# Capstone Project: Space Pirates

## Section 1: Statement of the problem to be solved

Using the classic Asteroids game from the 1980s as inspiration, Space Pirates is a game where the user pilots a small ship through a galactic arena of asteroids, space stations, mines, booty, etc. The goal is to collect as much booty as possible without dying.

Given the minimum specification from the assignment document: (original specification in italics and a description of how it was met indented below it)

# Minimum Specifications

* *The project must have a graphical user interface based on the Swing library in Java (except for Android projects). The main window must be derived from JFrame and have a menu bar and at least one of each of the following: buttons, text fields, labels, and list or combo controls. It must have a caption and its own icon. You should make a conscious decision about whether the window should be resizable to fit the requirements of your application.* 
  + PirateFrame contains a menu bar (with file i/o options, game settings, and help )
  + PirateFrame contains a split panel on a boarder layout. The top part is a panel for buttons (PirateBtnPanel) which contains a button for game statistics and a text box displaying the score. The bottom part is the graphic game screen (SpacePanel)
  + The frame is resizable so players can see more of the game space (they can also zoom in and out using the mouse wheel)
* *Use control borders appropriately to improve the appearance of your application. Provide for scrollbars if/where they are needed.*
  + All control borders are the default.
* *The project must have appropriate listeners for events appropriate to your application.*
  + Multiple mouse listeners are used to control the main space ship
  + Custom listeners (interface classes) are defined for game specific events such as treasure being captured and docking. Anytime an event can be detected in one area of the program but is of real and immediate interest to another area, a listener is created so that classes remain de-coupled.
* *There should be an “About Box” which is derived from JDialog and more attractive and professional looking than a JOptionPane dialog.*
  + The “How to Play” button on the menu bar is meant to meet this specification.
* *You should use icons, pictures, and/or other graphic elements in appropriate ways to enhance your user interface. The window layouts should be attractive and professional looking.*
  + The SpacePanel contains the graphic elements that meet this expectation
* *You should use one or more text files in ways appropriate to your application. Use Java’s built-in dialogs such as JFileChooser, JColorChooser, and so forth as appropriate to enhance your project.*
  + The game can save the player’s progress and reload it later using text files selected from JFileChooser.
* *Your program should handle any exceptions thrown in appropriate ways.*
  + Exception handling exists in several areas (mostly around I/O and system interactions)
* *The application should have an appropriate look and feel. Choose colors and fonts carefully, remembering that your application will be graded on a computer different than yours, and it may have a different screen resolution and different default color schemes and fonts.*
  + This is subjective and hard to confirm it was met since we don’t have an array of test machines at our disposal. However, the game was developed on several different laptops and a desktop with screens of different sizes and quality. The default game window size is half of a normal screen size and can be resized by the user.
* *The application should make appropriate use of arrays and/or arraylists.*
  + Both arrays and ArrayLists are used throughout. Primarily primitive collections are made with arrays and object collections are made with ArrayLists (this is a general statement and not an absolute design rule).
* Use good object-oriented principles in your design and implementation. Consider using inheritance, polymorphism, and other object-oriented principles where they are appropriate in your project.
  + OOD/OOP principles and patterns were generally followed but additional refactoring could be done to further decouple the UI classes from the core game classes by moving non-UI elements and methods into SpacePirateGame
  + Having said that, one example of inheritance and polymorphism can be seen in that all objects that can exist in space are derived from a common class, SpaceObject. All weapons that are used (Laser, Torpedo, ClusterBomb) all derive from Weapon which is a subclass of SpaceObject. SpaceStation is a specialization of WeighStation which is also a subclass of SpaceObject.
  + This hierarchy allows all the common physics calculations to occur in the base classes leaving most of the specialized classes overriding only a few methods to customize certain behaviors.
* Be sure to include appropriate documentation with your project including the names of all team members with an indication of the projects components for which each team member was responsible.
  + Charles Kinser – Game concept and “product owner”. Primarily responsible for the physics behind the game elements including motion, collision detection/reaction, orbit detection/capture, and POV (point of view) window direction.
  + William Kinser – Overall layout of classes and initial structures, file I/O (save/loading of game status and loading of game images), and most of the UI supporting the game (i.e. menu bar, panels, etc.)
  + Sam Reynolds – Treasure creation/tracking, game statistics, images.

## Section 2: List of inputs, outputs, and processing required

The spaceship will be controlled by either the mouse. If time permits, screen touch may also be added.

Inputs :

1. Rotate ship clockwise or counter-clockwise (mouse movement and dragging)
2. Accelerate/decelerate the ship (left button click)
3. Fire weapons (multiple weapons available at higher levels) (right button double click)
4. Upgrade screens when docked to swap out weapons and other features of the ship
5. File navigation for saving and loading game progress

### Outputs:

1. A health bar at the bottom of the screen
2. A statistics popup displays
   1. Health of ship
   2. Weapon currently in use
   3. # of rounds for weapon
   4. Current payload (treasure collected)
   5. Space Coin currently on hand
   6. score
3. When docked at a space station, use has the option to
   1. Trade in treasure for space coins
   2. Buy health with space coins
   3. Buy weapons with space coins
   4. Buy ammo for weapons with space coins

### Processing:

1. As game play continues, the player accumulates items in their payload by mining the asteroids, collecting “loose” valuables, or raiding other ships.
2. Each “physical” object will obey a common set of “laws of physics”
3. Collision detection can happen between the ship(s), weapon’s fire, treasures, asteroids, etc. No collision will occur between the space stations or weigh stations with other objects (as they have a “force field” protecting them).
4. Space stations have a force field that protects docked ships from asteroids and disable all weapons on docked ships.
5. Player must be docked in order to buy upgrades and/or bank payload.
6. Shooting an asteroid will likely but not always release critical resources (aka booty points). The smaller the asteroid the greater the point value.

## Section 3: Identification of the main classes needed and a list of their individual responsibilities

**Class name: PirateFrame,**

**Responsibilities:**

* **The main class that starts the application, owner of all the GUI components**
* **Responsible for the menu bar and setting various menu items as enabled/disabled and/or selected/deselected**
* **Initializes game setup and play (loads icons, sets game stats, etc)**
* **Initiates save/load of game status**
* **Includes the Run method**
* **Handles game specific events**
  + **Ship enters orbit**
  + **Treasure captured**

**Class name: PirateBtnPanel,**

**Responsibilities:**

* **Panel holding the statistics button, score text box and the How to Play button**

**Class name: SpacePanel,**

**Responsibilities:**

* **Displays and manages all the graphical objects in the game**
* **Handles all mouse events**
  + **Mouse move**
  + **Mouse drag**
  + **Button 1 click**
  + **Button 2 double click**
  + **Mouse Wheel**
* Detects collisions

**Class name: SpaceObject (and its subclasses)**

**Responsibilities:**

* Holds common attributes like position in space, mass, and speed/acceleration
* Performs physics for collisions and orbits
* Performs default calculations for damage from collision

**Class name: SpaceShip**

**Responsibilities:**

* Extends SpaceObject by adding logic
  + for detecting whether it is orbiting a station (and calling the registered listener for this event),
  + for detecting whether it has captured space treasures (and calling the registered listener for this event)
  + managing it’s ammo supply for weapons on board

## Section 4: UML class diagram showing the classes and depicting the relationships among the classes (in parts and then as a whole)

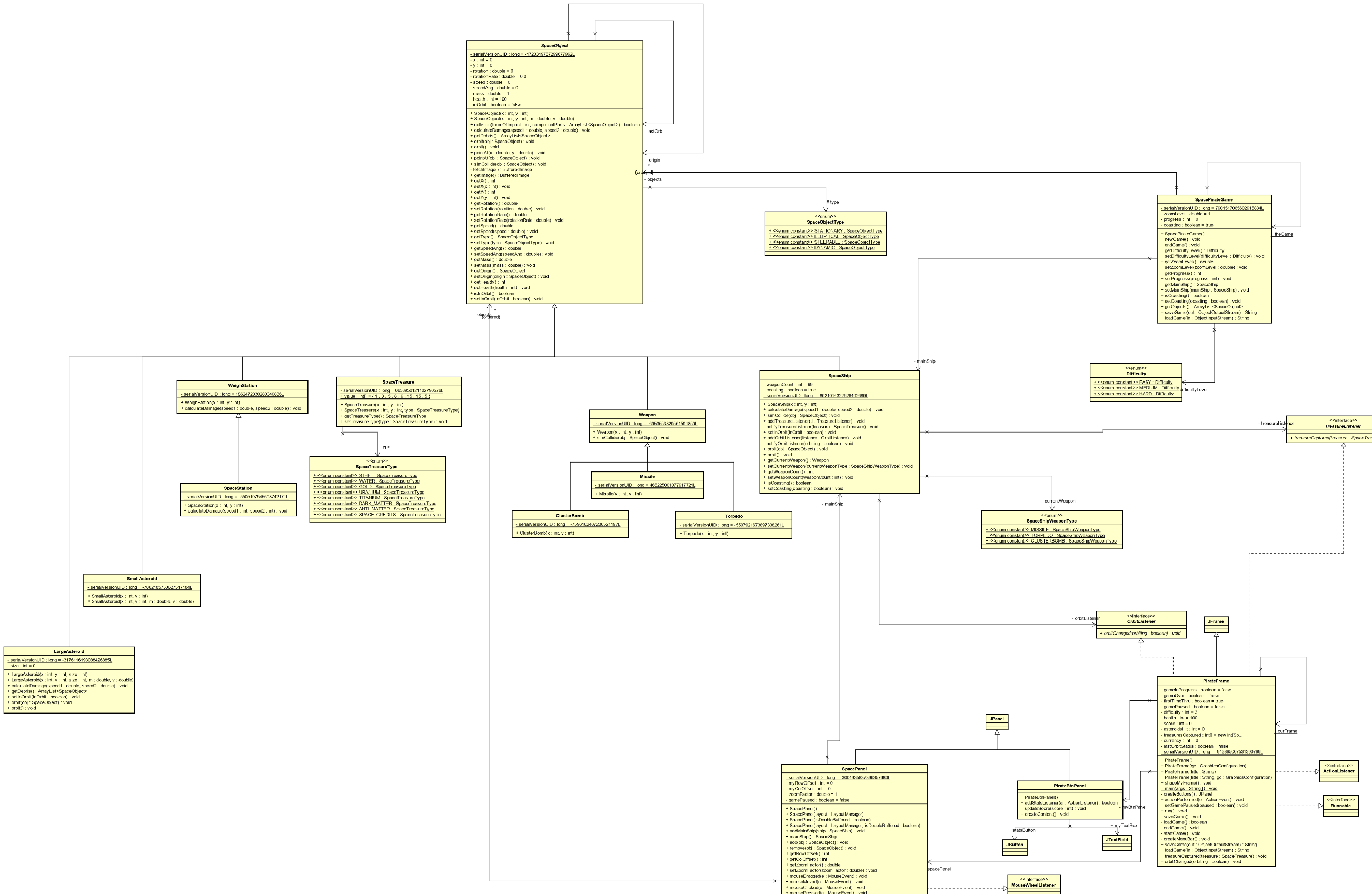


Figure SpaceObject and subclasses

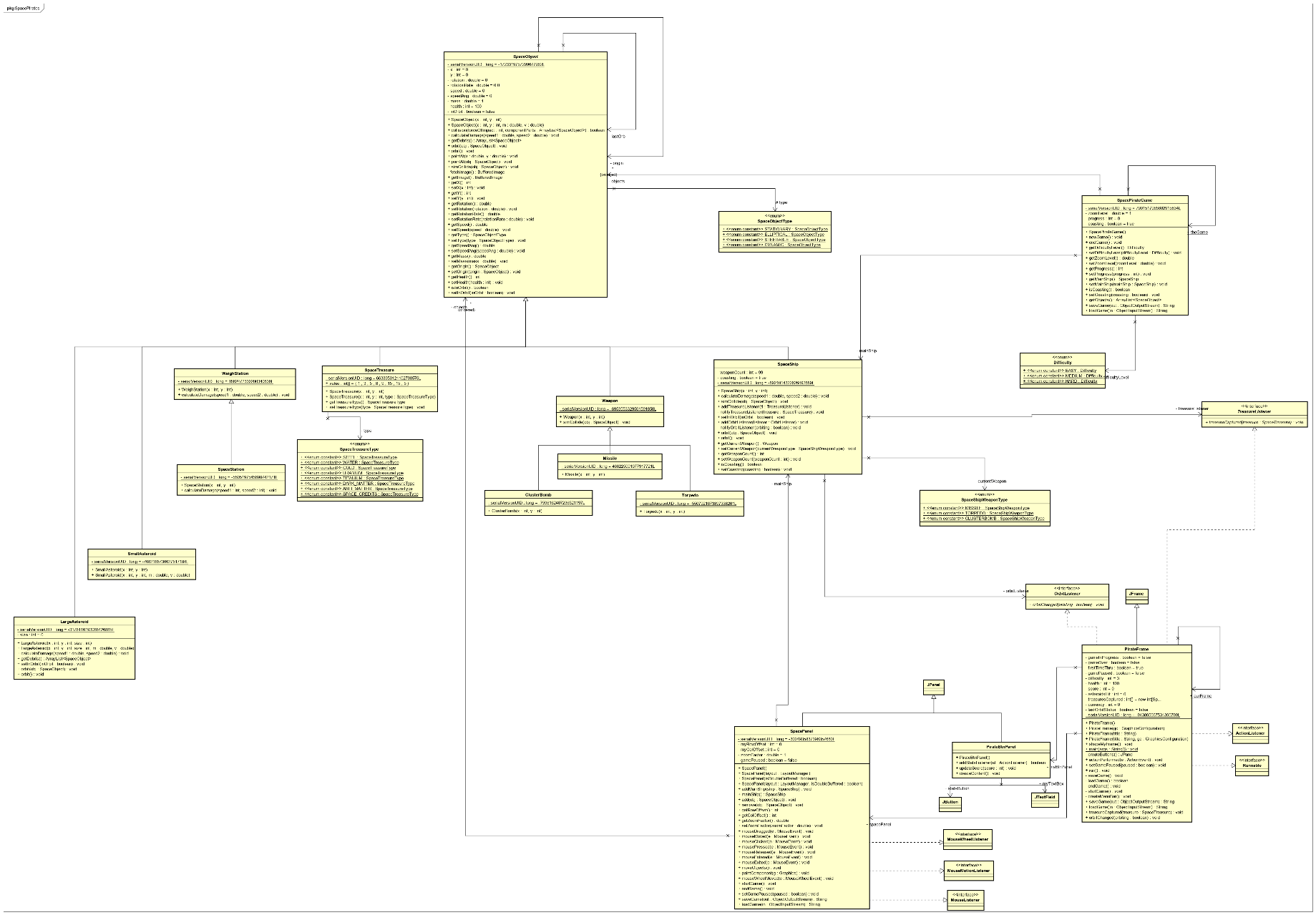
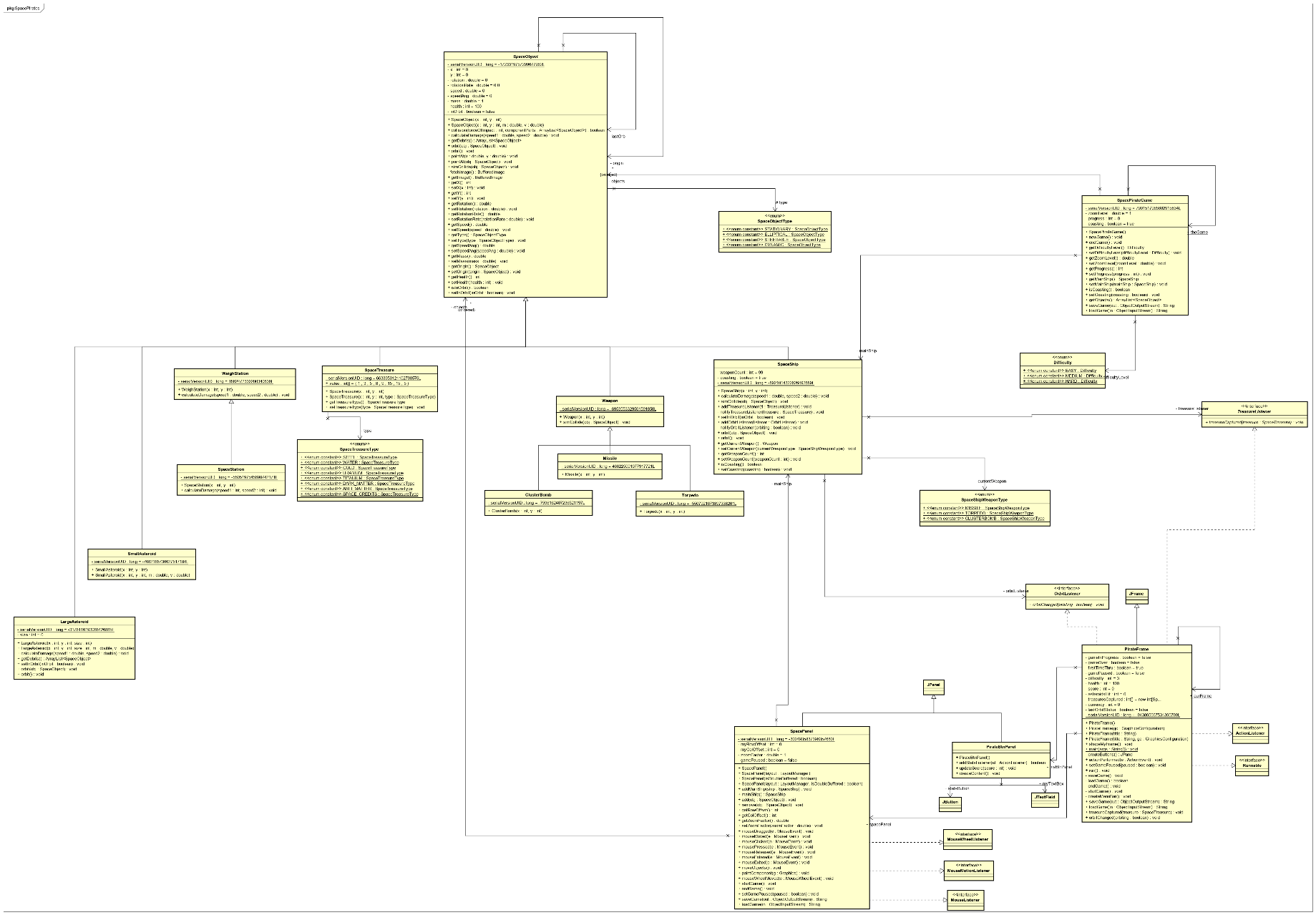


Figure 2 UI classes (Frame, Panel, etc)

Figure 3 Entire Class Diagram

## Section 5: Design of the main and major algorithms

The main control flow logic in SpacePirates will be as follows:

1. Create Frame and all the UI components for the main screen
2. First time running? Prompt user to start a new game, load existing game, or set options (this is to help speed the start and reduce clicks in initial game)
   1. If new game,
      1. reset all game variables for a new game using default options,
      2. start game
   2. If load game,
      1. present file popup to select a game,
      2. load game variables,
      3. start game
   3. If set options,
      1. do not start a game yet (user can use menu bar to set options and start a game)
3. Game setup
   1. Based on the difficulty level, a varying number of LargeAsteroids are created and sent spiraling in various directions (each LargeAsteroid is made up of a random number of SmallAsteroids)
   2. A minimal number of SpaceStation objects are created (one near the ship to start)
   3. WeighStations are created along the perimeter of the initial game space (this is for possible use of cargo ships which will be added if time permits in the project; where the cargo ships carry treasure and credits between WeighStations and can be “raided”)
   4. The main ship is set in the “center” of the game space
   5. Some small number of SpaceTreasure objects will be created to float aimlessly about the game space
4. Game in progress
   1. Display space centered on main space ship (the player)
   2. Display all space objects within view of the ship
   3. Monitor mouse movements
      1. Mouse wheel up and down will zoom the main screen in and out
      2. Mouse movement will change the orientation of the ship (rotate it) and change the direction if the ship is in motion (changes in direction follow rules of physics)
      3. Mouse button 1 pressed will accelerate the ship a fixed amount each time it is clicked up to a maximum amount
      4. Mouse button 2 double click will fire whichever weapon is on the ship
   4. Detection of Collisions
      1. Iterate through all object in space to determine if they are colliding with another object
      2. SPECIAL ALGORITHM LOGIC: To detect a collision, the Images of both objects are converted to Rectangle objects with the same X,Y coordinates. These are used to detect overlaps (collisions). This approach is not a perfect fit since there are gaps between the visual shape and the Rectangle especially at corners but it works for the time given. Also, for this logic to work properly, the main ship must ALWAYS be the first object in the collection followed by any weapon fire (e.g. Lasers).
      3. Because some objects may be destroyed as a result of a collision, they must be removed from the collection. Because of this, loops are used instead of enhanced loops or iterators (removing an object from a collection affects the indexes)
      4. The SpaceObject base class includes methods to handle collision events and collision reaction (including default damage calculations)
         1. Subclasses of SpaceObject can override these methods to customize behavior. For example:
            1. LargeAsteroid will override spaceDebris() to return an array of SmallAsteroids when it is destroyed
            2. WeighStation and SpaceStation override calculateDamage to do nothing since they have a force field and cannot be damaged
            3. SpaceShip overrides simCollide to detect what it is colliding with and respond accordingly

If colliding with a WeighStation or SpaceStation, it goes into orbit

If colliding with a SpaceTreasure, it adds it to its cargo hold (i.e. inventory)

Otherwise, it defaults to base class behavior

1. Move Objects
   * 1. Each object in space has a position in space, a speed and acceleration based on direction and angular momentum (this is described more in the section on the Physics of Space).
     2. SpaceStations and WeighStations are fixed in space, not moving
     3. SpaceShips coast to a stop over time if acceleration isn’t applied.
2. Physics of Space

<< need Charles to describe the physics involved in motion and collisions>>

## Section 6: List of test cases: the data and expected results used to verify the final product is working as expected

Test cases Summary:

The purpose of the test cases is to exercise the SpacePirate’s main functionality of:

1. Player being able to control the space ship
2. Docking and undocking with stations
3. Upgrading health and capability while docked
4. Ability to save and load game progress
5. Ability to shoot and destroy Asteroids (Large and Small), other space ships (if available) and various other space objects
6. Accumulation of points, credits and treasure for destroying stuff and collecting treasure. (that is, points for being a pirate)