CSCI 5448 - Project Final Report

Michael R. Shannon

November 28, 2018

Name: Michael R. Shannon

Title: HexMek

Project Summary: A board game engine based on the Classic BattleTech ruleset. A tactical strategy game where units move on a hexagonal grid and fire various weapons (with hits controlled by dice rolls) in an attempt to eliminate the opposing team.

Features

Implemented Features

ID	Description	Category	Priority
REQ.USR.4	Player's can view a list of all units in the game, with unique ID, type, coordinates, and health/armour percentage.	Views	High
REQ.USR.5	Player's can view the game board, showing location and orientation of units.	Views	High
REQ.USR.6	Player's can view the game board, showing the terrain type of each tile.	Views	High
REQ.USR.7	Player's can view the health/armour of any unit.	Views	High
REQ.USR.8	Player's can view the movement points of any unit.	Views	High
REQ.USR.9	Player's can view a list of weapons of any unit, showing location, range, damage, and whether it has been destroyed.	Views	High
REQ.USR.10	Player's can view percent to hit ratio against any other unit for all weapons on the current unit.	Views	High
REQ.USR.11	Player's can choose which unit to move when it is their turn.	Movement	High
REQ.USR.12	Player's can choose to not move, cruise or flank.	Movement	Low
REQ.USR.13	Player's can rotate (in any of the hexagonal directions), move forward, or (move backward if not flanking) until out of movement points.	Movement	High
REQ.USR.14	Player's can choose which unit will fire when it is their turn.	Combat	High
REQ.USR.15	Player's can choose which weapons to fire.	Combat	High

Not Implemented Features

ID	Description	Category	Priority
REQ.USR.1	Player's can choose from a list of units when starting the game by spending from the pool of battle value points.	Game Setup	Low
REQ.USR.2	Player 1 can place their units within 3 tiles of the south end of the board.	Game Setup	Low
REQ.USR.3	Player 2 can place their units within 3 tiles of the north end of the board.	Game Setup	Low

Final Class Diagram

The overall structure only changed in a couple places, that is the MVC (Model View Controller) and the movement classes. In the former case it was primarily because when the original diagram was constructed the models and views were in the initial design phase, with very little time given to finalize either the controllers or the views.

In the case of the movement classes the change is a complete change of design patterns. Originally, movement was implemented using the Command pattern allowing the player to each step of a move. For simplicity both in implementation (mainly due to a lack of friend functions) and in user interface this was changed to the Memento pattern seen in Figure 1 and in more detail in Figure 10. This is the single biggest change from the start of the project to the end.

A small change is that the logic of navigating and line drawing in the hexadecimal coordinate system is no longer shared between the Coordinate class (now the Hex class) and the HexMap class and is now entirely in the Hex class. The HexMap itself (which represents the game board/map) is now a simple container for Tile's (representing types of terrain) that are referenced by a Hex coordinate key.

Another change is that while a partial Composite pattern was initially part of the design for Unit's and their components it was not complete. This was changed to a complete Composite pattern with the introduction of the Damageable interface. This will be covered in more detail in the Composite Pattern section.

The Direction class also changed from a container for an enumeration to an interface with 6 implementations representing the directions of: north, north-east, south-east, south, south-west, and north-west. In this way it was changed from more of a data based implementation to and object oriented implementation, in particular, because the rotateLeft and rotateRight methods return a new Direction it is a small implementation of the state pattern with an operation on each state returning a new state. This is a pattern that is also used for the controllers, allowing each controller to decide on what the next phase (controller) should be.

The final change I will cover is the addition of the RandomSingleton class to solve insecure seeding issues with Java's Random implementation.

As always with a project of this scope, up front design is great for overall structure but in my experience with this and past projects at least 20% of the design will end up changing, especially the details and this project was no exception. Given this, the class diagram did prove invaluable in providing a direction and served as my method of tracking progress as I would change the color of completed classes. If I were to do a project of this size without such a diagram the number of refactoring commits would be much higher.

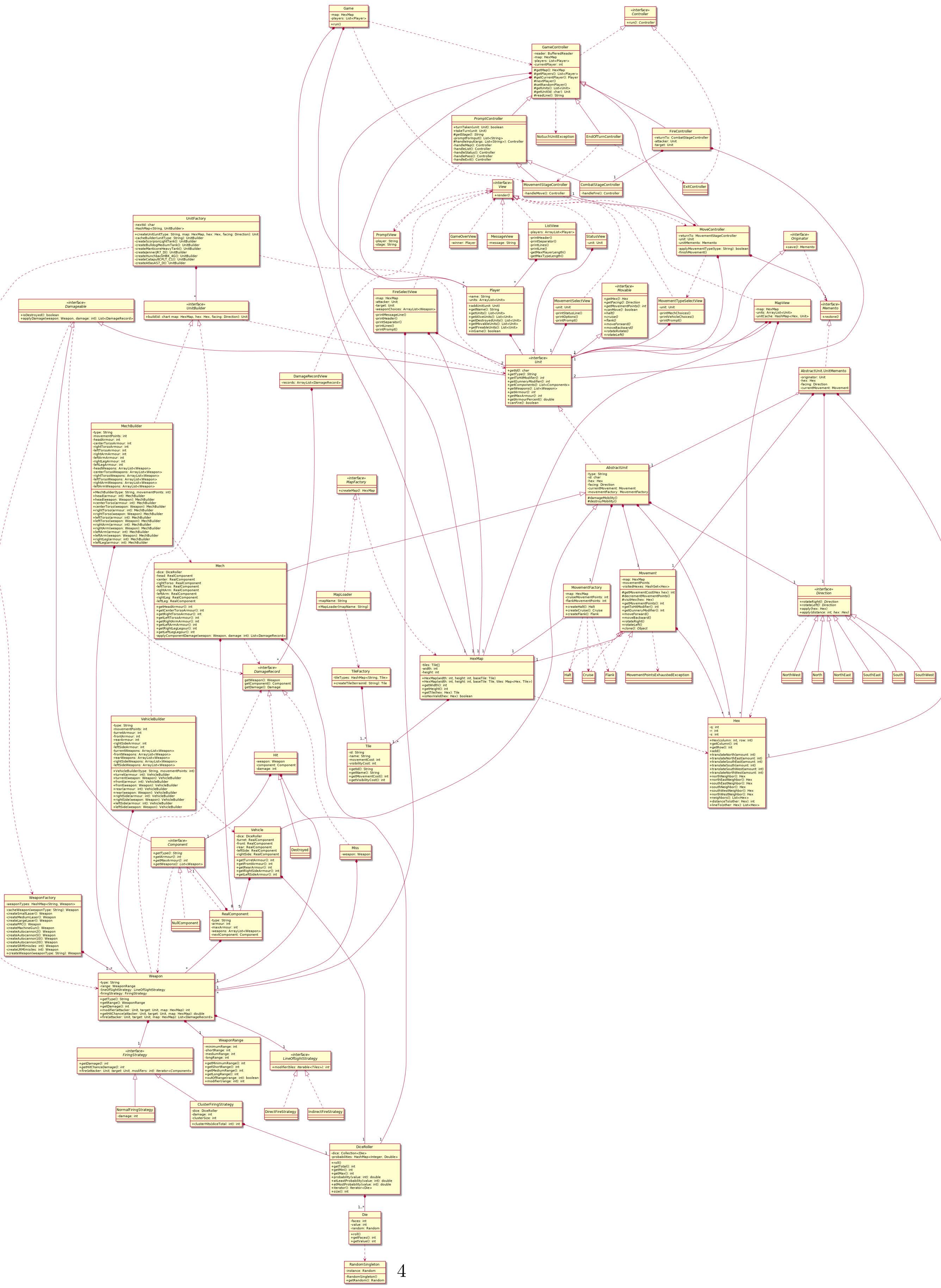

Figure 1: Final class diagram of the HexMek project.

Design Patterns Used

There are 8 design patterns implemented in the project a total of 15 times. Therefore, for the most part only one of the implementations will be show.

State Pattern

The State pattern is used to allow the state of an object to behave differently (via composition) at different times.

The State pattern is used twice. It is used to change movement states between halting, cruising, and flanking, allowing each type of movement to behave differently. It is also used for the controllers in the MVC pattern to allow the controller to change throughout the game. In this latter case the State has a method that returns the next State, making the State the Client. I will be covering the first case in detail.

Figure 2 shows the use of the State pattern for movement. The methods of the Unit interface have been removed to make the diagram fit on the page, they are not part of the state pattern. The Context is the AbstractUnit class and the State is the Movement class. The concrete states are the Halt, Cruise, and Flank classes which change which type of moves are allowed. In this case the state is changed by the halt, cruise, and flank methods declared in the Movable interface and thus in the AbstractUnit class. Therefore, in this case the Context is also the Client. This also shows the Abstract Factory pattern in the form of the MovmentFactory which is used to create the states.

The reason I used the State pattern here is that I needed different behavior depending on which type of movement was chosen. By choosing the State design pattern I was able to move the type specific logic into the Movement (State) classes and avoid conditional usage in the AbstractUnit class.

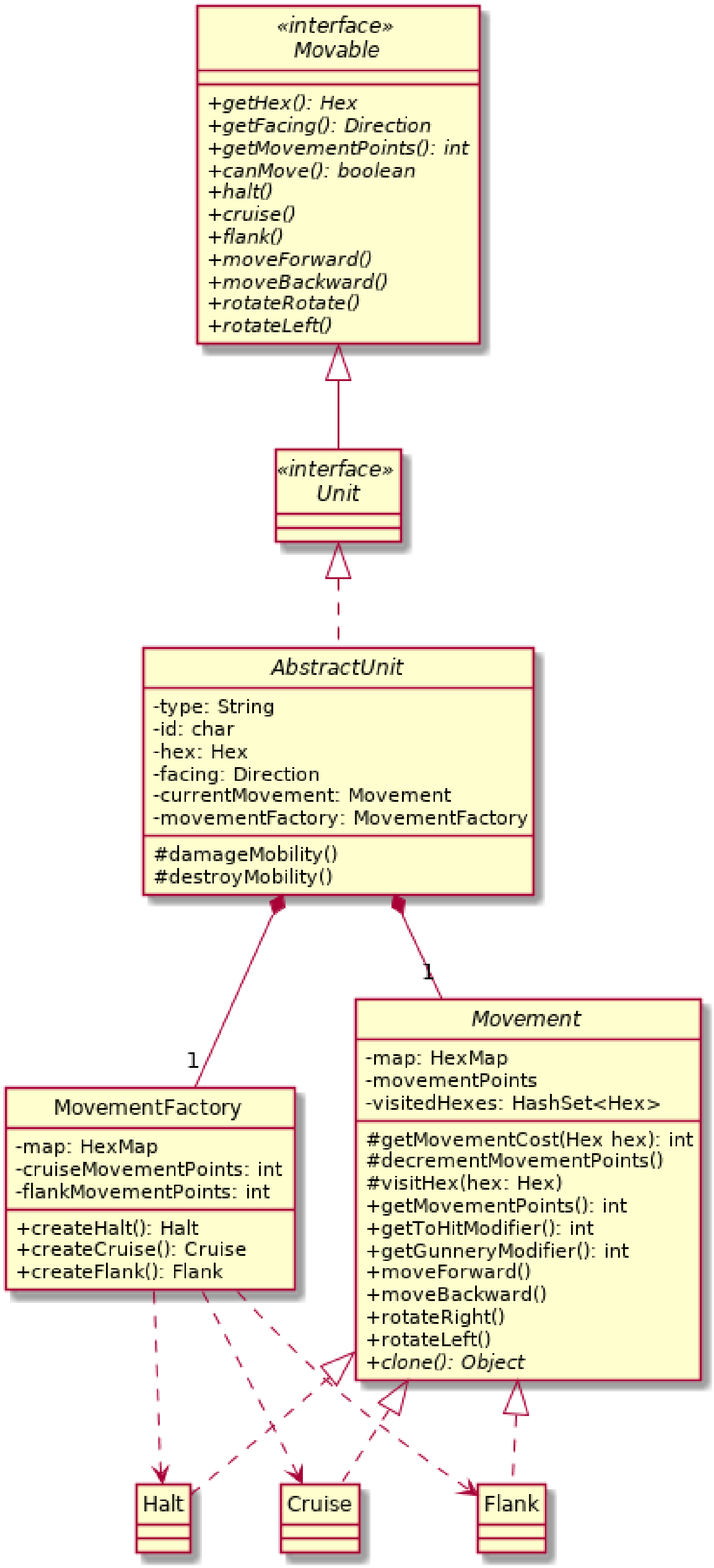

Figure 2: Application of State pattern.

Strategy Pattern

The Strategy pattern is used to allow for pluggable strategies and is typically used to allow the same code to use multiple algorithms. It is similar to the State pattern but is about polymorphic algorithms instead of polymorphic state.

The Strategy pattern is used twice, with the same context, that is the Weapon class. It is used for both the firing strategy (single shot vs cluster firing) and line of sight strategy (direct fire vs indirect). This is shown in Figure 3, where Weapon is the context and both the LineOfSightStrategy and FiringStrategy interfaces are strategies, the former with DirectFireStrategy and IndirectFireStrategy concrete strategies and the latter with NormFiringStragetgy and ClusterFiringStrategy concrete strategies. These two separate applications of the strategy pattern allow the Weapon class to be constructed by composition instead of inheritance. This is very convenient since there are many types of weapons.

The line of sight strategies provide a to-hit modifier based on intervening terrain, allowing weapons that are fired in a straight line and weapons that are fired in a high arc, ignoring terrain. The firing strategies allow single shot and cluster firing weapons. I chose to use a strategy pattern here in order to avoid excessive inheritance and reimplementation in subclasses. In a language with multiple inheritance it would be possible to achieve the same thing with inheritance but is typically frowned upon.

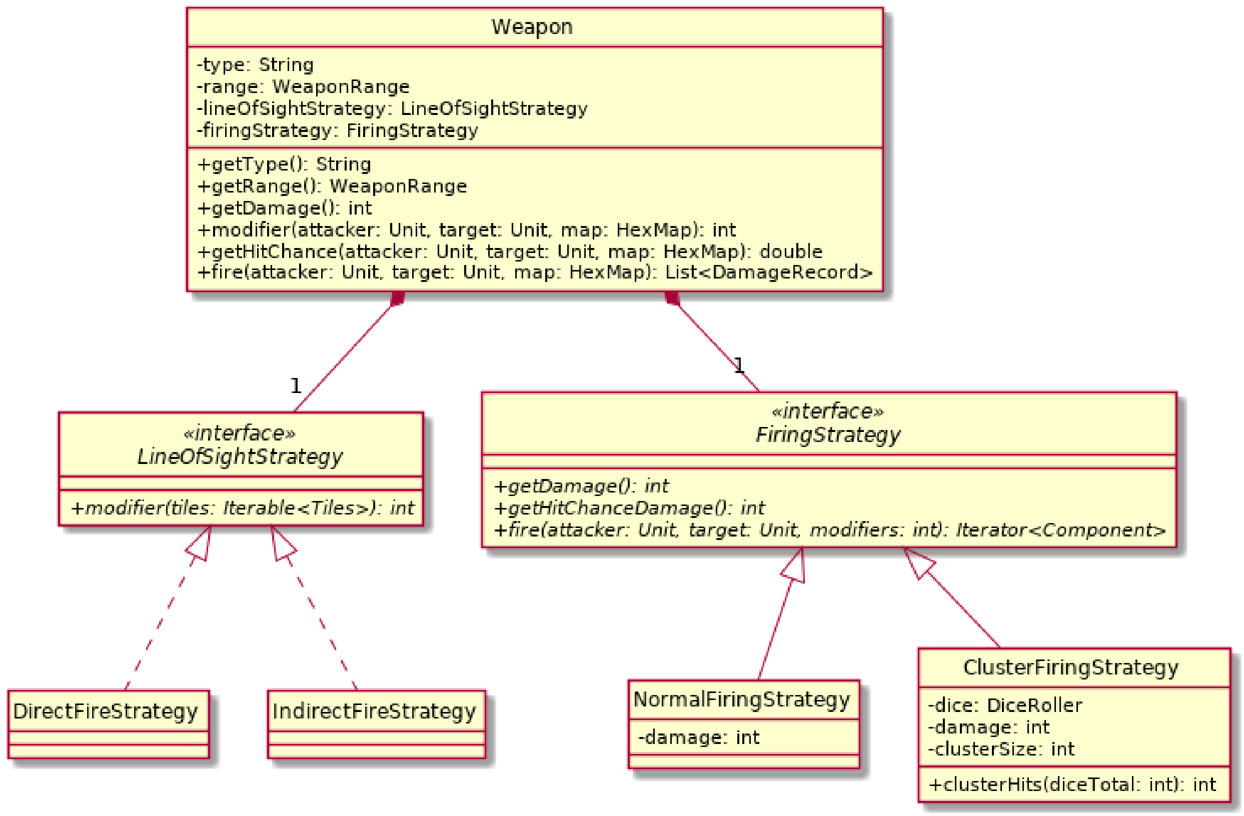
Weapon
-type: String
-range: WeaponRange
-lineOfSightStrategy: LineOfSightStrategy
-firingStrategy: FiringStrategy
+getType(): String
+getRange(): WeaponRange
+getDamage(): int
+modifier(attacker: Unit, target: Unit, map: HexMap): int
+getHitChance(attacker: Unit, target: Unit, map: HexMap): double
+fire(attacker: Unit, target: Unit, map: HexMap): List<DamageRecord>
1
1
«interface»
LineOfSightStrategy
+modifier(tiles: Iterable<Tiles>): int
«interface»
FiringStrategy
+getDamage(): int
+getHitChanceDamage(): int
+fire(attacker: Unit, target: Unit, modifiers: int): Iterator<Component>
DirectFireStrategy
IndirectFireStrategy
NormalFiringStrategy
-damage: int
ClusterFiringStrategy
-dice: DiceRoller
-damage: int
-clusterSize: int
+clusterHits(diceTotal: int): int

Figure 3: Application of Strategy pattern.

Factory Method Pattern

The Factory Method pattern is about allowing subclasses to control the creation of an object.

The canonical application of the Factory Method pattern in the project is with map creation. There are other factories but they are either the related Abstract Factory pattern of factories used to create and store flyweights. Figure 4 shows the Factory Method applied to map creation. In this case the HexMap is the product and is created by implementations of the MapFactory interface which is the abstract factory. The design currently has only one concrete factory which is the MapLoader class.

How this works is that any class implementing the MapFactory interface has a createMap method that returns a new hexagonal grid map. The current concrete factory (MapLoader) is constructed given a map name and loads a map from files in the resources folder. Using the Factory Method pattern is not necessary for this project but does provide expandability, allowing other methods of map loading to be used in the future. This is the only reason that I chose to implement the Factory Method pattern here.

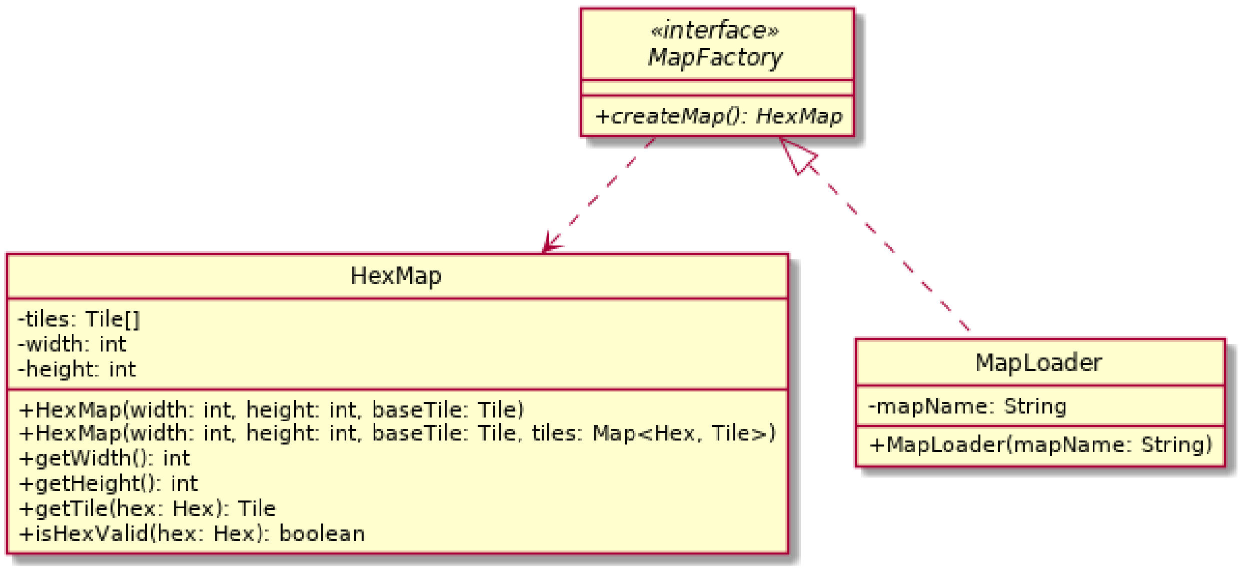
«interface»
MapFactory
+createMap(): HexMap
HexMap
-tiles: Tile[]
-width: int
-height: int
+HexMap(width: int, height: int, baseTile: Tile)
+HexMap(width: int, height: int, baseTile: Tile, tiles: Map<Hex, Tile>)
+getWidth(): int
+getHeight(): int
+getTile(hex: Hex): Tile
+isHexValid(hex: Hex): boolean
MapLoader
-mapName: String
+MapLoader(mapName: String)

Figure 4: Application of Factory Method pattern.

Abstract Factory Pattern

The Abstract Factory pattern is used to allow different subclasses of a common base class (or interface) to be returned from a single construction method based on some criteria not directly relating to the class. This allows the construction method to choose the proper concrete implementation of the abstract class/interface returned from the construction method.

In this project the Abstract Factory pattern is used to construct units (along with the Builder pattern detailed later). All unit's in the game implement the Unit class which is produced by the UnitFactory (the actual factory) and it's createUnit method that builds a unit based on the name of the unit. In reality, all current units inherit from the AbstractUnit class but the factory returns objects that implement the Unit interface. There are two types of Unit's in the game, Mech's and Vehicle's. By use of the Abstract Factory pattern the UnitFactory can return both types of units (cast to the Unit interface) without the type of the unit necessarily being known by the caller of the createUnit method. This layout can be seen in Figure 5 which shows the implementation of the Abstract Factory pattern for unit creation.

I chose this pattern to allow a single method (createUnit) to create all the units in the game without detailed knowledge of unit types being required by the user of the factory. This allows, for the most part, the types of units to be abstracted away simplifying game setup code.

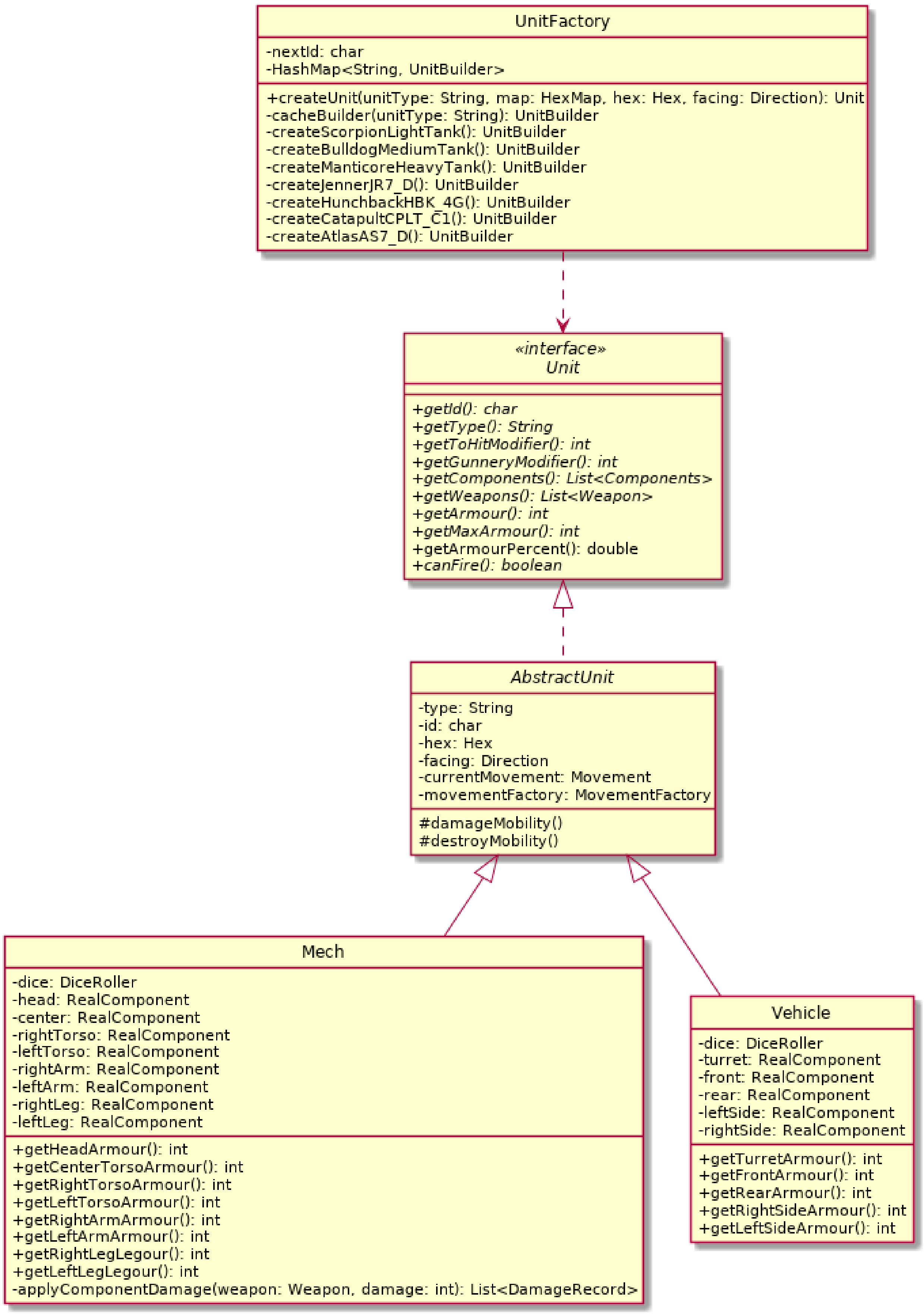
UnitFactory
-nextId: char
-HashMap<String, UnitBuilder>
+createUnit(unitType: String, map: HexMap, hex: Hex, facing: Direction): Unit
-cacheBuilder(unitType: String): UnitBuilder
-createScorpionLightTank(): UnitBuilder
-createBulldogMediumTank(): UnitBuilder
-createManticoreHeavyTank(): UnitBuilder
-createJennerJR7_D(): UnitBuilder
-createHunchbackHBK_4G(): UnitBuilder
-createCatapultCPLT_C1(): UnitBuilder
-createAtlasAS7_D(): UnitBuilder
«interface»
Unit
+getId(): char
+getType(): String
+getToHitModifier(): int
+getGunneryModifier(): int
+getComponents(): List<Components>
+getWeapons(): List<Weapon>
+getArmour(): int
+getMaxArmour(): int
+getArmourPercent(): double
+canFire(): boolean
AbstractUnit
-type: String
-id: char
-hex: Hex
-facing: Direction
-currentMovement: Movement
-movementFactory: MovementFactory
#damageMobility()
#destroyMobility()
Mech
-dice: DiceRoller
-head: RealComponent
-center: RealComponent
-rightTorso: RealComponent
-leftTorso: RealComponent
-rightArm: RealComponent
-leftArm: RealComponent
-rightLeg: RealComponent
-leftLeg: RealComponent
+getHeadArmour(): int
+getCenterTorsoArmour(): int
+getRightTorsoArmour(): int
+getLeftTorsoArmour(): int
+getRightArmArmour(): int
+getLeftArmArmour(): int
+getRightLegLegour(): int
+getLeftLegLegour(): int
-applyComponentDamage(weapon: Weapon, damage: int): List<DamageRecord>
Vehicle
-dice: DiceRoller
-turret: RealComponent
-front: RealComponent
-rear: RealComponent
-leftSide: RealComponent
-rightSide: RealComponent
+getTurretArmour(): int
+getFrontArmour(): int
+getRearArmour(): int
+getRightSideArmour(): int
+getLeftSideArmour(): int

Figure 5: Application of Abstract Factory pattern.

Flyweight Pattern

The Flyweight pattern is used to reduce the amount of memory usage when objects have a large amount of intrinsic (shared state).

There are two implementations of the Flyweight pattern in the project. These are for weapons and for terrain tiles. Because the Tile class is simpler it will be presented, however the implementation is technically the same, it's just that Weapon's are composite objects that also implement the Strategy pattern which makes the class diagram too complex.

Figure 6 shows the implementation of the Flyweight pattern applied to the Tile class which represents terrain tiles. In this implementation the flyweight is the Tile class and the flyweight factory is the TileFactory class. The TileFactory has a static method createTile that returns a Tile given the ID string of the tile. Because the Tile class only has intrinsic state there is no context to hold the extrinsic state as is typical in the Flyweight pattern. Therefore, this implementation is only half of the Flyweight pattern. When the createTile method is invoked a static dictionary of terrain tiles is queried, if the tile is found it is simply returned (as if it where a new tile), if the tile is not it is constructed and cached. This is possible because Tile is immutable.

I chose this pattern for the Tile and Weapon classes for the same reason. Because they both lack extrinsic state there is no reason to create and store many versions of the same object. For the terrain tiles this is important because there are only 3 types of tiles but over 200 tiles in the game. Therefore, it is by quantity that using the Flyweight pattern makes sense. For the Weapon class (not shown) it makes sense not because there are many instances but because there are multiple instances of very large objects.

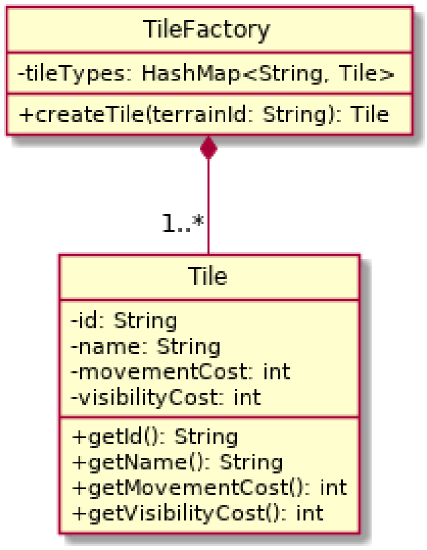

Figure 6: Application of Flyweight pattern.

Partial Builder Pattern

The builder pattern is about reducing the construction of an object into steps so that different types can be constructed from a common interface.

In this project I only use the first part of the pattern, that is breaking construction into steps. This can be seen in Figure 7 where we see the VehicleBuilder and MechBuilder builders that allow building of units in a step by step fashion. How it works is by setting the armour and weapons of the components of a unit and then finally building the unit with the build method of the UnitBuilder interface. The UnitFactory class is responsible for performing the building steps.

The reason I chose this pattern is not for the typical benefits of the Builder pattern but to get around the lack of keyword arguments in Java. By using half of the builder pattern I was able to achieve a similar interface to keyword arguments. This allowed zero or more weapons to be added to each component of the unit (such as the left torso) without requiring setters on the units themselves. This allows the Unit classes (Mech and Vehicle) to be self contained and not expose to much of their implementation.

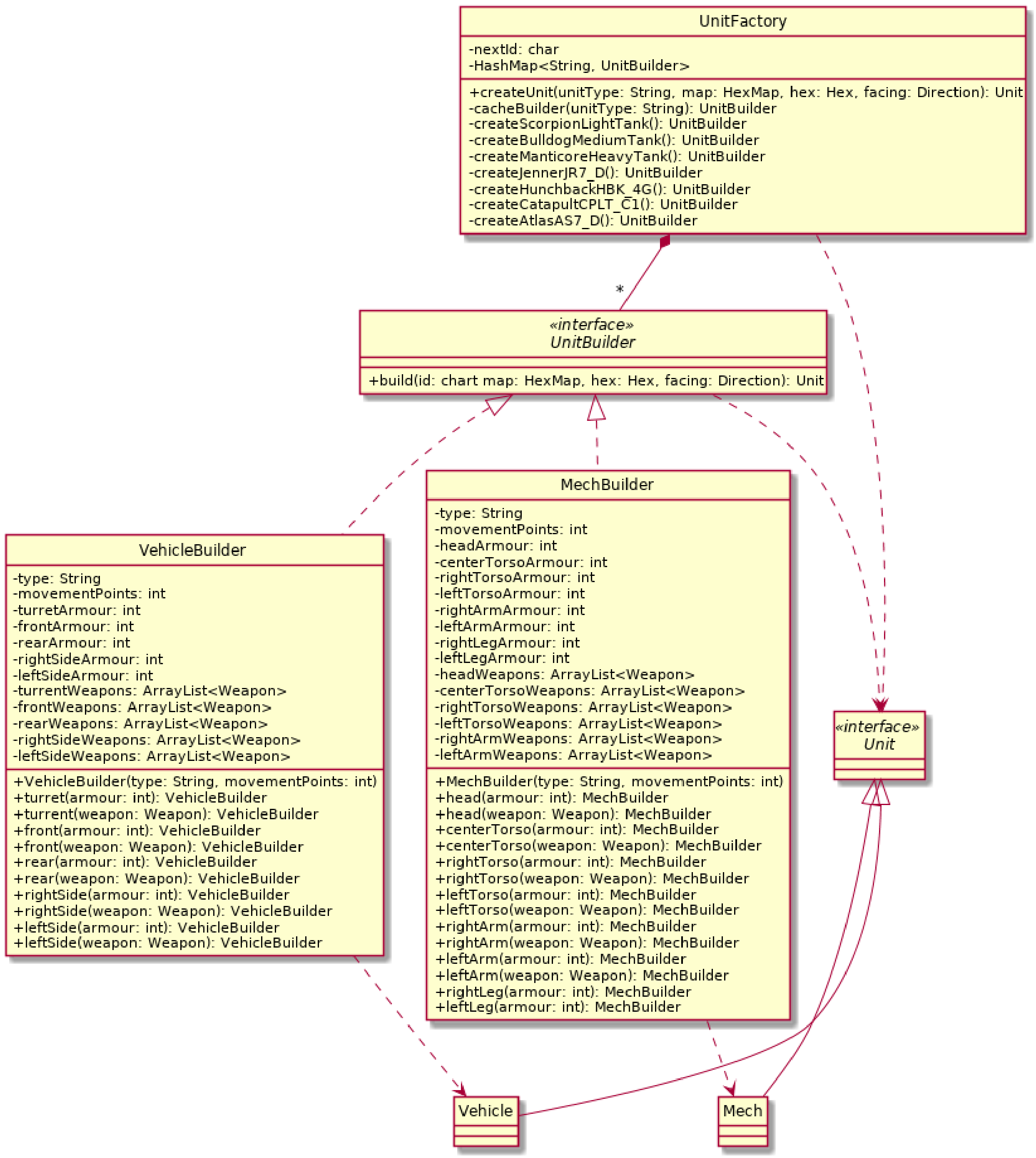
UnitFactory
-nextId: char
-HashMap<String, UnitBuilder>
+createUnit(unitType: String, map: HexMap, hex: Hex, facing: Direction): Unit
-cacheBuilder(unitType: String): UnitBuilder
-createScorpionLightTank(): UnitBuilder
-createBulldogMediumTank(): UnitBuilder
-createManticoreHeavyTank(): UnitBuilder
-createJennerJR7_D(): UnitBuilder
-createHunchbackHBK_4G(): UnitBuilder
-createCatapultCPLT_C1(): UnitBuilder
-createAtlasAS7_D(): UnitBuilder
*
«interface»
UnitBuilder
+build(id: chart map: HexMap, hex: Hex, facing: Direction): Unit
VehicleBuilder
-type: String
-movementPoints: int
-turretArmour: int
-frontArmour: int
-rearArmour: int
-rightSideArmour: int
-leftSideArmour: int
-turrentWeapons: ArrayList<Weapon>
-frontWeapons: ArrayList<Weapon>
-rearWeapons: ArrayList<Weapon>
-rightSideWeapons: ArrayList<Weapon>
-leftSideWeapons: ArrayList<Weapon>
+VehicleBuilder(type: String, movementPoints: int)
+turret(armour: int): VehicleBuilder
+turrent(weapon: Weapon): VehicleBuilder
+front(armour: int): VehicleBuilder
+front(weapon: Weapon): VehicleBuilder
+rear(armour: int): VehicleBuilder
+rear(weapon: Weapon): VehicleBuilder
+rightSide(armour: int): VehicleBuilder
+rightSide(weapon: Weapon): VehicleBuilder
+leftSide(armour: int): VehicleBuilder
+leftSide(weapon: Weapon): VehicleBuilder
MechBuilder
-type: String
-movementPoints: int
-headArmour: int
-centerTorsoArmour: int
-rightTorsoArmour: int
-leftTorsoArmour: int
-rightArmArmour: int
-leftArmArmour: int
-rightLegArmour: int
-leftLegArmour: int
-headWeapons: ArrayList<Weapon>
-centerTorsoWeapons: ArrayList<Weapon>
-rightTorsoWeapons: ArrayList<Weapon>
-leftTorsoWeapons: ArrayList<Weapon>
-rightArmWeapons: ArrayList<Weapon>
-leftArmWeapons: ArrayList<Weapon>
+MechBuilder(type: String, movementPoints: int)
+head(armour: int): MechBuilder
+head(weapon: Weapon): MechBuilder
+centerTorso(armour: int): MechBuilder
+centerTorso(weapon: Weapon): MechBuilder
+rightTorso(armour: int): MechBuilder
+rightTorso(weapon: Weapon): MechBuilder
+leftTorso(armour: int): MechBuilder
+leftTorso(weapon: Weapon): MechBuilder
+rightArm(armour: int): MechBuilder
+rightArm(weapon: Weapon): MechBuilder
+leftArm(armour: int): MechBuilder
+leftArm(weapon: Weapon): MechBuilder
+rightLeg(armour: int): MechBuilder
+leftLeg(armour: int): MechBuilder
«interface»
Unit
Vehicle
Mech

Figure 7: Partial application of partial Builder pattern.

Singleton Pattern

The Singleton pattern is about ensuring there can only be one instance of a class and providing global access to that instance.

In this project the Singleton pattern, as seen in Figure 8, is used to overcome the issue of rapid creation of Random instances. This is required because Java neither provides a global random nor does it use /dev/random for it's seed (which would ensure each new instance had a different seed).

In this project the Die class is the only client of the singleton and RandomSingleton is the singleton. RandomSingleton is not a typical singleton, but instead exposes an instance of Java's Random as a singleton. A typical singleton would provide an instance of itself. I could have used the canonical Singleton pattern by subclassing Random but composition achieves the same effect without the complexities of inheritance.

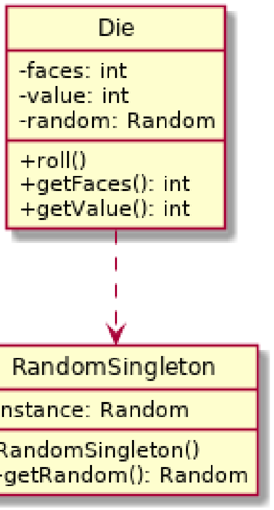

Figure 8: Application of Singleton pattern.

Composite Pattern

The Composite pattern is about treating a tree structure of objects, sometimes of different classes as if they are a single object.

In this project the Composite pattern is primarily used for application and propagation of damage. This is seen in Figure 9 where all parts of the composite implement the Damageable interface. This takes the place of the Component interface in the typical structure of the Composite pattern. Because of this all the classes shown in the diagram: Unit, AbstractUnit, Mech, Vehicle, RealComponent, Component, and NullComponent can take damage via the applyDamage method which is the common interface of the Composite pattern. This is used in two ways, when applying damage to a Unit the unit will apply this damage to the components it contains, which are of the RealComponent class. It is also used to propagate damage from outer components to inner components of a unit. For example, when the left torso of a mech is destroyed any further damage to that component must be propogated to the center torso. Therefore, not only does the Mech class contain a RealComponent that represents the center torso the left torso contains a reference to the center torso. This is the nextComponent property of RealComponent that takes a Component interface and can thus be a RealComponent or a NullComponent. In this way the NullComponent represents a leaf of the tree and thus no damage should be propagated further.

I chose the Composite pattern here to keep the Weapon class, which calls the applyDamage method from needing to know about the internals of Unit's while maintaining consistency so that application of damage can continually be passed down the tree of components.

There is also a smaller application of the Composite pattern in Figure 9 that is used to retrieve the armour. In this parallel Composite pattern the Unit class provides the common interface in the form of the getArmour and getMaxArmour methods. These methods will sum the armour from all of the Components contained within the Unit, again applying the Composite design pattern.

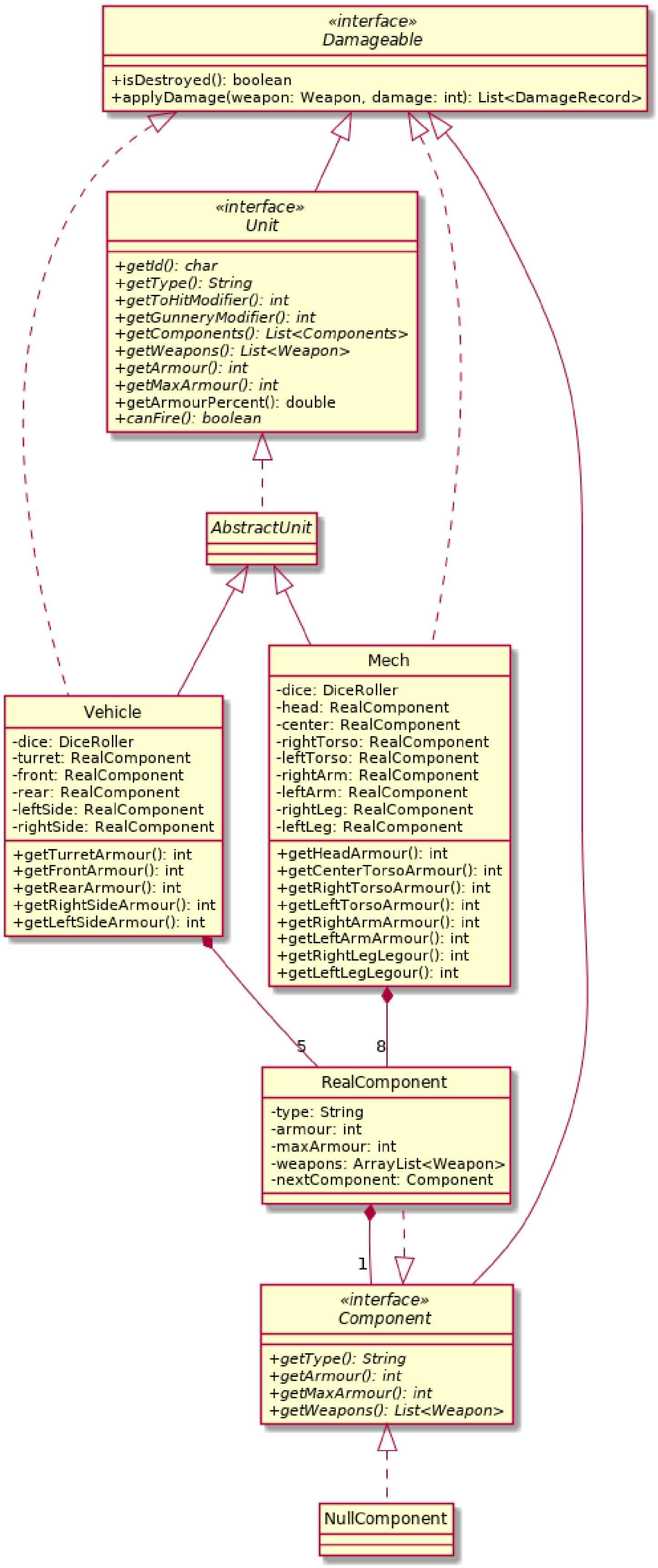

Figure 9: Application of Composite pattern. The properties and methods of the intermediate AbstractUnit class has been removed to avoid clutter.

Memento Pattern

The Memento design pattern is used to allow the state of an object to be saved (without knowing the internal structure of the object to be saved or even it's concrete type) and then restored at a later time.

In this project the Memento pattern is used to save and restore the location and movement of units. Looking at Figure 10 it can be seen that the implementation in this project uses canonical naming for the interfaces with an Originator interface providing a save method that returns a Memento interface. The Originator interface is implemented by any class that can be saved. The Memento interface has a restore method that is intended to restore the state of the Originator that created it.

In the context of the project all Unit's are Originator's and can therefore have their state saved. However, the actual implementation of the Memento is in AbstractUnit.UnitMemento which only saves the location and movement state of the unit. The reason it is not a complete Memento (does not save damage state) is because that would require a tree of Memento's and it is not necessary for this project. That is because I chose the Memento pattern to allow the movement state to be rolled back to before a movement began. This is important because unlike firing, which is an atomic operation, movement requires multiple inputs from the player over time and thus it is necessary to allow them to revert the movement.

The UnitMemento class must be an inner class of AbstractUnit so as to avoid the use of setters in the AbstractUnit. This is a workaround because Java does not have friend methods. By making it an inner class the UnitMemento class can use it's saved AbstractUnit reference to restore the Unit's Hex location, facing Direction, and Movement state by simply calling the restore method of the Memento.

Another pattern I could have used (the pattern I started the design with) is the Command pattern which would have saved every step of the movement. The reason I chose the Memento pattern over the Command pattern is to avoid an excessive amount of inner classes in the AbstractUnit class. By choosing the Memento pattern I only require a single inner class, while if I were to use the Command pattern I would need an inner class for each type of movement, as well as the setting of the Movement state.

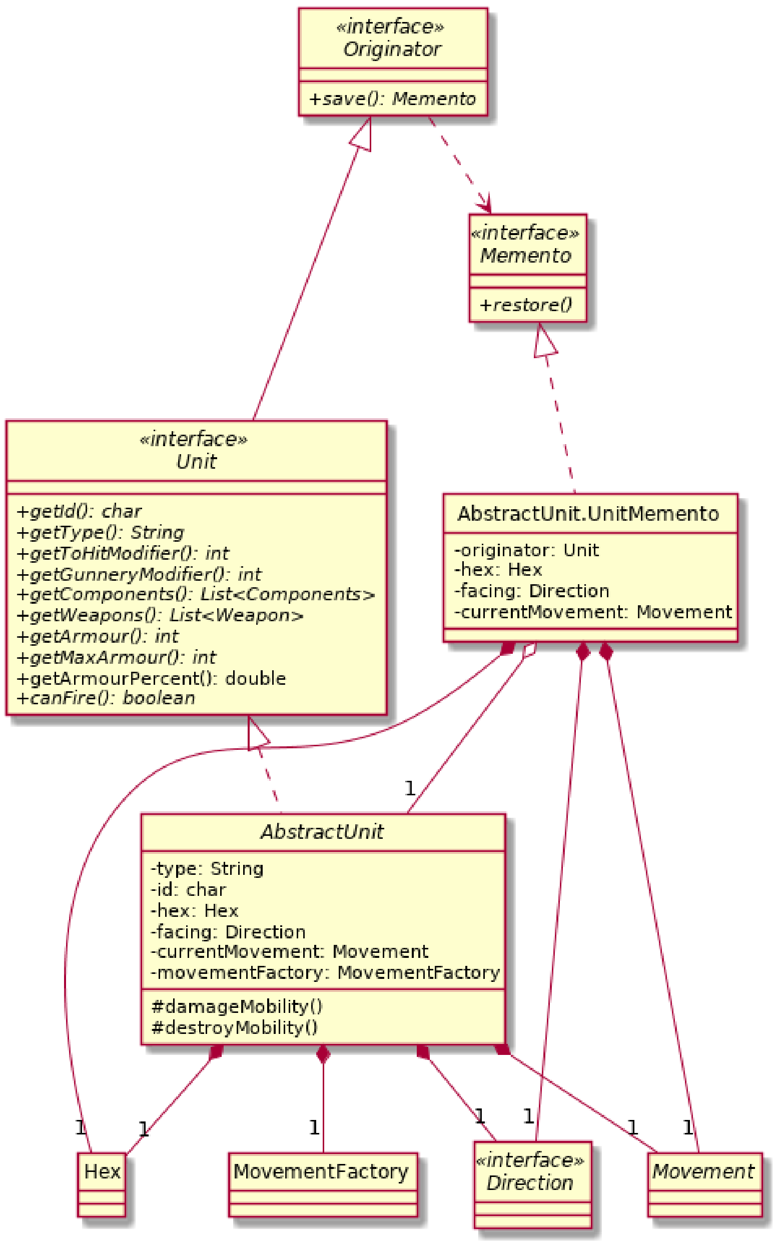

Figure 10: Application of Memento pattern.

Lessons Learned

I have only learned one thing from this project, that is that if you lack knowledge of the working language (Java in my case), or of a pattern used (in my case MVC as it applies to non web applications) the process of designing the application can be difficult and lead to designs that are not possible given the chosen tools/patterns. I had two elements of the design that had to be modified due to a lack of features in Java that I did not know about during the design phase. In particular, my original design assumed the friend feature (not available in Java) which allows one class to manipulate the internals of another class. This is typically bad practice but it is a good technique for implementing the Memento pattern in C++.

The reason I have only learned one lesson from doing this project is because I had previously completed an object oriented project of about 4 times the size using the same methods of analysis and design. In order to give a somewhat legitimate answer I have attempted to remember what I learned while working on that project last year.

An initial class diagram is highly useful, providing guidance until a good working memory is achieved. However, when they become large they are hard to read, especially when one or more classes permeate the code. This happened with my project last year with a class representing a network packet and this year with the Hex class which is a coordinate class in hexadecimal space.

I have also learned that maintaining the UML while sometimes required by management is typically a huge waste of time but if not done the diagrams quickly become useless. A good CASE (Computer-Aided Software Engineering) tool that could update the diagram from the source code would be a great help with maintaining the UML and greatly reduce workload.