Trusted Components

Bertrand Meyer

ETH Zürich / Eiffel Software

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

- Help move software technology to the next level through
- Trusted Components
- Advanced O-O techniques
- Teaching (including introductory)

Other activities

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

Approaches of special interest

- Eiffel
- .NET
- B
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”

The challenge

What does it take to bring software engineering to the next level?
# Software “engineering”

- The building of **quality** software

## Ways to quality

<table>
<thead>
<tr>
<th>Technical</th>
<th>Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A priori</strong></td>
<td><strong>A posteriori</strong></td>
</tr>
<tr>
<td>Design methods</td>
<td>White-box testing</td>
</tr>
<tr>
<td>O-O</td>
<td>Static analysis</td>
</tr>
<tr>
<td>Programming language choice</td>
<td>Proofs (of existing programs)</td>
</tr>
<tr>
<td>Formal development</td>
<td>Testing, validation, acceptance procedures</td>
</tr>
<tr>
<td></td>
<td>User involvement</td>
</tr>
<tr>
<td></td>
<td>Executive support</td>
</tr>
<tr>
<td></td>
<td>Education (engineers, managers...)</td>
</tr>
</tbody>
</table>

*Chair of Software Engineering*
Obstacles to achieving top quality

- 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)

Al Davis, IEEE Computer, March 2003

“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

Buffer overflow

- 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

- A software engineering issue:
  - Methodology
  - Programming languages
  - Verification

- Revealed through security problems

Good idea: Process models

CMM, ISO...

- Good: force a systematic process
- But: concentrate on form, not substance
Good idea: eXtreme Programming

“Agile” methods, refactoring, test-based development

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

B, Abstract State Machines

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

Good idea: open source

GNU, Linux...

- Good: energy, enthusiasm, collaboration
- But: quality not central concern
Today’s software is often good enough

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?

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

![Graph showing quality over time](chart.png)
From “good enough” to good?

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

The economic argument

- 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

- The good news:
  Reuse scales up everything

- The bad news:
  Reuse scales up everything
Trusted components

- Confluence of
  - Quality engineering
  - Reuse

Hennessy

- “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

Component quality: the inevitable issue

- 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?

Applications

Specialized components

Basic components

Compilers, operating systems

Perfectionism

- Component design should be Formula-1 racing of software “engineering”.

- In component development, perfectionism is good.
What exactly is a component?

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...

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

- 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

- 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)
Our experience: Eiffelbase

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

Eiffelbase hierarchy
How to get there

- Low road:
  - Component Certification
    → Component Certification Center
  - Component Quality Model

- High road:
  - Proofs of correctness

A Component Certification Center

- Principles

- Methods and processes

- Standards: Component Quality Model

- Services for component providers and component consumers
Component Quality Model

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

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

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

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
Contract levels

- Type

- Functional specification

- Performance specification

- Quality of Service

  (Source: Jézéquel, Mingins et al.)

Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

D: Design

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

E: Extension
Component Quality Model

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?

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

- 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

- Components should be good
- Proofs should be economical!
“Proving classes”

EiffelBase libraries (fundamental data structures and algorithms):

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

Hennessy

- “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”
Ongoing work on proofs

- 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

Related work: components

- 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

- **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

- **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

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

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

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