Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 445–454
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Evaluation of object-oriented design patterns in game development

Apostolos Ampatzoglou, Alexander Chatzigeorgiou *

Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

Received 19 January 2006; received in revised form 30 May 2006; accepted 5 July 2006
Available online 22 August 2006

Abstract

The use of object-oriented design patterns in game development is being evaluated in this paper. Games' quick evolution, demands great flexibility, code reusability and low maintenance costs. Games usually differentiate between versions, in respect of changes of the same type (additional animation, terrains etc). Consequently, the application of design patterns in them can be beneficial regarding maintainability. In order to investigate the benefits of using design patterns, a qualitative and a quantitative evaluation of open source projects is being performed. For the quantitative evaluation, the projects are being analyzed by reverse engineering techniques and software metrics are calculated.

Keywords: Game development; Design patterns; Software evaluation; Software metrics

1. Introduction

Recently, games have become one of the most profitable factors in the software industry. More specifically, during the last few years the game industry has been considered to produce revenue greater than the movie industry and its development rate has been one of the most fast growing in the United States economy [19,24]. Furthermore, game design and the methods used for easier and more efficient development constitute a very interesting open research field [4]. It goes without saying that computer games play a very important role in modern lifestyle. Therefore, it is no longer necessary to explain what a computer game is. On the other hand, it is not so obvious why game research is an extremely interesting field and simultaneously why game development is a very complicated task to accomplish.

The answer to the first question has many levels. As mentioned above, even though game development is a very strong industry, the research on this field is in its infancy. This fact leads game programming professionals to demand better developing methodologies and software engineering techniques [11]. Furthermore, games are the first and sometimes the only market for advanced graphics techniques to demonstrate the quality of graphics they produce [16,22]. It has been acknowledged [21] that game industry draws on research from academia, corporate R&D labs and in-house work by game developers. Several papers pointing out the need for transfer from research to industry appear at SIGGRAPH [25] conferences.

In order to prove what a complex task game development is, we will present the minimum personnel that a typical such company needs. The discrete roles of personnel do not prove the complexity of the task itself, because this is a common tactic for software development teams. The distinction between games and other forms of software is that, in games, the development groups consist of people with different fields of expertise. First of all, a script writer is required; this person will write the game script and fill in a document usually called "concept paper" [15]. The lead game designer will convert information from the concept paper into another called "design document" which will be the guide throughout the development process. Apart from that, the company employs a group of programmers with several skills and expertise, such as engine and graphics programmers, artificial intelligence programmers, sound

* Corresponding author. Tel.: +30 2310 891886; fax: +30 2310 891875.
E-mail addresses: ampatzoglou@doai.uom.gr (A. Ampatzoglou), achat@uom.gr (A. Chatzigeorgiou).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.07.003

programmers and tool programmers. In collaboration with the sound programmers, the game development company will hire a musician and a sound effects group. In addition, the art of the game will be created by graphic artists, such as the character artists, the 3D modelers and the texture artist. Finally, the company must hire testers who would play the game in order to find bugs and make suggestions for changes in gameplay, graphics and the story [10,19,27].

Game programming is a main course in many Universities, at one or two semesters, and there are quite a few Master of Science programmes related to that field, but not many PhD theses on this subject. This fact reveals the industry need for game development methodologies but also the lack of relevant scientific research. A speculation about the absence of scientific research is that games are widely considered a "soft skill" topic. During the last few years this situation has slightly changed with a few publications about game design patterns and the use of game engines in creating virtual worlds and GIS applications. The ways and the extend of teaching game programming in graduate and postgraduate studies has also been examined in a few papers [18,19,24].

In the next sections, the way object-oriented design patterns could be used in computer games and an extended evaluation of their benefits and drawbacks are being examined. More specifically, in chapter 2 a short introduction to general game architecture is being presented. In chapter 3, there is a brief literature review of object-oriented design patters and game mechanics design patterns. In addition, four examples of how object-oriented design patterns could be used are discussed. In chapter 4, two real open-source games are being evaluated. Finally, future research plans and conclusions are being presented.

2. Game architecture

One of the most interesting aspects of game research is the architecture that the developer will use. In recent papers, there are a few references to the modules that the programs are being decomposed to, however, without extensive discussion of maintainability and code reusability issues. Such issues have been examined in detail in classical object-oriented programming, but those ideas are extremely immature in game programming.

Designing and programming large-scale software is a very complicated job that requires many human work hours. Consequently, software is usually divided, logically, into subprograms that are autonomously designed, programmed and tested by separate programmers' groups. These subprograms are called modules. Decomposing software into modules is an important decision that plays a main role in the architecture and further design of the program. In this section, the modules proposed for games are examined and briefly discussed.

In [3], Bishop et al., described a general game's architecture as shown in Fig. 1. This schema presents an interactive game's vital modules. The items with solid outlines

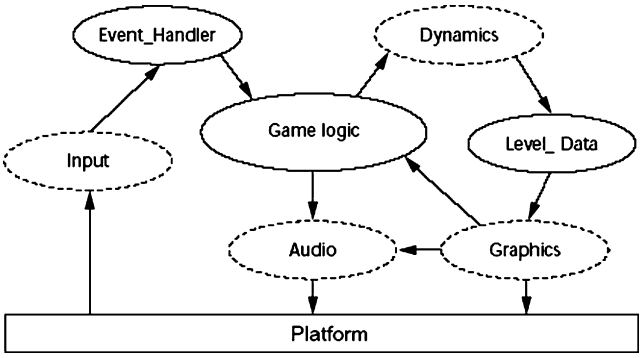

Fig. 1. Generic game architecture [3].

are essential to every game while the dashed outlines refer to modules that are found in more complicated and demanding games. The game logic is the part that holds the game's story. The audio and graphics are the modules that help the writers narrate the story to the player. The event-handler and the input modules, supply the game logic with the player's next action. The level data module is a storage module for details about static behaviour and the dynamics module configures dynamic behaviour of game's characters.

3. Design patterns

With the term design patterns one refers to identified solutions to common design problems. The notion of patterns was first introduced by Christopher Alexander, who identified and proposed solutions to common architectural problems. In his work he dealt with the question whether quality in architecture can be objective. By examining several architectural artifacts he discovered that "good" quality designs shared some common characteristics, or shared "common solutions to common problems" [1]. Patterns can also be used in software architecture and, if applied properly, they increase the flexibility and reusability of the underlying system. Object-oriented design patterns specify the relationships between the participating classes and determine their collaboration. Such solutions are especially geared to improve adaptability, by modifying the initial design in order to ease future changes [12]. Each pattern allows some aspect of the system structure to change independently of other aspects. In [20,29] the authors investigated the effect of design patterns on comprehensibility and maintainability. Their experiment analyzed the consumed time and the correctness of the outcome for the implementation of a given set of requirements employing systems with and without design patterns. The results have indicated that some patterns are much easier to understand and use than others and that design patterns are not universally good or bad. However, it is implied, that if patterns are used properly and in appropriate cases, they prove extremely beneficial regarding maintainability and comprehensibility.

In contrast to the former opinion, Bieman et al. [2], claimed that classes participating in design patterns are more change-prone than the other classes of the system. Their evaluation was based on analyzing existing systems and one of its results suggested that *"classes that play roles in design patterns are changed more often than other classes. The case study data does not show that design patterns support adaptability"*. According to the authors, the pattern participant classes provide key functionality to the system, which may explain why these classes tend to be modified relatively often.

Design pattern usage in game development is an open research field. In literature, it is not easy to find catalogues with patterns that could be used as *"common solutions for common problems"* in games. Such catalogues would make the communication between developers easier and the documentation of this kind of programs more understandable. Design patterns for games can be approached from two different perspectives; firstly as patterns used for describing the game mechanics (gameplay and game rules) and secondly as the use of object-oriented design patterns in programming games.

Concerning game mechanics, Bjork et al. [4] introduced a set of design patterns which essentially are descriptions (employing a unified vocabulary) of reoccurring interaction schemes relevant to game's story and gameplay. As such, these patterns are not related to the software architecture or code. The proposed patterns are collected from interviewing professional game programmers, from analyzing existing games and from transforming game mechanics. As mentioned, "*The way to recognize patterns is playing games, thinking games, dreaming games, designing games and reading about games*" [5]. An example of such a pattern is the *Paper – Rock – Scissors pattern* that is well known in game design community (also as *triangularity*). This pattern is used when there are three discrete states (or options for a player) and option A defeats option B, option B defeats option C and option C defeats option A.

Concerning game programming, in the next sections, examples of object-oriented design patterns in two game modules will be presented. The design patterns examined will be the Strategy and the Observer Pattern concerning the game logic module (Fig. 1) as well as the State and the Bridge pattern concerning 3D aspects in the graphics module (Fig. 1). It goes without saying that object-oriented design patterns can be applied in designing and coding of any game module. The selection of game logic and graphics module does not imply that patterns are more applicable to those fields.

3.1. Object-oriented design patterns in game logic

Although until now there is not much work found on object-oriented design patterns' use in games, we believe that such a use can be proven very useful in this domain. This fact can be examined by investigating the source code of games for the existence of design patterns. As a first approach, we will provide examples of how object-oriented design patterns [12,23] could be used in simple games. In addition to that, in Section 4 we will present real games that could use these object-oriented design patterns and improve their design.

The *strategy pattern*, *defines a family of algorithms*, *encapsulates each one*, *and makes them interchangeable* [12]. Strategy seems a very efficient way to design the way a chess game could simulate different artificial intelligence players' behaviors. Let's assume that a game presents a chess tournament. Every player uses a different heuristic algorithm to calculate the min–max value in order to decide the next computer move.

In the UML diagram of Fig. 2, the class `ComputerPlayer` represents the context. It maintains a reference to a strategy object; it is configured with a concrete strategy object and may define an interface that lets strategy access data. The `SelectMoveStrategy` class declares an interface common to all supported algorithms. It is used by context (`ComputerPlayer`), to call the algorithm defined by a concrete strategy. Finally, the concrete strategy (`Heuristics1MinMax`, `Heuristics2MinMax`, `Heuristics3MinMax classes`) implements the algorithms according to the strategy (`SelectMoveStrategy`) interface [12].

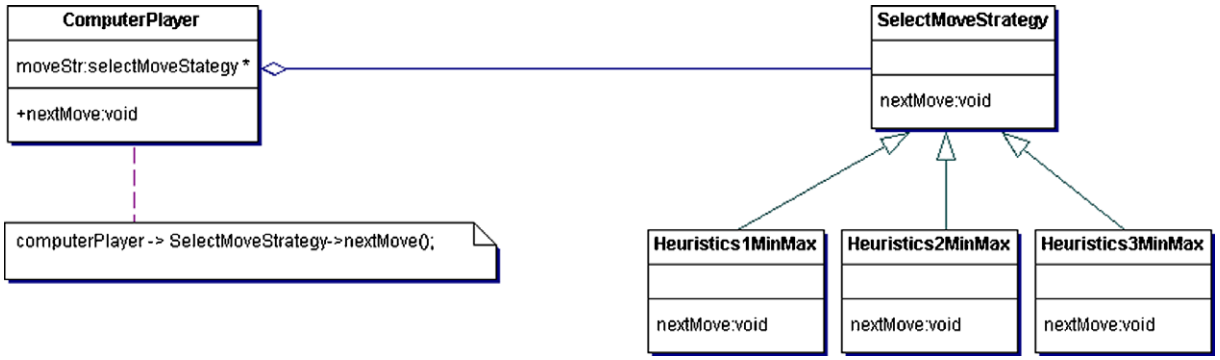

Fig. 2. Strategy pattern – chess example.

Employing this pattern, an alternative to conditional statements is offered for selecting behavior. Additionally, this design is more understandable, more easily maintained and more extendable because of the discrete subclasses [12]. This pattern could have also been used in games such as backgammon, card games and anywhere artificial intelligence algorithms can be applied (e.g. A* algorithms for computer controlled players in RPG – Role Playing Games, adventure and strategy games). However, Strategy pattern is one of the most commonly used patterns and can prove useful in many cases.

Additionally, the *observer pattern defines a one-to-many dependency between objects that when one object changes state, all it's dependants are notified and updated automatically* [12]. This pattern's example is a bit more complicated and is used in more demanding games. Suppose that in a football manager game a team hires trainers that upgrade players' attributes. This does not happen only once and not exactly when the trainer is hired. The upgrade is applied when a special variable reaches a certain value, and its value is increasing according to the coach's skills and the teams training schedule. The team players' attributes must be upgraded automatically when the training level variable reaches the aforementioned value.

In Fig. 3, the class `Team` is the subject that knows its observers and provides an interface for attaching and detaching them. The observer (the class `Players`) defines an updating interface for objects that should be notified of changes in the subject. The class `Training` stores the information about the subjects' current state. The concrete observers (`Attacker`, `Midfielder` and `Defender` classes) implement the observer's updating interface to keep their state consistent with the subject's [12].

This pattern lets the developers vary subjects and observers independently. This way they can reuse subjects without reusing their observers, and vice versa. Additionally, there is an abstract coupling between subject and observers. Moreover, the update notification is broadcasted and as a result the subject is not interested in which observers care about the change, since it is their responsibility to react to it [12].

3.2. Object-oriented design patterns in game graphics

The *state pattern allows an object to alter its behavior when its internal state changes* [12]. This pattern can be used in games where an object changes the level of detail (LOD) of its appearance with respect to its distance from the camera. For example, in any first person shooter game, all objects of a scene should look more elegant as the main character reaches them. Consequently, the image that represents the texture of the object will be of greater resolution, while the object approaches the main character's position.

In the diagram of Fig. 4, the context (`Object` class) maintains a concrete state at any given time. The state (`LOD` class) defines an interface for encapsulating the behavior associated with a particular state of context. The concrete state subclasses (`HighLOD`, `MediumLOD` and `LowLOD` classes) implement the behaviors associated with the states of the context [12].

With the use of this pattern, every behavior associated with a particular state is attached to one object. So, new states and transitions can be added easily by defining new subclasses. In addition to that, the state objects can be shared if they have no instance variables. Finally, state objects protect the context from inconsistent internal states, because state transitions are atomic (the transition between states happen by changing only one variable's value, not several) [12]. This pattern could be applied in the pacman game where states represent the behavior and the velocity of the enemies, in an arkanoid game where the bar's size and attributes are attached to

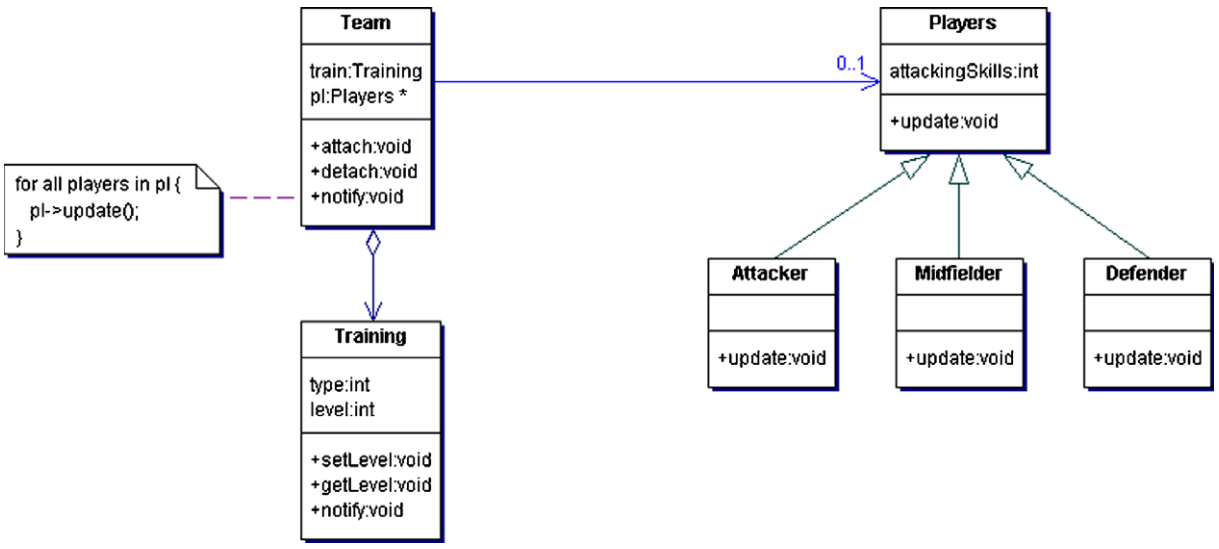

Fig. 3. Observer pattern – football manager example.

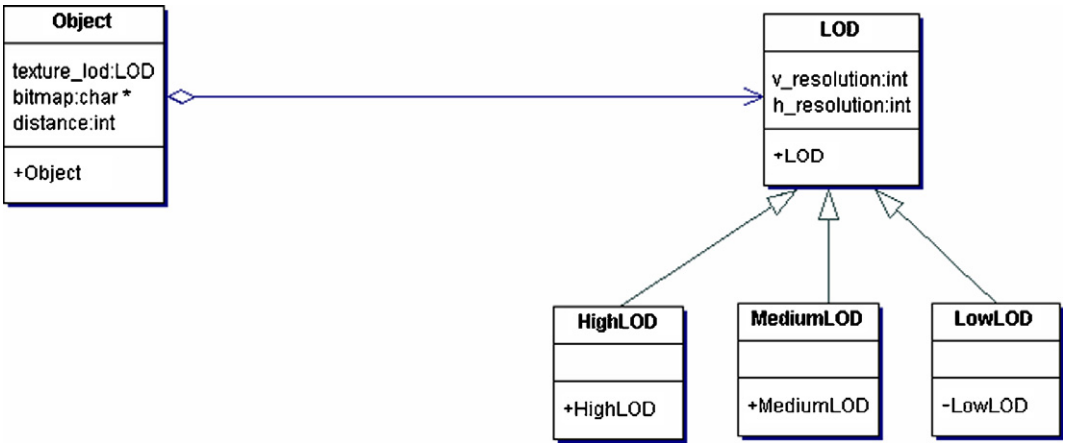

Fig. 4. State pattern – first person shooter example.

concrete state subclasses and also in simple "shoot them up" games such as space invaders and brain wave. In them the player shoots several targets and earns points. During the game he can also change weapons, put armor, shields etc. All these changes can be represented as different states.

The *bridge pattern decouples an abstraction from its implementation so that the two can vary independently* [12]. This pattern can be used in any game with 3D objects that can be painted in multiple ways (texture, material etc). At the same time the 3D object might also vary since it can be a primitive (sphere, cube, etc) or an export from a 3D package (.3ds file, .m2 model, etc).

The abstraction (the `Object` class) defines an abstraction's interface that maintains a reference to an object of type implementor (the `Style` class). The Implementor defines the interface for implementation classes. This interface does not have to correspond exactly to abstraction's interface. Typically the Implementor interface provides only primitive operations, and abstraction high-level operations based on these primitives. The refined abstractions (`model_3ds` class, `primitive` class and `model_m2` class) extend the interface defined by abstraction. Finally, the concrete implementor (`Texture` class and `Material` class) implements the implementor interface and defines its specific implementation [12].

By applying this pattern an implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time and it is even possible for an object to change its implementations at run-time. This way the pattern is decoupling interface and implementation and eliminates compile-time dependencies on the implementation. Additionally, a system with a bridge pattern is more extensible. At any time you can extend the abstraction and the implementor hierarchies independently [12]. In the example mentioned in Fig. 5, the Object and the Style abstractions can vary independently. So, if later on, a new 3D package appears and a loader that handles its models is available it can be added in the program without major difficulty. Similarly, if there is a new requirement for materials of a color type different than RGB (Red–Green–Blue), for example indexed colors (used in openGL) the client can use the structure of Fig. 5 with minor changes.

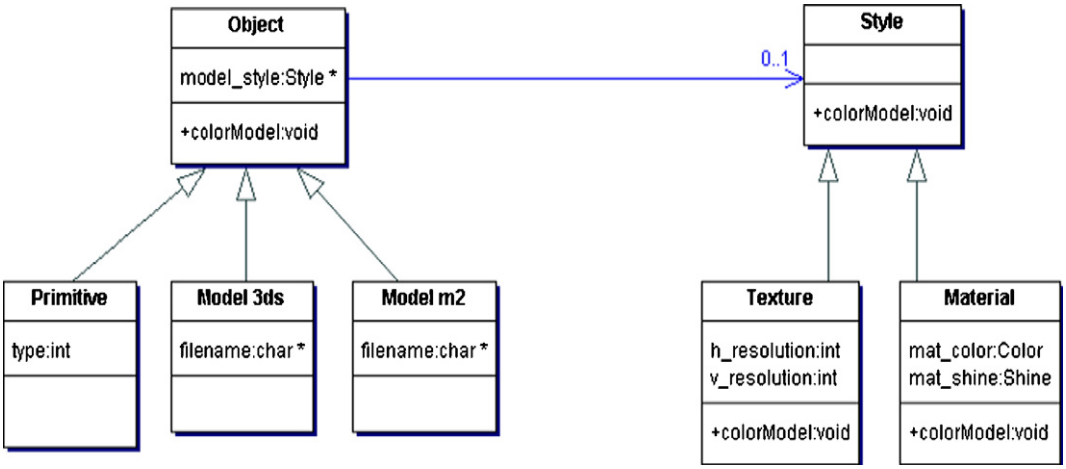

Fig. 5. Bridge pattern – 3D game example.

4. Evaluation

To evaluate the benefits of object-oriented design patterns in game development, we have studied two existing games (Cannon Smash and Ice Hockey Manager) released as open-source in the web [26]. These games have been selected because their code is publicly available and multiple versions of them are released. This fact in combination with the existence of, at least one, design pattern in, at least one, version made the evaluation possible. The evaluation has been performed on two versions of the programs, the first one with the design pattern under study implemented and the other one without it. The two programs evaluated have been developed using different programming languages (C++ and Java). The design patterns in the C++ program have been detected by manual inspection and employing reverse engineering tools. In the Java program a design pattern detection tool has also been used [28]. The evaluation criteria will not only be qualitative but also quantitative, employing well-known metrics. In the first section of this chapter an overview of how this metrics are calculated and why those metrics have been selected will be presented.

4.1. Software quality metrics

The software metrics that have been calculated can be divided into four main categories: size, complexity, coupling and cohesion metrics. These categories have been selected in accordance to [9], where the authors using metrics from the same categories performed an exploratory analysis of empirical data concerning productivity. The effect of design patterns on coupling metric scores has also been investigated in [14]. In [13], size, cohesion and coupling metrics have been used for identifying pattern instances, implying that the presence of a pattern affects their values. Coupling measures is the most obvious choice since according to [12], the use of abstractions in design pattern primarily aims at reducing the dependencies between classes. Patterns also conform to the Single Responsibility Principle [17] and as a result it is reasonable to expect that cohesion increases when patterns are properly used. Moreover, the application of patterns certainly involves the introduction of new classes/interfaces, moving of methods and in most cases requires additional code. Consequently, size metrics are expected to indicate an increase in code size. Finally, the polymorphism involved in many patterns is expected to eliminate complex pieces of code (such as cascaded if statement or switch statements) and thus to reduce the complexity of the corresponding classes.

The size metrics calculated are the *Lines Of Code* (LOC) and the *Number Of Classes* (NOC). The LOC metric is the traditional measure of size. It counts the number of code lines. The NOC metric counts the number of classes in the system [6,8].

The complexity metrics calculated are three, the *Attribute Complexity* (AC), the *Weighted Methods per Class 1* (WMPC1) and the *Weighted Methods per Class 2* (WMPC2). The AC metric is defined as the sum of each attribute's value in the class (each type and array type has a predefined complexity value), so that the complexity is increasing while the value is increasing. The WMPC1 metric is the sum of complexity of all methods of a class, where each method is weighted by its *Cyclomatic Complexity* (CC). CC represents the number of possible paths through an algorithm by counting the number of distinct regions on a flow graph, meaning the number of `if`, `for` and `while` statements in the operation's body. Therefore, the complexity increases if the value of the metric increases. The WMPC2 metric is intended to measure the complexity of a class, assuming that a class with more methods than another is more complex, and that a method with more parameters than another is also likely to be more complex [6,8].

The coupling is being evaluated by using one metric, the *Coupling Factor* (CF). More specifically, the CF metric is calculated as a fraction, where the numerator represents the number of non-inheritance couplings and the denominator is the maximum possible number of couplings in a system [6–8].

Finally, the cohesion is being evaluated by the *Lack Of Cohesion Of Methods* (LCOM) metric. This measure examines for each pair of methods in the class whether they access a common attribute [6,8]. In that case, the two methods are considered to be coherent.

Measurements have been performed for the classes involved in the patterns, the complete program and for Java-based software also for the package that the pattern-participating classes belong to. The aggregate metrics are calculated as the average of their members' metric values.

4.2. Evaluation example 1 – Cannon smash

The game is currently in version 0.6.6 and supports multiplayer gaming. This feature was added in version 0.4.5 where the programmers used an `ExternalData` class in order to handle messages coming from the network, such as instructions from remote players (this operation belongs to the input module of the game shown in Fig. 1). The `Event` class, which is used as an event-handler, using openGL input functions, had an association through reference to an `ExternalData` structure (Fig. 6).

The drawback in the presented approach is that the incoming data could be received in three different types and according to the current type a different reading strategy should be used. In this version the programmers used a *switch* statement to implement this mechanism as shown in Fig. 7.

From Fig. 7, it is obvious that the sample code suffers from "needless repetition" [17]. Therefore, if later a demand for more "reading strategies" appeared, the `switch` state-

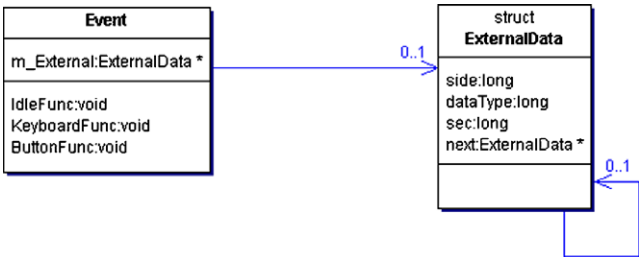

Fig. 6. Cannon smash 0.4.5 approach for multiplayer.

ment should be modified. This proves that the sample code is not reusable, because the program is not flexible enough to easily adapt the extra strategies. Such demands appeared in the next generations of the program. So, in the current version there are five reading strategies in contrast to the three strategies appeared in version 0.4.5.

The above description yields for polymorphism and in particular for using the strategy pattern as described in Fig. 2 (chapter 3.1). This way the complexity of the model will be decreased whereas the flexibility will increase. The developers applied the pattern to the next version of the program (version 0.4.6) allowing them to easily attach more strategies for reading data from network. The UML diagram representing the design of classes handling the multiplayer data in version 0.4.6 is shown in Fig. 8.

The two design approaches have been analyzed using a CASE tool with reverse engineering capabilities in order to calculate the metrics mentioned in Section 4.1. The results are presented in Table 1.

From Table 1, it is clear that the complexity of the `Event` class is decreasing. The most significant value proving this fact is the WMPC1 metric, which is decreasing by 23.6% with the use of the design pattern. The decrease of

```
switch ( theEvent.m_External->dataType ) {

 case DATA_PV:
      targetPlayer->Warp( theEvent.m_External->data );
      if ( targetPlayer == thePlayer )
        fThePlayer = true;
          else if ( targetPlayer == comPlayer )
        fComPlayer = true;
      break;
case DATA_PS:
   targetPlayer->ExternalSwing( theEvent.m_External->data );
   if ( targetPlayer == thePlayer ) fThePlayer = true;
      else if ( targetPlayer == comPlayer )
   fComPlayer = true;
   break;
case DATA_BV:
      theBall.Warp( theEvent.m_External->data );
      fTheBall = true;
      break;}

externalOld = theEvent.m_External;
theEvent.m_External = theEvent.m_External->next;
delete externalOld;
}
```

Fig. 7. Cannon smash 0.4.5 approach for multiplayer – source code.

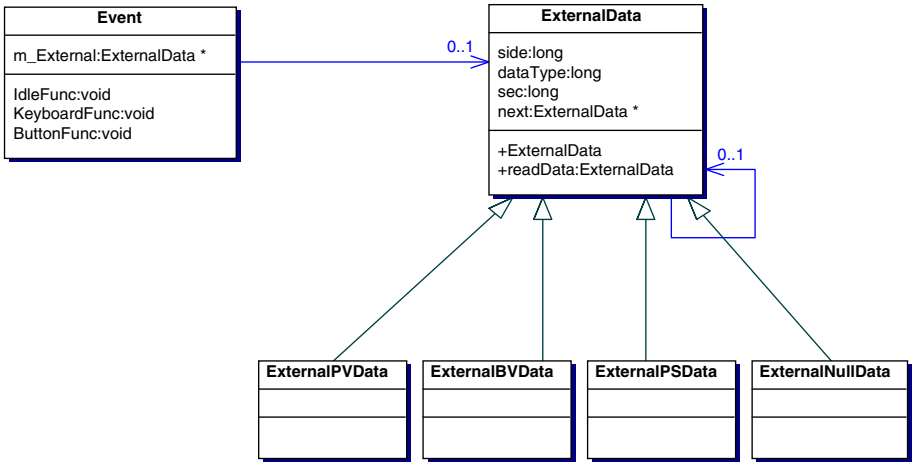

Fig. 8. Cannon smash 0.4.6 multiplayer approach.

Table 1
Metrics cannon smash project

	Version	LOC	NOC	AC	WMPC1	WMPC2	CF	LCOM
Event class	0.4.5	743	–	36	55	44	–	213
	0.4.6	472	–	43	42	43	–	211
Csmash project	0.4.5	8711	43	17	26	19	6	36
	0.4.6	9711	54	14	23	19	5	37

complexity is also reflected by the metrics at the project level, however, at a lower degree, due to the involvement of other classes. The only complexity metric that increases is the Attribute Complexity. However, this is completely irrelevant to the use of the pattern because this increment is caused by the addition of a variable which is not involved in the pattern.

The LCOM metric value's decrease for the `Event` class (even it is minor, 0.9%) suggests that the cohesion between methods is increased and the pattern is correctly used as mentioned in [17]. Moreover, the coupling for the complete program has decreased as it is suggested by the CF metric's value. The corresponding improvement is calculated as 16.6%.

Although the size (lines of code) of the `Event` class is decreasing, the use of the design pattern has increased the LOC and NOC of the complete program. The pattern is responsible for the 4 out of 9 (meaning 44.4%) classes added in the improved version and about 3% of the lines added. Even though this increment cannot be neglected, the used pattern contributes to the reusability, the flexibility and the decrease of complexity of the code.

The results are in agreement with previous studies [20,29] which claimed that in general the application of design patterns leads to more comprehensible and maintainable code. The results of Table 1 imply that a game employing patterns has reduced complexity, reduced coupling and slightly increased cohesion and in that sense is easier to understand, test and maintain. Moreover in [20] it has been observed that code size increases when patterns are used, something which is also verified by the results in Table 1.

4.3. Evaluation example 2 – ice hockey manager

Ice Hockey Manager (IHM) is a simulation of coaching a hockey team. The game's current version is 0.2 and there are two more versions released open-source in [26]. In version 0.1.1 the developers used eight (8) instances of design patterns. More specifically, one (1) factory pattern, three (3) observer patterns, two (2) strategy patterns and two (2) template patterns. Pattern recognition was possible through the use of an appropriate tool [28]. Even though that version's 0.1.1 design was extremely well-structured, some packages and some classes suffered from increased WMPC1, as shown in Table 2.

This problem was alleviated with the use of more design patterns despite the fact that additional functionality was applied. In version 0.1.2 twenty-six (26) instances of design patterns have been identified. More specifically, two (2) factory patterns, one (1) prototype pattern, three (3) adapter patterns, one (1) composite pattern, two (2) decorator patterns, six (6) observer patterns, eight (8) state patterns and three (3) template patterns. Although it is beyond the scope of this paper to describe the use of all 26 patterns, two patterns will be discussed (a bridge and a state), because of their former description in chapter 3.2. Despite that, in Table 2 there are cumulative results that show the aggregative effects of the eighteen (18) additional patterns.

The state pattern represents the fact that a hockey player can either be a goalkeeper or a field player (game logic module, shown at Fig. 1). This means that all players rrespectively of their position, have some common characteristics and they vary in several others. Therefore, a superclass `Player` is created, that encapsulates all the common characteristics and the common behaviour of hockey players. This class is inherited by two sub-classes, the `Goalie` and the `FieldPlayer`, that represent the variant behaviour and characteristics of each category. This way, any client of the program can create or handle a player without having to know in prior whether he is a goalkeeper or a field player. For example, a `Team class`, which holds a reference to an array of players, does not have to handle them separately but through a common interface.

Table 2
Metrics for ice hockey manager project

	Version	LOC	NOC	AC	WMPC1	WMPC2	CF	LCOM
Player attributes class	0.1.1	535	–	91	78	110	–	2237
	0.1.2	169	–	81	41	35	–	218
Player package	0.1.1	1107	6	36	26	34	–	404
	0.1.2	1173	13	26	15	18	–	91
IHM project	0.1.1	8680	86	37	14	14	9	50
	0.1.2	10522	132	33	14	17	5	28

The bridge pattern is used in order to provide a link between the `Player` and the `Player_Attributes` interfaces so that the implementation of class Player can be configured at run-time. Hence, the sub-class `Field_Player` can be linked to the sub-class `Field_player_Attributes` through a common interface for both goalkeepers and field players. This fact can be proven extremely useful for any routines that want to access the player's attributes without knowing the position of the player in the squad (Fig. 9).

The combination of these patterns affects many classes of the project. One of the most characteristic is the `PlayerAttributes` class. In version 0.1.2, the design patterns eliminated two conditional statements decreasing the WMPC1 of the package (one of these conditionals is shown in Fig. 10).

The results of the quantitative evaluation of the project are shown in Table 2. The metrics used are identical to those described in chapter 4.1. The project has been evaluated in three layers. First, metrics for the *PlayerAttributes* class have been calculated, then metrics for the player package and finally metrics for the complete IHM project.

From Table 2, it becomes obvious that design patterns have decreased the complexity of the pattern participating class. This is proven by the WMPC1 and WMPC2 metrics, which reduced their values by 47.4% and 68.2%, respectively, regarding the `PlayerAttributes` class. The `ihm.game.player.*` is the most commonly used package in the added design patterns (in 6 out of 26 patterns, 23.1%), so it shows the most significant reduction of complexity (43.6%, and 57.3%). On the other hand, the complexity of the system remains stable or slightly increases, possibly due to added functionality.

Similarly to the example in 4.2, the cohesion has improved according to the LCOM metric which decreased by 44%, 77.4% and 90.2% regarding the project, the package and the class, respectively. These results seem quite extreme but they reflect the consequences of eighteen (18) additional patterns and not only the bridge that is discussed above. Furthermore, the coupling in the system has decreased by 44.4% as it is implied by the CF metric value.

In contrast to this, the design pattern usage has increased the size of the project regarding the number of classes and the lines of code. In package level, these metrics increased by 116.6% and 5.9%, respectively, and in project level by 25.6% and 21.2%.

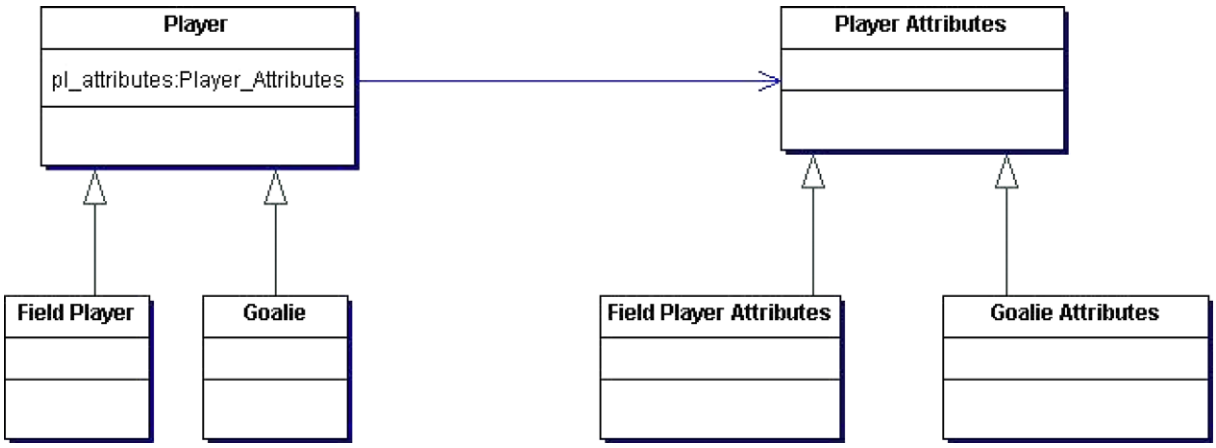

Fig. 9. Ice hockey manager – version 0.1.2 bridge-state pattern.

```
public int get Specific Attributes Average() {
  int specific Attributes = 0;

  // Specific Attributes average
  if (get Position() == POSITION_GOALTENDING) {
    int total = 0;
    total += goalie_glove Left;
     . . .
    specific Attributes = total / 7;
  } else {
    int total = 0;
    total += field_shotQuickness;
     . . .
    total += field_scoring;
    specific Attributes = total / 10;
  }
```

Fig. 10. Ice hockey manager version 0.1.1 – source code.

5. Future research

To draw safe conclusions about the use of patterns in game development, the applicability of other object-oriented design patterns should be examined. Additionally, it should be investigated whether classes participating in patterns are more change prone. In order to achieve this task several games' and game engines' source code is currently being examined and reverse engineering techniques are applied to them. Although sessions 4.2 and 4.3 examined the game logic part of the corresponding programs for the existence of design patterns, there is a belief that modules related to 3D rendering are an excellent domain for design pattern detection. Finally, a complete game engine, or a game, that uses all conclusions from the above research, is planned to be implemented.

6. Conclusions

This paper aimed at evaluating the use of object-oriented design patterns in game development. In order to achieve this goal we examined two open-source games. The results extracted by the two games were almost identical and indicate that patterns can be beneficial with respect to maintainability. The game version that includes the pattern under study has reduced complexity and coupling compared to a prior version without the pattern. Additionally, the application of patterns tends to increase the cohesion of the software. In contrast to that, the size of the projects has increased in the pattern version. Consequently, due to the evolving nature of games we believe that the appropriate employment of design patterns should be encouraged in game programming.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language – Town, Buildings, Construction, Oxford University Press, New York, 1977.
[2] J.M. Bieman, D. Jain, H.J. Yang, OO design patterns, design structure, and program changes: an industrial case study, in: International Conference on Software Maintenance (*ICSM 2001*), Florence, Italy, November 2001, pp. 580–590.
[3] L. Bishop, D. Eberly, T. Whitted, M. Finch, M. Shantz, Designing a PC game engine, IEEE Computer Graphics and Application (1998) 46–53.
[4] S. Bjork, S. Lundgren, J. Holopainen, Game design patterns, in: Lecture Note of the Game Design track of Game Developers Conference 2003, March 4–8, San Jose, CA, USA, 2003.
[5] S. Bjork, S. Lundgren, J. Holopainen, Game design patterns, in: Proceedings of Digital Games Research Conference 2003, Nov. 4–6, Utrecht, The Netherlands, 2003.
[6] Borland Together Control Center 6.1 product documentation, http://info.borland.com/techpubs/together/tcc61/, 2005.
[7] F. Brito e Abreu, The MOOD Metrics Set, in: Proceedings of the Ninth European Conference Object-Oriented Programming (ECOOP '95) Workshop Metrics, Aug. 1995.
[8] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[9] S.R. Chidamber, D.P. Darcy, C.F. Kemerer, Managerial use of metrics for object oriented software: an exploratory analysis, IEEE Transactions on Software Engineering 24 (1998) 629–639.
[10] C.E. Crooks II, Awesome 3D Game Development, Charles River Media, Hingham, Massachusetts, 2004.
[11] M. Doherty, A software architecture for games, University of the Pacific Department of Computer Science Research and Project Journal (RAPJ) 1 (1) (2003).
[12] Gamma, Helms, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, Reading, MA, 1995.
[13] Y.G. Gueheneuc, H. Sahraoui, F. Zaidi, Fingerprinting design patterns, 11th Working Conference on Reverse Engineering (WCRE'04), pp. 172–181.
[14] B. Huston, The effects of design pattern application on metric scores, The Journal of Systems and Software 58 (2001) 261–269.
[15] J. Laird, Game production time line, University of Michigan http://ai.eecs.umich.edu/soar/Classes/494/talks/Game-timeline.pdf.
[16] M. Lewis, J. Jacobson, Game engines in scientific research, Communications of the ACM 45 (1) (2002) 27–31.
[17] R.C. Martin, Agile software development: principles,patterns and practices, Prentice Hall, Upper Saddle River, NJ, 2003.
[18] M. Masuch, M. Rueger, Challenges in collaboration game design developing learning environments for creating games, Proceedings of the 3rd International Conference on Creating, Connecting and Collaborating through Computing (C5'05), Jan. 2005, Kyoto, Japan, pp. 67–74.
[19] G. Pleva, Game programming and the myth in a child's play, Journal of Computing Sciences in Colleges 20 (2) (2004) 125–136.
[20] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, L.G. Votta, A controlled experiment in maintenance comparing design patterns to simpler solutions, IEEE Transactions on Software Engineering 27 (12) (2001) 1134–1144.
[21] C. Raynolds, Game research: the science of interactive environment, SIGGRAPH Conference, July 2000, New Orleans, USA. http://www.siggraph.org/s2000/conference/courses/crs39.html.
[22] T.M. Rhyne, P. Doenges, B. Hibbard, H. Pfister, N. Robins, The impact of computer games on scientific and information visualization: if you can't beat them, join them (panel), IEEE Visualization, Proceedings of the conference on visualization '00, Salt Lake City, Utah, USA, 2000, pp. 519–521.
[23] Shalloway, J. Trott, Design Patterns Explained. A New Perspective on Object Oriented Design, Addison-Wesley Professional, Boston, MA, 2001.
[24] G.A. Shultz, The story engine concept in CS education, Journal of Computing Sciences in Colleges 20 (1) (2004) 241–247.
[25] SIGGRAPH Conference, http://www.siggraph.org.
[26] Sourceforge.net, http://www.sourceforge.net.
[27] P. Stacey, J. Nandhakumar, Managing projects in a games factory: temporality and practices, Proceedings of the 38th Hawaii International Conference on System Sciences, 2005, pp. 1–10.
[28] N. Tsantalis, Design Pattern detection",http://java.uom.gr/~nikos/pattern-detection.html, 2005.
[29] M. Vokác, W. Tichy, D.I.K. Sjøberg, E. Arisholm, M. Aldrin, A controlled experiment comparing the maintainability of programs designed with and without design patterns – a replication in a real programming environment, Empirical Software Engineering 9 (2004) 149–195.