Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

9-12-2014 12:00 AM

A Software Design Pattern Based Approach to Auto Dynamic Difficulty in Video Games

Muhammad Iftekher Chowdhury, *The University of Western Ontario*

Supervisor: Dr. Michael Katchabaw, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science
© Muhammad Iftekher Chowdhury 2014

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Graphics and Human Computer Interfaces Commons, and the Software Engineering Commons

Recommended Citation
Chowdhury, Muhammad Iftekher, "A Software Design Pattern Based Approach to Auto Dynamic Difficulty in Video Games" (2014). *Electronic Thesis and Dissertation Repository*. 2522.
https://ir.lib.uwo.ca/etd/2522

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

A SOFTWARE DESIGN PATTERN BASED APPROACH TO AUTO DYNAMIC DIFFICULTY IN VIDEO GAMES

(Thesis format: Monograph)

by

Muhammad Iftekher Chowdhury

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Muhammad Iftekher Chowdhury 2014

Abstract

From the point of view of skill levels, reflex speeds, hand-eye coordination, tolerance for frustration, and motivations, video game players may vary drastically. Auto dynamic difficulty (ADD) in video games refers to the technique of automatically adjusting different aspects of a video game in real time, based on the player's ability and emergence factors in order to provide the optimal experience to users from such a large demography and increase replay value. In this thesis, we describe a collection of software design patterns for enabling auto dynamic difficulty in video games. We also discuss the benefits of a design pattern based approach in terms of software quality factors and process improvements based on our experience of applying it in three different video games. Additionally, we present a semi-automatic framework to assist in applying our design pattern based approach in video games. Finally, we conducted a preliminary user study where a Post-Degree Diploma student at the University of Western Ontario applied the design pattern based approach to create ADD in two arcade style games.

Keywords

Video game, software design pattern, reusability, case study, empirical studies, auto dynamic difficulty, adaptive games

Dedication

To my beautiful daughter Raaha. She has nothing to do with this thesis; but everything to do with everything else in my life.

Acknowledgments

First and foremost, I want to thank Almighty Allah for His blessings and mercy upon me. He gave me the strength, knowledge, enthusiasm, determination, persistence, and patience to finish this work.

I would like to thank my supervisor, Dr. Michael Katchabaw, for his guiding support in the carrying out of this research. It has been an honor to work under his supervision, and I am truly indebted for his encouragement during this time.

I am particularly thankful to James Anderson for voluntarily participating in the study related to my research. In the study, he applied the design pattern based approach to two different video games and kept track of the effort spent and provided a useful thorough critical analysis of the design patterns. I greatly appreciate his participation and feedback, and hope that the experience will be helpful for him as well.

To my parents, brother, mother and father in law, brother in laws, sister in law and other family members, I appreciate the support and encouragement you have given to me. I would like to thank our friends who have always been there for encouragement.

Last but not the least; I would like to thank my wife, Sharmin, and my daughter, Raaha, for all their support and sacrifices.

Table of Contents

Abstract ii

Dedication iii

Acknowledgments iv

Table of Contents v

List of Tables viii

List of Figures ix

List of Appendices x

Chapter 1 1

Introduction 1

- **1.1 Motivation** 3
- **1.2 Type of Research** 4
- **1.3 Organization of Thesis** 4

Chapter 2 6

Related Work 6

- **2.1 Auto Dynamic Difficulty** 6
- **2.2 Software Design Patterns in Video Games** 17
- **2.3 Research Gap** 18

Chapter 3 20

Research Organization 20

- **3.1 Research Goals** 20
- **3.2 Research Studies** 21

Chapter 4 24

Design Patterns 24
4.1 Monitoring Pattern 25
4.2 Decision Making Patterns 28
4.3 Reconfiguration Pattern 31
4.4 Integration of Patterns 34
Chapter 5 36
Games Studied 36
5.1 Pac-Man 37
5.2 TileGame 39
5.3 Minecraft 40
5.4 Space Invaders 41
5.5 Tetris 42
5.6 Adaptations Implemented 44
5.7 Reusable Solution across Multiple Games 45
Chapter 6 47
Source Code and Process Reusability 47
6.1 Process 48
6.2 Source Code 53
6.3 Summary 57
Chapter 7 58
Automation Framework 58
7.1 Automation Framework 59
7.2 Proof-of-concept Prototype 63
7.3 Prototype Usage 71
7.4 Summary 74
Chapter 8 75

Preliminary User Study 75
8.1 Study Artifacts 75
8.2 Participant 77
8.3 Adaptations Implemented 77
8.4 Analysis Conducted 80
8.5 Results and Interpretations 81
8.6 Summary 88
Chapter 9 89
Conclusions 89
9.1 Key Contributions 89
9.2 Implications 90
9.3 Future Directions 93
9.4 Concluding Remarks 96
Appendices 103

List of Tables

Table 1: Decomposed executable studies from research goals 21

Table 2: Creating sensors using Java reflection 27

Table 3: Bypassing access modifier using Java reflection 27

Table 4: Summary of ADD design patterns 34

Table 5: Examples of adaptation implemented 44

Table 6: Example of artifacts produced through the ADD process activities 52

Table 7: Source code analysis of ADD design pattern implementation 56

Table 8: Categorization of the ADD source code 59

Table 9 : Interaction between each tables and other framework components 64

Table 10 : Pseudo code for generating sensor class in Java 69

Table 11: Custom source coded to integrate the framework generated source code to an existing game 73

Table 12: Demography of the preliminary user study participant 77

Table 13 : Average clearance rate based scenario in the Tetris game 78

Table 14 : Stack height based scenario in the Tetris game 79

Table 15 : Combination of scenarios in the Space Invaders game 80

Table 16 : Ease of usage of each of the design patterns on the Tetris Game 81

Table 17 : Ease of usage of each of the design patterns on the Space Invaders game 82

Table 18 : Effort spent of implementing ADD in the Tetris and the Space Invaders games 82

List of Figures

Figure 1: *Sensor factory* design pattern.......26

Figure 2: *Adaptation Detector* design pattern.......28

Figure 3: *Case based reasoning* design pattern.......30

Figure 4: *Game reconfiguration* design pattern.......32

Figure 5: ADD design patterns working together.......34

Figure 6: Screen captured from the Pac-Man game.......38

Figure 7: Screen captured from the TileGame game.......38

Figure 8: Screen captured from the Minecraft game.......41

Figure 9: Screen captured from the Space Invaders game.......42

Figure 10: 7 Tetriminos (top) and Screen captured from the Tetris game (bottom).......43

Figure 11: Components of the semi-automatic framework.......60

Figure 12: Schema of the MySQL database for the relational model.......64

Figure 13 : Number of attributes at different depths on the Tetris game.......67

Figure 14 : Screenshot of attribute tree visualization for the Tetris game.......68

Figure 15 : Screenshot of a sample session timeline visualization.......68

Figure 16 : Summary of participant's feedback about the design patterns and base level implementation.......85

Figure 17: Concept of multi-dimensional adaptive gameplay.......94

List of Appendices

Appendix A: Programmer's Manual for the Usage of the Base Level Implementations of the Design Patterns 103

Appendix B: User's Manual for the Proof-of-concept Automation Tool 112

Appendix C: PHP Source Code for the Translator Component 126

Appendix D: Example Source Code from the Base Level Implementation of the Design Patterns 132

Appendix E: Source Code Generated by the Proof-of-concept Automation Tool 154

Chapter 1

Introduction

Building dynamic video games is surprisingly complex; 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 the virtual game 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 [1]. 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 [1], especially as these needs and expectations change as players mature, refine their skills, and form new experiences [2]. In the end, this leads to a loss of engagement, a break of immersion, and an overall disappointing player experience [3][4].

It has been recently reported [5] that 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 players become frustrated when the games are too hard and bored when they are too easy [6]. From the point of view of skill levels, reflex speeds, hand-eye coordination, tolerance for frustration, and motivations, video game players may

vary drastically [7]. These factors together make it very challenging for video game designers to set an appropriate level of difficulty in a video game. Traditional static difficulty levels (e.g., easy, medium, hard) often fail in this context as they expect the players to judge their ability themselves appropriately before playing the game and also try to classify them in broad clusters (e.g., what if easy is too easy and medium is too difficult for a particular player?).

Auto dynamic difficulty (ADD), also known as dynamic difficulty adjustment (DDA) or dynamic game balancing (DGB), refers to the technique of automatically changing the level of difficulty of a video game in real time, based on the player's ability (or, the effort s/he is currently spending) in order to provide them with an "optimal experience", also sometimes referred to as "flow". If the dynamically adjusted difficulty level of a video game appropriately matches the expertise of the current player, then it will not only attract players of varying demographics but also likely to 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 [7][8]. How ADD is delivered in these games from a gameplay perspective can only be discerned through reviewing these games or from official strategy guides (or, occasionally in presentations such as [9]). Unfortunately, given the highly competitive nature of the games industry, no information is publicly available about how ADD is implemented in these games from a software design perspective. While others have studied ADD in games, this has been done in an ad-hoc fashion in terms of software design and is therefore not reusable or applicable to other games. Recreating an ADD system on a game-by-game basis is both

expensive and time consuming, ultimately limiting its usefulness. For this reason, we were motivated to leverage the benefits of software design patterns[1] [10][11] to construct an ADD framework and system [12] that is reusable, portable, flexible, and maintainable.

1.1 Motivation

The concept of auto dynamic difficulty is used in variety of games from commercially successful third person shooter games (i.e., “Max Payne”) to research based therapeutic games [13]. In terms of genre, their applicability is shown in prey and predator (e.g., Pac-Man) [6], cognitive (e.g., Pong) [14], first-person shooter (e.g., Half-life) [15], and platform (e.g., Super Mario Bros) [16] games. Also, the nearly universal presence of different difficulty levels in video games suggests the broader applicability of auto dynamic difficulty research. As we have mentioned earlier, unfortunately, from a software design perspective, there is no information publicly available about how ADD is implemented in commercial games. Furthermore, research in this area has largely been done in an ad-hoc fashion and is therefore not reusable or applicable to other games. As we have prior expertise in empirical software engineering research, we quickly identified our opportunity to contribute in this broader area by bringing knowledge from similar adjacent domain

---

[1] In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is a template for how to solve a problem that can be used in many different situations. They are formalized best practices that the programmer can use to solve common problems when designing an application or system.

such as self-adaptive systems (please see Chapter 4 for details), and applying it in video games, and gaining further knowledge through empirical study.

1.2 Type of Research

Our research has both knowledge seeking and solution building components. Examples of our solution building research include deriving software design patterns from other domains in the context of video games, applying those design patterns in different games, implementing a source code generation based proof-of-concept tool to assist in applying those design patterns, and so on. Examples of knowledge seeking research include collecting metrics through source code analysis of our implementation, textual content analysis of feedback from a preliminary user study, and so on. In Chapter 3, we will provide a more detailed overview of the organization of our research.

1.3 Organization of Thesis

The rest of this thesis is organized as follows. In Chapter 2, we discuss the state of the art of auto dynamic difficulty. In Chapter 3, we discuss the organization of our research in terms of research goals and studies. In Chapter 4 and Chapter 5, we describe the design patterns and the games that we used in our studies respectively. In Chapter 6, we present a step-by-step process for applying our design pattern based approach in a video game. In this chapter, we also present results from our empirical studies regarding the source code reusability achieved through our approach. In Chapter 7, we present a semi-automatic framework based on our design pattern based approach and a proof-of-concept prototype realizing that

framework. In Chapter 8, we discuss a preliminary user study and the feedback from the participant. Finally, in Chapter 9, we discuss future directions and conclude the thesis.

Chapter 2

Related Work

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

2.1 Auto Dynamic Difficulty

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

In [17], Demasi and Cruz explored the potential of co-evolutionary algorithms[2] to create a user-driven evolution of agents in an ANSI C based online action game. Here user-driven evolution means the enemies evolve and get smarter by the same proportion as the player gets better by playing the game. The game scenario is a square room (480 x 480 pixels) where the player character needs to survive against some 16 little monsters (a touch from any monster kills the player character). The player character has a gun to fight the monsters. When the player character kills a monster, another one enters the room, so that there are always 16 enemies alive. The player character starts with 20 shots in the gun and every 15 seconds a new cartridge with 20 shots appears in a random location in the game. The player character can teleport once in every 30 seconds from its local position to a random location. The player character and the enemies have the same speed. The enemies can move only in four directions (up, down, left, right), but the player character can walk or shoot in any one of the eight directions including diagonals. The player character has three lives; once all lives are lost the game is over. The final score is the number of enemies killed. These 16 non-player characters (NPC) are monitored and evolved when they die or reach their "time to live". The authors proposed four different methods for the online evolution of the agents: (i) using game specific information; (ii) online evolution using offline-evolved data; (iii) using online data

---

[2] Co-evolutionary algorithms (CEAs) are defined by their interaction-driven fitness, which means an individual fitness is determined based upon the interaction with other individuals in the population. That interaction can be cooperative, which means that individuals are evolving towards a common goal, or it can be competitive, which means that individuals are competing among themselves to win some sort of resource.

only; and (iv) using method-iii after method-i or ii. The authors used a heuristic fitness function for agent evolution and analyzed different game based values. The results indicated that method-iii (i.e., using online data only) can yield good results for online games which require real-time interaction and are unpredictable to some degree.

In [6], Hao et al. proposed the use of Monte-Carlo Tree Search (MCTS) algorithms to generate the intelligence of NPCs. The performance of the NPCs controlled by MCTS is adjusted by modulating the simulation time of MCTS. The authors use a slightly modified version of the popular prey and predator game genre of Pac-Man as the test bed. The specific modifications were: (i) the original maze is replaced by a simplified 16 x 16 maze and power ups are removed; (ii) two ghosts instead of four are designed and they move at the same speed as Pac-Man so that it is impossible for a single ghost to finish the task of eating Pac-Man (cooperation is required); (iii) ghosts win when they catch Pac-Man and Pac-Man must eat 45 pellets to win; if 55 steps have been finished and still neither Pac-Man nor ghosts have won, the result is considered as a draw. In the experiment, the authors controlled the ghosts with different simulation times and then collected the win rates of the ghosts. Based on the results, the authors proposed a precise regression function that could be used to dynamically match the game challenge with the ability of different types of players. However, because of the great computational intensiveness and consumption of system resources, such an ADD approach is only applicable to standalone games. An alternative approach, involving adjustments based on Artificial Neural Network (ANN) from MCTS, is proposed to realize ADD for online games [6]. The feasibility of

such an ADD approach is validated trough a study where the movements of the NPCs were monitored based on data from the simulation of the MCTS. Attributes that indicate the states of Pan-Man and the two ghosts, as well as the environment in each move are selected as inputs of the ANN for the two ghosts. Similar inputs are selected for the two ghosts except one more input (i.e., the first ghost's direction) for the second ghost so as to control cooperative behavior. The direction of each ghost is the output of the ANN.

In [13], Hocine and Gouaïch described a generic ADD approach for pointing tasks in therapeutic games. They have explained how this approach can meet therapeutic requirements such as:

(i) The ability assessment: by proposing an in-game kinematic evaluation mechanism taken from traditional rehabilitation practices,

(ii) The variability: by introducing a game abstract level which provides various ambiances and task themes with the same therapeutic objective behind,

(iii) The difficulty adjustment and continuity: by introducing a dynamic difficulty adaptation technique founded on a motivation model and the assessment of player's capabilities, which aims to overcome the playability challenge.

A Wii board based balance game is used for the proposed approach. A pilot study was conducted on healthy patients with one group using the proposed ADD

technique and the other using a random task difficulty. For the ADD approach, the player's profile containing his/her ability data as well as general information about the player such as age, gender, and whether he/she is right or left handed, is used to choose the appropriate adaptation strategy according to the proposed game goal associated with the therapeutic objective. The difficulty of a task is considered to be related with its probability of success. According to the player's profile and motivation, the difficulty adaptation module makes one of the following decisions: increase, decrease, or maintain the current difficulty level of the training session. Three criteria are taken into account when adjusting the difficulty level: the first two criteria S +local (n), S -local (n) measure the local instability of the motivation in both increase and decrease directions respectively. The third criterion measures the overall trend of motivation, T global. A least squares method is used to calculate this trend on the cumulative motivation. With these elements, the algorithm makes the following decisions:

(i) S-local (n)$\wedge$ $\neg$T global (n): decrease the difficulty. This is interpreted as a local decrease in motivation and a global trend indicating demotivation.

(ii) S +local (n)$\wedge$ T global (n): increase the difficulty. This is interpreted as a local and global increase in motivation: the patient is succeeding too easily. The difficulty is increased to keep an acceptable level of challenge.

(iii) In other cases, do not change the difficulty.

Results from statistical analysis such as the Chi-square goodness of fit test and a t-test indicate that ADD influences the player's motivation not only by challenging him/her but also by maintaining his/her success rate and influencing his/her perceived difficulty.

In [18], Hunicke and Chapman explored the computational and design requirements for an ADD system. They named the system "Hamlet" which was developed using Valve's Half Life game engine. Using techniques drawn from Inventory Theory and Operations Research, it analyzes and adjusts the supply and demand of game inventory in order to control the overall game difficulty. As the player moves throughout the game world, the system uses statistical metrics to monitor incoming game data. Over time, it estimates the player's future state from this data. When an undesirable but avoidable state is predicted, the system intervenes and adjusts the game settings as necessary. The system is designed to keep the player in the flow channel by encouraging certain states, and discouraging others. The authors proposed two types of adjustment actions:

(i) Reactive actions will adjust elements that are "in play" or "on stage" (i.e. entities that have noticed the player and are attacking). This includes directly manipulating the accuracy or damage of attacks, strength of weapons, and level of health, amongst others.

(ii) Proactive actions will adjust "off stage" elements (i.e. entities that are spawned but inactive or waiting to spawn). This includes changes to the

type, spawning order, health, accuracy, damage and packing properties of entities that have not yet appeared on screen.

In [15], Hunicke described how the “Hamlet” ADD system from [18] is used in a Half Life game engine based custom game called “Case Closed”. The author experimented with how ADD can affect player progress by manipulating the supply and demand of various items, and established a probability distribution of damage done to the player by his/her enemies during combat. Using this, the likelihood of player death is predicted in a given encounter – this helps to decide when to intervene. Intervention is conducted by a control policy, which is designed to affect the underlying game economy based on estimated player performance. Each control policy consists of an estimation algorithm, an adjustment goal, and a set of intervention strategies. A simple “comfort zone” policy is implemented, which works to keep players a relatively safe distance from death. During combat, if the estimator determines that the player’s probability of death is greater than 40%, it begins to intervene. The goal of the example policy is to keep the player’s mean health at 60, with a standard deviation of 15 points. During combat, the policy will add health in 15 point segments, at 100 clock intervals. The target is actively helping struggling players, without making the game too easy. Threshold values for adjustment, intervention increment, and lag were all set based on user testing and observation with respect to this goal. A preliminary study was conducted on an exploratory sample of 20 subjects of mixed skill (novice to expert). The results indicated that experts familiar with Half-Life and novices who rarely play shooters

both rated the game as somewhat to extremely difficult (3-5 on a scale of 1-5), however, trends indicate that expert players report elevated levels of enjoyment.

In [19], Qin et al. investigated the impacts of difficulty in video games on player immersion based on an experiment conducted with 48 participants, each playing the same experimental games with different difficulty settings. Warriors of Fate, the game used as a test bed is an English adaptation of the Japanese arcade action game Tenchi wo Kurau II. This game is a horizontal-scrolling game where common enemies keep popping up from everywhere. In each round there are 3–7 scenes. The task of each participant in the experiment was to overcome all enemies in each round. Key factors in this study were direction of difficulty changes, including three directions (up and down, down and up, and continuously increasing) and difficulty of rate changes, with three rates (slow, medium, and fast). The dependent variables were player immersion, playing time, and hit points. The player immersion was obtained through an instrument where the player scored his/her immersion in the computer game narrative on a scale of 1–7. The playing time was the total time required to overcome enemies in a round. The hit points were measured by the amount of life spent by the player-character in the experimental games to overcome all enemies in one round. The difficulty of each scene in each round of the original game is determined by the number of enemies and the length of their life bar. Then, according to the difficulty of the scenes in the original game and the requirement of the change directions and change rate of a round in the experimental games, scenes in the WOF were recomposed for the experiment. There were three experimental games each having six rounds. Each game represented a different change rate. The

six rounds in a game represented three types of change directions under an easy or hard level of game difficulty. Every participant played one experimental game. The results indicate that the players have better immersion when the difficulty changes up and down and the changes happen at a medium rate.

In [20], Missura formalized the problem of an adaptive agent in the context of the popular strategy game Connect Four. Connect Four is a game for two players each having 21 identical stones. One set of stones is white and the other is black. The game is played on a rectangular board consisting of 7 vertical columns of 6 squares each. If a stone is “dropped" in one of the columns, it will “fall down" to the lowest unoccupied square. The players make their moves in turn consisting of placing one stone in one of the columns. The goal of the game for each player is to get four of her own stones connected either vertically, or horizontally, or diagonally. If all 42 stones are placed on the board and no such group was created, the game is a draw. Missura described four Connect Four playing agents: Naive, Simple, Mini-Max, and Optimal. Then, in [21], Missura and Gartner presented “Adaptive Mini-Max” (AMM), an adaptive agent for playing Connect Four. AMM is a modified version of a pre-existent algorithm known as Mini-Max[3]. Instead of always taking the optimal move based on its investigation of the game tree, AMM first evaluates the moves that were

---

[3] Mini-Max uses a game-tree to decide which move to make on any given turn. This means that a directed tree of available actions is calculated, where the nodes are possible game states and the edges are possible moves, and the tree is investigated to determine which available move is optimal. For efficiency, Mini-Max only investigates a sub-tree of the available game tree, and then decides which move to make based on a ranking of the available moves.

available to its opponent. AMM does this by investigating a sub-tree of the game tree of choices that their opponent could have made, just as it would do for itself on its turn. The actions available to the opponent are ranked in terms of optimality, and the actual move that they made is noted. AMM does this throughout the game, and, at each turn, calculates an average of the ranking of all of the moves made by their opponent. This ranking enables AMM to determine the ability of their opponent. Then, at each of AMM's turns, AMM will choose to make not the optimal move, but rather the available move whose ranking is closest to the average ranking of the opposition's moves.

In order to evaluate AMM's adaptive mechanism, AMM was used to play against all the other Connect Four playing agents. AMM won approximately 50 percent of the time when playing against all but the optimal agent. The reason that AMM was not able to adapt to the ability of the optimal agent is that AMM is fundamentally based on Mini-Max, and so cannot perform any better than Mini-Max, which is a weaker agent than optimal. Finally, an evaluation of AMM's "fun-factor" and ability to adapt against human players was performed. The study found that users preferred playing against AMM over the non-adaptive agents. Additionally, no significant correlation between the players' skill levels calculated and their winning rate against AMM was found which means AMM successfully adapted to the players' skill levels.

Bailey and Katchabaw [7] developed an experimental test-bed based on Epic's Unreal engine that can be used to implement and study ADD in games. The core components of the test-bed is discussed below.

A) Game Engine Core: The game engine core is used to provide all of the fundamental technologies including graphics, audio, animation, artificial intelligence, networking, and physics required to drive a game or gameplay scenario. It allows developers to create new gameplay logic and content on top of this engine to have a complete game, without developing all of the underlying technologies.

B) Monitoring, Analysis, and Control: Monitoring, analysis, and control services are used in the test bed to support both ADD experimentation and software developed to implement new ADD algorithms and methodologies. These services are used by gameplay scenarios, and directly make use of the game engine core. To conduct experimentation within a particular gameplay scenario, the experimental environment must monitor and collect the appropriate player and progression data. The analysis service is used to provide support in the aggregation and correlation of data collected through monitoring. The control service is used to manipulate the experiment in the gameplay scenario, including starting, suspending, resuming, and halting a particular experiment.

C) Gameplay Scenarios: Gameplay scenarios are used to contain playable elements of games and game content. These can range in scale from mini-games depicting as few as one game activity for the player, all the way up to entire games.

They have also implemented a variety of mini-game gameplay scenarios using UnrealScript and UnrealEd for preliminary validation of the test bed. These include two jumping mini-games, a timed maze navigation mini-game, a turret mini-game requiring the player to navigate a short hallway lined with automated, indestructible gun turrets, and a fighting mini-game requiring the player to make their way through a room full of heavily armed enemy non-player characters.

Rani et al. [14] 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 signal data was collected from 15 participants each spending six hours in cognitive tasks (i.e., anagram and Pong tasks) and these were analyzed offline to train the system.

Orvis et al. [22], 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.

2.2 Software Design Patterns in Video Games

In a number of works, video games have been proposed as a tool to teach software engineering in general and design patterns in particular. On the other hand, unfortunately, work focusing on how game developers can benefit from the usage of

software design patterns is relatively rare. Here we discuss examples of both types of research.

Gestwicki and Sun [23] presented a video game based approach to teach software design patterns to computer science students. They developed an arcade style game, EEClone, which consists of six key design patterns and then used these patterns in their case study. Student participants analyzed the game to learn the usage of those patterns.

Antonio et al. [24] described their experience in teaching software design patterns using a number of incremental abstract strategy game design assignments. In their approach, each assignment was completed by refactoring and using design patterns on previous assignments.

Narsoo et al. [25] described the usage of software design patterns to implement a single player Sudoku game for the J2ME platform. They found that through the use of design patterns, new requirements could be accommodated by making changes to fewer classes than otherwise possible.

2.3 Research Gap

As we can see from the above discussion, the work on ADD in video games focuses on tool building (e.g., framework (Bailey and Katchabaw [7]), algorithms (Hunicke [15]; Hao et al. [6]) etc.) and empirical studies (e.g., Rani et al. [14]; Orvis et al. [22] etc.), but they all use an ad-hoc approach from a software design point view. On the other hand, research on using software design patterns in video games is mostly

limited to using video games as a means for teaching design patterns in undergraduate computer science courses (e.g., Gestwicki and Sun [23]; Antonio et al. [24]). In contrast, much work has been done towards game design patterns, such as the foundational work of (Björk and Holopainen [26]) and many others, but the focus there is game design and not software design, which is a subtle, yet important distinction. Thus, motivated by this research gap, in this thesis, based on empirical studies, we explore a software design pattern based approach to enable auto dynamic difficulty in video games.

Chapter 3

Research Organization

In this chapter, we discuss the overall research goals and how these goals are devised into a number of incremental studies, and provide a brief description of each study.

3.1 Research Goals

Our primary research goal is:

Research goal, ***G****: To develop a set of software design patterns, a process for applying those design patterns, a tool for using these design patterns effectively, for implementing auto dynamic difficulty in video games, and to empirically validate the overall approach.*

We decompose this high level overall research goal to following atomic sub-goals:

G1*: To develop a set of software design patterns for implementing ADD in video games.*

G2*: To validate that the proposed design patterns provide a reusable solution for implementing ADD in video games.*

***G3**: To analyze the source code reusability achieved through the usage of these design patterns to implement ADD in video games.*

***G4**: To define a concrete set of activities (possibly step-by-step) needed for applying our design pattern based approach in video games.*

***G5**: To develop a source code generation based semi-automatic framework that will assist in applying the ADD approach in video games.*

3.2 Research Studies

We have organized four different studies to achieve the above research goal. Our intention for each study is to address one or more sub-goals discussed in Section 3.1. Each study involves some development and empirical study around a specific game. Here, in Table 1, we briefly describe each of these studies:

Table 1: Decomposed executable studies from research goals

Study-1:	**Associated goals:** G1, G2 **Activities:** • Derive a set of design patterns to implement auto dynamic difficulty in video games. • Apply those design patterns in a proof-of-concept prototype Java game. **Game studied:** Pac-Man **Achievements:** • We have a set of design patterns for implementing auto dynamic difficulty in video games. • We have a Java implementation of those design patterns for a prototype game. • We have a preliminary validation of design patterns based approach for auto dynamic difficulty.

<table>
<tr><td>Study-2:</td><td>Associated goals: G2, G3
Activities:
Generalize the implementation from Study-1, so that it can be applied to other games.
Apply the generalized implementation to a third party game developed in Java with minimal modifications.
Based on the implementations from Study-1 and Study-2, measure and discuss how different software qualities (e.g., reusability, maintainability etc.) are impacted by the design pattern approach.
Game Studied: TileGame
Achievement:
We have a more generic Java implementation of the design patterns.
We have validated that the design patterns based approach for auto dynamic difficulty can easily be applied to games that were not implemented with any such prior motivation.
We, based on empirical grounds, have discussed how different software qualities are positively impacted by the usage of the proposed approach.</td></tr>
<tr><td>Study-3:</td><td>Associated goals: G2, G3, G4
Activities:
Based on the experience from Study-1 and Study-2, describe a step-by-step process to use the design patterns in a game.
Follow the described process to apply the generalized implementation to a commercial Java game with minimal modifications.
Based on the implementations from Study-1, Study-2 and Study-3, measure and discuss to what extend the implemented source code are reusable.
Game Studied: Minecraft
Achievement:
We have described a step-by-step process to apply the design patterns.
We have validated that on following the process, the design patterns based approach for auto dynamic difficulty can easily be applied to large-scale commercial game such as Minecraft.
We have further analyzed the effectiveness of the design pattern based approach by empirically investigating the reusability of the source code and the process across multiple games.</td></tr>
</table>

Study 4:	**Associated goal:** *G5* **Activities:** • Analyze the instantiation and specialization related artifacts (i.e., source code) that were identified as not reusable in prior studies. • Define a relational model to represent the dynamic information necessary to implement those artifacts. • Develop a framework, which will allow collecting required information from a game, create instance of the model based on that information, and provide an effective way for managing and fine tuning the model and finally generating source code based on the model. **Game Studied:** TileGame **Achievement:** • We have a semi-automatic tool, which with the help of code generation allows us to implement the design pattern based approach on a video game with minimum effort.
Study 5:	**Associated goal:** *G2* **Activities:** • Conduct a case study where an external developer uses our design pattern based approach to implement ADD. • Analyze the data collected from this case study to understand the ease of usage and effort associated in applying our design pattern based approach. • Identify potential issues from the critical feedback from the developer about the design patterns and/or the base level implementations provided to the developer and plan to address the issues. **Games Studied:** Tetris and Space Invaders **Achievement:** • From a preliminary user study, we have verified that a developer with no prior knowledge of our research can learn and apply our design pattern based approach to develop ADD in games with minimal effort.

Please note that the organization described in the above table is for execution purposes only and, while we discuss the results from these studies in upcoming chapters, we will not always follow this organization, and findings from different studies will be discussed together in certain chapters.

Chapter 4

Design Patterns

As we discussed earlier in Chapter 2, related work on ADD in video games has focused on tool building (e.g., framework (Bailey and Katchabaw [7]), algorithms (Hunicke [15]; Hao et al. [6]) etc.) and empirical studies (e.g., Rani et al. [14]; Orvis et al. [22] etc.), but they all use an ad-hoc approach from a software design point of view. On the other hand, research on using software design patterns in video games is mostly limited to using video games as a means for teaching design patterns in undergraduate computer science courses (e.g., Gestwicki and Sun [23]; Antonio et al. [24]). In contrast, much work has been done towards game design patterns, such as the foundational work of (Björk and Holopainen [26]) and many others, but the focus has generally been on game design and not software design, which is a subtle, yet important distinction. Relying on the success of software design patterns in different software domains, we can say that game developers could benefit from both game design patterns and software patterns for games.

Ramirez and Cheng [11] presented 12 design patterns that could assist in enabling adaptability in a software system. These design patterns were developed through

the generalization of design solutions found in the self-adaptive system literature. We found four of these 12 design patterns to be necessary for enabling ADD in video games. In this chapter, we derive these design patterns in the context of ADD in video games to achieve our first sub-goal "G1: To develop a set of software design patterns for implementing 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 [7]. In Sections 4.1, 4.2, and 4.3, we discuss monitoring, decision making, and reconfiguration patterns respectively. In Section 4.4, we discuss how these design patterns work together to enable the implementation of ADD in a game.

4.1 Monitoring Pattern

The key purpose of ADD is to provide more enjoyment to a broader demography of players. 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, amongst others.

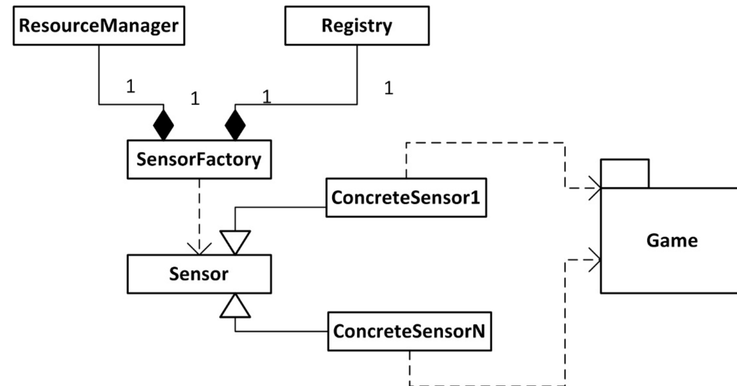


Figure 1: *Sensor factory* design pattern

Sensor factory: Sensors are objects that periodically read data from the game[4] 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. A concrete sensor may also override other attributes of the *Sensor* class. 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. An example of overriding an attribute from the base *Sensor* class can be redefining the default monitoring interval. The *SensorFactory* class uses

---

[4] 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., [10]) also suggests biological feedback can be included in this context.

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. If the *object* is not specified, then it uses the default game object. In Table 2, we provide a code snippet that demonstrates how Java reflection can be used to create a sensor without using the constructor directly. As we can see, unlike traditional implementations of the factory method pattern, this implementation does not require modification when new *ConcreteSensor* classes are created.

Table 2: Creating sensors using Java reflection

```
Class sensorClass = Class.forName(sensorName);
Constructor sensorConstructor  = sensorClass.getConstructor( new Class[]{ Object.class } );
Sensor sensor = (Sensor)sensorConstructor.newInstance( new Object[]{ object } );
```

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 (please see Table 3).

Table 3: Bypassing access modifier using Java reflection

```
Class objectClass = object.getClass();
Field field = objectClass.getDeclaredField(“fieldName”);
field.setaccessible(true);
Object fieldValue = field.get(object);
```

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.

4.2 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 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.

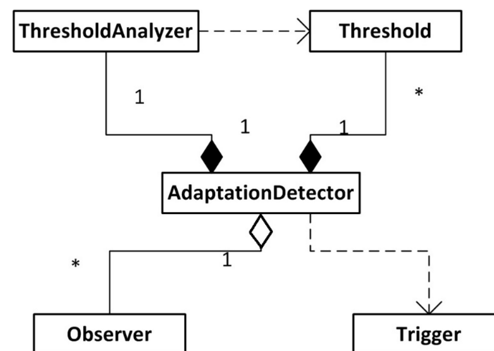


Figure 2: *Adaptation Detector* design pattern

Adaptation detector: With the help of the sensor factory pattern, the *AdaptationDetector* (please see Figure 2) deploys a number of sensors in the game

and attaches observers[5] to each sensor. *Observer* encapsulates the data collected from sensors, 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 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*.

[5] 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.

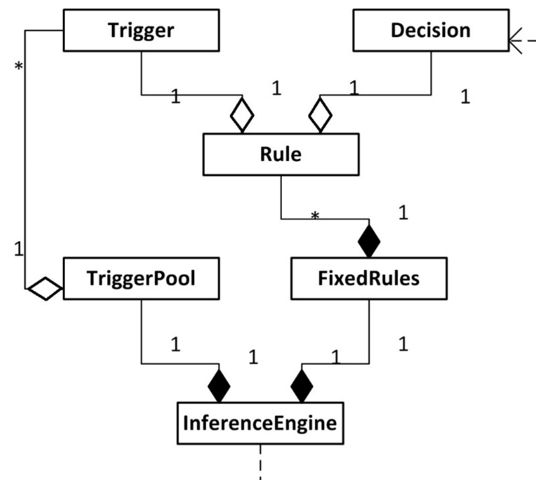


Figure 3: *Case based reasoning* design pattern

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 *Rules*[6]. 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 *Trigger*s 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*

---

[6] 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.

that appropriately responds to the *Trigger*. Note that all the attributes of two *Trigger*s need not be the same for them to match. For example, depending on the game, a "player's life value is below 20%" trigger created at two different time points might be considered the same trigger. Thus, 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*. Another optional component (not shown in Figure 3) for the case based reasoning pattern is a learner attached to the inference engine, which can learn new rules based on monitoring the sequence and effectiveness of different rule executions on the game.

4.3 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 is a runtime entity. The ADD system needs to adjust the game difficulty while the player is progressing through the game. If the adjustment is drastic, it can disturb the player's immersion. Also, there is the risk of leaving the game in an inconsistent state. Below we discuss the game reconfiguration pattern, which provides a systematic approach to reconfigure the game. Traditionally the pattern was designed for a client-server model. The reason we choose this pattern is because typically a video game is very analogous to a client-server model. In a client-server model, the 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 a loop (i.e., the game loop) for inputs from input devices (such as the keyboard, mouse, gamepad, sensors, etc.) and behaves according to those inputs.

***Game reconfiguration*:** This pattern is based on the server reconfiguration pattern described in [11]. The server reconfiguration pattern assumes that the object that needs to be configured will implement a specific interface. With the help of the adapter design pattern, this assumption can be eliminated (as we show in Figure 4 and discuss hereafter). The *AdaptationDriver* receives a *Decision* selected by the *InferenceEngine* (please see case based reasoning in Section 4.2) 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 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.

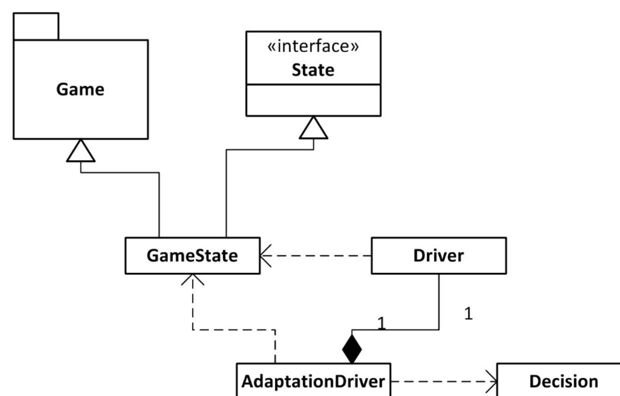


Figure 4: *Game reconfiguration* design pattern

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[7]) 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. When the game is in an inactive state, it will not be able to respond to the inputs it receives from the player through the input devices, but it should not discard those requests either because that might expose an unexpected behavior to the player. 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. When the *GameState* is requested to be *INACTIVE*, it is transferred to *BEING_INACTIVE*. While in the *BEING_INACTIVE* state, the game-loop finishes its current execution and then goes to the *INACTIVE* state. In the *INACTIVE* state, the game-loop does not get executed. If the game is not in the *ACTIVE* state, inputs are stored in the *RequestBuffer* instead of being processed. When the game is requested to be *ACTIVE*, it is transferred to the *BEING_ACTIVE* state first. In the *BEING_ACTIVE* state, the inputs stored in the *RequestBuffer* are retrieved and processed. The game goes to the *ACTIVE* state from the *BEING_ACTIVE* state only after the *RequestBuffer* becomes empty. The game can

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

be requested to go to the *INACTIVE* state only at a time when it is in the *ACTIVE* state, and vice versa. It is important to note that in a reasonable implementation, all these changes 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.

4.4 Integration of Patterns

In this Section, we briefly re-discuss how the four design patterns discussed in Sections 4.1, 4.2, and 4.3 work together to create a complete ADD system (please see Figure 5 and Table 4).

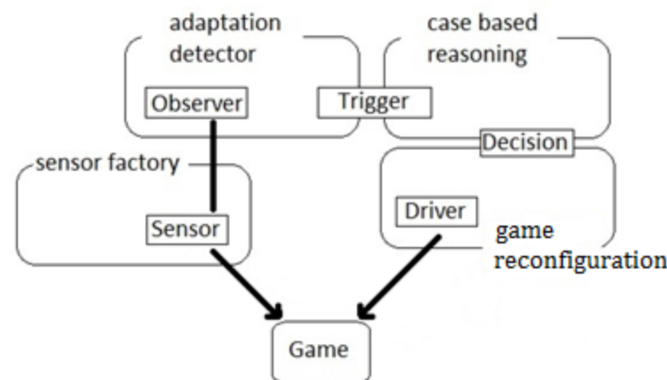


Figure 5: ADD design patterns working together

Table 4: Summary of ADD design patterns

Design Pattern	Role	Interacts With
Sensor factory	Collecting data from the game	Game, Adaptation detector
Adaptation detector	Deciding when to adjust the game	Sensor factory, Case based reasoning
Case based reasoning	Deciding what to adjust in the game and how to adjust	Adaptation detector, Game reconfiguration
Game reconfiguration	Implementing the adjustment	Case based reasoning, Game

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

Chapter 5

Games Studied

To date, we have used five games developed in Java for studying the design patterns described in Chapter 4. In our early work (please see studies 1 and 2; also reported in[12] and[27]), 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 [28]. Even though we were successful in using the design pattern based 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, in a later study (please see Study 3; also reported in[29]) we have selected a commercially successful sandbox game – Minecraft[8] [30] to extend our study. Also, we designed a class project, where a student used our designed pattern based approach to implement ADD in open source variants of two popular arcade games: Space Invaders and Tetris. In sections 5.1 to 5.6 below we

[8] Minecraft is commercially available for several platforms, but we focus on the desktop version developed in Java.

briefly describe each of the games and examples of adaptations that were implemented. In sub-section 5.7, we discuss the second sub-goal “G2: To validate that the proposed design patterns provides a reusable solution for implementing ADD in video games”.

5.1 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 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.

Figure 6: Screen captured from the Pac-Man game

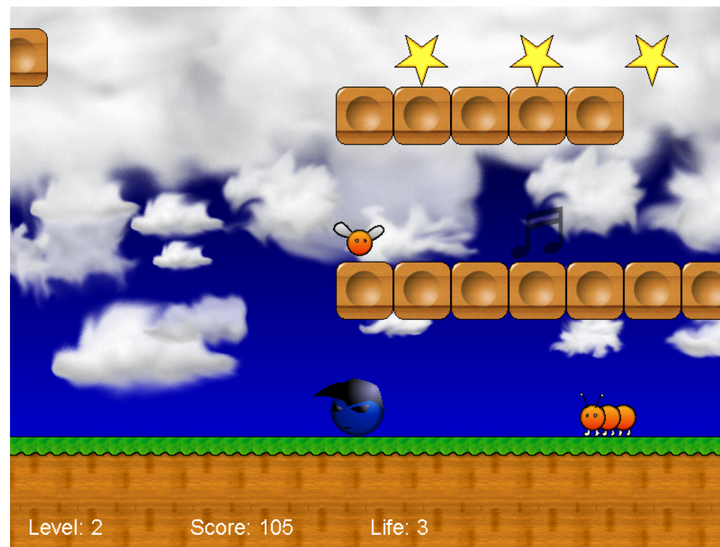


Figure 7: Screen captured from the TileGame game

5.2 TileGame

The level structure and game-play 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. 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 six lives. When the player character dies, it loses one life and the game restarts from the beginning of that level. The player character and ants are affected by gravity; flies are only affected by gravity when they die. 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.

5.3 Minecraft

Minecraft [30] 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 (e.g., dirt, stone, tree trunks, water, etc.) 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 (e.g., furnaces, bricks, stairs, etc.) and items (e.g., sticks, axes, buckets, etc.). Players can place collected or crafted blocks and items elsewhere to build structures. The world is divided into biomes (e.g., deserts, jungles, snow fields, etc.). The time in the game goes through a day-night cycle every 20 real time minutes. There are various NPCs known as mobs (e.g., animals, villagers, hostile creatures, etc.). Non hostile animals (e.g., cows, pigs, chickens, etc.) spawn during the daytime and can be hunted for food and crafting materials. Hostile mobs (e.g., spiders, zombies, creepers (a Minecraft-unique creature), etc.) 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 (please see Figure 8) played in the survival mode of the game.

Figure 8: Screen captured from the Minecraft game

While Minecraft is not open-source, its source code can be readily obtained through the use of a toolchain [31] provided by an active and extensive developer community that decompiles the game back to its source code. This practice is accepted by the creators of Minecraft while an official modding interface is under development.

5.4 Space Invaders

Space Invaders is a two dimensional fixed shooter game[9]. In this game, the player controls a canon by moving it horizontally across the bottom of the screen and firing at invader alien ships descending from top of the screen. In the used variant, there are 24 alien ships organized in 4 rows (please see Figure 9). The player can shoot

[9] In fixed shooter games, (i) the level fits within a single screen, (ii) the protagonist's movement is fixed to a single axis of motion, and (iii) enemies attack in a single direction (such as descending from the top of the screen).

one missile at a time and he can only shoot the next one when the previous one hits an alien ship or the top of the screen. Each alien ship can randomly drop one bomb at a time until it is destroyed. It can only drop the next bomb when the previous bomb hits the player's canon or the ground. The player starts with 5 lives and each time a bomb touches the player's canon, one life gets decreased. To win the game, the player needs to destroy all the alien ships before losing all of his/her lives and the alien ships reach the ground.

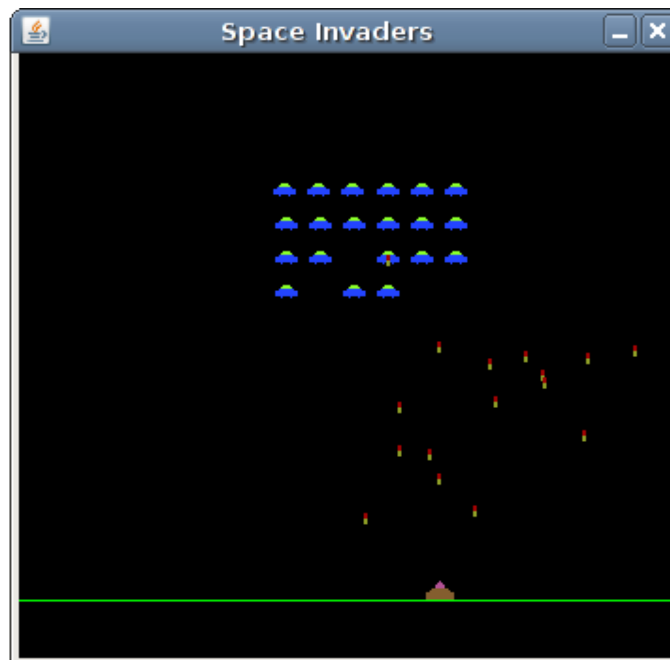


Figure 9: Screen captured from the Space Invaders game

5.5 Tetris

Tetris is a falling block puzzle game in which there are 7 different shapes (i.e., I, J, L, O, S, T and Z shapes – please see in Figure 10) called Tetriminos. Tetriminos are game pieces shaped like tetrominoes, geometric shapes composed of four square blocks each. A random sequence of Tetriminos fall down the playing field from the

top of the screen. A player can control these shapes by moving them sideways or rotating them at 90 degree units, with the intention of creating horizontal lines of blocks without any gaps. Such lines disappear immediately as they form and all the blocks above that line fall by one line and the player earns points. The game continues until the stack of block reaches the top of the screen such that no new Tetriminos can enter.

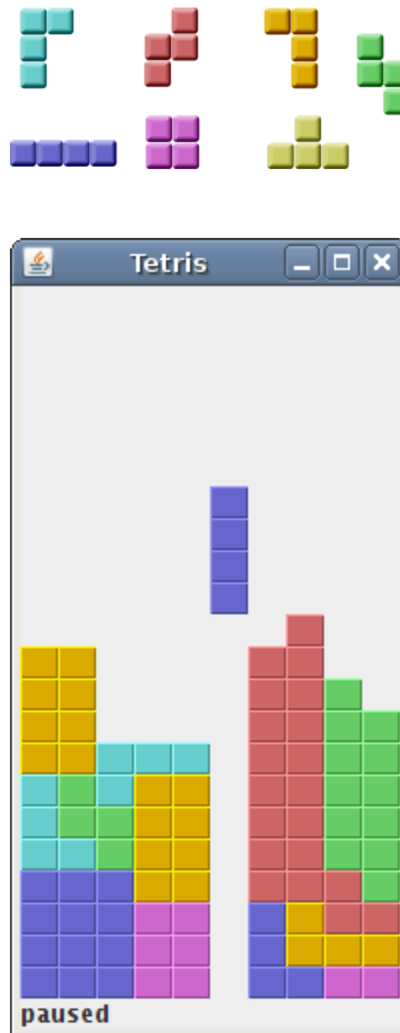


Figure 10: 7 Tetriminos (top) and Screen captured from the Tetris game (bottom)

5.6 Adaptations Implemented

In Table 5, we give examples[10] of different adaptations that were implemented in these 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 a possible way of eliciting essential requirements for an adaptive software [32].

Table 5: Examples of adaptation 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 etc.	Modify ghost's speed, duration of Pac-Man's predator mode etc. based on how the average score per life compares 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 (i.e., when the player character dies) based on scores and life 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 players 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.

---

[10] Here, we discuss one or more non-trivial examples from each of the games. Few more scenarios will be discussed in Chapter 6. Other trivial ones were intentionally left out, as they do not provide any additional value to this discussion.

Game	Metrics for Sensors	Attributes for Modification	Adaptation Scenarios
Tetris	Average number of shapes falling between two rows being cleared	Relative frequency ratio between desirable and undesirable shapes	Give undesirable (please see section 8.3 for details of the classification of the shapes) shapes to the player when he/she is clearing rows quickly and give desirable shapes for the opposite.
	Height of the stack	Speed of the shapes	Descend the shapes faster if the stack is not very high and decrease the speed if the stack is high.
Space Invaders	Alien ships' height from the ground, number of alien ships remaining	Alien ships' speed towards ground	Gradually increase or decrease the alien ships' speed and player's missile's speed based on the remaining size of the alien force and their distance from the ground.
		Speed of player's missile	

5.7 Reusable Solution across Multiple Games

Design patterns are a general reusable solution for commonly occurring problems. Typically, design patterns are elicited by analyzing implemented solutions across multiple systems rather than being designed and thus their reusability as a solution does not need to be demonstrated. However, this general approach of eliciting design pattern is not applicable for our specific problem. Popular games such as "Max Payne", "Half-Life 2" and "God Hand" use the concept of auto dynamic difficulty. How ADD is delivered in these games from a gameplay perspective can only be discerned through reviewing these games or from official strategy guides (or, occasionally in presentations such as [9]). Unfortunately, given the highly competitive nature of the games industry, no information is publicly available about

how ADD is implemented in these games from a software design perspective. There are no adequate open source examples of auto dynamic difficulty implementations to be analyzed. Thus, we have derived the necessary design patterns from the self-adaptive system literature in the context of ADD in video games (please see Chapter 4). In this chapter, we discussed five different games where the design pattern based approach was used to implement ADD. One of the games (i.e., Minecraft) among them is a highly successful sandbox game. Most adaptations that were implemented primarily focus on modifying attributes of the game (please see Pac-Man and Minecraft examples in Table 5) whereas others focus on content modifications (please see TileGame example of usage of different version of maps in Table 5). Thus, in this chapter, through empirical evidence (i.e., the usage of the design patterns to implement ADD in 5 different games), we have addressed our second sub-goal "G2: To validate that the proposed design patterns provides a reusable solution for implementing ADD in video games".

Chapter 6

Source Code and Process Reusability

In [27], we examined, based on a case study involving Pac-Man and TileGame, how the use of our design patterns as discussed in Chapter 4 impacted different software qualities of a game. One of the findings of that study was that, for small games such as Pac-Man and TileGame, using these design patterns to develop ADD may result in more than 75% source code reusability. In this chapter, we want to examine whether our design pattern approach can be applied to a large commercial game such as Minecraft and to what extent the reusability quality of these patterns remain valid (i.e., our third sub-goal "G3: To analyze the source code reusability achieved through the usage of these design patterns to implement ADD in video games").

In Section 6.1 below, we describe the process of using our design patterns approach to develop an ADD system including examples from our work and existing literature (i.e., our fourth sub-goal "G4: To define a concrete set of activities (possibly step-by-step) needed for applying our design pattern based approach in video games"). The process was developed to formalize our experiences from [27] to assist in the ADD-enablement of larger games like Minecraft. By taking a step-by-step methodological

approach, a seemingly monumental task was accomplished without difficulty. A well-defined process such as this is also important for industrial adoption for several reasons such as measuring progress, planning, and automation.

Following this process, we then carried out a source code reusability analysis on these games using four metrics: Number Of Methods (NOM [33]), Weighted Methods per Class (WMC [34]), Coupling Between Objects (CBO [34]) and amount of reuse [35]. These metrics are taken from the software metrics literature (e.g., [33], [34]) and are frequently used to analyze the reusability of source code. The first three metrics can be applied to an individual piece of software and were applied to the source code of the Minecraft ADD implementation constructed here. The amount of reuse metric requires comparing multiple pieces of software to each other and, in this case, the Minecraft ADD implementation was compared to the ADD implementations of Pac-Man and TileGame. Logical Source Lines of Code (SLOC) was used for this measurement. We used three different tools for collecting these metrics. The Eclipse plugin Metrics [36] was used for calculating NOM and WMC. The Understand [37] and Unified Code Count (UCC [35]) tools were used for calculating CBO and amount of reuse respectively. In Section 6.2, we discuss the results from this analysis.

6.1 Process

With our design pattern based architecture in hand, we can essentially follow a step-by-step process to develop the rest of the system. In this section, we describe that process.

1) Define Sensors: Identify metrics to assess the skill of the player and the perceived level of difficulty based on failure and success rates. Examples of data to be collected for this purpose may include the player's score, player's life level, time spent on activities, inventory, number of enemies killed, and so on. There can be reactive and proactive metrics. Reactive metrics measure a player's performance based on success or failure on a particular activity. For example, for the Pac-Man game, we used an average-score-per-life sensor. On the other hand, for Minecraft, we have created sensors to monitor a player's inventory and current time of the world, which can be proactively used to predict whether the player will have enough resources to build a shelter before nightfall. These metrics can be identified intuitively (e.g., level completion time), as a design artifact of game play (e.g., amount of life remaining), or as described in specific algorithm or technique (e.g., average win rate of ghosts in Pac-Man [6]). Furthermore, any analysis method such as plotting various attributes over time, using a debug mode, or analyzing log files can be helpful for identifying these metrics.

2) Identify attributes to modify game difficulty: Identify attributes of the game that can be adjusted to modify the level of difficulty of the game. Here we provide examples of such attributes:

a) Player character attributes: For example, the durability of items and the amount of damage the player experiences from hostile mobs' attacks in Minecraft, or the duration that Pac-Man's predator mode can be increased or decreased to modify the level of difficulty.

b) Non-Player character attributes: For example, in the Pac-Man game, the attributes of ghost speed, ghost vision length, and the amount of time that a ghost stays in the centre of the maze after being eaten by Pac-Man in predator mode can be increased or decreased to change the game difficulty.

c) Game world and level attributes: For example, in the TileGame game, loading different versions of the map can be used to modify game difficulty. For procedurally generated levels, either unexplored parts of the world can be generated to match player expertise (e.g., [38]) or attributes of already generated game world objects can be adjusted. For example, in Minecraft, the hardness of a particular type of block can be modified within a believable range to modify the difficulty of gathering that particular resource.

d) Puzzle attributes: For example, in Minecraft, if the player fails to build a shelter in the first few nights, hints can be provided when daytime is drawing to a close.

The techniques described in Step 1 can be used to identify these attributes as well.

3) Identify adaptation scenarios: Identify game adaptation scenarios involving metrics and attributes identified from Step 1 and Step 2. Please note that this is more of a game design activity than a software design activity, as the focus is on adjusting elements of gameplay to optimize player experience. Thus, existing literature on game design (e.g., [8]) can provide great insight for this step.

4) Define observers and thresholds: Define thresholds based on the scenarios identified in Step 3 for the sensors defined in Step 1, resolve any boundary value problems raised by the threshold definitions, and define observers to relate thresholds to sensors. Analysis techniques described in Step 1 can be used to find appropriate threshold values. Also, user trials can be useful here.

5) Define triggers and adaptation detectors: Define triggers to represent each scenario, including any necessary contextual information with the trigger (for example, in the TileGame game, a trigger representing game-world-too-easy may include map difficulty and speed of NPCs), and develop the adaptation detector logic based on the scenarios.

6) Define decisions: Use attributes identified in Step 2 to create decisions to modify game difficulty according to the scenarios identified in Step 3. Please note that, existing literature on game difficulty can useful here. For example, Bostan and Öğüt [39], based on lessons learned from a number of role playing games, suggested using a convex-shaped difficulty curve. Similarly, Qin et al. [19], suggested up and down directions and a medium rate of difficulty change based on an experiment involving 48 participants using Warriors of Fate, an action game.

7) Define rules: Define rules to relate triggers to decisions based on the adaptation scenarios. It is important to analyze any dependency between rules and take actions if there are any contradictions. For example, two rules should not be each other's preconditions. Techniques for analyzing correlations between two software artifacts, such as a traceability matrix, can be useful here.

In Table 6, we show examples of artifacts produced during the first three steps of the process described above, when applied to Minecraft. Other artifacts from the process are very much code specific and are difficult to describe here. We present a source code analysis of all the artifacts in the next section.

Table 6: Example of artifacts produced through the ADD process activities

Metrics for Sensors (Step 1)	**Attributes for Modification (Step 2)**	**Adaptation Scenario (Step 3)**
Player's experience points (i.e., *EntityPlayer.experienceTotal)*	Built in game difficulty settings (i.e., *GameSettings.difficulty*)	Modify the built in game difficulty settings as the player earns or loses experience points to make the game easier or harder.
Number of items of particular materials in player's inventory (i.e., *InventoryPlayer.mainInventory*)	Hardness of those particular items (i.e., *Block.blockHardness*)	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.
Number of items in player's inventory, which day in game, time of the day (i.e., *WorldInfo. worldTime*)	Pace of time in the game (i.e., Timer.timerSpeed)	Slightly modify the passage of time during the afternoon of the first day based on whether players have collected enough resources to build a shelter for the night. Players are given more time to make the game easier and less to make it harder.
Number of times player dies (ie., EntityLiving.deathTime), which day in game	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.

Player's food level (i.e., *FoodStats.foodLevel*)	Maximum health of animals (e.g., *EntityChicken.getMaxHealth()*), fleeing attribute of animals (i.e., *EntityAnimal. fleeingTick*) when attacked (i.e., *EntityAnimal. attackEntityFrom()*), number of items dropped when dies (e.g., *EntityCow.dropFewItems()*)	If they player is facing continuous low food level, food collection can be made easier by modifying number of attacks required to kill an animal or modifying flee behavior of an animal when attacked or the number of items dropped when killed.

6.2 Source Code

In Table 7, we show a reusability analysis of the source code of the ADD system that we have developed for Minecraft. In the first column, we show the class name or pattern name. In the second column we show the number of classes in each category (i.e., specified in column 1). In the next three columns we show the corresponding NOM, WMC and CBO values. In the sixth column we show the total logical SLOC in the ADD system for Minecraft. In the seventh column we show the reused Logical SLOC (i.e., those lines that remained unchanged from the Pac-Man and TileGame games) and the associated percentage. In the last column we show the game-specific Logical SLOC (i.e., specific to ADD system for Minecraft and cannot be directly reused) and the associated percentage. For clarity, we combined certain rows of 100% reused classes within a particular pattern. In those cases, the maximum values of NOM, WMC and CBO were reported because the thresholds for these metrics are defined as upper bounds (please see the discussion below). After all of the rows of a particular class or pattern, we present a summary. The last row of the table is a summary across all the classes and patterns.

1) Number of Methods (NOM): NOM is simply a count of the number of methods in a class, with 20 and 40 being the preferred and acceptable thresholds respectively [33]. We can see from the third column in

Table 7, that the maximum number of methods in a class from our implementation is 10.

2) Weighted Methods per Class (WMC): WMC [34] is a weighted sum based on complexity[11] of each of the methods in a class and is defined as:

$$WMC = \sum_{i=1}^{n} C_i$$

Where n is the number of methods and *Ci* is the complexity of method *i*. The preferred and acceptable thresholds for these metrics are defined as 25 and 40 respectively [33]. We can see from fourth column in Table 7 that all the classes are within the acceptable thresholds and only two classes (i.e., Registry and Game State) are above the preferred threshold.

3) Coupling between Objects (CBO): CBO [34] is the measure of number of classes to which a class is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. We can see from fifth column in Table 7 that CBO of only two classes (i.e., Adaptation Detector and Inference Engine) are above the preferred threshold of 5 [33].

---

[11] Cyclomatic complexity is a software metrics used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent path through a program's source code.

4) Amount of Reuse: We can see from Table 7 that SensorFactory, Sensor, Registry and ResourceManager classes in the sensor factory design pattern were completely reused across all three games. Similarly, classes for the Observer, Trigger, Threshold and ThresholdAnalyzer in the adaptation detector pattern were completely reused. Three classes (i.e., Rule, FixedRules and Decision) in the case based reasoning pattern, and three classes (i.e., Driver, AdaptationDriver and State) in the game reconfiguration pattern were also completely reused. Furthermore, the classes required to implement AdaptationDetector, InferenceEngine and GameState were partially reused. Only the concrete sensors (seven classes) and the concrete decisions (2 classes) were very specific to the game and could not be reused.

As we discussed earlier, only two classes have WMC values above the preferred threshold and only two classes have CBO values above the preferred threshold. This is indicative of high source code reusability potential. For amount of reuse, we can see from the last row in Table 7, the ADD system for Minecraft contains 28 classes comprised of 808 logical SLOC. Among these 808 logical SLOC, 600 logical SLOC (74.26%)[12] are exactly the same as Pac-Man and TileGame and thus are considered reusable. Only 208 (25.74%) logical SLOC are specific to the game.

---

[12] These 600 lines were separated as a base level implementation and were given to a voluntary participant for a preliminary user study. The participant acknowledged that he managed to use this source code for creating ADD in two other video games with very minimal changes. A detailed source code analysis was out of the scope of that study. Please see Chapter 8 for more details about the study.

Table 7: Source code analysis of ADD design pattern implementation

Class/ Pattern Name	# of Classes	NOM	WMC	CBO	Logical SLOC		
					Total	Reusable (%)	Specific (%)
SensorFactory, Sensor, Resource Manager	3	9	15	3	145	145(100)	0(0)
Registry	1	10	27	2	73	73(100)	0(0)
ConcreteSensors	7	4	10	1	64	0(0)	64(100)
Sensor Factory	**11**				**282**	**218(77.3)**	**64(22.7)**
Observer, Trigger, Threshold, Threshold Analyzer	5	8	10	2	97	97(100)	0(0)
AdaptationDetector	1	4	20	8	91	21(23.08)	70(76.92)
Adaptation Detector	**6**				**188**	**118(62.8)**	**70(37.23)**
Rule, Fixed Rules, Decisions	3	10	10	3	75	75(100)	0(0)
InferenceEngine	2	4	7	7	57	46(80.7)	11(19.3)
ConcreteDecisions	2	2	2	0	22	0(0)	22(100)
Case-based Reasoning	**7**				**154**	**121(78.57)**	**33(21.43)**
Driver, Adaptation Driver, State	3	4	22	3	99	99(100)	0(0)
GameState	1	10	27	1	85	44(51.8)	41(48.2)
Game Reconfiguration	**4**				**184**	**143(77.7)**	**41(22.3)**
Grand Total	**28**				**808**	**600(74.26)**	**208(25.74)**

Overall, more than 70% of the logical SLOC required to implement the ADD systems are considered reusable. Previously, in the Pac-Man and TileGame games we experienced 77.52% and 79.68% code reusability [27], and so our findings with Minecraft are reasonably consistent with our prior experience. Considering that Minecraft is significantly larger and more complex than either Pac-Man or TileGame, this further strengthens our confidence in the reusability benefits of our approach to ADD, and demonstrates significant potential for commercial applications.

6.3 Summary

In this chapter, we described a step-by-step process for using our design pattern based approach to develop an ADD system including examples from our work and existing literature. Following the process, we then carried out a source code reusability analysis using four metrics taken from the software metrics literature that are frequently used to analyze the reusability of source code. The results indicated that using these design patterns to develop ADD should result in a high degree of source code reusability. A repeatable process and source code reusability provide clear motivation for adopting our design pattern based approach to creating ADD in video games.

Chapter 7

Automation Framework

We have enjoyed success in our initial works (i.e., Pac-Man [12] and Tilegame [27]) in enabling ADD in simple, small, proof-of-concept casual games. In these cases, however, the code was either originally written by us or well documented and simple enough to be easily understood and reshaped accordingly. Applying our software design pattern based framework for ADD to a large commercial-scale game such as Minecraft [30], on the other hand, seemed to be a daunting task, at least on the surface. Thus, the process described in Chapter 6 was developed to formalize our experiences from using them in Pac-Man and TileGame to assist in the ADD-enablement of larger games such as Minecraft. In practice, we found that applying such a methodical process enabled ADD in Minecraft quite readily, and that our framework 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 section 6.1) are easier to build a tool upon.

We have also carried out a source code analysis of these games. In Section 6.2 (also reported in [29]), the Minecraft ADD implementation was compared to the ADD implementations of Pac-Man and TileGame. During this analysis, we have noticed that a large fraction of the resultant code is generalization and instantiation of other high level classes (e.g., Sensors, Triggers, Thresholds, and Decisions etc.). The following table is a summary of the analysis derived from the results presented in section 6.2. 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 codes of the ADD system are similar looking classes and statements. This result motivates us to create a tool which will allow us to develop and maintain these artifacts in a semi-automatic manner (i.e., our fifth sub-goal "G5: To develop a source code generation based semi-automatic framework that will assist in applying the ADD approach in video games").

Table 8: Categorization of the ADD 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**	

7.1 Automation Framework

Figure 11 depicts a high level decomposition of our semi-automatic system. The key idea is to represent part of the ADD logic as a relational model which 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.

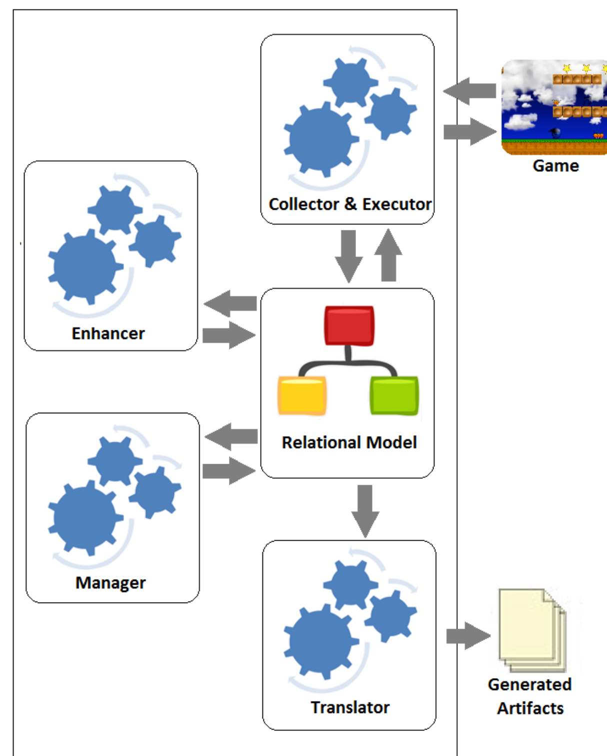


Figure 11: Components of the semi-automatic framework

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 a storage for a set of objects and relations which represent much of the dynamic information (e.g., Sensor's name, relations between sensors and attributes, etc.) for an intended ADD 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.

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 model and the collector can collect more information based on those modifications.

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.

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 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 [40], Southey et al. described an active learning based semi-automatic gameplay analysis tool. The tool is highly platform and game independent and 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 FIFA'99).

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.

7.2 Proof-of-concept Prototype

We have developed a web-based proof-of-concept prototype as an instance of the semi-automatic framework described in Section 7.1. In this section, we briefly describe how each component of the framework was instantiated in the prototype.

Relational Model: The relational model was realized using a MySQL[41] relational database. We have also created a REST API using PHP[42] to read and write on this database. All of the other components in the prototype interact with the database through this API. In Figure 12 we show the schema of the database. In Table 9, we show how different components of the framework interact with each table. As we can see, the *sessions* and *session_attributes* tables are for recording log information. Information in these tables are written by the Collector and read by Enhancer module for analyzing data. Information from all the other tables get translated to source code in some form. We will discuss these interactions in more details in the sections below.

Table 9 : Interaction between each tables and other framework components

Tables	Written By	Read By
attributes	Collector, Manager	Enhancer, Manager, Translator
sessions_attributes	Collector	Enhancer
Sessions	Collector	Enhancer
sensors, sensors_attributes, observers, observers_sensors, thresholds, observers_thresholds, triggers, rules, decisions, decisions_attributes	Manager	Manager, Translator

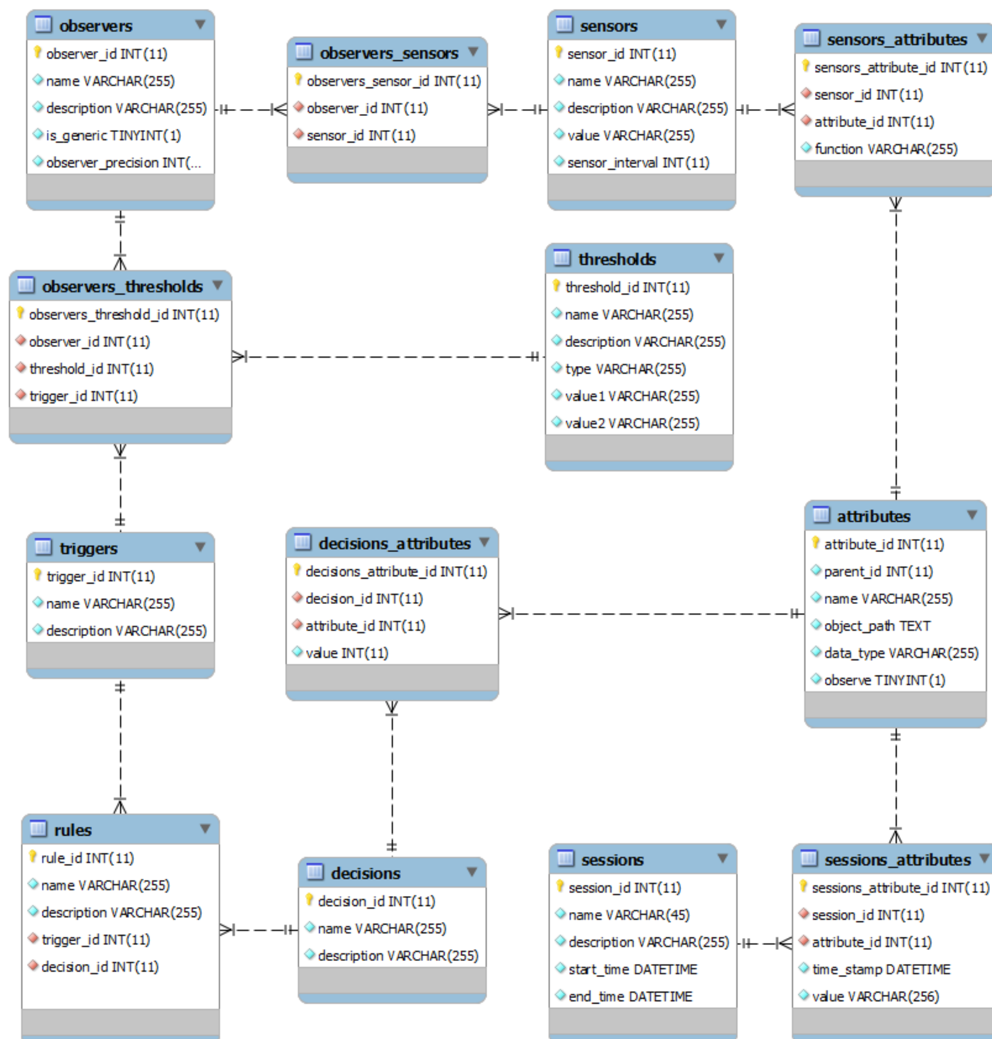


Figure 12: Schema of the MySQL database for the relational model

Collector: We have created a collector in Java for interacting with games implemented in Java. It has two sub components named *ObjectInformationCollector* and *RunTimeInformationCollector*. Given a base level object and a maximum depth, the *ObjectInformationCollector* recursively inspects all of the attributes of that object until it reaches to the maximum depth or finds primitive attributes (e.g., Integer, Boolean, etc.). While traversing, it records each attribute's name, parent, data type, and object path (i.e., a dotted notation to reach from the base level object) in the *attributes* table. Given a set of attributes to monitor and frequency of monitoring, the *RunTimeInformationCollector* creates a session (i.e., an entry in the *sessions* table with a start time), monitors the change of values of those attributes, records them with time stamps, and associates them to the session (in the *sessions_attributes* table) while the game is being played. At the end of the session, the end time is also recorded. Please note that in our current implementation of the prototype there is no Executor component. The task of Executor is achieved through repeated deployment of the generated source code and further monitoring.

Manager: We have created the manager component using the ajaxCRUD [43] library which allows faster user interface creation using PHP [42] and Javascript [44] for CRUD (i.e., Create, Read, Update, and Delete) operations on a MySQL [41] database. Once the attributes are recorded by the Collector component, we can mark them to be monitored using the *observe* flag on the *attributes* table using this component. It also allows all the required use cases for manipulating the relational model. Below we discuss one example. For an extensive list of use cases, please see the user manual in Appendix B.

The Manager facilitates creating a sensor, defining the frequency of monitoring (i.e., *sensor_interval*) for that sensor, and defining the function for calculating the value of the sensor (i.e., *value*). It also allows associating multiple attributes to a sensor. There are some built in functions that can be applied to these associations. For example, in the Pac-Man game, there is an array *ghost_speed[]* and a variable *pacman_speed* to hold the ghosts' and pac-man's speed respectively. If we are creating a sensor *PacManSufficientSpeed* to know whether the pac-man's speed is more than all the ghosts' speed or not, we will associate the *ghost_speed[]* and *pacman_speed* to the sensor using a MAX function (i.e., to calculate the maximum of a Collection) and no function respectively. In doing so, the value in the sensor should be pacman_speed > max_ghost_speed.

Enhancer: For Enhancer, we have created two visualizations for visualizing the data collected by the Collector component. We used the Data Driven Documents [45] visualization library also known as d3js [46] created by the Stanford Visualization Group. We briefly discuss each of the visualizations below.

1. **Attribute Tree Visualization:** In Figure 13**,** we show the number of attributes at different depths[13] of the Tetris game collected by the Collector component. As we can see from the figure, the number grows very quickly, which makes it very difficult to locate an attribute from the list of all attributes to mark it for

[13] Depth refers to the distance of an attribute from the root level object (i.e., the object from which we start the attribute discovery). So, the root level object will be considered at depth zero, any attributes of the root level object will be considered at depth one and their attributes will be considered at depth two and so on.

observing or association to other entities such as sensors or decisions. Thus, we have created this visualization where the attribute hierarchy is represented in a tree structure where nodes with children attributes can be expanded or collapsed.

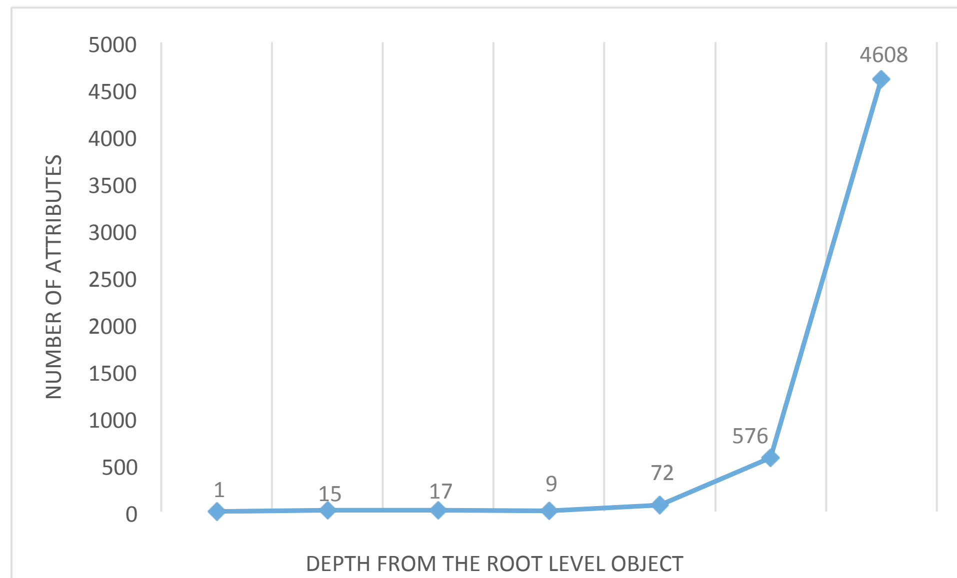


Figure 13 : Number of attributes at different depths on the Tetris game

In Figure 14, we show screenshot of the attribute tree visualization where all the attributes up to depth four are expanded for the Tetris game.

2. **Session Timeline Visualization:** After we collect a list of attributes from the game using the Collector, we intuitively select some attributes for monitoring. Our intention is to use some of these attributes as sensors (to understand the level of difficulty that the player is facing), and then use the Collector again to monitor their value changes during a session. Now, from the raw collection of data, it is very difficult to understand whether our selection is useful or not. Thus, we have created another visualization (please see Figure 15) where value changes for multiple attributes in one session, or one attribute in multiple sessions, can be seen as line charts in a time line.

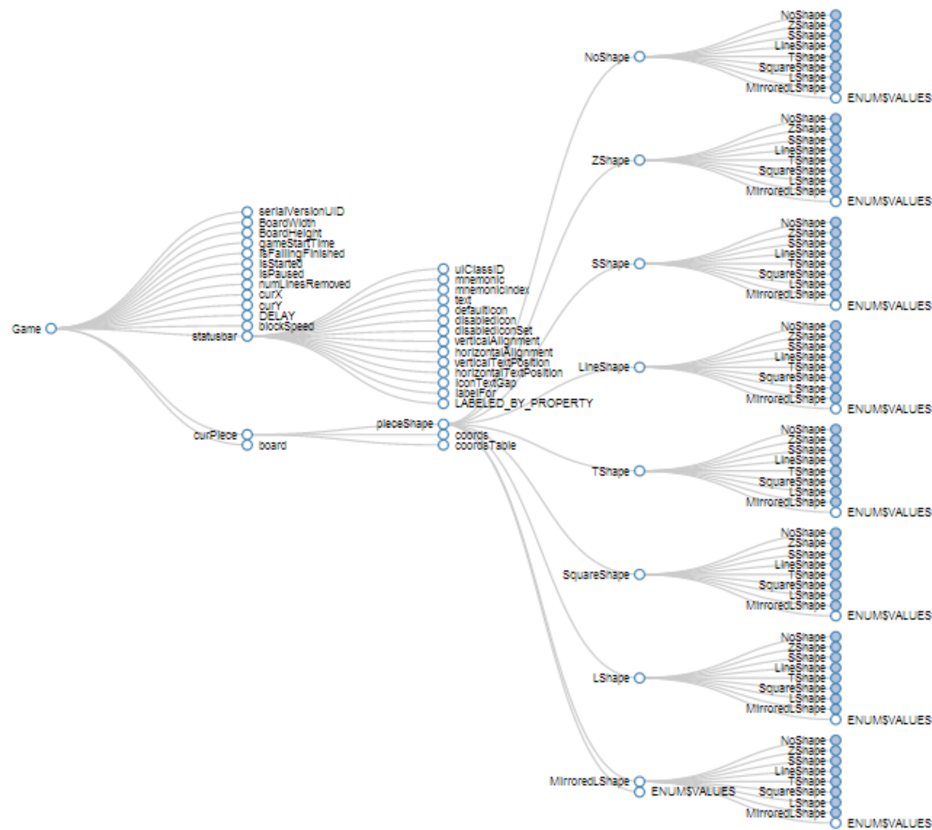


Figure 14 : Screenshot of attribute tree visualization for the Tetris game

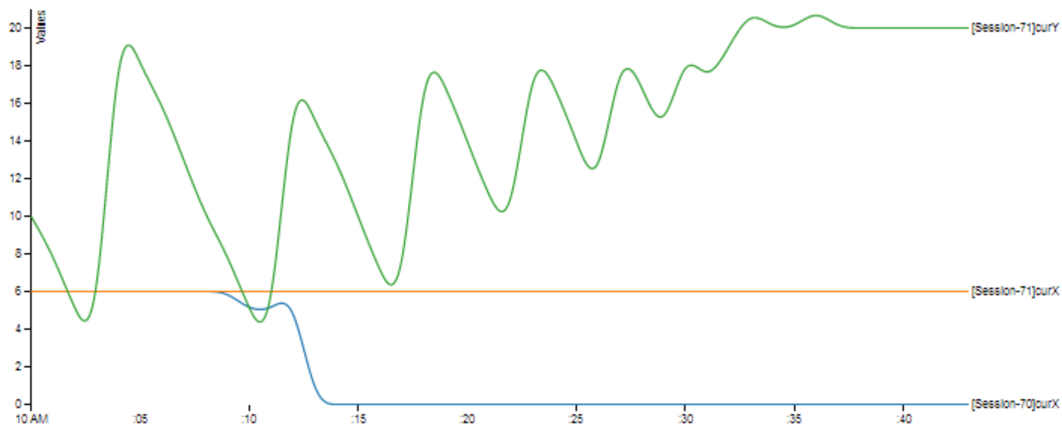


Figure 15 : Screenshot of a sample session timeline visualization

Translator: We have created the Translator component using PHP. It interacts with the REST API to fetch the required data from the MySQL database and then generates corresponding Java source code. The code generation logic is often quite simple. For each Java class, we predefine the static parts of the code and the Translator injects the dynamic parts as necessary. In Table 10, we show the pseudo code for generating a sensor class in Java. In lines 1 to 9, we print the Java class and constructor definition. In lines 8 and 9, we print the override for the refreshValue method (the parent Sensor class periodically calls this method to get the updated value) and the exception-handling block for accessing different attribute values. If there are some attributes attached to the sensor (line 10), in lines 11 to 13, we print the declaration for accessing those attributes. In lines 14 to 31, we print the logic for calculating any functions attached to the attribute such as MAX, MIN, AVG and so on. In lines 32 and 33, we print the overall value calculation for the sensor. The rest of the lines are for ending the exception-handing block, and method and class declaration. Please see Appendix C for the actual PHP code of all the sub components of Translator.

Table 10 : Pseudo code for generating sensor class in Java

Executed PHP Code	Printed Java Code	Injected Data Value

```
1. public class <sensor_name> extends Sensor{

2.      public <sensor[name]>(Object object){
3.           this.object = object;
4.           this.fieldName = "<sensor[name]>";
5.           this.setInterval(<sensor[interval]>);
6.           this.setValue(0);
7.      }

8.      public void refreshValue(){//Java method declaration starts
```

```
9.       try{  // Java try block starts
10.      if(sensor[attributes]!=""){ // PHP external if block starts
11.      foreach(sensor[attributes] as attribute){ // PHP foreach block-1 starts
12.      <attribute[data_type]> <attribute[name]> = <attribute[attribute_path]>;
13.      } // PHP foreach block-1 ends

14.      foreach(sensor[attributes] as attribute){ // PHP foreach block-2 starts
15.               if(attribute[function]!="NONE"){ //PHP internal if block starts
16.      <attribute[element_data_type]> <attribute[name]><attribute[function]> = 0;

17.      for(int i = 0; i < <attribute[name]>.length; i++){ // Java for loop starts
18.               if(attribute[function]=="SUM" || attribute[function]=="AVG"){
19.      <attribute[name]><attribute[function]> = <attribute[name]><attribute[function]> +
<attribute[name]>[i];
20.               }
21.               elseif(attribute[function]=="MAX"){
22.      <attribute[name]><attribute[function]> =
Math.max(<attribute[name]><attribute[function]> , <attribute[name]>[i]);
23.               }
24.               elseif(attribute[function]=="MIN"){
25.      <attribute[name]><attribute[function]> =
Math.min(<attribute[name]><attribute[function]> , <attribute[name]>[i]);
26.               }

27.               if(attribute[function]=="AVG"){
28.      <attribute[name]><attribute[function] = <attribute[name]><attribute[function]> /
<attribute['name']>.length;
29.               }
30.      }// Java for loop ends
31.               } // PHP internal if block ends

32.      double value = <sensor[value]>;
33.      this.setValue(value);
34.      }//PHP for each block-2 ends
35.      }//Java try block ends
36.      catch(Exception ex){ // Java catch block starts
37.               System.out.print("Exception in Sensor: <sensor[name] >:"+ex.getMessage());
38.               this.setValue(0);
39.      }//Java catch block ends

40.      }//PHP external if block ends
41.      }//Java method declaration ends

42. }//Java class declaration ends
```

Please note that the prototype described in this section is just a proof of concept and does not define the limits of the actual framework described in Section 7.1.

7.3 Prototype Usage

Here we discuss how the prototype can be used to create ADD logic for a game:

1. Configure the Collector component so that it can collect information from the game. In our experience, it was only a few lines of code changes to pass the game object as a parameter to the Collector.

2. Run the *ObjectInformationCollector* to obtain all the attributes up to a certain depth inthe game (please see Figure 13 and related discussion in Section 7.2 on growth of number of attributes with the depth).

3. Intuitively select attributes for monitoring and mark them to be observed using the *observe* flag from the Manager (using the Attribute Tree Visualization to help locate the intended attributes). In doing so, we can attempt to select two types of attributes:

    a. Potential attributes for sensors: These are the attributes that shows how much difficulty the player is facing but cannot be easily modified (modification of these attributes usually seems unfair to the player). For example, the score of the player, number of lives remaining, and so on.

    b. Potential attributes for decisions: These are the attributes that can be modified to make the game more difficult or easier to the player. For example, the map of the game, speed of the enemies, and so on.

4. Run the *RunTimeInformationCollector* and let different players play the game multiple times and record those sessions. Another option is to create different bots[14], each representative of different class of players such as beginner, intermediate, expert, and so on, and let the bots play the game.

5. Use the Session Timeline Visualization to narrow down the number of attributes for the sensors. Use the Manager to define sensors. Associate each sensor to one or more attributes.

6. Use the Manager to define observers and mark them either as generic or not generic using the *is_generic* flag. Generic observers can be only associated with one sensor and its corresponding source will be generated by the tool, whereas the custom observers (those marked as *is_generic*=false) can be associated to multiple sensors, but the developer will have to code the observer definition later with the same name as used in the Manager. Use the Session Timeline Visualization to identify the boundary values of when the adaption should take place. Define Thresholds based on the boundary values.

7. Define Triggers and associate them to observer-threshold combinations using the Manager.

---

[14] In a video game, a bot is a type of weak AI (i.e., a non-sentient computer intelligence, typically focused on a narrow task) software to control a player character.

8. Define decisions and associate one or more attributes to each decision using the Manager. In each association, define the modified attribute values (using the Session Timeline Visualization to identify what modification should take place).

9. Define rules as associations between triggers and decisions using the Manager.

10. Generate source codes for Sensors, Adaptation Detector, Inference Engine, and Decisions using the Translator. Place the generated code with the game's original source code and make any additional modifications. Configure the game to use this adaptation logic.

We used our framework to recreate the ADD scenario we have implemented earlier for the TileGame. We generated source code for one sensor class, three decisions class, the AdaptationDetector class and the InferenceEngine class (please Appendix E for the generated source code). We used the GameState class that we have implemented during our initial work on the TileGame. We only needed few lines of code (please see Table 11) to integrate the generated source code with the game.

Table 11: Custom source coded to integrate the framework generated source code to an existing game

```
TileGameState tileGame = new TileGameState();
AdaptationDriver adaptationDriver = new AdaptationDriver(tileGame);
GameInferenceEngine inferenceEngine = new GameInferenceEngine(adaptationDriver);
inferenceEngine.start();
SensorFactory sensorFactory=
    new SensorFactory(tileGame, new ResourceManager(), new Registry());
AdaptationDetector adaptationDetector =
    new AdaptationDetector(inferenceEngine, sensorFactory);
adaptationDetector.start();
tileGame.run();
```

11. Build the game and resolve any build errors. Once the game is built, run the RunTimeInformationCollector and let different players or bots play the game.

12. Repeat step-3 to step-11 above until satisfied with the result of the adaptations.

7.4 Summary

In this chapter, we presented a semi-automatic framework that would assist in applying our design pattern based approach. It also reduces developer effort by generating source code for some of the artifacts. We discussed different components of the framework and corresponding implementation choices. Additionally, we discussed a proof-of-concept prototype that we have implemented to realize the framework.

Chapter 8

Preliminary User Study

In our initial work, we used our design pattern based approach to implement auto dynamic difficulty in three different games; two of them are prototypical in nature and one of them is a commercial game. Regardless, in all of these studies, the primary researcher played the role of the game developer. This raises the concerns of whether these design patterns are useful for a developer without prior knowledge of them and how much effort it would take for a developer to gain sufficient familiarity to make effective and efficient use of them. Thus, we conducted a preliminary user study where a Post-Degree Diploma student at the University of Western Ontario voluntarily participated. This study was a course project for the student and he was not involved with this particular research prior to the study. In this chapter, we will discuss this study in detail.

8.1 Study Artifacts

In this section, we briefly discuss each of the input and output artifacts. The following artifacts were provided to the student at the beginning of the study:

- Open Source Games: Two open source games (i.e., Tetris and Space Invaders) from[47] were provided to the student. The task was to introduce auto dynamic difficulty to those games. We did not provide any restrictions on the kinds of modifications that could be done to the game. The adaptation scenarios to be implemented were also left open ended and unspecified.

- Base Level Implementation: The base level implementation that we have found to be reusable across different games (please see section 6.2) was provided to the student. In Appendix D, we include examples from this implementation.

- Programmer's Manual: A programmer's manual showing example usage of the design patterns was provided to the student. In Appendix A, we include the complete programmer's manual.

- Research Papers: To make the participant familiar with the design patterns, one of our published research paper (i.e., [29]) was provided to the student.

- Survey Questionnaire: A survey questionnaire comprising 10 questions was provided to the participant. The questionnaire had three different types of questions related to the developer, the games, and the experience of using the design patterns.

At the end of the study, the following artifacts were collected from the participant:

- Completed Implementations: The completed ADD implementations on top of the originally provided open source games were collected.

- Completed Survey Questionnaire: Two copies of the completed survey questionnaire, each based on one of the games, were collected.
- Critical Review: The participant was asked to provide a critical review of the design patterns and the base level implementation based on his experience.
- Developer Log: A brief description of the activities and associated effort to implement the adaptations scenarios on top of the games.

8.2 Participant

In Table 12, we show the demography of the participant. These information were collected from the developer section of the survey questionnaire.

Table 12: Demography of the preliminary user study participant

Department	Computer Science
Program	Post-Degree Diploma[15]
Experience of working with Object Oriented programming and design	1 year or more but less than 2 years
Experience of working with software design patterns	Less than 1 year

8.3 Adaptations Implemented

We have already discussed the games used for this study in Chapter 5 (please see Sections 5.4 and 5.5). Thus, in this section, we will only discuss the adaptation scenarios that were implemented. The participant implemented two scenarios in

[15] The student had a prior university degree in a field other than Computer Science and was completing a one-year diploma to provide background and obtain a credential in Computer Science.

each of the games. We did not recommend any specific order, but we found from the critical review document that the student worked on the Space Invaders game after finishing his work on the Tetris game. In the Tetris game, both scenarios use mutually exclusive sensors and decisions. In the first scenario, the participant categorizes some of the tetriminos[16] into two classes. The straight-line and T-shaped tetriminos are considered desirable while the S-shaped and Z-shaped tetriminos are considered undesirable. For the sensor, average clearance rate, a measure of the number of tetriminos dropped between one line being cleared and the next line to be cleared, is used. In Table 13, we show how the ratio of desirable and undesirable tetriminos changes based on the values observed from the average clearance rate sensor. Please note that a high clearance rate indicates that the player is performing quite well, and so the player is given fewer desirable tetriminos and more undesirable tetriminos so as to make the game more challenging.

Table 13 : Average clearance rate based scenario in the Tetris game

Average clearance rate	Ratio of desirable and undesirable tetriminos
Low	High : Low
Medium	Equal : Equal
High	Low : High

In the second scenario, the speed of the falling tetriminos is adjusted as it directly impacts the difficulty of the game by controlling the amount of time the player has to decide where to put the tetriminos. When to make the adjustment is determined

[16] Tetriminos are game pieces shaped like tetrominoes, geometric shapes composed of four square blocks each.

based on the stack height – the number of tetriminos from the baseline to the highest point, which is an indication of the player's perceived level of difficulty. In Table 14, we show how the speed of tetriminos is changed based on the value observed through the stack height sensor. Please note that a low stack height indicates that the player is performing well, and so the speed of the tetriminos are increased so as to make the game more challenging.

Table 14 : Stack height based scenario in the Tetris game

Stack Height	Speed of Falling Tetriminos
Low	Incremental Increase
Medium	Incremental Decrease
High	Slowest

In the Space Invaders game, three different aspects of the game are modified based on a combination of observations from two different sensors monitoring the aliens' height and number of aliens remaining. The aspects that are modified to create the appropriate level of difficulty are aliens' rate of descent, the speed of the player's shots and the overall speed of the game. Please note that as the alien height decreases, the player's perceived difficulty level increases whereas it decreases with the numbers of aliens remaining. As we see from Table 15, when the aliens' height and the number of aliens remaining are either (close to top, high) or (middle, medium), the player has equal likelihood of winning or losing and thus all the attributes are kept at their original values to give the player the opportunity to show his/her performance. When the aliens' height and aliens remaining are either (close to top, medium/low) or (middle, low), we can say the player has played well so far and thus the aliens' rate of descent, the speed of player's shots, and the game speed

are set to (fast, slow, slow) to make the game challenging for the player. On the contrary, when the aliens' height and aliens remaining are (middle/close to bottom, high), we can say the player is not playing as well and thus the aliens' rate of descent, speed of player's shots, and the game speed are set to (slow, fast, fast) to make the game easier to the player. Similarly, when the aliens' height is close to the bottom and the number of aliens remaining is medium, the last opportunity is given to the player whereas when the aliens' height is close to bottom and the number of aliens remaining is low, we can say the player probably had a hard time catching up with the aliens but is about to win, so the game speed is decreased to let him/her enjoy the end of the game.

Table 15 : Combination of scenarios in the Space Invaders game

Alien Height	Alien Remaining	Alien's rate of descending	Speed of player's shots	Game speed
Close to top	High	Normal	Normal	Normal
Close to top	Medium/Low	Fast	Slow	Slow
Middle	High	Slow	Fast	Fast
Middle	Medium	Normal	Normal	Normal
Middle	Low	Fast	Slow	Slow
Close to bottom	Low	Normal	Normal	Slow
Close to bottom	Medium	Slow	Fast	Normal
Close to bottom	High	Slow	Fast	Fast

8.4 Analysis Conducted

For analysis, we conducted a three-pass content analysis on the critical review and the developer notes document. For effort related information, in the first pass, we went through both documents and highlighted all time related information. In the second pass, we summarized them in tabular format for each design pattern and each game and compared between documents to verify that they do not have any conflicts. In the third pass, information from separate documents were combined

and compared against the ease of usage information collected from the survey for interpretation. For the critical review, in the first pass, we went through the critical review document and highlighted any feedback about the design patterns or the base level implementation. In the second pass, we combined multiple statements discussing the same aspects into one feedback item. In the third pass, the feedback items were categorized into one of the following five types: general praise, specific strength, improvement suggestion, critical feedback, and red flags.

8.5 Results and Interpretations

We asked the participant about how easy or difficult it was to use each of the design patterns. The participant's response was collected on a five-level Likert scale where 1 means extremely easy and 5 means extremely difficult. In Table 16 and Table 17, we show participant's rating of ease of usage of each of the design patterns based on his experience of applying them to the Tetris and the Space Invaders games respectively. Please note that working with the Tetris game is the student's first exposure to the design patterns whereas in the Space Invaders game he is applying them for the second time.

Table 16 : Ease of usage of each of the design patterns on the Tetris Game

Design Patterns	Ease of usage (1= extremely easy; 5 = extremely difficult)					
	1	2	3	4	5	N/A
Sensor factory	X					
Adaptation detector			X			
Case-based reasoning	X					
Game reconfiguration			X			

Table 17 : Ease of usage of each of the design patterns on the Space Invaders game

Design Patterns	Ease of usage (1= extremely easy; 5 = extremely difficult)					
	1	2	3	4	5	N/A
Sensor factory	X					
Adaptation detector			X			
Case-based reasoning	X					
Game reconfiguration	X					

The Sensor factory and case-based reasoning patterns were consistently rated as 1 (i.e., extremely easy to use) for both the games. The adaptation detector and game reconfiguration patterns were rated as 3 (i.e., moderately easy/difficult) based on the experience of applying them in the Tetris game. Among them, the rating of the game reconfiguration pattern has improved after applying the patterns in the Space Invaders game. The rating of the adaptation detector pattern remained the same at 3. We have verified that these ratings match with the descriptions in the critical review document which will be discussed later in this section.

Next, we present the effort-related information collected from the critical review document. We have also verified that it matches with the information provided in the developer notes document.

Table 18 : Effort spent of implementing ADD in the Tetris and the Space Invaders games

Design Pattern	**Implementation Time (HH:MM)**	
	Tetris	**Space Invaders**
Sensor factory	1:35	1:00
Adaptation detector	2:00	1:45
Case-based reasoning	0:30	0:30

Game reconfiguration	2:40	0:00
Post development testing and debugging	4:00	0:05
Total	**8:45**	**3:20**

As we see from Table 18, within just about 12 development hours (i.e., 8:45 + 3:20), provided the base level implementation, the participant managed to use the design pattern based approach to implement two auto dynamic difficulty scenarios in each of two different arcade style Java games. It is noteworthy that the overall development time has decreased to less than 50% (3:20 from 8:45) just after his first experience which is a clear indication of decrease in effort with more experience with the design patterns. We can also see that the patterns the participants rated to be comparatively difficult (i.e., adaptation detector and game reconfiguration) contributed towards more development time in the Tetris game. As the participant mentioned that he managed to reuse the game reconfiguration logic from the Tetris to the Space Invaders game, the effort spent was negligible and is recorded as 0 hours. Also, the participant reported that the exception messages in our base level implementation were not descriptive enough and for the adaptation detector pattern he had to add some custom code in the base level implementation. These two factors contributed towards more post development testing and debugging time in the Tetris game. The participant managed to mitigate these issues by updating the exception messages and abstraction in the adaptation detector implementation. Also, more testing was done during the development of the Space Invaders game. Thus, the post development testing and debugging time for that game has dropped to 5 minutes. Even though this study is not comparable to a commercial project because of the scope, it is noticeable that the participant spent

about 33% time in post development testing and debugging which is very close to the amount of time typically spent in such tasks in commercial projects (i.e., 30% [48]).

Lastly, we discuss the participant's critical feedback about the design patterns and the base level implementations. In Figure 16, we show a summary of participant's feedback. Each feedback item is categorized into one of the following five types: general praise, specific strength, improvement suggestion, critical feedback, and red flags. The general praise and specific strengths were related to the design patterns whereas the improvement suggestions, critical feedback, and red flags were related to the base level implementation as summarized by the participants on his own words:

"*The simplicity, modularity, and reusability of the design pattern based ADD framework enables an inexperienced user to generate a functional ADD enabled game within a reasonable time period. Furthermore, improvements to the time needed to implement various design patterns were observed after implementation of only one set of scenarios, an indication of the rapid learnability of the framework—a fact that can likely be attributed to its simplicity.*

One of the greatest strengths of this framework is the modularity. This separation of various aspects of the framework make it easier to focus on one aspect at a time—simplifying the task at hand, and reducing the learning curve required. Not only can each aspect of the framework be learnt and understood in progressive steps, but

decisions regarding the implementation and integration of the framework can be analysed and addressed in progressive steps as well.

Many of the obstacles to the learnability of the framework were unrelated to the framework itself, but rather a product of issues with the implementation and documentation used."

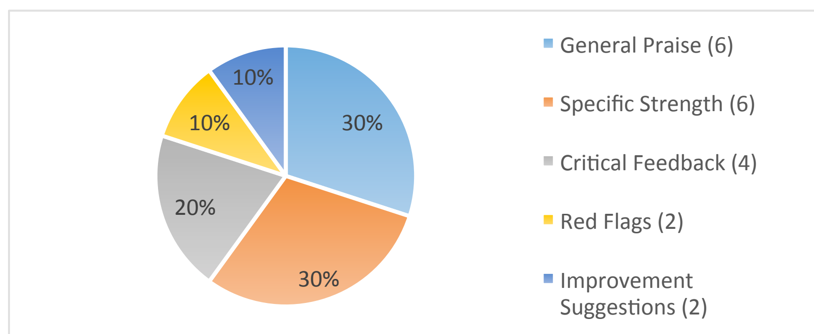


Figure 16 : Summary of participant's feedback about the design patterns and base level implementation

We will not discuss the feedback in the general praise and the specific strength section as they do not call for any further action from our end. We will discuss the feedback from the other three categories here:

Red-flags:

1. Partial implementation in the adaptation detector: In the base level implementation provided to the participant, the adaptation detector class in the adaptation detector pattern is partially implemented. We acknowledge this as a deficiency of our base level implementation and a source of confusion. We have already incorporated the participant-provided

suggestion of declaring that class as an abstract class and leaving the developer to extend from there (this approach is more in par with our other base level classes and has already been used in the sensor factory and the case based reasoning pattern).

2. Non-descriptive exceptions: Some of the exceptions in our base level implementation are not descriptive and do not provide enough contextual information. We acknowledge it as a deficiency of our base level implementation and are currently revising our code to address it.

Critical feedback:

1. Implementation order in adaptation detector: The adaptation detector class requires referencing code segments that are completed later. This accounted for confusion during the first scenario implementation in the Tetris game. We believe this issue can be overcome by “programming to an interface”. Also, our semi automation framework[17] (please see Chapter 7) can help manage this logic.

2. Implementation order in case based reasoning: The participant implemented the decisions after the inference engine and raised concern against this implementation order. Indeed we have already suggested a different

[17] The proof-of-concept for the semi automation framework was under development during this preliminary user study and thus could not be used here. We recognize the potential of a similar user study involving the framework in the future.

implementation order in the step by step process described in [29]. We take as an action item from this feedback to document our suggested implementation order in the source code as well.

3. Applicability of game reconfiguration pattern: The game reconfiguration pattern is very different from the other three patterns, as it does not contain much adaptation logic. On the contrary, it creates the foundation to push changes to the game from the case based reasoning pattern. Also, the participant noted that the implementation logic was directly transferrable to another game and thus should be part of the base level implementation. We acknowledge that this pattern requires a lot of boilerplate coding and for each game needs to be only implemented once in most cases. That said, we differ on the opinion of this logic being part of the base level implementation. The participant managed to port the implementation from the Tetris to the Space Invaders game as they both used similar threading and input handling techniques (a plausible reason for this could be that both of them were implemented by the same developer), which might not be true for games using different Java libraries for those purposes.

4. Incremental complexity in adaptation detector pattern: The participant noted that the complexity of the adaption detector dramatically increases with the number of sensors and if the adaptation scenarios are interrelated. We consider this problem analogous to a system having a large number of

potentially interrelated requirements and thus a traceability matrix and other validation techniques can help to mitigate this issue.

Improvement suggestions:

1. Adding typical modifications scenarios in decisions: The participant suggested that typical adjustments such as increment and decrement can be incorporated in the generic decision class to decrease the amount of custom code.

2. Analysis tools for finding threshold boundaries: The participant found it very time consuming to find the appropriate boundaries for threshold values. In our semi-automatic framework, we have a module called enhancer (please see Chapter 7) that encompasses such a task. The proof-of concept prototype also provides ways for basic analysis such as plotting based on user logs.

8.6 Summary

In this Chapter, we reported on a preliminary user study where the participant, without any prior knowledge of our design pattern based approach and minimal experience of working with design patterns in general, managed to implement two scenarios in each of the two games provided with minimal effort (about 12 hours). The participant provided a detailed feedback of his experience about using the design patterns and their base level implementation. We also conducted a survey on the participant for a quantitative rating of his experience and other complementary information. We also presented our analysis of his feedback.

Chapter 9

Conclusions

In this chapter, we highlight our contributions, discuss implications for using a design pattern based approach for ADD, and list some possible future research directions. Finally, we conclude the thesis with some final remarks.

9.1 Key Contributions

We derived four software design patterns namely, Sensor Factory, Adaptation, Detector, Case-based Reasoning, and Game Reconfiguration from the self-adaptive system literature in the context of auto dynamic difficulty (ADD) in video games. We have created a generic base level implementation of these design patterns in Java. We have applied the design pattern based approach and the base level implementation to three different games – Pac-Man, TileGame and Minecraft. Based on our experience from the first two games, we provided a step-by-by process for applying the design pattern based approach in a video game and verified the process by applying the process while developing ADD for Minecraft. We carried out a source code analysis on the implementations of ADD in these games for measuring reusability and amount of reuse. Through the analysis we found that reusability

metrics such as number of methods (NOM), weighted methods per class (WMC), and coupling between objects (CBO) indicated high reusability of our base level implementation and the amount of reuse can be as high as 74.26%, even for commercial games like Minecraft. We described a code-generation based semi-automatic framework that can be used to easily apply the design pattern based approach in a game with minimal manual effort. Additionally, we implemented a proof-of-concept prototype based on the framework and tested the integration of the prototype with multiple games. We also conducted a preliminary user study where a Post-Degree Diploma student at the University of Western Ontario voluntarily participated. The student was not involved with this particular research before the study and still he managed to apply the design pattern based approach to create ADD in two popular arcade style games: Space Invaders and Tetris.

9.2 Implications

In this section, we discuss the benefits of using a design pattern approach for implementing ADD in video games based on our work and their implications:

A) Reusable Source Code: Reusability refers to the degree to which existing applications 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 [27], we reported an empirical investigation involving source code analysis of two prototypical Java games (i.e., Pac-Man and TileGame). In that study, we noticed 77.52% and 79.68% code reusability in Pac-Man and TileGame respectively while implementing the

adaptive systems using these design patterns. In Chapter 6, we have extended this study to a commercially acclaimed game (i.e., Minecraft [30]) and experienced comparable results. 600 SLOC (i.e., 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 the 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 creating ADD in video games. From our experience on developing ADD for Pac-Man and TileGame, we formalized such a process and applied it on the Minecraft game. A well-defined process such as this is also important for industrial adoption for several reasons such as measuring progress, planning, and automation. Furthermore, developers can focus more on game play design and ADD logic design rather than implementation details. Unlike ad-hoc approaches, a well-defined process is repeatable with consistent results across various games.

   Since the process is defined in a step-by-step method with specific artifacts expected as outputs from each step, it will be possible to define specific metrics to estimate the project size and later measure the progress as the project moves forward.

C) Impact on Quality Factors: In [27], we examined how different software quality factors are impacted by the usage of these design patterns. We have already discussed the impact on reusability (please see Section 6.2). We briefly discuss the impact on few other quality factors below.

Integrability: Integrability refers to the ability to make the separately developed components of the system work correctly together. As we can see in Figure 5, the integration points among the design patterns and with the game are clearly defined. Because of these clearly defined integration points, the four design patterns can be integrated with each other and a game easily.

Portability: Portability is the ability of a system to run under different computing environments. A framework- or middleware-based approach for creating an ADD 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 [11]. These design patterns were derived from the self-adaptive systems literature in the context of ADD in video games. This indicates the portability of these design patterns across domains. Also, in our case study, we managed to port them (as a solution) from one game to another within the platform (Java). This indicates portability across systems on the same platform. In the future, we plan to examine the portability of these design patterns across platforms as well.

Maintainability: Maintainability refers to the ease of the future maintenance of the system. As discussed earlier, different parts of the design patterns have specific concerns (e.g., Sensors will collect data, Drivers will make changes to the game, etc.), and so the resulting source code will have high traceability and maintainability. Furthermore, as the use of these design patterns provides source code reusability (please see Section 6.2), this will increase the probability that prior testing has eliminated defects while being used in a new game.

D) Automation: In Chapter 7, we described a framework that will guide the developers through the process of applying the design patterns. It is essentially a semi-automatic tool that will help developers to easily integrate a game into the tool and then identify metrics for sensors, identify attributes to adjust game difficulty, maintain traceability between these artifacts, and so on. Such a framework works as motivation for adopting a new approach. The proof-of-concept for the framework is validated through a prototype.

9.3 Future Directions

In this section, we briefly discuss some possible future directions for our research:

A) 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. In principle, however, these design patterns should be sufficient to implement more complex forms of adaptability in game-play for other purposes. Figure 17 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 a number of rules and other associated artifacts (i.e., sensors, observers, triggers and decisions) focused on each of these aspects. In a scenario with 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.

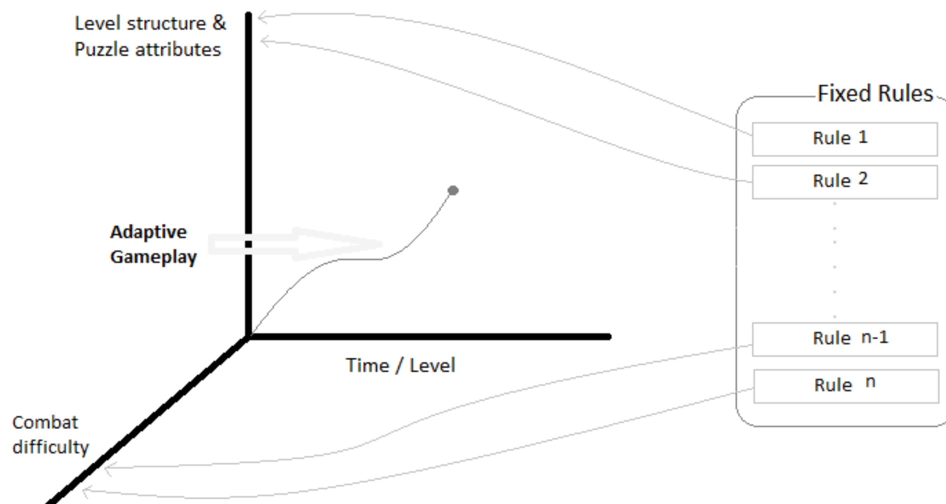


Figure 17: Concept of multi-dimensional adaptive gameplay

B) Achieving ADD in Multiplayer Games: To date, we have used these design patterns for implementation of ADD in single player games. Recently, Baldwin et al. [49] presented a classification framework for ADD in multiplayer games and, by applying that framework, found that many modern multiplayer games use some sort of ADD. To the best our knowledge, no existing scientific literature reports how to achieve ADD in multiplayer games. One of the key challenges for ADD system for a multiplayer game would be to provide different treatments to different players based on their expertise and still appear unbiased and fair. Our future plan is to extend (if necessary) and apply the design pattern based approach in a multiplayer game to achieve ADD. The multiplayer version of Minecraft would be a plausible test bed for such experimentation.

C) Further Empirical Studies: During our related work review, we noticed a number of studies where the researchers provided the implemented game to some external players and investigated their experience (e.g.,[18], [13], [19] etc.). We did not find any empirical study in ADD literature where the researchers provided their implemented artifacts to external developers and empirically investigated their experience about further developing with the help of those artifacts. We performed one such study in Chapter 8. Such studies are important as they provide more insight into applying those artifacts outside laboratory. We would like to conduct more such studies with more participants, including experienced developers from industry. We would also like to use the semi automation framework (please see Chapter 7)

for such a study. Additionally, we want to experiment on developers applying our design pattern based approach in platforms other than Java. The empirical research methods for such a study can be case-study, controlled experiments, focus groups, and so on.

9.4 Concluding Remarks

Design patterns are a formal approach of describing reusable solutions for a design problem. Game developers can benefit from two types of design patterns: game design patterns and software design patterns for video games. While popular commercial games such as "Max Payne", "Half-Life 2" and "God Hand" use the concept of auto dynamic difficulty, no information is publicly available about how ADD is implemented in these games from a software design perspective. Furthermore, research in this area has largely been done in an ad-hoc fashion and is therefore not reusable or applicable to other games. In this thesis, we presented a design pattern approach for implementing ADD in video games. We validated our approach through multiple case studies. We discussed benefits of adopting this approach based on results from our empirical investigations. Additionally, we have developed process and automation tools for applying this approach. We have also provided details of our research execution process and analysis tools used. We encourage other researchers to take advantage of our design pattern based approach and/or any other research artifacts.

References

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

[2] D. Charles and M. Black, "Dynamic Player Modelling: A Framework for Player-Centered Digital Games," in *International Conference on Computer Games: AI, Design and Education*, 2004.

[3] B. Pfeifer, "Creating Emergent Gameplay with Autonomous Agents," in *Game AI Workshop at AAAI-04*, 2004.

[4] B. Reynolds, "How AI Enables Designers," 2004. [Online]. Available: http://gamasutra.com/php-bin/news_index.php?story=11577. [Accessed 16 July 2014].

[5] B. Snow, "Why most people don't finish video games," 17 August 2011. [Online]. Available: http://www.cnn.com/2011/TECH/gaming.gadgets/08/17/finishing.videogames.snow/. [Accessed 13 April 2014].

[6] 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 *Sixth International Conference on Natural Computation (ICNC)*, Yantai, Shandong, 2010.

[7] C. Bailey and M. Katchabaw, "An experimental test bed to enable auto-dynamic difficulty in modern video games," in *2005 North American Game-On Conference*, 2005.

[8] E. Adams, Fundamentals of Game Design (2nd Edition), New Riders, 2010.

[9] M. Booth, "The AI systems of Left 4 Dead, Keynote on Fifth Artificial

Intelligence and Interactive Digital Entertainment Conference," 2009. [Online]. Available: http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf. [Accessed 16 July 2014].

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

[11] A. J. Ramirez and B. H. Cheng, "Design patterns for developing dynamically adaptive systems," in *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2010.

[12] M. I. Chowdhury and M. Katchabaw, "Software Design Patterns for Enabling Auto Dynamic Difficulty in Video Games," in *17th International Conference on Computer Games (CGAMES)*, Louisville, Kentucky, USA, 2012.

[13] N. Hocine and A. Gouaïch, "Therapeutic games' difficulty adaptation: An approach based on player's ability and motivation," in *16th International Conference on Computer Games (CGAMES)*, 2011.

[14] P. Rani, N. Sarkar and C. Liu, "Maintaining optimal challenge in computer games through real-time physiological feedback," in *11th International Conference on Human-Computer Interaction*, Las Vegas, 2005.

[15] R. Hunicke, "The case for dynamic difficulty adjustment in games," in *2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 2005.

[16] N. Shaker, G. Yannakakis and J. Togelius, "Towards Automatic Personalized Content Generation for Platform Games," in *Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[17] P. Demasi and A. J. d. O. Cruz, "Online Coevolution for Action Games," *International Journal of Intelligent Games & Simulation,* vol. 2, no. 2, pp. 80-88, 2003.

[18] R. Hunicke and V. Chapman, "AI for Dynamic Difficulty Adjustment in Games," in *Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence*, 2004.

[19] H. Qin, P. P. Rau and G. Salvendy, "Effects of different scenarios of game difficulty on player immersion," *Interacting with Computers,* vol. 22, no. 3, pp. 230-239, May 2010.

[20] O. Missura, "Adaptive agents in the context of connect four," in *LWA 2007: Lernen - Wissen - Adaption*, 2007.

[21] O. Missura and T. Gartner, "Online adaptive agent for connect four," in *Fourth International Conference on Games Research and Development (CyberGames 2008)*, 2008.

[22] 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, no. 5, pp. 2415-2433, September 2008.

[23] P. Gestwicki and F. Sun, "Teaching Design Patterns Through Computer Game Development," *Journal on Educational Resource in Computing,* vol. 8, no. 1, pp. 1-22, 2008.

[24] M. Antonio, G. Jiménez-Díaz and J. Arroyo, "Teaching Design Patterns Using a Family of Games," in *14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science*, Paris, France, 2009.

[25] J. Narsoo, M. Sunhaloo and R. Thomas, "The Application of Design Patterns to Develop Games for Mobile Devices using Java 2 Micro Edition," *Journal of Object Technology,* vol. 8, no. 5, pp. 153-175, 2009.

[26] S. Björk and J. Holopainen, Patterns in Game Design, Massachusetts, USA: Charles River Media, Inc. , 2004.

[27] M. I. Chowdhury and M. Katchabaw, "Improving software quality through design patterns : a case study of adaptive games and auto dynamic difficulty," in *Game-ON 2012*, 2012.

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

[29] M. I. Chowdhury and M. Katchabaw, "Bringing auto dynamic difficulty to commercial games: A reusable design pattern based approach," in *Computer Games (CGames'13)*, Louisville, KY, USA, 2013.

[30] "Minecraft," [Online]. Available: https://minecraft.net/. [Accessed 16 July 2014].

[31] "Main Page – Minecraft Coder Pack," [Online]. Available: http://mcp.ocean-labs.de/. [Accessed 16 July 2014].

[32] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Transactions on Autonomous and Adaptive Systems,* vol. 4, no. 2, pp. 1-42, May 2009.

[33] L. H. Rosenberg, R. Stapko and A. Gallo, "Risk based object oriented testing," in *24th annual Software Engineering Workshop, NASA*, 1999.

[34] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented

design," *IEEE Transactions on Software Engineering,* vol. 20, no. 6, pp. 476-493, 1994.

[35] "USC Code Count," [Online]. Available: http://sunset.usc.edu/research/CODECOUNT. [Accessed 16 July 2014].

[36] "Metrics 1.3.6," [Online]. Available: http://metrics.sourceforge.net. [Accessed 16 July 2014].

[37] "Understand Your Code," [Online]. Available: http://www.scitools.com. [Accessed 16 July 2014].

[38] M. Jennings-Teats, G. Smith and N. Wardrip-Fruin, "Polymorph: dynamic difficulty adjustment through level generation," in *PCG Workshop with FDG'2010*, 2010.

[39] B. Bostan and S. Öğüt, "Game challenges and difficulty levels: lessons learned From RPGs," in *ISAGA-2009*, 2009.

[40] F. Southey, G. Xiao, R. C. Holte, M. Trommelen and J. Buchan, "Semi-Automated Gameplay Analysis by Machine Learning," in *Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE-05)*, 2005.

[41] "MYSQL :: The world's most popular open source database," [Online]. Available: http://www.mysql.com/. [Accessed 16 July 2014].

[42] "PHP: Hypertext Preprocessor," [Online]. Available: https://php.net/. [Accessed 16 July 2014].

[43] "ajaxCRUD.com - Use PHP & AJAX to CRUD from a mysql database table (create / read / update / delete)," [Online]. Available: http://ajaxcrud.com/. [Accessed 16 July 2014].

[44] "JavaScript Web APIs - W3C," [Online]. Available: http://www.w3.org/standards/webdesign/script. [Accessed 16 July 2014].

[45] M. Bostock, V. Ogievetsky and J. Heer, "D3: Data-Driven Documents," *IEEE Transactions on Visualization & Computer Graphics,* vol. 17, no. 12, pp. 2301-2309, December 2011.

[46] "D3.js - Data-Driven Documents," [Online]. Available: http://d3js.org/. [Accessed 16 July 2014].

[47] "Java 2D games tutorial," [Online]. Available: http://zetcode.com/tutorials/javagamestutorial/. [Accessed 16 July 2014].

[48] S. A. Safavi and M. U. Shaikh, "Effort Estimation Model for each Phase of Software Development Life Cycle," in *Computer Engineering: Concepts, Methodologies, Tools and Applications*, IGI Global, 2012, pp. 238-246.

[49] A. Baldwin, D. Johnson, P. Wyeth and P. Sweetser, "A framework of Dynamic Difficulty Adjustment in competitive multiplayer video games," in *2013 IEEE International Games Innovation Conference (IGIC)*, 2013.

Appendices

Appendix A: Programmer's Manual for the Usage of the Base Level Implementations of the Design Patterns

A1. Defining a Sensor:

```
import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;
public class AverageScoreSensor extends Sensor{

    public AverageScoreSensor(Object object){
        this.object = object;
        this.fieldName = "AverageScore";
        this.setValue(0);
        this.setInterval(1000);
    }

    public void refreshValue(){
        try{
            int score = ((Game)this.object).getScore();
            int life = ((Game)this.object).getPlayer().getLife();
            int averageScore = score / (6 - life);
            this.setValue(averageScore);
        }
        catch(Exception ex){
            System.out.print("Exception in Sensor:"+ex.getMessage());
this.setValue(0);
        }
    }

}
```

A2. Defining the Adaptation Detector:

```
package
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector;
import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;

public class AdaptationDetector extends Thread{

    InferenceEngine inferenceEngine;
    SensorFactory sensorFactory;
    GenericObserver averageScoreObserver;
    Threshold lowAverageScoreThreshold;
    Threshold highAverageScoreThreshold;

    public AdaptationDetector(InferenceEngine inferenceEngine,
                                          SensorFactory sensorFactory){
        this.inferenceEngine = inferenceEngine;
        this.sensorFactory = sensorFactory; this.createObservers();
        this.createThresholds();
    }

    public void createObservers(){
        averageScoreObserver = new GenericObserver();
        Sensor averageScoreSensor =
sensorFactory.getSensorByName("AverageScoreSensor");
        averageScoreSensor.addObserver(averageScoreObserver);
    }

    public void createThresholds(){
        lowAverageScoreThreshold = new Threshold(ThresholdType.LESS_THAN, 4);
        highAverageScoreThreshold = new Threshold(ThresholdType.GREATER_THAN,
4);
    }

    public void run(){
        System.out.println("Adaptation Detector started");

        while(true){
            if(averageScoreObserver.isRecentlyUpdated()){
                if(ThresholdAnalyzer.analyze(lowAverageScoreThreshold,
```

```
averageScoreObserver.getValue())){
                    System.out.println("Adaptation Detector created
'MakeGameEasy' trigger");
                    Trigger makeLevelEasyTrigger = new Trigger("MakeGameEasy");
                    inferenceEngine.notifyTrigger(makeLevelEasyTrigger);
                }

                if(ThresholdAnalyzer.analyze(highAverageScoreThreshold,

averageScoreObserver.getValue())){
                    System.out.println("Adaptation Detector created
'MakeGameDifficult' trigger");
                    Trigger makeLevelDifficultTrigger = new
Trigger("MakeGameDifficult");
                    inferenceEngine.notifyTrigger(makeLevelDifficultTrigger);
                }
            }
            try{
                this.sleep(1000);
            }
            catch(InterruptedException e){
                System.out.println(e.getMessage());
            }
        }
    }
}
```

A3. Extending Inference Engine:

```
Import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;
import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;

public class PacManInferenceEngine extends InferenceEngine{

    public PacManInferenceEngine(AdaptationDriver adaptationDriver){
        super(adaptationDriver);
        this.fixedRules.addRule(new Rule("MakeGameEasy",
          new Trigger("MakeGameEasy"),
          new MakeGameEasyDecision()));
```

```
        this.fixedRules.addRule(new Rule("MakeGameDifficult",
          new Trigger("MakeGameDifficult"),
          new MakeGameDifficultDecision()));
    }

    public void run(){
        super.run();
    }
}
```

A4. Defining a Decision:

```
class MakeGameDifficultDecision extends Decision{

    public MakeGameDifficultDecision(){
        super("MakeGameDifficult");
    }

    public void compileDecision(){
        subDecisions.put("GHOST_SPEED", 5);
        System.out.println("Game is being difficult");
    }
}
```

A5. Extending the GameState:

```
import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;
import java.util.*;
import java.awt.event.*;

public class GameState extends Game implements State, KeyListener{

    int state = 1;
    // 1 = active; 2 = being inactive; 0 = inactive; 3 = being active;

    ArrayList<KeyEventEntry> keyEvents = new ArrayList<KeyEventEntry>();

    public void keyPressed(KeyEvent e){
        if(state == 1){
            super.keyPressed(e);
        } else {
```

```
            keyEvents.add(new KeyEventEntry("keyPressed", e));
        }
    }

    public void keyReleased(KeyEvent e){
        if(state == 1){
            super.keyReleased(e);
        } else {
            keyEvents.add(new KeyEventEntry("keyReleased", e));
        }
    }

    public void keyTyped( KeyEvent  e){
        if(state == 1){
            super.keyTyped(e);
        } else {
            keyEvents.add(new KeyEventEntry("keyTyped", e));
        }
    }

    public void makeInactive(){
        if(state == 1){
            state = 2;
        }
    }

    public void makeActive(){
              if(state == 0){
            state = 3;
        }
    }

    public int getState(){
        return state;
    }

    public boolean isActive(){
        return (state == 1);
    }

    public boolean isInactive(){
        return (state == 0);
    }
```

```
public void run(){
    long  starttime;

    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);

    while(true)
    {
        starttime=System.currentTimeMillis();

    try {
        if(state == 3 && !keyEvents.isEmpty()){
        // Game is being active
        //so all the stored requests need to be served
            if(keyEvents.get(0).getEventType() == "keyPressed"){
                super.keyPressed(keyEvents.get(0).getKeyEvent());
            }
            else if(keyEvents.get(0).getEventType() == "keyReleased"){
                super.keyReleased(keyEvents.get(0).getKeyEvent());
            }
            else if(keyEvents.get(0).getEventType() == "keyTyped"){
                super.keyTyped(keyEvents.get(0).getKeyEvent());
            }

            keyEvents.remove(0);
        }

        if(state != 0){
            // Game is not inactive so game loop needs to be executed
            super.run();
        }


        if(state == 1){
        // Game is active so game thread needs to
        // sleep after executing each time
            starttime += 40;
            Thread.sleep(Math.max(0,
            starttime-System.currentTimeMillis()));
        }

        if(state == 2){
            // Inactivate request made so inactivate
```

```
                    state = 0;
                }

                if(state == 3 && keyEvents.isEmpty()){
                // Activate request made and no pending requests left. So activate
                    state = 1;
                }
            }
            catch (InterruptedException e) {
                break;
            }
        }

    }

    class KeyEventEntry{
            private String eventType;
            private KeyEvent keyEvent;

            public KeyEventEntry(String eventType, KeyEvent keyEvent){
                this.eventType = eventType;
                this.keyEvent = keyEvent;
            }

            public String getEventType(){
                 return eventType;
            }

            public KeyEvent getKeyEvent(){
                 return keyEvent;
            }
        }

    }
```

A6. Integration of Patterns:

```
import javax.swing.JFrame;
import javax.swing.JPanel;
```

```
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector.
*;
import
com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.Registry;
import
com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.ResourceM
anager;
import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.AdaptationDriver;
import
com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.SensorFac
tory;

public class PacMan extends JFrame {

    public PacMan(JPanel game) {
        add(game);
        setTitle("PacMan");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 500);
        setLocationRelativeTo(null);
        setVisible(true);
        setResizable(false);
    }

    public static void main(String[] args) {
        GameState game = new GameState();
        new PacMan(game);
        AdaptationDriver adaptationDriver = new AdaptationDriver(game);
        PacManInferenceEngine inferenceEngine =
            new PacManInferenceEngine(adaptationDriver);
        inferenceEngine.start();
        SensorFactory sensorFactory=new SensorFactory(game,
            new ResourceManager(),
            new Registry());
        sensorFactory.setSensorBasePath("pacman");
        AdaptationDetector adaptationDetector =
            new AdaptationDetector(inferenceEngine,sensorFactory);
        adaptationDetector.start();
    }
}
```

Appendix B: User's Manual for the Proof-of-concept Automation Tool

1. Collector Component

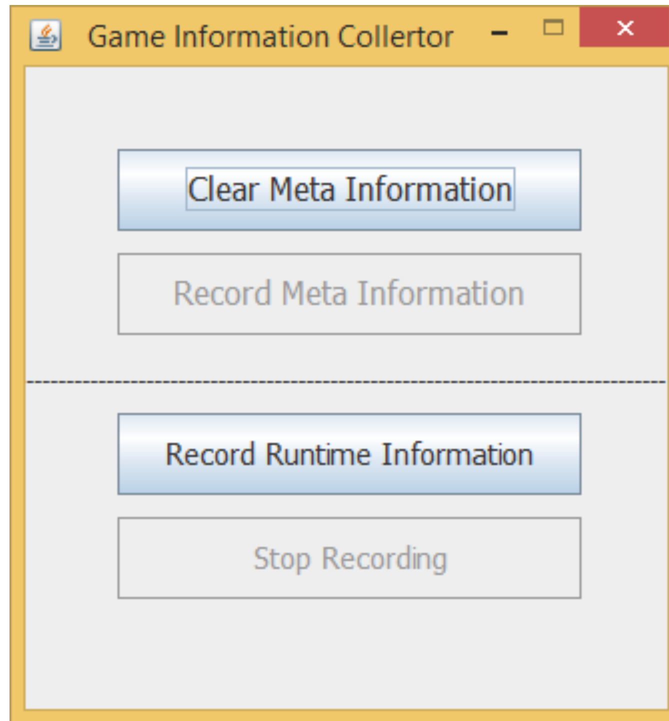
1.1. Configure the collector component

Configuring the Collector is basically passing the main game object to the ObjectInformationCollector and RauntimeInformationCollector and creating the CollectorFrame from them. Here we show how you can configure it with an existing game:

```
//The existing game object is called board

ObjectInformationCollector infoCollect =
      new ObjectInformationCollector(board);

RuntimeInformationCollector runtimeInformationCollector =
      new RuntimeInformationCollector(board);

runtimeInformationCollector.start();

new CollectorFrame(infoCollect, runtimeInformationCollector);
```

1.2. Use Collector Component

After configuring the Collector component with the game, if you run the game, then apart from the game window, the following "Game Information Collector" window will appear:



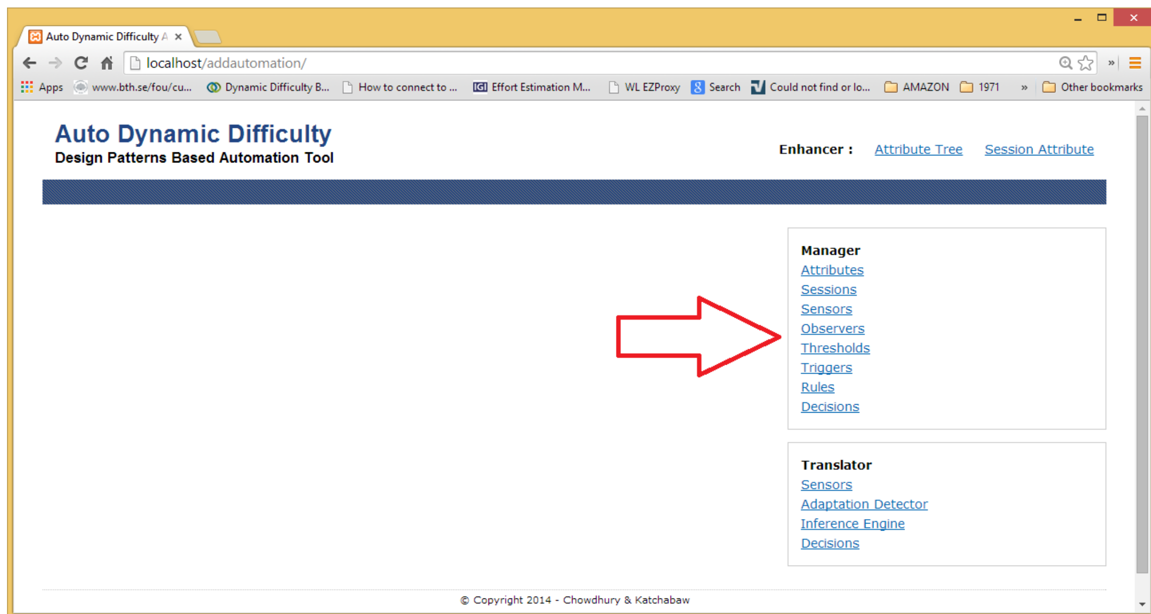
Click on the “Clear Meta Information” button to delete all the existing attribute information. Click on the “Record Meta Information” button to populate the attribute information up to a certain height. The height to discover can be configured in the ObjectInformationCollector class.

After collecting the attribute information, go to the Manager component to mark the attributes that needs to be monitored (please see section 0 below).

Click on the “Record Runtime Information” button to start a session and start recording values of all the attributes marked for monitoring at that moment. The frequency of recording can be adjusted in RuntimeInformationCollector class. Click on the “Sop Recording” button to stop recording attribute values and to end the session.

2. Manager Component

Open your browser and type the URL of the web UI to go to the Manager component (for local installation the URL will be http://localhost/addautomation/). Use the vertical "Manager" menu (shown with arrow in the picture below) to access each of the UIs within the Manager component.



2.1. Attributes

Click on the "Attributes" menu item from the "Manager" menu to get a list of all the attributes. Each page shows 10 attributes at a time. Use the pagination at the bottom of the page to go to the next page. Use the filters at the top of the page to quickly find an attribute. Click on the checkboxes under the observe column to mark an attribute for monitoring.

Auto Dynamic Difficulty A ×

localhost/addautomation/index.php?page=attributes

Apps www.bth.se/fou/cu... Dynamic Difficulty B... How to connect to ... Effort Estimation M... WL EZProxy Search

Auto Dynamic Difficulty

Design Patterns Based Automation Tool

Attributes

Filter for searching

Attributes (5297):

Name: **Data Type:**

Attribute ID	Parent Attribute	Name	Object Path	Data Type	Observe	Action
5297	--	board	board	[Ltetris.TShape$Tetr	☐	delete details
5296	curPiece	coordsTable	curPiece.coordsTable	[[[I	☐	delete details
5295	curPiece	coords	curPiece.coords	[[I	☐	delete details
5294	pieceS		pieceShape.	[Ltetris.TShape$Tetr	☐	delete details
5293	MirroredLShape	ENUM$VALUES	curPiece.pieceShape.	[Ltetris.TShape$Tetr	☐	delete details
5292	MirroredLShape	ENUM$VALUES	curPiece.pieceShape.	[Ltetris.TShape$Tetr	☐	delete details
5291	MirroredLShape	ENUM$VALUES	curPiece.pieceShape.	[Ltetris.TShape$Tetr	☐	delete details
5290	MirroredLShape	MirroredLShape	curPiece.pieceShape.	tetris.TShape$Tetrom	☐	delete details
5289	MirroredLShape	LShape	curPiece.pieceShape.	tetris.TShape$Tetrom	☐	delete details
528	hape	SquareShape	curPiece.pieceShape.	tetris.TShape$Tetrom	☐	delete details

List of attributes

Pagination

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83

2.2. Sessions

Click on the "Session" menu item from the "Manager" menu to get a list of all the sessions. You can use the filters at the top of the page to quickly find a session. Click on the "details" buttons under the "Action" column to get more details about the corresponding session.

Auto Dynamic Difficulty A ×
localhost/addautomation/index.php?page=sessions
Apps www.bth.se/fou/cu... Dynamic Difficulty B... How to connect to ... Effort Estimation M... WL EZProxy

Auto Dynamic Difficulty
Design Patterns Based Automation Tool

Sessions

Sessions (4):

Session Name: Description:

Session ID	Session Name	Description	Start Time	End Time	Action
73	--	--	2014-07-14 12:18:08	2014-07-14 12:19:06	delete details
72	--	--	2014-07-14 12:16:50	2014-07-14 12:18:00	delete details
71	--	--	2014-02-23 03:13:34	2014-02-23 03:14:21	delete details
70	--	--	2014-02-21 00:10:45	2014-02-21 00:10:58	delete details

Add Session

On the session details page, you will see list of values for different attributes along with timestamp within that session. Use the dropdown attribute filter to look at values for a specific attributes. Use the pagination at the bottom of the page to get to the next page of the list of attribute value pairs.

2.3. Sensors

Click on the "Sensor" menu item from the "Manager" menu to get a list of all the sensors. Use the filters at the top of the page to quickly find a sensor. Click on the "Add Sensor" button under the list of the sensors to open the form for adding sensor. Fill up the form and then click on "Save Sensor" to add a sensor. Once a sensor is created, click on the "details" button under the "Action" column to see more options.

Auto Dynamic Difficulty A

localhost/addautomation/index.php?page=sensors

Sensors

Filter for searching

Sensors (5):

Sensor Name: Description:

Sensor ID	Sensor Name	Description	Value Calculation	Interval	Action
9	SensorX	A	C/D	50	delete details source
8	SensorA	Sensor Desc	C/D	50	delete details source
7	SensorA		a/d	100	delete details source
6	D		C/A	10	delete details source
1	AverageScoreSensor	Some Description	curX curY	500	delete details source

List of sensors

Add Sensor

Add sensor form

Sensor Name	
Description	
Value Calculation	
Interval	
Save Sensor	Cancel

On the sensor details page, click on the "Add Related Attributes" button to open the form to associate an attribute to the sensor. In the form, select an attribute and a function for the corresponding dropdowns and click on "Save Related Attributes" button to associate that attribute to the sensor. Once an attribute is associated it will appear under the "Related Attributes" section and option for updating and/or deleting that association will appear.

Auto Dynamic Difficulty A ×

localhost/addautomation/index.php?page=sensors

Auto Dynamic Difficulty

Design Patterns Based Automation Tool

Sensors

Sensor Details:

Sensor ID	Sensor Name	Description	Value Calculation	sensor_interval
9	SensorX	A	C/D	50

Related Attributes (0):

No data in this table. Click add button below.

Add Related Attributes

Attribute	serialVersionUID
Function	SUM
Save Related Attributes	

Add related attribute form

2.4. Observers

Click on the "Observer" menu item from the "Manager" menu to get a list of all the observers. Use the filters at the top of the page to quickly find an observer. Click on the "Add Observer" button under the list of the observers to open the form for adding observer. Fill up the form and click on the "Save Observer" to add an observer. Once an observer is created, click on the "details" button under the "Action" column to see more options.

On the observer details page, click on the “Add Related sensors” button to open the form to associate a sensor to the observer. In the form, select a sensor from the dropdown and click on the “Save Related sensors” button to associate that sensor to the observer. Once a sensor is associated it will appear under the “Related sensors” section and option for deleting that association will appear.

Auto Dynamic Difficulty A ×

localhost/addautomation/index.php?page=observers

Auto Dynamic Difficulty

Design Patterns Based Automation Tool

Observers

Observer Details:

Observer ID	Observer Name	Description	Is Generic	Precision
3	AB	CD	0	12

Related sensors (2):

sensor	Action
AverageScoreSensor	delete

List of related sernors

Add Related sensors

sensor	AverageScoreSensor ▼
Save Related sensors	Cancel

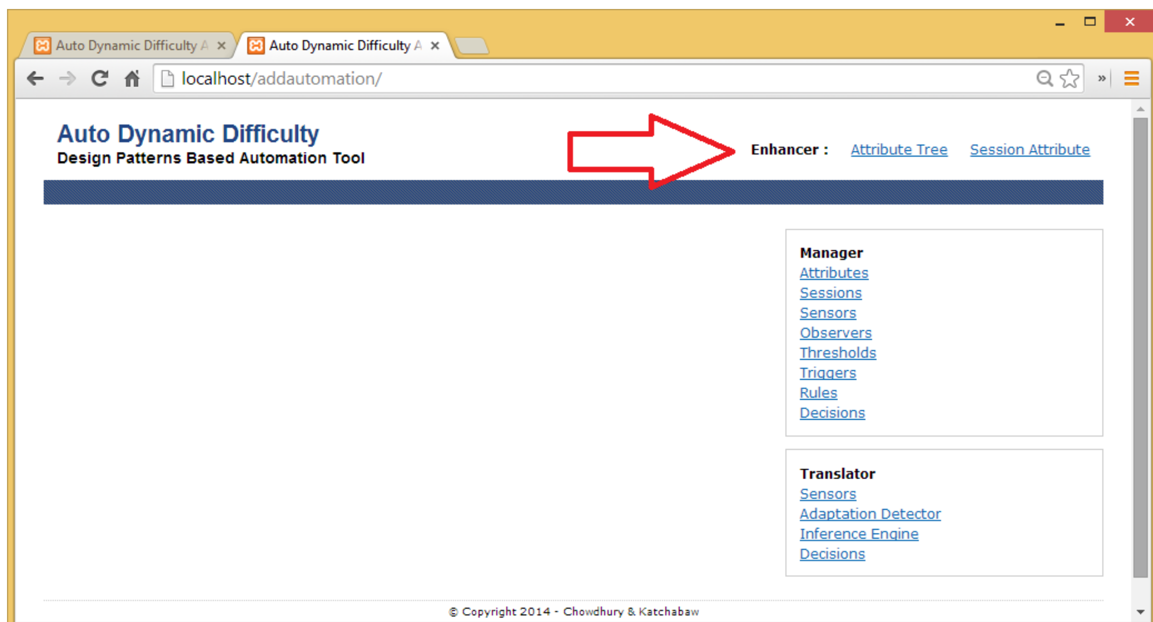
Add related sensor form

2.5. UIs for Thresholds, Triggers, Decisions, and Rules

The UIs for managing thresholds, triggers, decisions, and rules use the same patterns as the ones described above and thus are not discussed in details. From the trigger details page, combination of observer and thresholds can be added to a trigger. Similarly, a trigger and decision combination is created as rules.

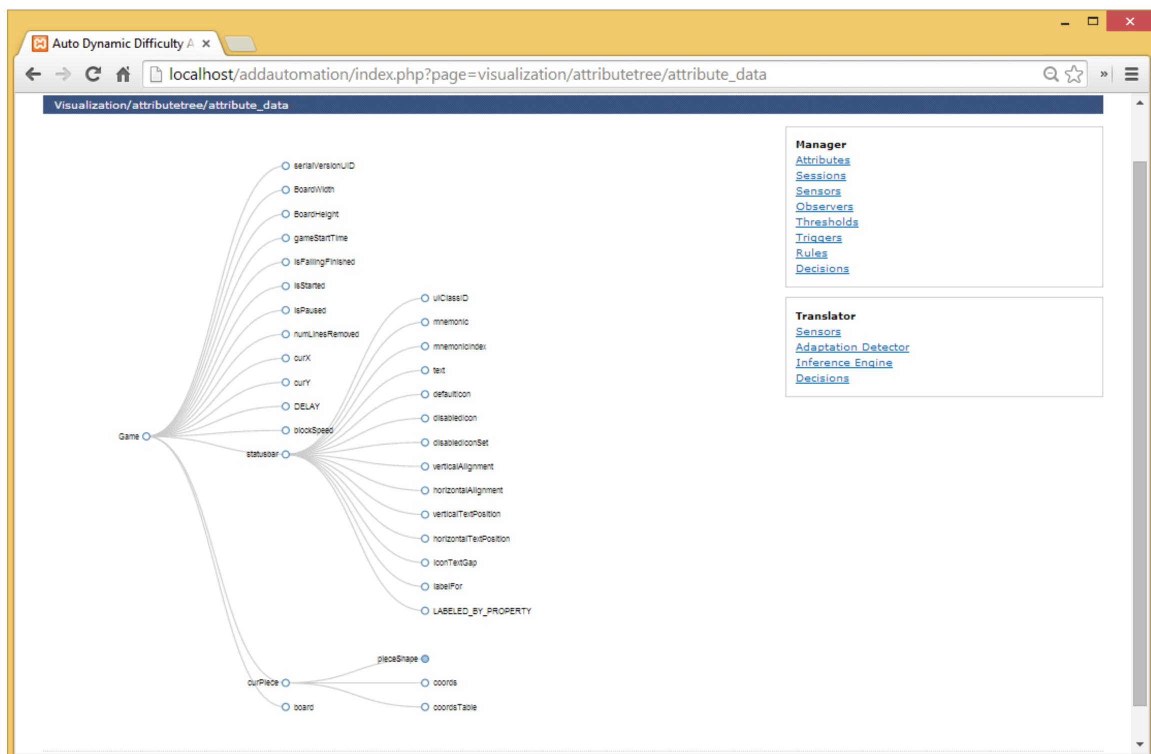
3. Enhancer Component

Use the horizontal "Enhancer" menu (shown with arrow in the picture below) to access each of the two visualization within the Enhancer component.



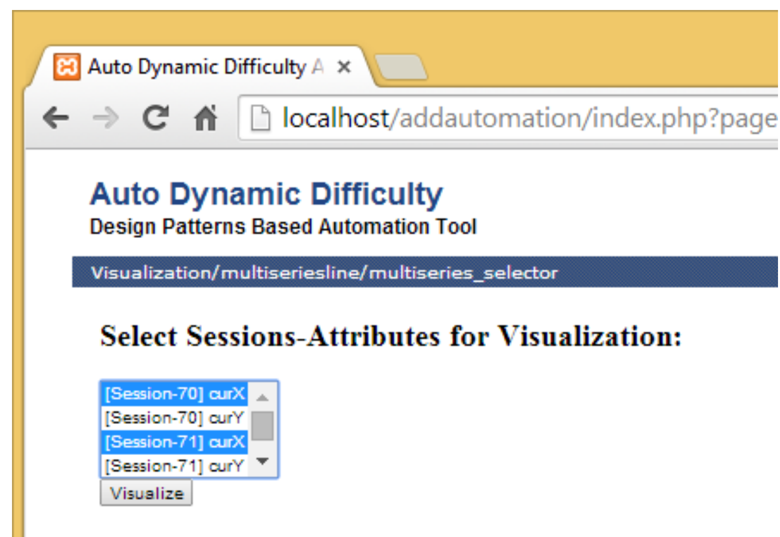
3.1. Attribute Tree Visualization

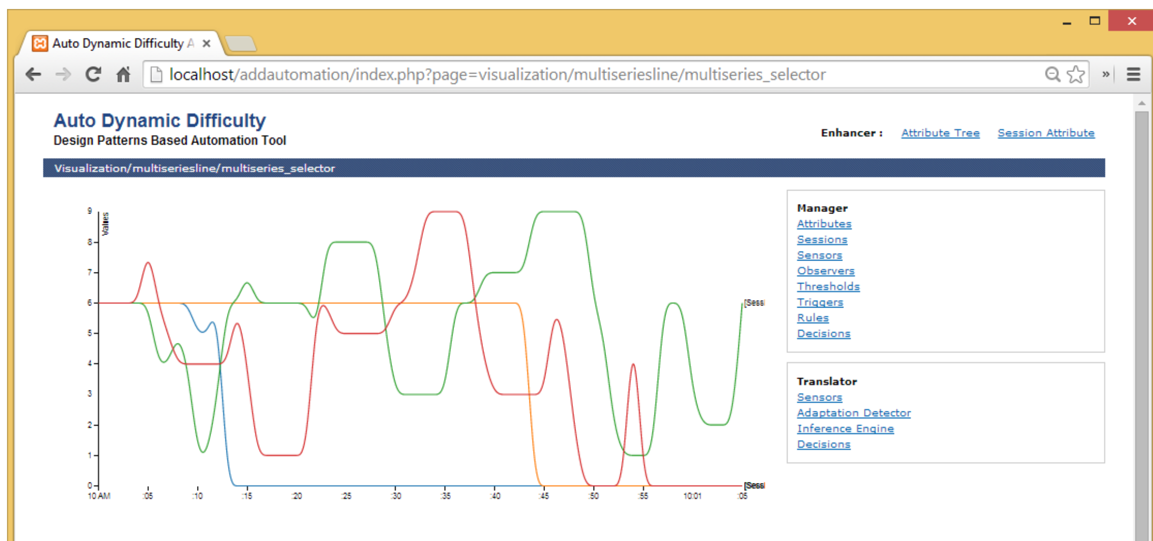
Click on the “Attribute Tree” menu item from the “Enhancer” menu to access the attribute tree visualization. This visualization shows all the attributes starting from the root object in a tree structure. Initially only the first level nodes are expanded. Any expandable node is blue colored and can be clicked to expanded.



3.2. Session Attribute Visualization

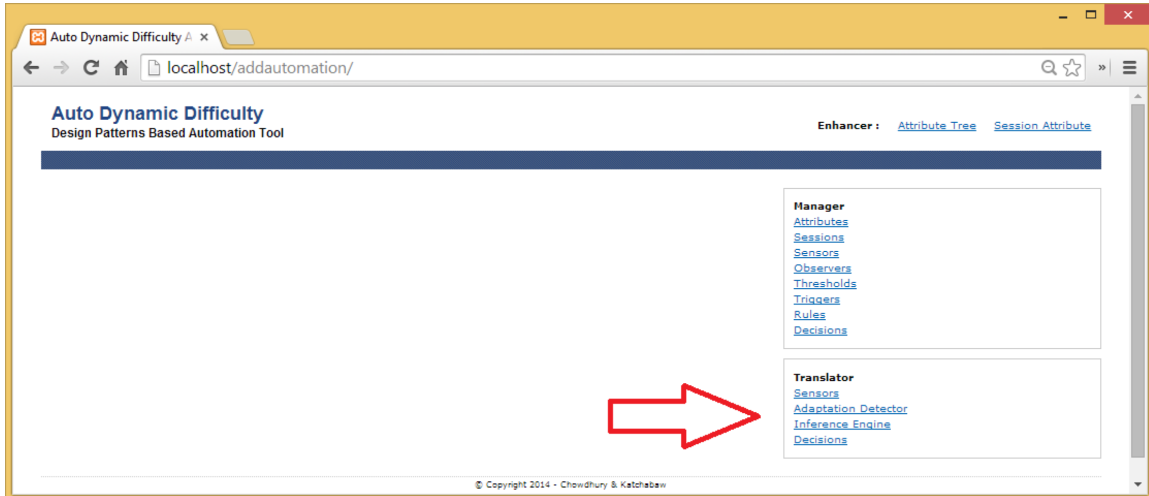
Click on the “Session Attribute” menu item from the “Enhancer” menu to access the session attribute visualization. It will require you to select one or more attributes from one or more sessions using a dropdown list. Once you select the attributes and click on the “Visualize” button, it will show the change of values for those attributes in a time line.





4. Translator Component

Use the vertical “Translator” menu (shown with arrow in the picture below) to access the generated source code from the Translator component.



Click on the “Sessions” menu item from the “Translator” menu to get a list of sensors with buttons titled “source” to generate source code. Click on any of the “source” button and download the generated source code for the corresponding sensor.

Click on the “Decision” menu item from the “Translator” menu to get a list of decisions with buttons titled “source” to generate source code. Click on any of the “source” button and download the generated source code for the corresponding decision.

Click on the “Adaptation Detector” or the “Inference Engine” menu item from the “Translator” menu to download the source of the Adaptation Detector or the Inference Engine correspondingly.

Copy the generated source codes to the game source code folder and make required code changes (i.e., very few lines of code from our experience) to integrate the ADD system to the game.

Appendix C: PHP Source Code for the Translator Component

C1. Sensor Code Generator (sensor.java.php):

```
<?php
	require_once('functions.php');
	$sensor = getSensorDetails($_REQUEST['sensor_id']);
?>
<?php
	header("Content-Type: text/plain");
	header('Content-Disposition: attachment;
filename="'.$sensor['name'].'.java"');
?>

public class <?=$sensor['name'] ?> extends Sensor{

	public <?=$sensor['name'] ?>(Object object){
		this.object = object;
		this.fieldName = "<?=$sensor['name'] ?>";
		this.setInterval(<?=$sensor['sensor_interval'] ?>);
		this.setValue(0);
	}

	public void refreshValue(){

		try{
<?php if($sensor['attributes']!=""): ?>
<?php foreach($sensor['attributes'] as $attribute): ?>
			<?=dataType($attribute['data_type']) ?>
<?=$attribute['name'] ?> = <?=$attribute['attribute_path'] ?>;
<?="\n" ?>
<?php endforeach; ?>
<?php foreach($sensor['attributes'] as $attribute): ?>
<?php if($attribute['function']!="NONE"): ?>

	<?=str_replace(array("[","]"),"",dataType($attribute['data_type'])) ?>
<?=$attribute['name'] ?><?=$attribute['function'] ?> = 0;

			for(int i = 0; i < <?=$attribute['name'] ?>.length; i++){
<?php if($attribute['function']=="SUM" || $attribute['function']=="AVG"): ?>
				<?=$attribute['name'] ?><?=$attribute['function'] ?>
= <?=$attribute['name'] ?><?=$attribute['function'] ?> + <?=$attribute['name']
?>[i];
<?php elseif($attribute['function']=="MAX"): ?>
				<?=$attribute['name'] ?><?=$attribute['function'] ?>
= Math.max(<?=$attribute['name'] ?><?=$attribute['function'] ?> ,
<?=$attribute['name'] ?>[i]);
<?php elseif($attribute['function']=="MIN"): ?>
				<?=$attribute['name'] ?><?=$attribute['function'] ?>
= Math.min(<?=$attribute['name'] ?><?=$attribute['function'] ?> ,
<?=$attribute['name'] ?>[i]);
<?php endif; ?>
			}
<?php if($attribute['function']=="AVG"): ?>
```

```
                    <?=$attribute['name'] ?><?=$attribute['function'] ?> =
<?=$attribute['name'] ?><?=$attribute['function'] ?> / <?=$attribute['name']
?>.length;
<?php endif; ?>
<?php endif; ?>
<?="\n" ?>
<?php endforeach; ?>

                    double value = <?=$sensor['value'] ?>;
                    this.setValue(value);
<?php endif; ?>
             }
             catch(Exception ex){
                    System.out.print("Exception in Sensor: <?=$sensor['name']
?>:"+ex.getMessage());
                    this.setValue(0);
             }

      }
}
```

C2. Decision code generator (decision.java.php):

```
<?php
      require_once('functions.php');
      $decision = getDecisionDetails($_REQUEST['decision_id']);
?>
<?php
      header("Content-Type: text/plain");
      header('Content-Disposition: attachment;
filename="'.$decision['name'].'.java"');
?>

public class <?=$decision['name'] ?> extends Decision{

      public <?=$decision['name'] ?>(){
             super("<?=$decision['name'] ?>");
      }

      public void compileDecision(){
<?php if($decision['attributes']!=""): ?>
<?php foreach($decision['attributes'] as $attribute): ?>
             subDecisions.put("<?=$attribute['attribute_path'] ?>",
<?=$attribute['value'] ?>);
<?="\n" ?>
<?php endforeach; ?>
<?php endif; ?>
             System.out.println("Decision <?=$decision['name'] ?> is being
execute!");
      }
}
```

C3. Adaptation Detector code generator (adaptationdetector.java.php):

```
<?php
```

```
        require_once('functions.php');
        $observers = getObservers();
        $thresholds = getThresholds();
        $sensors = getSensors();
        $observers_sensors = getObserverSensorAssignments();
        $observers_thresholds = getObserverThresholdAssignments();
?>
<?php
        header("Content-Type: text/plain");
        header('Content-Disposition: attachment;
filename="AdaptationDetector.java"');
?>

package
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector;

import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;

public class AdaptationDetector extends Thread{

        InferenceEngine inferenceEngine;
        SensorFactory sensorFactory;

<?php foreach($observers as $observer): ?>
<?php if($observer['is_generic']): ?>
        GenericObserver <?=strtolower($observer['name']) ?>Observer;
<?php else: ?>
        <?=$observer['name'] ?> <?=strtolower($observer['name']) ?>Observer;
<?php endif; ?>
<?php endforeach; ?>

<?php foreach($thresholds as $threshold): ?>
        Threshold <?=strtolower($threshold['name']) ?>Threshold;
<?php endforeach; ?>

        public AdaptationDetector(InferenceEngine inferenceEngine, SensorFactory
sensorFactory){
                this.inferenceEngine = inferenceEngine;
                this.sensorFactory = sensorFactory;
                this.createObservers();
                this.createThresholds();
        }


        public void createObservers(){
<?php foreach($observers as $observer): ?>
<?php if($observer['is_generic']): ?>
                <?=strtolower($observer['name']) ?>Observer = new
GenericObserver() ;
<?php else: ?>
                <?=strtolower($observer['name']) ?>Observer = new
<?=$observer['name'] ?>() ;
<?php endif; ?>
<?php endforeach; ?>

<?php foreach($sensors as $sensor): ?>
                Sensor <?=strtolower($sensor['name']) ?>Sensor =
sensorFactory.getSensorByName("<?=$sensor['name'] ?>");
<?php endforeach; ?>
```

```
<?php foreach($observers_sensors as $observer_sensor): ?>
		<?=strtolower($observer_sensor['sensor']['name'])
?>Sensor.addObserver(<?=strtolower($observer_sensor['observer']['name'])
?>Observer);
<?php endforeach; ?>


	}

	public void createThresholds(){

<?php foreach($thresholds as $threshold): ?>
		<?=strtolower($threshold['name']) ?>Threshold = new
Threshold(ThresholdType.<?=$threshold['type'] ?>, <?=$threshold['value1'] ?>
<?php if($threshold['value2']!="") echo ",".$threshold['value2'] ?>);
<?php endforeach; ?>
	}

	public void run(){

		while(true){

<?php foreach($observers_thresholds as $observer_threshold): ?>
			if(<?=strtolower($observer_threshold['observer']['name'])
?>Observer.isRecentlyUpdated()){

	if(ThresholdAnalyzer.analyze(<?=strtolower($observer_threshold['threshold
']['name']) ?>Threshold, <?=strtolower($observer_threshold['observer']['name'])
?>Observer.getValue())){
					Trigger
<?=strtolower($observer_threshold['trigger']['name']) ?>Trigger = new
Trigger("<?=$observer_threshold['trigger']['name'] ?>");

	inferenceEngine.notifyTrigger(<?=strtolower($observer_threshold['trigger'
]['name']) ?>Trigger);
				}
			}
<?php endforeach; ?>

			try{
				this.sleep(1000);
			}
			catch(InterruptedException e)
			{
				System.out.println("Exception in Adaptation Detector
: "+e.getMessage());
			}
		}
	}
}
```

C4. Inference Engine code generator (inferenceengine.java.php):

```
<?php
	require_once('functions.php');
	$observers = getObservers();
	$thresholds = getThresholds();
	$sensors = getSensors();
	$observers_sensors = getObserverSensorAssignments();
	$observers_thresholds = getObserverThresholdAssignments();
?>
<?php
	header("Content-Type: text/plain");
```

```
	header('Content-Disposition: attachment;
filename="AdaptationDetector.java"');
?>

package
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector;

import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;

public class AdaptationDetector extends Thread{

	InferenceEngine inferenceEngine;
	SensorFactory sensorFactory;

<?php foreach($observers as $observer): ?>
<?php if($observer['is_generic']): ?>
	GenericObserver <?=strtolower($observer['name']) ?>Observer;
<?php else: ?>
	<?=$observer['name'] ?> <?=strtolower($observer['name']) ?>Observer;
<?php endif; ?>
<?php endforeach; ?>

<?php foreach($thresholds as $threshold): ?>
	Threshold <?=strtolower($threshold['name']) ?>Threshold;
<?php endforeach; ?>

	public AdaptationDetector(InferenceEngine inferenceEngine, SensorFactory
sensorFactory){
		this.inferenceEngine = inferenceEngine;
		this.sensorFactory = sensorFactory;
		this.createObservers();
		this.createThresholds();
	}


	public void createObservers(){
<?php foreach($observers as $observer): ?>
<?php if($observer['is_generic']): ?>
		<?=strtolower($observer['name']) ?>Observer = new
GenericObserver() ;
<?php else: ?>
		<?=strtolower($observer['name']) ?>Observer = new
<?=$observer['name'] ?>() ;
<?php endif; ?>
<?php endforeach; ?>

<?php foreach($sensors as $sensor): ?>
		Sensor <?=strtolower($sensor['name']) ?>Sensor =
sensorFactory.getSensorByName("<?=$sensor['name'] ?>");
<?php endforeach; ?>

<?php foreach($observers_sensors as $observer_sensor): ?>
		<?=strtolower($observer_sensor['sensor']['name'])
?>Sensor.addObserver(<?=strtolower($observer_sensor['observer']['name'])
?>Observer);
<?php endforeach; ?>


	}
```

```
	public void createThresholds(){

<?php foreach($thresholds as $threshold): ?>
		<?=strtolower($threshold['name']) ?>Threshold = new
Threshold(ThresholdType.<?=$threshold['type'] ?>, <?=$threshold['value1'] ?>
<?php if($threshold['value2']!="") echo ",".$threshold['value2'] ?>);
<?php endforeach; ?>
	}

	public void run(){

		while(true){

<?php foreach($observers_thresholds as $observer_threshold): ?>
			if(<?=strtolower($observer_threshold['observer']['name'])
?>Observer.isRecentlyUpdated()){

	if(ThresholdAnalyzer.analyze(<?=strtolower($observer_threshold['threshold
']['name']) ?>Threshold, <?=strtolower($observer_threshold['observer']['name'])
?>Observer.getValue())){
					Trigger
<?=strtolower($observer_threshold['trigger']['name']) ?>Trigger = new
Trigger("<?=$observer_threshold['trigger']['name'] ?>");

	inferenceEngine.notifyTrigger(<?=strtolower($observer_threshold['trigger'
]['name']) ?>Trigger);
				}
			}
<?php endforeach; ?>

			try{
				this.sleep(1000);
			}
			catch(InterruptedException e)
			{
				System.out.println("Exception in Adaptation Detector
: "+e.getMessage());
			}
		}
	}
}
```

Appendix D: Example Source Code from the Base Level Implementation of the Design Patterns

D1. Abstract Sensor class (Sensor.java):

```
package com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory;

import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;

import
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector.
*;

import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;

import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;

import java.util.Observer;

public abstract class Sensor extends Thread{

	protected String fieldName;

	protected Object object;

	protected Object value;

	protected int interval;

	protected SimpleObservable observable = new SimpleObservable();


	public Object getObject(){

		return this.object;

	}

	public String getFieldName(){

		return this.fieldName;

	}

	public Object getValue(){

		return this.value;

	}
```

```
public int getInterval(){
       return this.interval;
}
public void setInterval(int interval){
       this.interval = interval;
}
public void setValue(Object value){
       if((this.value==null ||
           !this.value.equals(value)) &&
           this.observable.countObservers()!= 0){
              this.value = value;
              if(this.observable.countObservers()!= 0){
                     this.observable.setChanged();
                     this.observable.notifyObservers(this.value);
              }
              System.out.println(this.getFieldName()+"
                  value changed to = "+this.getValue().toString());
       }
}


public void addObserver(Observer o){
       this.observable.addObserver(o);
}


public abstract void refreshValue();


public void run(){
       while(true){
              this.refreshValue();
              try{
```

```
                        this.sleep(this.getInterval());
                }
                catch(InterruptedException e){
                        System.out.println(e.getMessage());
                        this.refreshValue();
                }
        }
    }
}
```

D2. Sensor Factory class (SensorFactory.java):

```
package com.add.monitoring.sensorfactory;
import java.lang.reflect.Constructor;
public class SensorFactory extends Thread{
    Object defaultObject;
    ResourceManager resourceManager;
    Registry registry;
    /*
     * If sensors are outside the sensor factory package then the
SensorBasePath needs to be set
     */
    String sensorBasePath;
    public SensorFactory(){
        this(null, new ResourceManager(), new Registry());
    }
    public SensorFactory(Object defaultObject, ResourceManager
resourceManager, Registry registry){
        this.defaultObject = defaultObject;
        this.resourceManager = resourceManager;
        this.registry = registry;
```

```
            sensorBasePath = "";
        }
        public void setDefaultObject(Object defaultObject){
            this.defaultObject = defaultObject;
        }
        public void setResourceManager(ResourceManager resourceManager){
            this.resourceManager = resourceManager;
        }
        public void setRegistry(Registry registry){
            this.registry = registry;
        }
        public Sensor getSensorByName(String sensorName) throws SensorException {
            return this.getSensorByName(sensorName, this.defaultObject);
        }
        public void setSensorBasePath(String sensorBasePath){
            this.sensorBasePath = sensorBasePath;
        }
        public String getSensorBasePath(){
            return sensorBasePath;
        }
        public Sensor constructSensorByName(String sensorName, Object object)
throws SensorException {
            if(!this.sensorBasePath.equals("")){
                sensorName = this.sensorBasePath + "." + sensorName;
            }
            try{
                @SuppressWarnings("unchecked")
                Class<? extends ReflectiveSensor> sensorClass
                    = (Class<? extends ReflectiveSensor>)
Class.forName(sensorName);
```

```
                    Constructor<? extends Sensor> sensorConstructor
                            = sensorClass.getConstructor( new Class[]{
Object.class } );


                    return (Sensor) sensorConstructor.newInstance( new
Object[]{ object } );
            }
            catch(Exception e){
                    throw new SensorException("Error: Unable to construct
sensor " + e.getMessage());
            }
    }


    public Sensor getSensorByName(String sensorName, Object object) throws
SensorException {
            Throwable t = new Throwable();
            StackTraceElement[] elements = t.getStackTrace();
            String client =
elements[1].getClassName()+"."+elements[1].getMethodName();
            if(this.resourceManager.sensorAllowed()){
                    Sensor sensor = null;
                    if(this.registry.doesSensorExist(sensorName)){
                            sensor = this.registry.getSensor(sensorName);
                            this.registry.addClient(sensor, client);
                            return sensor;
                    }
                    else{
                            sensor = constructSensorByName(sensorName, object);
                            sensor.start();
                            this.registry.addEntry(sensor, client);
                    }
                    return sensor;
```

```
		}
		else{
			throw new SensorException("Error: Sensor not allowed");
		}
	}
}
```

D3. Registry class (Registry.java):

```
package com.add.monitoring.sensorfactory;
import java.util.*;


public class Registry{
	ArrayList<RegistryEntry> registryEntries;
	public Registry(){
		registryEntries = new ArrayList<RegistryEntry>();
	}
	public boolean doesSensorExist(String sensorName){
		for(int i=0; i<registryEntries.size(); i++){

	if(registryEntries.get(i).getSensor().getClass().getName().equals(sensorN
ame)){
				return true;
			}
		}
		return false;
	}
	public boolean doesSensorExist(Object object, String fieldName){
		for(int i=0; i<registryEntries.size(); i++){

	if(registryEntries.get(i).getSensor().getObject().equals(object) &&
registryEntries.get(i).getSensor().getFieldName().equals(fieldName)){
```

```
                    return true;
                }
            }
            return false;
        }
        public Sensor getSensor(String sensorName){
            for(int i=0; i<registryEntries.size(); i++){

        if(registryEntries.get(i).getSensor().getClass().getName().equals(sensorN
ame)){
                    return registryEntries.get(i).getSensor();
                }
            }
            return null;
        }
        public Sensor getSensor(Object object, String fieldName){
            for(int i=0; i<registryEntries.size(); i++){

        if(registryEntries.get(i).getSensor().getObject().equals(object) &&
registryEntries.get(i).getSensor().getFieldName().equals(fieldName)){
                    return registryEntries.get(i).getSensor();
                }
            }
            return null;
        }
        public int getIndexOfSensor(String sensorName){
            for(int i=0; i<registryEntries.size(); i++){
        if(registryEntries.get(i).getSensor().getClass().getName().equals(sensorN
ame)){
                    return i;
                }
            }
```

```
        return -1;
    }
    public int getIndexOfSensor(Object object, String fieldName){
        for(int i=0; i<registryEntries.size(); i++){

    if(registryEntries.get(i).getSensor().getObject().equals(object) &&
registryEntries.get(i).getSensor().getFieldName().equals(fieldName)){
                return i;
            }
        }
        return -1;
    }
    public Sensor getSensorAtIndex(int index){
        return registryEntries.get(index).getSensor();
    }
    public void addClient(Sensor sensor, String client){
        for(int i=0; i<registryEntries.size(); i++){
            if(registryEntries.get(i).getSensor().equals(sensor)){
                registryEntries.get(i).addClient(client);
            }
        }
    }
    public void addEntry(Sensor sensor, String client){
        registryEntries.add(new RegistryEntry(sensor, client));
    }
    class RegistryEntry{
        Sensor sensor;
        ArrayList<String> clients;
        public RegistryEntry(){
            this.clients =  new ArrayList<String>();
        }
```

```
        public RegistryEntry(Sensor sensor){
               this.sensor=sensor;
               this.clients =  new ArrayList<String>();
        }
        public RegistryEntry(Sensor sensor, String client){
               this.sensor=sensor;
               this.clients =  new ArrayList<String>();
               clients.add(client);
        }
        public void setSensor(Sensor sensor){
               this.sensor = sensor;
        }
        public void addClient(String client){
               clients.remove(client);
               clients.add(client);
        }
        public void removeClient(String client){
               clients.remove(client);
        }
        public boolean isClient(String client){
               return (clients.indexOf(client)!=-1);
        }
        public int numberOfClients(){
               return clients.size();
        }
        public Sensor getSensor(){
               return sensor;
        }
    }
}
```

D4. Threshold class (Threshold.java):

```
package com.add.decisionmaking.adaptationdetector;

public class Threshold {

	public static enum ThresholdType{

		GREATER_THAN,

		GREATER_THAN_OR_EQUAL,

		LESS_THAN,

		LESS_THAN_OR_EQUAL,

		EQUAL,

		NOT_EQUAL,

		IN_BETWEEN,

		IN_BETWEEN_INCLUSIVE,

		NOT_IN_BETWEEN

	}

	protected ThresholdType thresholdType;

	protected Object firstBoundary;

	protected Object secondBoundary;

	public Threshold(ThresholdType thresholdType, Object firstBoundary,
Object secondBoundary){

		this.thresholdType = thresholdType;

		this.firstBoundary = firstBoundary;

		this.secondBoundary = secondBoundary;

	}

	public Threshold(ThresholdType thresholdType, Object firstBoundary){

		this(thresholdType, firstBoundary, null);

	}

	public Object getFirstBoundary(){

		return firstBoundary;

	}
```

```
	public Object getSecondBoundary(){

		return secondBoundary;

	}

}
```

D5. Threshold Analyzer class (ThresholdAnalyzer.java):

```
package com.add.decisionmaking.adaptationdetector;

public class ThresholdAnalyzer{

	public static boolean analyze(Threshold threshold, Object value){

		double objectDoubleValue = Double.parseDouble(value.toString());

		double firstBoundaryDoubleValue =
		Double.parseDouble(threshold.getFirstBoundary().toString());

		double secondBoundaryDoubleValue;

		switch(threshold.thresholdType){

			case GREATER_THAN:

				return (objectDoubleValue >
firstBoundaryDoubleValue);

			case GREATER_THAN_OR_EQUAL:

				return (objectDoubleValue >=
firstBoundaryDoubleValue);

			case LESS_THAN:

				return (objectDoubleValue <
firstBoundaryDoubleValue);

			case LESS_THAN_OR_EQUAL:

				return (objectDoubleValue <=
firstBoundaryDoubleValue);

			case EQUAL:

				return (objectDoubleValue ==
firstBoundaryDoubleValue);

			case NOT_EQUAL:

				return (objectDoubleValue !=
firstBoundaryDoubleValue);

			case IN_BETWEEN:
```

```
                        secondBoundaryDoubleValue =
                        Double.parseDouble(threshold.getSecondBoundary().toSt
                        ring());

                        return (objectDoubleValue > firstBoundaryDoubleValue
&& objectDoubleValue < secondBoundaryDoubleValue);

                    case IN_BETWEEN_INCLUSIVE:

                        secondBoundaryDoubleValue =
                        Double.parseDouble(threshold.getSecondBoundary().toSt
                        ring());

                        return (objectDoubleValue >= firstBoundaryDoubleValue
&& objectDoubleValue <= secondBoundaryDoubleValue);

                    case NOT_IN_BETWEEN:

                        secondBoundaryDoubleValue =
                        Double.parseDouble(threshold.getSecondBoundary().toSt
                        ring());

                        return !(objectDoubleValue > firstBoundaryDoubleValue
&& objectDoubleValue < secondBoundaryDoubleValue);

                }

                return false;

        }

}
```

D6. Abstract Adaptation Detector class (AdaptationDetector.java):

```
package com.add.decisionmaking.adaptationdetector;


import com.add.decisionmaking.casebasedreasoning.InferenceEngine;

import com.add.monitoring.sensorfactory.SensorException;

import com.add.monitoring.sensorfactory.SensorFactory;

public abstract class AdaptationDetector extends Thread {

        private final InferenceEngine inferenceEngine;

        private final SensorFactory sensorFactory;


        /**

         * Constructs a new AdaptationDetector object.
```

```
	 *
	 * @param inferenceEngine
	 * @param sensorFactory
	 * @throws SensorException
	 */
	public AdaptationDetector(InferenceEngine inferenceEngine, SensorFactory 
sensorFactory)
			throws SensorException{
		this.inferenceEngine = inferenceEngine;
		this.sensorFactory = sensorFactory;
		this.createObservers();
		this.createThresholds();
	}
	/**
	 * Instantiates sensors and observers and attach observers to the sensors
	 */
	public abstract void createObservers() throws SensorException ;
	/**
	 * Instantiates the thresholds
	 */
	public abstract void createThresholds();

	/**
	 * Adaptation logic
	 */
	public abstract void adapt();
	/* (non-Javadoc)
	 * @see java.lang.Thread#run()
	 */
	public void run() {
```

```
        System.out.println("Adaptation Detector started");
        while(true) {
            this.adapt();
            try {
                AdaptationDetector.sleep(1000);
            }
            catch(InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
    /**
     * Get the inferenceEngine
     *
     * @return the inferenceEngine
     */
    public InferenceEngine getInferenceEngine() {
        return inferenceEngine;
    }

    /**
     * Get the sensorFactory
     *
     * @return the sensorFactory
     */
    public SensorFactory getSensorFactory() {
        return sensorFactory;
    }
}
```

D7. Inference Engine class (InferenceEngine.java):

```
package com.add.decisionmaking.casebasedreasoning;

import com.add.reconfiguration.gamereconfiguration.*;

import java.util.*;


public class InferenceEngine extends Thread{

	ArrayList<Trigger> triggerPool;

	public FixedRules fixedRules;

	AdaptationDriver adaptationDriver;

	public InferenceEngine(AdaptationDriver adaptationDriver){

		this.triggerPool = new ArrayList<Trigger>();

		this.fixedRules = new FixedRules();

		this.adaptationDriver = adaptationDriver;;

	}

	public void notifyTrigger(Trigger trigger){

		System.out.println("Inference Engine Got Notified With A
Trigger");

		this.triggerPool.add(trigger);

	}

	public void implementDecision(Decision decision){

		System.out.println("implement decision "+decision.getName());

		this.adaptationDriver.implementDecision(decision);

	}

	public void run(){

		while(true){

			System.out.println("Inference Engine is waiting for
trigger");

			System.out.println("Inference Engine triggerPool
Size="+this.triggerPool.size());

			if(!this.triggerPool.isEmpty()){

				System.out.println("Inference Engine got a trigger");
```

```
                        Decision decision =
                        this.fixedRules.getDecision(this.triggerPool.get(0));

                        decision.setContextualInformation(this.triggerPool.ge
                        t(0).getContextualInformation());

                        this.triggerPool.remove(0);

                        System.out.println("Trigger removed from pool in
inference engine");

                        System.out.println("Going to implement
                        decision"+decision.getName());

                        this.implementDecision(decision);

                }

                try{

                        Thread.sleep(1000);

                }

                catch(InterruptedException e){

                        System.out.println(e.getMessage());

                }

        }

    }

}
```

D8. Decision class (Decision.java):

```
package com.add.decisionmaking.casebasedreasoning;

import java.util.*;

public abstract class Decision {

    String name;

    HashMap<String, Object> contextualInformation;

    public HashMap<String, Object> subDecisions;

    public Decision(String name){

        this.name = name;

        this.contextualInformation = new HashMap<String, Object>();
```

```
		this.subDecisions = new HashMap<String, Object>();

	}

	public void setName(String name){

		this.name = name;

	}

	public String getName(){

		return this.name;

	}

	public HashMap<String, Object> getContextualInformation(){

		return contextualInformation;

	}

	public void setContextualInformation(HashMap<String, Object>
contextualInformation){

		this.contextualInformation = contextualInformation;

	}

	public Object getContextualInformation(String informationPath){

		return contextualInformation.get(informationPath);

	}

	public void setContextualInformation(String informationPath, Object
value){

		this.contextualInformation.put(informationPath, value);

	}

	public Iterator<Map.Entry<String, Object>> getDecisionsIterator(){

		Set<Map.Entry<String, Object>> decisionsSet =
subDecisions.entrySet();

		return decisionsSet.iterator();

	}

	/**

	 * Defines and adds the decision's component effects to the subDecisions
map.

	 */

	public abstract void compileDecision();
```

```
}
```

D9. State interface (State.java):

```
package com.add.reconfiguration.gamereconfiguration;

public interface State {

	public void makeInactive();

	public void makeActive();

	public int getState();

	public boolean isActive();

	public boolean isInactive();

}
```

D10. Driver class (Driver.java):

```
package com.add.reconfiguration.gamereconfiguration;

import java.lang.reflect.Array;

import java.lang.reflect.Field;


public class Driver{

	State stateObject;

	Object object;

	public Driver(State stateObject){

		this.stateObject = stateObject;

	}

	public void update(String attributePath, Object value){

		/*

		while(stateObject.isInactive()){

		}

		while(!stateObject.isInactive()){

			stateObject.makeInactive();
```

```
            }
            */
            System.out.println("State object made inactive");
            object = stateObject;
            Class<?> objectClass = this.object.getClass();
            String[] objectPath = attributePath.split("\\.");
            Field field = null;
            for(int i=0; i<objectPath.length; i++){
                    try{
                            if(!isInteger(objectPath[i])){
                                    field =
getDeclaredOrInheritedField(objectClass, objectPath[i]);
                            }
                            if(i<objectPath.length-1){
                                    field.setAccessible(true);
                                    this.object = field.get(this.object);
                                    while(i<objectPath.length-2 &&
isInteger(objectPath[i+1])){
                                            this.object =
Array.get(this.object,Integer.parseInt(objectPath[i+1]));
                                            i++;
                                    }
                                    objectClass = this.object.getClass();
                                    field.setAccessible(false);
                            }
                    }
                    catch(Exception e){
                            System.out.println("Exception1: "+e.getMessage());
                            e.printStackTrace();
                    }
            }
```

```
            String fieldName = objectPath[objectPath.length-1];
            try{
                  if(isInteger(fieldName)){
                        System.out.println("Field is an index");
                        Array.set(this.object, Integer.parseInt(fieldName),
value);
                  }
                  else {
                        System.out.println("Field is an attribute");

                        if(field.isAccessible()){
                              System.out.println("Field is accessible");
                              field.set(this.object, value);
                        }
                        else{
                              System.out.println("Field is not accessible");
                              field.setAccessible(true);
                              field.set(this.object, value);
                              field.setAccessible(false);
                              System.out.println("Field value modified");
                        }
                  }
            }
            catch(Exception e){
                  System.out.println("Exception: "+e.getMessage());
            }
      }
      public Field getDeclaredOrInheritedField(Class<?> c, String fieldName){
            try{
                  return c.getDeclaredField(fieldName);
```

```
			}
			catch(Exception e1){
				try{
					return c.getField(fieldName);
				}
				catch(Exception e2){
					return getDeclaredOrInheritedField(c.getSuperclass(),
fieldName);
				}
			}
		}
		public boolean isInteger(String str) {
				if (str == null) {
					return false;
				}
				int length = str.length();
				if (length == 0) {
					return false;
				}
				int i = 0;
				if (str.charAt(0) == '-') {
					if (length == 1) {
							return false;
					}
					i = 1;
				}
				for (; i < length; i++) {
					char c = str.charAt(i);
					if (c <= '/' || c >= ':') {
							return false;
```

```
                    }
                }
            return true;
        }
    }
```

Appendix E: Source Code Generated by the Proof-of-concept Automation Tool

E1. AverageScoreSensor

```
import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector.
*;
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;
import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;


public class AverageScoreSensor extends Sensor{

	public AverageScoreSensor(Object object){
		this.object = object;
		this.fieldName = "AverageScoreSensor";
		this.setInterval(1000);
		this.setValue(0);
	}

	public void refreshValue(){

		try{
			int score =
(int)(((com.brackeen.javagamebook.state.ScoreManager)(((TileGameState)object).s
coreManager)).score);

			int level =
(int)(((com.brackeen.javagamebook.state.ScoreManager)(((TileGameState)object).s
coreManager)).level);

			double value = score/level;
			this.setValue(value);
		}
		catch(Exception ex){
			System.out.print("Exception in Sensor:
AverageScoreSensor:"+ex.getMessage());
			this.setValue(0);
		}

	}
}
```

E2. AdaptationDetector

```
package
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector;


import com.game.designpattern.autodynamicdifficulty.monitoring.sensorfactory.*;

import
com.game.designpattern.autodynamicdifficulty.decisionmaking.adaptationdetector.
*;
```

```
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;

import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;


public class AdaptationDetector extends Thread{


       InferenceEngine inferenceEngine;

       SensorFactory sensorFactory;


       GenericObserver averagescoreobserverObserver;


       Threshold lowaveragescorethresholdThreshold;

       Threshold mediumaveragescorethresholdThreshold;

       Threshold highaveragescorethresholdThreshold;


       public AdaptationDetector(InferenceEngine inferenceEngine, SensorFactory
sensorFactory){

              this.inferenceEngine = inferenceEngine;

              this.sensorFactory = sensorFactory;

              this.createObservers();

              this.createThresholds();

       }

       public void createObservers(){

            averagescoreobserverObserver = new GenericObserver() ;


            Sensor averagescoresensorSensor =

                 sensorFactory.getSensorByName("AverageScoreSensor");


            averagescoresensorSensor.addObserver(averagescoreobserverObserver);


       }
```

```
	public void createThresholds(){

		lowaveragescorethresholdThreshold =
			new Threshold(ThresholdType.LESS_THAN, 30 );

		mediumaveragescorethresholdThreshold =
			new Threshold(ThresholdType.IN_BETWEEN_INCLUSIVE, 30 ,60);

		highaveragescorethresholdThreshold =
			new Threshold(ThresholdType.GREATER_THAN, 60 );

	}



	public void run(){


		while(true){



			if(averagescoreobserverObserver.isRecentlyUpdated()){


	if(ThresholdAnalyzer.analyze(lowaveragescorethresholdThreshold,
averagescoreobserverObserver.getValue())){

					System.out.println("observer value
"+averagescoreobserverObserver.value.toString()+" considered easy");

					Trigger makeleveleasytriggerTrigger = new
Trigger("makeLevelEasyTrigger");


	inferenceEngine.notifyTrigger(makeleveleasytriggerTrigger);

				}

			}

			if(averagescoreobserverObserver.isRecentlyUpdated()){


	if(ThresholdAnalyzer.analyze(mediumaveragescorethresholdThreshold,
averagescoreobserverObserver.getValue())){

					System.out.println("observer value
"+averagescoreobserverObserver.value.toString()+" considered medium");

					Trigger makelevelmediumtriggerTrigger = new
Trigger("makeLevelMediumTrigger");


	inferenceEngine.notifyTrigger(makelevelmediumtriggerTrigger);

				}

			}

			if(averagescoreobserverObserver.isRecentlyUpdated()){


	if(ThresholdAnalyzer.analyze(highaveragescorethresholdThreshold,
averagescoreobserverObserver.getValue())){
```

```
                                System.out.println("observer value
"+averagescoreobserverObserver.value.toString()+" considered difficult");

                                Trigger makeleveldifficulttriggerTrigger = new
Trigger("makeLevelDifficultTrigger");


      inferenceEngine.notifyTrigger(makeleveldifficulttriggerTrigger);

                        }

                  }


                  try{

                        this.sleep(1000);

                  }

                  catch(InterruptedException e)

                  {

                        System.out.println("Exception in Adaptation Detector
: "+e.getMessage());

                  }



            }

      }


}
```

E3. GameInferenceEngine

```
import
com.game.designpattern.autodynamicdifficulty.decisionmaking.casebasedreasoning.
*;

import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;



public class GameInferenceEngine extends InferenceEngine{



      AdaptationDriver adaptationDriver;



      public GameInferenceEngine(AdaptationDriver adaptationDriver){

            super();

            this.adaptationDriver = adaptationDriver;
```

```
		this.fixedRules.addRule(new Rule("MakeLevelEasy", new
Trigger("makeLevelEasyTrigger"), new MakeLevelEasyDecision()));

		this.fixedRules.addRule(new Rule("MakeLevelMedium", new
Trigger("makeLevelMediumTrigger"), new MakeLevelMediumDecision()));

		this.fixedRules.addRule(new Rule("MakeLevelDifficult", new
Trigger("makeLevelDifficultTrigger"), new MakeLevelDifficultDecision()));


	}


	public void run(){

		super.run();

	}


	public void implementDecision(Decision decision){

		System.out.println("implement decision "+decision.getName());

		this.adaptationDriver.implementDecision(decision);

	}

}
```

E4. Decisions

```
import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;


public class MakeLevelEasyDecision extends Decision{


	public MakeLevelEasyDecision(){

		super("MakeLevelEasyDecision");

	}


	public void compileDecision(){

		subDecisions.put("scoreManager.mapDifficulty", "easy");

		System.out.println("Decision MakeLevelEasyDecision is being
execute!");

	}

}
```

```
import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;


public class MakeLevelMediumDecision extends Decision{


       public MakeLevelMediumDecision(){

              super("MakeLevelMediumDecision");

       }


       public void compileDecision(){

              subDecisions.put("scoreManager.mapDifficulty", "medium");


              System.out.println("Decision MakeLevelMediumDecision is being
execute!");

       }

}

import
com.game.designpattern.autodynamicdifficulty.reconfiguration.gamereconfiguratio
n.*;


public class MakeLevelDifficultDecision extends Decision{


       public MakeLevelDifficultDecision(){

              super("MakeLevelDifficultDecision");

       }


       public void compileDecision(){

              subDecisions.put("scoreManager.mapDifficulty", "difficult");


              System.out.println("Decision MakeLevelDifficultDecision is being
execute!");

       }

}
```

Curriculum Vitae

Name:	Muhammad Iftekher Chowdhury
Post-secondary Education and Degrees:	North South University Dhaka, Bangladesh 2003 - 2006 BS The University of Western Ontario London, Ontario, Canada 2008 - 2009 MS The University of Western Ontario London, Ontario, Canada 2010 - 2014 Ph.D.
Honors and Awards:	Western Graduate Research Scholarship 2008 - 2009, 2010 - 2012
Related Work Experience	Game Developer Infrablue Technology 2004 – 2004 Software Development Engineer Amazon 2013 - 2014

Publications:

M. I. Chowdhury and M. Katchabaw, "Software Design Patterns for Enabling Auto Dynamic Difficulty in Video Games," in *17th International Conference on Computer Games (CGAMES'12)*, Louisville, Kentucky, USA, pp. 76-80, 2012.

M. I. Chowdhury and M. Katchabaw, "Improving software quality through design patterns : a case study of adaptive games and auto dynamic difficulty," in *Game-ON 2012*, 2012.

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

M. I. Chowdhury and M. Katchabaw, "Bringing auto dynamic difficulty to commercial games: A reusable design pattern based approach," in *18th International Computer Games (CGames'13)*, Louisville, KY, USA, pp. 103-110, 2013.