Software Design Patterns for Enabling Auto Dynamic Difficulty in Video Games

Muhammad Iftekher Chowdhury, Michael Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada

Abstract—**Auto dynamic difficulty is the technique of automatically changing the level of difficulty of a video game in real time to match player expertise. In this paper, we describe a collection of software design patterns for enabling auto dynamic difficulty in video games. The benefits of a design pattern approach include more reusability and lower risk compared to traditional ad hoc approaches. We implemented these design patterns as a proof-of-concept prototype system using Pac-Man as a test-bed.**

Keywords-auto dynamic difficulty; game balancing; software design pattern

I. 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 [1], 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 [2]. From the point of view of skill levels, reflex speeds, hand-eye coordination, tolerance for frustration, and motivations, video game players may vary drastically [3]. 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 the "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, in our current work, we leverage the benefits of software design patterns [4] to construct an ADD framework and system that is reusable, portable, flexible, and maintainable. In this paper, we describe a collection of four design patterns from self-adaptive system literature, derived in the context of enabling auto dynamic difficulty in video games, and discuss their use in a proof-of-concept prototype system. We used a variant of Pac-Man as a test-bed for our study.

The rest of this paper is organized as follows. In Section II, we discuss the literature reviewed. In Section III, we describe the design patterns for enabling auto dynamic difficulty in video games. In Section IV, we describe the proof-of-concept implementation. In Sections V and VI, we highlight the benefit of a design pattern approach and conclude the paper.

II. RELATED WORK

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 [3] developed an experimental test-bed 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 game-play scenarios were developed in the test-bed and these were used in preliminary validation experiments.

Rani et al. [5] 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 [6] 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. [7], 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. [8] 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 [9] 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.

As we can see from above discussion, the work on ADD in video games focuses on tool building (e.g., framework [3], algorithm ([6], [8]) etc.) and empirical studies (e.g., [5], [7] etc.), but they all use an ad-hoc approach from a software design point of view. Thus, in this paper, we discuss the usage of software design patterns derived from self-adaptive system literature for enabling ADD in video games.

III. DESIGN PATTERNS

Ramirez and Cheng [10] presented 12 design patterns, developed through the generalization of design solutions found in the self-adaptive system literature, that would assist in enabling adaptability in a software system. We found four of these design patterns to be often necessary and sufficient for enabling ADD in video games. In this section, we derive these design patterns in the context of ADD in video games. We used the same classification scheme of adaptive design patterns as Ramirez and Cheng (i.e., monitoring, decision making, and reconfiguration patterns). Bailey and Katchabaw also used synonymous component names for their framework [3]. In Sections A, B, and C, we discuss monitoring, decision making, and reconfiguration patterns respectively. In Section D, we discuss how these design patterns work together.

A. Monitoring Pattern

The main purpose of ADD is to provide more enjoyment to a broader range of player types and skill levels. Even though it seems that there should be a direct mapping from a player's achievements to their enjoyment, the actual relationship is far more complicated. For example, high achievement with minimum effort can be boring for a hardcore player whereas low achievement with high effort can be frustrating for a novice player. Thus, before we dynamically adjust the difficulty level of a game, we need to know the player's perceived level of difficulty which requires collecting data from the game at runtime. The monitoring 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. Examples of data to be collected include the player's score, player's life level, time spent on activities, inventory, number of enemies killed, and so on.

Sensor factory: Sensors are objects that periodically read data from the game[1] and notify the rest of the ADD system. *Sensor* (please see Figure 1) is an abstract class which encapsulates the periodical collection and notification mechanism. It has the abstract method *refreshValue()* which child classes need to define. A concrete sensor realizes the *Sensor* and defines data collection and calculation inside the *refreshValue()* method. An example of a concrete sensor can be *AverageScorePerLifeSensor*, which reads score and number of life attributes from the game and divides the score by the number of lives. 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. It is good practice that the *object* will provide an appropriate interface so that it can be queried by the *ConcreteSensor* for the required attribute. If for some reason the *object* does not provide the required interface, then reflection can be used to bypass the access modifier.

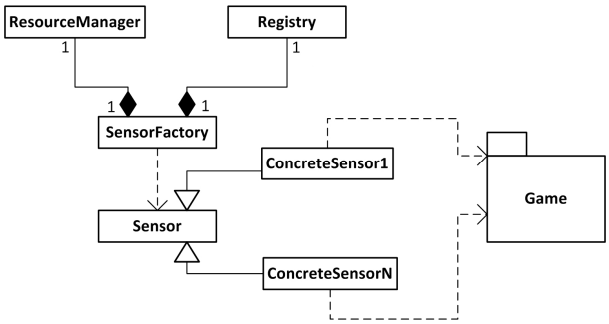

Figure 1. *Sensor factory* design pattern

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. Usually, the underlying platform and/or development environment provides wrappers for resource monitoring. For example, the *java.lang.Runtime* class and *java.lang.management* package provide such functionality.

B. Decision Making Patterns

After collecting raw data using the monitoring pattern (i.e., sensor factory), the ADD system must interpret what that information means in the context of a particular game and which game elements need to be adjusted to what degree to

[1] Please note that, with the advancements of HCI in games, the scope of sensors are no longer limited to the game world. Real world data collected from input devices such as Xbox's Kinect, Wii's controller, Playstation's Move, etc. might be useful to monitor for ADD. Research (e.g., [5]) also suggests biological feedback can be included in this context.

provide the player with an appropriate level of difficulty. Two decision making patterns: adaptation detector and case based reasoning are discussed below, encapsulating the tasks of "when to adjust the game" and "what to adjust in the game and how to adjust?" respectively.

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[2] to each sensor. *Observer* encapsulates the data collected from sensor, the unit of data, and whether the data is up-to-date or not. The unit of data represents the degree of precision necessary for each particular type of sensor data. For example, in a particular game, every tenth change in the player's inventory might be worth noticing, compared to changes in the player's remaining number of lives, which should be noted on each change. *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 might need. *Trigger* also holds book-keeping attributes such as the trigger creation time and so on. For example, if the average score per life is less than a particular threshold, then it might indicate that an adaptation is necessary. Now to give a bigger picture, the *Trigger* may include contextual information, such as the number of enemies left, their average speed, etc. *AdaptationDetector* needs to make sure that it does not repeatedly create the same *Trigger*.

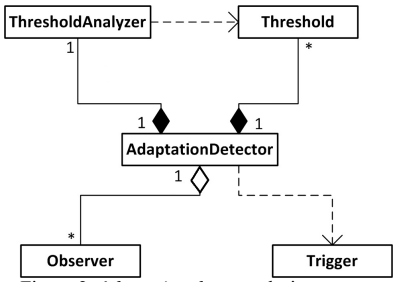

Figure 2. *Adaptation detector* design pattern

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. As the name of the pattern suggests, this pattern is best suited to games where the difficulty adjustment logic can be defined as a finite number of cases.

The *InferenceEngine* has two data structures: the *TriggerPool* and the *FixedRules*. *FixedRules* contains a number of *Rule*s[3]. 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 *Rules* held by *FixedRules* and find a *Decision* that appropriately responds to the *Trigger*. Please note that many programming languages will have built in implementation for typical instantiation of the *TriggerPool* and the *FixedRules*. A *Trigger* should provide the method (e.g., overriding the *equalsTo()* method in Java) to compare it with another one so that the *InferenceEngine* can find and take the appropriate *Decision*. Optionally, a learner component (not shown in Figure 3) can be attached to the inference engine, which can learn new rules based on monitoring the sequence and effectiveness of different trigger-decision executions on the game.

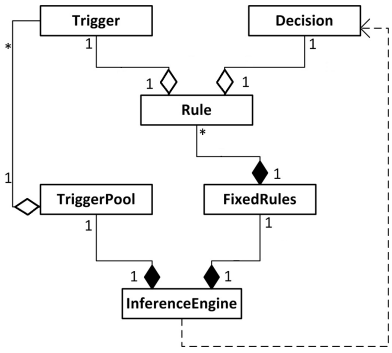

Figure 3. *Case based reasoning* design pattern

C. *Reconfiguration Pattern*

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 reconfiguration pattern to facilitate smooth execution of the decision. This task is non-trivial because the game difficulty needs to be adjusted while the player is progressing through the game. If the adjustment is drastic, it will disturb the player's immersion. Also, there is the risk of leaving the game in an inconsistent state. Here we discuss the game reconfiguration pattern, derived from the server reconfiguration pattern [10] for a client-server model, as a systematic approach to reconfigure the game. The reason we choose this pattern is because a video game often closely resembles a client-server model in which a server continuously checks in a loop for requests from clients and responds to the requests when they arrive. Similarly, in a video game, the game logic continuously checks in the game loop for inputs from input devices (such as the keyboard, mouse, gamepad, sensors, etc.) and behaves according to those inputs.

Game reconfiguration: The server reconfiguration pattern described in [10] assumes that the object that needs to be configured will implement a specific interface for reconfiguration. With the help of the adapter design pattern, this assumption can be eliminated from the game reconfiguration design pattern (as we show in Figure 4 and discuss below). The *AdaptationDriver* receives a *Decision* selected by the *InferenceEngine* (please see case based

[2] In an observer design pattern, the subject (i.e., sensors in this case) maintains a list of observers and notifies them of changes. Many programming languages provide a built in observer implementation mechanism.

[3] Please note that, the *Rule*s are very much specific to the game and the success of the ADD system highly depends on determining and using appropriate *Rule*s.

reasoning in Section III B) 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[4]) 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. The *GameState* overrides Game's event handling methods and game-loop to implement the *State* interface. It is important to note that in a reasonable implementation, all the state changes and reconfigurations can be done in less time than the game loop's sleeping period after each execution and, consequently, these changes are not noticeable to the player.

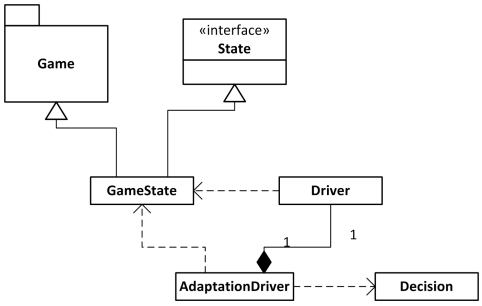

Figure 4. *Game reconfiguration* design pattern

D. ADD Design Patterns

In this Section, we briefly re-discuss how the four design patterns discussed in previous sections work together to create a complete ADD system (please see Figure 5).

The sensor factory pattern uses Sensors 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 Observers. When the adaptation detector finds situations where difficulty needs to be adjusted, it creates Triggers with appropriate additional information. Case based reasoning gets notified about required adjustments by means of Triggers. It finds appropriate Decisions associated with the Triggers 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.

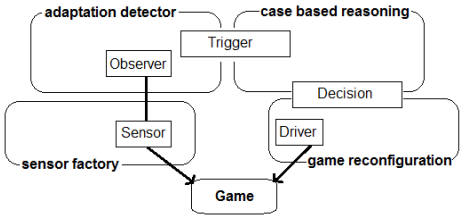

Figure 5. ADD design patterns

IV. PROOF OF CONCEPT

We implemented a proof-of-concept prototype ADD system in Java using the design patterns discussed in Section III and a variant of Pac-Man as a test-bed (please see Figure 6). We choose Pac-Man for our case study, as it is both a highly influential video game and a game used widely among researchers as a test-bed in their work.

Figure 6. Screen captured from the developed Pac-Man game[a]

a. The player controls Pac-Man in a maze. 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 benefit them by keeping the ghosts away 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.

Usually, a Pac-Man game is multi-level, but our implementation 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

[4] Example can be: object oriented dot notation like, attribute1.sub_attribute2[sub_attribute_index].sub_sub_attribute5.

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 ghost's speed, the ghost's 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.

V. DISCUSSION

In this section, we discuss the benefits of using a design pattern approach for implementing ADD in video games:

Reusable solution: 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 reusable across different platforms and programming languages [10].

Reusable source code: Generally, it is expected that reusable source code can be created for reusable solutions. Our preliminarily plans on how to reuse the ADD source code of the Pac-Man game (discussed in Section IV) indicates that a design pattern approach for implementing ADD in video games will result in highly reusable source code. We are also planning on some further studies with other games to confirm this idea.

Lower risk: As these design patterns have been developed based on generalizations of other researchers' work in the area of self-adaptive systems [10], this approach is less risky compared to an ad-hoc approach, and more likely to be effective in implementing ADD.

Separation of concerns: As different parts of the design patterns have specific concerns (e.g., Sensors will collect data, Driver will make changes to the game, etc.) the resulting source code will have high traceability and maintainability. So, creating test cases also becomes easier compared to a more ad-hoc approach.

Parallelizable: Since, in this approach, the game logic and ADD logic are clearly separate entities, they can be implemented and evolved in parallel. Also, it is possible to implement ADD logic on top of pre-existing games.

Defined process: Since the high level structure of the solution is already known, it is possible to create a step-by-step method for developing ADD (e.g., identifying the attributes to be monitored, identify attributes that controls the difficulty, etc.). Furthermore, developers can focus more on game play design and ADD logic design (please see Section IV for example) rather than implementation details.

VI. CONCLUSIONS AND FUTURE WORK

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. Thus, in this paper, we introduced four design patterns from the self-adaptive system literature derived in the context of enabling auto dynamic difficulty (ADD) in video games. We also described a proof-of-concept implementation as a means for validation of those design patterns. Even though our context of discussion was ADD, these patterns can be used in any situation where the game needs to be adaptive and reconfigures itself based on monitoring. In the future, we want to conduct case studies to evaluate the applicability of these design patterns across different genres and platforms. Our future plans also include developing a set of reusable components implementing these design patterns.

REFERENCES

[1] B. Snow, "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.video games.snow/. Last accessed: July 04, 2012.

[2] Y. Hao, S. He, J. Wang, X. Liu, J. Yang, and W. Huang, "Dynamic difficulty adjustment of game AI by MCTS for the game Pac-Man," In Proc. of 6th Int. Conf. on Natural Computation, 2010, pp. 3918-3922.

[3] C. Bailey, and M. Katchabaw, "An experimental test bed to enable auto-dynamic difficulty in modern video games," In Proc. of the 2005 North American Game-On Conf., 2005, pp. 18-22.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vissides, "Design patterns: elements of reusable object-oriented software," Addison - Wesley, 1995.

[5] P. Rani, N. Sarkar, and C. Liu, "Maintaining optimal challenge in computer games through real-time physiological feedback," In Proc. of 11th Int. Conf. on Human-Computer Interaction, 2005, pp. 184-192.

[6] R. Hunicke, "The case for dynamic difficulty adjustment in games," In Proc. of 2005 ACM SIGCHI Int. Conf. on Advances in computer entertainment technology, 2005, pp. 429-433.

[7] K. A. Orvis, D. B. Horn, and J. Belanich, "The roles of task difficulty and prior videogame experience on performance and motivation in instructional videogames," Computers in Human Behavior, vol. 24(5), pp. 2415-2433, September 2008.

[8] Y. Hao, S. He, J. Wang, X. Liu, J. Yang, and W. Huang, "Dynamic difficulty adjustment of game AI by MCTS for the game Pac-Man," In Proc. of 6th Int. Conf. on Natural Computation, 2010, pp. 3918-3922.

[9] N. Hocine, and A. Gouaïch, " Therapeutic games' difficulty adaptation: An approach based on player's ability and motivation," In Proc. of 16th Int. Conf. on Computer Games (CGAMES), 2011, pp. 257 - 261.

[10] A. J. Ramirez, and B. H. Cheng, B., "Design patterns for developing dynamically adaptive systems," In Proc. of 2010 ICSE Workshop on Software Eng. for Adaptive and Self-Managing Syst., 2010, pp. 49 - 58.

BIOGRAPHY

Muhammad Iftekher Chowdhury is a PhD candidate working in the area of game design at the University of Western Ontario. He finished his Masters in software engineering from the same university.

Dr. Michael Katchabaw is a Associate Professor at the University of Western Ontario. His research interest includes game design and development. He co-founded the Digital Recreation, Entertainment, Art, and Media (DREAM) research group at Western.