A Software Design Pattern Based Approach to Adaptive Video Games

Muhammad Iftekher Chowdhury, Michael Katchabaw
Department of Computer Science
University of Western Ontario
London, Canada
{iftekher.chowdhury, katchab}@uwo.ca

***Abstract*—To achieve success, it is becoming increasingly clear that modern video games must be adaptive in nature – malleable and able to reshape to the needs, expectations, and preferences of the player. Failure to adapt results in a game that is too inflexible, rigid, and pre-defined; one that is simply ineffective, particularly for a large and diverse player population. Developing and supporting adaptive games, however, introduces many challenges. In this paper, we describe a set of software design patterns for enabling adaptivity in video games to address these challenges. We also demonstrate the benefits of our pattern-based approach, in terms of software quality factors and process improvements, through our experience of applying it to a number of video games for enabling a particular type of adaptivity, auto dynamic difficulty.**

Keywords-adaptive video game; software design patterns; game development process; software quality

I. INTRODUCTION

Building rich and dynamic video games is surprisingly complex [1], so much of the existing research and development in this area has led to the creation of games that are largely deterministic in nature. What occurs in these worlds and how this is presented to the player is for the most part fixed, and quite unable to adequately react to the interactions of the player [2,3]. While interesting in their own ways, these games are often too inflexible and rigid to be able to effectively meet the needs and expectations of a large and diverse player population [2,4,5,6], especially as these needs and expectations change as players mature, refine their skills, and form new experiences [7]. In the end, this leads to a loss of engagement, a break of immersion, and an overall disappointing player experience [2,8,9]. The result is a game that is unsuccessful critically and commercially.

As work in this area continues, it is becoming increasingly clear that games must be adaptive in nature — malleable and able to reshape to the needs, expectations, and preferences of the player [2,3]. Adaptive systems are designed to excel at situations that cannot be completely or singularly modeled prior to development, and so they must be able satisfy requirements that arise only after they are put in use; this is very much the case in games. Nearly every aspect of a game can be made adaptive in this way: the game world (structural elements, composition); the population of the world (the agents or characters in the world); any narrative elements (story, history, or back-story); gameplay (challenges, obstacles); the presentation of the game to the player (visuals, music, sound); and so on. In being adaptive, games can provide more compelling, engaging, immersive, and perhaps personalized or customized experiences to their player, leading to a significantly better outcome for the player, and far more success for the game in the end [2,4,5,6,8,9,10].

Previous attempts at adaptivity can be characterized as ad hoc from a software engineering perspective; lacking rigor, structure, and reusability, with custom solutions per game, which is not acceptable [11,12]. There is a critical need for reusable software infrastructure to enable the construction of adaptive games [11,12]. Addressing this problem is the broad goal of our research. While this is a difficult goal to achieve [2,13], both from theoretical and practical perspectives, we have found success in this area by leveraging software design patterns [14].

In particular, we study adaptivity in games through an exploration of a particular problem in this space, that of auto dynamic difficulty. In this case, adaptations are focused on adjusting game difficulty to match the expertise of the player. According to the theory of flow or optimal experience [15], players who lack the skill to suitably deal with the challenges they face will feel anxiety or frustration in their experience, while players whose skills are excessive for the challenges faced will feel boredom or receive no sense of accomplishment from their experience. A game that is properly balanced, on the other hand, will be much better received by the player [16]. A single difficulty level has little chance of addressing the needs of a broad audience. Multiple static difficulty levels in games also 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 [11,12]. An adaptive game supporting auto dynamic difficulty circumvents these problems to deliver a more satisfying experience to players by providing per-player skill-appropriate challenge.

In this paper, we discuss our general approach to adaptive games and demonstrate the effectiveness of our approach by examining auto dynamic difficulty, extending our previous work in this area [11,12]. To do so, we leverage the benefits of software design patterns, derived from self-adaptive system literature [17], to construct an adaptive system for video games that is reusable, portable, flexible, and maintainable.

II. Related Work

In recent years, adaptive video games and auto dynamic difficulty have received notable attention from numerous researchers. In the subsections below, we review key work in this area and discuss the research gap that remains.

A. *Adaptive Game Systems*

The study of adaptive systems in a broader sense is not new. Unfortunately, it is difficult to directly apply adaptive systems work from other domains to video games [11,12]. Games do more than deliver functionality as in other software systems; there is a larger emphasis on engagement, immersion, and experience, as well as greater demands on interactivity and real-time performance and presence. These factors require careful consideration often not required in other domains. Furthermore, adaptations in games can go beyond the tuning found in most other domains; there can also be creative or generative aspects to adaptivity. There exists a separation of logic or processing and content in games; while both can be tuned, the content aspect can be altered in fundamentally different ways that fall outside of traditional approaches to adaptive systems. Consequently, there is a need to study adaptivity in the context of games. To date, efforts in doing so have been rather scant, with the work of Charles et al. [5] one of the few examples. Unfortunately, attempts in this area tend not to leverage progress from the adaptive systems literature, and so are typically too narrow, overly focused, and lack rigor from a software engineering perspective.

That said, while not studying adaptivity in games directly, many researchers studying other issues in this space have created work that has been adaptive, at least to a certain degree. This includes work on agent and story adaptation [18,19,20,21,22,23], varying the structure of the game world [10,24,25,26], and difficulty adjustment, as discussed at length in the next section. Unfortunately, this work is also quite ad hoc and cannot be readily generalized or reused for other purposes.

B. *Auto Dynamic Difficulty*

There have been numerous attempts made towards providing auto dynamic difficulty in video games over the years. In this section, we highlight several of these works.

Bailey and Katchabaw [16] developed an experimental testbed based on Epic's Unreal engine that can be used to implement and study auto dynamic difficulty in games. A number of mini-game gameplay scenarios were developed in the test-bed and these were used in preliminary experiments.

Rani et al. [27] 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 [28] used a probabilistic model to design adaptability in a first person shooter (FPS) game based on the Half Life SDK. They used the game in an experiment on 20 subjects and found that adaptive adjustment 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 that players did not notice the adjustments.

Hao et al. [29] proposed a Monte-Carlo Tree Search (MCTS) based algorithm for auto dynamic difficulty to generate intelligence of Non Player Characters (NPCs). 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 [30] described an adaptive approach for pointing tasks in therapeutic games. They introduced a motivation model based on job satisfaction and activation theory to adapt task difficulty. They also conducted preliminary validation through a control experiment on eight healthy participants using a Wii balance board game.

C. *Research Gap*

It is clear from surveying the literature that a structured, formalized study of adaptivity for video games is needed to continue advancing the state of the art in this area. Indeed, games could benefit greatly by having an infrastructure of frameworks, patterns, libraries, and support tools to enable adaptivity, as is the focus of this paper. In doing so, developers can focus on creating their games and choosing the adaptations desired, leaving the implementation of these adaptations to the provided infrastructure.

Research on auto dynamic difficulty in games focuses on tool building (including frameworks, algorithms, and so on) and empirical studies, but they all use an ad hoc approach from a software engineering perspective. Thus, in this paper, we discuss a software design patterns based approach for enabling adaptivity in games, and explore the application of this approach to auto dynamic difficulty in particular.

III. Design Patterns For Adaptive Games

In this section, we overview our collection of four design patterns for enabling adaptivity in video games. These patterns were derived from the self-adaptive system literature [17], and specialized and refined for games in particular. For further details, the reader is encouraged to refer to [11] for elaborated discussion and examples.

A. *Sensor Factory*

The sensor factory pattern is used to provide a systematic way of collecting data on a game and its players while satisfying resource constraints, and provide those data to the rest of the adaptive 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 calculations. The *SensorFactory* class uses the "factory method" pattern

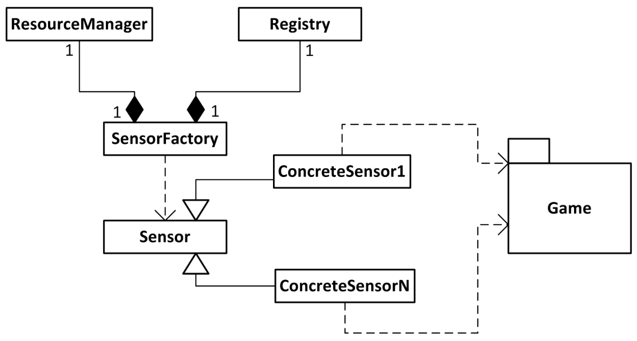

Figure 1. *Sensor factory* design pattern

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.

B. 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 adaptation process might need.

C. Case Based Reasoning

While the adaptation detector determines the situation when an adjustment is required by creating a *Trigger*, case based reasoning (please see Figure 3) formulates the *Decision* that contains the adjustment plan. The

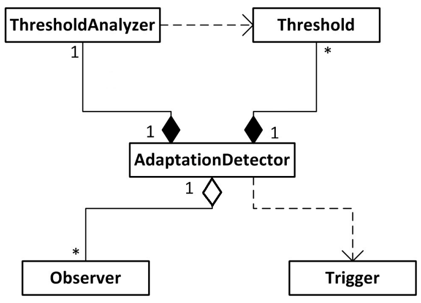

Figure 2. *Adaptation detector* design pattern

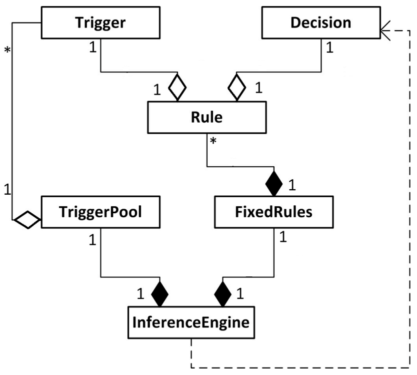

Figure 3. *Case based reasoning* design pattern

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 are 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*.

D. Game Reconfiguration

Once the adaptive system detects that an adjustment is necessary, and decides what and how to adjust the various game components, it is the task of the game reconfiguration pattern (please see Figure 4) to facilitate smooth execution of the decision. The *AdaptationDriver* receives a *Decision* selected by the *InferenceEngine* (please see case based reasoning in previous subsection) 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 behaviour whereas in the inactive state, the object stops its regular tasks and is open to changes.

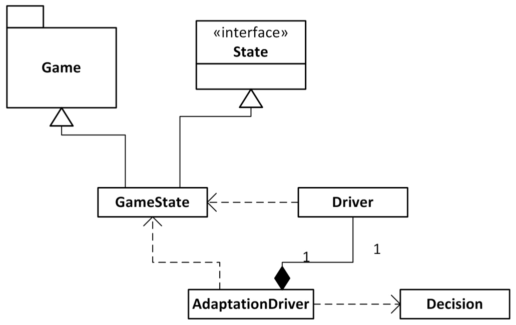

Figure 4. *Game reconfiguration* design pattern

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.

E. Integration of Design Patterns

In [31], Salehie and Tahvildari described integration of four generic steps for an adaptation process namely monitoring, detecting, deciding, and acting. The four design patterns discussed in previous sections work on the same process flow. In this Section, we briefly re-discuss how they work together to create a complete adaptive system (please see Figure 5). The sensor factory pattern uses Sensors to collect data from the game so that the player's state and the game's state can be measured. The adaptation detector pattern observes Sensor data using Observers. When the adaptation detector finds situations where the game needs to be adjusted, because either the player or the game is in a sub-optimal state, it creates Triggers with appropriate additional information. Case based reasoning is then 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 functioning of the game accordingly, 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 adaptive system for a particular game.

F. Enabling Auto Dynamic Difficulty

When used together, these software design patterns are sufficient to implement a wide range of adaptivity in gameplay. To demonstrate their use, we explore the particular adaptation of game challenge delivered to the player in the form of auto dynamic difficulty.

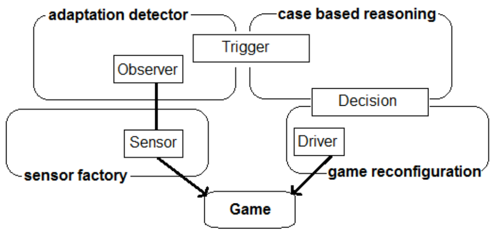

Figure 5. Four design patterns working together in a game

In this application, Sensors would be used to collect data from the game to assess the player's perceived level of difficulty. As above, the adaptation detector pattern observes Sensor data using Observers. When the adaptation detector finds situations where difficulty needs to be adjusted, because the game is currently too easy or too hard for the player, it creates Triggers with appropriate additional information. This information details the in-game activity that gave rise to the Triggers, provides more information on the player's state, and includes anything else needed to assist in formulating a Decision or carrying out reconfiguration. These Triggers are passed to case based reasoning, which in turn finds appropriate Decisions to bring game difficulty back in line with player skill and expertise. These Decisions are then passed to the adaptation driver, which applies the changes specified by each Decision to the game, to adjust the difficulty of the game appropriately, with the help of the Driver. In doing so, the situation is corrected, and game difficulty is tuned according to the needs of the player.

IV. OVERVIEW OF STUDIED GAMES AND ADAPTATIONS

The software design patterns in Section III have been implemented as a Java framework that can be used to enable adaptivity in games. As there is nothing Java-specific to our patterns, bringing this framework to other platforms with other language bindings is part of on-going work.

To date, we have used three very different games developed in Java for studying our approach to adaptivity, with a focus on auto dynamic difficulty. In our earlier work ([11,12]), two casual prototypical games were used. The first game is a variant of Pac-Man and was developed specifically for the purposes of our research. The second game, TileGame, is a slightly modified version of a platform game described in [32]. Even though we were successful in using our approach in these two games, the code for these games was either written by ourselves or well documented and simple enough to be easily understood and reshaped accordingly. Thus, recently we have selected a commercially successful sandbox game – Minecraft [33] to extend our study. Minecraft is commercially available for several platforms, but we focus on the desktop version also developed in Java. In the subsections below, we briefly describe each of the games and examples of adaptations that were implemented using our framework.

A. Pac-Man

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

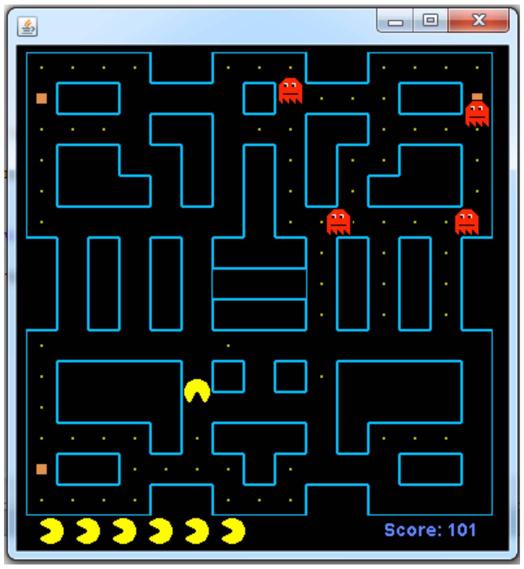

Figure 6. Screen captured from the Pac-Man game

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.

B. *TileGame*

The level structure and gameplay of this game is similar to the popular Super Mario game series. 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. 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.

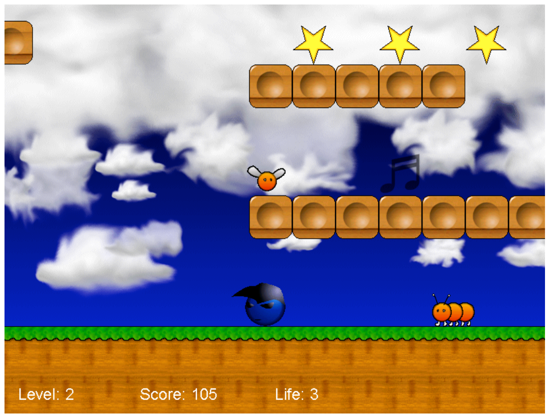

Figure 7. Screen captured from the TileGame game

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 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 not. 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.

C. *Minecraft*

Minecraft [33] is an exceptionally popular sandbox game that allows players to explore, gather resources, combat, craft and build constructions out of textured cubes in a procedurally generated 3D world. The terrain of the game world, consisting of plains, mountains, forests, caves, and waterways, are composed of rough 3D objects (primarily cubes) representing different materials (for example dirt, stone, tree trunks, water, and so on) and arranged in a fixed grid pattern. Players can break (please see Figure 8) and collect these material blocks and craft these blocks to form other blocks (for example, furnaces, bricks, and stairs) and items (for example sticks, axes, and buckets). Players can place collected or crafted blocks and items elsewhere to build structures. The world is divided into biomes (such as deserts, jungles, and snow fields). The time in the game goes through a day-night cycle every 20 real time minutes.

Figure 8. Screen captured from Minecraft

There are various NPCs known as mobs (including animals, villagers, and hostile creatures). Non hostile animals (such as cows, pigs, chickens, and so on) spawn during the daytime and can be hunted for food and crafting materials. Hostile mobs (such as spiders, zombies, and creepers, a Minecraft-unique creature) spawn during nighttime and in dark areas. There are two primary game modes: creative and survival. In creative mode, players have access to unlimited resources, and are not affected by hunger or environmental or mob damage. On the other hand, in survival mode, players need to collect resources (and craft them) and have both a health bar and a hunger bar that must be managed to stay alive and continue playing. The game also features single player and multiplayer options. For this research, we focused on the single player option played in survival mode.

While Minecraft is not open-source, its source code can be readily obtained through the use of a toolchain [34] provided by an active and extensive modding community that decompiles the game back to its source code. The creators of Minecraft accept this practice while an official modding interface is under development.

D. *Adaptations Implemented*

In Table I, we provide examples of different adaptations that we have implemented in the above games. The first column shows the name of the game. The next three columns show the details of the adaptations implemented. Please note that these columns: metrics for sensors, attributes for modification, and adaptation scenarios also represent the questions: when to adapt, what to adapt, and how to adapt respectively, which is part of the methodology for eliciting essential requirements for adaptive software [31].

TABLE I. EXAMPLES OF ADAPTATIONS IMPLEMENTED

Game	Metrics for Sensors	Attributes for Modification	Adaptation Scenarios
Pac-Man	Total score, Number of times player dies	Ghost's speed, the ghost's vision length, duration of Pac-Man's predator mode, and so on	Modify ghost's speed, duration of Pac-Man's predator mode and so on based on how the average score per life compared to specific thresholds
TileGame	Current level number, Total score, Number of times player dies	Load different versions of the map where default objects and enemies are placed in slightly different positions	Load different versions of the map when the player character goes to the next level or in the next loading of the same level (such as when the player character dies) based on score and lives lost in last level.
Minecraft	Which day in game, Number of times player dies	Display hints about collecting resources and building shelters	If the player is continuously dying during the first night, give the player some hints to progress through the game to make it easier.
	Number of items of particular materials in player's inventory	Hardness of those particular items	Modify the hardness of a particular resource in the game world as the player's inventory of that particular item changes, making it easier or harder to collect the resource.

Many adaptations that we have implemented focus primarily on tuning attributes of the game (please see Pac-Man and Minecraft examples in Table I), while others focus on content modifications (please see the TileGame example of usage of different versions of maps in Table I).

V. DISCUSSION

In this section, we discuss the benefits of using a software design pattern approach for implementing adaptivity in video games.

A. *Reusable Source Code*

Reusability refers to the degree to which existing code can be reused in new applications. Since design patterns provide a reusable solution, it is expected that reusable source code can be created for such solutions as well. In [12], we reported an empirical investigation involving source code analysis of the Pac-Man and TileGame games. In that study, we experienced 77.52% and 79.68% code reusability in Pac-Man and TileGame respectively while implementing the adaptive systems using our software design patterns. Recently, we have extended this study to the popular commercial game Minecraft [33] and found comparable results. In Figure 9, we show a summary of these studies, identifying reusable and application-specific logical Source Lines of Code (SLOC). As we can see, 600 SLOC (74.26% in Minecraft; 79.68% in TileGame; and 77.52% in Pac-Man) of the adaptive system remained unchanged across all three games.

Reusability of source code reduces implementation time and increases the probability that prior testing has eliminated defects.

B. *Repeatable Process*

In our design pattern-based approach, since the high level structure of the solution is already known, it is possible to create a step-by-step method for developing adaptive video games. From our experience in implementing adaptivity into Pac-Man and TileGame [11,12], we formalized such a process and applied it to the Minecraft game. In Table II, we provide a generalized description of the process to incorporate the concepts of adaptive gameplay discussed in the previous section.

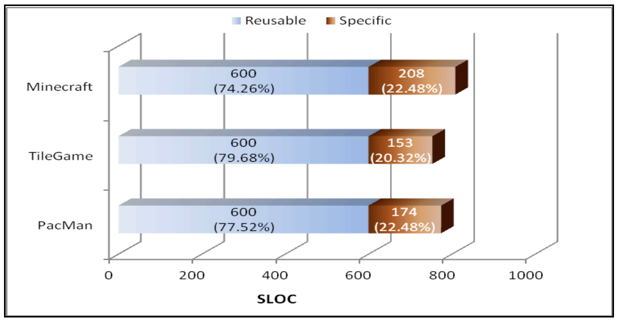

Figure 9. Source code reusability found in adaptive games developed using our design patterns

TABLE II. ADAPTIVE GAME IMPLEMENTATION PROCESS

#	Activity	Output
1	Identify the aspects of the game that will be adaptively adjusted.	
2	For each of the aspects identified in step-1 repeat step-3 to step-9.	
3	Define or reuse available sensors.	Sensors
4	Identify or introduce attributes that can be adjusted.	
5	Identify adaptation scenarios involving sensors and attributes from step-3 and step-4.	
6	Define thresholds based on the scenarios identified in step-5 for the sensors defined in step-3, and define observers to relate thresholds to sensors.	Thresholds, Observers
7	Define triggers to represent each scenario, and develop adaptation detector logic from the scenarios.	Triggers
8	Use attributes identified in step-4 to create decisions to modify game functionality according to the scenarios identified in step-5.	Decisions
9	Define rules to relate triggers to decisions based on the adaptation scenarios identified in step-5.	Rules

A well-defined process for adaptivity is important for industrial adoption as it enables progress tracking, planning, and automation. Furthermore, it allows developers to focus more on gameplay design and adaptive logic design, rather than implementation details. Unlike ad hoc approaches, a well-defined process is repeatable with consistent results across various games. Our study on three different games using the process described above is a primary validation of consistent repeatability of the process. Since the process is defined in a step-by-step method with specific artifacts expected as outputs from each step (please see the third column in Table II), it will be possible to define specific metrics to estimate project size and later measure progress as the project moves forward.

C. Impact on Quality Factors

In [12], we examined how different software quality factors are impacted by the usage of our design patterns. We have already discussed the impact on reusability in subsection A, and so we briefly discuss the impact on other quality factors below.

Integrability: Integrability refers to the ability to make the separately developed components of a 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. Because of these clearly defined integration points, the four design patterns can be integrated with each other and a game rather easily.

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 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 [17]. These design patterns were derived from the self-adaptive system literature in the context of adaptivity in video games, with a particular focus on auto dynamic difficulty. This indicates the portability of these design patterns across domains. Also, in our research, 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, and so on), 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 Figure 9), this will increase the probability that prior testing has eliminated defects while being used in a new game.

D. Automation

Using our approach, it is possible to implement tools that will guide developers through the process of enabling adaptivity in their games. We are currently designing a semi-automatic tool to help developers to easily integrate a game into the tool and then identify metrics for sensors, brainstorm adaptation scenarios, identify attributes to adjust in the game, maintain traceability between these artifacts, and so on. The benefits of such semi-automatic tools include reducing development effort and defects, standardization, ease of progress tracking, and improving maintainability.

VI. CONCLUDING REMARKS

Adaptivity is becoming increasingly essential to modern video games. Previous attempts at adaptivity in games can be characterized as ad hoc from a software engineering perspective; lacking rigor, structure, and reusability, with custom solutions per game. There is a critical need for software frameworks, patterns, libraries, and tools to enable adaptive systems for games. Thus, in this paper, we leverage the benefits of software design patterns to construct a framework for adaptive games. Based on studies of three different games, including the large commercial game Minecraft, we discussed how the usage of these software design patterns results in a reusable approach both in terms of source code and process and improves a number of other quality aspects.

There are many possible directions for future work in this area. We plan to extend our work, enabling auto dynamic difficulty in additional games, exploring other forms of adaptivity, and bringing our framework to other platforms. While our approach is designed to be generalizable, and work to date supports this, further work is necessary to fully assess this and identify limitations to our approach. To further assess the effectiveness and efficiency of our approach, we will conduct extensive user testing and performance testing. Since a key goal of adaptivity in games is an improved player experience, this user testing is essential. Lastly, to assist developers, we will continue developing semi-automatic and automatic tools to enable adaptivity with minimal effort on their part.

REFERENCES

[1] G. Dolbier and A. Goldschmidt, The Business of Interactive Entertainment. IBM Digital Media Solutions Technical Report G565-1461-00, May 2006.

[2] A. Glassner, Interactive Storytelling: Techniques for 21st Century Fiction. A K Peters, Ltd., 2004.

[3] P. Sweetser, Emergence in Games. Charles River Media, 2008.

[4] G. Andrade, G. Ramalho, H. Santana, and V. Corruble., "Challenge-Sensitive Action Selection: An Application to Game Balancing". In the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Compiègne, France, September 2005, pp. 194-200.

[5] D. Charles, M. McNeill, M. McAlister, M. Black, A. Moore, K. Stringer, J. Kücklich, and A. Kerr, "Player-Centred Game Design: Player Modelling and Adaptive Digital Games". Proceedings of DiGRA 2005 Conference: Changing Views Worlds in Play, June 2005, pp. 285-298.

[6] P. Langley, Machine Learning for Adaptive User Interfaces. Kunstiche Intellugenz, 1997.

[7] D. Charles and M. Black, "Dynamic Player Modelling: A Framework for Player-Centered Digital Games". In Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education, Microsoft Campus, 2004, pp. 8-10.

[8] B. Pfeifer, "Creating Emergent Gameplay with Autonomous Agents". Proceedings of the Game AI Workshop at AAAI-04, San Jose, California, July 2004, pp.20.

[9] B. Reynolds. How AI Enables Designers. Appeared in the Proceedings of the 2004 Game Developers Conference, San Jose, California, March 2004, pp. 20.

[10] G.N. Yannakakis and J. Hallam, "Real-time Game Adaptation for Optimizing Player Satisfaction". IEEE Transactions on Computational Intelligence and AI in Games, 1(2), 2009, pp. 121-133.

[11] M. Chowdhury and M. Katchabaw, "Software Design Patterns for Enabling Auto Dynamic Difficulty in Video Games". Proceedings of the 17th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games. Louisville, Kentucky. July, 2012, pp. 76-80.

[12] M. Chowdhury and M. Katchabaw, "Improving Software Quality Through Design Patterns: A Case Study of Adaptive Games and Auto Dynamic Difficulty". Proceedings of GameOn 2012. Magala, Spain. November, 2012, pp. 41-47.

[13] E. Adams, Fundamentals of Game Design, Second Edition. New Riders, 2010.

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

[15] M. Csikszentmihalyi., Creativity: Flow and the Psychology of Discovery and Invention. New York, NY: Harper Collins Publishers. 1996.

[16] C. Bailey and M. Katchabaw, "An Experimental Testbed to Enable Auto-Dynamic Difficulty in Modern Video Games". In Proceedings of the 2005 North American Game-On Conference. Montreal, Canada. August 2005, pp. 18-22.

[17] A. Ramirez and B. Cheng, "Design Patterns for Developing Dynamically Adaptive Systems". Proceeding of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. Cape Town, South Africa, May 2010, pp. 49-58.

[18] P. Baillie-de Byl, Programming Believable Characters in Games. Charles River Media, 2004.

[19] J. Dias, S. Mascarenhas, and A. Paiva, "FAtiMA Modular: Towards an Agent Architecture with a Generic Appraisal Framework". Workshop on Standards in Emotion Modeling, Leiden, Netherlands, August 2011, pp. 12.

[20] A. Guye-Vuilleme and D. Thalmann, A High-Level Architecture For Believable Social Agents. Virtual Reality, Volume 5, Number 2, 2001, pp. 95-106.

[21] M. Nelson, C. Ashmore, and M. Mateas, "Authoring an Interactive Narrative with Declarative Optimization Based Drama Management". Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE). Marina del Rey, California, June 2006, pp. 127-129.

[22] P. Spronck, "A Model for Reliable Adaptive Game Intelligence". IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games, 2005, pp. 95-100.

[23] R. Zhao, Applying Agent Modeling to Behaviour Patterns of Characters in Story Based Games. PhD Thesis, University of Alberta, 2010.

[24] K. Compton and M. Mateas, "Procedural Level Design for Platform Games". Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE). Marina del Rey, California, June 2006, pp. 109-111.

[25] C. Pedersen, J. Togelius, and G. Yannakakis, "Optimization of Platform Game Levels for Player Experience". Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), Oct., 2009, pp. 191-192.

[26] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards Automatic Personalised Content Creation in Racing Games". Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games. April 2007, pp. 252-259.

[27] P. Rani, N. Sarkar, and C. Liu. "Maintaining Optimal Challenge in Computer Games Through Real-time Physiological Feedback". Proceedings of the 11th Intl. Conf. on Human-Computer Interaction. Las Vegas, USA, July 2005, pp. 184-192.

[28] R. Hunicke, "The Case for Dynamic Difficulty Adjustment in Games". Proceedings of the 2005 ACM SIGCHI International Conf. on Advances in Computer Entertainment Technology. Valencia, Spain, June 2005, pp. 429-433.

[29] 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". Proceedings of the Sixth Int. Conference on Natural Computation. Yantai, China, August 2010, pp. 3918-3922.

[30] N. Hocine and A. Gouaïch, "Therapeutic Games' Difficulty Adaptation: An Approach Based on Player's Ability and Motivation". Proceedings of the 16th Intl. Conf. on Computer Games. Louisville, Kentucky, USA, July 2011, pp. 257-261.

[31] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges". In ACM Transactions on Autonomous and Adaptive Systems, Vol. 4, No. 2, Article 14, May 2009, pp. 1-42.

[32] D. Brackeen, B. Barker, and L. Vanhelsuwé, Developing Games in Java. New Riders, 2004.

[33] Mojang, Minecraft. Retrieved from: https://minecraft.net. Last accessed: Jan 29, 2013.

[34] MCP Team, Main Page – Minecraft Coder Pack, Retrieved from: http://mcp.ocean-labs.de/. Last accessed: Jan 29, 2013.