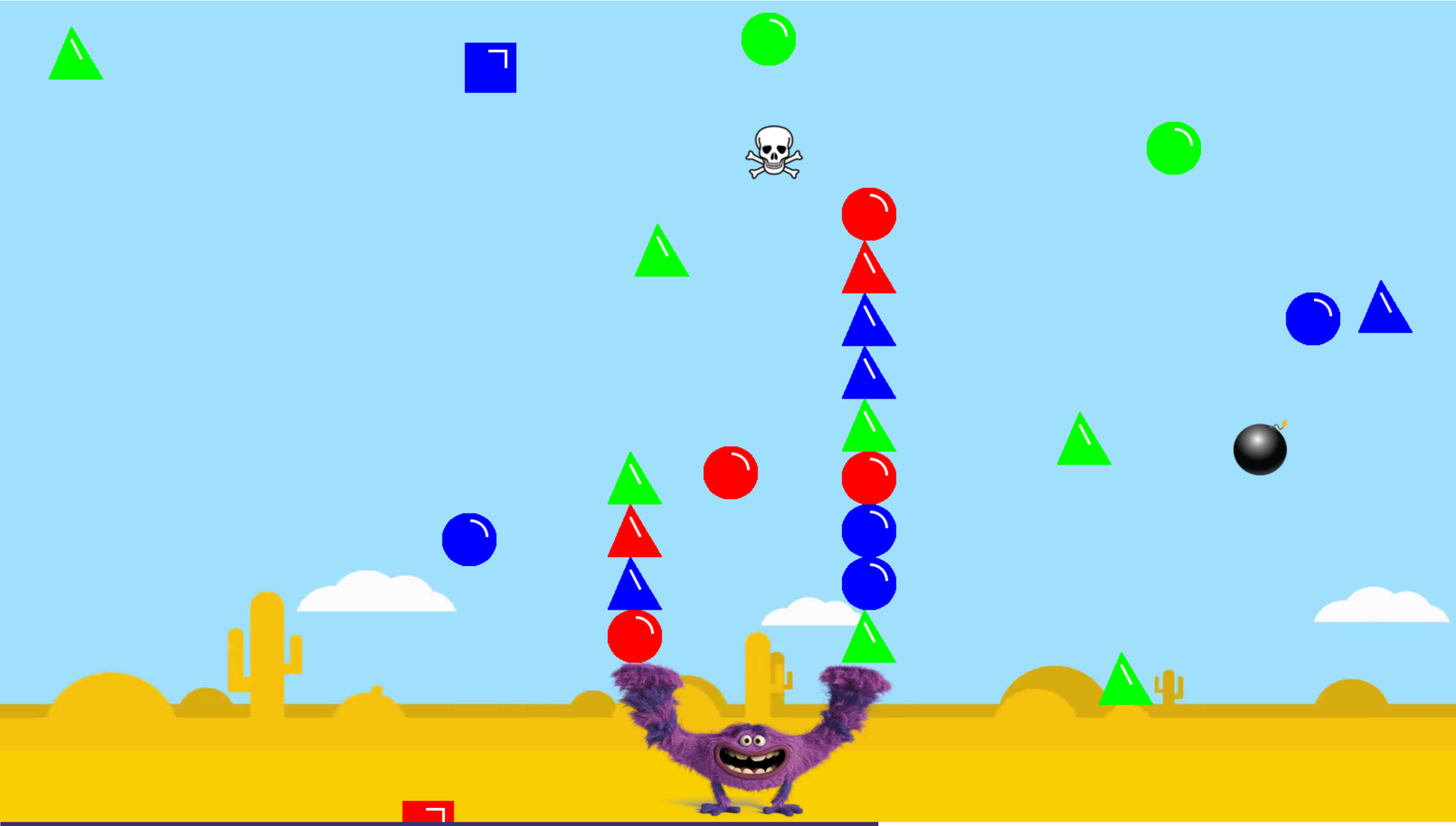


Circus of Plates OOP

Team members

Merna Mustafa [ID 54]	Febronia Ashraf [ID 29]
Neveen Samir [ID 58]	Menna Osman [ID 52]

TABLE OF CONTENTS

GAME DESCRIPTION:

It is a single player-game in which the player carries two stacks of shapes , and there are a set of colored shapes that fall and he tries to catch them. If he manages to collect three consecutive shapes of the same color , then they are vanished and his score increases by one point and the game time increases by 5 seconds.

There are three levels of difficulty :

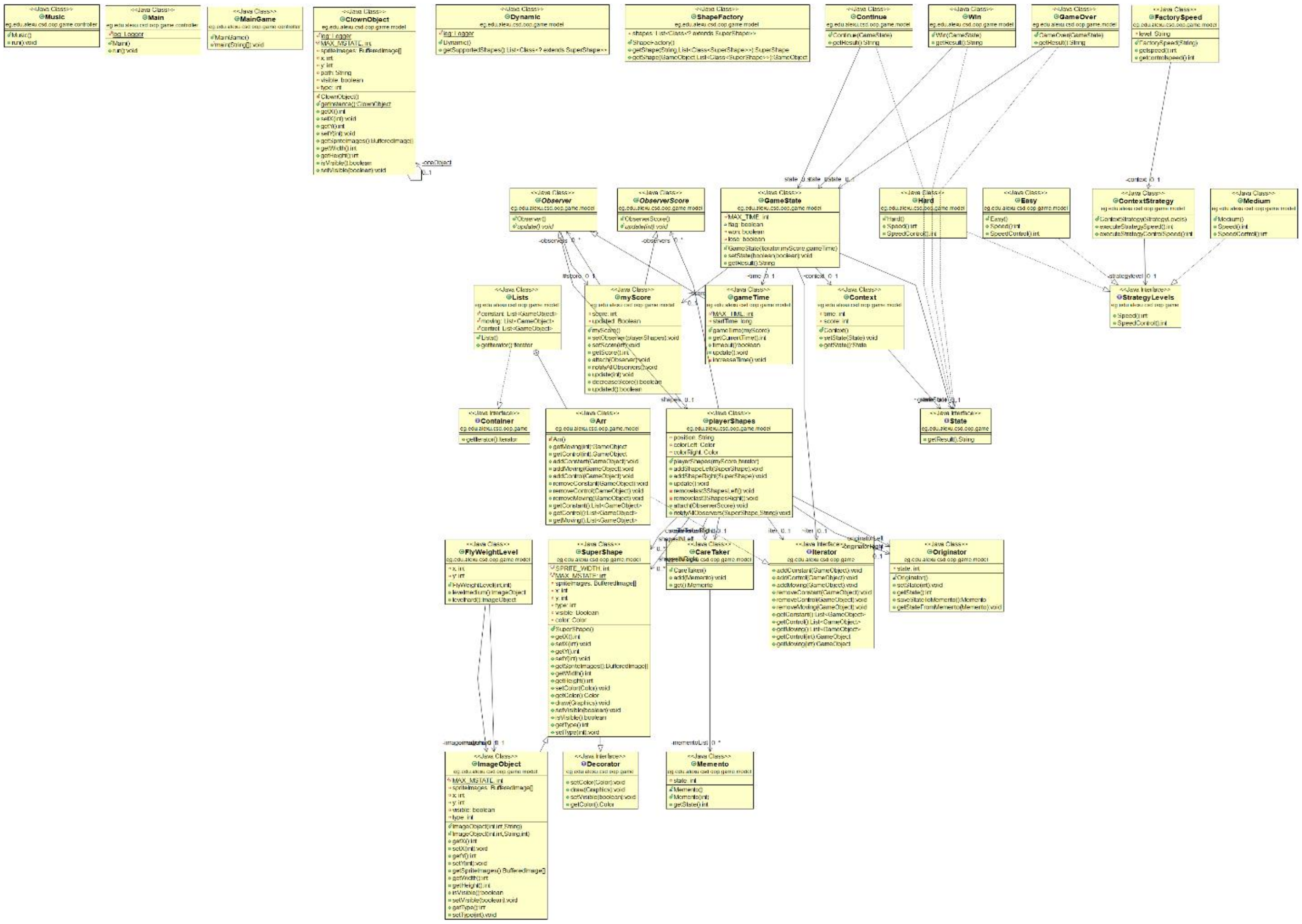
Easy : the shapes' speed is low .

Medium : high speed , there is a new shape "bomb" if the player hits it , the score will decrease.

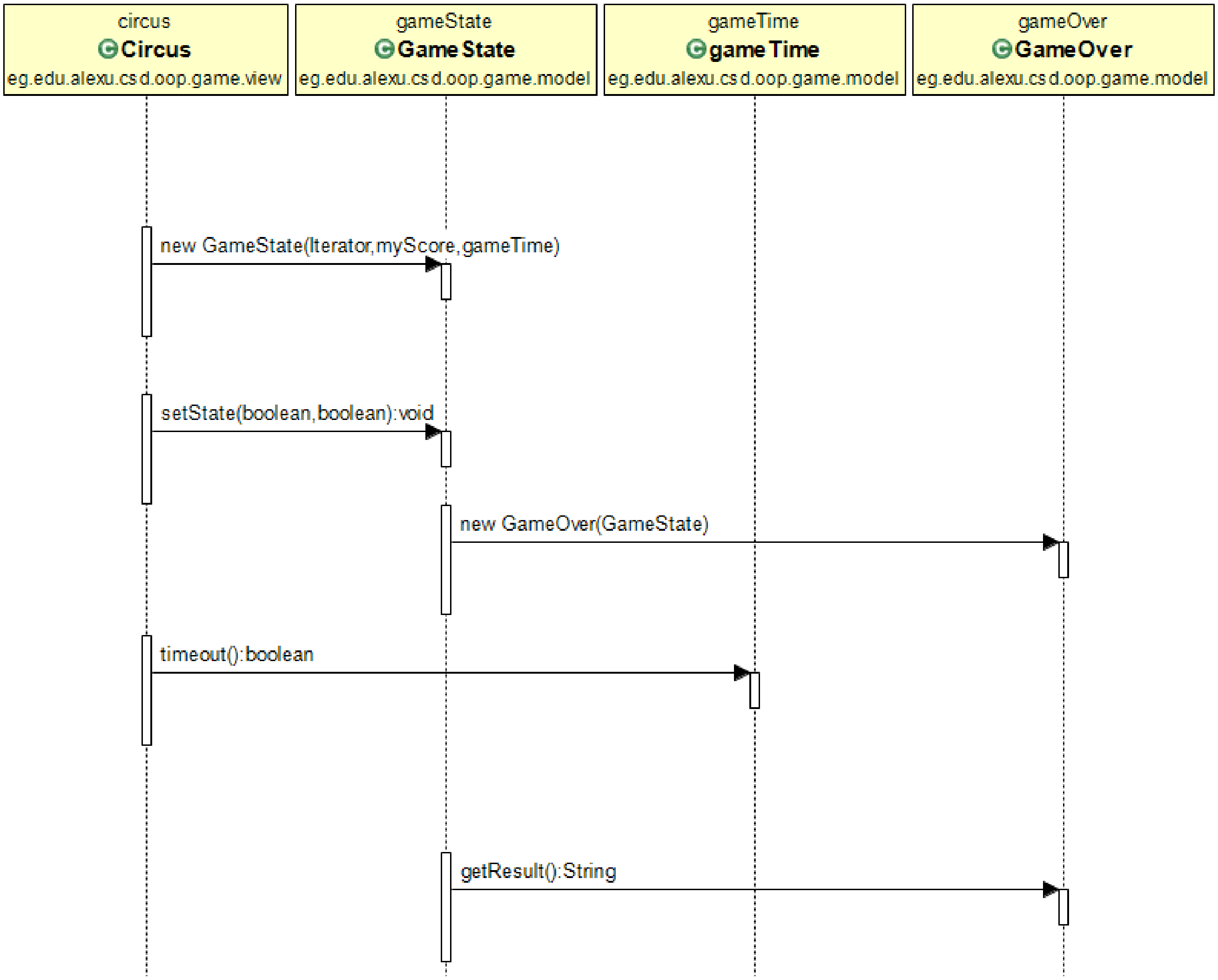
Hard : high speed , another new shape will appear "skull" .

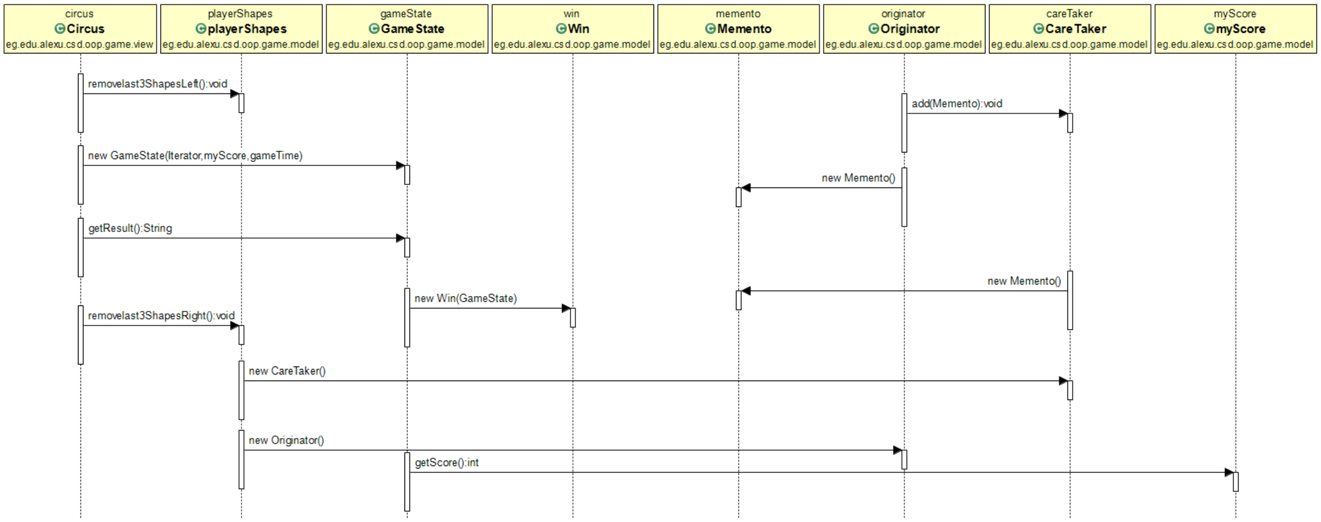
The player wins if his score became 5, and lose when the time ended or hits the danger shapes (bomb, skull) when his score is 0 or one of the two stacks is full .

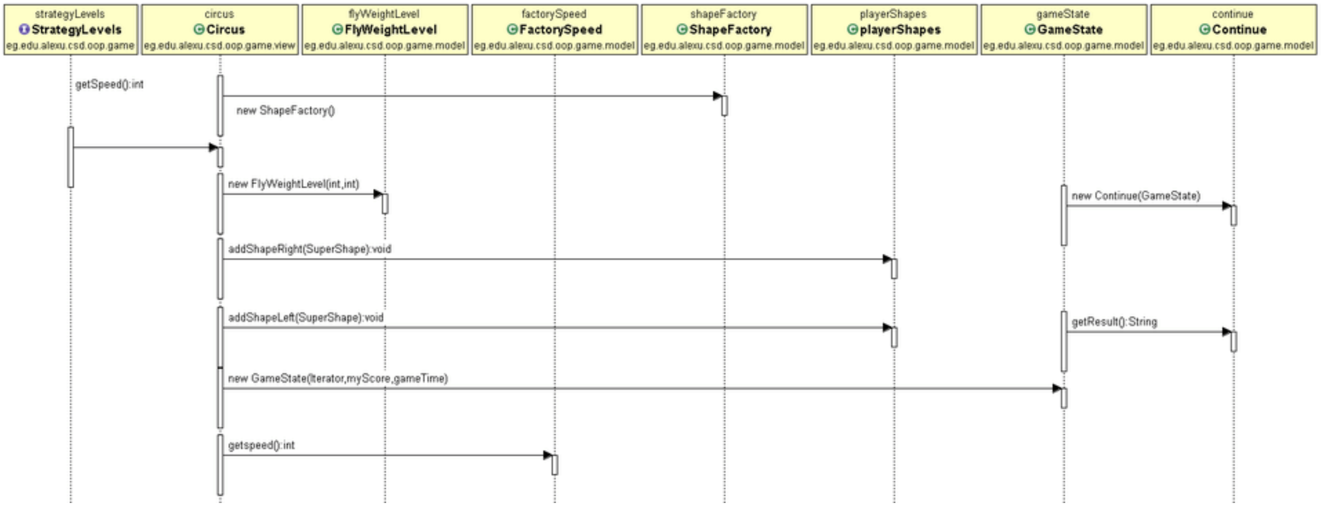
GAME DESIGN:

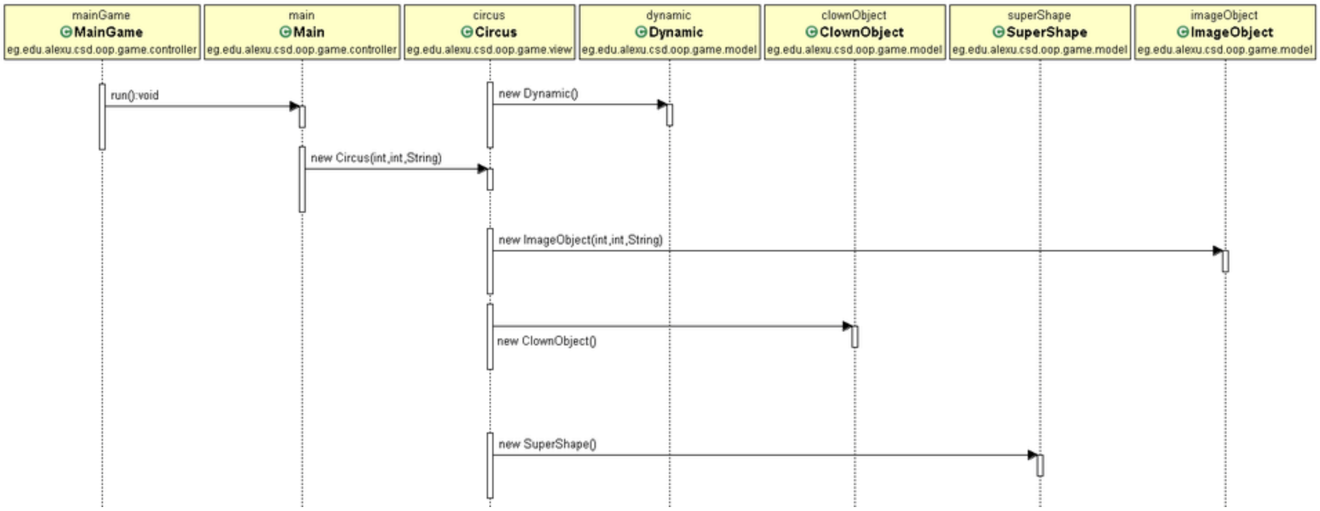

<<Java Class>>
Music
eg.edu.alexu.csd.oop.game.controller
Music()
run():void
<<Java Class>>
Main
eg.edu.alexu.csd.oop.game.controller
log: Logger
Main()
run():void
<<Java Class>>
MainGame
eg.edu.alexu.csd.oop.game.controller
MainGame()
main(String[]):void
<<Java Class>>
ClownObject
eg.edu.alexu.csd.oop.game.model
log: Logger
MAX_MSTATE: int
spriteImages: BufferedImage[]
x: int
y: int
path: String
visible: boolean
type: int
ClownObject()
getInstance():ClownObject
getX():int
setX(int):void
getY():int
setY(int):void
getSpriteImages():BufferedImage[]
getWidth():int
getHeight():int
isVisible():boolean
setVisible(boolean):void
-oneObject
0..1
<<Java Class>>
Dynamic
eg.edu.alexu.csd.oop.game.model
log: Logger
Dynamic()
getSupportedShapes():List<Class<? extends SuperShape>>
<<Java Class>>
ShapeFactory
eg.edu.alexu.csd.oop.game.model
shapes: List<Class<? extends SuperShape>>
ShapeFactory()
getShape(String,List<Class<SuperShape>>):SuperShape
getShape(GameObject,List<Class<SuperShape>>):GameObject
<<Java Class>>
Continue
eg.edu.alexu.csd.oop.game.model
Continue(GameState)
getResult():String
<<Java Class>>
Win
eg.edu.alexu.csd.oop.game.model
Win(GameState)
getResult():String
<<Java Class>>
GameOver
eg.edu.alexu.csd.oop.game.model
GameOver(GameState)
getResult():String
<<Java Class>>
FactorySpeed
eg.edu.alexu.csd.oop.game.model
level: String
FactorySpeed(String)
getspeed():int
getcontrolspeed():int
state 0..state 0..state 0..1
<<Java Class>>
Observer
eg.edu.alexu.csd.oop.game.model
Observer()
update():void
<<Java Class>>
ObserverScore
eg.edu.alexu.csd.oop.game.model
ObserverScore()
update(int):void
<<Java Class>>
GameState
eg.edu.alexu.csd.oop.game.model
MAX_TIME: int
flag: boolean
won: boolean
lose: boolean
GameState(Iterator,myScore,gameTime)
setState(boolean,boolean):void
getResult():String
<<Java Class>>
Hard
eg.edu.alexu.csd.oop.game.model
Hard()
Speed():int
SpeedControl():int
<<Java Class>>
Easy
eg.edu.alexu.csd.oop.game.model
Easy()
Speed():int
SpeedControl():int
-context 0..1
<<Java Class>>
ContextStrategy
eg.edu.alexu.csd.oop.game.model
ContextStrategy(StrategyLevels)
executeStrategySpeed():int
executeStrategyControlSpeed():int
<<Java Class>>
Medium
eg.edu.alexu.csd.oop.game.model
Medium()
Speed():int
SpeedControl():int
-observers 0..*
-observers 0..*
#score 0..1
-score 0..1
-time 0..1
-context 0..1
-strategylevel 0..1
<<Java Class>>
Lists
eg.edu.alexu.csd.oop.game.model
constant: List<GameObject>
moving: List<GameObject>
control: List<GameObject>
Lists()
getIterator():Iterator
<<Java Class>>
myScore
eg.edu.alexu.csd.oop.game.model
score: int
updated: Boolean
myScore()
setObserver(playerShapes):void
setScore(int):void
getScore():int
attach(Observer):void
notifyAllObservers():void
update(int):void
decreaseScore():boolean
updated():boolean
<<Java Class>>
gameTime
eg.edu.alexu.csd.oop.game.model
MAX_TIME: int
startTime: long
gameTime(myScore)
getCurrentTime():int
timeout():boolean
update():void
increaseTime():void
<<Java Class>>
Context
eg.edu.alexu.csd.oop.game.model
time: int
score: int
Context()
setState(State):void
getState():State
<<Java Interface>>
StrategyLevels
eg.edu.alexu.csd.oop.game
Speed():int
SpeedControl():int
shapes 0..1
-gameState 0..1
<<Java Interface>>
Container
eg.edu.alexu.csd.oop.game
getIterator():Iterator
<<Java Class>>
Arr
eg.edu.alexu.csd.oop.game.model
Arr()
getMoving(int):GameObject
getControl(int):GameObject
addConstant(GameObject):void
addMoving(GameObject):void
addControl(GameObject):void
removeConstant(GameObject):void
removeControl(GameObject):void
removeMoving(GameObject):void
getConstant():List<GameObject>
getControl():List<GameObject>
getMoving():List<GameObject>
<<Java Class>>
playerShapes
eg.edu.alexu.csd.oop.game.model
position: String
colorLeft: Color
colorRight: Color
playerShapes(myScore,Iterator)
addShapeLeft(SuperShape):void
addShapeRight(SuperShape):void
update():void
removelast3ShapesLeft():void
removelast3ShapesRight():void
attach(ObserverScore):void
notifyAllObservers(SuperShape,String):void
<<Java Interface>>
State
eg.edu.alexu.csd.oop.game
getResult():String
careTakerRight 0..1
shapesInLeft
shapesInRight
0..*
0..*
-iter 0..1
-iter 0..1
originatorLeft
originatorRight
0..1
<<Java Class>>
FlyWeightLevel
eg.edu.alexu.csd.oop.game.model
x: int
y: int
FlyWeightLevel(int,int)
levelmedium():ImageObject
levelhard():ImageObject
<<Java Class>>
SuperShape
eg.edu.alexu.csd.oop.game.model
SPRITE_WIDTH: int
MAX_MSTATE: int
spriteImages: BufferedImage[]
x: int
y: int
type: int
visible: Boolean
color: Color
SuperShape()
getX():int
setX(int):void
getY():int
setY(int):void
getSpriteImages():BufferedImage[]
getWidth():int
getHeight():int
setColor(Color):void
getColor():Color
draw(Graphics):void
setVisible(boolean):void
isVisible():boolean
getType():int
setType(int):void
<<Java Class>>
CareTaker
eg.edu.alexu.csd.oop.game.model
CareTaker()
add(Memento):void
get():Memento
<<Java Interface>>
Iterator
eg.edu.alexu.csd.oop.game
addConstant(GameObject):void
addControl(GameObject):void
addMoving(GameObject):void
removeConstant(GameObject):void
removeControl(GameObject):void
removeMoving(GameObject):void
getConstant():List<GameObject>
getControl():List<GameObject>
getMoving():List<GameObject>
getControl(int):GameObject
getMoving(int):GameObject
<<Java Class>>
Originator
eg.edu.alexu.csd.oop.game.model
state: int
Originator()
setState(int):void
getState():int
saveStateToMemento():Memento
getStateFromMemento(Memento):void
-imagemedium 0..1
-imagehard 0..1
-mementoList 0..*
<<Java Class>>
ImageObject
eg.edu.alexu.csd.oop.game.model
MAX_MSTATE: int
spriteImages: BufferedImage[]
x: int
y: int
visible: boolean
type: int
ImageObject(int,int,String)
ImageObject(int,int,String,int)
getX():int
setX(int):void
getY():int
setY(int):void
getSpriteImages():BufferedImage[]
getWidth():int
getHeight():int
isVisible():boolean
setVisible(boolean):void
getType():int
setType(int):void
<<Java Interface>>
Decorator
eg.edu.alexu.csd.oop.game
setColor(Color):void
draw(Graphics):void
setVisible(boolean):void
getColor():Color
<<Java Class>>
Memento
eg.edu.alexu.csd.oop.game.model
state: int
Memento()
Memento(int)
getState():int

SEQUENCE DIAGRAMS:


circus
Circus
eg.edu.alexu.csd.oop.game.view
gameState
Game State
eg.edu.alexu.csd.oop.game.model
gameTime
game Time
eg.edu.alexu.csd.oop.game.model
gameOver
Game Over
eg.edu.alexu.csd.oop.game.model
new GameState(Iterator,myScore,gameTime)
setState(boolean,boolean):void
new GameOver(GameState)
timeout():boolean
getResult():String


circus
Circus
eg.edu.alexu.csd.oop.game.view
playerShapes
playerShapes
eg.edu.alexu.csd.oop.game.model
gameState
Game State
eg.edu.alexu.csd.oop.game.model
win
Win
eg.edu.alexu.csd.oop.game.model
memento
Memento
eg.edu.alexu.csd.oop.game.model
originator
Originator
eg.edu.alexu.csd.oop.game.model
careTaker
Care Taker
eg.edu.alexu.csd.oop.game.model
myScore
myScore
eg.edu.alexu.csd.oop.game.model
removelast3ShapesLeft():void
add(Memento):void
new GameState(Iterator,myScore,gameTime)
new Memento()
getResult():String
new Memento()
new Win(GameState)
removelast3ShapesRight():void
new CareTaker()
new Originator()
getScore():int


strategyLevels
StrategyLevels
eg.edu.alexu.csd.oop.game
circus
Circus
eg.edu.alexu.csd.oop.game.view
flyWeightLevel
FlyWeightLevel
eg.edu.alexu.csd.oop.game.model
factorySpeed
FactorySpeed
eg.edu.alexu.csd.oop.game.model
shapeFactory
ShapeFactory
eg.edu.alexu.csd.oop.game.model
playerShapes
playerShapes
eg.edu.alexu.csd.oop.game.model
gameState
GameState
eg.edu.alexu.csd.oop.game.model
continue
Continue
eg.edu.alexu.csd.oop.game.model
getSpeed():int
new ShapeFactory()
new FlyWeightLevel(int,int)
new Continue(GameState)
addShapeRight(SuperShape):void
addShapeLeft(SuperShape):void
getResult():String
new GameState(Iterator,myScore,gameTime)
getspeed():int


mainGame
MainGame
eg.edu.alexu.csd.oop.game.controller
main
Main
eg.edu.alexu.csd.oop.game.controller
circus
Circus
eg.edu.alexu.csd.oop.game.view
dynamic
Dynamic
eg.edu.alexu.csd.oop.game.model
clownObject
ClownObject
eg.edu.alexu.csd.oop.game.model
superShape
SuperShape
eg.edu.alexu.csd.oop.game.model
imageObject
ImageObject
eg.edu.alexu.csd.oop.game.model
run():void
new Dynamic()
new Circus(int,int,String)
new ImageObject(int,int,String)
new ClownObject()
new SuperShape()

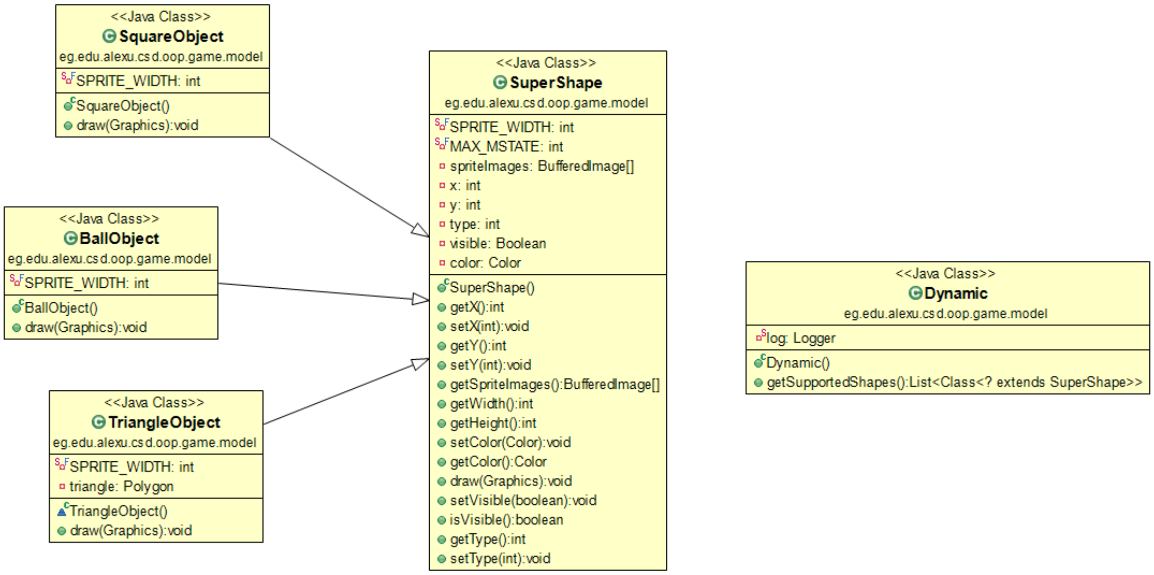
DESIGN PATTERNS:

1) Singleton

This pattern involves a single class which is responsible to create a player object while making sure that only single player gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

2) Dynamic linkage

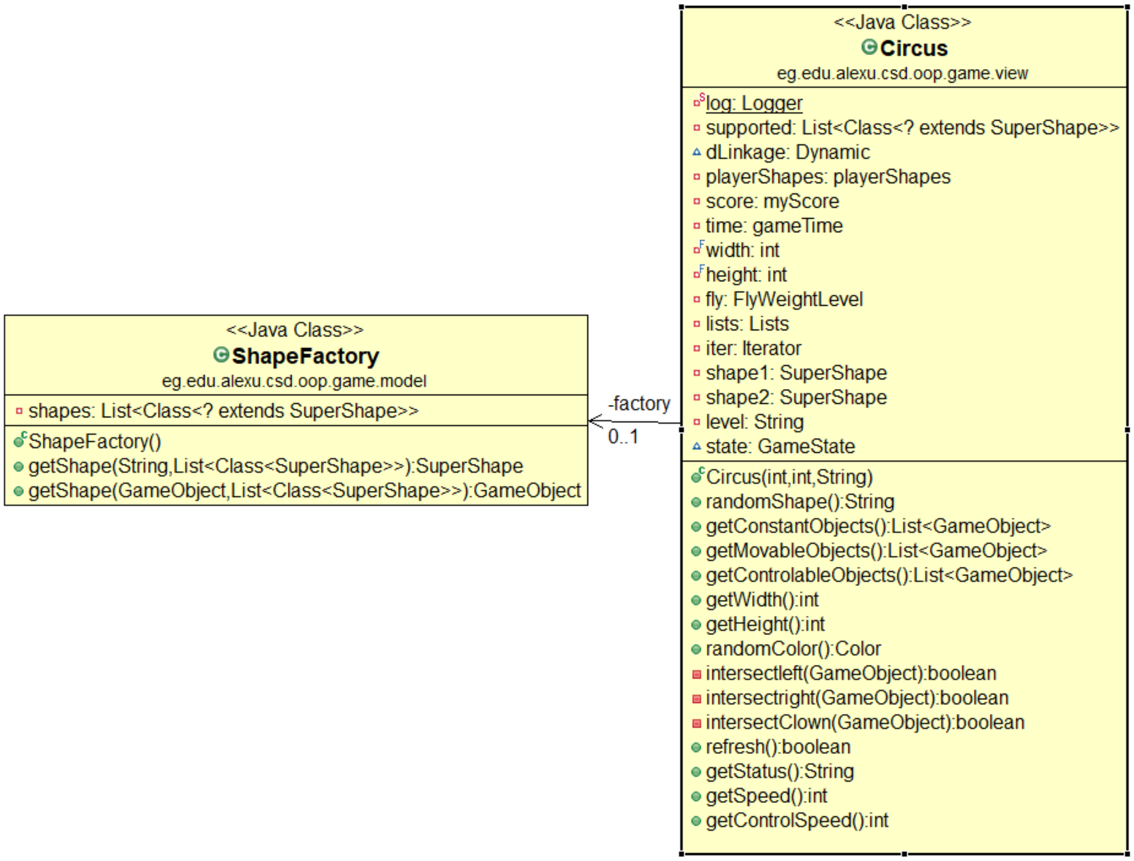
We used this pattern to load the shape classes from a jar file. this class return a list of shape classes (shapes).



3) Factory

In Factory pattern, we create shapes without exposing the creation logic to the client and refer to newly created object using a common interface.

We use it to create different shapes. The factory class takes the shapes list which loaded by dynamic linkage and create a new shape.


<<Java Class>>
Circus
eg.edu.alexu.csd.oop.game.view
log: Logger
supported: List<Class<? extends SuperShape>>
dLinkage: Dynamic
playerShapes: playerShapes
score: myScore
time: gameTime
width: int
height: int
fly: FlyWeightLevel
lists: Lists
iter: Iterator
shape1: SuperShape
shape2: SuperShape
level: String
state: GameState
Circus(int,int,String)
randomShape():String
getConstantObjects():List<GameObject>
getMovableObjects():List<GameObject>
getControlableObjects():List<GameObject>
getWidth():int
getHeight():int
randomColor():Color
intersectleft(GameObject):boolean
intersectright(GameObject):boolean
intersectClown(GameObject):boolean
refresh():boolean
getStatus():String
getSpeed():int
getControlSpeed():int
<<Java Class>>
ShapeFactory
eg.edu.alexu.csd.oop.game.model
shapes: List<Class<? extends SuperShape>>
ShapeFactory()
getShape(String,List<Class<SuperShape>>):SuperShape
getShape(GameObject,List<Class<SuperShape>>):GameObject
-factory
0..1

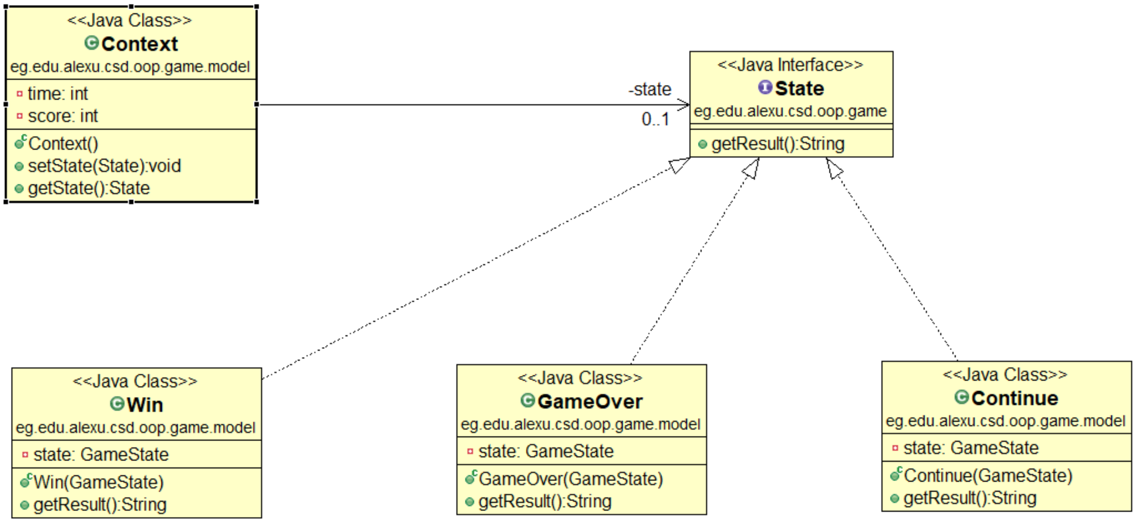
4) State

In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

We have 3 states : win state , game over state and continue state .

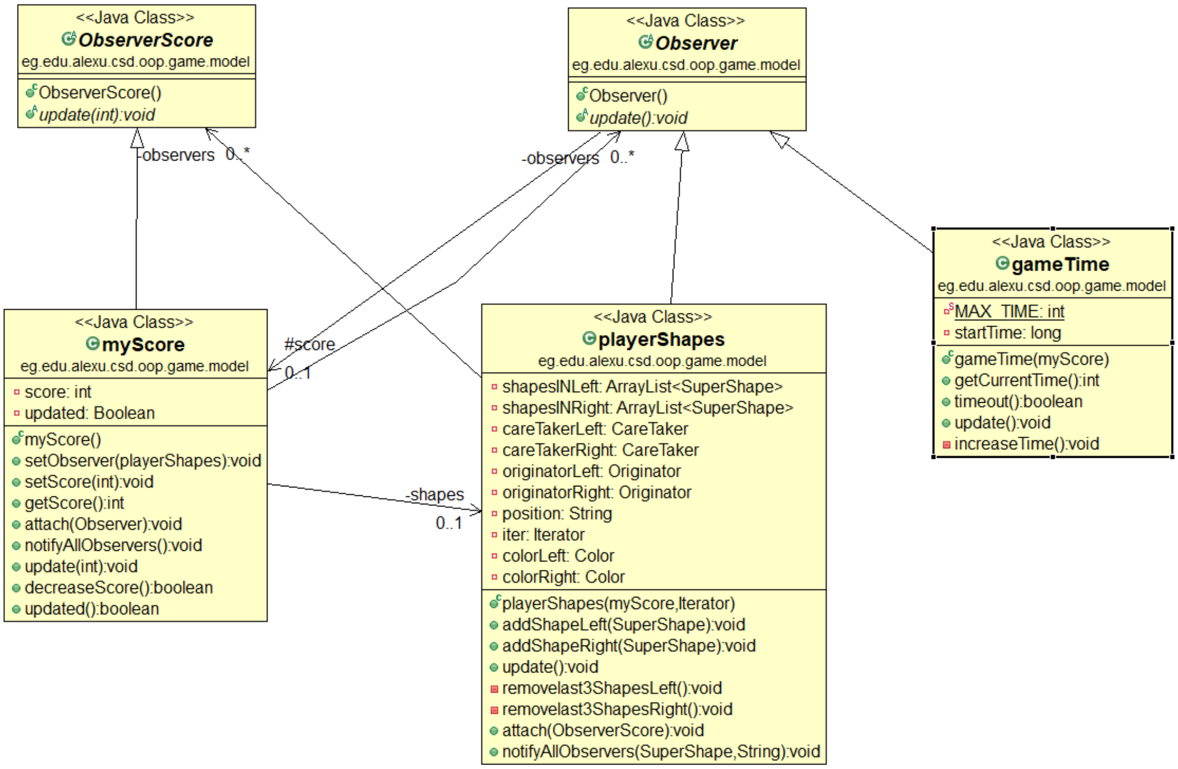
We use context and state objects to demonstrate change in Context behavior based on type of state it is in

States change according to the player’s score , time and danger shapes.



5) Observer

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. When the score increases , the shapes stack and the time will be modified automatically . the time will increase by 5 s and the last 3 shapes will vanish.

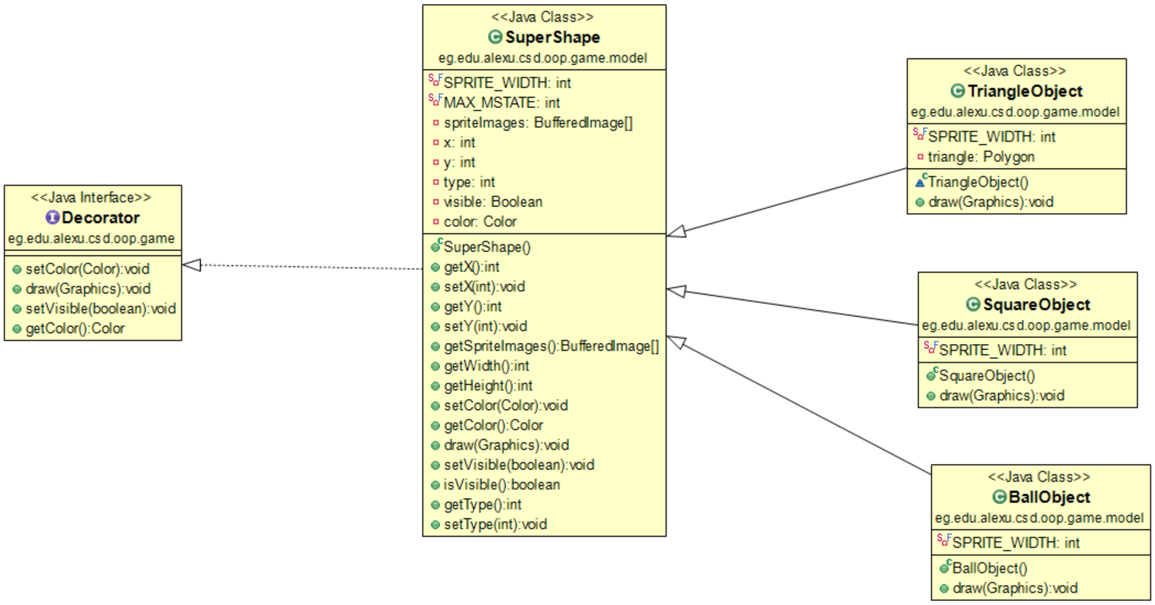


6) Decorator

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

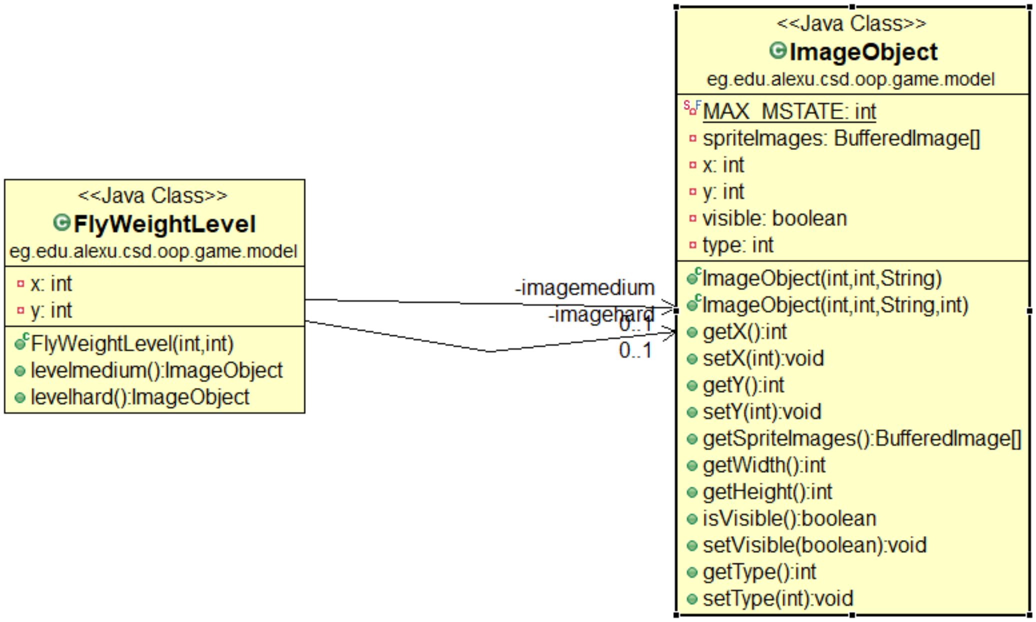
We create a *decorator* interface . We will then create an SuperShape implementing the decorator interface and having *Shape* object as its instance variable. We have three classes (ballObject, SquareObject, TriangleObject) concrete implementing SuperShape.



7) Flyweight

Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.

Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found. We will demonstrate this pattern to draw the danger shapes(bomb, skull).

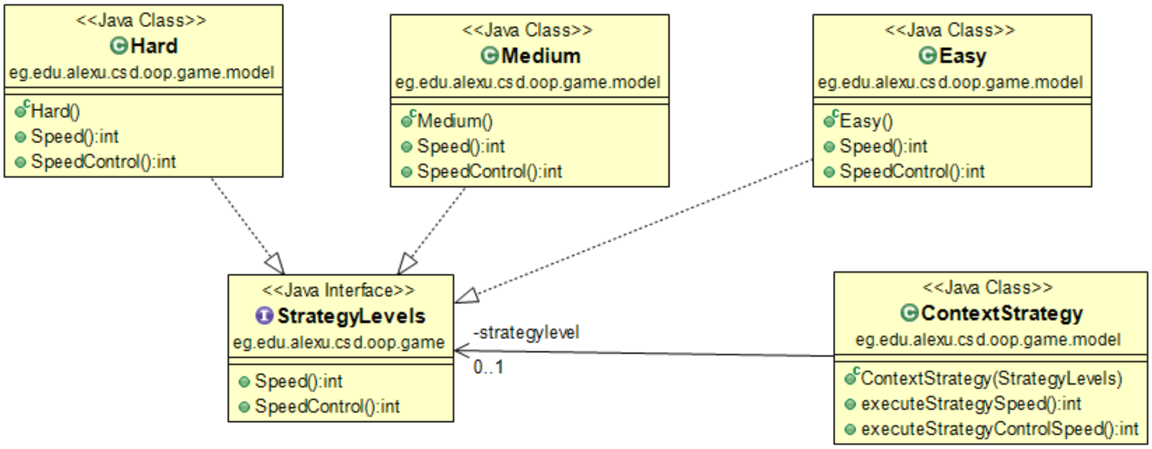


8) Strategy

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

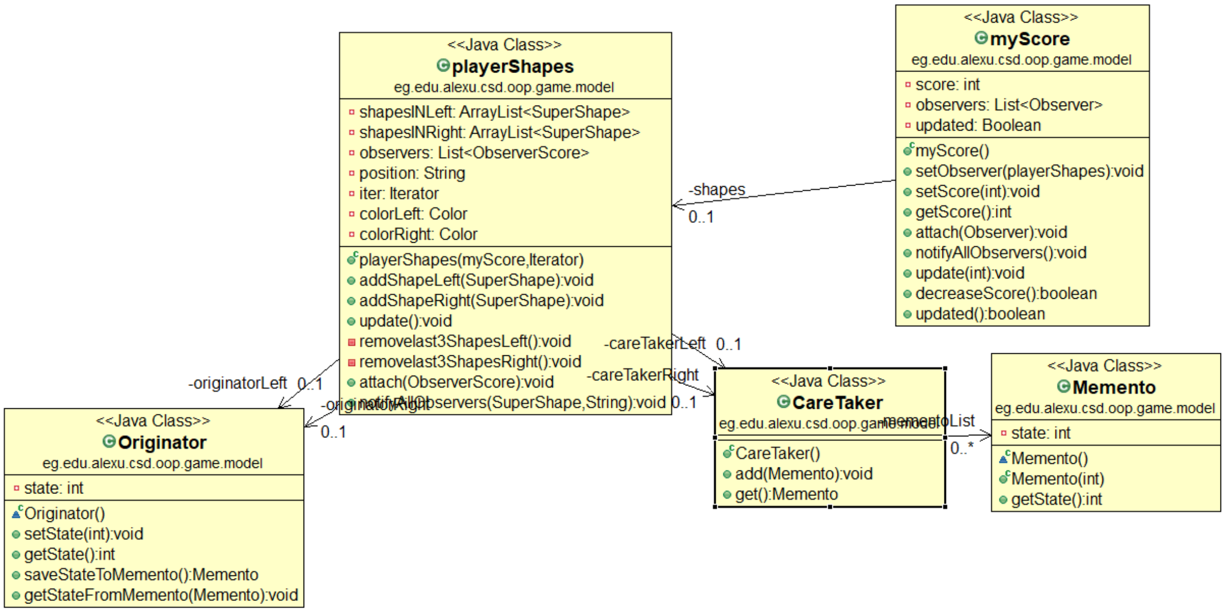
In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

We have three levels of difficulty implements *Strategy* interface defining an action. *Context* is a class which returns levels.



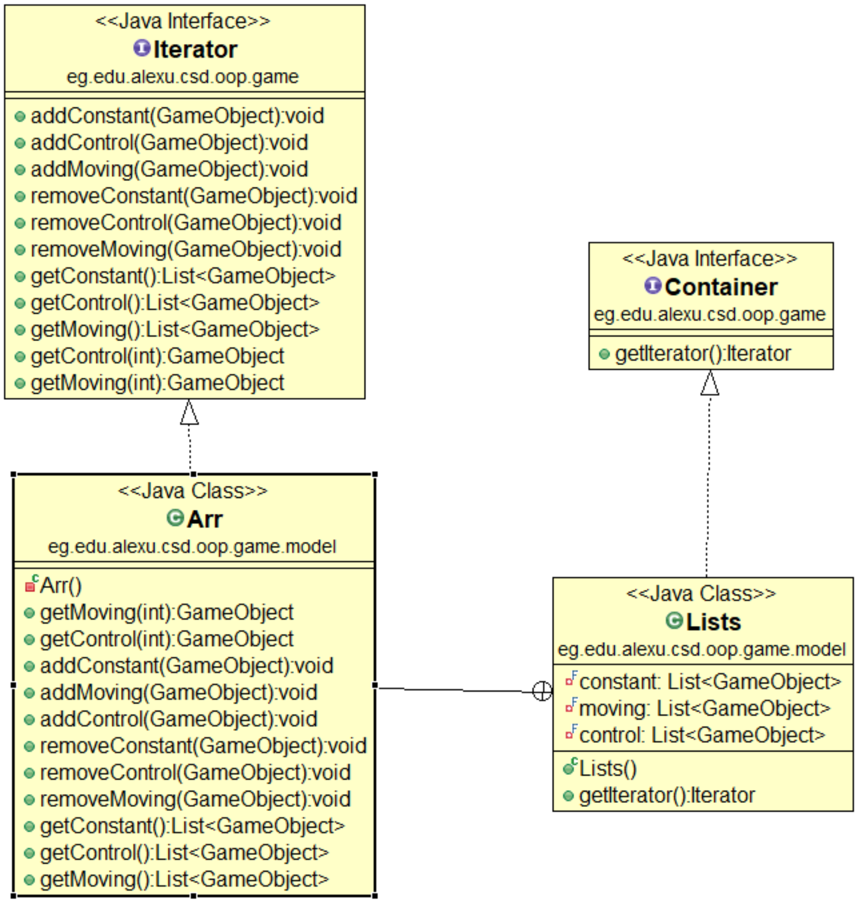
9) Memento

Memento pattern is used to restore state of an object to a previous state. Memento pattern uses three actor classes (CareTaker, Memento, Originator). When the player catch a shape the state becomes one ,then when player catches a new shape the state increases when the color of the new shape equals color of the old shape otherwise the state of new shape becomes one


<<Java Class>>
myScore
eg.edu.alexu.csd.oop.game.model
score: int
observers: List<Observer>
updated: Boolean
myScore()
setObserver(playerShapes):void
setScore(int):void
getScore():int
attach(Observer):void
notifyAllObservers():void
update(int):void
decreaseScore():boolean
updated():boolean
<<Java Class>>
playerShapes
eg.edu.alexu.csd.oop.game.model
shapesINLeft: ArrayList<SuperShape>
shapesINRight: ArrayList<SuperShape>
observers: List<ObserverScore>
position: String
iter: Iterator
colorLeft: Color
colorRight: Color
playerShapes(myScore,Iterator)
addShapeLeft(SuperShape):void
addShapeRight(SuperShape):void
update():void
removelast3ShapesLeft():void
removelast3ShapesRight():void
attach(ObserverScore):void
notifyAllObservers(SuperShape,String):void
-shapes
0..1
-careTakerLeft
0..1
-careTakerRight
0..1
-originatorLeft
0..1
-originatorRight
0..1
<<Java Class>>
Originator
eg.edu.alexu.csd.oop.game.model
state: int
Originator()
setState(int):void
getState():int
saveStateToMemento():Memento
getStateFromMemento(Memento):void
<<Java Class>>
CareTaker
eg.edu.alexu.csd.oop.game.model
CareTaker()
add(Memento):void
get():Memento
-mementoList
0..*
<<Java Class>>
Memento
eg.edu.alexu.csd.oop.game.model
state: int
Memento()
Memento(int)
getState():int

10) Iterator

We create an Iterator interface which narrates navigation methods and a Container interface which returns the iterator . Concrete classes implementing the Container interface is responsible to implement Iterator interface and use it. We use it to deal with our lists “ constant , moving , control”.


<<Java Interface>>
Iterator
eg.edu.alexu.csd.oop.game
addConstant(GameObject):void
addControl(GameObject):void
addMoving(GameObject):void
removeConstant(GameObject):void
removeControl(GameObject):void
removeMoving(GameObject):void
getConstant():List<GameObject>
getControl():List<GameObject>
getMoving():List<GameObject>
getControl(int):GameObject
getMoving(int):GameObject
<<Java Interface>>
Container
eg.edu.alexu.csd.oop.game
getIterator():Iterator
<<Java Class>>
Arr
eg.edu.alexu.csd.oop.game.model
Arr()
getMoving(int):GameObject
getControl(int):GameObject
addConstant(GameObject):void
addMoving(GameObject):void
addControl(GameObject):void
removeConstant(GameObject):void
removeControl(GameObject):void
removeMoving(GameObject):void
getConstant():List<GameObject>
getControl():List<GameObject>
getMoving():List<GameObject>
<<Java Class>>
Lists
eg.edu.alexu.csd.oop.game.model
constant: List<GameObject>
moving: List<GameObject>
control: List<GameObject>
Lists()
getIterator():Iterator

11)MVC

We use MVC architectural design pattern and divide our software into three interconnected parts Model - View - Controller. This allows us to work in parallel on different components without impacting or blocking one another and makes the modification easier.