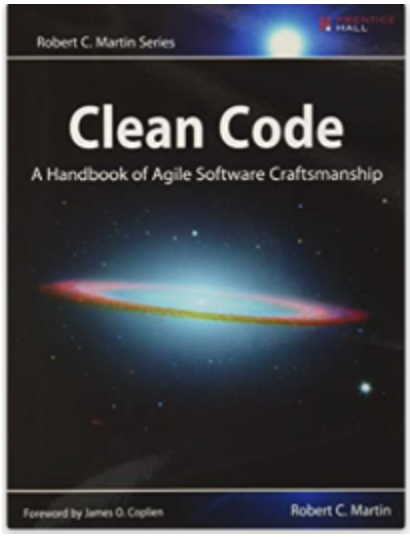
Clean Code Notes



The goal is to keep keep class and function count low, as well as have tests, eliminate duplication, and express ourselves.

Chapter 1

Just about why we need clean code:

As the mess builds, the productivity of the team continues to decrease, more and more mess wil be made.

LeBlanc's law: Later equals never.

Chapter 2 Meaningful Names

1. Name should tell you why it exists, what it does, and how it is used.
2. Avoid leaving false clues that obscure the meaning of the code, such as abbreviations which have different meanings.
3. Do not refer to a group of accounts `accountList` if it's not a `List`.
4. Do not use names which vary in a small way, such as our `MKTheme` and `MKThemer`.

5. Pay attention to lower-case `L` and uppercase `O` (because they are similar to 1 and 0).
6. Make meaningful distinctions, names for example `a1` and `a2`, `Info` and `Data` have no meaningful distinctions.
7. Make the names pronounceable.
8. Use searchable name, in this regard, longer names better than shorter name, searchable name better than a constant (such as 5, we sometimes need to guess what a number means), the length of a name should correspond to the size of its scope.
9. Don't use useless prefixes, such as the `m_p` prefixes in our code, the `p` looks useless and people tend to just ignore it.
10. This book says we don't need to indicate it's a interface in the name of an interface, like we do in the code putting `Interface` in the end of the name, the book suggests we just indicate it's an implementation in the name of the implementation. But I think it's better to keep doing what we do now, it's easier to read.
11. Try not use single-letter variable name except ina loop counter because other readers have to mentally map these names.

12. Class names use noun or noun phrase names, not a verb. Class names should be specific, not too general such as `Data`.
13. Method names use verb or verb phrase names. Accessors, mutators and predicator use `get`, `set` and `is` in the names. When constructors are overloaded, use static factory methods with names that describe the arguments and make the corresponding constructor private, for example:

    ```
    auto menu = MKMenu::contextMenu(label);
    ```

    instead of

    ```
    auto menu = new MKMenu(label)
    ```

14. Be consistent for the name, use one word for one abstract concept, for example, don't use `fetch`, `retrieve` and `get` if they mean one concept, just use get everywhere. Also don't use the same word for two purpose.

15. Use solution domain names such as CS terms, algorithm names.
16. Add meaningful context.
17. Don't add gratuitous context, for example, for enum `DrawFlags` in the `MKInputBoxInputControl`, `DrawTxt`, `DrawSelections`, `DrawCursor`, `DrawAll` can be changed to `Txt`, `Selections`, `Cursor` and `All` without putting `Cursor` in every name.

Chapter 3 Functions

1. Make the function as small as possible.
2. The blocks in the `if`, `else` amd `while` statement should be one line long(a function call with a nicely descriptive name maybe). For functions, the indent level should not be greater than one or two, for example, `MKDeviceConnectDialog::prepareDevice` and `MKInstrumentSettingsContainer::updateSettingsCache` in our code has too many levels.
3. Functions should only do one thing. It can be a few steps of one thing. It can't be steps at many different levels.
4. One level of abstration per function

```
void MKInstrumentSettingsContainer::updateSettingsCache()
{
    // Update m_settings if a panel has had a new setting added/removed since last setting
m_settings
    if (m_settingsCacheInvalidated) {
        m_settings.clear();
        for (auto _tab : tabs()) {
            if (auto tab = dynamic_cast<MKInstrumentSettingsTab*>(_tab)) {
                for (auto _panel : tab->panels()) {
                    if (auto panel = dynamic_cast<MKInstrumentSettingsPanelNew*>(_panel))
 {
                        m_settings.append(panel->settings());
                    }
                }
            }
        }
        m_settingsCacheInvalidated = false;
    }
}
```

in this theory, it should be:

```
void MKInstrumentSettingsContainer::updateSettingsCache()
{
    // Update m_settings if a panel has had a new setting added/removed since last setting
m_settings
    if (m_settingsCacheInvalidated) {
        m_settings.clear();
        updateAllTabsSettings();
        m_settingsCacheInvalidated = false;
    }
}
```

5. Reading code from top to bottom : The Stepdown Rule - every function is followed by those at the next level of abstraction so that the program can be read easily.
6. Try to avoid using `switch` statement, because it always do N things. If we have to, make sure it is buried in a low level class and never repeated.
7. Use descriptive names. Don't be afraid to make long names.
8. The ideal number of arguments for a function is 0, then followed by 1, 2. 3 should try to be avoided. arguments force reader to interpret them everytime they see them, harder to understand the code.
9. Use returning value instead of output argument to avoid confusion.
10. Avoid passing boolean into a function, which will force a function to do more than one thing. Change this kind of function into two functions, such as `MKBDGain::updateInvertButton(bool invert)` as below:

```
void MKBDGain::updateInvertButton(bool invert)
{
    if (m_buttonInvert.isNull()) return;

    m_invert = invert;
    if (invert) {
        m_buttonInvert->setStyleSheet("background-color: #11c3ff; border: none; font-weigh
t: bold; color: rgba(240, 240, 240)");
    } else {
        if (li::colors::scheme) {
            m_buttonInvert->setStyleSheet("background-color: rgba(240, 240, 240); border:
 none; font-weight: bold; color: #11c3ff");
        } else {
            m_buttonInvert->setStyleSheet("background-color: #383838; border: none; font-w
eight: bold; color: #11c3ff");
```

```
        }
    }
    repaint();
}
```

can be changed to

```
void MKBDGain::updateInvertButton()
{
    if (m_buttonInvert.isNull()) return;

    m_buttonInvert->setStyleSheet("background-color: #11c3ff; border: none; font-weight: b
old; color: rgba(240, 240, 240)");
    repaint();
}

void MKBDGain::updateNotInvertButton()
{
    if (m_buttonInvert.isNull()) return;

    if (li::colors::scheme) {
        m_buttonInvert->setStyleSheet("background-color: rgba(240, 240, 240); border: non
e; font-weight: bold; color: #11c3ff");
    } else {
        m_buttonInvert->setStyleSheet("background-color: #383838; border: none; font-weigh
t: bold; color: #11c3ff");
    }
    repaint();
}
```

11. for a function with two arguments for example `writeField(outputStream, name)`, we can simplify it by making the `writeField` method a member of `outputStream` like `outputStream.writeField(name)`, or making `outputStream` a member variable of the current class, or extract a new class `FieldWriter` that takes the `outputStream` in its constructor and has a `write` method.
12. Reduce the arguments by wrapping some of the arguments into a class. for example in below code, `title` and `text` can be wrapped into a simple struct `MKMessage`:

```
static MKMessageBox* information(const QString& title, const QString& text, MKDialogButto
n::MKDialogButtons buttons = MKDialogButtonType::Accept, MKDialogButtonType defaultButton
 = MKDialogButtonType::Accept);
```

13. Use verb/noun pair for function name such as `writeField(name)`. Use key word, for example, `assertExpectedEqualsActual(expected, actual)` is better than `assertEquals`, because you can see the argument order from the name.
14. No side effect.
15. Try to avoid output arguments, for example, `appendFooter(StringBuffer report)` can be `report.appendFooter()`.
16. Functions should do something (change something for an object) or answer something (return some information), but not both.
17. prefer exceptions to returning error codes.
18. Extract `try` and `catch` block into functions of their own to avoid confusion of the structure of the code.
19. Functions handling errors should do nothing else, because functions should do one thing.
20. Avoid duplication.

Chapter 4 Comments

Comments means the failure to express ourselves in code, and it takes time to maintain, we need to try to avoid comment.

1. Try to avoid comment by explain ourselves in code.
2. Good comment: Legal comments - refer to external document rather than putting all the terms and conditions in the comment; Informative comments; Explanation of intent; Clarification; Warning of consequences; TODO comment - scan through them regularly and eliminate the ones we can; Amplification.
3. Bad comment: Mumbling - force you to look at other modules for the meaning of the comment; Redundant comments - for example, there is some `// Draw text` or `// Draw icon` comments which the function's intent is pretty obvious; Misleading comment; Mandated comments; Journal comments - some kind of old fashion comment when there is no source code control system; Noise comments;.
4. Don't use comment when we can use a function or a variable to explain, give the functions and variables good names.

5. Don't overuse position marker, such as the class name comment we put in the beginning of every class, because the writer believes that overusing this kind of comment will make people just ignore them.
6. Don't do commented-out code.
7. Don't offer systemwide information in a local comment, because you will forget to change it when you change that information somewhere else.
8. Avoid too much infomation in the comment. For example below code in `mk_dialog.cpp`, maybe we just put these comment in the DESKTOP-1468 ticket and leave one lone comment that there is a TODO of DESKTOP-1468?

```
// TODO(DESKTOP-1468): Re-enable this when rendering issues have been solved
    // if (!canClose()) return;  // Already closing, prevents multiple close animations or
premature closing

    // First, we need to remove old effects, because Qt can only have 1 effect assigned to
a widget...
    // m_dialog->removeDropshadow();

    // Now start to fade out
    // auto fadeEffect = new QGraphicsOpacityEffect(this);
    // auto opacityAnimation = new QPropertyAnimation(fadeEffect, "opacity");
    // setGraphicsEffect(fadeEffect);
    // opacityAnimation->setDuration(250);
    // opacityAnimation->setStartValue(1);
    // opacityAnimation->setEndValue(0);
    // opacityAnimation->setEasingCurve(QEasingCurve::OutBack);
    // opacityAnimation->start(QPropertyAnimation::DeleteWhenStopped);
    // connect(opacityAnimation, &QPropertyAnimation::finished, [this, fadeEffect] {
    //     fadeEffect->deleteLater();
    //     close();
    // });
```

1. Giving a good name for a short function is better than giving it a coment.

Chapter 5 Formatting

Vertical formating

1. Use blank lines between concepts.
2. Avoid vertical density - line of code that are tightly related.

3. Variables should be declared as close to their usage as possible.
4. If one function calls another, they should be vertically close and the caller should be above the callee.
5. Functions with a certain conceptual affinity should be close together.

Horizontal Formatting

1. Unalign declarations and assignments.

No other useful information, just some rules we are always following like indentation.

Chapter 6 Objects and Data Structures

1. Use getters and setters and give getters and setters an abstract term. so that we don't expose the details of our data.
2. Procedural code (code using data structure) -easy to add new functions but hard to add new data structures.

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Geometry {
  public final double PI = 3.14;
  public double area(Object shape) throw NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    } else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
  throw new NoSuchShapeException();
  }
}
```

3. Object-oriented code - easy to add new classes but hard to add new functions. `MKMenu` is a good example for OO code.

```
public class Square implements Shape {
  public Point topLeft;
  public double side;

  public double area() {
    return side * side;
  }
}

public class Rectangle implements Shape {
  public Point topLeft;
  public double height;
  public double width;

  public double area() {
    return height * width;
  }
}
```

4. The law of Demeter (to hide object's internal structure)- a method of `f` of a class `C` should only call the methods of * `C` * An object created by `f` * An object passed as an argument of `f` * An object held in an instance variable of `C`. This law doesn't apply to the data structure with no behavior.
5. Use train wreck to solve the violation of the law of Demeter.

```
const auto markerX = m_referenceTrace->getXAxis()->get()->coordToPixel(m_cursor->m_value);
```

should be changed to

```
auto xAxis = m_referenceTrace->getXAxis();
const auto markerX = xAxis->get()->coordToPixel(m_cursor->m_value);
```

6. Use data transfer objects structure when you have a database or you need to parse messages from sockets.

Chapter 7 Error Handling

1. Use Exception rather than return code
2. Provide context with exceptions
3. Use a wrapper that catch same exceptions (an exception class) in terms of a Calle's needs to avoid duplicated exception codes.
4. Create a class to handle special case so that we don't need to deal with exception behavior.
5. Don't return null or pass null. Maybe use assertions instead.

Chapter 8 Boundaries

1. When using a boundary interface from a third-party code like `Map`, keep it inside the specific class, having very few places that refer to it, don't pass it around the system(use it directly), because the thirs-party code may change. We need to ensure that once it changes, we change as least code as possible.
2. Use test to learn third-party code. (This is too much work, we can just learn it by trying to write it).

Chapter 9 Unit Tests

1. The number of asserts in a test ought to be minimized.
2. Single concept per test.
3. F.I.R.S.T rules. Tests should be fast; Tests should not depend on each other; Tests should be repeatable in any environment. The tests should have a boolean output. The tests need to be written in a timely fashion.

Chapter 10 Classes

1. Follow class organization convention. For example ours are public functions, then private functions then private variables.
2. Avoid loosen encapsulation, try to keep variables and functions private.
3. Classes should be small. Small means not too many responsibilities.

4. The Single Responsibility Principle - a class should have only one responsibility, collaborates with a few others to achieve the desired behavior.
5. Cohesion - methods manipulate more variables, the more cohesive. High cohesion means the methods and variables of the class are co-dependent and hang together as a logical whole. To maintain cohesion, we extract one small part of the function into a separate function, promote the variables used in this function to instance variables of the class, make the function we extracted and the variables into a class in their own, then we have a cohesive class. This makes organization better and structure more transparent.
6. Open-closed principle, classes should be open for extension but closed for modification. Allow new functionality via subclassing.

```
public class Sql {
  public Sql(String table, Column[] columns)
  public String create();
  public String select(Column[] columns)
}
```

should be

```
abstract public class Sql {
  public Sql(String table, Column[] columns)
  abstract public String generate();
}

public class CreateSql extend Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate();
}

public class SelectSql extend Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate();
}
```

Chapter 11 Systems

1. Avoid containing hard-coded dependency in startup process.

2. Use the abstract factory pattern to give the application control of when to build an object, but keep the details of the construction seperate from the application code.
3. Dependency Injection - seperate construction from use, an object should not take responsibility for instantiating dependencies itself.
4. Test drive the system architecture.
5. Choose the architecture fit the application.
6. Use good DLS, which will lower the risk of incorrectly translate the domain into the implementation.
7. Aspect or aspect-like mechanism, such as Java AOP framework.

Chapter 12 Emergence

1. Four rules of simple design: runs all the tests, contains no duplication, expressing the intent of the programmer, minimizes the number of classes and methods.
2. Goal is to keep keep class and function count low, as well as have tests, eliminate duplication, and express ourselves.

Chapter 13 Concurrency

Defending your system from concurrent code:

To avoid two threads modifying the same field of a shared object and cause problems:

1. Limit the scope of data
2. Use copies of objects
3. One thread share no data with any other thread. Each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables.
4. Know your library. * use the provided thread-safe collection * use the executor framework for executing unrelated tasks * use nonblocking solutions

This chapter just has some overview of concurrency, not too many contents.