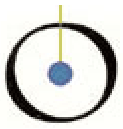
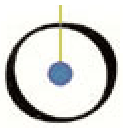


The Grand Challenge of Trusted Components

Bertrand Meyer

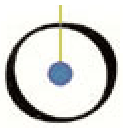
ETH, Zürich
& Eiffel Software, Santa Barbara



For numerous papers and other info

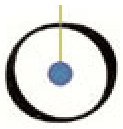
http://www.inf.ethz.ch/~meyer

http://se.inf.ethz.ch



Proposition

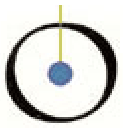
Major progress in software engineering requires switching to the systematic production and use of components of guaranteed quality.



John Hennessy, Stanford

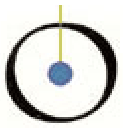
- "Most of the improvement in the reliability of computer systems has come from improvement in the basic components"

- "You'll see ever increasing portions of the effort devoted to design and verification"

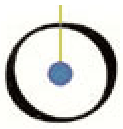


The challenge

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



Software engineering



Software "engineering"

ETH

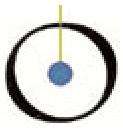
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Professor of
Software Engineering

Prof. Dr. Bertrand Meyer

ETH Zentrum, RZ F1
CH-8092 Zürich

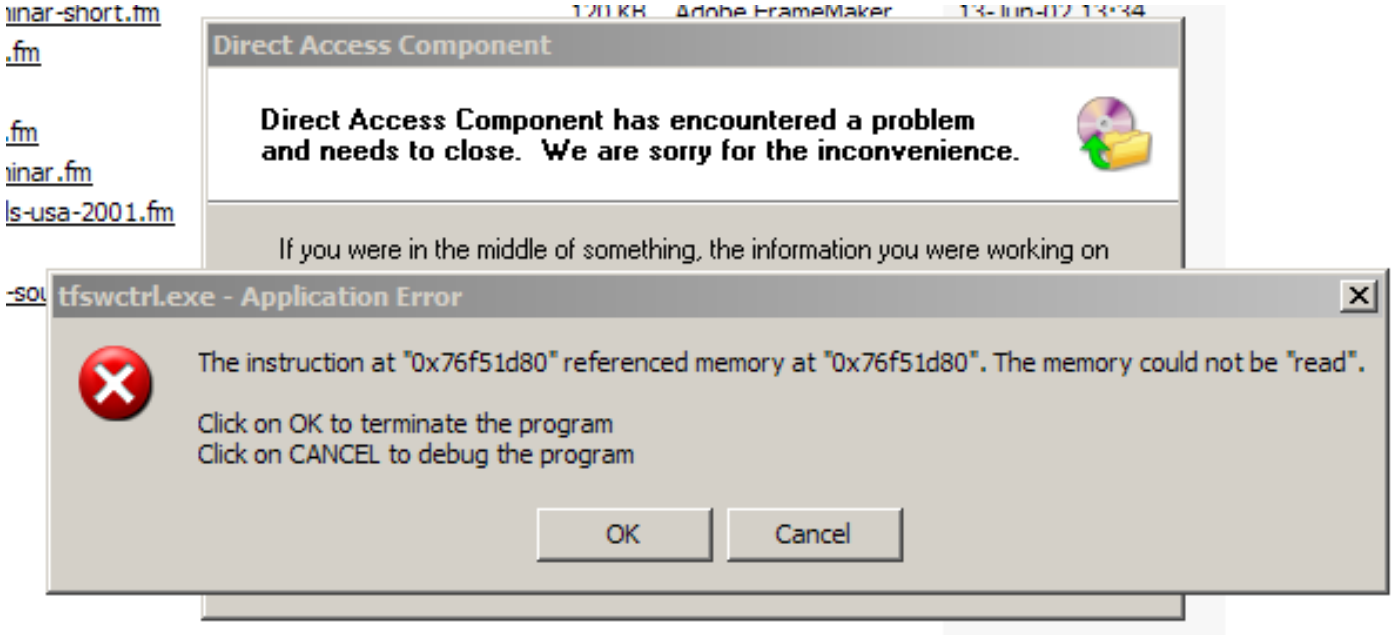
phone +411 632 04 10
fax +411 632 13 07
bertrand.meyer@inf.ethz.ch



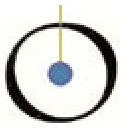
Software "engineering"

- The building of quality software

Coming back from standby...

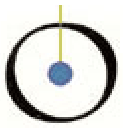


Trying to dial up...



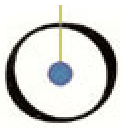
The challenge

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



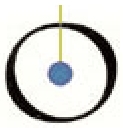
Ways to quality

	Technical	*Management*
A priori	▪Design methods ▪O-O ▪Programming language choice ▪Formal development	▪User involvement ▪Executive support ▪Education (engineers, managers...)
A posteriori	▪White-box testing ▪Static analysis ▪Proofs (of existing programs)	▪Testing, validation, acceptance procedures



Obstacles to achieving top quality

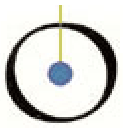
- Industry is not interested
  (not worth the investment)
  (except security)
- Anti-intellectual attitude
  e.g. formal methods
  "Worse is better"
  Fad effects
- Academia is not that interested either
  (hard to publish)



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..."

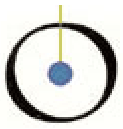


Security

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

But viewpoints are different:

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



Good idea: Process models

CMM, ISO...

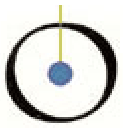
- Good: force a systematic process
- But: concentrate on form, not substance

Standish Group, 1998

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

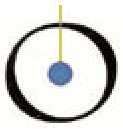
User Involvement	20 Points
Executive Support	15 Points
Clear Business Objectives	15 Points
Experienced Project Manager	15 Points
Small Milestones	10 Points
Firm Basic Requirements	5 Points
Competent Staff	5 Points
Proper Planning	5 Points
Ownership	5 Points
Other	5 Points



Good idea: eXtreme Programming

"Agile" methods, refactoring, test-based development

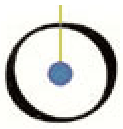
- Good: rehabilitates the act of programming
- But: tests are not specs!



Good idea: Formal methods

B, Abstract State Machines

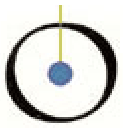
- Good: benefit from mathematics
  (**IF** accompanied with proofs!)
- But: expensive



Good idea: open source

GNU, Linux...

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



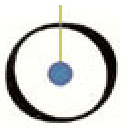
Today's software is often good enough

Overall:

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

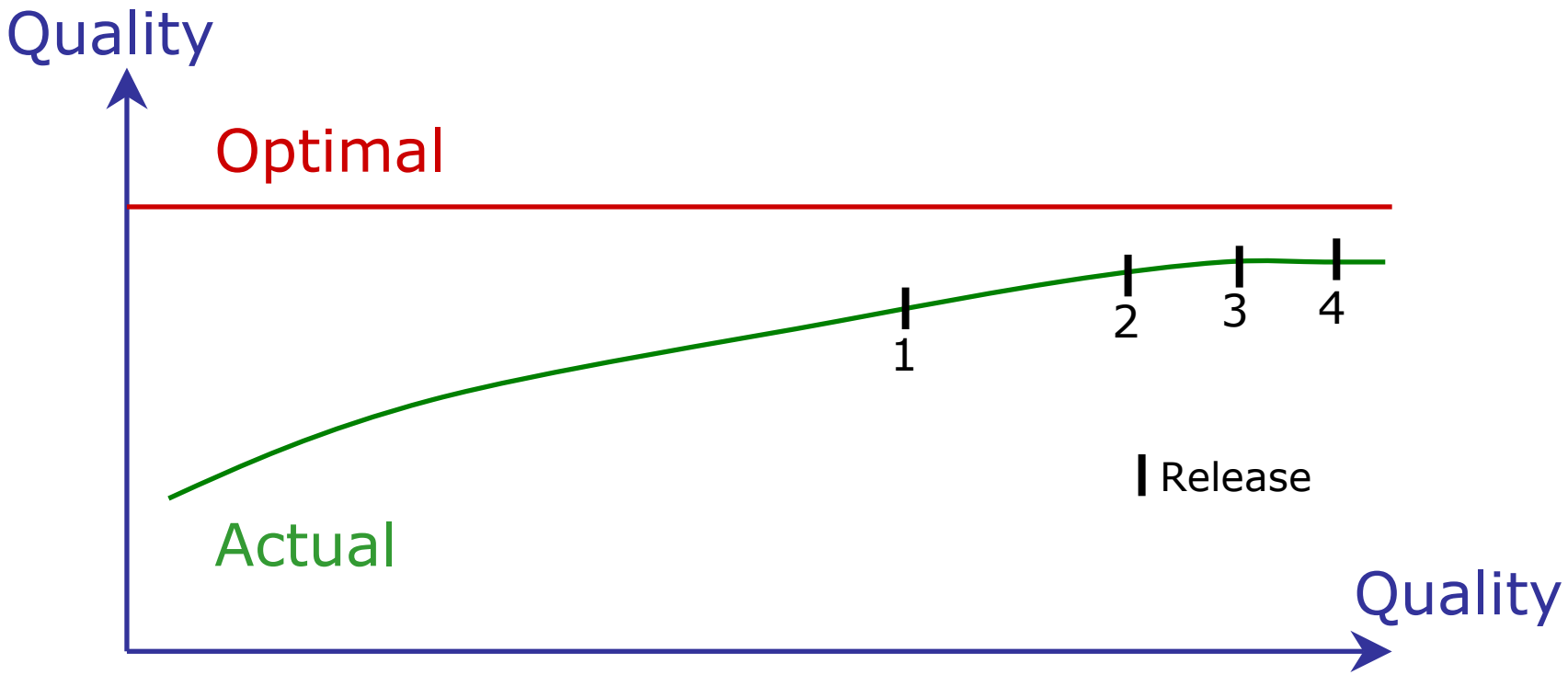
Significant improvements since early ICSEs:

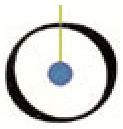
- Better languages
- Better tools
- Better practices (configuration management)



From “good enough” to good?

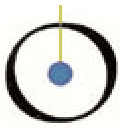
- Beyond “good enough”, quality is economically bad
- He who perfects, dies





The economic argument

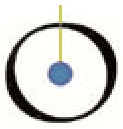
- Stable system:
  - Sum of individual optima = Global optimum
- Non-component-based development:
  - Individual optimum = “Good Enough Software”
  - Improvements: I am responsible!
- Component-based development:
  - Interest of both consumer and producer: Better components
  - Improvements: Producer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



Quality through reuse

- The good news:

  Reuse scales up everything



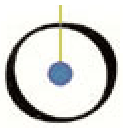
Quality through reuse

- The good news:

  Reuse scales up everything

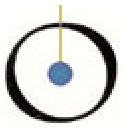
- The bad news:

  Reuse scales up everything



Trusted components

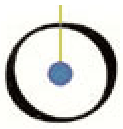
- Confluence of
  - Quality engineering
  - Reuse



Hennessy

- "Most of the improvement in the reliability of computer systems has come from improvement in the basic components"

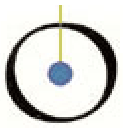
- "You'll see ever increasing portions of the effort devoted to design and verification"



Software design in the future

Component-based for

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

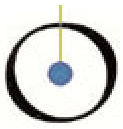


Component quality: the inevitable issue

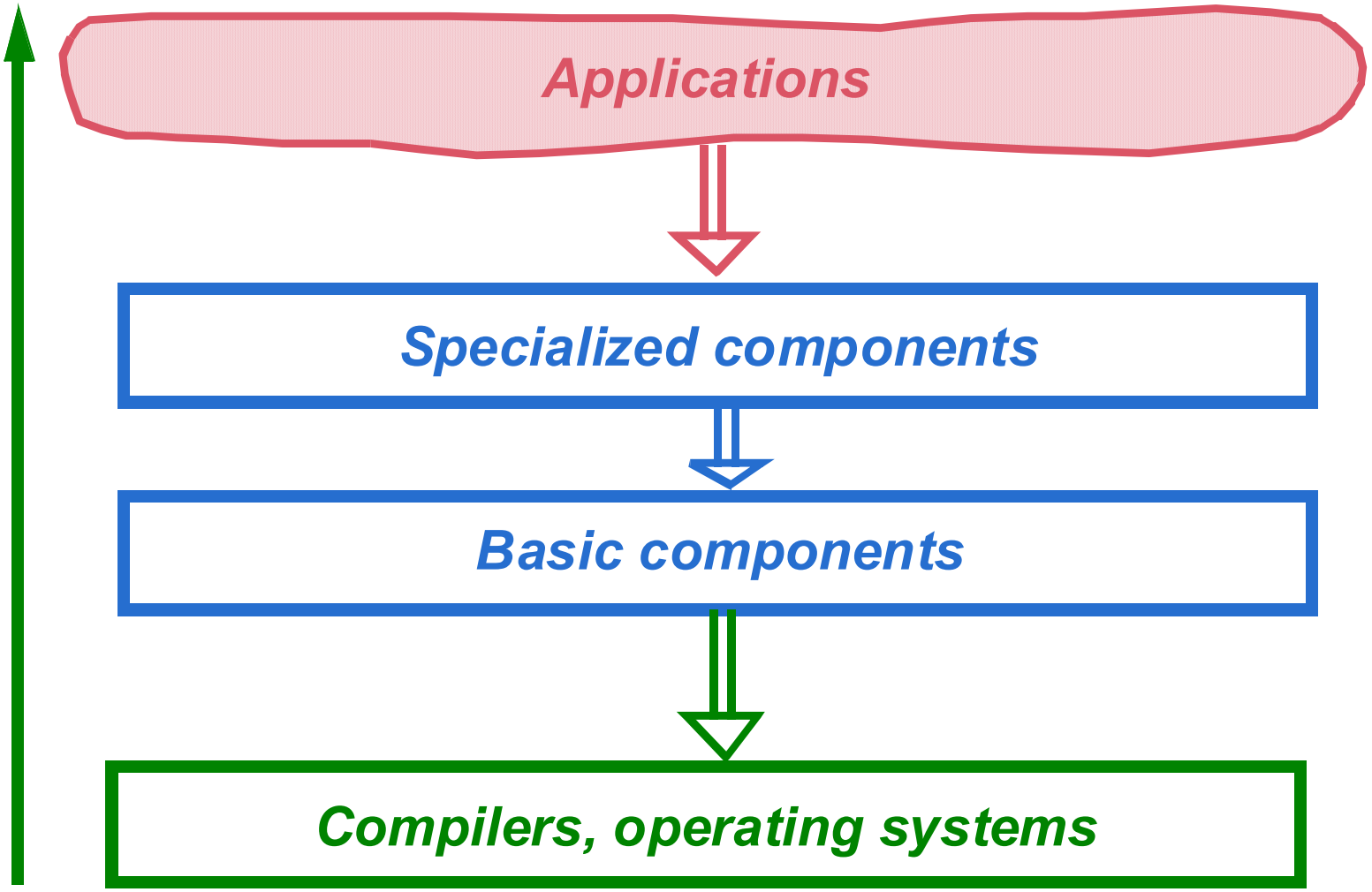
- The key issue
  - Bad-quality components are major risk

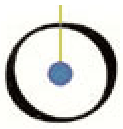
    Deficiencies scale up, too

  - High-quality components could transform the state of the software industry (if it wanted to — currently doesn't)



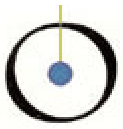
Where to focus effort?





Perfectionism

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

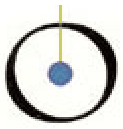


What exactly is a component?

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...

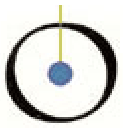
Lifecycle role:
- Analysis
- Design
- Implementation

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

Flexibility:
- Static
- Dynamic
- Replaceable

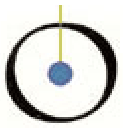
Form of use:
- Interface only
- Source only
- Source + hiding

Economics:
- Free
- Purchased
- Rented



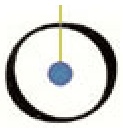
This is a broad view of components

- 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

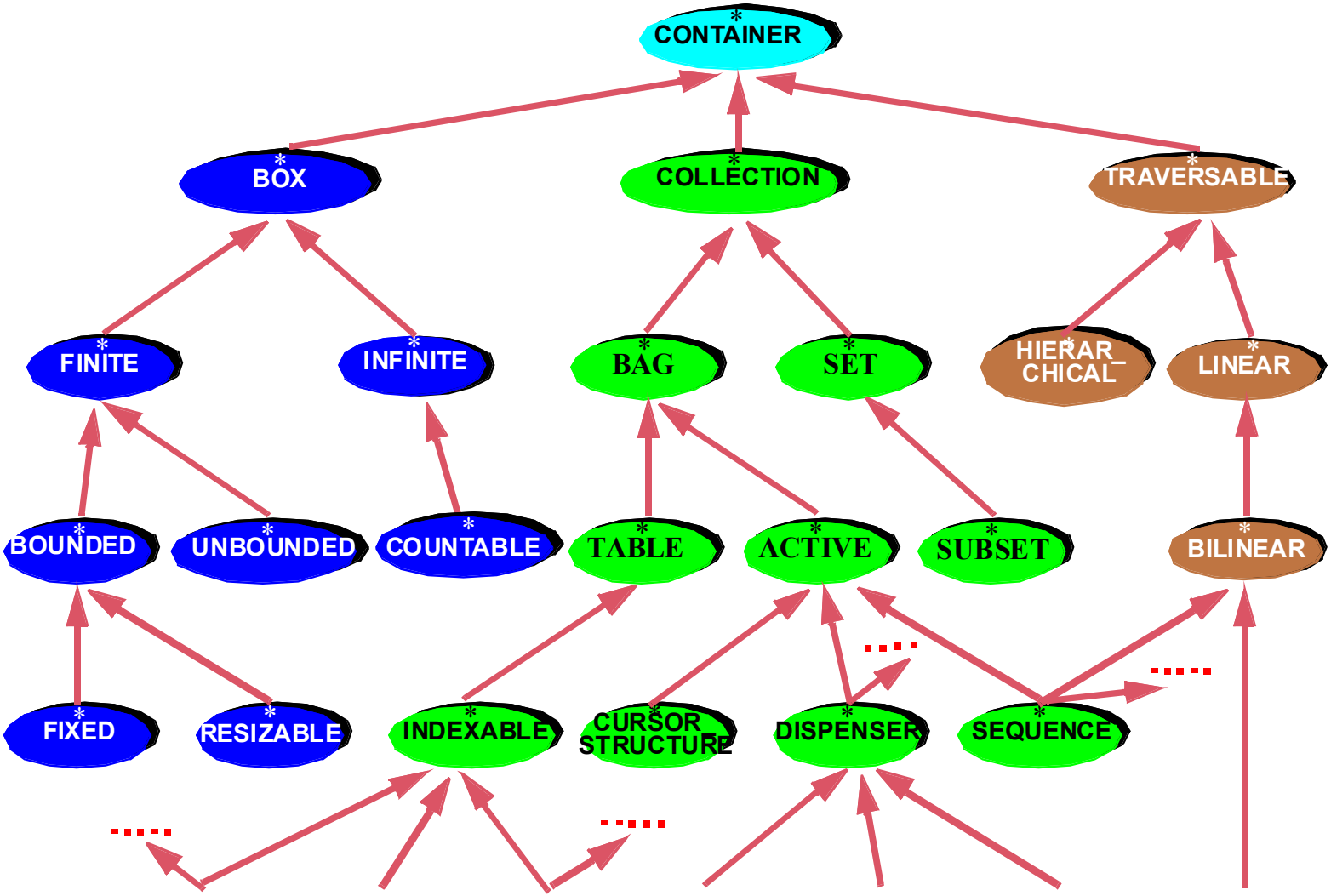
- Patterns are both one of the greatest advances in software engineering, and a step backwards from the push for reuse through object technology
- We should try to turn successful patterns into components!
- Language mechanisms (e.g. multiple inheritance, constrained genericity, agents etc. in Eiffel) make this possible in many cases.
- $\forall\, x \mid x$ considered harmful
- Systematic effort in progress (with Karine Arnout) on Gamma et al. book patterns. See paper on event library on web site.

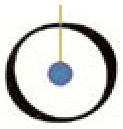


Our experience: Eiffelbase

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

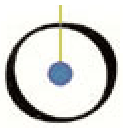
Eiffelbase hierarchy





How to get there

- Low road:
  - Component Certification
- High road:
  - Proofs of correctness



The Low Road: Component Quality Model

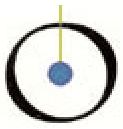
A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension



Component Quality Model

A: Acceptance

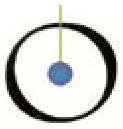
B: Behavior

C: Constraints

D: Design

E: Extension

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



Component Quality Model

A: Acceptance

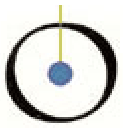
B: Behavior

C: Constraints

D: Design

E: Extension

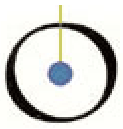
- B.1 Examples
- B.2 Usage documentation
- B.3 Preconditioned
- B.4 Some postconditions
- B.5 Full postconditions
- B.6 Observable invariants



Contract levels

1. Type
2. Functional specification
3. Performance specification
4. Quality of Service

(Source: Jézéquel, Mingins et al.)



Component Quality Model

A: Acceptance

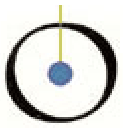
B: Behavior

C: Constraints

D: Design

E: Extension

- 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



Component Quality Model

A: Acceptance

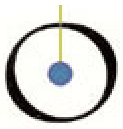
B: Behavior

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



Component Quality Model

A: Acceptance

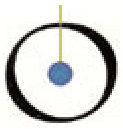
B: Behavior

C: Constraints

D: Design

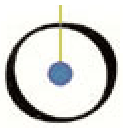
E: Extension

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



A Component Certification Center

- Principles
- Methods and processes
- Standards
- Services for component providers and component consumers



The high road: towards proofs?

A: Acceptance

B: Behavior

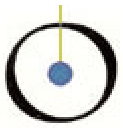
C: Constraints

D: Design

E: Extension

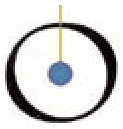
D.1 Precise dependency doc
D.2 Consistent API rules
D.3 Strict design rules
D.4 Extensive test cases

D.5 Some proved properties
D.6 Proofs of preconditions, postconditions & invariants



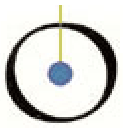
Proof technology and formal methods

- 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!

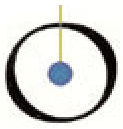


Proofs and O-O: Impedance mismatch

- Refine

or

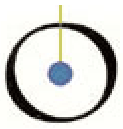
- Prove?



Dangers of "proven components"

- You might be believed!
- One doesn't prove a component

  You may at best be able to prove specific properties of the component
- Do not raise undue expectations

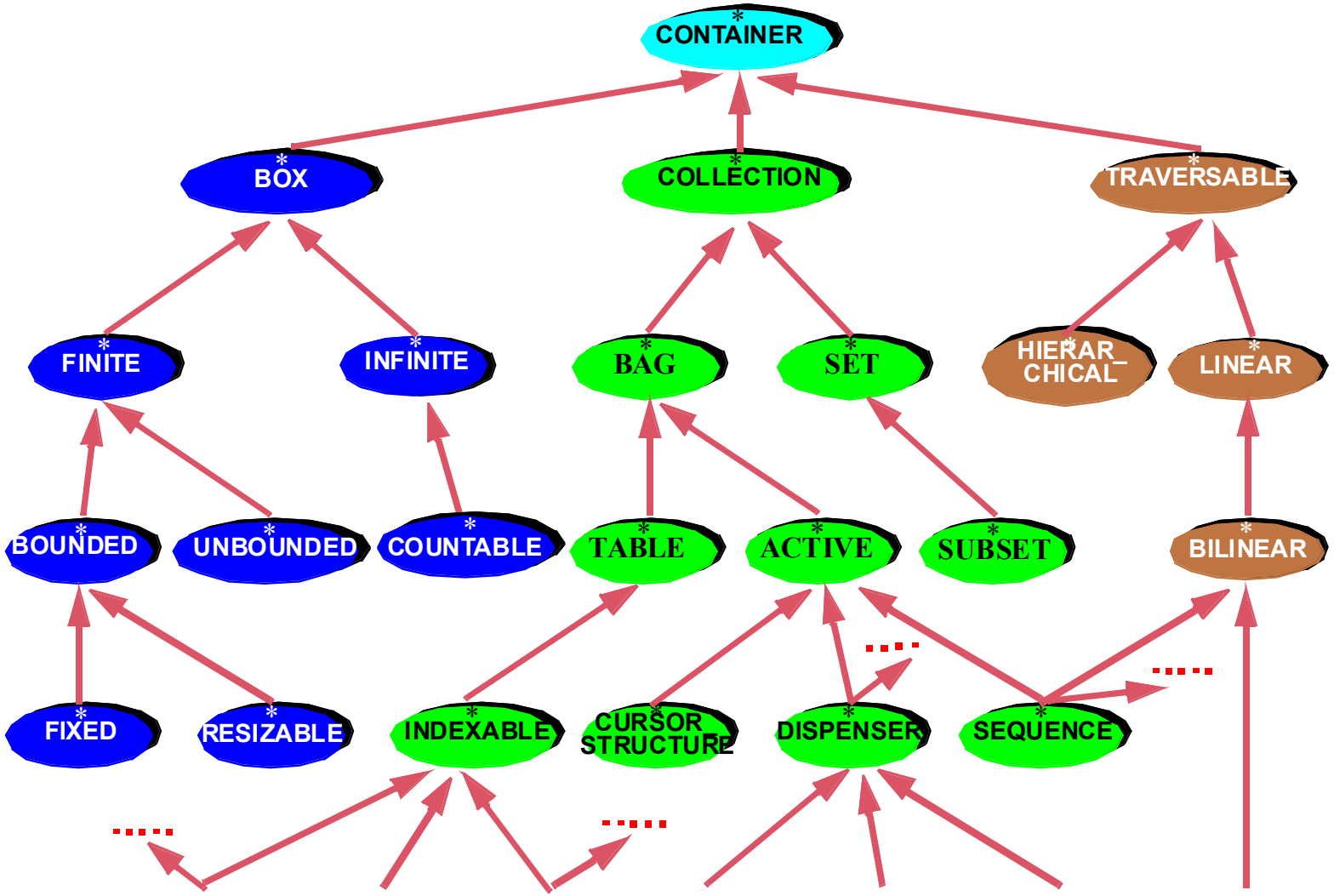


“Proving classes”

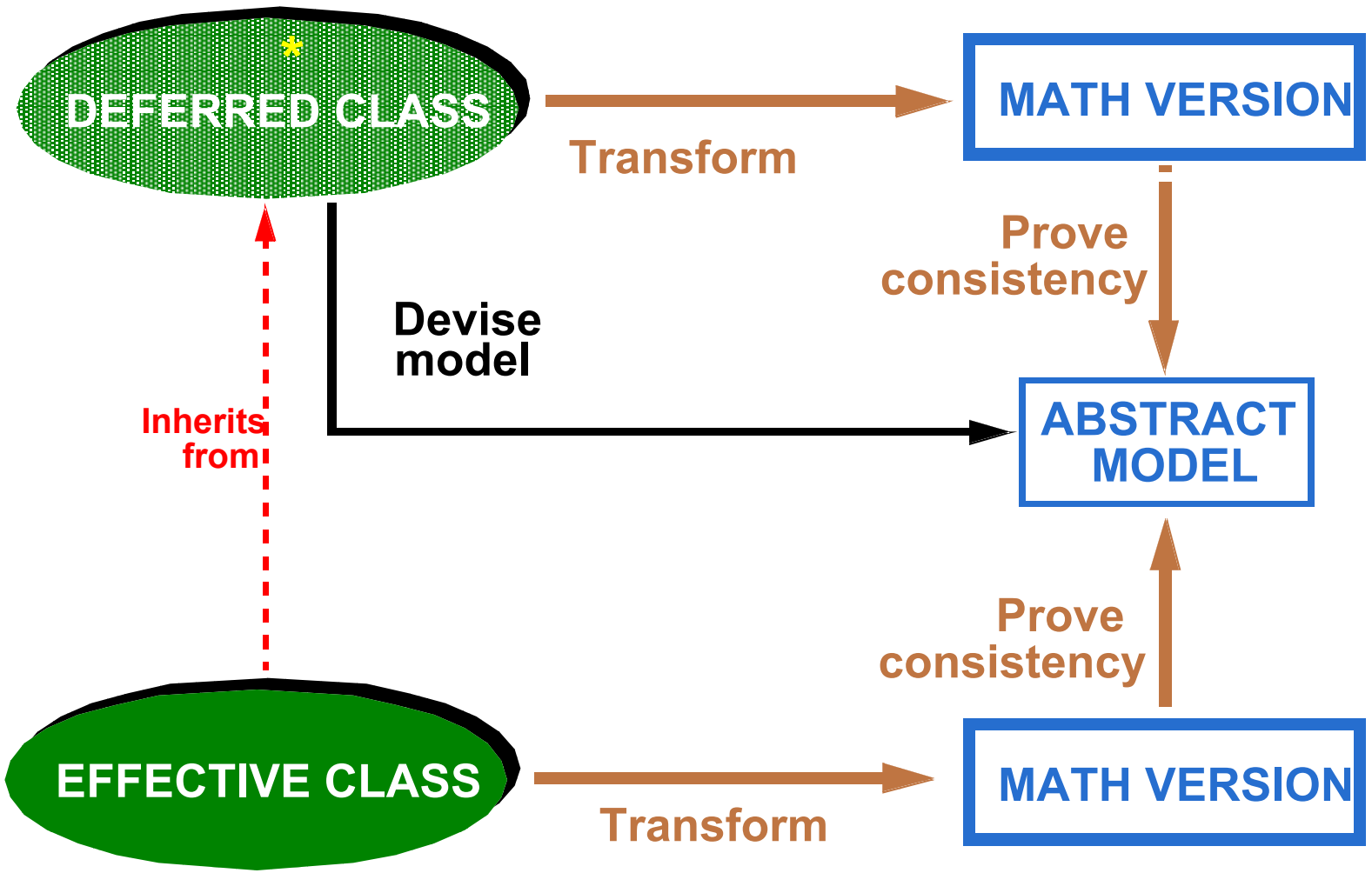
EiffelBase libraries (fundamental data structures and algorithms):

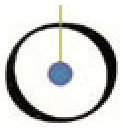
- Classes are equipped with contracts
- “Proving a class” means proving that the implementation satisfies the contracts

Top of the Eiffelbase hierarchy



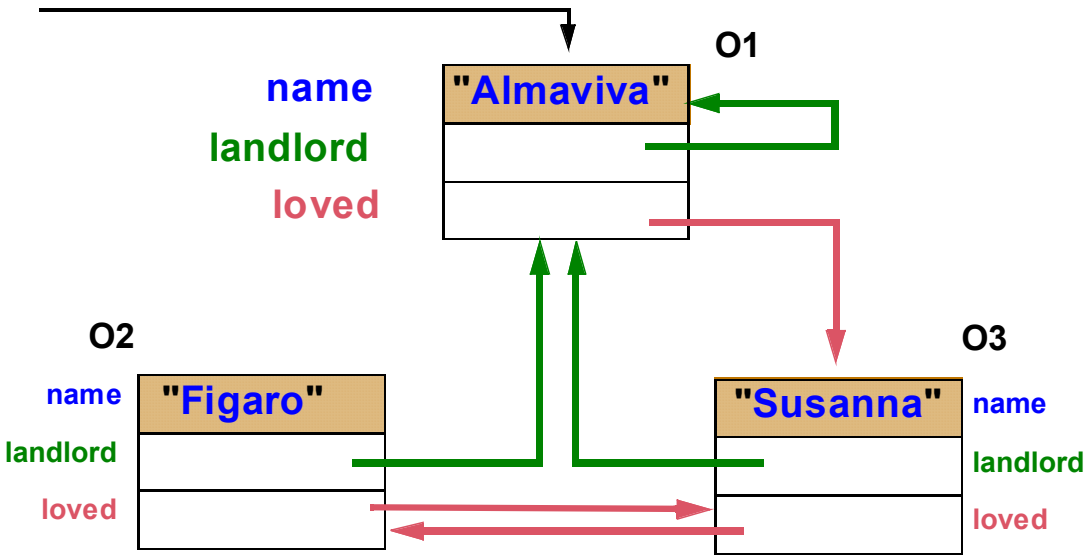
Proof strategy for classes





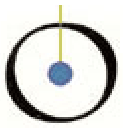
Proof strategy for classes

- The tough part is the object structure, especially pointers



- Object Calculus (based on partial functions and ideas from both axiomatic and denotational semantics)

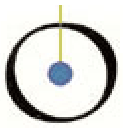
- Then add classes, inheritance, dynamic binding...



Hennessy

- "Most of the improvement in the reliability of computer systems has come from improvement in the basic components"

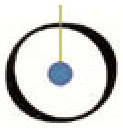
- "You'll see ever increasing portions of the effort devoted to design and verification"



Underlying work

- Semantic theory for full O-O language
- Language development (Eiffel 5)
- Contract extraction

  Cf. Karine Arnout, Bertrand Meyer,
  "Outing Closet Contracts from .NET libraries"
- From patterns to components
- Modeling efforts

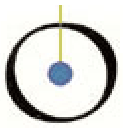


Teaching

"Inverted Curriculum" for introductory programming:

- Use libraries from the start
- Exciting application domain
- Give students heaps of code
- From consumers to producer (outside-in)
- Abstraction: teach, don't preach

There will be a textbook and supporting material



Some of the challenges ahead

General:

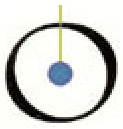
- Convince the software engineering community
- Convince industry (producers, consumers)
- Define ambitious, feasible objectives
- Achieve balance between high and low road

"High road":

- Finish up the theory
- Produce mechanized proofs

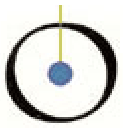
"Low road":

- Define standard terminology
- Get the economics right



Applied research in universities?

- In some areas, one cannot compete with industry
- Applied, tool-oriented work is necessary and possible
- Components are an ideal example

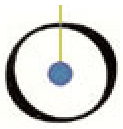


Proposition

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.



For numerous papers and other info

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

http://se.inf.ethz.ch