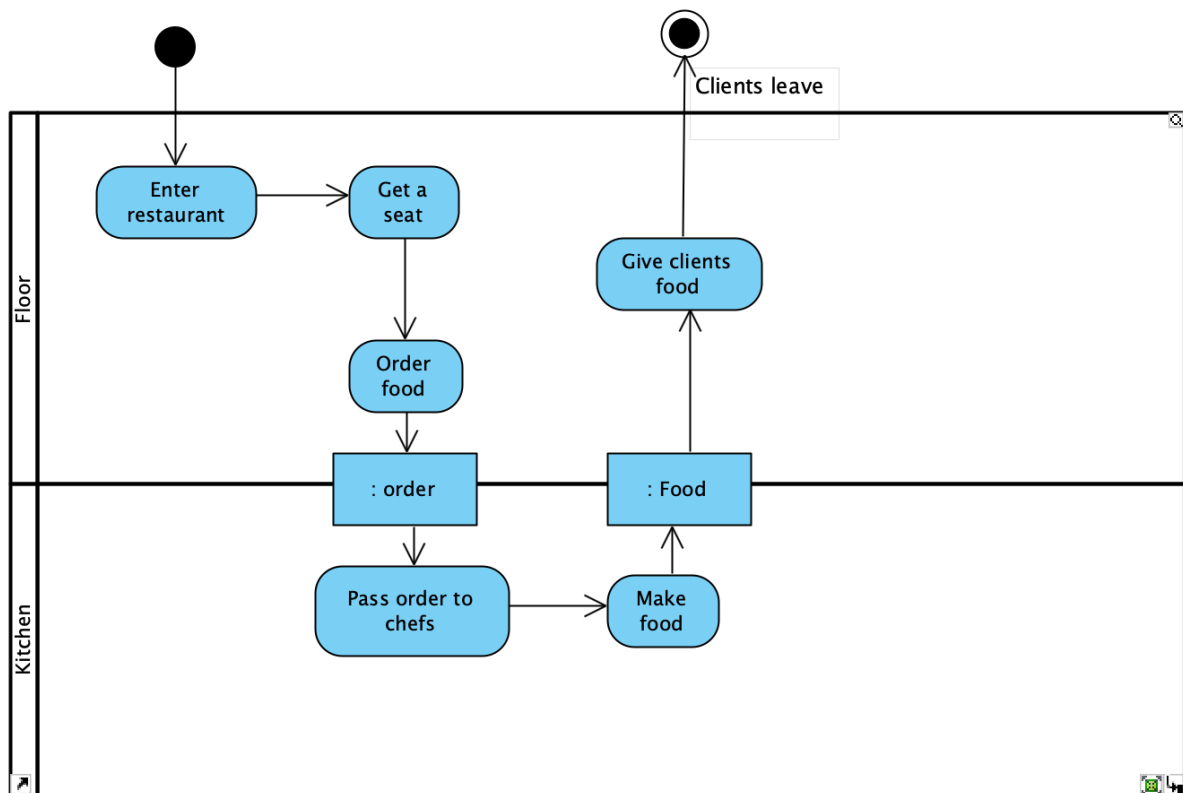


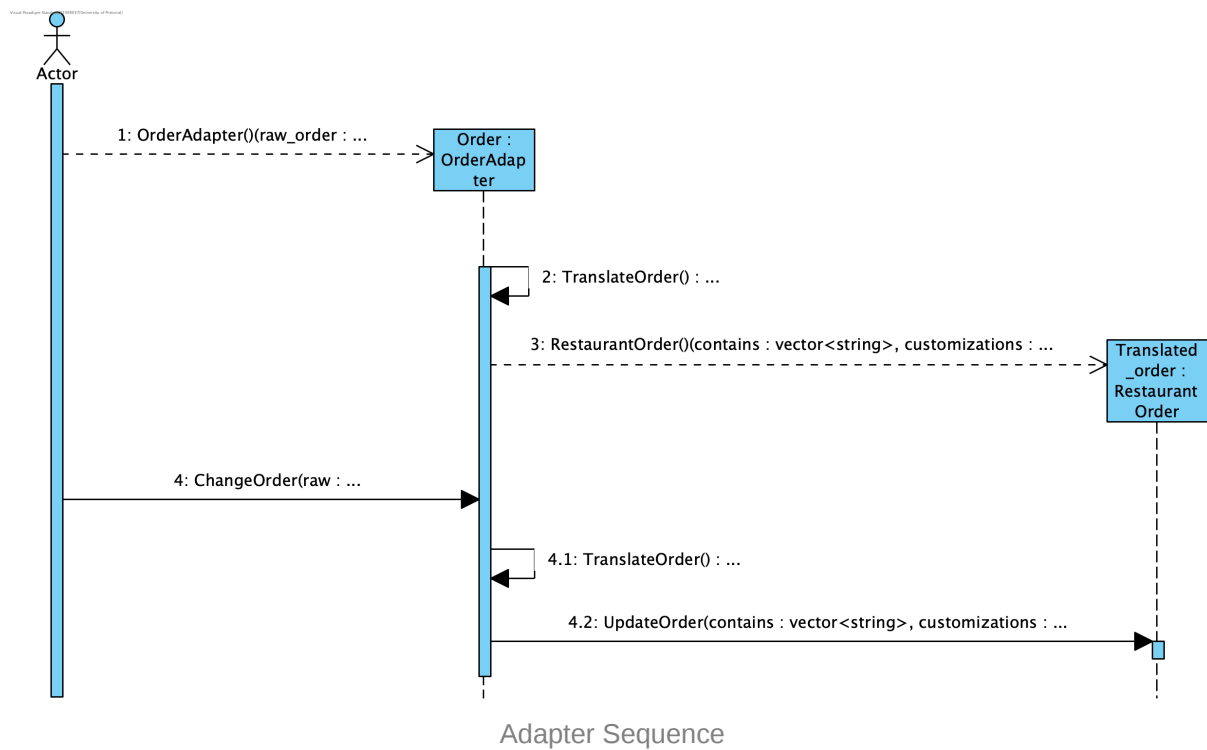
214 Project

<https://github.com/GeekGurusUnion/214-PA5.git>

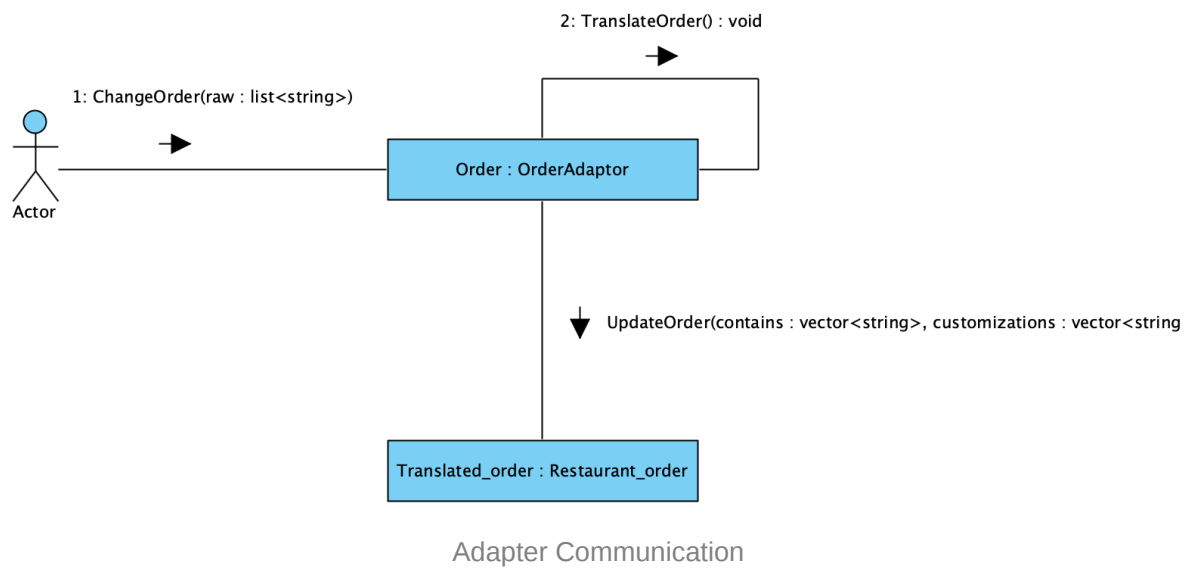
Activity Diagram



Sequence Diagram



Communication Diagram







Class Diagram

COS 214 Project

Group Organization

Members

- Tiaan Pouwels (u21675229@tuks.co.za)  @Tiaan2
- Dawie Reyneke (u21438112@tuks.co.za)  @ReynekeD
- Xavier Reynolds (u20526254@tuks.co.za)  @Xavier893
- Keith Homan (u21473103@tuks.co.za)  @d1scrd
- Jacobus Smit (u21489476@tuks.co.za)  @SW1F7YY
- Iwan de Jong (u22498037@tuks.co.za)  @iwandejong

Coding Standards

1. Organizational

1.1. Use a version control system

- Never keep files checked out for long periods (small incremental updates (see Git Standards 1.1. for more info)).
- Ensure that checked-in code doesn't break the build (Github Actions doesn't do everything).

1.2. Code Reviews

- Peer-review other's work so that you understand what is going on and ensure their code isn't breaking your code.
- Make a Github issue to describe the problem and assign the responsible person to the issue.

1.3. File Names

Filenames should be all lowercase and can include underscores.

Examples of acceptable file names:

```
my_useful_class.cpp  myusefulclass.cpp
```

2. Design Style

2.1. Give entity *one* cohesive responsibility

- For each entity, focus on one thing at a time.
- Give each entity (variable (member), class, function) one well-defined responsibility.
- As the entity grows, the scope increases, but it should not diverge.

2.2. KISS (Keep It Simple Software)

- Correct is better than fast.
- Simple is better than complex.
- Clear is better than cute.
- Safe is better than insecure

2.3. Minimize global and shared data

Sharing causes contention. Avoid shared data, like global data. This increases coupling which reduces maintainability.

2.4. Ensure resources are owned by objects

Never allocate more than one resource (pointer) in a single statement. This eases the process of memory deallocation.

2.5. Optimize for the reader, not the writer

More time is spent reading code than writing it.

3. Coding Style

3.1. Use `const` proactively

`const` (immutable) variables are easier to understand and to track. It's safe and checked at compile time.

3.2. Declare Variables as locally as possible

Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible.

3.3. Always initialize variables

This is a common source of C++ hotfixes. Initialize variables upon definition.

3.4. Avoid long functions

Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility (As explained in 2.1).

3.5. Minimize Definitional Dependencies

Don't be over-dependent: Don't #include a definition when it is not needed (or included by its parent anyways).

3.6. Always write internal #include guards

Prevent unintended multiple inclusions by using #include guards.

3.7. Don't use `using namespace std;`

Rather use a using-declaration which lets you use cout/cin/string without qualification

```
using std::cout;  
cout << "Values:";
```

4. Functions and Operators

4.1. Order parameters according to their value, pointer or reference

Distinguish among input, output, and input/output parameters, and between value and reference parameters.

4.2. Avoid overloaded operators as far as possible

Overload operators only for good reasons. It's easy to misuse operator overloading and cause confusion among fellow coders.

5. Class Design and Inheritance

5.1. Use design patterns!

Because we should xD.

5.2. Each `new` should be coupled with an `delete`

Basic thought for memory deallocation.

5.3. The `#define` guard

As an example, the file `foo/src/bar/baz.h` in project `foo` should have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif // FOO_BAR_BAZ_H_
```

5.4. Declaration Order

Within each section, prefer grouping similar kinds of declarations together, and prefer the following order:

- Types and type aliases (typedef, using, enum, nested structs and classes, and friend types)
- (Optionally, for structs only) non-static data members
- Static constants
- Factory functions
- Constructors and assignment operators
- Destructor
- All other functions (static and non-static member functions, and friend functions)
- All other data members (static and non-static)

6. Construction, Destruction and Copying

6.1. Define and initialize member variables in the same order

Agree with your constructor's parameters: member variables are initialized in the order they are

declared.

7. Error Handling and Exceptions

Prefer using exceptions over `cout` . This keeps the output clean.

Sources

<http://micro-os-plus.github.io/develop/sutter-101/>

<https://google.github.io/styleguide/cppguide.html>

<https://stackoverflow.com/questions/1452721/why-is-using-namespace-std-considered-bad-practice>

Git Standards

1. Repository Rules

- Fork the Organization's repo to work and test code locally - **fork ONLY the `main` branches & pull request to the respective branch on the Organization's repo**
- Reduce the frequency of pull requests unless the advancement is impeded by a required feature from a particular team member.

1.1. Basic Workflow

1. Make sure to fetch latest updates from the organization's repo onto your forked repo by running

```
git fetch upstream && git merge upstream/<branch>
```

2. **Make sure you are working in `origin/main`**
3. Commit frequently to the respective branch, but also narrow-down commits to provide clarity
4. After a night's work, create a pull request to merge onto organization's repo such that other members can sync with your latest changes.
5. **Make sure your pull request directs to the correct branch**

Example workflow: When working on the `dev` branch, use `git fetch upstream/dev` . You can

make your own branches in your forked repo as needed. When creating a Pull Request, select the `dev` branch on the Organization side (see image above). This applies to all branches (e.g. working with `hotfix` branch causes a PR to be directed to the organization's `hotfix` -branch)

2. Committing Code

- Make atomic commits of changes, even across multiple files, in logical units. That is, as much as possible, each commit should be focused on one specific purpose.
- As much as possible, make sure a commit does not contain unnecessary whitespace changes. This can be checked as follows:

```
$ git diff --check
```

3. Commit Messages

For consistency, try and use the imperative present tense when creating a message. Examples:

- Use "Add tests for" instead of "I added tests for"
- Use "Change x to y" instead of "Changed x to y"

4. Branching

4.1. Main Branches

Our main repository will have `main` as the evergreen branch.

4.2. Supporting Branches

To aid in ease of tracking new features, a few sub-branches have been added:

- Feature branches
- Dev branches
- Hotfix branches

These branches may have a limited lifetime and will be removed eventually.

4.3. Feature Branches

4.3.1. Naming Convention

Feature Branches must be named `feature-<featureClassification>` (referred to as `feature-id`).

4.3.2. Merging Feature Branches

These feature-branches must be merged onto the `dev` -branch.

```
$ git checkout -b feature-id dev    // creates a local branch for the new feature
$ git push origin feature-id       // makes the new feature remotely available
```

And for actually merging onto `dev` -branch:

```
$ git merge dev
```

When development on the feature is complete, the lead (or engineer in charge) should merge changes into master and then make sure the remote branch is deleted.

```
$ git checkout dev                // change to the master branch
$ git merge --no-ff feature-id    // makes sure to create a commit object during merge
$ git push origin dev             // push merge changes
$ git push origin :feature-id     // deletes the remote branch
```

4.4. Hotfix Branches

4.4.1. Naming Convention

Feature Branches must be named `hotfix-<hotfixClassification>` (referred to as `hotfix-id`).

4.4.2. Merging Hotfix Branches

These hotfix-branches must be merged onto the `main` -branch.

```
$ git checkout -b hotfix-id master    // creates a local branch for the new hotfix
$ git push origin hotfix-id           // makes the new hotfix remotely available
```

And for actually merging onto the `main` -branch:

```
$ git merge master // merges changes from master into
```

When development on the hotfix is complete, [the Lead] should merge changes into master and then make sure the remote branch is deleted.

```
$ git checkout master // change to the master branch
$ git merge --no-ff hotfix-id // makes sure to create a commit
$ git push origin master // push merge changes
$ git push origin :hotfix-id // deletes the remote branch
```

Sources

<https://gist.github.com/digitaljhelms/4287848>

<https://gist.github.com/digitaljhelms/3761873>