Abstract—An arcade, two-dimensional single player fixed shooter game application can be considered as a classic in computer games area. In this paper an existing game of this type, namely Space Chickens are refactored using Aspect-Oriented Software Design (AOSD) techniques. Existing Space Chickens implementation includes polymorphism and interface to hold object-oriented design patterns. Also it is mainly a robust system. Because of its object-oriented design it holds the concept of reusability to some extent. Most of it can be easily modified and implemented for later usage, meaning the design allows programmers to change the code with minimal effort. However, object-oriented design paradigm has insufficient ability to resolve all of the scattered and tangling concerns [2]. Thus, Space Chickens also involve some critical cross-cutting aspects.

AOSD is a promising technique to resolve these problems. It provides high cohesion and low coupling in software design. Aspect-oriented programming realizes the cross-cutting concerns as first class elements which results in encapsulating them in explicit modular units that are called as aspects. The aspects in this project are implemented by using AspectJ Development Tools [3] which is a plug in to Eclipse [4] and it supports implementing AOSD with AspectJ. AspectJ is a general purpose Aspect-Oriented Programming Language and an extension to the Java Programming Language.

In this paper, we will first apply some object-oriented design patterns to the Space Chickens game in order to satisfy object-oriented requirements. Then we will detect the cross-cutting concerns in the game. At last we will apply the AOSD techniques in order to solve those cross-cutting concerns i.e. tangling and scattered codes.

The previous design of Space Chickens satisfies most of the object-oriented design goals in classical sense. It is implemented in Java and it modularizes most of the concerns by dividing them into appropriate classes. Each main entity in the game is implemented as a standalone class such as Bullet, Chicken, Egg etc. The previous implementation has a neatly designed abstraction-occurrence and hierarchial class design as it can be seen in Fig. 1. It is mainly a robust system. Because of its object-oriented design, it holds the concept of reusability to some extent. Most of it can be easily modified and implemented for later usage, meaning the design allows programmers to change the code with minimal effort. However, object-oriented design paradigm has insufficient ability to resolve all of the scattered and tangling concerns [2]. Thus, Space Chickens also involve some critical cross-cutting aspects.

AOSD is a promising technique to resolve these problems. It provides high cohesion and low coupling in software design. Aspect-oriented programming realizes the cross-cutting concerns as first class elements which results in encapsulating them in explicit modular units that are called as aspects. The aspects in this project are implemented by using AspectJ Development Tools [3] which is a plug in to Eclipse [4] and it supports implementing AOSD with AspectJ. AspectJ is a general purpose Aspect-Oriented Programming Language and an extension to the Java Programming Language.

In this paper, we will first apply some object-oriented design patterns to the Space Chickens game in order to satisfy object-oriented requirements. Then we will detect the cross-cutting concerns in the game. At last we will apply the AOSD techniques in order to solve those cross-cutting concerns i.e. tangling and scattered codes.

The organization of the paper will be as follows: Sec. II presents the requirement analysis of the legacy code and the object-oriented design. Sec. III continues with the explanation of the Cross-cutting concern in the base code and Sec. IV explains the solution of these problems in AOSD paradigm and evaluates the resultant design. Discussion about the refactoring approach is presented in Sect. V. Finally, paper is concluded in Sec. VI.

II. OBJECT-ORIENTED DESIGN

This section introduces the design of the legacy code. Firstly, requirements of the Space Chickens are presented and the object-oriented approach is summarized. Also object-oriented refactoring of the legacy code is presented in this section.
A. Requirement Analysis

Functional and nonfunctional requirements are presented in this section. A broad overview of use case model can be seen in Fig. 2.

1) Functional Requirements:
   - The game has a player profile system in order to keep track of the players’ information. The player profile system provides one or more profiles for each user. Each profile is stored by a unique player name in order to store player’s game settings, achieved high scores. The game system offers creation of a new profile. User can enter a profile name if it exists, the old profile is used otherwise a new profile is formed.
   - The system provides game settings that can be stored in each player profile. These game settings are:
     - Choose game difficulty
     - Sound on/off
     - Choose spacecraft type
     - Background change
     - Keyboard Controls
   - The game settings system offers options to choose the game difficulty level, enable or disable the sound effects, choose spacecraft’s type, change background and an option to customize the direction and fire keys from the keyboard.
   - The system allows user to have the spacecraft move to right and left and shoot via keyboard.
   - The chickens drop bonus packets during the game. User can collect these bonus packets to have a weapon that shoots well.
   - The system keeps a high score list, where top-scorers of the game are recorded. The user can see this list.
   - The player can pause the game and continue from where he pauses.

2) Nonfunctional Requirements:
   - The game will have a good, user-friendly interface to be understandable by the users. The players will be able to find out what they must do with the game whenever they browse to the interface of the game.
   - The game will have a high performance rate.

B. Design of the Legacy Code

This subsection presents the design of the base code. The object-oriented class diagram of the game engine can be seen in Fig. 3. The game engine is the main class of the system and it manages all of the functions of the system. It is a singleton class and it is a façade between user interface and the other classes. It has lists for each entity in the game (Egg, Chicken etc.) and also has an instance of a SpaceCraft object. The manager classes manage the concepts as their names suggest (ProfileManager manages the profile of the user), MainGameFrame is the main frame of the game where the user interacts, Egg, Chicken, SpaceCraft, Bullet and BonusItem are the entities of the game, where each of them can move except Chicken and each of them can collide with the most of other entities.
C. Object-oriented Design Patterns

Design patterns name, abstract and identify the key aspects of a design structure for making reusable object-oriented designs. They identify the roles, collaborations and the responsibilities of the participating classes and instances. Each design patterns focuses on a different object-oriented problem and each have its own solutions to them [5].

1) Façade Pattern: A façade is an object that provides an interface to some other objects. The façade pattern gives a higher level interface to some objects and simplifies the usage of the system [5]. The original (legacy) Space Chickens has a façade pattern which can be seen from Fig. 3. GameEngine is the façade that is the interface between GUI classes and Game Application classes. Those classes cannot directly call the other classes; they should access them via GameEngine.

2) Singleton Pattern: Singleton Pattern guarantees that only one instance of a class is created and provides a global access to that class [5]. In the legacy system, GameEngine is one of the singleton classes and every other object interacts with the unique instance of the GameEngine.

D. Object-Oriented Refactoring

1) State Design Pattern: The state pattern is a behavioral software design pattern, also known as the objects for states pattern. This pattern is used in computer programming to represent the state of an object. This is a clean way for an object to partially change its type at runtime [5]. In the original Space Chickens game different types of firing actions may happen due to the current firing type, these firing operations can alternate through out the game, and the type checking was done in the code by simply if clauses. For the sake of modularity, simplicity and reusability, the state pattern is applied to this concern.

Eventually, firing types are abstracted away from the fire operation by changing the current state with new states at each firing type moments, therefore only one fire operation works as different types of fire according to the current firing state. Class diagram of the implemented state pattern can be seen in Fig. 5.

III. CROSS-CUTTING CONCERNS

This section presents problems that cannot be solved by object-oriented design paradigm in an efficient way. Original Space Chickens has a carefully designed software architecture. The previous section describes the original code and presents some improvements over it. It is seen that the modularity can be increased by applying some design patterns for a subset of concerns. After it is revised in object-oriented approach (OOA), State Pattern is added and a more flexible code is achieved. Up to this point, it is fair to say that the modularity is established and design paradigms and patterns are grounded well in terms of object-oriented design. However lack of expressiveness in object-oriented paradigm makes it impossible to achieve high cohesion and low coupling in some parts of the code.

Cross-cutting concerns are the main responsible of this problem. Using Aspect-Oriented Software Design (AOSD), these concerns are tried to be captured. Following Cross-cutting concerns are solved using AOSD:

- Collision Detection
- Façade Enforcement
- Constraint Check
- Score Update

A. Collision Detection

Collision detection is one of the most problematic part of the Space Chickens implementation. The game has various objects that are either moved by user or move autonomously. Creating and destroying lots of objects depends on the detection of the collision between that object and possible many others. For example a space craft can collide
with egg or two different bonus items. There are total of 5 different objects that can cause collision with each other namely bullet, bonus item, chicken, space craft and egg.

Each possible collision between these 5 objects are handled differently in the original code. All of these classes has their own collision detection methods implemented in their classes. A future change in collision detection will need change in these 5 classes. Collision detection concern is scattered throughout the code and cannot be handled by object-oriented programming techniques.

An aspect-oriented approach is required to overcome this problem. Details of the solution is presented in the next section.

### B. Façade Enforcement

The design of Space Chickens requires the façade pattern. There are lots of inter-class operations in SpaceChickens. Majority of the classes requires objects of other classes to perform some action during the game. However calling methods of other classes in order to reach these objects can make the code very obscure.

Original Space Chickens implements façade design pattern to overcome this problem. It provides convenient methods for common tasks. Nevertheless, the implementation of façade is change prone. The integrity of the pattern is not guaranteed when the code is modified. This may result obfuscated code at the later use.

A better approach would be warning the programmer about the façade design pattern and encouraging the usage of façade class. However, this requires addition of a warning message to each implemented method on by one in the classical sense. Moreover the methods that are implemented later by other programmer may not include these warnings. Aspect-oriented design can solve this problem in a simple way as describes in Sec. 5.2

### C. Constraint Check

Objects in Space Chickens are highly dynamic. Nearly all of the objects that are seen in the game screen constantly move. Bullets, bonus items, space craft and eggs have certain speed and direction constraints. Changing the values of these objects may result in malfunction of the application.

Base code of Space Chickens does not have any constraint check about the values of these objects’ critical variables. Implementing these bounds in object-oriented paradigm requires adding codes to all of the setter methods of each variable. This results in scattering the constraint check concern throughout multiple classes. A future change in the constraint check strategy will result excessive coding in each of these classes. Furthermore, if the programmer sets the value of the variables directly (without using setter methods) further precautions must be implemented like checking the value of these variables before calling the move methods.

Again aspect-oriented approach provides a noble way to overcome this problem. AOSD provides a coherent way to check all of the critical fields. Please refer to Sec. 5.3 for the details of the solution.

### D. Score Update

Gathering as many points as possible is the main aim of the Space Chickens. Users have various options to collect points throughout the game. Most of the actions triggered by the user cause a change in the total points and each action has a different value. For example, shooting a bullet decreases the score of the user by 5 and killing a chicken provides 65 points.

Score update in the base code is handled separately. There are total of 7 actions that can affect the game score in Space
Chickens which means that the scoring concern is scattered to 7 different parts of the code in different classes. Adding a new action to get score or changing an existing one cannot be handled easily in the object-oriented approach.

Likewise the other 3 problems, the scoring concern is handled in AOSD paradigm. A better approach that suggests easy modification is presented in the following section.

IV. ASPECT-ORIENTED APPROACH

The previous section summarized the main problematic concerns in the base code. Fig. 6 visualizes the Cross-cutting concerns in the legacy code. There is one more concern; namely Façade Enforcement Aspect that can not be seen in aspect visualizer. The reason for omission is that the implementation of the application totally respects the façade enforcement and there is no problematic area. Hence the aspect visualizer omits it as there is no problematic area to point out.

Cross-cutting concerns in the legacy code are handled systematically using AOSD. AspectJ[3] is used to implement the concerns as an AOSD language. AspectJ can be downloaded as an Eclipse[4] plug in as AJDT and it is easy to integrate it with the AspectJ framework. The solutions that are implemented using AJDT is presented below:

A. Collision Detection Aspect

The scattered collision detection aspect is solved by defining rational pointcuts. At the beginning of the game there are no collided objects and after that point, a two object can collide only if one of them moves. Hence the methods that cause objects to move are critical in terms of collision detection.

```java
public aspect CollisionDetectionAspect {
  pointcut bulletMovePoint(Bullet b): this(b) && execution(* Bullet.move(. .));
  pointcut bonusItemMovePoint(BonusItem i): this(i) && execution(* BonusItem.move(. .));
  pointcut eggMovePoint(Egg e): this(e) && execution(* Egg.move(. .));
  pointcut spaceCraftMovePoint(SpaceCraft s): this(s) && execution(* SpaceCraft.move(. .));
  pointcut restrictCheckCollision(): call(* (BonusItem || Egg || SpaceCraft || Bullet).checkCollision(. .));

  after(Bullet b): bulletMovePoint(b) {
    // Collision Check
  }

  declare error: restrictCheckCollision():
  "You cannot call check collision from the object itself – instead use aspect to check collision";
}
```

Code 1. Collision Detection Aspect

In our implementation, all of the move joinpoints of objects that can collide are caught. After the move operations are performed, an advice mechanism is added to catch...
the collided objects if there are any. The details of the implementation can be seen in Code 1.

This implementation handles changes in the code very effectively. For example, if the collision strategy changes (like allowing space craft to collide with chickens) only the collision detection aspect will be changed and the rest of the code will remain as is. On the other hand, object-oriented design requires change in multiple classes which may result in excessive coding. Also if a new element that can collide with multiple objects is added to a code, inserting related code into collision detection aspect is sufficient in AOSD paradigm. On contrary code addition to each class of collidable objects is required in such a scenario in object-oriented programming.

B. Façade Enforcement Aspect

Space Chickens abstracts the classes under façade design pattern. Otherwise the resultant code may become very obscure as there are lots of inheritance and aggregation among classes. Moreover multiple operations from different classes are required in order to perform one action during the game. Hence preserving the façade design against possible code change and modification is an important concern.

Object-oriented design does not provide an efficient way to embed this protection mechanism into code. In order to achieve this in the classical sense, a warning or notification message must be added to each method in each class. It is evident that such a precaution requires lots of coding replication of the same warning code over each class. This solution scatters one concern to each class which cause a very impractical implementation.

```java
public abstract aspect FacadeEnforcementAspect {
    pointcut gameApplication() {
        (call (* SoundManager.*(.)) ||
        call (* LevelManager.*(.)) ||
        call (* ProfileFrame.*(.)) ||
        call (* MainGameFrame.*(.)) ||
        call (* HighScoreFrame.*(.)) && !within(
            SoundManager || LevelManager ||
            ProfileManager || MainGameFrame ||
            HighScoreFrame);
    }

    pointcut Gui() : (call( * Bullet.*(.)) ||
    call( * BonusItem.*(.)) ||
    call( * Chicken.*(.)) ||
    call( * SpaceCraft.*(.)) ||
    call( * Egg.*(.)) && !within(Bullet ||
        BonusItem || Chicken || SpaceCraft || Egg);
    }

    pointcut facade() : within(GameEngine || *Aspect);
    declare warning: gameApplication() && Gui() && !
        facade()
        "You cannot call encapsulated methods directly
        use GameEngine(facade) methods instead" ;
}
```

Code 2. Façade Enforcement Aspect

Solution of this problem becomes quite easy when it is attacked by AOSD techniques. As it can be seen in Code 2, by only catching appropriate joinpoints, all enforcement aspect can be added using just a few lines of code. Additionally, this method collects all façade enforcement aspect under one concrete class, thus eliminates the scattering concern by providing a coherent piece of code.

Changes in façade enforcement again can be handled very effectively. All needed to be done is to change the required field in façade enforcement aspect. On the other hand, object-oriented design requires changing related parts in each class of the application. Similarly adding or deleting new classes to project can handled much more easily in AOSD paradigm.

After writing this Façade Enforcement Aspect, we realized that there are some lines in the code that do not obey the Façade Pattern. We realized this with the help of newly declared warnings. Then we modified the code to satisfy the Façade Pattern.

C. Constraint Check Aspect

Values of variables in Space Chickens has critical limits. Some of the methods has pre or post conditions regarding the values of these variables. However there is not an assurance mechanism about the values of these. This make Space Chickens very vulnerable against modification for the future use.

Addition of the constraint check in object-oriented paradigm has same disadvantages with addition of façade enforcement. Nearly each classes of the Space Chickens are affected from this operation. As a result a future change in constraint check strategy has risk to affect all of these classes. This scattered concern can be solved elegantly using AOSD.

```java
public abstract aspect FatalConstraintCheckAspect {
    abstract pointcut constraintCheck(Object o);
    void around (Object o): constraintCheck(o) {
        if(o.getClass().getName().equals("Chicken")){
            Chicken c = (Chicken)o;
            if(c.speed == 0)
                proceed(e);
            else{
                int speed = (Integer)o;
                if(speed > 0)
                    proceed(o);
            }
    }

    public aspect SpeedConstraint extends
        FatalConstraintCheckAspect {
        pointcut constraintCheck(Object o): args(o) &
            set(= (SpaceCraft || Egg || BonusItem ||
                Bullet || Item).speed); }
    }

    public aspect ChickenConstraint extends
        FatalConstraintCheckAspect {
        pointcut constraintCheck(Object o): target(o) &
            call(* Chicken.move(...));}
}
```

Code 3. Constraint Check Aspect

A new abstract aspect class like in Code 3 is enough to check all constraint variables. Required variables can be captured by joinpoints implemented in concrete constraint aspects and restriction on them can be enforced in the advice part of the abstract aspect class. Modifications and additional constraints can be handled easily on this class by only
modifying the required concrete aspect class or adding a new concrete class to FatalConstraintCheckAspect. No other parts of the code need to be changed for this operation.

D. Score Update Aspect

Score update handling is scattered in the code in original Space Chickens implementation. As described in the previous section, various operations change the score in various values. These features make it hard to modify the scoring operation.

AOSD again provides a fine solution to modularize scoring concern. All of the score modifications are captured in a score update aspect like in Code 4. Note that the pointcuts capture each action that results in a change in the game score. Then the required advice section modifies the game score accordingly.

The aspectual approach to this not only provides concrete implementation of scoring but it also allows for easily modification. For example adding a new score (i.e. adding a penalty for moving space craft) can be handled smoothly by just adding a pointcut and advice section to score aspect. On the other hand, object-oriented programming may require change in multiple classes for implementing this functionality (Assume that space craft can be moved by keyboard, mouse or joystick.).

```java
privileged public aspect ScoreUpdateAspect {
    pointcut init(): call(* GameEngine.new(...));
    pointcut chickenHit(): call(* GameEngine.
collisionDetected(Bullet, Chicken));
    pointcut chickenKilled (Chicken c): target(c) &&
call(* Chicken.isDead(...));
    pointcut spaceCraftHit(): call(* GameEngine.
collisionDetected(Egg));
    pointcut bonusItemHit(): call(* GameEngine.
collisionDetected(BonusItem));
    pointcut levelUp (): call (* GameEngine.levelUp
(...));
    pointcut fireUp(FiringStateContext f): target(f)
&& call(* FiringStateContext.fire (...));
    after(): init()
    ...
    void around(): chickenHit()
    { //Update Score
    }
    before(Chicken c): chickenKilled(c)
    { //Update Score
    }
}
```

Code 4. Constraint Check Aspect

V. DISCUSSION

One important discussion is on the performance issue. The developed aspects caused a considerable performance drop. We have tried the final version on different computer setups, and the results were somewhat frustrating. Computers with reasonable hardware setup did not show any change in terms of performance. However, setups with low-end hardware had performance drops. The upgraded code was running around 5% slower than the legacy code. Obviously, the aspect oriented approach brought some bulk to the legacy code, and this eventually affected the game experience.

To solve our performance issue, we could have narrowed down pointcuts to specific points. For example, instead of capturing all of the speed variable changes, we could have capture only the important ones, such as Chicken class' speed variable, Chicken class' move function does not apply speed factor. We preferred this approach for increasing future reusability of our aspects, yet this unnecessary approach affected the current performance.

Another major issue is, the aspects developed for this project are not robust enough to handle future changes. Annotation based aspects could have solved the fragile aspects problem, yet that would change the legacy code's integrity. What is meant by integrity is the coding style, the legacy code was designed and coded in a simple way, and adding a complex annotation system would take the code to a different level. We simply did not want to disrupt the simplicity of the legacy code.

The last issue should be about why AOP?, and why not OOP and design patterns?, although our renovations are also solvable with object-oriented approach and design patterns, aspect oriented approach is better due to its consistency and extraction of specific points to aspects. If we had tried to solve the problems with OOP approach, for the future refactoring cases, our newly introduced solutions would probably be the future cross-cutting concerns, since introduction of new classes to an existing system requires too much change in the legacy code. Besides, OOP approach would take much more engineering and time than AOSD approach. In short, AOSD saved our precious time and created more consistent and reliable code.

VI. CONCLUSION

In this paper, our aim was to improve the current functionalities of a well-implemented legacy code for a game called "Space Chickens". Our major purpose was to go over the code in a way that reusability and modularity would increase. A feasible and easy approach for this purpose was the aspect-oriented approach, due to its ability of moving the cross-cutting concerns to one point and dealing with them in there, and also gathering up the scattered concerns into one aspect.

In the legacy code, first we applied an object-oriented design refactoring. At this step we found out that fire operation was tangled, and it is easily solvable by introducing a state design pattern. After this step, we applied an aspect-oriented refactoring, and found out some major problems that preventing the modularization and code reusability of
the legacy code. We tackled this problem via aspect-oriented approach. These major points were:

- Collision detection
- Façade enforcement
- Constraint checking
- Score update

Although these problems are solvable with object-oriented approach, aspect-oriented approach is better due to its consistency and extraction of specific points to aspects.

ACKNOWLEDGMENT

We would like to thank to Asst. Prof. Bedir Tekinerdoğan for his contributions, and suggestions throughout the refactoring phases. The last but not least, we would like to thank the colleagues, who supplied this legacy code for us, Kamil Akhüseyinoğlu, Ömer Erdil Albayrak, Ayşê Akay, Ahmet Akçakoca, Burçak Aktay.

REFERENCES