Monster Game

Our biggest factor on design tradeoffs and decisions was time. We were very limited with time so we opted to create a simple version of the project. With whatever we could not complete we added in comments and pseudo code to show our plans for when we had more time.
Since every member had worked closely with Model-View-Adapter in project 4 we had grown accustomed to it. Model-View-Adapter does not allow any information flow between Model and view while Model-View-Controller in a triangular shape allows the model and the view to be connected. However we still closely modeled this project to project 4 which used Model-View-Adapter, the difference being we had to shape our controller accordingly in our file TouchMe. This like the adapter accounted for the creation of the activity along with creating events when a user touched it.

In the creation of our classes we abided by the single responsibility principle. We made sure each class had only one responsibility so it could be easier to extend classes and for easier for the whole group to understand different people's contributions. A good example of this is how our within in States file, we have divided up each class based on its State rather than having all the states in one file. This relays into the Open/Closed Principle, while we did extend our classes but our extensions did not alter any of the original code. With the multiple interfaces we have, we utilized the interface segregation principle. This allowed us to create classes that only used methods it needed it. This is seen with our interfaces for AbstractMonsterModel and then again for MonsterListener. We continually used dependency inversion principle where both our modules and details were based off of abstractions or the idea of the game and different functions of that game. We also used Liskov substitution principle allowing subtypes of objects to be replaceable. Overall we organized our design in such a way that no little changes rippled through the program making it unusable and different parts worked side by side without too much reliance on little details.

We did try to do testing but we were unable to complete and get them running in our program. However we did write some sample ones as ideas of what we would might have implemented. As our design changed rapidly as we got closer to the due date, so we were unable to add to tests and choose to focus on the actual functionality of the game.

In this project we experience virtual concurrency due to the fact that we were using an emulator. We wanted to make sure all of different states along with the timer worked and updated with the view. One of the ways android facilitated our different states was through their lifecycle. Another way we could have facilitated this process was if we could have implemented async. With Async we could have done background operations while updating to the current thread. But thankfully we managed fine without it.