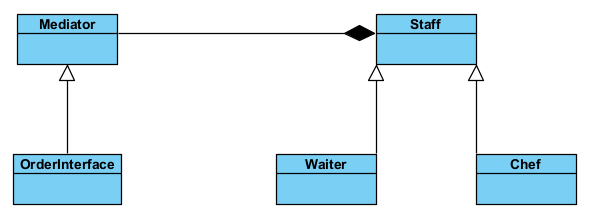
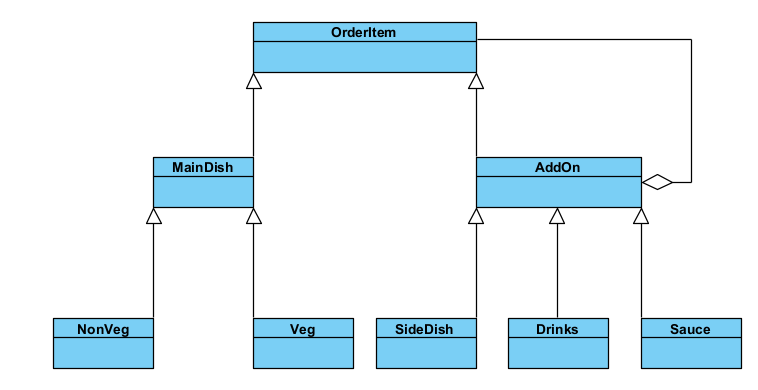
1.1

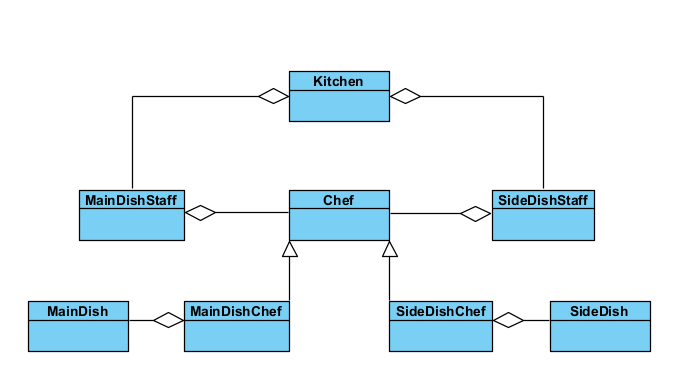
Mediator:



Decorator:

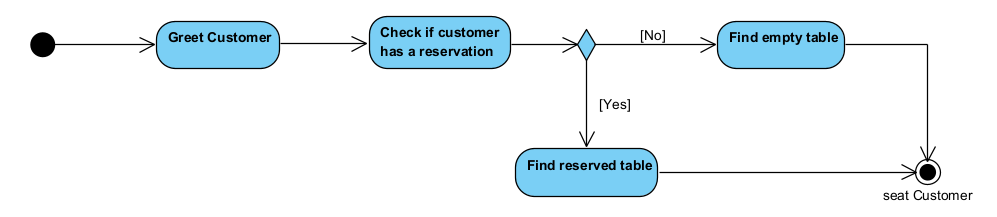


Builder:

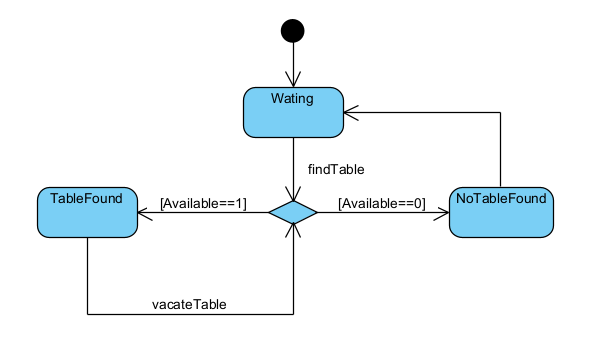


1.2

Activity Diagram:



State Diagram:



Sequence:

A diagram of a process

Description automatically generated

Communication Diagram:

A diagram of a work flow

Description automatically generated

1.3

Naming Standards:

Naming standards are agreements and best practices that define what to name your code elements. For example, some standards recommend using verbs for functions, nouns for classes, and adjectives for variables.

Avoid using abbreviations or acronyms

A common mistake that can make your code hard to read and understand is using abbreviations or acronyms that are not widely known or obvious. Unless the abbreviation or acronym is very common and clear, such as HTML, URL, or API, avoid using it in your names. Instead, use the full word or phrase that explains what the code element does or represents. For example, instead of naming a variable numCust, name it numberOfCustomers. Instead of naming a function calcAvg, name it calculateAverage.

Use consistent and meaningful prefixes and suffixes

Sometimes, you may need to use prefixes or suffixes to add more information or context to your names. For example, you may use prefixes to indicate the type, scope, or visibility of your variables, such as int, g\_, or m\_. You may use suffixes to indicate the unit, format, or role of your variables, such as \_sec, \_txt, or \_btn. However, you should use prefixes and suffixes only when they are necessary and helpful, and not when they are redundant or confusing. You should also use them consistently and meaningfully throughout your code.

Avoid using magic numbers or strings

Another common mistake that can make your code hard to read and understand is using magic numbers or strings. Magic numbers or strings are literal values that are not explained or defined in your code. For example, instead of using 3.14, 42, or "Hello", you should use constants or variables that have meaningful names, such as PI, ANSWER, or GREETING. This way, you can avoid hard-coding values that may change or have different meanings in different situations.

Refactor and rename your code as needed

Finally, one of the best practices of clean code is to refactor and rename your code as needed. Refactoring is the process of improving your code without changing its functionality. Renaming is one of the simplest and most effective ways of refactoring your code. As your code evolves and grows, you may find that some of your names are no longer accurate, relevant, or consistent. In that case, you should not hesitate to change them to better reflect the current state and logic of your code.

Snakecase:

Words are delimited by an underscore.

first\_name

Pascalcase:

Words are delimited by capital letters.

FirstName

Camelcase:

Words are delimited by capital letters, except the initial word.

firstName

Git — Commit Standards

commit Types: Some commonly used types in commit messages include:

feat: Used when adding a new feature.

fix: Used when fixing a bug.

refactor: Used when reorganizing or restructuring existing code.

docs: Used when making changes related to documentation or comments.

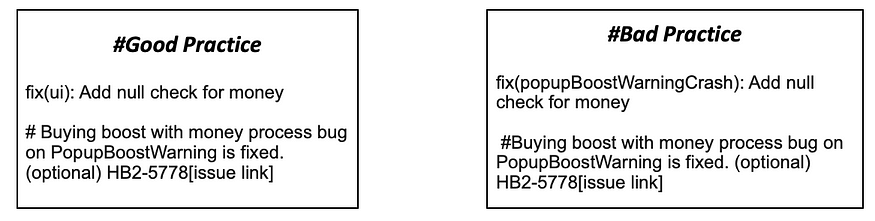
style: Used for changes in code formatting, whitespace, punctuation, etc.

test: Used when adding or updating test code or test scenarios.

chore: Used for changes related to auxiliary tools, configuration files, or project organization.

Commit comments should start with “#”. If you don’t use the “#” symbol in your commit messages, automated systems like “danger bot” can perceive these messages as incorrect or inappropriate rather than code changes. Therefore, it is important to use the “#” symbol correctly when writing your commit messages.

The Scope should indicate the specific feature or area of the code changes. If it’s not a specific feature, it should provide a broader context, such as ui, data, utils, ai, gameplay, shader. The Scope name should be written in camelCase format and ideally be a single word.



Branch Merge

When working on a branch and needing to merge the changes from the “develop” branch, conflicts may arise. When resolving this conflict, the commit message should be as follows:

fix(conflict): Resolve conflicts

# Merge develop into A branch. Resolve conflicts on GameManager.

Commonly used git commands:

git init: Creates a new empty Git repository in a directory.

git clone [repository]: Copies a remote Git repository to a local directory.

git add [file]: Adds a specific file or all changes to the “stage” for tracking.

git commit -m “[description]”: Records the staged changes as a “commit.” The description should be a text that explains the changes made.

git status: Shows the current state of the repository. It displays which files have been modified, staged, or are still unstaged.

git push: Sends the commits from the local repository to a remote repository. It is used to share the changes made in the local repository.

git pull: Fetches the latest changes from a remote repository to the local repository.

git branch: Lists the current branches. It can be used to create a new branch or switch to an existing branch.

git checkout [branch]: Switches to a specific branch. It can also be used to create a new branch or delete an existing branch.

git merge [branch]: Merges the changes from a specific branch into the current branch.

git log: Shows the commit history in the repository. It displays who made the commits, when they were made, and the changes included.

git reset [commit]: Resets to a specific commit. It can undo commits or discard changes.

git stash: Saves temporary changes in the working directory and reverts to a clean state. Later, these temporary changes can be reapplied