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

Routine exercising using ART

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

Objects pool
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

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

```plaintext
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

```eiffel
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
```
Demo – Test Execution
Demo – Application Class

class APPLICATION
  inherit
    ARGUMENTS
  create
    make
    feature {NONE} -- Initialization
      make
      -- Run application.
      do
        create my_account
        my_account.deposit (660)
        my_account.withdraw (100)
      end
      my_account := BANKACCOUNT
    end

Outputs

Degree 1: Generating Metadata
Merging System Changes

Eiffel Compilation Succeeded
Demo – Failed Execution
Demo – Test Extraction
Demo – Extracted Test

```plaintext
// bank_test - [test] (TEST_BANK_ACCOUNT_002) (c:\users\christiano\documents\effel user files\7.2\projects\bank_test\tests\tests\test_bank_account_002.e)

File Edit View Favorites Project Execution Refactor Tools Window Help

Class TEST_BANK_ACCOUNT_002

inherit EDA_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]]

  do
  Result := <<
  [(BANK_ACCOUNT), |
  "balance", (INTEGER_32) 800 |
  ], False]
  >>
  end

end
```
Demo – Test Generation
Demo – Generated Test

```
PREP Inventory

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))
```