CoP

Clown of Plates

Omar Emam	44
Ahmed Waleed	9
Mohamed Zidan	57
Abd-Elrahman Adel	37

Report Content:

Program Specifications:

-Program has interactive GUI which will help the user change the setting (size, quality, sound …),the level (easy,medium,hard)and inside the game the user will be able to pause , un/mute sound and see his time and score all in very friendly GUI.

-The user will have variety in plates shapes and speeds in each level

-A logging file is created each game to save the logger output in side it

-After one is over the user will be able to exit or return to start frame where he will be able to change the setting or start a new game

User Guide:

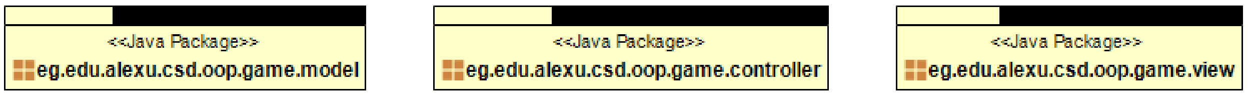
-To play the game:

1. Choose your preferred setting
2. Click Start
3. Choose your preferred level
4. Click play button
5. Use arrows to move your clown right and left
6. Each 3 plates give you a point
7. Collect as much points as you can before time is up

UML Diagrams & Design Patterns:

- MVC :

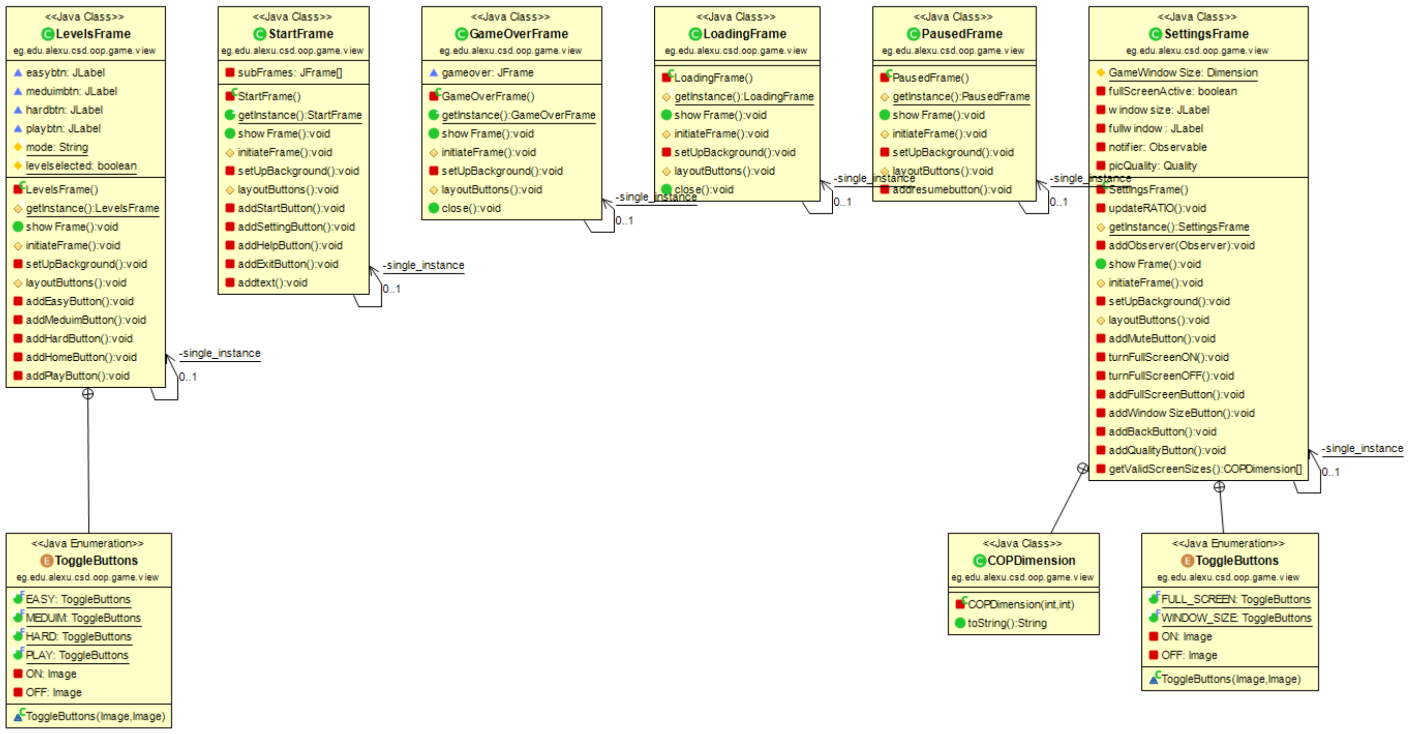
  The Model View Controller (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information.



<<Java Package>>
eg.edu.alexu.csd.oop.game.model
<<Java Package>>
eg.edu.alexu.csd.oop.game.controller
<<Java Package>>
eg.edu.alexu.csd.oop.game.view

- Singleton :

  It's used in all view elements that will allow us to have only one frame at once with no duplicates



- <u>Factory :</u>

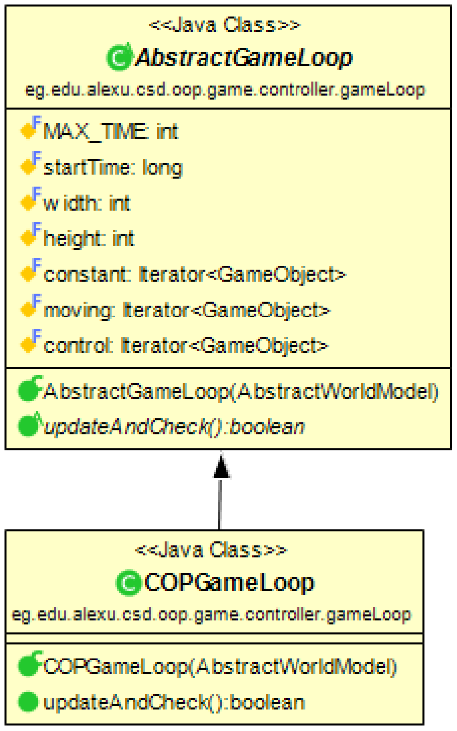
In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

We used this pattern generating the plates

- <u>Iretrator:</u>

The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements

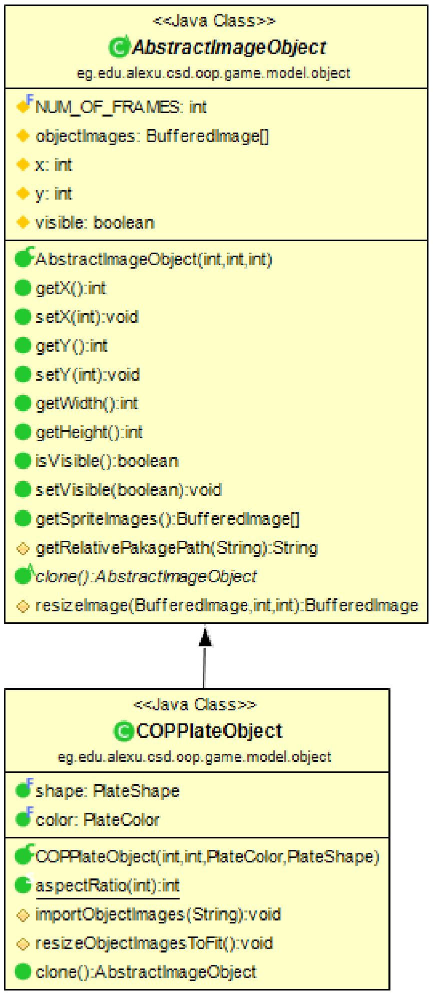
We used this pattern looping over game elements



<<Java Class>>
AbstractGameLoop
eg.edu.alexu.csd.oop.game.controller.gameLoop
MAX_TIME: int
startTime: long
width: int
height: int
constant: Iterator<GameObject>
moving: Iterator<GameObject>
control: Iterator<GameObject>
AbstractGameLoop(AbstractWorldModel)
updateAndCheck():boolean
<<Java Class>>
COPGameLoop
eg.edu.alexu.csd.oop.game.controller.gameLoop
COPGameLoop(AbstractWorldModel)
updateAndCheck():boolean

- ## Dynamic Linkage:

We used this pattern for the function of all plates

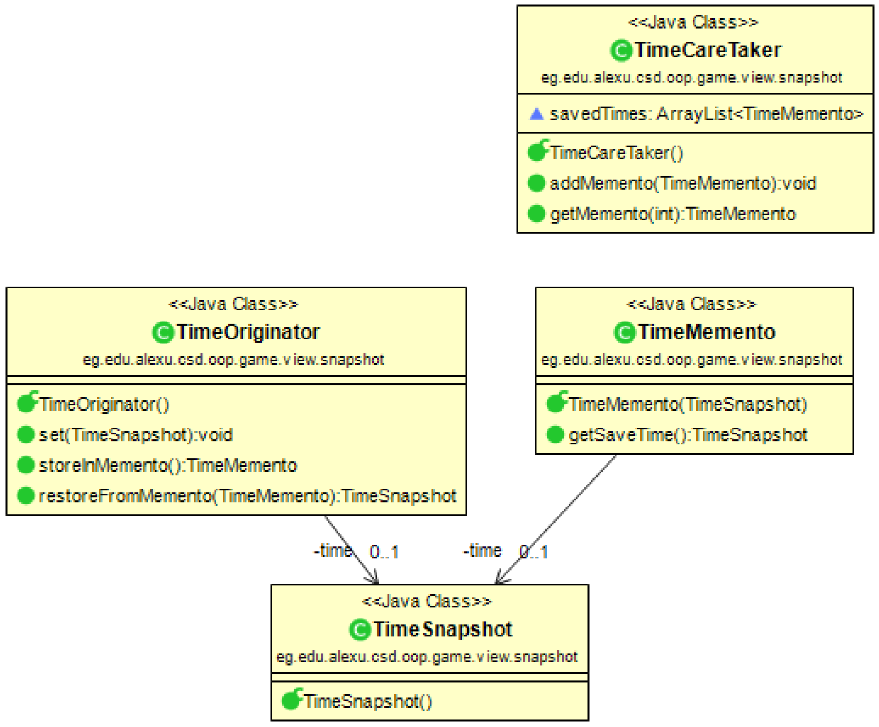


<<Java Class>>
AbstractImageObject
eg.edu.alexu.csd.oop.game.model.object
NUM_OF_FRAMES: int
objectImages: BufferedImage[]
x: int
y: int
visible: boolean
AbstractImageObject(int,int,int)
getX():int
setX(int):void
getY():int
setY(int):void
getWidth():int
getHeight():int
isVisible():boolean
setVisible(boolean):void
getSpriteImages():BufferedImage[]
getRelativePakagePath(String):String
clone():AbstractImageObject
resizeImage(BufferedImage,int,int):BufferedImage
<<Java Class>>
COPPlateObject
eg.edu.alexu.csd.oop.game.model.object
shape: PlateShape
color: PlateColor
COPPlateObject(int,int,PlateColor,PlateShape)
aspectRatio(int):int
importObjectImages(String):void
resizeObjectImagesToFit():void
clone():AbstractImageObject

• Snapshots:

Memento pattern is used to restore state of an object to a previous state. It is also known as snapshot pattern. A memento is like a restore point during the life cycle on the object, which the client application can use to restore the object state to its state.

We used this pattern to overcome the pause issue in the engine

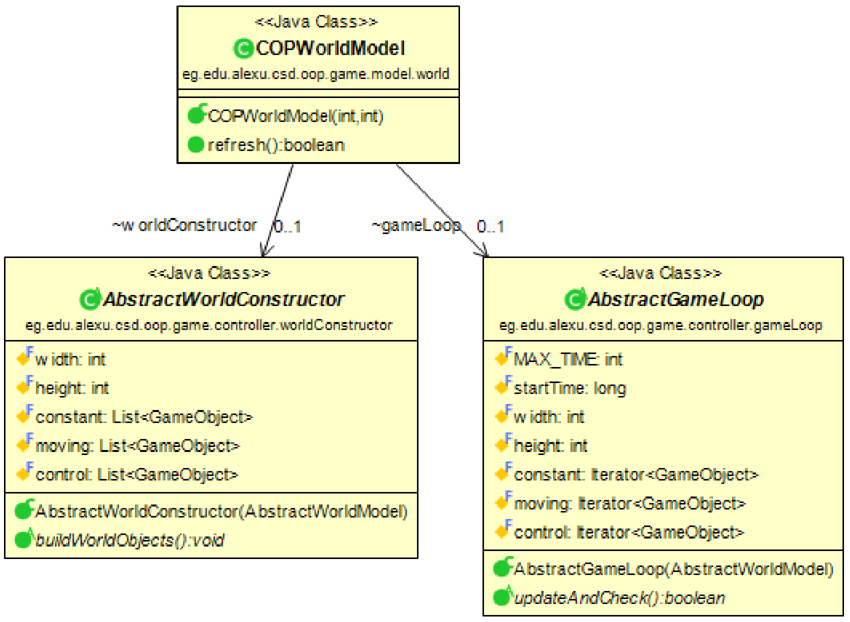


<<Java Class>>
TimeCareTaker
eg.edu.alexu.csd.oop.game.view.snapshot
savedTimes: ArrayList<TimeMemento>
TimeCareTaker()
addMemento(TimeMemento):void
getMemento(int):TimeMemento
<<Java Class>>
TimeOriginator
eg.edu.alexu.csd.oop.game.view.snapshot
TimeOriginator()
set(TimeSnapshot):void
storeInMemento():TimeMemento
restoreFromMemento(TimeMemento):TimeSnapshot
<<Java Class>>
TimeMemento
eg.edu.alexu.csd.oop.game.view.snapshot
TimeMemento(TimeSnapshot)
getSaveTime():TimeSnapshot
-time 0..1
-time 0..1
<<Java Class>>
TimeSnapshot
eg.edu.alexu.csd.oop.game.view.snapshot
TimeSnapshot()

• Strategy:

The strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

We used this pattern to distribute the logic of the world among various components



<<Java Class>>
COPWorldModel
eg.edu.alexu.csd.oop.game.model.world
COPWorldModel(int,int)
refresh():boolean
~worldConstructor 0..1
~gameLoop 0..1
<<Java Class>>
AbstractWorldConstructor
eg.edu.alexu.csd.oop.game.controller.worldConstructor
width: int
height: int
constant: List<GameObject>
moving: List<GameObject>
control: List<GameObject>
AbstractWorldConstructor(AbstractWorldModel)
buildWorldObjects():void
<<Java Class>>
AbstractGameLoop
eg.edu.alexu.csd.oop.game.controller.gameLoop
MAX_TIME: int
startTime: long
width: int
height: int
constant: Iterator<GameObject>
moving: Iterator<GameObject>
control: Iterator<GameObject>
AbstractGameLoop(AbstractWorldModel)
updateAndCheck():boolean

- State:

The state pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes.

We used this pattern to control the pictures quality

- Flyweight:

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

We used this pattern to avoid loading the pictures more than once

- Observer:

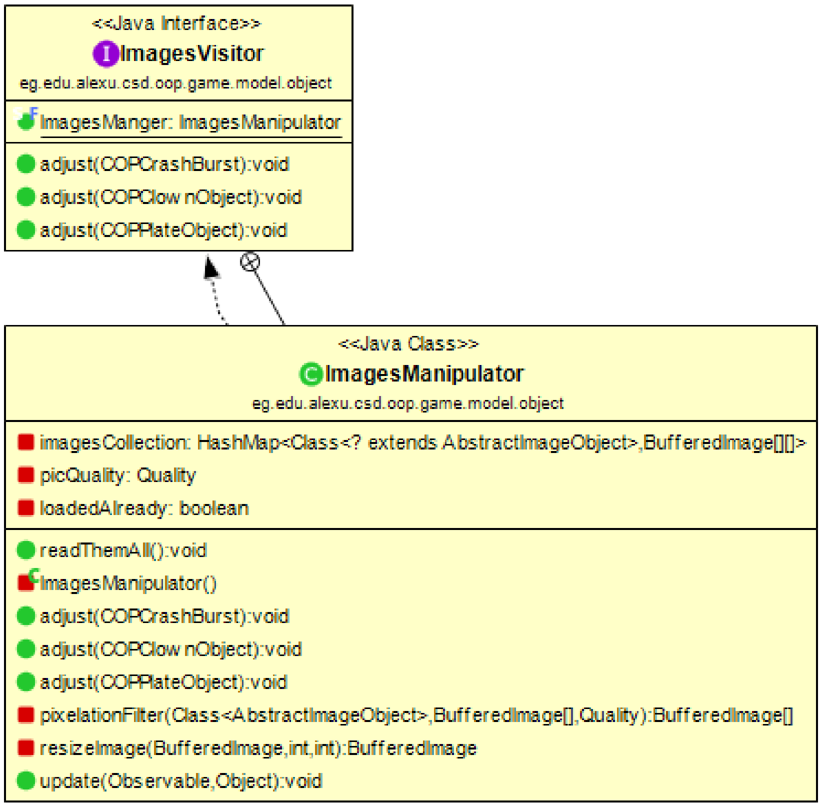
The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

We used this pattern to notify the image manager with quality needed

- Visitor:

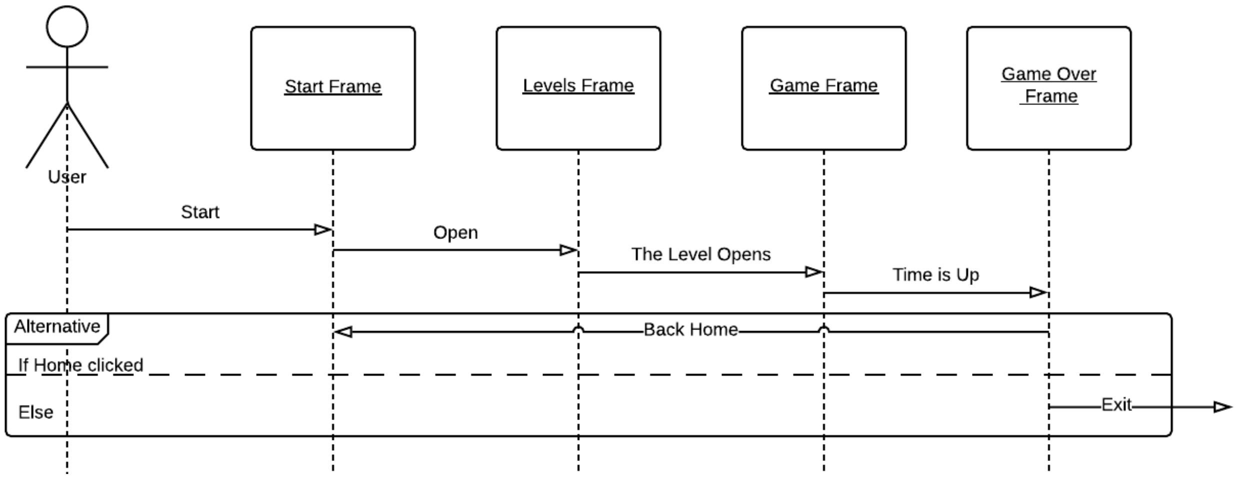
The visitor design pattern is a way of separating an algorithm from an object structure on which it operates

We used this pattern to allow the image manager to give each element its image



<<Java Interface>>
ImagesVisitor
eg.edu.alexu.csd.oop.game.model.object
ImagesManger: ImagesManipulator
adjust(COPCrashBurst):void
adjust(COPClownObject):void
adjust(COPPlateObject):void
<<Java Class>>
ImagesManipulator
eg.edu.alexu.csd.oop.game.model.object
imagesCollection: HashMap<Class<? extends AbstractImageObject>,BufferedImage[][]>
picQuality: Quality
loadedAlready: boolean
readThemAll():void
ImagesManipulator()
adjust(COPCrashBurst):void
adjust(COPClownObject):void
adjust(COPPlateObject):void
pixelationFilter(Class<AbstractImageObject>,BufferedImage[],Quality):BufferedImage[]
resizeImage(BufferedImage,int,int):BufferedImage
update(Observable,Object):void

- ## Sequence Diagram :



User
Start Frame
Levels Frame
Game Frame
Game Over Frame
Start
Open
The Level Opens
Time is Up
Alternative
Back Home
If Home clicked
Else
Exit