Test Design Patterns
Class scope

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

Design by Contract

- Precondition
  - Conditions that must hold before a method can execute
  - Client class must fulfill precondition
- Postcondition
  - Specify conditions that must hold after a method completes
  - Server class must fulfill postcondition
- Class invariant
  - Specifies a condition that must hold anytime a client class could invoke an object's method
  - Must be true on entry and exit of public methods

Design by Contract Structure

- Constructor
  - Before: check class precondition
  - After: check class invariant, check constructor postcondition
- Destructor
  - Before: check class invariant, check destructor precondition
  - After: check destructor postcondition
- Method
  - Before: check class invariant, check method precondition
  - After: check method postcondition, check class invariant

Design by Contract
Subcontracting
(Contracts and Inheritance)

- Contracts are inherited
- Subclass must fulfill its superclass's public type contract
- „*Require no more ensure no less!*"
- Inheritance rules
  - Subclass may keep or strengthen the class invariant
  - Subclass may keep or weaken the precondition
  - Subclass may keep or strengthen the postcondition

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

Invariant Boundaries

- Intent
  - Select test-efficient test value combinations for classes, interfaces, and components composed of complex and primitive data types
- Context
  - The valid and invalid combinations of instance variable values may be specified by the class invariant
  - The class invariant typically refers to instance variables that are instances of primitive and complex data type
  - The Invariant Boundaries pattern does not consider input/output relationship or message sequence

Invariant Boundaries – cont.

- Fault Model
  - Bugs in implementation of constraints needed to define and enforce a domain formed by several complex boundaries
  - Subclass invariants are often implicit or accidental, leading to misinterpretation and misuse
- Strategy
  1. Develop the class invariant
  2. Develop on points and off points for each condition in the invariant using 1x1
  3. Develop in points for variables not referenced in a condition
  4. Represent the results in a domain matrix

Invariant Boundaries – cont.

- Entry Criteria
  - The class invariant exists or can be created
  - test suite's entry criteria designed with another pattern must be fulfilled
- Exit Criteria
  - A complete set of domain tests has been developed

Invariant Boundaries – cont.

- Consequences
  - The class invariant development is difficult and time consuming
  - After the class invariant is developed test suite creation is fast
  - The number of test cases grows linearly with the number of conditions in the invariant

Invariant Boundaries Example

```
class ClientProfile {
    Account account = new Account();
    Money creditLimit = new Money();
    short trCounter;
...
}
```

Account abstract states: Opened, Closed, Idle, Locked, Debit
creditLimit is in range ±999 999 999 999.99
creditLimit >= trCounter * 100 + 100
trCounter must be >= 0 and <= 5000

Invariant Boundaries Example

- Define Class Invariant

```
assert (trCounter >= 0) && (trCounter <= 5000)
    && (creditLimit>= trCounter * 100 + 100)
    && !account.isClosed());
```

- Develop on points and off points

Condition	On point	Off point
trCounter >= 0 trCounter <= 5000 creditLimit >= trCounter * 100 + 100 !account.isClosed()	0 5000 100 Opened	-1 5001 250100.01 Closed

Invariant Boundaries Example – cont.

- Develop on points and off points – cont.

$\texttt{creditLimit}_{min} = (0 * 100) + 100 = 100.00$

$\texttt{creditLimit}_{max} = (5000 * 100) + 100 = 500\ 100.00$

$\texttt{trCounter}_{central} = (0 + 5000) / 2 = 2500$

$\texttt{creditLimit}_{central} = (2500 * 100) + 100 = 250\ 100.00$

$\texttt{creditLimit}_{off} = \texttt{creditLimit}_{central} + 0.01 = 250\ 100.01$

Invariant Boundaries Example – cont.

- Represent the results in a domain matrix

Constraint			Test case							
Variable	Condition	Point	1	2	3	4	5	6	7	8
trCounter	>= 0	On	0							
		Off		-1						
	<= 5000	On			5000					
		Off				5001				
	Typical	In					3500	8	99	1037
creditLimit	>= trCounter * 100 + 100	On					100			
		Off						2..1		
	Typical	In	8327.62	9999,99	104.0	732.86			783.0	700.15
Account	!isClosed()	On							open	
		Off								closed
	Typical	In	open	open	debit	Lockd	idle	open		

Invariant Boundaries Example – cont.

- Represent the results in a domain matrix – cont.
  - Each test case has only one on or off-point
  - Select in-points for all other values in the test case
  - Avoid to repeat in points (increase chance of finding bugs)

Class Modalities Nonmodal

- *„Accept any message in any state"*
- No domain state constraint
- No message sequence constraint
- Classes that implement basic data types are often nonmodal i.e. Time

Class Modalities
Nonmodal class example

```
class Time {
    private int hh24, mm, ss;
    public Time(int hh24, int mm, int ss) { ... }
    public void setHour(int value) { ... }
    ...
    public void getSecond() { ... }
    public boolean equals(Time t) { ... }
    public Time subtract(Time t) { ... }
}
```

- Only one abstract state (any possible time) and many concrete states (e.g. 23:00:01, 12:15:49 etc.)
- Any possible message can accur after any possible message with exception for constructor and destructor

Class Modalities Unimodal

- No domain state constraint
- Message sequence constraint
- Classes that are application controllers are usually unimodal
- Example: traffic lights
  turnOnRedLight can only be accepted after turnOnYellowLight
  turnOnGreenLight – only after turnOnRed, turnOnYellowLight

Class Modalities Quasi-modal

- Imposes sequential constraints on message acceptance that change with the contents of the object
- Many container and collection classes are quasi-modal i.e. Stack
- **Behaves like modal class, but message sequence constraints are implicitly connected with the contents of the object** i.e. stack - size determines whether stack is full, empty, or loaded. It doesn't matter what are the contents of the stack or in what order.
- Test model is focused on parameters describing the contents i.e. stack size
- **In modal class the contents control the behavior directly**

Class Modalities
Modal

- Domain state constraint
- Message sequence constraint
- Example Account

Class Modalities
Modal example

```
class Account {
    private Money saldo;
    private int accountNumber;
    private Date lastOperation;
    public void open() { ... }
    public Money getSaldo() { ... }
    public void credit(Money creditAmount) { ... }
    public void debit(Money debitAmount) { ... }
    public void lock() { ... }
    public void unlock() { ... }
    public void regulate() { ... }
    public void close() { ... }
}
```

- If [saldo == 0] then close()
- If isClosed it cannot accept close()

Nonmodal Class Test

- Intent
  - Develop a class scope test suite for a class that does not constrain message sequences
- Context
  - Nonmodal class imposes few constraints on message sequence but usually has a complex state space and a complex interface
  - Few or no message sequences are illegal; most message sequences are legal

Nonmodal Class Test – cont.

- Fault Model
  - Sequential constraints on messages are not imposed so the state control model is trivial
  - State-based testing (see Modal Class Test) is inappropriate
  - Allowed sequence is rejected
  - Allowed sequence produces bad value
  - Methods reporting about abstract state are inconsistent
  - Allowed modifier sequence is rejected
  - Not allowed modifier's argument is accepted resulting in corrupted state
  - Accessor method has incorrect side effect, which changes or corrupts object state
  - Not allowed calculation violates class invariant
  - Some modifier or accessor methods cause inconsistent abstract state view

Nonmodal Class Test – cont.

- Strategy
  1. Develop a set of test cases using **Invariant Boundaries** pattern
  2. Select a message sequence strategy: define-use (setters and getters), random (setters and getters in random order), or suspect (only suspected combinations of setters and getters)
  3. Set the Object Under Test to a test case from the domain matrix
  4. Send all the accessor (get) messages
  5. Verify that the returned and resulting values are consistent
  6. Repeat until all sequences have been exercised

Nonmodal Class Test – cont.

- Basic test strategy
  1. To set the Object Under Test to a test vector value with a modifier (set, constructor)
  2. Verify that modifier has produced a correct state
  3. Try each accessor (get) to see that it reports the state correctly without buggy side effect

Nonmodal Class Test – cont.

- Entry Criteria
  - Alpha – omega cycle
- Exit Criteria
  - Object Under Test has taken values of each test case at least once
  - Achieve at least branch coverage on each method in the Class Under Test

Nonmodal Class Test – cont.

- Consequences
  - The number of tests is m^{n+1} where m is the number of methods in CUT, n is a number of modifier – accessor pairs – typically n = 2

Nonmodal Class Test Example

```
public void testTime1() {
    Time temp = new Time(0, 4, 9);
    assertTrue(0, temp.getHour());
    assertTrue(4, temp.getMinute());
    assertTrue(9, temp.getSecond());
}
public void testTime2() {
    time.setMinute(4);
    time.setHour(0);
    time.setSecond(9);
    assertTrue(0, temp.getHour());
    assertTrue(9, temp.getSecond());
    assertTrue(4, temp.getMinute());
}
```

```
public void testTime3() {
    try {
        Time temp = new Time(-1, 6, 8);
        fail();
    } catch (SomeException ex) { }
}
...
```

Quasi-modal Class Test

- Intent
  - Develop a class scope test suite for a class whose constraints on message sequence change with the state of the class
- Context
  - A quasi-modal class has sequential constraints that reflect the organization of information used by the class
  - Container and collection classes are often quasi-modal
  - **Effective testing must distinguish between content that determines behavior and content that does not affect behavior**

Quasi-modal Class Test – cont.

- Fault Model
  - Quasi modal class failures occur
    - When invariants related to all members of a collection are not observed
- Strategy

1. Create N+ state model test suite
   - Use N+ test strategy
   - Model constraint parameters as states
   - Develop test data using invariant boundaries
2. Run class specific operation-pairs

Quasi-modal Class Test – cont.

- Strategy detailed

2. Develop FREE state model. Characterize state with constraint params not content. Treat each constraint param as state variable.
3. Generate transition tree. There is no need to indicate loops (*).
4. Tabulate events and actions along each path.
5. Develop test data for each path using **Invariant Boundaries** pattern for events, messages and actions.
6. Execute conformance test suite until all tests pass.
7. Develop sneak path test suite. Add all forbidden transitions in all states and define exception to be thrown.
8. Execute sneak path test suite until all tests pass.
9. Run class specific operation-pairs.

Quasi-modal Class Test – cont.

- Entry Criteria
  - Alpha – omega cycle
- Exit Criteria
  - Achieve at least branch coverage on each method
  - Provide N+ coverage
  - Optionally develop a class flow graph
    - Identify uncovered alpha-omega paths
    - Excersise those paths

Quasi-modal Class Test – cont.

- Consequences
  - Testable behavior model is available or can be developed
  - CUT is state observable – we have trustworthy built-in tests or feasible access to the CUT so we can determine the resultant state of each test run

Quasi-modal Class Test – Example

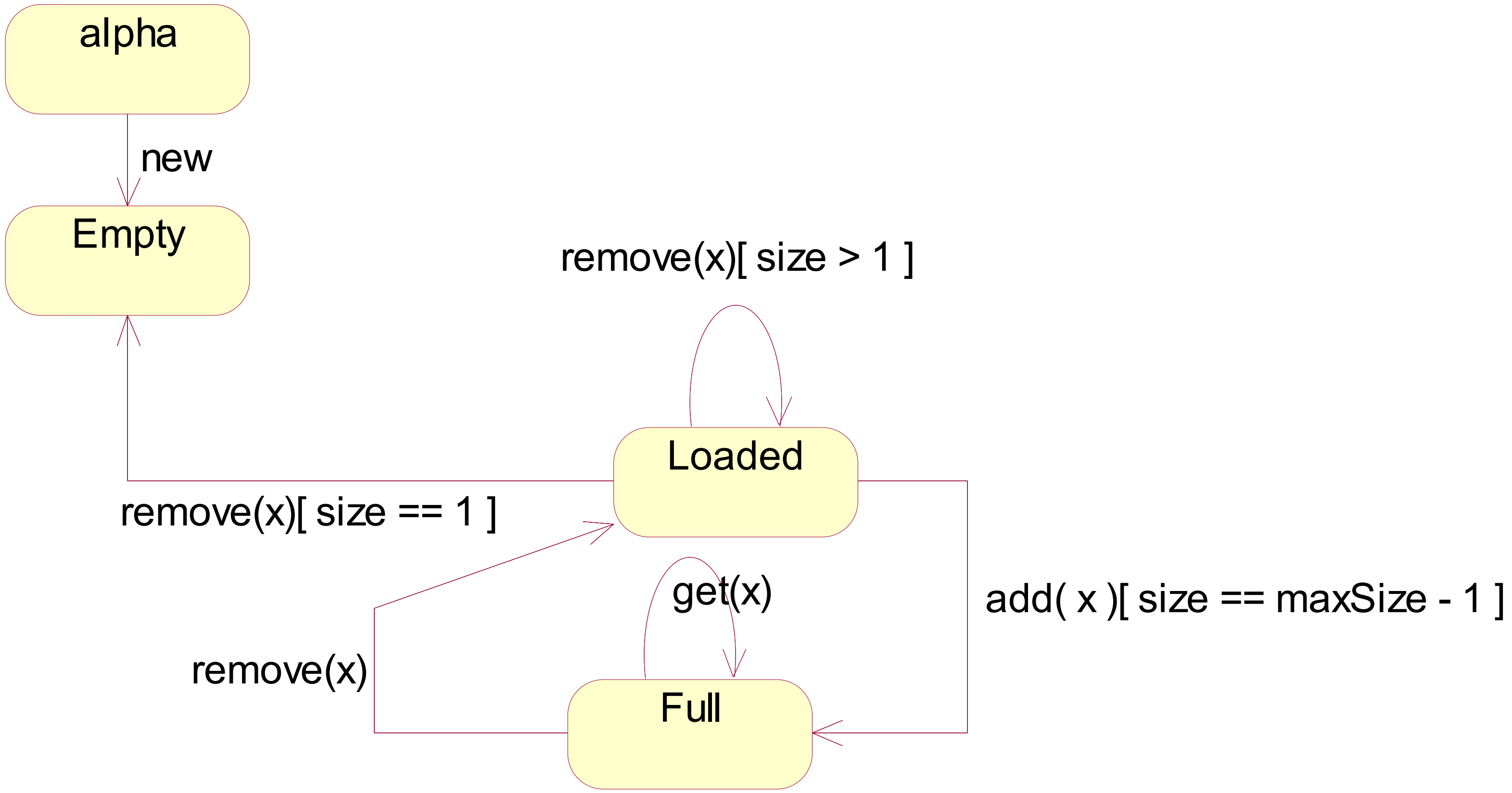
```
class MyList {
    protected int size;
    final public static int MAX = 1000;
    protected List list = new Vector();

    public void add(Object obj)
        throws DuplicateException { ... }

    public void remove(Object obj) { ... }
    public Object get(int idx) { ... }
}
```

Class Invariant *(not content)*

```
size >= 0 && size <= MAX
```

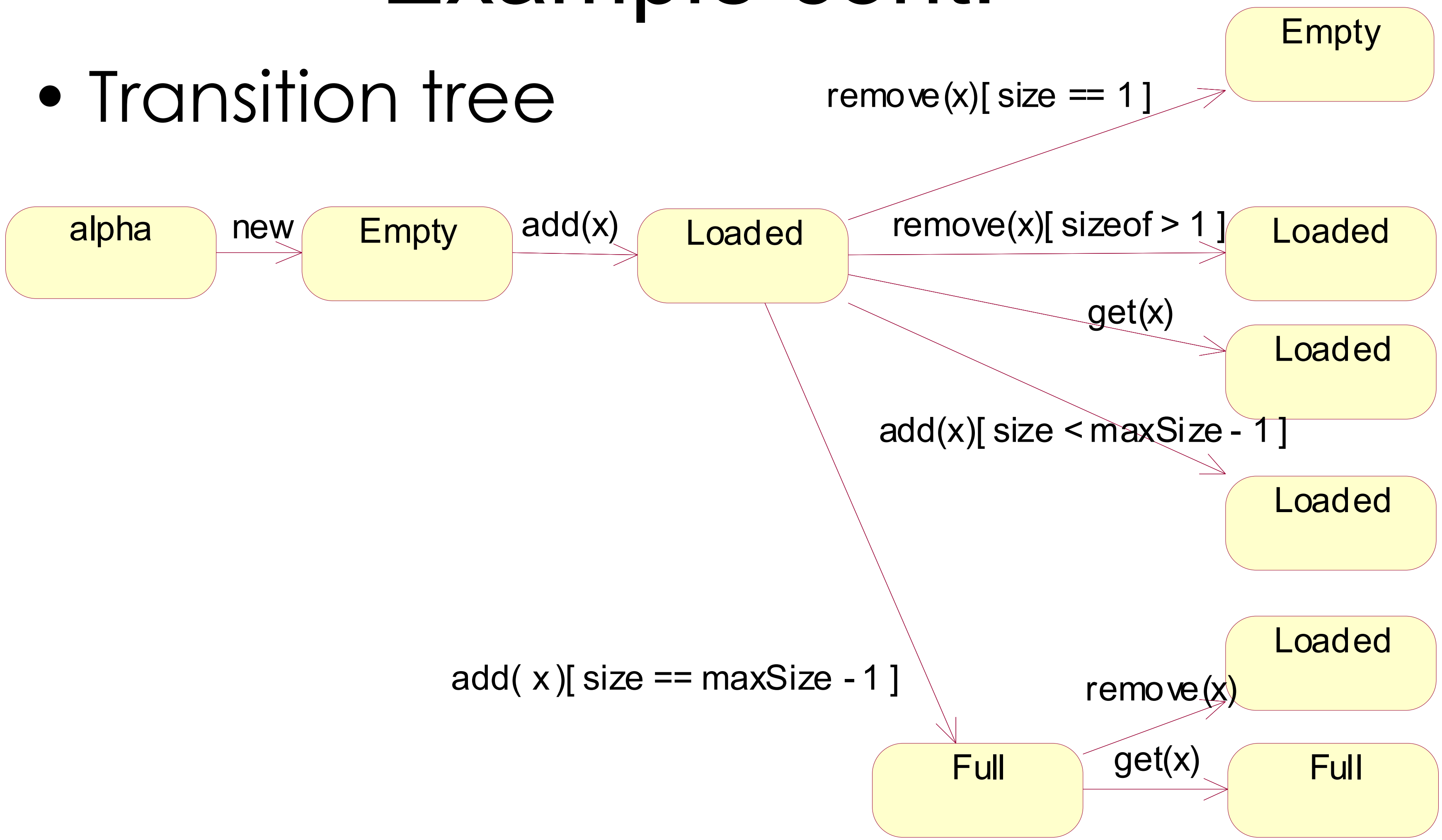
Quasi-modal Class Test – Example cont.

- State chart (not all transitions are shown)



Quasi-modal Class Test – Example cont.

- Transition tree



Quasi-modal Class Test – Example cont.

- Invariant Boundaries for add

Condition	On point	Off point
size >= 0 size < MAX – 1 size == MAX –1	0 MAX – 1[1] MAX – 1[1]	-1 MAX[1] MAX – 2, MAX[1]

[1]create only one test case for this value instead of two ☺

Quasi-modal Class Test – Example cont.

Conformance test suite

```
public void testAdd1() {
    MyList myList = new MyList();
    myList.add(new Object());
    assertTrue(myList.isLoaded());
    assertFalse(myList.isEmpty());
    assertFalse(myList.isFull());
    assertEquals(1, myList.size());
}
public void testAdd2() {
    MyList myList = new MyList();
    // set size to MAX - 1
    myList.add(new Object());
    assertTrue(myList.isFull());
    assertFalse(myList.isEmpty());
    assertFalse(myList.isLoaded());
    assertEquals(MyList.MAX, myList.size());
}
```

```
public void testAdd3() {
  MyList myList = new MyList();
  // set size to MAX - 2
  myList.add(new Object());
  assertTrue(myList.isLoaded());
  assertEquals(myList.maxSize - 1,
myList.size());
  assertFalse(myList.isEmpty());
  assertFalse(myList.isFull());
}
```

Quasi-modal Class Test – Example cont.

- Response matrix for add

Events and guards			Accepting state – expected output			
			a	Empty	Loaded	Full
add(x)	size < MAX - 1	size == MAX - 1				
	DC	DC	X	✓	X	4
	False	False	X	✓	4	4
	True	False	X	✓	✓	4
	False	True	X	✓	✓	4
	True	True	X	X	X	X

DC – don't care
X – excluded
✓- explicit transition
4 – Rejection - throw IllegalEventException

Quasi-modal Class Test – Example cont.

Sneak path test suite

```
public void testAdd4() {
  MyList myList = new MyList();
  // set size to MAX
  assertTrue(myList.isFull());
  try {
      myList.add(new Object());
      fail();
   } catch (IllegalEventException ex)
{ }
   assertTrue(myList.isFull());
  assertFalse(myList.isEmpty());
  assertFalse(myList.isLoaded());
  assertEquals(MAX, myList.size());
}
```

```
public void testAdd5() {
    MyList myList = new MyList();
    myList.add(new Object());
    assertTrue(myList.isLoaded());
    myList.size = MyList.MAX;
    try {
        myList.add(new Object());
        fail();
    } catch (IllegalEventException ex)
{ }

// check state analogically
}
...
```

Quasi-modal Class Test – Example cont.

Run class specific operation-pairs

```
public void testAddDuplicate() {
    assertEquals(x, y);
    myList.add(x);
    // ... check state
    try {
        myList.add(y);
        fail();
    } catch (DuplicateException ex)
   { }
    // ... check state
}
```

```
public void testAddRemove() {
    Object elem = new Object();
    myList.add(elem);
    // ... check state
    myList.remove(elem);
    assertEquals(0, myList.size());
 assertFalse(myList.contains(elem));
    // ... check state
}
...
```

Modal Class Test

- Intent
  - Develop a class scope test suite for a class that has fixed constraints on message sequence
- Context
  - A modal class has both message and domain constraints on the acceptable sequence of messages
  - Interactions between message sequences and state are often subtle and complex, therefore error prone

Modal Class Test – cont.

- Fault Model
  - Omited transition – message is rejected in allowed state
  - Incorrect action – bad response is chosen, although the state and method used were correctly accepted
  - Resulting state is not allowed – method develops a bad state in a transition
  - Corrupt state is developed
  - Sneak path allows to accept the message although it should be rejected

Modal Class Test – cont.

- Strategy
  1. Create FREE state model
  2. Generate transition tree
  3. Equip the transition tree with full explication of conditional transition cases
  4. Tabulate events and actions along each path
  5. Develop test data for paths – use Invariant Boundaries for message events and actions
  6. Develop Conformance tests until all tests pass
  7. Develop Sneak path test suite until all tests pass

Modal Class Test – cont.

- Entry Criteria
  - Alpha – omega cycle
  - If critical method/scope functions exist, they should be tested before running the modal because lead-node cannot be reached until all the predecessor tests have passed
- Exit Criteria
  - Achieve branch coverage on each method in the Class Under Test
  - A full modal class suite provides N+ coverage
  - A higher level of confidence can be obtained by developing a class flow graph to identify and any uncovered alpha-omega paths

Modal Class Test – cont.

- Consequences
  - A testable behavior model is available or can be developed
  - The Class Under Test is state-observable

Literature and Links

- Robert V. Binder: „Testowanie systemów obiektowych. Modele wzorce i narzędzia.” WNT 2003
- Jtest - Automatic test generator (commercial) http://www.parasoft.com/jsp/products/home.jsp?product=
- Jcontract – Design by Contract in Java (commercial) http://www.parasoft.com/jsp/products/home.jsp?product=
- Icontract – Design by Contract in Java http://www.reliable-systems.com/tools/iContract/iContract htm
- Assertion Definition Language http://adl.opengroup.org/
- JML Specs http://www.jmlspecs.org/

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?