Item objects are difficult to work with when extended. One issue we’ve encountered was trying to extend the item into another usable item, such as the water gun. There is no clear distinction to define what can be done with an item on the floor, and what can’t. For example, we want to use the spray water action only when we pick it up. But you can use it when it’s on the ground. To fix this, we could use other lists of actions to store actions that can be done when picked up. This list would be swapped with the other list of actions whenever the item was dropped or picked up. This allows us to have items with multiple sets of actions that can be used for a greater variety of item diversity.

Similarly, while there are constructors to create items without the “pick up” action, or items with just the “drop” action, this doesn’t work very well when you extend to items with more actions, like with a key. The constructors are only good for making items, not anything extended from item. For example, if I wanted to make two keys on a map, one on the floor and one on an enemy, I’d need to redefine a new constructor just to make an “inventory” key. To remedy this, we could have a function that checks through the allowable actions, deletes the pick-up action and optionally replaces it. Then we’d make the static constructors use the given functions when creating furniture or inventory items. This would make the process of extending items to new items much simpler.

Accessing and iterating through the GameMap is difficult. The X and Y values used to create the GameMap are protected, which means that scanning the map for a specific tile is difficult. For example, when we tried to look for a launchpad on another map, we couldn’t search properly and end up getting index out of bounds errors. The GameMaps should have a getter for X and Y, so that the actions impacting that map could realistically search the map for specific tiles, thus creating a clean way to search the map without round about error handling.

Actions have a very narrow scope. It gets difficult to do actions affecting objects outside the current map. For example, our implementation of the MoveMap action requires an actor to jump to and the map it’s on. However, that is very difficult to do from an actor level. You’d need to start referencing all the way from the application level. To fix this, you could have actions take in the whole list of maps in the world as an optional parameter. This allows actions to easily find and affect other maps without much hassle. The main disadvantage of this is that you’d need to be careful that you don’t affect something in another map that shouldn’t be possible, e.g. attacking another actor on the moon.

The world seems to display actions done by all actors, even on other maps. If the game gets bigger with more maps, the turn logs would get very cluttered with all the actions done by every actor. For example, in our game Yugo Maxx is still taunting the player even when the player is on the moon. Additionally, there are some actors that shouldn’t display any messages at all. For example, in our game the Oxygen Dispenser is an actor, but is made to do nothing but skip turns, meaning a lot of messages will be displayed of the Oxygen Dispenser doing nothing. To remedy both these issues, we could have a “loudness” attribute to actors, a default level of noise that can be used to make some actions appear global, local or silent. Then we make actions have a loudness level too, which can either be pre-set or put in as a parameter. For example, the Do-Nothing action could have a parameter, allowing regular actors to show they do nothing locally, but allow others to be silent. This would allow multiple actors to do what they need to do without cluttering the display.

There is a lack of triggers in the engine. It would be much easier to implement events if there was something in the World object that can check the events that happen. So far, the only way to do something like this would be to create an actor somewhere with an action to check every turn, but even so it’s only once per turn and only on said actor’s turn. For example, if I wanted to trigger some sort of game over event, I’d prefer it to trigger when the player dies at that moment. To remedy this, we could just add some sort of “Check Triggers” function right after processActorTurn(actor) in the run function. This would iterate through a list of triggers input from the application level and check their conditions. If those conditions are met, the trigger would run the appropriate action. This would allow for more scripted events, give players drops without giving the actors the actual items and so on and so forth.

Actors should be given intrinsic actions they can do on their own. For example, when we tried to create a Quit action, we couldn’t just attach it to the Player object to use. We circumvented that by creating an un-droppable item which gave the quit action, however should we need many more intrinsic actions like spells or menus, we can’t just keep adding it to items and giving them to the player. To fix this, we could have actors have a list of actions, and add that to the list of actions to check in the World object during the processActorTurn function.

We feel that the method for dealing with unconscious actors is a little bit counterproductive. On the one hand, we have an isConscious function to check if an actor is Concious, however if the method used to make an actor unconscious also removes the actor, replacing it with a generic item. This makes it difficult to work with mechanics involving consciousness. For example, we thought of making our spray water action have an extra feature to wake up unconscious enemies, however since they were items, we couldn’t really do much to them. To fix this, we could add functions to actors similar to items, so we don’t need to replace the actors with items. This way, we could do more interesting things to bodies and increase the flexibility of what actions can reasonably do.

The way hitting enemies works does not leave much room for expansion. Right now, all attacks have a 50% chance to hit, and there are no special effects for weapons, such as a longer range or a status effect. For example, we originally wanted to be able to attack an enemy with the water gun, dealing no damage but giving it a soaked debuff. To allow for a greater functionality, the action for attacks should be linked to the weapon used, so the Attack action would call the weapon’s OnHit action, which could give accuracy or additional effects like the stun effect. This way, weapons would have more purpose compared to their current iteration as a plus to damage.

The way the engine gets attack actions right now also makes it difficult to integrate range weaponry. Right now, the world only checks for actors directly by the exits of the current location. This is fine for melee weapons, but if we wanted to use a water gun at a range, we’d need to code in a specific range attack function onto the weapon item itself. To fix this, we could instead mark certain actors as “attackable” instead of them returning an Attack action. Then the world could iterate across all actors on the current map, check for attackable actors within a certain range, check for obstacles and create an attack action accordingly. This would allow for weapons of any range and won’t require the creation of specific range attack functions to work, making it much easier to create ranged weaponry of any type.

The skills in the engine right now are very limited right now. Since they only return Enums, you need separate classes to deal with the ramifications of said skill. For example, we might have wanted to make a swimming skill that would allow you to move through water, however the higher the swimming level the longer you could go without taking damage, however we’d need to specifically create the skill, and several different levels of that skill, then alter the water to work with the skill and calculate damage based on the enum returned. It’s not very practical. To remedy this, we could have skills be stored as a separate skills class, which would hold its own set of values depending on what skill it was, a function to return what skill it is, a function alter the skilled object’s properties and a function to add actions to a skilled object, for example. That way we can make skills modular, and makes it easier to test and implement skills in the game.