# Recommendations for extensions to the game engine

1. Improve the extensibility of World class run() method
   - We noticed the logic of World class run() method could not be extended without first extending the class then rewriting the entire method as the initial setup of the world and the main loop logic are both placed within the same method. This violates the Don't Repeat Yourself (DRY) principle and there is no workaround except modifying the engine. This is bad because any modifications or improvements done on the engine will not reflect on the extended World class. In our implementation, we could not add the global rain event without rewriting the run() method.
   - We propose extracting everything within the while(stillRunning()) loop in World class run() method into a protected method mainLoop(), and everything before that to a protected method setup(). The run() method will then call setup() and repeatedly call mainLoop() until stillRunning() is false via a while-loop.
   - With this implemented, to add new functionalities (e.g. rain, in our case) to the main loop of the game, the developer could extend the World class, and only override the mainLoop() method and place the new logic there, without having to rewrite the entire run() method. The same goes for adding functionalities in the setup part. This ensures adherence to the DRY principle.
   - We do not find any disadvantage to this improvement except added code complexity than might put additional cognitive load on the developer, though insignificant. Considering the benefit this could bring, we can safely disregard this disadvantage.

2. Implementing design by contract
   - There is no checking for negative values in Actor class heal() & hurt() method. If something went wrong when computing the points, and points go negative, the heal() method will hurt the actor, and the hurt() method will heal the actor. This is an unwanted and unexpected behaviour. Since no exception will be raised, the error cannot be easily found until much later. Furthermore, in methods of many other classes, the methods do not check whether the value passed into them are bad values (null), and will fail without leaving behind an informative stack trace. It has been a frustrating experience debugging these errors. These have violated the Fail Fast (FF) principle, where a system should fail immediately when an unexpected behaviour is expected, and fail locally if possible to ease the debugging process.

- We propose methods to perform precondition checking before executing the main method body. For example, the Actor class heal() method should check if points is negative, if yes, an exception should be raised immediately.
- With the implementation of design by contract, the developer does not have to worry about the engine displaying unexpected behaviours. If anything went wrong, it has to be the developer's code. It limits the scope of debugging, reduces the amount of time spent on debugging, and more time will be spent developing the features.
- We see no significant downsides to this proposal. It might be confusing for inexperienced programmers. However, an experienced programmer can spot the design by contract pattern straight away and have full faith in the engine.

3. Inclusion of native quit game support
   - As of now, there is no native support for quitting the game. If the game is running from the command line interface, the user needs to perform a keyboard interrupt to quit the game. This can confuse the player as a game should require no technical knowledge of the player to play it.
   - We propose including the QuitAction we implemented as part of the game engine. The action removes the player from the game, which halts the running world as the world could not run without a player in it. An instance of QuitAction will be added to the player's actions via Player class playTurn() method, so it will be added to the list of menu selections available to the player every turn. This implementation also adheres to Single Responsibility Principle (SRP), where the QuitAction only does one thing which is quitting the game.
   - With this implemented, the player can now quit the game via the quit option in the displayed menu (with an assigned hotkey) without performing a keyboard interrupt. This can create a more user–friendly experience to the players, and the developers do not have to design this functionality themselves.
   - We could not think of a scenario where the game does not need to be ended. This is true even for games that can run indefinitely (e.g. Conway's Game of Life), as the player will definitely want to quit at one point. Therefore, we see no disadvantage in this proposal.

4. Addition of more interfaces
   - The interfaces provided in the interfaces package allow extension to the corresponding classes in the game engine without modifying the classes themselves. This removes the need to downcast references in the game. This is an application of Open–closed Principle, as the involved classes are open to extensions to new features, but closed to any modifications, as

any new functionalities can be implemented through the provided interfaces – we do not modify the classes that contain the main logic which will risk breaking the engine. In some cases, if all subclasses share the same implementation, a default implementation can be supplied through the interfaces.

- Without those interfaces, to access the extended functionalities of a class, we either need to extend the classes that utilise that class to take in the extended class' instance, or perform downcasting to match the required type, of which both are suboptimal. We had this issue when attempting to implement 'Dirt, Trees and Bushes' in assignment 2.
- We propose the addition of WorldInterface, GameMapInterface and LocationInterface, and place them within the interfaces package, and have their corresponding classes implement them.
- This makes the game engine more flexible, as simple functionalities can be added to the classes without extending them unnecessarily. Furthermore, the interfaces can be used as reference types instead of the concrete classes. It is better to depend more on abstractions as the code can be more flexible. For example, if both A and B implement an interface, and the interface is used as the reference type, then the code will work with both classes. However, if type A is used as the reference type, then the code will only work with class A. This is an application of the Dependency Inversion Principle (DIP).
- We do not see any downside with this implementation, except the introduction of 3 new classes that might complicate the logic flow.

5. Built-in support for crossing maps
    - We do not understand why this functionality is not built-in to the game engine. Both the World and ActorLocations objects, of which both will only have one copy each throughout the game's runtime, only track one player. This implies the game does not support multiplayer, as any more players will not be shown in the UI if they are on other maps, although they can still be controlled via the displayed menu in the UI. However, the GameWorld stores the GameMaps in a list, which implies there can be more than 1 GameMap. This means that if we have more than one map, the player needs to be able to traverse to other maps somehow, but the functionality is not supported by the engine.
    - An easy fix would be to move the connectMaps() method we wrote in DinosaurMap class to the GameMap class in the engine, to have native support for this functionality. To add the second map and connect it to the North of the current map (connect by x-axes of the maps), we added Exits to the bottommost row of the new map that have their destination set to the topmost row of the current map, and the other way around. Do the opposite and we can

connect the maps by their y-axes. A map can also be connected to itself, so if an Actor walks past the West edge then he will end up at the East edge. This can be done for arbitrary number of maps, if their connecting axes are of equal length.

- This implementation also reduces the dependencies (ReD) of the system. Had this not be implemented, a naïve developer will connect the maps in Application class, which makes the Application class depends on Exit class. As this is implemented within the GameMap class itself, it does not have any added dependency.
- With this implemented, in the future, any future games that use this game engine can traverse through the maps without manually adding this functionality by extending the GameMap class. Developers should be focusing on adding new features, not adding features that should have been built-in.
- We see this as a functionality missing from the game engine. There should not be any disadvantage in adding it back.

6. Inclusion of built-in pathfinding / BFS algorithm in Location class
   - The game engine provides support for hitpoints, healing, hurting other actors etc., which implies pretty much every game that uses this game engine will need to pathfind or search for a location that satisfy a certain condition or some sort, to heal or hurt other actors. Without this, how can an actor locate another actor? As of now, there is no built-in pathfinding or any algorithm to support this, and the developer needs to design it himself. Ideally, the game developer should be focusing on developing new functionalities, instead of pondering with algorithms or low-level implementations.
   - Therefore, we propose the inclusion of built-in pathfinding and/or BFS algorithms in the Location class. In our implementation, we made the extended GameLocation class an Iterable. Iterating through a GameLocation now iterates through locations from nearest to farthest from that location. This is done through the Breadth First Search (BFS) algorithm on the exits of the current location and their exits and so on, and implemented with Iterator Pattern. This can be moved to the Location class so the developer does not have to downcast Location to GameLocation every time this has to be done.
   - With this implemented, the developer can focus on implementing the functionalities without worrying about the algorithms. This speeds up the development process.
   - A clear disadvantage of this proposal is, having complicated algorithms in the game engine can confuse the developer, but this problem can be overcame with good documentations.

Conclusion

- In general, the game engine adheres to the SOLID principles, which makes the code of the game engine highly scalable and readable.
- For example, the engine adheres strictly to Single Responsibility Principle (SRP) — the entities do exactly what their name implies and nothing more than that. This has made the code highly readable, as there is no hidden logic, and in our case, understanding the code was easy. Its adherence to Open–closed Principle (OCP) — functionalities of classes can be extended via inheritance (e.g. Dinosaur class extending Actor class) and overriding implementations, allows changes to what the code does without modifying the logic within the engine. The engine's functionalities can be extended but closed for modifications. Us able to implement all features without modifying the engine is proof. Moreover, it also complies to the Liskov Substitution Principle for maintaining the expected behaviour of the base Action class in the extended classes. The interfaces are all very specific and we do not see in any case any unnecessary methods being forced on classes. This is an implementation of Interface Segregation Principle (ISP) and it increases the code readability as the displayed behaviour of classes are more predictable. We could also see Dependency Inversion Principle (DIP) implemented as the arguments of methods use interfaces as reference type instead of concrete classes. This means any added classes that implement the interfaces will be compatible with the current code without the need of modifying code that risk breaking it.
- With that being said, the game engine still has some limitations and naïve usage of the game engine can cause unexpected behaviours. For example, the ActorLocation instance is shared across all GameMaps instances of a World object, which means it is possible to get the location of actors from other maps. This is usually impossible due to the way the game engine is programmed, but if functionalities are extended without care, we can see this causing unwanted behaviours.
- We believe an ideal game engine should allow developers to implement the desired features without limiting constraints which might lead to developers spending more time on understanding the engine and debugging more than designing the game.