Reusable Aspect Models in practice using TouchRAM

Joeri Exelmans[a]

[a]*University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium*

Abstract

This paper starts as a summary of 2 papers written on the theory of Reusable Aspect Models. The authors of these papers have also developed a tool called TouchRAM. I will use this tool to model a minimalistic role-playing game. I will use my newly acquired experience to asses whether RAM and TouchRAM make it possible to create reusable software packages, to keep consistency among them and to provide overall scalability.

Keywords: Reusable Aspect Models, TouchRAM, Aspect-oriented Modeling

1. Introduction

Reusable Aspect Models are a proposed way of dealing with the consistency and scalability issues that came with existing multi-view modeling techniques. Aspect-oriented techniques have the potential to tackle these weaknesses, as they

- let the developer reason about each concern individually (Scalability), and
- allow for a clear composition of concerns (Consistency). Concern interactions as well as conflicts are explicitly modeled.

Existing Aspect-oriented modeling techniques operated only within a single modeling notation.

Section 2 gives a summary of what RAM is and what it is supposed to deliver. Section 3 introduces TouchRAM, a tool made by the original authors who proposed RAM. Section 4 is a detailed description of a system I developed with TouchRAM. Section 5 is about some ideas I've gathered from my experiences. Section 6 contains thoughts about future work.

2. What is RAM?

The Reusable Aspect Models (RAM) approach described in ORIGINAL PAPER

1. Integrates different formalism techniques into 1 coherent approach
2. Packages aspect models

3. Supports aspect dependency chains
4. Checks consistency
5. Defines weaving algorithm.

2.1. A RAM package

The basic unit of a module in RAM is called a package. A package defines a certain amount of functionality. A package consists of:

Structural View Formalism: UML Class Diagram. In the structural view, one or more classes are defined. Classes contain a number of methods, attributes and associations which are only relevant to the concern of the package. A class can incomplete, i.e. it does not contain enough functionality to be instantiated already. Incomplete classes are preceded by the character | in their name.

State View Formalism: State Charts. For every class which has public methods, a state view must be present. Defines a usage protocol. In case it describes an incomplete class, the state view has a pointcut and an advice section, in order to extend the state view of another package instantiating the incomplete class (see later).

Message View Formalism: UML Sequence Diagram. At most 1 message view per public method in structural view. Describes interactions between classes in structural view. Always in the form of pointcut (typically a call to the public method) + advice (what to do when called).

2.2. Dependencies and Reuse

Several concepts:

Mandatory Instantiation Parameters Incomplete classes that have their name starting with a | character, supplemented with incomplete methods, incomplete states and incomplete objects make up the mandatory instantiation parameters of the package.

Instantiation When a package A depends on a package B, it has to instantiate all of B's mandatory instantiation parameters. This means A must provide a mapping from those parameters to either complete or incomplete elements in A. This has the semantics of B supplying extra functionality to (elements of) A. This can be in the form of extra methods, attributes and associations in the state view, states and transitions in the state view, and message sequences in the message view.

Binding If A depends on B, it can also map complete elements of B to elements in A. This results in elements from A's views being merged with mapped elements from B's views.

2.3. Weaving algorithm

Dependencies of many RAM packages form a directed acyclic graph (DAG). The weaving of a node A in such a graph with the packages it depends on is also called creating an independent model of A. An independent model is one from which no arrows depart. To create an independent model of A, we first create independent models of all the packages A depends on (recursively). Then, finally, for each node B on which A depends, A is woven with B, i.e. pair-wise. The order in which a node is woven with its dependencies usually doesn't matter.[3]

2.4. Consistency checks

When weaving a pair of models, there are 3 levels of checks:

1st level Checks consistency within individual aspect models. State view checks: All methods must be present, all fields in state view must be declared in structural view, ... Message view: For each public method in state view, max. 1 message view. Checks whether message view behaves according to state view protocol.

2nd level Checks consistency between pair of models. Checks if binding rules and instantiations are compatible. Checks if all mandatory instantiation parameters are supplied.

3rd level Checks level-1 consistency again, but this time in the new model that resulted from the weaving.

3. TouchRAM

TouchRAM provides an environment where the user can model RAM packages. It is intended to be suitable for use on many platforms. Its interface is promoted as being able to get a high amount of productivity out of both (multi)touch screens and the combination of keyboard and mouse.[1] The version used in this paper includes the following functionality[2]:

- Aspect hierarchies
- Structural view
- Simple consistency checks

Additionally, the user can have a look at the message view of packages which already include a message view, and a weaving algorithm for this has also been developed. However, creation and editing isn't supported (yet). I will thus restrain myself to the creation of RAM packages which only consist of a structural view. Other missing features (like state view editing and weaving) are planned for the near future[2].

4. Solution

4.1. System Requirements

I will model a minimalistic Role-Playing Game. It is based on the assignments during the course *Model Driven Engineering*, but the requirements have been relaxed a bit[1]. They are as follows:

- A `Game` consists of a number of connected `Tile`s, 1 `Hero` and 1 or more `Villain`s.
- A `Tile` either contains the `Hero`, a `Villain`, or nothing.
- A `Game` is turn-based. The `Hero` and `Villains` get to play their turn in a fixed order (`Hero` first, then the `Villain`s), round-robin.
- During their turn, the `Hero` and `Villain`s can either move to a tile next to their current tile, or, if possible, attack a `Villain`/the `Hero` respectively. They do this randomly.
- A `Game` ends if the `Hero` is dead (game lost). or if he has killed all the `Villain`s (game won).

4.2. Overview

The packages that make up a complete system can always be classified in one of the following categories:

1. Application-specific. Non-reusable.
2. Domain-specific. Reusable in other applications that operate in the same domain.
3. Domain-independent. Highly reusable.

This solution consists of 10 RAM packages. Figure 1 shows the dependencies of all packages in the solution, with these 3 categories marked.

In [3], the authors explain their solution bottom-up, i.e. they start with the lowest-level packages, and as they progress, they talk about higher-level packages that depend on the ones previously explained. This was an excellent way to introduce the reader to the concepts of RAM, as the low-level packages are usually very small and simple.

In this paper, a different strategy will be used: The packages on the domain-specific level will first be described, staying as high-level as possible, and going down to deeper levels if dependency chains are encountered. This order is more consistent with the order in which the different packages where created[2]. Finally, the 2 application-specific packages will be discussed.

[1]Scenes, doors, keys, traps, obstacles and goals have been removed.

[2]This is because in the RAM way of doing things, we start with the full requirements and keep splitting reusable functionality off into different packages, see Section 5.3.

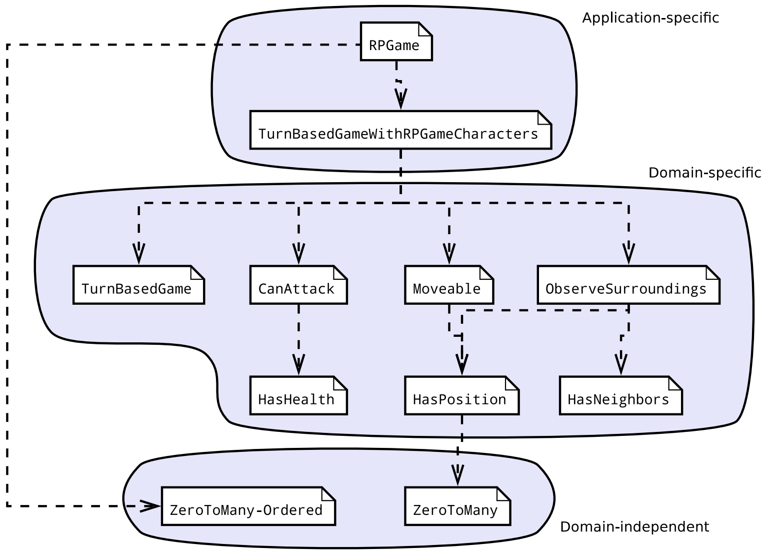

Figure 1: Package dependencies

4.2.1. TurnBasedGame

If we try to reason about the most high-level, yet application-independent functionality that RPGame relies upon, there are probably a number of possibilities. I chose to create an aspect model which represents the concept of a *turn-based game*. What are the properties of a turn-based game?

- There has to be some mechanism that determines which player should have his turn
- During his turn, the player generates a set of actions he can perform
- For each set of generated actions (= each turn), some mechanism has to pick one action
- That action is then executed by the player
- Finally, there has to be a condition for the game to end.

This results in the RAM package seen in Figure 2. Note that the mandatory instantiation parameters start with a | character.

While designing this package, I had an association between |TurnSelector and |Player in mind, i.e. |TurnSelector keeps a list of |Players. However, that would lock our implementation into a turn-based game where the order of the turns must be represented by a list. This is indeed what we want for RPGame, but it would make the TurnBasedGame package less reusable. Such an association would also be completely useless in the context of this package: as long as we don't implement |nextTurn(), we don't *require* that association to exist. It is

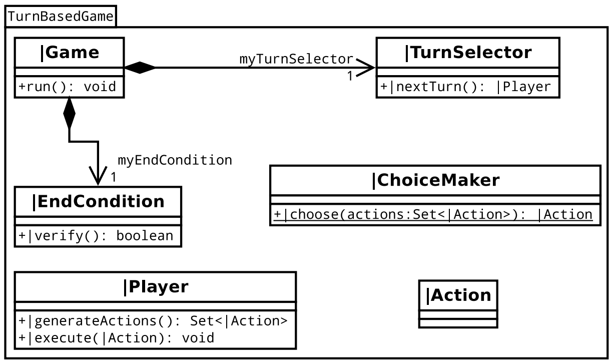

Figure 2: Structural view of the TurnBasedGame package.

good RAM design to only specify the elements (classes, methods, attributes, associations, ...) which are relevant to the provided functionality.

In this case, the provided functionality (and also the only complete element in the package) is the *run()* method of |Game. Its message view is shown in Figure 3. The game object first checks its |EndCondition. If it evaluates to true, the game loop ends[3]. The game object queries |TurnSelector for a player. |TurnSelector comes up with a player somehow and returns it to the game object. The game object queries the player for a set of possible actions. The game object then calls a static method in the |ChoiceMaker class to choose between those actions. This way, we can instantiate the |TurnBasedGame package to be interactive, random, or something completely different[4]. The game object finally calls the *execute()* method on the player with the chosen action as argument.

Finally, one interesting fact: TurnBasedGame does not depend on any other package. The functionality it provides is in fact so simple that it completely stands on its own.

4.2.2. CanAttack and HasHealth

To be situated on the same layer as TurnBasedGame, the CanAttack package introduces 2 incomplete classes: |CanAttack and |Victim. |CanAttack has a method to attack an object of type |Victim. This is achieved by calling |Victim's *receiveDamage()* method.

It is clear that, for the "receive damage"-concept to make sense, the |Victim class must have some awareness of its remaining health. Therefore, we make CanAttack depend on the HasHealth package. |HasHealth is the only mandatory instantiation parameter. It denotes the class to be equipped with the functionality of having health. It is thus instantiated to be |Victim.

[3] *run()* is supposed to be a loop which keeps running until *verify()* evaluates to true, however, my UML sequence diagram program didn't allow me to visualize a loop.

[4] It would have been even better if we had one |ChoiceMaker per player, so some players could be interactive and others randomized.

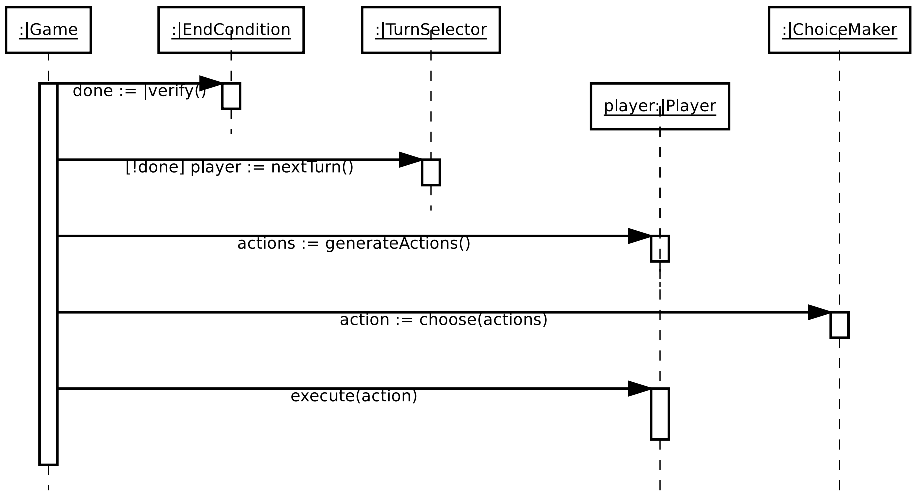

Figure 3: Message view of Game.run(). Please note that this was not explicitly modeled in TouchRAM for reasons explained in section 3. This figure is solely intended to give the reader a better idea of what the TurnBasedGame package does.

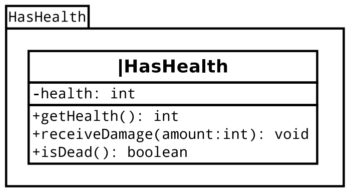

Figure 4: State view for HasHealth.

Because HasHealth already implements the *receiveDamage()* method, we also add a binding from HasHealth.receiveDamage to |Victim.receiveDamage().

4.2.3. Movable and HasPosition

Moveable defines the possibility of a class to be moved. There is no such thing as absolute movement. An object can only move relative to another. We need some reference, hence an empty incomplete class |Position is introduced alongside the |Movable class.

The *move()* method changes the |Moveable's position to the *destination* argument.

It would be logical to model |Moveable's position as an association from |Moveable to |Position, labeled e.g. *myPosition.* But this is actually a piece of information that could make up a RAM package itself. Indeed, Moveable depends on a package called "HasPosition" in order to implement this functionality.

HasPosition also has an association in the opposite direction. This represents the fact that a |Position keeps a set of objects that are located at it. TouchRAM

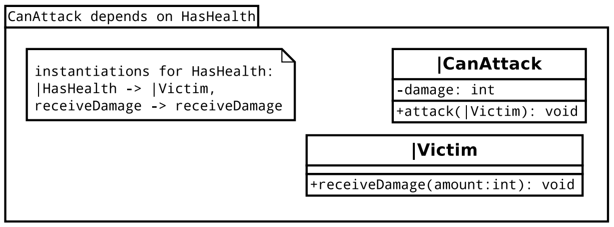
CanAttack depends on HasHealth
instantiations for HasHealth:
|HasHealth -> |Victim,
receiveDamage -> receiveDamage
|CanAttack
-damage: int
+attack(|Victim): void
|Victim
+receiveDamage(amount:int): void

Figure 5: State view for CanAttack.

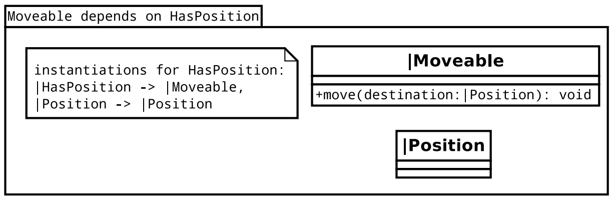
Moveable depends on HasPosition
instantiations for HasPosition:
|HasPosition -> |Moveable,
|Position -> |Position
|Moveable
+move(destination:|Position): void
|Position

Figure 6: State view for Moveable.

comes with a library containing some very low-level reusable packages. One of those, ZeroToMany, provides the functionality of a zero-to-many association by keeping a Set object, along with some typical methods that operate on a set (testing presence, etc). HasPosition depends on the ZeroToMany package and instantiates it. It also binds ZeroToMany's complete *getAssociated()* method to make it public under the new name *getAtPosition()*.

With the packages seen until now, we can instantiate a turn-based game framework, we can provide objects with health and attack functionality, and move them around. One crucial reusable piece of functionality remains, though: The possibility of objects to observe their surroundings.

4.2.4. ObserveSurroundings and HasNeighbors

The package ObserveSurroundings depends on HasNeighbors and HasPosition.

HasNeighbors defines a class with a zero-to-many association with itself[5].

Now, the ObserveSurroundings package instantiates the |HasNeighbors class to be (yet another incomplete) |Position class. This means the |Position class keeps a list of neighbor |Positions. Later on, the |Position class will be instantiated to be a Tile, so that makes sense. We bind *getNeighbors()* to the public (so |Actor can call it) *getNeighborPositions()* method.

[5]This could have been accomplished by making the package depend on the ZeroToMany package mentioned in Section 4.2.3, but a potential bug in the weaving algorithm causes TouchRAM come up with a wrongly woven model when having the mandatory instantiation parameters |Data and |Associated instantiated as being the same class.

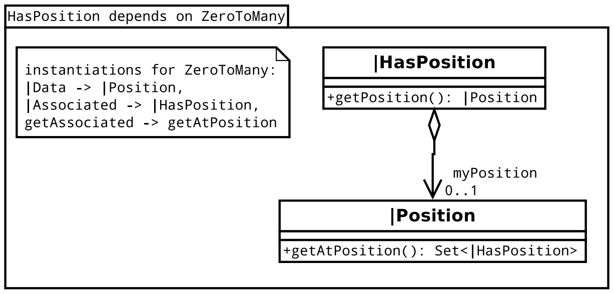
HasPosition depends on ZeroToMany
instantiations for ZeroToMany:
|Data -> |Position,
|Associated -> |HasPosition,
getAssociated -> getAtPosition
|HasPosition
+getPosition(): |Position
myPosition
0..1
|Position
+getAtPosition(): Set<|HasPosition>

Figure 7: State view for HasPosition.

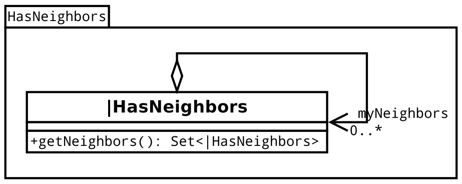

Figure 8: State view for HasNeighbors.

ObserveSurroundings also instantiates 2 parameters from HasNeighbors: |Position becomes |Position, |HasPosition becomes |Actor. There is also a binding from *getAtPosition()* to *getActorsAtPosition()*. Like with *getNeighbors()*, this is necessary because |Actor will use it for its implementation of *getNeighbors()*.

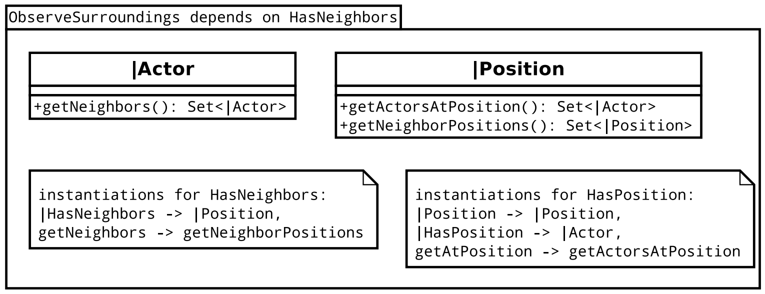
ObserveSurroundings depends on HasNeighbors
|Actor
+getNeighbors(): Set<|Actor>
|Position
+getActorsAtPosition(): Set<|Actor>
+getNeighborPositions(): Set<|Position>
instantiations for HasNeighbors:
|HasNeighbors -> |Position,
getNeighbors -> getNeighborPositions
instantiations for HasPosition:
|Position -> |Position,
|HasPosition -> |Actor,
getAtPosition -> getActorsAtPosition

Figure 9: State view for ObserveSurroundings.

|*Actor.getNeighbors()* is the actual piece of functionality in this package. Its message view can be seen in Figure 10. Clearly, the actor object first asks its position for neighboring positions. The actor object then asks each of those neighboring positions for the actors located at it. The results of these calls are merged and returned.

All of the packages seen until now were application independent. We will now focus on application-specific packages.

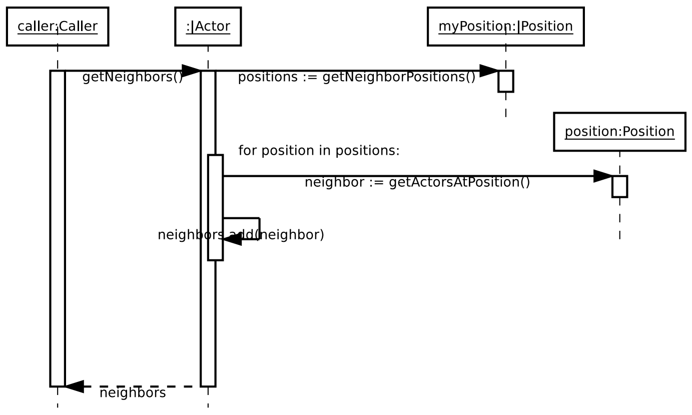

Figure 10: Message view for *|Actor.getNeighbors()*.

4.2.5. TurnBasedGameWithRPGameCharacters

This package contains most of the application-specific functionality of RPGame. It depends directly or indirectly on all of the previously discussed packages (see Figure 1).

The package defines the behavior of the characters in an RPGame. More specifically, it instantiates TurnBasedGame's |Player class, along with that class' *generateActions()* and *execute()* methods, as elements in the abstract (but complete) Character class. Villain and Hero derive from that class and implement Character's abstract methods. This is a nice example of mixing Aspect Oriented Modeling and O.O. to get the benefits of both paradigms. In this case we needed O.O.'s benefit of polymorphism.

Hero and Villain both have their own implementation of *generateActions()*, because that way, we will be able modify them separately if the requirements change.

TurnBasedGame's |Action class is also instantiated. An object of the Action class is nothing but a container in which all the information needed to execute the action is stored. Action has 2 subtypes: Attack and Move. Both were already associated with the |Actor type (in this case Character). Attack is additionally associated with a "victim" character. Move is additionally associated with a destination position.

All other classes are redeclared as mandatory instantiation parameters.

4.2.6. RPGame

Finally, our most high-level package simply instantiates everything that wasn't instantiated by TurnBasedGameWithRPGameCharacters already.

|**Game** is instantiated as RPGame. This is simply a renaming.

|**Position** is instantiated as Tile. This is also simply a renaming.

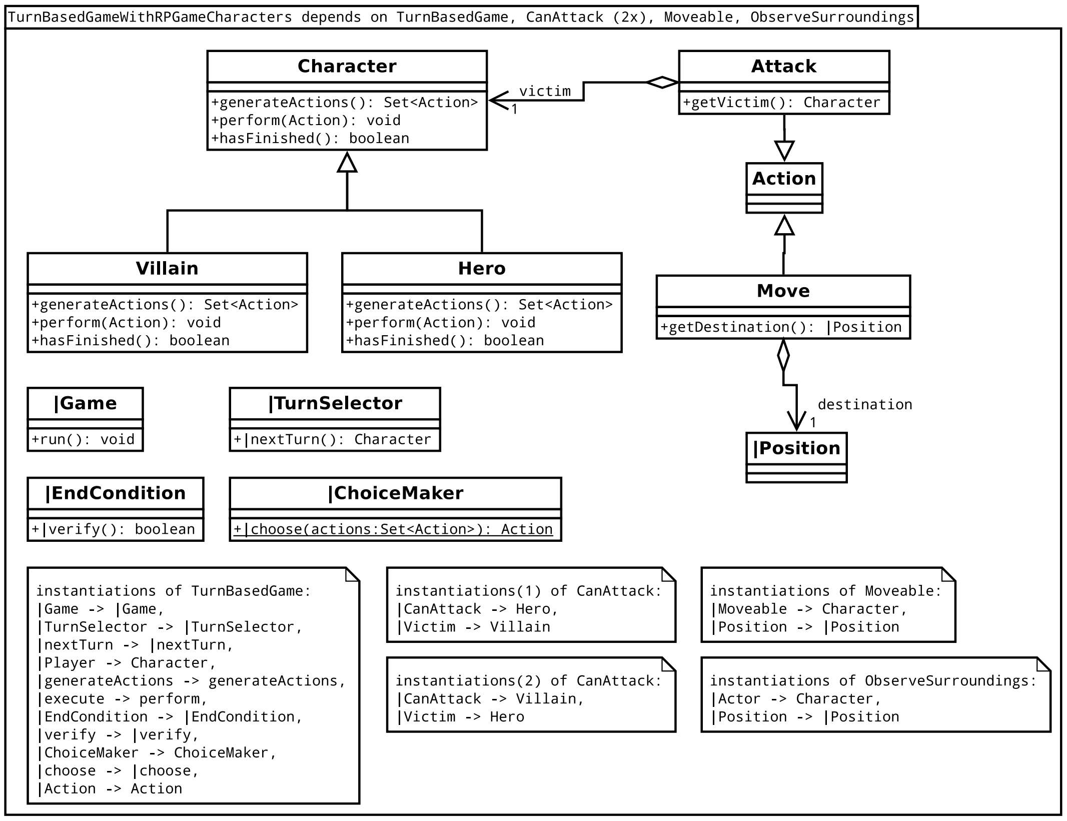

Figure 11: State view for TurnBasedGameWithRPGameCharacters.

|TurnSelector is instantiated as RoundRobinTurnSelector. The functionality to select characters round-robin is provided by depending on ZeroToManyOrdered.

|EndCondition is instantiated as RPGEndCondition. This introduces the check whether the Hero is dead or has won the game.

|ChoiceMaker is instantiated as RandomChoiceMaker. Given an input of type Set¡Action¿, a random element is returned.

Character is bound to the empty Character class to be able to use this typename in TouchRAM.

Action is bound to the empty Action class for the same reason as Character.

Figure 13 shows all packages woven together.

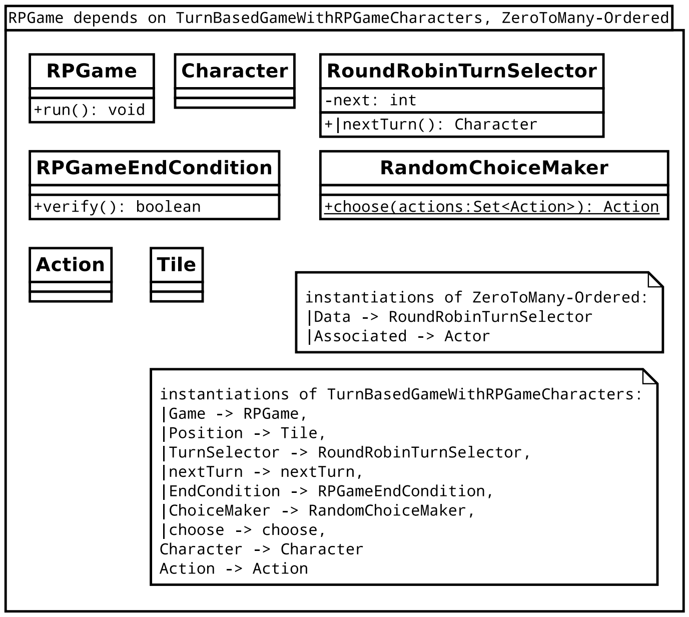
RPGame depends on TurnBasedGameWithRPGameCharacters, ZeroToMany-Ordered
RPGame
+run(): void
Character
RoundRobinTurnSelector
-next: int
+|nextTurn(): Character
RPGameEndCondition
+verify(): boolean
RandomChoiceMaker
+choose(actions:Set<Action>): Action
Action
Tile
instantiations of ZeroToMany-Ordered:
|Data -> RoundRobinTurnSelector
|Associated -> Actor
instantiations of TurnBasedGameWithRPGameCharacters:
|Game -> RPGame,
|Position -> Tile,
|TurnSelector -> RoundRobinTurnSelector,
|nextTurn -> nextTurn,
|EndCondition -> RPGameEndCondition,
|ChoiceMaker -> RandomChoiceMaker,
|choose -> choose,
Character -> Character
Action -> Action

Figure 12: State view for RPGame.

5. Conclusion

5.1. Improvements

There are a couple of things that could be improved upon in the current design.

- The TurnBasedGame package has a single instantiation parameter (a static method) to pick from a list of actions. That makes it impossible to have different "choice" functionality for different players. Multiplayer or human vs. computer is impossible. It would be better to have a choice function associated with every player.

- The |Game object in TurnBasedGame is composed of two other objects, namely |TurnSelector and |EndCondition. After discussing all the packages and analyzing their synergies, it has become clear to me that they could all be merged into the game object. Advantages of this approach:

  - List or set of players becomes an attribute of the game object itself (instead of |TurnSelector). That means |EndCondition can also access this information, in order to query players for their state (e.g. if Hero dead, then end condition evaluates to true).
  - Smaller amount of classes and instantiation parameters in TurnBasedGame package.

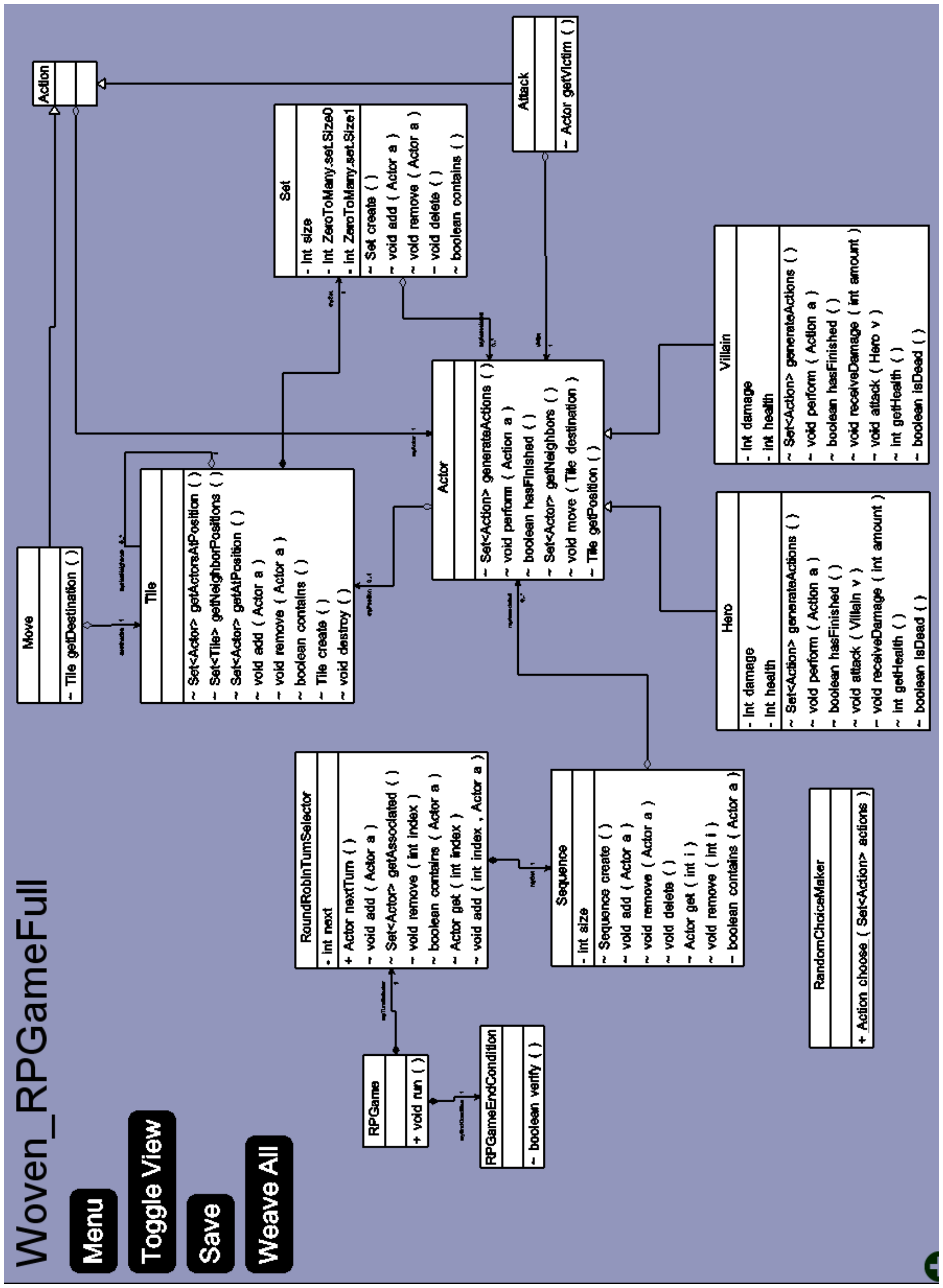
Woven_RPGameFull
Menu
Toggle View
Save
Weave All
Action
Attack
~ Actor getVictim ( )
Set
- int size
- int ZeroToMany.set.Size0
- int ZeroToMany.set.Size1
~ Set create ( )
~ void add (Actor a)
~ void remove (Actor a)
~ void delete ( )
~ boolean contains ( )
Move
~ Tile getDestination ( )
Tile
~ Set<Actor> getActorsAtPosition ( )
~ Set<Tile> getNeighborPositions ( )
~ Set<Actor> getAtPosition ( )
~ void add (Actor a)
~ void remove (Actor a)
~ boolean contains ( )
~ Tile create ( )
~ void destroy ( )
Actor
~ Set<Action> generateActions ( )
~ void perform (Action a)
~ boolean hasFinished ( )
~ Set<Actor> getNeighbors ( )
~ void move (Tile destination)
~ Tile getPosition ( )
Villain
- int damage
- int health
~ Set<Action> generateActions ( )
~ void perform (Action a)
~ boolean hasFinished ( )
~ void receiveDamage (int amount)
~ void attack (Hero v)
~ int getHealth ( )
~ boolean isDead ( )
Hero
- int damage
- int health
~ Set<Action> generateActions ( )
~ void perform (Action a)
~ boolean hasFinished ( )
~ void attack (Villain v)
~ void receiveDamage (int amount)
~ int getHealth ( )
~ boolean isDead ( )
RoundRobinTurnSelector
- int next
+ Actor nextTurn ( )
~ void add (Actor a)
~ Set<Actor> getAssociated ( )
~ void remove (int index)
~ boolean contains (Actor a)
~ Actor get (int index)
~ void add (int index , Actor a)
Sequence
- int size
~ Sequence create ( )
~ void add (Actor a)
~ void remove (Actor a)
~ void delete ( )
~ Actor get (int i)
~ void remove (int i)
~ boolean contains (Actor a)
RandomChoiceMaker
+ Action choose (Set<Action> actions)
RPGame
+ void run ( )
RPGameEndCondition
~ boolean verify ( )

Figure 13: State view of independent model of RPGame, generated by TouchRAM.

5.2. Reusability

The most impressive advantage of RAM. When developing an application using, e.g. O.O. programming, the user is encouraged to split common behavior into a superclass, in the hope that it would yield a reusable module. When building a model with RAM, the act of decomposing itself is thinking about reusability.

With O.O., multiple inheritance is often badly supported, e.g. in the case of Java, only virtual multiple inheritance is supported. With RAM, aspect packages are so small that the modeler is forced to make his higher-level modules depend on many lower-level ones.

Generic programming provides some means to make a piece of code independent of a type it operates on. RAM takes this to the next level by making any kind of element (class/method/state/object) from a package available for instantiation outside that package.

5.3. The "RAM intuition"

For someone who is new to RAM, the decomposing might need some practice. For instance, someone used to decomposing systems into abstraction-specialization modules, could be tempted to write a package A which provides an interface, and then have another package B which depends on A and implements that interface. This is absolutely wrong! There should be no such thing as a package which only declares an interface. A package provides functionality. An interface is not functionality.

In [3], the authors mention that the easiest way to break down a system into RAM packages, is to start with the full requirements. The modeler then tries to detect a reusable piece of functionality, and creates a high-level RAM package which fulfills this functionality. Recursively, the modeler tries to find a piece of reusable functionality in the newly created package, going deeper and deeper until he hits a barrier at which the functionality of a package is so low-level that he can not subdivide any further. A typical example of such a barrier is a package which only consists of a one-way association between two incomplete classes.

I have found this top-down technique to be extremely useful. Actually, it felt like it was the only possible way to get the job done. I've tried a bottom-up approach as well, but every time I got stuck. This novel way of thinking was very interesting to do.

5.4. Consistency and Scalability

In [4] the authors intended to tackle the consistency and scalability issues that came with existing multi-view modeling techniques.

Because message view editing isn't supported yet, consistency checks in TouchRAM don't seem to do much. According to [1], if a check fails the weaving ends with an error. I've never had such an error, so from my practical experience I cannot assess this quality.

Trying to make every package as small and reusable as possible should have a good influence on the scalability of models. The average number of classes in my packages is 3.4. That number would be even lower if the |Game class got merged with |TurnSelector and |EndCondition, as described in Section 5.1. I potentially see one problem: When decomposing a very large system, the complete requirements can maybe be overwhelming. Is it still possible to detect reusable functionality? Further research required.

6. Future work

6.1. RAM

Some research is still needed to verify whether the RAM approach is suitable for huge systems.

6.2. TouchRAM

The people behind TouchRAM have planned a lot of extra features for the future.[2]

References

[1] W. Al Abed, V. Bonnet, M. Schöttle, O. Alam, and J. Kienzle. Touchram: A multitouch-enabled tool for aspect-oriented software design. In *submitted to the 5th Intl. Conference on Software Language Engineering (SLE 2012).*

[2] J. Kienzle. Touchram website. `http://www.cs.mcgill.ca/~joerg/SEL/TouchRAM.html`, November 2012.

[3] J. Kienzle, W. Al Abed, F. Fleurey, J.M. Jézéquel, and J. Klein. Aspect-oriented design with reusable aspect models. *Transactions on aspect-oriented software development VII*, pages 272–320, 2010.

[4] J. Kienzle, W. Al Abed, and J. Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98. ACM, 2009.