Finals Weak: Project Part 5
Final Report

Team: Brooke Robinson, Dylan Cole, Michael Feller, Jesus Ortiz Tovar

Title: Finals-Weak: *a game to prove who is weak and who is strong in the face of FINALS!*

Description: We will use the Lightweight Java Game Library (LWGL) to code a 2D, top-down game called “Finals-Weak” where the player must navigate various parts of campus to survive finals week at CU.

Git-Hub Repository Link: https://github.com/brobinson124/finals-weak.git

Table of Contents:

List of the features implemented	2
List of the features that were not implemented	4
Class Diagrams	6
Class Diagram from Part 2	7
Updated Class Diagram	8
Design Patterns Used	9
What We Learned	11

List of the features implemented

Use Case ID	Use Cases	Comments
UC-01	1. Play Game	Game loads and player can move around in world
UC-02	2. Select Level	Press 1, 2, or 3 to switch to a new level
UC-07	6. Character Select	Push ‘j’, ‘b’, ‘m’, or ‘c’ to select a character for the enemy avatar.
UC-08	8. Quit Game	User can press “ESC” to exit the game at any point.

User Requirements				
ID	**Requirement**	**Actor**	**Priority**	**Comments**
UR-01	Easy to learn/Intuitive	User	Medium	The only controls are player movement with the arrow keys.
UR-02	As a user I would like simple and responsive controls	User	Medium	Game is responsive to one or multiple key inputs at a time.
UR-03	As a user I would like a way for the game for end	User	Low	User can press “ESC” to exit the game at any point.

Functional Requirements				
ID	**Requirement**	**Actor**	**Priority**	**Comments**
FR-01	Be able to handle I/O operations and move player accordingly	Developer	High	Game is responsive to one or multiple key inputs at a time.
FR-03	To prevent the user from putting the game in an unusable state	Developer	Medium	Player can step out of level boundaries but game will continue to run

Non-functional Requirements				
ID	**Requirement**	**Actor**	**Priority**	**Comments**
NR-02	Running on different operating systems	Developer	High	Game is built with Java and libraries that are not specific to a particular

				operating system. Compiled with Eclipse, so in theory yes, but only tested on Windows environment.
NR-03	Multiple level of variety within game	Developer	Low	Different levels implemented.
NR-04	Comment code and make it readable	Developer	Medium	Every team member commented code their wrote to make it easy to who to ask for help/blame.

List of the features that were not implemented

Use Case ID	Use Cases	Owner	Comments
UC-03	3. Pause Game	No	Insufficient time to develop this functionality, not essential
UC-04	4. Resumes Game	No	Without UC-03 this feature didn't need to be implemented
UC-05	5. View High Scores	No	Insufficient time to develop a system to score the player's performance and give them a score for the level
UC-06	6. View Controls Available	No	Insufficient time to develop information screens before playing a level.

User Requirements				
ID	**Requirement**	**Actor**	**Priority**	**Comments**
UR-04	As a user I would like to play a game that provides sufficient challenge	User	High	The game currently allows the user to navigate a maze-like room with obstacles that impede movement

Functional Requirements				
ID	**Requirement**	**Actor**	**Priority**	**Comments**
FR-02	End the game/level when predefined tasks	Developer	Medium	There is only one level and no end goal. The game allows the user to walk in a room.
FR-04	Inform the user how to use the game (inform the gamer)	Developer	Low	Insufficient time to develop information screens before playing a level.
FR-05	Provide a UI that adequately informs user of game events	Developer	Medium	There no specific events to show in the final version besides enemies and world layout

Non-functional Requirements				
ID	**Requirement**	**Actor**	**Priority**	**Comments**
NR-07	Save user data (high scores, etc)	Developer	Low	Insufficient time to develop objectives and tasks to record.

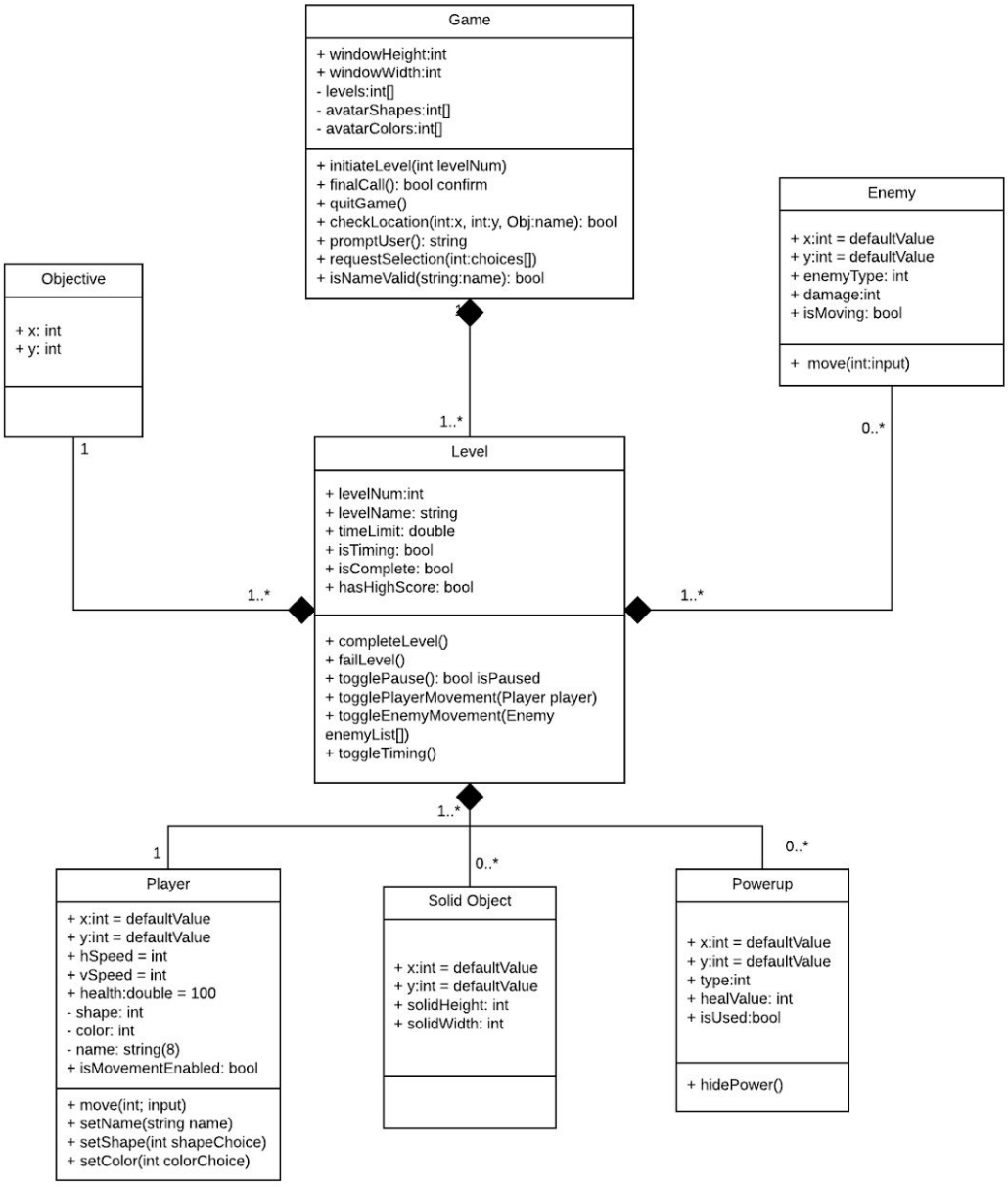
Class Diagrams

Our class diagram ended up expanding quite a bit over the course of our game development. We added multiple new classes for functionality that we originally assumed would have been included within existing objects, but we ultimately broke plenty of classes up in order to avoid creating any “Blobs”.

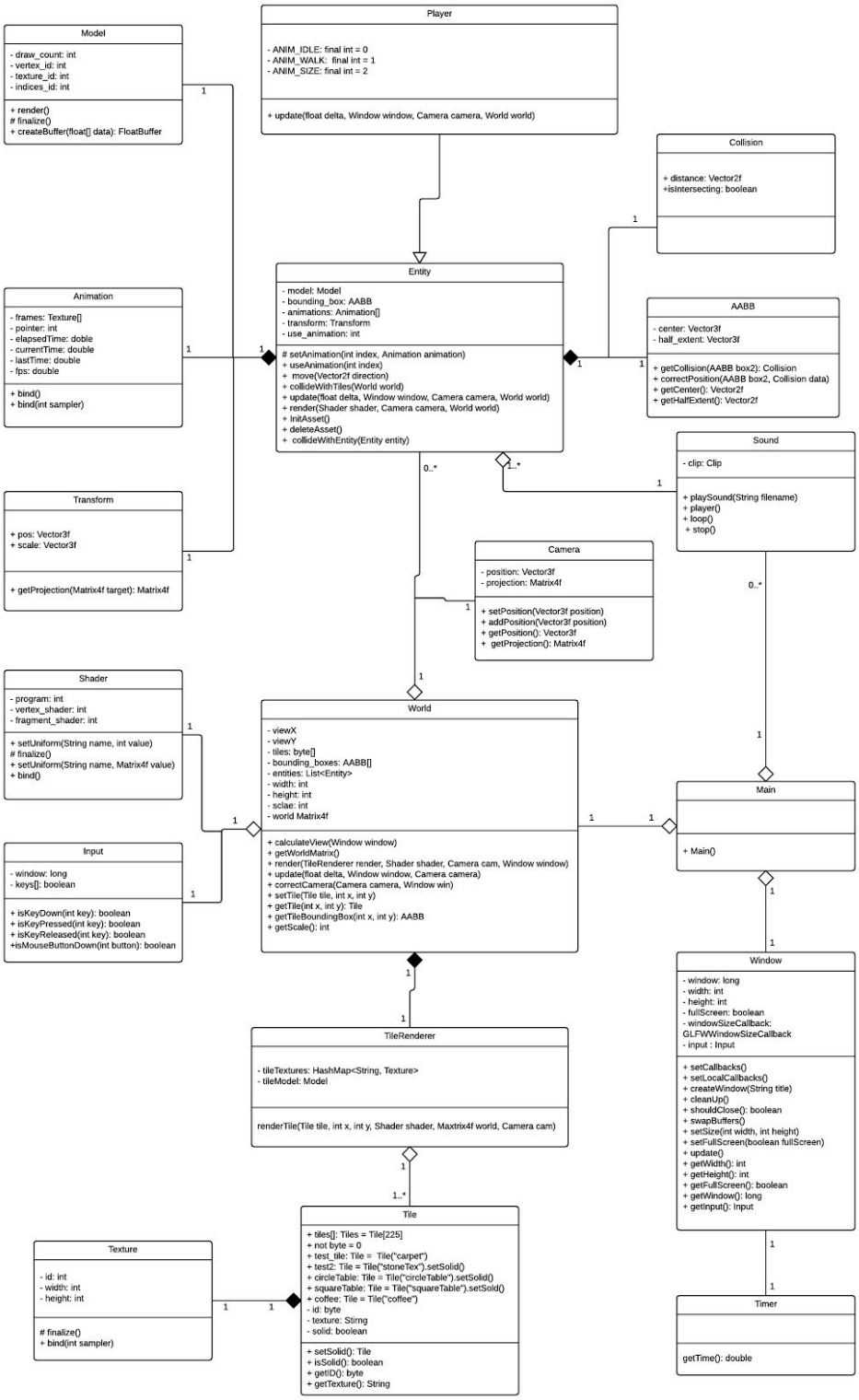
We also ended up adding methods we didn’t even know we would need; frequently a seemingly trivial implementation proved to need far more calculations and attributes than expected (we’re looking at you, camera), but that’s largely due to the fact that we weren’t initially familiar with the library we used. The OpenGL aspects of our project were absurdly finicky, but they taught us to utilize good object oriented programming techniques in order to keep everything straight.

Ultimately, we coded more classes and methods than expected to implement less functionality than we initially thought we would be able to. We thought this would disappoint us, but we were actually quite happy with what we were able to accomplish. The veritable explosion of unexpected new classes opened our eyes to the complexity of game programming, and we realized the importance of keeping our ever-growing web of classes organized and clean; we had to work hard not to make spaghetti code.

Class Diagram (from Part 2)



Updated Class Diagram [Link to full size on Github]



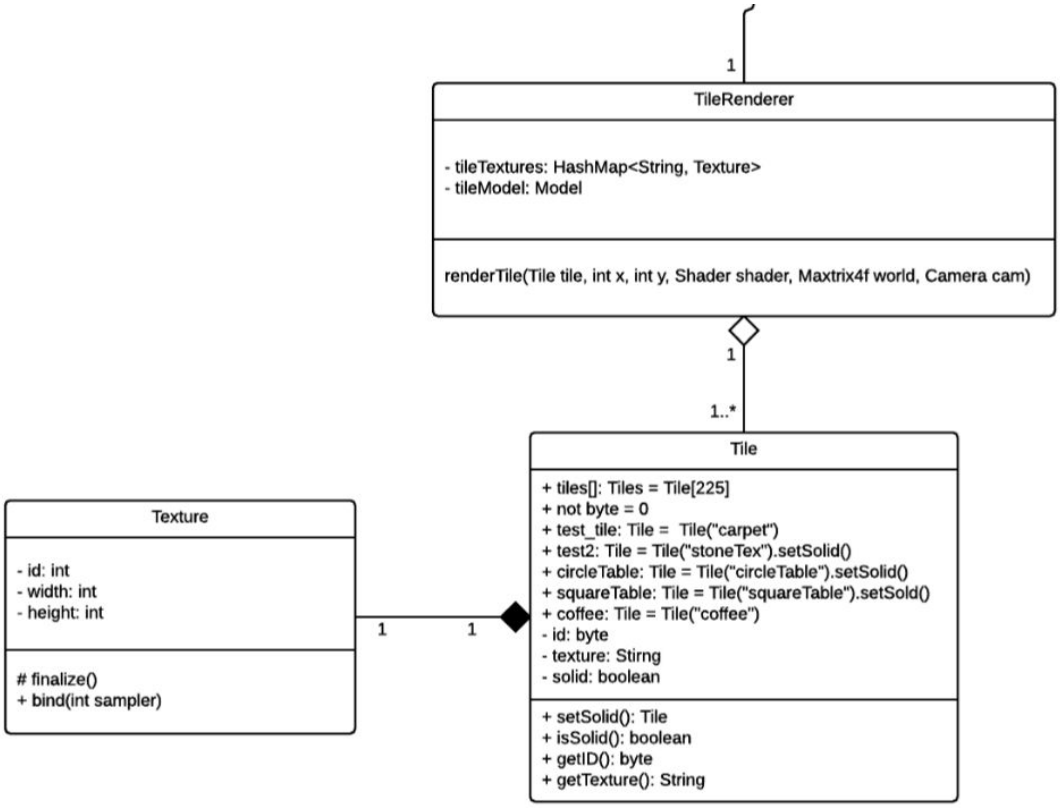
Design Patterns Used

The two main design patterns we utilized were Flyweight and Template. The most complex aspect of our game turned out to be the generation and construction of the world, which we accomplished through the use of tiles. We would create each tile by assigning a size, location, texture, and solidity, and then stitch these tiles together to build our levels.

Flyweight

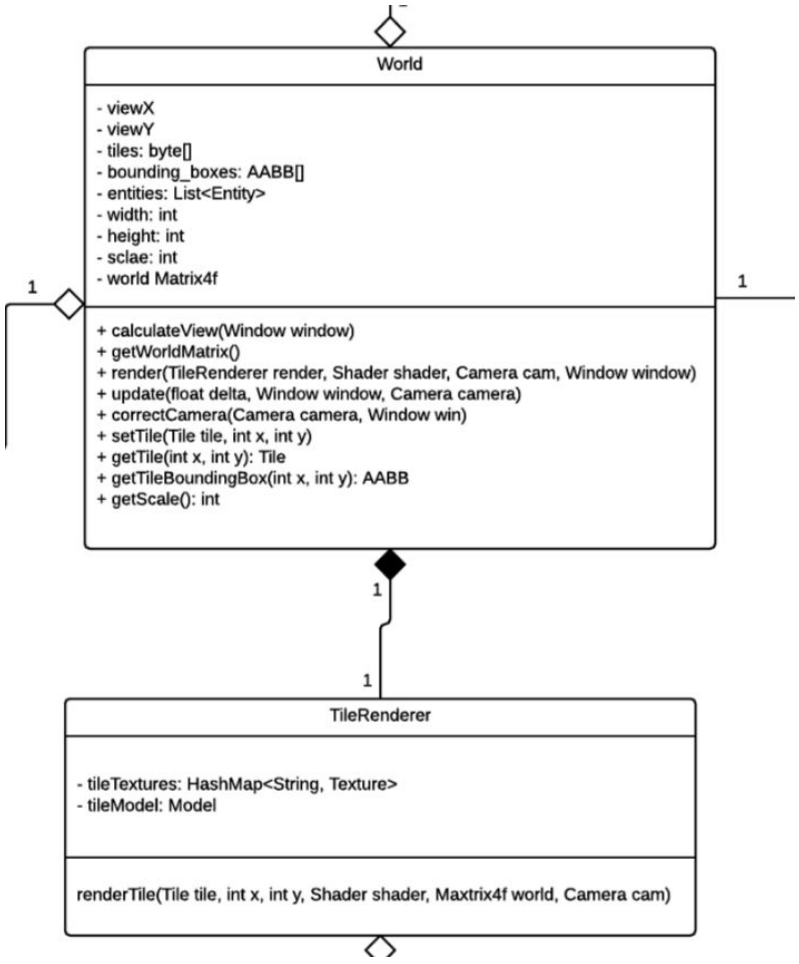
Flyweight classes helped us build these tiles by generating each one with certain attributes (the aforementioned size, texture, etc.). This structure made it exceedingly easy to generate new tiles on the fly, and place them wherever we pleased with a single TileRenderer object.

We used a similar structure for our entities (player, enemies, animated objects), but we felt that the world construction better displayed the principles of the Flyweight design pattern.



Template

While our flyweight classes made it easy to generate these tiles, we didn't want to place every single tile individually to build a level. To make the process more efficient, we built a class structure that would take in an actual template .png file with the layout of our level, and then use Template design pattern principles to call our previously described flyweight classes to flesh out and construct our world. Declaring upfront how these classes would be used made it very easy to build whatever level design we could dream up.



What We Learned

1. Designs change! While it helped to have the starting point we defined in project part 2, our end product differed in some unexpected ways. Classes we thought were simple turned out to be far more complex, and objects we thought were very separate could in fact implement similar interfaces.

2. Some design patterns seem like common sense, while others require some forethought. At times we realized we were accidentally mimicking a design pattern with some class relationships (the Flyweight classes), and at others we had to think “this seems wrong, what pattern could make it better?”(the Template classes).

3. Game programming is HARD. Countless objects and “moving parts” have to work together perfectly to make an enjoyable experience for the player. It made us appreciate just how much work goes into making legitimate, fully-fledged gaming experiences.

4. Anti-patterns take some serious effort to avoid; They seemed to just naturally arise as our code developed and we often had to stop what we were doing to refactor a method or class.