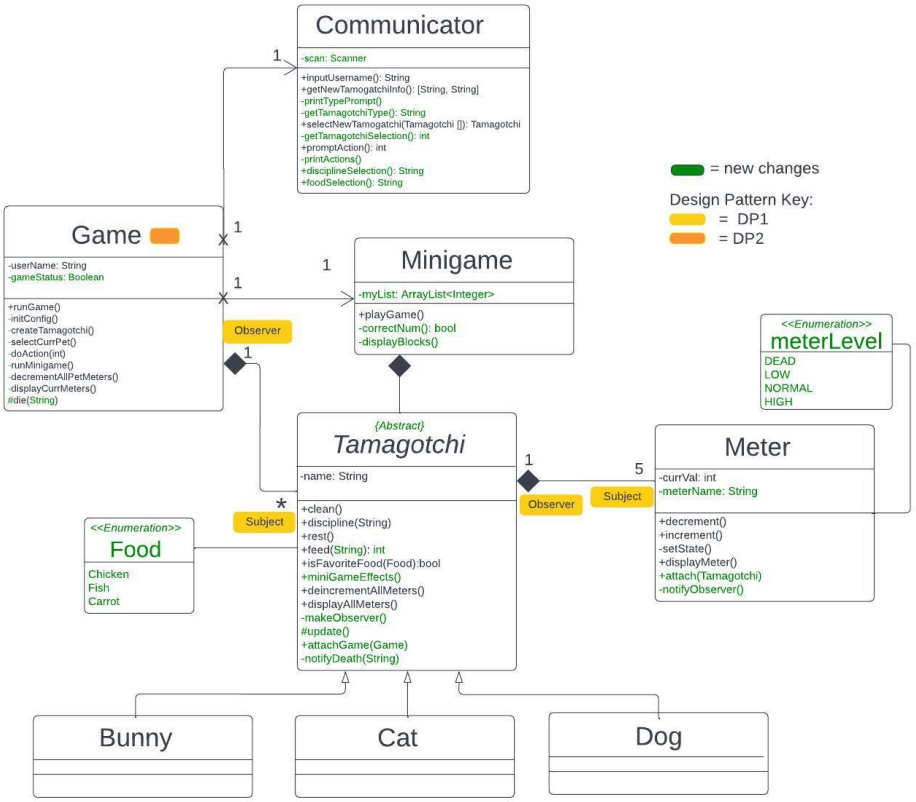
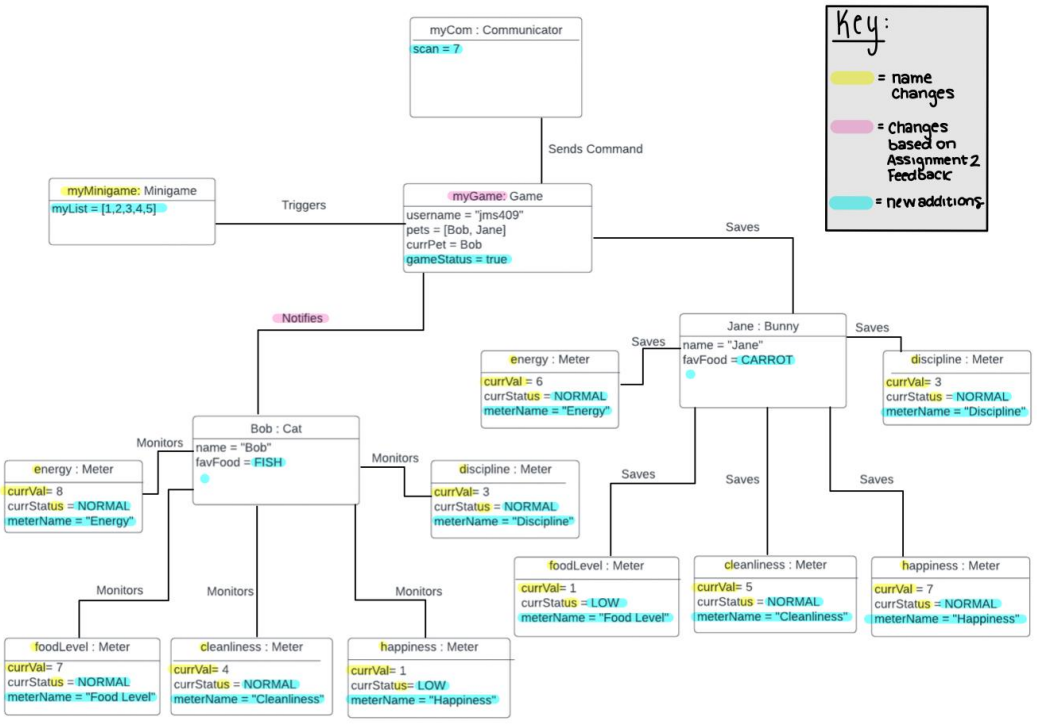
Class Diagram

Author: Casey Goldberg



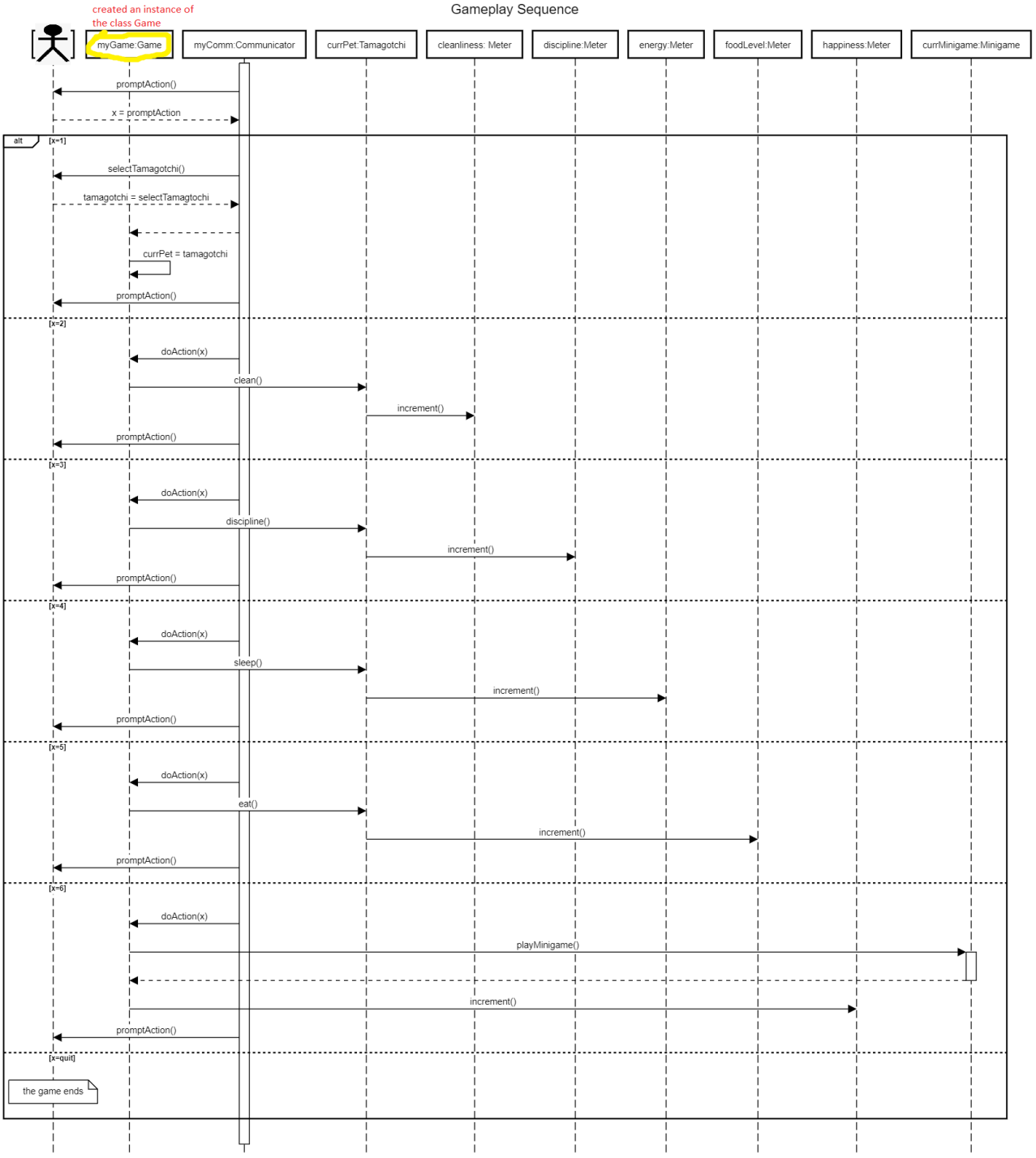
Object Diagram

Author: Jeannemarie Milmerstadt

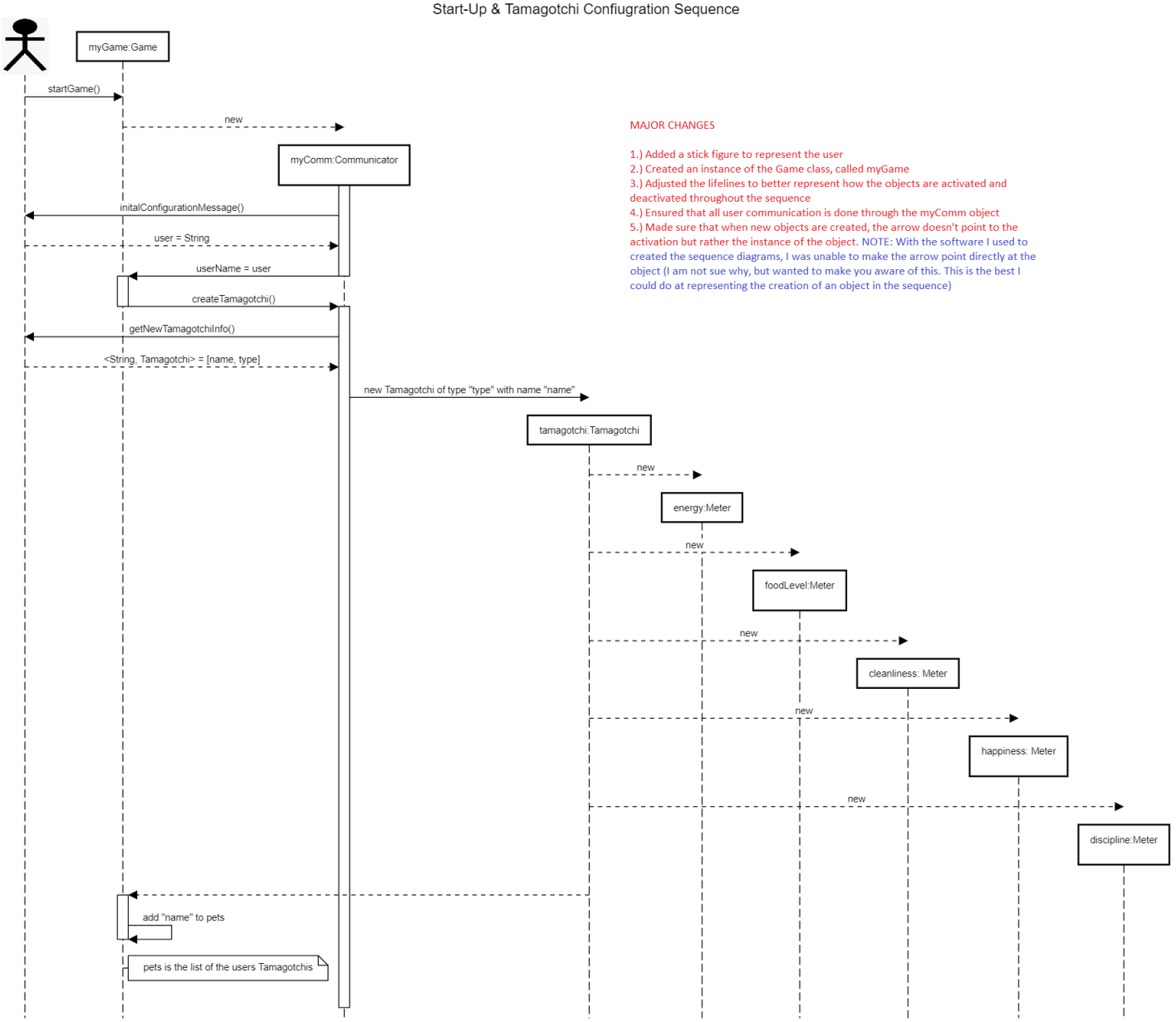


Sequence Diagrams

Author: Nick Cannone



Start-Up & Tamagotchi Confiugration Sequence



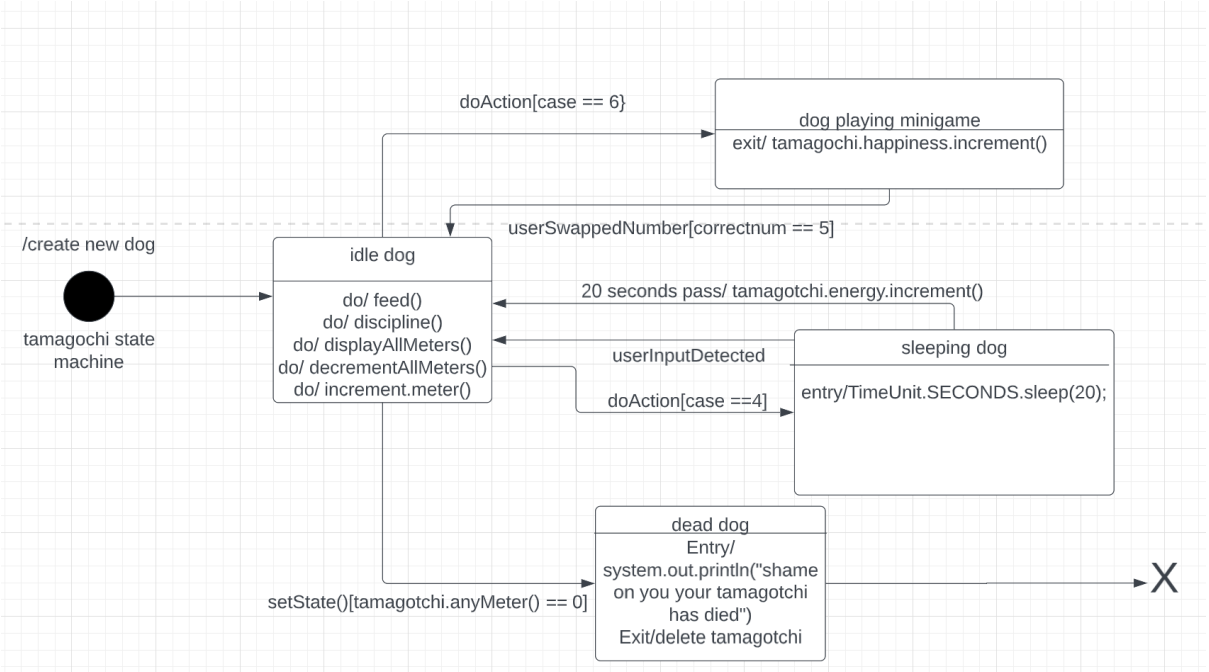
MAJOR CHANGES

1.) Added a stick figure to represent the user
2.) Created an instance of the Game class, called myGame
3.) Adjusted the lifelines to better represent how the objects are activated and deactivated throughout the sequence
4.) Ensured that all user communication is done through the myComm object
5.) Made sure that when new objects are created, the arrow doesn't point to the activation but rather the instance of the object. NOTE: With the software I used to created the sequence diagrams, I was unable to make the arrow point directly at the object (I am not sue why, but wanted to make you aware of this. This is the best I could do at representing the creation of an object in the sequence)

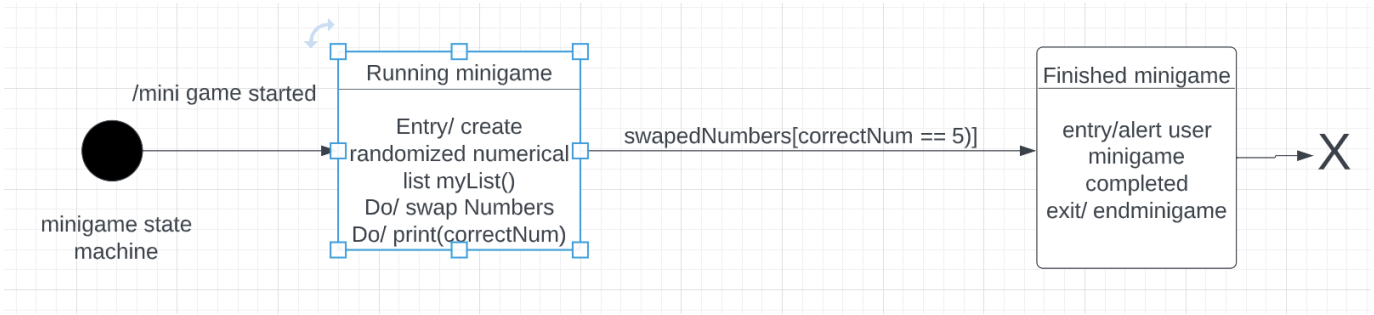
State Machine Diagrams

Author: Peter Schulman

Example: Tamagotchi States



Example: Minigame



Design Pattern Applications

Author: Casey, Nick

	Design Pattern 1
Design pattern	Observer Pattern
Problem	In our game, it is crucial that the Tamagotchi pet knows when one of its associated meter levels is at a High, Low or Dead level so that it can inform the user and the user can use this knowledge to select the most advantageous command next. The Game class also needs to know when one of its active pets dies (one of its meter levels hits 0) so that it can delete the pet from the game and prevent the user from giving it more commands. We wanted to implement these state change notifications without having to have the higher-level classes continuously check the state of its associated classes.
Solution	Implementing the observer pattern allows the Meter class and Tamagotchi class to notify higher level classes that they are associated with that their newly-changed states imply danger/death for the tamagotchi. Instead of requiring the tamagotchi object, for example, to constantly check its meter levels, we implemented the observer pattern to notify the tamagotchi pet as soon as one of its meters changes to a dangerous level. This same logic applies to the Game/Tamagotchi observer/subject relationship.
Intended use	This pattern implementation does not affect any runtime sequences. Since all meters are only associated with one tamagotchi and all tamagotchi's are only associated with one game session, we don't need to implement attach and detach methods that would allow us to add and remove observers at run-time as many observer pattern implementations include. involved in the applied design patterns (you can refer to small sequence diagrams here if you want to detail how the involved parties interact at run-time. This implementation only affects the inner-workings of the code by reducing coupling and increasing efficiency.
Constraints	Doesn't allow for multiple observers, but we don't require that for our system anyways.

	Design Pattern 2
Design pattern	Singleton
Problem	We want to avoid other Game objects being created, so need to find a way to ensure that only one can ever be created.
Solution	Made the Game constructor private, and created a private static variable called INSTANCE that is the only construction of a Game object. We then created a public method called getInstance() that calls the instance whenever we need to use the Game object.
Intended use	In our main method, we will only ever use the single instance of the Game object, rather than creating a new one every time the program is run.
Constraints	The possibility of having multiple games running at once.