**Objects and Properties**

For each location in the Simulation, there will be an object that extends **Location**. **Location** is an abstract class that contains the information that applies to all locations, including information such as x and y coordinates, name, type, display value and whether it is probable or not. It also contains some abstract methods that all locations will have – these are methods like getting the configuration string for the specific location. The reason this method is abstract is because each location will be of a different type that will have different configuration settings.

From this, I have two subclasses: **Object** and **Property**. These represent the two types of elements that can be in any given space. **Object** represents the items that there can only be one of per space (Rock, Energy Pill, Robot etc). **Property** represents the items that there can be many of in a space (Water, Jammer, Hole, Fog etc). **Property** itself doesn’t have any special fields or methods that **Location** doesn’t have, but it allows for more specificity when passing to functions. **Object** has two extra specializations that **Location** doesn’t have. These include indicating whether the object is movable or not, and the color information. Both **Property** and **Object** are abstract.

For both **Property** and **Object**, there are further specializations. For **Object**, these specialized, concrete subclasses are **Block**, **Ball**, **EarthRock**, **EnergyPill**, **Robot**, and **RomulanRock**. Each of these has their own specializations for their needs. These include functionality such as energy. **Property** has its own set of specialized concrete subclasses – **Fog**, **Hole**, **Jammer**, **Mud**, **Lava**, and **Water**. These contain specializations that allow their own functionality.

The reason I chose to follow this model was because it seemed logical. I thought that both properties and objects have similarities, so I extracted that into a superclass. Then there was functionality that was the same among objects, and functionality that was the same among properties. I extracted this functionality out into the **Object** and **Property** classes. All the logic specific to a given type is limited to the lowest level of abstraction.

**Simulation**

The simulation is a standalone class. It contains a 2D array of **Objects** and **Properties**. The simulation knows of the objects and properties it contains, and it can take care of the probing and getting seen items. I didn’t want the objects to have knowledge of the board, which is why the simulation handles the functionality that needs access to the board.

**Simulate**

This is facilitates the coordination between a Simulation and the robot brains. These methods can be found under the **Simulate** box on the UML diagram. This simulate contains a **Simulation** and then lists of robots/robot brains that it communicates with. When a robot needs to update it’s location, it will do bounds checking, and then update its own location. Then **Simulate** will call a method in the **Simulation** that will update the robot’s location, move any objects, and communicate back to the **Simulate** whether or not the movement was successful. **Simulate** will then pass that info back to the robot brains.

**Brains**

Each brain is a standalone product. They have their own **Robot** instance, and they have knowledge of the sim width and height. From this, they can make educated decisions. There is a **Brain** superclass that contains the similar functionality held between brains. This includes functionality such as parsing requests and initialization messages and sending the basic commands. It contains an abstract method called makeMoveDecision that takes in a list of seen objects, and the probing information from the last turn. Each robot brain (**PillAddict**, **Sentry**, and **Tribot**) implements this method in order to make decisions that will help complete the task at hand.