Software Verification

Bertrand Meyer
Carlo Furia
Sebastian Nanz

Testing, part 2

Mutation testing

How do you count the Eggli in the Zürichsee?

Mutation testing

Purpose: estimate quality of a test suite

Principle: make small changes to the program source code (so that the modified versions still compile) and see if successful test cases still succeed

If they do, the test suite is not good enough!

Terminology

Mutant: a modified version of the program, obtained by injecting a fault

- We only consider mutants that are not equivalent to the original program

Killed mutant: At least one test case detects the injected fault

Alive mutant: no test case detects the injected fault

Mutation score : measurement of effectiveness of test, defined next

Mutation operators

Mutation operator: a rule that specifies a syntactic variation of the program text so that the modified program still compiles

A mutant is the result of an application of a mutation operator

The quality of the mutation operators determines the quality of the mutation testing process

Mutation operator coverage (MOC): For each mutation operator o, there is at leas one mutant using o

Examples of mutants

Original program:

```
if (a < b)
        b := b – a;
else
        b := 0;
```

Mutants:

```
if (a < b)
if (a <= b)
if (a > b)
if (c < b)
        b := b – a;
        b := b + a;
        b := x – a;
else
        b := 0;
        b := 1;
        a := 0;
```

OO mutation operators

Polymorphism- and dynamic binding-related:

- Change creation type
  create x.make → **create** {T} x.make
- Redefinition
  Replace inherited routine or attribute by redefined version

Various:

- Argument order change
  If types match, e.g. f (x, y: INTEGER)
- Replace assignment by copy
  list1 := list2.twin → list1 := list2

System test quality (STQ)

S: system composed of n components, denoted C_i
d_i: number of killed mutants after applying test sequence to C_i
m_i: total number of mutants

Mutation score for C_i and test sequence T_i: $MS(C_i, T_i) = d_i / m_i$

System test quality:

$$STQ(S) = \frac{\sum_{i=1,n} d_i}{\sum_{i=1,n} m_i}$$

STQ provides a measure of test suite quality

If contracts are used as oracles, STQ is a combined measure of test and contract quality

Mutation tools

muJava - http://ise.gmu.edu/~ofut/mujava/

Test Coverage

Coverage

How extensive is a test?

Coverage measures a percentage of elements of a certain kind exercised by a test suite.

"Achieving coverage" means reaching 100% for the chosen criterion

Purposes of measuring code coverage

Code coverage analysis makes it possible to:

- Find sections of code not exercised by test cases
- Create additional test cases that exercise properties not previously tested
- Obtained a (hoped for) estimate of test suite quality

Code coverage analyzer

A code coverage analyzer is a tool that automatically computes the coverage achieved by a test suite

Steps involved:

1. Instrument source code by inserting trace instructions that write to a trace file
2. Run tests
3. Parse trace file to produce a coverage report

Standard measures of coverage

Instruction coverage, branch coverage etc.

Instruction (statement) coverage

Percentage of instructions (executable statements) executed

- Disadvantage: insensitive to control structures

Branch (or "decision") coverage

Percentage of conditionals whose boolean expression has evaluated to both true and false

- Disadvantage: insensitive to individual components of boolean expression
- The most commonly used in practice (easy to achieve)

Condition coverage

Percentage of elementary boolean conditions that have evaluated to both true and false

- Disadvantage: Not all combinations

Example:

if a **and** b **then** ...

Multiple-condition coverage

Percentage of combinations of true and false values of elementary boolean conditions

- Disadvantage: difficult to achieve, widely different number of tests for similar expressions

Examples*

a **and** b **and** (c **or** (d **and** e))

1.	F	–	–	–	–
2.	T	F	–	–	–
3.	T	T	F	F	–
4.	T	T	F	T	F
5.	T	T	F	T	T
6.	T	T	T	–	–

((a **or** b) **and** (c **or** d)) **and** e

1.	F	F	–	–	–
2.	F	T	F	F	–
3.	F	T	F	T	F
4.	F	T	F	T	T
5.	F	T	T	–	F
6.	F	T	T	–	T
7.	T	–	F	F	–
8.	T	–	F	T	F
9.	T	–	F	T	T
10.	T	–	T	–	F
11.	T	–	T	–	T

*From Steve Cornett

Modified Condition/Decision coverage (MC/DC)

Percentage of combinations of elementary conditions that have evaluated to both true and false value for one of the conditions, with all the other conditions unchanged, leading to both true and false for the overall expression (decision)

Example:

(a **or** b) **and** (c **or not** d)

- Advantage: easier to achieve than multiple condition
- The standard at Boeing

Generalization: predicate coverage

A predicate is covered if at least one test run makes it true and at least one makes it false

Example:

$$a \lor b \lor (f(x) \land x > 0)$$

is covered by the following two test cases:

- {a=true; b=false; f(x)=false; x=1}
- {a=false; b=false; f(x)=true; x=-1}

Clause coverage (CC)

Satisfied if for every clause of the predicate at least one test run makes the clause true and at least one to false

Example:

$$x > 0 \vee y<0$$

Clause coverage is achieved by:

- {x=-1; y=-1}
- {x=1; y=1}

Does clause coverage imply predicate coverage?

No: consider following variant:

- {x=-1; y=2}
- {x=1; y=1}

Combinatorial coverage (CoC)

The test runs must include all possible combination of clause values

Example:

$((A \vee B) \wedge C)$

	A	B	C	$((A \vee B) \wedge C)$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Determination

A clause c_M (called major clause) of a predicate p determines p if the remaining clauses $c_m \in p$, $m \neq M$ (called minor clauses) have such values that changing the value of c_M changes the value of p. c_M will be the active clause.

Example:

$p = a \vee b$

	a	b
$c_M = a$	T F	f f
$c_M = b$	f f	T F

Correlated Active Clause Coverage (CACC)*

For each $p \in P$ and each major clause $c_M \in C_p$, choose minor clauses c_m, $m \neq M$ so that c_M determines p

The test runs must include at least one that makes c_M true and one that makes it false

The values chosen for the minor clauses do not need to be the same for these two runs

Example:

$p = a \wedge (b \vee c)$

We satisfy CACC for a if we choose one test case out of rows 1, 2, or 3, and one out of rows 5, 6, or 7.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

***Variant of "MCDC"**

Restricted Active Clause Coverage (RACC)

For each $p \in P$ and each major clause $c_M \in C_p$, choose minor clauses c_m, $m \neq M$ so that c_M determines p.

The test runs must include one that makes c_M true and and one that makes it false

The values chosen for the minor clauses must be the same for these two runs

Example:

$p = a \wedge (b \vee c)$

We satisfy RACC for a if we choose (1,5), or (2,6), or (3,7).

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

Path coverage

Percentage of paths taken

A path is a unique sequence of branches from routine entry to exit

- Disadvantage: exponential
- Does not take loops into account

Can be impossible to achieve 100%

```
if c then a end
other_instructions
if c then b end
```

(if other_instructions do not affect c.)

Limits of coverage measures

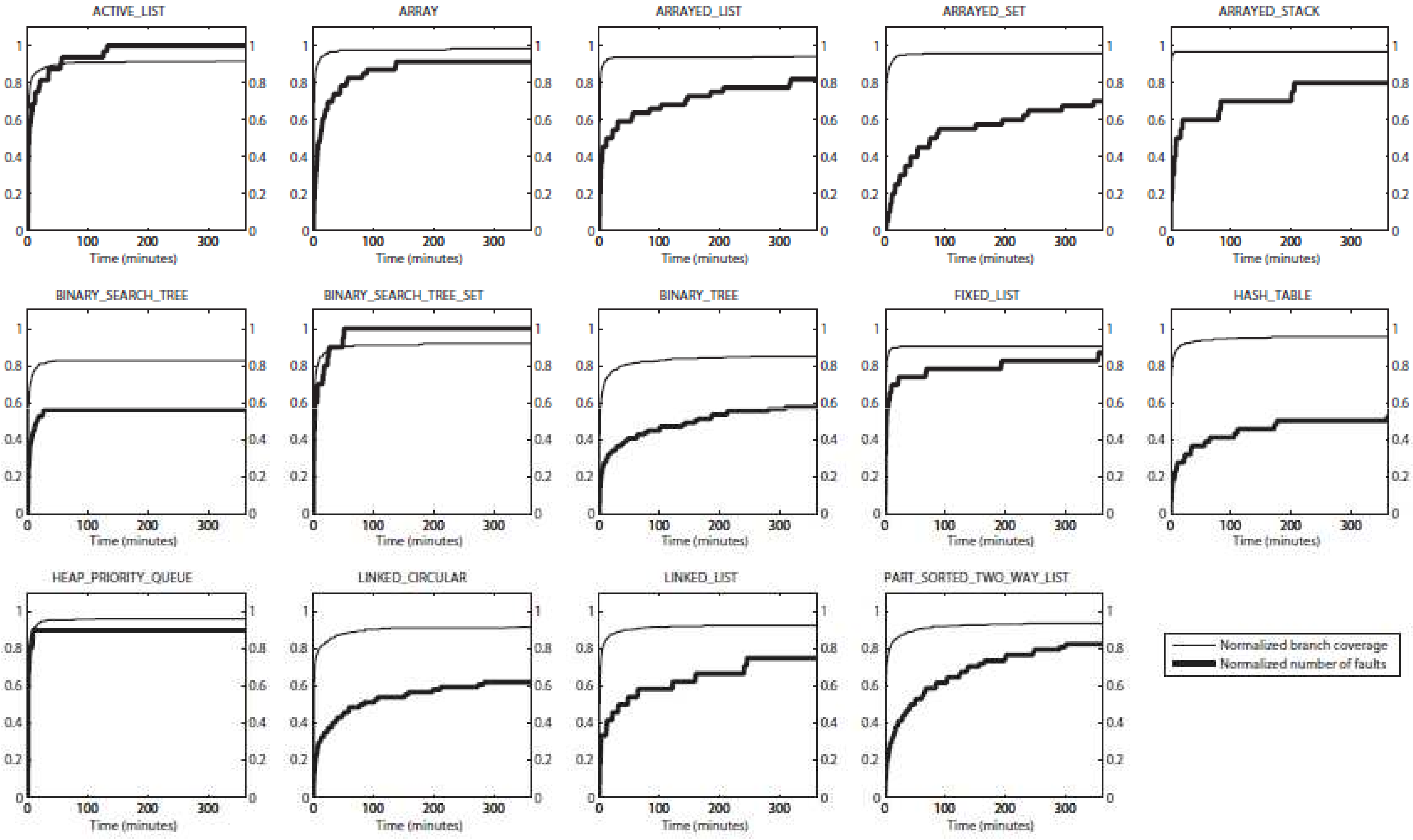


Figure 8: Median of the branch coverage level and median of the normalized number of faults for each class over time

Yi Wei, M. Oriol, B. Meyer (2009)

Code coverage tools

Emma
- Java
- Open-source
- http://emma.sourceforge.net/

JCoverage
- Java
- Commercial tool
- http://www.jcoverage.com/

NCover
- C#
- Open-source
- http://ncover.sourceforge.net/

Clover, Clover.NET
- Java, C#
- Commercial tools
- http://www.cenqua.com/clover/

See also http://www.codecoveragetools.com/

Dataflow-oriented testing

Focuses on how variables are defined, modified, and accessed throughout the run of the program

Looks for faults resulting from wrong paths between a definition of a variable in the code and certain uses of that variable

Access-related potential bugs

Examples:

- Using an uninitialized variable
- Assigning to a variable more than once without an intermediate access
- (C++) Deallocating a variable before it is initialized
- (C++) Deallocating a variable before it is used
- Modifying an object more than once without accessing it

Types of access to variables

Definition (def) : change value of variable (constructor, assignment, procedure)

Use: read value of variable

- Computational use (c-use): in a computation
- Predicative uses (p-use): in a test
- Kill: instruction that results in a variable being deallocated, undefined, released or no longer visible

Examples:

- z := x * y // c-use of y; c-use of x; def of z
- **if** x > 0 **then** ... // p-use of x

Data flow graph

All measures of dataflow coverage are defined in terms of the data flow graph

- Sub-path: sequence of consecutive nodes
- Path: sub-path starting at entry node and ending at exit node

Path properties:

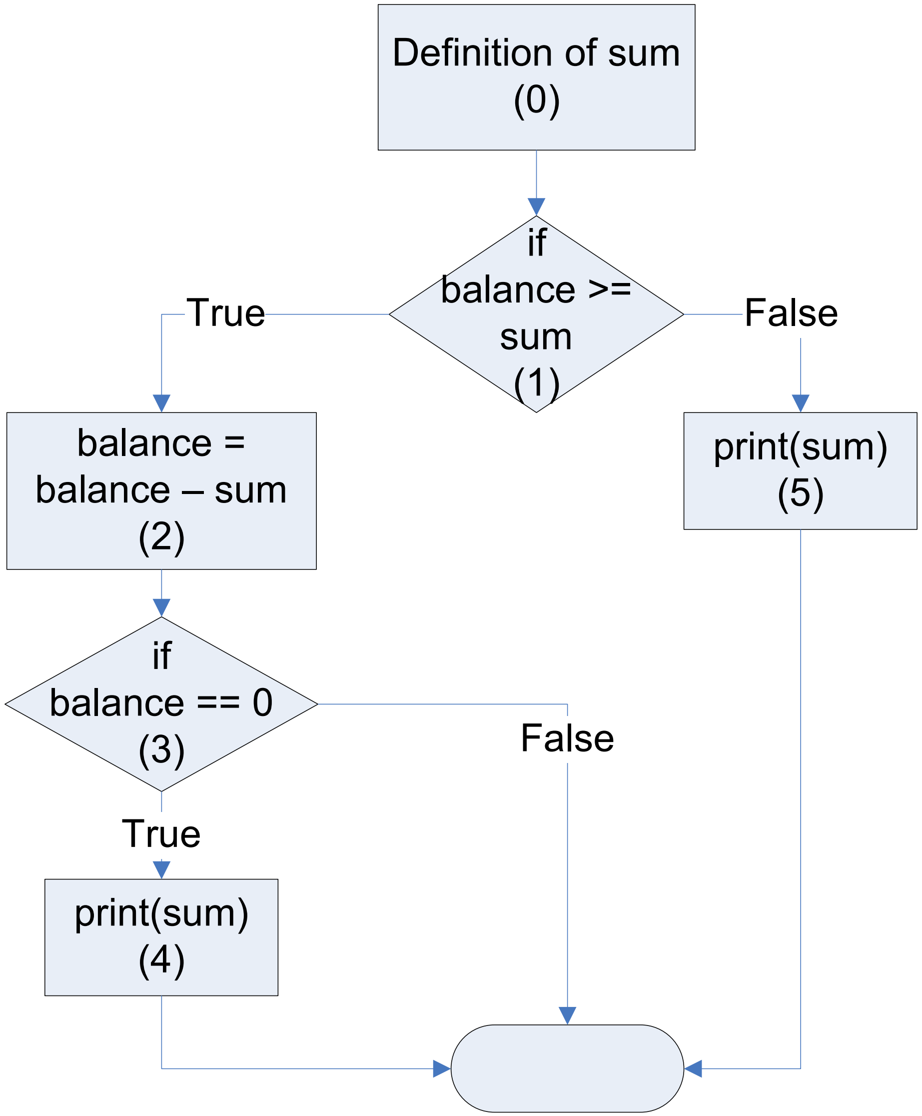
- A sub-path is *def-clear* for a variable *v* if it contains no definition of *v*
- A sub-path *p* starting with a def of variable *v* is a *du-path* for *v* if *p* is def-clear for *v* except for the first node, and *v* encounters either a c-use in the last node or a p-use along the last edge of *p*

Example: source code

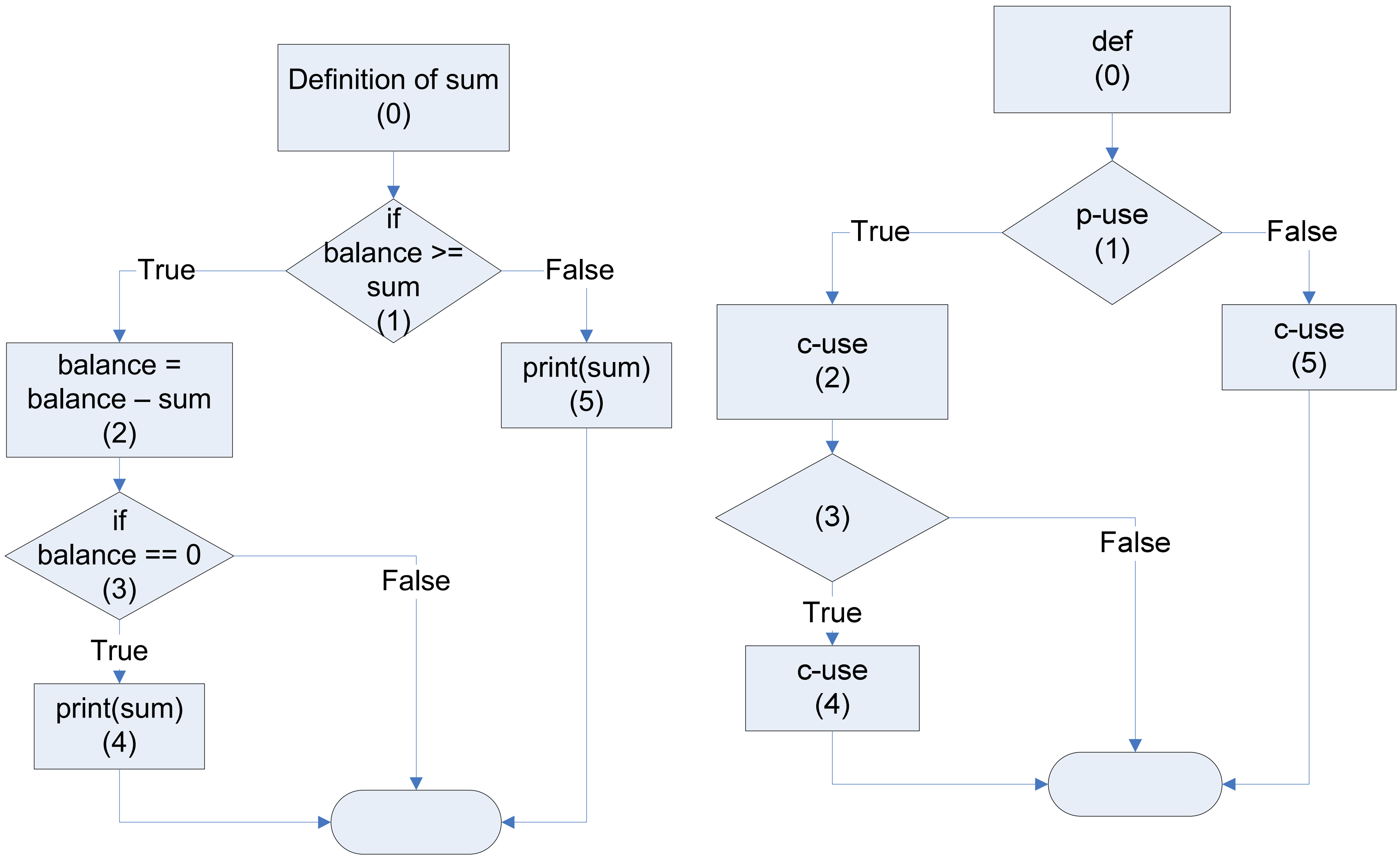
```
class ACCOUNT feature
	balance: INTEGER

	withdraw (sum: INTEGER)
		do
			if balance >= sum then
				balance = balance - sum
				if balance = 0 then
					io.put_string (“There were only " + sum +
						"CHF in the account. The account is now empty.%N")
				end
			else
				io.put_string (“There is less than ” + sum + “CHF in the account.”)
			end
		end
end
```

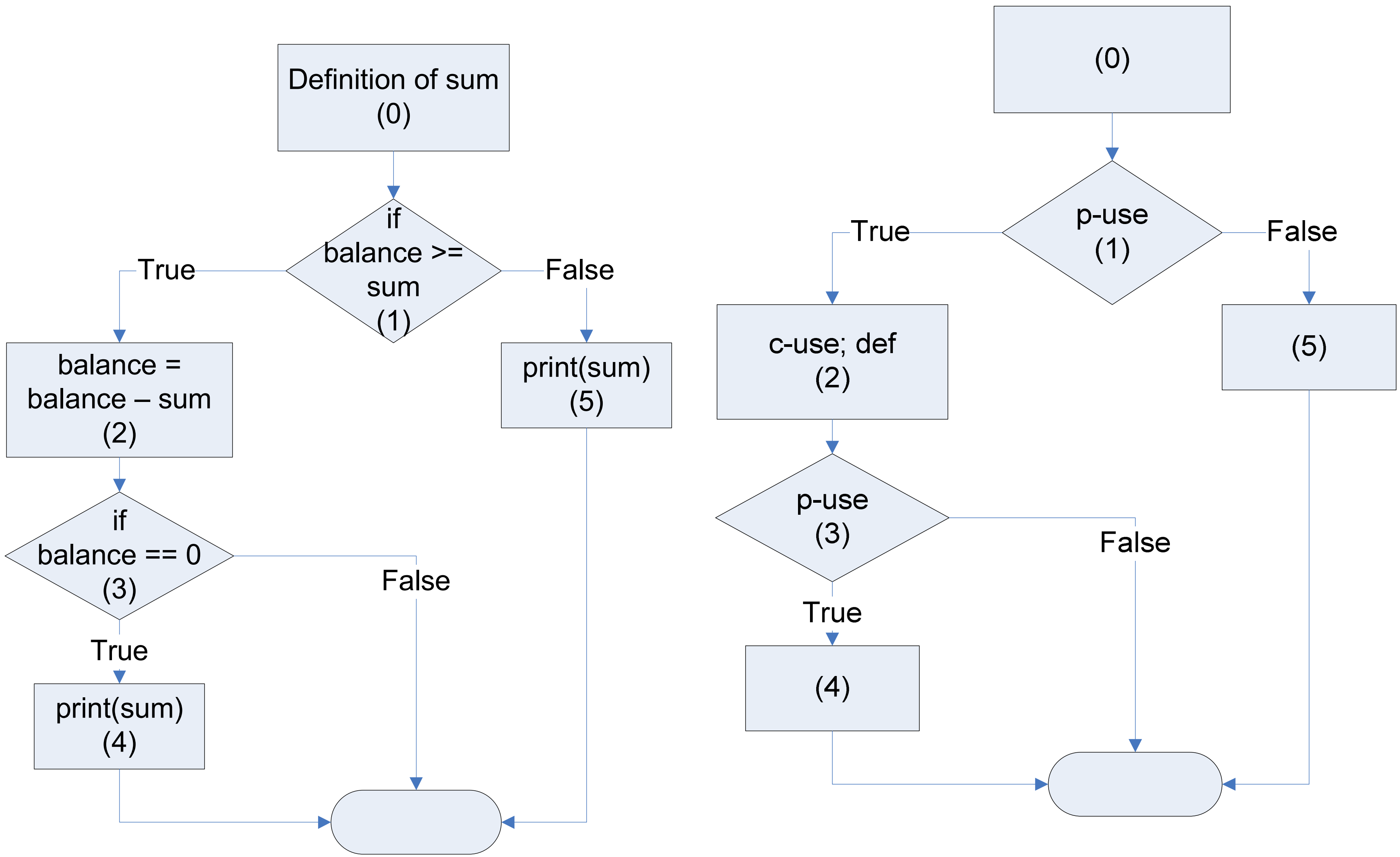
Control flow graph for `withdraw`



Data flow graph for `sum` in `withdraw`



Data flow graph for **balance** in **withdraw**



Dataflow coverage criteria

all-defs: execute *at least one* def-clear sub-path between *every* definition of every variable and *at least one* reachable use of that variable.

all-p-uses: execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable p-use of that variable.

all-c-uses: execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable c-use of the respective variable.

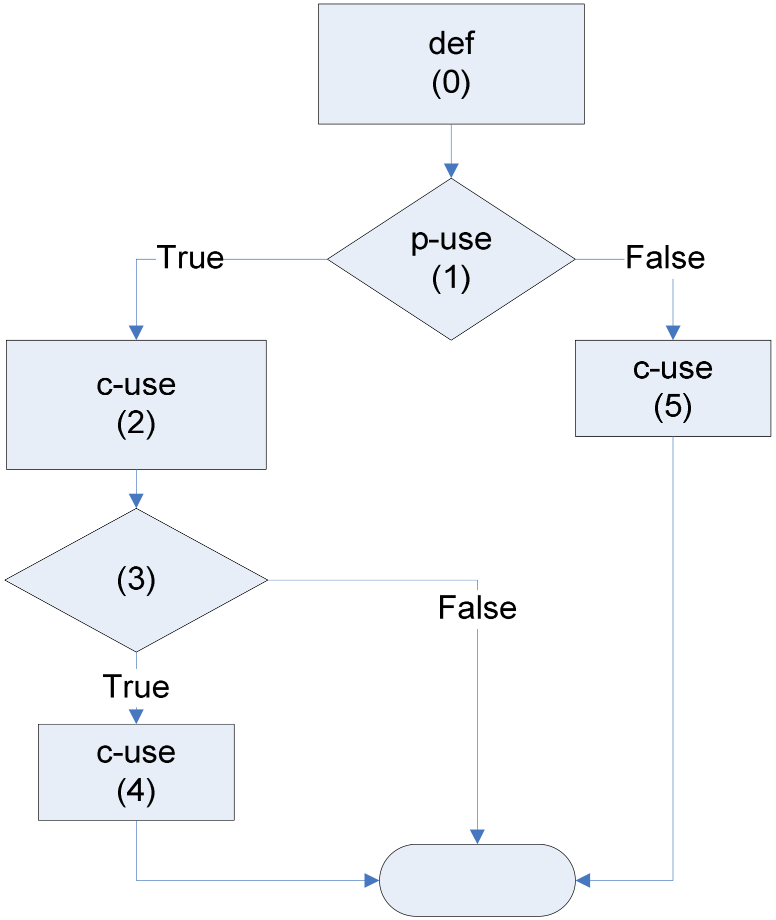
Dataflow coverage criteria (continued)

all-c-uses/some-p-uses: apply all-c-uses; then if any definition of a variable is not covered, use p-use

all-p-uses/some-c-uses: symmetrical to all-c-uses/some-p-uses

all-uses: execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable use of that variable

Dataflow coverage criteria for **`sum`** in **`withdraw`**



all-defs: *at least one* def-clear sub-path between *every* definition and *at least one* reachable use

(0,1)

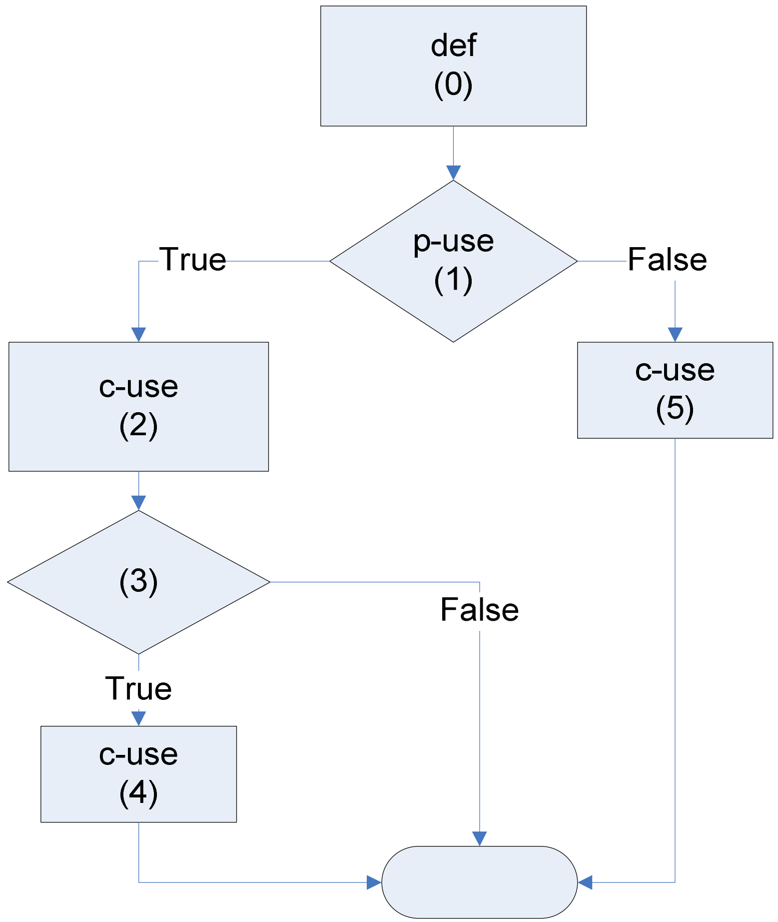
all-p-uses: *at least one* def-clear sub-path from *every* definition to *every* reachable p-use

(0,1)

all-c-uses: *at least one* def-clear sub-path from *every* definition to *every* reachable c-use

(0,1,2); (0,1,2,3,4); (0,1,5)

Dataflow coverage criteria for `sum` in `withdraw` (cont.)



all-c-uses/some-p-uses: apply all-c-uses; then if any definition of a variable is not covered, use p-use

(0,1,2); (0,1,2,3,4); (0,1,5)

all-p-uses/some-c-uses: symmetrical to all-c-uses/some-p-uses

(0,1)

all-uses: *at least one* def-clear sub-path from *every* definition to *every* reachable use

(0,1); (0,1,2);(0,1,2,3,4);(0,1,5)

Specification coverage

Predicate: an expression that evaluates to a boolean value

- e.g.: $a \vee b \vee (f(x) \wedge x > 0)$

Clause: a predicate that does not contain any logical operator

- e.g.: $x > 0$

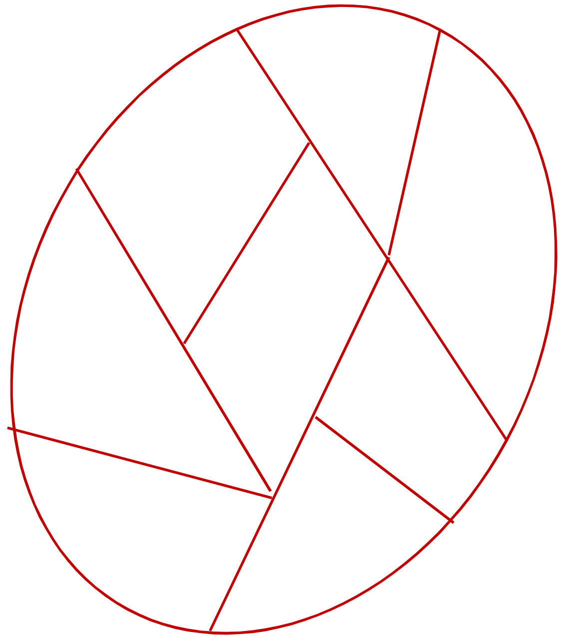
Notation:

- P = set of predicates
- C_p = set of clauses of predicate p

If specification expressed as predicates on the state, specification coverage translates to predicate coverage

Partition testing

If we cannot test *every* value of the input domain, how do we choose inputs?



A partition divides input space into subsets (equivalence classes) satisfying:

- Completeness (covers all input)
- Disjointness

Expectation (hope) behind partition testing:

- If any value in the subset produces a failure, any other value in the subset does too

Examples of partition testing

Boundary value analysis

Special values testing

Choosing values

Each Choice (EC):

- Test suite includes at least one test case from every equivalence class for every input

All Combinations (AC):

- Test suite includes at least one test case from every combination of equivalence classes for all inputs

Partition testing

Applicable to *all levels* of testing: unit, class, integration, system, etc.

Based only on the *input space* of the program, not the implementation (i.e. black box concept)

Many testers intuitively apply a similar concept

- 9 -

Contract-based & random testing

Test automation

Testing is so difficult and time consuming…

So why not do it automatically?

What is most commonly meant by "automated testing" currently is automatic test *execution*

But actually…

What can we automate?

Test execution

- Run test suite without step-by-step actions
- Should be parameterizable
- Recover from failures (multi-process architecture)

Test management

- Let user adapt process to needs and preferences
- Save tests for regression testing

Test result evaluation (applying oracles)

- Classifying tests as pass/no pass
- Other info about test results

What can we automate?

Regression testing

- Re-run previous tests
- May require minimization

Estimation of test suite quality

- Report a measure of code coverage
- Other measures of test quality
- Feed this estimation back to the test generator

Test generation

- Generation of test data (objects used as targets or parameters for feature calls)
- Procedure for selecting the objects used at runtime
- Generation of test code (code for calling the features under test)

"Push-button testing"

Never write a test case, a test suite, a test oracle, or a test driver

Automatically generate

- Objects
- Feature calls
- Evaluation and saving of results

The user must only specify the system under test and the tool does the rest (test generation, execution and result evaluation)

Testing strategy

How do we plan and structure the testing of a large program?

- Who is testing?
  - Developers / special testing teams / customer
  - It is hard to test your own code
- What test levels do we need?
  - Unit, integration, system, acceptance, regression test
- How do we do it in practice?
  - Manual testing
  - Testing tools
  - Automatic testing

xunit

The generic name for any test automation framework for unit testing

- Test automation framework – provides all the mechanisms needed to run tests so that only the test-specific logic needs to be provided by the test writer

Implemented in all the major programming languages:

- JUnit – for Java
- cppunit – for C++
- SUnit – for Smalltalk (the first one)
- PyUnit – for Python
- vbUnit – for Visual Basic

JUnit: **resources**

Unit testing framework for Java

Written by Erich Gamma and Kent Beck

Open source (CPL 1.0), hosted on SourceForge

Current version: 4.0

Available at: www.junit.org

Very good introduction for JUnit 3.8: Erich Gamma, Kent Beck, *JUnit Test Infected: Programmers Love Writing Tests,* available at http://junit.sourceforge.net/doc/testinfected/testing.htm

For JUnit 4.0: Erich Gamma, Kent Beck, *JUnit Cookbook,* available at http://junit.sourceforge.net/doc/cookbook/cookbook.htm

JUnit: Overview

Provides a framework for running test cases

Test cases

- Written manually
- Normal classes, with annotated methods

Input values and expected results defined by the tester

Execution is the only automated step

How to use JUnit

Requires JDK 5

Annotations:

- **@Test** for every method that represents a test case
- **@Before** for every method that will be executed before every **@Test** method
- **@After** for every method that will be executed after every **@Test** method

Every **@Test** method must contain some check that the actual result matches the expected one – use asserts for this

- **`assertTrue`, `assertFalse`, `assertEquals`, `assertNull`, `assertNotNull`, `assertSame`, `assertNotSame`**

Example: basics

```
package unittests;

import org.junit.Test; // for the Test annotation
import org.junit.Assert; // for using asserts
import junit.framework.JUnit4TestAdapter; // for running

import ch.ethz.inf.se.bank.*;

public class AccountTest {
        @Test public void initialBalance() {
                Account a = new Account("John Doe"
                Assert.assertEquals(
                        "Initial balance must be the one set through the
constructor",
                        1000,
                        a.getBalance());
        }

        public static junit.framework.Test suite() {
                return new JUnit4TestAdapter(AccountTest.class);
        }
}
```

To declare a routine as a test case

To compare the actual result to the expected one

Required to run JUnit4 tests with the old JUnit runner

Example: set up and tear down

```
package unittests;

import org.junit.Before; // for the Before annotation
import org.junit.After; // for the After annotation
// other imports as before…

public class AccountTestWithSetUpTearDown {

        private Account account;

        @Before public void setUp() {
                account = new Account("John Doe", 30, 1, 1000);
        }
        @After public void tearDown() {
                account = null;
        }
        @Test public void initialBalance() {
                Assert.assertEquals("Initial balance must be the one set through the
constructor",
                        1000,
                        account.getBalance());
        }
        public static junit.framework.Test suite() {
                return new JUnit4TestAdapter(AccountTestWithSetUpTearDown.class);
        }
}
```

Must make account an attribute of the class now

To run this routine before any @Test method

To run this method after any @Test method

@BeforeClass, @AfterClass

A routine annotated with @BeforeClass will be executed once, before any of the tests in that class is executed.

A routine annotated with @AfterClass will be executed once, after all of the tests in that class have been executed.

Can have several @Before and @After routines, but only one @BeforeClass and @AfterClass routine respectively.

Checking for exceptions

Pass an argument to the @Test annotation stating the type of exception expected:

```
@Test(expected=AmountNotAvailableException.class) public void overdraft ()
throws AmountNotAvailableException {
                    Account a = new Account("John Doe", 30, 1, 1000);
                    a.withdraw(1001);
          }
```

The test will fail if a different exception is thrown or if no exception is thrown.

Setting a timeout

Pass an argument to the @Test annotation setting a timeout period in milliseconds. The test fails if it takes longer than the given timeout.

```
@Test(timeout=1000) public void testTimeout () {
        Account a = new Account("John Doe", 30, 1, 1000);
        a.infiniteLoop();
    }
```

Testing is tedious!

From a survey of 240 software companies in North America and Europe:

- 8% of companies release software to beta sites without any testing.
- 83% of organizations' software developers don't like to test code.
- 53% of organizations' software developers don't like to test their own code because they find it tedious.
- 30% don't like to test because they find testing tools inadequate.

Parts of a test case

Create input

- Instructions
- Data

Execute tests

Evaluate result (Oracle)

- Compare
- Compute

(Tear down)

Degress of automation

No automation

Automated execution

Automated input generation

Automated oracle

Challenges of automated testing

Vast input space

Is this input good?

- Precondition

Is this output good?

- Postcondition

The quality of the test is only as good as the quality of the assertions

Vast input space

- Input space typically unbounded
- Even when finite, very large
- Exhaustive testing impossible
- Number of test cases increases exponentially with number of input variables

```
foo (c: CHARACTER)
    do
        ...
    end

bar (c1: CHARACTER;
     c2: CHARACTER)
    do
        ...
    end
```

Automatic testing tools

- TestEra (MIT)
- Korat (MIT)
- AutoTest (ETH)

AutoTest

Fully automated testing framework

- Actual strategies are extensions

Based on Design By Contract

Robust execution

Integration of manual unit tests

AutoTest: three parts

1. Generated tests

2. Extracted tests

3. Manual tests

AutoTest: strategies

Random Strategy

- Use random input

Planning Strategy

- Employ information from postcondition to satisfy preconditions

...

AutoTest: automatic test framework

Ilinca Ciupa
Andreas Leitner
Yi Wei

- 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, +/-10, max and min)
- Add manual tests if desired
- Any test (manual or automated) that fails becomes part of the test suite

Minimization through dynamic slicing

```
auto_test system.ace –t 120 ACCOUNT CUSTOMER
```

```
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 strategies

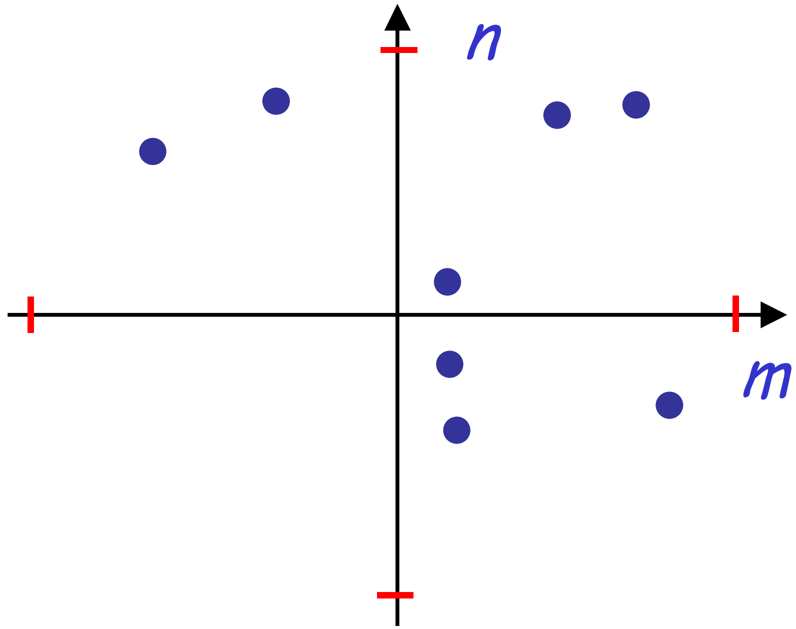
- Object pool
  - Get objects through creation procedures (constructors)
  - Diversify through procedures
- Routine arguments
  - Basic values: heuristics for each type
  - Objects: get from pool
- Test all routines, including inherited ones ("Fragile base class" issue)

Adaptive Random Testing (Chen et al.)

Conjecture:

Random testing may find faults faster if inputs evenly spread

So far: basic types

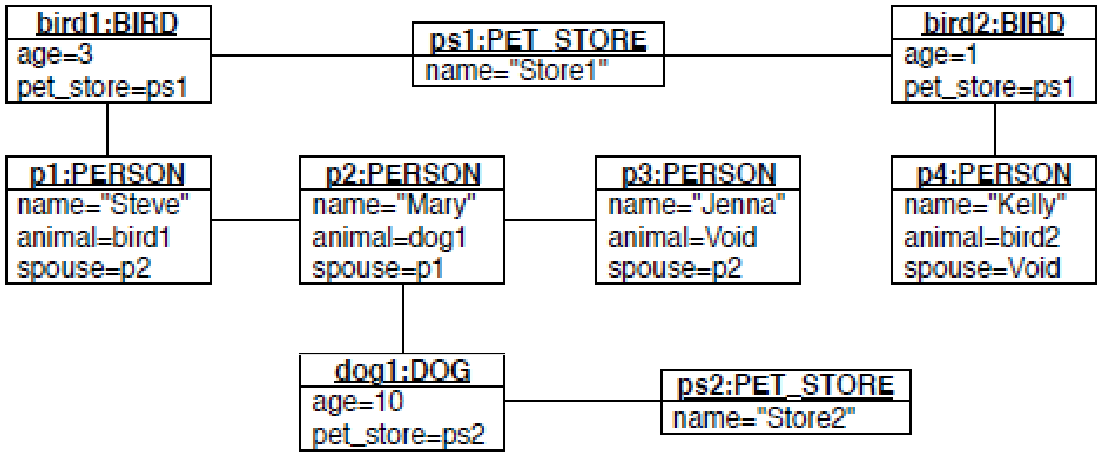


Our contribution: extend this to objects

Need to define notion of distance between objects

Object distance

Ilinca Ciupa
(ICSE 2007)



$p \leftrightarrow q \triangleq$

combination (

type_distance $(p.type, q.type)$,

field_distance (p, q),

recursive_distance (

$\{[p.r \leftrightarrow q.r] \mid r \in$

Reference_attributes $\}$)

ART vs pure random

Results so far:

- Does not find more faults
- Does not find faults faster
- Finds other faults!

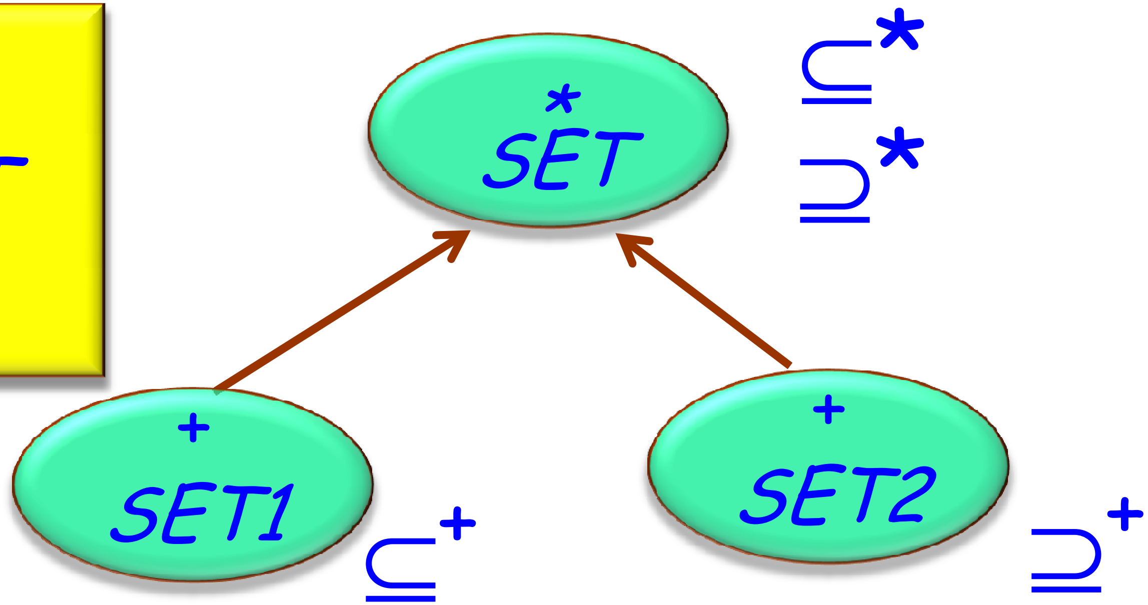
Random testing: example bug found

Bernd Schoeller

Test:

s1, s2 : SET

s2 $\subseteq$ *s1*



*: Deferred

$^+$: Effective

The testbed: EiffelBase

- Version of September 2005
- 20-year history
- Showcase of Eiffel technology
- About 1800 classes, 20,000 SLOC
- Extensive (but not complete) contracts
- Widely used in production applications
- Significant faults remained

Some AutoTest results (random strategy)

Library	TESTS Total	TESTS Failed	ROUTINES Total	ROUTINES Failed
EiffelBase (Sep 2005)	40,000	3%	2000	6%
Gobo Math	1500	1%	140	6%

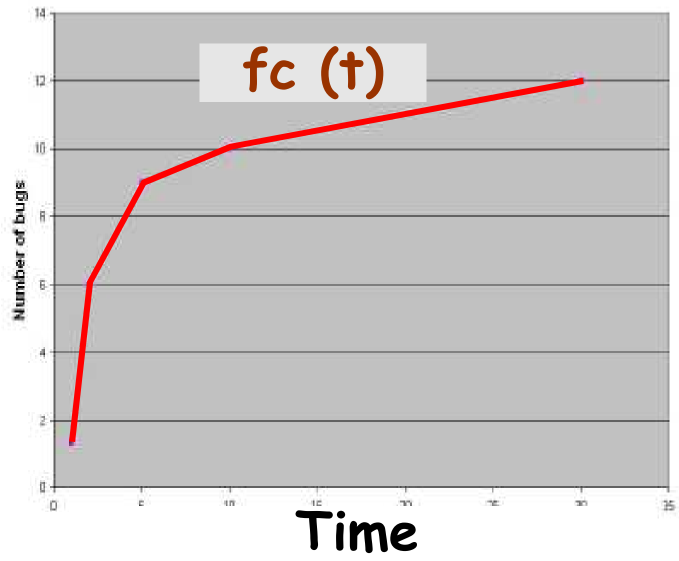
Testing results and strategy

"Smart" ideas not always better

Don't believe your intuition

Measure and assess objectively

Class *STRING*



Define good assessment criteria:

- Number of faults found
- Time to find all faults

Experimental law:

$$fc(t) = a \;/\; (1 + b * t)$$

Fault categories

Specification faults -- examples:

- Precondition:
  - Missing non-voidness precondition (will go away)
  - Missing min-max precondition
  - Too strong precondition
- Postcondition:
  - Missing
  - Wrong

Implementation faults -- examples:

- Faulty supplier
- Missing implementation
- Case not treated
- Violating a routine's precondition
- Infinite loop

Who finds what faults?

I.Ciupa, A. Leitner, M.Oriol, A. Pretschner (submitted)

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

AutoTest vs manual testers

On three classes (two with seeded bugs):

- Humans found 14 faults, AutoTest 9 of them
- AutoTest found 2 faults that humans did not (in large class)
- 3 faults not found by AutoTest found by 60% of humans (one is infinite loop)
- 2 faults not found by AutoTest are missing preconditions (void, min-max)

AutoTest vs user reports

On 39 EiffelBase classes:

- AutoTest found 85 faults,
  Plus 183 related to RAW_FILE,
  PLAIN_TEXT_FILE, DIRECTORY (total 268)
- 4 of these also reported by users
- 21 faults solely reported by users
- 30% of AutoTest-found bugs related to extreme values; users never report them

AutoTest finds only 1 out of 18 (5%) of implementation faults and 3 out of 7 specification faults

AutoTest bad at over-strong preconditions, wrong operator semantics, infinite loops, missing implementations

Users never find faulty suppliers (blame on client)

AutoTest developments

- Large-scale extensive tests, empirical assessment of criteria & strategies
- Comparison with manual efforts
- Complete integration with EiffelStudio IDE
- Background, unobtrusive, continuous testing
- Distributed cooperative testing ("Testi@home")

Test Extraction

Andreas Leitner, Arno Fiva

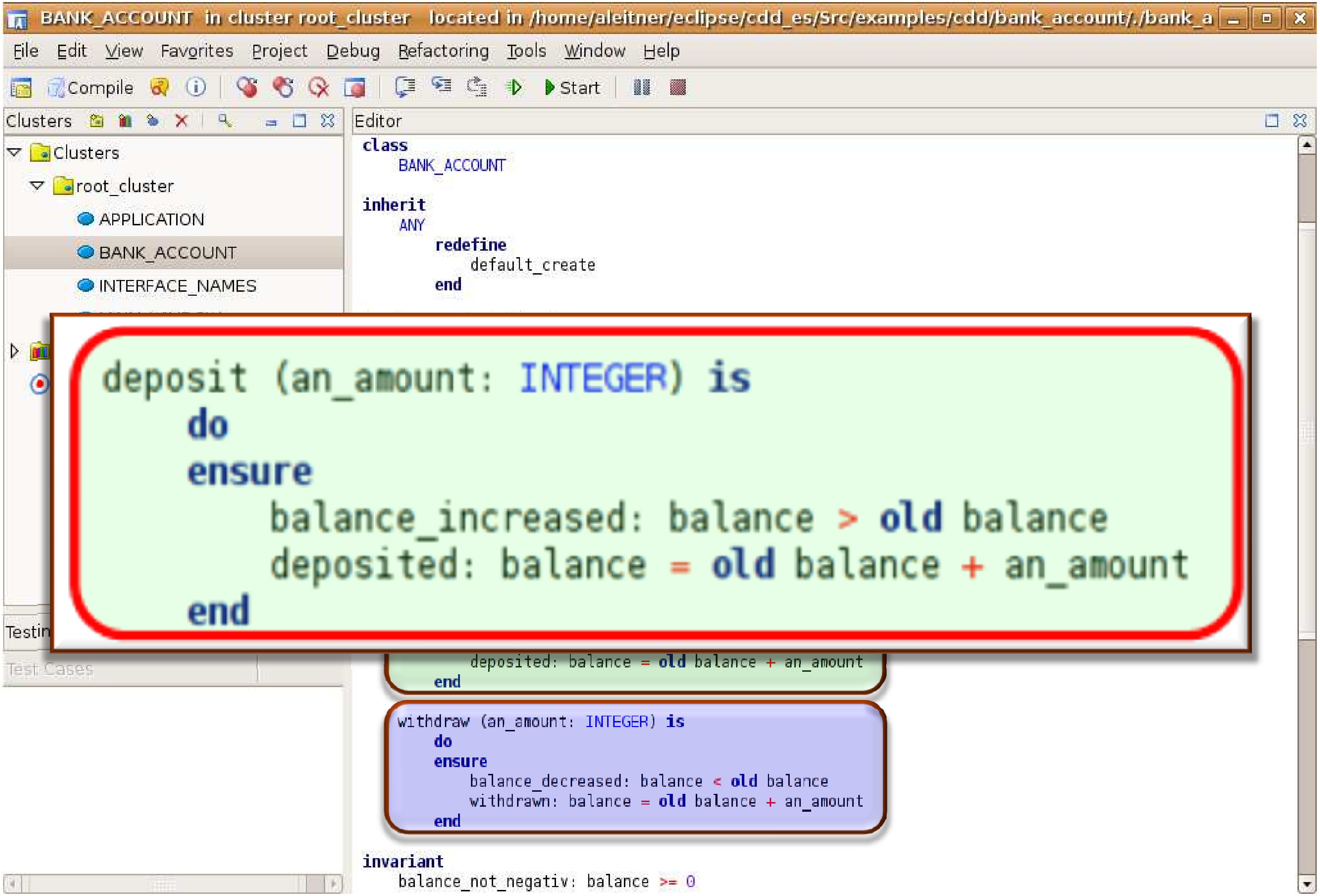
Like Test-Driven Development, but

- Tests derived from spec (contracts)
- Not the other way around!

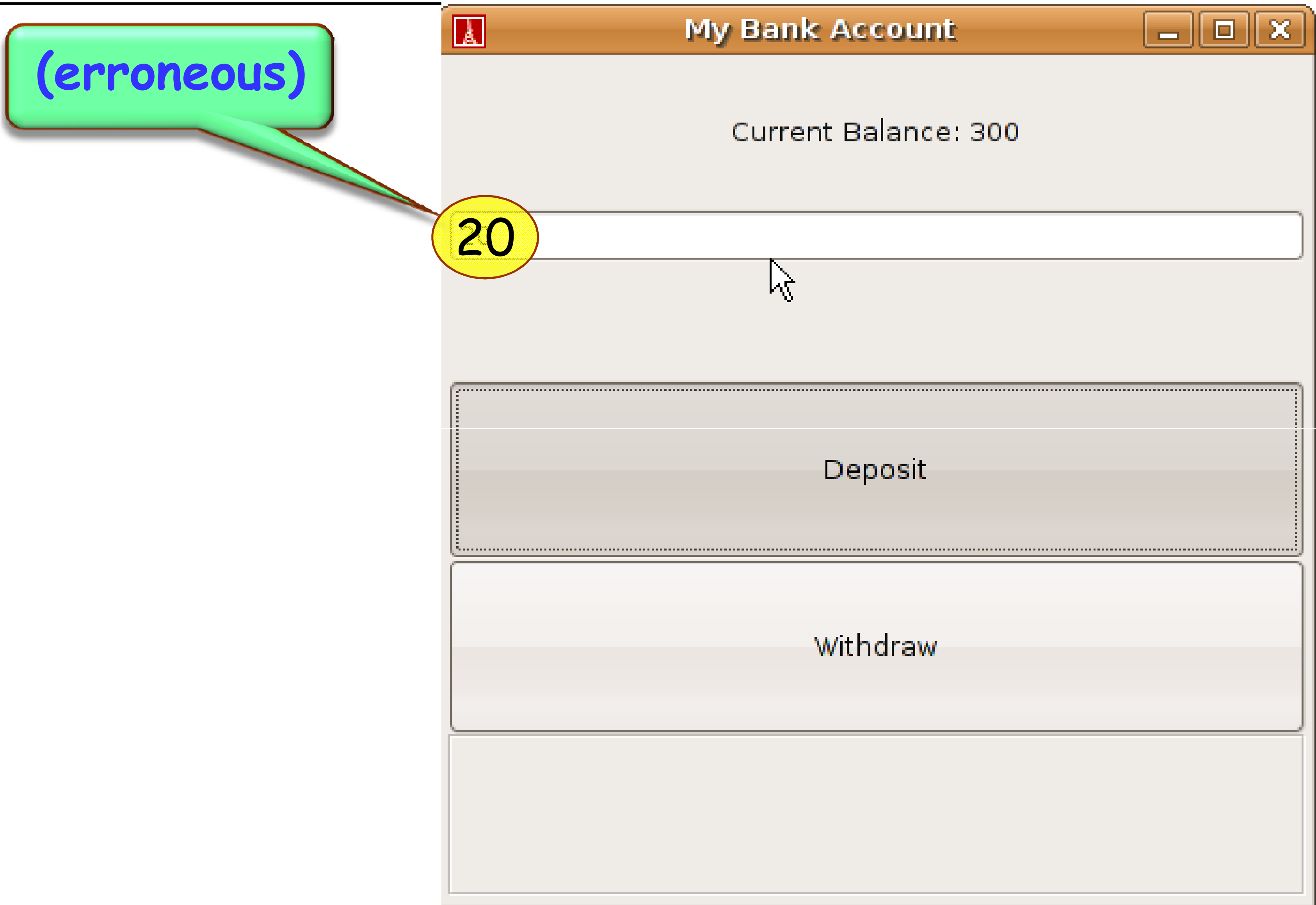
Record every failed execution, make it reproducible by retaining objects

Turn it into a regression test

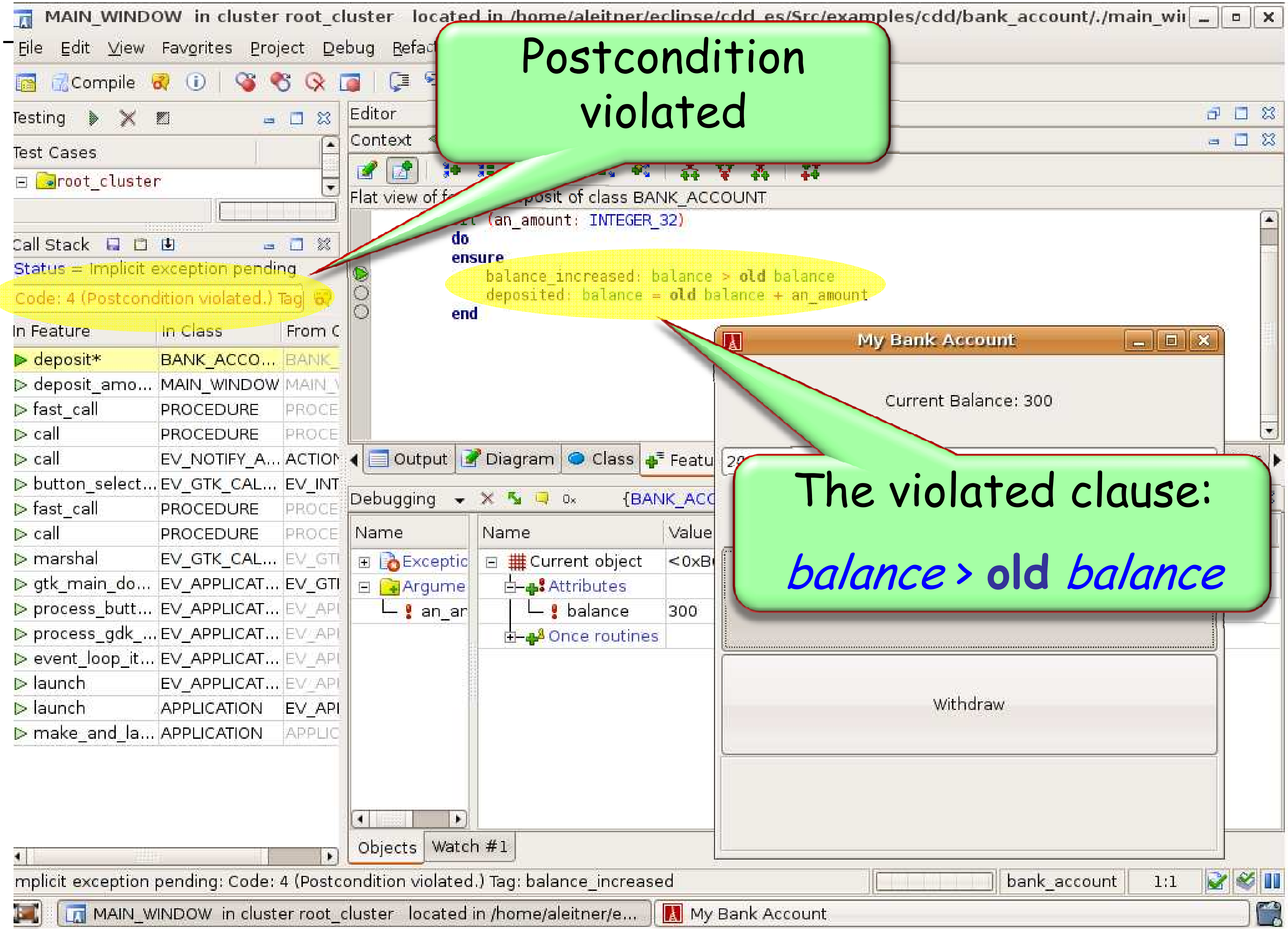
Specified but unimplemented routine



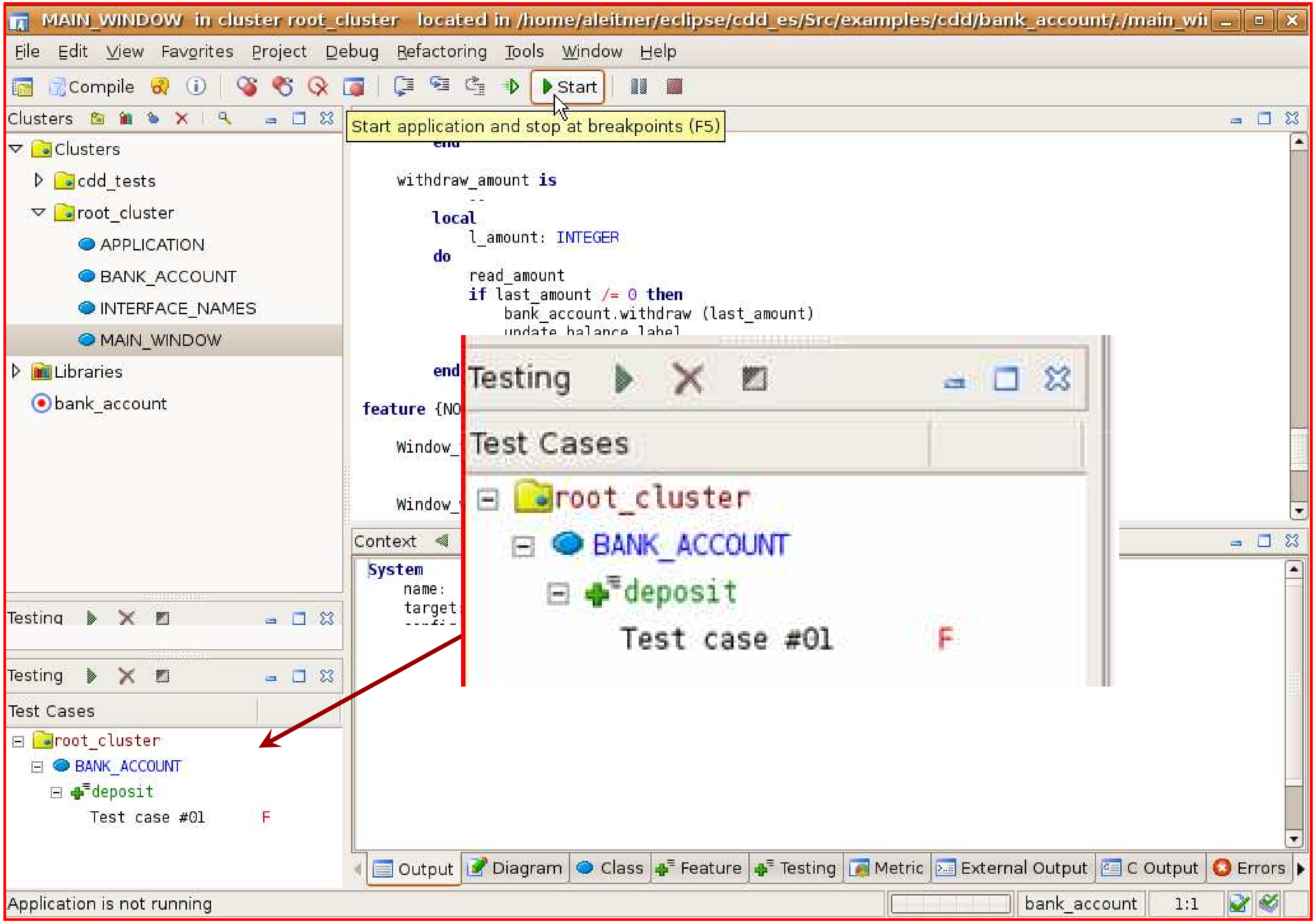
Running the system and entering input



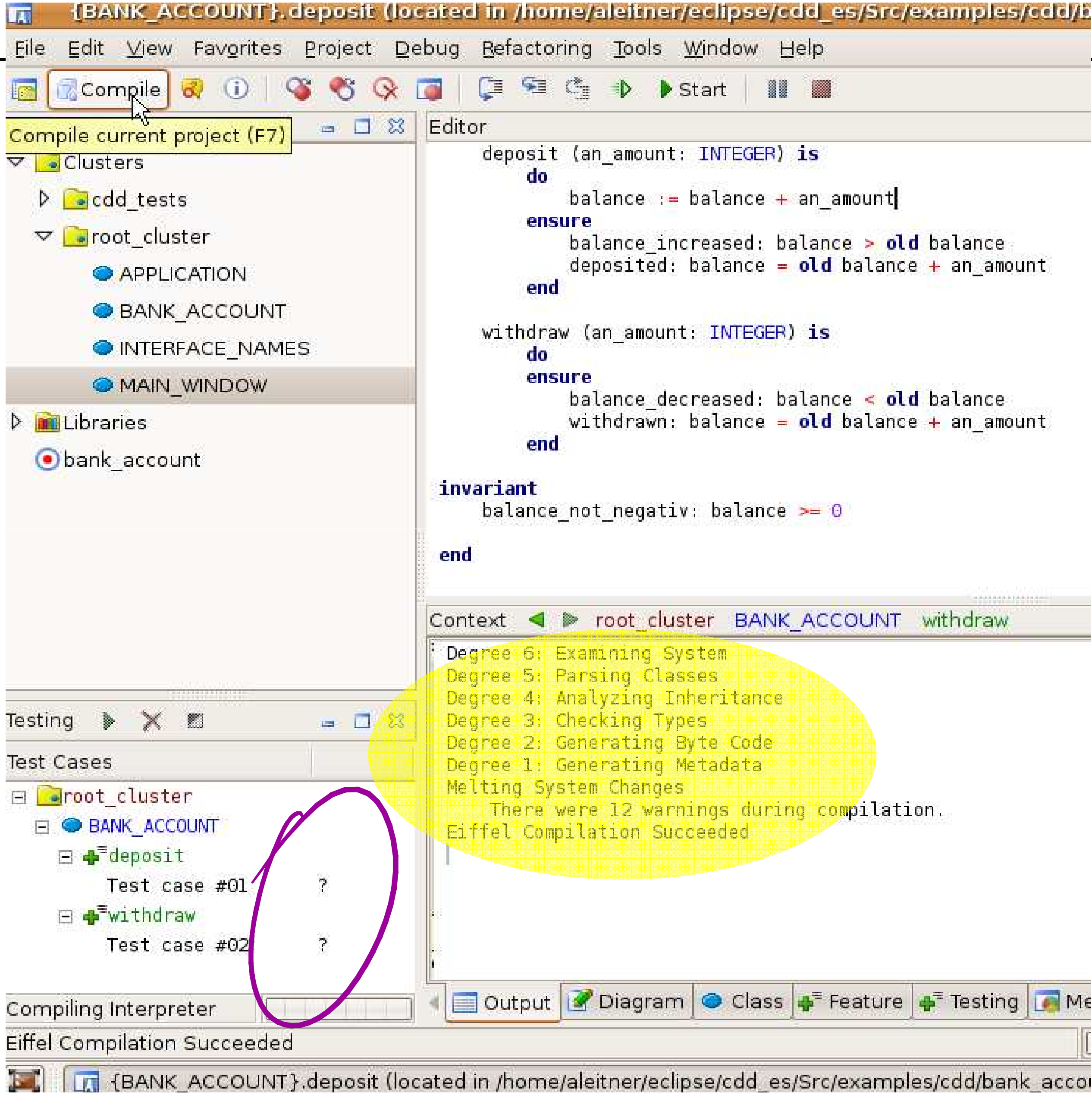
Error caught at run time as contract violation



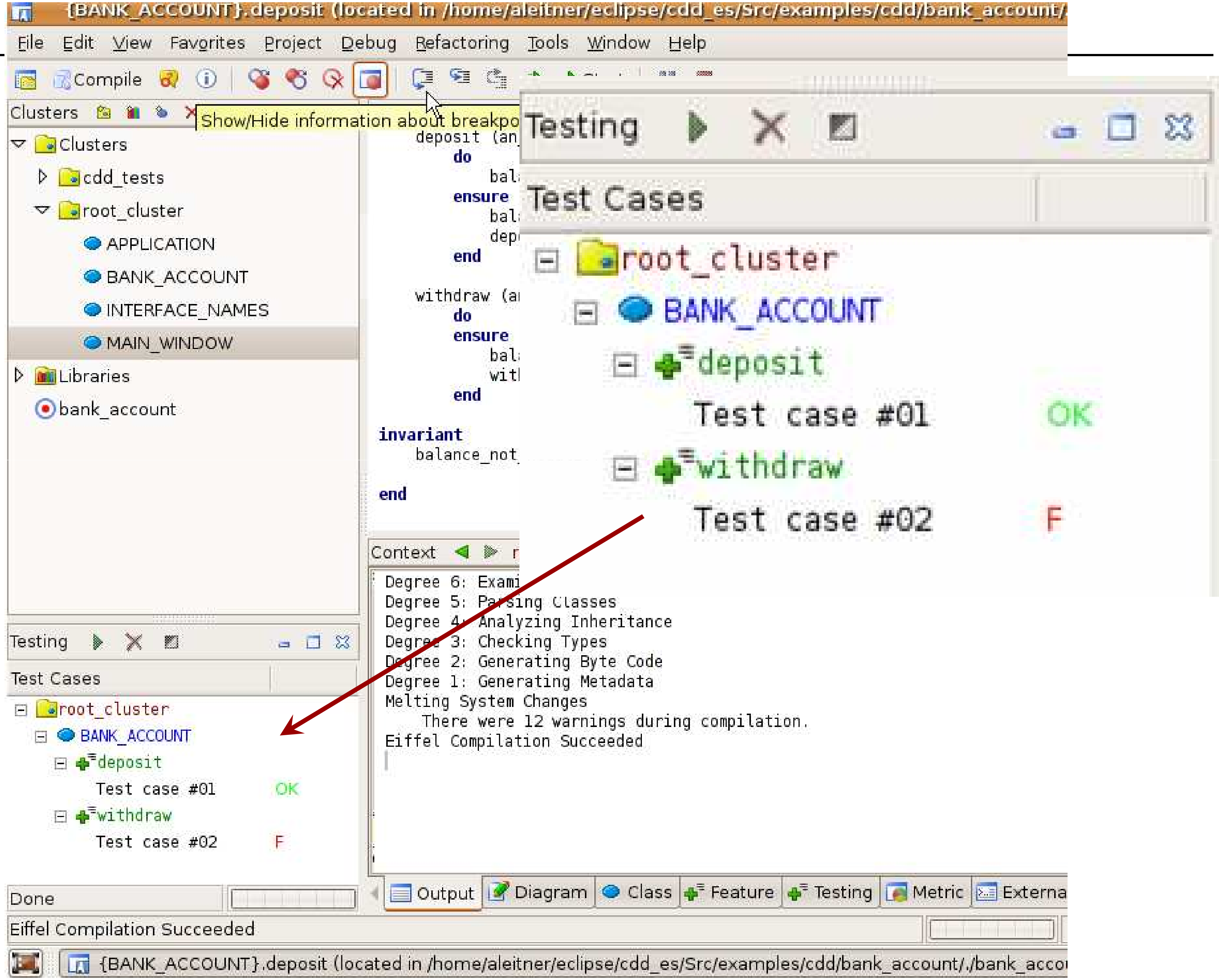
This has become a test case



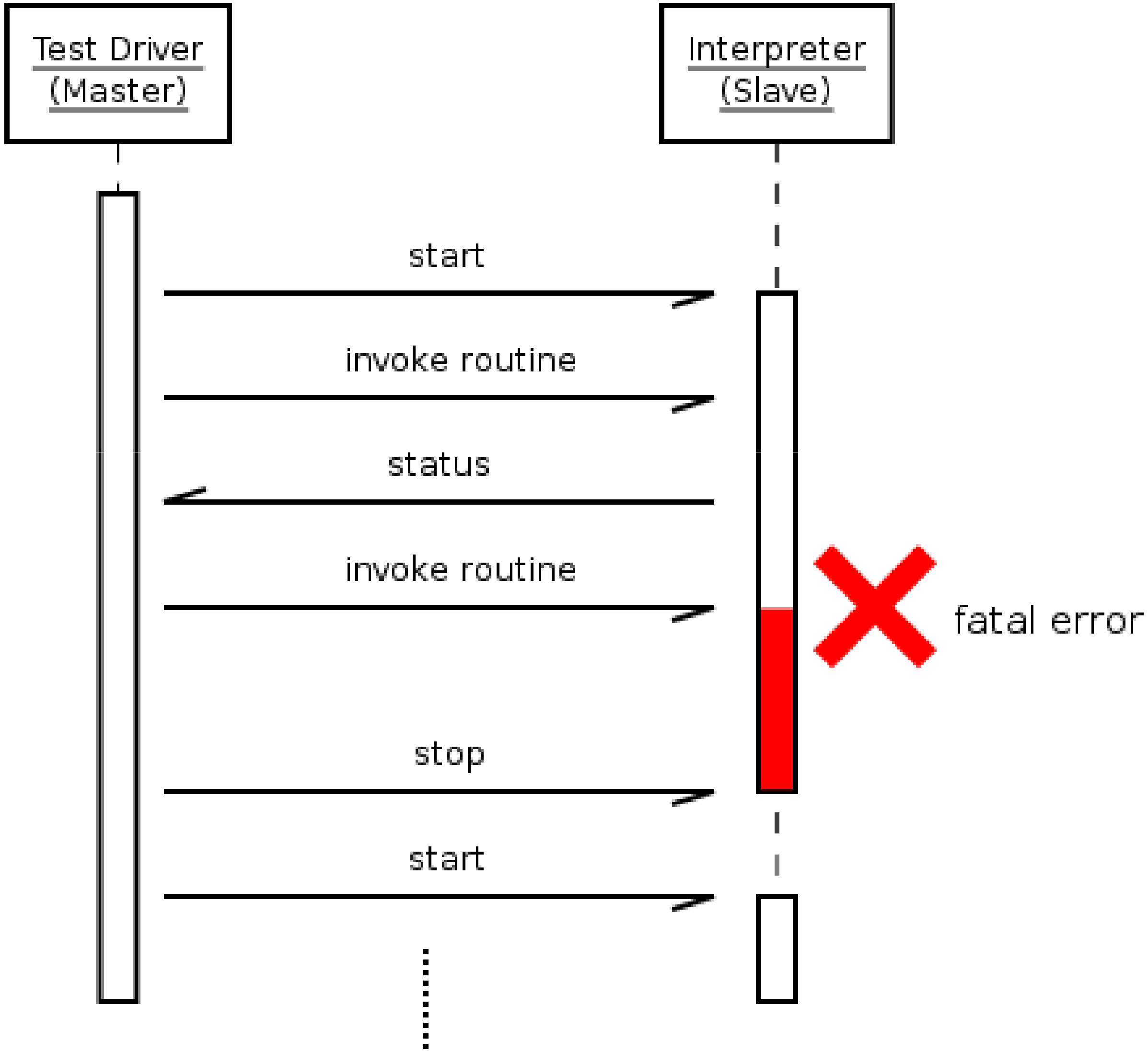
Correcting and recompiling



One fault corrected, the other not



AutoTest: robust execution



Hands-on!

Automated Testing:

A session with AutoTest

Hands-on with AutoTest: overview

- Tool: AutoTest
  - Implements Contract Based Testing
  - Chair of Software Engineering
  - Framework

Hands-on with AutoTest: **resources**

- Home page: se.inf.ethz.ch/people/leitner/auto_test/
- Documentation: se.inf.ethz.ch/people/leitner/auto_test/toc.html

Automatic test case generation: **assessment**

Testing is tedious

Automation can help

Challenges involved

Tools are getting there!

Automatic test case generation: bibliography

TestEra

D. Marinov and S. Khurshid: *TestEra: A Novel Framework for Automated Testing of Java Programs*. 16th IEEE Conference on Automated Software Engineering (ASE 2001), San Diego, CA. Nov 2001.

Korat

C. Boyapati, S. Khurshid and D. Marinov. Korat: *Automated Testing Based on Java Predicates* ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy, July 2002. See: mulsaw.lcs.mit.edu/

AutoTest

Several articles and online descriptions available from se.ethz.ch/research/tests.html