Software Quality Factors

Alexandre Bergel
abergel@dcc.uchile.cl

Source

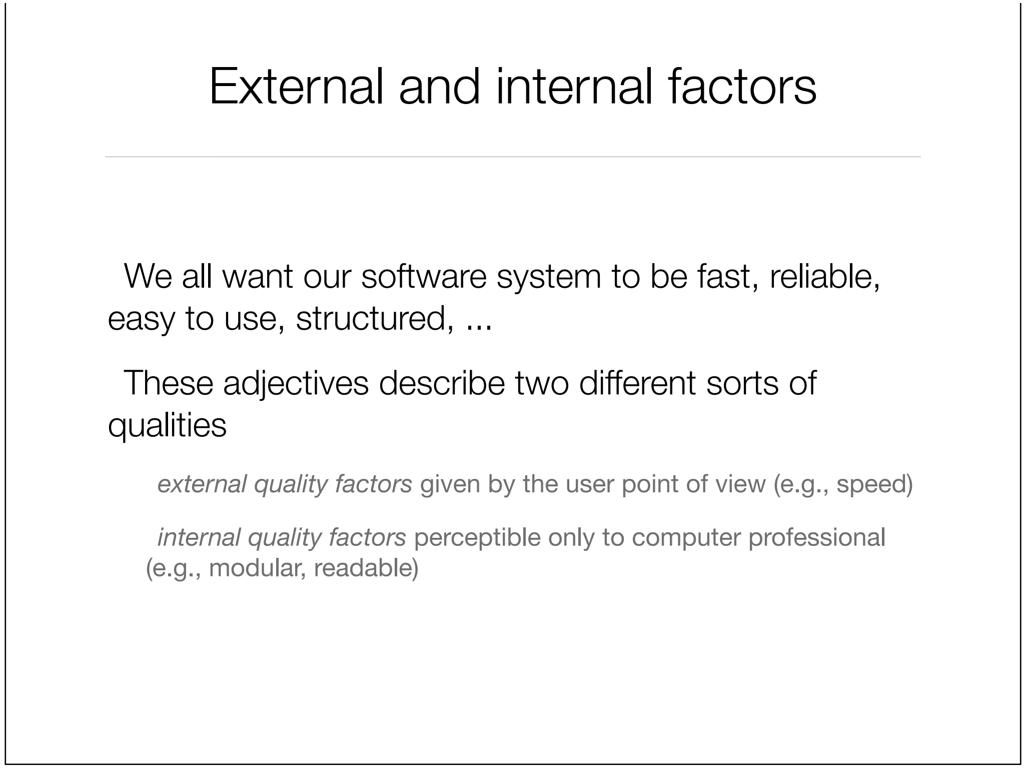
Object-Oriented Software Construction, second edition, Bertrand Meyer

OBJECT–ORIENTED
SOFTWARE CONSTRUCTION
SECOND EDITION
CD-ROM INCLUDED
The Most Comprehensive, Definitive O-O Reference Ever Published
An O-O Tour de Force by a Pioneer in the Field
CD-ROM Includes Complete Hypertext Version of Book AND Object-Oriented Development Environment
Winner
Software Development
PRODUCT EXCELLENCE AWARD 1997
BERTRAND MEYER

Engineering seeks quality

Software engineering is the production of quality software

Software quality is best described as a combination of several factors



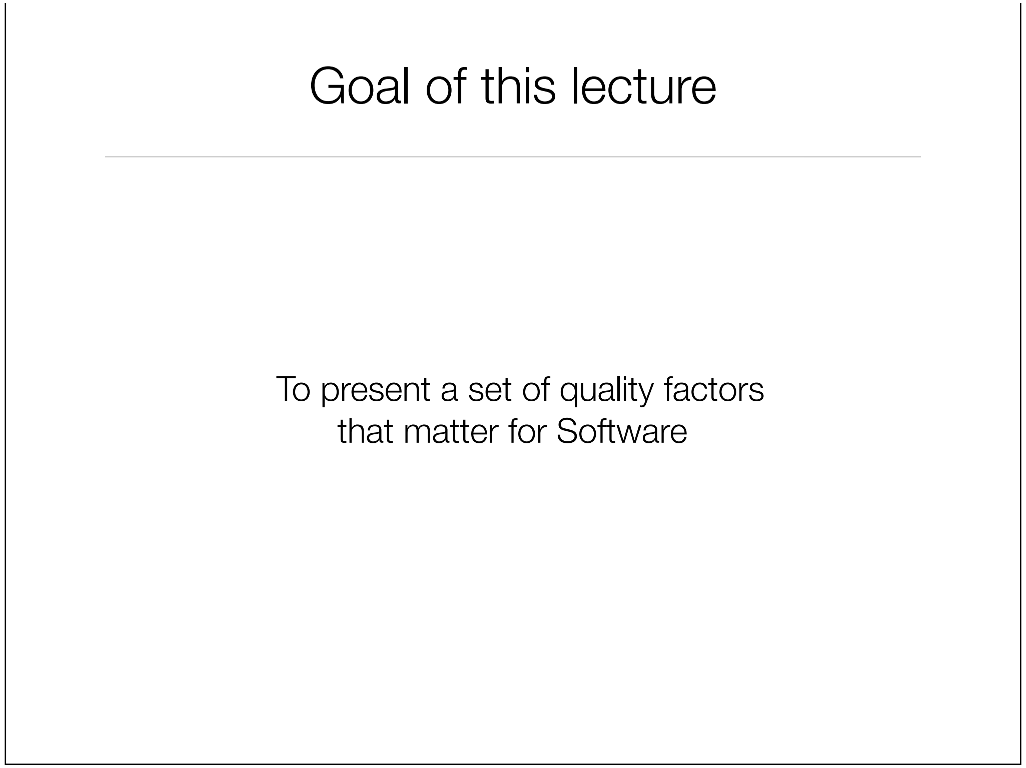
External and internal factors

We all want our software system to be fast, reliable, easy to use, structured, ...

These adjectives describe two different sorts of qualities

- *external quality factors* given by the user point of view (e.g., speed)
- *internal quality factors* perceptible only to computer professional (e.g., modular, readable)

In the end, only external factors matter. If I use a web browser, little do I care whether the source program is readable or modular. But the key to achieving these external factors is in the internal ones: for the users to enjoy the visible qualities, the designers and implementers must have applied internatl techniques that will ensure the hidden qualities



Goal of this lecture

To present a set of quality factors
that matter for Software

We should not however lose track of the global picture; the internal techniques are not an end in themselves, but a means to reach external software qualities

A review of external factors

Correctness

Robustness

Extendibility

Reusability

Compatibility

Efficiency

Portability

Ease of use

Timeliness

Correctness if the ability of software products to perform their exact tasks, as defined by their specification

Correctness is the prime quality

If a system does not do what it is supposed to do, everything else matters little

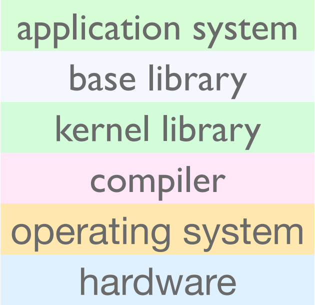
But this is easier said than done...

Requirement specification plays an important rôle

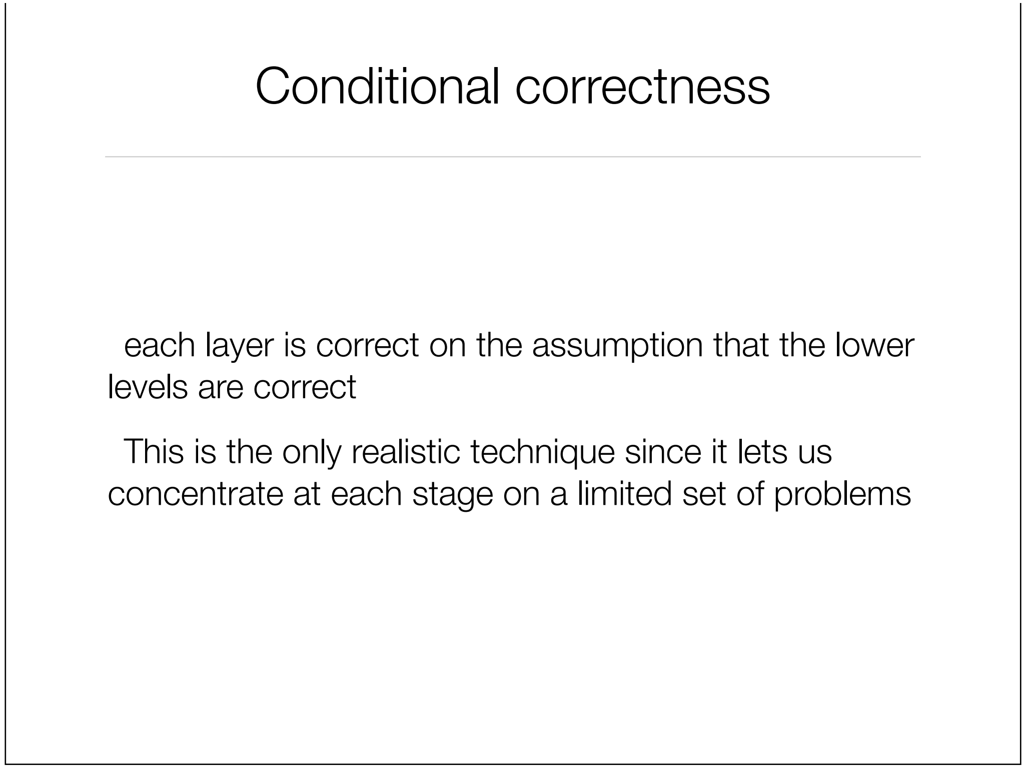
Layers in software development

Methods for ensuring correctness will usually be *conditional*

A software touches so many areas that it is often impossible to guarantee its correctness



application system
base library
kernel library
compiler
operating system
hardware



Conditional correctness

each layer is correct on the assumption that the lower levels are correct

This is the only realistic technique since it lets us concentrate at each stage on a limited set of problems

You cannot usefully check that a program in a high-level language X is correct unelss you are able to assume that a program in a high-level language X is correct unless you are able to assume that the compiler on hand implements X correctly. This does not necessarily mean that you trust the compiler blindly, simply that you separate the two components of the problem: compiler correctness, and correctness of your program relative to the language’s semantics.

Note that some attempts to prove the correctness of compilers are being conducted at INRIA with the OCaml compiler.

Some techniques to gain correctness

testing

debugging

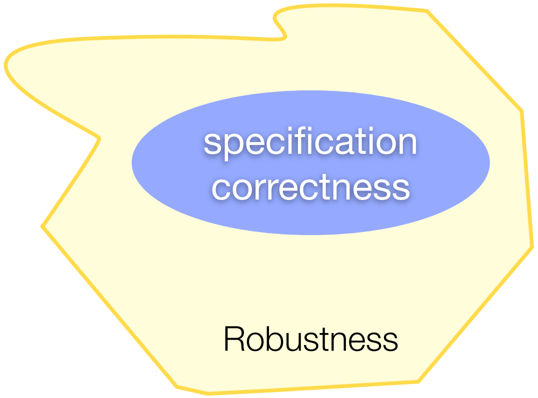
type checking

use of assertions

use of formal methods

Robustness is the ability of software systems to react appropriately to abnormal conditions

Robustness complements correctness



Correctness addresses the behavior of a system in cases covered by its specification

Robustness characterizes what happens outside of that specification

Some techniques to gain robustness

Providing erroneous input

Getting a community of users

...

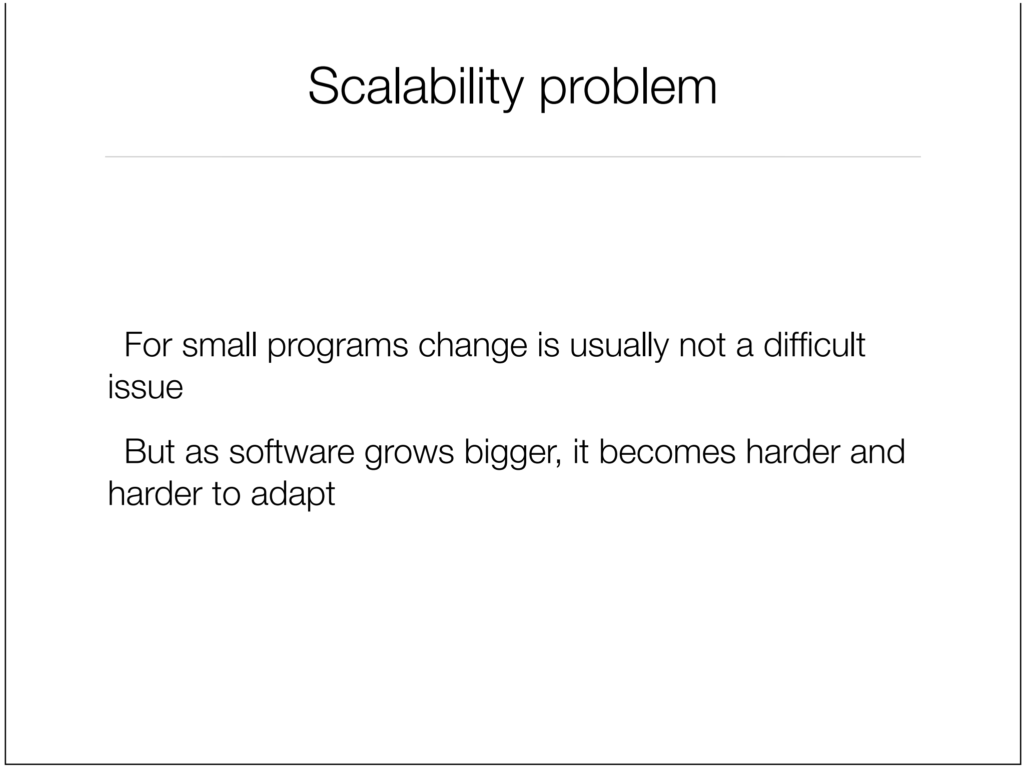
Extensibility is the ease of adapting software products to changes of specification

Software is supposed to be *soft*

- Nothing can be easier than to change a program if you have access to its source code

- Just use your favorite text editor

- The problem of extendibility is one of scale



Scalability problem

For small programs change is usually not a difficult issue

But as software grows bigger, it becomes harder and harder to adapt

We need extendibility because at the basis of all software lies some human phenomenon and hence fickleness. The obvious case of business software where passage of a law of a company’s acquisition may suddenly invalidate the assumptions on which a system rested.

A large software system often looks to its maintainers as a giant house of cards

Not a new problem

Traditional approaches to software engineering did not take enough account of changes

Two principles for extendibility

- *design simplicity*: a simple architecture will always be easier to adapt to changes than a complex one
- *decentralization*: the more autonomous the modules are, the higher chance a simple change will affect a small number of modules only. Chain of reaction must be avoided

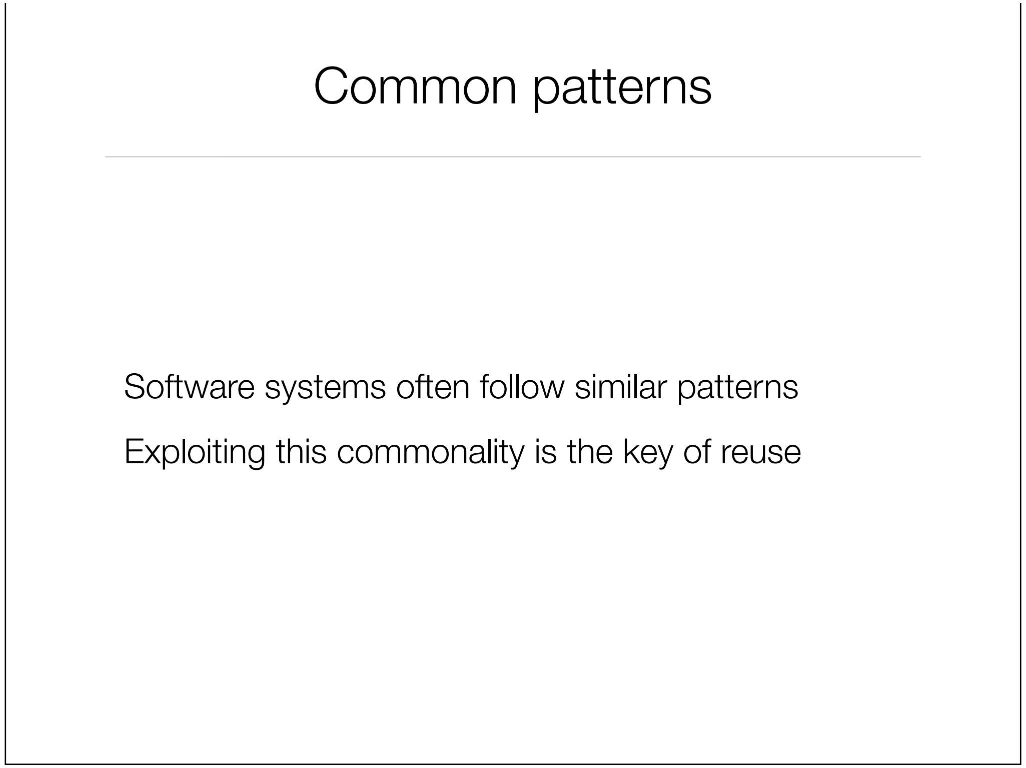
Some techniques to gain Extensibility

Language constructs

- classes, modules, functions, packages

Design patterns

Reusability is the ability of software elements to serve for the construction of many different applications



Common patterns

Software systems often follow similar patterns

Exploiting this commonality is the key of reuse

It should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before.
By capturing such a pattern, a reusable software element will be applicable to many different developments.

Some techniques to gain reusability

Language support

- classes, modules, functions, packages

Documentation

- unit test, textual description, formal description

Compatibility is the ease of combining software elements with others

Some techniques to gain reusability

Standardized file format

- text format, xml

Standardized data structure

- in Lisp, all data and programs are represented by binary trees

Standardized user interfaces

Standardized access protocols

- Corba, Ole-com ActiveX, Service Web

Efficiency is the ability to place as few demands on hardware resources

Some techniques to gain efficiency

Adoption of better algorithm

Memory profiling

Network monitoring

Execution benchmarking

Portability is the ease of transferring software products to various hardware and software environments

Not only hardware

Portability addresses variations not just of the physical hardware but more generally of the *hardware-software machine*

Ease of use is the ease with which people of various backgrounds and qualification can learn to use software product

Structural simplicity

As with many other qualities, one of keys to ease of use is structural simplicity

A well-designed system, built according to a clear, well thought-out structure, will be easier to use than a messy one

The condition is not sufficient of course, but it helps considerably

Importance of OO techniques

Object-oriented languages appear at first to address design and implementation

But it yields powerful new interface ideas that help the end users

User interface design principle

Do not pretend you know the user;
you just don’t

Functionality is the extent of possibilities provided by a system

A good software product is based on a small number of powerful ideas

Timeliness is the ability of a software system to be released when or before its users want it

Where Moose is useful?

Correctness

Robustness

Extendibility

Reusability

Compatibility

Efficiency

Portability

Ease of use

Where Moose is useful?

Correctness

Robustness

Extendibility

Reusability

Compatibility

Efficiency

Portability

Ease of use