The Trash Panda Dilemma Part 6

Audrey Randall, Tarah Peltz, Peter Gutenko, Chance Roberts

Implemented Project Requirements:

Business Requirements

ID	Requirement
BR-01	Username for high score must be an alphanumeric and not blank

User Requirements

ID	Requirement
UR-01	As a user, I can move forwards and backwards within a level.
UR-02	As a user, I can jump over obstacles
UR-04	As a user, I can pick up coins
UR-05	As a user, I can add more health to the character currently in use by picking up trash
UR-06	As a user, I can maintain my health by picking up food items specific to the character I'm currently playing.
UR-07	As a user, I can rotate through characters so I can play as the character that I will need for a specific scenario.
UR-08	As a user, I can view my high scores and the high scores of other players to see how I stack up.
UR-09	As a user, I can equip my characters with hats that I have found in the levels to make my characters look cuter.

UR-10	As a user, I can pause the game at any point.
UR-11	As a user, I can find a help menu if I pause the game just in case I forget how to play or I find a bug.
UR-13	As a user, I also want to view the leaderboard from the main menu so I can see how I stand up to the rest without playing the game.
UR-14	As a user, I want to be able to start the game from the main menu
UR-15	As a user, I want to be able to exit the game from the pause menu so I can easily stop playing.
UR-16	As a user, I want to be able to access the leaderboard from the pause menu to see how I'm doing in comparison to other players
UR-18	As a user, I want to able to add my score to the leaderboard when I finish a game
UR-20	As a user, I want to avoid my enemies while I'm playing by using movements like moving forward and backward, jumping, or climbing
UR-21	As a user, I want to try to make it to the goal while I'm playing by using movements like moving forward and backward, jumping, or climbing
UR-22	As a user, I want to play as the rat sometimes, who can fit through small pipes as a special power and is the fastest character
UR-23	As a user, I want to play as the raccoon sometimes, who can claw through obstacles as a special power
UR-24	As a user, I want to play as the opossum sometimes, who can play dead when enemies approach as a special power and is the slowest character
UR-25	As a user, I want to be able to unpause the game after I have paused it

UR-26	As a user, I want to be able to pick up new hats when they randomly appear in game
UR-27	As a user, the health of my character currently in use will decrease the longer I use it
UR-28	As a user, the health of characters not currently in use will increase slowly since I'm letting the characters rest
UR-29	As a user, I can't use a character when it runs out of health

Non-Functional Requirements:

ID	Requirement
NFR-01	The game shall not drop below 60 FPS on any computer 5 years old or newer to allow our game to feel as though it is playable.
NFR-02	If the database goes down, leaderboard may not be viewed but the game itself shall still be playable.

Requirements that were not implemented:

UR-03	As a user,I can climb up ladders
UR-12	As a user, I want to be able to alert the developers of any bugs that I run into so that they could potentially get fixed.
UR-17	As a user, I want to be able to climb walls to access different parts of the areas and possible secrets.
UR-19	As a user, when I play as a rat and am inside

	a pipe, I can't rotate to play as other characters
NFR-03	Scores shall be stored locally in the case of a database issue and shall be uploaded after the database is restored

Class Diagrams:

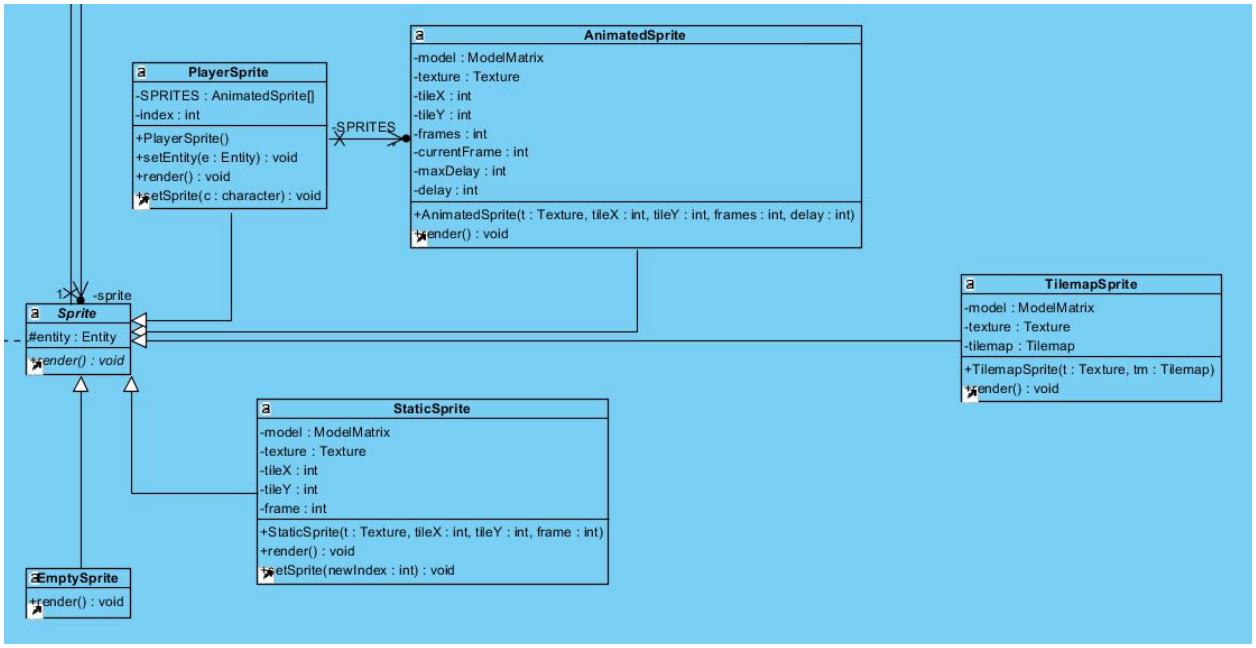
We apologize for the size of our diagrams and the complexity of our code, but there's no way they will fit into an 8.5x11 PDF. Please see our Original Class Diagram and New Class Diagram on Github.

One of the biggest changes to the class diagram was the addition of quite a few functions to PlayerBehavior and several other classes. While the overall structuring of the code and the purpose of the classes stayed consistent, it would have been difficult to accurately anticipate all of the functions and variables we would need. One such instance was the special behaviors. We realized that an array of function pointers would be an elegant way to handle the special behavior for each of the animals. However, there aren't true "pointers" in java, of course, so we ended up implementing the closest solution, which uses anonymous sublasses and is described here (https://stackoverflow.com/questions/2752192/array-of-function-pointers-in-java).
We also ended up adding a number of sprite classes because we discovered differences in the functionality that required separate implementations. The player in particular was tricky, as we needed a way to flip the sprites when they were facing right, we needed 3 different sprites depending on which character the player was without cluttering the Player class, which was abstracted into the PlayerSprite class.
We also adjusted some of the way that items were created. We added an Entity Factory to make the creation of Entities easier, and we added a Level abstraction to abstract the level creation process away from GameWorld, which just manages the different entities within the level.

Design Patterns:

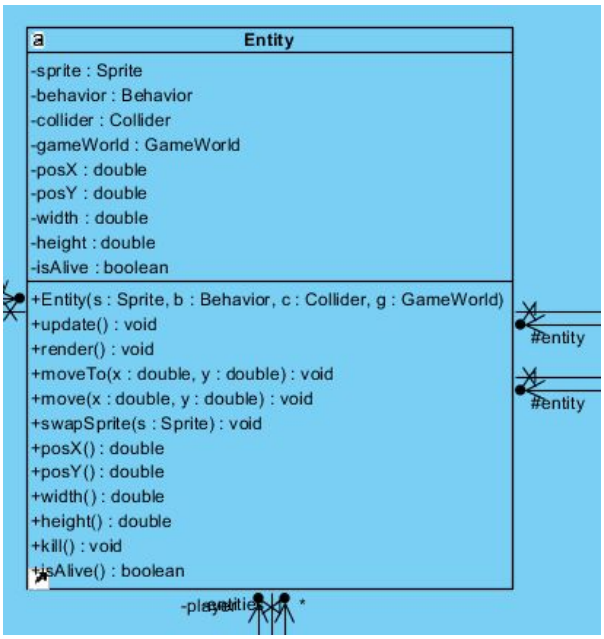
Flyweight

In order to prevent unnecessary re-loading of textures, we used the Flyweight pattern to implement our sprites. The extrinsic portion of our sprites (for example, in classes AnimatedSprite and StaticSprite) is their position in the gameworld, represented by tileX and tileY. The intrinsic portions are the texture each one uses, variables related to their animation behavior, their size, and so on.



Factory

All objects that the player interacts with or sees rendered on the screen in “The Trash Panda Dilemma” are represented as abstract Entities. To facilitate the ability to choose what sort of Entity to create at runtime, we used the Factory pattern to create EntityFactory. This can return an Entity of any type. The classes that inherit from Entity are listed in the EntityType enum so that viewers of the diagram can easily see what types EntityFactory can produce.

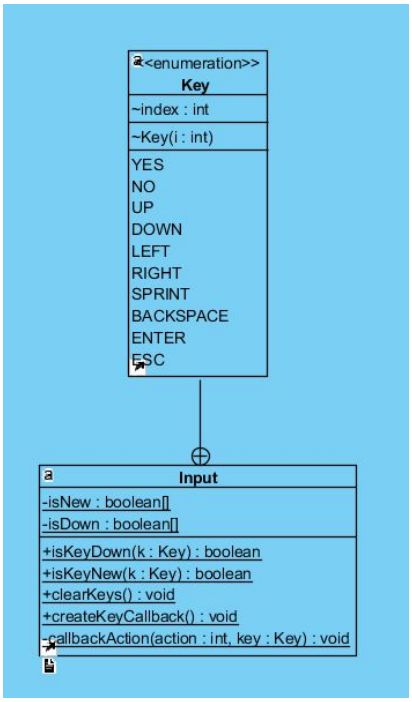


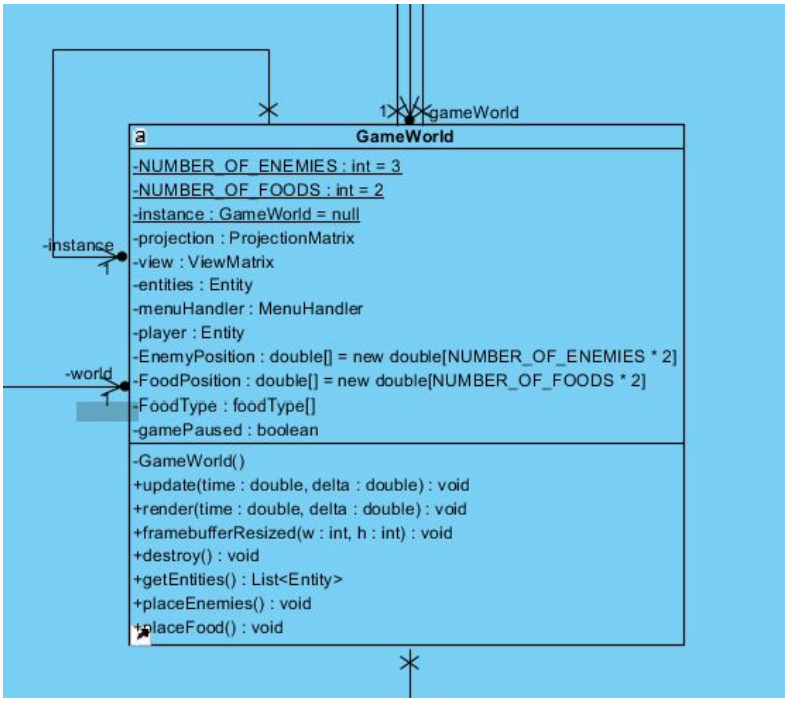
Strategy

In order to encapsulate the variation in the behavior of Entities, each Entity has an instance of a Behavior class. Behavior is an abstract class that implements Component and is inherited from by PlayerBehavior, FoodBehavior, CoinBehavior, and several others. The algorithm that determines an Entity's behavior is selected at the time the Entity is created, and could be changed dynamically at runtime if necessary. Please see the Behavior package in the full class diagram; it is too large to put in this PDF at a visible font size.

Singleton

Two of our classes only makes sense if they are instantiated exactly once: the Input class, which handles key presses, and GameWorld, which knows things like how many enemies the game contains, camera positions, etc.. Most classes in the game need to be able to access these. In the case of Input, in order to avoid race conditions, only one Input handler should be created.





Takeaways

One takeaway from this class is the usefulness of planning out your code before starting. By investing the time upfront to plan out the code, we were able to ensure that each class served a very specific purpose, that there was no redundant code, and that the overall structuring of the code made sense and successfully followed such good practices as low coupling and high cohesion.

Ramping up on projects as a full time software engineer when you start on a new team is also notoriously hard. We learned from this class that many IDE's will generate a class diagram of your codebase for you. We think that (if the code allows) generating the class diagram of the

codebase you want to familiarize yourself with would be an excellent tool during ramp up to see how all the classes work together and where different functionalities are located.
At the same time, refactoring or adding new features to the program requires a bit of planning ahead to make sure that you are still, or that you are now following good design practices when adding or reimplementing these features. This could potentially be difficult, and could take a while to get it so you wouldn't really have to refactor things in the future, but it ends up being worth it to have a nice and easy to understand code base, and the knowledge that your addition or refactoring is good practice.