

# Programs that test themselves

Bertrand Meyer, *ETH Zurich and Eiffel Software*

Arno Fiva, Ilinca Ciupa, Andreas Leitner, and Yi Wei, *ETH Zurich*

Emmanuel Stapf, *Eiffel Software*

Presenter: Papastergiou Christos

# Introduction

- Modern engineering products continuously test themselves
- They are designed for testability
- Software design pays little attention to testing needs

**Idea:** Design software for testability

# Autotest

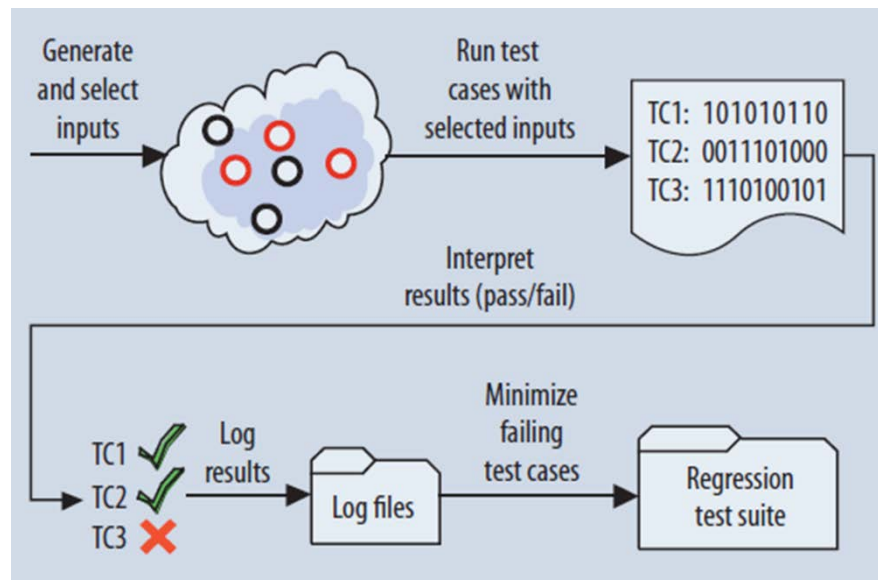
- Autotest is a set of components that
  - automates testing process
  - relies on programs with contracts
  - is integrated into the EiffelStudio
- Components
  - test generation
  - test extraction
  - integration of manual tests

# Automated testing

- Levels of automation:
  - test execution (JUnit, PHPUnit ...)
  - regression testing
  - resilience
  - test case generation
  - test oracles
  - minimization
- Most frameworks support only the first three
- Autotest innovates also on the last three

# Test generation

- The unit of a generated test is a failed routine call
- Each routine is exercised with different targets and arguments
- Use contracts as oracles
- Log results
- Create minimized tests for the failed routines



# Exercising a routine (1)

- Objects are needed for target and possibly for arguments
- When an object T is needed, Autotest decides:
  - to create a new one
  - to use an existing one
- To create a new object Autotest
  - selects a constructor
  - makes sure invariant holds

# Exercising a routine(2)

- The arguments of a routine might be of primitive types.  
Autotest decides:
  - random selection from the domain
  - selection from preset values for each type
- Random but still powerful

# Contracts as oracles

- Contracts in the code serve as oracles
- A contract violation signals a flaw either in:
  - the caller of a routine or
  - in the routine itself
- Benefits
  - software is tested as it is
  - no further programming skills needed

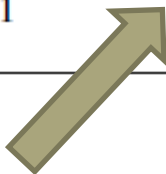


# Optimizations

- Adaptive random testing
  - use values equally spaced out across a domain
  - introduction of a distance metric for objects
  - complements rather than replaces the random algorithm

```
ba1: BANK_ACCOUNT, ba1.owner="A", ba1.balance=675234  
ba2: BANK_ACCOUNT, ba2.owner="B", ba2.balance=10  
ba3: BANK_ACCOUNT, ba3.owner="O", ba3.balance=99  
ba4 = Void  
i1: INTEGER, i1 = 100  
i2: INTEGER, i2 = 284749  
i3: INTEGER, i3 = 0  
i4: INTEGER, i4 = -36452  
i5: INTEGER, i5 = 1
```

Objects pool



Routine exercising  
using ART

```
ba3.transfer(ba1, i5)  
ba1.transfer(ba4, i2)  
ba2.transfer(ba2, i4)  
...
```

# Minimization

- Keeping the whole failed test is impractical
- Keep only the instructions that involve the target and the arguments of the failing routine
  - statically analyze the failed test
  - calculate backward slice
  - use the slice as the failed test

```
...
67 v_61.forget_right
68 create {PRIMES} v_62
69 v_63 := v_62.lower_prime ({INTEGER_32} 2)
70 create {STRING_8} v_64.make_from_c (itp_default_pointer
   )
...
146 create {ARRAY2 [ANY]} v_134.make ({INTEGER_32} 7, {
   INTEGER_32} 6)
147 v_134.enter (v_45, v_131)
148 create {RANDOM} v_135.set_seed (v_63)
149 v_136 := v_135.real_item
```

Initial test

```
68 create {PRIMES} v_62
69 v_63 := v_62.lower_prime ({INTEGER_32} 2)
148 create {RANDOM} v_135.set_seed (v_63)
149 v_136 := v_135.real_item
```

Minimized test

# Test generation results

- Autotest was experimented on classes with different semantics and sizes

Tested library	Faults	Percent failing routines	Percent failed tests
EiffelBase	127	6.4 (127/1984)	3.8 (1513/39615)
Gobo libraries	26	4.4 (26/585)	3.7 (2.928/79886)
Specification library	72	14.1 (72/510)	49.6 (12860/25946)

# Test extraction

- Failed runs are candidate test cases
- Autotest can turn a failure into a test by
  1. creating a trace abstraction of the debugger (a called\_by tree with <invocation,context> nodes)
  2. selecting the invocation that received the failure
  3. extracting a snapshot of the state that is required for this invocation

Demo

# Conclusions

- Advantages
  - nice features on automatized testing
  - discovers unfound software failures
  - helps investigate questions
  - does not require extra knowledge
  - all tests are treated the same regardless of their origin
- Disadvantages
  - cannot guarantee absence of faults
  - not suitable for integration testing
  - generated and extracted tests less robust and readable

**Manual tests should still form the majority of your testing suite!**

Questions?

# Demo – Bank Account Class

```
bank_test - [bank_test] [BANK_ACCOUNT] (C:\Users\Christiano\Documents\Eiffel User Files\7.2\projects\bank_test\bank_account.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
Search Compile Run View bank_test
Class BANK_ACCOUNT Feature withdraw View bank_test
BANK_ACCOUNT
    redefine
        default_create
    end
feature
    default_create
    do
        balance := 0
    end

    balance: INTEGER

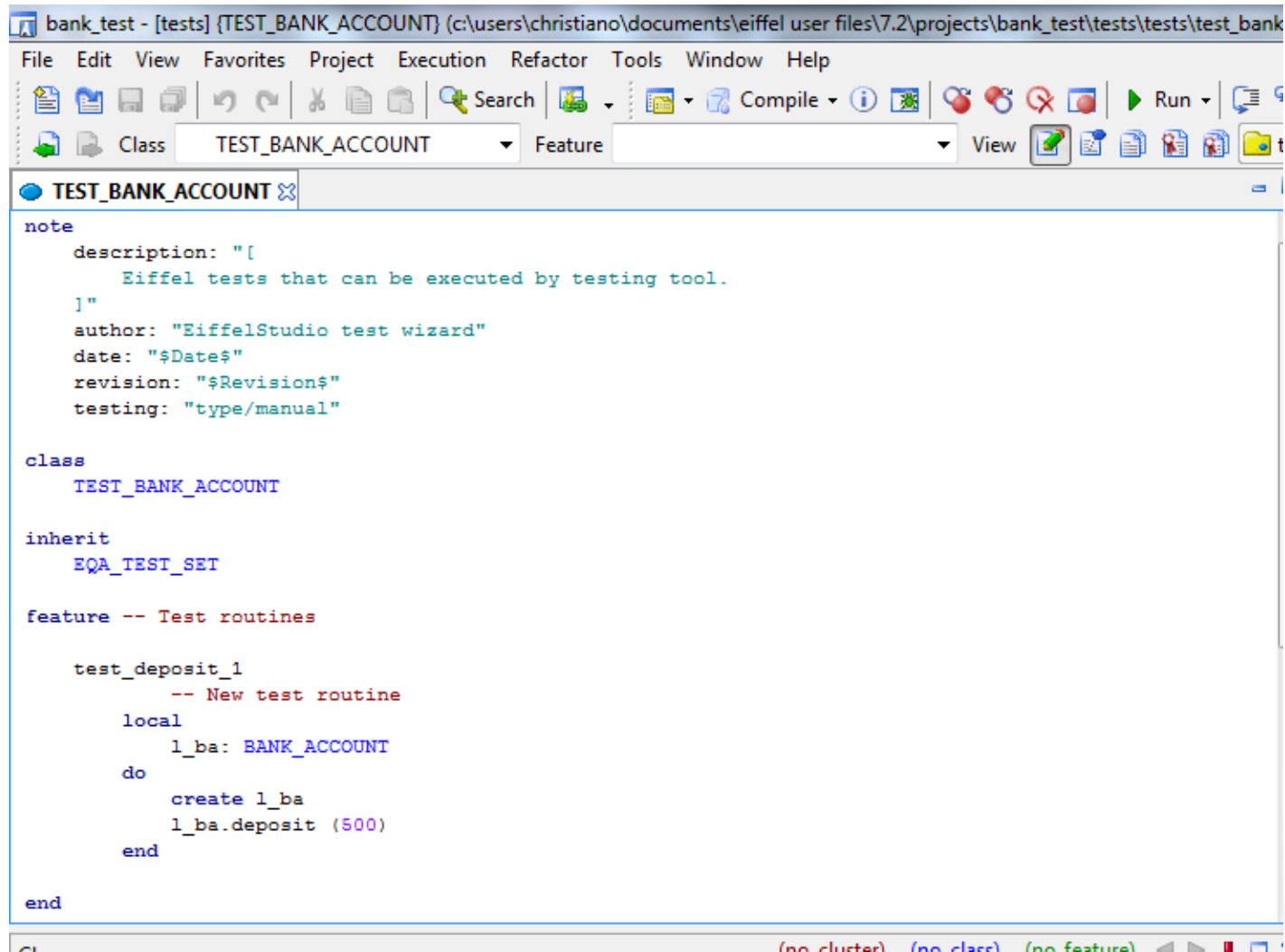
    deposit (an_amount: INTEGER)
        -- Deposit 'an_amount'.
        require
            amount_large_enough: an_amount > 0
        do
            balance := balance + an_amount;
        ensure
            balance_increased: balance > old balance
            deposited: balance = old balance + an_amount
        end

    withdraw (an_amount: INTEGER)
        -- Withdraw 'an_amount'.
        require
            amount_large_enough: an_amount > 0
            amount_valid: balance >= an_amount
        do
            balance := balance - an_amount
        ensure
            balance_decreased: balance < old balance
            withdrawn: balance = old balance - an_amount
        end

invariant
    balance_not_negative: balance >= 0
end
```



# Demo – Manual Test Case



The screenshot shows the EiffelStudio IDE interface. The title bar indicates the file path: `bank_test - [tests] {TEST_BANK_ACCOUNT} (c:\users\christiano\documents\ Eiffel user files\7.2\projects\bank_test\tests\tests\test_bank`. The menu bar includes File, Edit, View, Favorites, Project, Execution, Refactor, Tools, Window, and Help. The toolbar contains icons for file operations, search, compile, and run. The main editor window displays the following code:

```
note
  description: "[
    Eiffel tests that can be executed by testing tool.
  ]"
  author: "EiffelStudio test wizard"
  date: "$Date$"
  revision: "$Revision$"
  testing: "type/manual"

class
  TEST_BANK_ACCOUNT

inherit
  EQA_TEST_SET

feature -- Test routines

  test_deposit_1
    -- New test routine
    local
      l_ba: BANK_ACCOUNT
    do
      create l_ba
      l_ba.deposit (500)
    end

end
```

At the bottom of the editor, there are navigation icons and status text: `(no cluster) (no class) (no feature)`.

# Demo – Test Execution

The screenshot displays the EiffelStudio IDE interface during a test execution. The main editor shows the source code for the `TEST_BANK_ACCOUNT` class, which includes a `test_deposit_1` routine. The `AutoTest` panel on the right indicates that 1 out of 1 tests were run, with 0 unresolved and 1 failure. The `Outputs` panel at the bottom shows the execution details for the failed test, including a postcondition violation in the `BANK_ACCOUNT.deposit` method.

```
note
  description: "[
    Eiffel tests that can be executed by testing tool.
  ]"
  author: "EiffelStudio test wizard"
  date: "$Date$"
  revision: "$Revision$"
  testing: "type/manual"

class
  TEST_BANK_ACCOUNT

inherit
  EQA_TEST_SET

feature -- Test routines

  test_deposit_1
    -- New test routine
    local
      l_ba: BANK_ACCOUNT
    do
      create l_ba
      l_ba.deposit (500)
    end

end
```

**AutoTest**

Filter: ^class

Tests	Status	Last executed
class		
bank_test		
tests		
tests		

Run: 1/1    Unresolved: 0    Fail: 1

test\_deposit\_1 (TEST\_BANK\_ACCOUNT) balance\_increased [0.0140s]

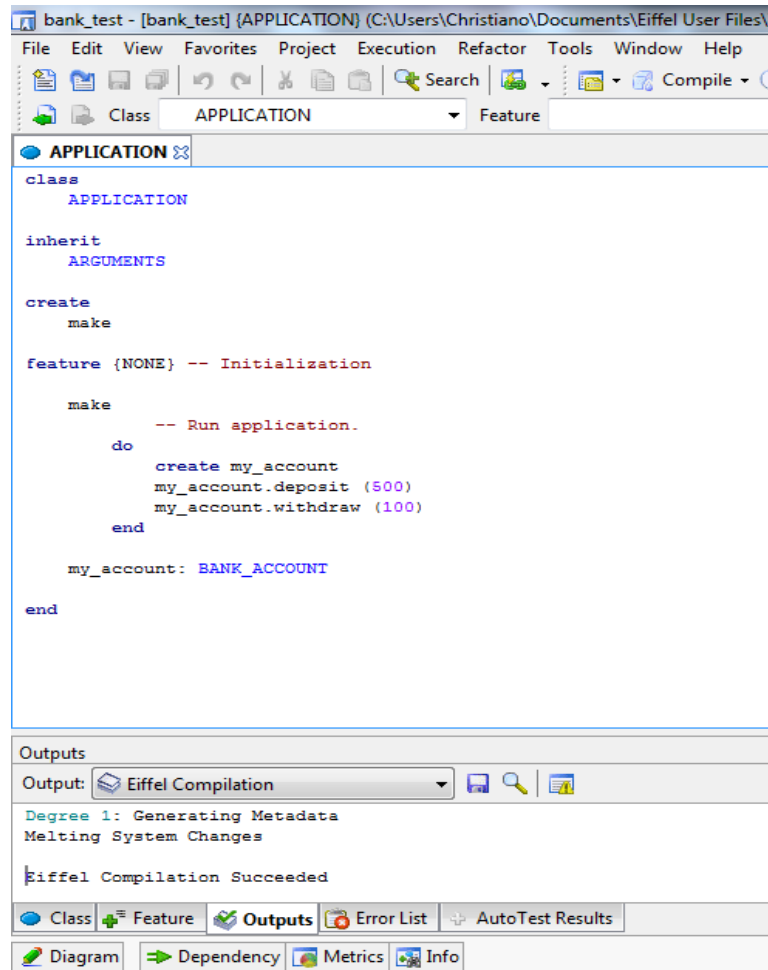
**Outputs**

Output: Testing

```
test routine: exceptional (Postcondition violation in BANK_ACCOUNT.deposit)
on_clean: ok

Execution complete
```

# Demo – Application Class



The screenshot displays the Eiffel IDE interface. The main editor window shows the source code for the `APPLICATION` class. The code defines a class that inherits from `ARGUMENTS` and includes a `make` feature for initialization. The `make` feature contains a `do` block that creates a `my_account` object, deposits 500, and withdraws 100. The `my_account` attribute is of type `BANK_ACCOUNT`.

```
class
  APPLICATION

inherit
  ARGUMENTS

create
  make

feature {NONE} -- Initialization

  make
    -- Run application.
  do
    create my_account
    my_account.deposit (500)
    my_account.withdraw (100)
  end

  my_account: BANK_ACCOUNT

end
```

The bottom panel shows the `Outputs` window, which displays the following text:

```
Output: Eiffel Compilation

Degree 1: Generating Metadata
Melting System Changes

Eiffel Compilation Succeeded
```

The IDE interface includes a menu bar (File, Edit, View, Favorites, Project, Execution, Refactor, Tools, Window, Help), a toolbar with icons for file operations and compilation, and a bottom toolbar with buttons for Class, Feature, Outputs, Error List, AutoTest Results, Diagram, Dependency, Metrics, and Info.

# Demo – Failed Execution

The screenshot shows the EiffelStudio IDE with a debugger window. The 'Outputs' window shows the application launch. The code editor displays the following code:

```
do
  balance := balance + an_amount
ensure
  balance_decreased: balance < old balance
  withdrawn: balance = old balance - an_amount
end
```

An 'EiffelStudio Warning' dialog box is open, displaying the following text:

**EiffelStudio Warning**

Contract violation occurred. Do you want to

- [break] into debugger,
- or [continue] to let the application handle the violation,
- or [ignore] the violation and continue as if it was not violated?

Note: You can always ignore contract violation later using the drop down menu from "Run" button.

Do not show again (always break into debugger on contract violation.)

Buttons: Break, Continue, Ignore

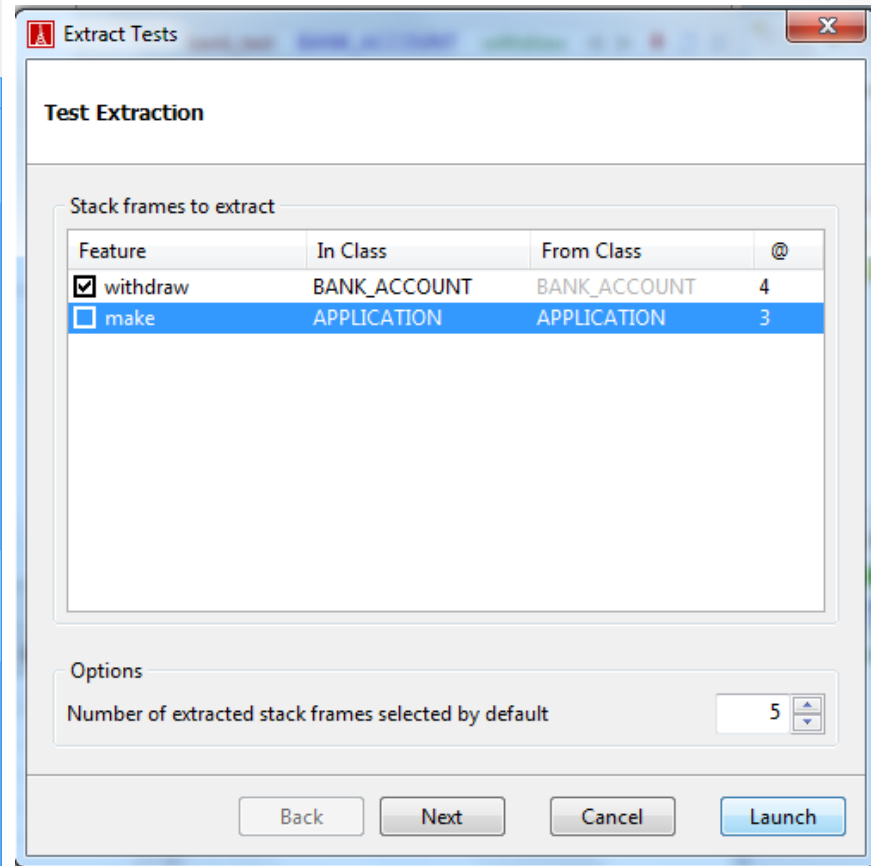
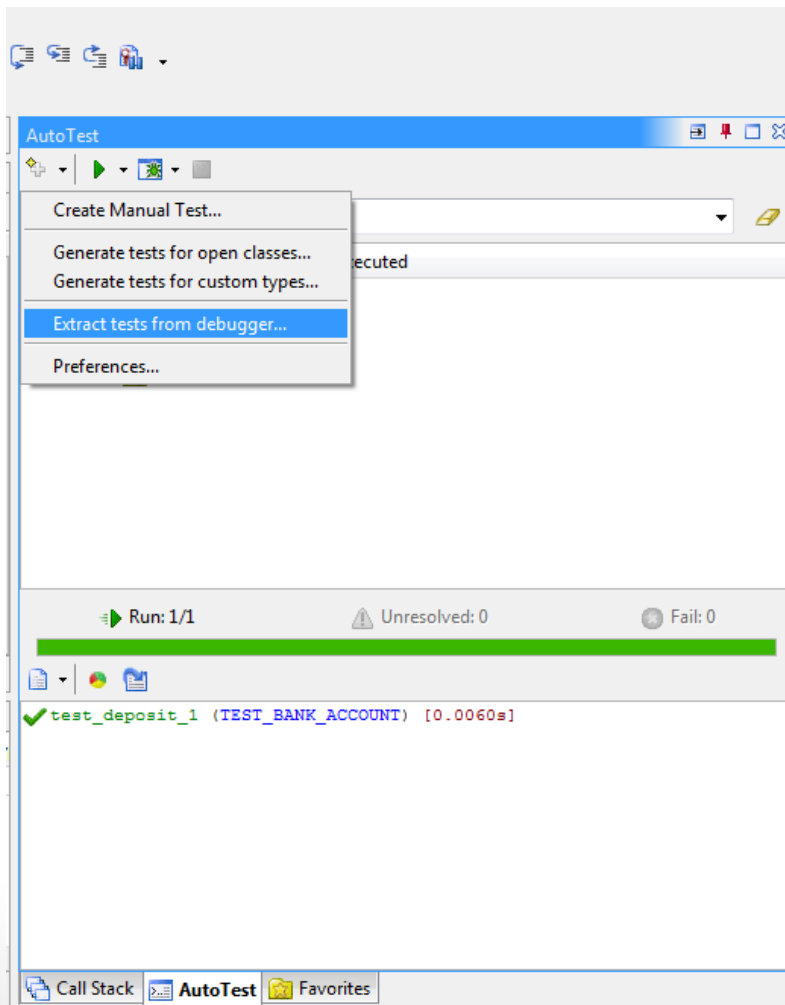
The background interface shows the 'Objects' window with the following data:

Name	Value	Type	Address
Exception raised	balance_decreased: POST...		
Message	balance_decreased		
Code	4		
Type	POSTCONDITION_VIOLAT...		
Exception object	balance_decreased: POST...	Exception data	0x2D94C10
Current object	<0x2D94C08>	BANK_ACCOUNT	0x2D94C08
balance	600	INTEGER_32	

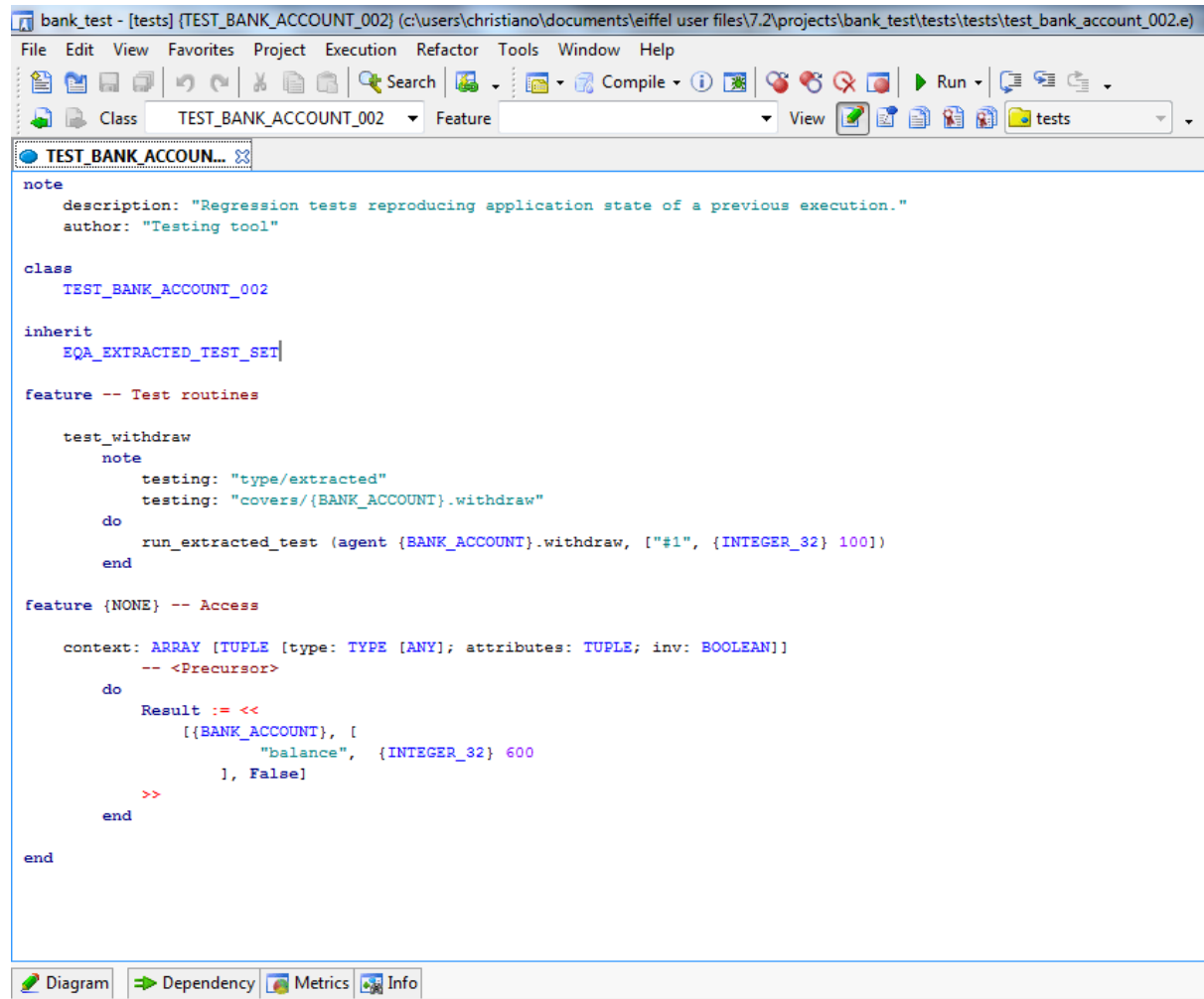
The 'Call Stack' window shows the following stack:

- balance\_decreased: POSTCONDITION\_VIOLAT...
- withdraw BANK\_ACCOUNT BANK\_A...
- make APPLICATION APPLICA...

# Demo – Test Extraction



# Demo – Extracted Test



```
bank_test - [tests] [TEST_BANK_ACCOUNT_002] (c:\users\christiano\documents\ieffell user files\7.2\projects\bank_test\tests\tests\test_bank_account_002.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
[Icons] Search [Icons] Compile [Icons] Run [Icons]
Class TEST_BANK_ACCOUNT_002 Feature View [Icons] tests
TEST_BANK_ACCOUNT_002
note
  description: "Regression tests reproducing application state of a previous execution."
  author: "Testing tool"

class
  TEST_BANK_ACCOUNT_002

inherit
  EQA_EXTRACTED_TEST_SET

feature -- Test routines

  test_withdraw
    note
      testing: "type/extracted"
      testing: "covers/{BANK_ACCOUNT}.withdraw"
    do
      run_extracted_test (agent {BANK_ACCOUNT}.withdraw, ["#1", {INTEGER_32} 100])
    end

feature {NONE} -- Access

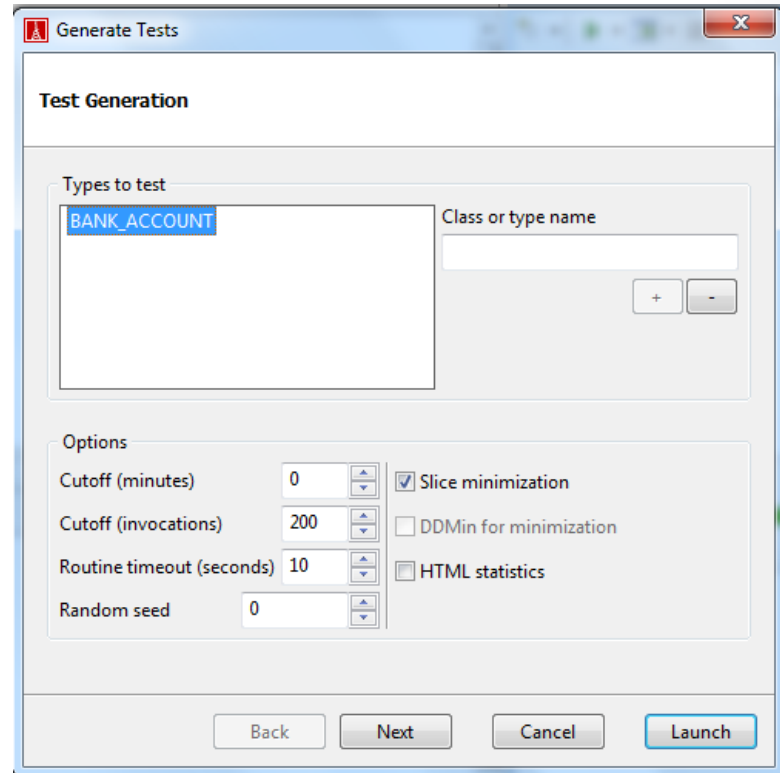
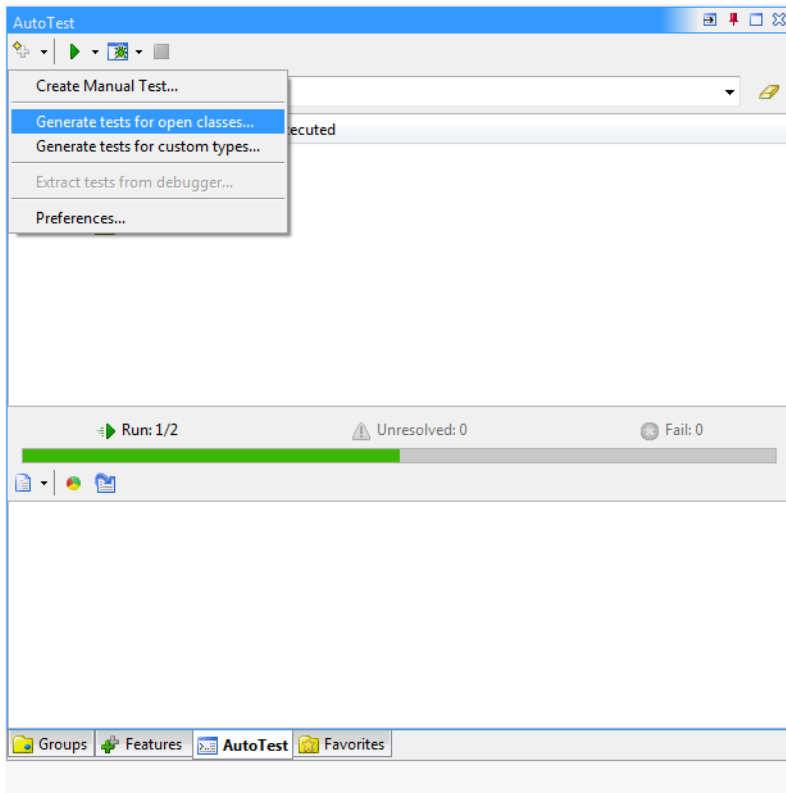
  context: ARRAY [TUPLE [type: TYPE [ANY]; attributes: TUPLE; inv: BOOLEAN]]
  -- <Precursor>
  do
    Result := <<
      [{BANK_ACCOUNT}, [
        "balance", {INTEGER_32} 600
      ], False]
    >>
  end

end

end

Diagram Dependency Metrics Info
```

# Demo – Test Generation



# Demo – Generated Test

```
Outputs
Output: Testing
194: BANK_ACCOUNT.deposit (passed)
193: create BANK_ACCOUNT.default_create (passed)
192: BANK_ACCOUNT.balance (passed)
191: BANK_ACCOUNT.default_create (passed)
190: create BANK_ACCOUNT.default_create (passed)
189: BANK_ACCOUNT.withdraw (invalid test)
188: BANK_ACCOUNT.deposit (invalid test)
187: BANK_ACCOUNT.withdraw (invalid test)
186: BANK_ACCOUNT.default_create (passed)
185: BANK_ACCOUNT.balance (passed)
184: BANK_ACCOUNT.deposit (invalid test)
183: BANK_ACCOUNT.default_create (passed)
182: BANK_ACCOUNT.balance (passed)
```

Class Feature Outputs Error List AutoTest Results

```
TEST BANK ACCOUN...
inherit
  EQA_GENERATED_TEST_SET

feature -- Test routines

  generated_test_1
    note
      testing: "type/generated"
      testing: "covers/{BANK_ACCOUNT}.withdraw"
    local
      v_22: BANK_ACCOUNT
      v_23: INTEGER_32
      v_27: detachable ANY
      v_40: INTEGER_32
      v_75: detachable ANY
      v_106: INTEGER_32

    do

      execute_safe (agent: BANK_ACCOUNT
        do
          create {BANK_ACCOUNT} Result
        end)
      check attached {BANK_ACCOUNT} last_object as l_ot1 then
        v_22 := l_ot1
      end
      v_23 := {INTEGER_32} 5
      execute_safe (agent v_22.deposit (v_23))
      execute_safe (agent v_22.balance)
      v_27 := last_object
      v_40 := {INTEGER_32} 9
      execute_safe (agent v_22.deposit (v_40))
      execute_safe (agent v_22.balance)
      v_75 := last_object
      v_106 := {INTEGER_32} 6

      -- Final routine call
      set_is_recovery_enabled (False)
      execute safe (agent v_22.withdraw (v_106))
```