**CODING STYLEGUIDE**

**FOR**

****

***A Member-Managed Limited Liability Company***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Introduction**

This document serves as the definition of Tatum Games coding standards. The style guide covers formatting, syntax, naming conventions and best practices.

**2. Formatting**

Code formatting provides you with quick and detailed readability so that lots of code can be interpreted in depth. We follow Allman style.  
  
**Allman Style**

The Allman style is named after Eric Allman. It is also sometimes termed BSD style since Allman wrote many of the utilities for BSD Unix. This style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Example** | **Open/Close Equal Height** | **Open/Close Equal Indent** | **Open/Close Don’t Share Lines With Content** |
| Allman | while (x == y)  {    performAction();  } | No | Yes | Yes |

**2.1. Braces**

Braces are used with if, else, for, do, and while statements, even when the body is empty or contains only a single statement.

**2.1.2. Non-empty Blocks**

|  |  |
| --- | --- |
| **Line Break** | **True/False** |
| Line break before the opening brace | True |
| Line break after the opening brace | True |
| Line break before the closing brace | True |
| Line break after the closing brace, only if that brace ends a statement or terminates the body of a method, constuctor, or named class | True |

Example:

public void function( )

{

if(condition())

{

DoSomething();

}

else

{

DoSomethingElse();

}

}

**2.1.3. Empty Blocks**

Empty blocks may be closed on same line as opening bracket, unless part of a multi-block statement.

Examples:

void DoNothing( ){ }

void DoNothingElse( ) {

}

Not Acceptable

try {

DoSomething( );

} catch (Exception e) { }

**2.1.4. Single Line Statements**

Single line statements are allowed to have braces that begin and end on the same line.

Example:

get { return variable1; }

**2.2. Block Indentation**

Each time a new block or block-like construct is opened, the indent is increased by one tab. When block ends the indent returns to previous indent level. Also applies to comments.

**2.3. One Statement per line**

Each statement is followed by a line break.

**2.4. Column Limit: 60-80**

Limit columns to 60-80 characters long. Any code going over must be line-wrapped.

**2.5. Line-wrapping**

Line should be cut in the most readable way. Breaks should come before symbols, like "." or "::". A method or constuctor name stays with its parentheses, "( )". A "," sticks with the token that proceeds it.

**2.6. Whitespace**

Vertical: A blank line appears in between constructors, methods, nest classes, static initializers, and instance initializers. Variables are group together based on type, (public/private) separated by a blank line.

Horizontal: Spaces will appear within valid code written, as well as tabs based on coding standards.

**2.7. Horizontal Alignment**

Adding spaces so that variables line up horizontally is never required but is allowed.

Example:

private int x;

private Color color; or

private int x;

private Color color;

**2.8. Variable Declaration**

Every variable should be declared at a time, declarations like int a, b; should be avoided.

**2.9. Comments**

Comments can appear the line before a statement, or following the statement in line. Single-line comments allow narrative on only one line at a time. Single-line comments can begin in any column of a given line and end at a new line or carriage return. The // character sequence marks the text following it as a single-line comment.

Multi-line comments have one or more lines of narrative within a set of comment delimiters. The /\* delimiter marks the beginning of the comment, and the \*/ marks the end. You can have your comment span multiple lines and anything between those delimiters is considered a comment.

Example:

/\* Constructor.  
 In it we call another function

\*/

void Function ( )

{

// single-line comment

AnotherFunction ( );

}

2.10. Switch Statements

Switch statements will follow this format:

switch (variable) {

case 1:

break;

case 2:

break;

default:

break;

}

}

**3. Naming**

Naming conventions for best practice when you are developing. In short examples that do not include using directives, use namespace qualifications. If you know that a namespace is imported by default in a project, you do not have to fully qualify the names from that namespace.   
  
**Terminology**

* **Camel Case (camelCase):** In this the first letter of word always in small letter and after that each word with capital letter.
* **Pascal Case (PascalCase):** In this the first letter of every word is in capital letter.
* **Underscore Prefix (\_underScore):** For underscore ( \_\_ ), the word after \_ use camelCase terminology.

|  |  |
| --- | --- |
| **Kind** | **Rule** |
| Private field | \_lowerCamelCase |
| Public field | UpperCamelCase |
| Protected field | UpperCamelCase |
| Internal field | UpperCamelCase |
| Property | UpperCamelCase |
| Method | UpperCamelCase |
| Class | UpperCamelCase |
| Interface | IUpperCamelCase |
| Local variable | lowerCamelCase |
| Parameter | lowerCamelCase |

**3.1. Package Names**

Package names are in UpperCamelCase. First letter is capitalized, as well as first letter of every following word.

**3.2. Class Names**

Class names are in UpperCamelCase.

**3.3. Method Names**

Method names are in UpperCamelCase.

**3.4. Variable Names**

Variables are written in lowerCamelCase, first letter is lowercase with first letter of following words capitalized. Exception of constant variables.

**3.4.1. Constant Variables**

Constants are in CONSTANT\_CASE. All uppercase letters with "\_"between words.

**3.5. Parameter Names**

Parameter names are in lowerCamelCase.

**3.6. Local Variables**

Local variables are in lowerCamelCase.

**4. Best Practices**

A set of informal rules that must be followed.

**4.1. Use Descriptive Names**

Write names as specific as possible, so later developers can easily understand use and meaning.

Example: PauseButton, ExitButton instead of Button1, Button2

**4.2. Proper Parenting**

Hierarchies should always follow common sense. Avoid grouping together unrelated objects.

**4.3. Avoid Abbreviations**

Avoid shortening words for convenience, it makes the code more complicated and harder to read.

Example: Use SmallGun, LargeGun not 'sg', 'lg'

**4.3. Prefix Underscores to Highlight Items**

Add a '\_' to important scripts that will be use a lot. For example, \_ScoreManager keeps track of score changes from multiple sources. These type of objects will be used most often so it isa good idea to give them a distinct and visible name.

**4.4. Use Yes/No Questions for Booleans**

It is much easy to understand the boolean and it's states if yes/no questions is used. Also avoid use of double negative names.

Eaxample: canJump, isAlive, isGrounded

**4.5. File Organization**

Source files should contain only one public type, although multiple internal classed are allowed. Source files should be given the name of the public class in the file. Classes member should be alphabetized, and grouped into sections (Fields, Constuctors, Properties, Events, Methods, Private Interface Implementations, Nested types).

**5.0. Definition of Done**

The definition of done (DoD) is when all acceptance criteria that Tatum Games products must satisfy are met and ready to be accepted by a user, customer, team, or consuming system. We must meet the definition of done to ensure quality. It lowers rework, by preventing user stories that don’t meet the definition from being promoted to higher level environments. It will prevent features that don’t meet the definition from being delivered to the customer or user.

**Product is not considered completed until the following is met**

|  |  |
| --- | --- |
| **ID** | **Acceptance Criteria** |
| 1 | Unit tests must be included, and test coverage must be a minimum 70%. |
| 2 | Developer documentation in README.md markdown. Include pertinent engineering-level documentation that will help integrator understand how to use SDK. |
| 3 | All Tatum Games mobile game products must include Mikros SDK. |
| 4 | Create “docs” folder in all Tatum Games game products and include necessary architecture diagrams, sequence diagrams and flow charts. |
| 5 | Bitrise integration for weekly APK distribution |
| 6 | In-line documentation as well every method/function fully documented. |
| 7 | Follow Tatum Games coding styleguide |