C# coding standard for Intelligent agents, TME285 v 1.1, 2018-01-05

Mattias Wahde

1 Introduction

When writing code, in most cases you are not only writing for the computer to understand and make use of your code, but people as well! This, of course, includes yourself. Will you understand your code a month after having written it? By writing clear and highly readable code you reduce the risk of introducing unwanted errors. It is the aim of this coding standard to help you write such code.

To some extent, any coding standard represents a subjective choice. That is, several different coding standards would be possible. However, for this course, you *must* follow the coding standard as specified in this document. Note, however, that the coding standard is not complete. Instead, it describes only some of the most important aspects that you should keep in mind while coding. You should also look at the C# software libraries associated with the course (in particular, you should examine the code in the `ExamplesSolution` as well as Appendix A of the compendium), and strive to write your code in a similar way.

As a general rule (for the course, at least), always strive for *clarity*, without, of course, sacrificing too much performance. Try to put yourself in the position of another person seeing your code for the first time: Is the code sufficiently clearly written to be self-explanatory? Is it easy to modify the code, if that should be required?

2 Classes

C# is an object-oriented programming language, in which code concerning a specific topic is generally placed in a *class* that, in turn, typically contains constants, variables (fields), methods, and properties. In order to organize

the code for readability, one can use the concept of a *region*, whereby similar parts (such as all constants or all fields) can be grouped together. As an example, see the `ImageProcessor` class in the `ImageProcessing` library. You should strive to use this kind of grouping.

3 Naming practices

When naming a variable (or a constant, a method etc.) you should always strive to use a *meaningful* name that clearly describes the purpose of the variable. For example a good name for the variable used to store a speech synthesizer is `speechSynthesizer`, whereas simply naming the variable `s` is not recommended. Using long names is perfectly acceptable; When given a choice, a long but descriptive variable name is to be preferred over a short name with unclear meaning. Ugly abbreviations (such as e.g. `spSnt`) are not acceptable either.

3.1 Classes

For class names, the first character should be in upper case. If the class name consists of several words, all words should begin with an upper case letter, and all other letters should be lower case, e.g. `SomeClass`. Class names should be descriptive and specific. Thus, for example, the isolated word recognizer is called `IsolatedWordRecognizer` (rather than just `SpeechRecognizer`) etc. Instead, the `SpeechRecognizer` class is a base class from which other classes can be derived. For example, one may wish (later) to add a class called `ContinuousSpeechRecognizer` etc.

Also, in order to avoid conflicts between namespaces (in which case one would have to write out not only the class name but also specify the namespace when a corresponding variable is declared) avoid using very generic class names, or class names that are already used in the standard distribution of C# .NET.

3.2 Fields

For names of (private and protected) fields (variables), the first character should be in lower case. If the field name consists of several words, all words except the first should begin with an upper case letter, and all other letters should be lower case, e.g. `aLocalVariable`. Note: There should be no other characters (e.g. underscore) between the words in a field name. As stated above, using meaningful names should always have higher priority

than using short names and this is especially so in the case of variables with a large scope.

However, variables with small scope can have short names. For example, a variable used for storing a temporary value within an `if-then` statement (containing only a few lines of code between `if(...)` and `end`) may very well be named `tmpVal`. Also, in the specific case of mathematical equations for which a set of variable names is generally accepted, it is sufficient (at least if the scope is small) to use the same name for the corresponding fields in your code. For example, in case you are considering a mathematical function of two variables x_1 and x_2, it is perfectly acceptable to use the variable names `x1` and `x2`, rather than, say, `firstVariable` and `secondVariable`.

3.2.1 Iterator variables

Iterator variables should be named using `i`, `j` and `k` (or `ii`, `jj` and `kk`). However, in the case of a loop consisting of many lines of code, a longer and more meaningful name should preferably be used (e.g. `iIteration`) and should be prefixed with either `i`, `j` or `k`.

3.2.2 Abbreviations

Abbreviations in variable names (as well as in names for methods etc.) are acceptable in the case of very common abbreviations, e.g. *max* and *html*. Note that the upper and lower case rule for variable names (see above) still applies, e.g. `cthStudent` *not* `CTHStudent`.

3.3 Properties

For (public) properties, the first character should be in upper case, and should normally match the name of the corresponding field. Thus, for example, if a class contains a field called `position`, the corresponding property should be called `Position` etc.

3.4 Constants

In many cases, you will need to specify numerical values. Such values should (normally) be specified using a named constant and, as mentioned above, you should strive to specify all relevant constants on consecutive lines in a `Constants` region near the top of the file. This makes it easy to find the constants and, when necessary, change the respective values. Constants

should be named using capital letters, and with an underscore (_) between words.

You should *avoid* specifying the same constant, as a numerical value, at various places in the code. Thus, for example, if you need a constant enumerating, say, the number of iterations in some numerical operation, you should introduce a named constant (for example called `NUMBER_OF_ITERATIONS`) as

```
NUMBER_OF_ITERATIONS = 1000;
```

and then refer to `NUMBER_OF_ITERATIONS` wherever this constant is needed, rather than introducing the numerical value at various places in the code. The obvious reason for this is that, if a constant is defined in several different places, it is easy to make an error when changing the value (forgetting to change one instance of the constant, for example).

Note that one may, of course, have several constants taking the *same* value. For example, if two constants refer to completely different things, they should indeed be specified as *two* different entities:

```
NUMBER_OF_ITERATIONS = 1000;
CHROMOSOME_LENGTH = 1000;
```

As in the case of variables, constants should be given clear, descriptive names.

3.5 Methods

Methods should be named using upper case for the first character of *every* word in the method name, e.g. `RenderTriangles`. Function parameters (specified in the method interface) are named as variables. While variables are often named using nouns, methods names should preferably include at least one verb, since a method is intended to perform some action.

3.5.1 Auto-generated methods

When C# autogenerates methods, such as for example the code for a button click, it will generally do so using names that do *not* adhere strictly to the standard specified above. For example, if a button is called `startButton`, the corresponding autogenerated click method will be `startButton_Click`. Such method names *are* allowed here (for simplicity), but if you wish, you can simply write the corresponding method yourself (using a more appropriate method name, such as `StartButtonClick`), and then manually associate the corresponding event (`Click`) with this method.

4 Code organization and layout

4.1 Whitespace and other layout topics

Use whitespace to group your code in order to make it more readable. Use vertical whitespace (i.e. blank lines) to form blocks of lines of code, quite similar to paragraphs when writing normal text. Note that the lines of code that constitute a block should be cohesive (i.e. the lines of code should be strongly related to each other) and the formation of the block should be logical. Use horizontal whitespace (i.e. indentation) to group statements, such as `if-then-else` and `for`-loops. Note that, in general, the C# IDE will handle the indentation by itself. Example:

```
if (object3DList != null)
{
     foreach (Object3D object3D in object3DList)
     {
          object3D.Render();
     }
}
```

4.2 Avoid complex statements

In order to improve readability and avoid errors in code, avoid writing statements that each perform many steps of computation. For example, the code snippet

```
output = Math.Pow(abs(timeSeries.ValueList[i] - (delta*average +
        (1-delta)*gamma)), kappa)* weightLeft/
        (1+exp(alphaLeft*(timeSeries.ValueList[i] - (delta*average +
        (1-delta)*gamma)-betaLeft)))+weightRight/(1+exp(-alphaRight*
        (timeSeries.ValueList[i] - (delta*average + (1-delta)*gamma)-betaRight)));
```

should preferably be written using several statements and temporary variables

```
double x = timeSeries.ValueList[i];
double z = x - (delta*average + (1-delta)*gamma);
double wR = weightRight/(1+exp(-alphaRight*(z-betaRight)));
double wL = weightLeft/(1+exp(alphaLeft*(z-betaLeft)));
double w = wR+wL;
output = Math.Pow(abs(z),kappa)*w;
```

4.3 Conditional expressions

Avoid complex conditional expressions spanning over several lines. Instead, introduce temporary boolean variables.

```
if (!(location == BUNKER) && ((sustainedWinds == CATEGORY_2_SUSTAINED_WINDS)
   &&(centralPressure == CATEGORY_2_CENTRAL_PRESSURE)))
  ...
end
```

should instead be coded as

```
Boolean isCategory2Winds = (sustainedWinds == CATEGORY_2_SUSTAINED_WINDS);
Boolean isCategory2Pressure = (centralPressure == CATEGORY_2_CENTRAL_PRESSURE);
Boolean isCategor2Hurricane = (isCategory2Winds && isCategory2Pressure);
Boolean outsideBunker = !(location == BUNKER);
if (isCategory2Hurricane && outsideBunker)
  ...
end
```

4.4 Inputs to methods

When specifying the input to a method, use (when necessary) temporary variables, rather than passing a method call as input to another method. For example, instead of

```
faceRecognizer.Recognize(faceDetector.GetImage());
```

write

```
Bitmap face = faceDetector.GetImage();
faceRecognizer.Recognize(face);
```

The second option makes the code much clearer and easier to debug, if necessary.

4.5 Comments

You should strive to write code that is self-explanatory. However, sometimes it is needed to add information to the code, such as an explanation of a complex algorithm, information regarding limits or perhaps a motivation. An explanation concerning an entire method (or a class) can preferably be placed in its own `Comments` region just before the method (or class) code, so that the comments can be collapsed (hidden) if needed; see for example the class `LevisonDurbinRecursion` in the `MathematicsLibrary` (under `Matrices`). In the rare cases where an explanation is required inside a method (on a given line), write the comment using preceded by `//`, either on the same line as the code, or as an inserted comment line just before the line with the code. Do not overuse comments! Including unnecessary

comments such as

```
i = 1; // Assigns 1 to the variable i
```

is not a good idea.