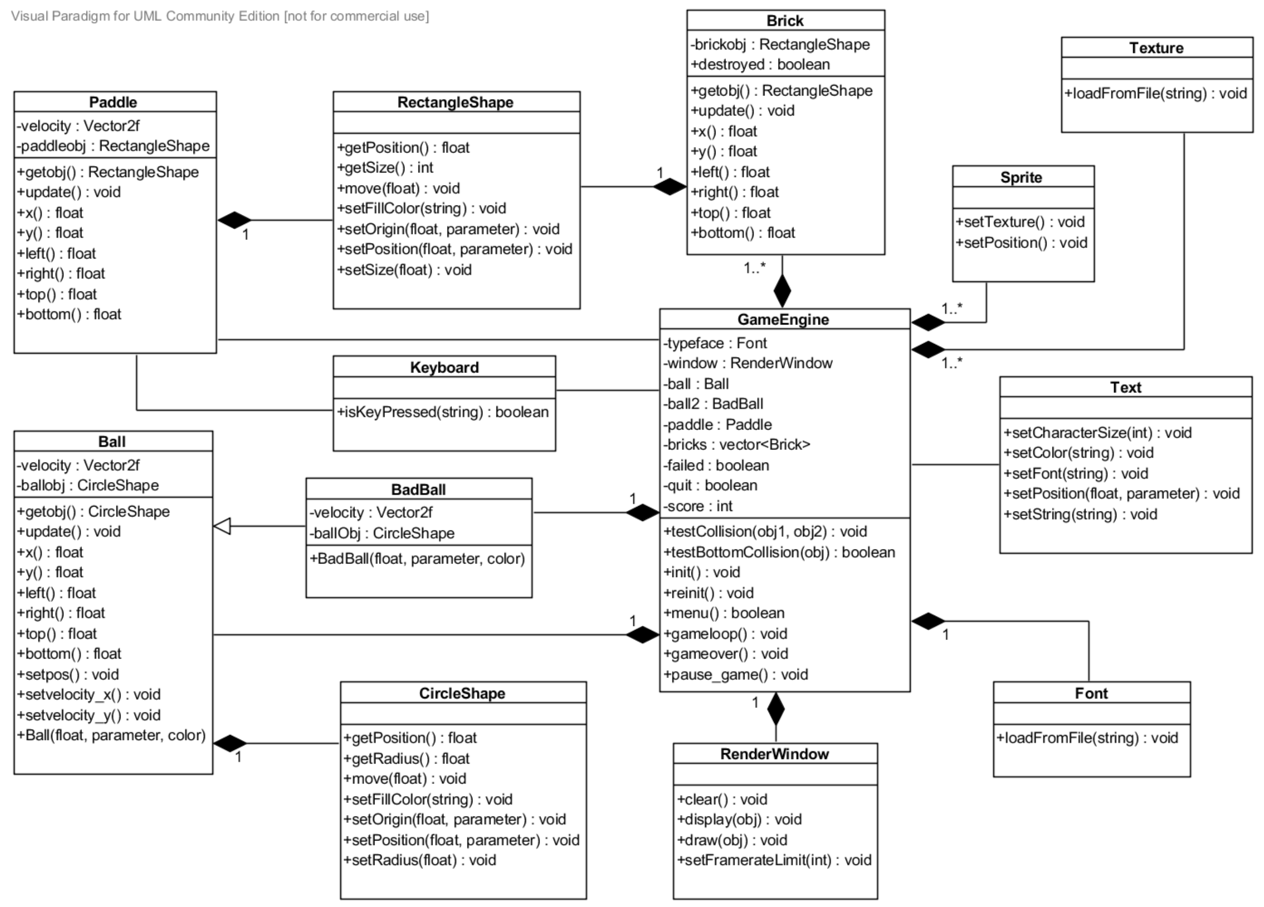
OOAD Project Part 4 - <Arkanoid clone>

Team Members: Josh Weaver, Stephen Ham

***Implemented Features & Class Diagram**:*

- Filled-out main menu, complete with a graphical element
- Scoring system
- Entity that must be avoided (a second ball)
- “Game over” screens for success & failure game-over conditions
- Refactoring of original code for design patterns and organization/clean-up

We lacked sufficient time to complete a Change Settings screen, which would have necessitated much more complex work with the SFML API than what was originally expected (specifically, handling user input on a graphical window, without adding mouse support). It was also determined that a Help screen wasn’t very practical for this game.



Design Patterns:

Facade: we used Facade to structure the core part of the program, effectively using the class GameEngine as the facade for access to the complex system of related classes and methods, as the original code did not use a facade pattern and instead filled the C++ program's main() method with lines of unnecessary code that made it unwieldy to manage.

We did not use any other design patterns, though given additional time we could have implemented any one of these:

- Prototype: to "spawn" more instances of a ball instead of using inheritance
- Observer: to autonomously update the ball, paddle, & brick attributes on every change
- Singleton: feasible on the Paddle and GameEngine classes to keep their objects to a single instance

Change in design since Project Part 2:

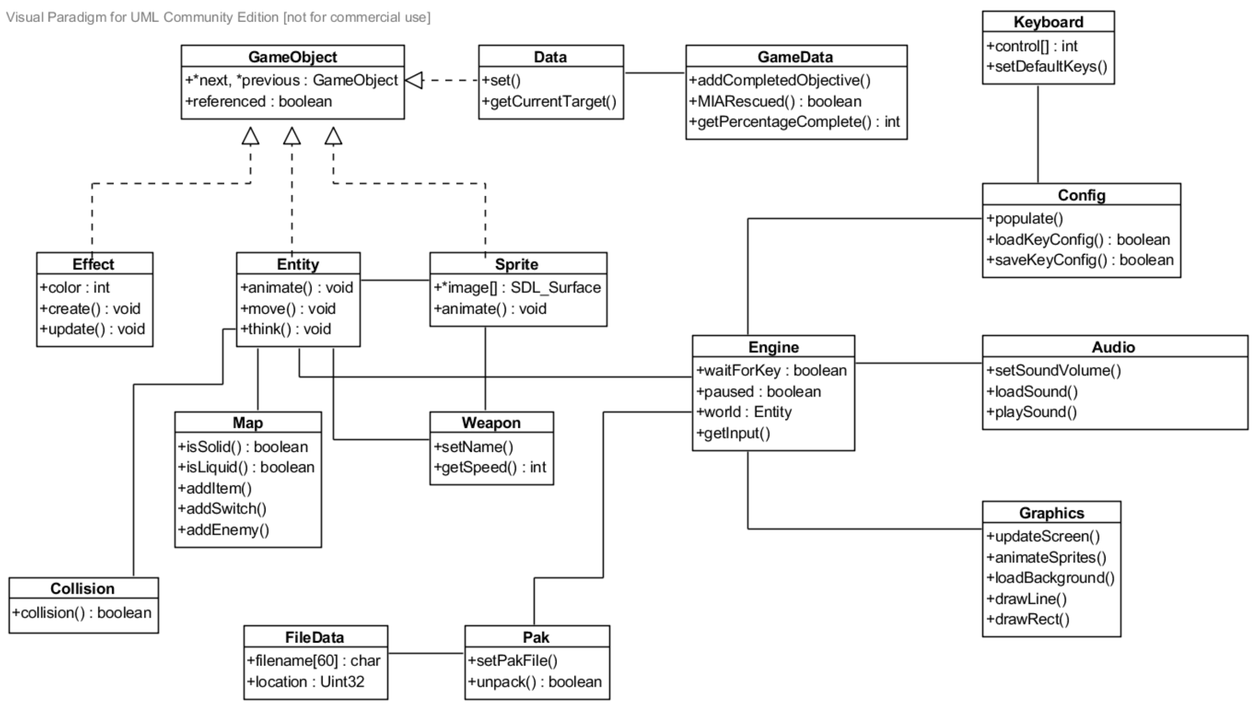
Since we changed our project since Part 2, our Part 4 class diagram naturally changed as well, though there were some similarities. Both class diagrams use a main "Engine" type class that interfaces with a variety of other classes that all provide operations & attributes for the objects in the game "world", both internally (graphics objects, data objects, calculation methods), and externally (interfacing with the keyboard).

The Part 2 diagram also showed some disorganization, specifically with the number of different classes, some of which made little sense to separate from the Engine class, like Collision and Weapon, which could have both been attributes (with corresponding methods to access/mutate them), and Entity & Data, which did not seem like they should have been separated from each other.

The diagram for part 2 unfortunately also had to be compacted for space due to the sheer number of classes, methods, & attributes, which made it more difficult to determine which classes were merely interfacing with others, versus either composition or aggregation relationships.

It was also interesting to note that in our new class diagram, composition relationships were shown more clearly, versus the inheritance relationships in the previous iteration of our project, and almost no classes were "inherited" (per se) in our new project.

As far as similarities, we noted that both diagrams relied on a few functionally identical classes and methods, like Keyboard, Sprite, collision() methods, and "entity" type classes, which made sense, as almost all games have some type of entity system. Most notable was the difference in the level of complexity between the Game Engine classes, as the first had fewer attributes & operations while the second had much more.



It was clear though that the programmer(s) of the original game did not sufficiently “design” their project to be easily maintainable, and that the programmer for the other game we selected put more thought into his design. We noted later revisions of his code that did, in fact, use class inheritance as well, to make Paddle and Brick inherit from RectangleShape, instead of them merely instantiating an object based on that class (composition relationship).

What we learned about the process of analysis & design:

We learned that going through the analysis steps is essential for listing out every part of a project to ensure that progress on all parts is tracked, and that creating the core UML diagrams (use cases, activity, sequence, etc) is also very helpful to verify that the software system behaves as intended when completed, or to identify any implementation issues that might exist.

Creating the full complement of UML diagrams allowed us to see how the classes interacted with each other and how to plan for user interaction, which revealed the depth of our first project idea and the better design of our second idea, but which also went to show that depth & features are not always better than a system that is well-designed, because in fact the first idea had many features that made it appear more thoroughly-implemented but turned out to have negative effects and we found that the code was largely sloppy and disorganized when we tried to compile & build it. Through this we also found that writing proper documentation for other programmers can also be invaluable, especially when it comes to open-source software that will inevitably be attempted to be modified by others, as the first project idea had incomplete documentation (and was thus more difficult to create the diagrams for).

We also learned that applying a design pattern on pre-written code took a fair amount of upfront time to implement, quite a bit longer than expected, but once it was done it had a visible positive effect on the cleanliness of the code and made it much easier to follow.