Analyzing Aspects of a Gaming Application

Cem AKSOY, Mücahid KUTLU, Bahri TÜREL
Department of Computer Engineering, Bilkent University
Bilkent 06800, Ankara, Turkey
{cem, mucahid, bahri}@cs.bilkent.edu.tr

Abstract

Gaming applications can be considered as one of the complex applications in software development. This is because for developing gaming applications we have to deal with many different concerns. Traditionally, gaming applications are developed using general purpose or specific languages that cannot modularize these concerns with their proposed abstraction mechanisms. In this paper, we propose an aspect-oriented approach to modularize these crosscutting concerns by using the explicit language abstractions defined by aspect-oriented programming (AOP). We identify the key concerns in an object-oriented (OO) game, Chicken Invaders, and model these as aspects. We report our observations on aspect-oriented (AO) game programming and discuss the lessons learned.

1. Introduction

Gaming applications have a wide range of usage area. PC and mobile games are used very frequently. Technological developments bring improvements to game properties day by day. Developing graphical engines and artificial intelligence components cause the games to become more complex. These bear design issues in gaming applications [2].

The significant improvement in computer games started after usage of high level languages for game implementations. High level languages helped the programmers to modularize the concerns in the games, which causes more reusable and maintainable codes to occur in programs. In video gaming applications OpenGL also is a good example of modularization of concepts. Object-oriented programming (OOP) is the one of the most important steps in these improvements. Abstraction components called, classes are now used to separate concerns specific to games such as collision detection and resource management [2].

Separation of concerns as pointed by Dijkstra in [3] is separating one aspect from the others and seeing the other concerns as irrelevant from the separated concern's point of view. In programming, this is applied by modularization. Each concern should be mapped to a single module and each module should be coherent in itself. In gaming applications, the complexity rises from the big number of objects and the interactions among them. Current abstraction mechanisms like OO cannot fully modularize the concerns that crosscut over multiple modules. As an example, collision detection usually requires some operations to follow it which crosscut over all the objects which may collide.

In OOP, trying to accomplish all gaming tasks ends up in scattered concerns and tangling code. Also, game engine usually becomes a large object which is responsible for many different tasks. This decreases the coherency in the modules. We aim to overcome these deficiencies by applying AOP to specified game, named "Chicken Invaders". We define the concerns and modularize those using aspects and decrease the load of game engine.

The paper is organized as follows: Section 2 shows the case study for application of AOP in detail. Section 3 follows with the specification of the problem and identification of the game specific aspects. Section 4 describes and illustrates the aspects implemented with AspectJ for solution of the problem. Section 5 explains aspect-oriented solution with JBoss as an alternative approach for AOP and Section 6 briefly indicates the related work in the literature. Finally, Section 7 concludes by discussing the effects of AOP on the case study.

2. A Case Study: Chicken Invaders

In this section, we describe a gaming application named Chicken Invaders that we use as the case study for our work. A sample screen view of the game can be seen in Figure 1.

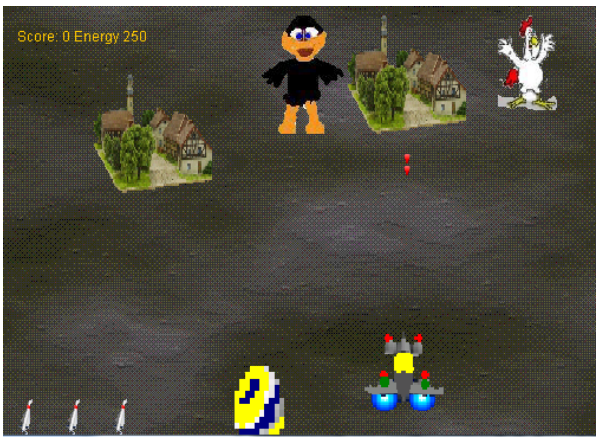

Figure 1. A snapshot from the game Chicken Invaders.

Chicken Invaders is a single player arcade game in which the player aims to complete the levels by destroying the enemy objects (chickens). The player uses an aircraft. The aircraft can use several weapons which have different destructive power and range. The enemy objects are classified into different categories such as ground, air and bonus objects. All these objects have different abilities, properties and vulnerabilities. Chickens come from above of the screen and move to down. The player tries not to have a crash with air objects and destroy as much as enemy objects in order to get a higher score. As the player shoots the enemy objects by using its bullets or bomb, objects may leave a bonus object for the aircraft which may heal the aircraft or give the ability to use a different type of weapon. The player also must be aware of the eggs of chickens which reduces aircraft's health. At the end of each stage, player has to defeat the boss enemy.

2.1. Object-Oriented Design

Chicken Invaders is a rich game in terms of objects so it is suitable for observing most of the properties of OOP paradigm.

In the proposed OO design of the game, there are two packages: GUI and GameComponents. GUI classes are responsible for user interface and GameComponents classes are GUI-independent representations of the game objects.

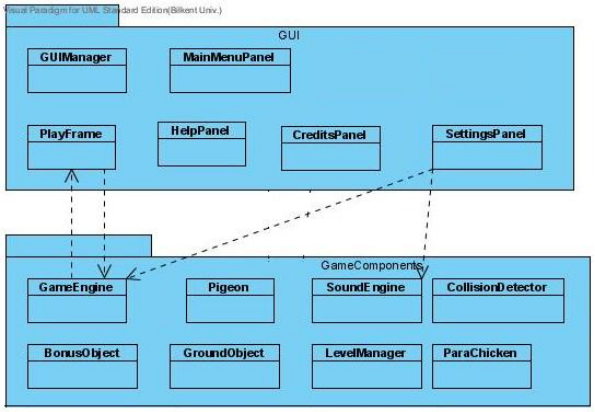

Figure 2. Package Dependency Diagram of Chicken Invaders, not including all classes.

As dependency details are shown in Figure 2, only a few classes from different packages depend on each other. During the game, main interaction between packages occurs between PlayFrame, GUI class of game play, and GameEngine. In each interval, GameEngine triggers PlayFrame to draw its objects. SettingsPanel changes static music/sound level attributes of SoundEngine and static game speed attribute of GameEngine. Thus, SettingPanel depends on those two objects.

As can be seen in class diagram of GameComponent package, Figure 3, GameEngine class is considered as the controller class. It has to control flow of the game and be connected to every class in order to co-ordinate them and provide connection between them. That is to say, GameEngine class has many responsibilities during the game which makes it the most complicated class. Responsibilities of GameEngine can be listed as the following:

- Getting new game objects from GameLevel class.
- Updating game objects positions.
- Checking if there is any crash between aircraft and chickens/bonus objects or between bullets and "ShootableObjects" by using CollisionDetector class.
- Running SoundEngine in proper times like after firing or any crash or sending rocket.
- Removing destroyed game objects and changing the score of the aircraft
- Removing game objects that are out of frame.

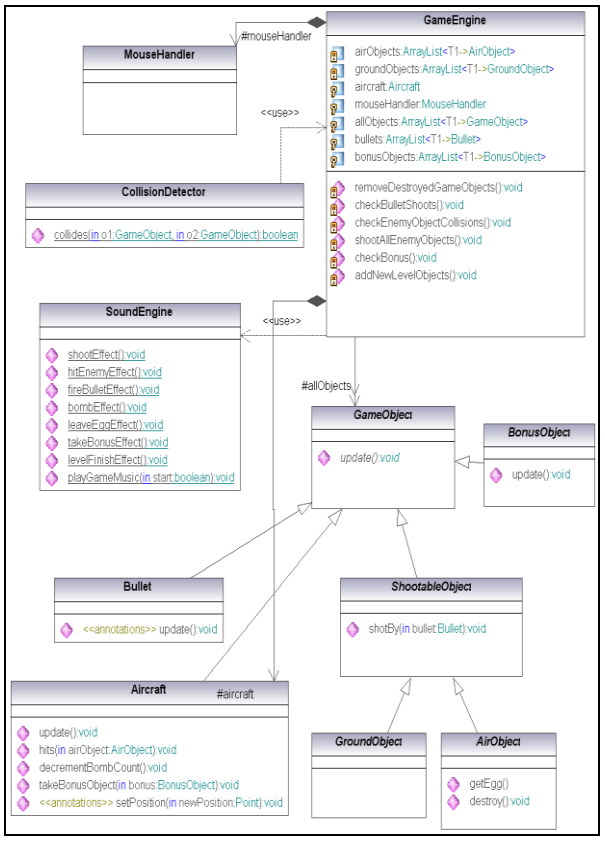

Figure 3. Class Diagram of Game Component Package. including important classes

- Setting bombs of enemy objects (eggs)
- Getting inputs from user via mouse.
- Setting speed of game and pausing the game.

All drawn objects on the GUI are part of a big hierarchy in which they all extend from a main class, GameObject. The intermediate level in this hierarchy consists of two classes, BonusObject and ShootableObject. These two are needed because of the different natures of two categories. BonusObjects cannot be shot by the aircraft, they disappear only when they move out of the frame or they collide with the aircraft. ShootableObject is also divided into two categories which are GroundObject and AirObject. As can be understood from the category names, the main distinction between these two categories is AirObject may collide with the aircraft where GroundObject cannot collide. They both can be shot by aircraft.

2.2. Applied Patterns

In OO design patterns play an important role to maintain easy development of software. In this section we describe design patterns which are used in Chicken Invaders.

2.2.1. Singleton Pattern. Singleton pattern is used to restrict instantiation of an object so that only one active instance of that class can be found on the system.

In Chicken Invaders, GameEngine is defined as a singleton object because it is only needed to be instantiated once. It has a lot of responsibilities over many classes, in other words it assures the game to flow. So as a coordinator, one GameEngine object is necessary and sufficient.

2.2.2. Visitor Pattern. The visitor pattern is used for separating operations from object structures. We can easily add new operations to those object structures without any modification.

In Chicken Invaders, we have applied visitor pattern to GameObject structure. In Figure 4, illustration of visitor pattern in Chicken Invaders is given. GameObjectUpdateVisitor and GameObjectShootVisitor are two visitor classes each of which implement GameObjectVisitor interface. GameObjectUpdateVisitor updates objects locations and GameObjectShootVisitor decreases health of visited object by a certain amount. By having visitor pattern, we can easily add new operations to GameObjects.

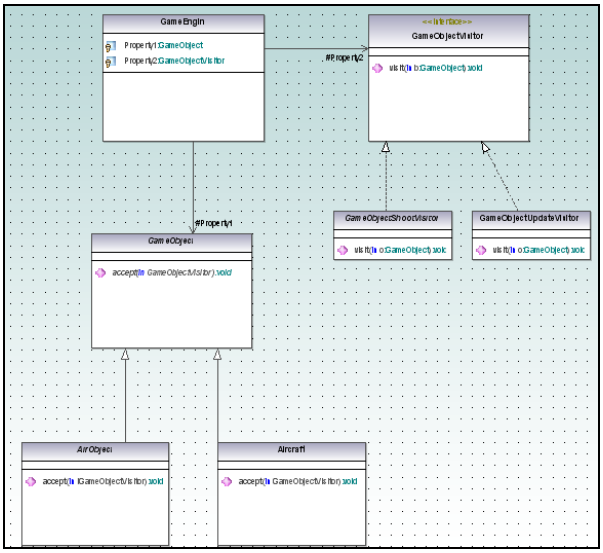

Figure 4. Illustration of visitor pattern in Chicken Invaders.

3. Identifying Crosscutting Concerns

In Section 1, we explained that in gaming applications, some concerns cannot be modularized and

they crosscut over multiple modules. Also, some objects may become non-coherent because of the big responsibility load. This results in a tangled code. In this section, we explore these problems encountered in OO design of Chicken Invaders.

3.1. Detecting Collisions

In Chicken Invaders, multiple objects may collide with each other, such as, aircraft and an air object or a bullet with a "ShootableObject". Detection of these collisions is significant for flow of the game.

These collisions are regularly checked by GameEngine after updating positions of all game objects. GameEngine uses `checkBulletShoots()` to detect collision of bullets and "ShootableObjects", `checkBonus()` to detect collision of aircraft and bonus objects and uses `checkEnemyObjectCollisions()` method to detect collision of aircraft and air objects. All these three methods use CollisionDetector class's `collides(GameObject,GameObject)` method. Handling concern of detection of collisions in GameEngine class causes low coherency for GameEngine which results tangled code.

Low coherency causes some maintenance issues such as finding the relevant part of code in a large code. For example, if all checks are completed by GameEngine it would be hard to find the relevant part in case of a change.

3.2. Removing Out-of-Frame Objects

Enemy objects, bonus packages move from up to down and bullets move from down to up. Whenever they get out of frame, we do not need to hold them in memory since we won't use them anymore. Therefore, after updating positions of game objects, except aircraft, we have to check whether they are out of frame or not. And the ones out of frame should be removed from memory.

GameEngine performs this action in its method `RemovedDestroyedGameObjects()`. This concern increases load of GameEngine.

3.3. Playing Sound

In Chicken Invaders, different sound effects are used for some specific actions like a crash, shooting an enemy object with a bullet or using a bomb etc. In addition, background music starts just after the game starts and continues to play during the execution time of the game.

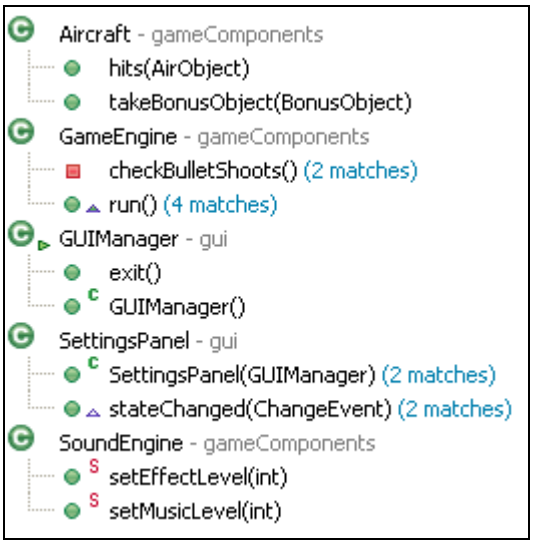

Figure 5. References to SoundEngine.

SoundEngine class coordinates all sound effects by using different methods for each type of sound. These methods are called from many different classes of the game which can be seen in Figure 5. Although, playing sound concern is modularized into a class, SoundEngine, because of these multiple calls from different classes, it becomes a scattering concern. These calls cause maintenance problems because in order to change a SoundEngine procedure, we need to visit all the modules that refer to that procedure. As a concrete example, to change `hitEnemyEffects()`, we may need to change `hits(AirObject)` method of Aircraft class. In simple changes it may not be required but for example if we want to completely remove sounds played after shooting enemy objects, we must change the corresponding code segment in `hits(AirObject)` method.

3.4 Throwing Bomb

In Chicken Invaders, aircraft throws a bomb which decreases health of all game objects that are currently in frame by a certain amount.

This concern is handled in GameEngine class with `shootAllEnemyObjects()` and in Aircraft class with `decrementBombCount()`. It is the responsibility of GameEngine to decrease health of all enemy objects following a bomb. All these decrements, like other tangling modules, reduces the coherency of GameEngine by increases its work load.

4. Aspect-Oriented Programming with AspectJ

In this section, we describe our aspect oriented solutions with AspectJ to crosscutting concerns which is mentioned in Section 3. In addition, two development aspects are described.

4.1 Detecting Collision Concern

To detect collision we need two advices, one for collisions of aircraft and air/bonus objects and other one is for collision of bullets and air/ground objects.

4.1.1 Collision of Aircraft and Air/Bonus Objects. The pointcut and advice for detecting collision of aircraft and air/bonus objects are shown in Figure 6. Pointcut selects joinpoints where position of aircraft is changed. By using `target()` and `this()` methods, we are able to get properties and functions of Aircraft and GameEngine classes. o In advice part, for each air object that is hold in GameEngine, we perform collision check(line 5). If there exists a collision, `hits(AirObject)` method of Aircraft is called (line 6) for decreasing health of Aircraft. Air object is destroyed (line 7). Same process is performed for bonus objects. Here, we call `takebonusObject(BonusObject)` method of Aircraft class (line 13) and remove the collided bonus object from memory(line 14-15).

```
1  pointcut updateInPositions( Aircraft a, GameEngine g):
2          target(a) && this(g) && call( void Aircraft.setPosition(..));
3  after(Aircraft a,GameEngine g):updateInPositions(a, g){
4      for (AirObject ao : g.airObjects){
5          if (CollisionDetector.collides(a, ao)){
6              a.hits(ao);
7              ao.destroy();
8          }
9      }
10     for (int bi = 0; bi < g.bonusObjects.size(); bi++){
11         BonusObject bo = g.bonusObjects.get(bi);
12         if (CollisionDetector.collides(bo, a)){
13             a.takeBonusObject(bo);
14             g.bonusObjects.remove(bo);
15             g.allObjects.remove(bo);
16         }
17     }
18 }
```

Figure 6. Pointcut and advice for collision of Aircraft and Air/Bonus Objects.

4.1.2 Collision of Bullet and Game Objects. The pointcut and advice for detecting collision of aircraft and air/bonus objects are shown in Figure 7. Pointcut selects joinpoints where position of bullet is changed. In advice, we check collision of bullet and for air and ground objects (line 21 and 26). If collision is found, we call `visit(GameObject)` method of GameObjectShootVisitor class(line 23 and 29).

```
17 pointcut bulletCollision(Bullet b, GameEngine g):
18     args(b) && this(g) && call(void GameObjectUpdateVisitor.visit(GameObject));
19
20 after( Bullet b, GameEngine g ):bulletCollision(b, g){
21     for (int i = 0; i < g.airObjects.size(); i++) {
22             if (CollisionDetector.collides(g.airObjects.get(i), b)) {
23                 visitor.visit(g.airObjects.get(i));
24
25             }
26     }
27     for (int i = 0; i < g.groundObjects.size(); i++) {
28             if (CollisionDetector.collides(g.groundObjects.get(i), b)) {
29                 visitor.visit(g.groundObjects.get(i));
30             }
31     }
32 }
```

Figure 7 Pointcut and advice for collision of Bullet and Game Objects.

By having these aspects, load of GameEngine is decreased. GameEngine does not deal with methods of Bullet objects and Aircraft. Interaction between CollusionDetector ends and so a better modularization is provided.

4.2 Removing Out-of-Frame Objects

The pointcut and advice for removing out-of-frame objects are shown in Figure 8. The pointcut selects joinpoints where position of any GameObject is changed. In advice, we get boundary of frame (line 6-

```
1  pointcut RemoveOutOfFrameObjects(GameEngine g,GameObject o) :
2      args(o) && this(g) && call ( void GameObjectUpdateVisitor.visit(..) );
3
4  after ( GameEngine g,GameObject o ) : RemoveOutOfFrameObjects(g,o){
5
6      double screenWidth = g.playFrame.getBounds().getWidth();
7      double screenHeight = g.playFrame.getBounds().getHeight();
8
9          Rectangle oRect = o.getRectangle();
10         if ((oRect.x + (oRect.width / 2)) < -10
11                 || (oRect.y + (oRect.height / 2)) < -10
12                 || ((oRect.y) - screenHeight) > 5
13                 || (oRect.x - screenWidth) > 5) {
14
15             g.allObjects.remove(o);
16             g.bonusObjects.remove(o);
17             g.airObjects.remove(o);
18             g.groundObjects.remove(o);
19         }
20 }
```

Figure 8. Pointcut and advice for Removing-Out-Of-Frame Objects.

7) and check if is out of frame (line 10) and remove the object from proper lists.(line 15-18). By having this aspect, we are now able to delete unused objects immediately. In bigger problems, deletion of these objects has higher priority because of filling memory space.

4.3 Playing Sound

There are several advices for playing sound aspect. The pointcut and advice for playing sound are shown in Figure 9. Advices that play sounds for firing a bullet, shooting an object, sending an egg, throwing a bomb, taking a bonus object are not shown for simplicity.

```
7   pointcut soundAircraft() : call (void Aircraft.hits(..));
8
9   pointcut aspectGUIManager() :  call (public gui.GUIManager.new() );
10
11  after():soundAircraft()
12  {
13      SoundEngine.hitEnemyEffect();
14  }
15
16  after() : aspectGUIManager()
17  {
18      SoundEngine.loadSounds();
19  }
```

Figure 9 Pointcut and advice for Playing Sound.

By having this aspect, we can easily manage playing sound concern like adding new sounds or canceling a sound without searching in codes. In addition, development gets easier because of better modularization.

4.4 Throwing Bomb

The pointcut and advice for throwing bomb are shown in Figure 10. The pointcut selects joinpoints where number of bombs of aircraft is decremented by method.

```
14  pointcut throwBomb(GameEngine e) :
15      this(e) && call ( void Aircraft.decrementBombCount() );
16
17  after(GameEngine e) : throwBomb(e)
18  {
19      for (int i = 0; i < e.airObjects.size(); i++) {
20          e.airObjects.get(i).shotBy(new Bomb());
21      }
22      for (int i = 0; i < e.groundObjects.size(); i++) {
23          e.groundObjects.get(i).shotBy(new Bomb());
24      }
25  }
```

Figure 10 Pointcut and advice for Throwing Bomb.

`decrementBomb()`. In advice part, we call `shotBy(Bomb)` method of each object in game(line 20-23). By having this aspect, load of GameEngine is decreased.

4.5 Development Aspects

Development aspects are used to help developers for understanding flow of software, such as tracing or logging. In this section, we describe two development aspects which we use to control flow of the program and detect the mistakes.

4.5.1 Checking Creation of Game Objects. Controlling creation of objects is significant for the developer since number of objects is high. The pointcut and advice for this concern are shown in Figure 11. Pointcut selects constructor joinpoints of each class which is a sub-class of GameObject. Advice method prints message for creation of the new object by using `thisjoinpoint`.

```
4  pointcut objectIsCreated(): call ( public GameObject+.new(..) );
5  after():objectIsCreated()
6  {
7      System.out.println(thisJoinPoint.toShortString()+" is created");
8  }
```

Figure 11. Pointcut and advice for Checking Creation of Game Objects

4.5.2 Checking Deletion of Game Objects. Controlling deletion of object is important in order to understand usage of memory. The pointcut and advice for this concern are shown in Figure 12. Pointcut selects joinpoints in calling `remove(GameObject)` methods in GameEngine class and RemoveOutOfframe aspect. In advice part, we give proper message about which object is deleted.

```
1  pointcut objectIsRemoved(GameObject g) :
2          args(g) && call(* *.remove(..)) &&
3          (within(gameComponents.GameEngine) ||
4           within(gameComponents.GamaAspect));
5  after(GameObject g): objectIsRemoved(g)
6  {
7      System.out.println(g.getClass() + " is removed");
8  }
```

Figure 12. Pointcut and advice for Checking Deletion of Game Objects.

5. Alternative Aspect Oriented Approach: JBoss

JBoss is a Java Aspect-oriented framework. JBoss and AspectJ both share similar capabilities and semantics. They only vary in syntax and implementation approaches. [5] There is our aspect-oriented solution code with JBoss for playing sound concern in Figure 13. Since only syntax for pointcuts is a little different than AspectJ, we do not mention

again all our aspects. In XML files we define our pointcuts. In Figure 13.a, we get joinpoints where `hits(GameObject)` method of Aircraft is called. Program calls AircraftCrash method of JBossSound class. In Figure 13.b, program calls `hitEnemyEffect()` method of SoundEngine class and returns.

a)

```
<bind pointcut="call(void gameComponents.Aircraft->;hits(..) )">
    <after aspect="src.gameComponents.JBossSound" name="AircraftCrash"/>
</bind>
```

b)

```
public Object AircraftCrash( Invocation invocation  ) throws Throwable
{
    SoundEngine.hitEnemyEffect();
    return invocation.invokeNext();
}
```

Figure 13. Pointcut and Advice for playing sound after crash of aircraft. a, pointcut in XML file, b, advice code in Java.

We can change our program without recompilation by using JBoss since pointcuts are defined in XML files. In case of having a bug in a game, we can debug by changing our advices or at least deactivate bugged module by deleting proper advice from XML.

6. Related Work

There exist few studies on the effects of AOP on gaming applications. In [2], general problems of game developments are mentioned. It explains the crosscutting concerns encountered while building a game engine (See Figure 14. for a sample crosscutting problem in a gaming application). As we mentioned in problem statement part, trying to insert many tasks into a single class as a harmony is not possible. Sound management and Collision Detection are also classifies as crosscutting as an out study. They also point to new paradigms like AOP are trying to solve these problems but there is not an explicit example.

The author in [3] applies AOP to an OO game called Invaders. Aspect is applied to a single concern and its effect on the speed of the game is evaluated. No significant overhead on the frame per second of the game is reported.

Alves et al.[1] discuss AOP's effect on mobile games which are wanted to be system independent. They are trying to reduce power cost of mobile phones applying some different methodologies.

As a result of these related works, we see that AOP's effects on gaming applications are not fully explored yet.

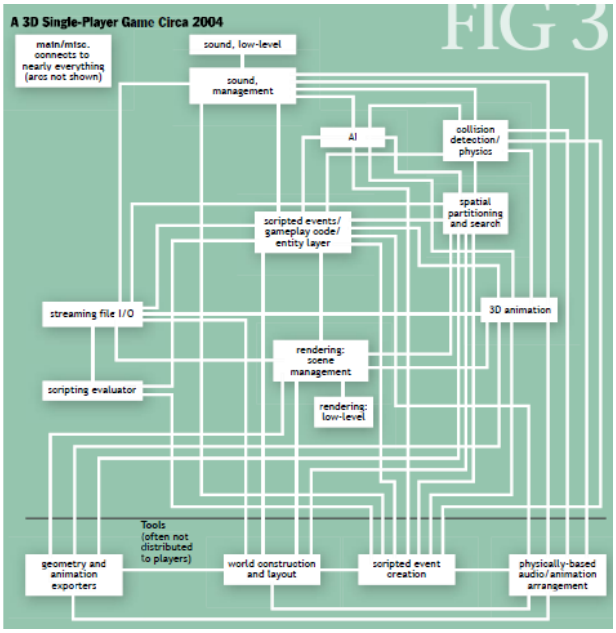

Figure 14. Dependency of Modules in a 3D Single-Player Game Circa 2004. [1]

7. Conclusion

In this paper, we first analyze the main concerns of gaming applications and claim that current abstraction mechanisms like classes are not sufficient to modularize these. We then conduct a case study on an OO game, Chicken Invaders, to identify the crosscutting concerns of the game and then create an AO design by modularizing these concerns into aspects.

We identify the crosscutting concerns of the game as detecting collision, removing out-of-frame objects, playing sound and throwing bomb. The concerns except throwing bomb are common in most of the games. Collision detection and the updates following detected collisions are very frequent in games with object interactions. Sound effects are also a part of almost all of the games. Finally, handling inactive objects is also a general concern of most of the games with moving objects. So, we can say that this case study is illustrative in terms of all games.

We show that all these concerns can be implemented by using aspects to reduce problems rising from crosscutting concerns. We use change scenarios for scattered concerns like SoundEngine to prove it decreases maintainability of the code. Tangling

[1] Adapted from [2]

ones are also shown to decrease coherency of GameEngine.

We implement aspects to handle these concerns by eliminating their actual implementations in OOP. So we modularize these concerns into aspects. This causes increase in coherency of GameEngine and fully modularize SoundEngine. SoundEngine is now easily maintainable because changes in one aspect would be enough to change, remove or add new sound operations. Additionally, for removing out-of-frame objects, AOP style is more realistic because it removes each object just after their position is updated instead of updating all objects and then removing the out-of-frame ones. This may cause some memory save especially in big games.

Development aspects' effects to game development are also examined. Especially, in big games development aspects are proved to be effective for tracing operations like object creation. Usage of these aspects eases the development process.

We demonstrate that some crosscutting concerns specific to most of the games may be eliminated by using AOP. By using AOP, both maintainability and modularity of the code increases. Since most of the concerns are common to all games, we think that AOP usage in gaming applications should be explored further. Some comprehensive study on problems of gaming application development may be conducted.

We believe that AOP should be involved in game programming more and hope that our work may be an inspiration for other studies.

7. Acknowledgments

We want to thank Asst. Prof. Bedir Tekinerdogan for his precious advices and contributions during this study and also his contributions to this workshop.

8. References

[1] Vander Alves, P.M.J., Paulo Borba, *An Incremental Aspect-Oriented Product Line Method for J2ME Game Development*, in *OOPSLA '04: Workshop on Managing Variability Consistently in Design and Code*. 2004: Vancouver, Canada

[2] Blow, J. 2004. Game Development: Harder Than You Think. *Queue* 1, 10 (Feb. 2004), 28-37.

[3] Ernest Criss, "Aspect Oriented Programming and Game Development", *Technical Report at Department of Information and Computer Sciences University of Hawai at Manao*, Honolulu, USA, pp. 1-5.

[4] Dijkstra, E.W.: On the role of scientific thought. In: Edsger W. Dijkstra: *Selected Writings on Computing: A Personal Perspective*. (Springer-Verlag 1982).

[5] Cristiano Breuel, Francisco Reverbel. *Join point selectors. In Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT 2007), pages 14-21. ACM Press, 2007.*