Clean C++20

Sustainable Software Development Patterns and Best Practices

Second Edition

Stephan Roth

Clean C++20

Sustainable Software Development Patterns and Best Practices

Second Edition

Stephan Roth

Apress®

Clean C++20: Sustainable Software Development Patterns and Best Practices

Stephan Roth
Bad Schwartau, Schleswig-Holstein, Germany

ISBN-13 (pbk): 978-1-4842-5948-1 ISBN-13 (electronic): 978-1-4842-5949-8
https://doi.org/10.1007/978-1-4842-5949-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Jay Mantri on Unsplash (www.unsplash.com)

To Caroline and Maximilian: my beloved and marvelous family.

Table of Contents

About the Author

Stephan Roth, born on May 15, 1968, is a passionate coach, consultant, and trainer for Systems and Software Engineering with the German consultancy company *oose Innovative Informatik eG,* located in Hamburg. Before he joined oose, Stephan worked for many years as a software developer, software architect, and systems engineer in the field of radio reconnaissance and communication intelligence systems. He has developed sophisticated applications, especially for distributed systems with ambitious performance requirements, and graphical user interfaces using C++ and other programming languages. Stephan is also a speaker at professional conferences and the author of several publications. As a member of the *Gesellschaft für Systems Engineering e.V.*, the German chapter of the international Systems Engineering organization INCOSE, he is also engaged in the Systems Engineering community. Furthermore, he is an active supporter of the Software Craftsmanship movement and concerned with principles and practices of Clean Code Development (CCD).

Stephan Roth lives with his wife Caroline and their son Maximilian in Bad Schwartau, a spa in the German federal state of Schleswig-Holstein near the Baltic Sea.

You can visit Stephan's website and blog about systems engineering, software engineering, and software craftsmanship via the URL roth-soft.de. Please note that the articles there are mainly written in German.

On top of that, you can contact him via email or follow him at the networks listed here.

Email: `stephan@clean-cpp.com`

Twitter: `@_StephanRoth` (`https://twitter.com/_StephanRoth`)

LinkedIn: `www.linkedin.com/in/steproth`

About the Technical Reviewer

Marc Gregoire is a software engineer from Belgium. He graduated from the University of Leuven, Belgium, with a degree in “Burgerlijk ingenieur in de computer wetenschappen” (equivalent to a master of science degree in computer engineering). The year after, he received the *cum laude* degree of master in artificial intelligence at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working on international teams stretching from South America and the United States to Europe, the Middle East, and Asia. Currently, Marc works for Nikon Metrology on industrial 3D laser scanning software.

Acknowledgments

Writing a book like this one is never just the work of an individual person, the author. There are always numerous, fabulous people who contribute significantly to such a great project.

First, there was Steve Anglin of Apress. Steve contacted me in March 2016 for the first edition of *Clean C++*. He persuaded me to continue my book project with Apress Media LLC, which had been self-published at Leanpub. The self-publishing platform Leanpub served as a kind of "incubator" for a few years, but then I decided to finish and publish the book with Apress. Steve was also the one who contacted me in 2019 and asked me if I wanted to release a second edition that would take into account the emerging C++20 language standard. Well, he was obviously quite successful.

Next, I would like to thank Mark Powers, Editorial Operations Manager at Apress, for his great support during the writing of the manuscript for both editions. Mark was not only always available to answer questions, but his incessant follow-up on the progress of the manuscript was a positive incentive for me. I am very grateful to you, dear Mark.

In addition, many thanks also to Matthew Moodie, Lead Development Editor at Apress, who has provided proper help throughout the whole book development process.

A special thank you goes out to my technical reviewer Marc Gregoire. Marc critically examined every single chapter of both editions. He found many issues that I probably would have never found. He pushed me hard to improve several sections, and that was really valuable to me. Thank you!

Of course, I would also like to say a big thank you to the whole production team at Apress. They've done an excellent job regarding the finalization (copy editing, indexing, composition/layout, cover design, etc.) of the whole book up to the distribution of the final print (and eBook) files.

Last but not least, I would like to thank my beloved and unique family, especially for their understanding that a book project takes a great deal of time. Maximilian and Caroline, you're just wonderful.

CHAPTER 1

Introduction

"How it is done is as important as having it done."

—Eduardo Namur

Dear readers, I introduced the first edition of this book with the words: "It is still a sad reality that many software development projects are in bad condition, and some might even be in a serious crisis." That was a little over three years ago, and I am pretty sure that the general situation has not improved significantly since then.

The reasons that many software development projects are still having difficulties are manifold. There are a lot of risk factors that can cause software development projects to fail. Some projects, for example, are afflicted because of lousy project management. In other projects, the conditions and requirements constantly and rapidly change, but the development process does not support this high-dynamic environment. Furthermore, the all-important requirements elicitation and use case analysis is given little space in some projects. In particular, communication between external stakeholders, such as between domain experts and developers, can be difficult, leading to misunderstandings and the development of unnecessary features. And as if all this were not bad enough, quality assurance measures, such as testing, are given too little importance.

STAKEHOLDER

The term stakeholder in systems and software engineering is commonly used to refer to individuals or organizations that can potentially contribute requirements to a development project or that define important constraints for the project.

Usually, a distinction is made between external and internal stakeholders. Examples of *external stakeholders* are the customers, all users of the system, domain experts, system administrators, regulatory authorities, the legislators, etc. *Internal stakeholders* are those

S. Roth, *Clean C++20*, https://doi.org/10.1007/978-1-4842-5949-8_1

from within the development organization and can be the developers and software architects, business analysts, product management, requirements engineers, quality assurance, marketing personnel, etc.

The previously listed points are all typical and well-known problems in professional software development, but beyond that, another fact exists: **In some projects the code base is poor quality!**

That does not necessarily mean that the code is not working correctly. Its *external quality,* measured by the quality assurance (QA) department using integration or acceptance tests, can be pretty high. It can pass QA without complaints, and the test report might state that they found nothing wrong. The software users might also possibly be satisfied and happy, and the development may even have been completed on time and on budget (… which is rare, I know). Everything seems to be fine on first sight … really, everything?!

Nevertheless, the *internal quality* of this code, which works correctly, can be very poor. Often the code is difficult to understand and horrible to maintain and extend. Countless software units, like classes or functions, are very large, some of them with thousands of lines of code, making their comprehensibility and adaptability a serious challenge. Too many dependencies between software units lead to unwanted side effects if something changes. The software has no perceivable architecture. Its structure seems to be randomly originated and some developers speak about "historically grown software" or "architecture by accident." Classes, functions, variables, and constants have bad and mysterious names, and the code is littered with lots of comments: some of them are outdated, just describe obvious things, or are plain wrong. Developers are afraid to change something or to extend the software because they know that it is rotten and fragile, and they know that unit test coverage is poor, if there are any unit tests at all. "Never touch a running system" is a statement that is frequently heard from people working within such kinds of projects. The implementation of a new feature doesn't just need a few hours or days until it is ready for deployment; it takes several weeks or even months.

This kind of bad software is often referred to as a *big ball of mud*. This term was first used in 1997 by Brian Foote and Joseph W. Yoder in a paper for the Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97). Foote and Yoder describe the big ball of mud as "… a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle." Such software systems are costly and time-wasting maintenance nightmares, and they can bring a development organization to its knees!

The pathological phenomena just described can be found in software projects in all industrial sectors and domains. The programming language they use doesn't matter. You'll find big balls of mud written in Java, PHP, C, C#, C++, and other more or less popular languages. Why is that so?

Software Entropy

First of all, there is the natural law of entropy, or disorder. Just like any other closed and complex system, software tends to get messier over time. This phenomenon is called *software entropy*. The term is based on the second law of thermodynamics. It states that a closed system's disorder cannot be reduced; it can only remain unchanged or increase. Software seems to behave this way. Every time a new function is added or something is changed, the code gets a little bit more disordered. There are also numerous influencing factors that can contribute to software entropy:

- Unrealistic project schedules raise the pressure and abet developers to botch things and to do their work in a bad and unprofessional way.
- The immense complexity of today's software systems, both technically and in terms of the requirements to be satisfied.
- Developers with different skill levels and experience.
- Globally distributed, cross-cultural teams, enforcing communication problems.
- Development mainly pays attention to the functional aspects (functional requirements and the system's use cases) of the software, whereby the quality requirements (non-functional requirements), such as performance, efficiency, maintainability, availability, usability, portability, security, etc., are neglected or at worst are fully ignored.
- Inappropriate development environments and bad tools.
- Management is focused on earning money quickly and doesn't understand the value of sustainable software development.
- Quick and dirty hacks and non-design-conformable implementations (*broken windows*).

THE BROKEN WINDOW THEORY

The *Broken Window Theory* was developed in connection with American crime research. The theory states that a single destroyed window at an abandoned building can be the trigger for the dilapidation of an entire neighborhood. The broken window sends a fatal signal to the environment: "Look, nobody cares about this building!" This attracts further decay, vandalism, and other antisocial behavior. The Broken Window Theory has been used as the foundation for several reforms in criminal policy, especially for the development of zero-tolerance strategies.

In software development, this theory was taken up and applied to the quality of code. Hacks and bad implementations, which are not compliant with the software design, are called "broken windows." If these bad implementations are not repaired, more hacks to deal with them may appear in their neighborhood. And thus, code dilapidation is set into motion.

Don't tolerate "broken windows" in your code—*fix them!*

Why C++?

"C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, you blow away your whole leg!"

—Bjarne Stroustrup, Bjarne Stroustrup's FAQ: Did you really say that?

First and foremost, phenomena like software entropy, code smells, anti-patterns, and other problems with the internal software quality, are basically independent of the programming language. However, it seems to be that C and C++ projects are especially prone to messiness and tend to slip into a bad state. Even the World Wide Web is full of bad, but apparently very fast and highly optimized, C++ code examples. They often have a cruel syntax that completely ignores elementary principles of good design and well-written code. Why is that?

One reason for this might be that C++ is a multi-paradigm programming language on an intermediate level; that is, it comprises both high-level and low-level language features. C++ is like a melting pot that blends many different ideas and concepts together. With this language, you can write procedural, functional, or object-oriented programs, or even a mixture of all three. In addition, C++ allows *template metaprogramming* (TMP), a technique in which so-called templates are used by a compiler to generate temporary source code that is merged with the rest of the source code and then compiled. Ever since the release of ISO standard C++11 (ISO/IEC 14882:2011 [ISO11]) in September 2011, even more ways have been added; for example, functional programming with anonymous functions are now supported in a very elegant manner by lambda expressions. As a consequence of these diverse capabilities, C++ has a reputation for being very complex, complicated, and cumbersome. And with each standard after C++11 (C++14, C++17, and now C++20), a lot of new features were added, which have further increased the complexity of the language.

Another cause for bad software could be that many developers didn't have an IT background. Anyone can begin to develop software nowadays, no matter if they have a university degree or any other apprenticeship in computer science. A vast majority of C++ developers are (or were) non-experts. Especially in the technological domains automotive, railway transportation, aerospace, electrical/electronics, or mechanical engineering domains, many engineers slipped into programming during the last decades without having an education in computer science. As the complexity grew and technical systems contained more and more software, there was an urgent need for programmers. This demand was covered by the existing workforce. Electrical engineers, mathematicians, physicists, and lots of people from strictly nontechnical disciplines started to develop software. They learned to do it mainly by self-education and hands-on, by simply doing it. And they have done it to their best knowledge and belief.

Basically, there is absolutely nothing wrong with that. But sometimes just knowing the tools and the syntax of a programming language is not enough. Software development is not the same as programming. The world is full of software that was tinkered together by improperly trained software developers. There are many things on abstract levels a developer must consider to create a sustainable system, for example, architecture and design. How should a system be structured to achieve certain quality goals? What is this object-oriented thing good for and how do I use it efficiently? What are the advantages and drawbacks of a certain framework or library? What are the differences between various algorithms, and why doesn't one algorithm fit all similar problems? And what the heck is a deterministic finite automaton, and why does it help to cope with complexity?!

But there is no reason to lose heart! What really matters to a software program's ongoing health is that someone cares about it, and clean code is the key!

Clean Code

What, exactly, is meant by “clean code”?

A major misunderstanding is to confuse clean code with something that can be called “beautiful code.” Clean code doesn’t have necessarily to be beautiful (...whatever that means). Professional programmers are not paid to write beautiful or pretty code. They are hired by development companies as experts to create customer value.

Code is clean if it can be understood and maintained easily by any team member.

Clean code is the basis of fast code. If your code is clean and test coverage is high, it only takes a few hours or a couple of days to implement, test, and deploy a change or a new function—not weeks or months.

Clean code is the foundation for sustainable software; it keeps a software development project running over a long time without accumulating a large amount of technical debt. Developers must actively tend the software and ensure it stays in shape because the code is crucial for the survival of a software development organization.

Clean code is also the key to being a happier developer. It leads to a stress-free life. If your code is clean and you feel comfortable with it, you can keep calm in every situation, even when facing a tight project deadline.

All of the points mentioned here are true, but the key point is this: ***Clean code saves money!*** In essence, it’s about economic efficiency. Each year, development organizations lose a lot of money because their code is in bad shape. Clean code ensures that the value added by the development organization remains high. Companies can earn money from its clean code for a long time.

C++11: The Beginning of a New Era

> *“Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever.”*
>
> —Bjarne Stroustrup, C++11 - the new ISO C++ standard [Stroustrup16]

After the release of the C++ language standard C++11 (ISO/IEC 14882:2011 [ISO11]) in September 2011, some people predicted that C++ would undergo a renaissance. Some even spoke of a revolution. They predicted that the idiomatic style of how development was done with this “modern C++” would be significantly different and not comparable to the “historical C++” of the early 1990s.

No doubt, C++11 has brought a bunch of great innovations and changed the way we think about developing software with this programming language. I can say with full confidence that C++11 has set such changes in motion. With C++11, we got move semantics, lambda expressions, automatic type deduction, deleted and defaulted functions, a lot of enhancements of the Standard Library, and many more useful things.

But this also meant that these new features came on top of the already existing features. It is not possible to remove a significant feature from C++ without breaking large amounts of existing code bases. This means that the complexity of the language increased, because C++11 is larger than its predecessor C++98, and thus it is harder to learn this language in its entirety.

Its successor, C++14, was an evolutionary development with some bug fixes and minor enhancements. If you plan to switch to modern C++, you should at least start with this standard and skip C++11.

Three years later, with C++17, numerous new features were added again, but this revision also removed a few. And in December 2020, the C++ standardization committee completed and published the new C++20 standard, which is called “the next big thing” by some people. This standard again adds lots of new features besides many extensions to the core language, the Standard Library, and other stuff, especially the so-called “big four”: Concepts, Coroutines, Ranges Library, and Modules.

If we look at C++ development over the past 10 years, we can see that the complexity of the language has increased significantly. In the meantime, C++23 development has already begun. I question whether this is the right way to go about things in the long run. Perhaps it would be appropriate at some point not only to permanently add functionalities, but also to review the existing features, consolidate them, and simplify the language again.

Who This Book Is For

As a trainer and consultant, I have had the opportunity to look at many companies that are developing software. Furthermore, I observe very closely what is happening in the developer scene. And I’ve recognized a gap.

My impression is that C++ programmers have been ignored by those promoting software craftsmanship and clean code development. Many principles and practices, which are relatively well known in the Java environment and in the hip world of web or game development, seem to be largely unknown in the C++ world.

This book tries to close that gap a little, because even with C++, developers can write clean code! If you want to learn about writing clean C++, this book is for you. It is written for C++ developers of all skill levels and shows by example how to write understandable, flexible, maintainable, and efficient C++ code. Even if you are a seasoned C++ developer, there are interesting hints and tips in this book that you will find useful in your work.

This book is not a C++ primer! In order to use the knowledge in this book efficiently, you should already be familiar with the basic concepts of the language. If you just want to start with C++ development and still have no basic knowledge of the language, you should first learn the basic concepts, which can be done with other books or with a good C++ introduction training. This book also does not discuss every single new C++20 language feature, or the features of its predecessors, in detail. As I have already pointed out, the complexity of the language is now relatively high. There are other very good books that introduce the language from A to Z.

Furthermore, this book doesn't contain any esoteric hack or kludge. I know that a lot of nutty and mind-blowing things are possible with C++, but these are usually not in the spirit of clean code and should not be used to create a clean and modern C++ program. If you are really crazy about mysterious C++ pointer calisthenics, this book is not for you.

Apart from that, this book is written to help C++ developers of all skill levels. It shows by example how to write understandable, flexible, maintainable and efficient C++ code. The presented principles and practices can be applied to new software systems, sometimes called *greenfield projects*, as well as to legacy systems with a long history, which are often pejoratively called *brownfield projects*.

Note Please consider that not every C++ compiler currently supports all of the new language features, especially not those from the latest C++20 standard, completely.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic font is used to introduce new terms and names.

Bold font is used within paragraphs to emphasize terms or important statements.

`Monospaced font` is used within paragraphs to refer to program elements such as class, variable, or function names, statements, and C++ keywords. This font is also used to show command line inputs, an address of a website (URL), a keystroke sequence, or the output produced by a program.

Sidebars

Sometimes I pass on small bits of information that are tangentially related to the content around it, which can be considered separate from that content. Such sections are known as sidebars. Sometimes I use a sidebar to present an additional or contrasting discussion about the topic around it.

THIS HEADER CONTAINS THE TITLE OF A SIDEBAR

This is the text in a sidebar.

Notes, Tips, and Warnings

Another kind of sidebar for special purposes is used for notes, tips, and warnings. They are used to provide some special information, to provide a useful piece of advice, or to warn you about things that can be dangerous and should be avoided.

Note This is the text of a note.

Code Samples

Code examples and code snippets appear separately from the text, syntax-highlighted (keywords of the C++ language are bold), and in a monospaced font. Longer code sections usually have numbered titles. To reference specific lines of the code example in the text, code samples sometimes include line numbers (see Listing 1-1).

Listing 1-1. A Line-Numbered Code Sample

```
01  class Clazz {
02  public:
03    Clazz();
04    virtual ~Clazz();
05    void doSomething();
06
07  private:
08    int _attribute;
09
10    void function();
11  };
```

To better focus on specific aspects of the code, irrelevant parts are sometimes obscured and represented by a comment with an ellipsis (…), like in this example:

```
void Clazz::function() {
  // ...
}
```

Coding Style

Just a few words about the coding style I use in this book.

You may get the impression that my programming style has a strong likeness to typical Java code, mixed with the Kernighan and Ritchie (K&R) style. I've spent nearly 20 years as a software developer, and even later in my career, I have learned other programming languages (for instance, ANSI-C, Java, Delphi, Scala, and several scripting languages). Hence, I've adopted my own programming style, which is a melting pot of these different influences.

Maybe you will not like my style, and you instead prefer Linus Torvald's Kernel style, the Allman style, or any other popular C++ coding standard. This is of course perfectly okay. I like my style, and you like yours.

C++ Core Guidelines

You may have heard of the *C++ Core Guidelines*, found at `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html` [Cppcore21]. This is a collection of guidelines, rules, and good practices for programming with modern C++. The project is hosted on GitHub and released under a MIT-style license. It was initiated by Bjarne Stroustrup, but has a lot more editors and contributors, e.g., Herb Sutter.

The number of rules and recommendations in the *C++ Core Guidelines* is pretty high. There are currently 30 rules on the subject of interfaces alone, about the same number on error handling, and no less than 55 rules on functions. And that is by no means the end of the story. Further guidelines exist on topics such as classes, resource management, performance, and templates.

I first had the idea of linking the topics in my book to the rules from the *C++ Core Guidelines.* But that would have led to countless references to the guidelines and might even have reduced the readability of the book. Therefore, I have largely refrained from doing so, but would like to explicitly recommend the *C++ Core Guidelines* at this point. They are a very good supplement to this book, even though I do not agree with every rule.

Companion Website and Source Code Repository

This book is accompanied by a companion website: `www.clean-cpp.com`.

The website includes:

- The discussion of additional topics not covered in this book.
- High-resolution versions of all the figures in this book.

Some of the source code examples in this book, and other useful additions, are available on GitHub at:

`https://github.com/Apress/clean-cpp20`

You can check out the code using Git with the following command:

```
$> git clone https://github.com/clean-cpp/book-samples.git
```

You can get a .ZIP archive of the code by going to `https://github.com/clean-cpp/book-samples` and clicking the Download ZIP button.

UML Diagrams

Some illustrations in this book are UML diagrams. The *Unified Modeling Language* (UML) is a standardized graphical language used to create models of software and other systems. In its current version, 2.5.1, UML offers 15 diagram types to describe a system entirely.

Don't worry if you are not familiar with all diagram types; I use only a few of them in this book. I present UML diagrams from time to time to provide a quick overview of certain issues that possibly cannot be understood quickly enough by just reading the code. Appendix A contains a brief overview of the used UML notations.

CHAPTER 2

Build a Safety Net

"Testing is a skill. While this may come as a surprise to some people, it is a simple fact.:

—Mark Fewster and Dorothy Graham,
Software Test Automation, 1999

That I start the main part of this book with a chapter about testing may be surprising to some readers, but this is for several good reasons. During the past few years, testing on certain levels has become an essential cornerstone of modern software development. The potential benefits of a good test strategy are enormous. All kinds of tests, if well engineered, can be helpful and useful. In this chapter, I describe why I think that unit tests, especially, are indispensable to ensure a fundamental level of high quality in software.

Note that this chapter is about what is sometimes called POUT ("plain old unit testing") and not the design-supporting tool test-driven development (TDD), which I cover in Chapter 8.

The Need for Testing

1962: NASA MARINER 1

The Mariner 1 spacecraft was launched on July 22, 1962, as a Venus flyby mission for planetary exploration. Due to a problem with its directional antenna, the Atlas-Agena B launching rocket worked unreliably and lost its control signal from ground control shortly after launch.

This exceptional case had been considered during design and construction of the rocket. The Atlas-Agena launching vehicle switched to automatic control by the on-board guidance computer. Unfortunately, an error in the software of that computer led to incorrect control commands that caused a critical course deviation and made steering impossible. The rocket was directed toward Earth and pointed to a critical area.

S. Roth, *Clean C++20*, https://doi.org/10.1007/978-1-4842-5949-8_2

At T+293 seconds, the Range Safety Officer sent the destruct command to blow the rocket. A NASA examination report[1] mentions a typo in the computer's source code, the lack of a hyphen (-), as the cause of the error. The total loss was $18.5 million, which was a huge amount of money in those days.

If software developers are asked why tests are good and essential, I suppose that the most common answer would be the reduction of bugs, errors, or flaws. No doubt this is basically correct: testing is an elementary part of quality assurance.

Software bugs are usually perceived as an unpleasant nuisance. Users are annoyed about the wrong behavior of the program, which produces invalid output, or they are seriously ticked off about regular crashes. Sometimes even odds and ends, such as a truncated text in a dialog box of a user interface, are enough to significantly bother software users in their daily work. The consequence may be an increasing dissatisfaction with the software, and at worst its replacement by another product. In addition to a financial loss, the image of the software manufacturer suffers from bugs. At worst, the company gets into serious trouble and many jobs are lost.

But the previously described scenario does not apply to every piece of software. The implications of bugs can be much more dramatic.

1986: THERAC-25 MEDICAL ACCELERATOR DISASTER

This case is probably the most consequential failure in the history of software development. The Therac-25 was a radiation therapy device. It was developed and produced from 1982 until 1985 by the state-owned enterprise Atomic Energy of Canada Limited (AECL). Eleven devices were produced and installed in clinics in the United States and Canada.

Due to bugs in the control software, an insufficient quality assurance process, and other deficiencies, three patients lost their lives caused due to radiation overdoses. Three other patients were irradiated and suffered permanent, heavy health problems.

An analysis of this case determined that, among other things, the software was written by only one person who was also responsible for the tests.

[1]NASA National Space Science Data Center (NSSDC): Mariner 1, http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1, retrieved 2021-0305.

When people think of computers, they usually have a desktop PC, laptop, tablet, or smartphone in mind. And if they think about software, they usually think about web shops, office suites, or business IT systems.

But these kinds of software and computers make up only a very small percentage of all systems with which we have contact every day. Most software that surrounds us controls machines that physically interact with the world. Our whole life is managed by software. In a nutshell: **There is no life today without software!** Software is everywhere and an essential part of our infrastructure.

If we board an elevator, our lives are in the hands of software. Aircrafts are controlled by software, and the entire, worldwide air traffic control system depends on software. Our modern cars contain a significant amount of small computer systems with software that communicates over a network, responsible for many safety-critical functions of the vehicle. Air conditioning, automatic doors, medical devices, trains, automated production lines in factories … no matter what we're doing nowadays, we permanently come in touch with software. And with the *digital revolution* and the *Internet of Things* (IoT), the relevance of software in our life will again increase significantly. This fact could not get more evident than with the autonomous (driverless) car.

It is unnecessary to emphasize that any bug in these software-intense systems could have catastrophic consequences. A fault or malfunction of an important system can be a threat to lives or physical condition. At worst, hundreds of people could lose their lives during a plane crash, possibly caused by a wrong `if` statement in a subroutine of the Fly-by-Wire subsystem. Quality is under no circumstances negotiable in these kinds of systems. **Never!**

But even in systems without functional safety requirements, bugs can have serious implications, especially if they are subtler in their destructiveness. It is easy to imagine that bugs in financial software could trigger a worldwide bank crisis. Imagine if the financial software of an arbitrary big bank completed every posting twice due to a bug, and this issue was not noticed for a few days.

1990: THE AT&T CRASH

On January 15th, 1990, the AT&T long distance telephone network crashed and 75 million phone calls failed for the following nine hours. The blackout was caused by a single line of code (a wrong `break` statement) in a software upgrade that AT&T deployed to all 114 of its computer-operated electronic switches (4ESS) in December 1989. The problem began the afternoon of January 15 when a malfunction in AT&T's Manhattan control center led to a chain reaction and disabled switches throughout half the network.

The estimated loss for AT&T was $60 million. There were also probably a huge amount of losses for businesses that relied on the telephone network.

Introduction to Testing

There are different levels of quality assurance measures in software development projects. These levels are often visualized in the form of a pyramid—the so-called *test pyramid*. The fundamental concept was developed by the American software developer Mike Cohn, one of the founders of the Scrum Alliance. He described the test automation pyramid in his book, *Succeeding with Agile* [Cohn09]. With the aid of the pyramid, Cohn describes the degree of automation required for efficient software testing. In the following years, the test pyramid has been further developed by different people. The one depicted in Figure 2-1 is my version.