Tamagotchi Pet Game-
-Post Report.

Authors- Robert Loomes 55938778,
Mna Chalabi 11322914

Structure

Base classes used- Pet, Food, Toy. These are called upon to create single Objects that, whose variables are maintained internally within the class methods. Most of these methods can be called from the main GUIfinal class.

A limited choice at Pet, Food, and Toy objects are available to the player (6 of each) and are all initialized at game start and held in the interface gameObjects. This means that 18 objects are loaded right away. This could potentially create loading issues if one were to ever change the game to house a lot more than 18 object choices.

In the case of the Food class, only 6 of theses objects exist, so only 6 references are ever made, despite the player potentially owning much more than that. This is because Food objects are never unique, they are merely added or removed from the player when bought used. A simple counter is used to track how many Food objects the player owned at one time.

In contrast, in the Toy class, objects created here each have be unique due to the presence of the durability attribute. The way unique references to these pre-made objects waere acheived was through a second cloning constructor in the Toy class. This lets players contain multiple unique copies of a pre-made Toy object.

Pets are similarly established using cloning constructors, as pets obviously need to be unique to each player.

The species subclass (of Pet) give each species unique modifers e.g a multiplier that makes the pet more or less tired after playing with a toy. All of these multipliers are already in the base Pet class, but are all initialized to 1, which makes it so that the Pet class can reference multipliers in it’s equations, but is overwritten when evoked by multipiers given by the Species subclass.

The alternatives we had considered to this design choice were creating 6 different Pet class, with all unique modifers that are unique; or requiring all the multiplier parameters on object creation. The former was not favourable due to the large amount of duplicate code required, and the latter beacuse we felt that creating objects with 10+ parameters was not a clean and understandable design choice.

The higher classes in the heriechy were the Player and GUIfinal classes.

The Player class is mainly used for creating Player objects and contain a few unique variables, and holding Maps/Lists that contained Pet/Food/Toy objects. Most of the methods in the Player class are simply adding or removing objects to these Maps/Lists.
No other design choice was realistically considered here.

GUIfinal which is the overlying class that evokes all of the other classes, contains all the GUI elements created in Swing, as well as all the main logic in the game revolving around the other lower classes. This leaves the class very large, with over 2000 lines of code for a fairly simple game. An interface was considered for containing most of the Swing elements, was was decided against due to the issues caused with Swing not being able to work with a class/interface GUI combo cleanly. This deisgn would be fine if coding all the GUI manually, but ultimately for future maintance purposes we decided to contain all elements in once large class. Maps were the main collection choice to contain the majority of the objects in the game. This let make things are little more optimal by not having to iterate through objects as much, and was helpful having a key/value pair to reference e.g key (Toy object)/ value (Toy durability).

The unsorted nature of the Maps were not a concern as Toy/Food orders in the Player collections were not a concern as it was not relevent in this game.

JUnit test files were written for all the base classes, which made sure all the the main methods were working correctly e.g Food items being added/removed to players.
We felt that test coverage was reasonable high. There was a low potential for exceptions overall due to the GUI being event driven, and any input required usually only accepted 1 type data type.

Feedback

This project was a learning experience, so a lot of design choices were improvised on the go, since we had no previous experience to draw from.

One of the main issues I had was converting the command-line application to the GUI application. The sequential logic did not transfer cleanly into the GUI event driven methods/buttons. The command-line was dominantly read by user input which checked that the input was valid, and then used switch/cases to evoke other methods from that input. This wasn't really relevant in the GUI design, as input errors were handled by pop-out windows, and user choices were driven by button events.

Most of the lower classes orignally output strings to the system based on what was given to it when called from the command-line app. While this was simply a case of re-wiring the output to the GUI labels etc, a lot of the lower class logic had to be moved directly into the GUI so that it could output directly.

Writing the GUI from scratch almost would have been easier.
In hindsight, most of the logic should have been contained the command-line game environment already, with the lower classes having no output. The switch/cases wern't horrible, and could have still existed in the GUI with a lot of work, but felt that it didn't fit the overall structure of the program.

In the future, more care should be taken to keep written logic updatable and transferrable to any setting it's in.

Robert
50% contribution.
Key contributions included class design, structure choices, inheritence design, gameplay design.
Was largely responsible for any algoritms or collection methods contained within the code.

Signed:

Mna
50% contribution.
Key contributions included unit-testing, GUI layouts, art design, gameplay design.
Was largely responsible for making sure classes worked as intended, as well as reporting bugs/ making sure nothing was largely intrusive to the overall design goal.

Signed: