Knowledge Guided Development of Videogames

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

Abstract
Due to the changing nature of videogames, the component-based architecture is the design of choice for managing game entities instead of the traditional static class hierarchies. A component-based architecture lets programmers edit entities as collections of components, which provide the entity with new functionalities. Such architecture promotes flexibility but makes the code more difficult to understand because entities are built at runtime by linking components.

In this paper we present a semi-automatic process for moving from a class hierarchy to a component-based architecture. Through the application of Formal Concept Analysis we propose a novel technique for automatically identifying candidate distributions of responsibilities among components.

Introduction
Game development may involve many months of work by a large team of people from different disciplines, resulting in large and complex software systems. To make things worse, game design is a hard design domain (Thomas and Carroll 1978) with the very ill-defined goal of “being fun”, which causes that game requirements change throughout the development of a game as designers try new ideas. From a software engineering point of view, the module responsible for the management of the game entities should be carefully implemented since it is the most affected module by those continuous changes, and therefore it must be flexible enough to be adapted to unexpected changes in the specification.

Traditionally, the code layer responsible of the management of the game entities took the form of an inheritance hierarchy of C++ classes. These classes represent the hierarchy of entities and procedures that, in some sense, may be viewed as the actions that these entities were able to perform. In the last few years, however, the component-based software architecture is the design of choice for managing entities in modern video games (Garcés 2006; West 2006). Instead of having entities of a concrete class, which define their exact behaviour, each entity is just a component container where every functionality, skill or ability that the entity has is implemented by a component.

Such architecture promotes flexibility, reusability and extensibility but makes the code more difficult to understand, since now the behaviour of a given entity is built at run-time by linking components. However, this may be confusing for programmers due to the loss of the class hierarchy where the entity distribution is seen at a glance. At the same time, compilers will lose important datatype information, decreasing the amount of errors detected at compile time.

We have been developing Rosette (Llansó et al. 2011), a visual authoring tool that eases the design of the game domain. Rosette allows experts to define entities on a hierarchical structure, and translates it into a component-based model. This facilitates the game design work since the entity distribution can be seen at a glance. Our tool also creates a knowledge-rich representation of the game domain using OWL (Web Ontology Language)©. This fully-fledged domain model will have a positive impact during different phases of the development process, bringing the ability to use reasoning engines for several tasks that go from checking inconsistencies in domain models to creating better AIs that reason over this annotated knowledge. Finally, generating code assets from the modelled game domain, Rosette is based on a Ontology Driven Architecture methodology (Tetlow et al. 2006) that enables validation and automated consistency checking (Zhu and Jin 2005).

In this paper we concentrate on the non-trivial process of going from an entity hierarchy specified in Rosette to an the entity functionality distribution implemented in a component-based design. We apply a novel technique that identifies the best candidate component distribution between entities using Formal Concept Analysis (FCA) (Ganter and Wille 1997), a mathematical method for data analysis.

The rest of the paper runs as follows. The next section describes both the component-based architecture for videogames, and FCA. The main contribution of the paper evolves in the two following sections which explain how to use Rosette for generating a conceptual hierarchy, the process of applying FCA to the conceptual hierarchy and how to generate a candidate distribution of components. The last section presents related work and concludes the paper.

---

©Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03)
Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

http://www.w3.org/TR/owl-features/
inspired in the Java 2 game (Gómez-Martín et al. 2007), can architecture, we can see how a class hierarchy (Figure 1), where, every functionality, skill or ability that the entity has, is implemented by a component.

The straight-forward approach of organizing entities in class hierarchies is inefficient. Component-based architecture is an approach that abstracts entities into components, which are self-contained pieces of logic that can perform different tasks (Bilas 2002). Common entity examples are enemies, players’ avatars or interactive items. Not so obvious entities are pure logic elements like camera sequences, waypoint markers, or triggers that control plot points in the story. The module implementing and managing the entities in a game is at the core of the game engine and, due to its size and complexity, is responsible for a good part of the game development effort.

Combining such a software size and complexity with the moving nature of the system specification as determined by an evolving game design, we face the need for a highly flexible and extensible software architecture. The component-based architecture manages the state of the game through a set of components as shown in Figure 2. Notice how methods from classes in the hierarchy of entities now become methods in components. For example, the walk functionality moves from the Player class to the PlayerControllerComp component.

From the entity point of view, every component that belongs to it is just an IComponent. So, component methods such as walk() cannot be externally invoked. For that purpose, the component-based architecture incorporates a version of the Command design pattern, which transform method invocation into objects that are passed around (Gamma et al. 1995). These objects are usually called messages. Therefore, the walk() method is only invoked when a walk message is processed by the PlayerControllerComp component.

As entities are now just a list of components, the concrete components that constitute them are specified in an external file that is processed in execution time, making the game entities data-driven.

**Formal Concept Analysis**

Formal Concept Analysis (FCA) is a mathematical method for data analysis, knowledge representation and information management. It was proposed by Rudolf Wille (Wille 1982) and during the last decades it has been used in numerous applications in different domains, like Psychology, Medicine, Linguistics and Computer Science among others.

FCA is applied to any collection of items (or formal objects according to FCA nomenclature) described by means of the set of properties (or formal attributes) they have. When FCA is applied over a set of objects, data is structured and grouped into formal abstractions called formal concepts. These formal concepts can be seen as a set of objects that share a common set of attributes. Therefore, the result of the application of FCA over a collection of items provides an internal view of the conceptual structure and allows finding patterns, regularities and exceptions among them. Moreover, formal concepts may be sorted using the subconcept-supерconcept relationship, with the set of objects of subcon-
cepts being a subset of the ones of superconcepts, and reciprocally, the set of attributes of subconcepts being a superset of the ones of superconcepts.

The items and attributes are given to the FCA in form of a formal context. A formal context is defined as a triple \( \langle G, M, I \rangle \) where \( G \) is the set of objects, \( M \) the set of attributes, and \( I \subseteq G \times M \), is a binary (incidence) relation expressing which attributes describe each object (or which objects are described using an attribute), i.e., \((g, m) \in I\) if the object \( g \) carries the attribute \( m \), or \( m \) is a descriptor of the object \( g \). When the sets are finite, the context can be specified by means of a cross-table, where rows represent objects and columns attributes. A given cell is marked when the object of that row has the attribute of the column.

We can define a concept by enumerating the set of objects that belong to that concept (its extent) or listing the attributes shared by all those objects (its intent). Formally, we define the prime operator that when applied to a set of objects \( A \subseteq G \) returns the attributes that are common to all of them:

\[
A' = \{ m \in M \mid (\forall g \in A)(g, m) \in I \}
\]

and when applied over a set of attributes, \( B \subseteq M \), its result is the set of those having attributes:

\[
B' = \{ g \in G \mid (\forall m \in B)(g, m) \in I \}
\]

With this definition, we may now define the formal concept: a pair \( \langle A, B \rangle \) where \( A \subseteq G \) and \( B \subseteq M \), is said to be a formal concept of the context \( \langle G, M, I \rangle \) if \( A' = B \) and \( B' = A \). \( A \) and \( B \) are called the extent and the intent of the formal concept, respectively.

The set of all the formal concepts of a context \( \langle G, M, I \rangle \) is denoted by \( \beta(G, M, I) \) and it is the output of the FCA. The most important structure on \( \beta(G, M, I) \) is given by the subconcept - superconcept order relation denoted by \( \leq \) and defined as follows \( (A_1, B_1) \leq (A_2, B_2) \) where \( A_1, A_2, B_1, B_2 \) are formal concepts and \( A_1 \subseteq A_2 \) (which is equivalent to \( B_2 \subseteq B_1 \) see (Ganter and Wille 1997)). This relationship takes the form of a concept lattice. Nodes of the lattice represent formal concepts and the lines linking them symbolize their subconcept-superconcept relationship.

A concept lattice may be drawn for further analysis by humans (Figure 4). This representation uses the so called reduced intents and reduced extents. The reduced extent of a formal concept is the set of objects that belong to the extent and do not belong to any subconcept. On the other hand, the reduced intent comprises attributes of the intent that do not belong to any superconcept. In that sense to retrieve the extension of a formal concept one needs to trace all paths which lead down from the node to the bottom concept to collect the formal objects of every formal concept in the path. By contrast, tracing all concepts up to the top concept and collecting their reduced intension gives its intension.

### Methodology for Obtaining the Lattice

In order to alleviate the handicaps of the component-based architecture, we let experts define a conceptual entity hierarchy through a visual interface (Figure 3). This is similar to define a UML class diagram: entities will be placed into

<table>
<thead>
<tr>
<th>Entity</th>
<th>setPositon</th>
<th>setAnimation</th>
<th>touched</th>
<th><em>physicalclass</em></th>
<th><em>aiscript</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persona</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResourceBearer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Partial formal context of the game domain

the hierarchy and will be populated with attributes and functionalities.

Before applying FCA, the first step is to automatically extract the information of the entities from the entity hierarchy modelled by the experts and express it in terms of a formal context. This formal context \( \langle G, M, I \rangle \) is built in such a way that \( G \) contains every entity type (such as Persona) and \( M \) will have every attribute and functionality (setAnimation or setPosition). Finally, \( I \) is the binary (incidence) relation \( I \subseteq G \times M \), expressing which attributes of \( M \) describe each object of \( G \) or which objects are described using an attribute.

Recovering our running example, Table 1 shows a partial view of the formal context in a cross-table. It shows the features of entity types at a glance. For example, the Persona entity type has the setPosition and the setAnimation functionalities and the _physicalclass_ attribute among others, as also shown in Figure 1.

The application of FCA over such a formal context gives us a set of formal concepts and their relationships, \( \beta(G, M, I) \). Figure 4 shows a hierarchical structure of formal concepts ordered by the subconcept-superconcept relationship and for every formal concept it indicates its reduced intent and extent. Starting from the lattice (Figure 4) and with the goal of extrapolating the formal concepts to a programming abstraction, a naïve approach is to generate a hierarchy of classes with multiple inheritance. Unfortunately, the result is a class hierarchy that makes an extensive use of multiple inheritance, which is often considered as undesirable due to its complexity.

### From Entities to Components

Although the approach of converting formal concepts into classes has been successfully used by others (Hesse and Tilley 2005; Godin and Valtchev 2005), it is not valid in our context. This approach is based on the analysis of the extents of the formal concepts. Our approach is the opposite; instead of focusing on the objects (reduced extents of the formal concepts), we set the foundations of the technique in the attributes (reduced intents). After all, our final goal is to remove inheritance and to identify common entity features in order to create independent classes (components).

The novel technique implemented in Rosette has two phases. The first one is an automatic process that presents a candidate set of components to the expert. In the second phase we provide the expert with some mechanisms in case
he/she wants to alter somehow the generated component distribution.

**Inferring Components from the Lattice**

When automatically inferring a component-based architecture, the first approach is to create an Entity base class, with no attributes but a list of components, and propose the set of components needed for the actual game by analysing the formal lattice. However we can benefit from it also when building the generic entity. If the top concept (⊤) has a non-empty intent, their attributes and functionalities are, by definition, found in every entity type and, therefore, it makes sense to have them in the Entity base class. This means that the generic entity class may have some general attributes and may be able to carry out some functionality (_name attribute or setPosition action in our example of Figure 4).

After that we can proceed with the analysis of the formal concepts. The idea is based on the fact that when a formal concept has a non-empty reduced intent this means that the concept contributes with some attributes and/or functionality that has not appeared so far (when traversing the lattice top to bottom). The immediate result is that the reduced extent of objects differs, from the objects in the superconcepts, in those properties. So, the attributes and functionalities in the reduced intent of every of these formal concepts should be consider to build a component. At the same time, Rosette states that all the instances of the entities in the extent of the formal concept will include the new created component.

Summarizing, in Figure 4, the reduced intent of the formal concept number 1 would represent the generic Entity whilst reduced intent of the formal concepts 2, 3, 4, 5, 7, 8, 9, 11 and 12 would represent the components of the architecture. For example, when analysing the formal concept labelled 11, our technique will extract a new component containing the features turn, stopWalk and walk, and will annotate that all entities (generic instances) of the concept Player will need to include this component. The Player will also contain other components extracted from the formal concepts above it: 2, 3, 5, 7 and 9.

Before the candidate component-based system is presented to the expert, the components are automatically named. Component names are created by concatenating each attribute name of the component or, when no one is available, by concatenating all the message names that the component is able to carry out. Thus, the resultant candidate distribution of components proposed by Rosette for the Figure 1 is the one shown in Figure 5, where components have been renamed for legibility.

**Expert Tuning**

The automatic process detailed above ends up with a collection of components with autogenerated names and the Entity base class that shares some common functionality. This result is presented to developers, who may edit it using up to four different operators:

1. **Rename**: proposed components are automatically named according to their attributes. The first operator users may perform is to rename them in order to clarify its purpose.

2. **Split**: in some cases, two functionalities not related to each other may end up in the same component due to the entity type definitions (FCA will group two functionalities when both of them appear together in every entity type created in the formal hierarchy). In that case, Rosette gives developers the chance of splitting them in two different components. The expert will then decide which features remain in the original component and which ones are moved to the new one (which is manually named).
3. **Move features**: this is the opposite operator. Sometimes some features lie in different components but the expert considers that they must belong to the same component. In this context, features of one component (some elements of the reduced intent) can be transferred to a different component. In the lattice, this means that some arrows are moved from a node to another one. When this movement goes up-down (for example from node 9 to node 10), *Rosette* will detect the possible inconsistency (entities extracted from node 11 would end with missed features) and warns the user to clone the feature also in the component generated from node 11.

4. **Add features**: some times features must be copied from one component to another one when FCA detects relationships that will not be valid in the long run. In our example, the dependency between node 3 and 4 indicates that all entities with a graphic model (4, *GraphicsComp*) will have physics (3, *PhysicsComp*), something valid in the initial hierarchy but that is likely to change afterwards. With the initial distribution, all graphical entities will have an _scale attribute. The original component is renamed as a PhysicsComp, but experts could envision that this should be a native feature of the *GraphicsComp* too. This operator let them add those “missing” features in order to avoid dependencies.

Comparing the set of components automatically inferred by *Rosette* (Figure 5) with the real distribution of components in the Javy 2 game (Figure 2) we realize that they are quite similar. The example presented in the paper only contains a portion of the total number of entity types in the game but, in our experiments, with more entity types the number of real and proposed components is more alike. In the case that the expert wanted to modify the proposed component architecture, he has the means to do it although the proposed component distribution is a totally functional one.

With the purpose of demonstrating how the expert would use the previous operators to transform the proposed set of components, we apply some modifications to the automatically proposed architecture (Figure 5) that turn it into the real classes of Javy 2 (Figure 2).

First of all, the *SpeedAttComp* has the _speed attribute but no functionalities. In designing terms this is acceptable, but rarely has sense from the implementation point of view. *Speed* is used separately by *PlayerControllerComp* and *AIAndMovementComp* to adjust the movement, in our experiments, with more entity types the number of real and proposed components is more alike. In the case that the expert wanted to modify the proposed component architecture, he has the means to do it although the proposed component distribution is a totally functional one.

With the purpose of demonstrating how the expert would use the previous operators to transform the proposed set of components, we apply some modifications to the automatically proposed architecture (Figure 5) that turn it into the real classes of Javy 2 (Figure 2).

First of all, the *SpeedAttComp* has the _speed attribute but no functionalities. In designing terms this is acceptable, but rarely has sense from the implementation point of view. *Speed* is used separately by *PlayerControllerComp* and *AIAndMovementComp* to adjust the movement, so we will apply the MOVE FEATURES operator moving (and cloning) the _speed feature to both components, and removing *SpeedAttComp* completely. This operator is coherent with the lattice (Figure 4): we are moving the intent of the node labelled 9 to both subconcepts (11 and 12).

Then, the SPLIT operator is applied over the *AIAndMovementComp* component twice. In the real implementation the *AIAndMovementComp* is divided in three components whilst in the *Rosette* suggestion resides in the same component. In the first application of the SPLIT operator, the _goToEntity_ and _goToPosition_ actions are moved to a new component, which is named *GoToComp*. The second application results in the new *SteeringToComp* component with the _steeringTo_ action and the _speed attribute. The original component is renamed as *AIComp* by the RENAME operator and keeps the _aiscript attribute.

Finally, although the *Entity* class has received some generic features (from the top concept, T), they are especially important in other components. Instead of just use those features from the entity, programmers would prefer to maintain them also in those other components. For this reason, we have to apply the ADD FEATURES operator over the *GraphicsComp*, *PhysicsComp* and *SpeakerComp* components in order to add the _setPosition_ and the _setOrientation_ functionalities to them.

![Figure 4: Concept lattice](image)

![Figure 5: The candidate components proposed by Rosette](image)
When the expert feedback is ended, the entity description should be stored. Rosette stores the inferred description in the same fully-fledged OWL domain model that contains the conceptual entity hierarchy, enriching this way the semantic knowledge of the game. This kind of domain is our solution to store not only the definitions of the entities but also extra semantic knowledge about entities, components, messages (actions to carry out) and attributes. This means that, at semantic level, the domain is thoroughly described knowing which components belong to every entity type but also which messages and attributes describe every component. In typical approaches this knowledge remained hidden in the code. This knowledge is then used to check errors in the domain and in future steps of the game development (as creating AIs that reason over the domain).

Related Work and Conclusions

Regarding related work, we can mention other applications of FCA to software engineering. The work described in (Hesse and Tilley 2005) focuses on the use of FCA during the early phases of software development. They propose a method for finding or deriving class candidates from a given use case description. Also closely related is the work described in (Godin and Valtchev 2005), where they propose a general framework for applying FCA to obtain a class hierarchy in different points of the software life-cycle: design from scratch using a set of class specifications, refactoring from the observation of the actual use of the classes in applications, and hierarchy evolution by incrementally adding new classes. The main difference with the approach presented here is that they try to build a class hierarchy while we intend to distribute functionality among sibling components, which solve the problem with multiple inheritance in FCA lattices.

Javy 2, our test-bed educational game was initially developed using an entity hierarchy (a portion was shown in figure 1), and afterwards manually converted to a component-based architecture (Figure 2). When Rosette was available, we tested it using the original Javy 2 hierarchy, and the initial component distribution was quite acceptable when compared with the human-made one. We could have saved a significant amount of time if Rosette had been available at the time.

Although it is out of the focus of this paper, we are successfully working in an iterative software design of videogames using FCA. This methodology will extend the contribution of this paper by letting experts modify the entity hierarchy sometimes during the game development, where Rosette provides new component suggestion retaining every modification the expert did during the previous steps.

In the long term, our goal is to support the up-front development of games with a component-based architecture where entities are connected to a logical hierarchical view. Such goal will require to somehow make information flow not only from logical entities to components but also the other way around. The initial entity hierarchy is sure to change once some code has been filled in the generated place holders or low-level attributes have been added. Although some effort has been done in order to ensure coherence (Llansó et al. 2011), we need to improve the code generation phase to do it reversibly. Rosette needs a way to preserve hand-made code if the hierarchy is changed and classes must be regenerated. Even more, operators applied to the initial component distribution should be automatically reapplied into the lattice in a coherent way if changes in the hierarchy allow them. We could thus maintain simultaneous views of an implementation in both paradigms (hierarchical an component-based), allowing intuitive edits in whichever is the most natural representation without losing the ability to track inconsistencies created in the other view.

References


Gamma, E.; Helm, R.; Johnson, R. E.; and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Massachusetts, USA: Addison Wesley Professional.


