CS2212B Group Project Specification

Version 1.1.0

Winter Session 2025

Important!

At the end of the project your team will review each others' contributions based on your team contract. To pass this course you must obtain a grade of at least 40% on this peer review. If you are having any issues with your team members, they must be reported to the course instructor as soon as possible (this can't be left until the last weeks of the course).

1. Overview

Virtual pets are digital companions that simulate the experience of caring for a real pet, providing players with a range of responsibilities like feeding, grooming, and interacting with their pet. These simulations, popularized by games like Tamagotchi, Neopets, Nintendogs, and Bitzee, give players a low-stakes environment to manage daily tasks and routines associated with pet care. The success of these virtual pet games lies in their ability to engage users through simple, repetitive tasks that grow more rewarding over time as the pet's well-being reflects the player's efforts.

For this project, students will design and implement a virtual pet application using the Java programming language. The game will involve managing a digital pet's needs, such as hunger, happiness, and health, while encouraging users to maintain regular interaction. Although the primary aim is to create an engaging and interactive game, there is an educational element that teaches players simple life skills like responsibility and routine management.

Teams will have the flexibility to design their virtual pet, its personality, and some of the specific gameplay mechanics that will drive the player's interaction. For example, players might need to feed their pet regularly, play with it to keep it happy, and ensure it stays healthy. While the game does not need to feature advanced graphics, it should include a rudimentary Graphical User Interface (GUI) to facilitate player interaction and provide a visually appealing experience.

This document outlines the technical specifications and requirements for the virtual pet project, including project milestones and deliverables. Further details on the specific work products required at each milestone will be provided in subsequent documents. The focus of the project is on building a functional, engaging virtual pet game that showcases your understanding of software development principles using the Java programming language.

2. Objectives

This project is designed to give you experience in:

- Applying the principles of software engineering towards a real-world problem.
- Working with, interpreting, and following a detailed specification provided to you.
- Creating models of requirements and design from such a specification.
- Implementing your design in Java and having to deal with decisions made earlier in the design process.
- Creating graphical, user-facing content and applications.
- Writing robust and efficient code.
- Write good, clean, well-documented Java code that adheres to best practices.
- Working on and with a team.
- Reflecting on good/bad design decisions made over the course of the project.

The project is intended to give you some freedom in design and programming to explore the subject matter, while still providing solid direction towards reaching a specified goal. See section 5.5 for how to request changes from the project specification for your team.

3. Requirements

Your project will need to adhere to a collection of functional and non-functional requirements. In essence, the functional requirements outline what your application will need to do, while the non-functional requirements specify how you're supposed to go about doing things.

As a team you will also gather, refine, and develop additional requirements specific to your game but all teams must satisfy the requirements in this section.

3.1 Functional Requirements

Functional requirements include required functionality, as discussed in the sections below. You must implement all of the required functionality for your project to be considered complete. How the below functionality is delivered is for the most part up to you, but must follow proper software engineering practices.

While we will not be grading visual appeal or aesthetics directly, if things slide to the point where your application is unintuitive, difficult to use, or unreadable, then this could impact your overall grade. User experience (UX) and the usability of your application are important factors that will be considered and are not equivalent to visual appeal or aesthetics.

3.1.1 User Interface

Your application needs to provide a Graphical User Interface (GUI) that players of the intended age group can navigate easily. This interface should be at least partly mouse based, inform players about the pet's state, and respond to the user's inputs. The following UI requirements must be included:

Multiple Screens: The game must have at least five distinct screens/pages including: a main menu screen (requirement 3.1.2), a gameplay screen, an instructional/tutorial screen (requirement 3.1.3), a load game / new game screen (requirement 3.1.4), and a parental controls screen (requirement 3.1.11). Your team is free to add additional screens as required.

Mouse-Based Interaction: The interface must support mouse-based interactions, enabling users to navigate through the game's screens, make selections, and control game elements primarily with a mouse (e.g., click a button that feeds the pet, etc.).

Keyboard Shortcuts and Commands: The interface should include keyboard shortcuts and commands for common actions to facilitate ease of use, especially for power users or when a mouse is not the most efficient input method or for accessibility reasons.

Where relevant, include keyboard controls for gameplay actions (e.g., pressing 'P' to pause, 'F' to feed, 'G' to give a gift, 'Esc' to return to the main menu, etc.).

Feedback Systems: The UI must provide visual or auditory feedback in response to user actions, such as clicking buttons or entering commands, to confirm that the desired action has been taken.

Your GUI can be created using a standard library such as Java Swing or JavaFX, be a custom interface developed by your team, or some combination of the two. It is acceptable to use tools to help generate and build your GUI code (such as your IDE's GUI Builder[1] or Scene Builder) so long as you cite their use in your documentation and code.

Java Swing is easier to use and learn, but a bit outdated and less customizable. JavaFX is newer and more customizable, but has a higher learning curve and more dependencies to setup (as it is not built into Java like Swing is).

Teams are expected to put some effort into learning the tool kits and libraries they will be using. They will not be covered in-class. This is intended to simulate how software development works in the real world, where you will be expected to read documentation for new libraries and APIs that you may have not experienced before and learn how to use them in your projects. Some links and resources will be shared in the Group Project unit on OWL to get you started, including a simple GUI demo.

[1] Many Java IDEs such as NetBeans, IntelliJ, and Eclipse come with tools or have plugins to assist you with building a GUI and generating some of the code for you.

3.1.2 Main Menu

Your game must have a main menu screen that displays the title of your game and provides options for the user to select from. These options must include 1) start a new game, 2) load a previously saved game, 3) tutorial or instructions, 4) parental controls, 6) exit. Your team may add additional options. Each option should take the player to a new screen in your application.

The main menu should also display: 1) some graphic or visual that is representative of your game, 2) list the names of the developer's who created it (this would be the members of your team), 3) your team's number, 4) term it was created in (e.g. "Fall 2024), and 5) mention that this was created as part of CS2212 at Western University.

3.1.3 Instructions and/or Tutorials

You must include at least one screen with detailed instructions on how to play your game and use this application. This can be text based, include pictures, or be an interactive tutorial. In all cases it must be comprehensive enough to inform players with no previous experience with your software, how to play your game. It should document all major features from the players perspective (i.e. it does need to include low level technical details).

When creating your instructions or tutorial, you should keep in mind the target demographic for your game and write these instructions appropriately for their age level.

3.1.4 New Game & Pet Selection

When a new game is selected, the player is presented with a choice of picking from at least 3 different virtual pet types. An image representing each pet type should be displayed on the screen and well as some basic information about each pet type. The user should be able to select one of the pets and give it a name.

Once named, a new save file (see requirement 3.1.5) is created, and the user is brought to the main game screen.

Each pet should have slightly different characteristics that impact gameplay. For example, one pet might become hungry faster than the others, another might require less gifts to stay happy, and so on. The characteristics of each pet are up to your team, but they should be noticeably different in terms of game paly and described to the player when selecting a pet.

As mentioned, each pet should be represented by an image. The image should match up with the sprite[2] used to represent the pet in the game. Your team is not required to create their own images, and it is acceptable to use premade sprites found online so long as you credit where they were obtained from.

A great resource for this is https://www.spriters-resource.com which includes sprite sheets you may find useful such as this set of Tamagotchi Smart sprite sheets: https://www.spriters-resource.com/lcd_handhelds/tamagotchismart/. The site contains many other sprite sets, and you are not limited to just using Tamagotchi sprites or sprites from spriters-resource.com. Keep in mind that you will need images of each pet showing different expressions or emotions for requirement 3.1.10.

3.1.5 Save/Load Game State

It is important to provide players with the ability to save their progress and the current state of their pet and inventory (requirement 3.1.8). To achieve this, the game must feature a Save Game State mechanism that specifically captures the player's progression and the current state of the pet. This should include any vital statistics about the pet, the type of pet being used, the pet's name, their current health, hunger, the current score, and the player's inventory, etc. The primary goal is to enable players to resume their progression from their last play session.

To facilitate this, you can either automatically save progress after set check points or provide players with a manual save option that is easily accessible somewhere in the GUI.

When restarting the game, players should have the capability to load their previously saved state. This loading process should load the pet back into the same state it was in when the game was last saved.

A clear confirmation message should be displayed each time the game state is saved, reassuring players that their progress has been successfully recorded.

Your Save/Load system must be designed in such as way that a player can have multiple pets that they can switch between by saving and then loading a new pet. You may either have the player select a file that contains the save state to load or have a set number of save slots for the user to choose from (at least 3).

[2] A sprite in games is a two-dimensional image or animation that represents characters, objects, or other elements within a virtual environment. In virtual pet games, the sprite typically serves as the visual representation of the pet, displaying its actions, expressions, and movements on the screen.

3.1.6 Vital Statistics & Rules

During gameplay the following statistics about the pet should be tracked and displayed to the user on the gameplay screen:

a) **Health**: A positive numerical value that represents how healthy the pet currently is. A zero value is dead, a maximum value is perfectly healthy.

b) **Sleep**: A positive numerical value that represents how tired the pet is. A low value is very sleepy, and a high value is perfectly awake.

c) Fullness: A positive numerical value that represents how hungry the pet is. A low value is very hungry, and a high value is perfectly full.

d) Happiness: A positive numerical value that represents how happy the pet is with the player. A low value is angry, and a high value is very happy.

These statistics can be displayed to the player as numbers, a progress bar, or pictorially. The choice is up to your team, but it must be done in a way that is meaningful to the player.

Each statistic should have a maximum value that may be different for each type of pet. All of the statistics **except health** should slowly decline at a set rate during game play. This rate may be different for each type of pet and for each statistic (e.g. sleep points might decline faster than fullness points).

When a statistic is less than 25% of the maximum value, a warning should be visually indicated to the user such as changing the color of the text or progress bar to red, flashing the text or progress bar, making a sound, etc.

When a statistic reaches zero, the following should happen depending on the statistic:

a) **Health:** If health points reach zero, the pet dies (enters the dead state) and the game is over. The sprite of the pet should change to one that indicates it is dead. The only option the player will have will be to start a new game or load a game.

b) **Sleep:** If sleep reaches zero, a health penalty is applied (a set number of health points are removed), and the pet will fall asleep and can no longer be interacted with. In the sleeping state the sleep value will slowly increases until it hits the maximum value. Once the max is reached, the pet wakes and returns to its normal state. During the sleeping state the other statistics still decline normally. When in the sleep state the sprite used should be changed to one of a sleeping pet.

c) Fullness: When fullness reaches zero**,** the pet enters the hungry state and the rate that happiness decline should be increased. Health points should also start decreasing until the pet exits the hungry state. The exact value of the increases/decease is up to you and can be dependent on the pet type. Once the appetite value is above zero, the pet exits the hunger state.

d) Happiness: When happiness reaches zero, the pet will enter the angry state. In the angry state the pet will refuse all commands of the player except ones that increases happiness

(see requirement 3.1.17) until their happiness is at least ½ of the maximum happiness value. The player will be forced to only issue commands that increases happiness to exit this state. The sprite of the pet should change to indicate it is angry.

There is some room for interpretation in the above rules. For example, what happens if the pet is sleeping and angry at the same time? It is up to your team to make and document your design decisions regarding your interpretation and implementation of the above rules and how they interact. You might need to do some play testing to ensure these are balanced, so making them easily configurable might be a good idea.

In addition to changing the pet's sprite, icons or text should be displayed to indicate all active states in the GUI. For example, if the pet is in an angry state and a hungry state, you might display an angry icon and a hungry icon in your GUI.

All vital statistics should be saved when the game is saved as per requirement 3.1.5.

Your team may add additional statistics and rules regarding them if they wish, but at a minimum the ones described in this section must be present.

3.1.7 Commands

The player interacts with the pet by issuing commands. The commands available are dependent on the pet's state (see requirement 3.1.6). At a minimum you should provide the following commands:

a) **Go to bed:** The pet enters the sleeping state (see requirement 3.1.6) and remains in that state until their sleep value researches the maximum.

b) **Feed:** The pet is fed food, and their fullness value increases (they are less hungry). The player should be able to select from different food types each with their own properties (e.g., some might add more to the fullness value than others). The food available should be based on the player's inventory (see requirement 3.1.8).

c) **Give Gift:** The pet is given a gift by the player and their happiness value increases. The player should be able to select from different gift types each with their own properties (e.g., some might add more happiness than others). The gifts available should be based on the player's inventory (see requirement 3.1.8).

d) **Take to the Vet:** The player takes the pet to the vet and the pet's health is increased by a set amount. Once this command is used, it should be unavailable for a set time until a cool down is over.

e) **Play:** The player plays with the pet increasing the pet's happiness by a set amount. Once this command is used, it should be unavailable for a set time until a cool down is over.

f) **Exercise:** The player takes the pet for a walk (or what makes sense for your pet type). This lowers their sleepiness and hunger values but increases their health points.

Based on the pet's current state, the following commands are available to the player:

1. **Dead State:** No commands.
2. **Sleeping State**: No commands.
3. **Angry State**: Give Gift and Play
4. **Hungry State:** All commands.
5. **Normal State**: All commands.

The states should be given precedence in the above order. For example, if the pet is in the sleeping and hungry state, no commands should be available. If the pet is angry and hungry, the only commands available should be Give Gift and Play.

When a player issues a command, there should be some indication to the player that something happened (e.g. playing an animation, changing the pet sprite, a sound, or an image/text flashing on the screen).

3.1.8 Player Inventory

The game should keep track of and display the player's current inventory of items. Items fall into two categories:

a) **Food Items:** This category of items increases the pet's fullness value (making them less hungry). Each food item should have different properties and increases the fullness value by a different amount.

b) **Gift Items:** This category of items increases the pet's happiness level. Each gift item should have a different property and increase the happiness level by a different amount.

The game should have a minimum of three items from each category.

The inventory should display a list of the items currently in the player's inventory including a count of how many they have of each item type. This can be a text-based listing or displayed pictorially (e.g. with icons). The inventory can be its own screen in the application or part of the gameplay screen.

Using the Give Gift and Feed commands (from requirement 3.1.7) should lower the amount of that item in the inventory.

The game should include a mechanic that allows the player to obtain items. This can be as simple as giving the player items periodically after a set amount of time, rewarding them with items at set score levels (requirement 3.1.9), or as complex as requiring them to complete a minigame first. The details of the mechanic are up to your team, but there must be a way of obtaining items.

The players inventory should be saved as per requirement 3.1.5.

3.1.9 Keeping Score

The game should keep track of the player's score. The score should start at zero and increase when the player does a positive action such as feeding the pet, giving it a gift, or playing with it. Some actions may also reduce the score, such as subtracting points if the pet had to be taken to the vet. The exact details of what adds and subtracts score, or how much is up to your team but should be clearly documented.

The current score should be displayed on the game play screen and saved in the save file as per requirement 3.1.5.

3.1.10 Pet Sprite

The pet should be represented visually by a sprite. The sprite can be a premade sprite that you found online (a great resource for this is https://www.spriters-resource.com) or one created by your team. If you use an existing sprite, you must credit where you obtained it in your documentation and code.

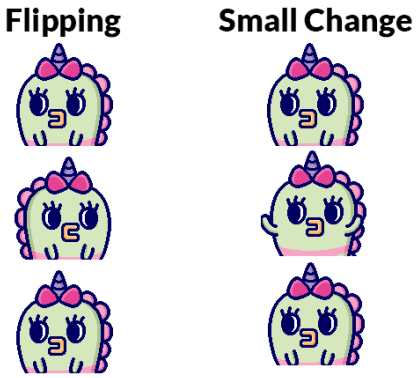
The sprite should change based on the pet's current state.

Example:

The sprite should also be different for each pet type based on the player's section in requirement 3.1.4.

The sprite is not required to be fully animated but should switch periodically on a timer or randomly with another sprite to give the illusion of movement. This could be accomplished by flipping the sprite horizontally to switching it back and forth with a sprite with a small and simple change.

Examples:



3.1.11 Parental Controls

The game should provide a separate area accessible via the main menu (requirement 3.1.2 Main Menu) that provides controls and statistics for the parent of the player. This screen should be password protected to prevent the player from accessing it. You may have a hardcoded password or pin; user accounts are **not** required.

Parent users are also allowed to play the game as a player would and can perform all tasks a player can.

The features of the parental controls are broken into three sections:

3.1.11.1 Parental Limitations

The parent should be able to set certain times of the day the player is allowed to play the game as well as enable or disable this feature. If the feature is enabled and the time is not in the allowed range, the player should get a message about not currently being allowed to play the game when they attempt to load a game or start a new game.

The time setting and if this feature is enabled or disabled should be saved and persist when the application is restarted.

3.1.11.2 Parental Statistics

The game should keep track of the players total play time as well as their average play time per gaming session and display this in the parental controls screen. These statistics should be saved and persist if the application is closed and reloaded. Any time the application is running should count towards the total play time.

A session is considered the time between the application is started and exited. You can assume the game is always safely exited (e.g. by clicking the exit button in the main menu) and not terminated in a way that would not allow you to save this data.

The parent should have the option of resetting the total and average play times back to zero.

3.1.11.3 Revive Pet

The parent should be able to select a save file or save slot (depending on how you implemented requirement 3.1.5) and revive the pet back to a normal state and with maximum values for each statistic.

3.1.12 Housekeeping & Error Handling

Your application also needs to do basic housekeeping things that are part of any decently put together application. For example, the user must be able to exit your application cleanly and data must be saved correctly on exit such that it is available the next time the application is started (e.g. save states should be persistent, parental statistics should not be lost, and so on).

If you allow the user to minimize or maximize the window, UI elements should scale correctly. It is ok if you disable this and do not allow the user to minimize or maximize the window. You may also force the game to be full screen or a set window size.

Any errors that could be raised by your application should be handled and clear messages in simple English should be shown to the user, that may help them understand and correct the error. Ideally, your application should avoid crashing without explanation. Any user input that could potentially be incorrect or cause faults in your software, must be validated and checked for errors.

There must always be a way to navigate between the screens of the application. The user should never get stuck on a screen with no way back to the main menu.

You may add other housekeeping functions as you see fit to help support your application. How you choose to make these available to the user is up to you.

3.1.13 One Extra Functional Requirement ← Don't forget about this!

Your team must pick a nontrivial feature to add to the game beyond what is described in this document. The feature should not be trivial to implement and should be a sensical addition to the game. The feature is up to your team, and you are not limited to the following list but here are some suggestions:

- Add a minigame when issuing a certain command (e.g. have a minigame when you play with the pet).
- Add sound to the game including background music and sound effects.
- Add more states, vital statistics, and commands for the pet that have their own rules and effects.
- Make the animations more advance and have the pet wander around the screen and interact with objects.
- Have additional unlockable pets to pick from. Unlockable pets should have new properties and set goals that need to be accomplished to be unlocked.
- Add more items, currency, and an item store.
- Add additional parental controls and statistics such as an overall play time limit, a play time per day limit, more advanced controls for when the player can play (e.g. only one week ends).

- Add different rooms the pet can be in with different properties (e.g. pet gains sleep faster when sleeping in the bedroom) and objects to interact with.

Part of the evaluation of this requirement will be based on the extra effort your team puts into the additional requirements. It must not be trivial and should require some work and planning to implement.

3.2 Non-Functional Requirements

Your application will need to adhere to the following non-functional requirements. These requirements will be taken into consideration in the assessment of your project.

3.2.1 The application must be developed for and work correctly in **Java 23** (current version as of writing) or newer.

3.2.2 **Your game must utilize an object-oriented approach** and make use of design patterns. You will be required to explain which design patterns you are using and why.

3.2.3 **There should be no objectional or offensive content** in the game. The content should be appropriate for players of age 7 and up.

3.2.4 The application is **required to have a Graphical User Interface** (GUI). This interface must be user-friendly and aligning with best UX practices to ensure intuitiveness and ease of navigation. It should present a cohesive design with clear labeling and a consistent visual language to minimize user effort and facilitate engagement. The GUI's effectiveness will be part of your project grade.

3.2.5 The application must store all data (e.g. vital statistics, player inventory, etc.) **locally** and not require an internet connection. The file format is up to you, but some good options are JSON, XML, CSV, or TSV. Some of these formats have standard Java framework methods or third-party libraries to handle them for you, others will require your own code for working with them. While a third-party library might make some things easier or more efficient, they can also be complex and difficult to use, costing you more time in the end than you saved. A database (e.g. SQLite, MySQL, etc.) can only be used if you are storing the data locally and no internet connection is required (using a database is strongly discouraged for this project as it will add unnecessary complexity).

3.2.6 Your game should not use any third-party libraries that are not freely available. All tools and libraries used must be easy to obtain and install (or ideally included with your application). The TA marking your project will need to be able to run it to test it, so you will be responsible for providing instructions on compiling and running your application including any required libraries.

3.2.7 **All code and files for your project must be stored in the GitLab repository** created for your team (details on this will be announced once available). **Your team must actively use this repository and not simply commit the files at the end of the project.**

3.2.8 **All design work and diagrams for your project must be stored and developed on the Wiki on GitLab** (details on this will be announced once available).

3.2.9 **All tasks and issues related to your project must be tracked using GitLab** (details on this will be announced once available) and be updated as the project progresses.

3.2.10 Your code must be commented using Javadoc. At a minimum every function/method should have a comment, and every file should have a comment at the top explaining it's purposes, author, etc. Code generated by a tool does not need to be commented other than a comment clearly citing the tool used to generate it. Any code not created by your team members must have a comment citing its source.

3.2.11 The majority of your code should be unit tested with a sufficient number of JUnit 5 tests. Code that can only be tested via a GUI actions or events does not need to be unit tested.

3.2.12 As a team you must choose and follow coding conventions and styles that you adopt (e.g. naming things, indentation, etc.). You must remain consistent in applying those conventions and styles across all files in the application.

3.2.13 The application must be executable on a Windows 11 system with a standard Java installation (Java 23 or newer), and each team member must be able to compile it and run it from a development environment they have access to. All team members must use the same development environment and tools as agreed to in your team contract.

3.2.14 The application must be self-contained and not create, modify, or delete files outside of the directory in which the application is installed, and subdirectories of this directory. Simply put the application should not alter or delete files that do not belong to it or were not created by it.

3.2.15 The application must present a visible response to every user action. Erroneous actions or actions that could not succeed for some reason must be met with a useful, professional error message.

3.2.16 The file size of the project as a whole should be under 500 megabytes.

3.2.17 The application should run efficiently and not use unnecessary computing resources. The interface should be responsive and not "lag" or freeze while the user is playing the game.

3.2.18 Your application must take accessibility into account when designing the user interface. Allowing tasks to be completed with both a keyboard or mouse would be ideal, as well as ensuring UI elements have a logical tab order. Colour use in the UI should also be carefully considered (e.g. for colour blind players).

3.2.19 Your code should be maintainable and as reusable as possible. Code should be structured to allow for easy updates and maintenance.

3.2.20 All content in the game, supporting documentation, design work, comments, etc. must be written in English. While your spelling and grammar will not be graded directly, any work that is unclear, incomprehensible, or otherwise not easily decipherable by the marker will be assigned a zero grade. All communications between team members should also be in English.

3.2.21 The application must be otherwise designed with sound software engineering principles in mind as discussed in the course.

You are also required to gather, refine, and document additional non-functional requirements related to your specific game idea and educational objectives.

4. Assessment

4.1 Teaching Assistant Marking

Your project will be assessed at set milestones throughout the term (tentative due dates for these project components are included in the course syllabus) as well as at the end of the term.

Teaching Assistants (TAs) will grade each project component based on a rubric provided by the course instructor. In addition to assessing the work submitted, TAs will also assign a grade for project management. Project management includes how well you function as a team, your use of the GitLab software, compliance with your team contract, your team minutes, your team's attendance at meetings, how tasks are assigned on GitLab, management of your code in your repository, and your use of the GitLab wiki to document your software.

At the end of the term, you will meet with your assigned TA for acceptance testing where you will demonstrate your software and show how it meets each of the requirements in this document. All team members must be present for this meeting and your performance during this meeting will be part of your "Project Video & Acceptance Testing" grade.

4.2 Peer Review

At the end of the term, you will assess and also be assessed by your team members in a peer review. This peer review is worth 10% of your final grade but you must obtain at least a 40% grade from your team members to pass the course.

Your team will also be responsible for creating a short video that demonstrates your software. This video will be shared with students from other teams. These students will provide a peer assessment of your work based on the video. This peer assessment will compromise part of your "Project Video & Acceptance Testing" grade.

5. Tips, Suggestions, & Advice

5.1 Files

The file format you use for saving the game's state or parental statistics is up to your team. The format need not be complicated and could be as simple as a plain text file with some values written in it such as a CSV or TSV file. You may also use more complex and powerful formats such as JSON or XML, or create your own. If you wish to use JSON, you will likely need a third party library such as Jackson or Gson.

We strongly recommend **not** using a SQL based database, as this would add a lot of complexity to your project and the functional requirements ask for everything to be local.

5.2 Sprites

You are not expected to create your own sprites. You can find many sets of sprite sheets to download on sites such as https://www.spriters-resource.com. For example, you can find sprite sheets from the game Tamagotchi Smart (a virtual pet game) here: https://www.spriters-resource.com/lcd_handhelds/tamagotchismart/

These sprite sheets contain different emotions, actions, poses for the virtual pets such as this one for Gaogaltchi:

Sprite sheets such as the one shown above are images that contain multiple smaller graphics or animations in a grid, used in 2D games to efficiently manage and display animations. By combining the different images, you can create new expressions and poses for your virtual pet.

You can take two approaches to handling this in your game. You can either load the whole sprite sheet into your game and manipulate it programmatically or alternatively you can edit the sprite sheet manually in an image editor such as GIMP or Photoshop. Doing this programmatically will require more complex image processing code but require less manual image manipulation. Doing this manually will require more time spent manually creating sprites in an image editor but will greatly simplify your Java code. Either option is acceptable for the project.

You are also not limited to using Tamagotchi Smart sprites. spriters-resource.com has a large collection of many different types of sprites and you are free to use any that are appropriate for the project. You may also use sprites from sites other than spriters-resource.com.

5.3 Graphics and Gameplay

Your game is not required to be graphically intensive or have complex gameplay mechanics beyond those described in the requirements. It is completely acceptable for the virtual pet to remain mostly static in one location and only flip the image back and forth horizontally

periodically to give the appearance of movement. The pet does **not** have to move around the screen or directly interact with anything on the screen.

Most of the game play will involve the player clicking buttons that issue a command to the pet (feed it, play with it, etc.). The player does **not** have to be able to control the pet directly.

5.4 GUI

Your team has free choice in the GUI toolkit/library you use, but you might wish to consider Java Swing. Java Swing is a bit outdated and has far less customization than modern toolkits such as Java FX, however, Java Swing is built into Java and easier to learn. Being built into Java means fewer dependencies to manage and less configuration required. Keep in mind that your GUI is being evaluated on useability and not visual aesthetics (so long as they do not impact useability), so choosing Java Swing will not be put you at a disadvantage.

You are free to use GUI Builders and tools to help generate parts of your GUI code. NetBeans and many other Java IDEs have built in GUI Builders for Java Swing, that could help speed your development and allow you to create GUIs visually.

We won't be covering Java GUI libraries in-class, but we will talk about general UX concepts such as useability and best practices. This is intentional as teams are expected to put some effort into learning the toolkits and libraries they will be using. This is intended to simulate how software development works in the real world, where you will be expected to read documentation for new libraries and APIs that you may have not experienced before and learn how to use them in your projects. Libraries and toolkits come and go, but the ability to understand technical documentation and learn new tools, will serve you your whole career.

5.5 Deviations from the Specification

In some cases, your team may wish to take an approach to the project that would require changing or altering the requirements in this document. Any significant changes or deviations from the requirements given in this document must be approved in writing by the course instructor. Such requests must be submitted to the course instructor (and not your TA) **via e-mail** well in advance of any deadline and not reduce the complexity or difficulty of project. All team members must agree to these changes unanimously.

6. Academic Dishonesty

All documentation for your project (including the requirements documentation, design documentation, testing documentation, UML diagrams, and wireframes) must be created by your team members this term. Using the documentation created by another team, from a past term, generated by AI, or for another course is a scholastic offence. While you are encouraged to discuss the project with other teams, you may not share or receive any documentation or diagrams from those teams. It is your responsibility to protect and secure your Western accounts to ensure no work can be taken without your knowledge and you must not store any work for this course publicly (all work should exist on your Western GitLab project).

In terms of code and implementation, you may use tools to generate code and the user interface, but this use must be cited in both your documentation and code comments. You may also use small exerts from publicly available code and examples so long as the use is cited (in both your code comments and documentation) and the code is not from another team or student who has taken this course in the past. Note that while using generated code is not an academic offence in this case (if properly cited), significant use of generated or borrowed code may result in a lower grade as your mark will be based on the code and documentation your team did write.

Violating the academic dishonesty policy will result in a zero grade for that project component/milestone for your whole team as well as all team members being reported to the department's integrity committee. It is important for your team self enforce this policy among your team members and report any concerns to the course instructor to avoid a penalty being applied to the whole team.