



# E-Health Service
# Coding policies

**Version 1.0**

# Revision History

| Date | Version | Description | Author |
|---|---|---|---|
| 2012-02-11 | 0.01 | Initial Draft | Vedran Šikić |
| 2012-13-11 | 1.0 | Added C# coding policies | Petar Kekez |
| | | | |
| | | | |

# Table of Contents

# 1. Introduction

## 1.1 Purpose of this document

The purpose of this document is to provide coding standardization for our project team. The document also offers a lot of code examples and gives some advices for successful programming. Main goal is to reach unique uniform coding style for whole team.

## 1.2 Document organization

The document is organized as follows:

- Section 1, *Introduction*, describes contents of this guide and purpose of this document.
- Section 2, General guidelines, describes platform independent advices.
- Section 3, Java guidelines, describes standards for Java and Android.
- Section 4, C# guidelines, describes standards for C# language.

## 1.3 Intended Audience

The intended audience is:

- Members of this project.
- Supervisor and other persons involved with the project.

## 1.4 Scope

This document includes advices and standards for Java, Android, C# programming languages with many examples.

## 1.5 References

General guidelines: http://www.merrioncomputing.com/Programming/7Secrets.htm

Android guidelines: http://source.android.com/source/code-style.html

Java guidelines: http://geosoft.no/development/javastyle.html#Layout of the Recommendations

# 2. General Guidelines

These guidelines are not language specific and their main use is to improve quality and understandability of the code.

## 2.1 Code for human consumption

It is one of the most pervasive misunderstandings in computing that the source code is for the computer's consumption. Computers work with low-level binary code, a series of impenetrable 1's and 0's or hexadecimal numbers, not the structured high level languages we code in. The reason that these languages were developed was to help the programmer. That means: code for clarity first, over efficiency and speed second.

## 2.2 Comment often and comment well

The purpose of the comment is to tell you (and any future developer) what the program is intended to do. Write comments with this in mind - and avoid simply restating the code. Comments should be understandable by everyone, and written in common language – English in this case.

## 2.3 Layout code to increase legibility

Good indentation helps finding possible problems with code branching and increases readability of the code in general.

## 2.4 Expect the unexpected and deal with it

Before you open a file, make sure that the file is present. Before you set focus to a control, make sure that the control is visible and enabled. Try to work out what conditions could cause your code to fail and test for them before they cause the program to fall over, and do not rely on error handling to.

## 2.5 Name your variables to aid readability

There are a number of strategies to variable naming. The key is to be consistent and to be as informative as possible. Choose one convention and stick with it – consistency is the key.

## 2.6 Keep your functions and subroutines simple

A function or subroutine should ideally only do one thing. One of the greatest sources of misunderstandings, in my experience, is a subroutine that does a number of different operations. This should be split into separate functions for each different thing it is doing so that these in turn are easy to reuse, and the scope of a code change is easy to understand.

## 2.7 Scope functions and variables appropriately

Functions and variables that are only used in one module should not be visible outside that module. Variables that are only used in a function or subroutine should not be visible outside that function or subroutine. This prevents any use of a variable or function outside of where it makes sense to use it.

## 2.8 Never stop listening and learning

Keep an eye out for new opinions and methodologies and to evaluate them in an impartial and rational way. This applies to your choice of language, operating system, back-end database and development methodology.

# 3. Java coding guidelines

## 3.1 Naming conventions

### *3.1.1 General naming conventions*

*Names representing packages should be in all lower case.*
**All code should be in dsd.ehealth.* package and sub packages.**

*Names representing types must be nouns and written in mixed case starting with upper case.*

```
Line, AudioSystem
```

*Variable names must be in mixed case starting with lower case.*

```
line, audioSystem
```

*Names representing constants (final variables) must be all uppercase using underscore to separate words.*

```
MAX_ITERATIONS, COLOR_RED
```

*Names representing methods must be verbs and written in mixed case starting with lower case.*

```
getName(), computeTotalWidth()
```

*Abbreviations and acronyms should not be uppercase when used as name.*

```
exportHtmlSource(); // NOT: exportHTMLSource();
openDvdPlayer();    // NOT: openDVDPlayer();
```

*Private class variables should have underscore suffix.*

```
class Person
{
  private String name_;
  ...
}
```

*Generic variables should have the same name as their type.*

```
void setTopic(Topic topic) // NOT: void setTopic(Topic value)
                           // NOT: void setTopic(Topic aTopic)
                           // NOT: void setTopic(Topic t)
```

*All names should be written in English.*
*Variables with a large scope should have long names, variables with a small scope can have short names*

```
line.getLength();   // NOT: line.getLineLength();
```

*The name of the object is implicit, and should be avoided in a method name.*

```
line.getLength();   // NOT: line.getLineLength();
```

### *3.1.2 Specific naming conventions*

*The terms get/set must be used where an attribute is accessed directly.*

```
employee.getName();
employee.setName(name);
```

*is prefix should be used for Boolean variables and methods.*

```
isSet, isVisible, isFinished, isFound, isOpen
```

*The term compute can be used in methods where something is computed.*

```
valueSet.computeAverage();
matrix.computeInverse()
```

*The term find can be used in methods where something is looked up.*

```
vertex.findNearestVertex();
matrix.findSmallestElement();
node.findShortestPath(Node destinationNode);
```

*The term initialize can be used where an object or a concept is established.*

```
printer.initializeFontSet();
```

*Plural form should be used on names representing a collection of objects.*

```
Collection<Point>  points;
int[]              values;
```

*n prefix should be used for variables representing a number of objects.*

```
nPoints, nLines
```

*No suffix should be used for variables representing an entity number.*

```
tableNo, employeeNo
```

*Iterator variables should be called i, j, k etc.*
*Complement names must be used for complement entities*

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down, min/max,
next/previous, old/new, open/close, show/hide, suspend/resume, etc.
```

*Abbreviations in names should be avoided.*

```
computeAverage();               // NOT: compAvg();
ActionEvent event;              // NOT: ActionEvent e;
catch (Exception exception) {   // NOT: catch (Exception e) {
```

*Negated Boolean variable names must be avoided.*

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

*Associated constants (final variables) should be prefixed by a common type name.*

```
final int  COLOR_RED   = 1;
final int  COLOR_GREEN = 2;
final int  COLOR_BLUE  = 3;
```

*Exception classes should be suffixed with Exception.*

```
class AccessException extends Exception
{
  :
}
```

*Default interface implementations can be prefixed by Default.*

```
class DefaultTableCellRenderer
  implements TableCellRenderer
{
  :
}
```

*Singleton classes should return their sole instance through method getInstance.*
*Classes that creates instances on behalf of others (factories) can do so through method new[ClassName]*

```
class PointFactory
{
  public Point newPoint(...)
  {
    ...
  }
}
```

*Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do.*

## 3.2 Files

*Special characters like TAB and page break must be avoided.*
*The incompleteness of split lines must be made obvious*

```
totalSum = a + b + c +
           d + e;

method(param1, param2,
       param3);
```

## 3.3 Statements

### *3.3.1 Package and Import Statements*

*The import statements must follow the package statement. import statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups.*

*Imported classes should always be listed explicitly.*

```
import java.util.List;      // NOT: import java.util.*;
```

### *3.3.2 Classes and Interfaces*

*Class and Interface declarations should be organized in the following manner:*

- Class/Interface documentation.
- *class* or *interface* statement.
- Class (static) variables in the order public, protected, package (no access modifier), private.
- Instance variables in the order public, protected, package (no access modifier), private.
- Constructors.
- Methods (no specific order).

### *3.3.3 Methods*

*Method modifiers should be given in the following order: <access> static abstract synchronized <unusual> final native The <access> modifier (if present) must be the first modifier.*

```
public static double square(double a);  // NOT: static public double square(double
a);
```

### *3.3.4 Types*

*Type conversions must always be done explicitly. Never rely on implicit type conversion.*

```
floatValue = (int) intValue; // NOT: floatValue = intValue;
```

*Array specifiers must be attached to the type not the variable.*

```
int[] a = new int[20];   // NOT: int a[] = new int[20]
```

### 3.3.5 Variables

*Variables should be initialized where they are declared and they should be declared in the smallest scope possible.*
*Variables must never have dual meaning.*
*Class variables should never be declared public.*
*Arrays should be declared with their brackets next to the type.*

```
double[] vertex;  // NOT: double vertex[];
int[]    count;   // NOT: int    count[];
```

*Variables should be kept alive for as short a time as possible.*

### 3.3.6 Loops

*Only loop control statements must be included in the for() construction.*

```
sum = 0;                          // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)                   sum += value[i];
  sum += value[i];
```

*Loop variables should be initialized immediately before the loop.*

```
isDone = false;           // NOT: bool isDone = false;
while (!isDone) {         //      :
  :                       //      while (!isDone) {
}                         //        :
                          //      }
```

*The use of do-while loops can be avoided.*
*The use of break and continue in loops should be avoided.*

### 3.3.7 Conditionals

*Complex conditional expressions must be avoided. Introduce temporary boolean variables instead*

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
}
```

*The nominal case should be put in the if-part and the exception in the else-part of an if statement*

```
boolean isOk = readFile(fileName);
if (isOk) {
  :
}
else {
  :
}
```

*The conditional should be put on a separate line.*

```
if (isDone)       // NOT: if (isDone) doCleanup();
  doCleanup();
```

*Executable statements in conditionals must be avoided.*

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
  :
```

```
}

// NOT:
if (File.open(fileName, "w") != null)) {
  :
}
```

### 3.3.8 Miscellaneous

*The use of magic numbers in the code should be avoided. Numbers other than 0 and 1can be considered declared as named constants instead.*

```
private static final int  TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = new Player[11];
```

*Floating point constants should always be written with decimal point and at least one decimal.*

```
double total = 0.0;    // NOT:  double total = 0;
double speed = 3.0e8;  // NOT:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

*Floating point constants should always be written with a digit before the decimal point.*

```
double total = 0.5;  // NOT:  double total = .5;
```

*Static variables or methods must always be referred to through the class name and never through an instance variable.*

```
Thread.sleep(1000);    // NOT: thread.sleep(1000);
```

## 3.4 Comments

*Tricky code should not be commented but rewritten.*
*All comments should be written in English.*
*Javadoc comments should have the following form:*

```
/**
 * Return lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x    X coordinate of position.
 * @param y    Y coordinate of position.
 * @param zone Zone of position.
 * @return     Lateral location.
 * @throws IllegalArgumentException  If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
  throws IllegalArgumentException
{
  ...
}
```

*There should be a space after the comment identifier.*

```
// This is a comment    NOT: //This is a comment

/**                     NOT: /**
 * This is a javadoc           *This is a javadoc
 * comment                     *comment
 */
```

*Use // for all non-JavaDoc comments, including multi-line comments.*

```
// Comment spanning
// more than one line.
```

*Comments should be indented relative to their position in the code.*

```
while (true) {        // NOT:  while (true) {
  // Do something               // Do something
  something();                    something();
}
```

*All public classes and public and protected functions within public classes should be documented using the Java documentation (javadoc) conventions.*

# 4. C# coding guidelines

## 4.1 Naming conventions

### *4.1.1 General naming conventions*

*Names representing packages should be in all lower case.*
**All code should be in dsd.ehealth.* namespace and sub namespaces.**

*Names representing types must be nouns and written in mixed case starting with upper case.*

```
Line, AudioSystem
```

*Variable names must be in mixed case starting with lower case.*

```
line, audioSystem
```

*Names representing constants (final variables) must be all uppercase using underscore to separate words.*

```
MAX_ITERATIONS, COLOR_RED
```

*Names representing methods must be verbs and written in pascal case starting with upper  case.*

```
GetName(), ComputeTotalWidth()
```

*Abbreviations and acronyms should not be uppercase when used as name.*

```
ExportHtmlSource(); // NOT: ExportHTMLSource();
OpenDvdPlayer();    // NOT: OpenDVDPlayer();
```

*Private class variables should have underscore preffix.*

```
class Person
{
  private String _name;
  ...
}
```

*Generic variables should have the same name as their type.*

```
void SetTopic(Topic topic) // NOT: void SetTopic(Topic value)
                           // NOT: void SetTopic(Topic aTopic)
                           // NOT: void SetTopic(Topic t)
```

*All names should be written in English.*
*Variables with a large scope should have long names, variables with a small scope can have short names*

```
line.GetLength();   // NOT: line.GetLineLength();
```

*The name of the object is implicit, and should be avoided in a method name.*

```
line.GetLength();   // NOT: line.GetLineLength();
```

### *4.1.2 Specific naming conventions*

*is prefix should be used for Boolean variables and methods.*

```
isSet, isVisible, IsFinished(), IsFound(), IsOpen()
```

*The term compute can be used in methods where something is computed.*

```
valueSet.ComputeAverage();
matrix.ComputeInverse()
```

*The term find can be used in methods where something is looked up.*

```
vertex.FindNearestVertex();
matrix.FindSmallestElement();
node.FindShortestPath(Node destinationNode);
```

*The term initialize can be used where an object or a concept is established.*

```
printer.InitializeFontSet();
```

*Plural form should be used on names representing a collection of objects.*

```
Collection<Point>  points;
int[]              values;
```

*n prefix should be used for variables representing a number of objects.*

```
nPoints, nLines
```

*No suffix should be used for variables representing an entity number.*

```
tableNo, employeeNo
```

*Iterator variables should be called i, j, k etc.*
*Complement names must be used for complement entities*

```
Get/set, add/remove, create/destroy, start/stop, Insert/Delete,
Increment/Decrement, Old/New, Begin/End, First/Last, Up/Down, Min/Max,
Next/Previous, Old/New, Open/Close, Show/Hide, Suspend/Resume, etc.
```

*Abbreviations in names should be avoided.*

```
ComputeAverage();               // NOT: CompAvg();
ActionEvent event;              // NOT: ActionEvent e;
Catch (Exception exception) {   // NOT: Catch (Exception e) {
```

*Negated Boolean variable names must be avoided.*

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

*Associated constants (final variables) should be prefixed by a common type name.*

```
final int  COLOR_RED   = 1;
final int  COLOR_GREEN = 2;
final int  COLOR_BLUE  = 3;
```

*Exception classes should be suffixed with Exception.*

```
class AccessException extends Exception
{
  :
}
```

*Default interface implementations can be prefixed by Default.*

```
class TableCellRenderer : DefaultTableCellRenderer
{
  :
}
```

*Singleton classes should return their sole instance through method GetInstance.*
*Classes that creates instances on behalf of others (factories) can do so through method New[ClassName]*

```
class PointFactory
{
  public Point NewPoint(...)
```

```
  {
    ...
  }
}
```

*Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do.*

## 4.2 Files

*Special characters like TAB and page break must be avoided.*
*The incompleteness of split lines must be made obvious*

```
totalSum = a + b + c +
           d + e;

Method(param1, param2,
       param3);
```

## 4.3 Statements

### 4.3.1 Namespace and Using Statements

*The using statements must follow the namespace statement. Using statements should be sorted with the most fundamental namespace first, and grouped with associated namespaces together and one blank line between groups.*

*Imported classes should always be listed explicitly.*

```
Using System.Web.Optimization;
```

### 4.3.2 Classes and Interfaces

*Class and Interface declarations should be organized in the following manner:*

- Class/Interface documentation.
- *class* or *interface* statement.
- Class (static) variables in the order public, protected, private.
- Instance variables in the order public, protected, private.
- Constructors.
- Methods (no specific order).
- Attributes

### 4.3.3 Methods

*Method modifiers should be given in the following order: <access> static abstract synchronized <unusual> final native The <access> modifier (if present) must be the first modifier.*

```
public static double square(double a);  // NOT: static public double square(double
a);
```

### 4.3.4 Types

*Type conversions must always be done explicitly. Never rely on implicit type conversion.*

```
floatValue = (float) intValue; // NOT: floatValue = intValue;
```

*Array specifiers must be attached to the type not the variable.*

```
int[] a = new int[20];   // NOT: int a[] = new int[20]
```

### 4.3.5 Variables

*Variables should be initialized where they are declared and they should be declared in the smallest scope possible.*

*Variables must never have dual meaning.*
*Class variables should never be declared public.*
*Arrays should be declared with their brackets next to the type.*

```
double[] vertex;  // NOT: double vertex[];
int[]    count;   // NOT: int    count[];
```

*Variables should be kept alive for as short a time as possible.*

### 4.3.6 Loops

*Only loop control statements must be included in the for() construction.*

```
sum = 0;                          // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)                 {
{                                            sum += value[i];
  sum += value[i];                        }
}
```

*Loop variables should be initialized immediately before the loop.*

```
isDone = false;           // NOT: bool isDone = false;
while (!isDone)           //      while (!isDone)
{                         //      {
   …                      //        …
}                         //      }
```

*The use of do-while loops can be avoided.*
*The use of break and continue in loops should be avoided.*

### 4.3.7 Conditionals

*Complex conditional expressions must be avoided. Introduce temporary boolean variables instead*

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry)
{
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement)
{
  :
}
```

*The nominal case should be put in the if-part and the exception in the else-part of an if statement*

```
boolean isOk = readFile(fileName);
if (isOk)
{
  :
}
else {
  :
}
```

*The conditional should be put on a separate line.*

```
if (isDone)       // NOT: if (isDone) doCleanup();
  doCleanup();
```

*Executable statements in conditionals must be avoided.*

```
InputStream stream = File.open(fileName, "w");
```

```
if (stream != null)
{
  :
}

// NOT:
if (File.open(fileName, "w") != null))
{
  :
}
```

### 4.3.8 Miscellaneous

*The use of magic numbers in the code should be avoided. Numbers other than 0 and 1can be considered declared as named constants instead.*

```
private static final int  TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = new Player[11];
```

*Floating point constants should always be written with decimal point and at least one decimal.*

```
double total = 0.0;    // NOT:  double total = 0;
double speed = 3.0e8;  // NOT:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

*Floating point constants should always be written with a digit before the decimal point.*

```
double total = 0.5;  // NOT:  double total = .5;
```

*Static variables or methods must always be referred to through the class name and never through an instance variable.*

```
Thread.sleep(1000);    // NOT: thread.sleep(1000);
```

## 4.4 Comments

*Tricky code should not be commented but rewritten.*
*All comments should be written in English.*
*Methods should be comented with the description, parameter and return coment:*

```
/// <summary>
/// Coments
/// </summary>
/// <param name="property">Coments</param>
/// <returns>Coments</returns>
public int Method(string property)
{
  :
}
```

*There should be a space after the comment identifier.*

```
// This is a comment    NOT: //This is a comment
```

*Multi-line comments are allowed.*

```
// Comment spanning
// more than one line.
```

*Comments should be indented relative to their position in the code.*

```
while (true) {        // NOT:  while (true) {
  // Do something              // Do something
  something();                   something();
}
```

*All public classes and public and protected functions within public classes should be documented using the Java documentation (javadoc) conventions.*