CSC 413 Project Documentation

Fall 2018

Cory Lewis

ID: 917359162

CSC413.01

Tank Game Repo: https://github.com/csc413-01-fa18/csc413-tankgame-mecharmor

Second Game Repo: https://github.com/csc413-01-fa18/csc413-secondgame-mecharmor

Table of Contents

1 Introduction

1.1 Project Overview

The term project consisted of two games. The Tank Game, and the Second Game (Galactic Mail) which are basic 2D arcade games that allow the player to move around and attack or collect any given set of objects. The Tank Game is multiplayer and allows two people to control their own tanks which they can use to attack each other. The Galactic Mail Game is a single player game that is a space ship which needs to deliver mail to the given set of moons, this game has multiple levels and can store your high score.

1.2 Introduction of the Tank Game

The Tank Game is a single leveled game that can be controlled by two people. The Goal of the game is for one tank to destroy the other. Extra Lives can be attained from running over red packs. Tanks are confined to the size of the map but are free to move around within those bounds. The map is generated by a .csv file which allows the game to populate objects at specific locations given certain numbers used for reflection during object generation.

1.3 Introduction of the Second Game (Galactic Mail)

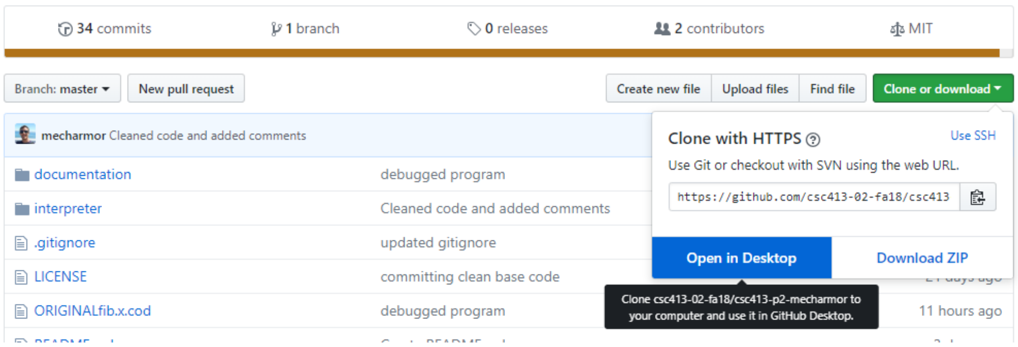
Galactic Mail is a single player game consisting of many levels that gradually get more challenging as the player progresses. The player earns points when landing on moons and delivering mail, but be careful because while on the moon you can lose points very quickly for not launching off of the planet (spacebar). This game contains background music and allows for the option to replay the game without having to relaunch the .jar.

2 Development Environment

a. Version of Java Used: jre-10.0.2
b. IDE Used: IntelliJ IDEA 2018.2.3
c. Main Java Libraries: awt, swing

3 How to Build/Import your Project

1) intelliJ IDE download link: https://www.jetbrains.com/idea/
2) GitHub desktop download link: https://desktop.github.com/
3) Clone Repository through GitHub desktop
    a. Navigate to this link (if you have permissions to view):
        i. (Tank Game Link) https://github.com/csc413-01-fa18/csc413-tankgame-mecharmor
    b. (Galactic Mail Link) https://github.com/csc413-01-fa18/csc413-secondgame-mecharmor
    c. Click "Clone or download"
    d. Choose "Open in Desktop" (Photo illustration below)



mecharmor Cleaned code and added comments

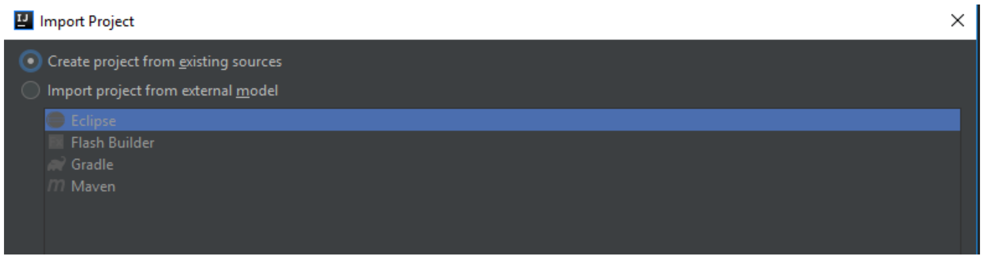
documentation	debugged program	
interpreter	Cleaned code and added comments	
.gitignore	updated gitignore	
LICENSE	committing clean base code	
ORIGINALfib.x.cod	debugged program	11 hours ago

Use Git or checkout with SVN using the web URL.

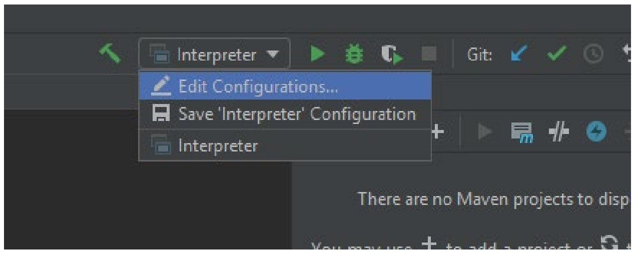
https://github.com/csc413-02-fa18/csc413

Clone csc413-02-fa18/csc413-p2-mecharmor to your computer and use it in GitHub Desktop.

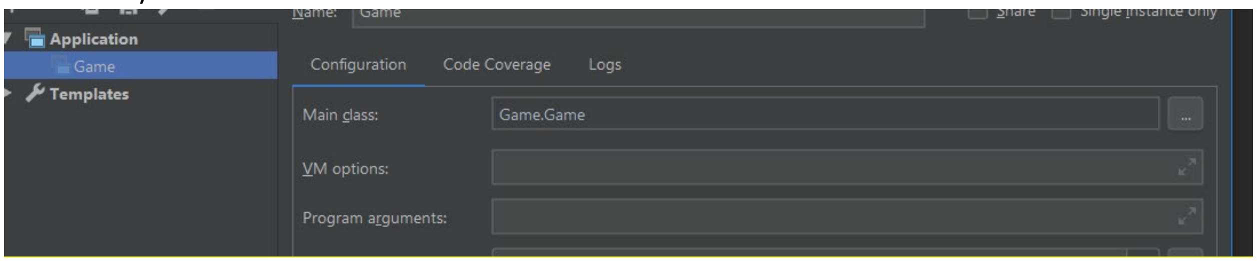
4) Once your repository is cloned you can launch IntelliJ and import using "existing sources".
5) When importing select: "Create project from existing sources" (photo illustration below)


Import Project
Create project from existing sources
Import project from external model
Eclipse
Flash Builder
Gradle
Maven

6) Once the project is imported you will need to “edit configuration” on the top right of the screen


Interpreter
Git:
Edit Configurations...
Save 'Interpreter' Configuration
Interpreter
There are no Maven projects to disp

7) Make sure your “Main class” is located at: Game.Game



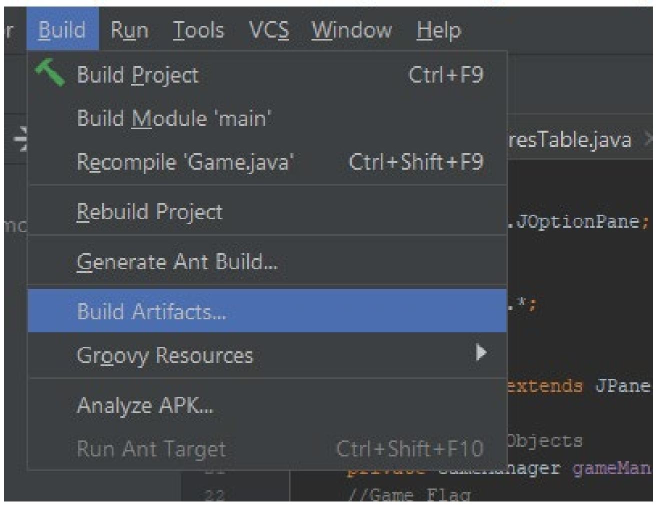
Name: Game
Main class: Game.Game
VM options:
Program arguments:

8) After that you are ready to Build!

4 How to Run your Project

Running the project within intelliJ:

1. Building project into a jar:
   a. Navigate to “Build” then select “Build Artifacts”, you will see a popup in the center of the screen that prompts you to build. “press build”. The project will build after a few seconds and the compiled jar will be located in the directory: csc413-secondgame-mecharmor\GalacticMail\out\artifacts\main_jar



   b. Copy the jar into the jar folder located in the GitHub project and replace the old jar. You are ready to go!
2. Running the Jar

a. In the fetched files from GitHub there is a file named 'Jar'. Go inside that folder and double click the 'main.jar'. Note: This jar relies on the 'resources' folder which is in the same directory as main.jar. If you copy the jar outside the main directory, the game will not launch!

5 Rules and Controls

- **Tank Game**
  - b. **Rules**:
    - i. Break Destructible Walls to get past obstacles. Collide with Red health boxes to get an extra life. Objective of the game is to destroy the other tank (Ex| Player 2).
  - c. **Controls**
    - i. Player 1
      1. W-**Forwards**, A-**LEFT**, S-**Backwards**, D-**Right**, E-**Shoot**
    - ii. Player 2
      1. Up Arrow–**Forwards**, Left Arrow–**Left**, Down Arrow–**Backwards**, Right Arrow–**Right**, Enter Key-**Shoot**
- **Galactic Mail**
  - d. **Rules**:
    - i. Land on moons to gain points and deliver mail to the residents of that planet. Green checkmarks denote which planets have had mail delivered to them. Once mail is delivered to the next planet you will be taken to the next level. If your score is high enough, you will get on the High-Score list! If an asteroid hit you then the game is over (but you might still have a chance to make it onto the high scores list). Avoid Asteroids at all costs!
  - e. **Controls**:
    - i. Player (Ship)
    - ii. W-**Forwards**, A-**LEFT**, D-**Right**, **Spacebar**–Launch from planet while docked

6 Assumption Made When designing and implementing Game(s)

Tank Game:

When trying to plan for the general structure of this program I knew I needed Interfaces: Collidable, Observable, and Drawable which can be used to help identify which objects have functions pertaining to their purpose.

I had to write my program structure down on roughly 13 pages, so I can plan properly and avoid any issues with classes having low Cohesion and high Coupling. During the development of the Tank Game, the main focuses were: getting objects to paint properly, dynamically generating objects from a .csv file, managing collisions properly, and optimizing program flow for best overall performance.

Second Game:

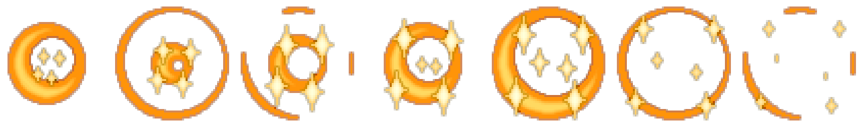
When designing the Galactic Mail game. I knew the overall design was going to be the same but what became more of a challenge is the 'next level' scenarios where Game Objects would need to be dealt with when a new map is generated.

To meet the requirements of 'delivering mail to moons' I had to create an interface called 'Flag' which essentially would allow me to give certain game objects the ability to 'throwFlag' when a ship collides with them. This proved handy because inside my GameManager.java class I was easily able to tally up all the thrown flags and compare them with the total amount of flags that can be thrown for that specific level.

During the thought process for designing the game. I decided I wanted to incorporate Animations and Music. After a few drawings I managed to determine that there are two specific types of animations pertaining to 2D games such as this. “State Animations” and “Overlay Animations”. The state animations are simply a slight change in the image for a specific game object (this is easy). The Overlay Animations are more challenging because they usually have many components to them and need to be lightweight, so they run smoothly. I noted down that I would need to create an Animation class that *extends* Game Object and *implements* the Drawable interface. This worked beautifully because I was able to use my Animation Table class to invoke object creations of type Animation at any point on the screen with any array of images at any speed with 1 line of code.

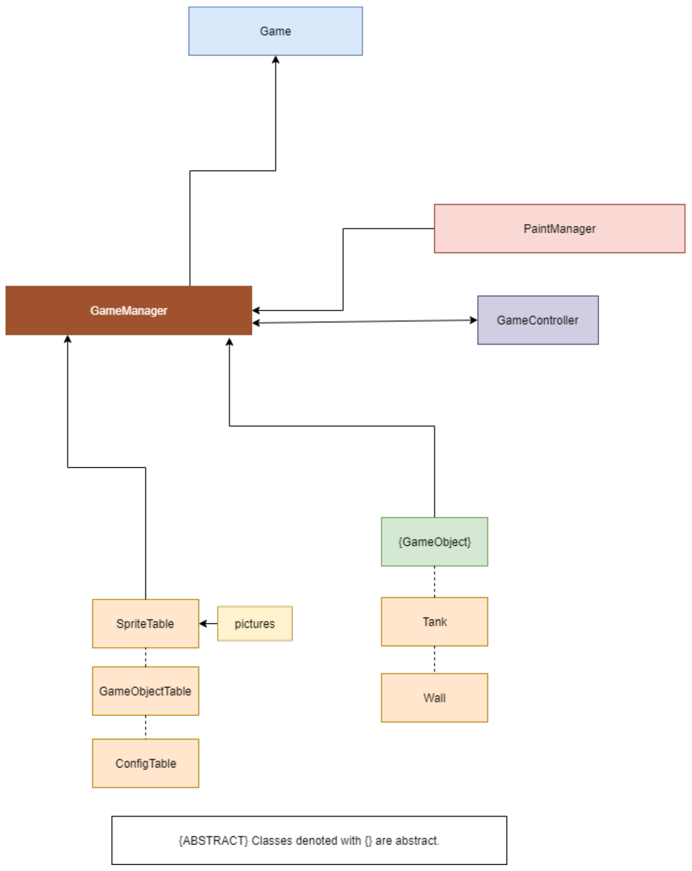
I knew animations were going to be challenging so instead of loading images of type .gif, I decided to create a long strip of images and splice them based on a naming convention of: “NameOfPic_Frames”. The frames would determine how much I would splice by. Img.getWidth()/frames would be what I used to get subImages of those x values.

Ex| Photo: “Explosion_7”

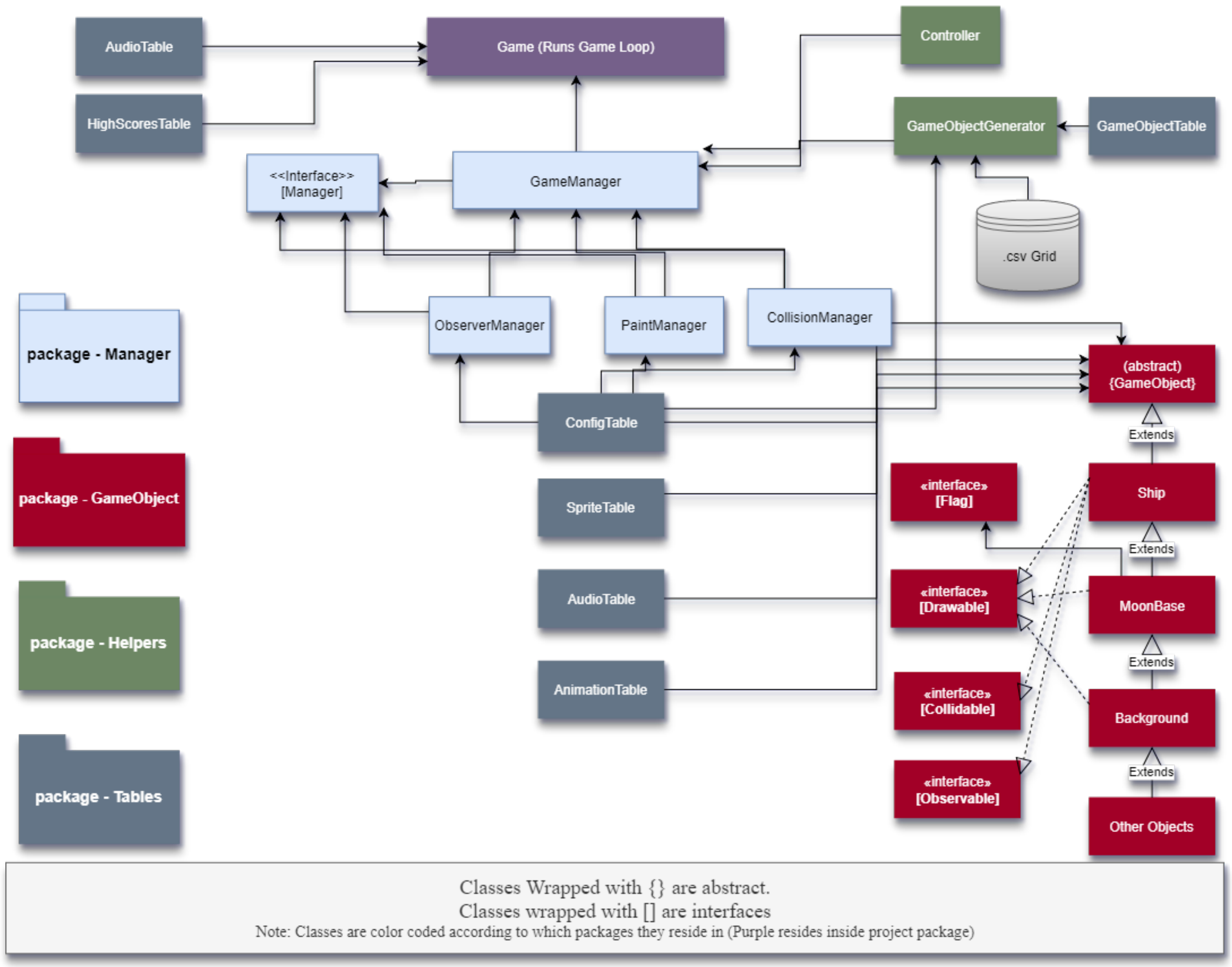


My previous design with the tank game contained many Tables that held my data. This was nice for performance, but I noticed every time I added a new resource, I had to manually type the path and file name to get the loading setup properly. I decided this would need to be done in a dynamic way.

7 Tank Game Class Diagram



8 Second Game Class Diagram (Galactic Mail)



9 Class descriptions of all classes shared among both Games

Before I dig into the nitty gritty of what each class does. I want to explain the **general flow** of both games. Run Program → Game Class initializes ALL tables → Game Manager is created and map is generated via GameObjectGenerator → GameObject children will invoke their super() constructor that builds basic properties for itself → The child GameObjects will publish themselves to the Game Manager → The Game Manager populates arrays in the following Lists inside: “CollissionManager”, “ObserverManager”, and “PaintManager”→The Game loop will continue to run and invoke methods in GameManager → the Invoked methods in GameManager will relay the call to one of the other managers to execute their specific function. → Game Loop continues until gameOver Boolean becomes true, inside Game.

- Game
  - This class runs the main game loop, initializes all the Tables, and Overrides the paintComponent method which paints all the Game Objects on the screen
- MapLoader aka (GameObjectGenerator) “Name was changed for better readability”
  - This class generates objects given a specific csv file or when requested to spawn an object and certain x, and y coordinates.
- PaintManager
  - This class’s sole purpose is to paint all objects of type “Drawable” to the screen. The paint method it contains loops though the known Drawables and invokes their own paint method.
- GameObject {abstract}

    - This class is used as a way of standardizing classes that extend this. It contains all common attributes shared among it's child classes. This helps reduce redundant code since every game object has common physical properties such as: Coordinates, rectangle hitbox, move methods that control movement of the object, and many more.
    - Methods inside GameObject can be Overwritten if a child class needs a permutation of that ability. A good example of this is the moveForward() method which moves specific quantities when called. The Asteroid class in (Galactic Mail) needs to randomly assign speeds so I overwrote this method for simplicity.
- Collidable <<Interface>>
    - This interface is a guarantee that the following methods will be implemented:
        - Void handleCollision(Collidable obj);
            - Used when two objects Rectangles collide.
        - Boolean isCollidable();
        - Rectangle getBounds();
            - Returns hit box for this specific instance of a Collidable
    - Used to determine if the specific object is collidable or not.
- Drawable <<Interface>>
    - This interface is a guarantee that the following methods will be implemented:
        - Void drawImage(Graphics g);
            - Used to paint the implementing object onto the main graphic of the game.
        - Boolean isDrawable();
    - Used to determine if the specific object is Drawable or not.
- Observable <<Interface>>
    - This interface is a guarantee that the following methods will be implemented:
        - Void update();
            - Used for moving object during keypress events
        - Void publish();
            - This method is used to push information to it's child objects for synchronization purposes.
        - Boolean GameOver();
            - Triggers game over sequence that stops game loop.
    - Note: Observable objects can trigger game over state which stops the game loop.
- ConfigTable (static class)
    - This table is designed to live during the lifetime of the program
        - Void Init()
            - Populates hash map with values used throughout the program
        - Int getConfig(String key)
            - Returns a specific configuration contained within this objects hashMap.
- SpriteTable (static class)
    - This table is used to read all BufferedImages in a specific folder and store them inside a hash map to be used throughout the program.
        - Void init()
            - Populates hash map with BufferedImages read from file at a constant path.
        - BufferedImage getBufferedImage(String key)
            - Returns a buffered image given a key (name of image) that is stored in the hash map.
- GameObjectTable → (GameObjectReflectionTable) "name changed for clarity"
    - This table contains a hash map with number keys that reference specific names of GameObject children. This table is ONLY used in the MapLoader (GameObjectGenerator) for reflection when abstract objects need to be instantiated.

        - String getObject(String key)
            - Returns a GameObject child class name given a specific number.
    - Note: Numbers were used for reflection because MapLoader (GameObjectGenerator) loads from a csv file populated with numbers.
- TankControl → (GameController) "renamed to increase abstraction so ANY game object can be passed in"
    - This class is used to connect ANY Game Object to a key listener.
    - This class implements many methods in KeyListener and is used to invoke move methods inside ANY Game Object that is coupled with this class.

10 Class Descriptions of classes specific to Tank Game

- Package
    - Helpers
        - GameManager *(This class is not specific to the tank game but I wanted to note its purpose)*
            - The game manager is used with strong coupling between itself and instantiated GameObjects, this helps game objects publish themselves to the game manager. (This caused low cohesion which looked like spaghetti code) "this was changed in the second game"
    - GameObject
        - Bullet (extends GameObject implements Collidable, Drawable)
            - Designed to simply move forward and destroy itself upon impact with any object
        - DestructibleWall (extends GameObject implements Collidable, Drawable)
            - This class is designed to destroy itself once it's "health" reaches zero after so many bullet collisions.
        - HealthBar (extends GameObject implements Drawable)
            - Used to keep track of health and display a red rectangle given a certain amount of health
            - *Used mainly as a child class of other Game Objects*
        - Lives (extends GameObject implements Drawable)
            - This class is designed to draw a simple circle given a certain offset.
            - *Used mainly as a child class of other Game Objects*
        - LivesBoost (extends GameObject implements Drawable, Collidable)
            - This class is designed to destroy itself upon impact with any object
        - Tank (extends GameObject implements Drawable, Collidable, and Observable)
            - This class is used as the main component of the game.
            - It has a series of child objects that determine it's lifespan (Healthbar, and Lives)
            - Void handleCollision(Collidable obj)
                - If bullet
                    - Lose health
                - Else If LivesBoost
                    - Increase Array List of lives by 1.

11 Class Descriptions of classes specific to Second Game

- Package
    - GameObject
        - Flag <<Interface>>
            - Void throwFlag()
                - This method is used to notify the Game Manager that a flag has been thrown.
        - Animation (extends GameObject implements Drawable)

            - Given any set of coordinates, animation name, and duration (in milliseconds). This class can iterate though an ArrayList with a timer to display a new image thus ‘animation’.
        - Asteroid (extends GameObject implements Drawable, Collidable)
            - Used to move around in a random direction and destroy itself upon collision with an instanceof Ship.
        - MoonBase (extends GameObject implements Drawable, Collidable, Flag)
            - This class is mainly used to throw flags once it collides with an instance of Ship.
            - Upon collision with an instance of Ship, this class will throw a flag and change the state of it’s image so it displays a green checkmark
        - Ship (extends GameObject implements Drawable, Collidable, Observable)
            - This class is a critical component to the program because it can notify the Game Manager when the game is over and has many different actions upon collision with specific Game Objects.
            - Void launchRocket()
                - Runs a timer that programmatically moves the space ship forward for a specific amount of time.
                - The timer invokes the method: “launchRocketHelper()”
            - Void launchRocketHelper()
                - Used to stop the timer once a specific amount of iterations is met.
            - Keypress toggle methods are overwritten so the default object image can be changed given a certain keypress
    - Helpers
        - Score *(used by HighScoresTable to manage score data)*
            - String getName()
                - Returns the name that is saved in this object
            - Int getScore()
                - Returns the score that is saved in this object
    - Manager
        - Manager <<Interface>>
            - Void stop()
                - Used to forcefully stop all actions of the managers (used when next level is loading)
        - CollisionManager (implements Manager)
            - Void checkCollisions()
                - Loops through all Collidable objects and determines if their rectangles intersect
                - Void throwFlag()
                    - Increases flags thrown counter. (used by Game Manager to determine when next level needs to be loaded)
                - Int getThrownFlags()
                    - Returns the amount of flags that have been thrown
                - Void syncCachedObjects()
                    - Merges cached collidable objects to the main colidables arraylist.
                - Void addCollidable(Collidable cb)
                    - Adds a colidable object to the collidablesCache arraylist to be later merged by (syncCachedObjects()).
        - GameManager (different from the Tank Games implementation because it delegates tasks to the other managers instead of runs everything by itself [high cohesion])
            - Void setGameOver(Boolean temp)
                - Sets game over status inside the observer manager class

        - Void updateObservers()
            - Triggers updateObservers method inside observerManager
        - Void addObservable(Observable obs)
            - adds observable to observerManager
        - void addObserverControls(GameController gc)
            - adds a game controller object to the observers control array list.
        - Boolean getGameOver()
            - Returns the game over Boolean contained within observer manager
        - Void addDrawable(Drawable dr)
            - Adds drawable object in paintManager
        - Void paint(Graphics g)
            - Invokes the paint method in paintManager
        - Void throwFlag()
            - Invokes the throwFlag() method in collisionManager
        - Void addCollidable(Collidable cb)
            - Adds collidable object to collisionManager
        - Void checkCollisoins()
            - Invokes checkCollisions() in collisionManager
        - Boolean allFlagsThrown()
            - Compares total flags in level with the amount of flags thrown from collisionManager. Returns true if the level has been completed
        - Void syncCachedObjects()
            - Invokes the syncCachedObjects() method in the other managers
        - Void stop()
            - Invokes the stop() method in the other managers
    - ObserverManager (implements Manager)
        - Void stop()
            - Iterates through all Observers and invokes their stop methods
        - Void updateObservers()
            - Iterates through all observables and checks if the game over status has been thrown.
            - Invokes each observers update method
    - PaintManager (implements Manager)
        - Void stop()
            - Iterates through all Drawables and invokes their sto methods
        - Void paint(Graphics g)
            - Loops through all drawables and invokes their drawImage method and passes a reference to the main games graphics object.
        - Void syncCachedObjects()
            - Merges all objects in drawablesCache with drawables arraylist
- Tables
    - AnimationTable
        - Void init()
            - Loops through all files in the directory “resources/animations/” and populates the hashmap with arraylist of images split by the splitImage method.
        - ArrayList<BufferedImage> splitImage(BufferedImage img, String name)
            - This method parses the name string and determines how many frames are in the buffered image.

        - The getSubImage() method is called and pieces of the original image are stored in an arraylist that will be returned after all the frames have been added
    - Void playAnimation(int x, int y, String name, int timeOfAnimationInMs)
        - Creates new instance of type Animation and adds it to the drawable arraylist.
- AudioTable
    - Void init()
        - This method reads all audio files in the path “resources/sounds/” and populates a hash map of type AudioInputStream.
    - Void playContinuously(String key)
        - Reads AudioInputStream from the audioTable hash map and converts it into a clip object that is ran continuously
    - Void stopALLContinuous()
        - Iterates through all playing music and invokes it’s stop() method.
    - Void playOnce(String key)
        - Retrieves AudioInputStream object and plays that sound one time.
- HighScoresTable
    - Void init()
        - Reads text file at path “resources/highscores.txt” and splits each line with the ‘:’ delimiter to store into an arraylist of scores of type score.
        - During file read, the lowestScore variable is set so lowest score can be retrieved in O(1) constant time.
    - Boolean isHighScore(int checkScore)
        - Determines if the current score is higher than the lowest score.
    - Void insertNewHighScore(String insertName, int insertScore)
        - Writes new high score to text file.
    - String getHighScores()
        - Iterates through all the high scores stored in the ArrayList and concatenates it into one large string.

12 Self-reflection on Development process during the term project

This project has challenged me in many ways. The biggest challenge of all was admitting to myself that I cannot think through the ENTIRE design process in my head and NOT write anything down on paper. That was the biggest take away from this project. Because before this term project, I would simply write code and not think through EXACTLY how I wanted to design the structure of my program. I noticed that with this term project I wrote much cleaner code than I have in the past mainly for spending hours thinking through how I wanted to design my project. Between my Tank Game and Second Game I had roughly 23 pages of notes used as a reference for the design of my project. Not going to lie, I’m tired. But this class has taught me a great deal in terms of being an independent programmer and attacking problems with a thinking first coding last mindset.

13 Project Conclusion

Both games meet the specified requirements of the projects and were designed in a way that is dynamic and scalable if future changes are made. The Second game design was much more dynamic than the tank game because I had more time to reflect on how I originally designed things and how they could be modularized even more. ALL my tables dynamically populate from files so hard coding is unnecessary. My Game Object Generator class allows me to create any object at any time at any place of the screen with one line of code. In addition, this class reads all the csv files in my ‘levels’ folder and can dynamically generate new levels without the need to hard code my maps into my classes. I took this term project as a challenge to modularize as much code as I could and pull ALL my resources in a dynamic way so scalability and mod ability is very easy to achieve. The level of abstraction I have been able to achieve is to the point where a new csv file can be created in my ‘levels’ folder and a new level will appear within the game without ANY modification of code.