Modeling History Sensitivity and Orthogonally Restricting Aspects

Nagehan Pala Er, Özgür Aydın Tekin, Ömer Köksal

Bilkent University, Department of Computer Engineering, Turkey

{nagehan, oatekin, omerkoksal}@cs.bilkent.edu.tr

Abstract. In this paper, we describe evolution problems of typical software, concentrating on history sensitivity. We start with a basic Object-Oriented design for a simple game. Then, history sensitivity problem is introduced gradually as evolution scenarios. If two different history sensitivity concerns are semantically related, the problem of orthogonally restricting aspects occurs. We elaborate on several solution approaches, namely object-oriented design approach, aspect-oriented design approach, and composition of events and actions approach to tackle the history sensitivity evolution problems in a modular and reusable manner.

Keywords: History sensitivity, Object-Oriented Design, Aspect-Oriented Software Engineering, Evolution, Events, Actions, Composition, Modularity, Reusability.

1 Introduction

Software systems are not static. They continuously evolve. In due time bugs are corrected and new features are added. Software systems have to cope with evolution. In order to cope with evolution various programming languages were introduced. They provided mechanisms to provide several quality factors such as reusability, reliability, maintainability, etc. For example, introducing "class" concept in object-oriented programming (OOP) enabled higher reusability. The mechanisms like inheritance and polymorphism in the object-oriented programming languages provided higher reusability and maintainability. But, the usage of these languages for years showed that existing mechanisms and hierarchy in the object-oriented programming may cause some side effects that decrease the modularity of the software systems.

The aim of evolution is to implement new changes. But, the new software may not be final and may continue to evolve. During evolution process, many features might be added to the current software due to new requirements. The new features to be added may require changes in the design and can cause side effects. As the software evolves the complexity of it grows unless there is a better solution available to solve these issues.

Preserving design modularity is another important issue. It is preferable that software is resilient to possible evolution. If the design is not modular, implementing several quality factors such as reusability and maintainability might be difficult in the evolution scenarios. In order to cope with evolution the system shall be decomposed into proper modules. If the modularity is not proper than the complexity may increase and evolution may require a lot of work.

Also, some features are difficult to add. For example: Adding history sensitivity. Generally, adding history sensitivity results in redefinitions. Also it may cause changes in software design. If you add several history sensitive objects the number of redefinitions increases. The problem becomes even worse if multiple history sensitive extensions somehow affect each other.

Usually, adding history sensitivity with OOP requires many redefinitions and results in tangling and scattering. Then, reusability and maintenance of the software becomes a problem in the future evolution scenarios.

Aspect Oriented Programming (AOP) languages can be used to implement history sensitivity in evolution scenarios. Using the AOP one can try to overcome difficulties faced within the object-oriented languages such as redefinitions. If the aspects can be defined independently, AOP might be a modular solution. However, in the case of semantically related (orthogonally restricting) aspects, AOP may also result in tangled and scattered code.

Composition Filter Model (CFM) and Events, Actions, and Composition Model (EACM) can be used to overcome tangling and scattering problem. EACM is an extension to CFM. CFM aims to provide better modularization and composition mechanisms with respect to object-oriented methodologies. In CFM method calls are processed by filters which are grouped as filter modules. A superimposition selector chooses a set of classes. A query language (such as Prolog) can be used to construct special filters and these filters can be applied on modules.

EACM is an extension of CFM introducing *event filters, action filters,* and *high-level events*. On contrary to event filters, action filters have side-effects on programs. Both of these filters may trigger high-level events as the result of their computation. Than these high level events can be processed by *high-level filters*. These mechanisms provide a hierarchy of events and filters such that resulting filters and filter modules are parametric.

In this paper we investigate the effects of adding history sensitivity to computer software that is already developed with the object-oriented methodology. As the case study we worked with a simplified computer game. We assumed that the computer game does not have any history sensitive features initially. Our first evolution scenario is adding history sensitive features to the current game. Then we assumed that these history sensitive features affect each other. In both cases we tried to investigate the effects of history sensitivity. We try to find solutions to avoid tangling and scattering in the evolution scenarios at the same time trying to minimize the

number of redefinitions. We would like to evaluate modularization and composition techniques to provide better concepts and mechanisms to separate concerns suitably.

The paper is organized as follows. Section 2 defines the object-oriented design of our case without history sensitivity. In Section 3, the problem statement is given in detail. We add history sensitivity to the object-oriented model and investigate the effects of it. We first discuss the required redefinitions in the Object-Oriented model. Then we investigate the problem when Aspect Oriented Programming is used comparing the number of redefinitions. We discuss the orthogonally restricted aspects if the aspects do affect each other. In Section 4, the notion of state machine design for composing several concerns is given. In Section 5, our solution approach for orthogonally restricting aspects problem is explained. We summarize the related work in Section 6. In Section 7, we evaluate the presented design approaches. Finally, section 8 concludes the paper.

2 An Illustrative Example

We defined a simplified computer game to illustrate the problems. The case study is a war game consisting of vehicles, weapons, and buildings, each of which has specific behavior to perform the scenarios given below.

- Player moves a vehicle.
- Player fires a weapon to destroy the buildings.
- Vehicle is destroyed if it hits a building.

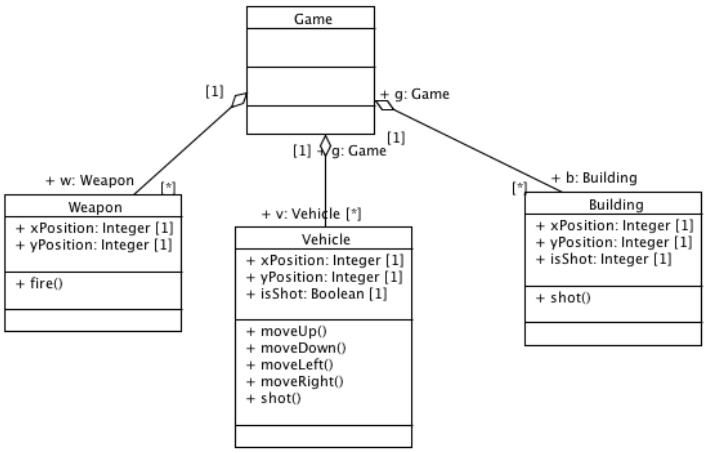

Fig. 1. Object-oriented design of the initial version of the game.

Object-oriented design of the war game that realizes the above scenarios is given in Fig. 1. For instance, Vehicle class shown in the figure encapsulates the following behavior. Its *move* methods update the position of the Vehicle by updating *xPosition*

and *yPosition* attributes. When a Vehicle is shot, *shot* method is called and its *isShot* attribute is set to *true*. We assume that after the Vehicle is shot, it cannot move. The following code snippet shows an example implementation of Vehicle class.

```
public class Vehicle {
  private int xPosition = 0;
  private int yPosition = 0;
  private boolean isShot = false;

  public void moveUp() {
    if (isShot == false)
      // update position in the up direction
  }

  public void moveDown() {
    if (isShot == false)
      // update position in the down direction
  }

  public void moveLeft() {
    if (isShot == false)
      // update position in the left direction
  }

  public void moveRight() {
    if (isShot == false)
      // update position in the right direction
  }

  public void shot() {
    isShot = true;
  }
}
```

After the first version of the game is released, following evolution scenarios are required to be implemented in the next versions of the game. According to evolution scenarios, history sensitivity is added to the game for graceful degradation. Wear/tear of systems is introduced as given below.

- **Vehicle runs out of fuel (fuel history):** When a vehicle moves, its fuel level decreases. If there is no fuel, the vehicle cannot move.
- **Vehicle is destroyed gracefully as it gets shot (damage history for vehicle):** Vehicle has a damage level and is destroyed gradually. If it touches the buildings, the damage level of the vehicle increases. If the damage level reaches its maximum value, its *isShot* attribute is set to *true*.
- **Weapon runs out of battery (battery history):** Weapon has a battery and as long as the user fires the weapon, the battery level decreases. If there is no energy left in the battery, weapon cannot fire.
- **Building is destroyed gracefully as it gets shot (damage history for building):** Buildings have damage levels as well. If a building is shot, the damage level increases. If damage level of the building reaches its maximum value, its *isShot* attribute is set to *true*.

These evolution scenarios have consequences to the operational behavior of the software. In the following section, the problems that arise due to reimplementation of the initial design, realizing the evolution scenarios, are discussed.

3 The Problem Statement

In the next version of the game, it is required that history sensitivity information is added to the game as described in the previous section. First, basic object-oriented design techniques are used to update the initial design. Then, aspect-oriented design approach is considered without changing the initial OO design. These approaches and the problems of them are discussed in the sequel.

3.1 Object-Oriented Design

First, we investigate the object-oriented techniques for adding the history sensitivity to our initial design. Generalization is one of the OO techniques that can be used to evolve each of the related classes according to the given evolution scenarios. Fig. 2 shows the updated design.

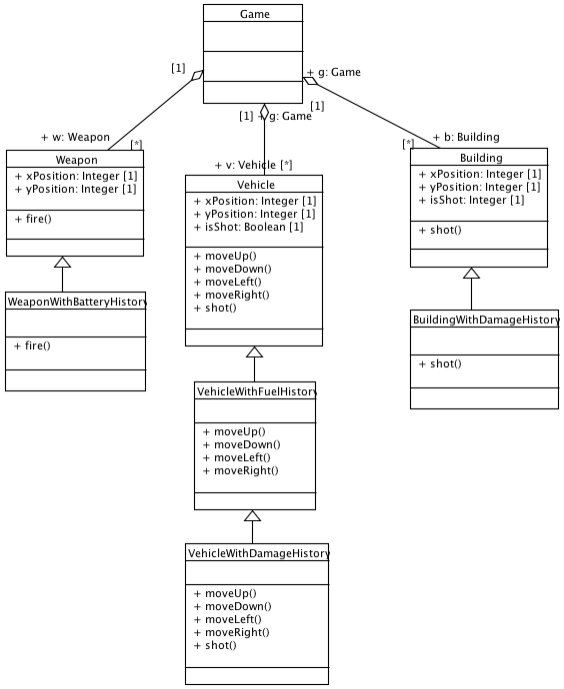

Fig. 2. Object-oriented design of the second version of the game.

Although, existing classes of the initial OO design can be re-used by generalization, most of the methods of the super class have to be redefined. For instance, move methods of Vehicle class must be re-implemented by VehicleWithFuelHistory class as shown in the following code snippet. Before related move method of Vehicle class is called, fuel level must be controlled. If there is enough fuel for movement, then the related move method of Vehicle is called. Otherwise, (if there is no fuel) the related move method is not called.

```
public class VehicleWithFuelHistory extends Vehicle {
  // set the fuel level to its maximum
  private int fuelLevel = 100;

  public void moveUp() {
    if (fuelLevel > 0) {
      // call moveUp method of Vehicle
      super.moveUp();

      // decrease fuel level
      fuelLevel = fuelLevel – 1;
    }
  }

  public void moveDown(){…}

  public void moveLeft(){…}

  public void moveRight(){…}
}
```

Damage history can be added to vehicle in a similar way. A new class, called VehicleWithDamageHistory is inherited from VehicleWithFuelHistory as shown in Fig. 2. This inheritance has the same problem: in order to add damage history we should redefine move methods of VehicleWithFuelHistory class and the shot method of Vehicle class. Similarly, damage level must be controlled before the related move method of VehicleWithFuelHistory class is called. If damage level is not equal to its maximum then the related move method is called. Each shot increases the damage level. While the damage level is less than its maximum, *shot* method of Vehicle class is not called. If damage level reaches its maximum, *shot* method is called and *isShot* attribute is set to *true*. After *isShot* attribute is set to true, vehicle cannot move. The following code illustrates an example implementation of VehicleWithDamageHistory class.

```
public class VehicleWithDamageHistory extends VehicleWithFuelHistory
{
  private int damageLevel = 0;
  private static int MAX_DAMAGE_LEVEL = 10;

  public void moveUp() {
    if (damageLevel < MAX_DAMAGE_LEVEL) {
      // call moveUp method of VehicleWithFuelHistory
      super.moveUp();
    }
  }
```

```
  public void moveDown(){…}

  public void moveLeft(){…}

  public void moveRight(){…}

  public void shot() {
    if (damageLevel != MAX_DAMAGE_LEVEL) {
      // increase damage level
      damageLevel = damageLevel + 1;
    }
    else {
      // call shot method of Vehicle
      super.shot();
    }
  }
}
```

Redefinition problem also exists for WeaponWithBatteryHistory and BuildingWithDamageHistory classes. *fire* and *shot* methods have to be redefined in WeaponWithBatteryHistory and BuildingWithDamage classes, respectively. The number of redefinitions introduced by the evolution scenarios in the OO design is given in Table 1.

Table 1. Number of redefinitions of OO design.

	# of Method Redefinitions	**# of Classes Introduced**
VehicleWithFuelHistory	4	1
VehicleWithDamageHistory	5	1
BuildingWithDamageHistory	1	1
WeaponWithBatteryHistory	1	1
TOTAL	**11**	**4**

Until now, we assumed that the introduced history sensitivity features do not affect each other. Now, assume that some of the defined history sensitivity features are semantically related. For instance, fuel and damage history can affect each other such that damage level changes fuel consumption rate and fuel level changes damage rate. Fuel consumption rate and damage rate are defined as 1 in the previous code snippets. As an OO technique, multiple inheritance can be applied as stated in [1]. Vehicle class can be inherited from both VehicleWithFuelHistory and VehicleWithDamageHistory. However, multiple inheritance does not solve the redefinition problem such that the methods of these two classes cannot be used directly and should be redefined. Further, we should also consider that multiple inheritance is not supported by all OOP languages.

3.2 Aspect-Oriented Design

In this section, aspect-oriented design techniques are investigated to solve the redefinition problem. For each of the evolution scenarios, a new aspect is defined. BuildingDamageAspect and VehicleDamageAspect are introduced to add damage history to the Building and to the Vehicle classes of the base design, respectively. VehicleFuelAspect is defined to add fuel history to the vehicle class and WeaponBatteryAspect is defined to add battery history to the weapon class. These aspects are illustrated in Fig. 3.

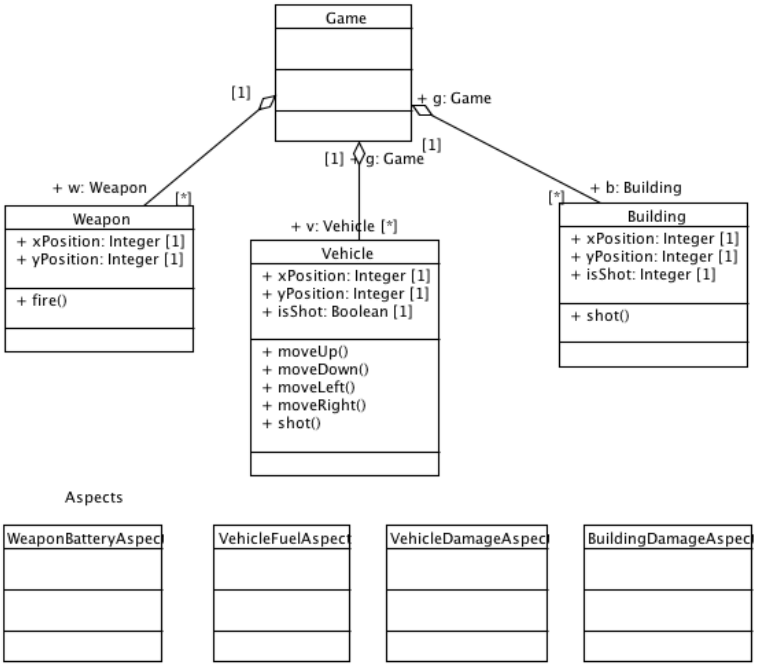

Fig. 3. Aspect-oriented design of the second version of the game.

VehicleDamageAspect and VehicleFuelAspect are defined in the following code snippets. Moreover, OrderingAspect have to be defined to declare the precedence ordering between these two aspects since both of them catch the same join points which are calls to the move methods.

```
public aspect VehicleDamageAspect {
  private int damageLevel = 0;
  private int MAX_DAMAGE_LEVEL = 100;

  pointcut triggerShot(): call (* Vehicle.shot(..));

  void around(): triggerShot() {
```

```
    damageLevel = damageLevel + 1;
    if (damageLevel > MAX_DAMAGE_LEVEL) proceed();
  }

  pointcut triggerMove():
    call (* Vehicle.move*(..));

  void around():triggerMove () {
    if (damageLevel < MAX_DAMAGE_LEVEL) proceed();
  }
}

public aspect VehicleFuelAspect {
  private int fuelLevel = 100;
  private int MIN_FUEL_LEVEL = 0;

  pointcut triggerMove():
    call (* Vehicle.move*(..));

  void around():triggerMove () {
    fuelLevel = fuelLevel - 1;
    if (fuelLevel > MIN_FUEL_LEVEL) proceed();
  }
}

public aspect OrderingAspect {
  declare precedence: VehicleDamageAspect, VehicleFuelAspect;
}
```

Using aspect-oriented techniques instead of object-oriented techniques solves the redefinition problem. Considering the aspects described above, the number of redefinitions vanishes as given in Table 2. Note that, each aspect is considered as a new class.

Table 2. Number of redefinitions of AO design

	# of Method Redefinitions	**# of Classes Introduced**
VehicleFuelAspect	0	1
VehicleDamageAspect	0	2
BuildingDamageAspect	0	1
WeaponBatteryAspect	0	1
TOTAL	**0**	**5**

When the OO design and the AO design approaches are compared with respect to the number of redefinitions, AO design seems superior. We see that AO approach solves the redefinition problem if the aspects are semantically unrelated, where aspects do not affect each other. However, AO approach may also have problems if they are semantically related, or in other words, orthogonally restricting aspects, as discussed in the following section.

3.3 Orthogonally Restricting Aspects Anomaly

If several aspects are defined independently and affect each other semantically, we call this problem orthogonally restricting aspects anomaly. This definition is adopted from orthogonally restricting specifications, which is defined in [1]. Initially, we assume that the introduced history sensitivity features do not affect each other. For instance, fuel history and damage history can be defined for Vehicle class independently:

- If a vehicle has fuel, it can move. Otherwise, it cannot move. Each move decreases fuel level by 1. (Fuel consumption rate is defined as constant and its value is equal to 1 as shown in the previous code snippets.)
- If a vehicle is shot, its damage level increases. As long as its damage is less than its maximum, the vehicle can move. After damage reaches to its maximum, then the vehicle cannot move. (Damage rate is also defined as constant and its value is equal to 1 as shown in the previous code snippets.)

Then, assume that it is required that some of the defined history sensitivity features should be semantically related. For instance, fuel and damage history of vehicle can affect each other such that damage changes fuel consumption rate and fuel level changes the damage rate:

- If damage is high, fuel consumption rate is also high. While damage increases, fuel consumption rate also increases.
- If fuel level is high, damage rate is also high. While fuel level decreases, damage rate also decreases.

According to this scenario, VehicleDamageAspect and VehicleFuelAspect affect each other semantically. Fuel Level (FL) must be adjusted considering the Damage Level (DL) of the vehicle, that is fuel consumption rate (FCR) increases proportionally with damage level. Damage level of the vehicle is adjusted considering the fuel level of the vehicle, that is Damage Rate (DR) increases as fuel level increases. The interrelation between the two orthogonally restricting aspects are shown in Table 3.

Table 3. Orthogonally restricting aspects

	Fuel Aspect	**Damage Aspect**
Fuel Aspect	-	FL ↑ → DR ↑
Damage Aspect	DL ↑ → FCR ↑	-

We can define only one aspect to control both damage and fuel. This aspect has two new attributes for damage rate and fuel consumption rate. When fuel level is updated, damage rate is also updated according to fuel level. Similarly, when damage level is updated, fuel rate is also updated according to damage level. This approach has three drawbacks. The first drawback is that, two different concerns (damage and fuel) are defined in the same aspect so tangling occurs. The second one is that it does not use previously defined two aspects so it is not a modular extension. The last

drawback is that, long if-else blocks are used to update fuel rate and damage rate as shown in the following code snippet:

```
public aspect VehicleDamageFuelAspect {
  private int damageLevel = 0;
  private int MAX_DAMAGE_LEVEL = 10;
  private int damageRate = 1;

  private int fuelLevel = 100;
  private int MIN_FUEL_LEVEL = 0;
  private int fuelRate = 1;

  pointcut triggerShot(): call (* Vehicle.shot(..));

  void around(): triggerShot() {
    damageLevel = damageLevel + damageRate;

    // update the fuel rate according to damage level
    updateFuelRate();

    if (damageLevel > MAX_DAMAGE_LEVEL) proceed();
  }

  pointcut triggerMove():
    call (* Vehicle.move*(..));

  void around():triggerMove () {
    if (damageLevel < MAX_DAMAGE_LEVEL) {
      fuelLevel = fuelLevel - fuelRate;

      // update the damage rate according to fuel level
      updateDamageRate();

      if (fuelLevel > MIN_FUEL_LEVEL) proceed();
    }
  }

  void updateFuelRate() {
    if (damageLevel >= 8 && damageLevel < 10) {
      // update fuel rate according to this condition
    } else if (damageLevel >= 6 && damageLevel < 8) {
      // update fuel rate according to this condition
    }
    …
  }

  void updateDamageRate() {
    if (fuelLevel >= 90 && fuelLevel < 100) {
      // update damage rate according to this condition
    } else if (fuelLevel >= 80 && fuelLevel < 90) {
      // update damage rate according to this condition
    }
    …
  }
}
```

4 Notion of state-machines, events and actions

We have introduced the problem of orthogonally restricting aspects, where Fuel Aspect and Damage Aspect are co-related in the previous sections. These two aspects have to be composed in a modular way for re-use, to implement the history sensitivity concern. We have discussed that such a modular composition is not possible. A logical solution that comes to mind is to write a new aspect that implements both damage and fuel history sensitivity concerns. This approach causes more and more tangling with if-else statements to express these concerns in the same aspect.

Here, we argue that one can try to remove the undesired if-else blocks for different concerns using a state-machine in the new aspect, toward a more flexible and re-usable design. In the sequel, we discuss that this is not a proper solution for orthogonally restricting aspects problem. However, it helps us recap the notion of events and actions, and introduce a different way of separation and composition of concerns.

Considering our case, we tackle the tangling problem of the new aspect by building a state-machine as in Fig. 4. Instead of checking the damage level in fuel aspect, or checking the fuel level in damage aspect, we use this state machine with the events "move" and "shot" that trigger the actions to be taken in the states. Such a state machine can be implemented as a finite state machine (JavaFSM) or a regular expression (JavaERE) facilitated by JavaMOP [2]. Extending the state-machine with more states (actions) for higher precision or with more events to implement another dimension or concern is possible.

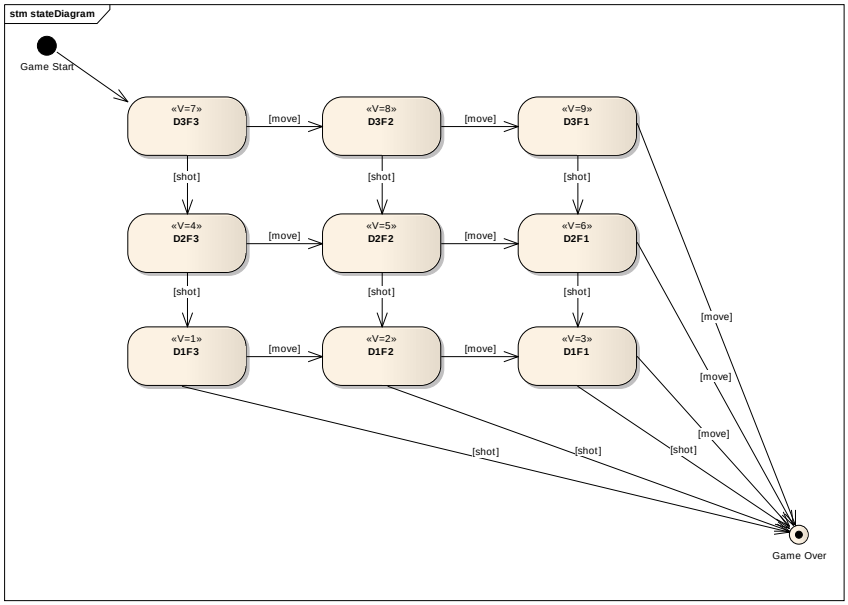

Fig. 4. State-machine, events and actions.

In our case, damage level influences fuel consumption rate and the fuel level influences the damage rate. Thus, the effects of "move" and "shot" events, as given in Fig. 4, are not independent. That is, the current state "D3F3" has to be changed to the next state "D3F1" when a "move" event happens since the damage level is high and the vehicle consumes more fuel when it moves. Similarly, we have to by-pass the state "D2F3" when a "shot" event occurs in state "D3F3" since the vehicle gets more damage when fuel level is high. In such an orthogonally restricting events-actions case, the state-machine solution is not well-structured, hence not scalable. Furthermore, the number of states may explode if higher resolution is required or more dimensions are introduced.

In the next section, we discuss a modular solution approach based on the events and the composition of the events that trigger the corresponding actions.

5 Composition of events and actions

In the previous section, we discussed AOP and the state-machine approaches which work only if the concerns are independent. However, for the orthogonally restricting aspects problem, neither presents a modular or re-usable extension mechanism. In this section we present a modular extension framework that tackles the conflicting aspects problem.

Composition of events has been introduced in [7]. Events are defined by a declarative language, filtered and the corresponding actions are triggered as illustrated in Fig. 5. When a new evolution scenario is introduced, new events or a composition of the events can be defined. Extending the existing design is intuitive and modular in this way. Dashed ellipses and boxes represent the extensions in Fig. 5.

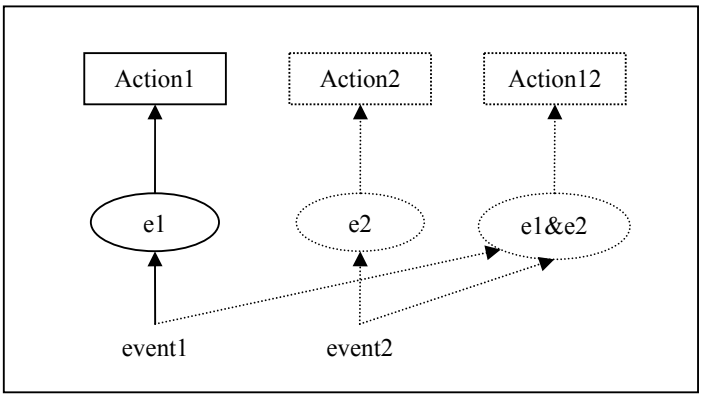

Fig. 5. Composition of events.

If we consider our case, game with history sensitivity problem, event1 and event2 stand for the move() method and shot() method calls. Respectively, action1 and

action2 stand for modifying the FuelLevel and the DamageLevel. The composition of semantically related concerns are discussed further in [3] as composition anomalies.

The composition of events and actions approach is extensible, where further events are defined on top of the actions. This approach is based on event-action-composition model (EACM) presented in [7]. Higher level events can be generated from actions as well as events. For our case, we introduce higher level events triggered by actions. This feature guides us to solve history sensitivity problem with orthogonally restricting aspects by using the higher level events that are generated by the corresponding actions. This provides more flexibility for reusable and modular extensions.

As explained in the previous sections, damage level affects fuel level and fuel level affects damage level. These effects can be expressed by writing long if-else structures or by introducing new variables. For our case, two new variables are introduced.

- Fuel consumption can be added as a new variable. While a vehicle moves, it decreases its fuel level according to its fuel consumption. In the initial version of the game, there is no fuel level. In the next version of the game, fuel level history is implemented but it is not affected by damage level such that each move operation decreases the fuel level by 1. Actually, each move operation decreases the fuel level by fuel consumption at that time. If we introduce a new variable called *FuelConsumption* and this variable is used directly in move operations, the only remaining problem is setting the fuel consumption. Damage level affects the fuel consumption. Damage level is changed when shot operation is called. When damage level changes, fuel consumption should also be changed.
- Damage rate can also be added as a new variable. While a vehicle is shot, it increases its damage level according to its damage rate. In the initial version of the game, there is no damage level. In the next version of the game, damage level history is implemented but it is not affected by fuel level such that each shot operation increases the damage level by 1. Actually, each shot operation increases the damage level by damage rate at that time. If we introduce a new variable called *DamageRate* and this variable is used directly in shot operations, the only remaining problem is setting the damage rate. Fuel level affects the damage rate. Fuel level is changed when move operation is called. When fuel level changes, damage rate should also be changed.

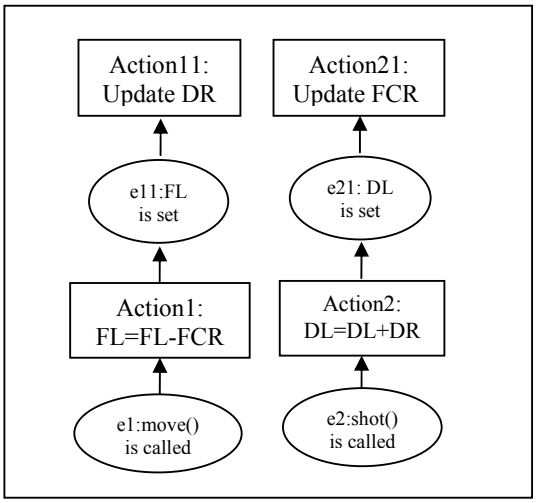

Fig. 6. Events on actions.

This scenario shows that actions can cause other events and these events results in other actions. Fig. 6 illustrates this case.

- Event1: move operation is called
- Action1: FuelLevel is decreased by FuelConsumptionRate
- Event11: Changing the FuelLevel
- Action11: May change DamageConsumptionRate

- Event2: shot operation is called
- Action2: DamageLevel is increased by DamageRate
- Event21: Changing the DamageRate
- Action21: May change FuelConsumptionRate

7 Related Work

In [1], authors discuss orthogonally restricting features problem from real-time perspective. In their work, several real-time specifications are defined independently and they are combined by using multiple inheritance. If these specifications affect each other semantically, these specifications may have to be re-defined. To solve this problem, each real-time specification has a parameter in order to state its real-time constraints. When two methods have to be combined, this parameter is checked to identify which specification is applicable.

Although Object Oriented Programming (OOP) solves many problems of Procedure Oriented Programming (POP), many OOP techniques are not sufficient to

capture some of the important design decisions that the developer must implement bringing scattered and tangled code [15].

Aspect Oriented Programming (AOP) is a methodology to improve the modularity of the software when the software has crosscutting concerns. AOP claims that change from POP to OOP is not complete and programmers need more dimensions [9].

AOP may solve many problems related with the object oriented programming. AOP aims to make the design and the code more modular. This means localizing concerns instead of scattering them. But on the other hand, superimposing aspects on software modules may cause side effects. New techniques are required to cope with this problem [8].

In [14], authors claim that the history-based aspects cause high runtime overhead and if domain knowledge is used compilers may apply powerful optimizations. Since current AOP languages do not provide tools for optimization they introduce "dependent advice" as a new extension to AspectJ language to preserve domain knowledge. They use a code generation tool to automatically generate dependency advice to lower the runtime overhead caused by history-bases aspects.

Akşit et. all [8] claim that one of the main problems of AOP is the aspect interference problem. They presented Composition Filter (CF) approach and introduce Compose* tool to detect and correct the semantic conflicts among aspects which are superimposed on the same join point.

Composition Filters (CF) uses the conventional object model and considers an object as an entity that performs some assigned functions [10, 12]. Within a system, entities interact with each other to achieve a common task. In the object model, most interactions are done by sending and receiving messages, that is where CF is introduced. By controlling messages (changing their targets and/or selectors) and through a well constructed interface, CF provides suitable solutions to many problems [10]. One of the CF approach strengths is the use of a uniform filtration mechanism to resolve these problems. From this viewpoint, CF is easy to understand and work with as it only adds few concepts to the object model. Like AspectJ, CF is one of the well-known and mature AOP approaches seeking new modularization concepts [13].

Events, Actions, and Compositions Model (EACM) is introduced as an extension to Composition Filter Model (CFM). In this model event filters and action filters are introduced both of which can trigger higher level events. Since filters and filter modules can be parametric they can be passed as arguments. The E-Chaser language is proposed to adopt EACM. Events, actions, and compositions are the first class linguistic constructs in this language. E-Chaser can be used in implementing runtime enforcement systems. Although it is based on CFM, E-Chaser provides dedicated constructs to define filter modules, filters and superimposition selectors which are defined as Prolog rules. Since it is possible to develop generic filter modules using parametric filter modules and filters, the codes can be evaluated free of tangling and scattering enabling software reuse and maintainability [7].

8 Evaluation

In this section, we evaluate the presented design approaches and make their advantages and disadvantages more clear. We begin with explaining the quality factors to compare different approaches. Our quality factors are related with modular extension. An approach is preferable if it enables the developers to make modular extensions. A modular extension can be defined by reuse and number of re-definitions. If a new functionality is added to the system as a new module and it re-uses the previously developed modules by reducing the number of re-definitions, it is a modular extension. If a proposed approach reuses the previously developed modules but also causes a lot of re-definitions instead of using them directly, its modular extension level is considered as low. Similarly, if an approach does not use the previous modules and all of the modules are defined from scratch, it is not definitely a modular extension because it is not an extension.

First, we evaluate object-oriented extension. We investigate OO techniques for adding history sensitivity to our initial object-oriented design in Section 3.1. Generalization technique is applied to re-use the previously developed classes. However, it is shown that generalization causes many redefinitions. As given in Table 1, 11 methods have to be redefined when 4 new classes are added. If we further assume that the added history sensitivity features affects each other, then the problem gets even worse. As an OO technique, multiple inheritance can be applied as stated in [1]. We should also notice that multiple inheritance does not supported by all OOP languages. If we assume that we apply multiple inheritance, it also causes redefinition problem.

In order to overcome redefinition problem, AOP techniques are investigated. Each new history sensitivity feature is defined as a new aspect instead of inherited classes. If these features are semantically unrelated, redefinition problem is solved by AOP. As shown in Table 2, AOP causes no redefinitions. However, if we assume that if some of the introduced features affect each other, a new problem called orthogonally restricting aspects problem occurs. If we define all semantically related history sensitivity features only as one aspect, several drawbacks occur such as tangling and long if-else blocks.

State machine based approach requires defining the each case as a new state, instead of writing long if-else blocks. State machines can be implemented by OO design or AO design. A state machine can also be written as a separate specification as stated in [6]. Events and state machine specifications are defined together and events are used to trigger the state machine. The most important problem of the state machines is extensibility. For example, if a state machine is defined for two parameters (assume that these parameters are boolean so there are four different states) and then if a new parameter is introduced (assume that the newly added parameter is also boolean so it increases the number of states to nine), the whole state machine should be defined from scratch.

Our proposed solution to overcome orthogonally restricting aspects anomaly is based on event compositions and events on actions [7], which is a promising solution for modular extension. It maximizes the reuse of the previously defined actions. It introduces new actions and new events while the system evolves. Events can be combined to make new events that can also be associated with actions. According to this approach, actions can yield new events and these new events can trigger new actions in turn. These events are called higher-level events. This hierarchy can be applied to higher-levels as needed. This approach solves both redefinition problem and orthogonally restricting aspects anomaly for our case study.

9 Conclusion

Evolution of software is irresistible. Main problem is how to design a software system that is flexible to accommodate the required extensions, minimally altering the base design. This flexibility is not possible with the existing object-oriented design methods since concerns or aspects cannot be expressed as first class entities in OOP. A better solution is to have modular extension mechanisms that do not tangle the base code, and that supports reliable integration of the new modules.

In this paper, we considered history sensitivity problem that is introduced on top of an existing object-oriented solution for a simple game. First, it is shown that object-oriented design techniques cannot undertake the history sensitivity based evolution scenarios without redefining the original code. Then, an aspect-oriented solution for modular adaptation of the evolution scenarios is given. As more scenarios are introduced, the problem of orthogonally restricting aspects occurs. Aspect-oriented solution loses its modularity and reusability, causing further tangling and scattering, hence breaking the principle of separation of concerns due to the mixed structure of events (pointcuts) and actions (advises) in the aspects.

As a modular and reusable extension mechanism, a state-machine approach is investigated in order to implement several concerns in the same module (aspect). State-machines do not provide standard extension mechanisms when new concerns are added as new dimensions of events and state. However, if such extensions grow in a well-structured way, regular expressions or finite-state automata can be used for extensions, reusing the original state-machine design.

Finally, the idea of separation of events and actions is discussed where events and actions are defined independently. Composition of events that trigger corresponding actions, and events on actions methods are discussed as a modular and more reusable way of supporting unavoidable evolution requirements.

References

1. Aksit, M., Bosh, J., Sterren, W., Bergmans, L.: Real-Time Specification Inheritance Anomalies and Real-Time Filters., pp. 386--407. Springer Verlag (1994)
2. F. Chen, C. Lee, and G. Roşu. Mining parametric speci_cations. Technical report, University of Illinois, 2010. URL http://hdl.handle.net/2142/15109.
3. Bergmans, L., Tekinerdogan, B., Magy, I., Aksit, M., An Analysis of Composability and Composition Anomalies, Internal Report.
4. Havinga, W., Bergmans, L., Aksit, M., Prototyping and Composing Aspect Languages Using an Aspect Interpreter Framework, ECOOP 2008 Object Oriented Programming, Lecture Notes in Computer Science, 2008, Volume 5142, pp. 180-206.
5. Havinga, W., Staijen, T., Rensink, A., Bergmans, L. Berg van den, K., An Abstract Metamodel for Aspect Languages, Internal Report.
6. Malakuti, S., Bockisch, C., and Akşit, M, "Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software", 20th International Symposium on Software Reliability Enginnering, pp. 31-40, 2009.
7. Malakuti, S., Akşit M., and Bockisch, C., "Events, Actions, and Compositions", University of Twente, Enschede, The Netherlands, 2011.
8. Durr, P., Stajlen T., Bergmans, L., and Akşit M., "Reasoning About Semantic Conflicts Between Aspects", University of Twente, The Netherlands.
9. http://c2.com/cgi/wiki?AspectOrientedProgramming
10. Akşit, M., and Tekinerdoğan, B., "Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters", University of Twente, Center for Telematics and Information Technology.
11. Elrad, T., Akşit, M., Kiczales, G., Lieberherr, K., and Ossher, H., et. all, "Discussing Aspect of AOP", Communications of the ACM, October 2001, Vol. 44, No:10.
12. Meslati, D., Kimour, M.T., and Ghoul S., "From Composition Filters to AspectJ", Journal of Computing and Information Technology, 2006, 2, 111-131.
13. Meslati, D., "On AspectJ and Composition Filters: A Mapping Concept", Informatica, 2009, Vol. 20, No: 4, pp555-578.
14. Bodden, E., Chen F., and Roşu, G., "Dependent Advice: A General Approach to Optimizing History-based Aspects", AOSD'09, March 2-6, 2009, Charlottesville, Virginia, USA, ACM 978-1-60558-442-3/09/03.
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier J.M., and Irwin, J., "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming, Finland, June 1997.