-based software architecture for managing the entities in a game (West 2006). Instead of having each entity implemented in a particular class which defines its exact behaviour, now entities are just components containers where every functionality, skill or ability that the entity has is implemented by a component. A component-based architecture facilitates the definition of new types of entities as collections of components that provide basic pieces of functionality, creating flexible software that can be adapted to changes in game design. However, such flexibility comes with a price, both in terms of software understanding and error checking: a game where entity types are just run-time concepts is harder to understand than one with an explicit hierarchy of entity types; and error checking that, in a more traditional inheritance-based architecture, would come from type safety at compile time is now lost.

In order to alleviate these problems we have previously proposed an extension to the component-based architecture with a declarative domain model, including a description of the entities, its attributes and components, along with the messages they exchange (Llansó et al. 2011a). Such extension promotes a new methodology for game design based on maintaining an ontological view of the entities in a game, and connecting it to aggregations of components in the source code. Allowing this explicit model of the semantic domain of the game, which is hidden in the source code in a pure component-based architecture, we bring into light information that facilitates the communication between programmers and designers, and helps in the task of adding and reusing functionality. Furthermore, we have developed *Rosette*, an authoring tool that supports the construction of such a model and includes a number of modules that leverage on this ontological representation to infer and provide additional feedback and suggestions that improve the software quality and make the game more flexible and robust.

In this paper we present the evaluation of a particular feature in *Rosette* which can automatically suggest a component distribution to implement a given declarative model of the entities in the game. Given a model of the entities, described in terms of their attributes and messages, the user may choose to manually identify a number of components to distribute such data and functionality or ask *Rosette* to do the work. The details of this automatic process have been described elsewhere (Llansó et al. 2011b), but in essence, an inductive technique, Formal Concept Analysis (FCA), is applied to obtain optimal groups of attributes and messages.

The goal of the work presented here is to empirically compare the quality of the component distributions designed by programmers using *Rosette* with those others automatically obtained by *Rosette* itself. Understanding game development as an iterative process, we measure quality of a component distribution in terms of coupling, cohesion, code duplication and modifiability and extensibility, as it evolves across several iterations of game development. The rest of the paper runs as follows. Next Section describes the main features and mechanisms in *Rosette*. The Experiment section describes the design of the experiment whilst the Evaluation section presents its results. Last Section draws some conclusions and presents lines of future work.

Rosette

Rosette is our game authoring tool and provides:

- *Easy authoring of the game domain*: Intuitive representation of the game domain with mechanisms that avoid duplication promoting flexible and reusable components.
- *Semantic knowledge stored in a formal domain model*: the game domain is transparently stored in a knowledge-

[*]Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03)

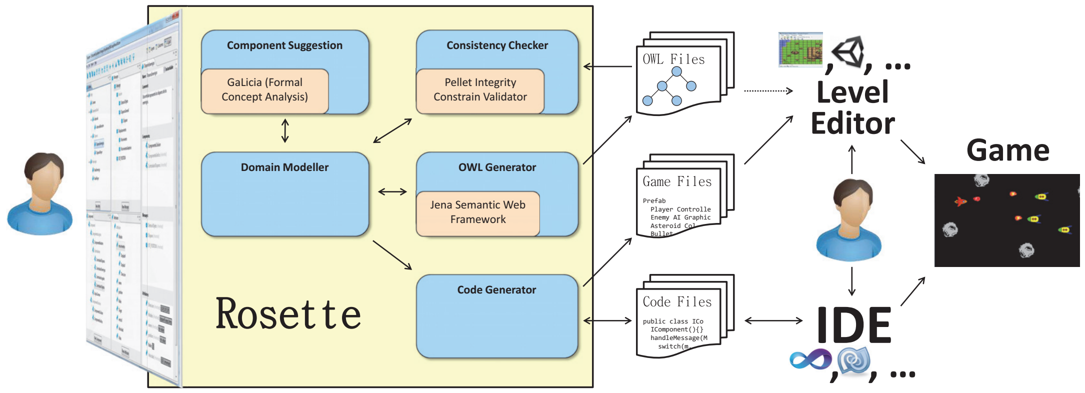

Figure 1: General view of the *Rosette* architecture

rich representation using OWL[1]. Therefore, *Rosette* is a working example of using ontologies for software engineering (Seedorf, Informatik, and Mannheim 2007) and introduces an Ontology Driven Architecture methodology (Tetlow et al. 2006) in the game development cycle.

- *Automatic component extraction*: using Formal Concept Analysis (FCA) (Ganter and Wille 1997), *Rosette* suggests the best component distribution for the entity hierarchy specified by programmers (Llansó et al. 2011c; 2011b) and allows manually fine-tune that components.
- *Consistency checking of the game domain*: the semantic knowledge allows validation and automated consistency checking (Zhu and Jin 2005). *Rosette* guarantees the coherence of the changes (Llansó et al. 2011a).
- *Code sketches generation*: as in Model Driven Architectures (MDA), *Rosette* boosts the game development creating big code fragments of the components. *Rosette* is able to generate code for the Unity engine[2] and for plain C++.
- *Agile methodologies*: videogame specification will surely change in the long run so *agile methodologies*, such as *scrum* (Schwaber and Beedle 2001) or *extreme programming* (Beck 1999; Beck and Andres 2004), take importance in videogame developments. *Rosette* greatly speeds up the iterative development and also eventual refactorizations in the component distribution. When the domain description is modified in *Rosette*, it automatically fires refactorization tasks in the code previously generated.

Figure 1 shows a general view of the *Rosette* architecture and how it is used in the development process. The user graphically defines the entities of the game domain, with state (attributes) and functionality (messages), along with their relations, specifying a model that *Rosette* internally represents as an OWL ontology. This knowledge-rich representation is then used to check inconsistencies and to provide feedback to the user through the *Consistency Checker* module. Using *Rosette* the user may also manually specify the component distribution or use the *Component Suggestion* module for getting a first version of that distribution that can afterwards be fine-tuned. Finally, the *Code Generator* module is used to generate code sketches of those components and other game files. The code sketches will be filled by the programmer outside of *Rosette* whilst the game files are used in the level editor to simplify the level creation.

As *Rosette* supports iterative development, this process can be repeated several times during the development of a game. This means that each time the user modifies the game domain, the *Code Generator* module re-creates the game assets taking care of preserving the game levels and the hand-written source code added in the assets of the previous iterations. *Rosette* maintains that code even when complex refactorizations are made.

In this paper, we present the evaluation of the automatic generation of component distributions through Formal Concept Analysis (FCA). FCA is a mathematical method for deriving a concept hierarchy from a collection of objects and their properties. Each concept in the hierarchy represents a group of objects that shares a set of properties. Since a component is a set of properties (messages and attributes) that is part of several objects (entities), *Rosette* uses this concept hierarchy to automatically identify a component distribution that will fit the requirements of the provided entity ontology (Llansó et al. 2011c; 2011b).

The candidate distribution is shown to the user through the visual interface, where he has the opportunity to modify it by merging or splitting components, adding or removing extra functionality, changing names, etc. Once all the desired changes have been done, *Rosette* generates a big amount of scaffolding code for the game entity manager. It also stores all the changes applied to the proposed component distribution using the lattices (Birkhoff 1973) created with FCA and represented with OWL in our domain. This information will be used in subsequent iterations to automatically replay all the changes into future proposed component distributions.

Experiment

The goal of the experiment that we carry out with *Rosette* was to answer the question "Does our component suggestion technique help to create better component distributions that improve the quality of the software?".

[1]http://www.w3.org/TR/owl-features/
[2]http://unity3d.com/unity/

The experiment simulates the initial stages of a game development. Its design reflects the volatile nature of the development process: over five different stages, the subjects of the experiment are faced to the development of a small game where the complexity of its entities increases. If they fail to design a good component system in the first iterations, they will be forced to rewrite a great amount of code in the refactorization process, or they will end up with a messy set of components. This scenario allows us to measure whether the component suggestion does or does not aid in the process.

The experiment consists in the development of a side scrolling shoot'em-up 2D arcade where the player controls a spacecraft, shooting enemies while avoiding collision of asteroids. However, as previously told, instead of having the entire design from the very beginning, the game specification evolves over five iterations. The description and their expected result in terms of components follow:

1. The first version of the game had to show a spaceship flying from left to right with the camera moving at constant speed in the same direction. Some static asteroids also appear. A reasonable component design includes (1) a graphics component for drawing entities, (2) a controller component that captures movement events and moves the player accordingly, (3) a camera component that updates the camera in the graphics engine and (4) a movement component that moves the owner entity at constant speed and that both player and camera use.
2. Next iteration consists in the creation of a random movement in the asteroids. That implies the development of a new component that, in some sense, plays the role of the simple artificial intelligence of asteroids that moves them randomly throughout the space.
3. Third iteration adds enemies in the form of other spaceships and the ability of both player and enemies to shoot bullets. This forces to (1) the creation of new kinds of entities for enemies and bullets, (2) developing a component that manages the spawn of the bullets, which is used in both player and enemy entities, (3) reusing the simple AI component for the spaceships movement, (4) reusing the component that moves the owner entity at constant speed for the bullets and (5) modifying AI component to emit the shoot messages when needed.
4. The fourth iteration asks for minimal collision detection in order for spaceships to react when bullets or asteroids collide against them. The expected solution is the addition of a physic/collision component and a life component that respond to the collision events.
5. In the last iteration an item is added to the scenario. The player is not able to shoot bullets until he picks this item. The development of this last gameplay mechanic is easy if the previously built components were well-thought. In particular, no other components are required, but only some modification to the existing ones.

For the experiment we gathered a group of 18 postgraduate students from our Master on Game Development at Complutense University of Madrid . All of them had taken master classes about game architecture, the pros and cons of the use of inheritance when developing the hierarchy of entities and component based architecture concepts.

The game had to be created in C++ and they had available a game skeleton that incorporates a minimal application framework that manages the main loop of the game, and a graphic server that encapsulates the inner workings of the Ogre render API[3]. It also had a small logic layer that manages component-based entities. The logic layer was able to load the description of the entities and the level map from external files. It, however, had no components, so students had the freedom to design the capabilities of every component from scratch. *Rosette* could aid in the creation of those components because it is able to generate the C++ code that is smoothly integrated within the logic layer provided.

In order to evaluate the component suggestion technique of *Rosette*, we separated the students in two groups where every student developed the game individually. The *control group* used a restricted version of *Rosette* where the *Component Suggestion* module was not available and the *experimental group* was able to use *Rosette* unabridged.

Evaluation

As discussed earlier, the goal of this study is to analyze the impact of our component suggestion technique and whether it helps to create better component distributions. In order to compare the distributions of the control and experimental group we need an unbiased way of measuring their quality.

A *good component distribution* is characterized by the reusability of their components, its flexibility against changes, and the maintainability and extensibility of the resulting software. Reusability and flexibility increase when the *cohesion* of each component is maximized and, at the same time, the coupling between components (and with other classes) is minimized. Code duplication is detrimental to maintainability, and extensibility is inversely proportional to the number of changes needed to add a new feature.

In order to measure all these aspects in our experiment, we have used these tools:

- **Code and generated software:** we collected the *Rosette* domains and the code of each iteration. We used it for searching *cohesion* and *coupling* issues that state flexibility deficiencies, and also to look for code duplicities that damage maintainability in the students' implementations.
- **Traces:** we also collected traces from *Rosette* that reflects the use of the tool during the different iterations. We have used this information to analyze the effort needed to modify or extend the software created by the students.
- **Questionnaire:** we gave students a questionnaire at the end of the experiment to gauge the usefulness of the tool as they perceive it.

In the next subsections we will detail our findings about each one of these aspects while using *Rosette*.

Cohesion

Cohesion refers to the degree to which the elements of a module (class or component) belong together (Yourdon and

[3]http://www.ogre3d.org/

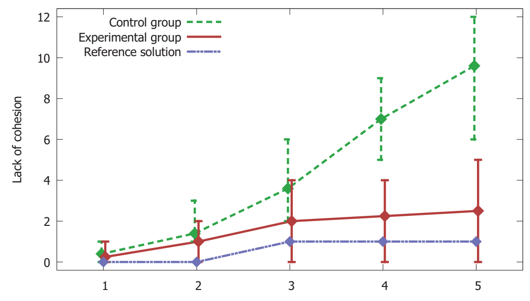

Figure 2: Lack of cohesion of the distributions per iteration

Constantine 1979). In order to evaluate the *lack* of cohesion of the software generated by the participants of the experiment we have used the *Lack of Cohesion in Methods (LCOM) measure*, described in (Hitz and Montazeri 1995; Chidamber and Kemerer 1994) as "the number of pairs of methods operating on disjoint sets of instance variables, reduced by the number of method pairs acting on at least one shared instance variable". LCOM is applied to individual classes, and is higher when they lack cohesion.

Here we have a simplified example extracted from the code of one of the participants:

```
class Controller : IComponent {
private:
   float _speed;
   Entity* _bullet;
public:
   void VerticalMove(float secs) {
      Vector2 pos = getPosition();
      pos += Vector2(_speed*secs,0);
      setPosition(pos);
   }
   void HorizontalMove(float secs) {
      Vector2 pos = getPosition();
      pos += Vector2(0,_speed*secs);
      setPosition(pos);
   }
   void Shoot() {
      EntityFactory f =
           EntityFactory::getInstance();
      Entity e = f->instanciate(bullet,
                          getPosition());
      f->deferredDeleteEntity(e, 2.0f);
   }
}
```

In this code, the `Controller` component provides two different functionalities: movement and shooting ability. This constitutes a lack of cohesion, as comes into light when LCOM is calculated: although both methods (`VerticalMove`, `HorizontalMove`) use the `_speed` attribute, the pairs (`VerticalMove`, `Shoot`) and (`HorizontalMove`, `Shoot`) do not share anything at all. Thus, LCOM has a value of 1 ($= 2 - 1$).

We have calculated the LCOM of every component of the software generated by the students in the five iterations and then we have added them up to illustrate the evolution of the cohesion during the game development. Figure 2 presents the average results of the two groups of participants where we can appreciate that in the first iterations the cohesion is quite similar between the two groups but, when the development evolves, the software generated by the students in the control group, without the help of *Rosette*, has much less cohesion. They usually have bigger components and, as their functionality is less united, they are less reusable, what means that these components are specific just for the game in current development (and even only for the current iteration). In order to test the statistical significance of the results, we calculated the P-value for the two groups in the last three iterations. The P-value is the probability, under the null hypothesis, of obtaining a result at least as extreme as the one that was actually observed (Goodman 1999). When the P-value is less than 0.05 (Stigler 2008) indicates that the result would be unlikely under the null hypothesis. The P-value for the iterations 3, 4 and 5 were 0.076675, 0.0014788 and 0.00099428 what shows that the probability that the values of the two groups belongs to the same population decreases with the game development and in the last two iterations are significantly different (P-val < 0.05). Figure 2 also shows the LCOM of the solution, described above in the Experiment Section, that was used as reference of a good distribution. The Figure reveals that the experimental group, aided by *Rosette*, ended with a distribution with similar cohesion to that in the reference one.

Coupling

Coupling is the degree to which each program module relies on other modules and is usually contrasted with cohesion. (Hitz and Montazeri 1995) define that "any evidence of a method of one object using methods or instance variables of another object constitutes coupling". They also propose a way to categorize the coupling between classes and object considering some rules; unfortunately they are not suitable for component-based architectures.

The definition given by (Lee et al. 2001) fits better: "When a message is passed between objects, the objects are said to be coupled. Classes are coupled when methods declared in one class use methods or attributes of the other classes". The paper also proposes a *coupling measure* similar to that one we have used for the cohesion. They define the *coupling value* (Pi) as the *number of methods invoked* (different to the number of *method invocations*) from one class to anotherwhere *interaction coupling* is defined as:

$$Interaction\_Coupling = \sum_{AllUseCase} Pi$$

Where each use case defines interactions between a component and the system to achieve a goal and they are extracted from the sequence diagrams.

Lee et al. also define *static coupling* which is caused by class association, composition or inheritance. They assign values to different relationships: 1 when two classes shares the same parent class, 2 when one class inherits from another, 3 when there is a directional association between two classes, 4 when the association is bidirectional and 5 when one class is an aggregation of another.

Component-based architectures add a new way of class coupling due to dynamic relationships. We consider that

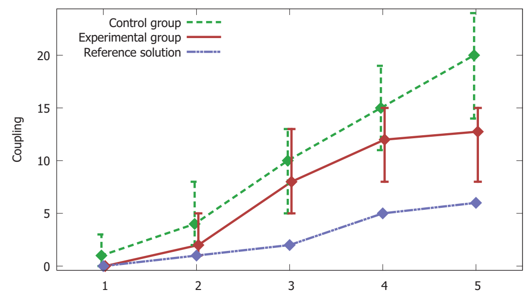

Figure 3: Coupling of the distributions per iteration

message passing between components constitutes a coupling of sibling classes, so are valued with 1. On the other hand, when a component accesses to the entity it belongs to is explicitly using the aggregation between them, so according with (Lee et al. 2001) the coupling is 5.

Once *interaction* and *static coupling* are defined, Lee et al. create the *two classes coupling* as:

$$Two\_Classes\_Coupling = Interaction\_Coupling * \\ Static\_Coupling$$

Using the next code extracted from one of the participants' solutions we will show an example of how $Two\_Classes\_Coupling$ is calculated:

```
class AI : IComponent {
public:
   void tick(unsigned int msecs) {
      float _hMove,_vMove;
      if(!_entity->getType().
            compare("MovingAsteroid")) {
         _hMove = rand() % 10 - 5;
         _vMove = rand() % 10 - 5;
      }
      else if(!_entity->getType().
                 compare("Enemy")) {
         _hMove = rand() % 20 - 10;
         _vMove = rand() % 20 - 10;
      }
      IMessage m =
           new HorizontalMove(_hMove);
      sendMessage(m);
      m = new VerticalMove(_vMove);
      sendMessage(m);
   }
}
```

$Interaction\_Coupling$ between `Entity` and `AI` classes has a value of 1, because `AI` only invokes the `getType()` method and there is just one use case that involves both classes.

On the other hand, the `AI` component accesses the entity it belongs to (inherited `_entity` attribute), using the knowledge about the aggregation between entities and components, which constitutes a static coupling of 5. These two values become a $Two\_Classes\_Coupling$ of 5 (= 1 * 5).

`AI` sends `HorizontalMove` and `VerticalMove` messages that are processed by the `Controller` component previously presented. That shows a *static coupling* between both components (`AI` and `Controller`) valued as 1. They both interact in just one use case, but they exchange those two types of messages, so they have an $Interaction\_Coupling$ of 2 that turns into the same value for $Two\_Classes\_Coupling$.

Using this procedure, we have carefully calculated for each participant the $Two\_Classes\_Coupling$ of each component with other classes, and we have added them up to obtain the *global coupling* of the component distribution. Figure 3 presents the results of the average of this global coupling in both groups during the five iterations. We can appreciate that the evolution of the coupling is similar in both groups and, from the statistical significance point of view, the iterations 3, 4 and 5 we have P-values of 0.34346, 0.12407 and 0.0015096. This shows a positive tendency but only in the last iteration the values between the two groups are significantly different. However, a closer look at the code has shown us that the reasons for this coupling are different. The coupling of the software generated by the experimental group is because the higher cohesion of their components prevents duplication in the functionality but makes the components communicate with each other. In fact coupling is usually contrasted with cohesion. For example, the `Controller` component of the previous example is used by the `AI` component but also by the player, which sends messages in the presence of keyboard events. In this way, the reusability of the `Controller` is good in this aspect although this slightly penalizes component coupling due to the message passing.

As components created by students in the control group were bigger and had more functionality (low cohesion) it would be expected that they had less coupling. However, those components, although big, were not usually self-contained. For example, some students implemented a component that mixed functionality of the `AI` and `Controller` components. This seems better from the coupling point of view but these implementations needed the *concrete entity type* (`Player`, `Enemy` or `Asteroid`) to decide how to behave, which creates unnecessary coupling.

Finally it was to be expected that our reference solution should have a bigger coupling since it was the solution with less cohesion. However, the other solutions have been very penalized by the *static coupling* because they invoke several times methods of the entity (generally to consult the entity type) and this multiplies by 5 the *interaction coupling*. So, although the number of messages exchanged by the components of our solution is bigger, the resulting coupling is smaller because it is considered less dangerous for reusability than accessing the entity type.

Duplication

Duplication of the functionality in the source code penalizes its maintainability. We have analyzed the components created by the participants in the experiment looking for duplicate code and, when a functionality (i.e. a method) was replicated in two or more places, we penalized that domain. In this sense, the duplication measure reveals the number of pieces of code that have been duplicated and could have been encapsulated and reused by different methods.

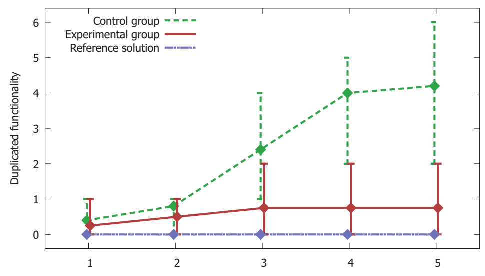

Figure 4: Duplicated functionality in the different iterations

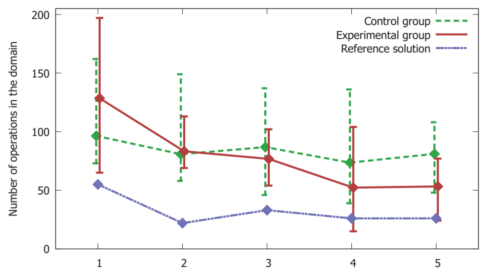

Figure 5: Number of operations done per iteration

Figure 4 presents the evolution of the duplicated functionality during the game development. It illustrates that participants, using the component suggestion of *Rosette*, maintained low code duplication during the whole development while people that did not use it duplicated more code each iteration. This figure confirms what cohesion and coupling measures revealed about the quality of the component distributions and proves that a bad distribution of the components leads to a bad reutilization of the functionality and to code duplication, which not only proves a poor reusability but also provokes a worse maintainability. The P-values in the iterations 3, 4 and 5 were 0.021433, 0.0017129 and 0.0012074 what prove that this results are not coincidence.

The big increment in the third iteration is mainly due to two reasons: (1) some participants were not aware of asteroids and enemy spaceships moving in the same way so, instead of reusing the functionality created in the second iteration (asteroids) they duplicated this code; (2) some of them also duplicate some code related with the shoot (both player and enemy) instead of parameterizing it in a new component.

During the fourth iteration duplicated code in the control group also increases significantly because many students did not group together the functionality related to collisions so they duplicated code in different components.

Modifiability and Extensibility

The last factor that should be taken into account to evaluate the quality of a component distribution is the modifiability and extensibility. A good measure of those aspects is the number of changes the component distribution suffers when a new functionality is added. We have used the traces generated by *Rosette* to collect this information. Those traces let us measure *semantic changes* in the domain, not just source code modifications that are less informative. *Rosette* tracked domain modifications such as components, attribute or messages creation, adding components to entities, attributes to components, elements removal, etc.

Paying attention to the average number of actions required to create the domains of the participants of both groups (Figure 5) we can observe that during the initial iterations, participants in the control group needed to apply less operations to the game domain to obtain the proposed functionality, but in the last iterations, the number of changes increased contrasted with the people in the experimental group.

The bigger number of operation carried out in the first iteration can be explained because creating a better component distribution needs a more complex domain model. But it is important to note that when using the component suggestion of *Rosette* (experimental group), many of the domain operations *are automatic*, so a richer domain does *not* imply more human time. Users using component suggestions got a better initial distribution that let them to require fewer and fewer changes into the domain when the game evolved. On the other hand, users that did not receive assistance created a poorer distribution that needed a nearly constant number of changes in the next iterations.

In this case, the statistical significance of the results is not relevant for the first iterations. Results for iterations 3, 4 and 5 were 0.35490, 0.16899 and 0.011173 what shows that only in the last iteration the results of the two groups were significantly different but let us think that, if the development had continued, in further iterations the experimental group would require fewer operations than the control group.

Conclusions and future work

Our intention with this experiment was proving that the component suggestion technique of *Rosette* helps to create a good distribution of the functionality in components and that the quality of the software generated is good enough in terms of flexibility, maintainability and reusability.

The data collected during the experiment reflects that, having two groups of people with the same skills in the development of videogames, those people that used the component suggestion module of *Rosette* ended with unbiased better software in every iteration. The key principle of maximizing the cohesion and minimizing the coupling is better accomplished by them and is also reinforced with the lack of code duplication. However, the main indication that this technique may improve and accelerate game development is that the number of changes the component distribution suffers when the requirements change is lower when following *Rosette*'s suggestions.

Nevertheless, the creation of an initial promising component distribution is expensive. Although the component suggestion of *Rosette* lighten this work, some students pointed out that this module is not intuitive, becoming hard to use. We could corroborate this fact because up to four people in the experimental group left the experiment, when only two of the control group abandoned. We, then, plan to study how to flatten the *Rosette* learning curve. In addition, we need to further evaluate the effectiveness of the tool and techniques with bigger games and larger user groups.

References

Beck, K., and Andres, C. 2004. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional.

Beck, K. 1999. Embracing change with extreme programming. *Computer* 32:70–77.

Birkhoff, G. 1973. *Lattice Theory, third editon*. American Math. Society Coll. Publ. 25, Providence, R.I.

Chidamber, S. R., and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20(6):476–493.

Ganter, B., and Wille, R. 1997. Formal concept analysis. *Mathematical Foundations*.

Goodman, S. N. 1999. Toward evidence-based medical statistics. 1: The p value fallacy. *Annals of Internal Medicine* 130(12):995–1004.

Hitz, M., and Montazeri, B. 1995. Measuring coupling and cohesion in object-oriented systems. In *Proc. Intl. Sym. on Applied Corporate Computing*.

Lee, J. K.; Seung, S. J.; Kim, S. D.; Hyun, W.; and Han, D. H. 2001. Component identification method with coupling and cohesion. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC '01, 79–. Washington, DC, USA: IEEE Computer Society.

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

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

Llansó, D.; Gómez-Martín, P. P.; Gómez-Martín, M. A.; and González-Calero, P. A. 2011c. Knowledge guided development of videogames. In *Papers from the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process (IDP)*. Palo Alto, California, USA: AIII Press.

Schwaber, K., and Beedle, M. 2001. *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st edition.

Seedorf, S.; Informatik, F. F.; and Mannheim, U. 2007. Applications of ontologies in software engineering. In *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006)*.

Stigler, S. 2008. Fisher and the 5 *CHANCE* 21(4):12–12.

Tetlow, P.; Pan, J.; Oberle, D.; Wallace, E.; Uschold, M.; and Kendall, E. 2006. Ontology driven architectures and potential uses of the semantic web in software engineering. W3C, Semantic Web Best Practices and Deployment Working Group, Draft.

West, M. 2006. Evolve your hierarchy. *Game Developer* 13(3):51–54.

Yourdon, E., and Constantine, L. L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1st edition.

Zhu, X., and Jin, Z. 2005. Ontology-based inconsistency management of software requirements specifications. In Vojtáŝ, P.; Bieliková, M.; Charron-Bost, B.; and Sýkora, O., eds., *SOFSEM 2005*, LNCS 3381. Springer. 340–349.