An Emerging Automation Framework for Adaptive Video Games

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

Abstract—**Previous attempts at adaptive video 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. In this paper, we present architecture of a semi-automatic framework that leverages code generation based on design patterns to introduce adaptability in video games. We also discuss key responsibility and implementation choices for each components of the framework.**

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

I. INTRODUCTION

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]. Adaptive systems are designed to excel at situations that cannot be completely or singularly modeled prior to development, and so they must be able to 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].

The software engineering literature on adaptive systems provides various solutions focusing on software requirements, system architectures, software design patterns, and so on. Unfortunately, it is difficult to directly apply adaptive systems work from other domains to video games [10]. 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, the adaptive video game literature primarily focuses on algorithms, frameworks, empirical studies and game design activities but rarely takes any benefits from the progress in adaptive system literature. Previous attempts at adaptability in video games 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 [10]. There is a critical need for reusable software infrastructure to enable the construction of adaptive games [11]. Addressing this problem is the broad goal of our research. While this is a difficult goal to achieve [2], both from theoretical and practical perspectives, we have found success in this area by leveraging software design patterns [1].

In our earlier work [11], we discussed our design pattern based approach to adaptive games and demonstrate the effectiveness of our approach through case studies. Our current goal is to create tool support that will assist developers in introducing adaptability in video games using the design patterns. Existing literature (e.g., [14][20]) suggest that design patterns are specifically suitable for code generation. We also noticed a high percentage of code reusability while using these design patterns during our earlier case studies. Motivated from these points, in this paper, we present the architecture of a semi-automatic framework that leverage code generation based on design patterns to introduce adaptability in video games.

The rest of this paper is organized as follows. In Section II, we discuss the literature reviewed. In Sections III and IV, we describe the design patterns for adaptive video games and motivation behind our current work. In Section V, we present architecture of a semi-automatic framework that leverage code generation based on design patterns to introduce adaptability in video games. In Section VI, we conclude the paper.

II. RELATED WORK

In recent years, adaptive video games and auto dynamic difficulty have 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. In below sub-sections, we review some of these works.

A. *Adaptive Game*

In the highly influential work [4], Charles and Black propose a framework for adaptive video games incorporating ideas of player-centered game design comprising four key aspects: player modeling, adaptive game environments in

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-391-9

response to player needs, monitoring the effectiveness of any adaptation, and dynamic player classification. They also proposed several neural network approaches for instantiating this framework.

Andrade et al. [7] developed a 2D fighting game where players utilized one of the four strategies: random, state-based, traditional (optimal) reinforcement learning (ORL agent), and adaptive reinforcement learning (ARL agent). Their results showed that the ARL agent was able to adapt to all three types of opponents with a relatively small number of games played.

Togelius et al. [9] attempted to evolve tracks of racing games that fit the players' driving styles to increase overall entertainment. Tracks were given a number of control points based on different implementations and the adaptation algorithm used these control points as locations to alter the shape of the track. They found that using a segment based method of control point distribution resulted in tracks having long straight paths for beginner players and sharper turns for advanced players.

B. *Auto Dynamic Difficulty*

Bailey and Katchabaw [3] developed an experimental testbed based on Epic's Unreal engine that can be used to implement and study auto dynamic diificulty in games. A number of mini-game gameplay scenarios were developed in the test-bed and these were used in preliminary validation experiments.

Rani et al. [17] 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 [18] used a probabilistic model to design adaptability 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 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 the players did not notice the adjustments.

Hao et al. [19] proposed a Monte-Carlo Tree Search (MCTS) based algorithm for auto dynamic difficulty 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 [16] described an adaptive 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.

III. DESIGN PATTERNS

In this section, we briefly discuss the four design patterns for enabling adaptability in video games. For further details, the reader is encouraged to refer to [10] 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, 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 calculation. The *SensorFactory* class uses the "factory method" pattern to provide a unified way of creating any sensors. 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.

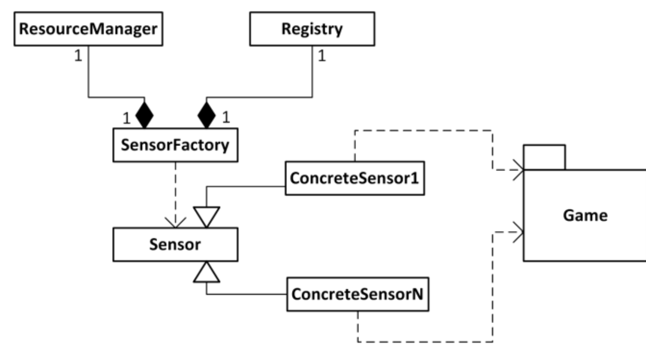

Figure 1. *Sensor factory* design pattern.

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.

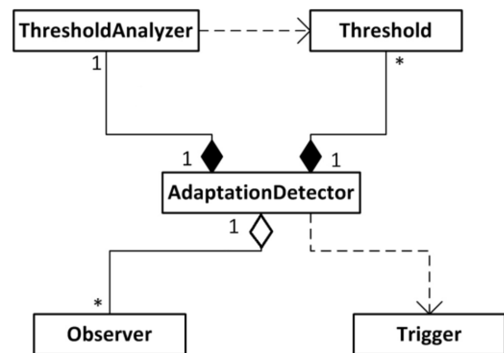

Figure 2. *Adaptation detector* design pattern

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 *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*.

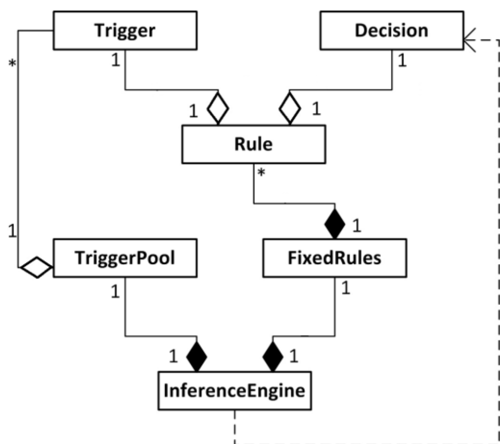

Figure 3. *Case based reasoning* design pattern

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 behavior whereas in the inactive state, the object stops its regular tasks and is open to changes. In the being inactive state, the game finishes the existing tasks based on the already processed player inputs but does not start any new task. In the being active state, the game does not start task based on player input and is not open to any new changes. The *Driver* takes the object to be reconfigured, details of the attribute to be changed and the changed attribute value as inputs. The *Driver* requests the object that needs to be reconfigured to be inactive. 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.

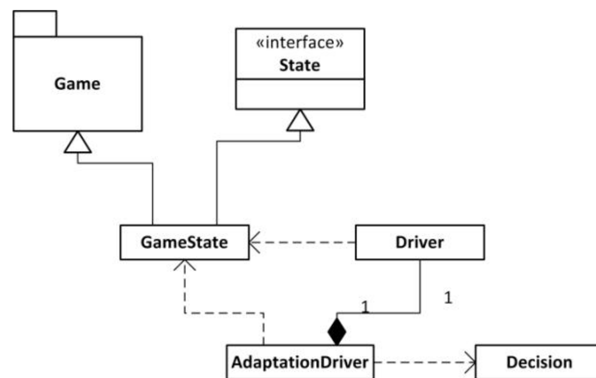

Figure 4. *Game reconfiguration* design pattern

E. Integration of Design Patterns

In [15], 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 sub-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).

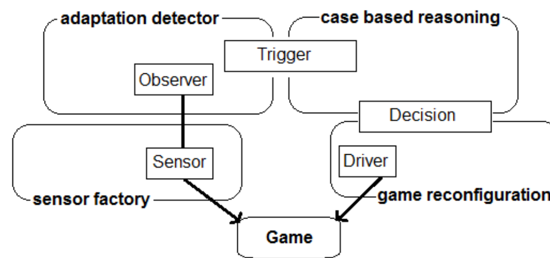

Figure 5. Four design patterns working together in a game

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 adaptive system for a particular game.

F. Achieving Adaptive Gameplay

So far we have used these design patterns for implementation of a specific type of adaptability in video games known as auto dynamic difficulty. But in principle these design patterns should be sufficient to implement more complex form of adaptability in game-play.

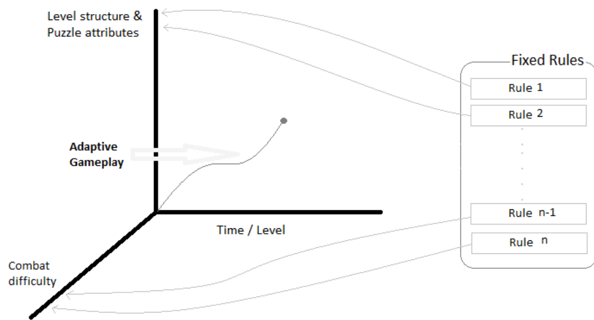

Figure 6. Concept of multi dimensional adaptive gameplay

Figure 6 depicts our position of a multidimensional adaptive game-play. For example, we have chosen two aspects of the game to adjust adaptively. One is *level structure and puzzle attributes*. And the other is *combat difficulty*. There are number of rules and other associated artefacts (i.e., sensors, observers, triggers and decisions) focused on each of these aspects. In a particular *level structure and puzzle attributes* with minimum *combat difficulty* the player may experience a maze type game whereas with a high *combat difficulty* and simple *level structure and puzzle attributes* the player may experience a fighting game. 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); game-play (challenges, obstacles); the presentation of the game to the player (visuals, music, sound); and so on.

IV. MOTIVATION FOR AUTOMATION

In this section, we discuss two key motivations behind our automation effort: the repeatable nature of the process for applying the design patterns and source code reusability achieved through the usage of the design patterns for implementing adaptability in video games.

A. Repeatable Process

Applying our software design pattern based framework for adaptability to a large commercial-scale game such as Minecraft [13], seemed to be a daunting task, at least on the surface. Thus, the process described in Table I was developed to formalize our experiences from using it in Pac-Man [10] and TileGame [11] to assist in the adaptability-enablement of larger games such as Minecraft. In practice, we found that applying such a methodical process enabled adaptability in Minecraft quite readily, and that our approach was easily adapted for use in this rather foreign environment with no more significant changes than we found in our earlier work with much simpler games. This is a key motivation for our current work as concrete activities (such as the ones in Table I) are easier to build a tool upon.

TABLE I. 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 the adaptation detector logic based on the scenarios.	Triggers
8	Use attributes identified in step-4 to create decisions to modify game difficulty 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

B. Reusable Source Code

We have also carried out a source code analysis of these games. In [11], the Minecraft adaptability implementation was compared to the adaptability implementations of Pac-Man and TileGame. During this analysis, we have noticed that a large percentage of the resultant code is generalization and instantiation of other high level classes (e.g., Sensors, Triggers, Thresholds, and Decisions etc.).

TABLE II. CATEGORIZATION OF THE ADAPTIVE SOURCE CODE

Category of Source Code	SLOC	%
Completely reusable	600	74.26
Specialization (Concrete Sensors (64) and Concrete Decisions (22))	86	10.64
Instantiation (Adaptation Detector (70) and Inference Engine (11))	81	10.02
Other logic	41	5.07
Total	808	

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-391-9

In Table II, we provide a summary of the analysis derived from the results presented in [11]. Here, we can see that 74.26% source code remained the same from earlier projects. Also, 10.64% source code is specializations and 10.02% code is for instantiation. Only 5.07% source code is other specific game logic. The specialization and instantiation (20.66%) related source code of the adaptive system consists of similar looking classes and statements. This result motivates us to create a tool that will allow us to develop and maintain these artifacts in a semi-automatic manner.

V. AUTOMATION FRAMEWORK

Figure 7 depicts a high level decomposition of our semi-automatic system. The key idea is to represent part of the adaptive logic as a relational model that is mutable. The core software elements are divided into four components: (i) Collector and Executor, (ii) Enhancer, (iii) Manager, and (iv) Translator. The collector and executor component interfaces the relational model with the game in question. It collects meta-information from the game's source code as well as runtime logging information and passes that to the model. It can also execute modification instructions presented in the model. The manager component provides graphical user interfaces to easily manipulate the model. The enhancer component facilitates the decision making process (i.e., when, how and to what degree to modify the game). The purpose of the translator component is translating the relational model, when finalized, to executable software artifacts (i.e., source code). In the following subsections, we discuss each of these components in further detail.

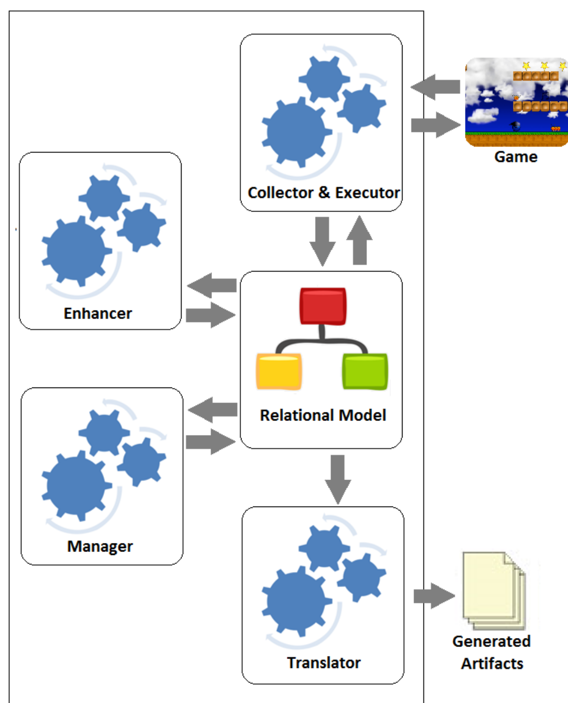

Figure 7. Components of the semi-automatic framework

A. Relational Model

Central to the framework is a relational model, as all the other components use it as a repository for all of their information. This is essentially storage for a set of objects and relations that represent much of the dynamic information (e.g., Sensor's name, relations between sensors and attributes, etc.) for an intended adaptive system as well as some meta-information (e.g., attributes, logging information, etc.). The structure of the model is derived from the design patterns described earlier and is not dependent on the platform or genre of the video game. There should be appropriate APIs for other components to collect information from the model. Implementation choices for the relational model include databases, XML storage, file based data structures, amongst others.

B. Collector and Executor

The collector and executor component interfaces the relational model with the game and thus should depend on the platform of the game. The collector needs to be configured with some base level objects (e.g., game world, player, enemies, inventory etc.). For the rest of the system to work, the collector needs to conduct a Breadth-First Search (BFS) starting from those base level objects and populate the model with a list of attributes and related data types using a hierarchical storage method such as recursive relations. Many languages provide programmatic ways (e.g., Java reflection) to collect such information with ease.

We have identified some key challenges regarding the implementation of the executor and the relational model:

- Identifying the depth of the object hierarchy to search,

- Representing relationships other than hierarchical ones and representing shared objects,

- Representing any run time changes on the hierarchy.

The executor can execute modification instructions presented as decisions in the relational model and the collector can collect more information based on those modifications.

C. Manager

The manager is another generic component that does not need to be aware of the details of the rest of the system and the platform other than the relational model. It is a collection of graphical user interfaces and business logic to easily manage the relational model. Once the attributes are recorded by the collector, they can be marked to be monitored using this component.

D. Enhancer

The enhancer is also a generic component and only needs to interact with the model and thus can be implemented in any language and need not be aware of the game's platform. It is a collection of tools that helps the game designer or developer to make decisions about which attributes to monitor, threshold values, which attributes to modify and to what degree, amongst others. It usually works on data

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-391-9

collected by the collector. Here we give examples of such tools:

- Statistical analysis: Such as factor and co-relation analysis.

- Graphical analysis: Such as curve fitting.

- Machine learning: For example, in [6], Southey et al. described an active learning based semi-automatic gameplay analysis tool that interacts with game-engine or frameworks like this one through an abstraction layer and mainly consists of a sampler, a learner and a visualizer component. The usage of the tool is demonstrated in commercial context (i.e., Electronic Art's [5] FIFA'99).

E. Translator

The translator component needs to be aware of the platform of the video game and needs to generate the artifacts accordingly. It can either directly translate to source code or generate an intermediate marked up description suitable for other code generation tools. The code generation logic is often quite straight-forward. For each file (e.g., Java class), the static parts of the code need to be predefined and the translator injects the dynamic parts as necessary. Please see literature on source code generation (e.g., [8]) for elaborated discussion and methodologies.

Benefits of such semi automatic tools include reducing efforts and defects, standardization, ease of progress measurability and improving maintainability, etc.

VI. CONCLUDING REMARKS

There is a critical need for software frameworks, patterns, libraries, and tools to enable adaptive systems for games. We have found success in this area by leveraging software design patterns. In this paper, we present architecture of a semi-automatic framework that leverage code generation based on design patterns to introduce adaptability in video games. Benefits of such a tool include minimizing developer efforts and increasing maintainability. We designed the framework following a loosely coupled architecture that is generalizable across various platforms. We will discuss a prototype (preliminary discussion of a proof-of-concept prototype and its usage can be found in [12]) based on this framework in our subsequent work. We also encourage other researchers to extend our framework as appropriate.

REFERENCES

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

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

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

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

[5] EA Sports, Retrieved from: http://www.easports.com/. Last accessed: Feb 14, 2015.

[6] F. Southey, G. Xiao, R. C. Holte, M. Trommelen, and J. Buchan, "Semi-Automated Gameplay Analysis by Machine Learning," in Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05), 2005, pp. 123-128.

[7] G. Andrade, G. Ramalho, H. Santana, and V. Corruble, "Challenge-sensitive action selection: an application to game balancing," In Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2005, pp. 194-200.

[8] Herrington, J. "Code Generation in Action,". Manning Publications Co., 2003.

[9] J. Togelius, R. D. Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," In Proceedings of the IEEE International Symposium on Computation Intelligence and Games (CIG 2007), 2007, pp. 252-259.

[10] M. I. Chowdhury and M. Katchabaw, "Software design patterns for enabling auto dynamic difficulty in video games," In 17 International Conference on Computer Games, 2012, pp. 76 - 80.

[11] M. I. Chowdhury and M. Katchabaw, " A Software Design Pattern Based Approach to Adaptive Video Games," In Proceedings of the 5th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE-2013), 2013, pp. 40 - 47.

[12] M. I. Chowdhury. 2014 "A Software Design Pattern Based Approach to Auto Dynamic Difficulty in Video Games," PhD Thesis, University Of Western Ontario. Can be accessed online at: http://ir.lib.uwo.ca/etd/2522/. Last accessed: Feb 14, 2015.

[13] Minecraft, Retrieved from: https://minecraft.net/. Last accessed: Feb 14, 2015.

[14] M. Ohtsuki, N. Yoshida, and A. Makinouchi, "A source code generation support system using design pattern documents based on SGML," in Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC '99) , pp.292-299.

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

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

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

[18] R. Hunicke, "The case for dynamic difficulty adjustment in games," In Proceedings of 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology (ACE 2005), 2005, pp. 429-433.

[19] 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 Proceedings of 6th International Conference on Natural Computation (ICNC 2010), 2010, pp. 3918-3922.

[20] Y. Seo and Y. Song. 2006. A study on automatic code generation tool from design patterns based on the XMI. In 2006 International Conference on Computational Science and Its Applications - Volume Part IV, pp. 864-887.