# Coding Standard

## Goals

Efficiently develop code and applications with the following qualities :

- **robust** : runs without crash and protects the data from being lost or corrupted.
- **secure** : protects the data from being stolen or hacked.
- **ergonomic** : can be used in an productive and intuitive manner.
- **efficient** : minimizes processing times to maximize the user productivity.
- **maintainable** : is easy to fix and enhance by any programmer in the team.
- **extensible** : is easy to extend with new features by reusing existing components.
- **consistent** : looks like it has been designed and implemented by a single developer.

## Specificity

This coding standard favors code readability over compactness, by :

- Forbidding the use of cryptic acronyms, abbreviations, prefixes and suffixes;
- Using different letter cases for classes, class members and local variables;
- Including the type name in the attribute and variable names.

## Rules

- Develop the application and its components with simple, robust and efficient code which will be easy to understand, extend and debug by any programmer in the team.
- Develop any piece of code so that it's :
  - **easy to understand** just by itself;
  - impossible to guess who has actually worked on it.
- Use **American English** for all the code, including comments.

```
InitializeColor();
MoveForward();
```

- Use the **meter** as the default distance unit.
- Use the **second** as the default time unit.
- Use **four spaces** instead of tabulations.
- Choose **short meaningful identifiers** for class, attribute, method and variable names.
- Use **standard prefixes** :
  - First, Last, Post
  - Prior, Next
  - Sub, Super, Base
  - Initial, Final

    - Old, New
    - Backward, Forward
    - Left, Right
    - Back, Front
    - Bottom, Top
    - Minimum, Maximum
    - Lower, Higher, Upper
    - Vertical, Horizontal
- Use **standard suffixes** :
    - Index, Count
    - Array, List, Map, Dictionary
- Use **standard verbs** :
    - Initialize, Update, Finalize
    - Is, Has
    - Reset, Set, Get, Find
    - Clear, Fill
    - Add, Remove
    - AddFirst, AddLast
    - Create, Destroy
    - Start, Stop
    - Begin, End
    - Enter, Exit
    - Open, Close
    - Read, Write
    - Load, Save
    - Pause, Resume
    - Enable, Disable
    - Lock, Unlock
    - Select, Deselect
    - Activate, Deactivate
    - Attach, Detach
    - Increment, Decrement
    - Increase, Decrease
    - Compress, Decompress
    - Connect, Disconnect
    - Send, Receive
    - Grant, Revoke
- Write **types** in **UPPER_CASE**, without articles.

```
class TANK_SHELL
{
}
```

- Write **type members** (methods, attributes, constants, etc) in **CamelCase**, without articles.

```
Tank.ShootShell();
```

- Write **local variables** and **method parameters** in **snake_case**, without articles.

```
player_name
```

- Don't use acronyms, abbreviations or single-letter variables.

```
TANK FindTank(
    int tank_identifier,
    int first_tank_index
    int post_tank_index
    )
{
    int
        tank_index;

    for ( tank_index = first_tank_index;
          tank_index < post_tank_index;    // no i, j, n, etc
          ++tank_index )
    {
        if ( TankArray[ tank_index ].Identifier == tank_identifier )
        {
            return TankArray[ tank_index ];
        }
    }

    return null;
}
```

- If you really have to use an acronym, capitalize it in member names.

```
DATABASE_URL
    DatabaseUrl;
```

- If a variable name collides with a predefined identifier, simply add a trailing underscore.

```
CLASS
    class_;

class_ = new CLASS;
```

- Include the class name (without numbers) in the attribute and variable names.

```
Dictionary<PLAYER, string>
    ActivePlayerDictionary;

List<ENEMY>
```

```
    CloseEnemyList;
TANK[]
    EnemyTankArray;
VECTOR_3
    InitialShellPositionVector,
    TankVelocityVector;

void ShootShell(
    )
{
    SHELL
        last_shot_shell,
        shot_shell;

    ...
}
```

- Start method names by a verb in the imperative mood (Set, Get, Find, ...).
- Use a verb in the indicative mood for boolean inquiries (Is, Has, Can, ...).
- Declare the method parameters in the same order as in the method name.

```
bool FindPlayerIndexByName(
    ref int player_index,
    string player_name
    )
{
    ...
}
```

- Use a positive affirmation for boolean variables and attributes.

```
if ( game_is_paused )
{
    ...
}
```

- When the attribute type starts like its owner type, omit the common prefix in the attribute name.

```
class TANK
{
    TANK_SHELL[]
        ShellArray;    // instead of TankShellArray
    bool
        IsDamaged;    // instead of TankIsDamaged
}
```

- Align matching braces.

```
bool CanShoot(
    )
{
    return ShotShellCount < MaximumShellCount;
}

// ~~

void ShootShell(
    VECTOR_3 initial_velocity_vector
    )
{
    ...
}
```

- Put braces around your repeated or conditional code even for one line of code.

```
if ( remaining_shell_count > 0 )
{
    ShootShell();
}
else
{
    Reload();
}
```

- Declare each attribute, variable and method parameter name on separate line.

```
int
    tank_count,
    tank_index;
```

- Try to declare all local variables at the start of the method, to improve the algorithm readability.
- Group local variables of the same type, and sort the declarations by ascending types and names, so that the declaration of a variable can be located at a glance.

```
int
    shell_count,
    shell_index,
    tank_count,
    tank_index;
string
    player_name,
    target_name;
```

- Try to split statements on several lines if they become wider than 100 characters, so that it's easy to edit two code files side by side on a single monitor.

- When splitting an expression on several lines, start the next lines with an operator and align it with the start of its left operand (or else indent it by 4 spaces).

```
if ( ( tower.GetDistance(
           tower_target,
           weapon_type
           )
       > tower.MaximumShootingDistance )
     || ( tank_distance > maximum distance
          && tank_health > 0.5 ) )
{

}
```

- Add exactly one space :
  - after `(` `[` `,`
  - before `)` `]`
  - after `if'` while for` ...
- Add exactly one empty line :
  - after a standard comment;
  - after the local variable declarations;
  - between a closing brace and the next statement.
  - between a return statement and the prior statement;
- Group the class elements by category, declared in the same predefined order :
  - Imports
  - Constants
  - Attributes
  - Constructors
  - Destructor
  - Operators
  - Inquiries (instance methods which won't change the instance attributes)
  - Operations (instance methods which can change the instance attributes)
  - Functions (static methods which don't belong to any instance)
- In a class, declare the called methods before the calling methods, so that the class code can be understood by a single sequential read.
- Use public attributes and methods, unless you really need to declare some of them as private.
- Use standard file extensions.
  - C# : cs
  - C++ : cpp, hpp
  - C : c, h
  - Javascript : js
  - HTML : html
  - CSS : css
- Declare one class per source code file.
- Use the class name in lowercase as file name.

```
tank_shell.cs
tank_shell.cpp
tank_shell.hpp
```

- Use the class name in uppercase for Unity source code files.

```
TANK_SHELL.cs
```

- Delimitate the code sections with standard comments.

```
// -- IMPORTS

...

// -- TYPES

class NAME
{
    // -- CONSTANTS

    ...

    // -- ATTRIBUTES

    ...

    // -- CONSTRUCTORS

    ...

    // -- DESTRUCTOR

    ...

    // -- OPERATORS

    ...

    // -- INQUIRIES

    ...

    // -- OPERATIONS

    ...

    // -- FUNCTIONS

    ...
}
```

- Don't use standard comments for empty sections.
- Align multiple lines comments with the surrounding statements, and write them as sentences.

```
/*
    A long explanation which is so long that it will have to be
    be split on several lines.
*/

...
```

- Align single line comments with the surrounding statements, and write them as sentences.

```
// A short explanation on a single line.

...
```

- Put end of line comments exactly four spaces after the statement, and start them in lowercase.

```
some_variable = some_magic_value;    // a short explanation
```

- Instead of adding comments to explain the code intent, refactor it to make it easy to understand without comments and improve its reusability.
- Begin C++ header files with `#pragma once`.

```
#pragma once

// -- IMPORTS

#include "tank.hpp"
#include "tank_shell.hpp"
...
```

- Name the unit test class by simply adding the `_TEST` suffix to the class name.

# Advices

- Design before you program, to avoid loosing precious time in developing the wrong solution to the wrong problem.
- First find what is really needed, by taking a few minutes to write :
    - a short text explaining how to use the application before implementing it, to optimize its interface.
    - a short text explaining what the application components will do before implementing them, to optimize their architecture.
    - a short text or test code explaining how the other programmers will use the application components, to optimize their class interface.

# Version

0.5

# Author

Eric Pelzer (ecstatic.coder@gmail.com).

# License