# Software Verification

Bertrand Meyer

## Lecture 14: Testing

*After material by Ilinca Ciupa*

# Testing basics

# Definition: testing

To test a software system is to try to make it fail

Testing is none of:

- Ensuring software quality
- Assessing software quality
- Debugging

# Exercise*

**Scenario:**

- A program reads three integers representing the lengths of a triangle's sides, and prints a message stating whether the triangle is scalene, isosceles or equilateral.

**Task:**

- Devise inputs to test the program as thoroughly as possible

*After Yuri Gurevich, LASER summer school 2009. Exercise originally from Glen Myers, "*The Art of Software Testing*", Wiley, 1979

# Myers: do you have these?

1. A scalene triangle
2. An isosceles triangle
3. An equilateral triangle
4. 3 permutations of 2
5. A zero-length side
6. A negative-length side
7. Three positive sides, sum of two = third
8. Three permutations of 7
9. Three positive sides, sum of two < third
10. Three permutations of 9
11. (0,0,0)
12. Noninteger values
13. Wrong number of initial values
14. The expected output in each case

# Seven principles of software testing

Bertrand Meyer, *Seven Principles of Software Testing*, IEEE *Computer*, August 2008

1. To test a program is to try to make it fail
2. Tests are no substitute for specifications
3. Any failed execution must yield a test case, to remain forever part of the regression test base
4. Determining success or failure (oracles) must be automatic

   4': Oracles should be part of the program, as contracts

5. A test suite must include both manual and automated cases
6. Don't believe your testing insights: evaluate any testing strategy through objective criteria
7. The most important criterion is number of faults found against time: fc (t)

# - *Intermezzo* -

# Test-Driven Development

# "The agile manifesto"

**agilemanifesto.org**

*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

*That is, while there is value in the items on the right, we value the items on the left more.*

# Agile methods: basic concepts

Principles:

- Iterative development
- Customer involvement
- Support for change
- Primacy of code
- Self-organizing teams
- Technical excellence
- Search for simplicity

Shunned: “big upfront requirements”; plans; binding documents; diagrams (e.g. UML); non-deliverable products

Practices:

- Evolutionary requirements
- Customer on site
- User stories
- Pair programming
- Design & code standards
- Test-driven development
- Continuous refactoring
- Continuous integration
- Timeboxing
- Risk-driven development
- Daily tracking
- Servant-style manager

# Test-Driven Development

Evolutionary approach to development

Combines

- Test-first development
- Refactoring

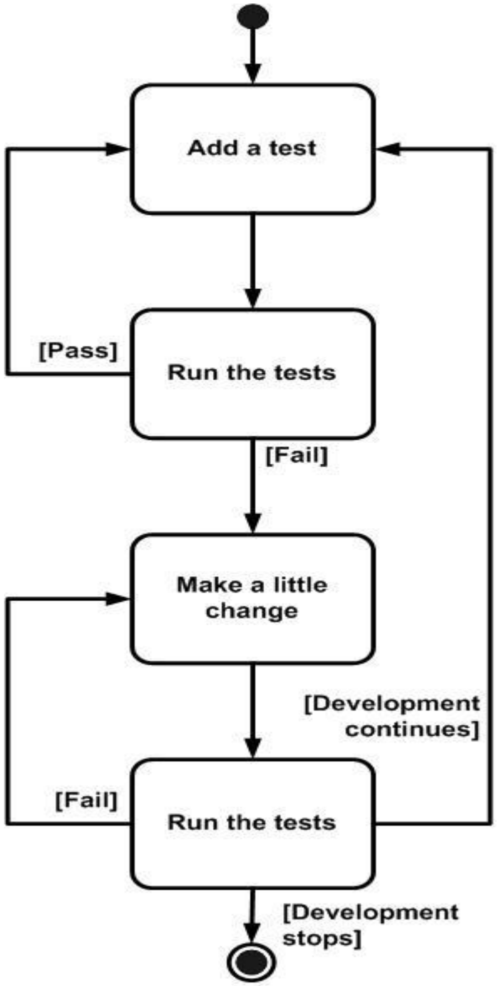
Primarily a method of software design

- Not just a method of testing

# TDD1: Test-First Development

**After Kent Beck***

1. Add a test
2. Run all tests and check the new one fails
3. Implement code to satisfy functionality
4. Check that new test succeeds
5. Run all tests again to avoid regression
6. Refactor code



Copyright 2003 Scott W. Ambler

* *Test Driven Development: By Example*, Addison-Wesley

# TDD 2: Refactoring

A change to the system that leaves its behavior unchanged, but enhances some non-functional quality:

- Simplicity
- Understandability
- Performance

Refactoring does not fix bugs or add new functionality.

# Examples of refactoring

Change the name of a variable, class, ...

Convert local variable to attribute

Generalize type

Introduce argument

Turn a block of code into a routine

Replace a conditional with polymorphism

Break down large routine

# TDD = TFD + Refactoring

Apply test-first development

Refactor whenever you see fit (before next functional modification)

# TDD: consequences on unit tests

Developers must learn to write good unit tests:

- Run fast (short setup, run, and tear-down)
- Run in isolation (reordering is possible)
- Use data that makes test cases easy to read
- Use real data when needed
- Each test case is one step towards overall goal

# TDD assessment

For:

- Reclaims central role of tests
- Continuous execution: reduce gap between decision and feedback
- Encourage developers to write code that is easily tested
- Yields extensive test repository
- Requires that all tests pass

But:

- Tests are not specs
- Some code difficult to test
- Risk that program pass tests and nothing else

*- End of Intermezzo -*

# Test-Driven Development

# What does testing involve?

- Determine system parts & properties to be tested
- Determine appropriate input values
- Determine expected outputs (oracles)
- Run system on selected input values
- Compare results to oracles
- Measure other execution characteristics: time, space…

# Components of a test

A test case specifies:

- The state of the implementation under test (IUT) and its environment before test execution
- The test inputs
- The associated oracle

An oracle defines:

- If possible, pass/no pass evaluation
- Expected returned values
- Expected messages
- Expected exceptions
- Resulting state of IUT and environment

# Test execution

Test suite: collection of test cases

Test driver: class or utility program that applies test cases to an IUT

Stub: partial, temporary implementation of a component

- May serve as a placeholder for an incomplete component or implement testing support code

Test harness : a system of test drivers and other tools to support test execution

# Types of tests: scope

Unit test

- Scope: program module, e.g. routine, class, cluster

Integration test

- Scope: subsystem or entire system, possibly including hardware
- Exercises interfaces between units to demonstrate that they are collectively operable

System test

- Scope: Complete, integrated application
- Focuses on characteristics that are present only at the level of the entire system
- Categories:
  - Functional
  - Performance
  - Stress or load

# Types of tests: intent

Fault-directed testing

- Intent: reveal faults through failures
- Unit and integration testing

Conformance-directed testing

- Intent: demonstrate conformance to required capabilities
- System testing

Acceptance testing

- Intent: enable customer to decide whether to accept software

# Types of tests: intent

Regression testing

- After a change., re-test program to find out if change has not introduced, re-introduced or uncovered faults

Mutation testing (also known as fault *seeding*)

- Test a modified program, with faults introduced
- Why would we do this?

# Black box vs white box testing (1)

| **Black box testing** | **White box testing** |
|---|---|
| Uses no knowledge of the internals of the SUT | Uses knowledge of the internal structure and implementation of the SUT |
| Also known as responsibility-based testing and functional testing | Also known as *implementation-based testing* and *structural testing* |
| Goal: to test how well the SUT conforms to its requirements<br>(Cover all the requirements) | Goal: to test that all paths in the code run correctly<br>(Cover all the code) |

# Black box vs white box testing (2)

| **Black box testing** | **White box testing** |
|---|---|
| Uses no knowledge of the program except its specification | Relies on source code analysis to design test cases |
| Typically used in integration and system testing | Typically used in unit testing |
| Can also be done by user | Typically done by programmer |

# 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 then
        b := b - a
else
        b := 0
end
```

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 for $C_i$

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
- Possibly obtain an 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
- Does not account for multiple executions of loops
- The most commonly used in practice (easy to achieve)

# Predicate coverage

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

Example:

a **or** b **or** (f(x) **and** $x > 0$)

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}

# Condition coverage

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

- Disadvantage: Not all combinations
- Does not imply predicate coverage
- Is not implied by predicate coverage

Example:

**if** a **and** b **then** …

# Clause coverage (CC)

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

Example:

$x > 0$ **or** $y < 0$

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 = -1}
- {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 |

# Multiple-condition coverage

***Source: Steve Cornett***

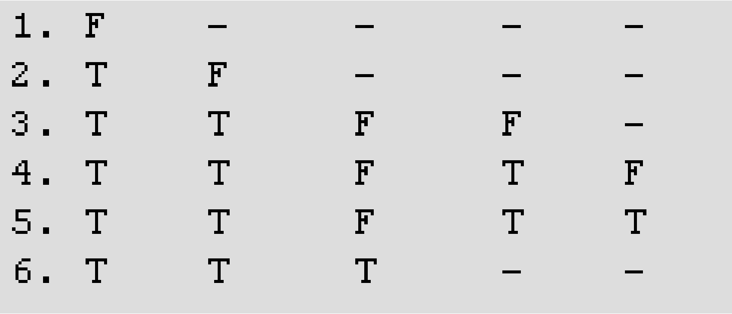
Percentage of combinations of values of elementary boolean conditions affecting the result

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

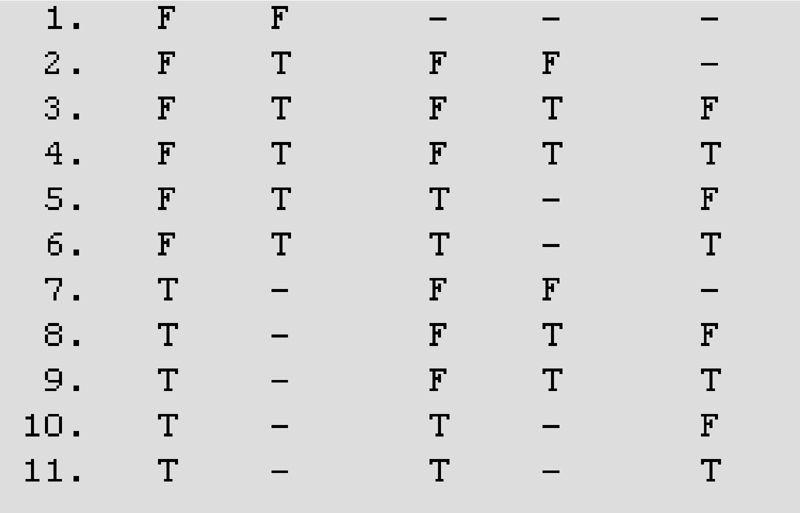
Examples*

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

((a **or** 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 | – | – |



| | | | | | |
|---|---|---|---|---|---|
| 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 |

# Modified Condition/Decision coverage (MC/DC)

Percentage of combinations of elementary conditions that affect the overall condition **independently**

We say that an elementary condition of a predicate "affects the predicate independently" if changing its value, without changing the values of other conditions, changes the value of the predicate

Example:

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

- Advantage: easier to achieve than multiple condition
- Required for safety-critical aviation software (FAA standard RCTA/DO-178B)

# Determination

For a predicate $p$ with:

- A clause $c_M$, the major (or "active" clause)
- The remaining "minor" clauses $c_m \in p$, $m \neq M$

we say that $c_M$ determines $p$ if, for some combination of the values of the minor clauses, changing the value of $c_M$ changes the value of $p$

Example:

$$p = a \vee b$$

|  | a | b |
|---|---|---|
| $c_M = a$ | T<br>F | f<br>f |
| $c_M = b$ | f<br>f | T<br>F |

# Restricted Active Clause Coverage (RACC)*

For each $p \in P$ and each major clause $c_M$, choose minor clauses $c_m$ 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, with the same values for the minor clauses

Example:

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

We satisfy RACC for a if we choose (1,5), or (2,6), or (3,7): three possibilities only

|  | 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 |

Often the interpretation of MC/DC in practice

****Variant of MCDC***

# Correlated Active Clause Coverage (CACC)*

For each $p \in P$ and each major clause $c_M$, choose minor clauses $c_m$ 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

Loosening of RACC: the values 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 and 3, and one out of 5, 6 or 7 (9 possibilities)

| | 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**

# 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 (numerous variants exist that unfold loops up to a maximum bound)

Can be impossible to achieve 100%

```
if c then a1 else a2 end
possible_other_instructions            -- Not affecting c
if c then b1 else b2 end
```

# Limits of coverage measures

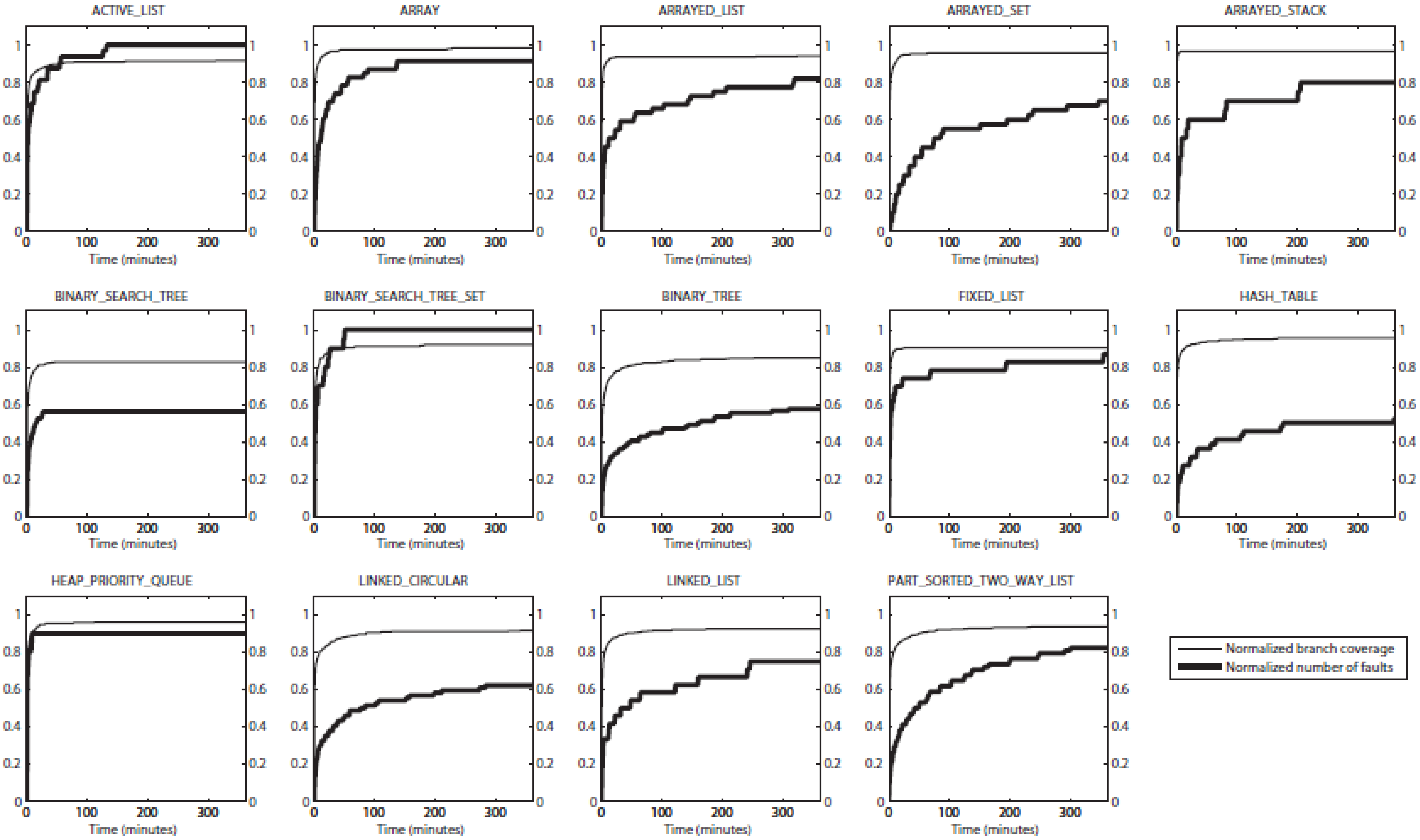


Figure 8: Median of the branch coverage level and median of the normalized number of faults for each class over time

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

# Complementing coverage

Keeping track of faults found in testing campaigns

Comparing the results to:

- Previous phases of the project
- Other projects in the company

# Are we shipping yet?

Possible criteria:

- Coverage (typical: 80% path coverage)
- No blocking faults
- Evolution of faults uncovered

# 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

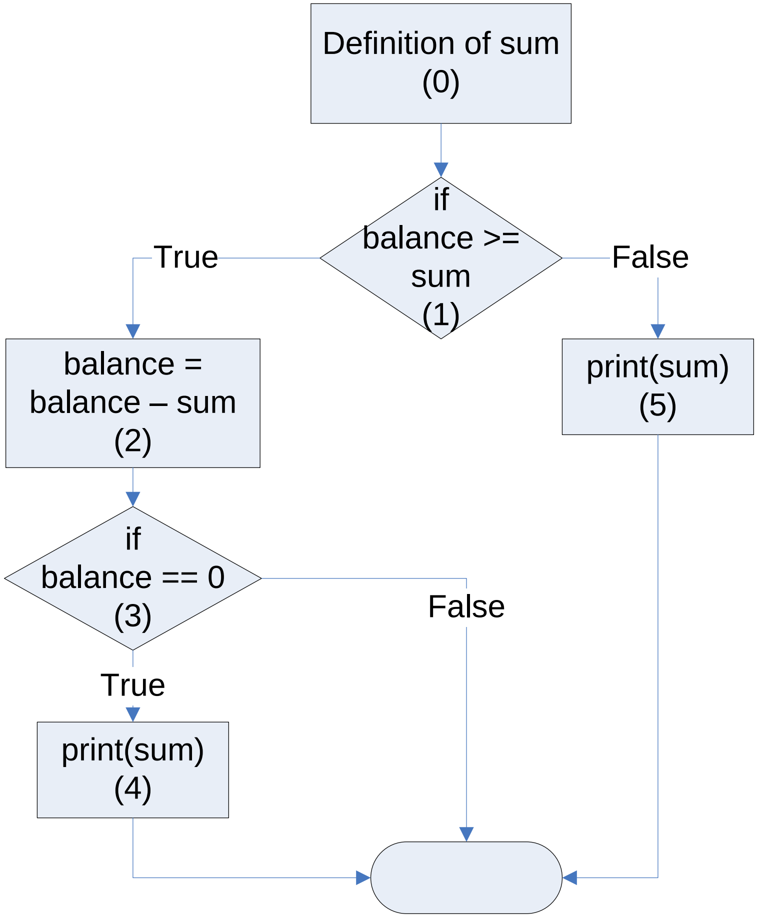
Path properties:

- 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*

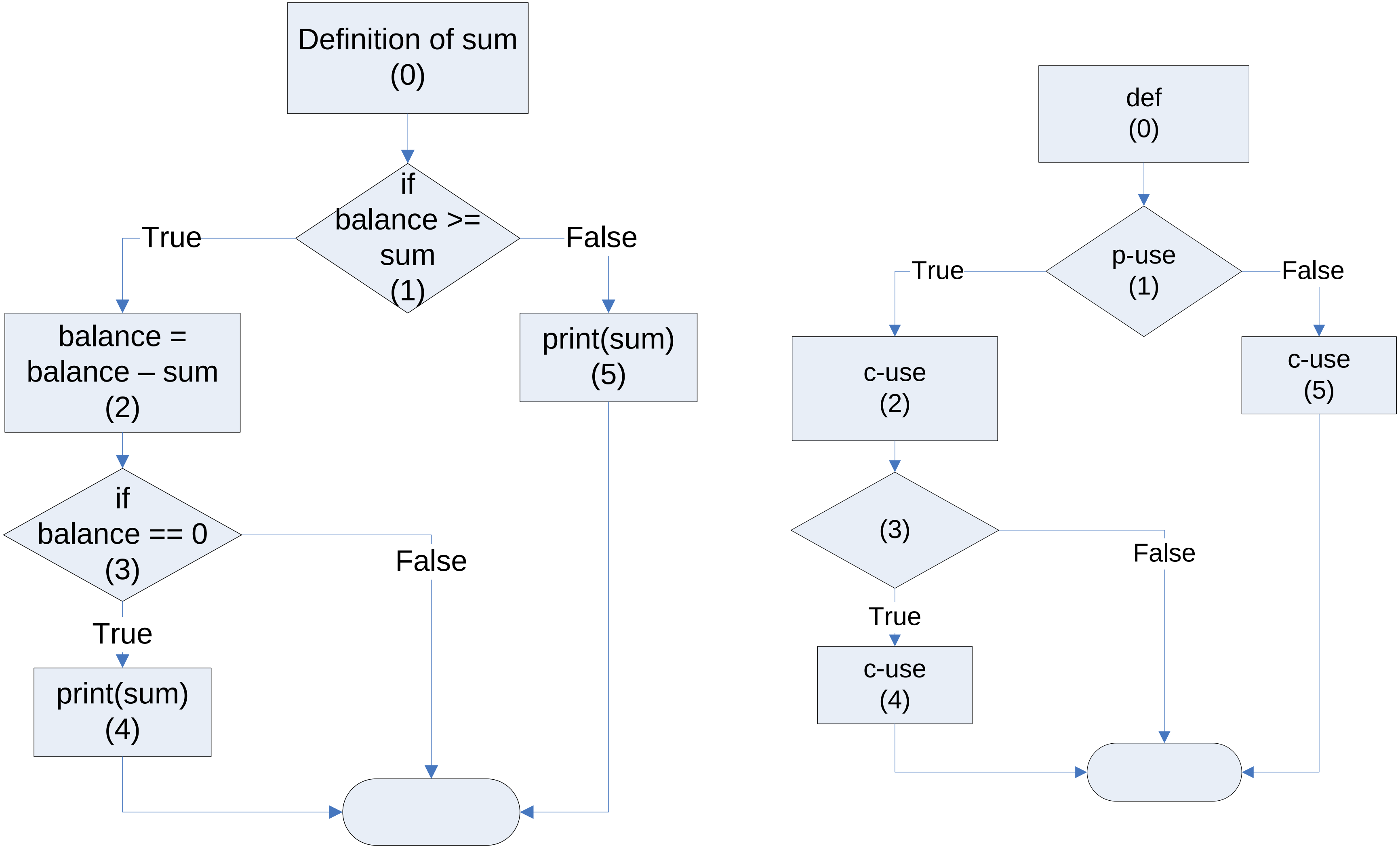
# Example: source code

```
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`**



Definition of sum
(0)

if
balance >=
sum
(1)

True

False

balance =
balance – sum
(2)

print(sum)
(5)

if
balance == 0
(3)

True

False

print(sum)
(4)

def
(0)

p-use
(1)

True

False

c-use
(2)

c-use
(5)

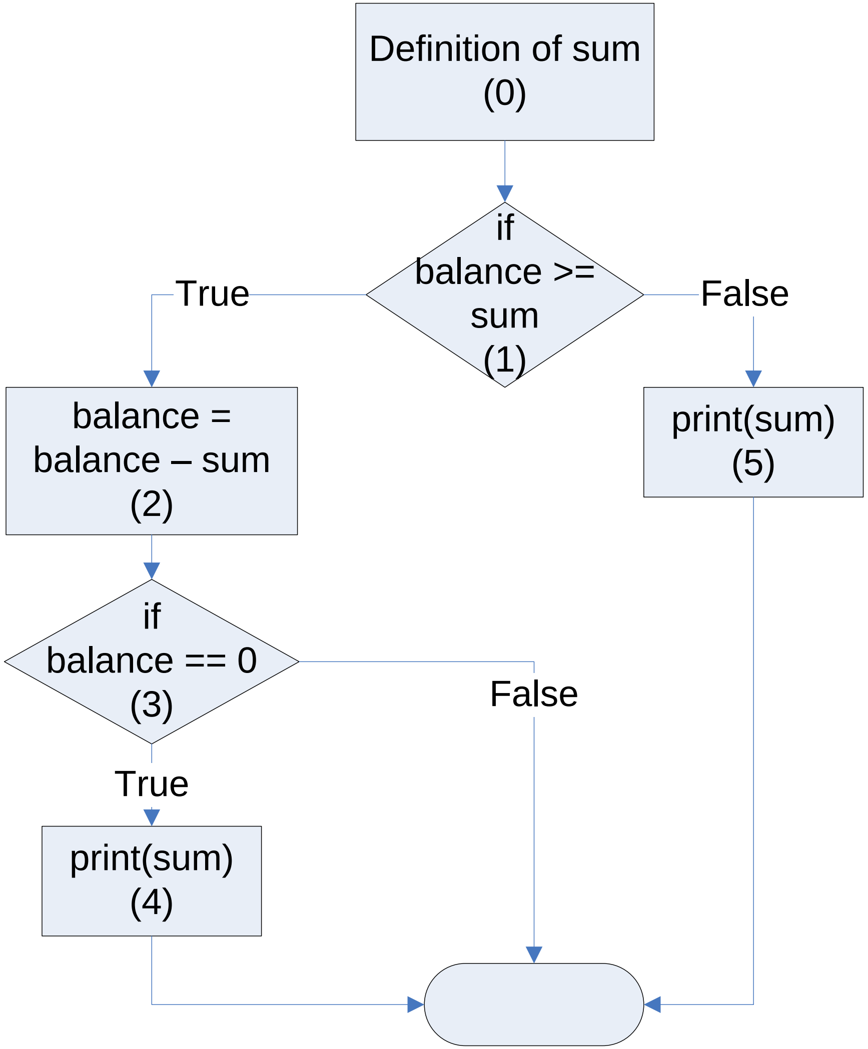
(3)

True

False

c-use
(4)

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



Definition of sum
(0)

if
balance >=
sum
(1)

True

False

balance =
balance – sum
(2)

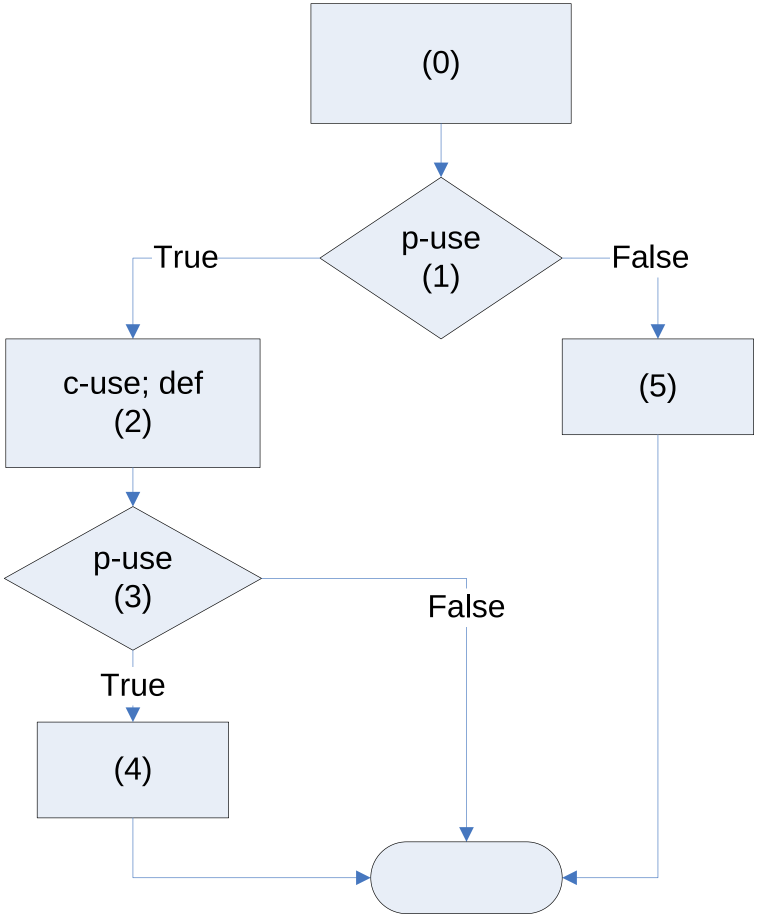
print(sum)
(5)

if
balance == 0
(3)

True

False

print(sum)
(4)



(0)

p-use
(1)

True

False

c-use; def
(2)

(5)

p-use
(3)

True

False

(4)

# 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, if any, 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.

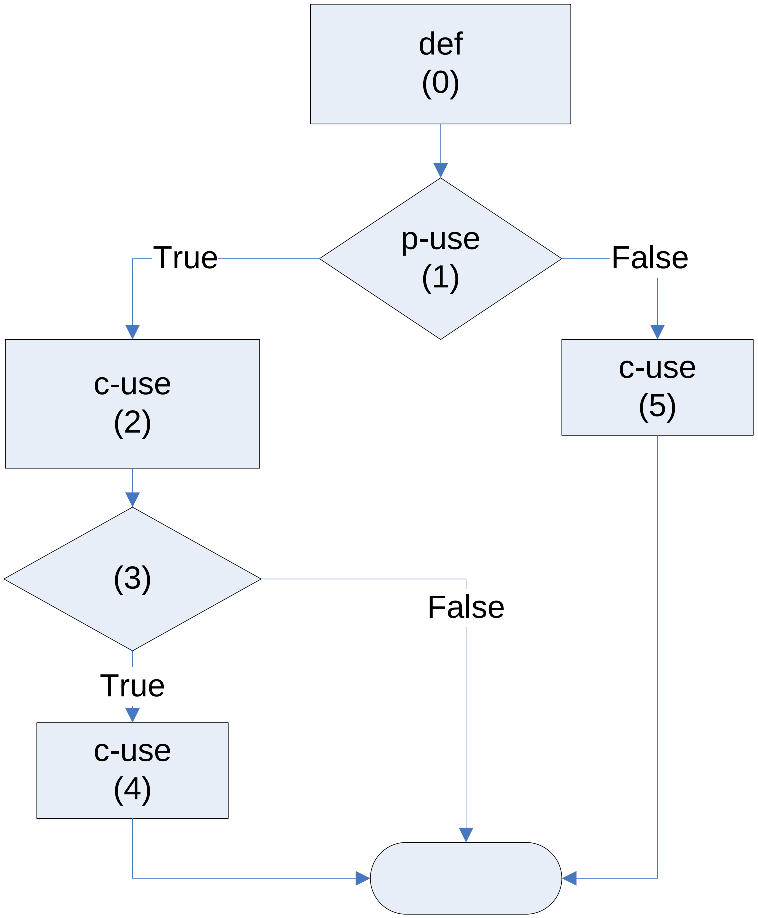
# Dataflow coverage criteria (continued)

**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)

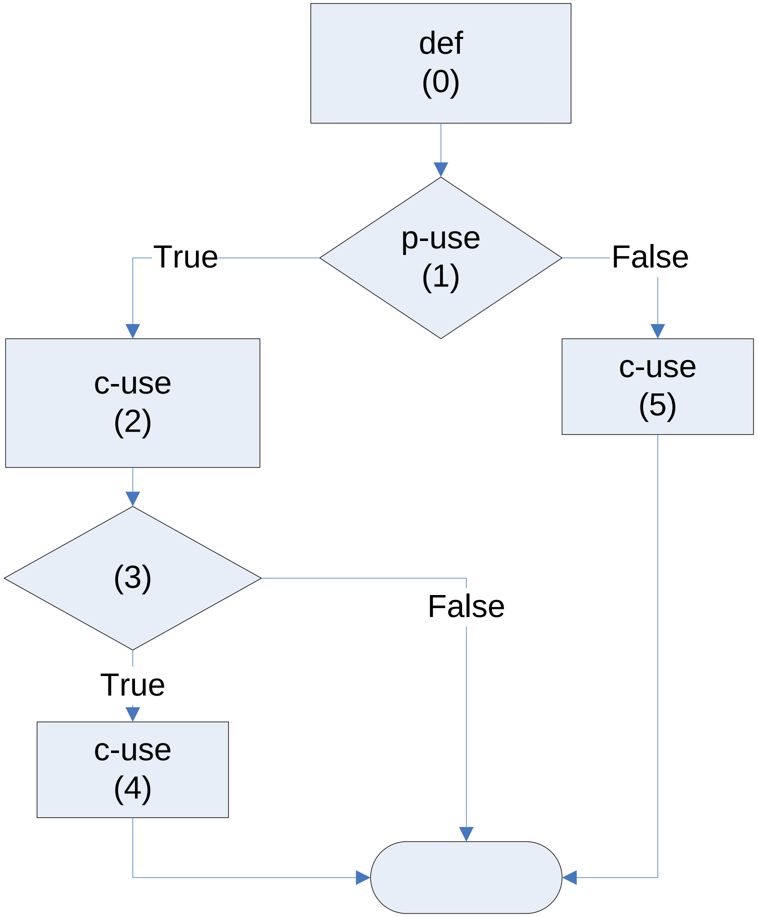
all-p-uses: *at least one* def-clear sub-path from *every* definition to *every* reachable p-use

(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

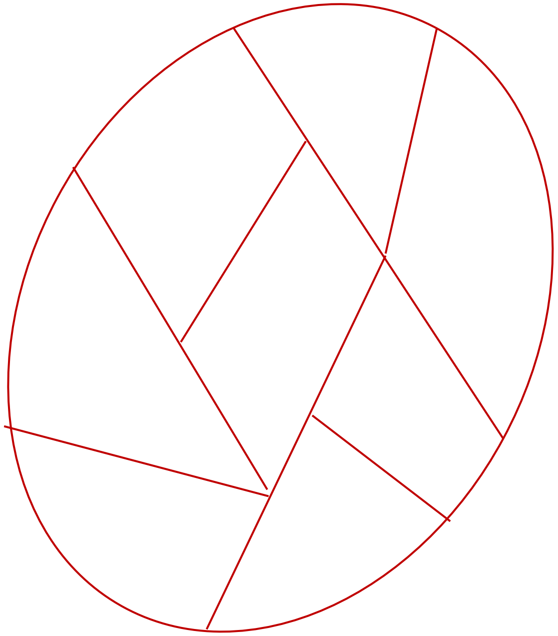
(0,1)

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)

# 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

# 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", 30
                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)

# Degrees 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

- PEX (.NET; Microsoft Research)
- Randoop (C#, Java, C; Mike Ernst)
- Yeti (Java, C#; Manuel Oriol)
- JTest (Java; Parasoft)
- JCrasher (Java; Christoph Csallner)
- SAGE (C, C++; Microsoft Research)
- AutoTest (Eiffel; 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

Precondition satisfaction Strategy

- Keeps track of created objects that satisfy non-trivial preconditions

...

# AutoTest: automatic test framework

**Ilinca Ciupa**
**Andreas Leitner**
**Yi Wei**

- Input: set of classes
- 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

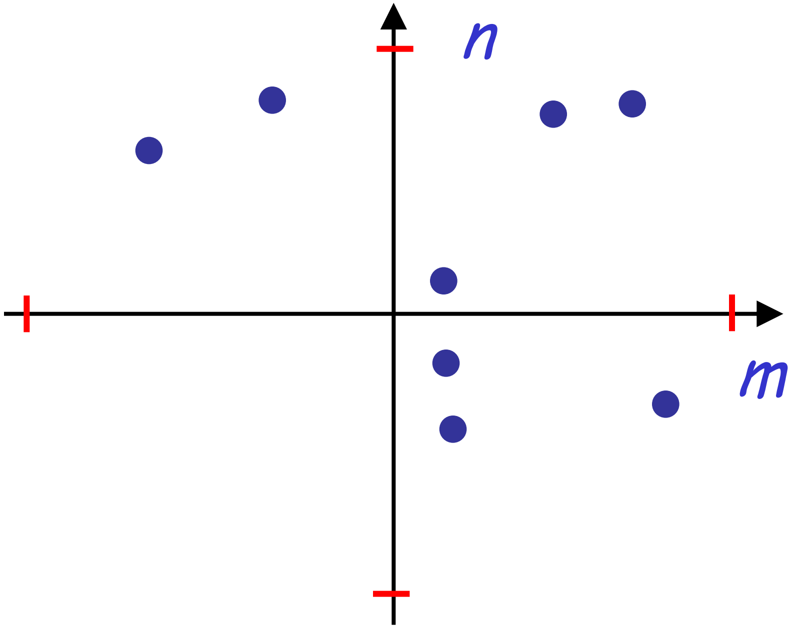
- Object pool
  - Get objects through creation procedures (constructors)
  - Diversify through procedures
- Routine arguments
  - Basic values: heuristics for each type
  - Objects: get from pool
- Test all routines, including inherited ones ("Fragile base class" issue)

# Adaptive Random Testing (Chen et al.)

Conjecture:

Random testing may find faults faster if inputs evenly spread

So far: basic types

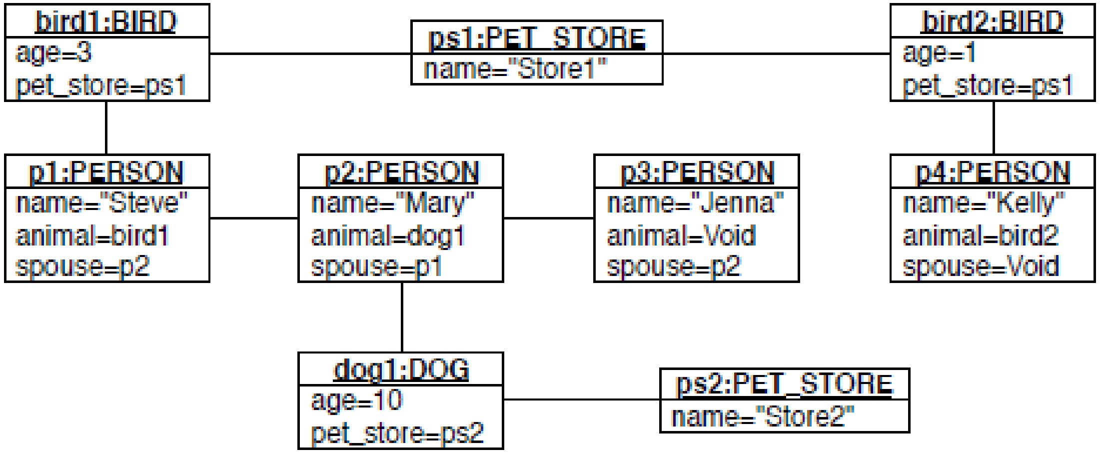


Our contribution: extend this to objects

Need to define notion of distance between objects

# Object distance

**Ilinca Ciupa**
**(ICSE 2008)**



$$p \leftrightarrow q \triangleq \mathit{combination}\,(\mathit{type\_distance}\,(p.\mathit{type}, q.\mathit{type}),\ \mathit{field\_distance}\,(p, q),\ \mathit{recursive\_distance}\,(\{[p.r \leftrightarrow q.r] \mid r \in \mathit{Reference\_attributes}\}\,)$$

# ART vs pure random

Results so far:

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

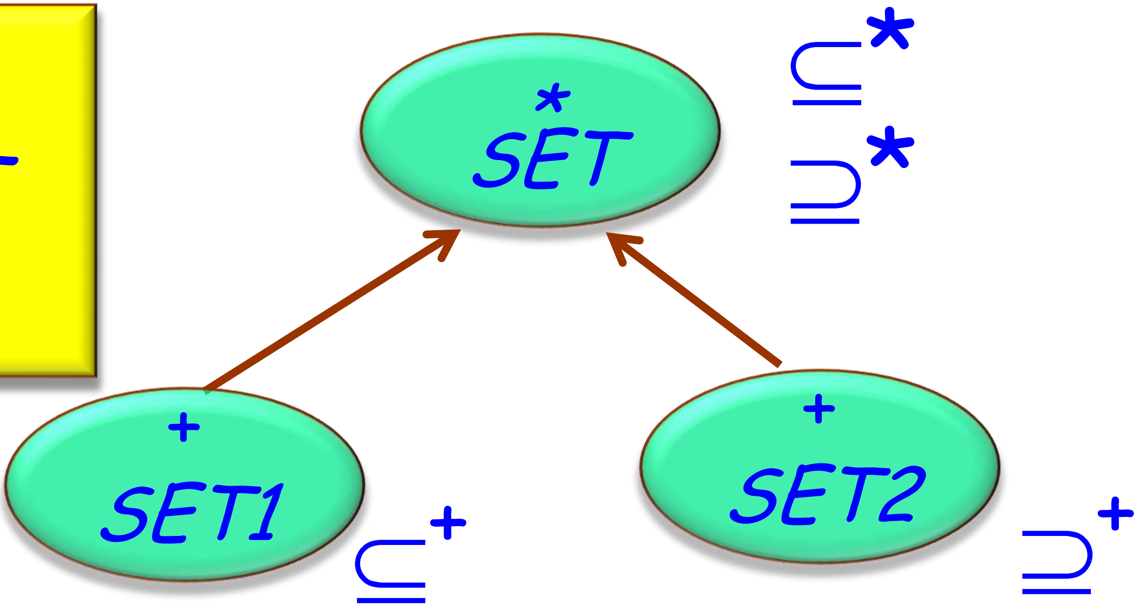
# Random testing: example bug found

**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<br>Total | <br>Failed | ROUTINES<br>Total | <br>Failed |
|---|---|---|---|---|
| EiffelBase (Sep 2005) | 40,000 | 3% | 2,000 | 6% |
| Gobo Math | 1,500 | 1% | 140 | 6% |

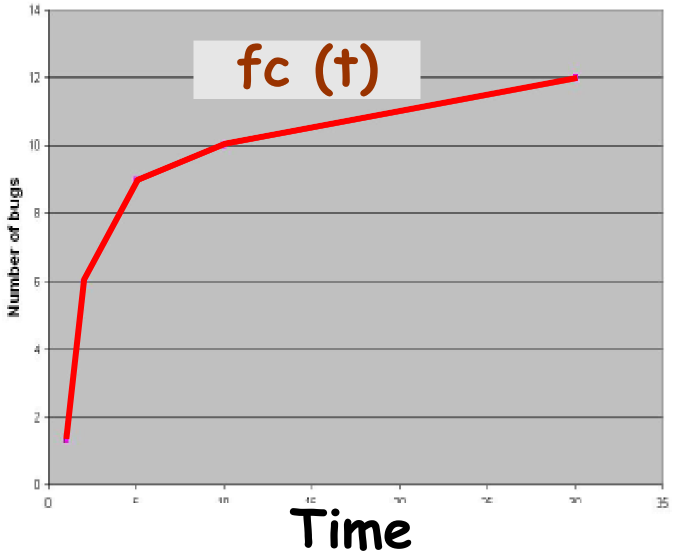
# Testing results and strategy

“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

# 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**

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)

# Test Extraction

**Andreas Leitner, Arno Fiva**

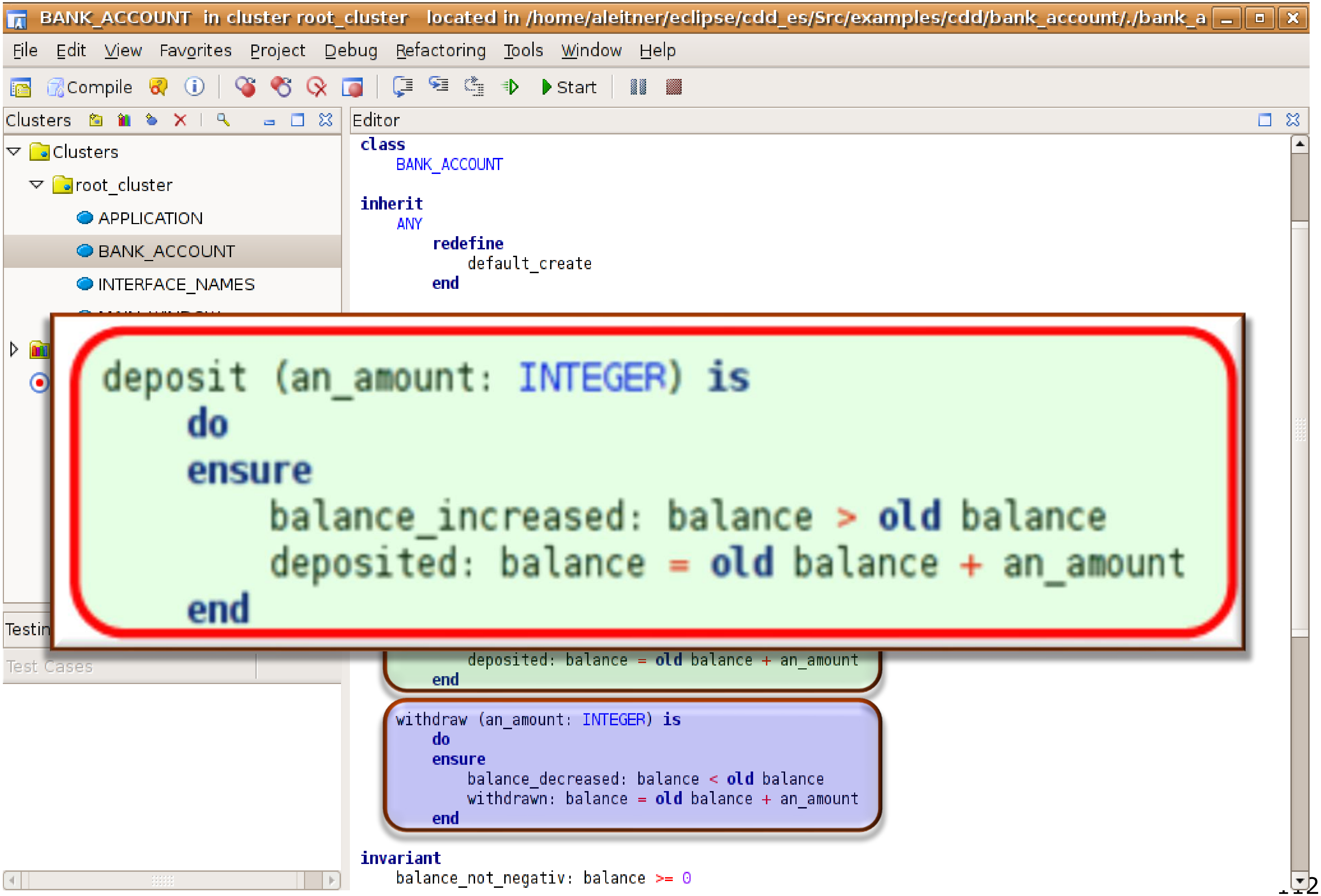
Like Test-Driven Development, but

- 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



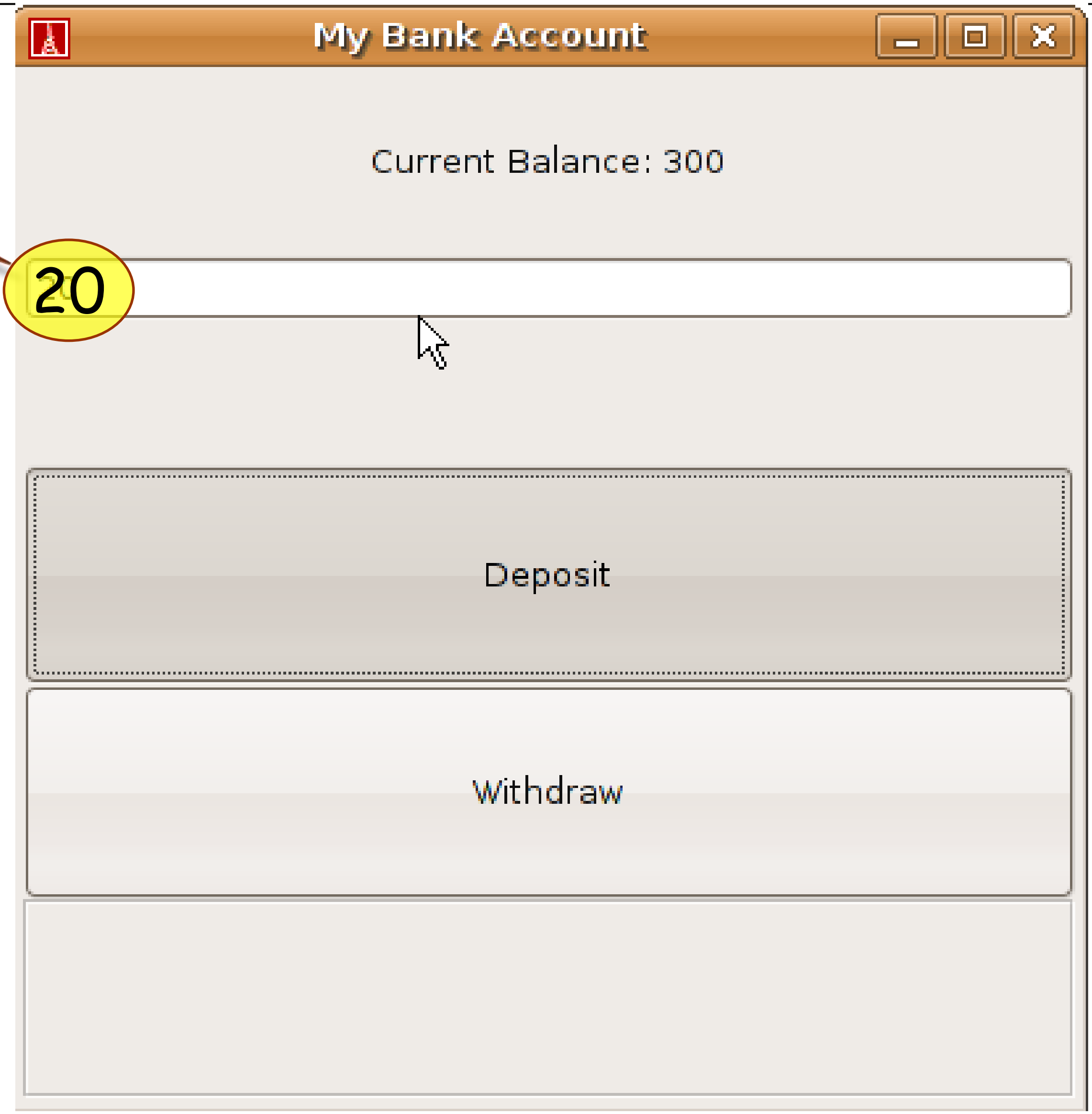
```
class
    BANK_ACCOUNT

inherit
    ANY
        redefine
            default_create
        end
```

```
deposit (an_amount: INTEGER) is
    do
    ensure
        balance_increased: balance > old balance
        deposited: balance = old balance + an_amount
    end
```

```
            deposited: balance = old balance + an_amount
        end
```

```
    withdraw (an_amount: INTEGER) is
        do
        ensure
            balance_decreased: balance < old balance
            withdrawn: balance = old balance + an_amount
        end

invariant
    balance_not_negativ: balance >= 0
```

# Running the system and entering input

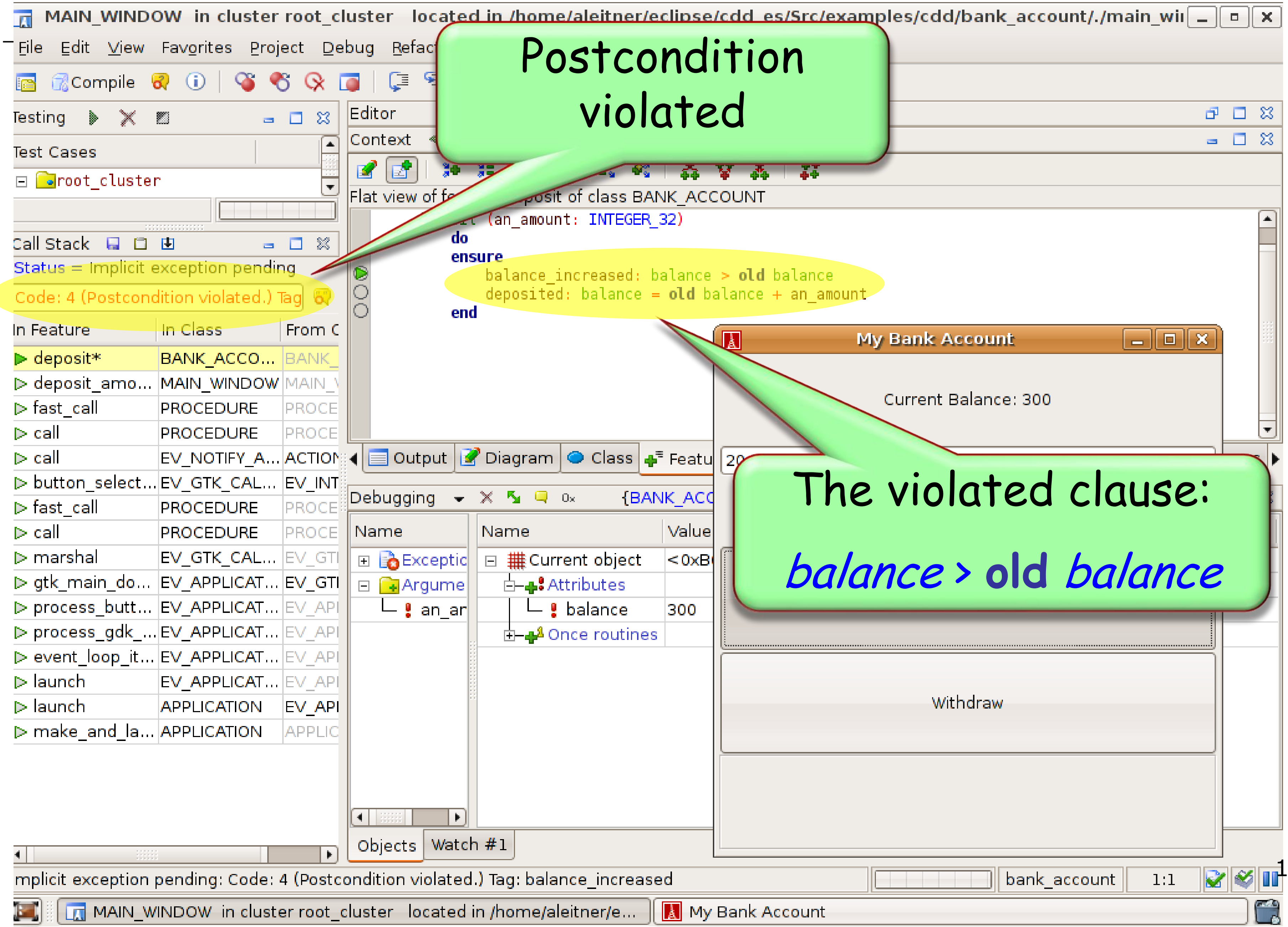


Current Balance: 300

20

(erroneous)

# Error caught at run time as contract violation



Postcondition violated

The violated clause:

*balance* > **old** *balance*

Status = Implicit exception pending

Code: 4 (Postcondition violated.) Tag

| In Feature | In Class | From C |
|---|---|---|
| deposit* | BANK_ACCO... | BANK_ |
| deposit_amo... | MAIN_WINDOW | MAIN_\ |
| fast_call | PROCEDURE | PROCE |
| call | PROCEDURE | PROCE |
| call | EV_NOTIFY_A... | ACTION |
| button_select... | EV_GTK_CAL... | EV_INT |
| fast_call | PROCEDURE | PROCE |
| call | PROCEDURE | PROCE |
| marshal | EV_GTK_CAL... | EV_GT |
| gtk_main_do... | EV_APPLICAT... | EV_GT |
| process_butt... | EV_APPLICAT... | EV_API |
| process_gdk_... | EV_APPLICAT... | EV_API |
| event_loop_it... | EV_APPLICAT... | EV_API |
| launch | EV_APPLICAT... | EV_API |
| launch | APPLICATION | EV_API |
| make_and_la... | APPLICATION | APPLIC |

Flat view of f… posit of class BANK_ACCOUNT

```
it (an_amount: INTEGER_32)
do
ensure
    balance_increased: balance > old balance
    deposited: balance = old balance + an_amount
end
```

Current Balance: 300

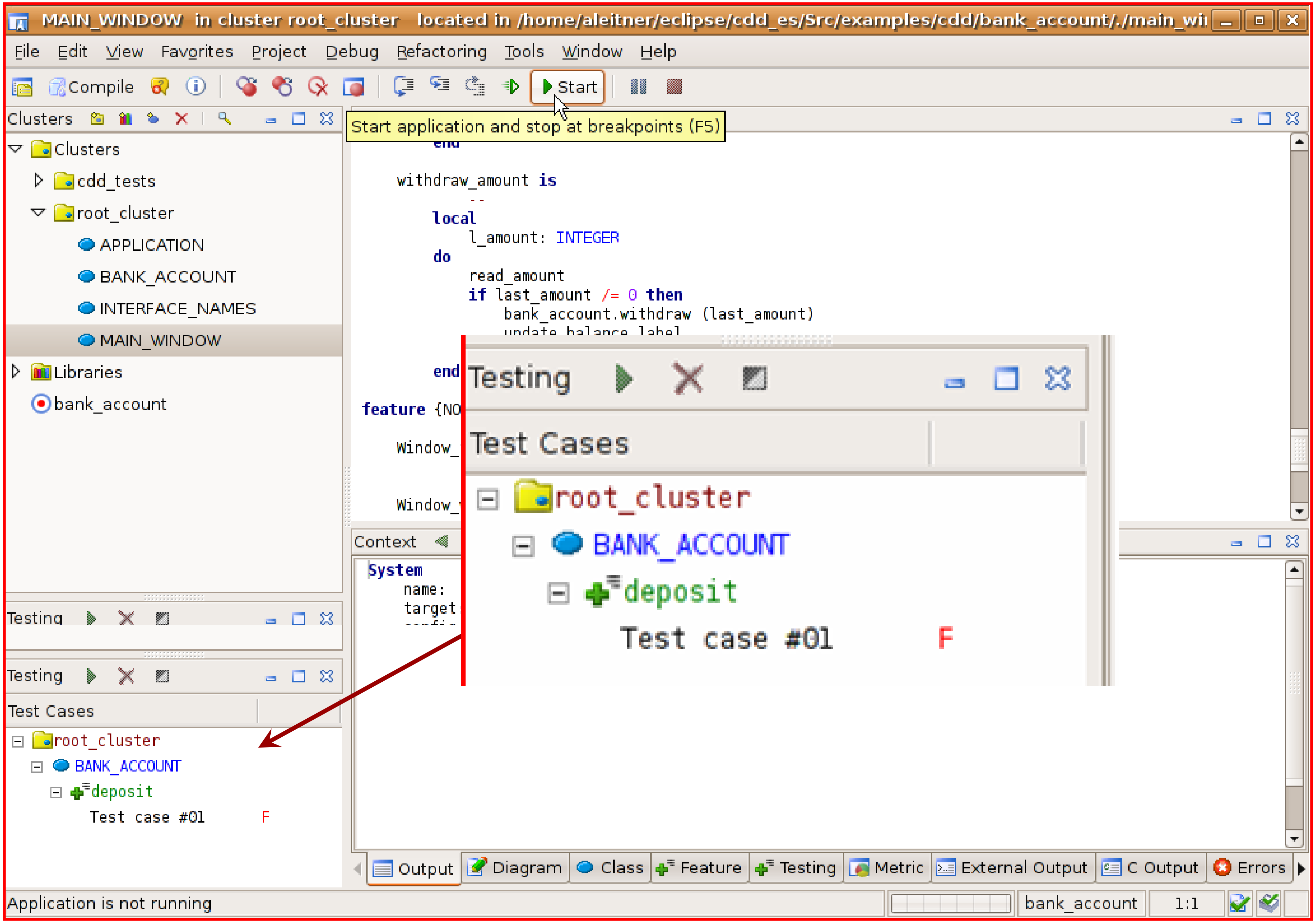
20

| Name |
|---|
| Exceptic |
| Argume |
| an_ar |

| Name | Value |
|---|---|
| Current object | <0xB |
| Attributes | |
| balance | 300 |
| Once routines | |

mplicit exception pending: Code: 4 (Postcondition violated.) Tag: balance_increased

# This has become a test case



MAIN_WINDOW in cluster root_cluster located in /home/aleitner/eclipse/cdd_es/Src/examples/cdd/bank_account/./main_wi

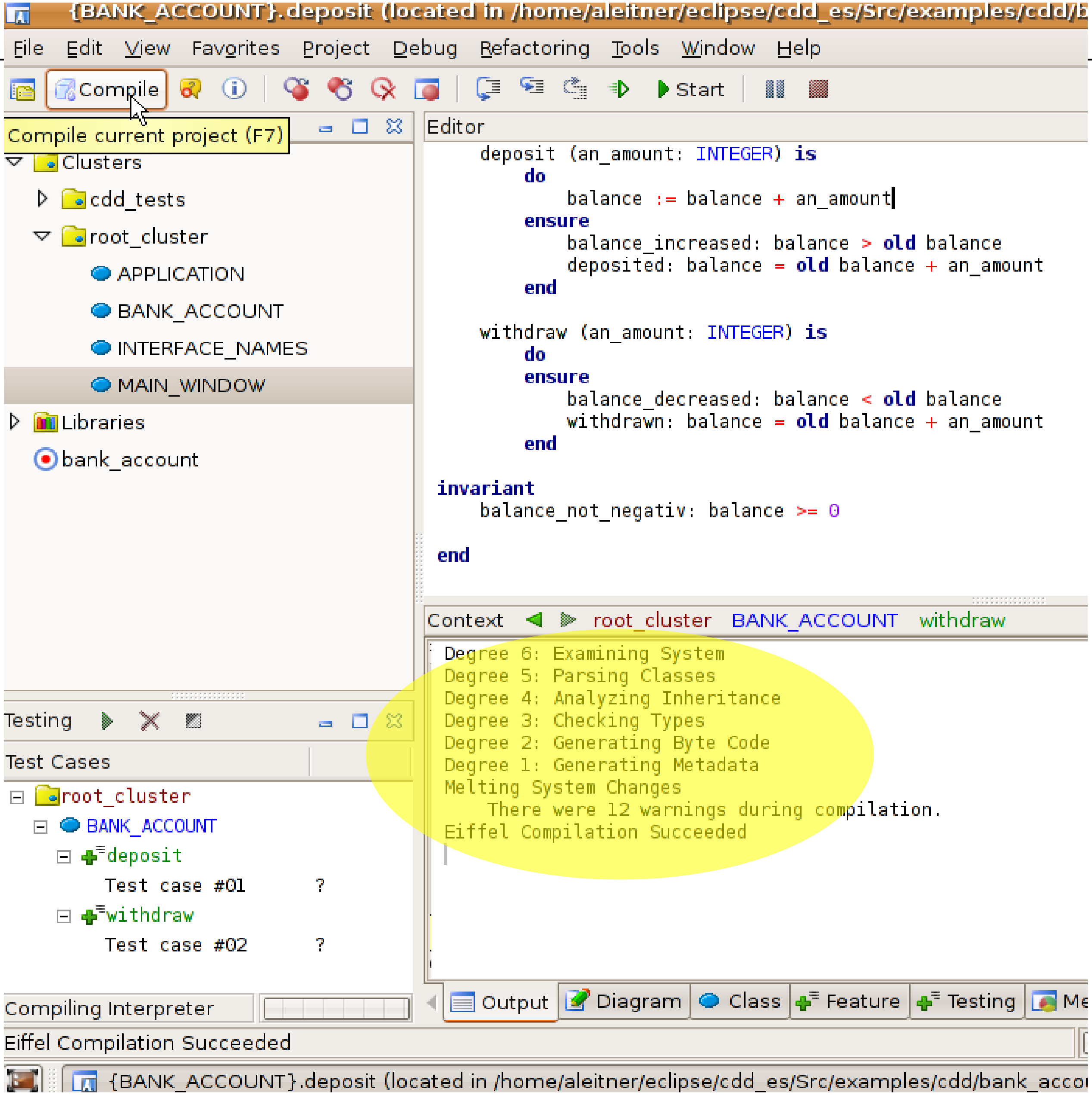
Start application and stop at breakpoints (F5)

```
    end

    withdraw_amount is
            --
        local
            l_amount: INTEGER
        do
            read_amount
            if last_amount /= 0 then
                bank_account.withdraw (last_amount)
                update_balance_label


        end

feature {NO

    Window_


    Window_
```

```
System
    name:
    target:
```

Application is not running

# Correcting and recompiling



```
    deposit (an_amount: INTEGER) is
        do
            balance := balance + an_amount
        ensure
            balance_increased: balance > old balance
            deposited: balance = old balance + an_amount
        end

    withdraw (an_amount: INTEGER) is
        do
        ensure
            balance_decreased: balance < old balance
            withdrawn: balance = old balance + an_amount
        end

invariant
    balance_not_negativ: balance >= 0

end
```

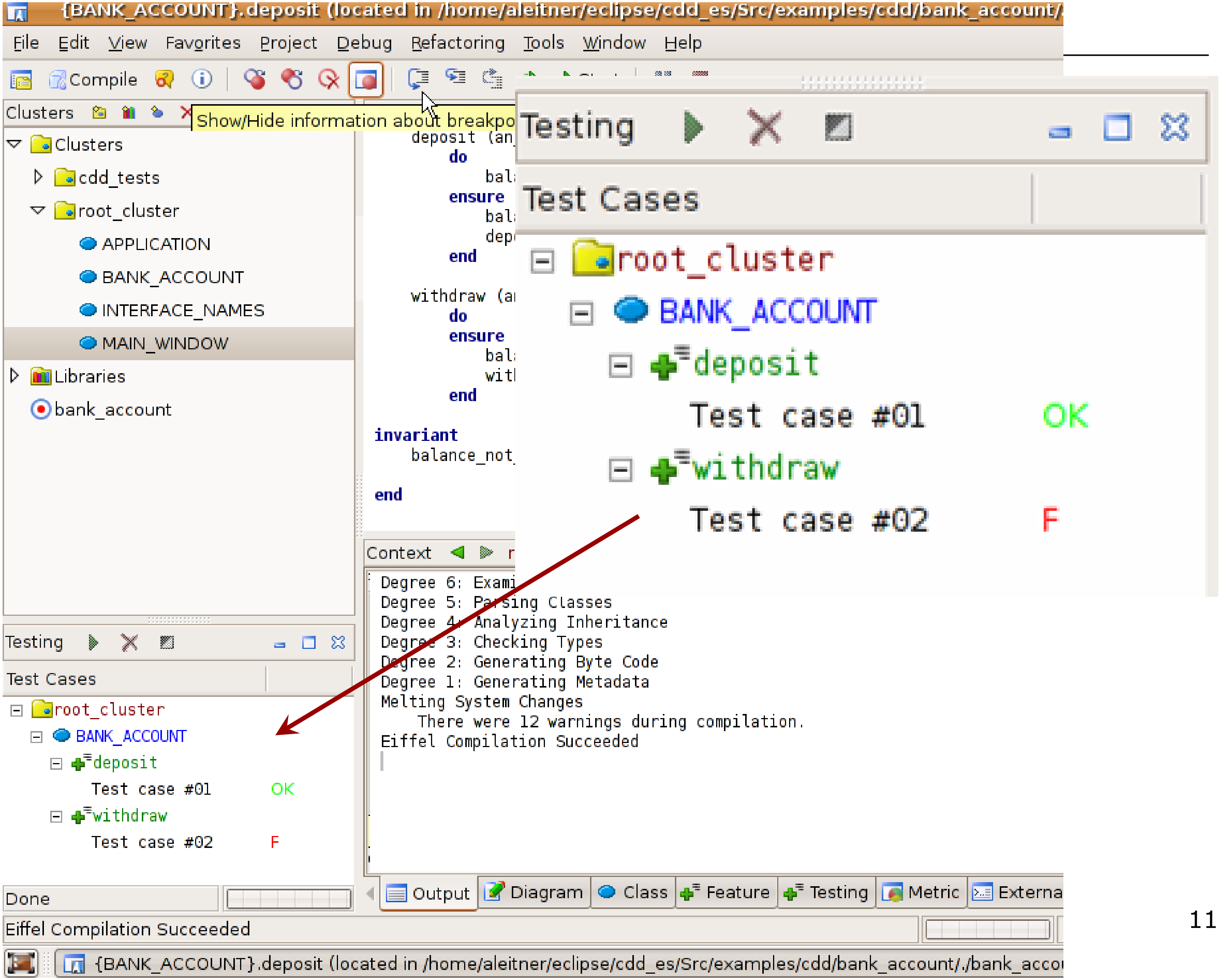
```
Degree 6: Examining System
Degree 5: Parsing Classes
Degree 4: Analyzing Inheritance
Degree 3: Checking Types
Degree 2: Generating Byte Code
Degree 1: Generating Metadata
Melting System Changes
    There were 12 warnings during compilation.
Eiffel Compilation Succeeded
```

Compiling Interpreter

Eiffel Compilation Succeeded

{BANK_ACCOUNT}.deposit (located in /home/aleitner/eclipse/cdd_es/Src/examples/cdd/bank_acco

# One fault corrected, the other not



```
deposit (an
    do
        bal
    ensure
        bal
        dep
    end

withdraw (a
    do
    ensure
        bal
        wit
    end

invariant
    balance_not_

end
```

```
Degree 6: Exami
Degree 5: Parsing Classes
Degree 4: Analyzing Inheritance
Degree 3: Checking Types
Degree 2: Generating Byte Code
Degree 1: Generating Metadata
Melting System Changes
    There were 12 warnings during compilation.
Eiffel Compilation Succeeded
```

- root_cluster
  - BANK_ACCOUNT
    - deposit
      - Test case #01 OK
    - withdraw
      - Test case #02 F

Eiffel Compilation Succeeded

{BANK_ACCOUNT}.deposit (located in /home/aleitner/eclipse/cdd_es/Src/examples/cdd/bank_account/./bank_acco

# Automatic test case generation: **assessment**

Testing is tedious

Automation can help

Challenges involved

Tools are getting there!