IMPROVING SOFTWARE QUALITY THROUGH DESIGN PATTERNS: A CASE STUDY OF ADAPTIVE GAMES AND AUTO DYNAMIC DIFFICULTY

Muhammad Iftekher Chowdhury and Michael Katchabaw
Department of Computer Science
University of Western Ontario
London, Ontario,
Canada
E-mail: {mchowd2, katchab}@uwo.ca

KEYWORDS
Auto dynamic difficulty, game balancing, software design patterns.

ABSTRACT

Auto dynamic difficulty (ADD) is the technique of automatically changing the level of difficulty of a video game in real time to match player expertise. Recreating an ADD system on a game-by-game basis is both expensive and time consuming, ultimately limiting its usefulness. Thus, we leverage the benefits of software design patterns to construct an ADD framework. In this paper, we discuss a number of desirable software quality attributes that can be achieved through the usage of these design patterns, based on a case study of two video games.

INTRODUCTION

In the last 30 years, the scope of video games has expanded considerably in terms of platforms, genres and size. Unfortunately, we still struggle with keeping players engaged in a game for a long period of time. According to a recent article (Snow 2011), 90% of game players never finish a game. One of the key engagement factors for a video game is an appropriate level of difficulty, as games become frustrating when they are too hard and boring when they are too easy (Hao et al. 2010). From the point of view of skill levels, reflex speeds, hand-eye coordination, tolerance for frustration, and motivations, video game players may vary drastically (Bailey and Katchabaw 2005). These factors together make it very challenging for video game designers to set an appropriate level of difficulty in a video game. Traditional static difficulty levels (e.g., easy, medium, hard) often fail in this context as they expect the players to judge their ability themselves appropriately before playing the game and also try to classify them in broad clusters (e.g., what if easy is too easy and medium is too difficult for a particular player?).

Auto dynamic difficulty (ADD), also known as dynamic difficulty adjustment (DDA) or dynamic game balancing (DGB), refers to the technique of automatically changing the level of difficulty of a video game in real time, based on the player's ability (or, the effort s/he is currently spending) in order to provide them with an "optimal experience", also sometimes referred to as "flow". If the dynamically adjusted difficulty level of a video game appropriately matches the expertise of the current player, then it will not only attract players of varying demographics but also enable the same player to play the game repeatedly without being bored. Popular games such as "Max Payne", "Half-Life 2" and "God Hand" use the concept of auto dynamic difficulty. While others have studied ADD in games, this has been done in an ad hoc fashion in terms of software design and is therefore not reusable or applicable to other games. Recreating an ADD system on a game-by-game basis is both expensive and time consuming, ultimately limiting its usefulness. For this reason, we leverage the benefits of software design patterns (Gamma et al. 1995) to construct an ADD framework and system that is reusable, portable, flexible, and maintainable.

In (Chowdhury and Katchabaw 2012), we introduced a collection of four design patterns originally from self-adaptive system literature (Ramirez and Cheng 2010), derived in the context of enabling auto dynamic difficulty in video games. Unfortunately, to date, the literature on the usage of software design patterns in developing video games is relatively scarce. Work in this area is mostly limited to using video games as a means for teaching software design patterns in undergraduate computer science courses (e.g., Gestwicki and Sun 2008; Antonio et al. 2009). Very little, if any, motivation of using software design patterns for implementing ADD is found in the video game literature. Thus, in this paper, we discuss the improvements to overall software quality that can be achieved through the usage of these design patterns, based on empirical evidence acquired through a case study involving implementation and source code analysis of two proof-of-concept video games.

The rest of this paper is organized as follows. In the next section, we overview key literature from the area. We then describe our design patterns for enabling auto dynamic difficulty in video games, as well as our case study. Finally, in the remaining sections, we present the results from our case study and conclude the paper.

RELATED WORK

Considering the variety of contexts and the focus of related research, we divide our related work discussion into three sub-sections. First we highlight the research that explores the use of ADD in video games. Afterwards, we discuss the literature on using software design patterns in video games. Finally, we discuss the research gap and put our work in the context of this other work.

Auto Dynamic Difficulty

In recent years, ADD has received notable attention from numerous researchers. Some of this research is primarily focused on knowledge seeking, whereas other works present solutions such as frameworks and algorithms. Additionally, in some research, new solutions are presented together with empirical validations. Here, we review some of these works.

(Bailey and Katchabaw 2005) developed an experimental testbed based on Epic's Unreal engine that can be used to implement and study ADD in games. It allows development of new ADD algorithms as well. A number of mini-game gameplay scenarios were developed in the test-bed and these were used in preliminary validation experiments.

(Rani et al. 2005) suggested a method to use real time feedback, by measuring the anxiety level of the player using wearable biofeedback sensors, to modify game difficulty. They conducted an experiment on a Pong-like game to show that physiological feedback based difficulty levels were more effective than performance feedback to provide an appropriate level of challenge. Physiological signals data were collected from 15 participants each spending 6 hours in cognitive tasks (i.e., anagram and Pong tasks) and these were analyzed offline to train the system.

(Hunicke 2005) used a probabilistic model to design ADD in an experimental first person shooter (FPS) game based on the Half-life SDK. They used the game in an experiment on 20 subjects and found that ADD increased the player's performance (i.e., the mean number of deaths decreased from 6.4 to 4 in the first 15 minutes of play) and the players did not notice the adjustments.

(Orvis et al. 2008), from an experiment involving 26 participants, found that across all difficulty levels, completion of the game resulted in an improvement in performance and motivation. Prior gaming experience was found to be an important influence factor. Their findings suggested that for inexperienced gamers, the method of manipulating difficulty level would influence performance.

(Hao et al. 2010) proposed a Monte-Carlo Tree Search (MCTS) based algorithm for ADD to generate intelligence of non player characters. Because of the computational intensiveness of the approach, they also provided an alternative based on artificial neural networks (ANN) created from the MCTS. They also tested the feasibility of their approach using Pac-Man.

(Hocine and Gouaïch 2011) described an ADD approach for pointing tasks in therapeutic games. They introduced a motivation model based on job satisfaction and activation theory to adapt the task difficulty. They also conducted preliminary validation through a control experiment on eight healthy participants using a Wii balance board game.

Software Design Patterns in Video Games

In a number of works, video games have been proposed as a tool to teach software engineering in general and design patterns in particular. On the other hand, unfortunately, work focusing on how game developers can benefit from the usage of software design patterns is relatively rare. Here we discuss examples of both types of research.

(Gestwicki and Sun 2008) presented a video game based approach to teach software design patterns to computer science students. They developed an arcade style game, EEClone, which consists of six key design patterns and then used these patterns in their case study. Student participants analyzed the game to learn the usage of those patterns.

(Antonio et al. 2009) described their experience in teaching software design patterns using a number of incremental abstract strategy game design assignments. In their approach, each assignment was completed by refactoring and using design patterns on previous assignments.

(Narsoo et al. 2009) described the usage of software design patterns to implement a single player Sudoku game for the J2ME platform. They found that through the use of design patterns, new requirements could be accommodated by making changes to fewer classes than otherwise possible.

Research Gap

As we can see from above discussion, the work on ADD in video games focuses on tool building (e.g., framework (Bailey and Katchabaw 2005), algorithm (Hunicke 2005; Hao et al. 2010) etc.) and empirical studies (e.g., Rani et al. 2005; Orvis et al. 2008 etc.), but they all use an ad-hoc approach from a software design point view. On the other hand, research on using software design patterns in video games is mostly limited to using video games as a means for teaching design patterns in undergraduate computer science courses (e.g., Gestwicki and Sun 2008; Antonio et al. 2009). In contrast, much work has been done towards game design patterns, such as the foundational work of (Björk and Holopainen 2004) and many others, but the focus there is game design and not software design, which is a subtle, yet important distinction. Thus, in this paper, we discuss the software quality attributes that can be achieved through the usage of software design patterns in the context of ADD, based on an empirical study.

DESIGN PATTERNS

In this section, we briefly discuss the four software design patterns for enabling ADD in video games. For further details, the reader is encouraged to refer to (Chowdhury and Katchabaw 2012) for elaborated discussion and examples.

Sensor Factory

The sensor factory pattern is used to provide a systematic way of collecting data while satisfying resource constraints, and provide those data to the rest of the ADD system. *Sensor* (please see Figure 1) is an abstract class that encapsulates the periodical collection and notification mechanism. A concrete sensor realizes the Sensor and defines specific data collection and calculation. The *SensorFactory* class uses the "factory method" pattern to

provide a unified way of creating any sensors. It takes the *sensorName* and the *object* to be monitored as input and creates the sensor. Before creating a sensor, the *SensorFactory* checks in the *Registry* data structure to see whether the sensor has already been created. If created, the *SensorFactory* just returns that sensor instead of creating a new one. Otherwise, it verifies with a *ResourceManager* whether a new sensor can be created without violating any resource constraints.

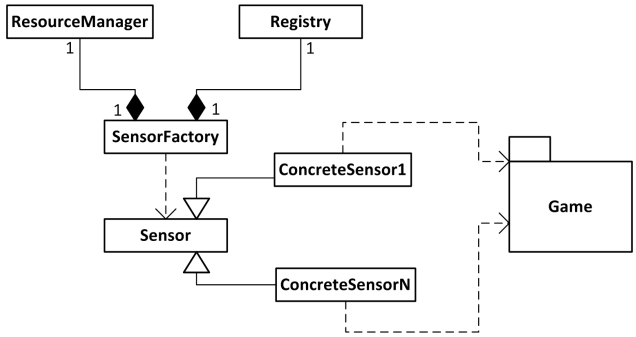

Figure 1: Sensor Factory Design Pattern

Adaptation Detector

With the help of the sensor factory pattern, the *AdaptationDetector* (please see Figure 2) deploys a number of sensors in the game and attaches observers to each sensor. *Observer* encapsulates the data collected from sensor, the unit of data (i.e., the degree of precision necessary for each particular type of sensor data), and whether the data is up-to-date or not. *AdaptationDetector* periodically compares the updated values found from *Observer*s with specific *Threshold* values with the help of the *ThresholdAnalyzer*. Each *Threshold* contains one or more boundary values as well as the type of the boundary (e.g., less than, greater than, not equal to, etc.). Once the *ThresholdAnalyzer* indicates a situation when adaptation might be needed, the *AdaptationDetector* creates a *Trigger* with the information that the rest of the ADD process needs.

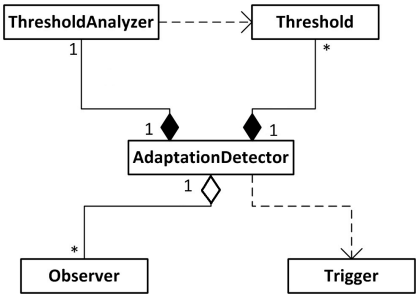

Figure 2: Adaptation Detector Design Pattern

Case Based Reasoning

While the adaptation detector determines the situation when a difficulty adjustment is required by creating a *Trigger*, case based reasoning (please see Figure 3) formulates the *Decision* that contains the adjustment plan. The *InferenceEngine* has two data structures: the *TriggerPool* and the *FixedRules*. *FixedRules* contains a number of *Rule*s. Each *Rule* is a combination of a *Trigger* and a *Decision*. The *Trigger*s created by the adaptation detector will be stored in the *TriggerPool*. To address the triggers in the sequence they were raised in, the *TriggerPool* should be a FIFO data structure. The *FixedRules* data structure should support search functionality so that when the *InferenceEngine* takes a *Trigger* from the *TriggerPool*, it can scan through the *Rule*s held by *FixedRules* and find a *Decision* that appropriately responds to the *Trigger*.

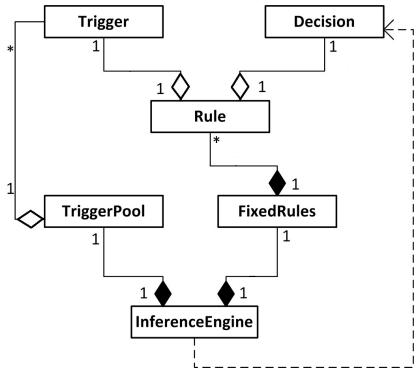

Figure 3: Case Based Reasoning Design Pattern

Game Reconfiguration

Once the ADD system detects that a difficulty adjustment is necessary, and decides what and how to adjust the various game components, it is the task of the game reconfiguration pattern to facilitate smooth execution of the decision. The *AdaptationDriver* receives a *Decision* selected by the *InferenceEngine* (please see case based reasoning in previous section) and executes it with the help of the *Driver*. *Driver* implements the algorithm to make any attribute change in an object that implements the *State* interface (i.e., that the object can be in ACTIVE, BEING_ACTIVE, BEING_INACTIVE or INACTIVE states, and outside objects can request state changes). As the name suggests, in the active state, the object shows its usual behavior whereas in the inactive state, the object stops its regular tasks and is open to changes. The *Driver* takes the object to be reconfigured (default object used if not specified), the attribute path (i.e., the attribute that needs to be changed, specified according to a predefined protocol such as object oriented dot notation) and the changed attribute value as inputs. The *Driver* requests the object that needs to be reconfigured to be inactive and waits for the inactivation. When the object becomes inactive, it reconfigures the object as specified. After that, it requests the object to be active and informs the *AdaptationDriver* when the object becomes active. The *GameState* maintains a *RequestBuffer* data structure to temporarily store the inputs received during the inactive state of the game. (If the reconfiguration is done efficiently, however, it should be completed within a single tick of the main game loop, and this buffering should be largely unnecessary.) The *GameState* overrides Game's event handling methods and game loop to implement the *State* interface.

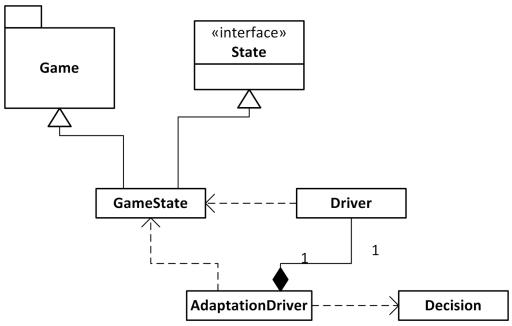

Figure 4: Game Reconfiguration Design Pattern

Integration of ADD Design Patterns

In this Section, we briefly re-discuss how the four design patterns discussed in previous sub-sections work together to create a complete ADD system (please see Figure 5). The sensor factory pattern uses *Sensor*s to collect data from the game so that the player's perceived level of difficulty can be measured. The adaptation detector pattern observes *Sensor* data using *Observer*s. When the adaptation detector finds situations where difficulty needs to be adjusted, it creates *Trigger*s with appropriate additional information. Case based reasoning gets notified about required adjustments by means of *Trigger*s. It finds appropriate *Decision*s associated with the *Trigger*s and passes them to the adaptation driver. The adaptation driver applies the changes specified by each *Decision* to the game, to adjust the difficulty of the game appropriately, with the help of the *Driver*. The adaptation driver also makes sure that the change process is transparent to the player. In this way, all four design patterns work together to create a complete ADD system for a particular game.

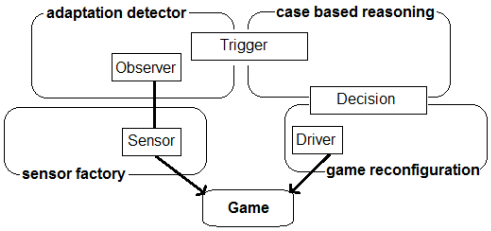

Figure 5: Four Design Patterns Working Together in a Game

CASE STUDY

In this section, we describe the case study used to assess software quality improvements achieved through our design patterns for ADD. We begin with a brief description of each of the two games that were used in the case study. We then describe the case study methods and the quality metrics that were collected from the case study.

Case Study Games

We used two arcade style single player games developed in Java for the case study. The first game is a variant of Pac-Man and will be referred to as Game-P from here onwards. Game-P was developed for the purposes of this research. The level structure and gameplay of the second game is similar to the popular Super Mario game series and will be referred to as Game-S from here onwards. Game-S is a slightly modified version of a platform game described in (Brackeen et al. 2004). In sub-sections below, we briefly describe the game logic and ADD logic of these two games.

Game-P

In this game, the player controls Pac-Man in a maze (please see Figure 6). There are pellets, power pellets, and 4 ghosts in the maze. Pac-Man has 6 lives. Usually, ghosts are in a predator mode and touching them will cause the loss of one of Pac-Man's lives. When Pac-Man eats a power-pellet, it becomes the predator for a certain amount of time. When Pac-Man is in this predator mode and eats a ghost, the ghost will go back to the center of the maze and will stay there for a certain amount of time. Eating pellets gives points to Pac-Man. The player tries to eat all the pellets in the maze without losing all of Pac-Man's lives. The player is motivated to chase the ghosts while in predator mode, as that will help them by removing the ghosts from the maze for a time, allowing Pac-Man to eat pellets more freely. Ghosts only change direction when they reach intersections in the maze, while Pac-Man can change direction at any time. A ghost's vision is limited to a certain number of cells in the maze. Ghosts chase the player if they can see them. If the ghosts do not see Pac-Man, they try to roam the cells with pellets, as Pac-Man needs to eventually visit those areas to collect the pellets. If the ghosts do not see either Pac-Man or pellets, they move in a random fashion.

Figure 6: Screen Captured from Game-P

Usually, a Pac-Man game is multi-level, but our implementation (i.e., Game-P) has only one level. The maximum possible score is 300 in our case, so the player will try to achieve the score of 300 without losing all of Pac-Man's lives. Our assumption is that if the player loses all lives (i.e., 6) before finishing the game, then the average score per life (i.e., total score / number of lives lost to achieve the score) would be less than 50 and the game would seem overly difficult to them. On the other hand, if the player finishes the game losing half of the lives or less, then the average score would be greater than or equal to 100, and the game would seem too easy to them. Thus, in this case, the ADD system monitors the average-score-per-life and changes game difficulty accordingly. It starts increasing the game difficulty when the monitored value is more than 50 and the game become most difficult when the value is more than 100. (Corresponding logic decreases the game difficulty when the average-score-per-life is less than 50.) The attributes of ghost speed, ghost vision length, duration of Pac-Man's predator mode, and the amount of time that a ghost stays in the centre of the maze after being eaten by Pac-Man in predator mode are increased or

decreased to change the game difficulty. Each of these attributes has lower and upper limits, so that the game includes the option of someone playing extremely well or extremely poorly.

Game-S

In this game, the player controls the player character in a platform world (please see Figure 7). There are three levels, each having different tile based maps. There are power ups and non-player characters (i.e., enemies) in each level. There are three different types of power ups: basic power ups, bonus power ups, and a goal power up. Basic power ups and bonus power ups give certain points to the player. In each level there is one goal power up that can be found at the end of the level. The goal power up takes the player from one level to another. There are two different types of non-player characters: ants and flies. Ants and flies move in one direction and change direction when blocked by the platforms. The player character can run on and jump from platforms. When the player character jumps on (i.e., collides from above) non-player characters, the non-player character dies. If the player character collides with a non-player character in any other direction, then the player character dies instead. The player character has 6 lives. When the player character dies, it loses one life and the game restarts from the beginning of that level. The player character and ants are affected by gravity; flies are only affected by gravity when they die.

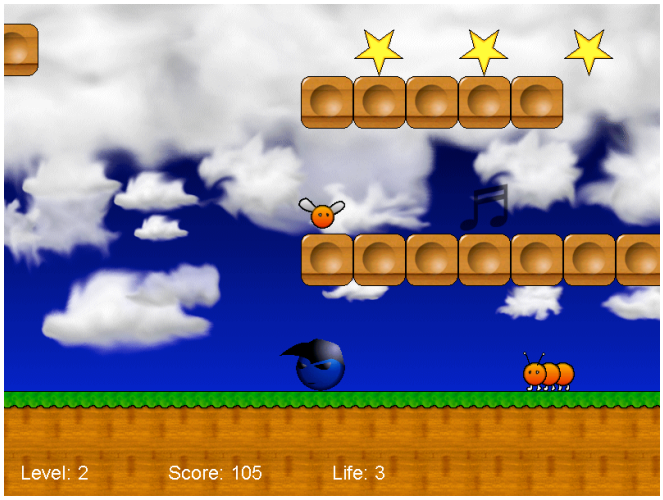

Figure 7: Screen Captured from Game-S

In this game, three map variants were created for each level. For a particular level, the same objects were placed in the map, but positioned slightly differently. One map variant was the default version and other two were easier and harder versions of the default map. The ADD system monitors score-per-level and life-lost-per-level, and adapts difficulty accordingly. One possible adaptation is the modification of the speed of the non-player characters and takes place during the game. Another adaptation is a change in level structure (i.e., loading a different version of the map) and takes place when the player character goes to the next level or in the next loading of the same level (i.e., when the player character dies). The modifications are minor and assumed to be transparent to the player, but altogether alter the game difficulty. Apart from this, each level is more difficult and lengthier than the previous level, but has more points to give the player a sense of progress and accomplishment. Similar to Game-P, modifications in Game-S have lower and upper limits, so that the game includes the option of someone playing extremely well or extremely poorly.

Case Study Method

Here we briefly discuss the steps that were taken during the course of the case study. Firstly, Game-P was developed without our pattern-based ADD system. Our ADD system was then developed and integrated with Game-P. The source code for the ADD system was then refactored within the scope of the design patterns. We manually tested Game-P separately and with the ADD system. A player simulation (i.e., a simple artificial intelligence playing the game itself using heuristic functions) was also created to test the game. Game-S was then chosen for study as it used the same Java platform as Game-P, but substantially differed in terms of gameplay, and was freely available and well documented in a book (Brackeen et al. 2004). Two default maps accompanied Game-S originally. We created one more default map ourselves, as well as two different variants of each map, as discussed earlier. There was no scoring mechanism in Game-S as originally written, so we developed scoring logic ourselves. After this, we took the source code of our ADD system used with Game-P and extended its abstract base classes (*Sensor*, and so on) to adapt the system for Game-S. We manually tested Game-S separately and with the ADD system. We then analyzed and compared the source code of the ADD systems of Game-P and Game-S to assess software quality according to a few key software metrics, as discussed in the next sub-section.

Analysis Tool and Metric

During the development of the ADD system for Game-P, we realized that much of its source code would be reusable across various games. During the extension of the ADD system for Game-S, we did not need to make modifications to many of the classes from the system for Game-P. To assess this quantitatively, we selected a metric and a tool. As a metric, we used Source Lines of Code (SLOC), as it is a widely accepted software metric and helps in estimating the development effort of a software product. For a tool, we used Unified Code Count (UCC) developed by the University of Southern California Center for Systems and Software Engineering. Features of UCC include both counting SLOC and comparing two versions of source code. UCC counts both the logical and physical SLOC. As seen from the two examples in Table 1, since logical SLOC disregards code formatting, it is more representative of the size of the software, and so we used logical SLOC as our metric in this study.

Table 1: Difference Between Physical and Logical SLOC

Example	Physical vs. Logical SLOC
if(a = = 0) foo();	Physical SLOC = 1,Logical SLOC = 2

if(a = = 0)
{
foo();
}
	Physical SLOC = 4,Logical SLOC = 2

SOFTWARE QUALITY ASPECTS

In this section, we describe how different desirable software quality attributes can be achieved through using the design patterns described earlier in this paper. For this discussion, we refer to case study results and observations.

Reusability

Reusability refers to the degree to which existing applications can be reused in new applications. Reusability of source code reduces implementation time and increases the probability that prior testing has eliminated defects.

In Table 2, we show our reusability analysis of the source code of the ADD systems of Game-P and Game-S. In the first column, we show the class name or pattern name. In the next four columns we show information related to Game-P. In the first Game-P column we show the number of classes in each category (i.e., specified in column 1). In the second column we show the corresponding total logical SLOC in Game-P. In the third column we show the reusable Logical SLOC (i.e., code that remained unchanged in Game-S) and the associated percentage. In the fourth column we show the game specific Logical SLOC (i.e., specific to Game-P and cannot be reused) and the associated percentage. The remaining columns report similar data, this time from the perspective of Game-S. For clarity, we combined 100% reusable classes within a particular pattern. After all the rows of a particular pattern we show the summary of that pattern. The last row of the table is the summary across all the patterns.

We can see from Table 2 that *SensorFactory*, *Sensor*, *Registry* and *ResourceManager* classes in the sensor factory design pattern are completely reusable. Similarly, classes required to implement the *Observer*, *Trigger*, *Threshold* and *ThresholdAnalyzer* in the adaptation detector pattern are completely reusable. Three classes (i.e., *Rule*, *FixedRules* and *Decision*) in the case based reasoning pattern, and three classes (i.e., *Driver*, *AdaptationDriver* and *State*) in the game reconfiguration pattern are also completely reusable. Furthermore, the classes required to implement *AdaptationDetector*, *InferenceEngine* and *GameState* are partially reusable. Only the concrete sensors (6 classes in Game-P and 3 classes in Game-S) and the concrete decisions (2 classes in Game-P and 5 classes in Game-S) are specific to the game and not reusable.

As we can see from the last row in Table 2, the ADD system in Game-P contains 27 classes comprised of 774 logical SLOC. Similarly, the ADD system in Game-S contains 753 logical SLOC in 27 classes. Between these two systems, 600 logical SLOC (77.52% in Game-P; 79.68% in Game-S) are exactly the same and thus are considered reusable. Only 174 (22.48%) logical SLOC in Game-P and 153 (20.32%) logical SLOC in Game-S are specific to the games. Overall, more than three fourths (75%) of the logical SLOC required to implement the ADD systems are considered reusable.

Integrability

Integrability refers to the ability to make the separately developed components of the system work correctly together. As we can see in Figure 5, the integration points among the design patterns and with the game are clearly defined. *Observer*s, *Trigger*s and *Decision*s are the integration points between the four design patterns. *Sensor*s and *Driver*s are the integration points between a game and the ADD system. *Sensor*s function as accessors to the game whereas *Driver*s function as mutators to the game. Because of these clearly defined integration points, the four design patterns can be integrated with each other and a game easily. One of the games in the case study (Game-S) was already developed without any prior consideration of these design patterns or even any ADD system. Regardless, we easily managed to extend and add our ADD system to that game using our design patterns, which demonstrates the integrability of the patterns (and also Game-S).

Table 2: Reusability Analysis of the Source Code of ADD Systems in Game-P and Game-S

Class/ Pattern Name	Game-P				Game-S			
	# of Classes	Logical SLOC			# of Classes	Logical SLOC		
		Total	Reusable(%)	Specific(%)		Total	Reusable(%)	Specific(%)
SensorFactory, Sensor, Registry, Resource Manager	4	218	218(100)	0(0)	4	218	218(100)	0(0)
ConcreteSensors	6	68	0(0)	68(100)	3	44	0 (0)	44(100)
Sensor Factory	**10**	**286**	**218(76.22)**	**68(23.78)**	**7**	**262**	**218 (83.21)**	**44(16.79)**
Observer, Trigger, Threshold, ThresholdAnalyzer	5	97	97(100)	0(0)	5	97	97(100)	0(0)
AdaptationDetector	1	68	21(30.88)	47(69.12)	1	65	21(32.31)	44(67.69)
Adaptation Detector	**6**	**165**	**118(71.52)**	**47(28.48)**	**6**	**162**	**118(72.84)**	**44(27.16)**
Rule, Decision, FixedRules	3	75	75(100)	0(0)	3	75	75(100)	0(0)
InferenceEngine	2	50	46(92)	4(8)	2	51	46(90.20)	5(9.80)
ConcreteDecisions	2	29	0(0)	29(100)	5	30	0(0)	30(100)
Case-based Reasoning	**7**	**154**	**121(78.57)**	**33(21.43)**	**10**	**156**	**121(77.56)**	**35(22.44)**
Driver, AdaptationDriver, State	3	99	99(100)	0(0)	3	99	99(100)	0(0)
GameState	1	70	44(62.86)	26(37.14)	1	74	44(59.46)	30(40.54)
Game Reconfiguration	**4**	**169**	**143(84.62)**	**26(15.38)**	**4**	**173**	**143(82.66)**	**30(17.34)**
Grand Total	**27**	**774**	**600(77.52)**	**174(22.48)**	**27**	**753**	**600(79.68)**	**153(20.32)**

Portability

Portability is the ability of a system to run under different computing environments. A framework- or middleware-based approach for creating a self-adaptive system (such as ADD in video games) is usually specific to a particular programming language and or platform, whereas a design pattern-based approach is highly portable across different platforms and programming languages (Ramirez and Cheng 2010). These design patterns were derived from the self-adaptive system literature in the context of ADD in video games. This indicates the portability of these design patterns across domains. Also, in our case study, we managed to port them (as a solution) from one game to another within the platform (Java). This indicates portability across systems on the same platform. In the future, we plan to examine the portability of these design patterns across platforms as well.

Maintainability

Maintainability refers to the ease of the future maintenance of the system. As discussed earlier, different parts of the design patterns have specific concerns (e.g., *Sensor*s will collect data, *Driver*s will make changes to the game, etc.), and so the resulting source code will have high traceability and maintainability. Furthermore, as the use of these design patterns provides source code reusability (please see Table 2), this will increase the probability that prior testing has eliminated defects when being used in a new game.

CONCLUDING REMARKS

Design patterns are a formal approach of describing reusable solutions for a design problem. To date, the literature on the usage of software design patterns in video games is relatively scarce. Little or no motivation of using software design patterns for implementing ADD is found in video game literature. Thus, in this paper, we presented a case study involving implementation and source code analysis of two proof-of-concept video games. We discussed how desirable software quality attributes such as reusability, integrability, portability, and maintainability can be achieved through the usage of these design patterns. Our case study results and methods have implications on both research and practice, giving practitioners motivation to use these design patterns for implementing ADD systems. Our analysis technique (i.e., a source code analysis to compare games) can be used in further research. Even though our context of discussion was ADD, these patterns can be used in any situation where a game needs to be adaptive and reconfigures itself based on monitoring.

REFERENCES

Antonio, M.; Jiménez-Díaz, G.; and Arroyo, J. 2009. "Teaching Design Patterns Using a Family of Games". In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science* (Paris, France, Jul. 6-9). 268-272.

Bailey, C.; and Katchabaw, M. 2005. "An Experimental Testbed to Enable Auto-Dynamic Difficulty in Modern Video Games". In *Proceedings of the 2005 North American Game-On Conference* (Montreal, Canada, Aug. 22-23). 18-22.

Björk, S., Holopainen, J. 2004. *Patterns in Game Design*. Charles River Media, Inc. Massachusetts, USA.

Brackeen, D.; Barker, B.; and Vanhelsuwé, L. 2004. *Developing Games in Java*. New Riders.

Chowdhury, M. I.; and Katchabaw, M. 2012. "Software Design Patterns for Enabling Auto Dynamic Difficulty in Video Games". In *Proceedings of the 17th Intl. Conf. on Computer Games* (Louisville, Kentucky, USA, Jul. 30 - Aug. 1). 76 - 80.

Gamma, E.; Helm, R.; Johnson, R.; and Vissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison - Wesley.

Gestwicki, P.; Sun, F. 2008. "Teaching Design Patterns Through Computer Game Development". *Journal on Educational Resource in Computing* 8, No. 1 (Mar), 1 -22.

Hao, Y.; He, S.; Wang, J.; Liu, X.; Yang, J.; and Huang, W. 2010. "Dynamic Difficulty Adjustment of Game AI by MCTS for the Game Pac-Man". In *Proceedings of the Sixth International Conference on Natural Computation* (Yantai, China, Aug. 10-12). 3918-3922.

Hocine, N.; and Gouaïch, A. 2011. "Therapeutic Games' Difficulty Adaptation: An Approach Based on Player's Ability and Motivation". In *Proceedings of the 16th Intl.Conf. on Computer Games* (Louisville, Kentucky, USA, Jul. 27-30). 257 - 261.

Hunicke, R. 2005. "The Case for Dynamic Difficulty Adjustment in Games". In *Proceedings of the 2005 ACM SIGCHI International Conf. on Advances in Computer Entertainment Technology* (Valencia, Spain, Jun. 15-17). 429-433.

Narsoo, J.; Sunhaloo, M. S.; and Thomas, R. 2009. "The Application of Design Patterns to Develop Games for Mobile Devices using Java 2 Micro Edition". *Journal of Object Technology* 8, No. 5 (Jul.,Aug.), 153 - 175.

Orvis, K. A.; Horn, D. B.; and Belanich, J. 2008. "The Roles of Task Difficulty and Prior Videogame Experience on Performance and Motivation in Instructional Videogames". *Computers in Human Behavior* 24, No.5 (Sep), 2415 -2433.

Ramirez, A. J.; and Cheng, B. H. C. 2010. "Design Patterns for Developing Dynamically Adaptive Systems". In *Proceeding of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems* (Cape Town, South Africa, May. 3-4). 49 - 58.

Rani, P.; Sarkar, N.; and Liu, C. 2005. "Maintaining Optimal Challenge in Computer Games Through Real-time Physiological Feedback". In *Proceedings of the 11th Int.l Conf. on Human-Computer Interaction* (Las Vegas, USA, July. 22-27). 184-192.

Snow, B. 2011. "Why Most People Don't Finish Video Games" Online publication in CNN, August 17, 2011. Retrieved from: http://www.cnn.com/2011/TECH/gaming.gadgets/08/17/finishing.videogames.snow/. Last accessed: July 04, 2012.

BIOGRAPHY

MUHAMMAD IFTEKHER CHOWDHURY is a PhD candidate working in the area of game design at the University of Western Ontario. He completed his Masters in software engineering from the same university.

MICHAEL KATCHABAW is an Associate Professor at the University of Western Ontario. His research interests include game design and development. He co-founded the Digital Recreation, Entertainment, Art, and Media (DREAM) research group at Western.