The Powder Game

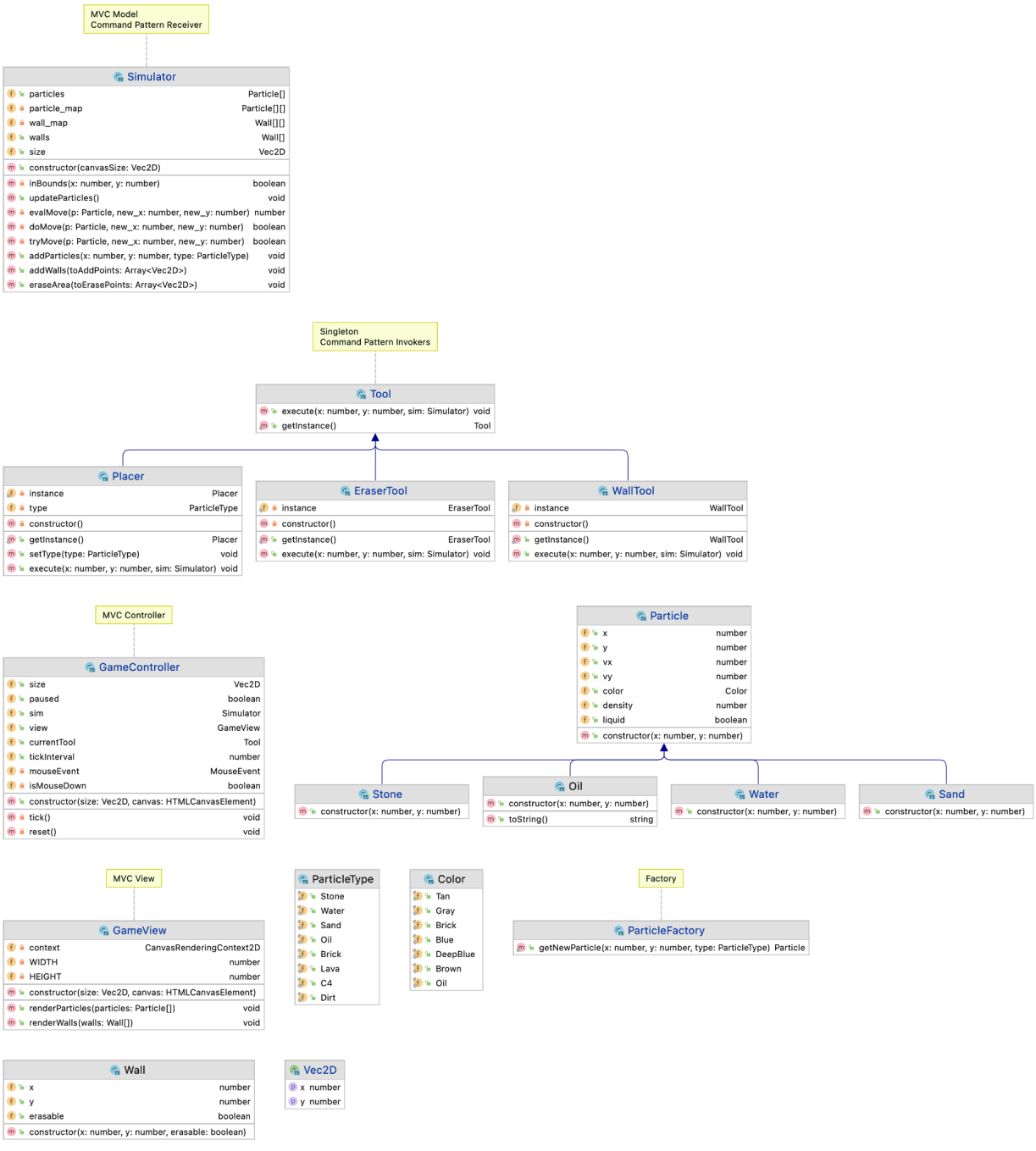
Lucas Webb and William Mardick-Kanter

Final State of System

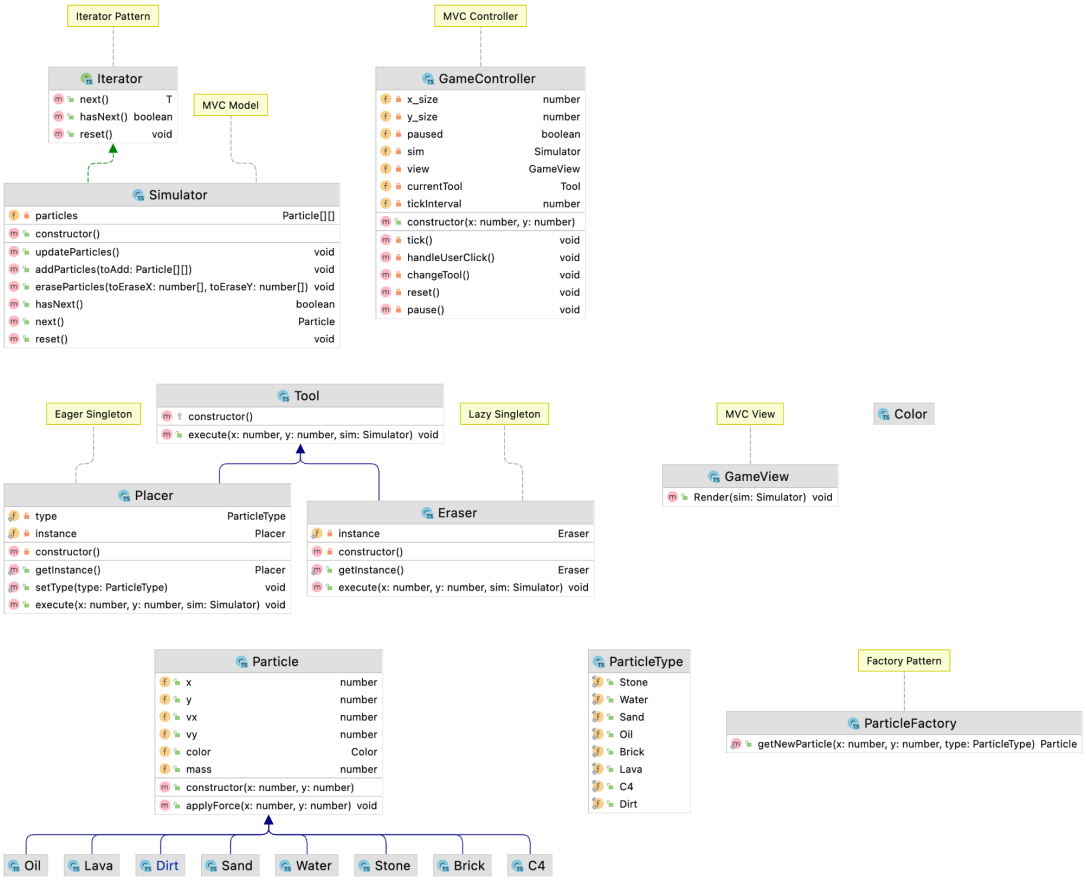
We ended up implementing the majority of the features we set out to. The only features that we did not implement were some of the more advanced physics capabilities such as explosions. One of the stand out features that is included is simulation of liquids. It's quite fun to play around with our app. Additionally, we included a slider that allows the user to change the pixel size, a feature we didn't originally plan on including. Between project 5, 6, and 7 not a lot changed as far as the structure of our program. We had planned on using the iterator pattern to iterate over the particles while rendering them but this didn't end up being necessary. We did end up using a command pattern to execute the different tool's functions.

Final Class Diagram

Current UML Diagram



Project 5 UML Diagram



Key Differences

There weren't any significant changes in our design. We had planned on using the iterator pattern to iterate over the particles while rendering them but this didn’t end up being necessary. One pattern we added was the command pattern, which is being used to execute the different tool’s functions.

Third Party code vs. Original code

The only third party code we referenced came from the actual Powder Toy that was implemented in C++, while ours was pure typescript. This code was used as a reference primarily for the physics simulation aspect. Of course our code is significantly different but still loosely modeled on the actual Powder Toy code.
https://github.com/The-Powder-Toy/The-Powder-Toy

OOAD Design Process

1. Strategic Programming

   Because we laid out the architecture before coding, it made designing and expanding our ideas a simple process. For example, once one class of particles was working as planned, adding more classes was as simple as adding the button in the html and adding it to the ParticleFactory switch statement.

2. Encapsulate What Varies

   Because of the nature of our game, we knew we were going to need to encapsulate a lot of logic in our GameController class. That class was mainly responsible for acting as the controller between the front and back end. We used encapsulation in how the tools were used and how the particles were placed/updated which kept this main class clean of messy logic.

3. Single Responsibility Principle

   We implemented the MVC pattern as the overarching pattern in the system. This meant we needed to keep all the classes responsible for solely what they were supposed to do to keep the controller from handling logic. By doing this, it's much easier to modify the code because all that's needed is the correct parameters to interact with another class, should the need arise.