Bilkent University

Department of Computer Engineering

Object-Oriented Software Engineering Course Project

CS319 Course Project: Virion

Design Report

Mert INAN, Ulugbek IRMATOV, Irmak YILMAZ

Supervisor: Uğur DOĞRUSÖZ

Design Report

Nov 28, 2016

This report is submitted to GitHub in partial fulfillment of the requirements of the Object-Oriented Software Engineering course CS319.

Contents

Design Report

CS319 Course Project: Virion

1 Introduction

In this report, high level design of the Virion system will be discussed and evaluated. It contains the overall introduction to the system and its design goals, and then the details of the architecture and subsystem components will be given.

1.1 Purpose of the System

Virion is an educational game where player tries to save a cell from virus attacks. Player can defend using proteins and other cell organelles such as mitochondria, vacuole, and cell wall. There are different kinds of viruses. Difficulty of the game is proportional to time. As time passes more dangerous viruses start coming. To get high score, player must keep cell alive as long as possible.

1.2 Design Goals

The main design goals of the system are dependent on the user-system interaction rather than the actual system configurations. As the implementation of the software does not involve complex algorithm design, and the crucial parts occur in the duration of the gameplay, design goals such as modifiability, adaptability, portability or maintainability are not the main concern, but will be incorporated into the implementation by itself.

On the other hand design goals such as user-friendliness, ease of learning, ease of remembering and ease of use are primary concerns. As Virion is an educational game, it should be easy for the user to understand the environment and the components involved. Even though the game can have a steep learning curve, it is necessary for the user to retain information learned from the game.

Other general design goals such as efficiency, high-performance are also important as the game need to perform its operations highly efficiently, as the outcome of the game depends on time, it is crucial to have a high-performing system.

During the design of the system, some design goals will need to be preferred, creating trade-offs. Here are two trade-offs that will need to be considered.

Functionality – Usability: As the system is a biology education game, it needs to be functional rather than usable at some instances. It is designed to maximize functionality and usability most of the times; however, if a choice is needed to be made, then functionality will be favored as the game should be adequate enough to model the real world interactions between cells and viruses. This may lead to a decrease in usability, such as extensive menus and logical links between organelles, which is not a significant decrease overall.

Time – Memory: Interactions between internal game components are heavily-dependent on time –scoring is also done according to time- hence, it is necessary to favor time instead of memory if the need arises. Even though time will be favored, overall memory consumption is not significant as there are no complex algorithms with high memory need, yet images of the components may sometimes slow down the processes which may lead to a delay during gameplay which is not a desirable scenario. Hence, if necessary, image quality will be shrunk to increase time reliability.

2 Software architecture

In this section of the design document, we will be analyzing the high level architecture of the Virion system by evaluating the subsystem components.

Among the widely applied architectures, most promising ones for this project were Model View Controller (MVC) and 3-Layer. We have discussed each architecture's drawbacks and advantages and decided on the one that best represents Virion's high level design. At the first glance, MVC seems to grasp the overall concept of the system but lacks the filesystem component to write the high score list in. Hence, we are choosing 3-Layer Architecture because we will be writing the high score list to the file in order for it to be persistent among sessions.

2.1 Subsystem decomposition

As the decided architecture of the system is 3-Layer, the subsystems will need to comply with the specific layering of that architecture. The topmost layer in this architecture corresponds to the user interface, middle layer has the actual business logic or the controllers of the game and the lowest layer contains the filesystem data management subsystem.

There are four subsystems present in the system: User Interface, Game Management, Game Components, and High Score List. This split is planned, at first, according to a MVC model; however, in the end it is transformed into a 3-Layer structure with High Score List subsystem being in the filesystem layer. Opaque layering is used in the design in order to increase flexibility, as the user interface layer or the high score list layer can change as the future stages of the project may turn it into a multi-user system.

High coherence and low coupling is tried to be achieved by making the boundaries of the layers as distinct as possible. UML Component diagram is used to show the relation among components at different layers as in Figure 1.

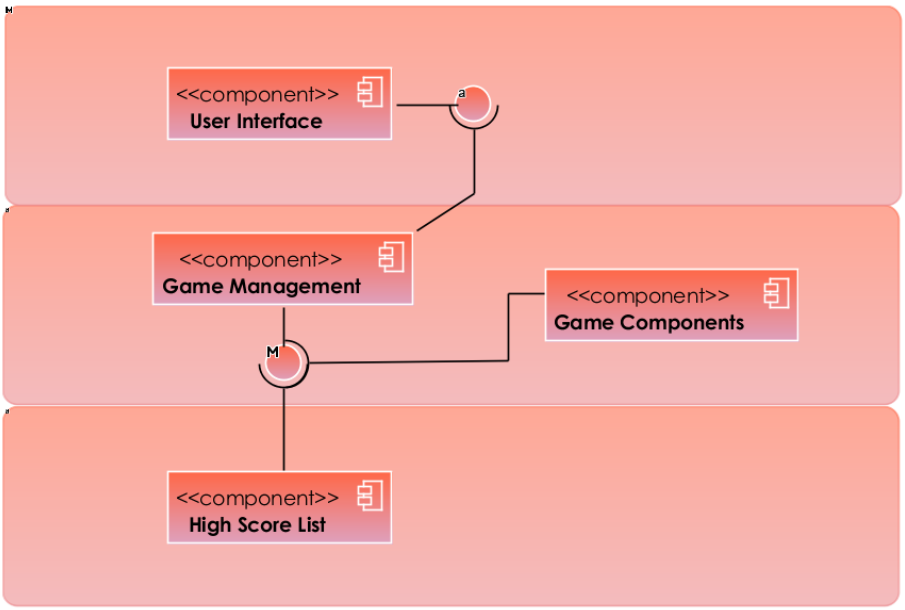


Figure 1 This is the UML Component Diagram of the 3-Layer Architecture of Virion system with subsystems.

2.2 Hardware/software mapping

As the Virion system does not demand a high processor power, it is enough to use the regular hardware with a single processor and optimized software to run the application. As there aren't multiple users, there won't be a lot of requests to filesystem which would not create a need for an extraordinary memory management. Hence, a regular computer with an operating system that is supporting Java applications will be adequate.

2.3 Persistent data management

The only data that needs to be persistent throughout different sessions of the program is the high score data. This data will not be regularly accessed either for read or write processes. Even in such a case, the writing will be done by the game management subsystem only, without any concurrency while writing. Two different options are available for the high score data management: filesystem and database. Because the database would require a lot of maintenance and is redundant without concurrencies in the system, it is decided to use filesystem database management to save high score data across different game executions.

Data does not need to be archived as the user will be able to see the top five high scores only which does not need to be stored throughout time intervals as the previous data is not needed and can be lost. Querying of the data will be done by interfaces inside the High Score List subsystem which would use regular file read/write operations.

2.4 Access control and security

As this is a single-player game, it has an open access to any player. It does not contain any authentication method. Accessing the game can be done by anybody who is authorized enough by the operating system to execute the game, itself. We will select procedure-driven centralized control for the control of the software itself as there are no multiple users, and control resides in one central controller object in the program code.

2.5 Boundary conditions

In the boundary conditions, the system will be acting according to the following.

Initialization: While the game is brought from a non-initialized state –such as the initial executable of the program- to the steady-state, the main processes will be done by the user interface subsystem. The user interface will display the splash screen during startup and then show the main menu of the game. Only the image data will be accessed for drawing the user interface during startup.

Termination: All cleanup procedures will be started by the main game controller in the Game Management subsystem. No single subsystem is allowed to terminate on its own. Every subsystem is notified if a single subsystem is terminated. Read and write to the filesystem will be completed before termination.

Failure: All game or computer related failures will result in loss of game state and score, as there is no save functionality. If a failure occurs during read and write to the high score list, warning messages will be shown and if necessary the highs core list will be erased. If failure occurs during getting the background images or component images, a restart of the game will be requested.

3 Subsystem services

The services that will be provided under the four main subsystems are as follows.

User Interface: Displays the screens, menu items, and game component images and notifies the game controller in the Game Management subsystem about any interactions between the screens, buttons, components and the player.

Game Management: Contains the main game controller class and other relevant controller classes that request high score updates to the list in the filesystem, screen updates to the user interface, game component state updates, movement updates, score and time updates.

Game Components: Contains all the model classes of the game objects and provide services to the system such as providing information about each game object, containing reference ids to the images of the game objects, and providing information about the availability of a game object.

High Score List: Provides services to read and write to the filesystem about the top five high scores upon instructions from the Game Management subsystem.

4 Low-Level Design

In this section of the report, we will be examining the final object model and the details of the classes. These details include class signatures such as method return types, method outputs, parameters, and attributes of the class. Furthermore, constraints on the design of the object model are also evaluated such as preconditions, invariants and postconditions to check the required state before and after the execution of a method in a class.

4.1 Object Design Trade-Offs

Design decisions for the object model required a revision of the previous model and addition of methods and attributes to the classes in order to circumscribe the services of the subsystems. The main trade-offs were along the lines of the main concepts of low-level design such as reuse. Reusing instead of reiterating similar methods was encouraged thorough out the design, while doing so the advantages of polymorphism were also taken into account. Hence, most of the methods like create, and destroy can be applied to the abstract class of GameObject which does not restrict the use of the function to a single object.

Another design trade-off was concerned about using design patterns such as abstraction-occurence. This pattern is used in two locations in the object model; one with the Ribosome class and another with the AttackerProtein class. As these objects are created during the gameplay with multiple amounts but only with different IDs, instead of creating copies, abstract and occurrence classes were implemented.

4.2 Final Object Design

In the following pages, the complete object model is presented. In order to represent it optimally, the complete model is broken down into four UML class diagrams. First diagram shows the extended controller subsystem classes. In the analysis phase, a single controller was doing all the necessary works. However, it is now decided upon to split the workload into different classes according to the category of the controller. In the following diagrams, virus and protein class families are shown which are followed by cell family game components and finally user interface classes.

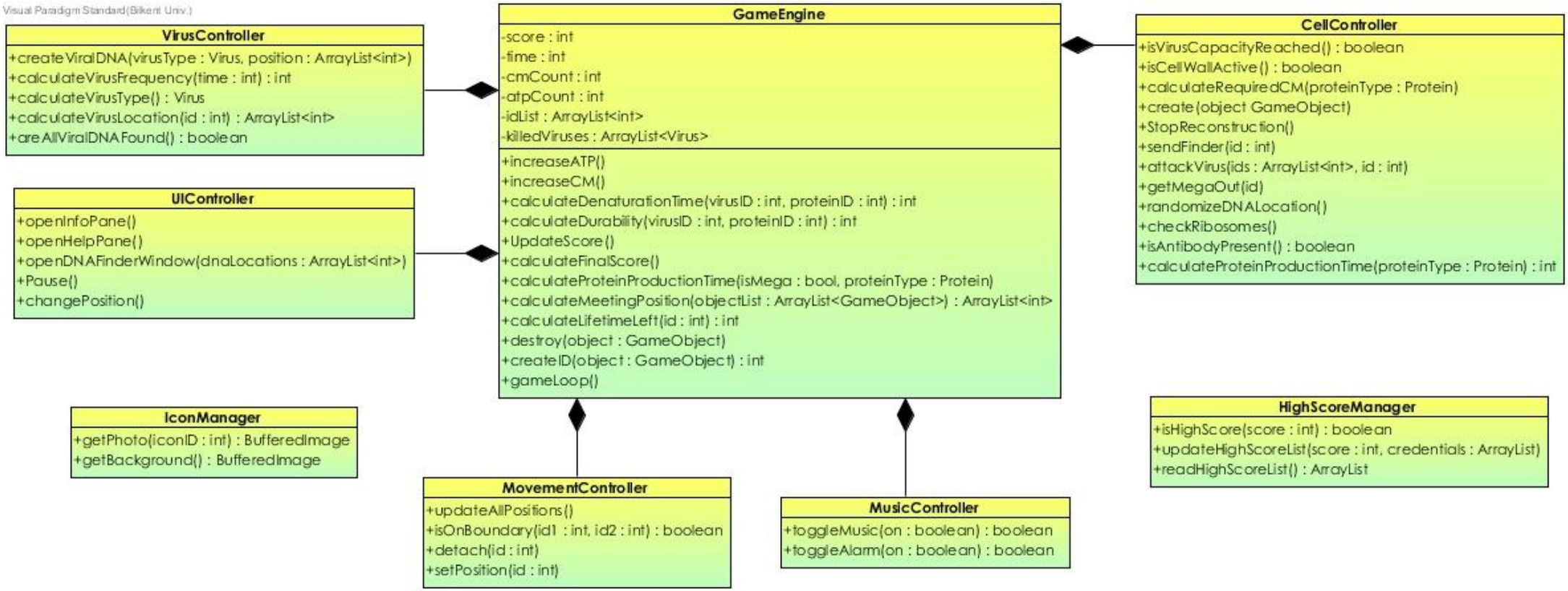


Figure 2 This diagram shows the classes in the game management and high score list subsystems. These controllers are separated in order to provide high coherence. HighScoreManager and IconManager classes belong to the HighScoreList subsystem.

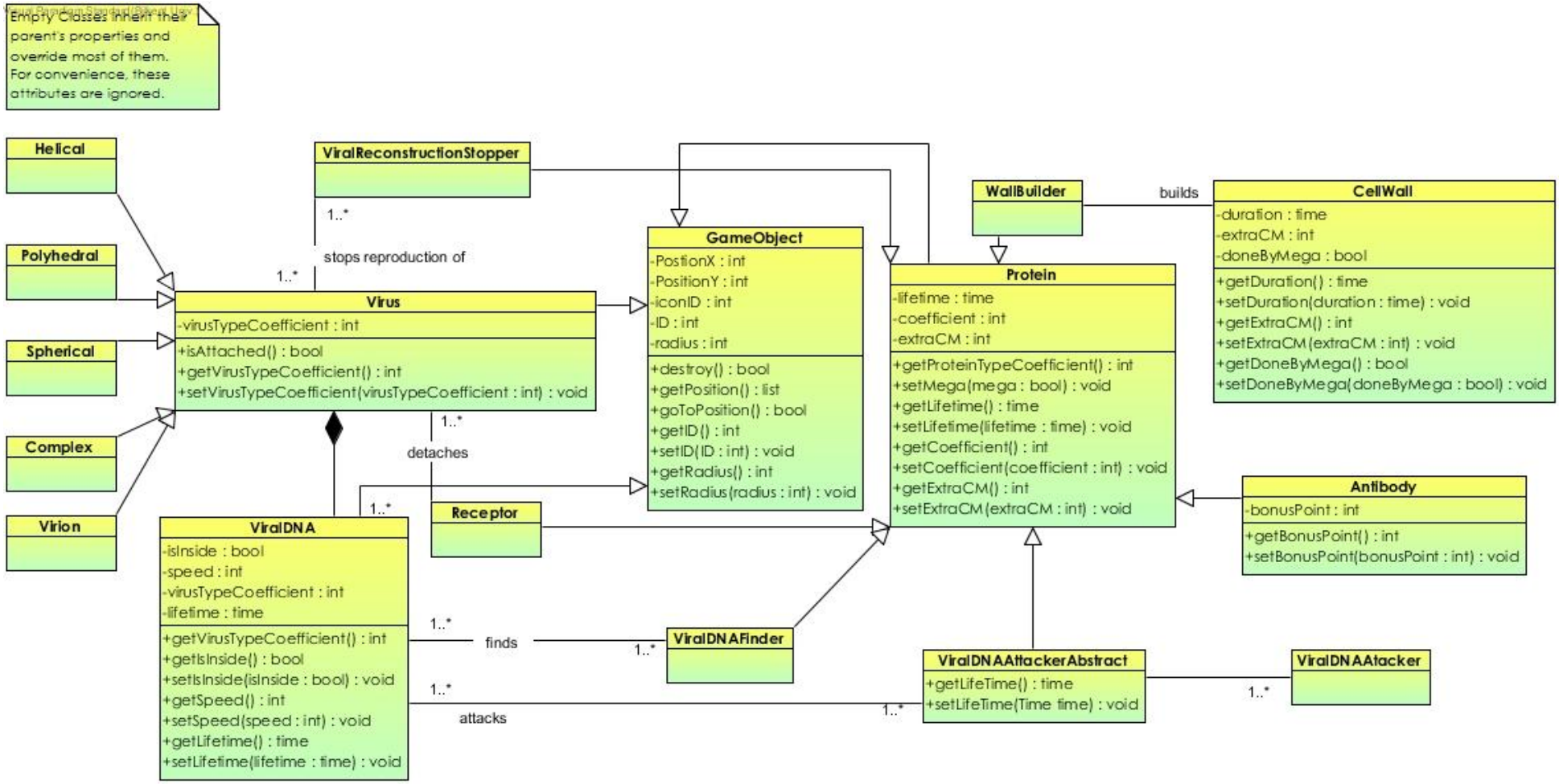


Figure 3 This is the diagram for representing major game objects in the object model. It contains the two main families of classes: virus and protein. Both of these families are inheriting the game object; hence, the game object has a crucial part as an abstract class.

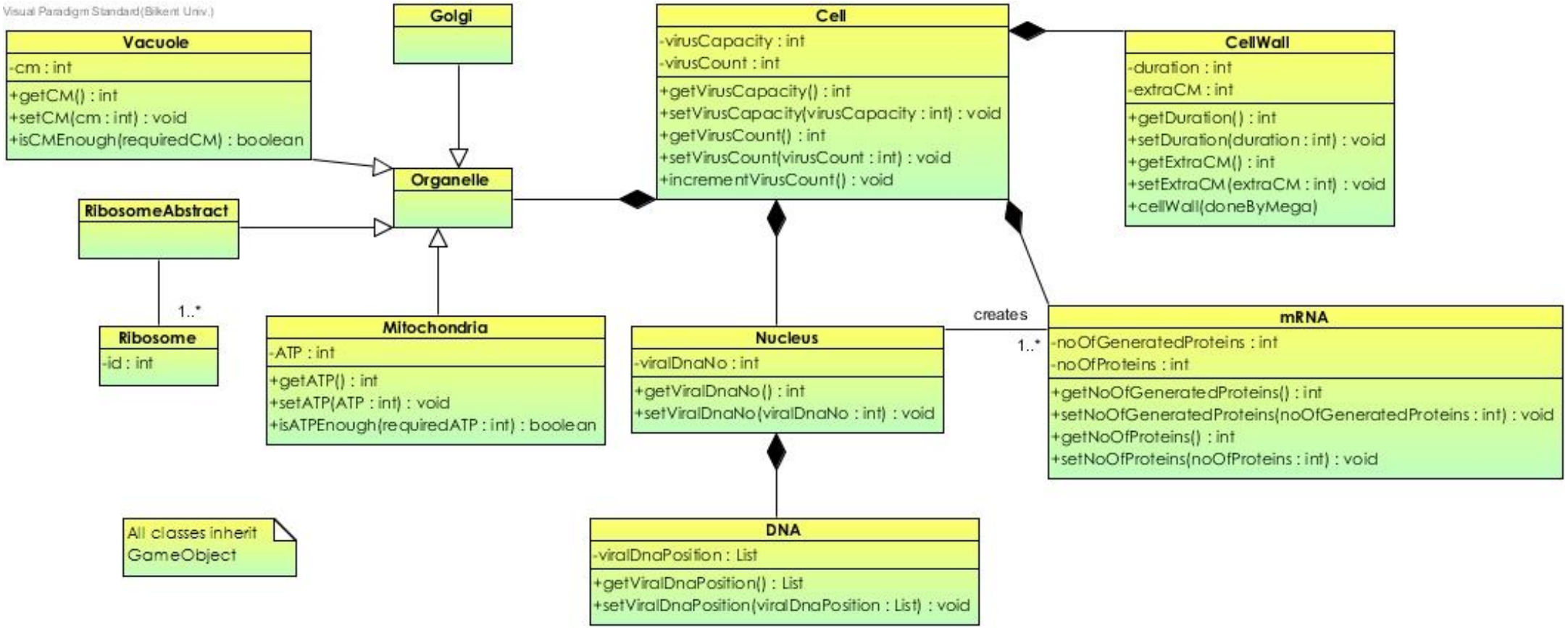


Figure 4 This diagram shows the third game component family: cell. All of these classes also inherit game object as it is also noted in the diagram.

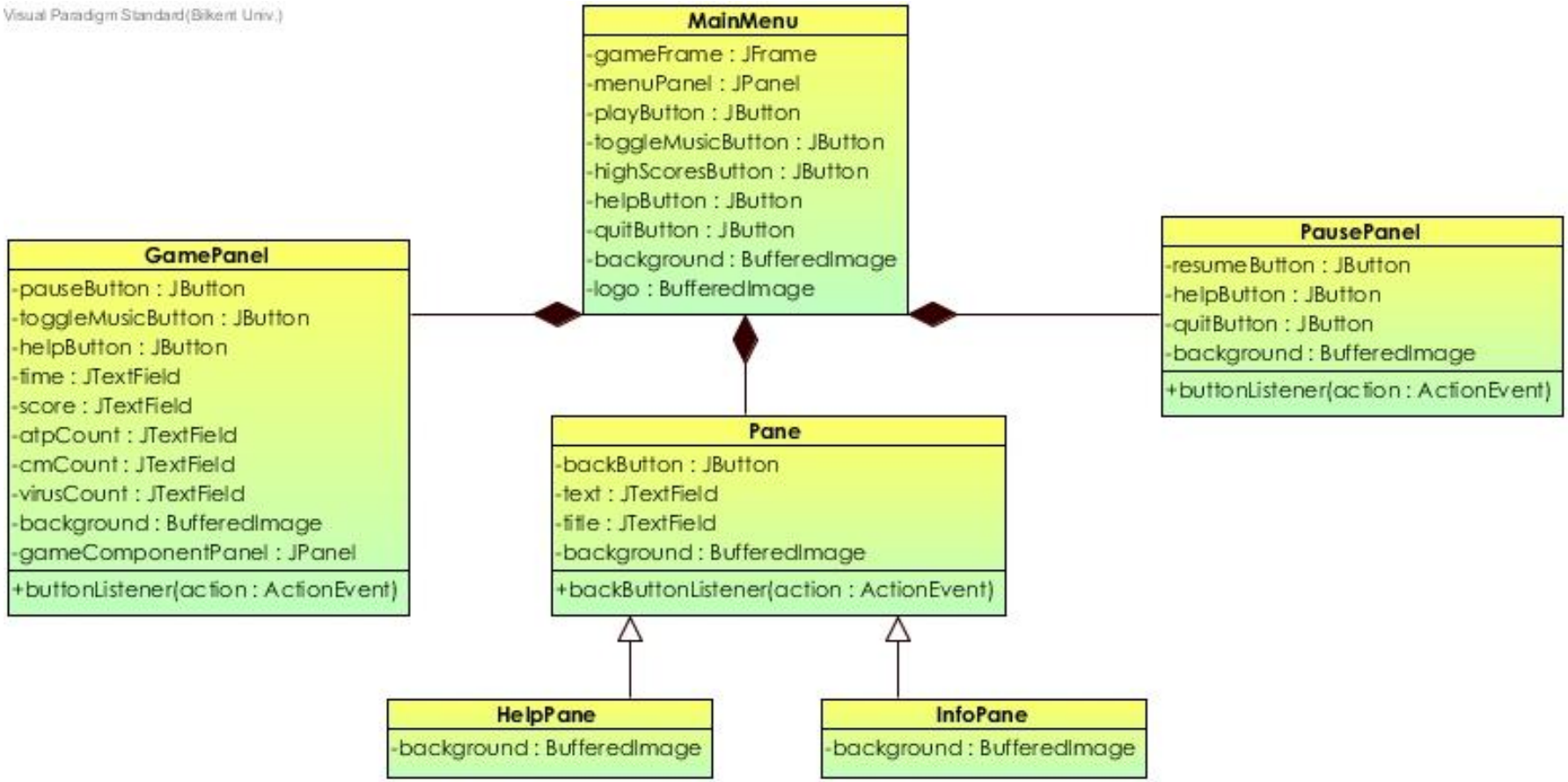


Figure 5 This is the UML diagram for the user interface subsystem's objects.

4.3 Packages

Because packages are used to divide the actual code into partitions, we have decided to categorize the actual code into three different components under the Virion System package. The general categories are Controllers, GameComponents and UserInterface. Its general structure is more like an MVC architecture. However, the architecture that we chose is 3-Layer as it includes a filesystem layer which will be used to persist data on the filesystem. The UML Package diagram can be seen in Figure 6.

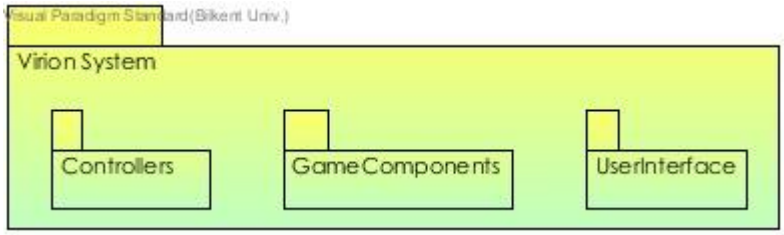
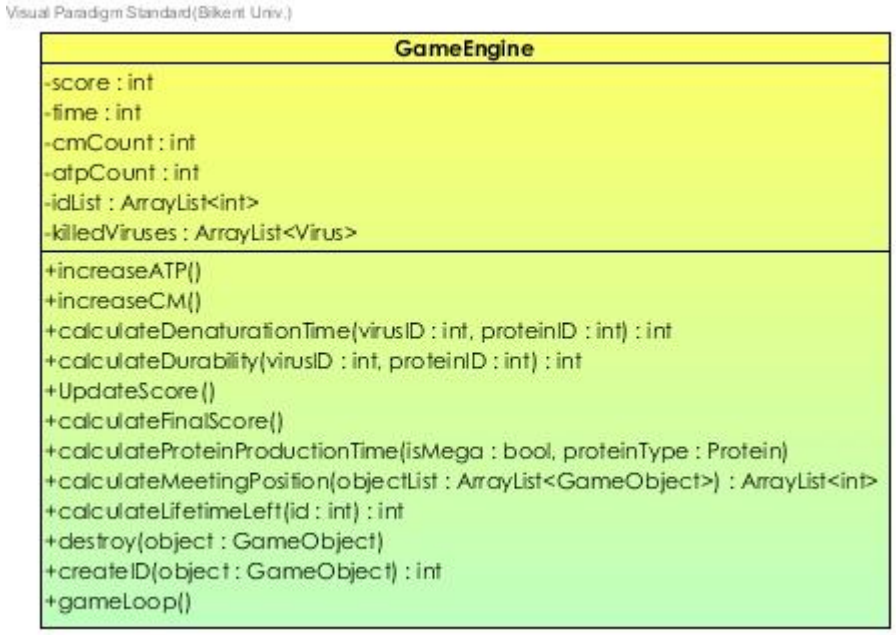


Figure 6 This is the general packaging of the code.

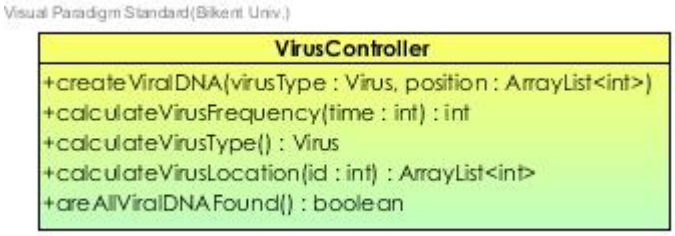
4.4 Class Interfaces

In this section, each class in the final object model will be discussed in detail. Each class is followed by a table explaining its attributes and methods with constraints. The order of the classes follow the same order in the grouping of the object model, which is Controllers, Game Components and User Interface.

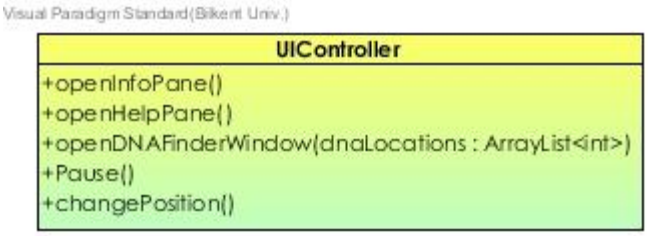


Class Name:	GameEngine
Attributes:	**int score:** Score of the player. **int time:** Time of the whole gameplay. **int cmCount:** Property that holds the amount of CM. **int atpCount:** Property that holds the amount of ATP. **ArrayList<int> idList:** Attribute that holds the IDs of all the GameObjects. ArrayList<Virus> killedViruses: Contains the killed viruses.
Methods:	**public void increaseATP():** Increases the atpCount by 1000 each minute. **public void increaseCM():**Increases the cmCount by 1000 each minute. **public int calculateDenaturationTime(int virusID, int proteinID):** This method calculates the duration of the denaturation of virus and proteins which are taken as inputs. After calculating the times according to the 90*(protein type coefficient) – (virus type coefficient) formula it returns the result as the protein denaturation time. **public int calculateDurability(int virusID, int proteinID):** This method understands the phase of the infection and calculates the durability of the virus according to the formulas: 5*(virus type coefficient) –(protein type coefficient), or 75*(virus type coefficient) –(protein type coefficient), or 20*(virus type coefficient) –(protein type coefficient) depending on the stage of infection. **public void updateScore():** Updates the score according to viruses killed at each timestamp –every half second.

	public void calculateFinalScore(): Calculates the overall score by accessing the virus type coefficients of each virus in the killedVirus list and adds the antibody bonus if there is one. **public int calculateProteinProductionTime(bool isMega, Protein proteinType):** Calculates the production time of the protein according to the formula, 90* (protein type coefficient). **public ArrayList<int> calculateMeetingPosition(ArrayList<GameObject> objectList) :** Gets the current positions of the objects and calculates a meeting position for them. **public int calculateLifetimeLeft(int id):** Subtracts the denaturation time of the protein from the overall lifetime of it and calculates its emaining lifetime. **public destroy(GameObject object) :** Destroys a given object and if it is a virus, adds it to the killed virus list. **public int createID(GameObject object):** Gives an ID to the object by checking the idList. **public void gameLoop():** This is the main method that runs the game. If the game is paused this loop halts execution. Otherwise, it updates the locations, does all the calculations at each timestamp –half a second- and continues until the game is over.
Constraints:	Invariants: • Time and score variables start at zero. • cmCount is 10,000 at the beginning of the game. • atpCount is 1000 at the beginning. Pre-conditions: • In order to execute updateScore, end of the gameLoop should be reached. • The IDs in each parameter should be a valid one. • Killed viruses should not be added to the list more than once. • Destroy and create methods should not destroy an already destroyed object or create an existent object. Post-condition: • calculateLifeTime function should update the remaining life of the object. • At the end of the gameLoop, the score and time values should be valid. • destroy function should add the virus to the list. • Create function should call createID in itself and successfully create an object with a valid ID.



Class Name:	VirusController
Methods:	**public void createViralDNA(virusType : Virus, position : ArrayList<int>)**: Gets type of virus, list of x,y, and creates viral DNA inside the cell. public int calculateVirusFrequency(int time): Computes the virus production frequency. public Virus calculateVirusType(): Sets the type of certain virus. **public ArrayList<int> calculateVirusLocation(int id) :** Gets virus object id and returns its position as list of x,y. **public boolean areAllViralDNAFound():** Returns true if player finds all viral DNAs in the post-infection phase.
Constraints:	Pre-condition: • Virus must be attached to cell for ceratin amount of time before injecting viral DNA inside the cell. Post-condition: • X and Y postion of the created viral DNA should be inside cell radius boundary.

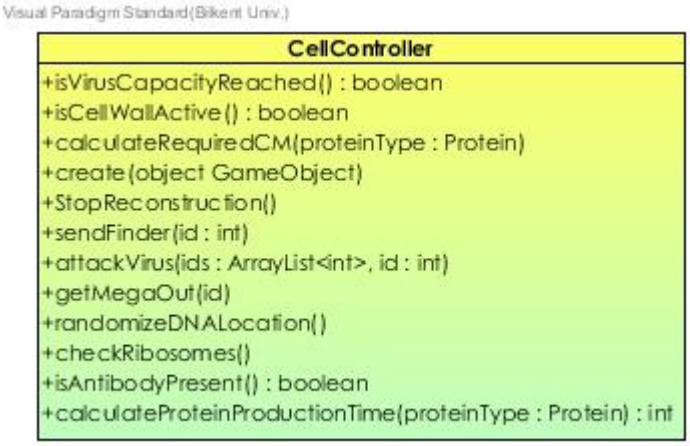


Class Name:	UIController
Methods:	**public void openInfoPane()**This method opens the information screen during the game or on main menu according to the user's processes. **public void openHelpPane():** This method opens the help screen according to the user's processes during game, on main menu or on pause mode. **public openDNAFinderWindow(ArrayList<int> dnaLocations)**This method opens the DNA Finder Window which will be used after viral DNA goes inside the DNA of cell. **public void pause():** This method pauses the game during the play mode. **public void changePosition():** This method changes the position of organelles and the cells according to the processes.
Constraints:	Pre-condition: • In order to execute pause() method, openHelpPane() shouldn't be executed at the same time. • In order to execute openInfoPane() method, pause() method should not be executed.

Class Name:	MovementController
Methods:	**public void updateAllPositions():** Constantly updates positions of viruses, viral DNAs,proteins, and organlles. Viral DNA move towards the nucleus constantly. Proteins and organelles don't stay still but instead stay like floating on water. **public boolean isOnBoundary(int id1, int id2) :** Returns true if one game object is attached to another object. When virus attaches to cell wall, it is on boundary of the cell. **public void detach(int id):** This method is called with virus id when a receptor detaches that virus. **public void setPosition(int id):** Sets the position of an object with id in the parameter.
Constraints:	Pre-condition: • Receptor should be attached to cell wall from inside of cel for certain amount of time before detach() function is called. Post-condition: • Detach() method should change position of virus a little away from cell wall.

Class Name:	MusicController
Methods:	**public boolean toggleMusic(boolean on) :** This method toggles the music of the game that the music can be turned on or off. **public boolean toggleAlarm(boolean on) :** This method toggles the alarm of the game that will be done during postinfection phase.

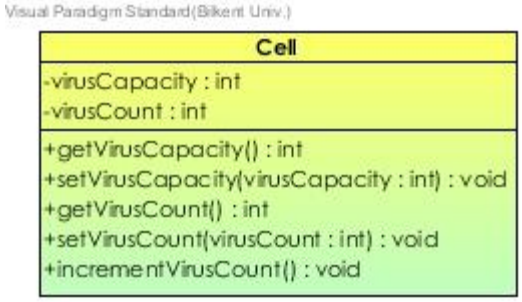
Class Name:	HighScoreManager
Methods:	**public boolean isHighScore(int score) :** Returns true if the score is the highest of all previous scores. **public void updateHighScoreList(int score, ArrayList credentials):** Updates high score list with newly added score and name of the player. public ArrayList readHighScoreList() : Returns the available high score list.
Constraints:	Pre-condition: • Player must enter at least some characters. The input form for name of the player can't be empty. Post-condition: • Highscore is updated successfully on the highscore list in the filesystem.



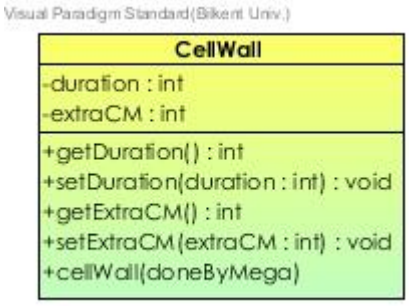
Class Name:	CellController
Methods:	**public boolean isVirusCapacityReached() :** This method checks whether the virus capacity is reached for the cell is reached or not at the end of each game loop. **Public boolean isCellWallActive() :** Checks whether the cell wall is active. **public void calculateRequiredCM(Protein proteinType):** This method returns the required core molecule in integer type for the protein passed in the parameter. **public void create(object GameObject):** This is the most general create function that can create any game object such as proteins or viruses. **public void stopReconstruction():** This method stops the reconstruction of viruses after post-infection. This method is called while there is an active ReconstructionStopper protein. **public void sendFinder(int id):** This method sends the selected viralDNAFinder protein into the nucleus to start the search for the viralDNA. **public void attackVirus(ids : ArrayList<int>, id : int) :** This method is attacks the virus with the proteins in the list. **public void getMegaOut(id):** This method moves the mega protein out of the Golgi. **public void randomizeDNALocation():** This method randomizes the locations of the viralDNA inside the nucleus of the host cell. **public void checkRibosomes():** Checks whether the selected ribosomes are able to produce the protein. **public boolean isAntibodyPresent() :** This method checks whether the user has produced antibody proteins or not. This method will be called by the calculateFinalScore() method in gameEngine to check for bonus points.

	public int calculateProteinProductionTime(Protein proteinType): This method calculates the production time of a protein by ribosome according to the formula of 10*(protein type coefficient).

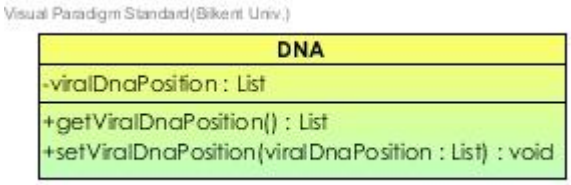
Class Name:	IconManager
Methods:	**public BufferedImage getPhoto(int iconID) :** This method finds the required photo from the filesystem and returns it. **public BufferedImage getBackground() :** This method finds the background image in the filesystem and returns it.
Constraints:	Pre-condition: • The required images should be present in the filesystem. Post-condition: • The image should be returned to the caller.



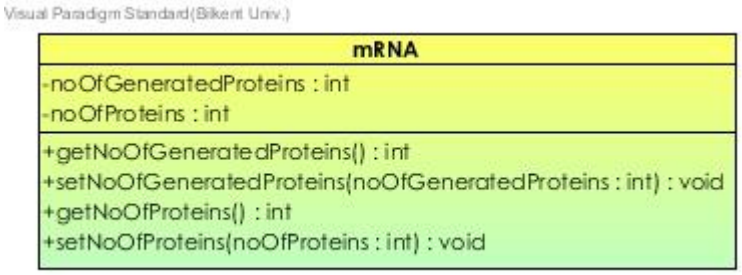
Class Name:	Cell
Attributes:	**int virusCapacity:** This is the attribution which includes the number of virus which each cell can have the most. If the virus number is bigger than this capacity, the cell will die. **int virusCount:** This is the attribution which includes the number of virus a cell has.
Methods:	**public int getVirusCapacity():** This method is the getter method of the attribution virusCapacity. It will return an integer which will be the number of viruses each cell will have at most. **public void setVirusCapacity(virusCapacity**): This method is the setter method of the attribution virusCapacity. With this method, the virus capacity will be setted to wanted number. **public int getVirusCount():** This method is the getter method of the attribution virusCount. It will return an integer which will be the number of viruses a cell has. **public void setVirusCount (virusCount):** This method is the setter method of the attribution virusCount. With this method, the virus count will be setted to wanted number. **public void incrementVirusCount**(): This method is about the interaction between virus class and the cell class. If any virus comes to the cell, the virus count will be incremented by this method.
Constraints:	**Pre-condition:** • For getVirusCount(),incrementVirusCount() should be used at least one. **Post-condition:** • After incrementVirusCount(), getVirusCount returns at least 1.



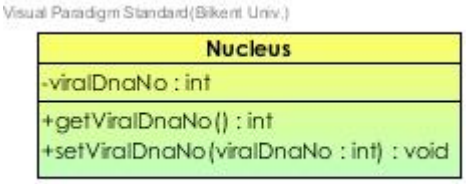
Class Name:	CellWall
Attributes:	**int duration:** This is an attribution that contains the time which the cell wall has to stay for the given seconds. **extraCM:** To produce a cell wall, it should be needed extra 200 Core Molecules. This attribution includes these extra core molecules.
Constructor:	**cellWall(doneByMega: bool):** The constructor of the class which is the cell wall done by mega protein. If the parameter is true, then this constructor is used.
Methods:	**public int getDuration():** This is the getter method for the duration. It returns an integer value which is the duration of the cell wall. **public void setDuration(duration: int):** This is the setter method of the duration attribution. Duration value can be setted to a needed value by this method. **public int getExtraCM():**This is the getter method for the attribution extraCM. It returns integer type value which is the extraCM needed for the cell wall making. **public void setExtraCM(extraCM: int):** This is the setter method for the extraCM attribution of cell wall class. It can be setted to needed value by this method.



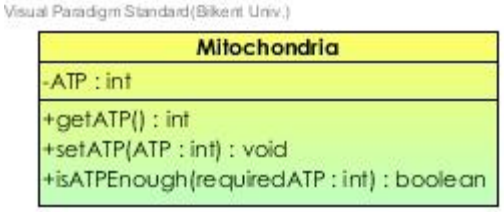
Class Name:	DNA
Attributes:	**ArrayList viralDNAPositions**: This is an attribution which contains the position of the viralDNA in the cell's DNA.
Methods:	**public List getViralDnaPosition():** The getter method of the viralDnaPosition attribution. It returns the position of the viral DNA in cell's DNA. By this method, the position of the viral DNA is known, and during the game user can search for it and destroy it. **public void setViralDnaPosition(viralDnaPosition: ArrayList):** The setter method of the attribution viralDnaPosition. The position of the viral DNA in cell's DNA is setted randomly by this method.



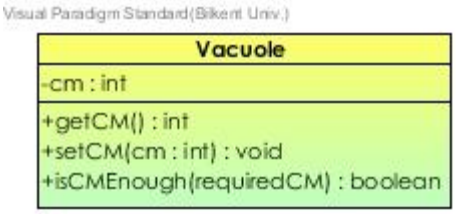
Class Name:	mRNA
Attributes:	**int noOfGeneratedProteins:** This attribution contains the integer value which is the number of proteins done by ribosome and gave mission by nucleus. **int noOfProteins**: The attribution contains the integer value which is the maximum number of proteins can be done.
Methods:	**public int getNoOfGeneratedProteins():**The getter method of the attribution noOfGeneratedProteins. It will return the integer value which is the number of proteins generated. **public void setNoOfGeneratedProteins(noOfGeneratedProteins):** The setter method of the attribution noOfGeneratedProteins. It will be setted to needed value of generated protein number by this method. **public int getNoOfProteins():** The getter method of the attribution noOfProteins. It will return the integer value which is the maximum number of proteins to be generated. **public void setNoOfProteins(noOfProteins):** The setter method of the attribution noOfProteins. It will be setted to needed value of maximum generated protein number by this method.
Constraints:	**Pre-condition:** • For getNoOfGeneratedProteins(), setNoOfProteins(noOfProteins) should be at least setted to 1.



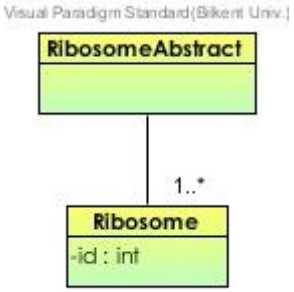
Class Name:	Nucleus
Attributes:	**int viralDNANo:** This is an attribution that contains the number of viral DNAs in nucleus.
Methods:	**public int getViralDnaNo():** This is the getter method of the attribution viralDnaNo. It will return the integer value which is the number of viral DNAs in the nucleus. **public void setViralDnaNo(viralDnaNo):** This is the setter method of the attribution viralDnaNo. The number of viral DNAs in the nucleus can be setted to needed value by this method.



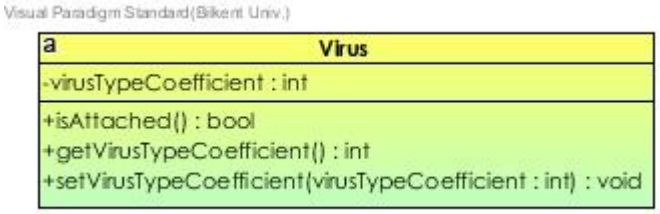
Class Name:	Mitochondria
Attributes:	**int atp:** This is the attribution that contains the number of ATP mitochondria produced.
Methods:	**public int getAtp():** The getter method of the atp attribution. It will return the integer value which is the number of ATPs mithochondria produced. **public void setAtp(atp : int):** The setter method of the atp attribution. The number of ATPs will be setted continuously by this method. **public boolean isAtpEnough(requiredAtp : int):** This method will be used to check whether a process can be done or not by looking the requiredAtp. If the number of required ATP for this process is already produced by mitochondria, it will return true, else false.



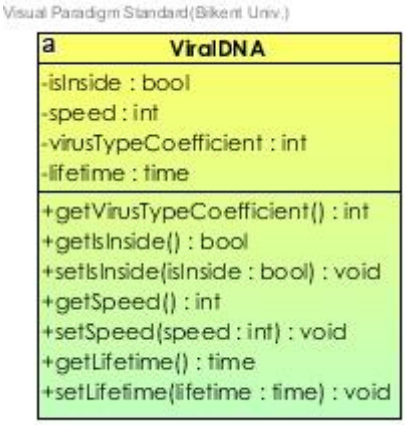
Class Name:	Vacuole
Attributes:	**private int cm:** This integer type attribute is used to make proteins and mRNA's in Protein class. It is the core molecules vacuole produces.
Methods:	**public int getCM():**This is the getter method of cm attribution. It will return integer type of value which is the number of core molecules. **public void setCM(cm : int):** This is the setter method of cm attribution. The number of core molecules can be setted to needed value by this method. **public boolean isCMEnough(requiredCM: int):** By the reference of requiredCM which is the multiplication of 100 and operation of GameController class calculateProteinProductionTime(proteinType), this method checks whether there is enough core molecule for required one or not and returns a boolean type .
Note:	In all organelle classes, there are all attributions, methods and constructors which game object abstract class has.



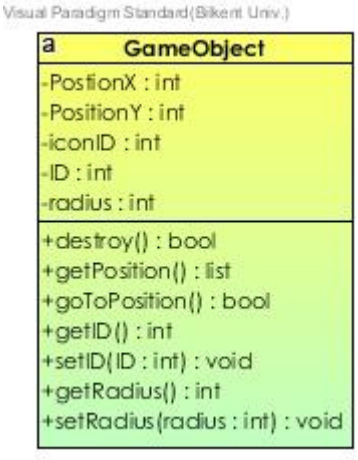
Class Name:	Ribosome & RibosomeAbstract
Attributes:	**int id: This is the ID of the ribosome copy given by the game engine.**
Methods:	All methods are inhereted from organelle abstract class
Note:	This structure is using the abstraction-occurrence design pattern.



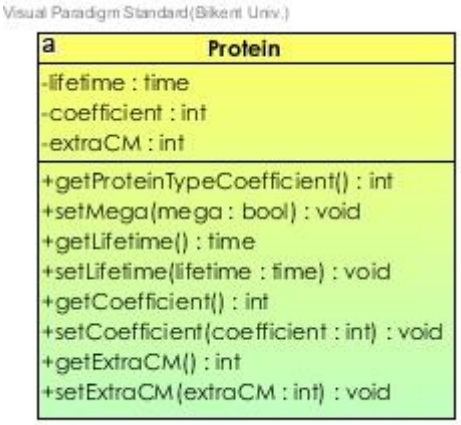
Class Name:	Virus
Attributes:	**int virusTypeCoefficient**: Attribute that holds virus's type coefficient value as there many types of viruses.
Methods:	**bool isAttached()**: Boolean function that returns true if virus is attached to cell wall and false otherwise.



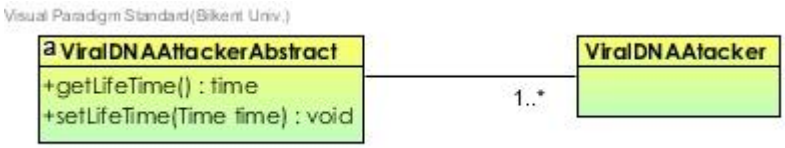
Class Name:	ViralDNA
Attributes:	**bool isInside**: Attribute that is set true if viral DNA is inside the cell. **int lifetime**: Important attribute of viral DNA that's decreased by an amount anytime it is attacked by proteins. When it's equal to zero viral DNA gets destroyed. **int virusTypeCoefficient:** This is the same as the attribute in Virus. **int lifetime:** This is the same attribute as the one in Virus.
Methods:	Basic getter and setter methods.
Constraints:	**Invariants:** • Speed should be bigger than zero. • virusTypeCoefficient and lifetime should be copied from virus upon instantiation.



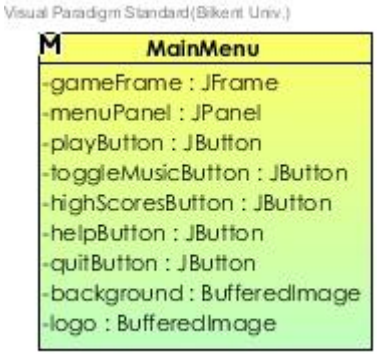
Class Name:	GameObject
Attributes:	**int positionX:** This attribute holds the x-coordinate position of the game object. **int positionY:** This attribute holds the y-coordinate position of the game object. **int ID:** This is the given ID of the object by the controller for ease of control. **int iconID:** Each object of the game look unique and each have unique id numbers of images located in image folder. **int radius:** This attribute sets radius value of each object. This will be helpful when handling collisions, attachments, and attacks.
Methods:	**public bool destroy():** This method destroys the given object when no longer needed. **public bool goToPosition():** This method assigns a position to the gameObject and it moves there using a Timer inside which provieds a smoothe move. ***Other methods are basic getters and setters.**



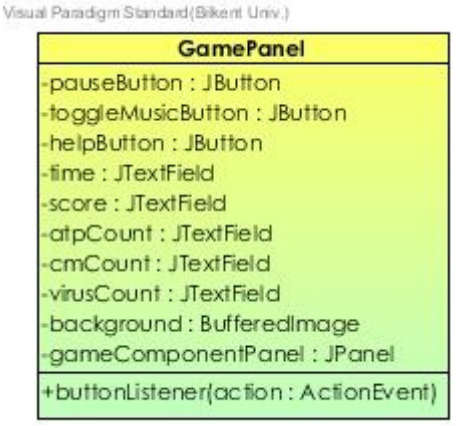
Class Name:	Protein
Attributes:	**int extraCM:** Core molucules required to produce proteins. **int lifetime:** Proteins, enemies of viruses, similarly have life time that decreases during fight with viruses. **int coefficient:** Similar to viruses, different types of proteins have different coefficients.
Methods:	Basic setter and getter methods.
Constraints:	Pre-condition: • Lifetime and coefficient of the protein is set during creation.



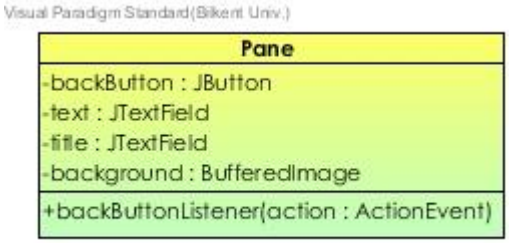
Class Name:	ViralDNAAttacker & ViralDNAAttackerAbstract
Attributes:	Attributes of the protein class are inherited.
Methods:	Basic setter and getter methods.
Note:	This structure is using the abstraction-occurrence design pattern.



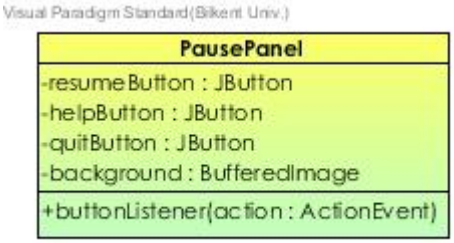
Class Name:	MainMenu
Attributes:	**JFrame gameFrame:** Frame for the game which all game components will be shown inside it. **JPanel menuPanel:** menuPanel shows the menu on the screen to the user. **JButton playButton:** This is the button for play that user should click for start to play. **JButton toggleMusicButton:** This is the attribution for button which will be clicked for opening or closing the music at the main menu screen. **JButton highScoresButton:** This is the high score button which will be clicked for showing the high scores. **JButton helpButton:** helpButton is the attribution which is a JButton that will be clicked to show the help screen. **JButton quitButton:** This is the attribution which has JButton type and clicked for quitting the game. **BufferedImaged background:** This is the background of the main menu screen. **BufferedImaged logo:** This is the logo of the game which has BufferedImaged type.



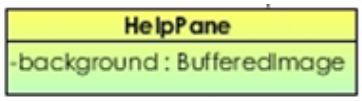
Class Name:	GamePanel
Attributes:	**JButton pauseButton:** This is the attribution of GamePanel class which is the button of the game which will be clicked to pause the game. **JButton toggleMusicButton:** This is the attribution for button which will be clicked for opening or closing the music during the game. **JButton helpButton:** This is the attribution for button which will be clicked to show the help screen during the game. **JTextField time:** This attribution is the time passed during the game. **JTextField score:** This attribution is the score of the user which is incremented during the game continuously. **JTextField atpCount:** This is the number of ATP which the user has during the game and it will be changed according to the processes user does and mithochondria's production of ATP. **JTextField cmCount:** This is the number of core molecules which the user has during the game and it will be changed according to the processes user does and vacuole's production of core molecule. **JTextField virusCount:** This is the number of viruses which is inside the cell. **BufferedImage background:** This is the background of the game screen. **JPanel gameComponentPanel:** This is the main panel where the gameplay occurs.
Methods:	**public void buttonListener(ActionEvent action):** This method gets the input from the button that is the clicks which user does.



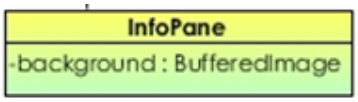
Class Name:	Pane
Attributes:	**JButton backButton:** This is the button which user will clicks on it to turn back to previous screen. **JTextField text:** This is the text field that shows the needed text. **JTextField title:** This is the text field that shows the needed title. **BufferedImage background:** This is the background of the needed screen.
Methods:	**public void backButtonListener(ActionEvent action):** This method gets the input from the back button that is the clicks which user does.



Class Name:	PausePanel
Attributes:	**JButton resumeButton:** This is button that user will click on to resume the game during the pause mode. **JButton helpButton:** This is the button that user will click on to show the help screen during the pause mode. **JButton quitButton:** This is the button that user will click on to quit the game during the pause mode. **BufferedImage background:** This is the background of the pause screen.
Methods:	**public void buttonListener(ActionEvent action):** This method gets the input from the button that is the clicks which user does.



Class Name:	HelpPanel
Attributes:	**BufferedImage background:** This is the background of the help screen.



Class Name:	InfoPanel
Attributes:	**BufferedImage background:** This is the background of the information screen.