Source Code Quality

Clean Code Heuristics and Guidelines

16 - 18 November 2021 | Guido Trensch (JSC, Simulation & Data Lab Neuroscience)

JÜLICH Forschungszentrum

Content

Content

“When was the last time you spent a pleasant evening in a comfortable chair, reading a good program?”

Jon Bentley, Communications of the ACM, 1986

“It is really hard to write good code … !”

Linus Torvalds

Introduction

Attributes of good Software

- Appropriate

  Software must be appropriate for the type of users.

- Reliability, security, safety

  Software should not cause physical or economic damage.

- Efficiency

  Software should not make wasteful use of system resources. Efficiency includes responsiveness, resource utilization, etc.

- Maintainability

  Software should be written in such a way that it can evolve to meet the changing needs of users.

[Ian Sommerville, *"Software Engineering"*]

Introduction

Software Quality vs Source Code Quality

- There is a subtle distinction between Software and Code

Software

- The end-user’s view
- Is the end product

Source Code

- The developer’s view
- Is the representation of the formal plan, expression of the design

- This distinction creates different perspectives on quality

SOURCE CODE QUALITY AFFECTS SOFTWARE QUALITY

Introduction

Quality Measures

Software Quality	Source Code Quality

• Test metrics
  • Code coverage
  • Test coverage
  • Unit test density
  • Defect density
  • …
	• General code quality metrics
	• Object oriented metrics
	• Complexity metrics

Introduction

Source Code Quality Metrics

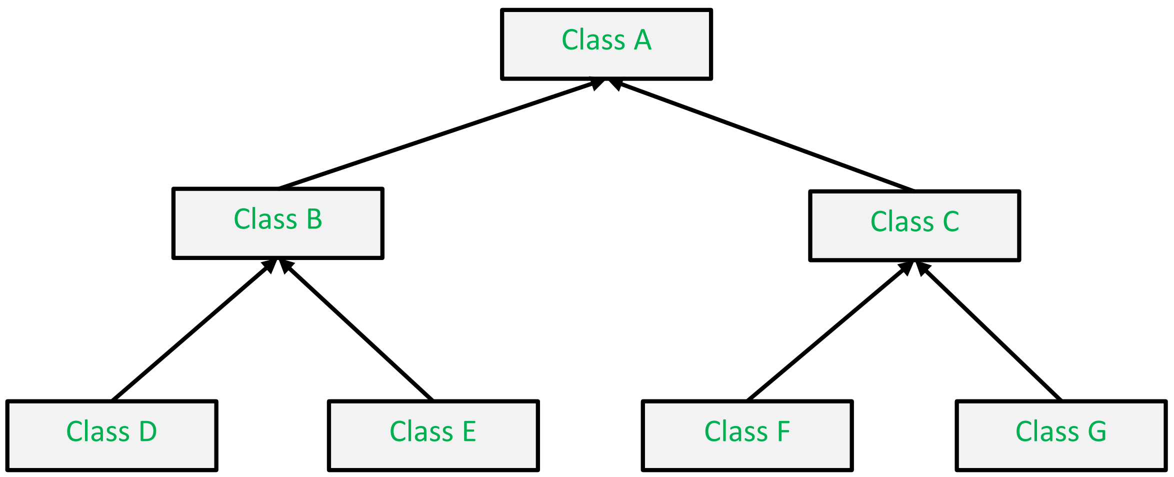
- Object oriented metrics
- Complexity metrics
- General code quality metrics

Introduction

Source Code Quality Metrics

- Object oriented metrics
- Complexity metrics
- General code quality metrics

- Depth of inheritance tree (DIT)
- Coupling between object classes (CBO)
- Number of children (NOC)

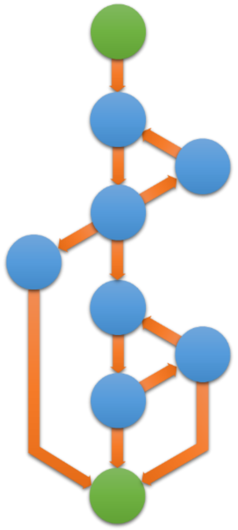


Introduction

Source Code Quality Metrics

- Object oriented metrics
- Complexity metrics
- General code quality metrics

- Afferent coupling
- Efferent coupling
- Cyclomatic complexity



Introduction

Source Code Quality Metrics

- Object oriented metrics
- Complexity metrics
- General code quality metrics

- Number of line defects (e.g., defects per 1000 lines)
  - Disabled code or "dead code“
  - Routine too long
  - ToDo annotations
  - Magic numbers
  - Nesting too deep
  - Duplicate code
  - Parameter not checked
  - ...
- Readability

Introduction

Thoughts on source code quality

- No consensus exists about the rules that define *“good source code”*.
- Therefore, it is a challenge to measure and judge the quality of source code.
- Source code quality measures are usually only an indicator for code smells.
- Developing software in teams and collaboration in projects add another dimension of complexity which also impacts quality.
- Source code quality is affected not only by the developers programming skills.
- **Improving the readability** of source code is a practical approach to improving source code quality.

... and there is another good reason to consider readability.

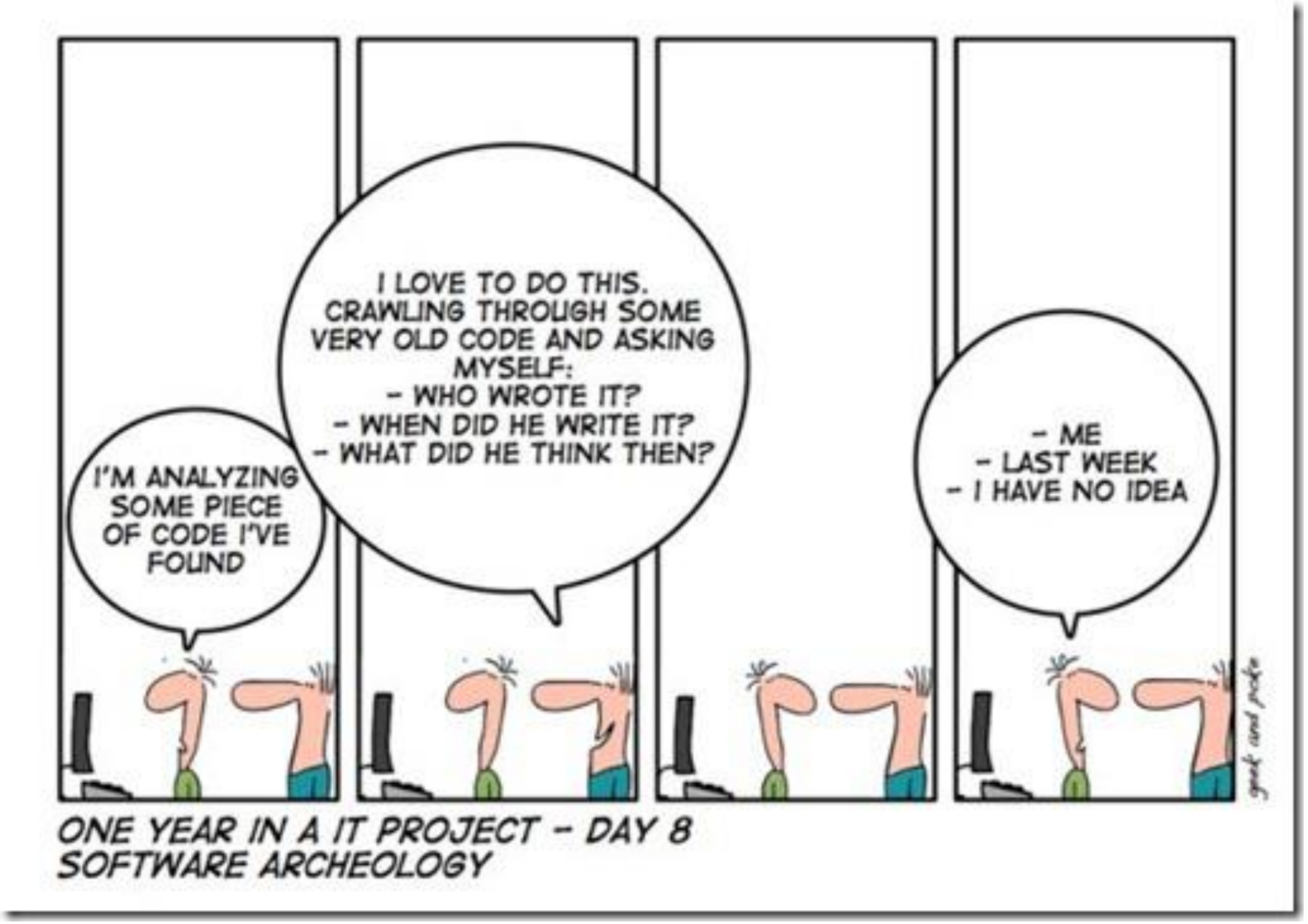
The ratio of time spent reading vs writing is well over

10:1 !

Because this ratio is so high, we want the reading of code to be easy !

Making it easy to read makes it easy to write !

Content



I'M ANALYZING SOME PIECE OF CODE I'VE FOUND
I LOVE TO DO THIS. CRAWLING THROUGH SOME VERY OLD CODE AND ASKING MYSELF:
- WHO WROTE IT?
- WHEN DID HE WRITE IT?
- WHAT DID HE THINK THEN?
- ME
- LAST WEEK
- I HAVE NO IDEA
ONE YEAR IN A IT PROJECT – DAY 8
SOFTWARE ARCHEOLOGY
geek and poke

Content

Clean Code Guidelines

Clean Code

- The objective of *"clean code"* is to bring coding guidelines to the lowest common denominator, regardless programming languages, platforms or technology.
- We will follow Robert C. Martin's heuristics from:

  Robert C. Martin, "Clean Code: A Handbook of Agile Software Craftmanship"

- Clean coding guidelines are **not dogmatic** rules.
- They imply a value system !

Clean Code Guidelines

Variable Names

Don’t be too quick to choose a name!

- What does this function do?

```
int func( int a ){
  int b = 0; int c;
  while( c = a & -a ){
    ++b; a &= ~c;
  }
  return( b );
}
```

[found in Circuit Cellar issue 250]

```
for( int the_element_idx = 0; the_element_idx < 10; ++the_element_idx ) {
   the_element_array[the_element_idx] = the_element_idx * 3.14159265359;
}
```

Clean Code Guidelines

Variable Names

Don't be too quick to choose a name!

```
int func( int a ){
  int b = 0; int c;
  while( c = a & -a ){
    ++b; a &= ~c;
  }
  return( b );
}
```

[found in Circuit Cellar issue 250]

- What does this function do?
- No descriptive names

```
for( int the_element_idx = 0; the_element_idx < 10; ++the_element_idx ) {
   the_element_array[the_element_idx] = the_element_idx * 3.14159265359;
}
```

Clean Code Guidelines

Variable Names

Don't be too quick to choose a name!

```
int func( int a ){
  int b = 0; int c;
  while( c = a & -a ){
    ++b; a &= ~c;
  }
  return( b );
}
```

[found in Circuit Cellar issue 250]

- What does this function do?
- No descriptive names
- Name too long

```
for( int the_element_idx = 0; the_element_idx < 10; ++the_element_idx ) {
   the_element_array[the_element_idx] = the_element_idx * 3.14159265359;
}
```

Clean Code Guidelines

Variable Names

Don't be too quick to choose a name!

```
int func( int a ){
  int b = 0; int c;
  while( c = a & -a ){
    ++b; a &= ~c;
  }
  return( b );
}
```

[found in Circuit Cellar issue 250]

- What does this function do?
- No descriptive names
- Names too long
- Raw number

```
for( int the_element_idx = 0; the_element_idx < 10; ++the_element_idx ) {
   the_element_array[the_element_idx] = the_element_idx * 3.14159265359;
}
```

Clean Code Guidelines

Variable Names

Don’t be too quick to choose a name!

```
for( int the_element_idx = 0; the_element_idx < 10; ++the_element_idx ) {
   the_element_array[the_element_idx] = the_element_idx * 3.14159265359;
}
```

```
for( int i = 0; i < 10; ++i ) {
   array[i] = i * MATH_PI;
}
```

Clean Code Guidelines

Variable Names

- Choose descriptive, explanatory and unambiguous names

  Make sure the name is descriptive and remember that meanings tend to drift as software evolves.
  Bad names, e.g.: execute(), handle

- Use long names for long scopes

  Variables and functions with short names lose their meaning over long distances.

- Choose names at the appropriate level of abstraction

  Choose names that reflect the level of abstraction of the class or function you are working in.

- Use standard nomenclature where possible

  Names are easier to understand if they are based on a convention.

- Avoid encodings with type or scope information

  e.g.: globalDict_PhoneBook, strName, iValue
  Today's environments provide all that information.

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names
- Typo, programming style ?

`if('+'= o)` compiler will complain

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names
- Typo, programming style ?

  `if('+'= o)` compiler will complain
- Mixed style

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names
- Typo, programming style ?

  `if('+'= o)` compiler will complain

- Mixed style
- Flag argument

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names
- Typo, programming style ?

  `if('+'= o)` compiler will complain
- Mixed style
- Flag argument
- Correctness

  try calling with, e.g.: `o = 'x'`

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names
- Typo, programming style ?

  `if('+'= o)` compiler would complain
- Mixed style
- Flag argument
- Correctness

  try calling with, e.g.: `o = 'x'`
- Raw number

Clean Code Guidelines

Find the 7 issues!

```
int func( float x, float y, float* z, char o  ) {
  if( o = '+' ) { *z = x + y; }
  else if( o == '-' ) { *z = x - y; }
  else if( o == '*' ) { *z = x * y; }
  else if( o == '/' ) { *z = x / y; }
  else {
    switch( o ) {
      case 'f': *z = faculty(x); break;
      case 's': *z = sqr(x); break;
      default: return(-1);
    }
  }
  return(0);
}
```

- No descriptive names
- Typo, programming style ?

  `if('+'= o)` compiler would complain

- Mixed style
- Flag argument
- Correctness

  try calling with, e.g.: `o = 'x'`

- Raw number
- Counterintuitive output argument

Clean Code Guidelines

```
enum operation_t { ADD, SUB, MULT, DIV, FACULTY, SQUARE };

float add( float operand_a, float operand_b ) { return(operand_a + operand_b); }
float sub( float operand_a, float operand_b ) { return(operand_a - operand_b); }
...
float square( float operand_a ) { return(operand_a * operand_a); }

float calculate( float operand_a, float operand_b, operation_t operation ) {
    switch( operation ) {
        case ADD:     return( add(operand_a, operand_b) );
        case SUB:     return( sub(operand_a, operand_b) );
        case MULT:    return( mult(operand_a, operand_b) );
        case DIV:     return( div(operand_a, operand_b) );
        case FACULTY: return( faculty(operand_a) );
        case SQUARE:  return( square(operand_a) );
    }
}

float c = calculate( a, b, ADD );
```

Functions, Names and Arguments

- Function names should express what the function does

  If you have to look at the implementation or documentation of the function to know what it does, then you should work to find a better name.

- One function should do one thing
- Avoid flag arguments
- Keep functions short

  All lines of the function should fit on your screen.

- Avoid too many arguments

  (No argument is best.)

- Avoid output arguments in the argument list when possible

  Output arguments are counterintuitive. (Much of the need for output arguments disappears in object-oriented languages.)

Clean Code Guidelines

Programming Style

Follow design principles !

- **KISS** – Keep it short and simple / stupid
- **DRY** – Don‘t repeat yourself
- **SoC** – Separation of concern
- **RAII** – Resource acquisition is initialization
- **S.O.L.I.D.**
  - *S – Single responsibility principle (SRP)*
  - *O – Open close principle (OCP)*
  - *L – Liskov substitution principle (LSP)*
  - *I – Interface segregation principle (ISP)*
  - *D – Dependency inversion principle (DIP)*

Clean Code Guidelines

Programming Style

- Avoid duplication when possible

  Duplication in the code may represent a missed opportunity for abstraction.

- Delete dead and unused code

  Don't be afraid, the VCS will remember!

- Expected behavior should be implemented

  Any function or class should implement the behaviors that another programmer could reasonably expect.

- Avoid too much information

  e.g.:

  - Avoid to create classes with lots of methods/member functions.

  - Concentrate on keeping interfaces very tight and small.

Clean Code Guidelines

Programming Style

- Avoid inconsistency in the implementation style

  If you do something a certain way, do all similar things in the same way.

- Do not use raw numbers in the code

  Replace numbers with named constants.

```
constexpr double MATH_PI = 3.141592653589793;
double a = MATH_PI;

      is preferable to

double a = 3.141592653589793;
```

Clean Code Guidelines

Programming Style

- Encapsulate conditionals

```
if( shouldBeDeleted(timer) ) { ... }
```

is preferable to

```
if( timer.hasExpired() && !timer.isRecurrent() ) { ... }
```

- Avoid negative conditionals

```
if( expressionIsTrue() ) { ... }
```

is preferable to

```
if( not expressionIsNotTrue() ) { ... }
```

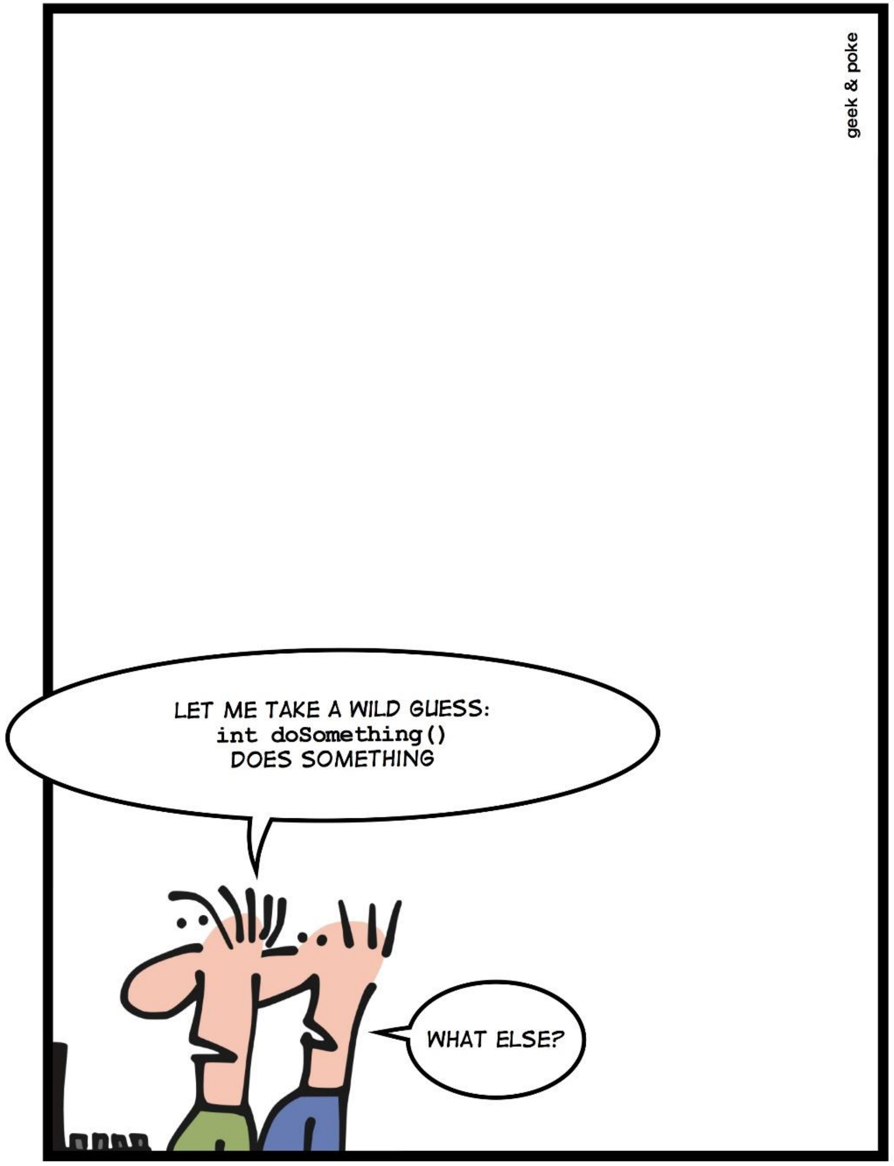
Clean Code Guidelines

Comments

Nothing can be such helpful as a well-placed comment.

Keep in mind, that the only truly good comment is the comment you found a way not to write.

SIMPLY EXPLAINED



SELF DOCUMENTING CODE

Clean Code Guidelines

Comments

- Avoid poorly written comments
- Avoid inappropriate Information

  e.g.: The change history of your code or any other meta-data.
- Avoid redundant comments

  msg(“WARNING: Too much comments!”)   // print warning message
- Remove obsolete comments or update them as quickly as possible
- Delete commented-out code

  Don’t be afraid, the VCS will remember.

Clean Code Guidelines

Artificial, Logical, Physical and Temporal Coupling

- Avoid artificial coupling

  Artificial coupling is a coupling between modules that serve no direct purpose.
  It is a result of putting a variable, constant or function in a inappropriate location.

- Make logical dependencies physical

  If one module depends upon another, that dependency should be physical, not just logical.
  The dependent module should not make assumptions about the module it depends on.

- Do not hide temporal coupling

  Temporal couplings are often necessary, but do not hide them.
  Structure the arguments of your functions such that the order in which they should be called is obvious.

Clean Code Guidelines

Source Code Structure

One of the most important decisions a software developer makes is where to put the code.

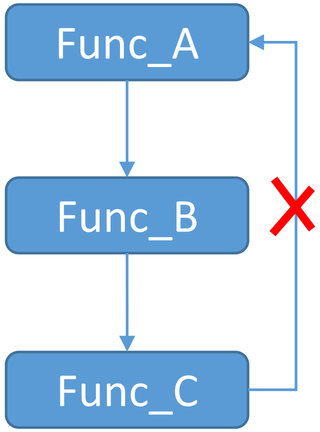
- Don‘t be arbitrary and avoid misplaced code

  Have a reason for the way you structure your code.

- Code at the right level of abstraction

  It is important to create abstractions that separate higher level general concepts from lower level detailed concepts.

- Functions should descent only one level of abstraction



Clean Code Guidelines

Source Code Structure

- Use vertical separation

  Local variables should be declared just above their first usage and should have a small vertical scope.

- Keep configurable data at high levels

  For example: If you have a configuration value, do not bury it in a low-level function. Expose it as an argument to that low level function called from the high-level function.

Clean Code Guidelines

Source Correctness

Lots of funny code is written because people don't take time to understand the source code.

- Understand the algorithm
- Ensure correct behavior at the boundaries

  Look for every boundary condition and write a test for it.
- Be precise and avoid ambiguities

  Ambiguities and imprecision in code are either a result of disagreements or laziness.

Clean Code Guidelines

General Considerations and Best Practices

- Use a style guide for your project

  It doesn't matter where you put your braces so long as all in the project agree on where to put them.

  - Google Style Guides: https://google.github.io/styleguide/
  - C++ https://google.github.io/styleguide/cppguide.html
  - Python https://github.com/google/styleguide/blob/gh-pages/pyguide.md
  - Java https://google.github.io/styleguide/javaguide.html
  - PEP 8 style guide for Python: https://www.python.org/dev/peps/pep-0008/

Clean Code Guidelines

General Considerations and Best Practices

- Use source code formatting and static code analysis tools
  - C, C++
    - **clang** static analyzer and **clang-format**.
    - **cppcheck** open-source tool for static analysis of C/C++ code.
    - **vera++** tool for verification, analysis and transformation of C++ source code.
  - Python
    - **PEP 8** formatter and checker.
    - **Pylint** error checker and looks for code smells: https://www.pylint.org/

Clean Code Guidelines

General Considerations and Best Practices

- Don‘t override safeties

  Turning off certain compiler warnings (or all warnings!) may help you to get the build to succeed, but at the risk of endless debugging sessions.

Clean Code Guidelines

General Considerations and Best Practices

Testing

- Implement sufficient tests

  A test suite should test everything that could possibly break.

- Don‘t skip trivial tests

  An ignored test is a question about ambiguity.

- Test boundary conditions

  Take special care to test boundary conditions.

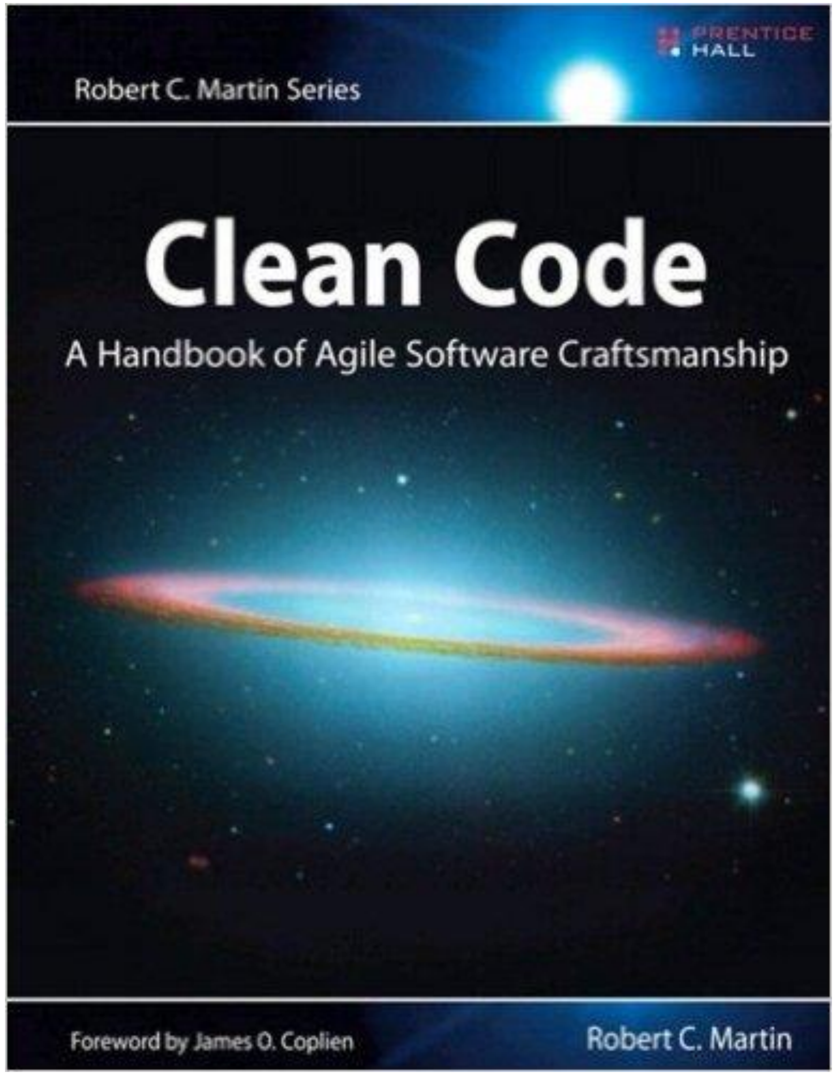
- Exhaustively test functions where bugs have occurred

  Bugs tend to congregate.

Content

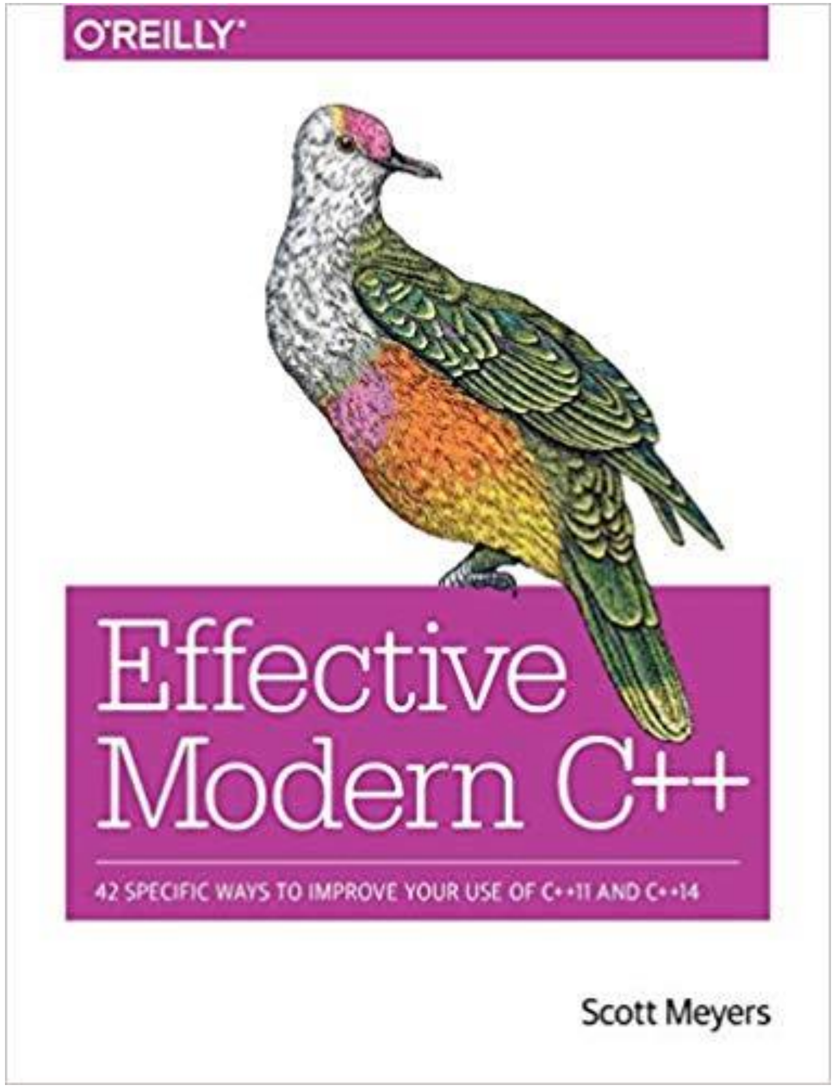
References

Every software developer should have read Robert C. Martin's book *"Clean Code: A Handbook of Agile Software Craftmanship"*, the standard reference for writing good code!



References

42 Specific Ways to Improve Your Use of C++11 and C++14.



References

- Google Style Guides: https://google.github.io/styleguide/
  - C++ https://google.github.io/styleguide/cppguide.html
  - Python https://github.com/google/styleguide/blob/gh-pages/pyguide.md
  - Java https://google.github.io/styleguide/javaguide.html

- PEP 8 style guide for Python: https://www.python.org/dev/peps/pep-0008/

"The next time you write a line of code, remember you are an author, writing for readers who will judge your effort."

Robert C. Martin