Alexandria University
Faculty of Engineering
Computer and Systems Engineering Dept
CS221: Programming-2

CIRCUS OF PLATES

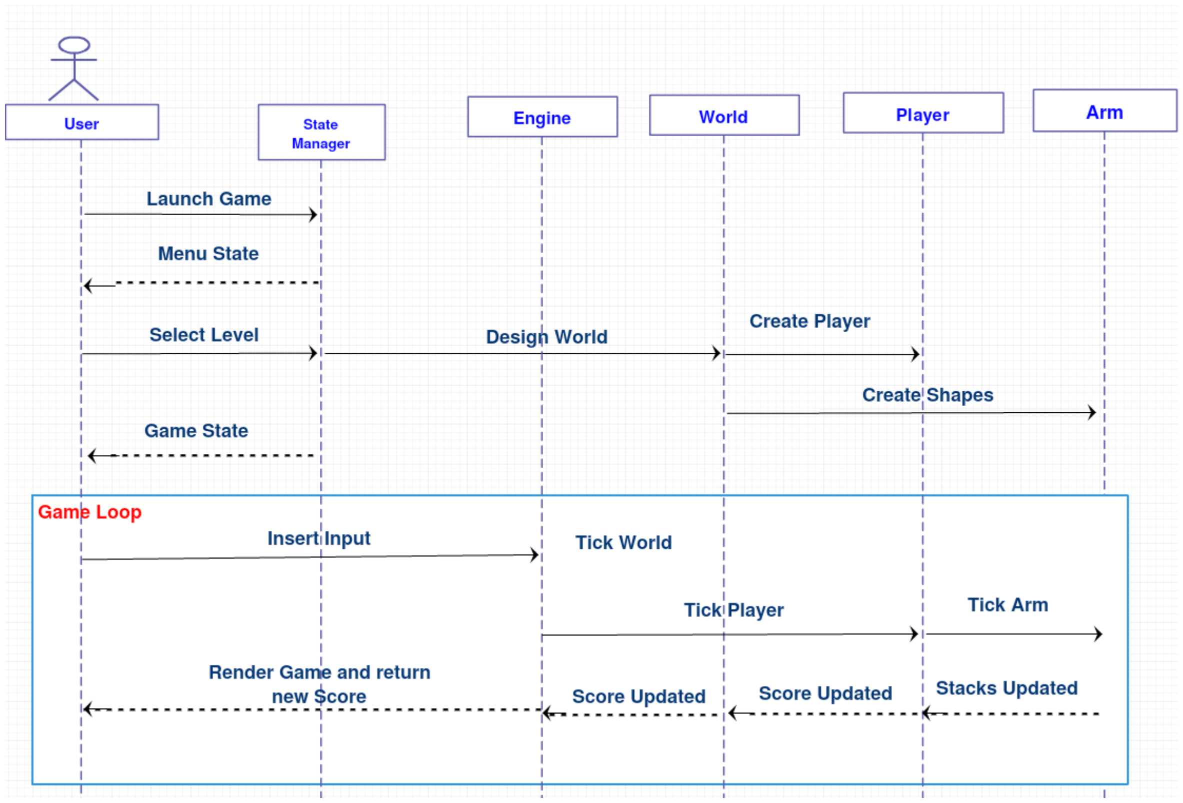
Names :

• Sherif Hassan Waly	(30)
• Omar Mohammed AbdelBaset	(42)
• Mohammed FathyAbdelFattah	(60)
• Marwan Alaa Mahmoud	(65)

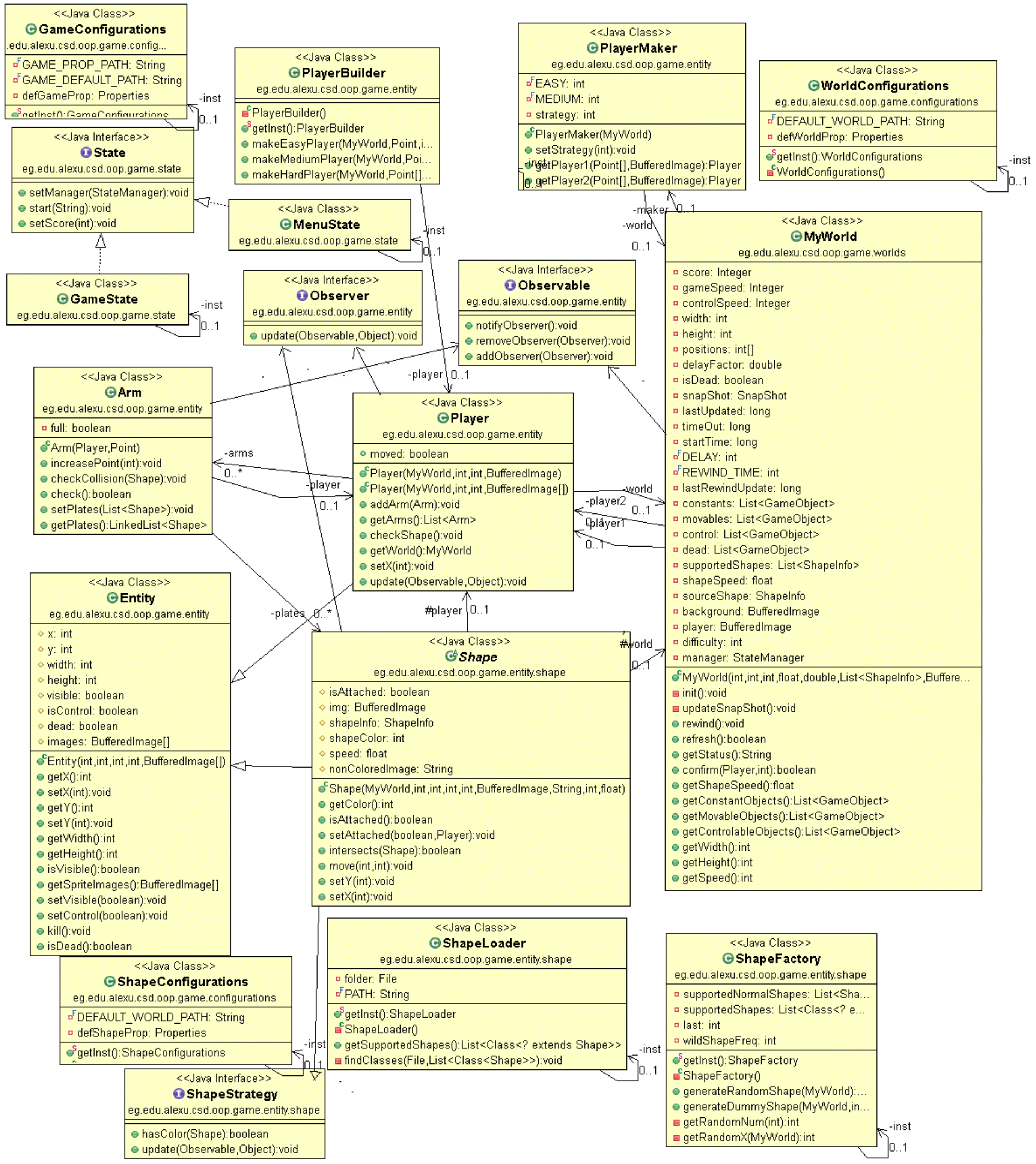
Game Description:

It is one player game in which every character carry one or more stack of plates, and there are a set of colored shapes queues that fall down and player tries to catch them, if player manages to collect three consecutive Shapes of the same color, then they will disappear and player's score increases. Game has three levels varying in theme and difficulty. User chooses level at Menu state.

Sequence Diagram:



UML Class Diagram:



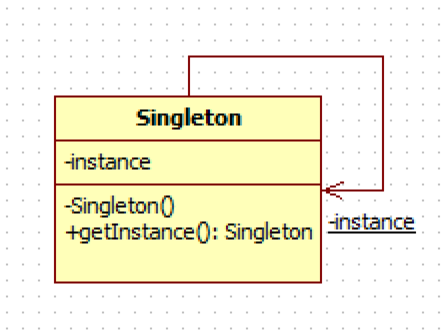
Used Patterns:

1. Singleton:

This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
This pattern involves a single class which is responsible to create an object while making sure that only single object 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.
In Game, Singelton used in classes which it's not needed to create more than one Object

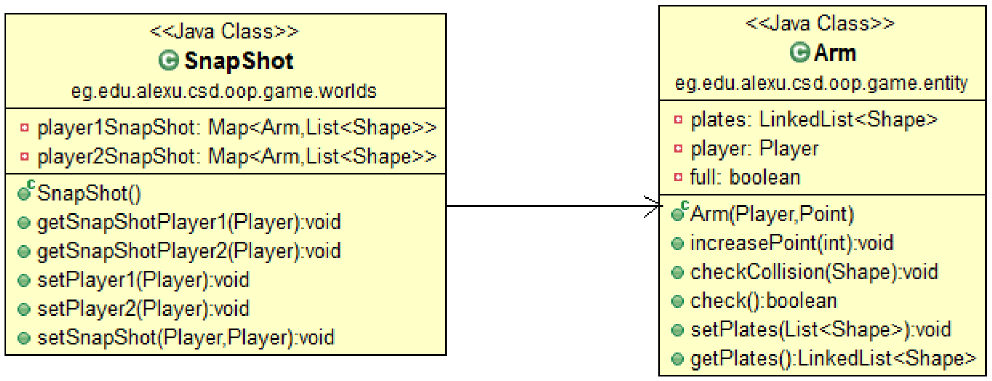
Classes using singleton:

- Configurations, ShapeConfiguration, WorldConfiguration.
- PlayerBuilder, ShapeFactory, ShapeImageHandler, ImageLoader.
- GameState, MenuState.



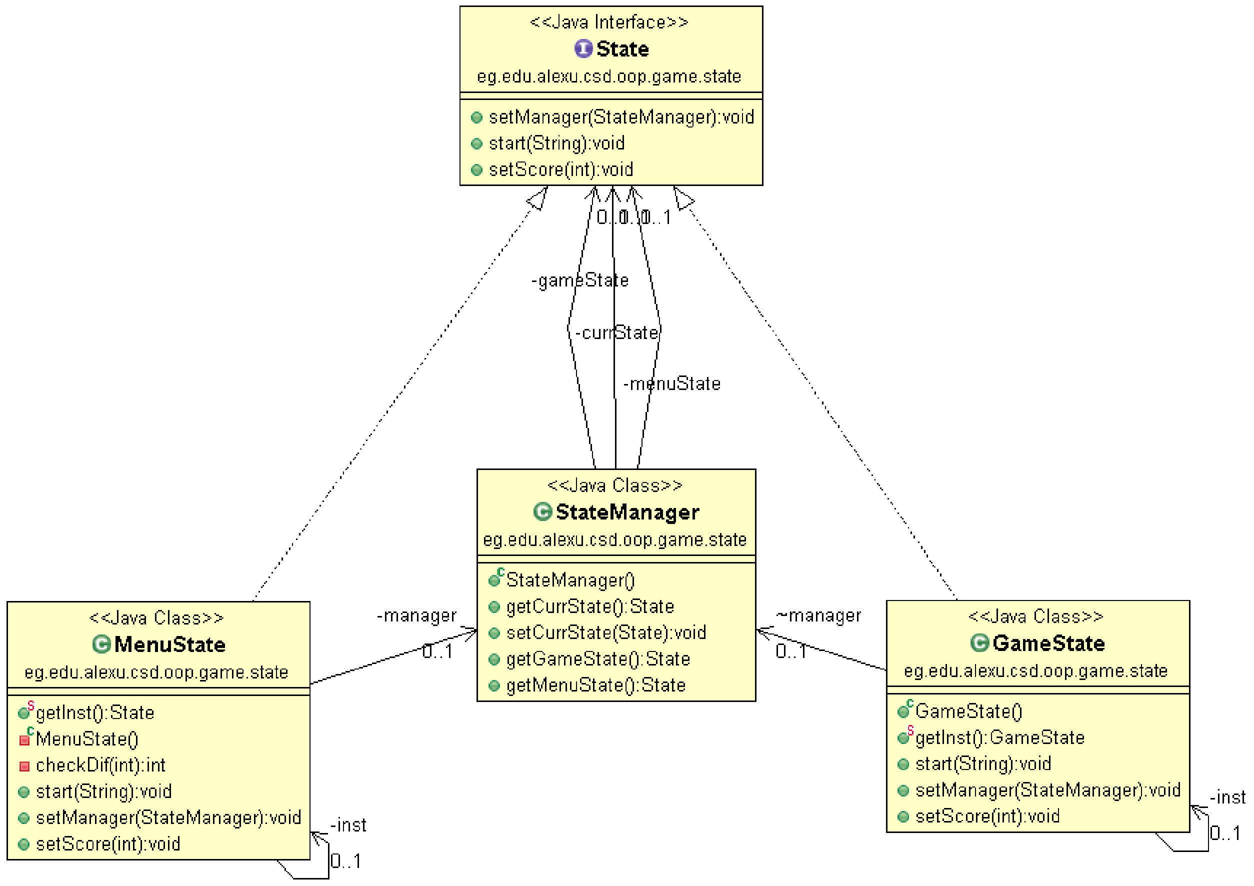
2. Snapshot:

- Used to restore state of an object to a previous state.
- Used to save current state for rewind case (save stacks of plates in arm and score).


<<Java Class>>
SnapShot
eg.edu.alexu.csd.oop.game.worlds
player1SnapShot: Map<Arm,List<Shape>>
player2SnapShot: Map<Arm,List<Shape>>
SnapShot()
getSnapShotPlayer1(Player):void
getSnapShotPlayer2(Player):void
setPlayer1(Player):void
setPlayer2(Player):void
setSnapShot(Player,Player):void
<<Java Class>>
Arm
eg.edu.alexu.csd.oop.game.entity
plates: LinkedList<Shape>
player: Player
full: boolean
Arm(Player,Point)
increasePoint(int):void
checkCollision(Shape):void
check():boolean
setPlates(List<Shape>):void
getPlates():LinkedList<Shape>

3. State:

In state design pattern we create an interface called state and classes implement that interface. There is a state manager which has an instance from each state and it changes the current state according to the state of the game such as game state and menu state, then is starts the current state. Each state can be changed from the other.

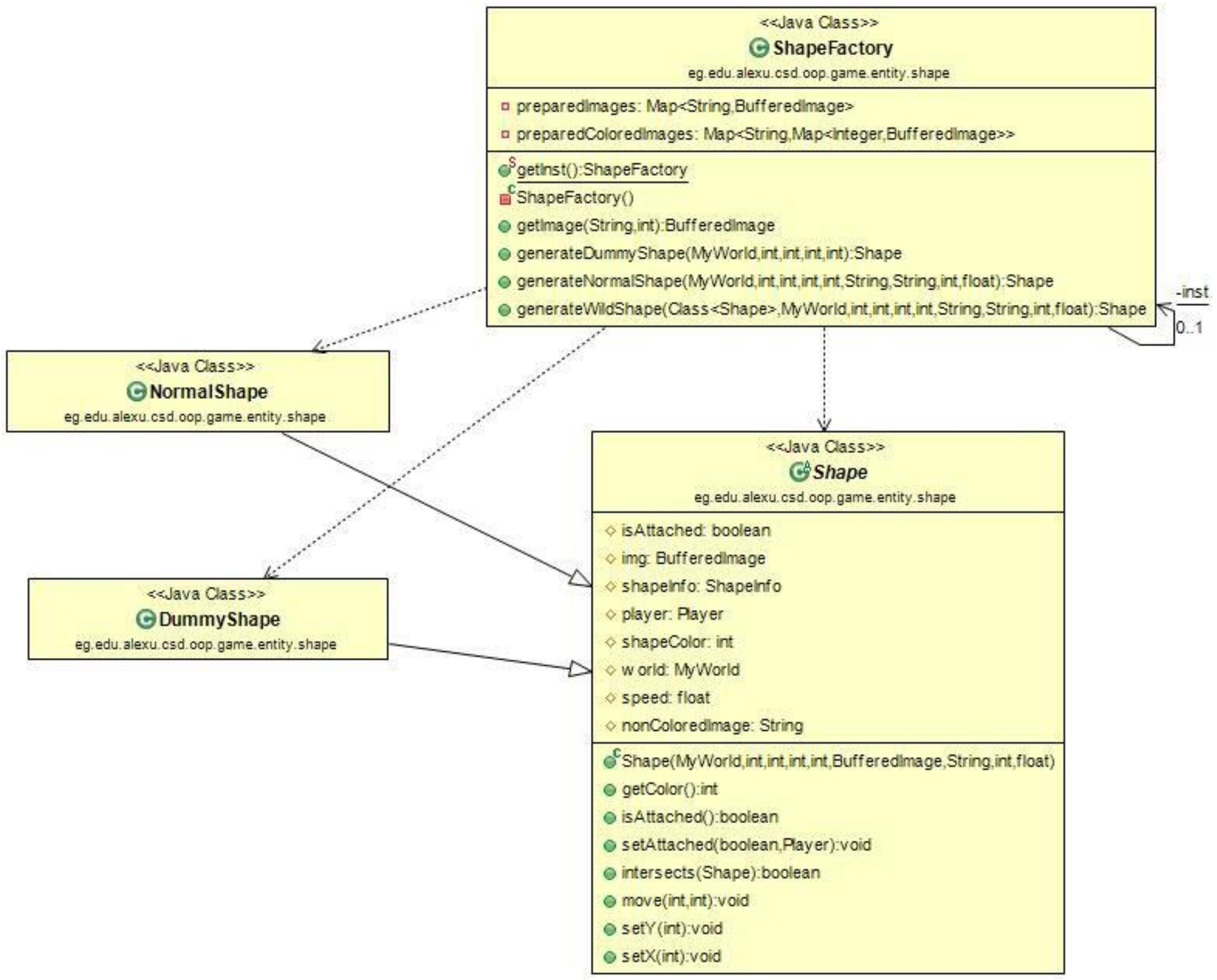


4. Factory:

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

Factory Classes in Project:

- Shape Factory: Responsible for formation of all shapes as; NormalShape, WildShape, DummyShape.

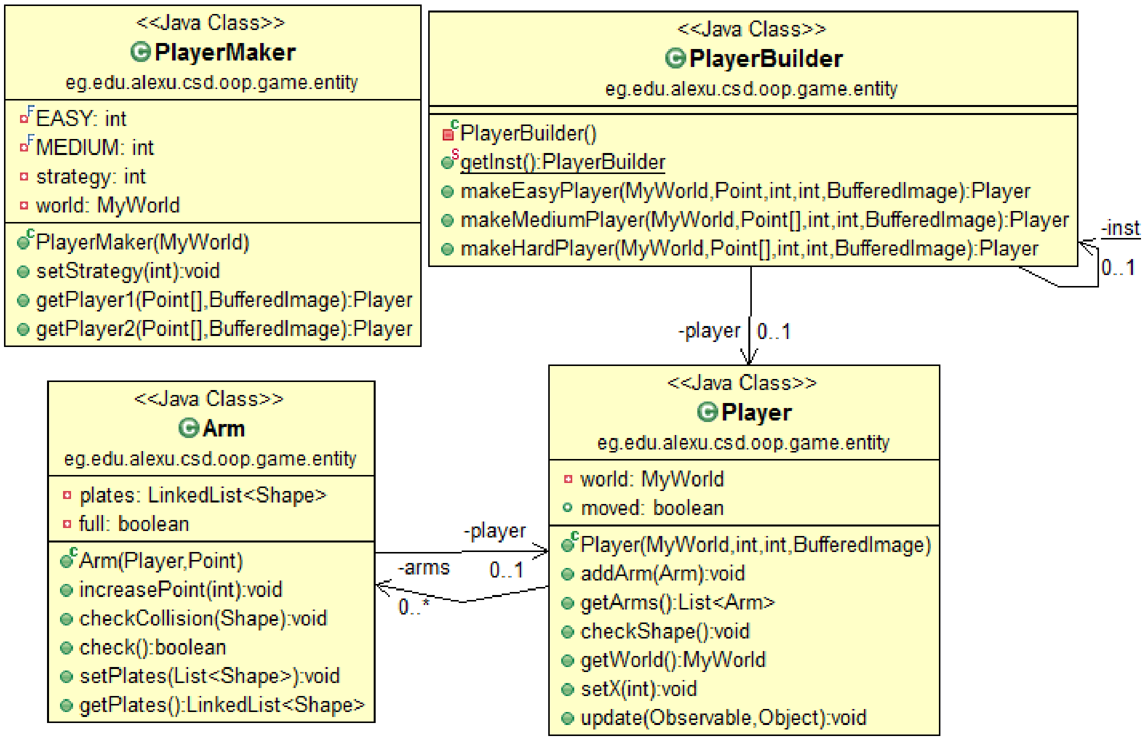


<<Java Class>>
ShapeFactory
eg.edu.alexu.csd.oop.game.entity.shape
preparedImages: Map<String,BufferedImage>
preparedColoredImages: Map<String,Map<Integer,BufferedImage>>
getInst():ShapeFactory
ShapeFactory()
getImage(String,int):BufferedImage
generateDummyShape(MyWorld,int,int,int,int):Shape
generateNormalShape(MyWorld,int,int,int,int,String,String,int,float):Shape
generateWildShape(Class<Shape>,MyWorld,int,int,int,int,String,String,int,float):Shape
-inst
0..1
<<Java Class>>
NormalShape
eg.edu.alexu.csd.oop.game.entity.shape
<<Java Class>>
DummyShape
eg.edu.alexu.csd.oop.game.entity.shape
<<Java Class>>
Shape
eg.edu.alexu.csd.oop.game.entity.shape
isAttached: boolean
img: BufferedImage
shapeInfo: ShapeInfo
player: Player
shapeColor: int
world: MyWorld
speed: float
nonColoredImage: String
Shape(MyWorld,int,int,int,int,BufferedImage,String,int,float)
getColor():int
isAttached():boolean
setAttached(boolean,Player):void
intersects(Shape):boolean
move(int,int):void
setY(int):void
setX(int):void

5. Builder:

Builds a complex object using simple objects and using a step by step approach.

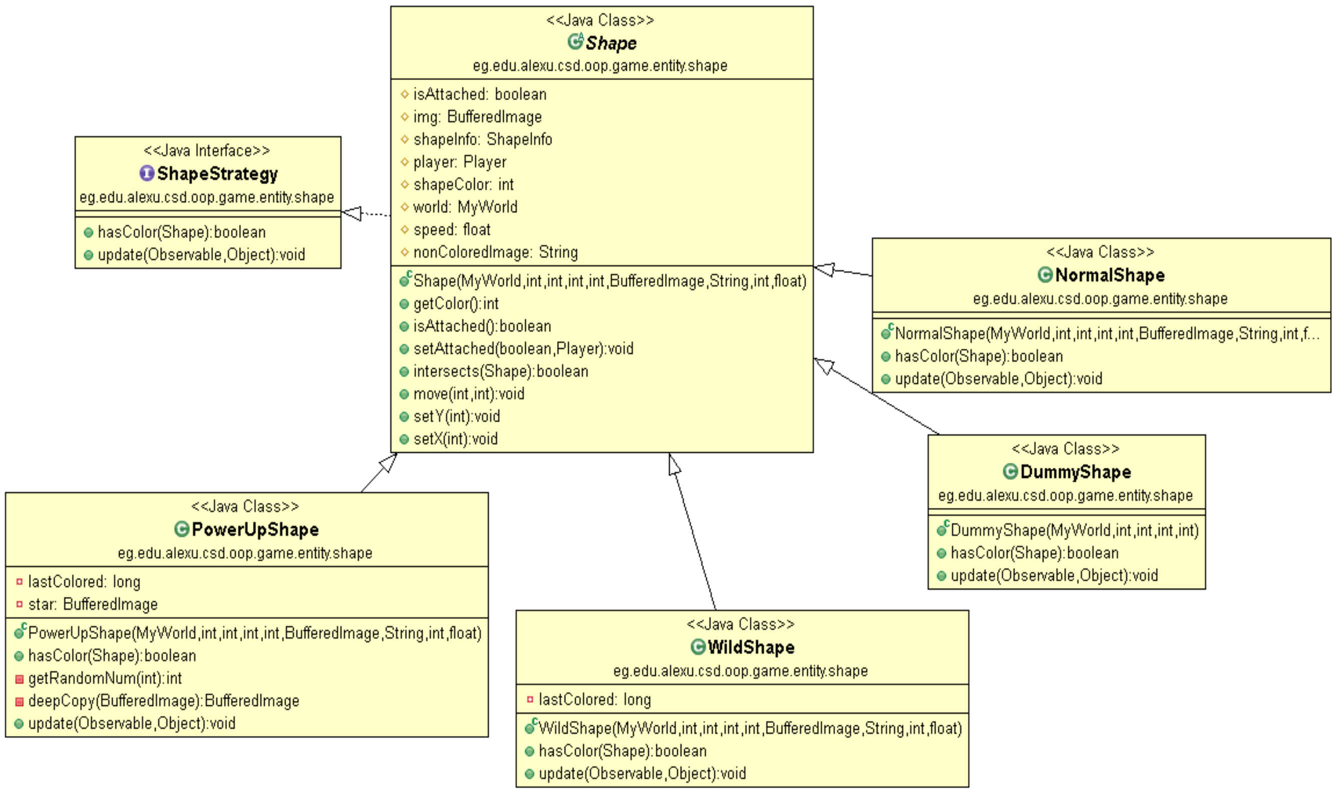
Builder is used in creating player structure consisting of concrete player and arms(plates Stacks).



<<Java Class>>
PlayerMaker
eg.edu.alexu.csd.oop.game.entity
EASY: int
MEDIUM: int
strategy: int
world: MyWorld
PlayerMaker(MyWorld)
setStrategy(int):void
getPlayer1(Point[],BufferedImage):Player
getPlayer2(Point[],BufferedImage):Player
<<Java Class>>
PlayerBuilder
eg.edu.alexu.csd.oop.game.entity
PlayerBuilder()
getInst():PlayerBuilder
makeEasyPlayer(MyWorld,Point,int,int,BufferedImage):Player
makeMediumPlayer(MyWorld,Point[],int,int,BufferedImage):Player
makeHardPlayer(MyWorld,Point[],int,int,BufferedImage):Player
-inst
0..1
-player
0..1
<<Java Class>>
Arm
eg.edu.alexu.csd.oop.game.entity
plates: LinkedList<Shape>
full: boolean
Arm(Player,Point)
increasePoint(int):void
checkCollision(Shape):void
check():boolean
setPlates(List<Shape>):void
getPlates():LinkedList<Shape>
-player
-arms
0..1
0..*
<<Java Class>>
Player
eg.edu.alexu.csd.oop.game.entity
world: MyWorld
moved: boolean
Player(MyWorld,int,int,BufferedImage)
addArm(Arm):void
getArms():List<Arm>
checkShape():void
getWorld():MyWorld
setX(int):void
update(Observable,Object):void

6. Strategy:

We create different shapes which represent various strategies and a Shape object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the Shape object.There is an interface called ShapeStrategy contains functions and this functions are implemented according to the shape behavior which is loaded dynamically.

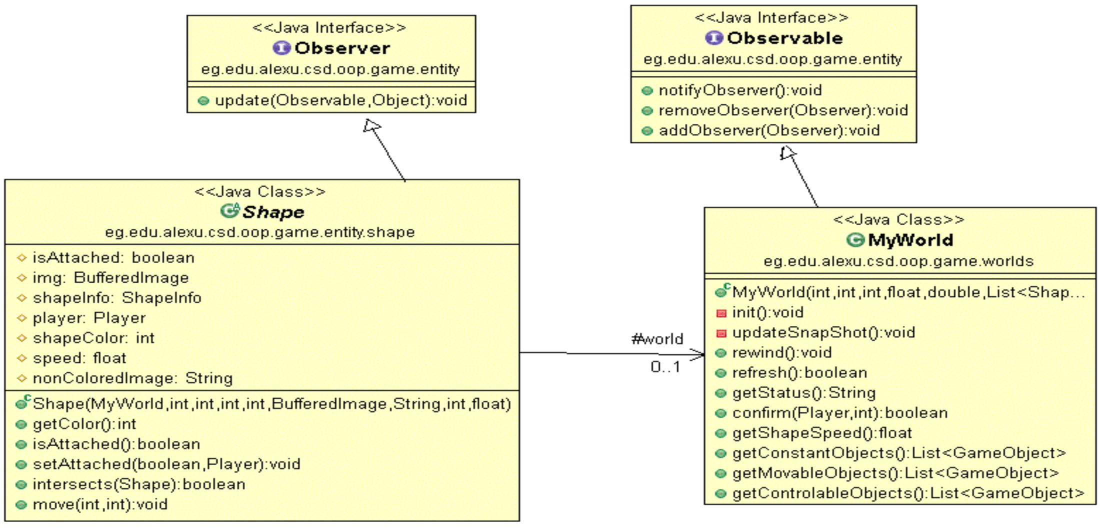


<<Java Class>>
Shape
eg.edu.alexu.csd.oop.game.entity.shape
isAttached: boolean
img: BufferedImage
shapeInfo: ShapeInfo
player: Player
shapeColor: int
world: MyWorld
speed: float
nonColoredImage: String
Shape(MyWorld,int,int,int,int,BufferedImage,String,int,float)
getColor():int
isAttached():boolean
setAttached(boolean,Player):void
intersects(Shape):boolean
move(int,int):void
setY(int):void
setX(int):void
<<Java Interface>>
ShapeStrategy
eg.edu.alexu.csd.oop.game.entity.shape
hasColor(Shape):boolean
update(Observable,Object):void
<<Java Class>>
NormalShape
eg.edu.alexu.csd.oop.game.entity.shape
NormalShape(MyWorld,int,int,int,int,BufferedImage,String,int,f...
hasColor(Shape):boolean
update(Observable,Object):void
<<Java Class>>
DummyShape
eg.edu.alexu.csd.oop.game.entity.shape
DummyShape(MyWorld,int,int,int,int)
hasColor(Shape):boolean
update(Observable,Object):void
<<Java Class>>
PowerUpShape
eg.edu.alexu.csd.oop.game.entity.shape
lastColored: long
star: BufferedImage
PowerUpShape(MyWorld,int,int,int,int,BufferedImage,String,int,float)
hasColor(Shape):boolean
getRandomNum(int):int
deepCopy(BufferedImage):BufferedImage
update(Observable,Object):void
<<Java Class>>
WildShape
eg.edu.alexu.csd.oop.game.entity.shape
lastColored: long
WildShape(MyWorld,int,int,int,int,BufferedImage,String,int,float)
hasColor(Shape):boolean
update(Observable,Object):void

7. Observer and Observable:

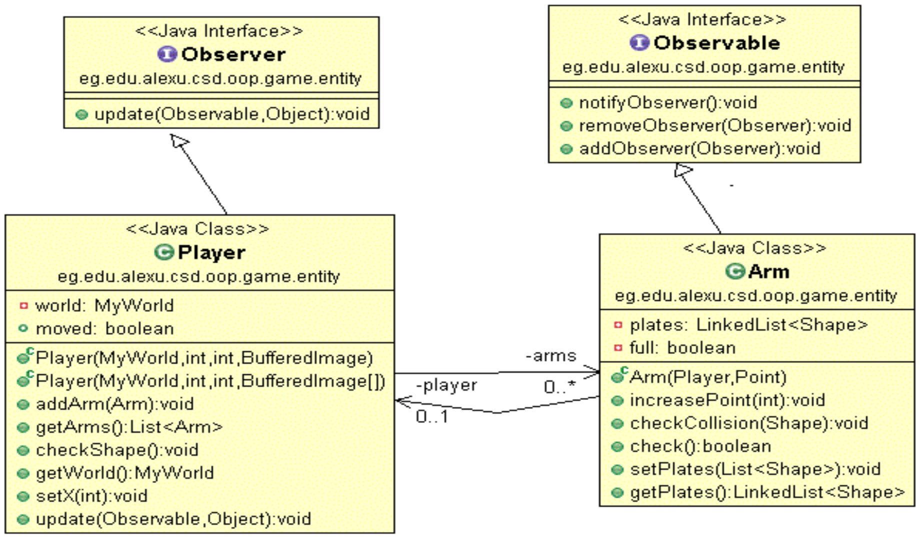
Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.In our game we used it in two times:

- Moving shapes downwards the shapes are the observers and the world is an observable. When refresh is called in the world we notify the shapes to update its self-move downwards.



<<Java Interface>>
Observer
eg.edu.alexu.csd.oop.game.entity
update(Observable,Object):void
<<Java Interface>>
Observable
eg.edu.alexu.csd.oop.game.entity
notifyObserver():void
removeObserver(Observer):void
addObserver(Observer):void
<<Java Class>>
Shape
eg.edu.alexu.csd.oop.game.entity.shape
isAttached: boolean
img: BufferedImage
shapeInfo: ShapeInfo
player: Player
shapeColor: int
speed: float
nonColoredImage: String
Shape(MyWorld,int,int,int,int,BufferedImage,String,int,float)
getColor():int
isAttached():boolean
setAttached(boolean,Player):void
intersects(Shape):boolean
move(int,int):void
#world
0..1
<<Java Class>>
MyWorld
eg.edu.alexu.csd.oop.game.worlds
MyWorld(int,int,int,float,double,List<Shap...
init():void
updateSnapShot():void
rewind():void
refresh():boolean
getStatus():String
confirm(Player,int):boolean
getShapeSpeed():float
getConstantObjects():List<GameObject>
getMovableObjects():List<GameObject>
getControlableObjects():List<GameObject>

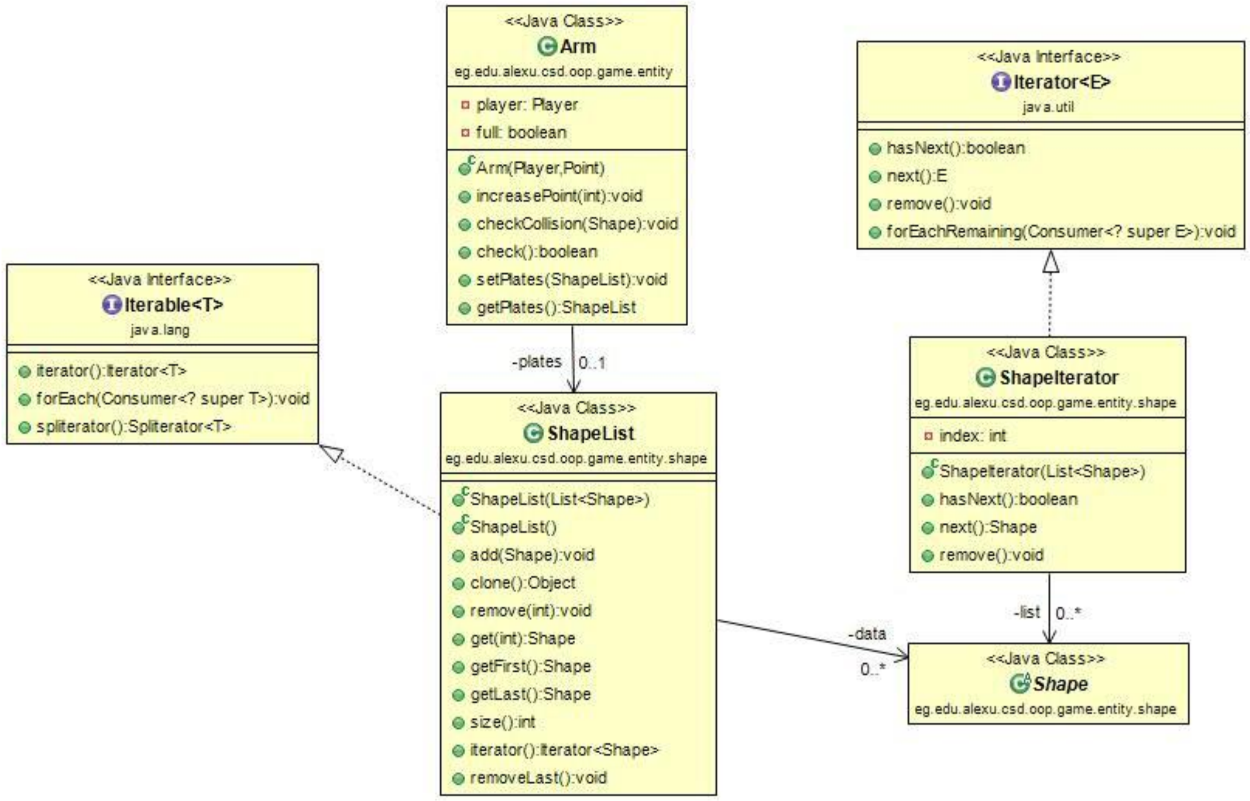
- Relation between the player and the arm, the player is an observer and the arm is observable, when three shapes are formed with the same color the arm of the player, the arm notifies the player to increase the score.



<<Java Interface>>
Observer
eg.edu.alexu.csd.oop.game.entity
update(Observable,Object):void
<<Java Interface>>
Observable
eg.edu.alexu.csd.oop.game.entity
notifyObserver():void
removeObserver(Observer):void
addObserver(Observer):void
<<Java Class>>
Player
eg.edu.alexu.csd.oop.game.entity
world: MyWorld
moved: boolean
Player(MyWorld,int,int,BufferedImage)
Player(MyWorld,int,int,BufferedImage[])
addArm(Arm):void
getArms():List<Arm>
checkShape():void
getWorld():MyWorld
setX(int):void
update(Observable,Object):void
-arms
0..*
-player
0..1
<<Java Class>>
Arm
eg.edu.alexu.csd.oop.game.entity
plates: LinkedList<Shape>
full: boolean
Arm(Player,Point)
increasePoint(int):void
checkCollision(Shape):void
check():boolean
setPlates(List<Shape>):void
getPlates():LinkedList<Shape>

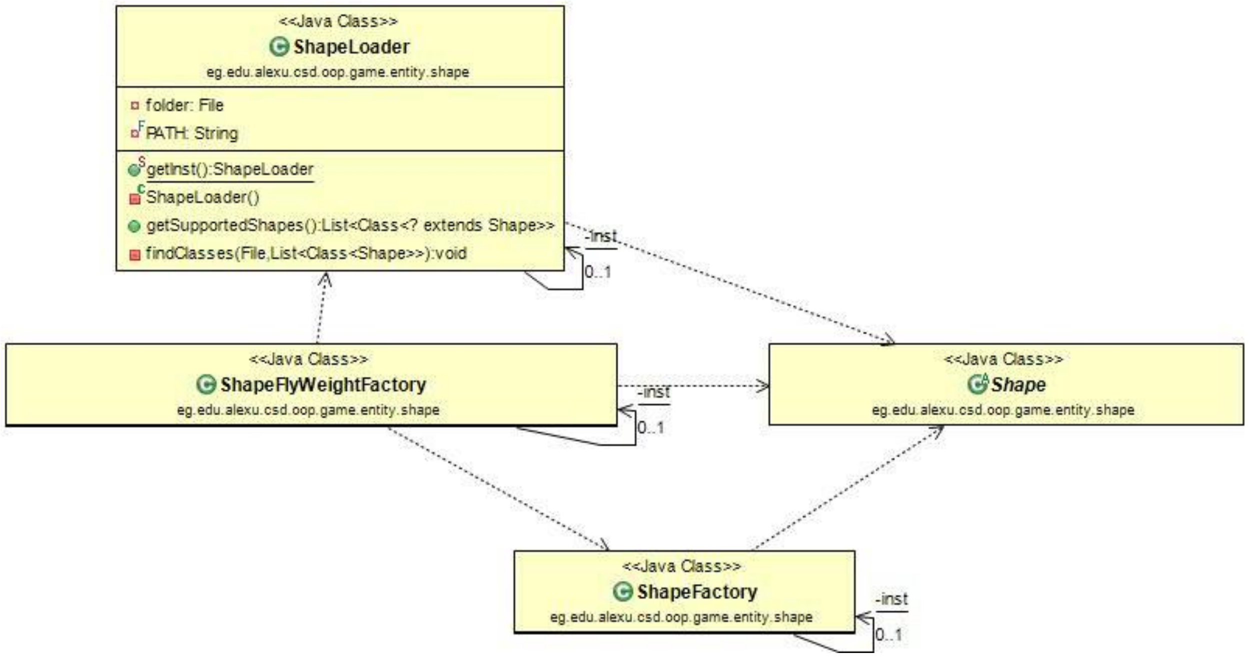
8. Iterator and Iterable:

Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation , in the game we need to access the shapes lying in the arms of the player alot (once each refresh) , so we needed to minimize the iteration time as much as we could , so we used the iterator allowing us to access them in much less time that regular method.



9. Dynamic Linkage:

It Defines procedure of arbitrary loading and usage of classes at runtime - with only requirement for the classes, that they implement an interface or an abstract class, known at compile-time. Dynamic linkage simply reduces compile-time restrictions on behalf of more run-time freedom regarding class implementation. The procedure allows change in implementation of classes at run-time, without neither re-compiling nor stopping the program , and in the game we need to add shapes to the game in the form of plugins to expand the variety of the shapes in the game , so we used the dynamic linkage to load external shapes in run-time .

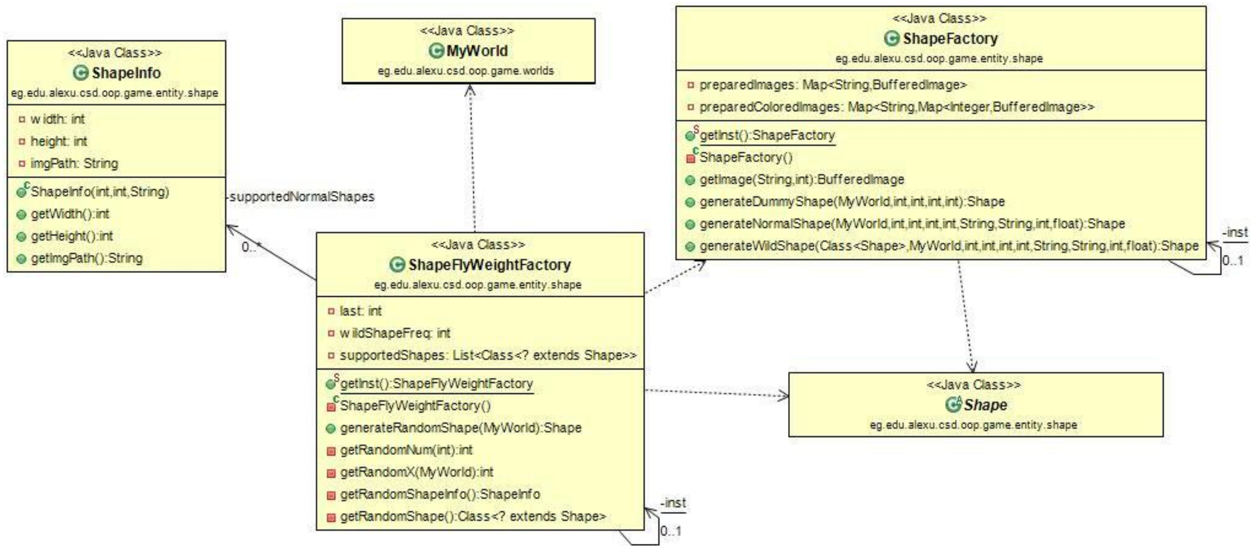

<<Java Class>>
ShapeLoader
eg.edu.alexu.csd.oop.game.entity.shape
folder: File
PATH: String
getInst():ShapeLoader
ShapeLoader()
getSupportedShapes():List<Class<? extends Shape>>
findClasses(File,List<Class<Shape>>):void
-inst
0..1
<<Java Class>>
ShapeFlyWeightFactory
eg.edu.alexu.csd.oop.game.entity.shape
-inst
0..1
<<Java Class>>
Shape
eg.edu.alexu.csd.oop.game.entity.shape
<<Java Class>>
ShapeFactory
eg.edu.alexu.csd.oop.game.entity.shape
-inst
0..1

10. Fly Weight:

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.

In the game each time we want to generate a shape we need to pass a colored image to it , normally we would load the image and then color it each time we generate a shape where the total number of shapes in the game is very small compared to the number of times we generate the images (polymorphism).

It is ideal to use the flyweight in this case on the images and the colored images, so we only load each image only once , and also color it only once using the map.


<<Java Class>>
ShapeInfo
eg.edu.alexu.csd.oop.game.entity.shape
width: int
height: int
imgPath: String
ShapeInfo(int,int,String)
getWidth():int
getHeight():int
getImgPath():String
-supportedNormalShapes
0..*
<<Java Class>>
MyWorld
eg.edu.alexu.csd.oop.game.worlds
<<Java Class>>
ShapeFactory
eg.edu.alexu.csd.oop.game.entity.shape
preparedImages: Map<String,BufferedImage>
preparedColoredImages: Map<String,Map<Integer,BufferedImage>>
getInst():ShapeFactory
ShapeFactory()
getImage(String,int):BufferedImage
generateDummyShape(MyWorld,int,int,int,int):Shape
generateNormalShape(MyWorld,int,int,int,int,String,String,int,float):Shape
generateWildShape(Class<Shape>,MyWorld,int,int,int,int,String,String,int,float):Shape
-inst
0..1
<<Java Class>>
ShapeFlyWeightFactory
eg.edu.alexu.csd.oop.game.entity.shape
last: int
wildShapeFreq: int
supportedShapes: List<Class<? extends Shape>>
getInst():ShapeFlyWeightFactory
ShapeFlyWeightFactory()
generateRandomShape(MyWorld):Shape
getRandomNum(int):int
getRandomX(MyWorld):int
getRandomShapeInfo():ShapeInfo
getRandomShape():Class<? extends Shape>
-inst
0..1
<<Java Class>>
Shape
eg.edu.alexu.csd.oop.game.entity.shape