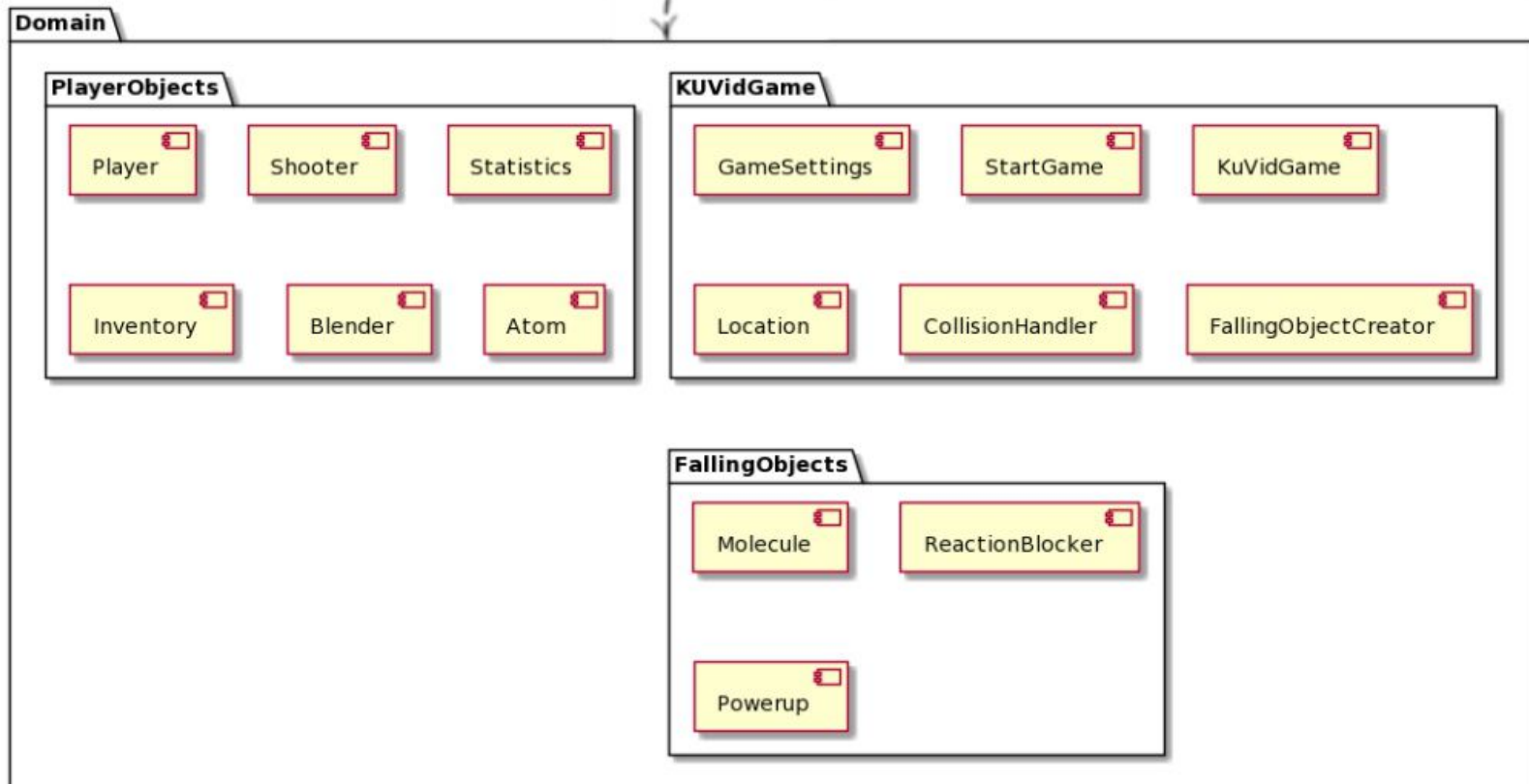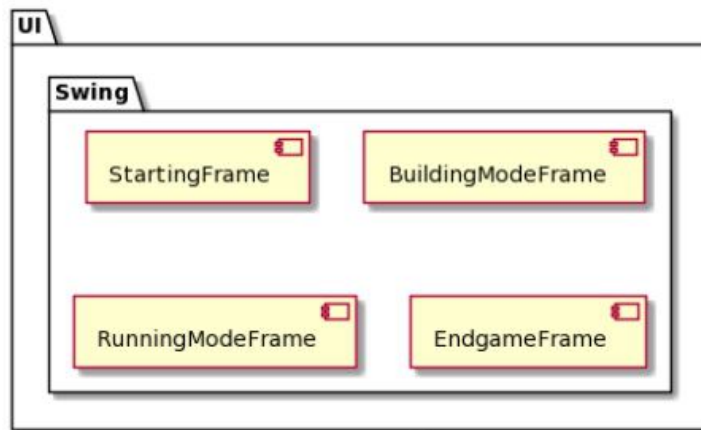Table of Contents

1. Logical Architecture

2. Interaction Diagrams

3. UML Class Diagram

4. Pattern Discussion

UI

Swing

StartingFrame

BuildingModeFrame

RunningModeFrame

EndgameFrame

Domain

PlayerObjects

Player

Shooter

Statistics

Inventory

Blender

Atom

KUVidGame

GameSettings

StartGame

KuVidGame

Location

CollisionHandler

FallingObjectCreator

FallingObjects

Molecule

ReactionBlocker

Powerup

UML Sequence Diagrams

Blend Atoms

Player	Blender	Inventory

blendAtoms(typ1, typ2) → Player

Player → Inventory: checkInventory(typ1)

Inventory ⇠ Player: existing

opt [existing]

Player → Blender: blendAtoms(typ1, typ2)

Blender ⇠ Player: num

Player → Inventory: decreaseAtomNum(typ1, num)

Player → Inventory: increaseAtomNum(typ2)

Break Atom

Player	Blender	Inventory

breakAtom(typ1, typ2) → Player

Player → Inventory: checkInventory(typ1)

Inventory ⇠ Player: existing

opt [existing]

Player → Blender: breakAtom(typ1, typ2)

Blender ⇠ Player: num

Player → Inventory: decreaseAtomNumber(typ1)

Player → Inventory: increaseAtomNumber(typ2, num)

Catch Powerup

Player	Shooter	Inventory

catchPowerup()

loop [isSame]

checkPosition()

isSame

catchPowerup()

type

increasePowerupNumber()

Player	Shooter	Inventory

Pause Game

KUVidGame

pause()

opt [not paused]

paused=true

KUVidGame

Resume Game

KUVidGame

resume()

opt [paused]

paused=false

KUVidGame

Pick Atom

```
:Player          :Inventory        :Shooter

pickAtom()
  →│
   │ randomAtom()
   │───────────────→│
   │   random        │
   │←-----------------│
   │ placeAtom(random)
   │────────────────────────────────→│
```

Pick Powerup

```
Player           Inventory         Shooter

pickPowerup(type)
  →│
   │ checkPowerup(type)
   │───────────────→│
   │   existing      │
   │←-----------------│
   ┌opt───┐ [existing]
   │ placePowerup(type)
   │────────────────────────────────→│
   └────────────────────────────────────┘
```

Rotate Shooter

```
:Player          :Shooter

rotateShooter(direction)
  →│
   │ updateAngle()
   │───────────────→│
```

Shoot Atom

Player → Shooter → Inventory

shoot()

Player → triggerShooter() → Shooter

Shooter ⇢ type → Player

Player → decreaseNumberOfAtoms(type) → Inventory

Shoot Powerup

Player → Shooter → Inventory

shoot()

Player → triggerShooter() → Shooter

Shooter ⇢ type → Player

Player → decreaseNumberOfPowerup(type) → Inventory

Move Shooter

Player → Shooter → Location

moveShooter(String direction)

Player → move(String direction) → Shooter

Shooter → updateLocation(int newX, int newY) → Location

UML Communication Diagrams

Blend Atoms

blendAtoms(typ1,typ2)

: Player

1: existing = checkInventory(typ1)

3 : **[existing = true]** decreaseAtomNumber(typ1, num)

4: **[existing = true]** increaseAtomNumber(typ2)

: Inventory

2: **[existing = true]** num = blendAtoms(typ1,typ2)

: Blender

Break Atom

breakAtom(typ1,typ2)

: Player

1 : existing = checkInventory(typ1)

3 **[existing=true]** : decreaseAtomNumber(typ1)

4 **[existing=true]** : increaseAtomNumber(typ2,num)

: Inventory

2 **[existing=true]**: num = breakAtom(typ1,typ2)

: Blender

Catch Powerup

catchPowerup()

:Player

1*[i=1..n]: isSame = checkPosition()
2: type = catchPowerup()

:Shooter

3: increaseNumberOfPowerup(type)

:Inventory

Pause Game

pause()

:KUVidGame

1 [not paused]: paused=true

Resume Game

resume()

:KUVidGame

1 [paused]: paused=false

Pick Powerup

pickPowerup(type)

:Player

1: existing = checkPowerup(type)

:Inventory

2 [existing=true]: placePowerup(type)

:Shooter

Pick Atom

pickAtom

:Player

1: random=randomAtom

:Inventory

2: placeAtom(random)

:Shooter

Rotate Shooter

rotateShooter(direction)

:Player

:Shooter

1: updateAngle

Shoot Atom

shoot()

: Player

1 : type = triggerShooter()

: Shooter

2 : decreaseNumberOfAtoms(type)

: Inventory

Move Shooter

moveShooter(String direction)

:Player

1 : move (String direction)

:Shooter

2 : updateLocation(int newX, int newY)

:Location

Shoot Powerup

shoot()

:Player

1: type = triggerShooter()

:Shooter

2: decreaseNumberOfPowerup(type)

:Inventory

Player
Inventory inventory
Shooter shooter
Blender blender
int healthPoints
—
catchPowerup()
shoot()
pickPowerup()
blend()
break()
triggerShooter()

GameSettings
+List<Integer> numberOfAtoms
+List<Integer> numberOfMolecules
+List<Integer> numberOfReactionBlockers
+List<Integer> numberOfPowerups
+int unitLength
+int level
+boolean isStationaryAlpha
+boolean isStationaryBeta
—
setAtomNumbers(List<Integer> atom)
setMoleculeNumbers(List<Integer> atom)
setPowerupNumbers(List<Integer> atom)
setReactionBlockerNumbers(List<Integer> atom)
setLevel(int level)
setUnitLength(int unitLength)

Shooter
ThrowableObject projectile
Location location
int speed

Blender
Atom source
Atom target
—
blend()
break()

Inventory
List<Integer> numberOfAtoms
List<Integer> numberOfMolecules
List<Integer> numberOfReactionBlockers
List<Integer> numberOfPowerups
—
increaseNumberOfPowerups(type)
decreaseNumberOfPowerups(type)
increaseNumberOfAtoms(type)
decreaseNumberOfAtoms(type)
checkInventory(type)

ThrowableObject (Interface)

KUVidGame
shoot()
pickAtom()
pickPowerup()
blend()
break()

Atom (Abstract)
String type
int speed
int width
int height
Location location
—
disappear()
move()
reflect()

Powerup (Abstract)
String type
int speed
int width
int height
Location location
—
fall()}

Statistics
int playerHealth
float score
int timeLeft
Map<Atom, Integer> availableAtoms
Map<Atom, Integer> availablePowerups
—
getHealth()
getScore()
getTimeLeft()
setHealth()
setScore()
setTimeLeft()

AlphaAtom
BetaAtom
GammaAtom
SigmaAtom

ReactionBlocker (Abstract)
String type
int speed
int width
int height
Location location
—
fall()
blockReaction()
explode()

Molecule (Abstract)
String type
int speed
int width
int height
Location location
—
fall()

Location
float xCoordinate
float yCoordinate

AlphaMolecule (Abstract)
BetaMolecule (Abstract)
GammaMolecule
SigmaMolecule

LinearAlphaMolecule
NormalAlphaMolecule
LinearBetaMolecule
NormalBetaMolecule

Powerup (Abstract)
String type
int speed
int width
int height
Location location
—
fall()

ReactionBlocker (Abstract)
String type
int speed
int width
int height
Location location
—
fall()
blockReaction()
explode()

AlphaPowerup
BetaPowerup
GammaPowerup
SigmaPowerup

AlphaReactionBlocker
BetaReactionBlocker
GammaReactionBlocker
SigmaReactionBlocker

Relationships:
- Player controls Shooter (1)
- Player controls Blender (1)
- Player has Inventory (1)
- Player plays KUVidGame
- Shooter shoots Atom (many)
- Blender blends Atom (many)
- Blender breaks Atom (1)
- Inventory records Atom
- Inventory records Powerup
- GameSettings records KUVidGame
- KUVidGame records Statistics (1)
- Shooter is at Location
- Atom is at Location
- Powerup is at Location
- ReactionBlocker is at Location
- Molecule is at Location
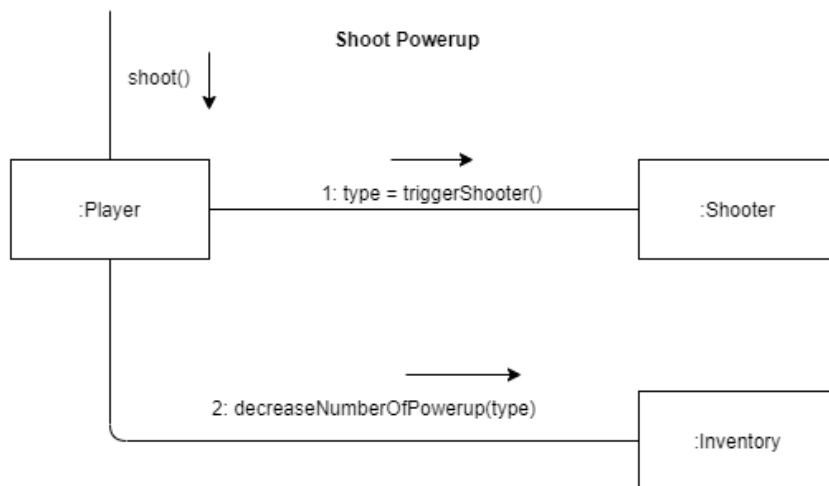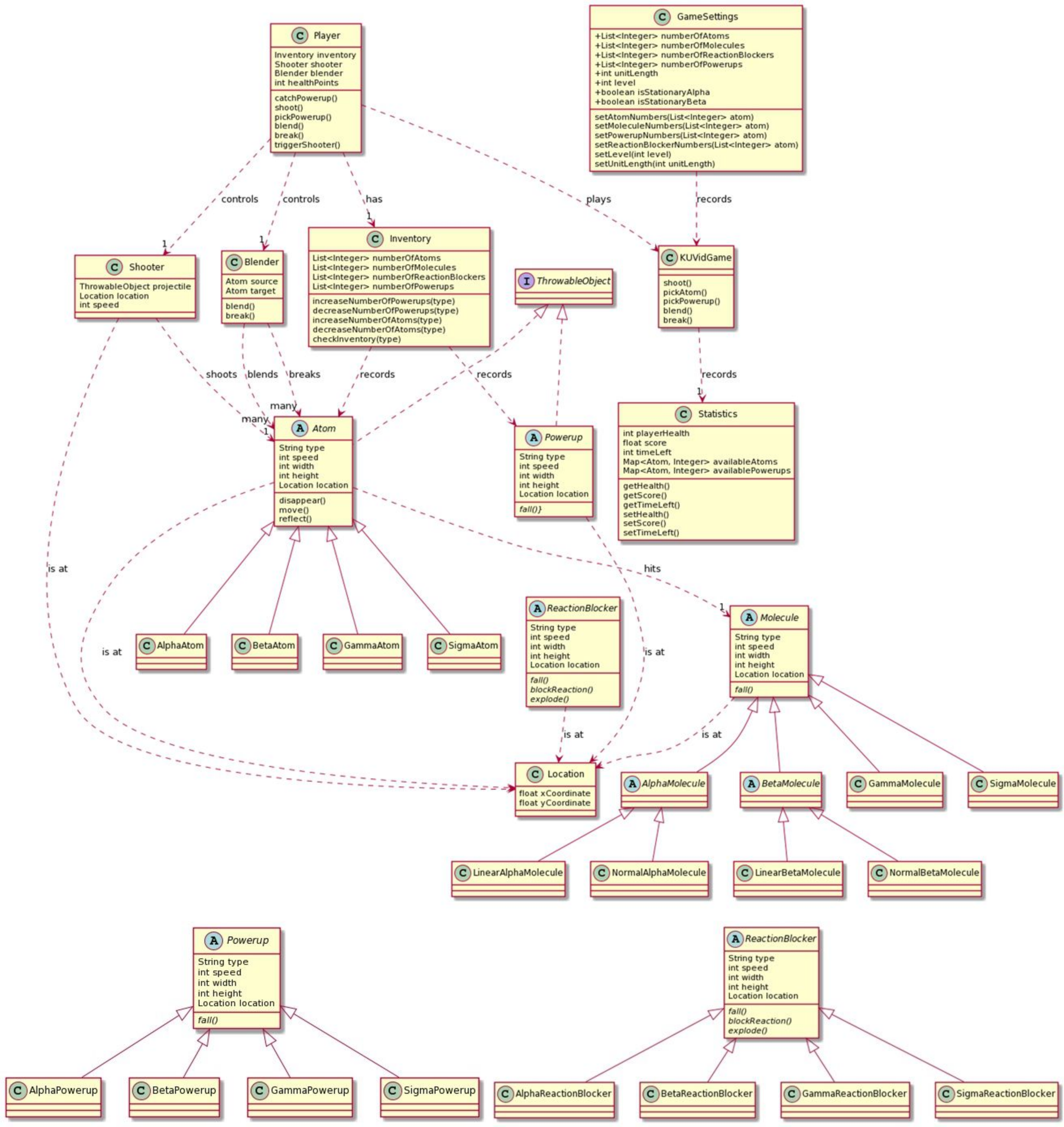- Powerup hits Molecule (1)

Patterns

Controller: In our design, we will be using the controller pattern to receive user inputs from the UI layer. The pattern allows us to delegate the inputs from the UI to the various classes in the domain layer without violating Model-View Separation. It represents the overall system. It is the main pattern that allows UI layer access to the domain layer, and therefore becomes the best option for controlling user inputs. So, we choose our Controller to be the KUVid Game class.

Creator: Our project requires instances of many types of objects to be constantly created, including Molecules, Powerups, Reaction Blockers. For this purpose, we will be implementing a FallingObjectCreator which will follow the GRASP pattern Creator. We decided to use a Creator pattern because FallingObjectCreator will have the initializing data for our falling objects when they are created. Thus making FallingObjectCreator an Expert with respect to creating falling objects. The use of a Creator pattern also supports Low Coupling.

Information Expert: In our design, to further reduce coupling, we will be using the Inventory class as an Information Expert. The Inventory class will be an expert in the sense that it will be the only class in our design that knows the amounts of the various objects that the player of the game possesses, e.g. number of atoms, number of powerups, number of atoms. For instance, the Player class will not have access to the number of atoms in their inventory directly. Another use of the Expert pattern will be the Blender class, which will have the information (rules) determining the relationship between various types of atoms while breaking and blending them. The distribution of responsibility between different classes also increases cohesion, while also making the code easier to follow. On the other hand, the pattern may result in designs that are problematic regarding cohesion, as can be seen in some of our sequence diagrams where the Player class calls the Shooter class and then calls the Inventory class with the return value.

High Cohesion: The best depiction of High Cohesion in our design is most likely the KUVidGame object, which instead of knowing what to do for all the possible actions for the game, delegates most of the commands, e.g. shoot, pick blender, etc., to different classes that are specifically designed to handle those methods; which enables us to maintain and reuse our code more frequently.

Low Coupling: The Player class does not shoot the atom directly, instead it triggers the Shooter to shoot what is currently loaded on it. If the Player class directly shoots an atom, it would be a high coupling. A similar example can also be given for **Indirection**, as rather than having the Player directly know/manipulate the number of items in the inventory, we have the inventory to do these tasks. Also, it enables reusing and understanding the code better without looking at other classes (just by looking at said class).

Polymorphism: Due to the nature of the project, where we have types (alpha, beta, sigma, gamma) of the same type of game objects (atom, molecule, powerup). The best way to handle this is by making use of the Polymorphism pattern. The main benefit of polymorphism is defining common elements and methods of objects in an interface or abstract class, therefore allowing us to write the bulk of the code once.

Strategy: In our design, the Strategy pattern is utilized in handling various falling patterns of some objects like Molecule, Powerup, ReactionBlocker. There are different strategies for falling objects. Also, the falling speed of some objects may change according to the difficulty of the game. While it may appear to make the design of the code more complicated, the Strategy pattern allows any change that may occur in any current or later stage of development to be carried out in a much easier fashion.

Observer: The Observer pattern is a must use pattern in most UI dependent projects, since it is the only relaxation of the most important principle, Model-View Separation. In our design, the use of the Observer pattern will be to update locations of the various game objects on the UI side, *according to the locations* which will be defined on the domain layer.