How you will be programming ten years from now

Bertrand Meyer
Professor of Software Engineering, ETH Zurich
Chief Architect, Eiffel Software, Santa Barbara

ACM Symposium on Applied Computing
Sierre, Switzerland, 23 March 2010

Note on these slides

The slides are a superset (about 10% more) of those presented at the conference.

The original extensively uses animations, so some of the slides will be hard to understand in the PDF version. If you would like to get a copy of the original PowerPoint slides with animation, please contact the author.

Past decades

Software engineering principles have spread widely

Programming methodology:

- Structured programming
- Object-oriented programming
- Design patterns
- Typed languages
  - Extension of type system, e.g. security (Java, .NET), void safety (Spec#, Eiffel)
- More dynamism
- Better tools, IDEs
- Integration with databases and the Web
- Influence of functional programming ideas
- Multithreading is the default, but unsatisfactory
- Security has changed the picture

x.f (…)

Goal of this talk

1. Sketch the picture for a decade from now

2. Present work being done at ETH and Eiffel Software:
    - Language advances
    - Integrated verification
    - Concurrency

Three forms of software development

- 1. Casual

Simple Web sites, spreadsheets, …

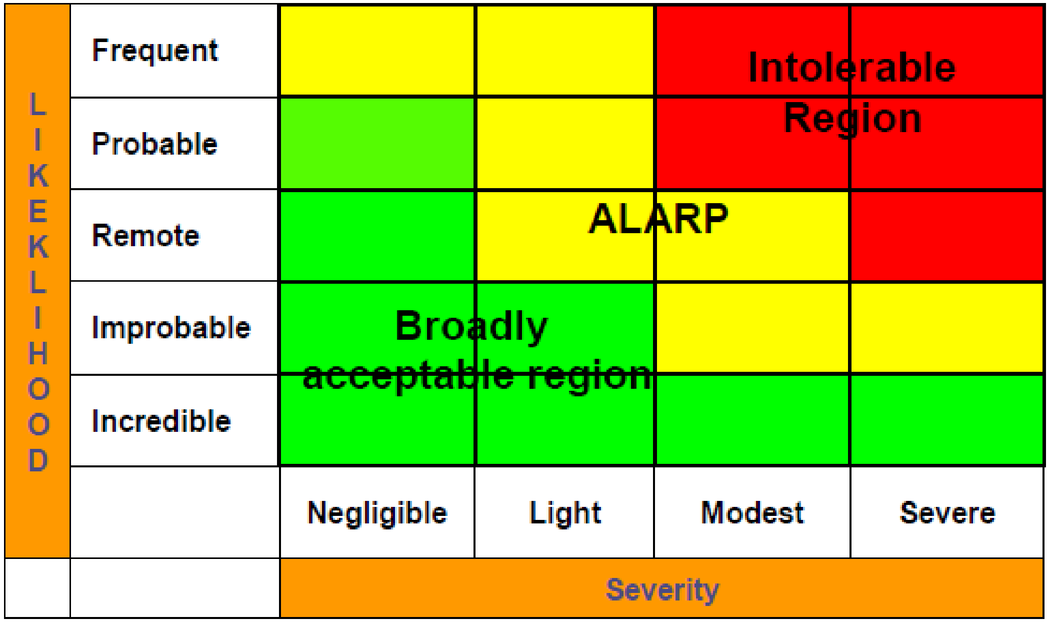
- 2. Professional

Enterprise computing, simulation, “soft” real-time, …

- 3. Critical

Transportation, critical infrastructure, “hard” real-time, …

How the rest of the world views software



ALARP = As Low As Reasonable Practical

Software (IEC 62304):

LIKELIHOOD = 100%

ISO 14971 (medical devices):

Risk = f (*LIKELIHOOD*, *Severity*)

Source: C. Gerber, Stryker Navigation

The standard excuse against progress in SE

(Image removed)

"Programmers will never accept this!"

Three developments that sketch the future

Spark (Praxis, UK)

- Verification technology integral part of development
- No commit without proof
- Downside: 1960-s level language

Spec# (MSR, Redmond)

- Proof assistant
- Verification integrated into software development

Daikon (Michael Ernst, now U. Washington)

- Infer assertions from programs
- Dynamic tool, based on catalog of patterns

Ten years from now

1. We will still be using O-O languages
2. Professional programming will be far more rigorous
3. Verification will be integrated in the development process
4. Every program will have a Web interface
5. Concurrency will be ubiquitous
6. More reliance on objective assessment
7. Software ***engineering***: not just process but technology

Professional programmers:

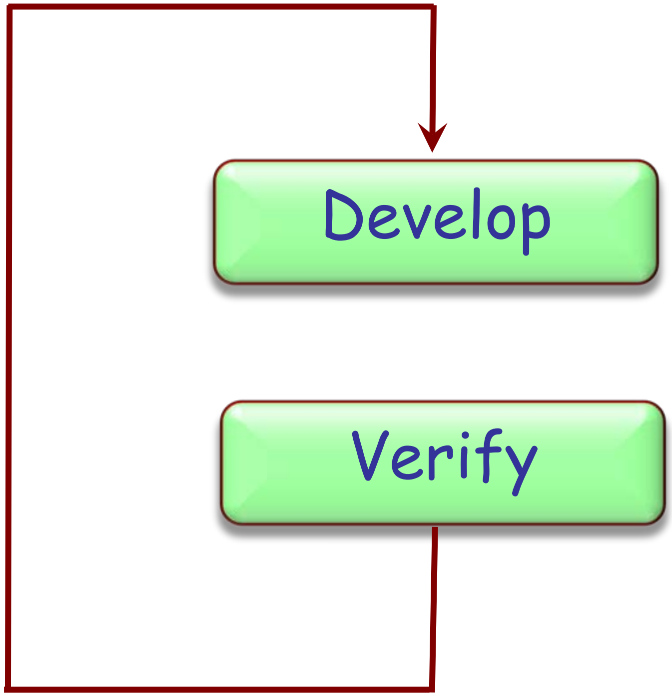
- Higher level of qualification
- Apply verification techniques routinely
- Not mathematics PhDs

Verified software

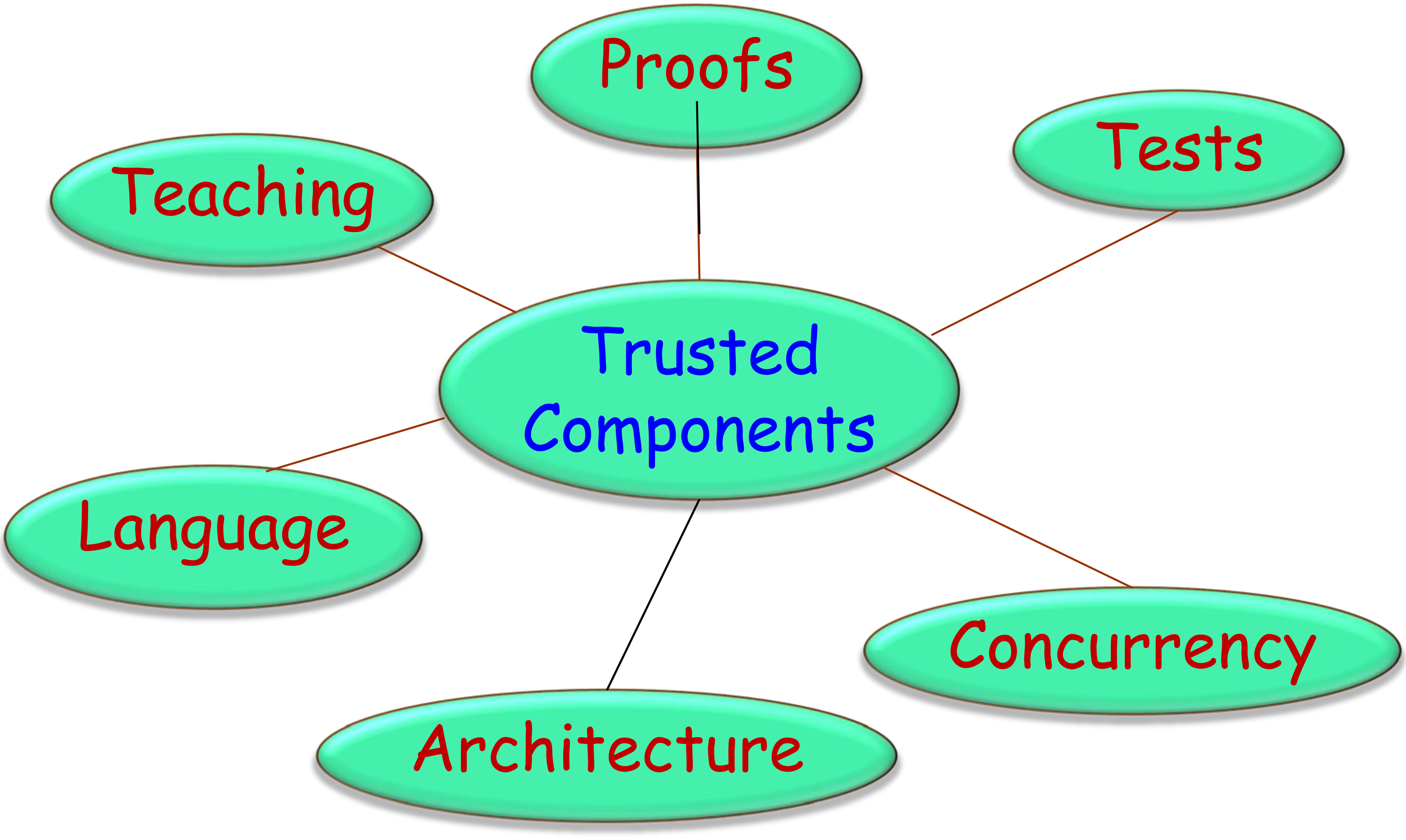
(See Tony Hoare's "Grand Challenge")

Two traditional approaches:

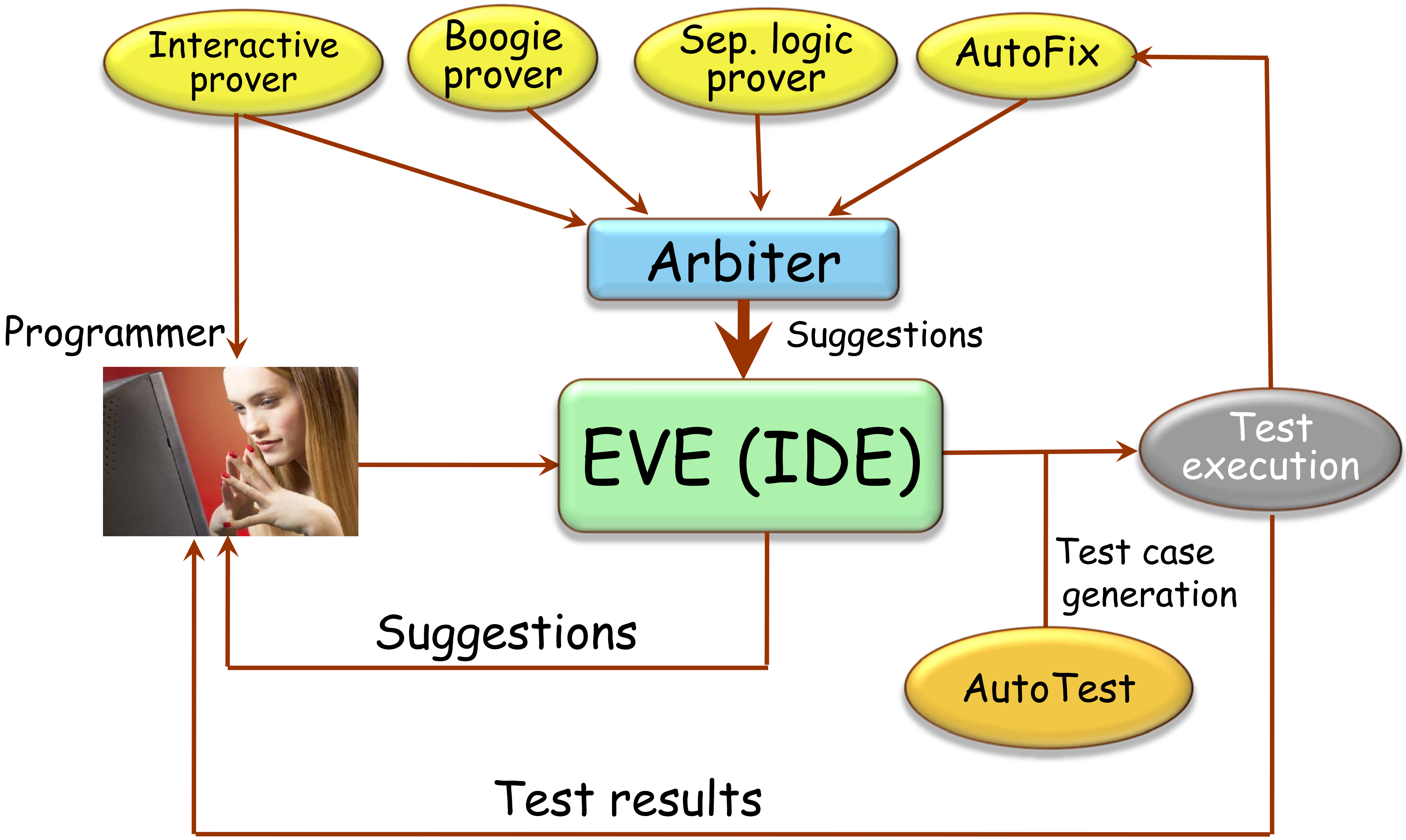
- Correctness by construction
- Verify a posteriori



Verification: our vision



The verification assistant



Verified software: the obstacles

1. Intellectual and managerial resistance
2. Poor integration of formal verification tools in normal development cycle and tools
3. Diversity and incompatibility of verification technologies
4. Split between tests and proofs
5. Remaining technical problems for modern programming language constructs:
    - Frame problem
    - Aliasing
    - Function objects (closures, agents, delegates…)
    - Exception handling
    - Devising loop invariants
    - Full specifications

Our language context: Eiffel

Modern object-oriented language

Constant refinement and evolution since 1985

ISO standard (2006, to be revised 2010), ECMA committee

Industrial usage, very large mission-critical applications

Applicable to teaching
(introductory programming at ETH since 2003)

Community large enough to matter, small enough to permit evolution

Eiffel: Features that help

Design by Contract mechanisms: built-in

Combination of genericity and inheritance

Multiple inheritance

Agents (high-level function objects)

Uniform access (no difference between functions & attributes)

Sophisticated type system (covariance)

Void safety: no more null pointer dereferencing

x.f (…)

Concurrency extension: SCOOP

Void safety: B. Meyer, E., Stapf, A. Kogtenkov, *Avoid a Void*, see http://bertrandmeyer.com/tag/void-safety/ (to appear in Hoare anniv. volume, 2010; earlier version of mechanism in ECOOP 2005)

Our research context: EVE

Eiffel Verification Environment

Open source

Developed at ETH, others' contributions welcome

6-month release schedule, following EiffelStudio

The verification assistant

Tools run *automatically* in *background*

Automatic verification using proofs and tests

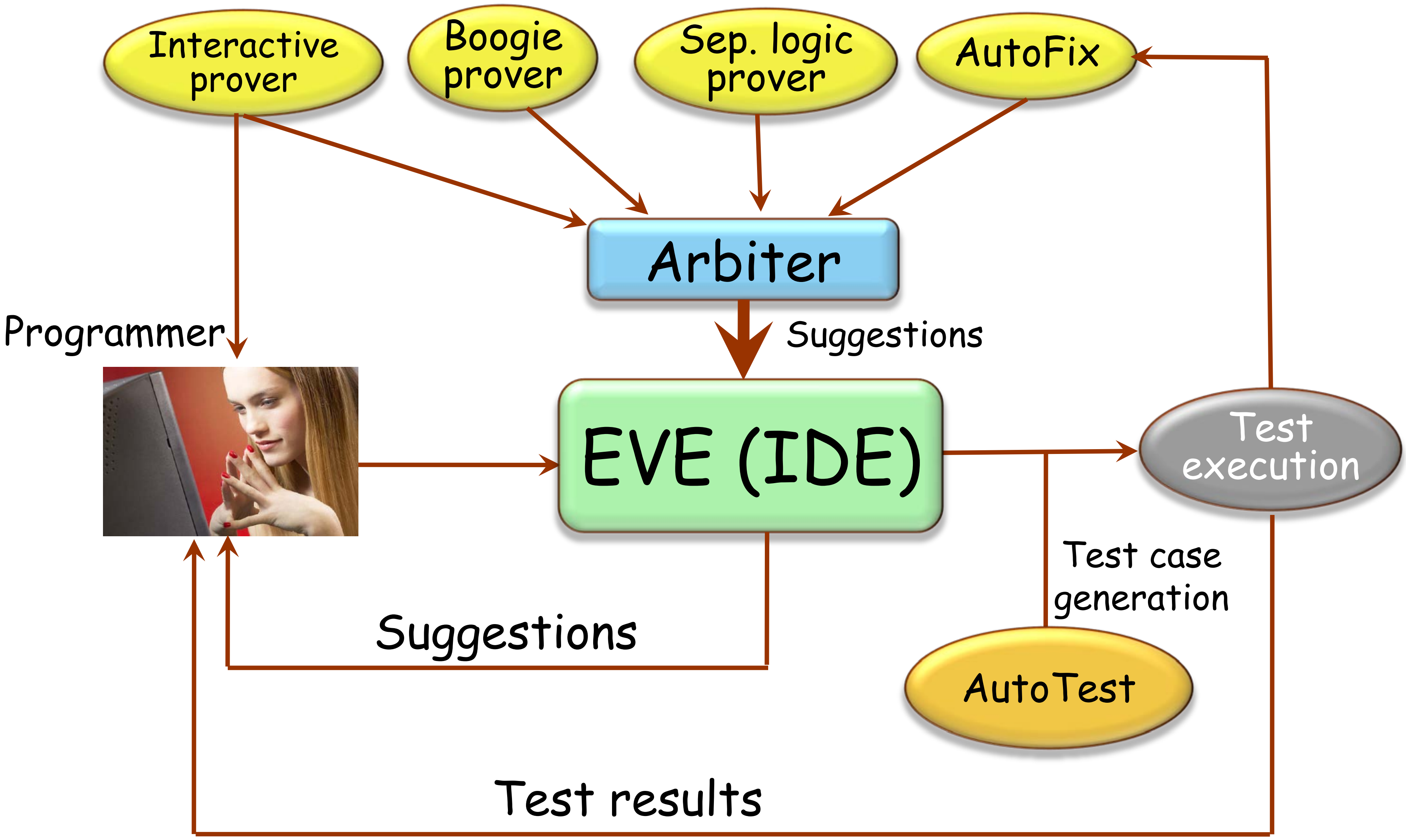
- Boogie, jStar, AutoTest

Automatic inference of code and contracts

- AutoFix, CITADEL, gin-pink

Present useful information to user

The verification assistant



The verification assistant: example use

Programmer writes code

```
class ACCOUNT feature
  balance: INTEGER
  deposit (amount: INTEGER)
    require
      amount > 0
    do
      balance := balance – amount
    ensure
      balance = old balance + amount
    end
end
```

This slide and the next four are by Julian Tschannen

Example use

Proof tool finds error & extracts counter-example

```
class ACCOUNT feature
  balance: INTEGER
  deposit (amount: INTEGER)
    require
      amount > 0
    do
      balance := balance — amount
    ensure
      balance = old balance + amount
    end
end
```

Postcondition does not hold if:

{balance = 0, amount = 1}

Example use

Test tool takes counter-example & generates failing test case

```
class ACCOUNT feature
  balance: INTEGER
  deposit (amount: INTEGER)
    require
      amount > 0
    do
      balance := balance — amount
    ensure
      balance = old balance + amount
    end
end
```

The program will fail if you run this:

```
test_1
  do
    create account
    account.deposit (1)
  end
```

Example use

AutoFix takes failing test case and generates a possible correction

```
class ACCOUNT feature
  balance: INTEGER
  deposit (amount: INTEGER)
    require
      amount > 0
    do
      balance := balance – amount
    ensure
      balance = old balance + amount
    end
end
```

Maybe you should change this to:

```
deposit (amount: INTEGER)
  do
    balance := balance + amount
  end
```

Example use

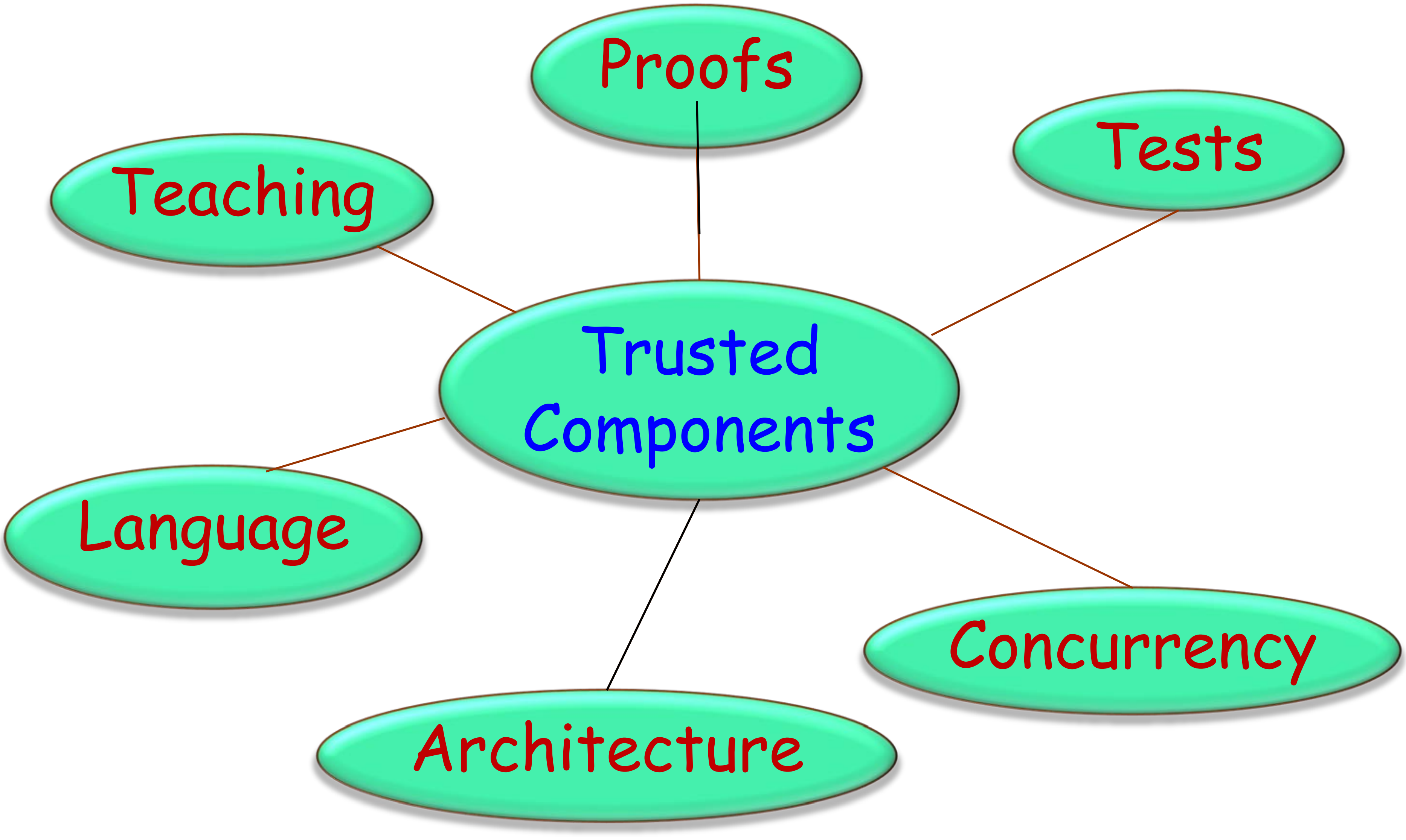
AutoTest & proof tool check fix to obtain a verified correction

```
class ACCOUNT feature
  balance: INTEGER
  deposit (amount: INTEGER)
    require
      amount > 0
    do
      balance := balance — amount
    ensure
      balance = old balance + amount
    end
end
```

Trying the fix now…
test_1 passes
deposit verified

The proposed fix works!
Do you want to use it?

Verification: our vision



Example language advance: void safety

Goal: remove null-pointer (void-reference) derefencing

Fully operational since 2009

See reference on slide 15

Basic idea: x.f (…) valid only if

x is "attached"

x.f (…)

1. Can be proved attached from context, e.g.

   if x /= **Void then** x.f (…) **end** (if x is a local variable)

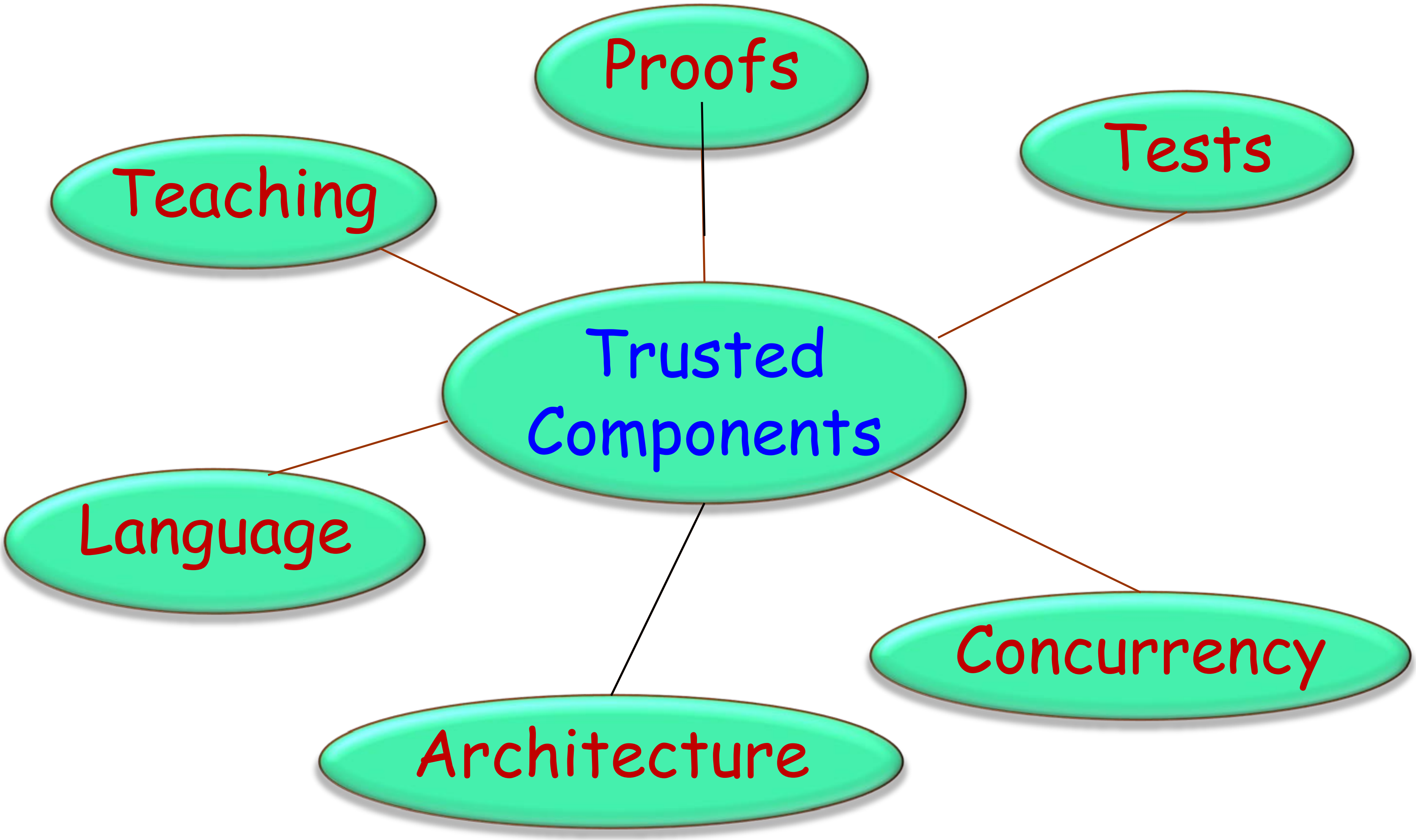
2. Use an attached type:

   x: PERSON -- As opposed to: x: **detachable** PERSON)
   (major problem: initialization of attached variables)

3. Use "object test":

   if attached exp **as** x **then** x.f (…) ; … **end**

Verification: our vision



AutoTest

Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Alexander Pretschner, Yi Wei, Arno Fiva, Emmanuel Stapf

Automatic testing:

- Automatic test case generation
- Automated testing process
- Test extraction from failure
- Regression testing

Initially research projects; main results now integrated in the standard EiffelStudio delivery.

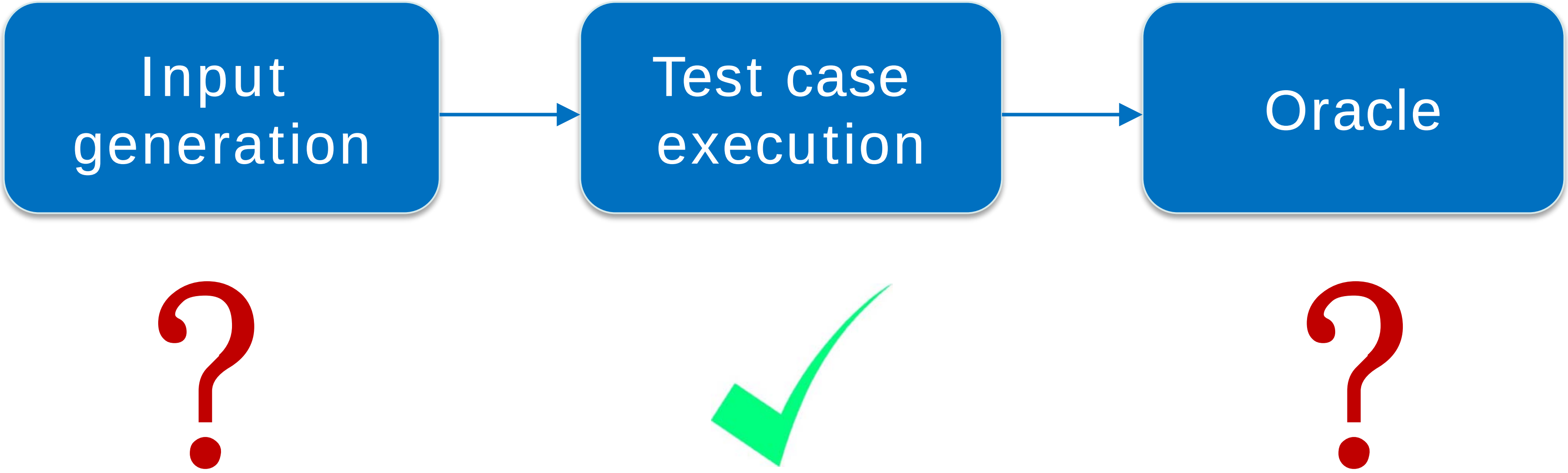
“Automated testing”

B. Meyer et al., *Programs that test themselves*, IEEE Computer, Sept. 2009, http://tinyurl.com/ybbsn2r

What can be automated:

- Test suite execution
- Resilience (continue test process after failure)
- Regression testing
- Test case generation
- Test result verification (*oracles*)
- Test extraction from failures
- Test case minimization

AutoTest: programs that test themselves



Contracts for testing

Contracts provide the right basis:

- A fault is a discrepancy between intent and reality
- Contracts describe intent

A contract violation always signals a fault:

- Precondition: in client
- Postcondition or invariant: in routine (supplier)

In EiffelStudio: select compilation option for contract monitoring at level of class, cluster or system.

30

Contract-based testing

Precondition & class invariant
Routine
Postcondition & class invariant

```
deposit (v: INTEGER )
    require
        v > 0                                  Input filter
    do
        balance := balance - v
    ensure
        balance = old balance + v              Oracle
    end

invariant     -- At class level
    balance >= 0
```

AutoTest: Test generation

Ilinca Ciupa
Andreas Leitner
Yi Wei
Manuel Oriol...

- Input: set of classes + testing time
- Generates instances, calls routines with automatically selected arguments
- Oracles are contracts:
  - Direct precondition violation: skip
  - Postcondition/invariant violation: bingo!
- Value selection: Random+ (use special values such as 0, +/-1, max and min)
- Add manual tests if desired
- Any test (manual or automated) that fails becomes part of the test suite

Minimization: an example

```
create {STRING} v1
v1.wipe_out
v1.append_character ('c')
v1.append_double (2.45)
create {STRING} v2
v1.append_string (v2)
v2.fill ('g', 254343)
...
create {ACCOUNT} v3.make (v2)
v3.deposit (15)
v3.deposit (100)
v3.deposit (-8901)
...
```

```
class
   ACCOUNT
create
   make
feature
   make (n: STRING)
      require
         n /= Void
      do
         name := n
         balance := 0
      ensure
         name = n
         balance = 0
   end
```

```
   name : STRING
   balance : INTEGER
   deposit (v : INTEGER)
      do
         balance := balance + v
      ensure
         balance =
            old balance + v
      end
invariant
      name /= Void
      balance >= 0
end
```

AutoTest (in EiffelStudio)

Demo

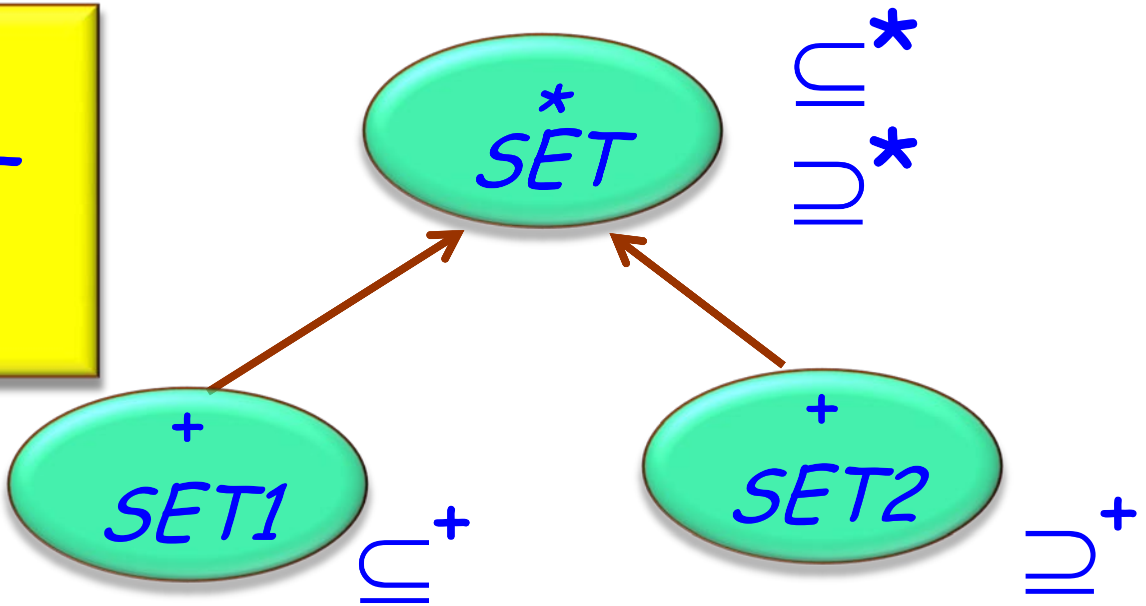
Random testing: example bug found

Bernd Schoeller

Test:

$s1, s2 : SET$

$s2 \subseteq s1$



*: Deferred

+: Effective

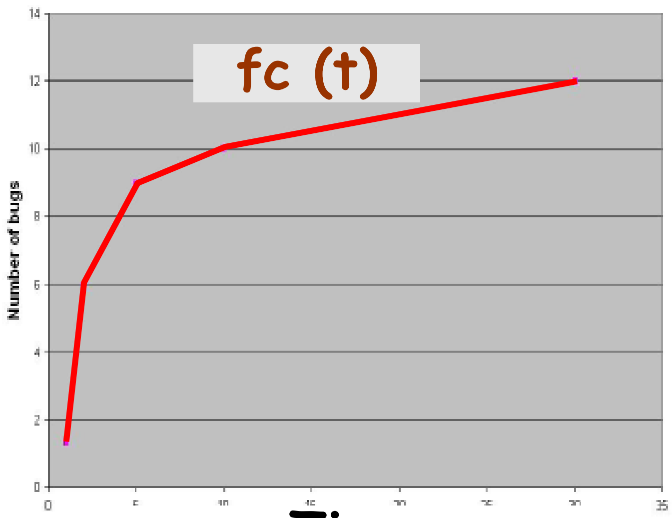
Test generation results

Library	TESTS Total	TESTS Failed	ROUTINES Total	ROUTINES Failed
EiffelBase	40,000	3%	2000	6%
Gobo Math	1500	1%	140	6%

36

Test generation results and strategy

Ilinca Ciupa, Yi Wei



Define good assessment criteria:

- Number of faults found
- Time to find all faults

Experimental law:

$$fc(t) = a - b / t$$

Who finds what faults?

I.Ciupa, A. Leitner, M.Oriol, A. Pretschner

On a small EiffelBase subset, we compared:

- AutoTest
- Manual testing (students) (3 classes, 2 with bugs seeded)
- User reports from the field

AutoTest: 62% specification, 38% implementation

User reports: 36% specification, 64% implementation

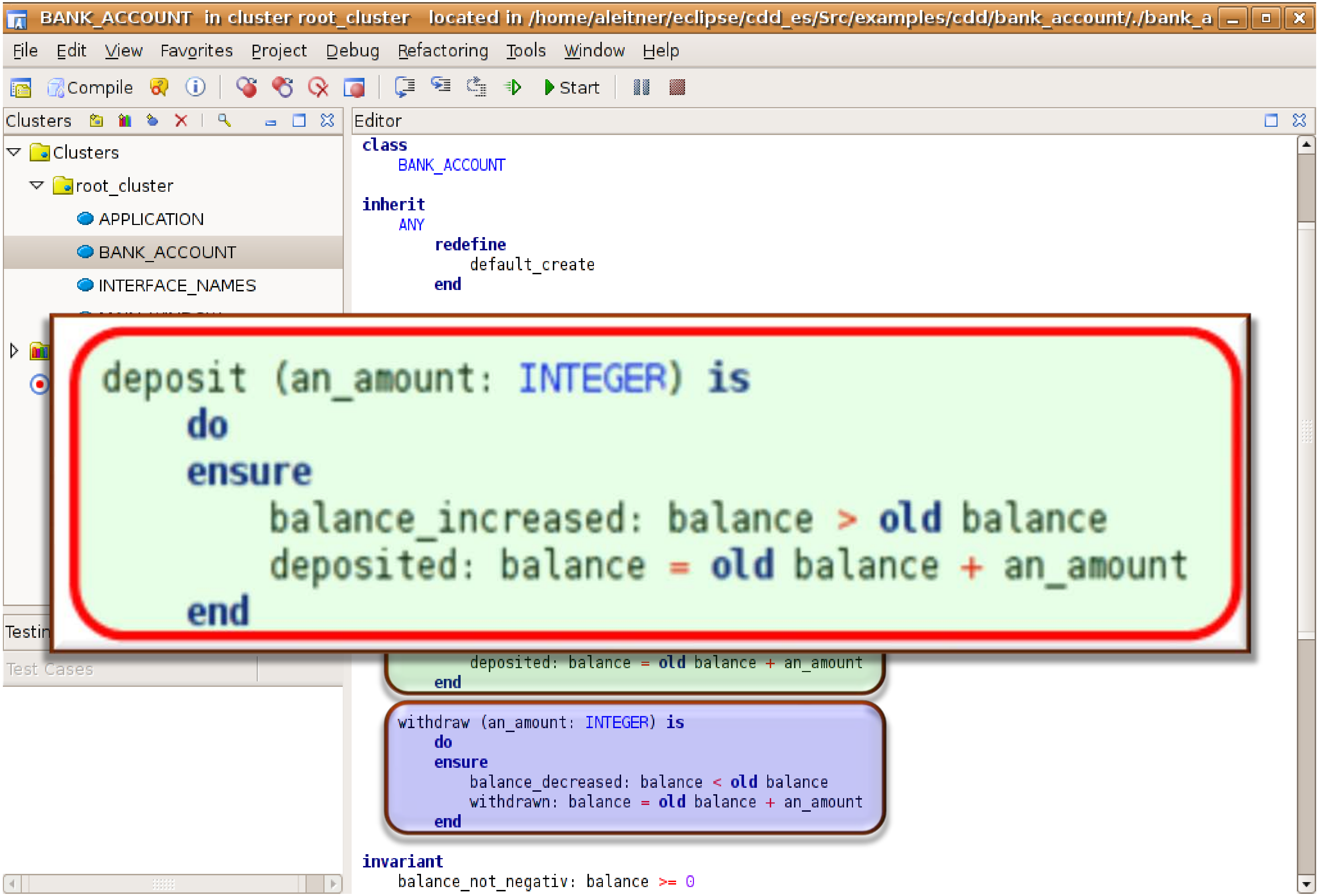
Test extraction

Andreas Leitner, Arno Fiva

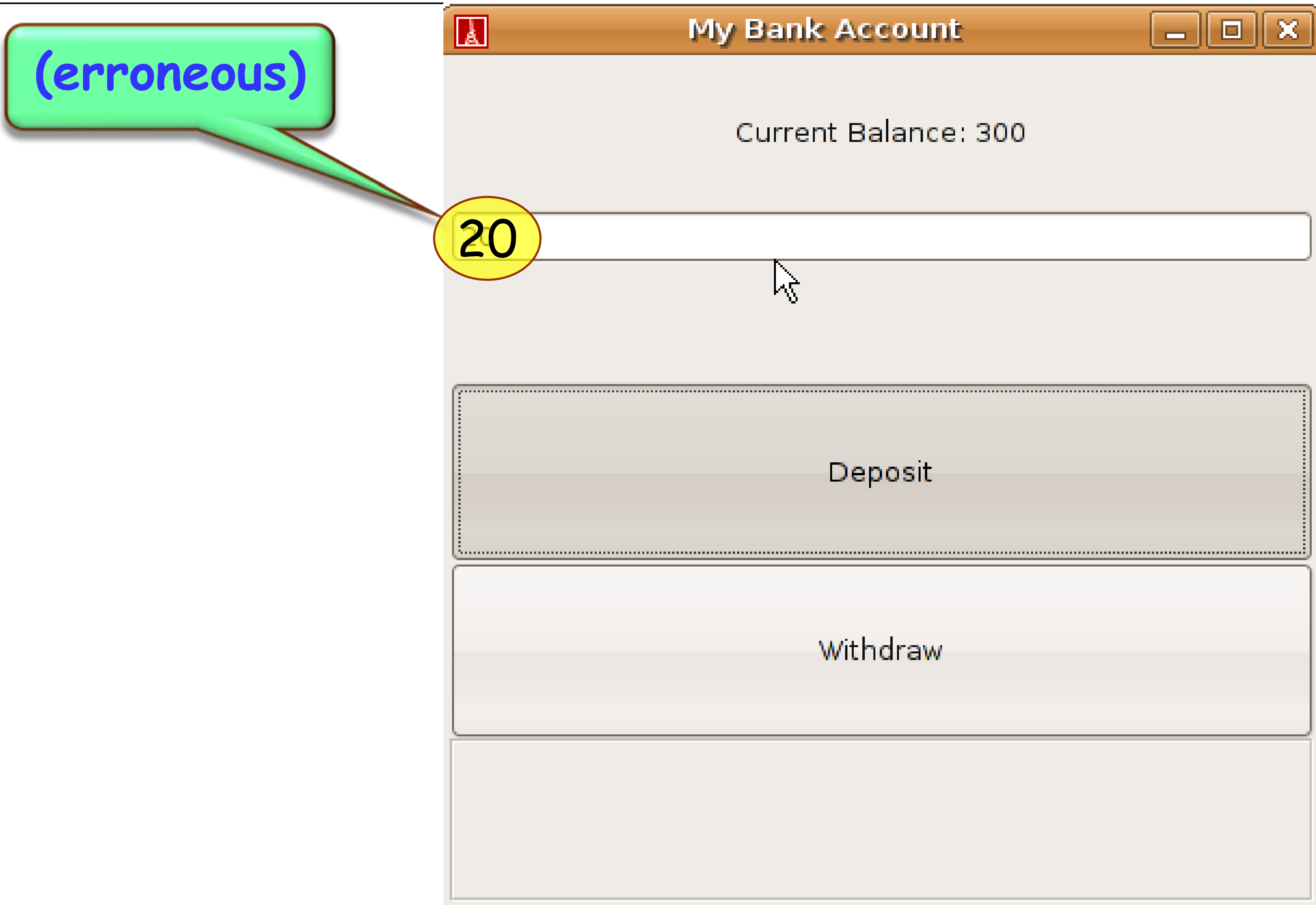
Record every failed execution, make it reproducible by retaining objects

Turn it into a regression test

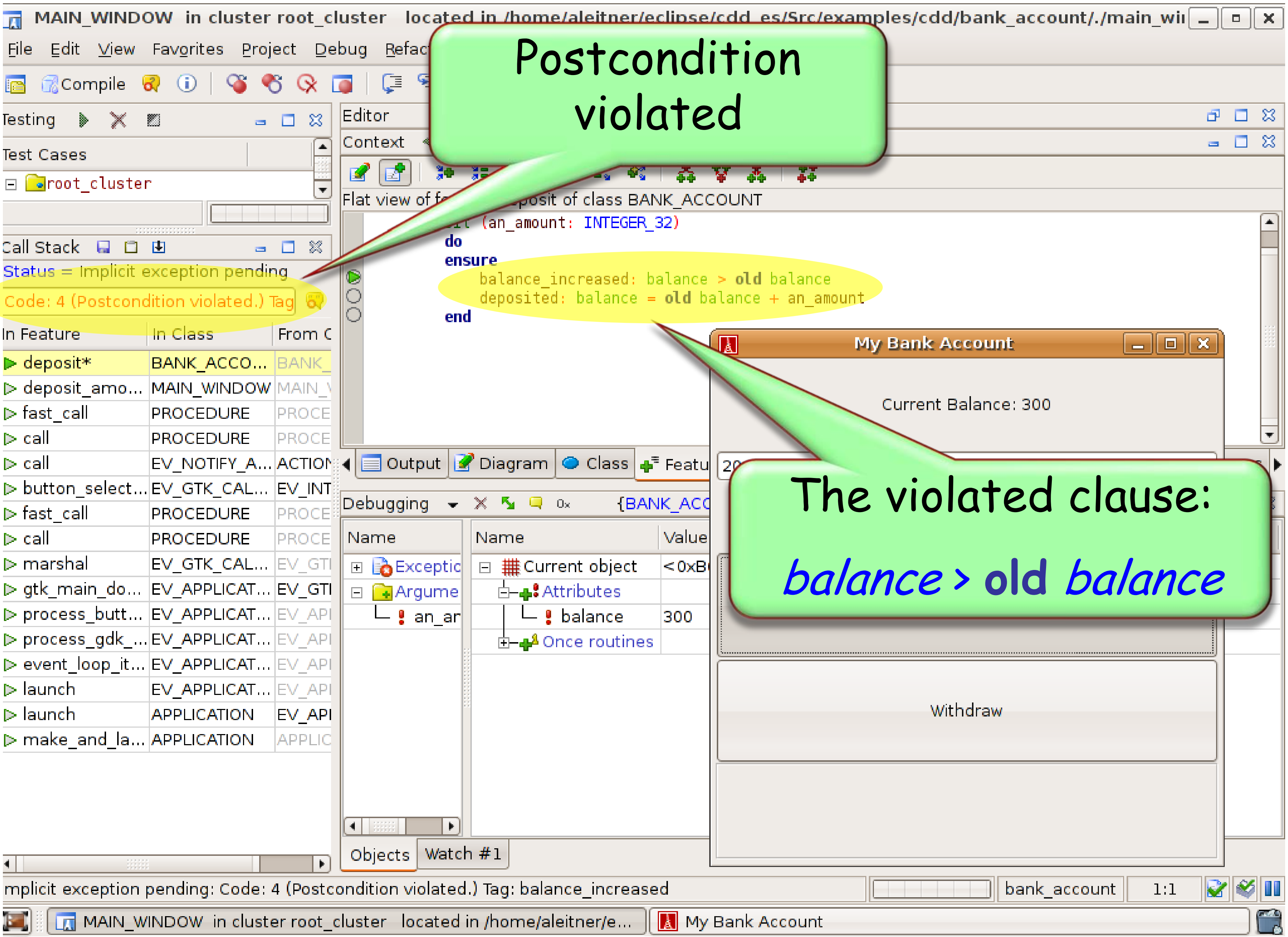
Specified but unimplemented routine



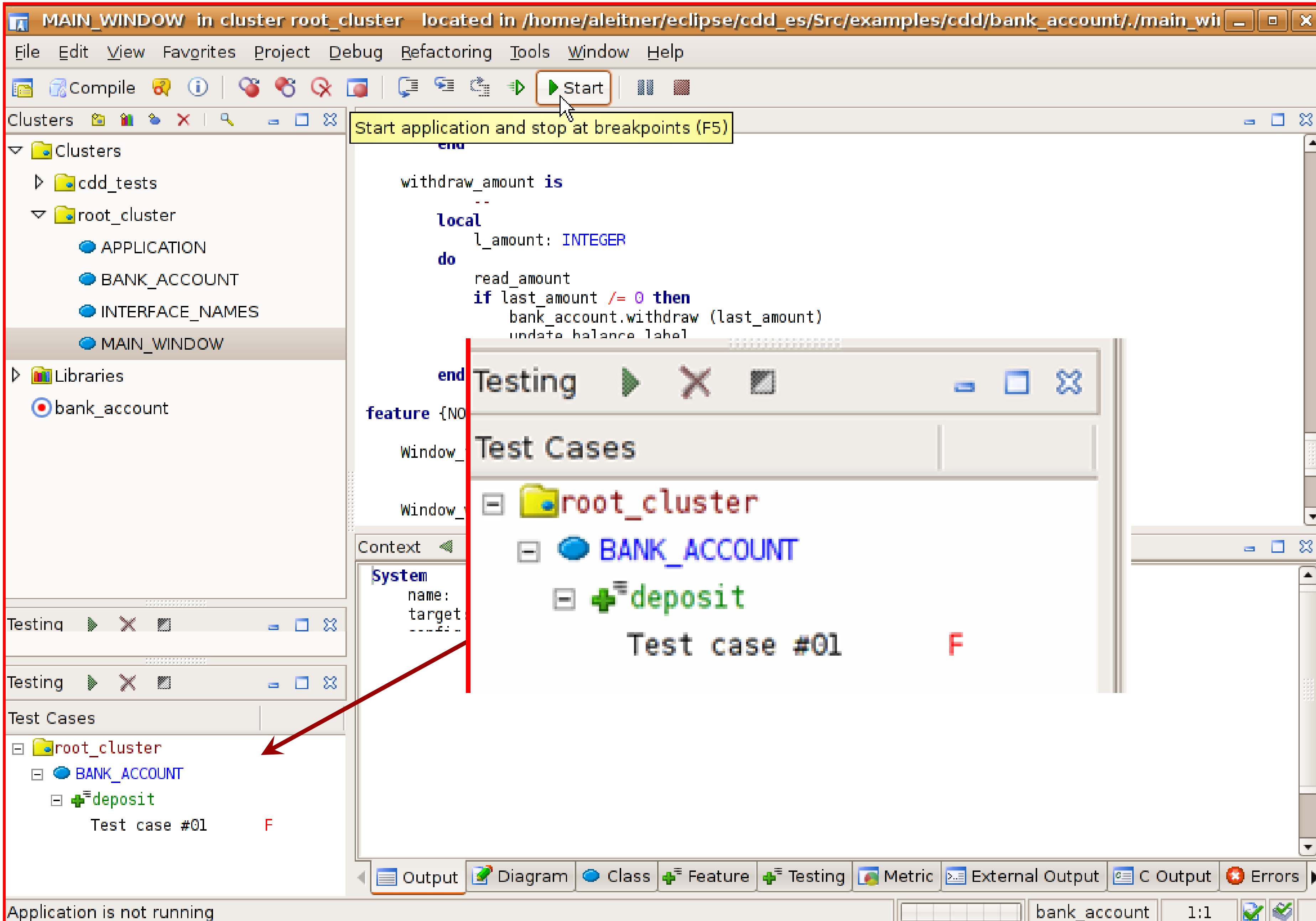
Running the system and entering input



Error caught at run time as contract violation



This has become a test case



CITADEL: Daikon for Eiffel

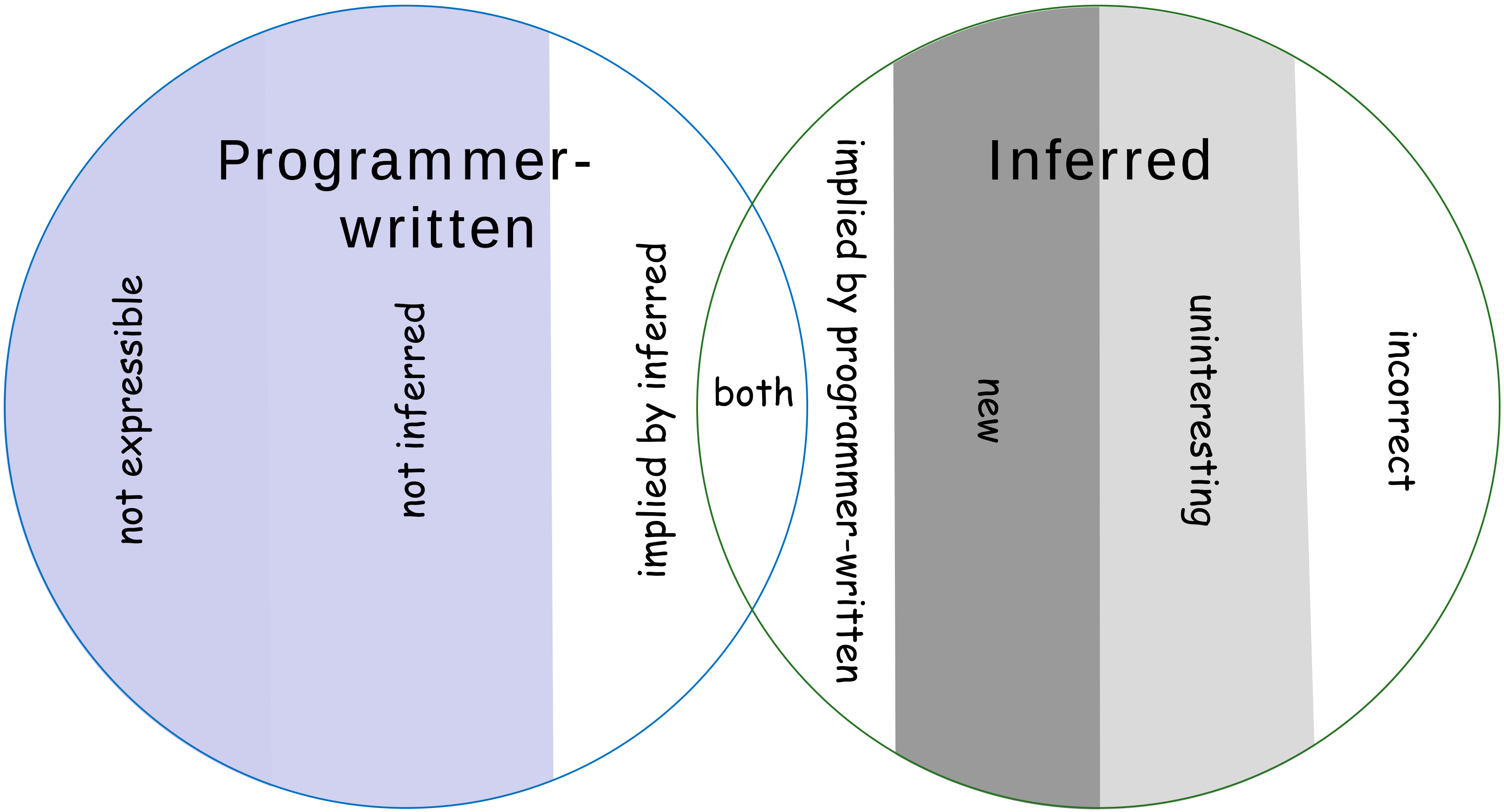
Nadia Polikarpova

How good are automatically inferred contracts?

How do programmer-written and inferred contracts compare?

How can contract inference be used to improve the quality of contracts in a language with Design by Contract support?

Classification



This slide and the next 4 are by Nadia Polikarpova

Observations

Daikon is good at inferring simple contract clauses: *97%* of relevant inferred clauses have the form

$$x \, \mathrm{R} \, y$$

- x is a variable
- y is a variable or a constant
- R is one of $=, \neq, <, \leq, >, \geq$

Reasons for inexpressible programmer-written contract clauses:

- *54%* due to calling a function with one argument
- *19%* due to implications

Observations

What kinds of clauses do programmers miss?

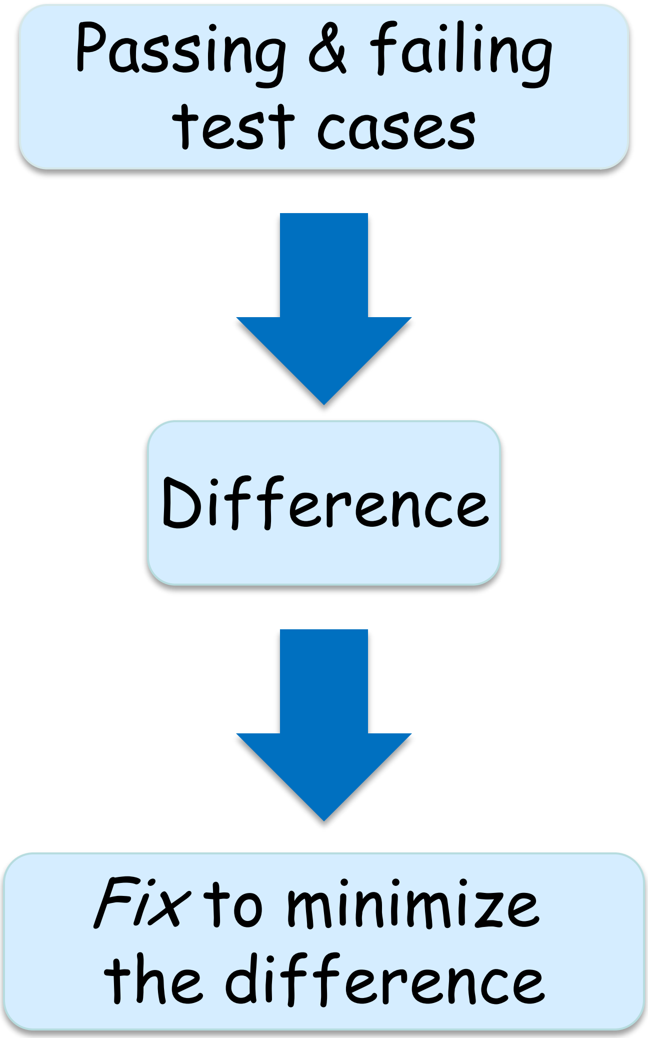
- Implementation properties
- Frame properties
- Theorems

Results

- A high proportion of inferred contract clauses are correct (*90%*) and relevant (*64%*) with the large test suite (*50* calls per method)
- Contract inference doesn't find all programmer-written contracts (only *59%*)
- Programmers don't write all inferred contracts (*25%*)
- These two types of contracts also differ qualitatively
- Contract inference can strengthen programmer-written postconditions and invariants
- Contract inference can be used to find missing precondition clauses

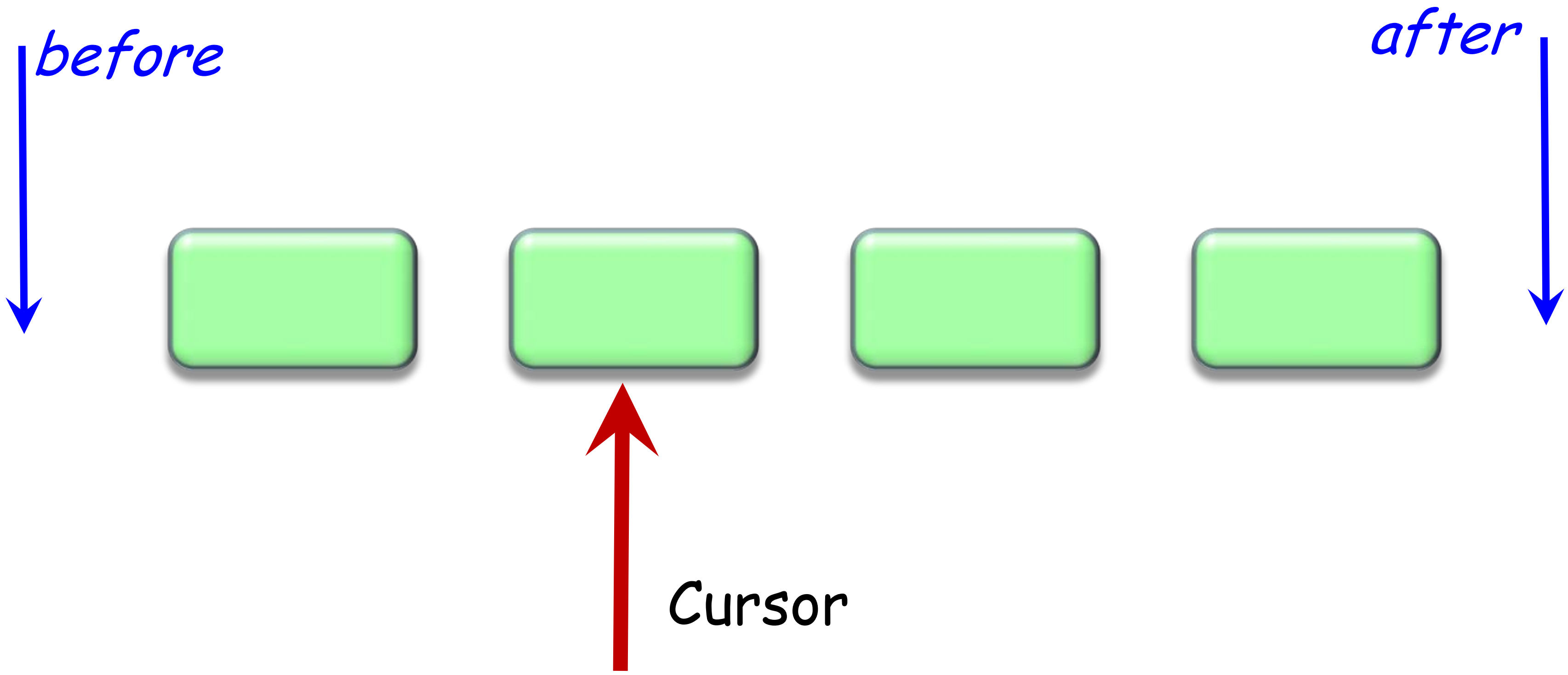
AutoFix: programs that fix themselves

Yi Wei, Yu Pei



- 16 faults fixed out of 42
- Some of the fixes are exactly the same as those proposed by professional programmers

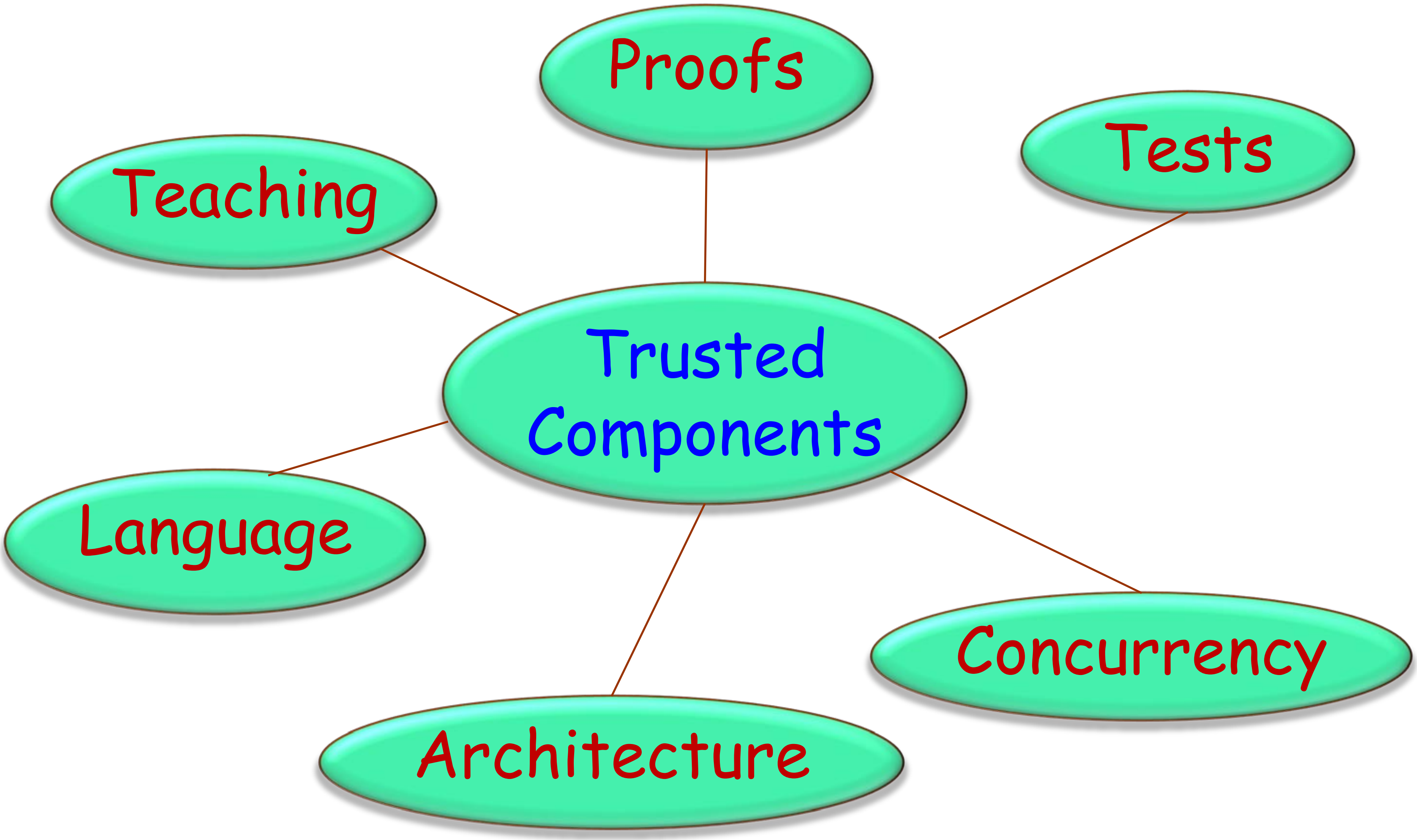
AutoFix demo: background



AutoFix

Demo

Verification: our vision



Proof technology

Bernd Schoeller, Martin Nordio, Julian Tschannen

Currently: Boogie (Microsoft Research, based on Z3)

Others possible in the future

Verifying agents (closures, delegates, function objects)

Martin Nordio, Julian Tschannen

Implemented an automatic verifier for a subset of Eiffel

- Same architecture as Spec# verifier
- Translation to Boogie
- Boogie verifier

Methodology for function objects

- Using abstract specifications

Julian Tschannen, Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, TOOLS 2010

Agent verification: demo

Demo context:

Formatting procedure applies variable formatting operation, which may be e.g. align_left, align_right

Precondition of formatting operation is unknown in general, but known for specific operations, e.g.

```
align_left
    require
        not left_aligned
```

Verifying agents

Can prove iterations, e.g.

```
my_integer_list.do_if
(agent is_negative, agent replace_by_square)
```

Verifying exceptions

Martin Nordio

"Exception invariants" to handle rescue-retry mechanism

Harnessing pointers

Stephan van Staden, Cristiano Calcagno

Approach 1: use separation logic

Extension of current separation logic techniques to O-O constructs (based on work of Matthew Parkinson and others)

Harnessing pointers: the alias calculus

Calculus for determining the alias relations that may exist at various points in a program

Simple, about a dozen rules

Experimental implementation

Appears efficient and scalable

Not yet included in EiffelStudio or EVE

May be an important step towards solving the frame problem

Bertrand Meyer: *Towards a Theory and Calculus of Aliasing*, Journal of Object Technology (JOT), vol. 9, no. 2, March-April 2010, pp. 37-74, http://www.jot.fm/issues/issue_2010_03/column5/

Towards full specifications

Nadia Polikarpova

Eiffel contracts (also JML, Spec#...) are typically **incomplete**
(unlike those of fully formal approaches such as Z)

Our solution:

Use models

A model is a mathematical interpretation of the structures

Model library: MML (Mathematical Model Library)

Fully applicative (no side effects, attributes, assignment etc.)

But: expressed in Eiffel (preserving executability)

This slide and the next five are by Nadia Polikarpova

LIST: contracts

```
class LIST [G]
    ...
    count: INTEGER
            -- Number of elements
    i_th (i: INTEGER): G
            -- Value at position `i'
        require
            1 <= i and i <= count
    index: INTEGER
            -- Cursor position
    put_right (v: G)
            -- Insert v to the right of cursor
        require
            index <= count
        ensure
            i_th (index + 1) = v
            count = old count + 1
            index = old index
            -- Old elements are still there
end
```

LIST: model-based contracts (1)

```
note
    model: sequence, index
class LIST [G]
    ...
    sequence: MML_SEQUENCE [G]
            -- Sequence of elements
    index: INTEGER
            -- Cursor position
    put_right (v: G)
            -- Insert v to the right of cursor.
        require
            index <= sequence.count
        ensure
            sequence = old (sequence.front (index).extended (x)
                + sequence.tail (index + 1))
            index = old index                              -- Theorem
        end
        ...
```

complete

LIST: model-based contracts (2)

```
...
    i_th (i: INTEGER): G
            -- Value at position i.
        require
            1 <= i and i <= count
        ensure
            Result = sequence [i]
           end
    duplicate (n: INTEGER): LIST [G]
            -- A copy of at most n elements starting at cursor position
        require
            n >= 0
        ensure
            Result.sequence =
                sequence.interval (index, index + n - 1)
                 Result.index = 0
        end
end
```

Model-based contracts: applications

On 7 of the most popular EiffelBase classes

Testing found 4 “functional” faults by violation of model-based contracts

EiffelBase: a data structures library with strong contracts

- Provides arrays, lists, sets, maps, stacks, queues
- 95% of features have complete contracts
- Aim is to prove the code against these contracts

Model-based contracts: future work

Contract more libraries and applications

Testing: more experiments

Proofs: model-based contracts need special support from the proof tool

Perform a user study:

- Do programmers understand model-based contracts?
- Can they write model-based contracts?

Loop invariant inference

Carlo Furia

Basic idea:

- Start from postcondition
- Infer loop invariant

Strategies for getting invariants

Constant relaxation

- Replace constant by variable
- Example: array maximum, …

Uncoupling

- Replace variable appearing twice by two variables
- Examples: square root, partition, …

Variable aging

- Use expression representing previous value
- Example: array maximum (other implementation)

Term dropping

- Remove part of precondition
- Example: partition

Computing a square root

Goal: Result^2 = a i.e.   **Result** * **Result** = a

Strategy: uncoupling; rewrite as

Invariant → $x * y = a$ **and** $x = y$ ← **Exit condition**

```
from
        "Establish invariant"
        -- Establish invariant:
            x := 1 ; y := a
until  x = y   loop
        "Bring x closer to y, maintaining invariant"
        -- Bring x closer to y, maintaining invariant:
                "Bring x closer to y"
                -- Bring x closer to y:
                    x := (x + y) / 2
                "Restore invariant"
                -- Restore invariant:
                    y := a / x
end     ; Result := x
```

The Assertion Inference Paradox

Correctness is consistency of implementation to specification

The paradox: if the specification is inferred from the implementation, what do we prove?

Possible retorts:

- The paradox only arises for correctness proofs; there are other applications, e.g. reverse-engineering legacy software
- The result may be presented to a programmer for assessment
- Inferred specification may be inconsistent

In this work, we only infer **loop invariants**

The border line

Programmer writes postcondition

Tool infers loop invariant

Implementation: gin-pink

gin-pink: Generation of INvariants by PostcondItioN weaKening

- written in Eiffel
- command-line tool
  - Boogie in / Boogie out
- works with any high-level language that can be translated to Boogie
- available for download from http://se.inf.ethz.ch/people/furia/

This slide and the next three are by Carlo Furia

Experiments on literature examples

Program	candididates	invariants	relevant invariants	Time (s)
array part. (v1)	38	9	3	93
array part. (v2)	45	2	2	205
array reversal	134	4	2	529
array rev. (ann)	134	6	4	516
bubble sort	14	2	2	65
coincid. count	1351	1	1	4304
dutch flag	42	10	2	117
dutch flag (ann)	42	12	4	122

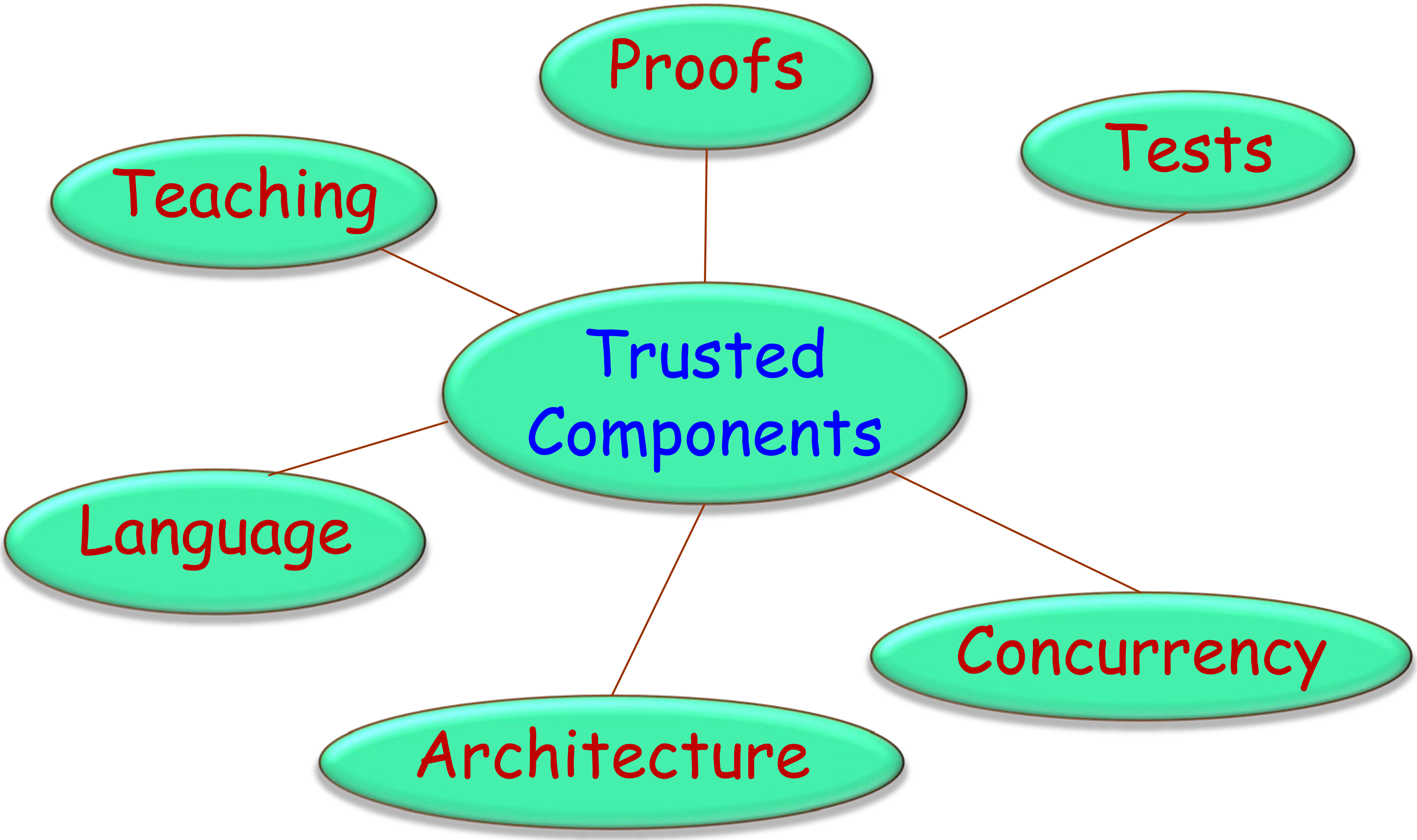
Experiments on literature examples

Program	candididates	invariants	relevant invariants	Time (s)
longest common sub. (ann)	508	22	2	4842
majority count	23	5	2	62
max of array (v1)	13	1	1	30
max of arr. (v2)	7	1	1	16
plateau	31	6	3	666
seq. search (v1)	45	9	5	120
seq. search (v2)	24	6	6	58

Experiments on literature examples

Program	candididates	invariants	relevant invariants	Time (s)
shortest path	23	2	2	53
stack search	102	3	3	300
sum of array	13	1	1	44
topolog. sort	21	3	2	101
welfare crook	20	2	2 (100%)	586

Verification: our vision



SCOOP mechanism

Piotr Nienaltowski et al.
e.g. Formal Asp. Comp, 2009

Simple Concurrent Object-Oriented Programming

Minimal extension to object-oriented programming (one keyword)

Assumption: programmers want to retain the ability to reason simply about programs ("reasonability")

Recent contributions: Sebastian Nanz, Benjamin Morandi, Scott West

Reasoning about objects: sequential

{INV **and** Pre_r} $body_r$ {INV **and** $Post_r$}

{Pre_r'} x.r (a) {$Post_r'$}

Priming represents actual-formal argument substitution

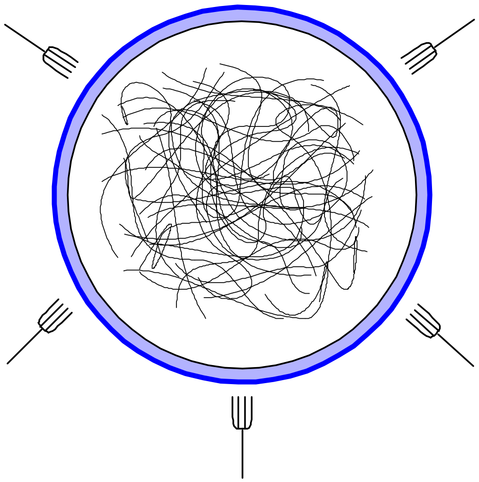
Only n proofs if n exported routines!

Dining philosophers

```
class PHILOSOPHER inherit
      PROCESS
            rename
                  setup as getup
            redefine step end

feature {BUTLER}
      step
            do
                  think ;   eat (left, right)
            end

      eat (l, r: separate FORK)
                  -- Eat, having grabbed l and r.
            do … end
end
```

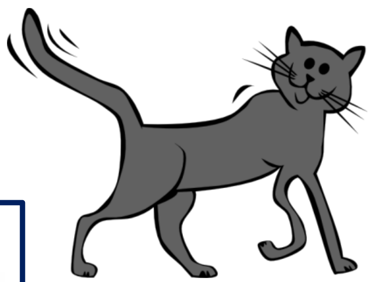


An application: hexapod robot

Sebastian Nanz, Benjamin Morandi,
Ganesh Ramanathan, Scott West

Distributed control

Load sensing

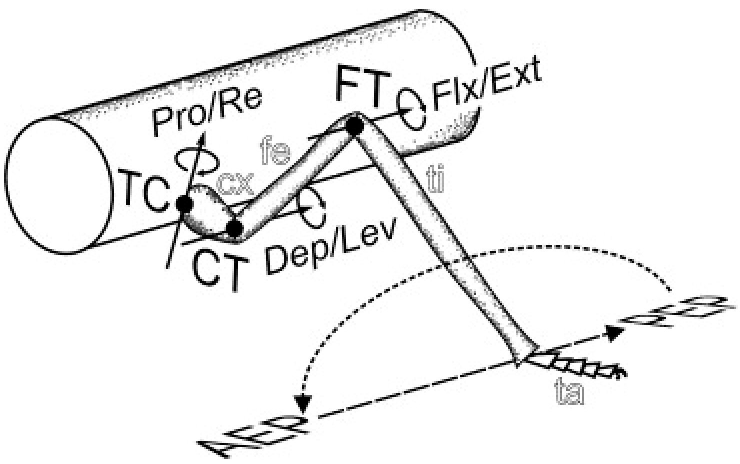


Centralised control

Balance sensing

This slide and next eight by above authors

Hexapod Locomotion

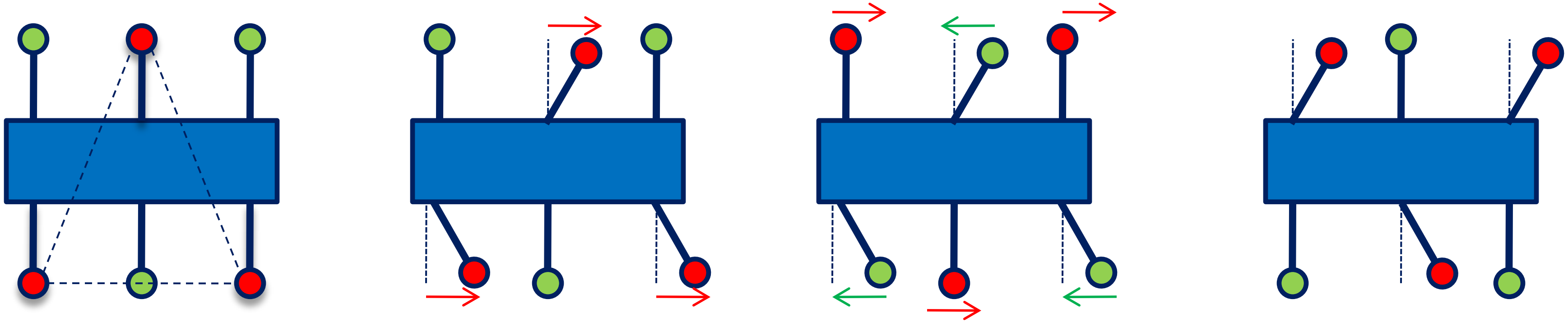


The hexapod should maintain the static stability by keeping the center of gravity within the bounds of the grounded legs.

Dragging of feet should be avoided.

Three degrees of freedom per leg, load sensor on feet, forward and rear angle sensing

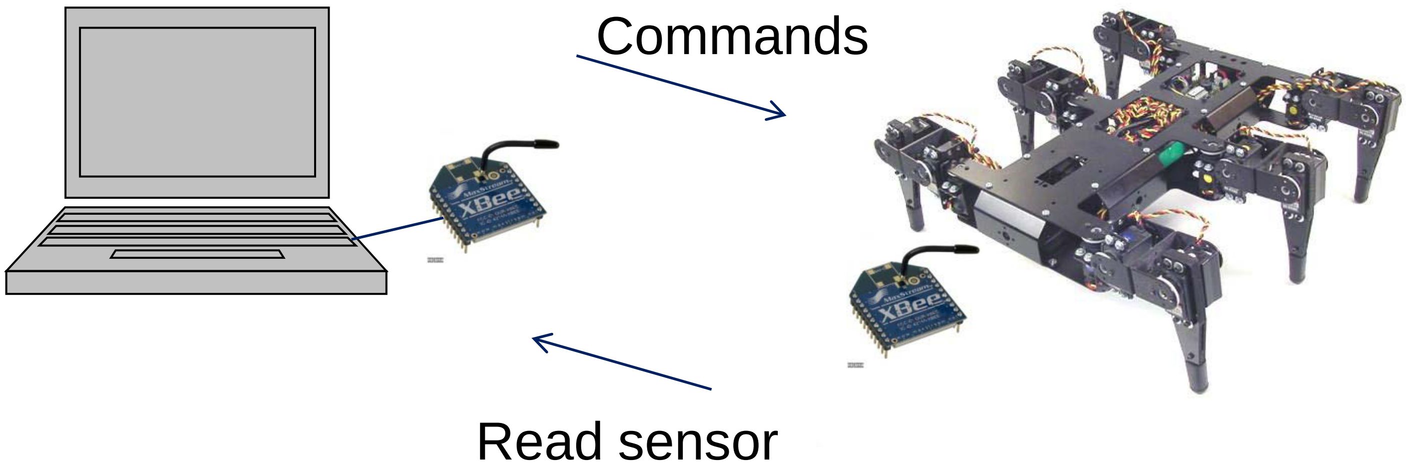
The Tripod Gait



Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up

The Hexapod Robot



The control program (SCOOP based or other variants) runs on the PC and transmits command to the on-board servo controller.

It also polls the inputs to obtain sensor information.

Implementation: Sequential Program

```
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true)
{
    lead.Raise();
    lag.Retract();
    lead.Swing();
    lead.Drop();

    TripodLeg temp = lead;
    lead = lag;
    lag = temp;
}
```

Implementation: Multi-Threaded Program

```
private object m_protractionLock = new object();

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState !=ThreadState.
          AbortRequested)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Partner.DroppedEvent.WaitOne();
            leg.Raise();
        }
        leg.Swing();

        // Waiting for partner retraction
        leg.Partner.RetractedEvent.WaitOne();
        leg.Drop();

        // Waiting for partner raise
        leg.Partner.RaisedEvent.WaitOne();
        leg.Retract();
    }
}
```

Hexapod implementation: SCOOP

```
walk
    do
        checklegs (my_signaler)
        from until my_signaler.stop_requested
        loop
            begin_protraction (partner_signaler, my_signaler)
            ensure_protraction (my_signaler)
            complete_protraction (partner_signaler)
            execute_retraction (partner_signaler, my_signaler)
        end
    end
```

Implementation: SCOOP

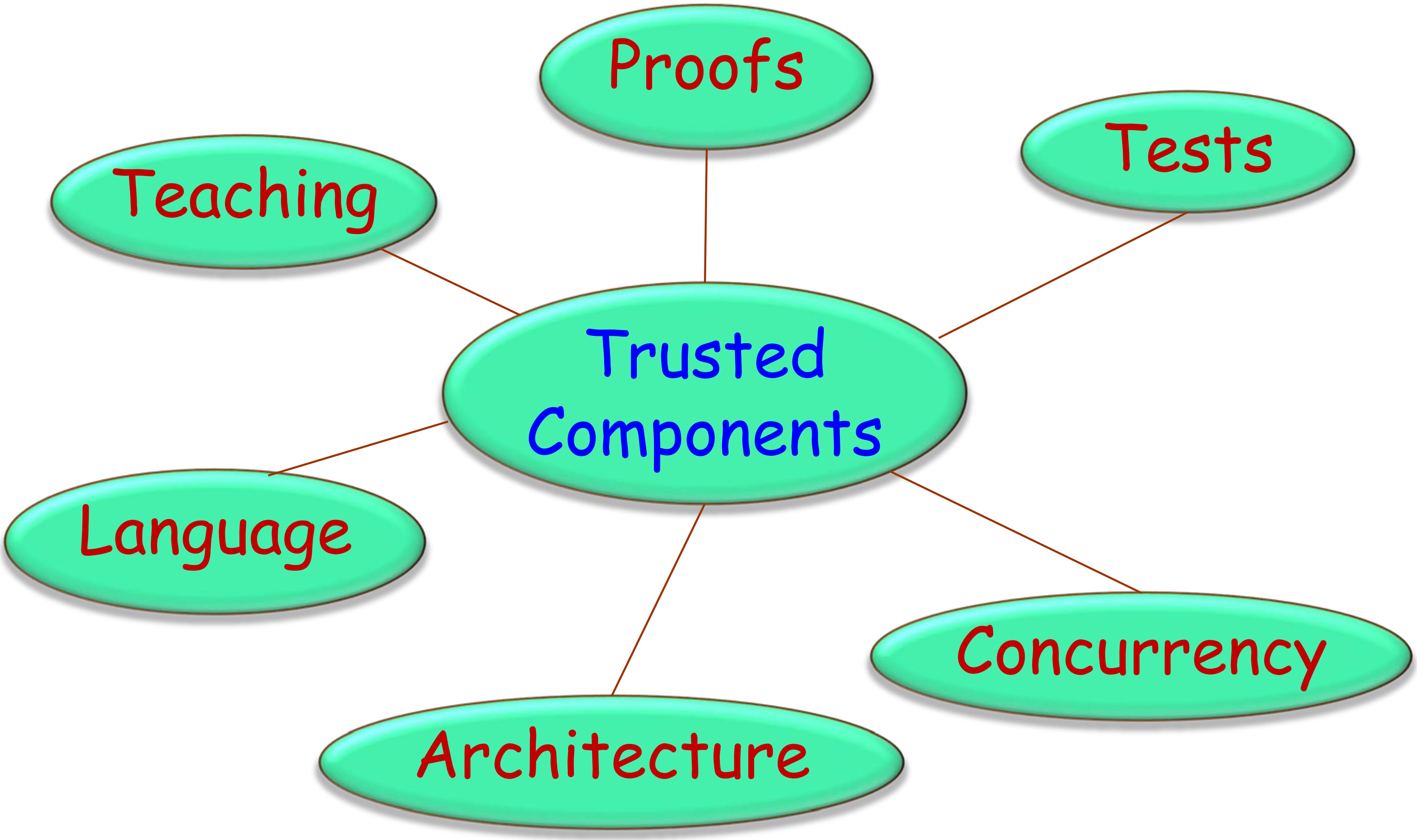
```
begin_protraction(partner, me:separate LEG_GROUP_SIGNALER) is
    --
    require
      my_legs_retracted : me.legs_retracted
      partner_down : partner.legs_down
      partner_not_protracting : not partner.protraction_pending
    do
      io.put_string (group_name)
      io.put_string (" : begin_protraction ")
      io.put_new_line

      tripod.lift

      me.set_protraction_pending(true)
    end
```

Demonstration

Verification: our vision



Teaching

Introductory programming (1st year)

Software architecture (2nd year)

Advanced courses:

- Software verification
- Distributed and Outsourced Software Engineering
- Concepts of Concurrent Computation
- Eiffel in depth
- Java & C# in depth
- Software engineering seminar

Introductory programming teaching

Teaching first-year programming is a politically sensitive area, as you must contend not only with your students but also with an intimidating second audience — colleagues who teach in subsequent semesters….

Academics who teach introductory programming are placed under enormous pressure by colleagues.

As surely as farmers complain about the weather, computing academics will complain about students' programming abilities.

Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing*,
10th Conf. on Australasian computing education, 2008

Concepts or skills?

Skills supported

by concepts

Principles of our course

- Fully object-oriented from the start, using Eiffel

- Outside-in ("Inverted Curriculum")

- Gentle introduction to formal techniques:
  Design by Contract

Outside-in

Michela Pedroni

The course gives students a large amount of software, right from the beginning

TRAFFIC library

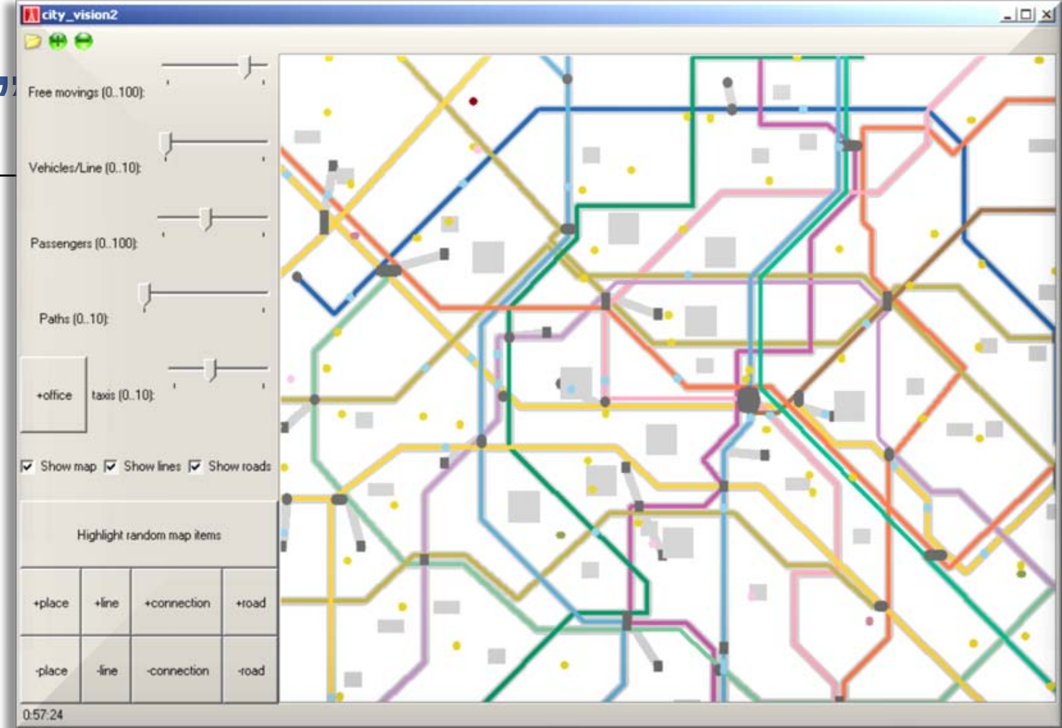
Michela Pedroni & numerous student projects; about 150,000 lines of Eiffel

- They start out as consumers
- They end up as producers!

"Progressive opening of the black boxes"

TRAFFIC is graphical, multimedia, extendible, and fun!

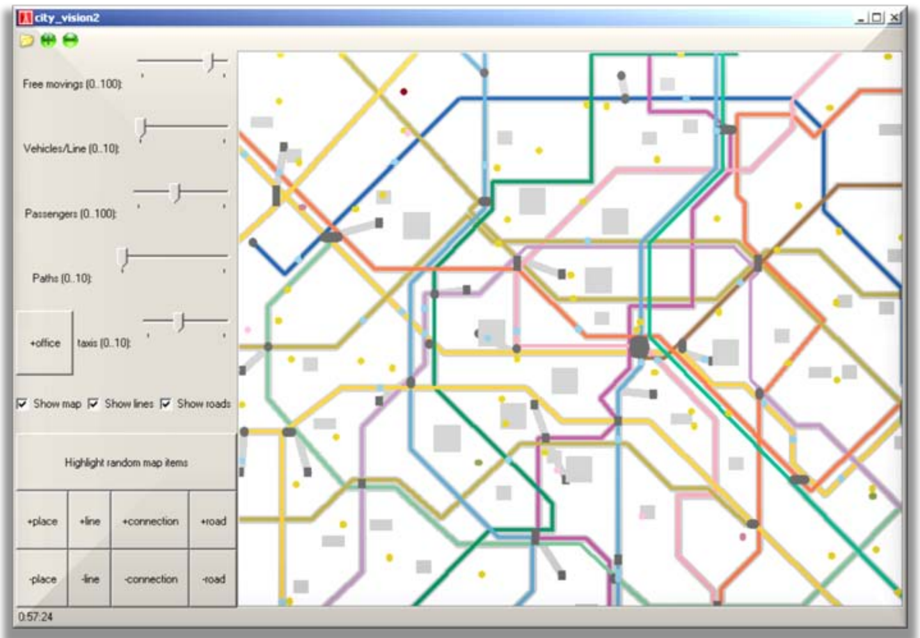
The first “program”




```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Prepare & animate route
        do
            Paris.display
            Louvre.spotlight
            Metro.highlight
            Route1.animate
        end
end
```

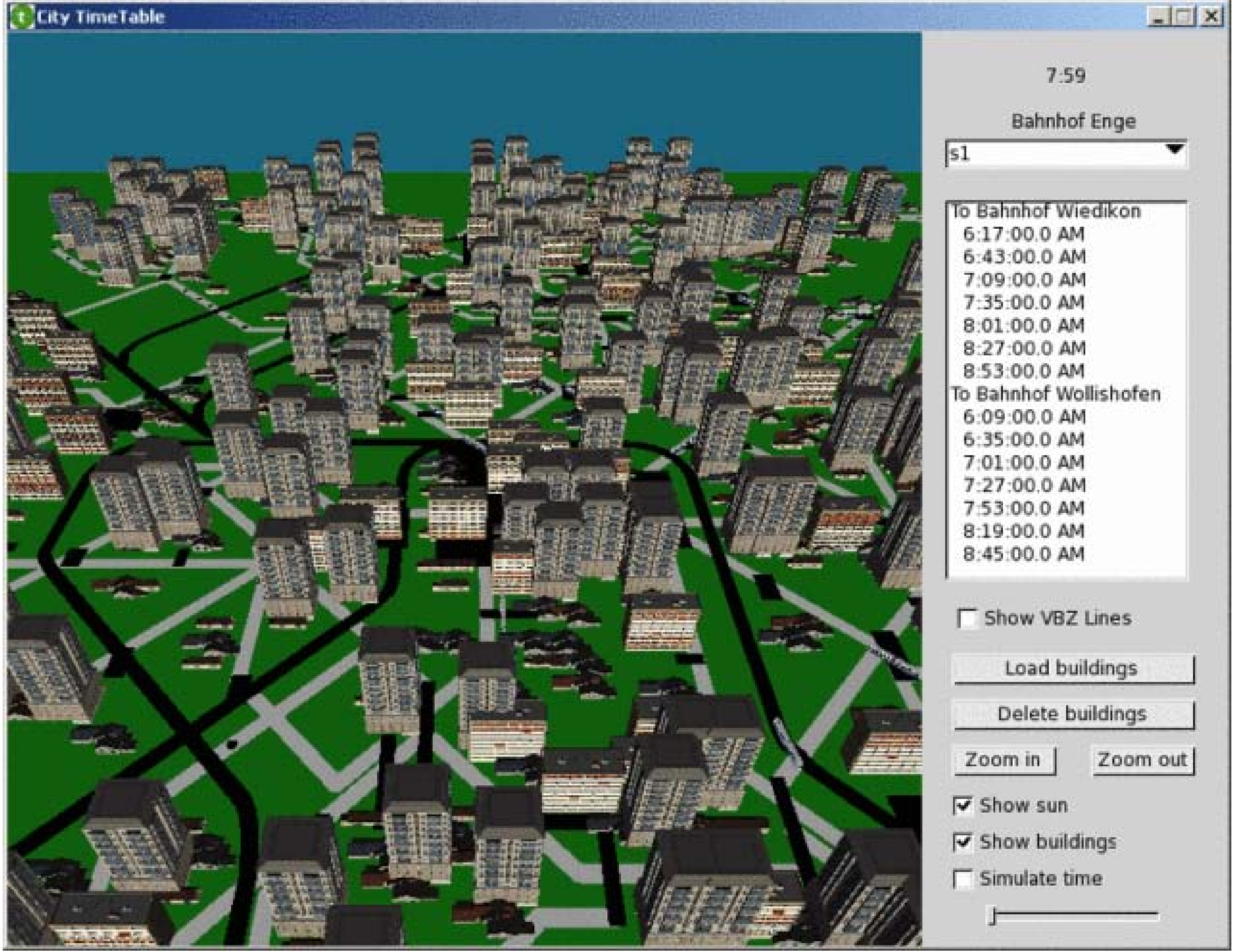
Text to input

The first “program”



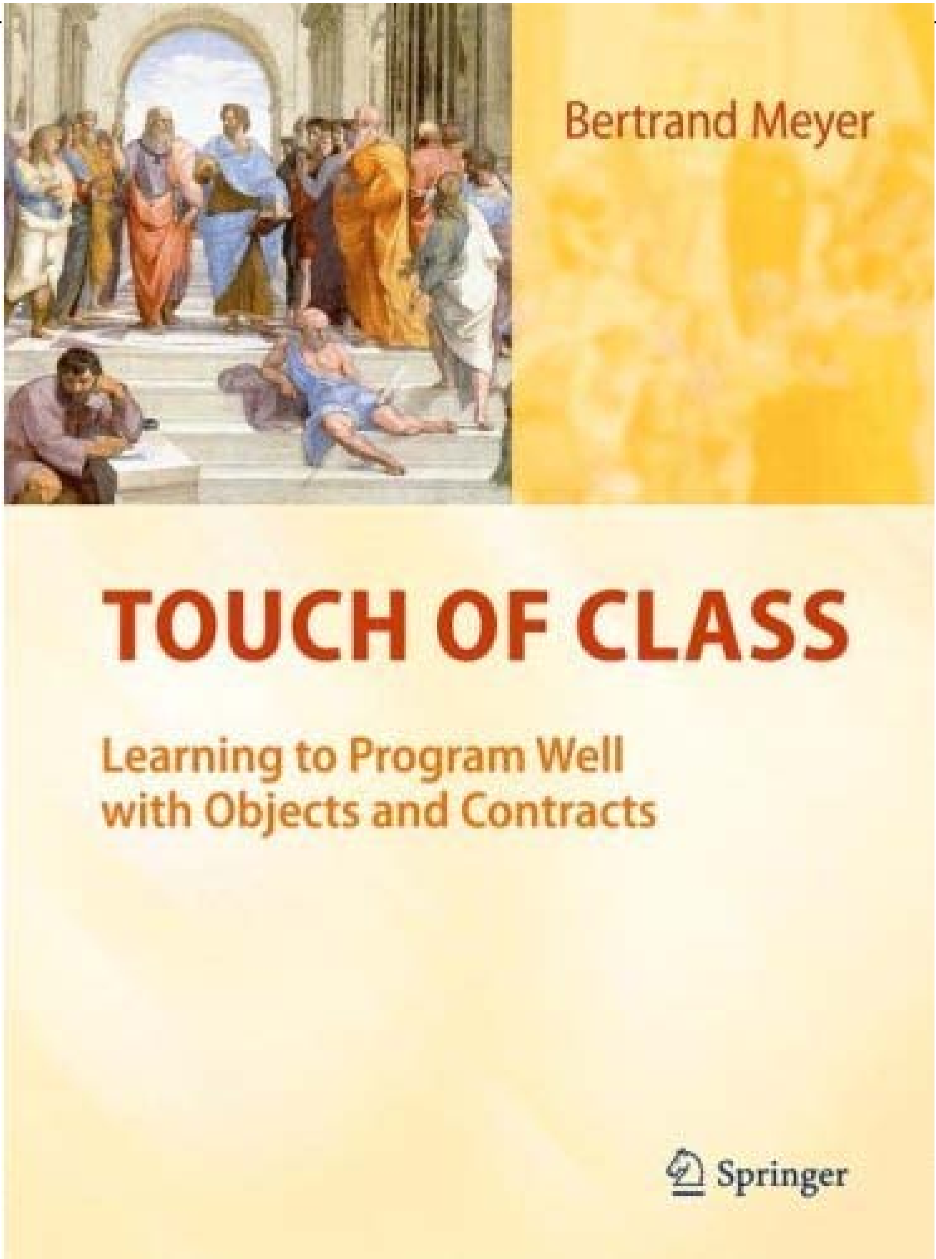

```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Prepare & animate route
        do
            Paris.display
            Louvre.spotlight
            Metro.highlight
            Route1.animate
        end
end
```

Text to input


City TimeTable
7:59
Bahnhof Enge
s1
To Bahnhof Wiedikon
6:17:00.0 AM
6:43:00.0 AM
7:09:00.0 AM
7:35:00.0 AM
8:01:00.0 AM
8:27:00.0 AM
8:53:00.0 AM
To Bahnhof Wollishofen
6:09:00.0 AM
6:35:00.0 AM
7:01:00.0 AM
7:27:00.0 AM
7:53:00.0 AM
8:19:00.0 AM
8:45:00.0 AM
Show VBZ Lines
Load buildings
Delete buildings
Zoom in
Zoom out
Show sun
Show buildings
Simulate time

Textbook

touch.ethz.ch



DOSE course

Peter Kolb, Martin Nordio, Roman Mitin

Distributed and Outsourced Software Engineering (since 2004)

Goal: Prepare students to the new, globalized world of software development

Some topics:

- Requirements in a distributed project
- Quality assurance
- Project models, CMMI
- Agile methods
- Managing relationships with suppliers, contract negotiation
- ...

Project: involving other universities

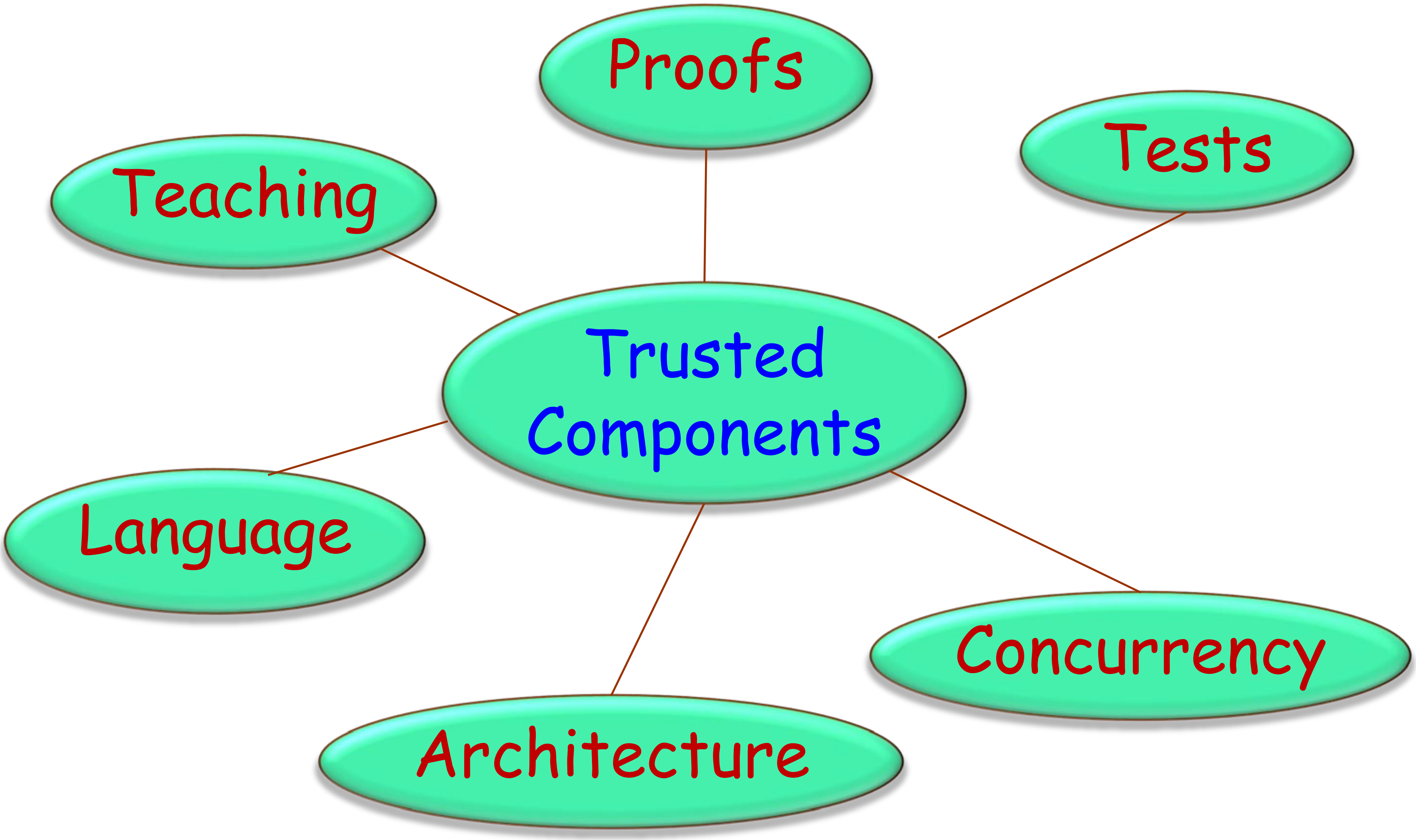
2009

- Politecnico di Milano (Italy)
- Hanoi University of Technology (Vietnam)
- Odessa National Polytechnic (Ukraine)
- University of Nizhny Novgorod (Russia)
- University of Zurich
- University of Debrecen (Hungary)\
- ETH

The main lessons students learn

The importance of APIs & specifications

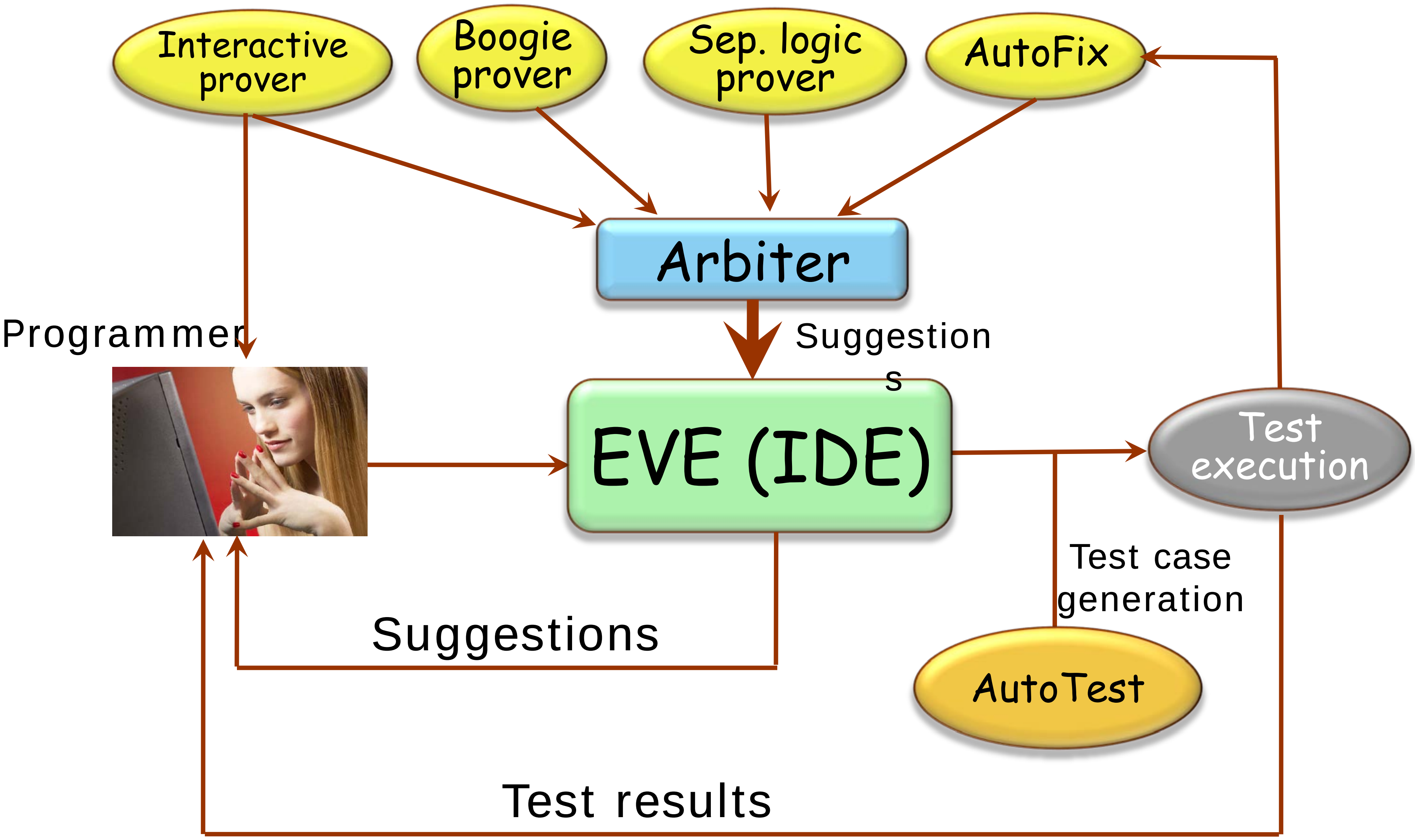
Verification: our vision



Ten years from now

1. We will still be using O-O languages
2. Professional programming will be far more rigorous
3. Verification will be integrated in the development process
4. Every program will have a Web interface
5. Concurrency will be ubiquitous
6. More reliance on objective assessment
7. Software ***engineering***: not just process but technology

The verification assistant



For more

http://se.ethz.ch

http://www.eiffel.com

Forthcoming conferences:

- TOOLS EUROPE (Malaga, June)
- SEAFOOD (distributed development), Saint Petersburg, June
- LASER summer school (Sept. 2010, Elba): experimental software engineering, see http://se.ethz.ch/laser