**2 About author**

Stephan Roth, born on May 15, 1968, is a consultant and trainer for Systems and Software Engineering . He 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.

3 **About Clean C++**

Stephan Roth: “My book *Clean C++* is about writing maintainable, extensible, and durable software with modern C++.”

To my mind, this book is a must for every developer, software architect, or team leader who is interested in good C++ code, and thus also wants to save development costs. If you want to teach yourself about writing clean C++ this book is exactly what you need. It is written to help C++ developers of all skill levels and shows by example how to write understandable, flexible, maintainable, and efficient C++ code.

**5 Clean C++ is not a C++ primer!**

It is not a book to introduce and learn the programming language C++. It is only if you want to teach yourself about writing clean and modern C++, this book is exactly what you need

**6 Contents at a Glance**

The book consists of nine chapters.

**7 Chapter one: Introduction**

How it is done is as important as having it done. —Eduardo Namur

The first chapter deals with the external and internal quality of the code.

This section includes topics such as:

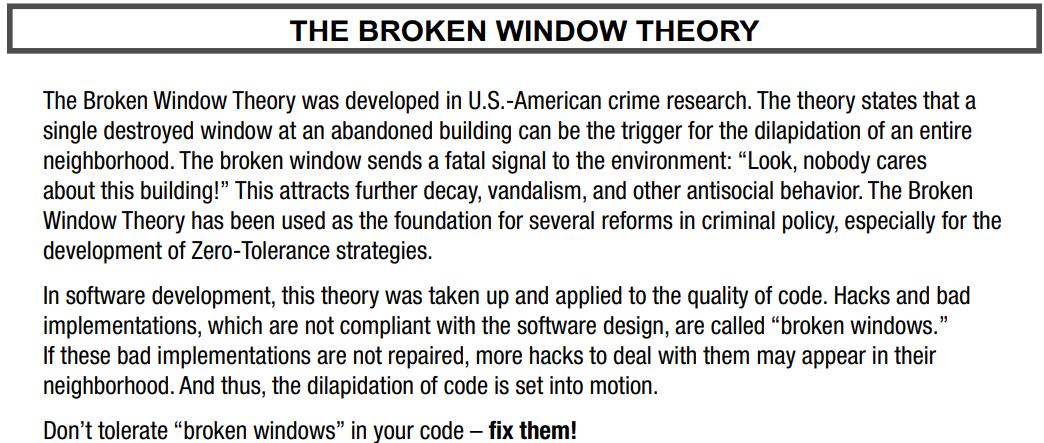
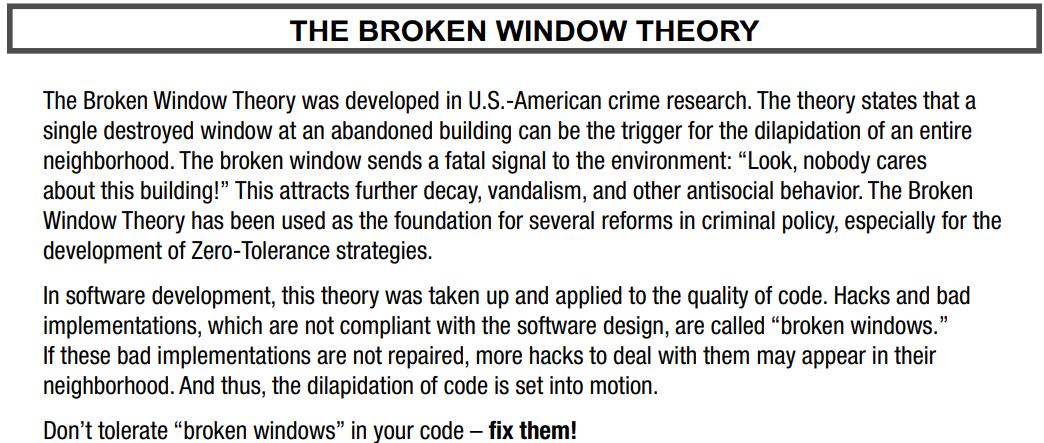
* Software Entropy
* Clean Code
* C++11 – The Beginning of a New Era
* Who this book is for
* Conventions used in this book
* UML Diagrams

**8 Why C++?**

## The creator of C++ Bjarne Stroustrup (Б'ярн Страуструп) said so about this language:

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

**9 Don’t tolerate “broken windows” in your code – fix them!**

To make the reader understand the importance of writing quality code, a story about the theory of a broken window is given.

**10 Chapter 2: Build a Safety Net**

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 the author describe why he think that Unit

Tests, especially, are indispensable to ensure a fundamental level of excellent quality in software

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

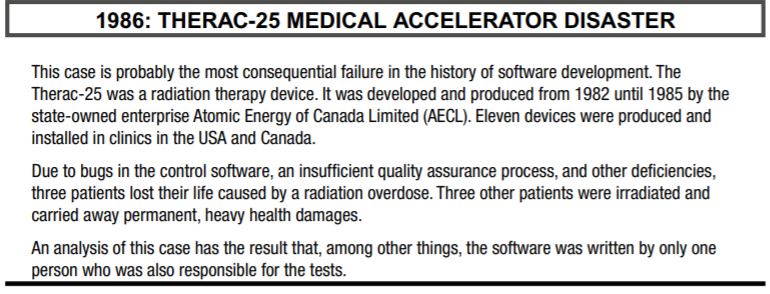
—Mark Fewster and Dorothy Graham, Software Test Automation, 1999

**This section includes topics such as:**

* The need for testing
* Introduction into testing
* Unit tests
* What about QA?
* Rules for good unit tests

**11 The Need for Testing**

To help readers understand the importance of code testing, Stephan Roth cites as an example the tragic story about THERAC-25 MEDICAL ACCELERATOR DISASTER and a costly mistake



Thanks to such a story, it becomes clear how important code testing is

**12 Test Pyramid**

The fundamental concept was developed by the American software developer Mike Cohn, one of the founders of the Scrum Alliance. With the aid of the pyramid, Cohn describes the degree of automation required for efficient software testing.

**13 Unit Tests**

A unit test is a piece of code that executes a small part of your production code base in a particular context.

The test will show you in a split second that your code works as you expect it to work. If unit test coverage

is pretty high, and you can check in less than a minute that all parts of your system under development are

working correctly.

There are several different unit test frameworks available for C++ development, for example, CppUnit,

Boost.Test, CUTE, Google Test, and a couple more.

In principle, all these frameworks follow the basic design of so-called xUnit, which is a collective name

for several unit test frameworks that derive their structure and functionality from Smalltalk’s SUnit.

**14 Rules for Good Unit Tests**

* Unit Test Naming

An expressive and descriptive naming of unit tests is very important. Author’s advice is to establish

naming standards for all tests

For general, multipurpose classes that can be used in different contexts, an expressive name could

contain the following parts:

• The precondition of the test scenario, that is, the state of the SUT before the test was executed.

• The tested part of the unit under test, typically the name of the tested procedure, function, or method.

• The expected test result.

That leads to a name template for unit test methods, as here on the slide

* Unit Test Independence

Each unit test must be independent to all the others. It would be fatal if tests must be executed in a specific

order because one test is based on the result of the previous one. Never write a unit test whose result is the

prerequisite for a subsequent test. Never leave the unit under test in an altered state, which is a precondition

for the following tests.

* One Assertion per Test
* Independent Initialization of Unit Test Environments
* Exclude Getters and Setters

These member functions are typically so simple that it would be foolish to write unit tests for them. Furthermore, usual getters and setters are implicitly tested by other and more important unit tests.

* Exclude Third-Party Code

Don’t write tests for third-party code! We don’t have to verify that libraries or frameworks do work as expected.

* Exclude External Systems

The same as for third-party code is true for external systems. Don’t write tests for systems that are in the context of your system to be developed, and thus not in your responsibility

* Don’t Mix Test Code with Production Code
* Tests Must Run Fast

Unit tests should establish a rapid feedback loop for developers. The execution of all unit tests for a large project should not last longer than about 3 minutes, and rather less time than that.

**15 Chapter 3: Be Principled**

I would advise students to pay more attention to the fundamental ideas rather than the latest technology. The technology will be out-of-date before they graduate. Fundamental ideas never get out of date.

—David L. Parnas

This section includes topics such as:

* KISS
* YAGNI
* DRY
* Information hiding
* Strong cohesion
* Loose coupling
* PLA
* The Boy Scout Rule

In this chapter, Stephan Roth introduce the most important and fundamental principles of well-designed and wellcrafted software. What is special about these principles is that they are not tied to certain programming

paradigms or programming languages. Some of them are not even specific to software development. For

instance, KISS principle can be relevant to many areas of life: generally speaking, it is not a bad

idea to make everything as simple in life as possible – not only software development.

That is, you should not learn of the following principles once and then forget them. These advices are

given for you to internalize. These principles are so important that they should, ideally, become second

nature to every developer. And many of the more concrete principles that the author discuss later in this book have

their roots in the following basic principles.

**16 Contents at a Glance**

* KISS - “ “Keep it simple and stupid”

In eXtreme Programming this principle is represented by a practice named “Do the simplest thing that could possibly work”

The KISS principle states that simplicity should be a major goal in software development, and that

unnecessary complexity should be avoided.

* YAGNI - You Aren’t Gonna Need It!

This principle is tightly coupled to the previously KISS principle. YAGNI is an acronym for “You Aren’t Gonna Need It!” It states that you should not write code that is not necessary at the moment, but might be in the future

* DRY - Don’t repeat yourself!

Although this principle is one of the most important. DRY is an acronym for “Don’t repeat yourself!” and states that we should avoid duplication, because duplication is evil.

* The Boy Scout Rule - Always leave the campground cleaner than you found it.

Whenever we find something in a piece of code that needs to be improved we should fix it immediately. And it does not matter who the original author of this piece of code was.

And the improvement doesn’t have to be a big deal. It may be a very small

clean-up, for example:

• Renaming a poorly named class, variable, function, or method

**17 Chapter 4: Basics of Clean C++**

*Programs must be written for people to read, and only incidentally for machines to execute.*

**This section includes topics such as:**

* Good names
* Comments
* Functions

In this chapter Stephan Roth describe the general basics of clean C++. These are sometimes universal things that are often programming language independent. For example, giving a good name is essential in all programming languages.

**18**

* Good names

Source code files, namespaces, classes, templates, functions, arguments, variables, and constants should have meaningful and expressive names

Use simple but descriptive and self-explaining names.

* Comments

Truth can only be found in one place: the code.

Code should tell a story and be self-explanatory. Comments must be avoided whenever possible.

Advice from the author: In Integrated Development Environment with syntax coloring, the color for comments is usually preconfigured to green or teal. You should change this color to red! A comment in the source code should be something special, which should attract the attention of the developer.

* Functions

Functions are the heart of any software system. They

represent the first organizational unit above the lines of code. Well-written functions foster the readability

and maintainability of a program considerably.

Functions should be pretty small. Ideally 4–5 lines, maximum 12–15 lines, but not more.

The name of a function should start with a verb. Predicates, that is, statements about an object that can be true or false, should start with “is” or “has.”

The name of a function should express its purpose, and not explain how it works.

Real functions should have as few arguments as possible.

Prefer simple object construction on the stack instead of on the heap

In a function’s argument list, use references instead of pointers but If it is inevitable to deal with a pointer to a resource, use a smart one

Also, pay attention to const correctness. Use const as much as possible, and choose always a proper declaration of variables or objects as mutable or immutable.

**19 Thank you for your attention!**