**Project 3 Report – Sydney Ng**

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual.

**Class: Student World**

|  |  |  |
| --- | --- | --- |
| **Function** | **Purpose** | **Why I defined it like this** |
| StudentWorld() | Creates a student world, which will hold everything in your game | Constructors don’t need to be made virtual or pure virtual |
| ~StudentWorld() | Destructs StudentWorld | This isn’t a base class so it doesn’t have to be pure virtual or virtual |
| cleanUp() | Deletes everything in the objectvector | This is specific to StudentWorld so don’t make it virtual or pure virtual |
| removeDead() | Removes anything that has been set to dead during the round from the vector | This is specific to StudentWorld so don’t make it virtual or pure virtual |
| randDouble() | Generates a random double between the numbers given | You don’t need to generate a random double anywhere else in the game |
| ProbabilityaddNewObjects() | Calculates if you should add new objects based on formulas given in spec | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| AddObjectToVector() | Adds an object to a vector by pushing it | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| CheckForAddingShips() | Calculates if you should add new ships based on formulas given in spec | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| incNumShipsDestroyed() | Increased the number of ships destroyed by one | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| decNumOnScreenShips() | Decrease the number of ships destroyed by one | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| AddShips() | Does the actual adding of ships to the game | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| getNachenblasterPointer() | Returns a pointer to the nachenblaster | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| setNBDead() | Sets the nachenblaster to dead | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| CalculateGameStatus() | Tells you what to return after each move() | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |
| numShipsLefToKill() | Calculates how many more aliens you need to kill to pass the level based on formulas given in spec | This is specific to StudentWorld so don’t make it virtual or pure virtual. No objects in the game can do this. |

**Class: Actor**

|  |  |  |
| --- | --- | --- |
| Actor() | It initializes Actor with everything you need:     m\_isAlive = true;     m\_ImageID = imageID;     m\_isProjectile = false;     whatWorld(world); | Constructors don’t need to be made virtual or pure virtual |
| ~Actor() | Destructs Actor | Destructors are always virtual for base class |
| doSomething() | Checks if the actor is dead, off the screen, and then will senFod it to the more specialized functions for this class | This is virtual because the ships have a different one. This isn’t pure virtual because stars don’t need it |
| setImageID() | Sets the member variable image id to the one you pass into it | This is set by the Actor class so it’s not virtual |
| getDamageValue() | Returns the member variable for damage | Every class is the same for this so no need to make it virtual |
| setIsProjectile | Sets if it is or isn’t a projectile | Every class is the same for this so no need to make it virtual |
| setDamageVal | Returns the damage value | This is set by the Actor class so it’s not virtual |
| getImageID() | Returns the ID of actor | Every class is the same for this so no need to make it virtual |
| getIsProjectile() | Returns if it’s a projectile | Every class is the same for this so no need to make it virtual |
| CheckIfOffScreen() | Uses X coordinates to determine if it’s on or off the screen | Every class is the same for this so no need to make it virtual |
| getWorld() | Returns a pointer to the studentworld | Every class is the same for this so no need to make it virtual |
| whatWorld(StudentWorld\* world) | Tells you what world it is | Every class is the same for this so no need to make it virtual |
| CheckForNBCollisions () | Uses ecludian distance to tell if you’ve run into everything | Every class is the same for this so no need to make it virtual |
| CollisionOccurred | Calls CalculateEcludianDistance | Every class is the same for this so no need to make it virtual |
| CalculateEcludianDistance | Uses formula in the spec to determine if there’s a collision | Every class is the same for this so no need to make it virtual |
| setLevelOver() | Sets the level as over | This isn’t specific for every class so no virtual |
| setStatusBar | Manipulates string for statusbar | This isn’t specific for every class so no virtual |

**Class: Ships**

|  |  |  |
| --- | --- | --- |
| Ships() | Calls actor, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Ships() | Destructs ships | Destructors are always virtual for base class |
| setHitPoints() | Sets hit points to whatever parameter you pass in | This can be made pure virtual because all ships have hit points |
| getHitPoints() | Returns hitpoints variable | Every class is the same for this so no need to make it virtual |
| doSomething() | Calls the body of somethingBody, checks for collisions & moves | This is pure virtual because every ship will do something, but it’s all different |
| CheckIfAlive() | Returns if the ship is alive/not | Every class is the same for this so no need to make it virtual |
| CheckForProjCollisions() | Bool if there was a collision using the distance formula given to us | Every class is the same for this so no need to make it virtual |
| UpdateHitPoints() | Will add hitpoints to the member variable hitpoints based on parameter | Every class is the same for this so no need to make it virtual |
| SufferDamage() | Causes the receiver to lose a certain number of points based on the ID passed in | Every class is the same for this so no need to make it virtual (since we’ve now passed ID into it) |

**Class: NachenBlaster**

|  |  |  |
| --- | --- | --- |
| NachenBlaster() | Calls actor & ships, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~NachenBlaster() | Destructs ships | Destructors are always virtual for base class |
| setNumTorps() | Sets the number of torps (mem variable) equal to the parameter passed in | This is something that only the NachenBlaster has so don’t make it virtual or pure virtual |
| setCabbagePoints() | Sets the number of cabbage points (mem variable) equal to the parameter passed in | This is something that only the NachenBlaster has so don’t make it virtual or pure virtual |
| CheckProperSide() | Makes sure that the projectiles are coming to the right | This is something that is virtual becuause the aliens do this, but from the opposite side |
| somethingBody() | Moves, checks cabbage points & checks for collisions | This is virtual because it’s derived from ship class |
| StatusBarBody() | Updates the status bar at the top based on NB’s health | This is something that only the NachenBlaster has so don’t make it virtual or pure virtual (since the status is based upon the NB’s health) |
| PostProjectileCollisionActions  () | Checks if there was a collision with a projectile | This is virtual because aliens have the same thing, but are affected differently |
| KeyPressMovement() | Moves the nachenblaster | This is something that only the NachenBlaster has so don’t make it virtual or pure virtual |
| UpdateNumTorps() | updates the number of torps (mem variable) equal to the parameter passed in | This is something that only the NachenBlaster has so don’t make it virtual or pure virtual |

**Class Aliens :**

|  |  |  |
| --- | --- | --- |
| Aliens() | Calls actor & ships, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Aliens() | Destructs ships | Destructors are always virtual for base class |
| getFlightPlan() | Checks length of flight plan | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual |
| getTravelSpeed() | Returns the speed alien is traveling at | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual |
| somethingBody() | Executes the body of the alien including: moving, collision checking, dying | This is a derived class and was already pure virtual |
| AttackNB() | Calls subfunctions attack NB | This is pure virtual because each ship attacks the NB differently |
| allShootingProjectileStuff() | Will shoot a projectile | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| CheckForAttacking() | Checks if NB is in range to attack | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| CheckProperSide() | Checks if projectiles coming from the left | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual |
| FireProjectile() | Fires a projectile | This is pure virtual because all aliens will fire, but just differently |
| CheckForNewFlightPath() | Checks if flightplan is less than 0 | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| NewFlightPathActions() | Chooses a new direction | This is pure virtual because they all move differently |
| MoveInDirection() | Moves in the new direction | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual |
| PostProjectileCollisionActions() | Dies, plays sound, possibly drops goodie (depending on which dervied class functions are called from this, that is) | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| PostNBCollisionActions() | Dies, plays sound, possibly drops goodie (depending on which dervied class functions are called from this, that is) | This is pure virtual because they all do different things after a collision |
| setFlightDirection() | Sets a new flight direction | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual |
| setFlightPlan() | Sets the new flight plan length | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| setTravelSpeed() | Sets a new travel speed | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual |
| AllAlienDeathStuff() | Takes care of everything of the alien’s death (including Dies, plays sound, possibly drops goodie (depending on which dervied class functions are called from this, that is) | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| AlienDeadActions() | This is different for every class, check the derived classes | This is pure virtual because they all do different things after a collision to die |

**Class Smallgon:**

|  |  |  |
| --- | --- | --- |
| Smallgon() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Smallgon() | Destructs smallgon | Destructors are always virtual for base class |

\*note Smallgon inherits all the default f(x) from aliens

**Class: Smoregon**

|  |  |  |
| --- | --- | --- |
| Smoregon() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Smoregon() | Destructs Smoregon | Destructors are always virtual for base class |
| AttackNB | Attacks the Nachenblaster by calling sub functions | This was pure virtual because every alien attacks differently |
| PossiblyCharge | Calculates probability of charging @ NB | This is specific to the smoregon so it’s not virtual or pure virtual |
| AllAlienDeathStuff | Dies, plays sound, causes NB damage | This is virtual because each alien dies differently |
| DropGoodie | Calculates probability & drops goodie | This is virtual because snagglegons also drop things |

**Class: Snagglegon**

|  |  |  |
| --- | --- | --- |
| Snagglegon() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Snagglegon() | Destructs Snagglegon | Destructors are always virtual for base class |
| AttackNB | Attacks the Nachenblaster by calling sub functions | This was pure virtual because every alien attacks differently |
| AlienDeadActions() | This is different for every class, check the derived classes | This is pure virtual because they all do different things after a collision to die |
| CheckForNewFlightPath() | Checks if flightplan is less than 0 | This is something that only the Aliens have (and is common to all of them) has so don’t make it virtual or pure virtual -- it will call sub functions that are virtual |
| AllAlienDeathStuff | Dies, plays sound, causes NB damage | This is virtual because each alien dies differently |
| DropGoodie | Calculates probability & drops goodie | This is virtual because snagglegons also drop things |
| NewFlightPathActions() | Chooses a new direction | This is pure virtual because they all move differently |
| FireProjectile() | Fires a projectile | This is virtual because the probability of firing are all different |

**Class: Projectiles**

|  |  |  |
| --- | --- | --- |
| Projectiles() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Projectiles() | Destructs Projectile | Destructors are always virtual for base class |

**Class: Cabbage**

|  |  |  |
| --- | --- | --- |
| Cabbage() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Cabbage() | Destructs Cabbage | Destructors are always virtual for base class |
| somethingBody() | Moves left | This is virtual because it’s derived from Projectile class |
| CheckIfOffScreen() | Makes sure you’re on the screen | Each one is a bit different (diff directions) so it’s virtual |

**Class: Turnip**

|  |  |  |
| --- | --- | --- |
| Turnip() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Turnip() | Destructs Turnip | Destructors are always virtual for base class |
| somethingBody() | Moves right | This is virtual because it’s derived from projectile class |

**Class: F\_Torpedo**

|  |  |  |
| --- | --- | --- |
| F\_Torpedo() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~F\_Torpedo() | Destructs F\_Torpedo | Destructors are always virtual for base class |
| somethingBody() | Moves left or right depending on who shot it | This is virtual because it’s derived from projectile class |

**Class: Star**

|  |  |  |
| --- | --- | --- |
| Star() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Star() | Destructs Star | Destructors are always virtual for base class |
| somethingBody() | Moves left | This is virtual because it’s derived from actor class |

**Class: Explosion**

|  |  |  |
| --- | --- | --- |
| Explosion() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Explosion() | Destructs Explosion | Destructors are always virtual for base class |
| somethingBody() | Gets bigger in size | This is virtual because it’s derived from actor class |

**Class: Goodies**

|  |  |  |
| --- | --- | --- |
| Goodies() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~Goodies() | Destructs Goodies | Destructors are always virtual for base class |
| somethingBody() | Checks for collisions, moves, checks for collision again | This is virtual because it’s common to all goodies, it will call separate subfunctions that are different |
| PostNBCollisionActions() | Gives the alien the powers of goodie since it picked it up | This is virtual because this does something different than aliens when it collides |

**Class: ExtraLife**

|  |  |  |
| --- | --- | --- |
| ExtraLife() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~ExtraLife() | Destructs ExtraLife | Destructors are always virtual for base class |
| goodiePowers() | Give the user an extra life | This is pure virtual because each goodie has a power and they’re all different |

**Class: RepairGoodie**

|  |  |  |
| --- | --- | --- |
| RepairGoodie() | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~RepairGoodie() | Destructs RepairGoodie | Destructors are always virtual for base class |
| goodiePowers() | Give the user extra points | This is pure virtual because each goodie has a power and they’re all different |

**Class: FT\_Goodie**

|  |  |  |
| --- | --- | --- |
| FT\_Goodie  () | Calls alien, since it’s a derived class | Constructors don’t need to be made virtual or pure virtual |
| ~FT\_Goodie  () | Destructs FT\_Goodie | Destructors are always virtual for base class |
| goodiePowers() | Give the user torpedos to use | This is pure virtual because each goodie has a power and they’re all different |

2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the Turnip class.” or “My Snagglegon doesn’t work correctly yet so I treat it like a Smoregon right now.”

1. There should be nothing that isn’t finished. Everything should be working! I have passed all the test cases & I wrote my own test cases & they worked when I tested them.

3. A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”

1. Created a function called setDamageVal because there was nothing in the framework that would automatically tell you how much to decrease the points by. We were just told to decrement it, but we were never told
2. I gave all my torpedos an extra member variable called owner, which would detect who it belonged to so that it wouldn’t hit someone on it’s own team & would know which way to travel. This wasn’t included in the spec as something to initialize
3. Also, there was nothing in the framework that would tell you who in general was shooting the projectile. To prevent an alien from dying from another alien’s projectile, I checked the X coordinates of the alien ship and the projectile. For example, only projectiles coming from the left can hit aliens and only projectiles coming from the right can hit the NachenBlaster
4. There was nothing to tell you when you should update the status bar. I did this once a tick. I also made this controlled by Nachenblaster class instead of studentworld class because I thought that it more closely aligns with the Nachenblaster class since this is just a reflection of the Nachenblasters’s current state (health, points etc).
5. There was nothing to check whether the things in the vector were projectiles, which made it hard when we had to check for projectile collisions. I made a member variable in Actor, which would tell you if each item was a projectile or not. That way you know whether to check it against projectile collisions.
6. It also doesn’t tell you when to update your status line, so I just assumed you do it after every move.

4. A description of how you tested each of your classes

|  |  |
| --- | --- |
| **Class Name** | **How Did I test** |
| StudentWorld | I put a break point after initialize() to make sure everything initialized correctly in preparation for the game. You can make sure that there are 30 stars and a Nachenblaster in your vector and that it’s status is all alive. Then you need to put a breakpoint inside of Move() for every object. That way you can see how exactly each one of them moves.  For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. For the functions that are given to us based on GameWorld, like increasing lives and taking care of score, you can just look at the screen on the actual game to make sure that it’s working, since that is an accurate representation of the values of all of the Nachenblaster’s parts. You can make sure that it removes things properly by checking the size of the vector and making sure that it decreased by the number of objects that had died in that round of move(). Also make sure that you are going to the next level or dying properly. Just play the game and make sure that you get the proper output. |
| Actor | I put a break point after the constructor of Actor to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. To test the doSomething(), I put a cerr statement that would print if it’s alive or not and if it was off the screen to make sure it was doing what it was supposed to do. To Check if Getworld() was working, Xcode will tell you because you can try to call the functions of Studentworld. If they don’t appear, you know that it isn’t working. For Checking Collisions function, you have it print the X & Y values and then if they are close and this function is called, you know it worked. To check SetLevelOver() you just can force the NB to be dead and see if this is called. To CheckStatus bar, just run the game and check if your numbers update. |
| Ships | I put a break point after the constructor of Ships to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. To test DoSomething, I would put a breakpoint before it calls somethingBody() to make sure that it has returned the proper values for checking off the screen and still alive. For Checking Collisions function, you have it print the X & Y values and then if they are close and this function is called, you know it worked. For suffering damage, you check it by printing out the values before and after calling it and making sure that it suffered the damage and that it suffered the correct number of points. |
| NachenBlaster | I put a break point after the constructor of NachenBlaster to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. For Checking that the projectiles are coming from the proper side, you introduce a projectile from the left and make sure that it didn’t cause any damage. For the set status bar body, put a breakpoint after Nachenblaster’s move() and check what the values are. Then make sure that this is the same value in the StatusBarBody(). For the somethingBody, you just need to make sure that you’re moving the correct way (by clicking the arrow keys when playing the game and making sure they correspond in the right way). For PostProjectileCollisionActions, make sure that you suffer the correct number of points depending on what kind of projectile it is. |
| Aliens | I put a break point after the constructor of Aliens to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. For Attack Nachenblaster, you need to put a breakpoint after you call each of the sub-functions. You need to make sure that you are generating the numbers correctly, so you put have it print out the min and max (or even hard-code a number) to make sure that when the number is called, it will do the correct action and either fire a projectile or attack the Nachenblaster. For the ProjectileActions, make sure that you suffer the correct number of points depending on what kind of projectile it is. For AllAlienDeathStuff(), you need to again make sure that you are generating the numbers correctly, so you put have it print out the min and max (or even hard-code a number) to make sure that when the number is called, it will do the correct action and possibly drop a goodie. |
| Smallgon | I put a break point after the constructor of Smallgon to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. It inherits everything from Alien, so you don’t need to do anything else. |
| Smoregon | I put a break point after the constructor of Smallgon to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. To check for Possibly Charging,  you put have it print out the min and max (or even hard-code a number) when you generate a random number or hard code it for the test case to make sure that when the number is called, it will do the correct action and ram the Nachenblaster. Do the same thing to make sure that it will do the correct action and possibly drop a goodie. |
| Snagglegon | I put a break point after the constructor of Snagglegon to make sure everything initialized correctly. For all of the functions that change the value of a member variable, I just put a cerr statement before and after that would print out its values to make sure that they did the proper thing. For checking new flight path, hard code each of the directions (since this one is different from the class of the other two) and make sure that it doesn’t go off the screen upwards or downwards when you play the game. When you generate a random number or hard code it for the test case to make sure that when the number is called, it will do the correct action and possibly drop a goodie. |
| Projectiles | This was a pretty simple class and it doesn’t do much. All you can do is put a break point after the constructor and one in the destructor to make sure it is called and initializes to everything correctly (for constructor). |
| Turnip | This is a pretty simple class too. I put a break point after the constructor of Turnip to make sure everything initialized correctly. Then to make sure that it’s going the right direction, you put a cerr statement before and after it moves to make sure that only the x coordinates are decreasing and not the y coordinates --which should not be moving at all (since it can only go left and no other combination of directions). |
| Cabbage | This is a pretty simple class too. I put a break point after the constructor of Cabbage to make sure everything initialized correctly. This has a special CheckoffScreen since it can go right (when most objects are going left, so you put a cerr statement before and after it moves to make sure that only the x coordinates are increasing and not the y coordinates  --which should not be moving at all (since it can only go right and no other combination of directions). |
| F\_Torpedo | This is a pretty simple class too. I put a break point after the constructor of F\_Torpedo to make sure everything initialized correctly. This projectile can move both left or right depending on who it is who is shooting it. You can give your Nachenblaster an inventory of F\_torpedos and then have it shoot. Put a cerr statement before and after it moves to make sure that only the x coordinates are increasing and not the y coordinates  --which should not be moving at all (since it can only go right and no other combination of directions). Then, you can force a Snagglegon to always shoot a torpedo (by manipulating the choosing of random nubmers). Put a cerr statement before and after it moves to make sure that only the x coordinates are decreasing and not the y coordinates  --which should not be moving at all (since it can only go left and no other combination of directions). |
| Star | I put a break point after the constructor of Star to make sure everything initialized correctly. The only thing stars do is move to the left, so that’s what the body of the main function will do. Put a cerr statement before and after it moves to make sure that only the x coordinates are decreasing and not the y coordinates  --which should not be moving at all (since it can only go left and no other combination of directions). You also want to make sure that stars are being created in the right place (within bounds, so make it cerr << the exact coordinates and make sure it’s greater than 0 and below the max height/width. |
| Explosion | I put a break point after the constructor of Star to make sure everything initialized correctly. This has its own somethingBody because it’s “movement” is increasing in size. To make sure that it’s increasing at the right rate, put a cerrr statement before and after you call this function and it will tell you the original size and the new size. Also put a break point after every tick to make sure that after 4 ticks the explosion will disappear by marking itself dead. |
| Goodies | I put a break point after the constructor of Goodies to make sure everything initialized correctly. Since this is a base class you have to make sure that it’s general enough. Luckily they basically all move in the same direction so you can just have it print the original and new locations to make sure it’s moving correctly. Also, you need to check if it’s getting picked up by a Nachenblaster so you can use the crash formula that we have used and tested in Actor to check if there is one. If there is you can make sure it disappears by putting a breakpoint in removeDead() and making sure that it is in there. |
| ExtraLife | I put a break point after the constructor of ExtraLife to make sure everything initialized correctly. The only thing different is dropping the goodie, which is an extra life. You need to make sure that it gets created at the exact location that the alien died, so have a break point where you drop the goodie and make sure that it’s the same as that of the alien. After you picked it up look at the status bar and make sure that you’ve upped your life by one. Also, you need to make sure that you’re calculating the chance correctly, so you can hard code it and then check to make sure the alien dropped one. |
| RepairGoodie | I put a break point after the constructor of RepairGoodie to make sure everything initialized correctly. The only thing different is dropping the goodie, which is extra points. You need to make sure that it gets created at the exact location that the alien died, so have a break point where you drop the goodie and make sure that it’s the same as that of the alien. Then you need to check to make sure that your score has been incremented correctly (can check on the status bar) Also, you need to make sure that you’re calculating the chance correctly, so you can hard code it and then check to make sure the alien dropped one. |
| FT\_Goodie | I put a break point after the constructor of FT\_Goodie to make sure everything initialized correctly. The only thing different is dropping the goodie, which is torpedos in your inventory (in your member variable). You need to make sure that it gets created at the exact location that the alien died, so have a break point where you drop the goodie and make sure that it’s the same as that of the alien. You can have it then cerr the new number of torpedos you have to make sure that it incremented correctly. Also, you need to make sure that you’re calculating the chance correctly, so you can hard code it and then check to make sure the alien dropped one. |