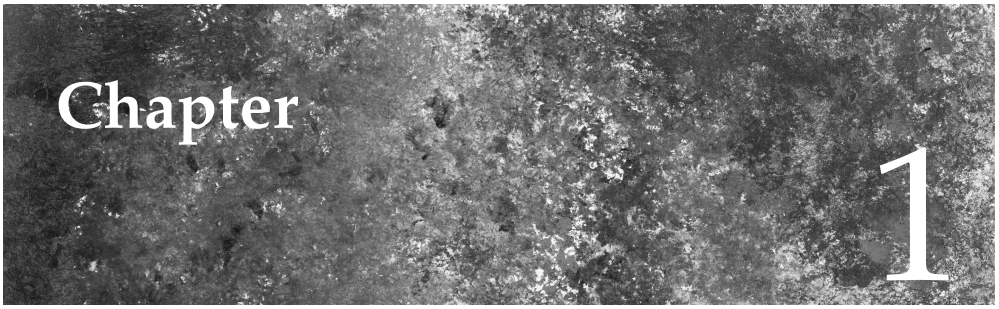
Uncle Henrik's Clean Code Principles

Henrik Bærbak Christensen

Status: Draft / Revision 19.04

August 7, 2019



Clean Code

In this document, I will outline some additional clean code principles beyond those of Martin (2009, Chapter 3) that I will argue improve code maintainability.

1.1 Do the Same Thing, the Same Way

Humans are not good at remembering large sets of random information. But if the information follows a template or a recognizable format or structure, it becomes much easier: Once the template is memorized, it can be used to create/understand a lot of examples.

1.1.1 The Problem

One war story of *not* using the standard template stems from a C project I was shortly enrolled in. The C standard library has a method to copy the contents of one string into another, strcpy, so you get a copy of the first string.

```
1  char *strcpy(char *dest, char *src)
```

(Java Strings are immutable objects, so Java does not have a similar method, as it does the copy 'behind the scenes'.)

Note that the type of the two parameters are the same (read 'char *' as something similar to String in java), but the important bit of information is that the destination string is the **first** argument, while the source string is the **second** one. After having coded using this function literally thousands of times, this template: destination is first parameter, source is second; was of course carved into my brain.

Now, I was asked to help out in a project in which my collegue had coded his own String implementation, DSEString, and of course provided

```
1  void strcpy(DSEString s1, DSEString s2) {... }
```

which was a method to copy the contents of one string into another. So I began writing a simple algorithm, using this function. It did not work to my surprise. I read the code again and again without being able to spot the problem, and finally had to debug for quite some time. It turned out—my collegue had *reversed the order of the parameters: the source string was the first parameter, and the destination was the second, in his* strcpy() *method!* **The same thing, done in a different way.**

A more recent example from the HotCiv system is the following implementation of moveUnit(Position from, Position to) that some students of mine made years ago.

The have introduced two Map structures for handling tiles (World) and for handling units (unitMap):

```
1 public class GameImpl implements Game {
2   private Map<Position,Tile> world = new HashMap<>();
3   private Map<Position,UnitImpl> unitMap = new HashMap<>();
4   ...
```

Even here, you will notice that the naming of instance variables are not "same naming for same things": Why not call them "tileMap" and "unitMap"? Why the Map suffix on one of the instance variables and not the other?

Then the first part of the implementation goes:

```
1 private boolean moveRedUnit(Position from, Position to) {
2   if (unitMap.get(from) != null &&
3       !world.get(to).getTypeString().equals(GameConstants.OCEANS) &&
4       unitMap.get(from).getOwner() == Player.RED &&
5       !getTileAt(to).getTypeString()
6           .equals(GameConstants.MOUNTAINS) &&
7       getUnitAt(from).getMoveCount() >= 1 &&
8       Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
9       Math.abs(from.getRow() - to.getRow()) <= 1 &&
10      ! to.equals(from) ) {
11    ...
```

As you know, Game has accessor methods for getting a unit at a specific position: getUnitAt(p) which is used in line 6. But the accessor is not used consequently, in line 2 a direct call into the data structure is used instead. So - two different ways of doing the same thing is used.

This is problematic - first because it makes the code less analyzable, the reader has to understand and know two different ways of getting units, and here actually know that they are algorithmic identical. Second, if we want to *change* the data structure used for storing units (say, into a matrix), the code will fail: the places in which getUnitAt(p) is used will work correctly but the direct accesses will of course not. This leads to many more places to make changes, lowering the changeability quality (Christensen 2010, Chapter 3).

So - this principle is often closely related to uncle Bob's "One Level of Abstraction": The moveUnit() method should keep its code at a higher level of abstraction than direct data structure access, namely only using accessor methods, shielding it from changes at the lower level of abstraction, the data structure choice.

1.1.2 The Solution

Cultivate a team and personal culture of sticking to an agreed schema, template, way of doing things.

In internal code, refactor the code as you spot any "same thing in a different way" code. In external code, say for a library that many people are depending upon, this is much more troublesome as any change you may make will affect a lot of people.

In the concrete HotCiv code above, I would *change all direct datastructure accesses into accessor method calls* so all accessing is done *the same way*, like

```
1  private boolean moveRedUnit(Position from, Position to) {
2    if (getUnitAt(from) != null &&
3        !getTileAt(to).getTypeString().equals(GameConstants.OCEANS) &&
4        getUnitAt(from).getOwner() == Player.RED &&
5        !getTileAt(to).getTypeString()
6            .equals(GameConstants.MOUNTAINS) &&
7        getUnitAt(from).getMoveCount() >= 1 &&
8        Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
9        Math.abs(from.getRow() - to.getRow()) <= 1 &&
10       ! to.equals(from) ) {
11     ...
```

1.2 Name Boolean Expressions

Some programmers may be good at overviewing complex boolean expressions with many NOT, AND, and OR operators. I am not and from helping out my students on countless occassions I guess I am not the only one.

1.2.1 The Problem

Maintainability is a measure of how easy it is to modify code, and a central aspect of that is analyzability (Christensen 2010, Chapter 3): my ability to understand the behavior of the code correctly and easily by simply *reading* it. Have a look at the moveRedUnit() method's many boolean expressions—it takes quite a while to read and fully grasp. And it is even of the simple variant, with only AND and NOT operators, no parentheses nor OR operators.

1.2.2 The Solution

My suggesting is to give meaningfull names to logically related boolean sub expressions. This is probably best explained by an example. Again, returning to the moveRedUnit(), there is a portion of the if-statement that reads:

```
1  private boolean moveRedUnit(Position from, Position to) {
2    if (getUnitAt(from) != null &&
3        ...
4        Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
5        Math.abs(from.getRow() - to.getRow()) <= 1 &&
```

Looking at lines 4–5, what does these two lines express? Well, they express that the distance we are trying to move a unit is less than or equal to one tile, which is the requirement of a legal move. So - I will increase analyzability by *naming the boolean expression*:

```
1 private boolean moveRedUnit(Position from, Position to) {
2   boolean moveDistanceIsOneOrLess =
3       Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
4       Math.abs(from.getRow() - to.getRow()) <= 1;
5
6   if (getUnitAt(from) != null &&
7     ... &&
8     moveDistanceIsOneOrLess &&
9     ...
```

Continuing this process with the rest of boolean expressions you may end up with an if-statement like:

```
1 private boolean moveRedUnit(Position from, Position to) {
2   ...
3
4   if (existsUnitOnFromTile &&
5     toTileIsValidTerrain &&
6     fromUnitIsMyOwn &&
7     moveCountIsOneOrMore &&
8     moveDistanceIsOneOrLess &&
9     ! toAndFromAreTheSameTile ) {
10      ...
```

Now, this expression is much closer to the language of the specification of HotCiv: A move is valid if there is a unit to move, if you move to a valid terrain, if you move your own unit, and that unit has sufficient moves left, you do not move more than one tile, etc.

One word of caution: Sometimes I have found myself naming an expression something like notValidTerrain, i.e. with a *not* in the variable name. However, then I often find myself using it in expressions like

```
1   if (! notValidTerrain) { ... }
```

Which is profoundly confusing as you figure out what NOT-NOT means. So, keep the NOT operators out of the named boolean expressions. Generally I like to *do the same thing, the same way* so I try to name my boolean expression in a *positive/desireable* way: battleWon, attackSucceed, toTileIsValidTerrain, etc., as it will make the NOT operator stand more clearly out. I find it easier to read

```
1   if (fileIsOpen) { /* read it */ }
```

than

```
1   if (! fileFailedInOpening) { /* read it */ }
```

It is desirable that the file has been opened so I can read it.

1.3 Bail Out Fast

If and while-statements can be nested to any level without the compiler or java runtime having problems. However, deep nesting levels are hard to understand and analyze.

1.3.1 The Problem

Below, the HotCiv code has been rewritten using nested if statements which is an alternative to using AND operators.

```
if (unitMap.get(from) != null) {
  if (!world.get(to).getTypeString().equals(GameConstants.OCEANS)) {
    if (unitMap.get(from).getOwner() == Player.RED) {
      if (!getTileAt(to).getTypeString()
             .equals(GameConstants.MOUNTAINS)) {
        if (getUnitAt(from).getMoveCount() >= 1) {
          if (Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
              Math.abs(from.getRow() - to.getRow()) <= 1) {
            if (! to.equals(from) ) {
              if ( getUnitAt(to) == null ) {

                // move is allowed to proceed to move it
                unitMap.put(to, unitMap.get(from));
                unitMap.put(from, null);

                // decrement move count
                unitMap.get(to).decrementMoveCount();
                return true;
              } else {
                return false;
              }
            } else {
              return false;
            }
          } else {
            return false;
          }
        } else {
          return false;
        }
      } else {
        return false;
      }
    } else {
      return false;
    }
  } else {
    return false;
  }
} else {
  return false;
}
```

It suffers from several analyzability issues. First, you have to keep track of how many previous conditions are already "in effect" when you try to understand at the code at

a given level. Second, finding the right "else" branch to modify code in is somewhat of a puzzle that you can easily get wrong (the code below is in that respect very well behaved because they are all just return statements, but consider that you need to add a statement in the else branch part of the condition about "not moving to a mountain"—will you hit the right code part?). I have seen people adopting the convention of putting a remark after the else to remind themselves like:

```
1  ...
2              } else { // moving to a mountain
3                return false;
4              }
```

If you find yourself using such "code crutches" then reconsider!

Finally, visually the "real" algorithmic code creeps more and more to the left side of the screen which makes me wonder "What does it do there?" Perhaps not much of an issue, but still...

Some old programming languages, notably Pascal, more or less forced you into this style, as it had no return statement.

1.3.2 The Solution

I prefer this kind of reasoning: "If I can compute an answer early, I will return it early, and then stop bothering about it in the code that comes next." If there is no unit on the 'from' tile then I know that the move is illegal, and the rest of the code in the method is irrelevant. I will code to *bail out fast*.

In our case, I would code it like[1]:

```
1  private boolean moveRedUnit(Position from, Position to) {
2    boolean existUnitOnFrom = getUnitAt(from) != null;
3    if (! existUnitOnFrom) return false;
4
5    fromUnitIsMyOwn = getUnitAt(from).getOwner() == getPlayerInTurn();
6    if (! fromUnitIsMyOwn) return false;
7
8    ...
9
10   boolean moveDistanceIsOneOrLess =
11     Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
12     Math.abs(from.getRow() - to.getRow()) <= 1;
13   if (! moveDistanceIsOneOrLess) return false;
14
15   ...
```

This principle also tend to allow me to code the "easy" parts first—the cases where an answer can be computed easily are at the top of the method, with an immediate return statement.

Bail out fast also often makes "guarding method calls" unnecessary in the later parts of the method's algorithm. For instance, the call to getMoveCount() in the second line of:

[1] Note that all boolean expressions are again expressed as *desirable* properties.

```
1  boolean moveCountIsOneOrMore =
2    getUnitAt(from).getMoveCount() >= 1;
```

will throw a null pointer exception in case there is no unit at the 'from' tile. Thus, you have to "guard it", either by yet-another if statement or by boolean short-circuit evaluation:

```
1  boolean moveCountIsOneOrMore =
2    getUnitAt(from) != null &&
3    getUnitAt(from).getMoveCount() >= 1;
```

By bailing out fast, the guard becomes unnecessary, as we already know as this point that there *is* a unit on the from tile.

Modern IDEs can help in the refactoring. For instance, IntelliJ allows marking an expression and then select 'Refactor/Extract/Variable' to ease the coding effort.

1.4 Arguments in Argument Lists

The last principle is about an issue that I sometimes stumble upon in student code. Maybe it is due to being novices in doing test-driven development because I can imagine a thinking along the lines of "Let us make this method work for red player first, (ongoing TDD work), Ok—it is working so to get it to handle blue player as well, let us copy it and substitute all occurences of 'red' with 'blue'."

1.4.1 The Problem

Consider the following implementation of the HotCiv moveUnit() method:

```
1  public boolean moveUnit(Position from, Position to) {
2    boolean v;
3    if (playerInTurn == Player.RED) {
4      v = moveRedUnit(from,to);
5    } else
6      v = moveUnitForBluePlayer(from, to);
7    return v;
8  }
9
10 private boolean moveRedUnit(Position from, Position to) {
11   [Long complex implementation here, often referring to Player.RED]
12 }
13 private boolean moveUnitForBluePlayer(Position from, Position to) {
14   [EXACT COPY of the above method, except all occurences of
15     Player.RED has been replaced with PLAYER.BLUE]
16 }
```

Of course, this is an obvious example of code duplication, and therefore the code does not obey the "Don't Repeat Yourself" principle by Martin (2009, Chapter 3). As argued at length in Christensen (2010, Chapter 7) duplicated code leads to all sorts of maintainability problem, and I will not repeat them here. Just consider what happens, when someone requires that the code can handle yellow and green player as well.

However, the indicator of the code duplication is in the naming of the methods: moveRedUnit and moveUnitForBluePlayer. The mentioning of blue and red directly in the method names indicate that the methods cannot cope with any type of player, only a specific one. The player color is an *argument* to a method, but now it appears in the method's name.

A similar and even more pronounced example is adding tax to a sales item. Different countries in EU have different tax rates so a suitable solution is to have the tax rate as an argument. And you do NOT do that by making a lot of methods

```
1  public double addTaxOf25Pct(double amount);
2  public double addTaxOf20Pct(double amount);
3  public double addTaxOf12Pct(double amount);
4  ...
```

1.4.2 The Solution

Arguments shall not be part of a method's name, instead put *arguments in argument lists*—provide it as a parameter to the method when calling it.

In our moveUnit example we actually do not need the player color parameter in the argument list at all—the method can of course query it from the unit it is trying to move.

For the salex tax case, put the *argument in the argument list*:

```
1  public double addTax(double amount, double taxrate);
```

And implement a general algorithm.

One exception to this rule are test case methods which are often named to be specific about parameters and values to highlight the exact test case being defined:

```
1  @Test
2  public void shouldAllowMoveRedArcherToEmptyPlain() { ... }
3  @Test
4  public void shouldNotAllowMoveRedArcherToMountain() { ... }
5  @Test
6  public void shouldAcceptLegionKillArcher() { ...}
7  ...
```

Hardcoding values is problematic in production code as algorithms are then not generalized; however test code algorithms must emphasize analyzability ("Evident Test" and "Evident Data" principles of TDD) and are meant to be a specific test case, not a general algorithm.

1.5 Video Tutorial

You can find ten video tutorials on my Vimeo account that refactor the HotCiv code using the principles in this paper as well as applying those of Martin (2009).

1. `https://vimeo.com/219059474` Session 1 / Intro

2. https://vimeo.com/220629645 Session 2 / Duplication I
3. https://vimeo.com/220630188 Session 3 / 'If' cleanup
4. https://vimeo.com/220632096 Session 4 / Duplication II
5. https://vimeo.com/220632562 Session 5 / Remove nesting
6. https://vimeo.com/220632796 Session 6 / One level of abstraction
7. https://vimeo.com/220633134 Session 7 / Commenting
8. https://vimeo.com/220633530 Session 8 / One level of abstraction
9. https://vimeo.com/220634076 Session 9 / Bail out fast
10. https://vimeo.com/234478442 Session 10 / One Level of Abstraction

Bibliography

Christensen, H. B. (2010). *Flexible, Reliable Software - Using Agile Development and Patterns*. Chapman & Hall, CRC.

Martin, R. C. (2009). *Clean Code - A Handbook of Agile Software Craftsmanship*. Pearson Education.