Test Design Patterns
Method scope

Błażej Pietrzak
blazej.pietrzak@cs.put.poznan.pl

Binder's Patterns

- Class (12)
  - Method Scope (4)
  - Class Scope (4)
  - Class Integration Scope (2)
  - Flattened Class Scope (2)
- Reusable Component (4)
- Subsystem (4)
- Integration (9)
- Application Scope (3)
- Regression Test (5)
- Assertions (1)
- Test Harness Design (16)
  - Test Case (3)
  - Test Control (2)
  - Driver (8)
  - Execution (3)

Why Test at Class Scope?

- Many bugs only found at class scope
- Cannot fully excersise a class through its clients
- Fix bugs close to creation
  - The longer you wait the more expensive it is
  - Debugging during system test sucks
- Reduces schedule risk
- Improves productivity
- Don’t delegate testing to your clients

Test Design Approach

1. Make a preliminary estimate of class under test
   - Plan budget for testing
2. Design and code a test driver
   - For non trivial classes begin with alpha-omega skeleton
   - After the alpha-omega tests pass, add additional tests
3. Select a class scope test pattern
4. Select test design patterns for each method
5. Arrange method test cases
   - According to sequence called for by the class scope pattern
6. Build the test package
   - When all tests pass, evaluate coverage
   - If coverage is insufficient, develop more tests

Class Scope Integration

Goal

- Demonstrate that class under test is ready to test

Two approaches

- Small pop
- Alpha-Omega cycle

Small pop

- A Big Bang integration at class level
- Excersise all untrusted components simultaneously
- Effective when
  - CUT is small and simple
  - Servers of the class are stable
  - Inherited features are stable
  - Few intraclass dependencies exist

Small pop – cont.

- Process
  - Develop entire class
  - Write a test driver using any appropriate test pattern
  - Run the test suite
  - Debug as needed

Alpha-Omega Cycle

- Alpha-Omega States:
  - Alpha State: object before it is constructed
  - Omega State: object after it has been deleted
- Alpha-Omega Cycle: Take an object from alpha to omega
  - Send a message to every method at least once
  - Shows that class is ready for more extensive testing
  - No attempt is made to achieve any coverage

Alpha-Omega states

- Order of calling methods:
  - Constructor method
  - Accessor (get) method
  - Boolean (predicate/is) method
  - Modifier (set) method
  - Iterator method
  - Destructor method
  - Order within steps: private, protected, public

Method Scope Pattern: Combinational Function Test

- Intent
  - Design test suite for behaviors selected according to combinations of state and/or message values
- Context

Method Scope Pattern: Combinational Function Test

- Fault Model
  - Incorrect or missing
    - Assignment to a decision variable
    - Operator or variable in predicate
    - Structure in a predicate („dangling else”, misplaced semicolon etc.)
    - Default case
    - Action(s)

Method Scope Pattern: Combinational Function Test

- Fault Model – cont.
  - Extra action(s)
  - Structural errors in decision table implementation (e.g. ommited or misplaced break in switch)
  - Bad type or incorrect value in object representing condition or action that can cause a binding that produces bad action (e.g. bad type in polymorphic method)
  - General errors (e.g. ambiguous requirements)

Method Scope Pattern: Combinational Function Test

- Strategy
  - Decision table with Conditions/Actions
  - At least one test for each action
  - Excersise boundaries of non-boolean variables
- Entry Criteria
  - Small pop. Minimal feasibility assures branch coverage within method

Method Scope Pattern: Combinational Function Test

- Exit Criteria
  - Produce every action at least once
  - Force each exception at least once
  - Exercise at least every branch
  - If polymorphic binding is used, select each binding at least once (branch coverage is not reliable for polymorphic binding)

Method Scope Pattern: Combinational Function Test

- Consequences
  - Detects faults that are incorrect response actions to test messages
  - Faults resulting from the order of messages to other methods or faults corrupting object variables hidden by the MUT interface may not be shown

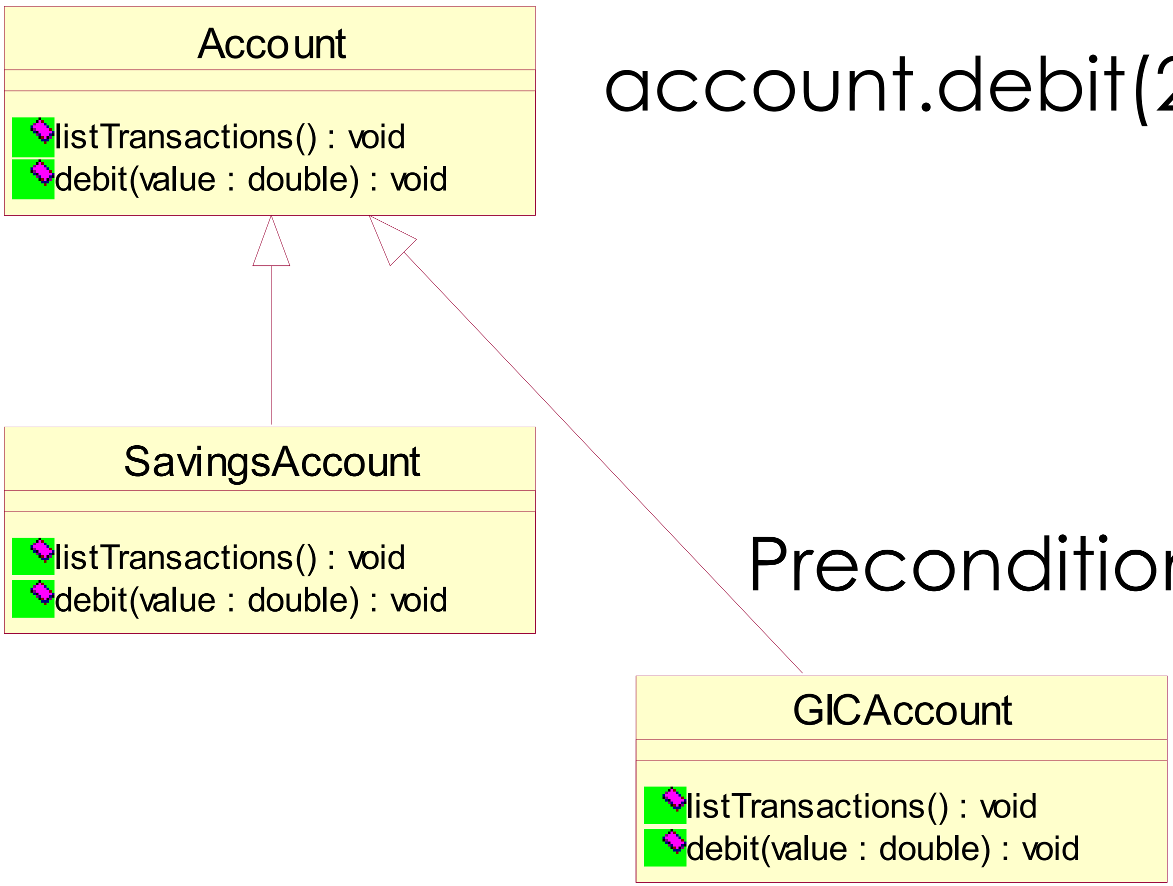
Method Scope Pattern: Polymorphic Message Test

- Intent
  - Design a test suite for a client of a polymorphic server that exercise all client bindings to the server
- Context
  - Would you have confidence in code for which only a fraction of statements or branches were executed?

Method Scope Pattern: Polymorphic Message Test

- Fault model
  - Client fails to meet all preconditions for all possible bindings
  - Unanticipated binding occurs – possibly because of an incorrect construction of a pointer (i.e. Incorrect indexing into an array)
  - Superclass is changed – subclasses are rendered inconsistent because of the change

Precondition Failure



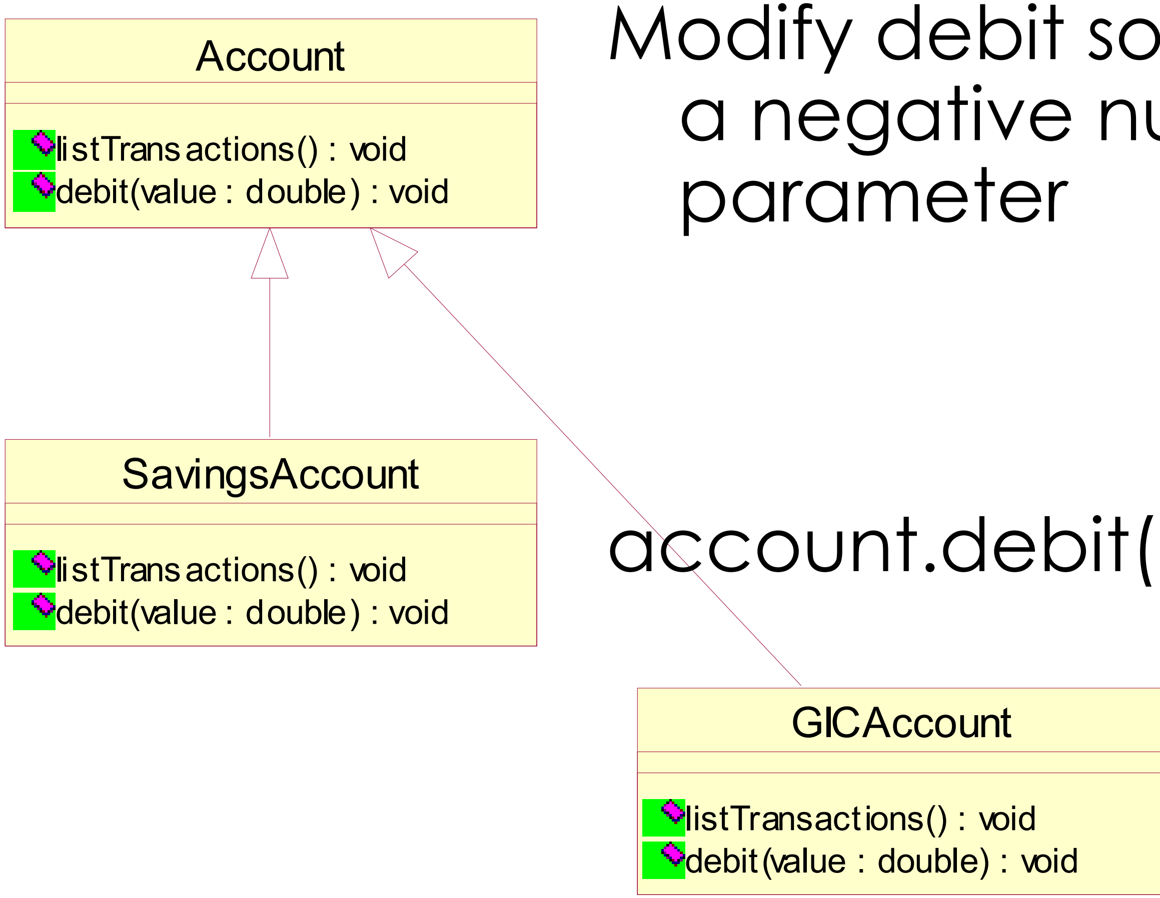
account.debit(2.0);

Precondition: debit > 500

Unanticipated binding

```
Customer.report() {
    Account[] list = new Account(MAX);
    int index = savingsIndex();
    list[index] = getSavingsAccount();
    ...
    index = gicIndex();
    list[index] = getGICAccount();
    ...
    System.out.println("Savings");
    index = getGICAccount(); // wrong function!
    list[index].listTransactions();
}
```

Superclass change



Modify debit so that it takes a negative number as parameter

account.debit(-2.0);

Method Scope Pattern: Polymorphic Message Test

- Strategy
  1. Determine the number of bindings for each message sent to a polymorphic server object
  2. Test for all possible bindings
  3. Test for run-time binding error

Method Scope Pattern: Polymorphic Message Test

- Entry Criteria
  - Small pop. Minimal feasibility is assured by branch coverage within method
  - Server class should be stable
- Exit Criteria
  - Each binding with polymorphic server should be tried at least once (achieve branch coverage of extended message flow graph)

Domain Testing Model

- Purpose: find test values
  - Subset of all possible values
  - Partition the input value set
  - Find best test values
- Proposed Methods
  - Equivalence Partitioning (classes of values causing same result)
  - Boundary Value Analysis (special class domain boundary)

Domain Testing Model – cont.

- Improved approach: domain testing model
  - Based on fault model
  - Select values based on value types and type domains
  - Solution for complex types (objects)

Boundary Values: Points

On, Off and Out Points

- On point (pol. *Punkt brzegowy*) lies on a boundary
- In point (pol. *Punkt wewnętrzny*) is within boundary and not on boundary
- Off point (pol. *Punkt pozabrzegowy*) lies not on a boundary
- Out point is outside boundary

Boundary Values: Points Example

Condition	Off point rule	On point	Off point
y >= 1.0	Closed, false	1.000000	0.999999
x <= 10	Closed, false	10	11
y <= 10.0	Closed, false	10.000000	10.00001
x > 0	Open, true	0	1
y <= 14.0 - x	Closed, false	7.000000	7.000001
!Stack.isFull()	Closed, false	32676	32767

Precision 0.000000

Boundary Values with 2 or more variables

- Solve equation
- constraints must be fulfilled
- Check central points of independent variable first

```
x > 0, x <= 10
y >= 1.0, y <= 10.0
y = 14.0 - x
```

so x must be x >= 4 and x <= 10

central point is x = 7

in point is y = 7.0 for x = 7

off point is y = 7.000001 for x = 7

Boundary Values: Points – cont.

- What about complex types (classes)?
  - Use state and state invariant
  - State on point
    - Smallest possible variable change produces state change
  - State off point
    - Valid state that is not boundary state and differs minimally
  - State in point
    - Any valid state that is not a state on point

1x1 Domain Testing Strategy

- One on point and one off point for each domain boundary
  - For each relational condition
  - For each nonscalar type
  - For each nonlinear boundaries
- Special case Equality condition
  - One on point and two off points for each equality condition
  - X == 10: on point is 10, off points are 9 and 11
- State invariant
  - One on point and at least one off point for each state invariant
  - Condition for invariant is once true, once false

Domain Test Matrix

- Add expected results and in points for other values
  - Each test case only has one off or on point
  - Select in points for all other values in the test case
  - Avoid to repeat in points (increase chance of finding bugs)
- Result
  - Minimal test cases
  - Input variables to excersise boundary conditions
  - For any type of variable types
  - Including abstract complex types (objects)

Method Scope Pattern: Category-Partition

- Intent
  - Design method scope test suites based on input/output values
- Context
  - How can we develop a test suite to exercise the functions implemented by a single method?
- Fault model
  - Combinations of message parameters and instance variables and these faults with result in missing or incorrect method output

Method Scope Pattern: Category-Partition

- Strategy
  1. Identify all functions of the method
     - Method may implement several functions
     - Other functions may be side-effects of the primary function i.e. The current position of List object may be incremented as side-effect of returning the next element
  2. Identify all input and output parameters of each function
  3. Identify categories for each input parameter
  4. Partition each category into choices
  5. Identify constraints on choices
  6. Generate test cases by enumerating all choice combinations

Method Scope Pattern: Category-Partition

- Entry Criteria
  - Small pop
- Exit Criteria
  - Every combination is tested once
  - Achieve branch coverage in the MUT

Method Scope Pattern: Category-Partition

- Consequences
  - Identification of categories and choices is subjective - some bugs may not be revealed
  - Many test cases for even moderately complex methods
  - The Invariant Boundaries pattern may be used to produced smaller test suite
  - With proper attention to interface details, a superclass Category-Partition method test suite may be reused to test an overriding subclass method i.e. getNextElement test suite could be repeated for List subclasses such as PersonList etc.

Category Partition Example

```
Object getNextElement();
```

- Functions
  - Return next element
  - Keep track of last position and wrap from last to first
  - Throw NoPosition and EmptyList exceptions if appropriate

Category Partition Example

- Inputs
  - Position of last referenced element
  - List state
- Outputs
  - Element returned
  - Incremented position pointer

Category Partition Example

- Categories and Choices
  - Position of last element
    - Category nth element
      - Min, in-between, max
    - Category Special Cases
      - Undefined, First, Last
  - State of the list
    - Category m-elements
      - Min, in-between
    - Category Special Cases
      - Empty, single entry, full (max)

Literature and Links

- Robert V. Binder: „Testowanie systemów obiektowych. Modele wzorce i narzędzia." WNT 2003
- Code coverage tools for Java:
  - Clover (commercial) http://www.thecortex.net/clover/
  - JCoverage (commercial, but also GPL license) http://www.jcoverage.com/
  - GroboUtils (MIT license) http://groboutils.sourceforge.net/
  - Hansel (BSD license) http://hansel.sourceforge.net/

Quality Assessment

Thank You for your attention ☺

- What is your general impression (1-6)
- Was it too slow or too fast?
- What important did you learn during the lecture?
- What to improve and how?