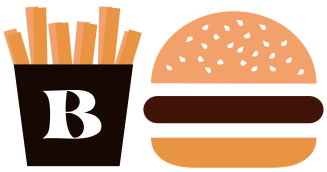
2023

COS 214

Practical 5

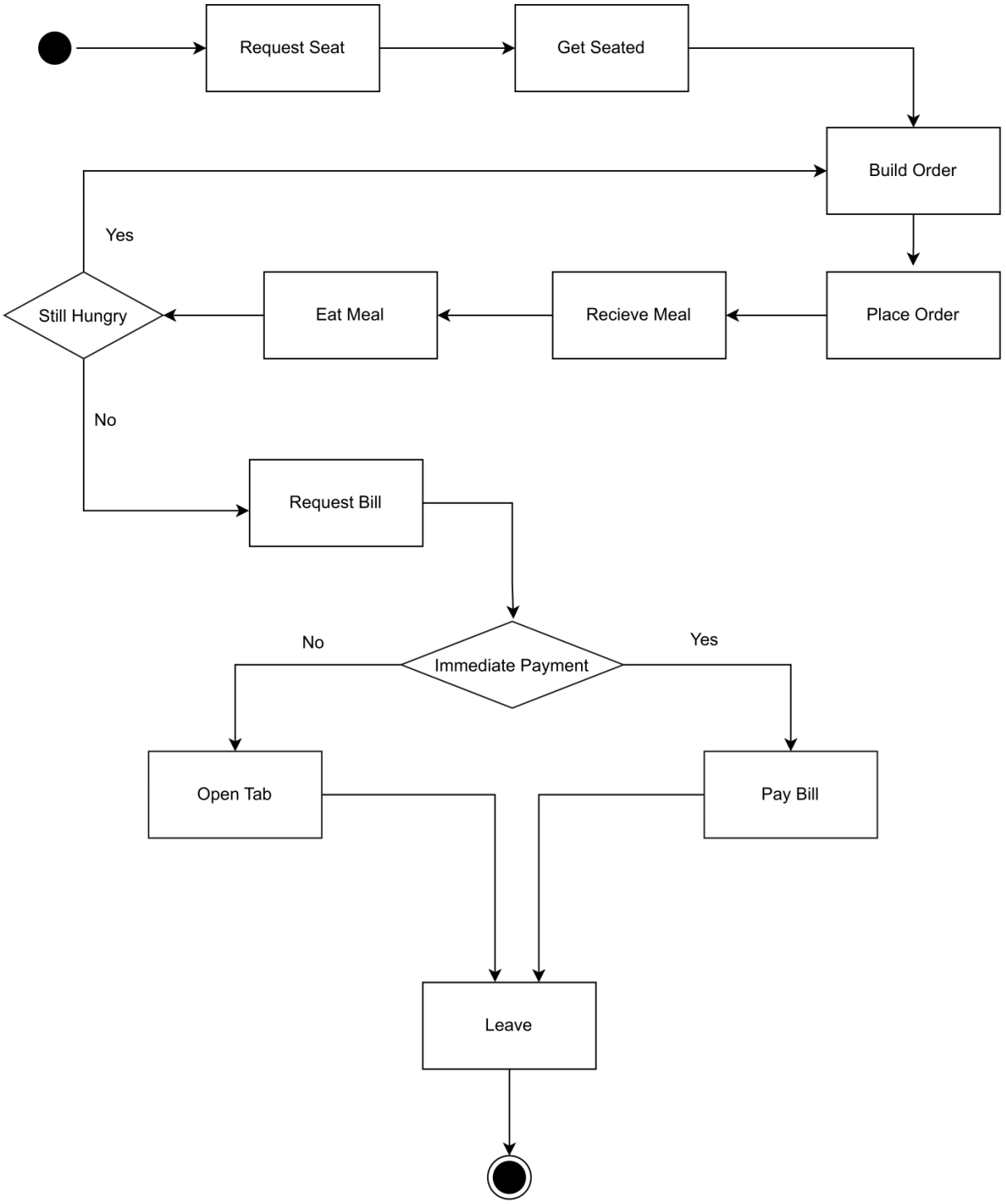


Burgundy

Team Members:

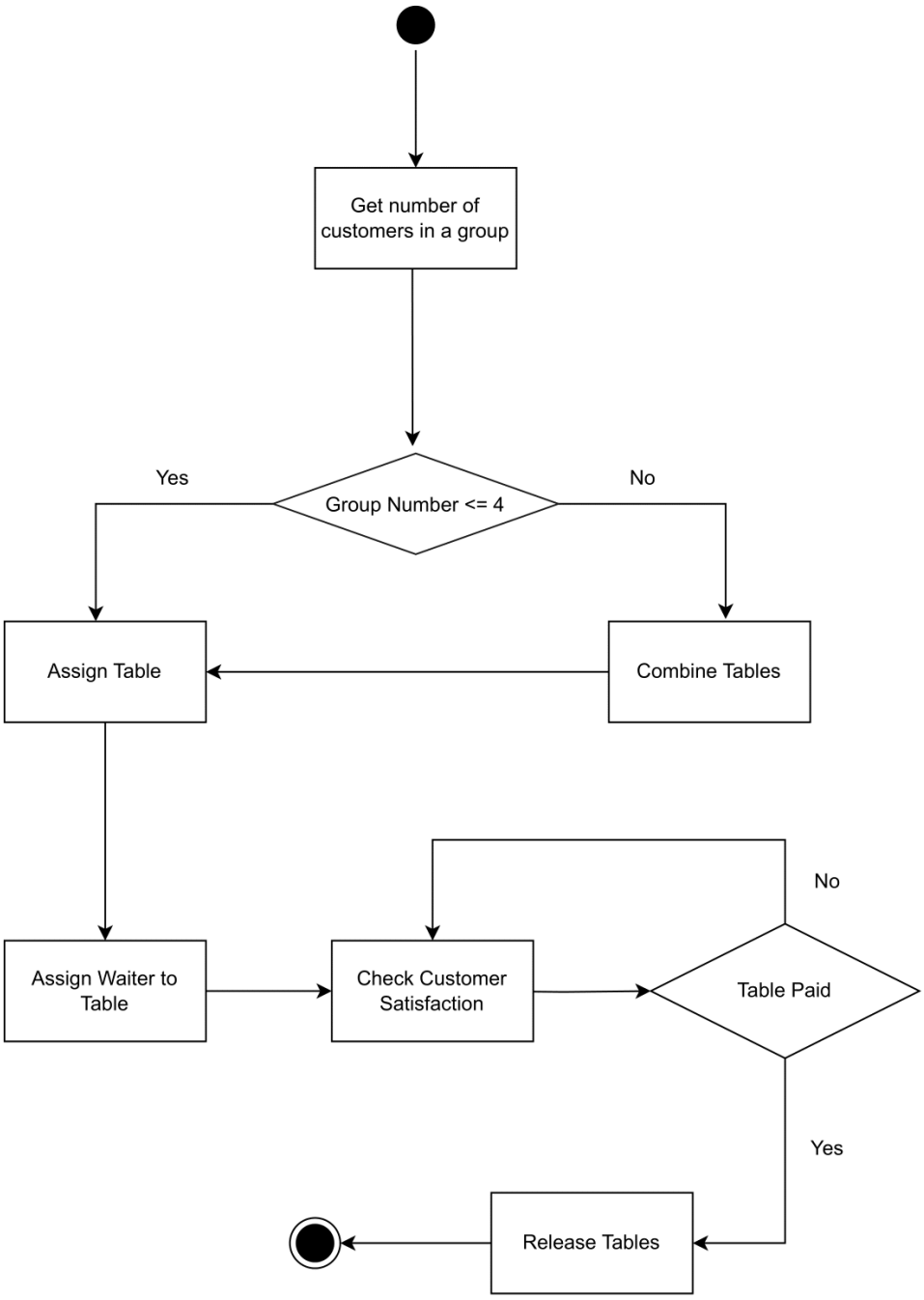
Lerato Gololo
Brenden van der Mescht
Siyamthanda Ndlovu
Katlego Zondo
Sechaba Ntshehi

Customer Activity Diagram

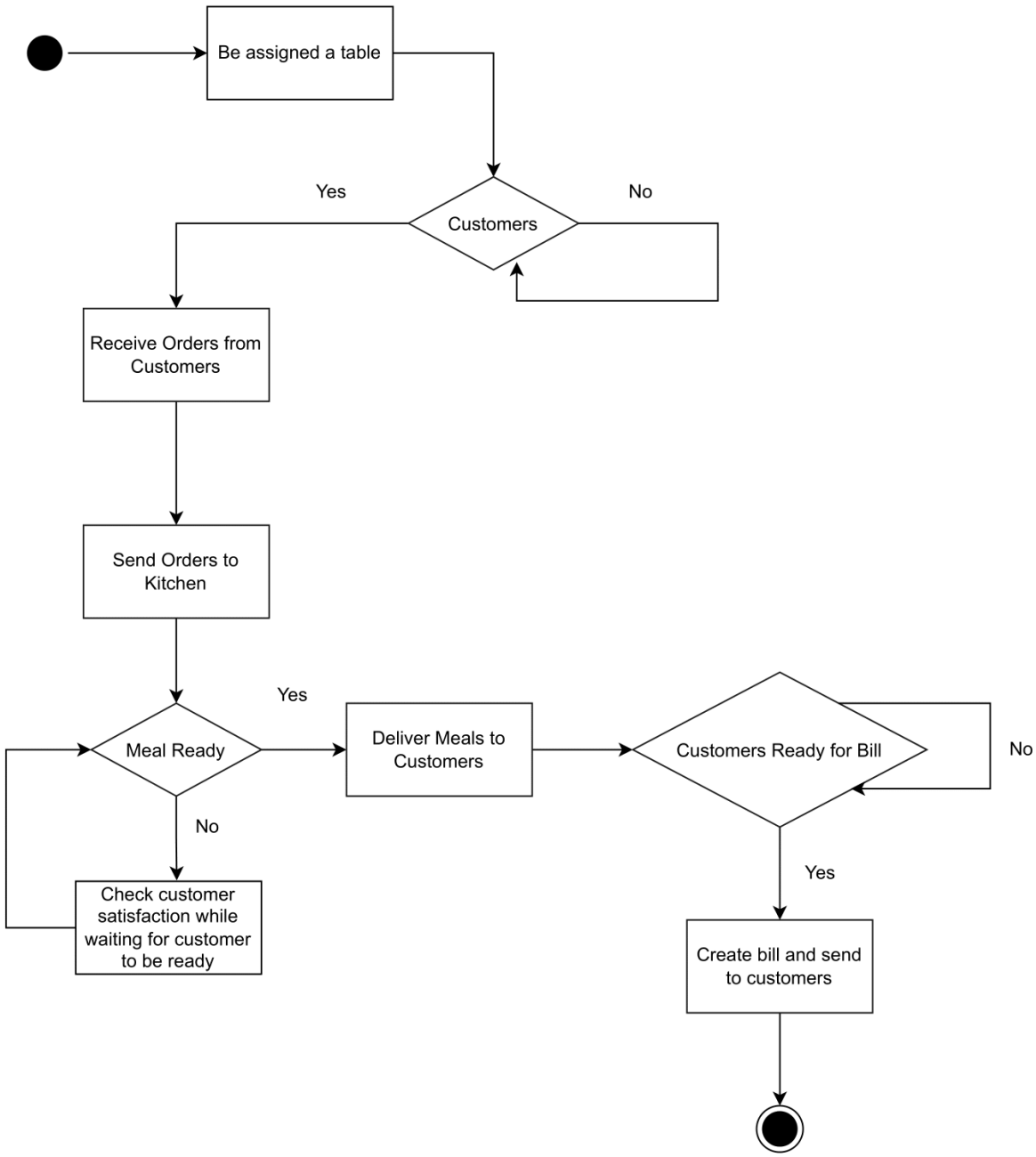


Note: Pay Bill is composite and paying immediately can be done by either one customer or by all customers at that table

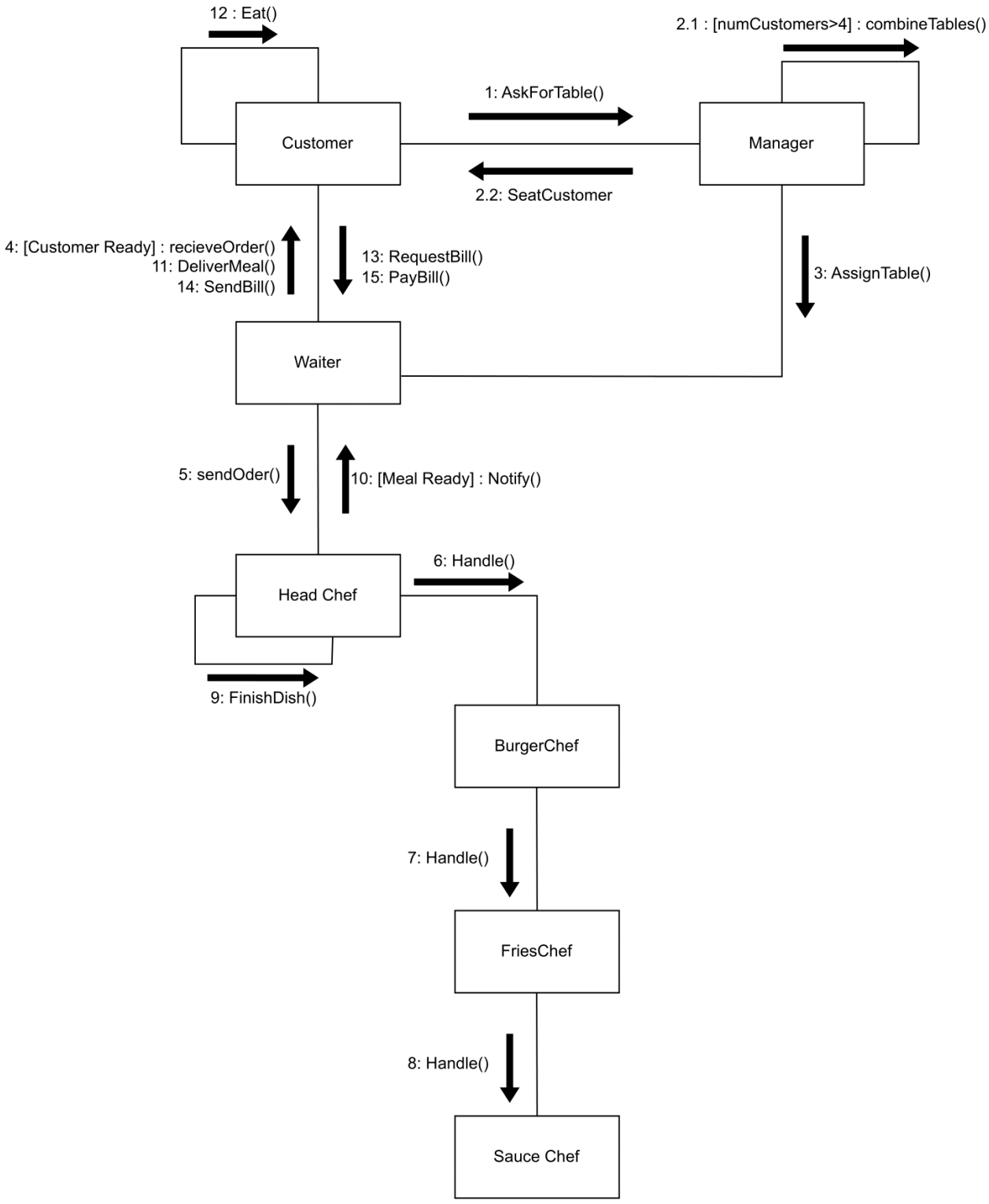
Manager Activtiy Diagram



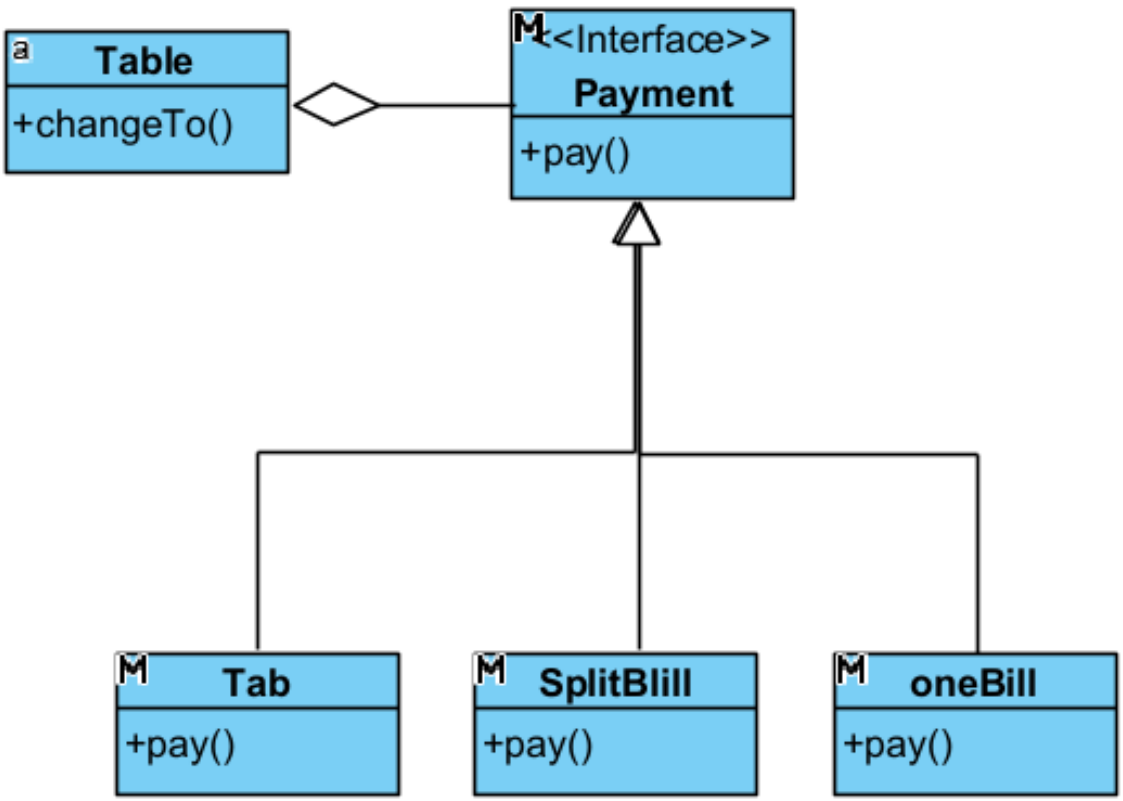
Waiter Activity Diagram



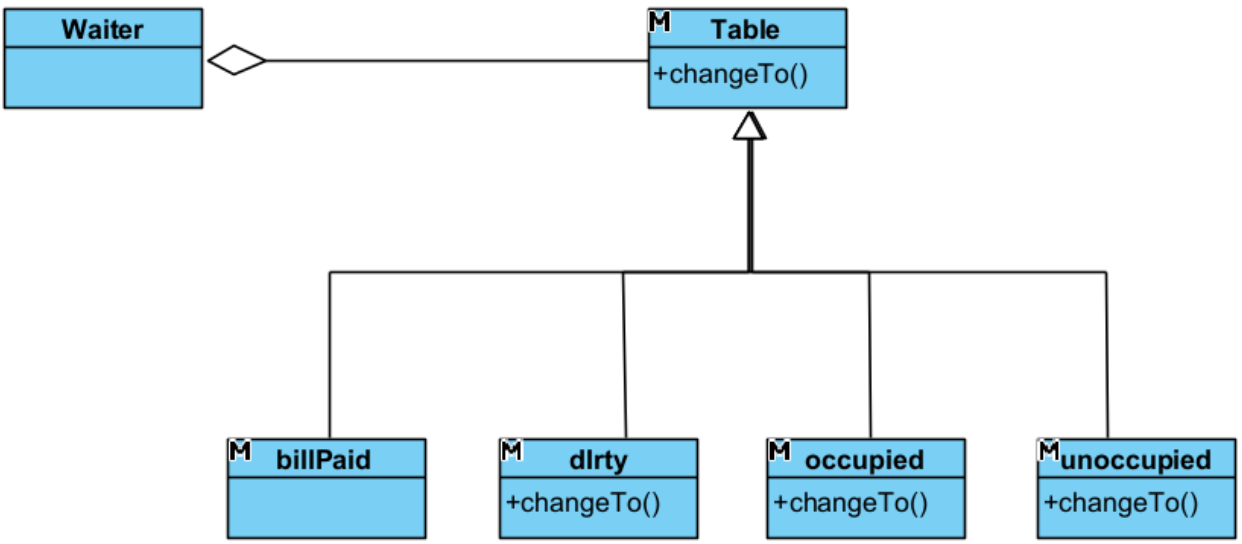
Kitchen-Floor Communication Diagram



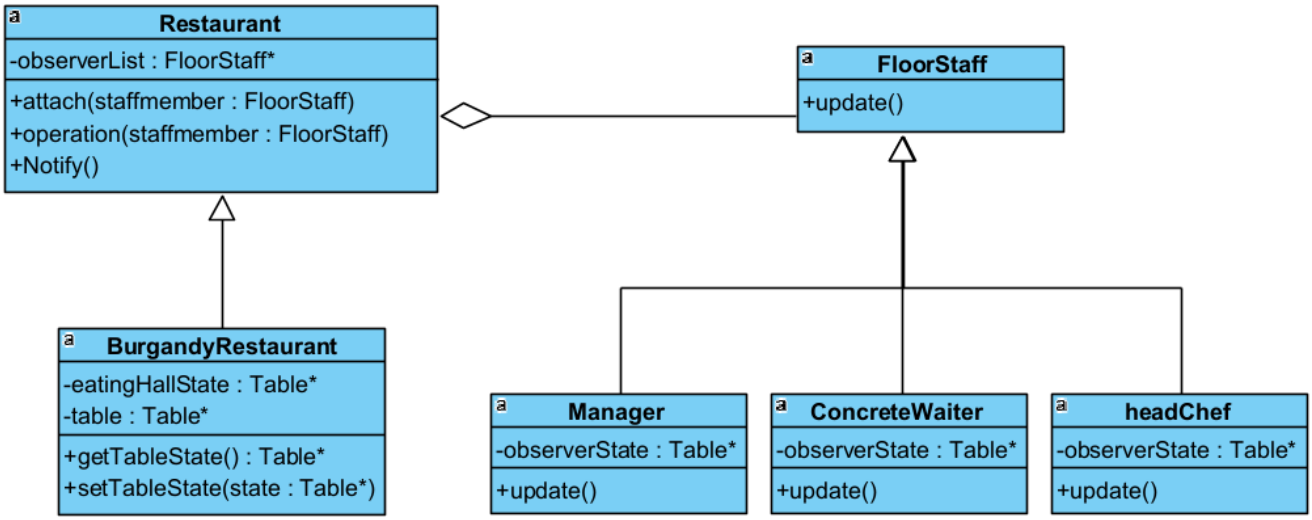
Strategy Pattern



State Pattern



Observer Pattern



Burgundy

COS 214 Final Project - Coding Standards

Naming Conventions

- **Function/ Variable Names:** Use camel case for naming. Start with a lowercase letter and capitalise the first letter of each subsequent word. Example:

```
int myVariableName;
```

Code Formatting

- **Spacing:** Use spaces on both sides of operators to improve code readability. Example:

```
z = x + y;
```

- **Variable Declaration:** Whenever possible, define variables at the beginning of the function. After declaring the variables, leave a line open before continuing with the rest of the function. This will improve code organisation and readability.

- **Pointer Declaration:** When using pointers, there must be no space between the datatype and the dereference operator. Example: `int* variableName`.

```
int* variableName;
```

- **Compound Statements:** Follow the Kernighan and Ritchie (K&R) style for compound statements. Place the opening brace on the same line as the function declaration or control statements. Use a consistent "Tab" indentation for the content within the braces. Example:

```
void functionName() {
    if (condition) {
        //Do something
    }
    else {
        // Do something else
    }
    return;
}
```

- **Functions**
  - Functions are designed to perform a specific task efficiently and effectively.
  - In order to promote modularity, functions are generally recommended to be limited to a length of 20-40 lines.
  - Constructor functions, unlike other functions, are exempt from the maximum of three parameters rule.
  - It is considered good11 practice for functions, even those with no return value (void functions), to always conclude with a return statement.
  - It is important for function names to accurately describe the purpose or action performed by the function.
  - Within their scope, functions are typically expected to have only one return statement.
  - Function names should be reasonably concise and descriptive.
  - It is advisable to validate pointers for null values to prevent potential issues.
  - Destructors, which handle the cleanup of resources, should be carefully managed to ensure proper execution.

- **Classes**
  - Classes must be defined using in a .h file and implementation must be in the .cpp
  - All classes must have their own header files
  - Copy constructor and assignment overloading will be implemented on a case by case basis.

Documentation

- **Doxygen Comments:**
  - For documentation, use Doxygen comments to describe the code.
  - Generally, high-level comments go in the header file of a class. Inline comments should still go inside the function definition inside the source file.
  - All comments that describe functions should contain a brief description, a longer and more detailed description and parameters and return values if applicable.

Error handling

- Throw exception when function expects an assigned pointer (NullpointerException)
- Input validation will be handled with exceptions (RuntimeException)
- Handling of exceptions will handled on a case by case basis

Testing

- Before merging a feature to the development branch, features should be fully tested, and functional
- Good testing standards must be followed, such as the ones detailed here: https://www.w3schools.in/software-testing/standards
- Trivial functions are exempt from testing
- Cover different use cases and user scenarios to ensure that your code handles various situations correctly.
- Using google test http://google.github.io/googletest/

Version Control using Git

- **Branching Strategy**
  - **Gitflow Workflow:** We will follow the Gitflow workflow for branch management. This workflow defines two main branches. The `master` and `develop` branches, along with other supporting branches for features, releases and hotfixes.
  - **Main Branches:**
    - **`master`:** Represents "production-ready" code. Code on this branch should be stable and executable.
    - **`develop`:** Integration branch where new features are merged and tested. This branch should also remain relatively stable.

- **Feature Branches**
  - **Feature Branches:** When adding new features, create feature branches from the **`develop`** branch. Give the feature branches descriptive names
  - **Pull Requests:** Submit a pull request to merge feature branches into the **`develop`** branch. The code in a feature branch should be reviewed and tested before merging.

- **Release Branches**
  - **Release Branches:** Release branches are used for final testing and any last-minute bug fixes. Release branches should also get descriptive names.
  - **Pull Requests:** Submit pull requests for changes in the release branches. Ensure proper testing and reviewing before merging to both **`develop`** and **`master`** branches.

- **Hotfix Branches**
  - **Hotfix Branches:** Used to fix critical issues or bugs in the production code. Should also use descriptive names.
  - **Pull Requests:** Submit pull requests for changes in the hotfix branches. Ensure proper testing and reviewing before merging to both **`develop`** and **`master`** branches.