



1

Trusted Components

Bertrand Meyer
ETH Zürich / Eiffel Software



2

My background

- Since 2001: Professor of Software Engineering at ETH Zürich
- Since 1985: Founder (now Chief Architect) of Eiffel Software, in Santa Barbara. Produces Eiffel tools and services
- Also adjunct professor at Monash University in Australia (since 1998)



Scope of our work at ETH

3

- Help move software technology to the next level through
 - Trusted Components
 - Advanced O-O techniques
 - Teaching (including introductory)
- Approaches of special interest
 - Eiffel
 - .NET
 - B



Other activities

4

- Journal of Object Technology JOT
www.jot.fm
- Numerous workshops and conferences
- LASER Summer School (Applied Software Engineering), starting September 2004



For numerous papers and other info

5

<http://www.inf.ethz.ch/~meyer>

<http://se.inf.ethz.ch>



Proposition

6

Major progress in software engineering requires switching to the systematic production and use of components of guaranteed quality.



- “Most of the improvement in the reliability of computer systems has come from improvement in the basic components”
- “You’ll see ever increasing portions of the effort devoted to design and verification”



- What does it take to bring software engineering to the next level?



- The building of **quality** software



	Technical	Management
A priori	<ul style="list-style-type: none">▪ Design methods▪ O-O▪ Programming language choice▪ Formal development	<ul style="list-style-type: none">▪ User involvement▪ Executive support▪ Education (engineers, managers...)
A posteriori	<ul style="list-style-type: none">▪ White-box testing▪ Static analysis▪ Proofs (of existing programs)	<ul style="list-style-type: none">▪ Testing, validation, acceptance procedures



Obstacles to achieving top quality

11

- Industry has not been that excited
(not worth the investment)
(except security)
- Anti-intellectual attitude
e.g. formal methods
“Worse is better”
Fad effects
- Academia is not that interested either
(hard to publish)



AI Davis, IEEE Computer, March 2003

12

“At a large telecommunications company, an operating division had contacted us about a project. The project manager analyzed the job and concluded that it could be done in 12 months. The customer wanted it in 9 months.

We could simply tell the customer that it couldn't be done. Or we could agree to 9 months. After all, it was not impossible, just extremely improbable...”



The new obsession with security may be the best thing that happened to software engineering

Example: Buffer overflows (again last week with Blaster...)

But viewpoints are different:

- Reliability engineer: it shouldn't crash
- Security engineer: if it crashes, we're safe



- Find a program that puts its argument into a finite-size buffer and doesn't check that the argument fits
- Use a big enough argument
- Overwrite return address...



Buffer overflow

15

- A software engineering issue:
 - Methodology
 - Programming languages
 - Verification
- Revealed through security problems



Good idea: Process models

16

CMM, ISO....

- **Good: force a systematic process**
- **But: concentrate on form, not substance**



What makes a project successful? The original CHAOS study identified 10 success factors. No project requires all 10 factors to be successful, but the more factors, the higher the confidence level.

CHAOS Ten	
User Involvement	20 Points
Executive Support	15 Points
Clear Business Objectives	15 Points
Experienced Project Manager	15 Points
Small Milestones	10 Points
Firm Basic Requirements	5 Points
Competent Staff	5 Points
Proper Planning	5 Points
Ownership	5 Points
Other	5 Points



“Agile” methods, refactoring, test-based development

- Good: rehabilitates the act of programming
- But: tests are not specs!



Good idea: Formal methods

19

B, Abstract State Machines

- Good: benefit from mathematics
(**IF** accompanied with proofs!)
- But: expensive



Good idea: open source

20

GNU, Linux....

- Good: energy, enthusiasm, collaboration
- But: quality not central concern



Today's software is often good enough

21

Overall:

- Works most of the time
- Doesn't kill too many people
- Negative effects, esp. financial, are diffuse

Significant improvements since early years:

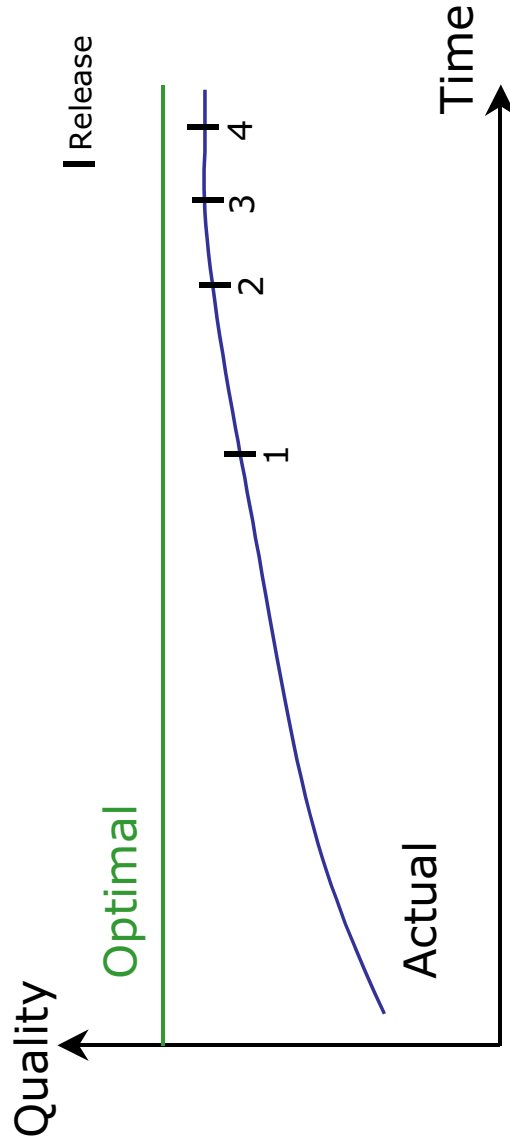
- Better languages
- Better tools
- Better practices (configuration management)



From "good enough" to good?

22

- Beyond "good enough", quality is economically bad
- He who perfects, dies

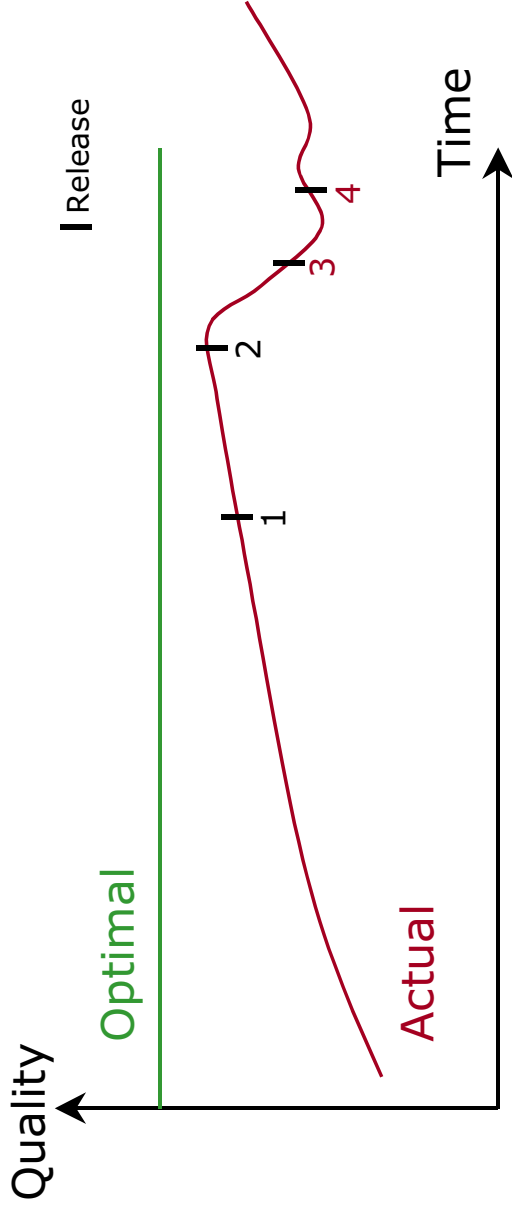




From “good enough” to good?

23

- Beyond “good enough”, quality is economically bad
- He who perfects, dies



The economic argument

24

- **Stable system:**
 - Sum of individual optima = Global optimum
- **Non-component-based development:**
 - Individual optimum = “Good Enough Software”
 - Improvements: I am responsible!
- **Component-based development:**
 - Interest of both consumer and producer: Better components
 - Improvements: Producer does the job



Quality through reuse

25

- **The good news:**

Reuse scales up everything



Quality through reuse

26

- **The good news:**

Reuse scales up everything

- **The bad news:**

Reuse scales up everything



Trusted components

27

- Confluence of
 - Quality engineering
 - Reuse



Hennessy

28

- “Most of the improvement in the reliability of computer systems has come from improvement in the basic components”
- “You’ll see ever increasing portions of the effort devoted to design and verification”



Component-based for

- Guaranteed quality
- Faster time to market
- Ease of maintenance
- Standardization of software practices
- Preservation of know-how

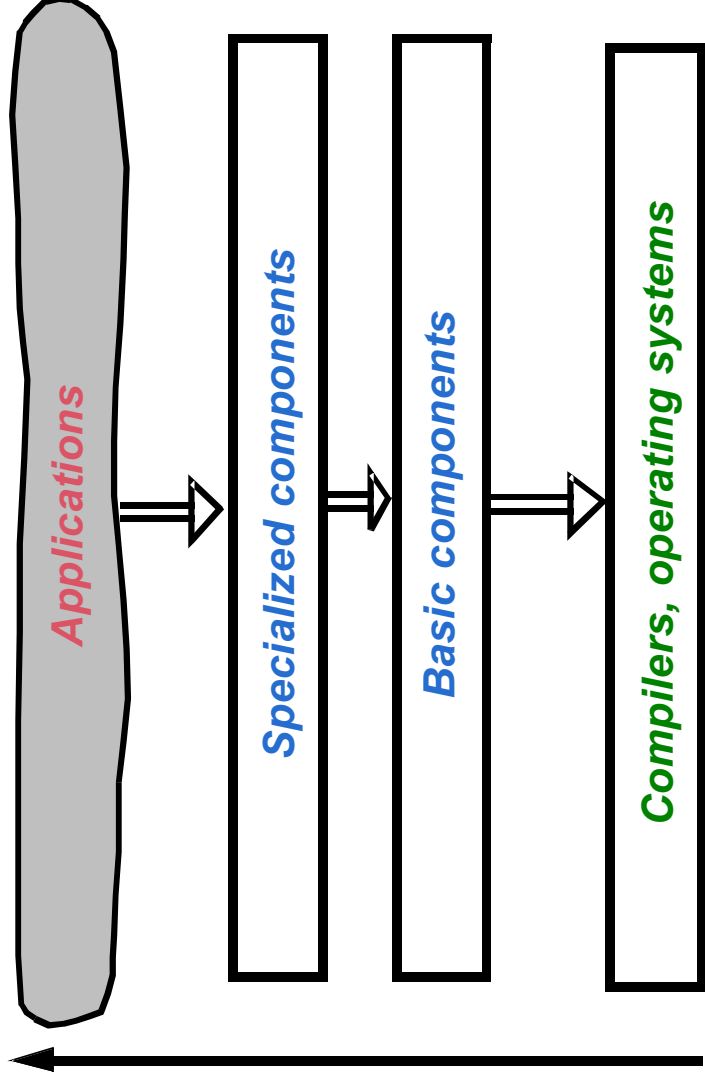


- **The key issue**
 - Bad-quality components are major risk
 - Deficiencies scale up, too**
 - High-quality components could transform the state of the software industry (if it wanted to — currently doesn't)



Where to focus effort?

31



Perfectionism

32

- Component design should be Formula-1 racing of software “engineering”.
- In component development, perfectionism is good.



What exactly is a component?

33

Working definition:

Program element such that:

- It may be used by other program elements (not just humans, or non-software systems).
These elements will be called “clients”
- Its authors need not know about the clients.
- Clients’ authors need only know what the component’s author tells them.



Classifying components by...

34

Lifecycle role:

- Analysis
- Design
- Implementation

Flexibility:

- Static
- Dynamic
- Replaceable

Abstraction level:

- Functional (subroutine)
- Casual (package)
- Data (class)
- Cluster (framework)
- System (binary comp.)

Form of use:

- Interface only
- Source only
- Source + hiding

Economics:

- Free
- Purchased
- Rented



This is a broad view of components

35

- Encompasses patterns and frameworks
- Software, especially with object technology, permits “pluggable” components (“don’t call us, we’ll call you), where client programmers can insert their own mechanisms.
- Supports component families



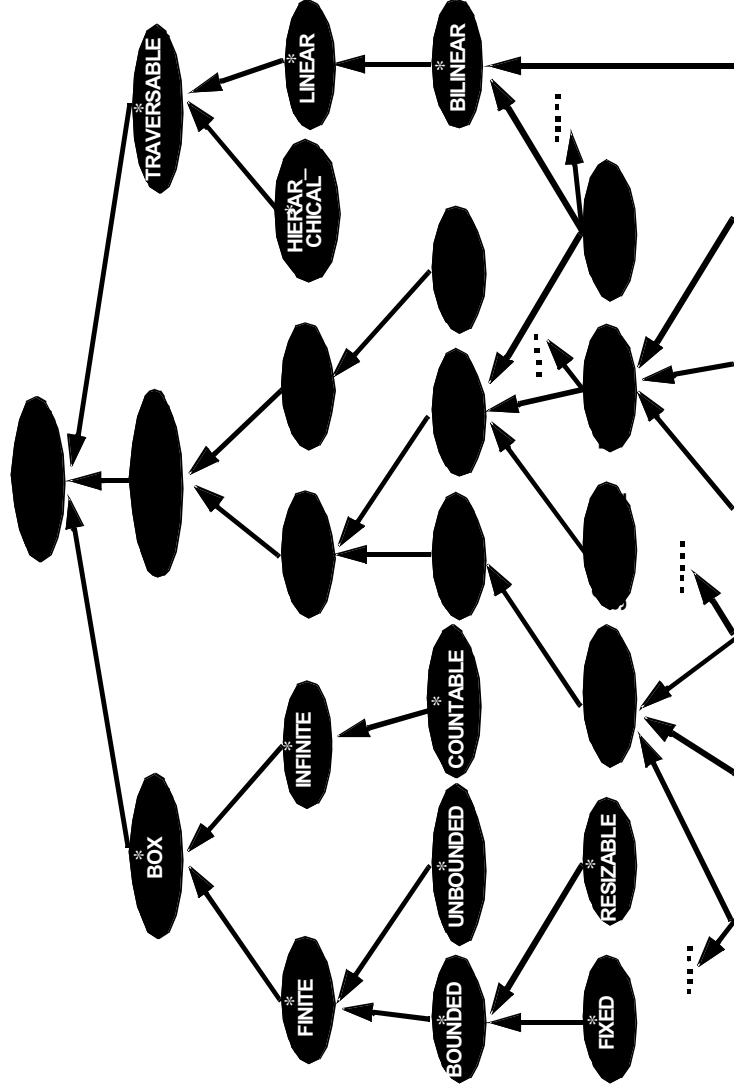
From patterns to components

36

- Patterns are both one of the greatest advances in software engineering, and a step backwards from the push for reuse through object technology
- We should try to turn successful patterns into components!
- Systematic effort in progress at ETH (Karine Arnout)



- Collection classes (“Knuthware”)
- Consistency principle
- Strict design principles: command-query separation, operand-option separation, taxonomy, uniform access...
- Strict interface and style rules





How to get there

39

- **Low road:**
 - Component Certification
→ Component Certification Center
 - Component Quality Model
- **High road:**
 - Proofs of correctness



A Component Certification Center

40

- Principles
- Methods and processes
- Standards: **Component Quality Model**
- Services for component providers and component consumers



Component Quality Model

41

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension



Component Quality Model

42

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

- A.1 Some reuse attested
- A.2 Producer reputation
- A.3 Published evaluations



Component Quality Model

43

A: Acceptance

B: Behavior

- B.1 Examples
- B.2 Usage documentation
- B.3 Preconditioned
- B.4 Some postconditions
- B.5 Full postconditions
- B.6 Observable invariants

C: Constraints

D: Design

E: Extension



Component Quality Model

44

A: Acceptance

B: Behavior

- C.1 Platform spec
- C.2 Ease of use
- C.3 Response time
- C.4 Memory occupation
- C.5 Bandwidth
- C.6 Availability
- C.7 Security

C: Constraints

D: Design

E: Extension



Contract levels

45



Type



Functional specification



Performance specification



Quality of Service

(Source: Jézéquel, Mingins et al.)



Component Quality Model

46

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

- E.1 Portable across platforms
- E.2 Mechanisms for addition
- E.3 Mechanisms for redefinition
- E.4 User action pluggability



Component Quality Model

47

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

- D.1 Precise dependency doc
- D.2 Consistent API rules
- D.3 Strict design rules
- D.4 Extensive test cases
- D.5 Some proved properties
- D.6 Proofs of preconditions, postconditions & invariants



The high road: towards proofs?

48

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

- D.1 Precise dependency doc
- D.2 Consistent API rules
- D.3 Strict design rules
- D.4 Extensive test cases
- D.5 Some proved properties
- D.6 Proofs of preconditions, postconditions & invariants



Proof technology and formal methods

49

- Constant advances in recent years
- PVS, Isabelle, Coq, ...
- B (method and tool)
- Most applications: life-critical systems in transportation, defense etc. Example: security system of Paris Metro METEOR line



Formal methods and reuse

50

- Components should be good
- Proofs should be economical!



“Proving classes”

51

EiffelBase libraries (fundamental data structures and algorithms):

- Classes are equipped with contracts
- “Proving a class” means proving that the implementation satisfies the contracts



Hennessy

52

- “Most of the improvement in the reliability of computer systems has come from improvement in the basic components”
- “You’ll see ever increasing portions of the effort devoted to design and verification”



- Semantic theory for full O-O language (Eiffel)
- General strategy for proving contract-equipped classes
- Mathematical basis: partial functions
- Build a **model** for each structure
- No need to extend assertion language
- Start from object structures, including pointers
- Calculus of Object Structures



- **Contracts in non-Eiffel libraries**
 - The “Closet Contract Conjecture”
 - Analysis of .NET Collection library (Karine Arnout)
 - Possible automation?
- **Contract-based test generation**
- **Trusted Reusable Components**
 - Design Patterns vs. Reusable components
 - Eiffel Event Library



Related work: Concurrency

55

- **SCOOP model**
 - Simple language extension supporting many different forms of concurrency and distribution
- **Research directions**
 - Access control
 - Real-time applications
 - Implementation for .NET multithreading



Teaching

56

- **Introduction to Programming (starting Fall 03)**
 - “**Inverted curriculum**”: outside-in
 - Based on reuse and imitation; give students heaps of code
 - Use Eiffel, Design by Contract
 - Use libraries from the start
 - Exciting application domain
 - Give students heaps of code
 - From consumers to producer (outside-in)
 - Abstraction: teach, don’t preach
- **Textbook: “*Touch of Class*”**
- Ongoing project, mailing list, instructor’s manual



Some of the challenges ahead

57

General:

- Convince the software engineering community
- Convince industry (producers, consumers)
- Define ambitious, feasible objectives
- Achieve balance between high and low road

“High road”:

- Finish up the theory
- Produce mechanized proofs

“Low road”:

- Define standard terminology
- Get the economics right



Proposition

58

The biggest hope and challenge for the software industry is at the confluence of quality engineering (especially formal methods) and reuse.

“Trusted Components”

Now is the time to do it.



For numerous papers and other info

59

<http://se.inf.ethz.ch>

<http://www.inf.ethz.ch/~meyer>