IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

An Approach to Auto Energetic Issues in Video Games Based on Program Plan Design

Santosh Shukla
IET, Dr RML Avadh University Ayodhya UP

Er. Ashish Kumar Pandey
Assistant Professor, Department of CSE, IET . Dr RL Avadh University Ayodhya UP

Er. Shobhit Srivastava
Assistant Professor, Department of CSE, IET . Dr RL Avadh University Ayodhya UP

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, andmaintainable.

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.

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 isthe 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

1 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.

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.

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.

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.*

Research Studies

We have organized four different studies to achieve the above research goal. Ourintention 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 specificgame. 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.

Study-2:	**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.
Study-3:	**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.

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]; Orviset 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

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.

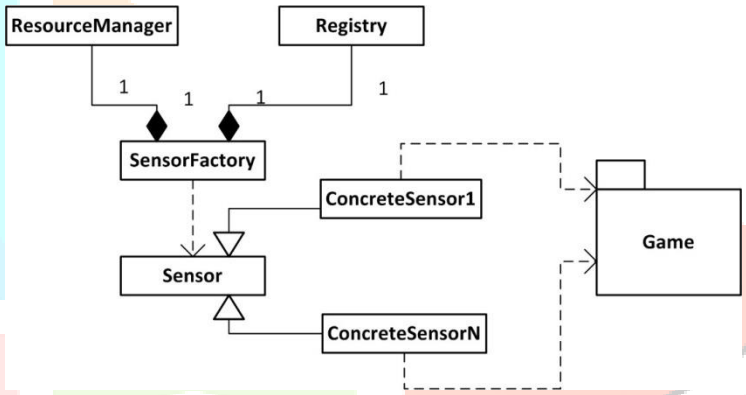

Figure 1: *Sensor factory* design pattern

***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

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);
```

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 aResourceManager 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.

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.

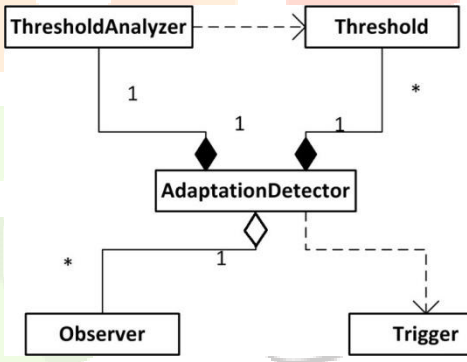

Figure 2: *Adaptation Detector* design pattern

***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*.

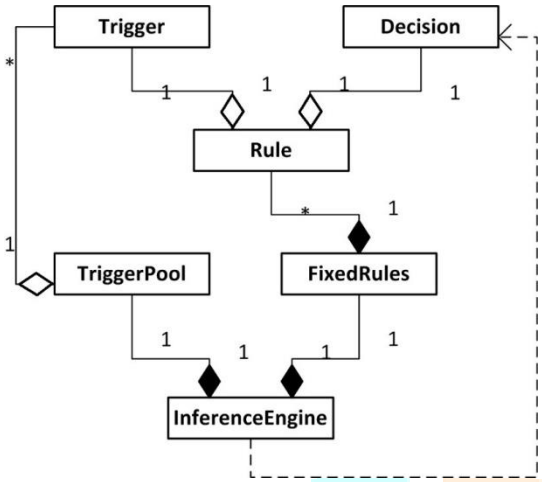

Figure 3: *Case based reasoning* design pattern

***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* anda *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*

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.

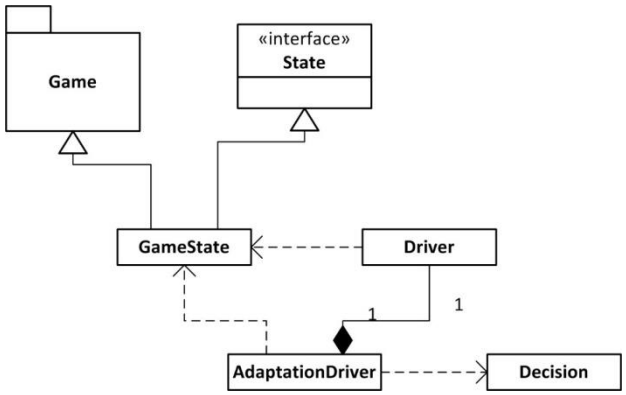

Figure 4: *Game reconfiguration* design pattern

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 *Request Buffer* becomes empty. The game can 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.

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).

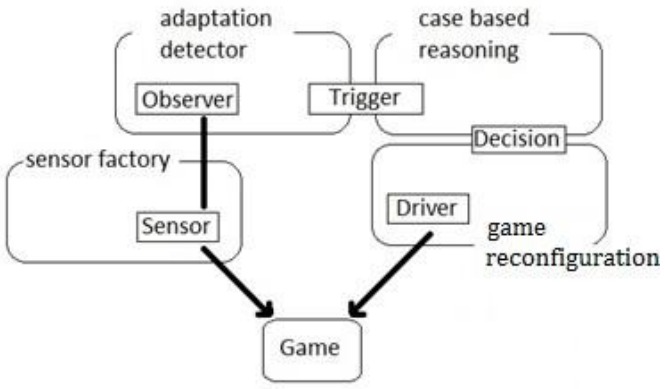

Figure 5: ADD design patterns working together **Table 4: Summary of ADD design patterns**

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 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".

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

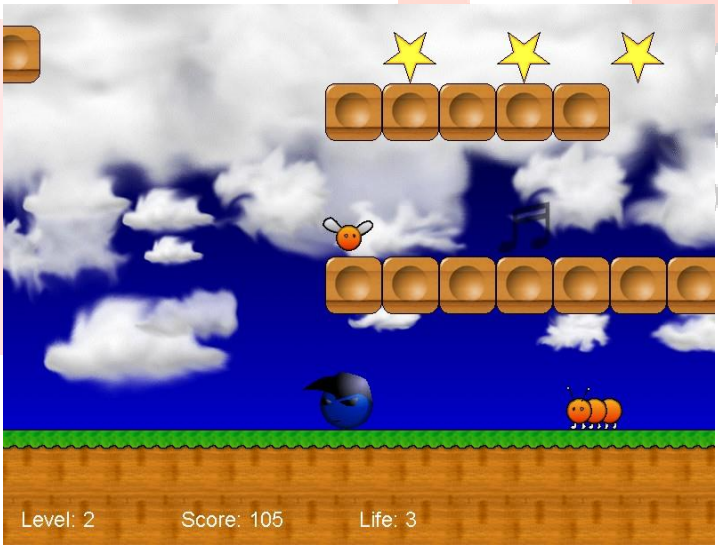

Figure 7: Screen captured from the TileGame game

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.

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.

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 firingat 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

[5] 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 bombat 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.

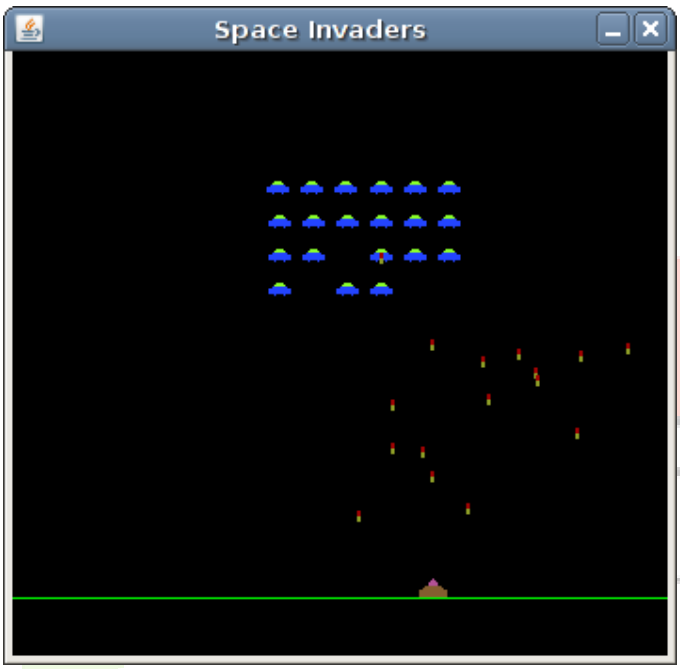

Figure 9: Screen captured from the Space Invaders game

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)

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 adaptrespectively, which is part of a possible way of eliciting essential requirements foran 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, Numberof times playerdies	Load different versions of the map where default objects and enemies are placed in slightly differentpositions.	Load different versions of the map when the player character goes to thenext level or in the next loading of thesame level (i.e., when the player character dies) based on scores and life lost in last level.
Minecraft	Which day in game, numberof times player dies	Display hints about collecting resources and building shelters	If the player is continuously dying during the first night, give the playersome hints to progress through the game to make it easier.
	Number of items of particular materials in players inventory	Hardness of those particularitems	Modify the hardness of a particular resource in the game world as the player's inventory of that particularitem changes, making it easier or harder to collect the resource.

6 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 quicklyand give desirable shapes for the opposite.
	Height of the stack	Speed of the shapes	Descend the shapes faster if the stackis 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 andtheir distance from the ground.
		Speed of player's missile	

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 implementationsto 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.

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 ashelter 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 ismore 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 ofthe 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.

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 aclass, 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 implementationis 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:

n

$$WMC = \sum_{i=1} Ci$$

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].

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.

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 toADD, and demonstrates significant potential for commercial applications.

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 creatingADD 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 andsimple enough to be easily understood and reshaped accordingly. Applying oursoftware 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.

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**	

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.

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.

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

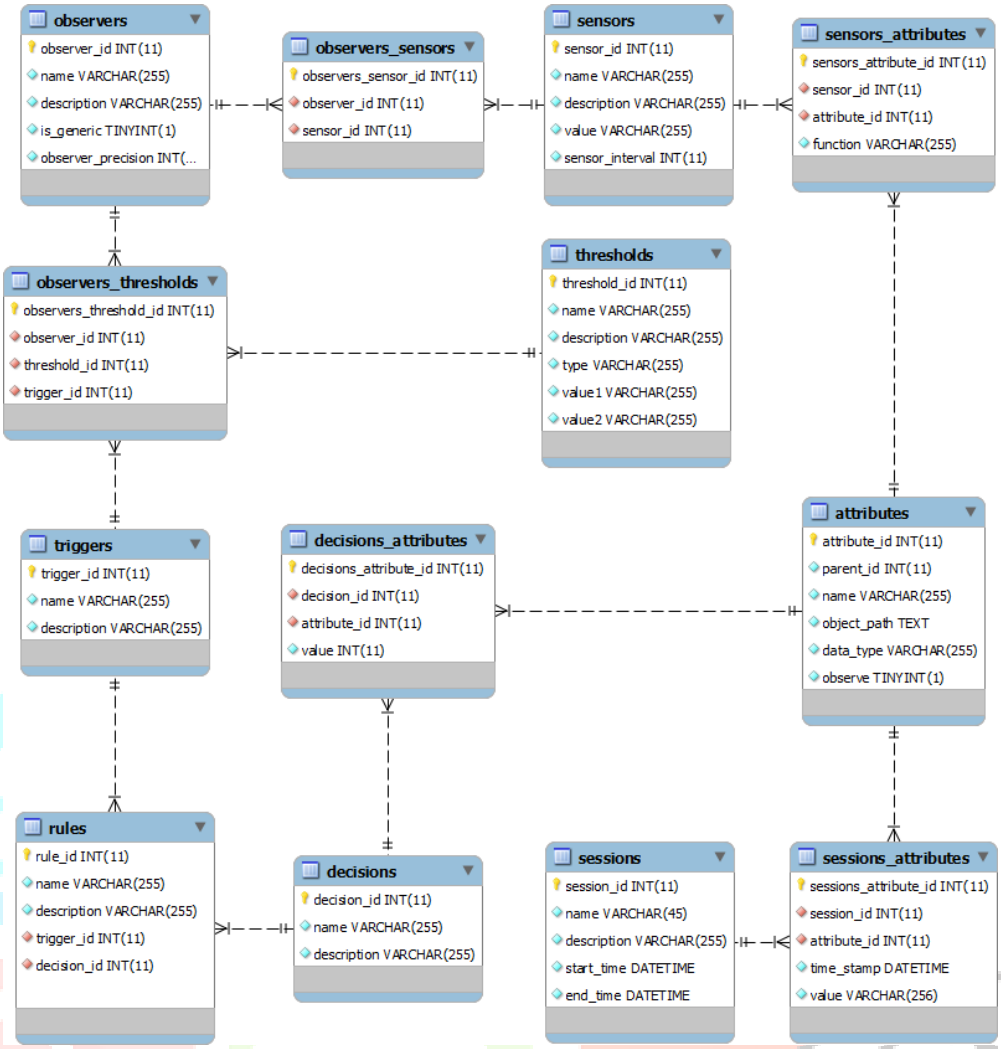

Figure 12: Schema of the MySQL database for the relational model

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

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 usermanual 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 ofa Collection) and no function respectively. In doing

so, the value in the sensor shouldbe 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 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.

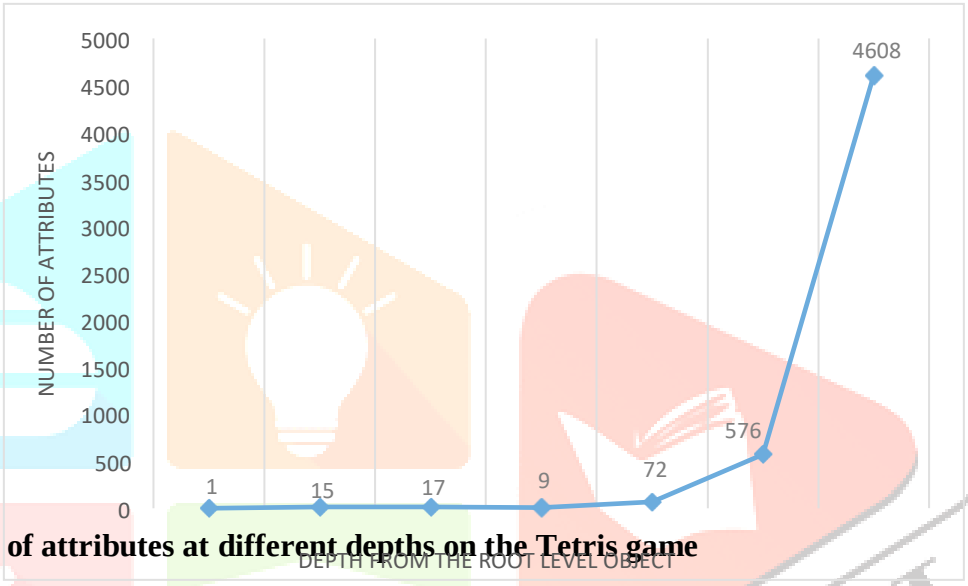

Figure 13 : Number of attributes at different depths on the Tetris game

In Figure 14, we show screenshot of the attribute tree visualization where all theattributes 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.

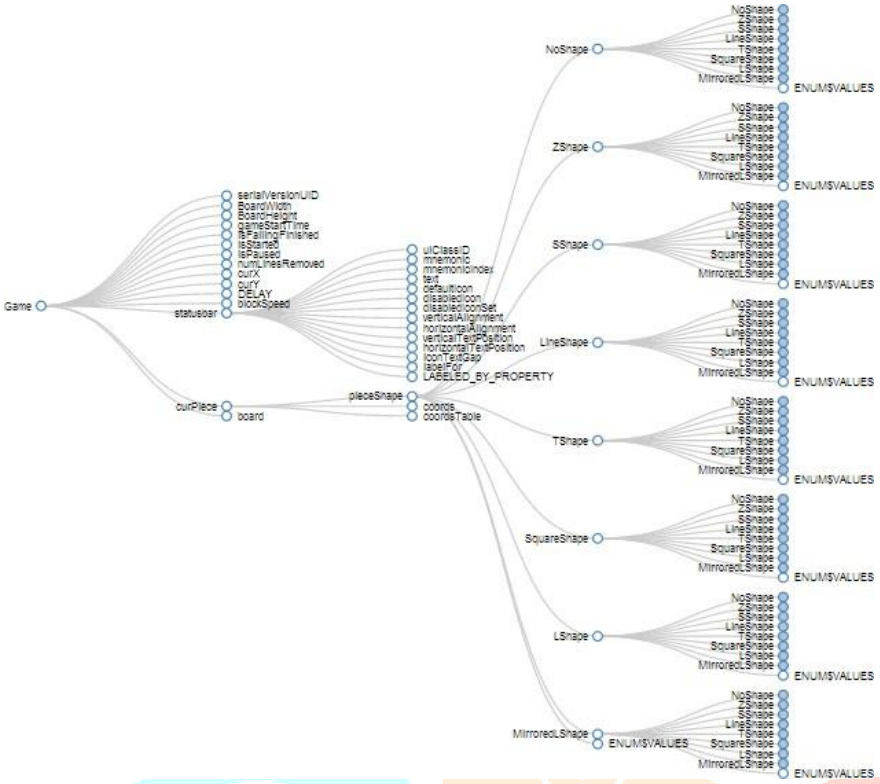

Figure 14 : Screenshot of attribute tree visualization for the Tetris game

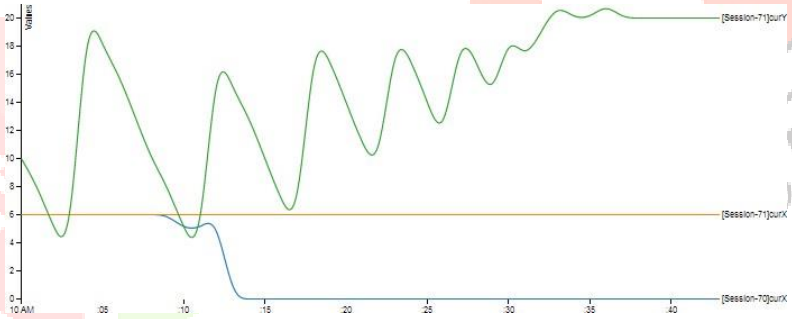

Figure 15 : Screenshot of a sample session timeline visualization

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 conceptand does not define the limits of the actual framework described in Section 7.1.

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 gameobject 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 ofattributes:

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.

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 originalsource 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.

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 gainsufficient 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.

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 tothe 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 thecomplete 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 surveyquestionnaire, each based on one of the games, were collected.

- Critical Review: The participant was asked to provide a critical review of thedesign patterns and the base level implementation based on his experience.

- Developer Log: A brief description of the activities and associated effort toimplement the adaptations scenarios on top of the games.

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 Invadersgame

Design Patterns	Ease of usage (1= extremely easy; 5 = extremelydifficult)					
	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 SpaceInvaders 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 testingand debugging	4:00	0:05
Total	**8:45**	**3:20**

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:

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 theframework itself, but rather a product of issues with the implementation and documentation used."

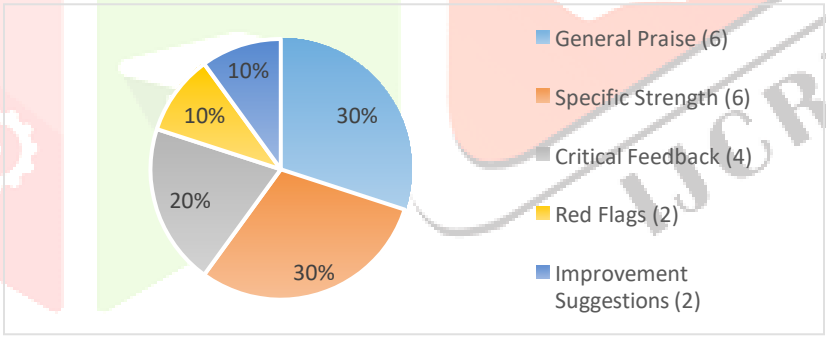

Figure 16 : Summary of participant's feedback about the design patterns and base level implementation

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

---

15 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.

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.

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.

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.
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.

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.

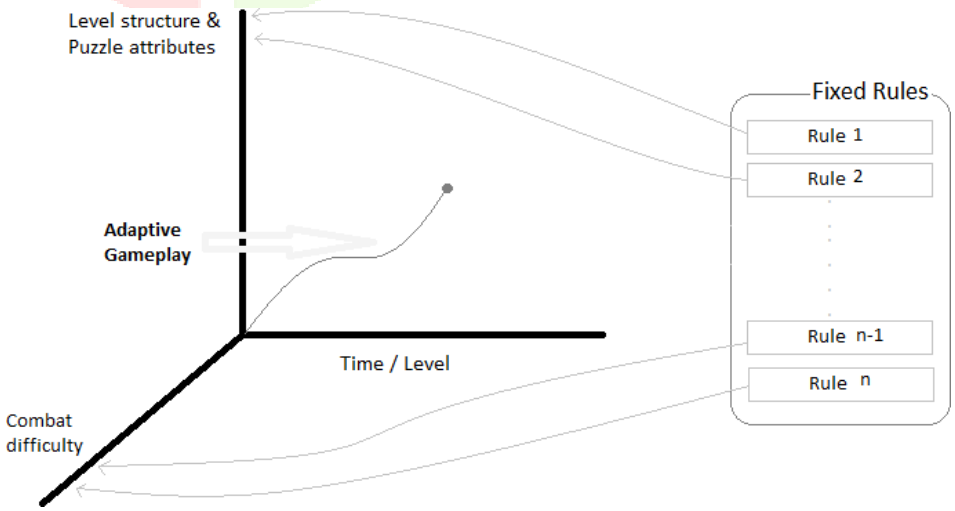

Figure 17: Concept of multi-dimensional adaptive gameplay

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.

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 *GameAI 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.