Tech Design Document

Last Updated: 11/19/24

Table of Contents

General Information

- Using Unity Version 2022.3.46f1
- The main branch should always be playable
- A second staging branch will be used for making branches and pull requests
- Features must undergo a code review prior to being merged into staging
- Leads will periodically review the staging branch before merging it into main
- Code style guide - except using underscores for private variables

Documentation

- **DOCUMENT YOUR CODE**
    - This includes both commenting and written documentation
    - Someone who has not seen your code before should be able to make sense of what it does
- Written code documentation will be split into internal documentation (for programming) and external documentation (for design)
    - Internal documentation should:
        - Explain, in English, why your code is structured the way it is
        - Explain, in English, how to safely work with and modify your code
        - List any dependencies your code has
    - External documentation should:
        - Briefly summarize what your feature does
        - Describe how it can be edited or worked with
        - List any dependencies your system may have
        - Inform a designer how to work with your feature without the need for programmer intervention
- Written code documentation should be done on a per feature/mechanic basis as opposed to being on a per branch/task basis
- These are living documents and should be updated as a feature is iterated upon

General Code Guidelines

- All files you create should have a header comment containing your name, the creation date, a space for the names of contributors, and a brief description of what the file does
    - /**
      * Author: [creator of file]
      * Contributors: [people who edited file, add your name if you edit]
      * Date Created: [date file was created]
      * Description: [brief description about what the script does]
      **/
- When modifying someone else's file, add your name to the header as a contributor

- If you adapt/take code from external source(s), place a link to that source in a comment above the code
    - Don't plagiarize
    - Also allows people reading your code to get further information if needed
- **Avoid public variables whenever possible**
    - If a variable should appear in the inspector while still being private, use the [SerializeField] attribute
        - Ex: [SerializeField] private int health = 100;
            - Will be editable in the Inspector
        - public bool TestBool { get; private set; }
            - Publicly accessible variable that can only be locally set
- Optimization
    - Avoid Update unless it's necessary
        - Avoid slow functions in Update, which include but aren't limited to:
            - GameObject.Find()
            - GameObject.GetComponent<>()
            - Camera.main

Code Maintainability and Expandability

- Code should be easily readable
    - This includes commenting code as well as maintaining consistent formatting
    - Classes, variables, and functions should have meaningful names that accurately describe what they do
    - Other people will end up working with your code. They should be able to read it and figure out how it works without your intervention
- Eliminate redundant code whenever possible
    - If you're using basically the same code in multiple places, consider turning it into a function
- Write modular code
    - Helps pieces of code to be more reusable
    - Ideally, functions and classes should be focused have one job
    - Smaller components are more manageable and easier to understand
- Avoid tightly coupling classes/objects when possible
    - Each system should ideally be standalone
    - The less objects are dependant on one another, the easier it is for systems to change
    - Unnecessary dependencies mean code is harder to debug
    - Avoid circular dependencies

Code Style and Syntax

- Follow this C# style guide except when stated otherwise in this document

- Names for private variables should be written in camelCase with an underscore before the name
- Lines of code should be at most around 80-100 characters long
    - This is not a hard limit, but rather a guideline. The most important thing is to avoid excessively long lines of code.
- Scripts do not need a copyright statement at the top
    - Instead use this header:
        - /**
            * Author: [creator of file]
            * Contributors: [people who edited file, add your name if you edit]
            * Date Created: [date file was created]
            * Description: [brief description about what the script does]
            **/
- Every function should have an XML comment above it
- Comment large blocks of code, especially ones whose function is not self-evident
- Any temporary or placeholder code should be commented
    - Use TODO or FIXME stubs to denote code that you know will need fixing or updating later
    - Can throw not implemented exception or debug error
- Guard Clauses in a single line are OK.
- Braces for fields can be on the same line
- Consts should be PascalCase, no underscores.
- Auto-properties can have their brackets on the same line.

File Organization

- Testing scenes should be named GYM_SceneName
    - These should be placed in the TestingGyms folder
- Folders and files should be given names related to their function
    - Folders and files should not be named after the IC who worked on them
        - Example: NickTestingScene would be a bad scene name
- Scripts related to the same mechanics or system should be organized into a folder

Branching

- **Branches will be used**
- All new branches should be made from staging branch
    - Branches should not be made off of main
- Each feature should have its own branch, even small features
    - Each branch should only modify the files needed for its feature
    - Bug fixes should also have their own branches
    - If two or more bugs are dependent on each other, they may be handled in the same branch

- Branch names should describe what the branch is used for (*Jira may change this*)
    - VfxImplementation would work as a branch name
    - NickBranch would be a bad branch name
- Once a feature is complete, pull request it into staging

Pull Requests

- A pull requests are made after your feature is complete and tested
- Pull requests are made into the staging branch and not main
- Leads will periodically review the staging branch and merge it into main
    - At this time, a build will also be made from main
- Branches can be updated while a pull request review is in progress without needing to make a new PR
- Documentation for a new feature/mechanic does not need to be written before a pull request is made, but should be completed before it is merged

Note: The following PR process will likely be modified as Jira - Github integration gets fleshed out.

PR Process - Reviewee

1) Finish your task
2) Merge staging into your branch and handle any conflicts
3) Make a pull request and fill out the description with:
    a) A link to related documentation (if it already exists)
    b) A brief description of the changes you made
    c) Details for how and where to test your feature
4) Make the pull request and move your task to be In Review
5) Go to the github-updates channel in Discord and make a thread off of the notification for your pull request
    a) This is a space to ping reviewers and discuss feedback regarding your changes
6) A pull request then needs approval by at least 2 reviewers
7) Once a pull request is fully approved, double check again that your branch is up to date with staging and there are no conflicts
8) A lead will give the approved PR one final check before merging it into staging
    a) This is the point by which documentation should be written
9) The pull request can now be merged and the task marked as Done

PR Process - Reviewer

1) Check the Jira ticket for the task
2) Make sure any related documentation is up to date with the information necessary to use the system or feature

3) Go in engine and confirm that the feature is ready to be merged into staging
    a) The feature should function as described in the documentation and Jira ticket
    b) There should be no game-breaking bugs
4) Go to the GitHub website and view the pull request
5) Review the code to ensure it meets code standards
    a) This is not just about functionality, but also commenting and readability, optimization, code architecture, and general best practices
6) Leave comments using the comment feature if changes need to be made
    a) Additionally, the reviewee should be notified that changes were requested
7) If the code meets standards and is ready to merge, mark the pull request as “approved”
8) If the code is not ready to merge and changes were requested, keep up to date with the reviewee as iterative changes are made and repeat this process again