Implementation

One key algorithm we implemented was the ability for the user to save and load their game. This was also in their requirements documents as #2. We chose to go with the JSON format for our save files because it's a very easy to use and common format, and LibGDX provides code for serializing objects into and deserializing objects out of JSON. This meant that it was quite straightforward to write our own implementations of LibGDX's Json.Serializer<?> interface for the objects in the game we wanted to save/load. The game as we inherited it pregenerated all the levels when the user creates a new game - we kept this mechanic and wrote our save system around it. This does mean that the save files do end up being quite large: around 400-500 kilobytes and 16000-17000 lines of JSON when pretty-printed, however our system handles this with ease.

One requirement that the previous team didn't implement in assessment 2 was items that the player can pick up to receive abilities. We used an enum to represent what a specific item represents, and then defined a function applyToPlayer() akin to the visitor pattern. This function takes a reference to the player and switches on the item type enum to apply specific logic for each type. If the effect can be applied, it returns true, else false. This signifies to the collision system that an item has been "picked up" and so should be removed from the game world (i.e. an item can only be picked up once). For saving the abilities the player has collected, we use a set of item types - this gets saved to the JSON as an array, and when loading, we iterate through the array and for each item type, create a new item with the type and call applyToPlayer. This has the end result of emulating the player picking up all the items again in sequence.

We chose to rewrite several elements of our chosen game, in order to help improve the readability and safety of our code. One of the main methods we employed to achieve this was the use of Java enums. Previously, they used integers to represent the different states that various parts of the game could be - these included enemy attack type (touch only, ranged only or both), enemy shot type (1 bullet in direction of movement, 1 bullet towards player, 2 bullets towards player etc.), enemy movement type (static, random or follow player) etc. Some problems with doing it this way include not being clear what 0 vs. 1 vs. 2 etc mean (they did clear this up in comments, however), and doesn't provide any way of constraining values, which could lead to runtime errors (and so requires testing what happens with erroneous values). This is why we chose to use enums; they give us the compile time safety of knowing that all potential values are accounted for, and in the case of switch statements, the compiler tells us if not all cases have been written (and there's no default case). They also provide us with named constants so that it's clear what EnemyShotType.TRIPLE_TOWARDS_PLAYER, for example, means.

Another change like this that we took a bit further was the "gameState" variable. Again it was originally an integer to represent main menu screen, level select screen, game over screen etc. Early in development, we took the same approach of making it an enum for the same benefits, but when it came to extending the GUI code, it was clear that this wasn't enough - a lot of the logic was being performed in switch statements in the main MuscovyGame class. We changed this to instead use LibGDX screen classes and polymorphism. This allows us to separate the logic and rendering for the different screens into different files, and their specific variables can be moved to those files so that the main MuscovyGame class can be decluttered.

Our architecture around the game's entities has an intricate level of inheritance. All entities inherit from the abstract class OnscreenDrawable, which provides a foundation for all drawable entities. Most classes that are expected to collide with other entities inherit from abstract class Collidable, which provides several helper methods to perform calculations for this purpose. We added another abstract class for all entities that move, MoveableEntity, into which we extracted common methods from PlayerCharacter and Enemy.
One noticeable exception to this hierarchy comes from the Projectile class. This inherits directly from OnscreenDrawable, a relic from the inherited codebase. Projectile defines its own collision box, separate from Collidable, as it is not intended to block or be blocked by other objects; it merely checks if it is in contact with another entity, and disappearing at this point.

A large amount of this code was entangled with with other parts of the codebase, and changing it would cause a lot of extra work which we deemed not necessary for this project. The Item class, which exhibits a similar behaviour to the Projectile class in terms of collision and does in fact extend from Collidable, is more of an example of how we would have implemented projectiles if given more time.

We used the classes LevelParameters and BossParameters as a collection of values that we used to create levels and bosses that shared logic but had somewhat different statistics,

for example bosses in later levels might have more health to make a more challenging experience for players. These classes provided a concise and logical interface for us to create, save and load new variants with ease.

Due to time constraints, we were unable to complete some features to our desired level of completion. These included the save system, objective selection and options control.

While we have implemented a working and robust save system, the system is not in a state which provides users with our desired level of control. We would have liked to provide the user with more feedback to which save slot was currently in use, and provided additional controls such as the ability to overwrite saves. This would share quite a lot of functionality with the LoadGameScreen - both of these would provide an interface to the user that allows them to select a save file, however whilst LoadGame gives the options for “load” and “delete”, overwrite would say “overwrite”. To accomplish this, we could create a base SaveSelectorScreen class that LoadGameScreen and OverwriteSelectionScreen inherit from. One more small feature that could be added here is asking the user for confirmation when deleting/overwriting a save - just prompting them “are you sure?” to prevent accidental deletion of their work. This would be straightforward to do using the ButtonList class and the example in LoadGameScreen of using multiple ButtonLists in a single screen.

We would also have liked to provide the user with some form of objective selection at round start. This would most likely have included presenting the user with a choice between two possible objectives for them to complete, providing a different reward powerup for a different choice. The UI for this would again be easy to make using the ButtonList class, but the difficulty in this lies in changing the level generation - currently all the levels are pregenerated when the user creates a new game, and this includes their objective and creating boss rooms in boss-objective levels (but not in other objectives). With this change, we could still pregenerate level layouts, just not say that a room is the boss room until the user chooses to do the boss objective.

Our final missing feature is the settings screen on the main menu. While this exists as an implemented object, it currently provides no functionality to the user. Our vision of this screen includes providing the user with the ability to completely rebind the controls used in the game to keys or controls of their choice. The ControlMap object is already set up to allow for easy changing of controls, and with the Action enum, we could iterate all possible actions and present the user with a button to change the binding for each. A lot of this would be similar to the ButtonList, however that was designed to just support a single list of buttons, whereas control binding would be more of a grid (actions as the rows and key/controller binding as columns). Code should still be shared between the two, however, to keep the consistency between all the user interface elements.