GOF design patterns applied to the Development of Digital Games

Roberto Tenorio Figueiredo
Faculdade de Ciências Aplicadas e Sociais de Petrolina
FACAPE
Petrolina, Brazil

Geber Lisboa Ramalho
Centro de Informática
Universidade Federal de Pernambuco – UFPE
Recife, Brazil

Abstract

The game market is very competitive, requiring companies to react rapidly to opportunities and demands. The adoption of libraries and frameworks has helped developers to focus on game logic, improving reuse and, consequently, fastening the development. Unfortunately, these tools do not solve all problems, since component reuse does not replace coding completely. In fact, developing game logic involves several elements that are particular to each game and cannot be easily generalized. In this context, game industry could take advantage design patterns, one of software engineering techniques to aid developers in coding recurrent situations or problems. A group of design patterns is quite famous as defined by the GoF (Gang of Four), composed of twenty-three patterns with recognized commercial applications use. Unfortunately, the adoption of GoF design patterns is very limited in game development. This work is a pioneer effort on explaining in details how to use some the GoF patterns in the development of games. This paper not only shows the final result, but it presents the "before and after" the application of these patterns and where they are assisting the programmer in his or her task. The positive impact of adopting design patterns has already been proved for the software industry in general, but in this paper, for sake of completeness, we illustrate this impact in a small experiment. The results have confirmed the interest in using design patterns in game development.

Keywords: Design Patterns, GOF, Game Development.

Authors' contact:
tenorio.petrolina@bol.com.br
glr@cin.ufpe.br

1. Introduction

The gaming industry was around 420 million dollars in 2011, only in Brazil [2014], which generated a heated battle between developer companies, who try to meet the demands of an increasingly demanding and competitive market. Looking at these numbers, plus the great competition in the industry, developers, every day, look for new forms of programming that combine low cost, agility, quality and acceptance in a competitive market.

In this context, reusability has an important role and there are several tools for improving reuse, such as game engines, libraries, etc. [Perucia et al. 2005]. These tools are already widely used in the production and development of games, but there are other tools and techniques that can help that are less used, for example, some DSLs (Domain Specific Languages) and design patterns [Perucia et al. 2005].
Design patterns are one of the software engineering techniques to aid developers in coding recurrent situations or problems. A group of design patterns is quite famous as defined by the GoF (Gang of Four) with recognized commercial applications use [Gamma et al. 2000]. Twenty-three design patterns by Gof were defined. They are: Abstract Factory, Builder, Factory Method, Prototype, Singleton, Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy, Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Unfortunately, the adoption of GoF design patterns is very limited in game development, according to the survey done in the section II. Many contest this assertion, due to Design Patterns are widely known and used, however, a research more depth confirms its veracity.

This work is a pioneer effort on explaining in details how to use some the GoF patterns in the development of games. This paper not only shows the final result of the application of pattern, but also presents the entire process of adding the patterns where it matters, the "before and after" the application of these patterns and where they are assisting the programmer in his task. This detailed explanation is achieved with the presentation of class diagrams, with and without the application of patterns, examples, and discussions, comments and relevant comments.

The positive impact of adopting design patterns has already been proved for the software industry in general [Deitel P. J.; Deitel H. M. 2005]. However, for sake of completeness, we present in this paper a small experiment presented on the topic "Experimentation" this paper. The results have confirmed the interest in using design patterns in game development.

2. Related Work

There are a lot of articles that comment about the application of design patterns in games, but many of them define their own patterns. Among the articles that comment about the application of the GOF patterns, we have a very limited amount of patterns worked. Articles include between one and nine patterns, many of them repeating itself among the articles having more than ten patterns that were not mentioned in any article.

The search of related work featured techniques of data mining over the Internet and periodic CAPES site, in addition to the annals of every year Congress on the topic, such as ICSE (International Conference on Software Engineerin), CSEE&T (Conference on software Engineering Education and Training), SEKE (Conference on software Engineering and Knowledge Engineering), CBSoft/SBES (Brazilian Conference on Software / Brazilian Symposium on software Engineering), SESRes (Software Engineering and Systems Research), GDC (Game Developers Conference), SBGames (Brazilian Symposium on Games and Digital Entertainment), among others. In addition, all references of retrieved articles were consulted in search of new references.

The works found were scrutinized. Follows the analysis of the works most relevant to the theme of the research.

In his work, Ampatzoglou and Chatzigeorgiou [2006] show the application and the use of design patterns in games, trying to make the code more flexible and reusable game, lowering maintenance costs, however, despite making quotation from eleven of the twenty and three GoF patterns, explains effectively only four (Strategy, Observer, State, Bridge). The research is not just the patterns, it is the focus on a few topics.

Have Trinidad and Fischer [2008] present the GoF design patterns Singleton, Observer and Adapter and further defines the Data Access Object and Monitor patterns which address structural aspects and applicability of each patterns as well as implementation examples, however, only the Singleton pattern have your example with a focus on game development.

Despite not speaking directly about the application of the GoF design patterns in games, the work of Björk and Holopainen [2001] investigates the relationship between the application of the design pattern and found bugs in software. To achieve its goal, an empirical study on games developed in Java was conducted. This research identified the number of defects, the clearance rate and the patterns used in games. The results show that the total number of use of patterns is not correlated directly with the bugs. However, some design patterns has a significant impact on the number of reported bugs. Among the GoF patterns discussed are the singleton, composite, adapter, observer, state, strategy, template method, decorator, prototype, proxy and abstract factory.

The article Kaae [2001] comments on how the design patterns can help the programmer in the early stage of game programming, but does not directly address the GoF patterns, only comments about how to apply the architectural pattern MVC (Model-View-Control). Another study showing the application of MVC in games is the work done by Wong and Nguyen [Wong S. B.; Nguyen D. 2002], however, despite the focus of his research is the MVC, it highlights the important use of GoF state, strategy, and visitor patterns in games.

Gestwicki [2007] presents a model to support the design, analysis and development of games with design patterns. The model consists of a structural framework for describing the components of the games and the interaction patterns that describe how components are used by the players (or computer). The study validates the use of five patterns in GoF building games. They are: state, facade, observer, strategy, and visitor.

In addition, some articles that comment about the use of design patterns in games are educational character, ie, using games to facilitate the learning of design patterns in the undergraduates. Among the works in this direction, it is worth highlighting Silveira and Silva [2006], showing how games can be used as motivators in learning design patterns, being its main focus the architectural patterns, citing and explaining just the decorator pattern but others without citing a study of its application.

In the works of Gestwicki and Sun [2007] and Gestwicki and Sun [2008] an approach for teaching design patterns that emphasizes object-orientation and the integration of patterns is presented. The context of the development of computer games is used to engage and motivate students, a case study is presented based on EEClone, a computer game in arcade style implemented in Java. These works focus on the GoF singleton, facade, observer, state, strategy, and visitor patterns.

According to Martín, Díaz and Arroyo [2009], the design of object-oriented software requires a combination of abilities that cannot be easily transferred to the students in traditional classes. Their studies show that can increase students' understanding of design patterns through an approach that consists in the development of a family of games in an incremental way strategy. In the development of these games, it is evident the use of architectural pattern MVC and GoF observer, strategy, template method, factory method, abstract method and proxy patterns.

Although this educational line, Wick [2005] discusses the complexity of the design patterns and learning how this complexity can be reduced with the use of digital games as practical examples of the application of patterns. This paper discusses the GoF patterns observer, state, singleton, command and visitor.

3. Design Patterns

Design patterns represent a considerable advance in the area of object-orientation, as it provides a catalog of project plans to admit reuse these solutions that have been tested and proven to be efficient for solving similar [Gamma et al. 2000] problems.

The use of the design pattern is of paramount importance as it offers a facility at the time of maintenance, since it leaves the project well documented, the scalability of the project, ie, the ability to manipulate the system and support the full load required by resources, the reuse of all materials, resulting in significant increases in productivity [Larsen S.; Aarseth E. 2006]. Design patterns are considered a way to represent record and reuse projects micro architectures repeated as often as necessary, and also the experience accumulated by designers throughout the development of the project [Larsen S.; Aarseth E. 2006]. The importance design patterns are to know exactly what's wrong and what's best for him. It is important to analyze the case and the solution to the situation, because it is through this analysis and knowledge of design patterns that you can decide which to use, how to use and why to use a particular design pattern, if it really is the best choice to solve this problem [Gamma et al. 2000].

Another point that should be taken into consideration is that most of the time a project is not developed alone. So if the entire development team already has knowledge of design patterns, if necessary explain how we developed a any functionality will be saved a long time [Gamma et al. 2000]. Design patterns used efficiently inheritance, polymorphism, composition, modularity and abstraction, very important for the development of object-oriented concepts projects, thus building a reusable, efficient code, high cohesion and low coupling [Gamma et al. 2000].

4. Applications of GOF design patterns in games

The intention of this chapter is to show in detail each of the 23 GoF patterns and their application in the development of digital games, however, due to the limitation of pages, only a few will be presented. The complete list of GoF patterns for game can be accessed in http://www.osfedera.com/get/federa/Dis_final.zip.
The examples this section are only didactic situations to explain how the patterns could be used.

4.1. Builder

Where to apply: In many games, many criticisms are made towards enemies, because they are exactly the same or are only slight variations of the enemies of previous stages. An example of this can be seen in the game "Street of Rage". The class diagram of a game designed to generate enemies without using patterns can be seen in Figure 4.1, where you can see a class "enemy", identified as "father" and several classes of enemies, who inherit this "father "and alter some characteristic. Every new feature to be changed, a new class must be created, thus occupying more space in memory. The creation of a wide variety of enemies is something costly and time-consuming the games, besides occupying memory and disk space [Perucia et al. 2005].

Solution proposed by the pattern: One way to optimize the creation of varied enemies, reducing the resource consumption of the machine is using the Builder pattern. The patterns proposes the creation of a complex enemy with various items, moves, weapons and garb quite different, various types of punches and kicks, as well as different styles of fighting. The concrete builder class will create several characters, starting this enemy "master", separating some of these features and elements to compose several distinct characters. The diagram with an example of applying the pattern builder can be seen in Figure 4.2.

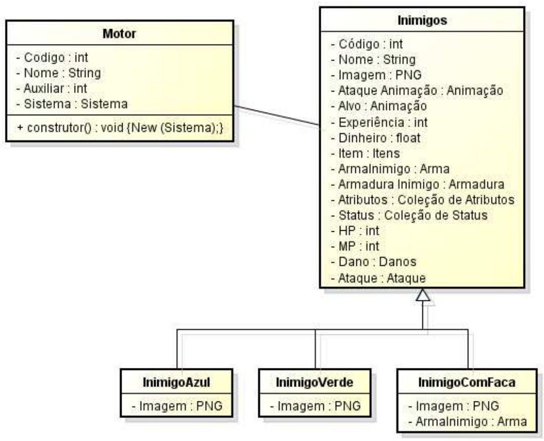


Fig. 4.1.: Partial class diagram of a game that shows the creation of enemies without using the Builder pattern. The enemies have slight variations of an enemy father.

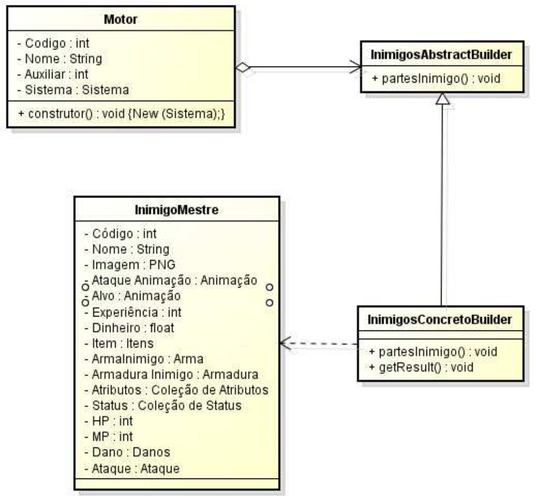


Fig. 4.2. Class diagram of a partial game, which shows the creation of enemies using the patterns Builder. Each enemy may have some features that made it unique in the game, and all these features come from a single class enemy.

4.2 Prototype

Where to apply: As previously discussed, the creation of identical or extremely similar enemies can generate criticism from his players, but on devices with limited resources (mobile phones, tablets, etc.) this idea can be the only way to create groups of enemies. One difficulty that can arise in creating enemies is replicated code duplication in each unit of these elements, generating effort in creating and possible adjustments to these elements may undergo during development. An example of replication can be seen in figure 4.3 of class diagram showing three enemies in a replicated stage.

Solution proposed by the patterns: A solution to this problem is proposed by default prototype. As can be seen in Figure 4.4, using the default causes the classes of enemies are independent of the choice phase and the amount of generated enemies is done at runtime and can be adjusted according to the need of the game and the load instantiated objects that the device supports. The gerarInimigo () method creates a new object requesting that the InimigoPrototipo class (one of his sons) be doubled.

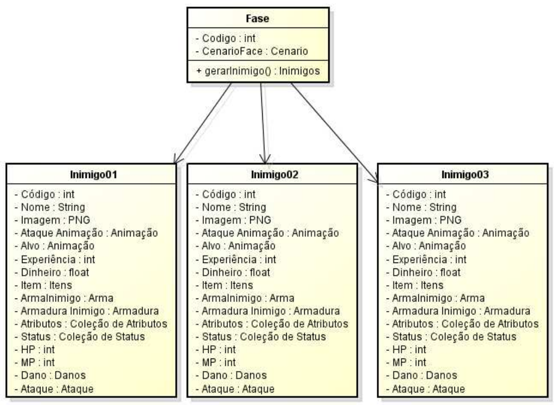


Fig. 4.3: Partial diagram of a game showing the generation of identical without the use of patterns Prototype enemies.

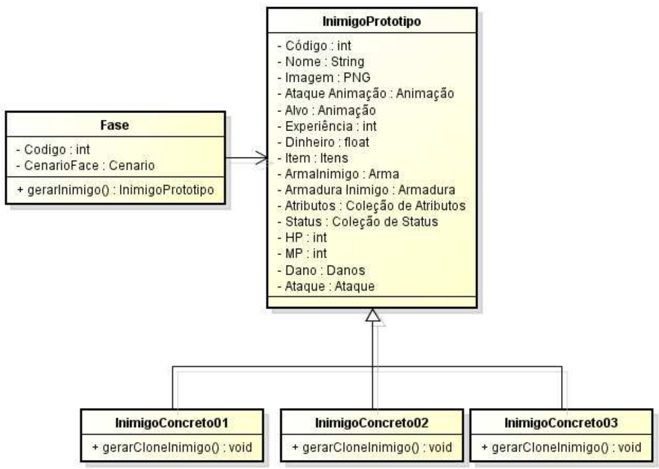


Fig. 4.4: Partial diagram of a game showing the generation of identical enemies using the Prototype pattern.

4.3 Singleton

Where to apply: Several elements of a game cannot be replicated anywhere else, such as replication can lead to inconsistencies, both logical, as the game's storyline. An example of this is the main character that the player will control during the game. In many games, the main character can only appear once. An example of a problem that can happen with a doubling of this character is the inconsistency of events.

Solution proposed by the pattern: One way to ensure a single instance of any object is to use the Singleton design pattern. Figure 4.5 shows a situation without the use of the pattern, where one wrong programming in some class method engine can generate more than one instance of the Character class. In Figure 4.6 has become the application of the Singleton pattern, which suggests a class to instantiate the class character through a static method that will not allow a second instantiation, thus ensuring the uniqueness of the instance of the Character class, avoiding inconsistencies and errors.

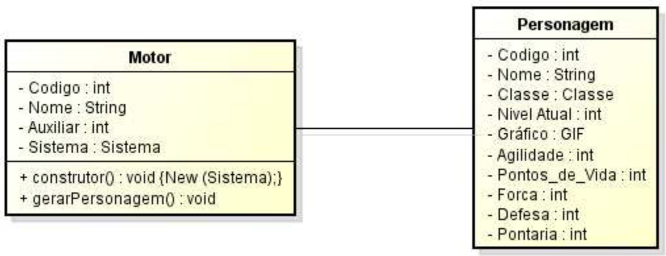


Fig. 4.5: Partial diagram of a game without using the Singleton pattern. The game engine can instantiate more than one character, because there is no guarantee of uniqueness.

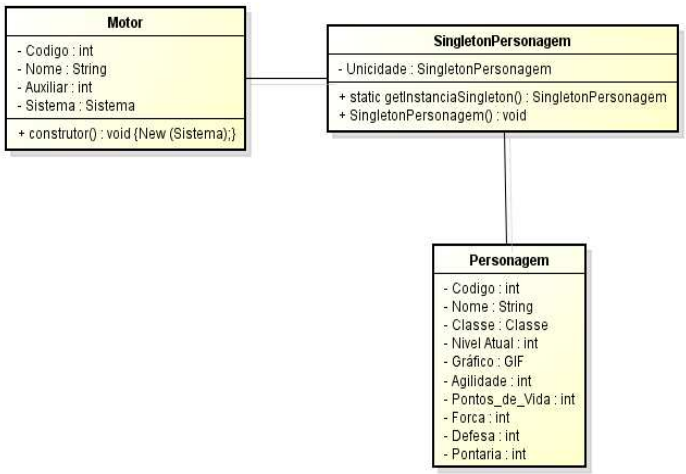


Fig. 4.6: Diagram of a partial match using the Singleton pattern. The character can only be instantiated via the Singleton class, which ensures the uniqueness of the instance of the class Character.

4.4 Flyweight

Where to apply: Each occasional enemies in a game is an instance of an object. These instances consume memory with information, often repeated, because the enemies usually have common characteristics. When these enemies go in droves and crowding the screen, several concurrent instances in real memory, which can cause slowdowns and even overhead will be generated. The first solution is to limit the number of simultaneous enemies, which can distort the game's plot, and impact on other areas of the game such as difficulty and dynamic game options.

Proposed solution by patterns: A solution to reduce the impact of these problems, increasing the ability to generate simultaneous enemies, using minimal memory is to use the Flyweight pattern. With the default, a shared resource that can be used by several enemies simultaneously object is created. This object has no specific link with any enemies, and contains only the information that is common among them. Thus, each instance of an enemy only has exclusive information, leaving the smaller objects, thus taking up less space in memory. The diagram shown in Figure 4.7 shows the generation of enemies the engine without the use of the pattern. Each enemy will have all the information object class "Enemy", occupying as much memory. In Figure 4.8 we have the implementation of the Flyweight pattern, it is possible to note the FlyweightAbstrato class that is nothing more than an interface in which the concrete Flyweights may aggregate the information specific to each enemy general information Flyweight. There is also the "InimigosInformacoesNaocompartilhadas" class that can store information that need not be shared by Flyweight. The engine maintains references to Flyweights because it stores or generates information specific to each enemy, but can only generate instances of Enemies by FlyweightFabricaDeInimigos class, thus ensuring the appropriate sharing of information.

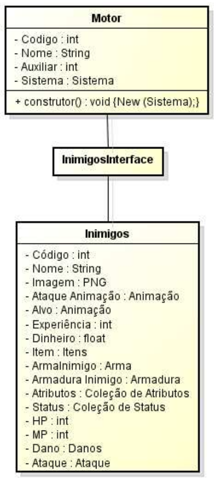


Fig.4.7: Diagram of a partial match without the Flyweight pattern. The engine generates only complete instances of enemies when necessary.

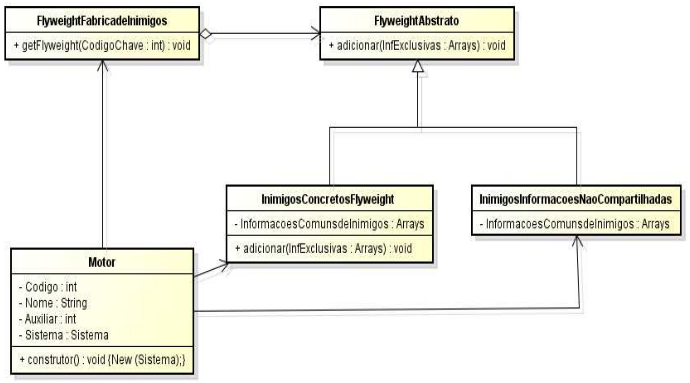
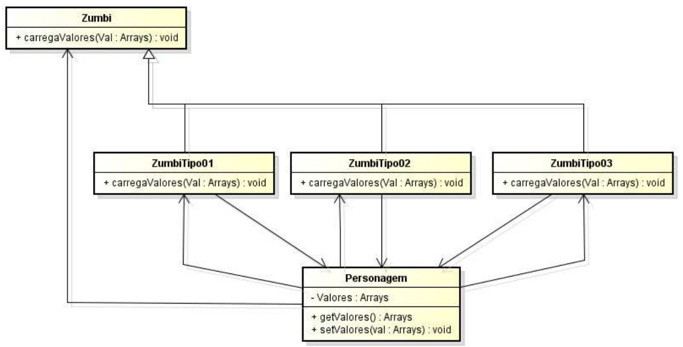

FlyweightFabricadeInimigos
+ getFlyweight(CodigoChave : int) : void
FlyweightAbstrato
+ adicionar(InfExclusivas : Arrays) : void
InimigosConcretosFlyweight
- InformacoesComunsdeInimigos : Arrays
+ adicionar(InfExclusivas : Arrays) : void
InimigosInformacoesNaoCompartilhadas
- InformacoesComunsdeInimigos : Arrays
Motor
- Codigo : int
- Nome : String
- Auxiliar : int
- Sistema : Sistema
+ construtor() : void {New (Sistema);}

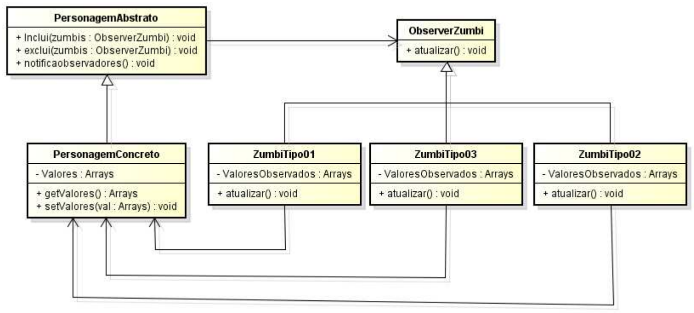
Fig. 4.8: Partial diagram of a game with the Flyweight pattern

4.5 Observer

Where to apply: In most current games, intelligent agents are widely used [Perucia et al. 2005]. These agents are usually the enemies of the player character, where the AI is applied. The strategy of attacking enemies is calculated based on the character's movement in order to avoid their attacks and surround it on all sides. The problem happens when objects of enemies must receive information handling and attack the player, because any change in the character must be reported, in addition, each map, the amount of zombies vary, so there is no simple way to number of objects that need to be notified. Obviously communication can not affect the consistency of communicating objects. One way to establish communications without the use of any patterns is illustrated in figure 4.9. A routine is implemented in character with a reference for each zombie one by one passing the necessary information.

Proposed solution by patterns: A solution to accomplish this communication, with weak coupling is the application of the Observer pattern. The character class knows its observers from a list and has methods to include them or exclude them, as these can vary with the progress of the game. The "ObserverZumbi" class defines an interface for objects that need to be notified. The "PersonagemConcreto" class stores the interesting information for the zombies and notifies them whenever a change occurs. Once informed of a change, the zombies consult the character and use the information obtained to adjust your strategy of action. The diagram of the pattern can be seen in Figure 4.10.


Zumbi
+ carregaValores(Val : Arrays) : void
ZumbiTipo01
+ carregaValores(Val : Arrays) : void
ZumbiTipo02
+ carregaValores(Val : Arrays) : void
ZumbiTipo03
+ carregaValores(Val : Arrays) : void
Personagem
- Valores : Arrays
+ getValores() : Arrays
+ setValores(val : Arrays) : void
Fig.4.9: Partial diagram without the use of the Observer pattern. There is a strong coupling between the character's class and the classes that represent the enemy.


PersonagemAbstrato
+ Inclui(zumbis : ObserverZumbi) : void
+ exclui(zumbis : ObserverZumbi) : void
+ notificaobservadores() : void
ObserverZumbi
+ atualizar() : void
PersonagemConcreto
- Valores : Arrays
+ getValores() : Arrays
+ setValores(val : Arrays) : void
ZumbiTipo01
- ValoresObservados : Arrays
+ atualizar() : void
ZumbiTipo03
- ValoresObservados : Arrays
+ atualizar() : void
ZumbiTipo02
- ValoresObservados : Arrays
+ atualizar() : void
Fig. 4.10: Partial Diagram using the Observer pattern. The objects of the enemies are reported every movement of the character, without the strong engagement by the proposed solutions without the use of the patterns.

4.6 State

Where to apply: In any game, the events that can occur with a character, will depend on certain characteristics of the transient at a given moment. An example of this happens in the game Sonic The Hedgehog, the event where the main character bump into an enemy may have different consequences according to your current feature. The figure 4.11 illustrates different states of the Sonic character. In figure 4.11 (a) have their normal state without rings to bump into an enemy in this state the character loses a life and return to the beginning of the stage, if you have even more lives. In figure 4.11(b) have the character with a shield, to bump into an enemy in this state the character simply lose the shield, will show a jump back and the game proceeds normally. The figure 4.11(c) shows the character with the call temporary invincibility, to bump into an enemy in this state, the enemy dies and no result will be transmitted to the character.

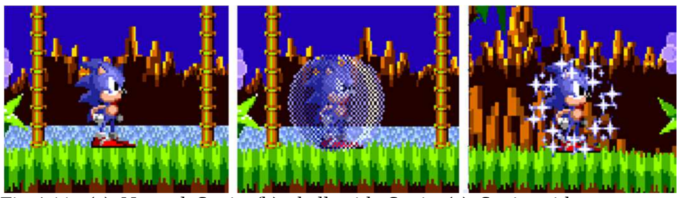


Fig.4.11: (a) Normal Sonic (b) shell with Sonic (c) Sonic with temporary invincibility

Another example can be seen in games Marvel Avengers Alliance. Upon receipt of enemies or aid allies attacks the character can have temporary gains or losses depending on what you have been given. In the case of earnings, for example, the character may get "Resists and Burn", the state where the character can not be burned; "Resistant to Poisons" state, where the character can not suffer poisoning; "Resistance to Bleeding" state, where the character suffers no damage because of bleeding; "Agil", where the character has been his ability to escape increased by 25%; "Strengthened" state where the character has his strength increased by 25%; "Protected" status, where your character has increased by 25% and "Focused" state where your character has increased by 25% marksmanship defense. In the case of losses, the character can be "burned", a state that makes you suffer damage each turn of struggle and has reduced its defense; "Poisoned", the state where the character takes damage each turn and has lowered his attack; "Bleeding" state, where the character takes damage each turn and every attack made; "Slow", where the character has been its ability to reduced leakage by 25%; "Weakened" state where the character has its strength reduced by 25%; "Exposed" state, where the character has their defense reduced by 25% and "Tonto" state, where the character is diminished on 25% shooting.

Proposed solution by default: Changes made to objects and reactions characters and events become independent of the state that the object is. The State object is responsible for assessing the state of the object and make the appropriate changes and reactions. In Figure 4.12 we can see the partial diagram of the game Marvel Avengers Alliance without applying the State pattern. You can see that the states of character are attributes and methods of the Character class, causing a strong coupling between the states and the individual character's actions, leaving the heavier and operators will not be used constantly class, since the character is not always burned, exposed, dizzy, weakened, poisoned, bleeding and reading simultaneously. In Figure 4.13 we have the diagram using the State pattern, there is now the states of character that are decoupled from the main class, allowing states can be removed or added without any change in the character class and whose actions may be modified without any concern for the rest of the system.

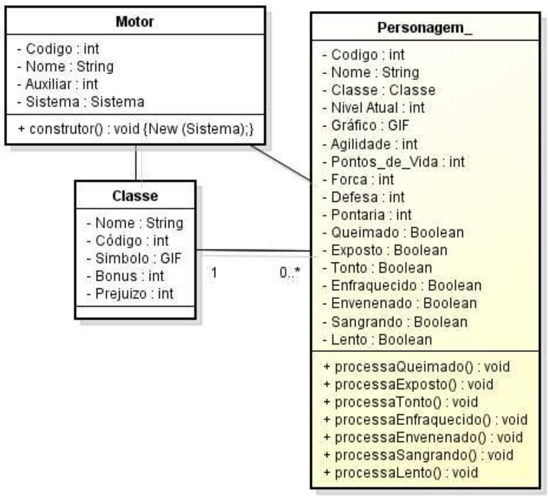


Figure 4.12: Partial diagram of the game Marvel Avengers Alliance, without the State pattern, in case of losses.

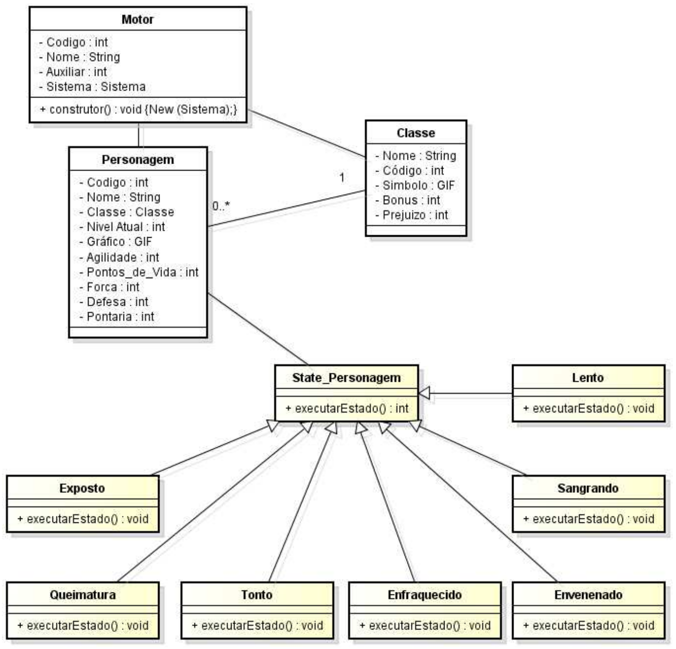


Figure 4.13: Partial diagram of the game Marvel Avengers Alliance, with the State pattern, in case of loss

5. Experimentation

As discussed, it’s been proved that the use of design pattern promotes a significant increase in productivity on software development [Larsen S.; Aarseth E. 2006]. However, to illustrate the impact of the application of design patterns proposed here in the development of digital games, the experiment was conducted. The objective of the experiment was to examine the improvements and quantize the adoption of design patterns in games, in relation to the reduction in development time, decreased in the presence of bugs and reducing lines of code. The participants were students of computing FACAPE. Students enrolled in the experiment were divided into groups in a total of six groups with three people each. The idea was to apply a AB test to compare the groups that used design patterns with those that did not. The division of the groups tried to balance the participants experience in C# language and in game development. Because of the time that students had and the complexity involved in the process, only three design patterns were used in the experiment, they were: Singleton, Prototype and Facade. These patterns were chosen because of the ease in its development and the possibility of being applied to a simple game. The day before the execution of the experiment, these three groups (referred to as groups A, B and C) received an explaination on the three patterns GoF, without no specific hints on how to use them in the game they will implement the next day. To the other groups (called D, E and F), nothing was explained. On the day of the experiment, this task description was given: "Each group will have a computer available and should develop a set of specifications that are being passed on to them in Visual Studio 2010 tool with XNA Game Development Library previously installed both in the laboratory. All of you have received an ongoing project and will use everything that the project offers. Whoever finishes first with all specifications, will win a prize. Get to work". The requirements of how the game should be and the images to be used, were provided to all groups. Groups A, B and C, who had received lessons on Design Patterns, also received the ready classes for the three design patterns that should be implemented.

The purpose of classes be delivered ready it is demonstrate that a key advantage of the design patterns, the code reuse, really decreases the development time and the number of lines typed code.

The game used in the experiment was a simple space ship game with a single screen where a player throws bullets to up to attempt to hit in the space ships enemies that shoot bullets down, trying to hit the space ship player. If the player is hit three times, he dead. If hit ten enemy space ships, will be deemed the winner.

Results:

1	2	3	4	5	6	7
A	Yes	06h50m	100%	0	246	21
B	Yes	06h36m	90%	1	253	23
C	Yes	06h05m	100%	0	218	25
D	No	07h40m	100%	1	345	15
E	No	09h45m	80%	1	259	1
F	No	06h40m	100%	1	251	6

1 – Group	2 – Use of patterns
3 – Approximate time	4 – Completeness
5 – Bugs	6 – Lines of Code
7 – Number of Classes	

With the results obtained can conclude that the average implementation time of the game with the use of patterns was 06:31, while the average of the teams that did not use patterns was 08:02. Regarding completeness, almost all teams managed to complete the project, having the balance between teams that used patterns and not used. Bugs were observed in all projects using no patterns and only a bug in projects that used patterns. Moreover, the average number of lines of code of the teams that used the patterns was 717 lines, while the average of the projects that did not use patterns was 855 lines. In relation to the quantity of classes, can be seen that the average number of classes of groups using the patterns was 23 classes. Well above the average of 07 classes, earned the group who did not use patterns.

Therefore we can measure the gain at development time using patterns was about 18.9% and the gain code lines was about 16.14%. In addition, improved software quality, reducing the amount of bugs generated. It is also possible to measure the amount of generated classes increased by more than three times when the patterns are used, this does not mean more work, since there was a reduction in the total number of lines typed. These results were expected since the literature on design patterns already says such gains in software and commercial applications.

Although the number of participants is not enough to come to definitive conclusions, the experimental data show clear evidence that the use of patterns reduces the programming effort and time required to develop a project of games, improving the quality of code. This is due to code reuse provided by the patterns, which, in addition to providing a ready solution to a recurring problem, allows the reduction in effort with the use of ready-made classes that other games can also reuse.

6. Conclusion

Despite, the potential of the adoption of design patterns in game development, this tool has been neglected by the game community. This work represents the first attempt to systematically shows how Gof patterns can be applied to game development. The diagrams show a choice of how programming is done without the pattern and the pattern, bringing feedback on the improvements that the use of patterns brought to the game design.

As future work will be organized a book describing the use of design patterns in games and a site where researchers can make their contributions, showing how a particular pattern GoF was used in game development.

References

AMPATZOGLOU, A.; CHATZIGEORGIOU, A. 2006. Evaluation of object-oriented design patterns in game development. University of Macedonia. Thessaloniki (Greece), p. 10. 2006.

AMPATZOGLOU, Apostolos; GORTZIS, Antonis; KRITIKOS, Apostolos; CHATZIASIMIDIS, Fragkiskos; ARVANITOU, Elvira M.; STAMELOS, Ioannis. 2011 *An empirical investigation on the impact of design pattern application on computer game defects.* In. XV International Academic MindTrek Conference: Envisioning Future Media Environments. Tampere, p. 8. 2011.

BJÖRK, S.; HOLOPAINEN, J. 2001. Patterns In Game Design. Ebook: The Game Design Reader, p.410 a 437. 2005.

BNDES, 2014. Relatório Final. Mapeamento da Indústria Brasileira e Global de Jogos Digitais. Fevereiro/2014. Contrato BNDES-FUSP 12.2.0431.1 Available in: <http://www.bndes.gov.br/SiteBNDES/bndes/bndes_pt/Galerias/Arquivos/conhecimento/seminario/seminario_mapeamento_industria_games042014_Relatorio_Final.pdf >. Accessed: 25 jun. 2014.

DEITEL, P. J.; DEITEL, H. M. 2005. Java Como Programar. Tradução de Edson Furmankiewicz. 6. ed. São Paulo: Pearson Prentice Hall, 2005.

GAMMA, E. et al. 2000. Padrões de Projeto. Porto Alegre: Bookman, 2000.

GESTWICKI, P. V. 2007. *Computer Games as Motivation for Design Patterns*. Ball State University. Muncie, p. 5. 2007.

GESTWICKI, P.; SUN, F.-S 2007. On Games, Patterns, and Design. In. Symposium on Science of Design. p. 17-18, 2007.

GESTWICKI, P.; SUN, F.-S. 2008. Teaching Design Patterns Through Computer Game Development. Journal on Educational Resources in Computing (JERIC), New York, 01 Março 2008.

KAAE, R. C. 2001. Using design patterns in game engines. TietoEnator Consulting A/S. Helsinki. Ebook, 2001.

LARSEN, S.; AARSETH, E. 2006. Level Design Patterns. Copenhagen: IT, 2006.

MARTÍN, M. A. G.; DÍAZ, G. J.; ARROYO, J. 2009. Teaching Design Patterns Using a Family of Games. In. 14th SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 268-272, 2009.

PERUCIA, A. S. et al. 2005. Desenvolvimento de Jogos Eletrônicos – Teoria e Prática. São Paulo: Novatec, 2005.

SILVEIRA, I. F.; SILVA, L. 2006. Aprendizagem de Padrões de Projeto em Ciência da Computação através de Jogos Digitais. In.XIV WEI - Workshop sobre Educação em Computação. São Paulo, p. 10. 2006.

TRINDADE, J. M. F.; FISCHER, L. G. 2008. Estudo e Aplicação de Padrões. Universidade Federal do Rio Grande do Sul. Rio Grande do Sul, p. 13. 2008.

WICK, M. R. 2005. Teaching Design Patterns in CS1: a Closed Laboratory Sequence based on the Game of Life. In. 36th SIGCSE Technical Symposium on Computer Science Education. Eau Claire, p. 487-491, 2005.

WONG, S. B.; NGUYEN, D. 2002. Design Patterns for Games. Rice University. Houston, p. 5. 2002.