Design Rationale For RobynsBizarreAdventure, part 3

Herein we detail the various design decisions made in order to implement the spec (and extras) of Assignment 3 in an extensible and maintainable manner. Repetition of code especially was aimed to be avoided as there were a lot of extra features added this time around.

Building On Assignment 2’s Design

In working on Assignment 2 and 3, it became very clear that the design we had come up with was really quite good - for the most part everything detailed in Assignment 3 was incredibly easy to implement (for example, all new dinos are just subclasses of the pre-existing abstract dino class). As such, very little has changed in terms of overall design in the system.

Upgraded FollowBehaviour

- FollowBehaviour now uses Predicates in order to search for various things an actor might be interested in
- Different kinds of items, ground or other actors could be of interest
- The class has been generified to support this extensibility
- This code gets used in a lot of places, so centralising searching into one behaviour was very useful

Util Class

- During assignment 2 we created an algorithm to search the map for edible items of interest. That was later moved into FollowBehaviour once it was generified and extended. However - there were other places where this code was useful, for example, using this search was our idea for implementing ranged attacks - and so the code got moved to Util so it can easily be used elsewhere.
- Util class also contains a method that gets used in the new Menu, since its a generic function that could be used in any part of the UI. It assigns and displays key shortcuts, essentially. Extensible to work beyond calling actions, which lets us have submenus.

Quest System

- The quest system implemented relies on the use of the lastAction argument passed in during playTurn
- To determine whether the desired action has been completed, the menuDescription of the two actions is compared (this avoids the problem of comparing classes, where for example any BuyActions would be the same, regardless of the item to be bought
- This allows for a huge amount of variety without stretching the bounds of the game engine

The one issue is that you can only accept the rewards for a quest the turn after you complete the action. This could be fixed by keeping a list of all the actions completed but I decided it wasn't worth the extra time to process through the whole list.

New kinds of Attacks

- There are 2 new kinds of attacks - ranged and radial
- Ranged attacks have their own behaviour that uses the search algorithm to find suitable targets, and returns these to the player for their actions list
- Radial attacks have their own Action, that gets the targets around the actor and then executes and returns the strings of attack actions for each actor found
- Again, this diversifies gameplay but integrates well with the existing engine

New Ground Types

- So since we added Water, and we don't want land dinos to be walking around on it, or water dinos to be swimming around on the land, we decided to use the skills system
- Each ground tile has skills that determine what kind of terrain they are
- Then, actors have their own skills that say which terrains they can go on (flying dinosaurs have both land and water, water dinos have water and land dinos have land)
- Giving us a super simple check in canActorEnter(), simply checking if the actor has the right skill.
- By doing it this way, it made it easy to add a new item, WaterBoots, that allows the player to walk on water by having the required skills.

Ending the Game

- The game is finished when the player character is removed from the game
- To include the required 3 game endings, the player is removed from the map under these circumstances:
  - When the quit game option is selected in menu
  - When the player is killed
  - When a captive bred, adult TRex is on the map (more info below)

The TRex problem

- The spec states that when a captive bred, adult TRex is on the map the game should end. It was a bit of a challenge to implement cleanly, but I think we did a pretty good job under the circumstances:
  - In breed-behaviour, baby dinos get the string "Jr." appended to their names. This would be our check for if a dino was bred.
  - All dinosaurs have an attribute called dinoStage to indicate if they are babies or adults. This let us check if the TRex was an adult or not

    - Player has a behaviour that every turn, uses the search algorithm to see if there is a T-Rex on the map who matches these conditions. If the check succeeds, the player is presented with the option to end the game

Now, unfortunately there are a couple of problems with this: mainly that it can only check the current player map. And also, if you instantiate a TRex with a name like, “Steve Jr.”, you will win immediately. But that requires the source code so it’s not really a problem for players, since you can’t name dinosaurs in game.

Overridden World

- World was subclassed in order to allow for 3 main things:
    - Periodic enemy spawning
    - A welcome screen, with an option to begin and also to quit
    - The ability to connect maps together
- The enemy spawning was an essential feature in terms of the extras we wanted, and a welcome screen was a nice QoL change. Connecting maps was part of the spec. Only 2 methods were overridden, run and processActorTurn. Everything else was left untouched in the super class to preserve original functionality.

Using inbuilt interfaces

There were several times when the in-built interfaces were used, as this keeps everything we add easily extensible.