# Sequence Diagrams

A sequence diagram doesn’t depend on any particular language and doesn’t mean to match the code strictly. Its one-and-only focus is to describe the MAIN FLOW of essential functions or methods in the program.

Here are some rules you should follow when making sequence diagrams:

* Design is made first and BEFORE any coding step. This means THE CODE DEPENDS ON THE DESIGN, not vice versa.
* Sequence is all about the FLOW of methods/functions in a program. So, only focus on main points (what and where to implement) which really help describe the flow, and skip all others (how to implement).
* Putting every detail to a sequence diagram is STUPID. Just need to focus on main details. This way, you will save lots of time and reduce lots of unnecessary mistakes. To make the flow concise and easy to read, following points should be excluded from sequence diagrams:
  + Declaration of local variables, except special variables (global or static).
  + Initialization of variables, except ones needing initial values to make the flow correct.

Eg: bool isActive = false, then at a point in the flow: if (isActive) { do something } or if (something) { isActice = true }

* + Memory allocation / de-allocation
  + …
* In a big flow which requires multiple sequence diagrams to be fully described, always make the most overall one first. Other smaller diagrams will be REFERRED by this overall one. Don’t forget to include a diagram ID after each sequence name for easy reference.
* In many cases, describing details using a NOTE is easier to understand than putting pseudo code in sequence diagrams. So, choose to use notes or code wisely.
* Notes are not just useful for describing details, but they also help explain the pseudo code in sequence diagrams (similar to block comments). You cannot draw out a bunch of code and expect people to understand it without an explanation.

# Class Diagram

Here are some rules you should follow when making sequence diagrams:

* When you have defined an enum for a set of data, use it as much as possible.
* For UI design, there are two main sections: "Implementation of GUI", and "Implementation of data".
  + In "Implementation of GUI", the control should be defined with clear detail of its properties, name, and basic implementation of events: update state, update value, create item, set attribute, on command.
  + The functions used by UI but belong to data part should be in "Implementation of data".
* For business or data logic design, the structure will be same as the part "Implementation of data", except it doesn't need to separate into two sections.

# Common

**CommonDef.h**

It’s where we put constant definitions (either in form of #define or const or enum or struct). In general, they are:

* Names (or paths) of commonly-used folders / files / modules
* Limitations (max path’s length, etc.)
* ID or index --> enum
* Group of global variables --> struct

**String Resources**

They’re where we put:

* UI context (labels of controls, tooltips, etc.)
* Error/warning messages

Commonly-used formats: TXT, INI, XML

**String Preferences**

They’re where we put:

* Program configuration (PC name, IP address, enable/disable flag for a specific function, etc.)

Commonly-used formats: TXT, INI, XML, JSON

# Utility

A utility file contains commonly-used implementation, which can be used anywhere throughout the program, such as writing logs to file, converting data types, getting and formatting time, getting paths, etc.

**Notes:**

**1.** A utility should be non-class (with .h and .cpp) to make it instance-independent. It’s better to have a namespace to make code clear. Also, it should only contain methods, no global variables.

**2.** Many functions in utility files are created to return stable values. Because they’re (mostly) called many times throughout the program, re-executing steps defined in these functions are wasteful. So for run-time optimization, we should check if the value is already valid; if so, directly return it. This means static variables will be used.

For example:

static TCHAR st\_moduleDirectory[\_MAX\_DIR] = {0};

LPCTSTR GetModuleDirectory()

{

    // if st\_moduleDirectory is already gotten, no need to re-execute below steps

    if (\_tcslen(st\_moduleDirectory) == 0) {

        TCHAR fullPath[\_MAX\_PATH];

        TCHAR dir[\_MAX\_DIR];

        TCHAR drive[\_MAX\_DRIVE];

        ::GetModuleFileName(NULL, fullPath, \_MAX\_PATH);

        \_tsplitpath(fullPath, drive, dir, NULL, NULL);

        \_sntprintf(st\_moduleDirectory, sizeof(st\_moduleDirectory) - 1, \_T("%s%s"), drive, dir);

    }

    return st\_moduleDirectory;

}

# Wrapper

Wrapping is a step in customizing other’s frameworks (or libraries) to fit our system or expanding the functionality of particular APIs.

**Example 1**: Wraps the fopen() API of stdio.h, so that error logs will be shown when things go wrong:

In FileUtil.cpp:

FILE\* IO\_fopen(const char \**filename*, const char \**mode*)

{

    if (!*filename* || !*mode*) {

        WriteToLogBuf(LOG\_DEBUG, "Error IO\_fopen()");

        return NULL;

    }

    FILE\* res = NULL;

    res = fopen(*filename*, *mode*); // a member function of stdio.h

    if (!res) {

        char log[256] = {0};

        \_snprintf(log, 255, "Could not open the file %s", *filename*);

        WriteToLogBuf(LOG\_ERROR, log);

    }

    return res;

}

**Example 2**: Wrap the GetModuleFileName() of Win32 to get the module path without the module name.

In AppUlt.cpp:

static TCHAR st\_moduleDirectory[\_MAX\_DIR] = {0};

LPCTSTR GetModuleDirectory()

{

    if (\_tcslen(st\_moduleDirectory) == 0) {

        TCHAR fullPath[\_MAX\_PATH];

        TCHAR dir[\_MAX\_DIR];

        TCHAR drive[\_MAX\_DRIVE];

        ::GetModuleFileName(NULL, fullPath, \_MAX\_PATH);

        \_tsplitpath(fullPath, drive, dir, NULL, NULL);

        \_sntprintf(st\_moduleDirectory, sizeof(st\_moduleDirectory) - 1, \_T("%s%s"), drive, dir);

    }

    return st\_moduleDirectory;

}

# Error Message Handling

Here is a good way to go:

* Put all error messages in an error resource file (e.g., ErrorMessages.ini).
* Open the file once, at the beginning of the program.
* Throughout the program, when an error happens, read its key and find its matched value.
* Close the file once, at the end of the program.

TODO: Open and close the file once, or do these steps each time we read a key? Which way is more optimized?

The ErrorMessages.ini contains following content:

[ERROR]

ERR\_START = Unable to start the program.

ERR\_INVALID\_PATH = The path is invalid.

...

[WARNING]

...

# Logger

In any application, there should be some logging mechanism to record errors happening throughout the app or store the flow of the app. They will be extremely helpful data for the developer to debug if any crash occurs during run time.

Notes when create loggers:

* Must be multi-thread safe.

Example: PizzaStore\DebugLog.cpp

Other examples:

<https://cppcodetips.wordpress.com/2014/01/02/a-simple-logger-class-in-c/>

<https://stackoverflow.com/questions/8337300/how-do-i-implement-convenient-logging-without-a-singleton>