Bugs in computer algebra systems: how to catch or even avoid

Jakob Kröker

April 3, 2015

Content

Motivation: why get obsessed with bug hunting
- famous software bugs
- observed CAS-bugs in our research project

What probably did went wrong

Bug hunting: do it yourself
- tests
- conservative programming
- code review

some observations
- error classes, error examples

Avoiding errors
- basic suggestions

the impact of software errors: famous software bugs

- faulty conversion from a float to 16 bit int responsible for the explosion of Ariane 5
- error in sovjet early warning system against a first-strike almost triggered nuclear war in 1983
- Patriot-missile-bug
- in medicine: error in THerac-25 system

reference selecton:

- `http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugs.html`
- `http://www.softwareqatest.com/qatfaq1.html`
- `http://www.ima.umn.edu/~arnold/disasters/`

Software bugs nearly brought on nuclear war in 1983

The software was supposed to filter out false missile detections caused by Soviet satellites picking up sunlight reflections off cloud-tops, but failed to do so.

Disaster was averted when a Soviet commander, based on what he said was a '...funny feeling in my gut', decided the apparent missile attack was a false alarm.

The filtering software code was rewritten.

why get obsessed with bug hunting

one of projects I'm working on:

resolution of singularities of arithmetic algebraic surfaces

- when tracking down a performance issue we observed that different CAS versions returned different results.

BUG alert!

why get obsessed with testing

one of projects I'm working on:

resolution of singularities of arithmetic algebraic surfaces

- when tracking down a performance issue we observed that different CAS versions returned different results.
- further investigation (mainly testing and reviewing) uncovered more bugs
  - about 30 bugs in our small package (one bug per 100 loc)
  - $>$100 bugs in library and kernel code (including unrelated)

meanwhile our project slowly converges to something usable, but it is not there yet

obvious observed bugs

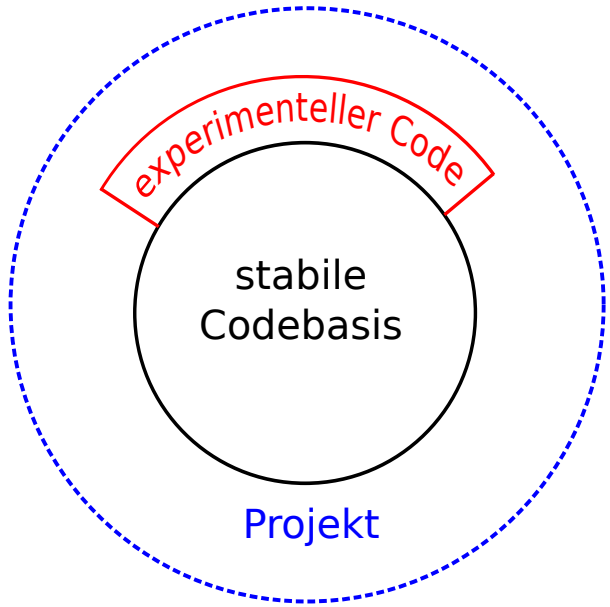
using a polynomial ring with more that one variable
over rational numbers:

- take the radical of zero ideal in Macaulay2
- compute minimal associated primes of the unit ideal in Sage
- check if the zero ideal is primary in Macaulay2

what happened (my point of view):

In summary: (partly) failed quality assurance

in particulary, groebner basis related computation over $\mathbb{Z}$ was/is experimental but was not marked as such any more



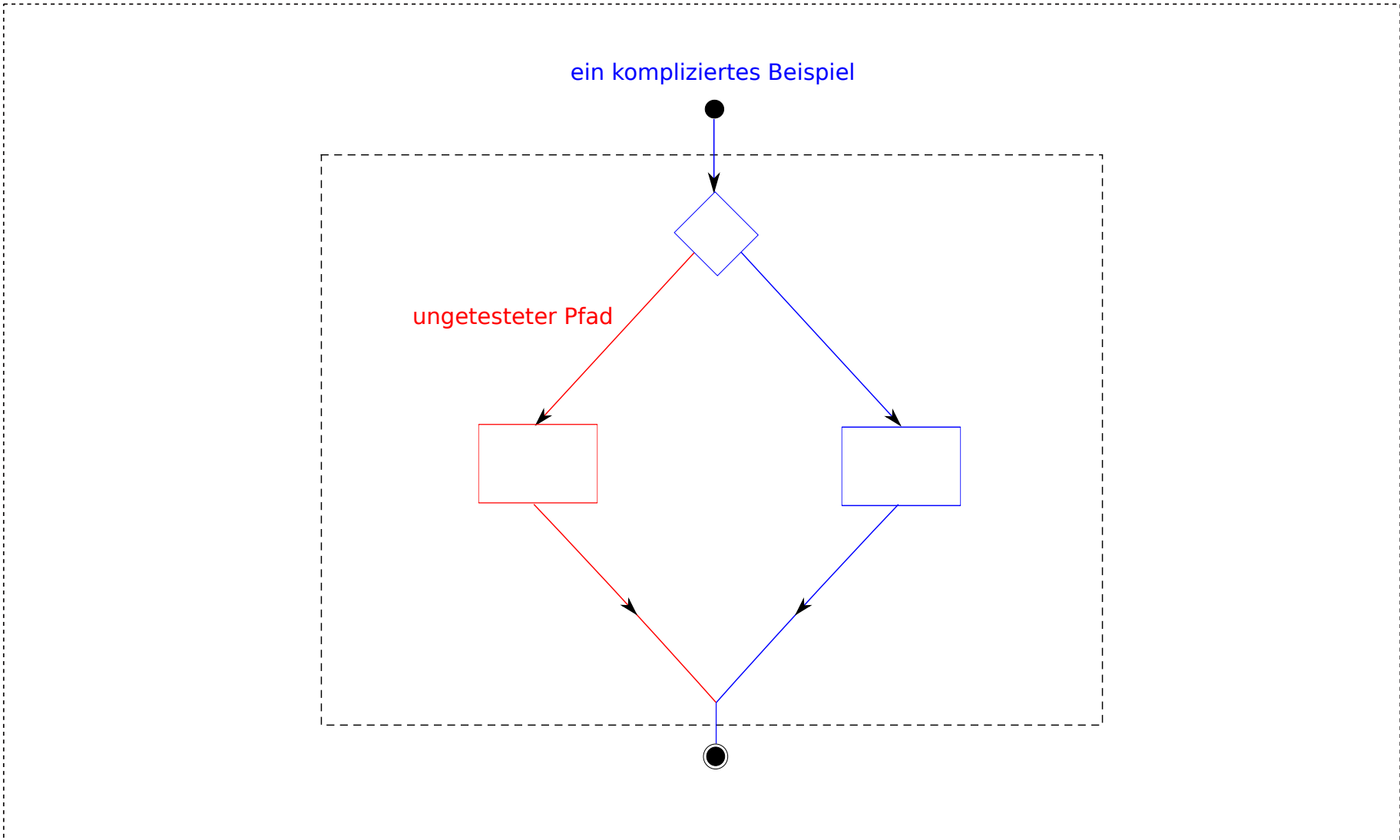
what happened (not only over $\mathbb{Z}$):

mainly incorrectly handled corner cases (not only)
(sometimes they were uninteresting from mathematical point of view)

false feeling of safety:

- often a (couple of) single examples were considered as sufficient tests
- high code coverage does not guarantee high quality code
- internal functions usually had no tests and no documentation
- not enough testers/ testing not adequate

a single example as a test



what happened (my point of view):

- parts of source code checked in without review
- some reviews were not properly performed
- people start use new functionality everywhere
- here and there sources show programming or engineering skill deficits or contain historical artifacts (Singular is old)
- no quality manager role ?
- unfortunate priorities caused by academic community pressure (publish or perish)

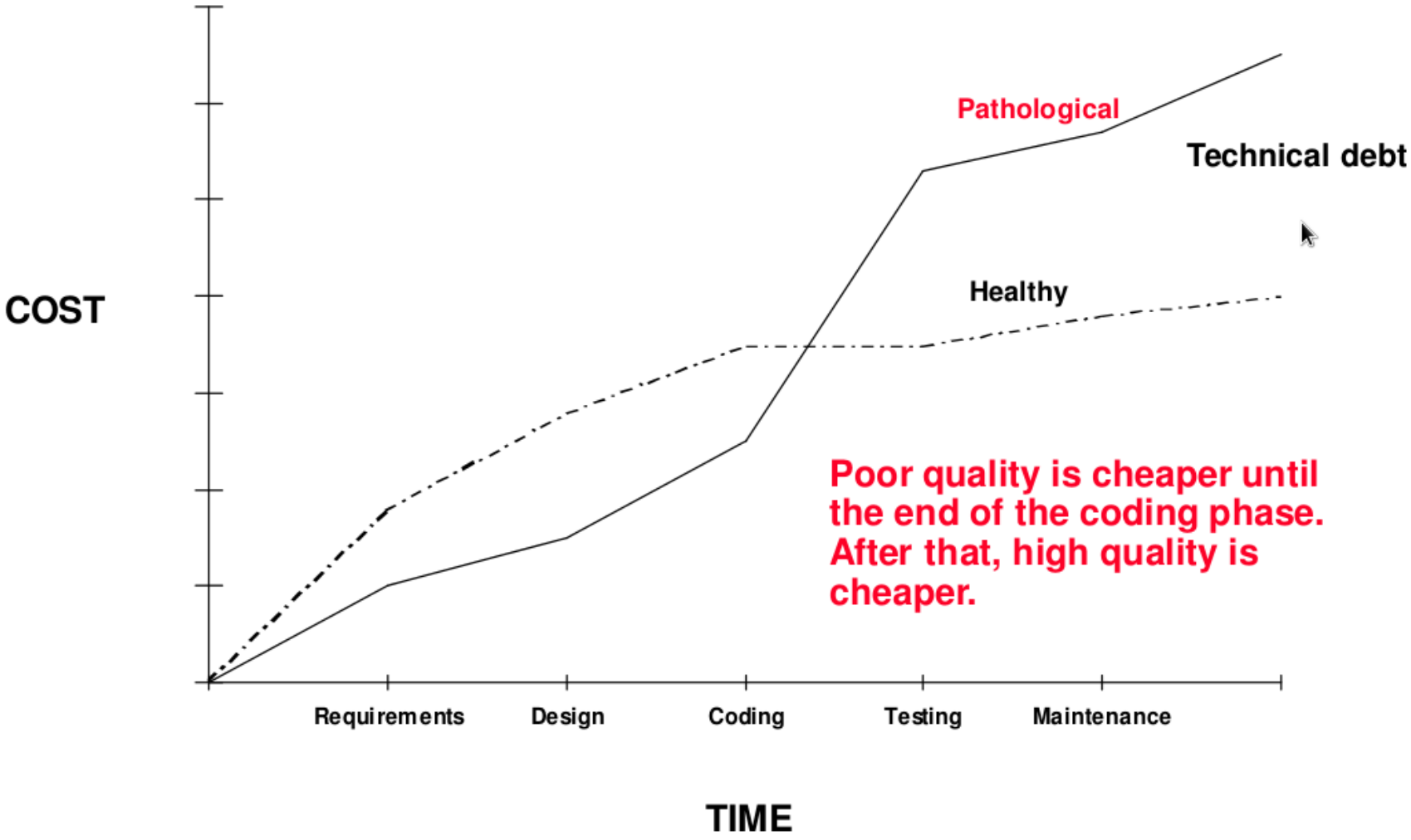
what happened:

bug reporting:

- users do rather work around bugs than report
- not all reports are recorded in bug tracker (and get lost over time)

influence of quality to project evolution

HOW QUALITY AFFECTS SOFTWARE COSTS



Why is it hard to get serious about quality assurance?

Why is it hard to get serious about quality assurance?

Solving problems is a high-visibility process;
preventing problems is low-visibility.

Bug hunting: do it yourself

why bug hunting and preventing

- wish for low error rate
- production increase in long term

looking for bugs? You can catch them, too!

Ideas for bug hunting

- write (automated) tests
- add consistency checks *(conservative programming)*
- perform source and specification reviews

almost every in following presented strategy did lead to uncovering bugs related to our project!

testing: ideas for bug hunting

- check corner and undefined cases

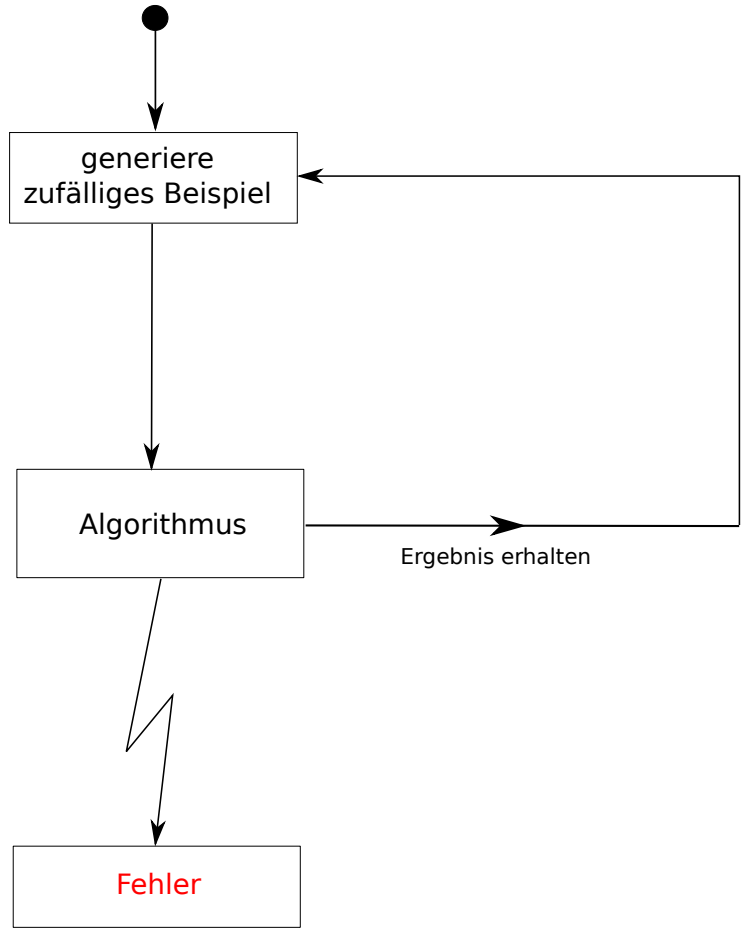
check corner cases

- primality test for nonegative integers
  - special cases: 0, 1,2
  - a while ago for CAS Axiom number 2 was not prime
- compute minimal associated primes
  - corner cases: unit ideal, zero ideal
  - sage tells that unit ideal has a minimal associated prime

testing: selection of ideas

- check corner and undefined cases
- check interfaces with random input

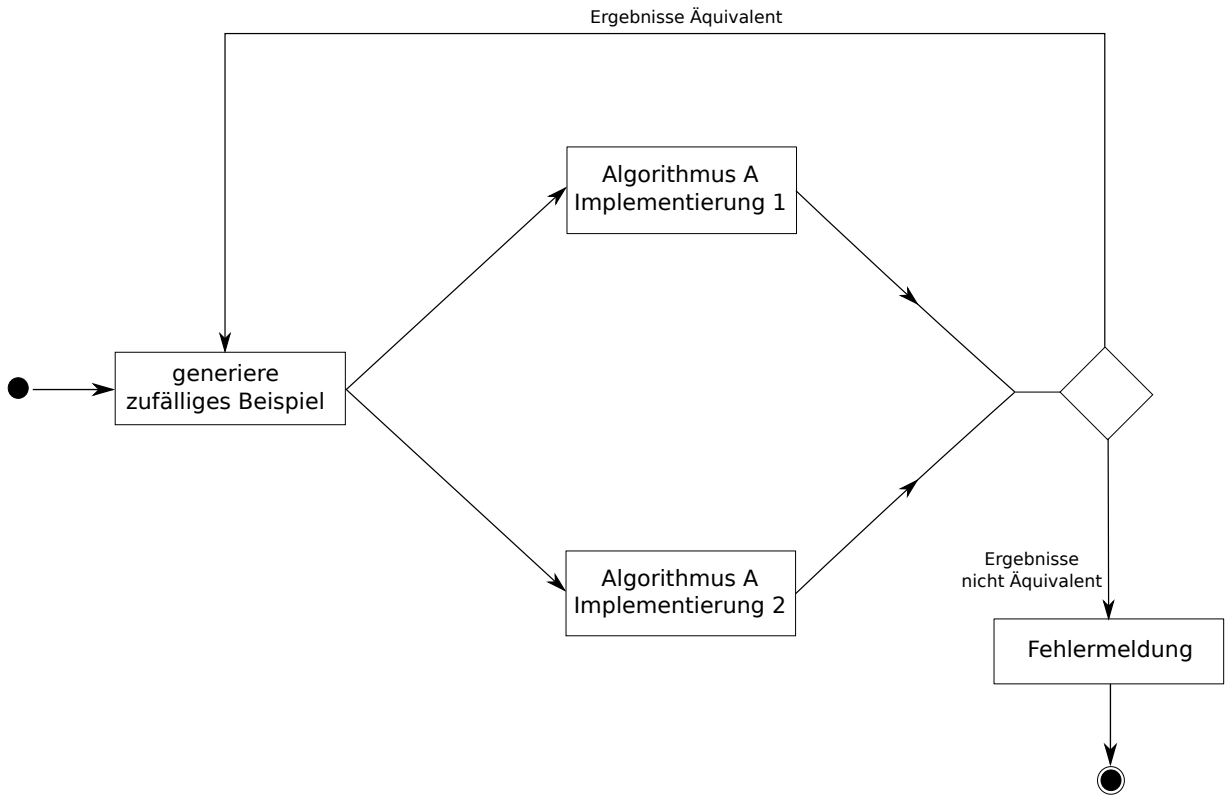
testing only by random input



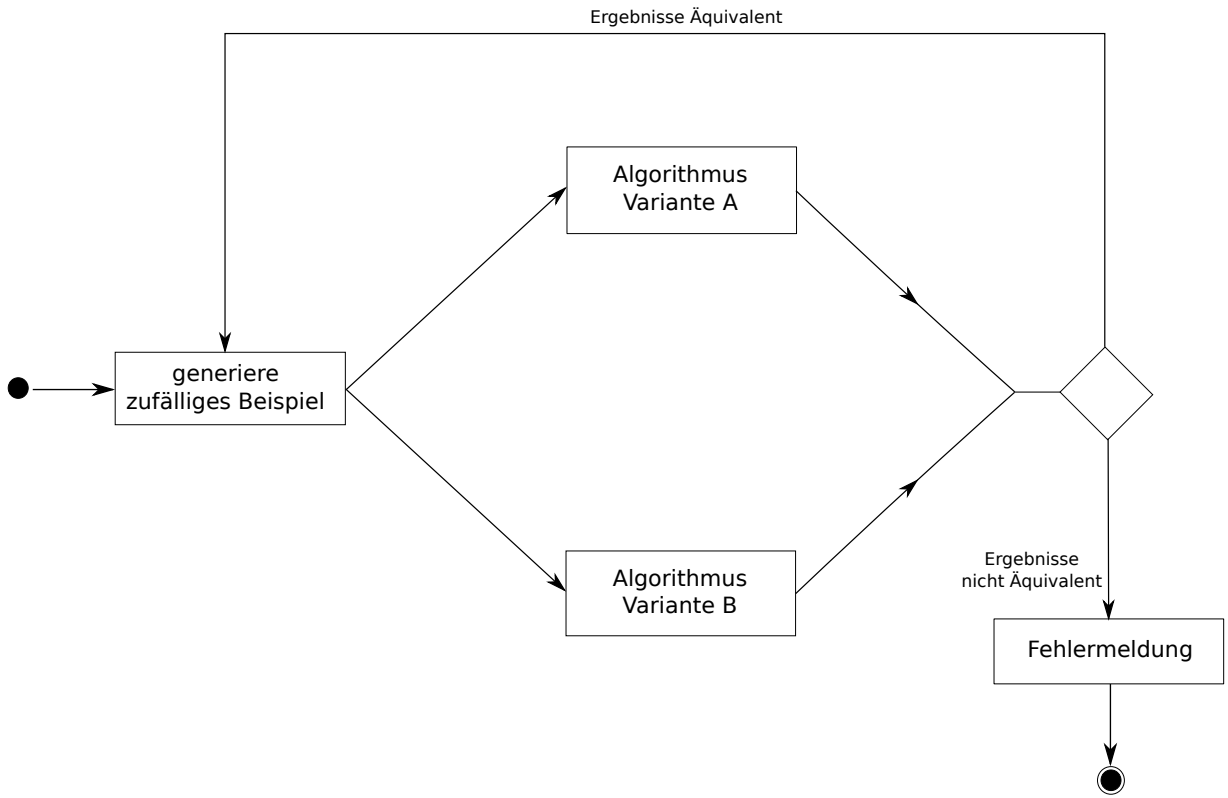
testing: selection of ideas

- check corner and undefined cases
- check interfaces with random input
- verify examples with known results
- test with random input by comparing outputs of
  - different algorithms (solving same problem)
  - differing implementations (of same algorithm)

testing via result comparison (1)



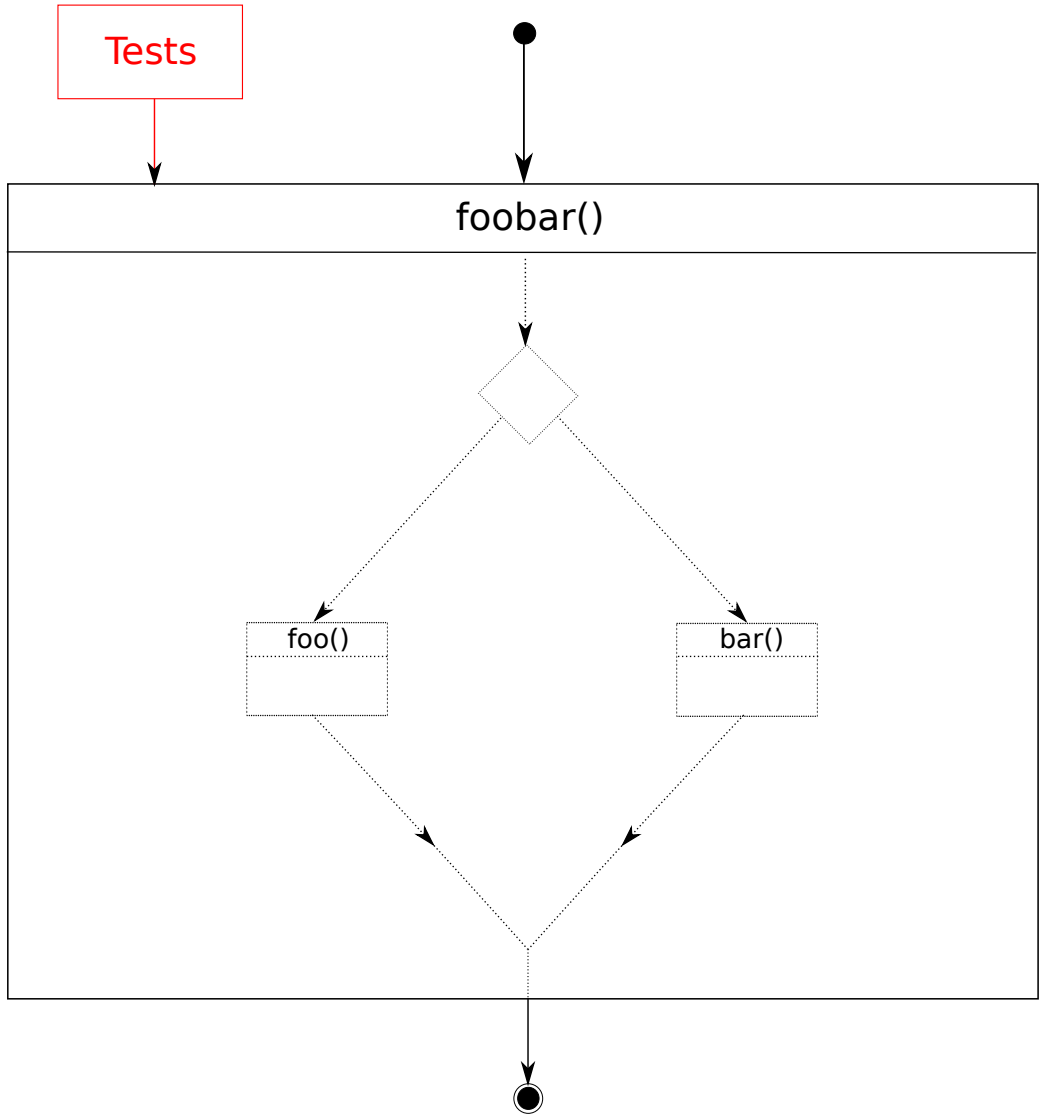
testing via result comparison (2)



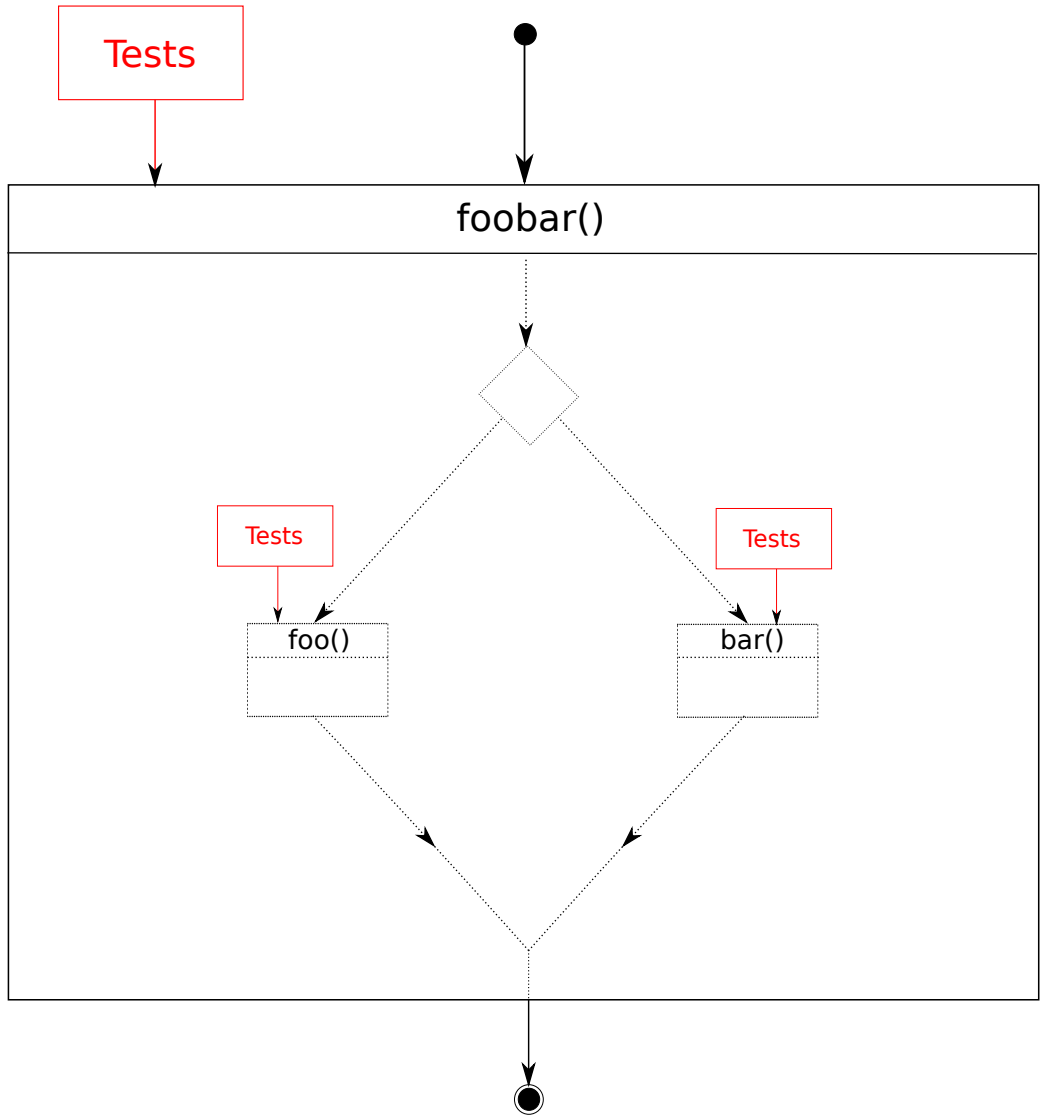
testing: selection of ideas

- check corner and undefined cases
- check interfaces with random input
- verify examples with known results
- test with random input by comparing outputs of
  - different algorithms (solving same problem)
  - differing implementations (of same algorithm)
- looking for bugs (siblings), which are similar to already known ones

coarse-mesh testing



finegrained testing



who should test

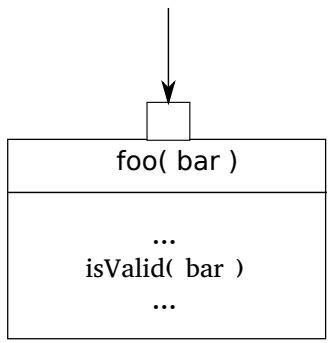
- at best, not the developer
- sceptical persons

tests extent

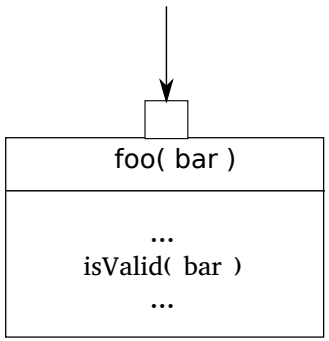
- rule of thumb:
  test to source ratio: 50/50

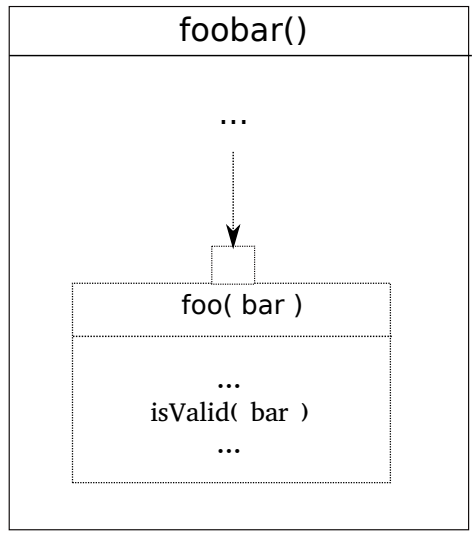
- extremal example SQLite:
  (test loc)/(src loc) is 1000 to 1!

conservative programming: validate parameter input



conservative programming: validate parameter input





conservative programming: check result

check result integrity (if asked for)

- by checking expected properties or known invariants
- by comparing with a result obtained by an alternative implementation

source code review

in the main sage project every added piece of sources is reviewed

however, it seems that issues sometimes slip through
- recent example: 'IntegerListsLex'

observed errors

selection of observed error types

- incorrecly handled corner case 27+
- memory management bug 11+
- design issue 6+
- incorrect usage of a function 5+
- incomplete documentation 4+
- new bugs as consequence of refactoring (changed behaviour) 2+
- too much intelligence 2+
- bugs caused by unfortunate naming 2+
- others 25+

example: unfortunate naming

Let $\mathbb{R} = \mathbb{Q}[x, y]$; ideal $I = \{x,\ 0,\ y\}$

What is the purpose of

- size(I) ?
- ncols(I) ?

example: unfortunate naming

Let $\mathbb{R} = \mathbb{Q}[x, y]$; ideal $I = \{x,\ 0,\ y\}$

what could be the purpose of

- nNonzeroGens(I)
- nGens(I)

?

example for too much intelligence

- Singular: autorenaming of ring variables in case of conflict

leads to bugs or shadows bugs, see

- http://www.singular.uni-kl.de:8002/trac/ticket/609
- http://www.singular.uni-kl.de:8002/trac/ticket/508
- ...

Avoiding errors

basic suggestions

- offer and ensure continuing education in programming, engineering and development process areas
- know and follow common best practices
- do not accept/use experimental (library) code or mark is as such
- should there be consequences for sharing broken code? (I don't know)
- prefer correctness to optimization
- conservative programming
- rate quality of libraries/packages
- apply for quality manager and tester position grants
- value high quality and performant software
- write a testing bot/framework

Thank you for attention!

Questions?