Tutorial – Collaboration Diagrams

Introduction:

In this tutorial (and in the tutorials for the remaining UML sessions) we are going to analyse a simple game so that we can document its design using UML diagrams.

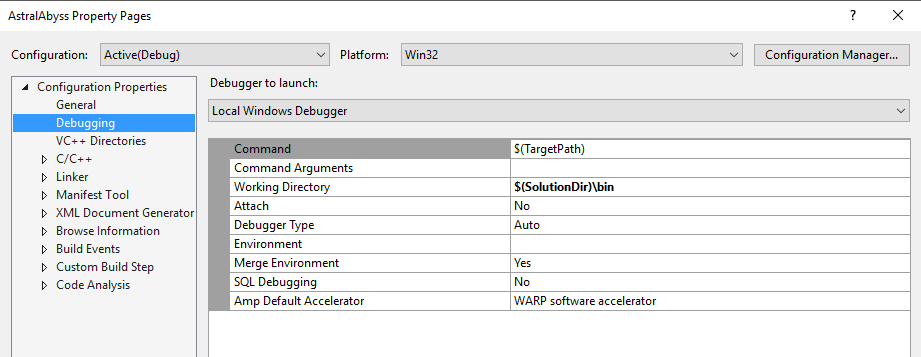
In this tutorial we will create a collaboration diagram for the game Astral Abyss.

In the tutorials for previous UML sessions you would have created various diagrams describing this game. Completing these exercises and tutorials will assist you in this tutorial, so you may find it useful to complete them first if you have not yet done so.

Set Up:

The Astral Abyss project is available on the Resources page for this subject. If you have already added this project to your *aieBootstrap* solution, you can skip this step.

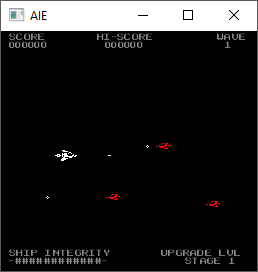
The game is provided as a project that will link into the *aieBootstrap* solution. If you do not yet have a copy of *aieBootstrap*, you will need to download that from this github repository: <https://github.com/AcademyOfInteractiveEntertainment/aieBootstrap>

1. Download *AstralAbyss.zip* from the *Resources* page for this subject
2. Extract the zip file to your computer. (A good place to extract it would be to the bootstrap solution folder)
3. Open the *aieBootstrap* solution
4. In Visual Studio, add the *AstralAbyss* project to the solution.  
   In the *Solution Explorer*, right-click on the solution and select *Add -> Existing Project*
5. Open the properties for the *AstralAbyss* project and ensure the debug *Working Directory* is set to **$(SolutionDir)bin\**  
   
6. Lastly, we need to copy the images and fonts this project uses to the solution’s *bin* folder.

In the AstralAbyss project folder you will find a *bin* folder containing images and fonts. Move these into the **$(SolutionDir)\bin** folder.   
  
If your game launches and you cannot see anything drawn, you have likely copied the resources to the wrong folder.

Once you have set up the project, set it as the active *Start Up Project*, compile and then run the project.

You should be able to launch and play the *Astral Abyss* game without errors.



Creating a Collaboration Diagram:

For the collaboration diagram we’re going to map the same event we described in the tutorial for the *Sequence Diagram* session. This should make it easier for you to compare and contrast the differences between these two diagrams so that you can select the appropriate one to use for your assessment.

The event we are describing is when the player presses the space bar to shoot a bullet.

We’ll start by listing the classes involved in this event:

* The Game class,
* The Player class,
* The Bullet class, and
* The Input class.

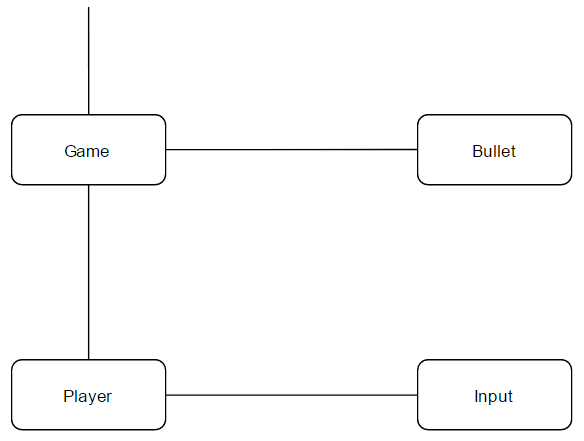
As with the *sequence diagram*, this diagram is an abstraction, so we are only concerning ourselves with the objects directly involved with spawning a bullet.

The *collaboration diagram* shows the relationship between these objects. Any class object that calls a function of another class should be joined to that class with a link.

Looking at the code (or the sequence diagram if you like), we can see that there are relationships (links) between the following sets of classes:

* Game and Player – the Game class will call the Player’s *update* function, as well as checking to see if the Player object is shooting, and its current upgrade level.
* Game and Bullet – the Game class will create an instance of the Bullet class when it spawns a bullet
* Player and Input – the Player class will make a call to the Input class to check if the space bar is being pressed

We can show this link information in the diagram as follows:



We’ve also listed an initial link coming into the Game class to show that this class has a relationship with another (as yet unknown) class that sends the initial message we are describing in this diagram.

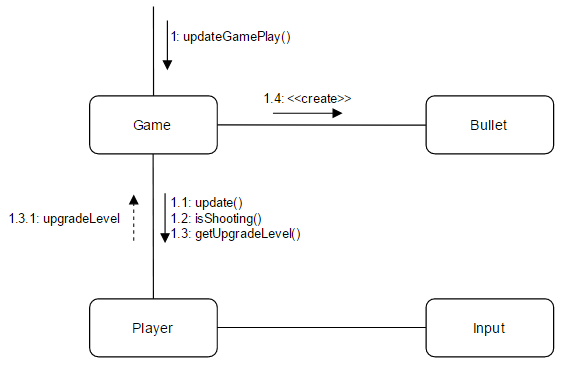
Next, we need to add the messages that are send back and forth between these objects. These messages are numbered, and the numbering needs to be very specific as it dictates the order in which these messages are sent.

The first message is the *updateGamePlay()* call to the *Game* class. We label this message 1. The Game class sends a number of messages in response to receiving this function call. It:

* Sends the *update()* message to the Player
* After the *update()* call finishes, it then sends the *isShooting()* message to check if the player is now shooting
* If so, it will call *getUpgradeLevel()* so that it knows what type of bullet to create, and finally it will,
* Create an instance of *Bullet*

These messages are labelled 1.1 through to 1.4.

We can add this information to the diagram:



Notice we haven’t even touched on what is happening with the *Player* and *Input* class.

There are many ways you could go about mapping the messages in your *collaboration diagram*. You could, as we’ve done here, draw these top level messages first (every message that is sent from the *Game* class in response to receiving the *updateGamePlay()* message).

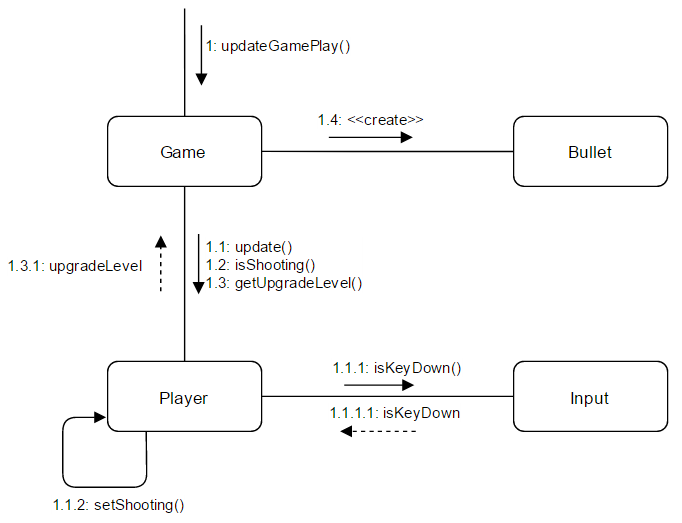
Or, you could trace the messages through completely (so list the messages the *Player* sends in response to *upate()*, and then list any messages that are spawned as a result of those) before coming back to the *Game* class and drawing the next messages.

In our diagram, we’re still missing the messages that are send by the *Player* class in response to receiving the *update()* message.

The Player will:

* Check if the space key is down, and then
* Update its shooting state according to the state of the space key

Let’s complete our diagram by adding that information now:



This is the final collaboration diagram for firing a bullet.

Notice that the final order of our messages exactly matches the sequence diagram we created for this event.

Exercises:

Create a collaboration diagram for another event in the game. You could draw a diagram describing what happens with the player shoots an enemy, or any other event of your choosing.