Clean code

Good practices in general coding

Alessandro Sciarra

Z02 – Software Development Center

06 July 2023

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

AxelKrypton

PUNCH 4NFDI

CRC-TR 211 Strong-interaction matter under extreme conditions

Prelude

Leave me a feedback

Was the talk clear, useful, inspiring?
Would you like to hear more on this/other topic? Which?
Any general comment? ✉ Alessandro

Disclaimer

1. Slides are quite full of text for later reading.
2. It is about common sense, exceptions might exist.
3. Let us discuss!

Outline

Summary & Conclusion

Take home message

Clean code

What it is and why it is worth aiming at it

- The importance of meaningful names, comments and formatting
- Strive to respect few very general rules (e.g. DRY, KISS, pathfinder)
- Functions and classes: small, Single Responsibility and IOSP!
- Test your code and keep your tests clean (F.I.R.S.T.)!

Have a reason for everything you type ⟶ refactor, refactor, refactor!

The first reader of your code is yourself

Try to minimise the time needed by a new reader to understand!

Tools of developer daily life

Your toolkit should contain more than you might think!

Not just a compiler and a mentor (e.g. Google, Stack Exchange, textbook)

Tools of developer daily life

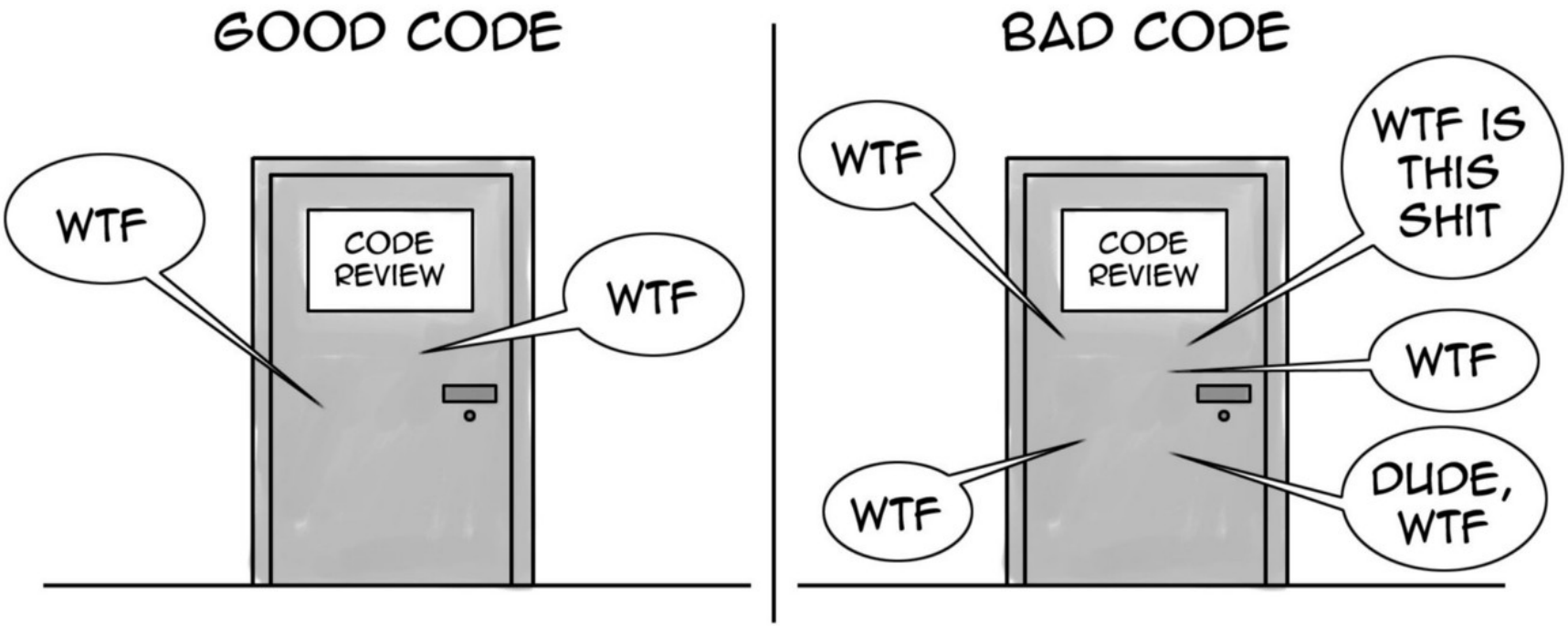
Pick one tool from each category and know how to use it

Your toolkit should contain more than you might think!

Not just a compiler and a mentor (e.g. Google, Stack Exchange, textbook)

- Integrated Development Environment (e.g. Visual Studio Code, Eclipse)
- Tests support – Build automation (e.g. CMake)
- Version control system (e.g. Git) → GitHub, Redmine, GitLab, etc.
- Profiler
- Static analysis – Formatter (e.g. clang suite)
- Different compilers
- Debugger { Clean code does not require to be debugged! }

What are we aiming to?



THE ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE

Clean code

Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. [. . .] if you try to imagine improvements, you're led back to where you are.

Michael Feathers

I like my code to be elegant and efficient [. . .] clean code does one thing well.

Bjarne Stroustrup

Clean code is simple and direct. [. . .] Clean code reads like well-written prose.

Grady Booch

You know you are working on clean code when each routine you read turns out to be pretty much what you expected.

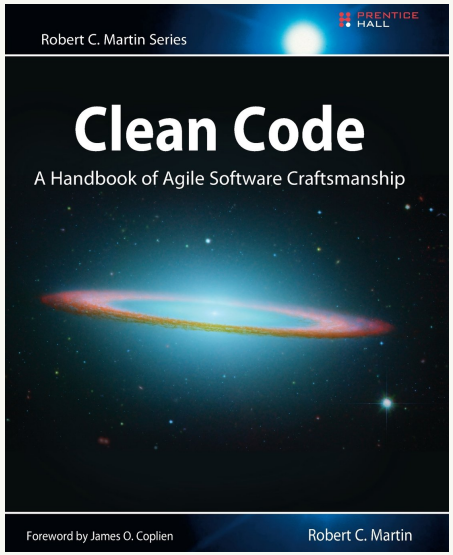
Ward Cunningham

Inventor of the SOLID principle, of the C++ language, of the Booch method and of the Wiki, respectively (top to bottom).

Clean code

> Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. [. . .] if you try to imagine improvements, you're led back to where you are.
>
> Michael Feathers

- Readable
- Maintainable
- Easy to extend
- Easy to use
- Hard to break
- Testable/Tested
- ...



The Clean Code developer road

Red → Orange → Yellow → Green → Blue → White

We will develop the needed code sense!

> A gut feeling is like a drill, a simple instrument whose force lies in the quality of its material.
>
> — Gerd Gigerenzer —

Why do we need to care?

Why do we need to care?

- We never write code for one time-invariant reader!
- Learning and applying few simple rules increases quality of code base.
- When a program is modified, its complexity will increase, provided that one does not actively work against this.

Why do we need to care?

The technical debt

It is a concept that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

Wikipedia

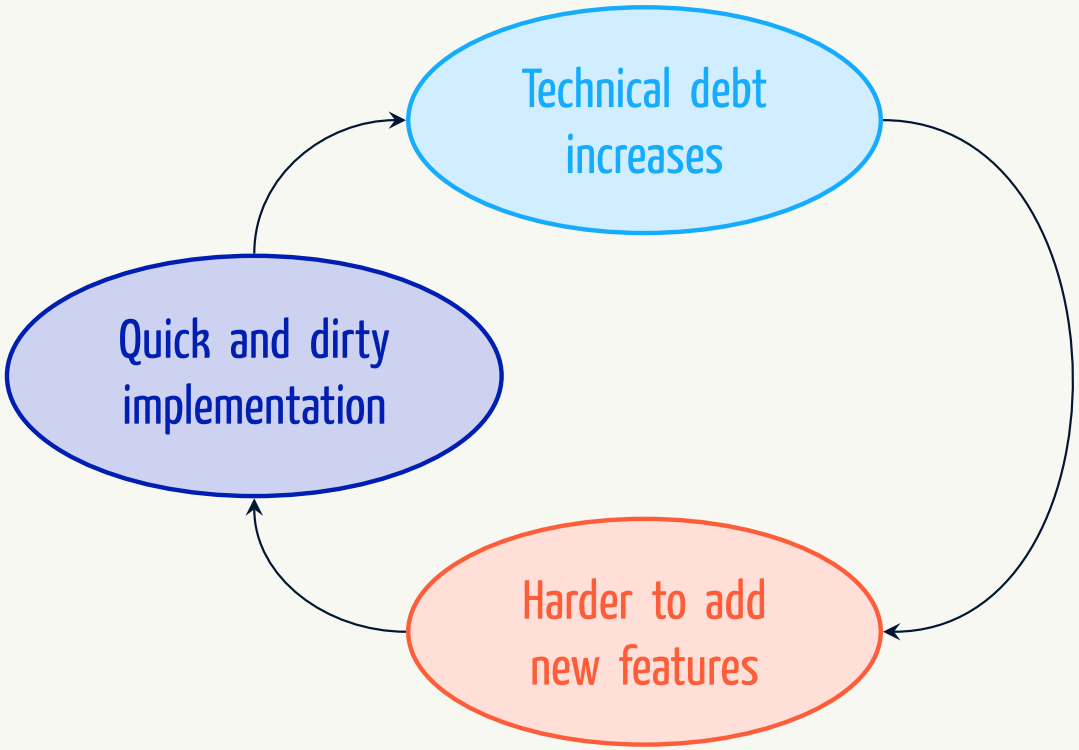
- We never write code for one time-invariant reader!
- Learning and applying few simple rules increases quality of code base.
- When a program is modified, its complexity will increase, provided that one does not actively work against this.

Why do we need to care?

The technical debt

It is a concept that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

Wikipedia

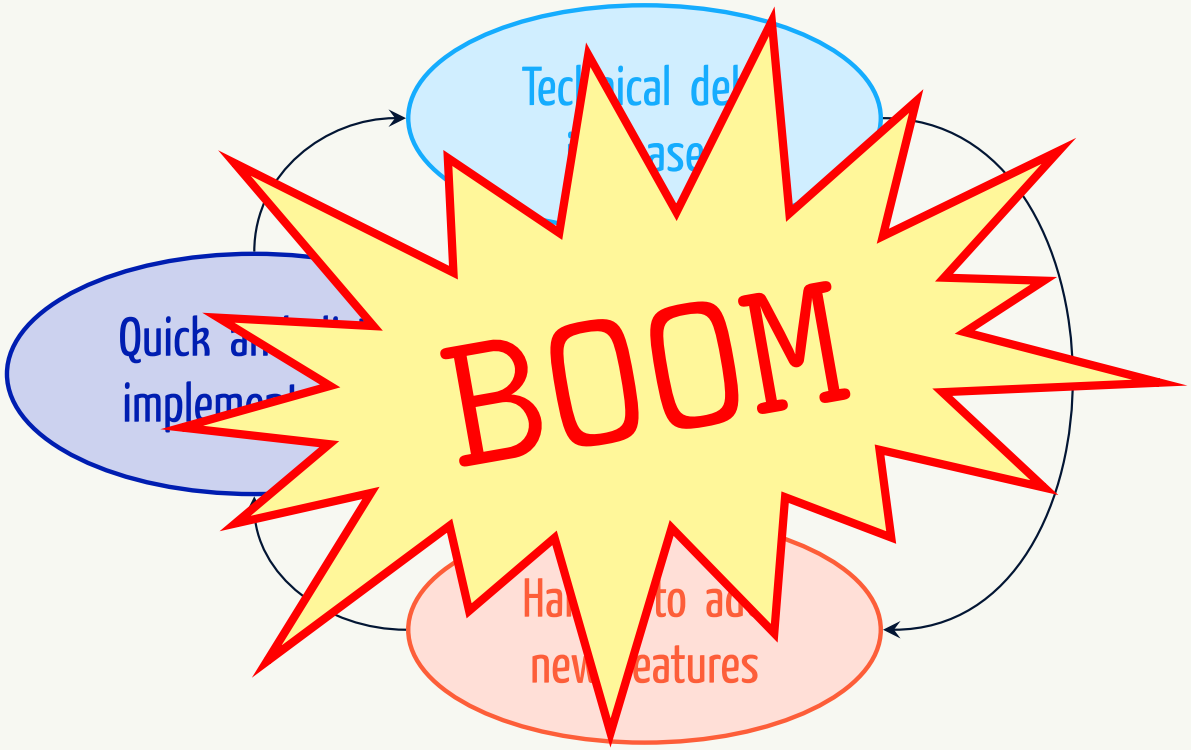


Why do we need to care?

The technical debt

It is a concept that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

Wikipedia

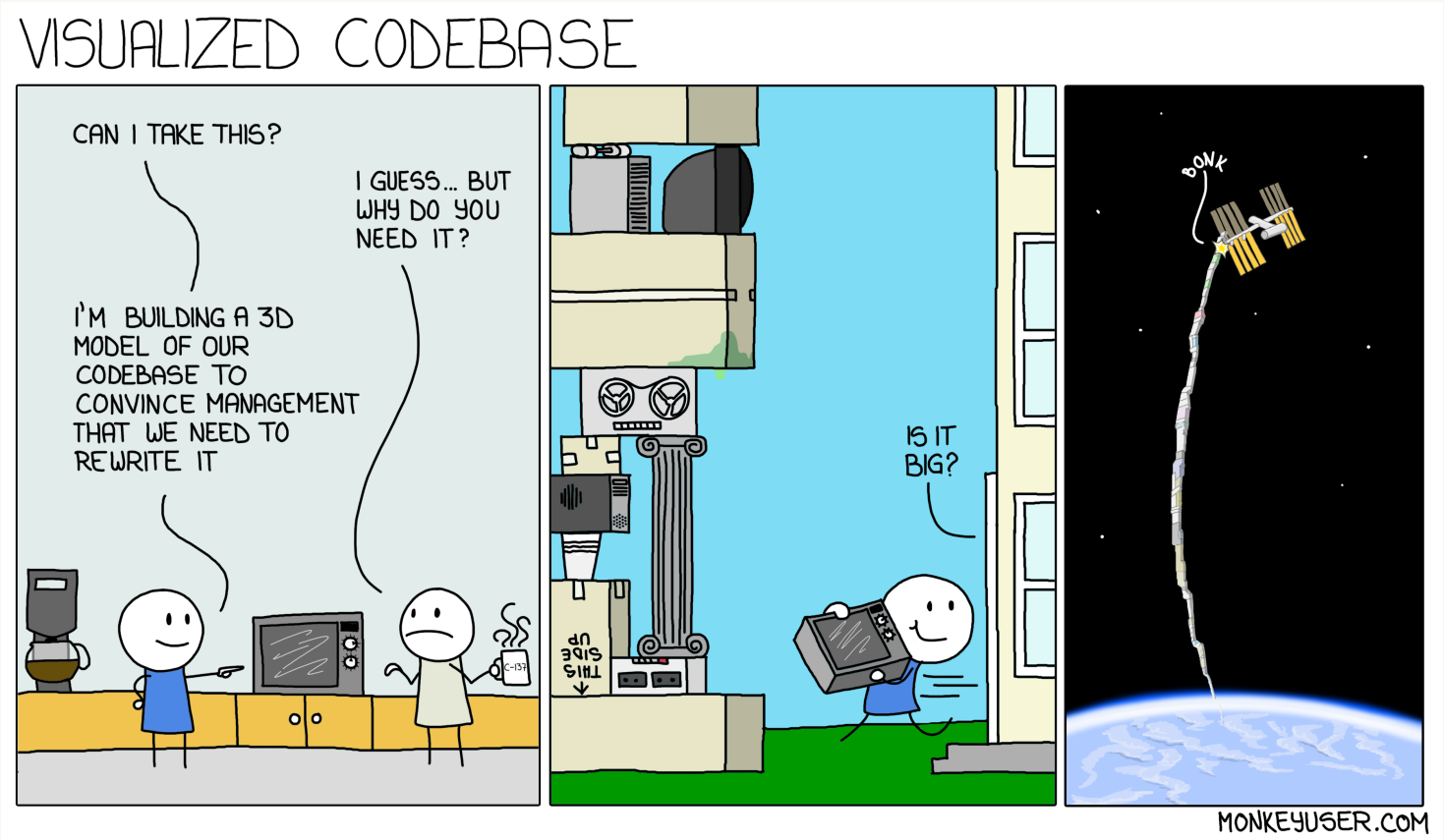


Why do we need to care?

The technical debt

It is a concept that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

Wikipedia

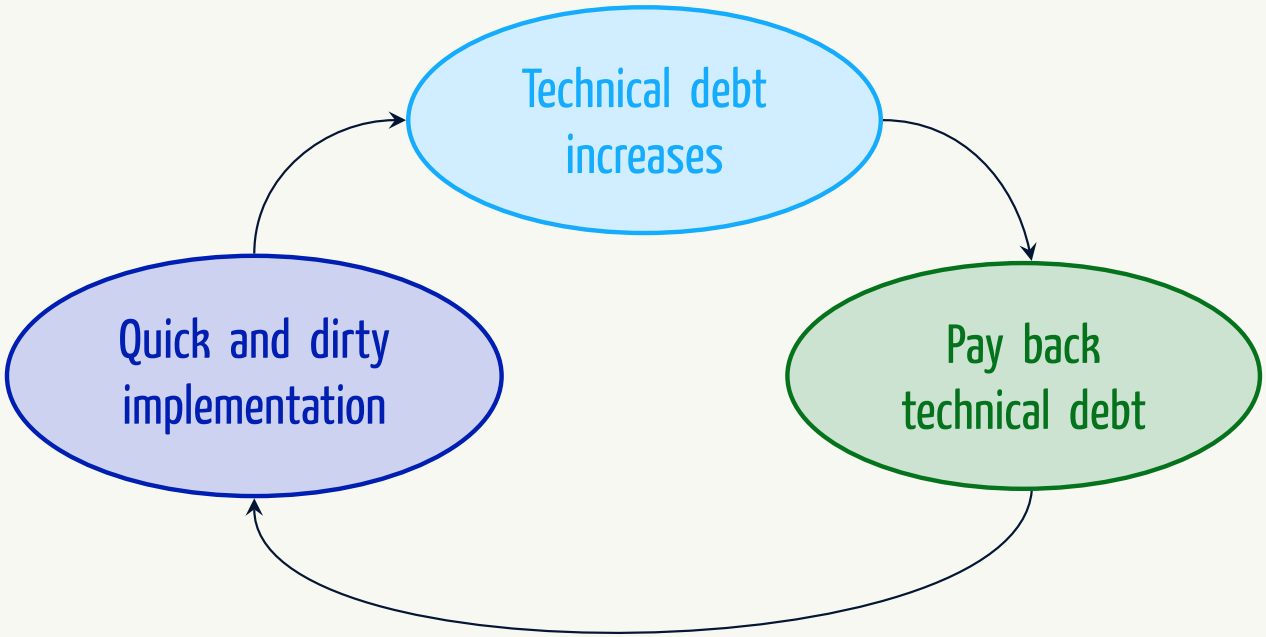


Why do we need to care?

The technical debt

It is a concept that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

Wikipedia



Technical debt naturally tends to increase

As physicist, why having high quality code?

- It is plenty of problems which need to be approached numerically
  → Programming is one of the major tasks in physics!
- Reproducibility should be a pillar of science
  → One of the main responsibilities of a scientist!
- Lots of bad code around and passed on across generations
  → A huge amount of time is wasted, subtracted to research!

It should not happen...

«Wow! There is definitely a new physics signal in these data!»

...few days/weeks/months/years later...

«Oh, no! It was just a bug in the code!»

The starting point...

The starting point...

Meaningful names

Use intention-revealing names

Why is it hard to tell what this code is doing?

```
using std::vector<std::vector<int>> = myVecVec;

myVecVec FC(const myVecVec& b)
{
    myVecVec list;
    for(const std::vector<int>& c : b){
        if(c[0] == 4)
            list.push_back(c);
    }
    return list;
}
```

Use intention-revealing names

Why is it hard to tell what this code is doing?

```
using std::vector<std::vector<int>> = myVecVec;

myVecVec FC(const myVecVec& b)
{
    myVecVec list;
    for(const std::vector<int>& c : b){
        if(c[0] == 4)
            list.push_back(c);
    }
    return list;
}
```

- What kind of object is contained in `list`?
- What does `0` represent in the `if`-clause?
- What does `4` mean in the `if`-clause?
- How would I use the object being returned?

Use intention-revealing names

Good names are crucial for readability!

```
using std::vector<std::vector<int>> = myVecVec;

myVecVec getFlaggedCells(const myVecVec& gameBoard)
{
    myVecVec flaggedCells;
    for(const std::vector<int>& cell : gameBoard){
        if(cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.push_back(cell);
    }
    return flaggedCells;
}
```

- What kind of object is contained in `flaggedCells`?
- What does `STATUS_VALUE` represent in the `if`-clause?
- What does `FLAGGED` mean in the `if`-clause?
- How would I use the object being returned?

Use intention-revealing names

Good names are crucial for readability!

```
using std::vector<std::vector<int>> = SetOfCells;

SetOfCells getFlaggedCells(const SetOfCells& gameBoard)
{
    SetOfCells flaggedCells;
    for(const Cell& cell : gameBoard){
        if(cell.isFlagged())
            flaggedCells.push_back(cell);
    }
    return flaggedCells;
}
```

- What kind of object is contained in `flaggedCells`?
- What does `STATUS_VALUE` represent in the `if`-clause?
- What does `FLAGGED` mean in the `if`-clause?
- How would I use the object being returned?

Use names which can be searched

Rule of thumb

The length of a name should be chosen w.r.t. to the size of its scope

Variables: Proportionally to their scope size

Functions: In inverse proportion to their scope (cf. C++ STL)

- Single-letter names **only** for local variables inside **short scopes**
- Single-word names **a must** for functions in widely used **libraries**
- IDE very good at renaming variables $\Rightarrow$ rename, rename, rename!

Use names which can be searched

Rule of thumb

The length of a name should be chosen w.r.t. to the size of its scope

Variables: Proportionally to their scope size

Functions: In inverse proportion to their scope (cf. C++ STL)

Something like this...

```
for(int j = 0; j < 34; j++)
    s += t[j]*4/5;
```

Use names which can be searched

Rule of thumb

The length of a name should be chosen w.r.t. to the size of its scope

Variables: Proportionally to their scope size

Functions: In inverse proportion to their scope (cf. C++ STL)

...or something like this?

```
const int realDaysPerIdealWeek = 4;
int totalRealWeeksNeeded = 0;
for(int j = 0; j < NUMBER_OF_TASKS; j++)
{
    int realDaysPerTask   = taskDaysEstimate[j] *
                            realDaysPerIdealWeek;
    int realWeeksPerTask  = realDaysPerTask/WORKING_DAYS_PER_WEEK;
    totalRealWeeksNeeded += realWeeksPerTask;
}
```

Other important principles

- Avoid disinformation
- Use pronounceable names → `timestamp` better than `ymdhms`
- Avoid mental mapping
- Pick one word per concept → `get`, `fetch`, `retrieve`, `obtain`
- Do not be cute
- ...

Take home lesson

Choose carefully every name!

↓

Rename when you find better ones!

Comments and Formatting

Do not comment bad code, rewrite it

Comments are an **unfortunate necessity**,
NOT a great achievement!

— Robert C. Martin —

Do not comment bad code, rewrite it

Avoid comments!

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. Comments are always failures. [...] Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often. The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong.

R. C. Martin

- To keep comments up-to-date costs energy and this energy should rather go into make the code clearer and more expressive!
- Inaccurate comments are far worse than no comments!
- Truth can only be found in one place: the code.

Explain yourself in the code

What you should strive to avoid...

```
Person e; // The employee we are considering

// Check if the employee is eligible for full benefits
if( (e.flags & HOURLY_FLAG) && e.age > 65 )
```

Explain yourself in the code

What you should strive to avoid...

```
Person e; // The employee we are considering

// Check if the employee is eligible for full benefits
if( (e.flags & HOURLY_FLAG) && e.age > 65 )
```

...in favour of expressive code

```
Person employee;

if(employee.isEligibleForFullBenefits())
```

Explain yourself in the code

What you should strive to avoid...

```
Person e; // The employee we are considering

// Check if the employee is eligible for full benefits
if( (e.flags & HOURLY_FLAG) && e.age > 65 )
```

...in favour of expressive code

```
Person employee;

if(employee.isEligibleForFullBenefits())
```

- It is simple to come up with good ideas
- It is often simply matter of creating a function whose name expresses the content of the comment!

Good comments

But try to limit them as much as possible!

Informative comment

```
// Format matched:      hh:mm:ss dd Month yyyy
std::regex timePattern("(\\d{2}:){2}\\d{2}, \\d{2} \\S+ \\d{4}");
```

Good comments

But try to limit them as much as possible!

Informative comment

```
// Format matched:      hh:mm:ss dd Month yyyy
std::regex timePattern("(\\d{2}:){2}\\d{2}, \\d{2} \\S+ \\d{4}");
```

Explanation of intent

```
// Checking the residuum possibly not every
// iteration allows for speed up on GPUs
if(iterationNumber % CG_CHECK_RESIDUUM_EVERY == 0)
```

Good comments

But try to limit them as much as possible!

Informative comment

```
// Format matched:      hh:mm:ss dd Month yyyy
std::regex timePattern("(\\d{2}:){2}\\d{2}, \\d{2} \\S+ \\d{4}");
```

Explanation of intent

```
// Checking the residuum possibly not every
// iteration allows for speed up on GPUs
if(iterationNumber % CG_CHECK_RESIDUUM_EVERY == 0)
```

Clarification when using cryptic external API

```
// Check if root was found up to desired precision
if(gsl_fcmp(functionValue, 0.0, epsilon) == 0)
```

Good comments

But try to limit them as much as possible!

- Informative comment
- Explanation of intent, Clarification

Amplification

```
// See https://stackoverflow.com/a/1736040 for more information.
template<typename T>
std::string getDefaultForHelper(const T& number)
```

```
// Removing trailing spaces is crucial since any would break the
// following algorithm. It might leave the string unchanged, but
// the absence of trailing spaces is at the moment not guaranteed!
boost::trim_right(keyToDecipher);
```

Good comments

But try to limit them as much as possible!

- Informative comment
- Explanation of intent, Clarification
- Amplification

TODO comments

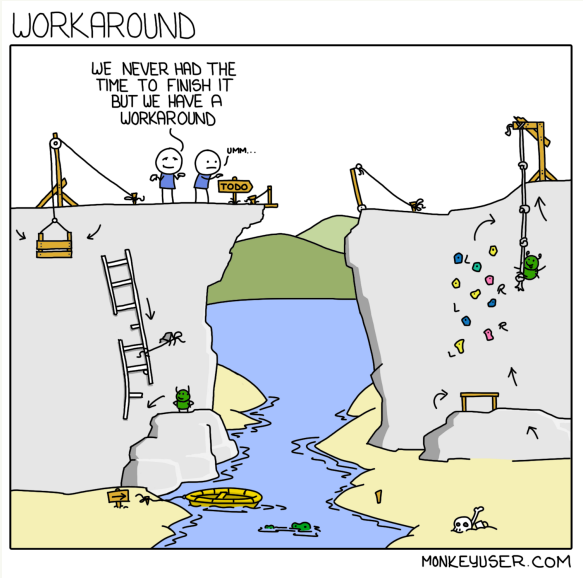
```
// TODO: The following two classes should be merged into one!
```

Legal comments

```
/*****************************************
 *  Copyright (c) 2018 Alessandro Sciarra  *
 *                                         *
 *  This file is part of BaHaMAS.          *
 *****************************************/
```

Good comments

But try to limit them as much as possible!



- Informative comment
- Explanation of intent, Clarification
- Amplification
- TODO comments → border line: really needed in a code? Do not check them in!
- Legal comments → border line: not legally required & overlap with version control.

Automatic documentation comments

If you are writing a public API, then you should certainly provide a good documentation. But be aware that documentation comments can be as misleading, misplaced and lying as any other comment.

Bad comments

Several bad habits in one example

```
/**
  * @param longTitle The complete title of the CD
  * @param longAuthor The complete author of the CD
  * @param nTracks The number of tracks of the CD
  * @param durationInMinutes The duration of the CD in minutes
  */
void Jukebox::addCD(std::string longTitle,
                    std::string longAuthor,
                    /*int nTracks,*/ int durationInMinutes)
{
    // Create new CD to be later added to the list
    CD cd;
    cd.title    = longTitle;
    cd.author   = longAuthor;
    //cd.tracks   = nTracks;
    cd.duration = durationInMinutes;
    if(cd.duration<60) // Added by James
        throw std::invalid_argument();
    cdList.push_back(cd);
} // addCD method
```

Bad comments

The same code cleaned up

```
void Jukebox::addCD(std::string longTitle,
                    std::string longAuthor,
                    int durationInMinutes)
{
    CD cd;
    cd.title    = longTitle;
    cd.author   = longAuthor;
    cd.duration = durationInMinutes;
    if(cd.duration<60)
        throw std::invalid_argument();
    cdList.push_back(cd);
}
```

The code becomes smaller and it reads well!
Why should you add a comment here?!

Bad comments

- Redundant comments → Avoid
- Noise comments → Avoid
- Misleading comments → Avoid
- Comments instead of a good name → Use good name
- Closing brace comments → Rarely
- Attribution comments → Delete, use version control
- Commented-out code → Delete, use version control
- Unclear, cryptic comments → Remove or clarify

Have **good reasons** before adding a comment!

Are they?

Formatting

Yes, formatting is important!

The 3 most important rules:

Formatting

Yes, formatting is important!

The 3 most important rules:

1. Stay coherent with what you find

Formatting

Yes, formatting is important!

The 3 most important rules:

1. Stay coherent with what you find
2. Stay coherent with what you find

Formatting

Yes, formatting is important!

The 3 most important rules:

1. Stay coherent with what you find
2. Stay coherent with what you find
3. Stay coherent with what you find

This starts with but it is not limited to formatting!

Especially important when working on an existing codebase!

Formatting

Yes, formatting is important!

- How big should be a file?
  → Try to stay below few hundreds, the reader will be grateful!
- Give some vertical breath to your code
- Care about length of lines (80 to 140 characters)
  → Crucial if the code is readable from browser (e.g. GitHub)
- Consider horizontal alignment and spacing (e.g. around operators)
- Be coherent (e.g. braces)

How to work when different people write in the same code base?

Agree on a style and use a tool (e.g. clang-format) to enforce it

Functions and Classes

Common general aspects

SMALL

Do one thing. Do it well. Do it only!

- Functions and classes interfaces should stay small (see next slide)
- Classes should fulfil the single responsibility principle
- One level of abstraction per function

Common general aspects

SMALL

Do one thing. Do it well. Do it only!

- Functions and classes interfaces should stay small (see next slide)
- Classes should fulfil the single responsibility principle
- One level of abstraction per function

Single-responsibility class

```
class Version {
  public:
    int getMajorVersionNumber();
    int getMinorVersionNumber();
    int getBuildNumber();

  // [...]
};
```

Common general aspects

SMALL

Do one thing. Do it well. Do it only!

- Functions and classes interfaces should stay small (see next slide)
- Classes should fulfil the single responsibility principle
- One level of abstraction per function

1. Let's start with an example of bad code
2. and then have a look to how it should be

Common general aspects

SMALL

Do one thing. Do it well. Do it only!

Impolite, rude code ⟶ up and down to understand it!

```
void MakeCoffee(const Coffee& typeOfCoffee){
    auto neededT = temperature(typeOfCoffee);
    if(neededT < T)
        warmUp(neededT);
    while(grinder != nullptr)
        prepareGrinder();
    grind(calculateGrams(typeOfCoffee));
    auto neededP = pressure(typeOfCoffee);
    prepareToBrew(neededT, neededP);
    if(status == READY_TO_BREW)
        brew(typeOfCoffee);
    else
        throw std::runtime_error("Unable to brew!");
}
```

Common general aspects

SMALL

Do one thing. Do it well. Do it only!

- Functions and classes interfaces should stay small (see next slide)
- Classes should fulfil the single responsibility principle
- One level of abstraction per function

Single-level-of-abstraction function

```
void MakeCoffee(const Coffee& typeOfCoffee){
    WarmUpMachineIfNeeded(typeOfCoffee);
    GrindCoffee(typeOfCoffee);
    SetPressureAndTemperature(typeOfCoffee);
    BrewCoffee(typeOfCoffee);
}
```

More about this one-thing idea

> How big should a function be?
>
> Robert C. Martin

More about this one-thing idea

Back to our example

```
void MakeCoffee(const Coffee& typeOfCoffee){
    // 1. WarmUpMachineIfNeeded
    auto neededT = temperature(typeOfCoffee);
    if(neededT < T)
        warmUp(neededT);
    // 2. GrindCoffee
    while(grinder != nullptr)
        prepareGrinder();
    grind(calculateGrams(typeOfCoffee));
    // 3. SetPressureAndTemperature, needs T again
    auto neededP = pressure(typeOfCoffee);
    prepareToBrew(neededT, neededP);
    // 4. BrewCoffee
    if(status == READY_TO_BREW)
        brew(typeOfCoffee);
    else
        throw std::runtime_error("Unable to brew!");
}
```

More about this one-thing idea

How big should a function be?

A function should do one thing. But what's one thing? A function does one thing, if you cannot meaningfully extract another function from it.

Robert C. Martin

Result after extraction

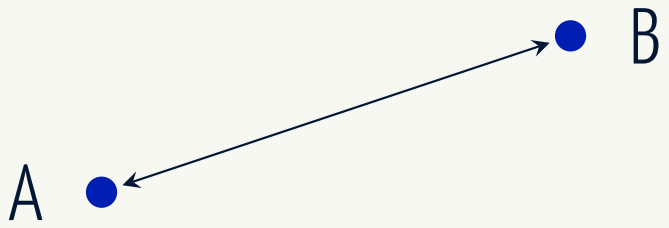
```
void MakeCoffee(const Coffee& typeOfCoffee){
    WarmUpMachineIfNeeded(typeOfCoffee);
    GrindCoffee(typeOfCoffee);
    SetPressureAndTemperature(typeOfCoffee);
    BrewCoffee(typeOfCoffee);
}
```

Limit the number of arguments

Make useful abstractions whenever it helps

When you call the function, which argument is what?

```
double Distance(double, double, double, double);
double Distance(double, double, double, double, double, double);
```

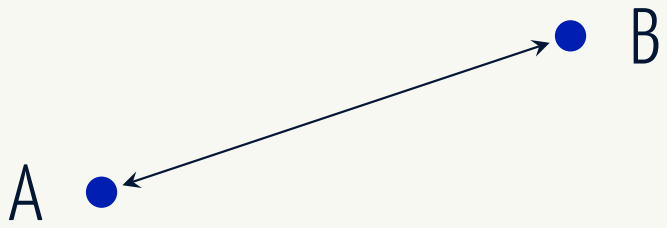


Limit the number of arguments

Make useful abstractions whenever it helps

When you call the function, which argument is what?

```
double Distance(double, double, double, double);
double Distance(double, double, double, double, double, double);
```



Isn't this much better?

```
struct Point2D {
    double x, y;
};
```

```
struct Point3D {
    double x, y, z;
};
```

```
double Distance(std::pair<Point2D, Point2D>);
double Distance(std::pair<Point3D, Point3D>);
```

Integration Operation Segregation Principle

Clear separation of:

Operation: A function which contains exclusively logic, meaning transformations, control structures or API invocations.

Integration: A function which does not contain any logic but exclusively calls other functions of the code base.

Operation

```
bool isPrime(unsigned int number)
{
    if(number == 1) return false;
    for(auto i = 2u; i <= std::sqrt(number); ++i)
    {
        if(number%i == 0) return false;
    }
    return true;
}
```

Integration Operation Segregation Principle

Clear separation of:

Operation: A function which contains exclusively logic, meaning transformations, control structures or API invocations.

Integration: A function which does not contain any logic but exclusively calls other functions of the code base.

Integration

```
int main(int argc, char *argv[])
{
    WelcomeUserToTheGame();
    Minesweeper minesweeper(argc, argv);
    minesweeper.playGame();
    PrintGameResult();
}
```

Testing your code

I have a full ~2h seminar on this topic

Software testing

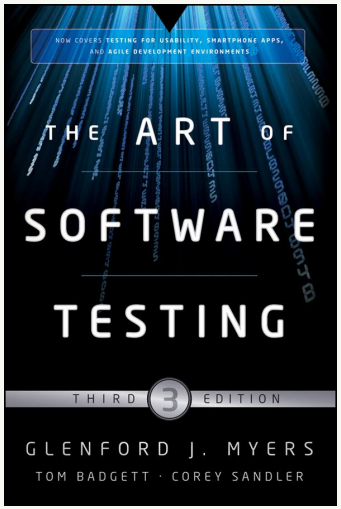
Test code is just as important as production code

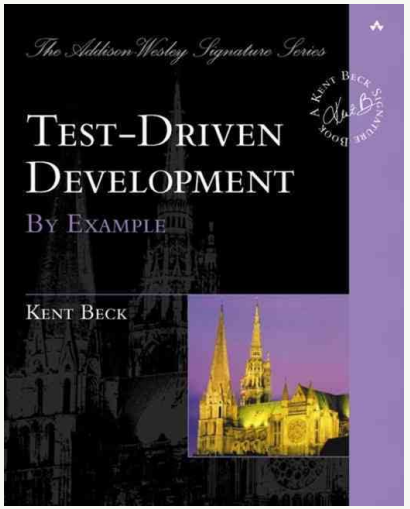
- Testing levels
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
- Testing types and techniques
  - Regression testing
  - Automated testing
  - Continuous testing
  - ...

Software testing

Test code is just as important as production code

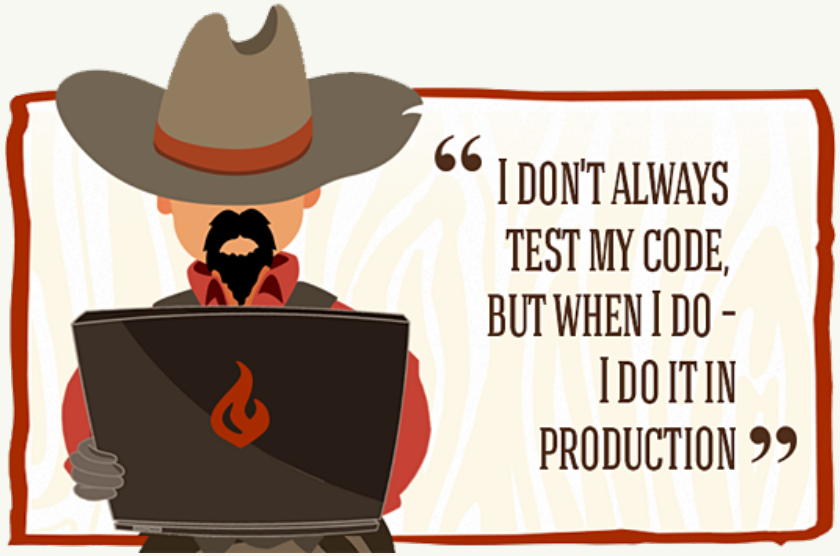
- Testing levels
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
- Testing types and techniques
  - Regression testing
  - Automated testing
  - Continuous testing
  - ...





The wrong mental approach to testing

- I do not have time to waste writing tests.
- The result is reasonable, the code should be correct!
- This part of code is simple, why should I test it?
- Let me finish to implement this feature so that I can produce some data and in the meanwhile I write some tests.



https://android.jlelse.eu/basics-of-unit-testing-affdd2273310

The wrong mental approach to testing

- I do not h[illegible] t[illegible]
- The re[illegible]
- Th[illegible]
- I d[illegible] [illegible] so that I can produce some data and in the meantime I write some tests.

Please, NO!

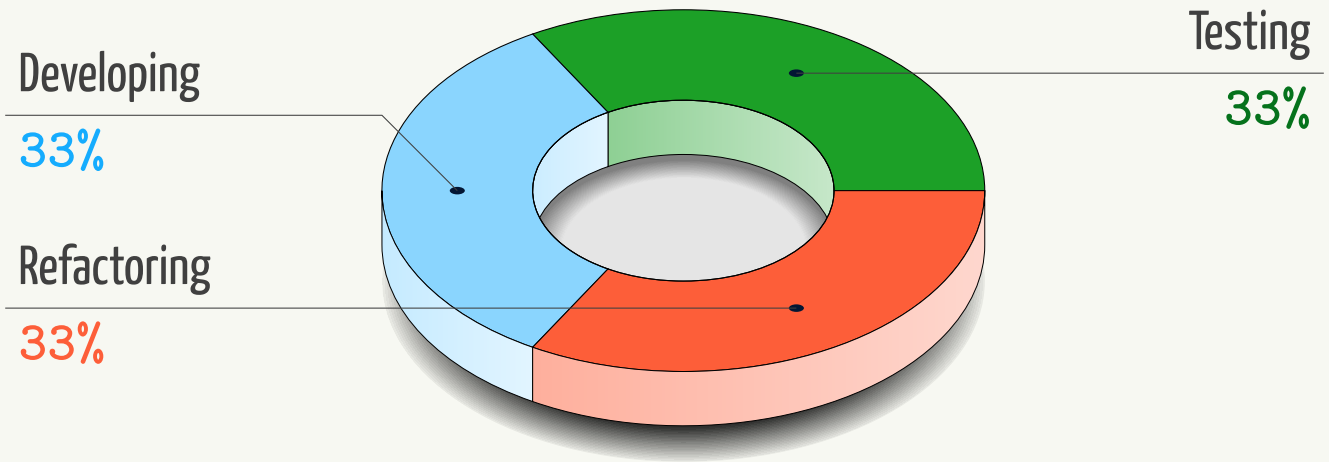
https://android.jlelse.eu/basics-of-unit-testing-affdd2273310

The correct mental approach to testing

- Testing is part of the natural evolution of the code (not an overhead)
- Test code has to fulfil the same clean code rules as production code
- Tests are the key to be free to change (improve) the production code!
- Keep balance between adding new code, writing tests and refactoring

The correct mental approach to testing

- Testing is part of the natural evolution of the code (not an overhead)
- Test code has to fulfil the same clean code rules as production code
- Tests are the key to be free to change (improve) the production code!
- Keep balance between adding new code, writing tests and refactoring



The five principles clean tests should fulfil

Fast:	To be run frequently
Independent:	To make failures meaningful
Repeatable:	To be run everywhere with same outcome
Self-validating:	To be automatised
Timely:	To make production code testable

Golden rule

Write (or at least think) F.I.R.S.T. tests first!

Unit tests

The deepest level of testing

Write tests for (every) unit of source code in the code base

- A unit: the smallest testable part of a code
- Strive to test a single concept per test function
- In most languages, unit tests frameworks are available
- Automatic build tools (e.g. CMake) offers a natural way to run tests
- Know how much code you tested
  → Code coverage tools (e.g. LCOV)

Unit tests

The deepest level of testing

Boost.Test ← Check me out!

Write tests for (every) unit of source code in the code base

Boost unit test framework

```
#define BOOST_TEST_MODULE Factorial
#include <boost/test/included/unit_test.hpp>
#include "factorial.hpp"

BOOST_AUTO_TEST_CASE(SpecialCases)
{
    BOOST_REQUIRE_EQUAL(factorial(0), 1);
    BOOST_REQUIRE_EQUAL(factorial(1), 1);
}

BOOST_AUTO_TEST_CASE(NormalCases)
{
    BOOST_REQUIRE_EQUAL(factorial(5), 120);
}
```

Unit tests

The deepest level of testing

Boost.Test ← Check me out!

Write tests for (every) unit of source code in the code base

./factorialTest

```
Running 2 test cases...
boostTest.cpp(12): fatal error in "SpecialCases":
    critical check factorial(0) == 1 failed [0 != 1]

*** 1 failure detected in test suite "Factorial"
```

Unit tests

The deepest level of testing

Boost.Test ← Check me out!

Write tests for (every) unit of source code in the code base

./factorialTest --report_level=short

```
Running 2 test cases...
boostTest.cpp(12): fatal error in "SpecialCases":
    critical check factorial(0) == 1 failed [0 != 1]

Test suite "Factorial" failed with:
  1 assertion out of 2 passed
  1 assertion out of 2 failed
  1 test case out of 2 passed
  1 test case out of 2 failed
  1 test case out of 2 aborted
```

Unit tests

The deepest level of testing

Boost.Test ← Check me out!

Write tests for (every) unit of source code in the code base

./factorialTest --report_level=detailed

```
Running 2 test cases...
boostTest.cpp(12): fatal error in "SpecialCases":
    critical check factorial(0) == 1 failed [0 != 1]

Test suite "Factorial" failed with:
  1 assertion out of 2 passed
  1 assertion out of 2 failed
  1 test case out of 2 passed
  1 test case out of 2 failed
  1 test case out of 2 aborted

  Test case "SpecialCases" aborted with:
    1 assertion out of 1 failed
  Test case "NormalCases" passed with:
    1 assertion out of 1 passed
```

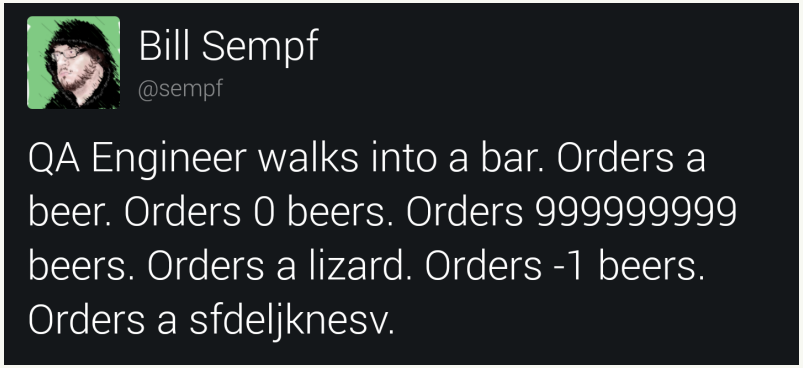
What should I test and how?

- Values

- Actions

What should I test and how?

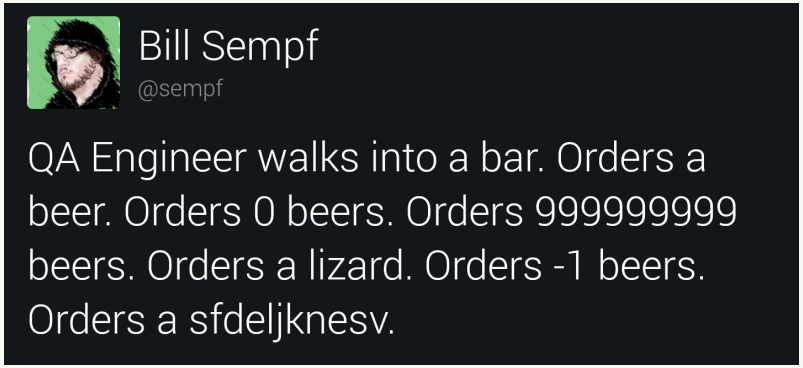
- Values



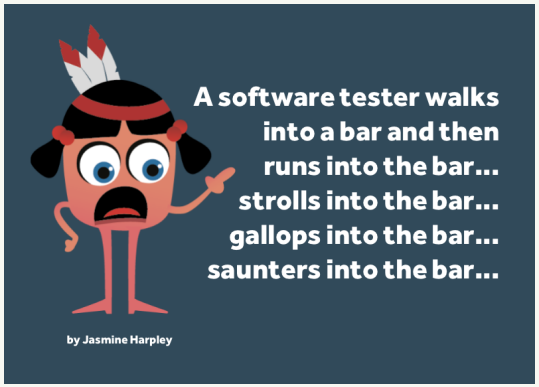
- Actions

What should I test and how?

- Values



- Actions



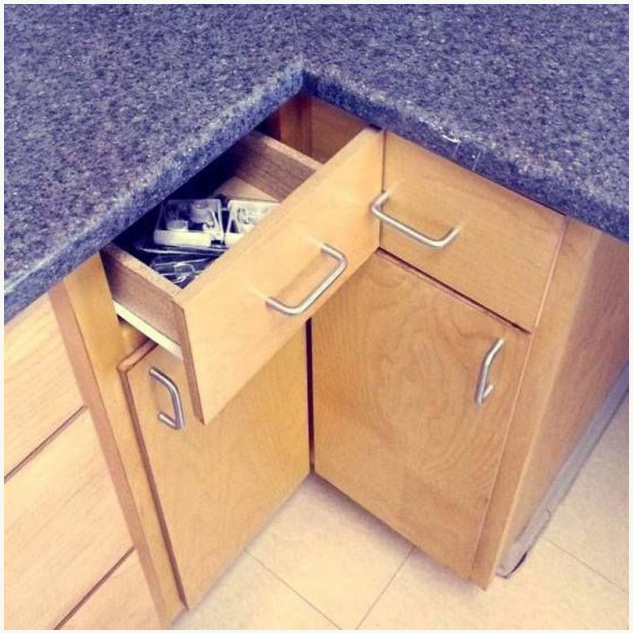
...and anything that makes sense!

Unit tests are not enough!

Unit tests are not enough!

Integration tests

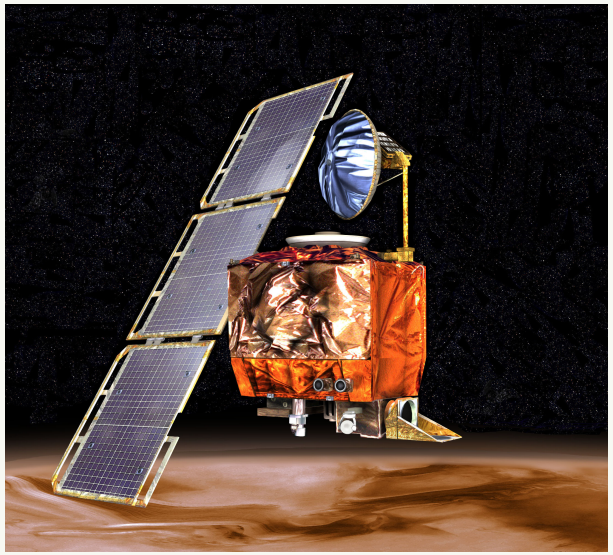
Once you are sure that all components work correctly alone, be sure they work as expected together!



Unit tests are not enough!

Integration tests

Once you are sure that all components work correctly alone, be sure they work as expected together!



Mars Climate Orbiter lost in 1999 – By NASA/JPL/Corby Waste – © Public Domain

General good practices

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Trivial example

```
double sum1=0;
for(const auto& data : dataSet1)
    sum1 += data;
sum1 /= dataSet1.size();


double sum2=0;
for(const auto& data : dataSet2)
    sum2 += data;
sum2 /= dataSet2.size();
```

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Trivial example

```
double calculateAverage(const std::vector<double>& dataSet)
{
    double sum=0;
    for(const auto& data : dataSet)
        sum += data;
    return sum / dataSet.size();
}

calculateAverage(dataSet1);
calculateAverage(dataSet2);
```

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Less trivial example

```
double calculateAverage(const std::vector<double>& dataSet)
{
    // ...
    throw std::runtime_error("Error in \"calculateAverage\"");
}
```

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Less trivial example

```
double calculateAverage(const std::vector<double>& dataSet)
{
    // ...
    throw std::runtime_error("Error in \"calculateAverage\"");
}
double calculateAverage(const std::vector<double>& dataSet)
{
    // ...
    using namespace std::string_literals;
    throw std::runtime_error("Error in \""s + __func__ + "\""s);
}
```


`__func__` in C++ is an implementation-defined string and is just one among other possibilities.

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Traversing a complex structure

```
// Imagine to have to do this very often
for(auto& line : wiki)
{
    if(line.isHeader())
        continue;
    for(auto& letter : line)
    {
        if(letter.isVowel())
            capitalize(letter);
    }
}
// How can you avoid duplication?
```

Don't Repeat Yourself → The DRY principle

- If you are doing Copy & Paste, you are doing it wrong!
- One of the most basic and yet most disregarded principle
- Code duplication makes the code less maintainable and easy to break

Traversing a complex structure

```
void traverseAndTransform(WikiPage& wiki,
                          std::function<void(Letter)> transform)
{
    for(auto& line : wiki){
        if(line.isHeader())
            continue;
        for(auto& letter : line)
            transform(letter);
    }
}

traverseAndTransform(wiki, /*lambda function*/);
```

Keep It Simple, Stupid → The KISS principle

Avoid complexity!

Everything should be done as simple as possible, but not simpler.

A. Einstein

- If you have a solution, ask yourself if there is an easier one
- Complexity reduces readability and maintainability
- Use libraries!
- Do not feel cool because you did something hard in a tricky way.
  Feel cool when you did the same in a trivial-to-understand way!
- If you really need something complicated (do you!?), then be sure everybody understands!

Keep It Simple, Stupid → The KISS principle

Simple example

```
std::vector<int> data = {1, -3, 2, 8, 4, -9};

auto first = data.begin(), last = data.end();
while (first != last) {
    if (*first == 4) {
        makeReportForUser();
        break;
    }
    ++first;
}
```

Keep It Simple, Stupid → The KISS principle

Simple example

```
std::vector<int> data = {1, -3, 2, 8, 4, -9};

auto it = std::find(data.begin(), data.end(), 4);
if(it != data.end())
    makeReportForUser();
```

Keep It Simple, Stupid → The KISS principle

Simple example

```
std::vector<int> data = {1, -3, 2, 8, 4, -9};

//Even better
if(isElementInVector(data, 4))
    makeReportForUser();
```

```
template<typename T>
bool isElementInVector(const std::vector<T>& data,
                       const T& value)
{
    auto it = std::find(data.begin(), data.end(), value);
    return it != data.end();
}
```

Keep It Simple, Stupid → The KISS principle

Example from real life (Quake III Arena, 1999)

Keep It Simple, Stupid → The KISS principle

Example from real life (Quake III Arena, 1999)

```
float Q_rsqrt(float number)
{
    long i;  float x2, y;  const float threehalfs = 1.5F;

    x2 = number * 0.5F;  y  = number;
    // evil floating point bit level hacking
    i  = * (long*) &y;
    i  = 0x5f3759df - (i >> 1);  // what the fuck?
    y  = * (float*) &i;
    // 1st iteration
    y  = y * (threehalfs - (x2 * y * y));
    // 2nd iteration, this can be removed
    // y  = y * (threehalfs - (x2 * y * y));
    return y;
}
```

Keep It Simple, Stupid → The KISS principle

Example from real life (Quake III Arena, 1999)

```
float Q_fastInverseSqrt(float number)
{
    float approximateResult
       = calculateApproximateInverseSqrt(number);
    return refineResult(approximateResult, number);
}
```

Keep It Simple, Stupid → The KISS principle

Example from real life (Quake III Arena, 1999)

```
float Q_fastInverseSqrt(float number)
{
    float approximateResult
       = calculateApproximateInverseSqrt(number);
    return refineResult(approximateResult, number);
}
```

```
float calculateApproximateInverseSqrt(float number)
{
    //Here some very low level operations are carried out.
    //Read here  for a detailed explanation.
    long floatAsInteger = * (long*) &number;
    floatAsInteger = 0x5f3759df - (floatAsInteger >> 1);
    return * (float*) &i;
}
```

Keep It Simple, Stupid → The KISS principle

Example from real life (Quake III Arena, 1999)

```
float Q_fastInverseSqrt(float number)
{
    float approximateResult
       = calculateApproximateInverseSqrt(number);
    return refineResult(approximateResult, number);
}
```

```
float refineResult(float result, float initialNumber)
{
    //Here one iteration of the Newton's method is used.
    //Read here for a detailed explanation.
    float refinedResult = initialNumber * result * result;
    return (3.0F - refinedResult) * result * 0.5F;
}
```

Beware of optimisations

Premature optimisation

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time:

Premature optimisation is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%.

D. Knuth

- Compiler do more and more a better job!
- Use optimisations flags (e.g. `-O1`, `-O3` with `g++` and `clang`)
- Never sacrifice clean code in name of **claimed** better performance!
- Only optimise after having used a profiler and having written tests!

The boy scout rule

Leave the campground cleaner than you found it

Always check in the code cleaner than you checked it out

- To clean an existing code base takes time and requires effort
- Aim for tiny but continuous progress
- Applying the boy scout rule is a way to pay back technical debt
- Any IDE will help you in quickly achieving easy refactoring

What to do, now?

DRY principle → `\againframe{<name>}`

Take home message

Clean code

What it is and why it is worth aiming at it

- The importance of meaningful names, comments and formatting
- Strive to respect few very general rules (e.g. DRY, KISS, pathfinder)
- Functions and classes: small, Single Responsibility and IOSP!
- Test your code and keep your tests clean (F.I.R.S.T.)!

Have a reason for everything you type → refactor, refactor, refactor!

The first reader of your code is yourself

Try to minimise the time needed by a new reader to understand!

Go back and immediately apply what you learnt!

Keep in mind good practices next time you code

- Use these slides as a starting point
- Try to apply principles day after day
- Feel guilty when you do something dirty
- Never give up in writing high quality code
- Refer to the literature and go beyond this material, keep improving
- Develop your code-sense, experience will make you quicker

Moral of the story

An old violinist got lost on his way to a performance. He stopped an old man on the corner and asked him how to get to Carnegie Hall. The old man looked at the violinist and the violin tucked under his arm and said: «Practice, son. Practice!»

R. C. Martin

Go back and immediately apply what you learnt!

Keep in mind good practices next time you code

- Use these slides as a starting point
- Try to apply principles day after day
- Feel guilty when you do something dirty
- Never give up in writing high quality code
- Refer to the literature and go beyond this material, keep improving
- Develop your code-sense, experience will make you quicker

Moral of the story

An old violinist got lost on his way to a performance. He stopped an old man on the corner and asked him how to get to Carnegie Hall. The old man looked at the violinist and the violin tucked under his arm and said: «Practice, son. Practice!»

R. C. Martin