Contract Driven Development

= Test Driven Development – Writing Test Cases

By
Andreas Leitner, et al.

Mikael Malm
mmal@cs.aau.dk

17 October 2007
The Article and Authors

- Published in proceedings of the 6th joint meeting of ESEC-FSE '07
- Written by 5 researchers at the Chair of Software Engineering, Swiss Federal Institute of Technology
- Research on Design by Contract, formal methods, proofs, contract-based automatic testing
- Andreas Leitner and Ilinca Ciupa
  - Ph.D. students
- Arno Fiva
  - Student
The Article and Authors

- Manuel Oriol
  - Non-tenured lecturer at CSE

- Bertrand Meyer
  - Professor and head of the CSE
  - Designer of the Eiffel language

- The authors has published several papers on automated testing within the last few years.
Context and Motivation

- Conducted study internally at ETH
  - Only about 20% always wrote unit tests either before or after implementation
  - Time to write and maintain unit tests was cited as the main reason for not writing them
- Several tools exist to (almost) fully automate the process of testing
  - AgitarOne, DART, etc.
  - But these do not leverage the developer’s knowledge of the semantics of the program
  - Some aspects of testing are inherently difficult to automate
Context and Motivation

- The developer typically runs the application during development, implicitly testing the program
  - To test the program in general
  - To do a quick test of the current feature
- The method presented in the paper captures these tests and makes them explicit
Context and Motivation

- **CDD vs. TDD**
  - Contracts are a lightweight, executable specification and can express more about intended semantics than test cases
  - But unlike test cases, they are not automatically executable
  - They lack concrete instances satisfying the preconditions of the method under test

- The cdd tool provides these and unifies the advantages of contracts (thorough specification) and test cases (concrete and executable)
Overview of CDD development

- Contracts are written as part of the coding activity before the implementation.
- The developer runs the application during development.
- If a failure occurs, the IDE breaks into debugger.
- Cdd tool extracts a test case that triggers the failure.
- This test case is then stored in a test suite and can be run continuously during development.
Example

class BANK_ACCOUNT
    inherit
        ANY
    redefine
        default_create
    end
feature
    default_create
        do
            balance := 300
        end
balance: INTEGER
deposit (an_amount: INTEGER)
    do
        ensure
            balance_increased: balance > old balance
            deposited: balance = old balance + an_amount
        end
    ...
    invariant
        balance_not_negative: balance >= 0
end
Example

class TEST_CASE_1
feature
  test
  local
    ba: BANK_ACCOUNT
  do
    ba := new_object ("BANK_ACCOUNT")
    set_field (ba, "balance", 300)
    checkInvariant (ba)
    ba.deposit (30)
  end
end
How it works

- Automatically extract a test case
- Minimizes it to the part of the application relevant for the failure
- Free it from external state (the GUI) and non-determinism (the user input)
How it works

- Test case extracted from program trace
  - Instructions and context

- Instructions
  - Object Creation (allocation and construction)
  - Method calls, delegates

- Context
  - Program state and bindings of instruction
  - For a method call this is the target object and arguments
How it works

- Every node in the tree can trigger a failure (origin)
- The recipient of the failure is the invocation responsible for the failure
How it works

- Recreate context $c$
- Check invariant of $c$
  - violation $=$ invalid test case
- Check precondition of instruction $i$ in $c$
  - violation $=$ invalid test case
- Run instruction $i$ in $c$
  - if normal termination and post-condition is satisfied pass, otherwise fail
Other Areas of Application

- Failure test suites
  - Create test suite by capturing whole call stack
  - Can aid debugging
- Extracting test suites from system level tests
  - High level black box tests usually achieve high coverage but are easier to maintain
Future Work

- Concurrency
  - Not yet implemented, but could be done with minimal effort

- Selective Capture and Replay
  - Pre-state extraction
  - Non-determinism
  - External state
My Impression of the paper

- Mostly a good article
- Didn’t see the need for the small study on unit testing habits in the motivation section.
- Liked that they described the implementation in a language agnostic way
Questions