Software Testing

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

Spring 2022

Lecture 3

Outline

Recap

- Input domain modeling: 5 major steps
  1. Identify testable functions
  2. Identify parameters
  3. Build the IDM (input domain model)
  4. Apply a test criterion
  5. Provide test inputs
- Building the IDM
  1. Identify characteristics
  2. Create a partition for each characteristics
  3. Two approaches:
     - Interface-based
     - Functionality-based

Step 3: building the IDM

- Most creative step of input-space partitioning
  - Iterative
  - Characteristics suggest blocks, blocks suggest characteristics
- Strategies
  - Valid, invalid, special values
  - Domain boundaries
  - Values that represent normal use
  - Sub-partitions of blocks
  - Balance number of blocks per characteristics
- Don't forget to check for completeness and disjointness

Example (interface-based approach)

```
public static Triangle triang (int Side1, int Side2, int Side3)
  // Side1, Side2, and Side3 represent the lengths of the sides of
      a triangle
  // Returns the appropriate enum value
```

- triang has one testable function and 3 integer inputs.
- Finding characteristics (first option):

Characteristics	b_1	b_2	b_3
relation of side 1 to 0	> 0	$= 0$	< 0
relation of side 2 to 0	> 0	$= 0$	< 0
relation of side 3 to 0	> 0	$= 0$	< 0

- Invalid triangles. Problem?

Example (interface-based approach), cont'd

```
public static Triangle triang (int Side1, int Side2, int Side3)
```

- Finding characteristics (second option)

Characteristics	b_1	b_2	b_3	b_4
length of side 1	> 1	$= 1$	$= 0$	< 0
length of side 2	> 1	$= 1$	$= 0$	< 0
length of side 3	> 1	$= 1$	$= 0$	< 0

- Refinement of the previous characteristics
  - Completeness property holds because parameters are of integer type
  - From 3^3 to $4^3 = 64$ possible combinations
- Next step: picking test values

Characteristics	b_1	b_2	b_3	b_4
Side1	5	1	0	-5

- Better choices?

Example (interface-based approach), cont'd

- Given the characteristics from before

Characteristics	b_1	b_2	b_3	b_4
length of side 1	> 1	$= 1$	$= 0$	< 0
length of side 2	> 1	$= 1$	$= 0$	< 0
length of side 3	> 1	$= 1$	$= 0$	< 0

- Compare

Characteristics	b_1	b_2	b_3	b_4
Side1	5	1	0	-5

vs.

Characteristics	b_1	b_2	b_3	b_4
Side1	2	1	0	-1

- Boundary values are better

Example (functionality-based approach)

```
public static Triangle triang (int Side1, int Side2, int Side3)
```

- Finding characteristics: geometric characterization

Characteristics	b_1	b_2	b_3	b_4
geometric classification	scalene	isosceles	equilateral	invalid

- Problem?
- Equilateral implies isosceles

Example (functionality-based approach), cont'd

- Fix (by refining b_2)

Characteristics	b_1	b_2	b_3	b_4
geometric classification	scalene	isoceles & not equilateral	equilateral	invalid

- Next step: picking values

Characteristics	b_1	b_2	b_3	b_4
triangle	(4,5,6)	(3,3,4)	(3,3,3)	(3,4,8)

Example (functionality-based approach), cont'd

- Alternatively, one can break the geometric characterization down into four separate characteristics:

Characteristics	b_1	b_2
scalene	true	false
isosceles	true	false
equilateral	true	false
valid	true	false

- Additional constraints needed
  - (equilateral = true) $\rightarrow$ (isosceles = true)
  - (valid = false) $\rightarrow$ (scalene = iscosceles = equilateral = false)

Using more than one IDM

- If a program has many parameters, it might make sense to build several small(er) IDMs
  - Could reduce number of invalid values
  - Allows for tests of different granularity
- Overlapping of IDMs no problem
  - A variable can occur in more than one IDM

Outline

Combination strategies

- Step 4, input domain modeling
- Systematic selection of test values and test sets: requires criteria
- We will discuss 6 sensible criteria.

ACoC: all combinations

Definition

In the All Combinations Coverage (ACoC) criterion all combinations of all blocks from all characteristics must be used.

Number of tests:

Q	no. of characteristics $(q_1, q_2, \ldots)$
Bl_i	no. of blocks of characteristic q_i

For the ACoC criteria ("all combinations of blocks"), the number of tests is the product of the number of blocks in each characteristic:

$$\Pi_{i=1}^{Q} \mathrm{Bl}_i$$

Example (ACoC)

- Consider the characteristics for the `triang()` function from before

Characteristics	b_1	b_2	b_3	b_4
length of side 1	> 1	$= 1$	$= 0$	< 0
length of side 2	> 1	$= 1$	$= 0$	< 0
length of side 3	> 1	$= 1$	$= 0$	< 0

- For convenience, we abbreviate:

Characteristics	b_1	b_2	b_3	b_4
A	A_1	A_2	A_3	A_4
B	B_1	B_2	B_3	B_4
C	C_1	C_2	C_3	C_4

- Number of tests: $4 \cdot 4 \cdot 4 = 64$
  - How many valid triangles?
  - All sides greater 0: only 8 cases

ECC: each choice coverage

Definition

In the Each Choice Coverage (ECC) criterion one value from each block for each characteristics must be used.

What is the number of tests for ECC?

q_i	characteristic
Q	no. of all characteristics
Bl_i	no. of blocks of characteristic q_i

ECC: number of tests

Assume one characteristic per parameter. For the ECC criteria ("each block must be used once"), the number of tests is the number of blocks in the largest characteristic:

$$\max_{1 \leq i \leq Q} \mathrm{Bl}_i$$

- Example: function triang() ($Q = 3$, $Bl_1 = Bl_2 = Bl_3 = 4$, thus 4 tests)
- The following combination of blocks defines possible test requirements

$$\begin{array}{ccc} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \\ A_4 & B_4 & C_4 \end{array}$$

- One possible test set

$$\{(2,2,2),(1,1,1),(0,0,0),(-1,-1,-1)\}$$

Pair-wise coverage criterion (PWC)

- ECC is quite weak: it does not require any kind of combination.
- The *Pair-wise Coverage Criterion* is stronger:

Definition

In the Pair-Wise Coverage (PWC) criterion, a value from each block for each characteristic must be combined with a value from every block for each other characteristic.

PWC: number of tests

Assume one characteristic per parameter. For the PWC criteria ("each block must be combined with every other block"), the number of tests with > 2 parameters is *at least* the product of the two largest characteristics:

$$\max_{1 \leq i \leq Q} \mathrm{Bl}_i \cdot \max_{1 \leq j \leq Q, j \neq i} B_j$$

- Example: function triang() ($Q = 3$, $Bl_1 = Bl_2 = Bl_3 = 4$, 16 tests)
- Test requirements: the following blocks must be covered

$$\begin{array}{lllllll}
(A_1, B_1) & (A_1, B_2) & (A_1, B_3) & (A_1, B_4) & (A_1, C_1) & .. & (A_1, C_4) \\
(A_2, B_1) & (A_2, B_2) & (A_2, B_3) & (A_2, B_4) & (A_2, C_1) & .. & (A_2, C_4) \\
\ldots & & & & & & \\
(A_4, B_1) & (A_4, B_2) & (A_4, B_3) & (A_4, B_4) & (A_4, C_1) & .. & (A_4, C_4) \\
(B_1, C_1) & (B_1, C_2) & (B_1, C_3) & (B_1, C_4) & & & \\
\ldots & & & & & & \\
(B_4, C_1) & (B_4, C_2) & (B_4, C_3) & (B_4, C_4) & & &
\end{array}$$

Example (PWC), (cont'd)

- The pairwise combination of pairs of blocks is well-defined:

$$
\begin{array}{lllllll}
(A_1, B_1) & (A_1, B_2) & (A_1, B_3) & (A_1, B_4) & (A_1, C_1) & .. & (A_1, C_4) \\
(A_2, B_1) & (A_2, B_2) & (A_2, B_3) & (A_2, B_4) & (A_2, C_1) & .. & (A_2, C_4) \\
... & & & & & & \\
(A_4, B_1) & (A_4, B_2) & (A_4, B_3) & (A_4, B_4) & (A_4, C_1) & .. & (A_4, C_4) \\
(B_1, C_1) & (B_1, C_2) & (B_1, C_3) & (B_1, C_4) & & & \\
(B_2, C_1) & (B_2, C_2) & (B_2, C_3) & (B_2, C_4) & & & \\
(B_3, C_1) & (B_3, C_2) & (B_3, C_3) & (B_3, C_4) & & & \\
(B_4, C_1) & (B_4, C_2) & (B_4, C_3) & (B_4, C_4) & & &
\end{array}
$$

- A test set derived from the following combination of pairs meets the PWC criterion:

$$
\begin{array}{llll}
A_1, B_1, C_1 & A_1, B_2, C_2 & A_1, B_3, C_3 & A_1, B_4, C_4 \\
A_2, B_1, C_2 & A_2, B_2, C_3 & A_2, B_3, C_4 & A_2, B_4, C_1 \\
A_3, B_1, C_3 & A_3, B_2, C_4 & A_3, B_3, C_1 & A_3, B_4, C_3 \\
A_4, B_1, C_2 & A_4, B_3, C_3 & A_4, B_3, C_4 & A_4, B_4, C_1
\end{array}
$$

T-wise coverage criterion (TWC)

- PWC could be generalized from 2 combinations to t.

Definition

In the t-Wise Coverage (TWC) criterion, a value from each block for each group of t characteristics must be combined.

- Number of tests
  - At least the product of the t largest characteristics

$$\geq \max_{1 \leq i_1 \leq Q} Bl_{i_1} \cdots \max_{1 \leq i_t \leq Q} Bl_{i_t}$$

  - If all characteristics have the same size B: B^t
  - If $t = Q$, then TWC=ACoC

Base-choice criterion (BCC)

- The *Base Choice Coverage Criterion* allows bringing in domain knowledge

Definition

In the Base Choice Coverage (BCCC) criterion, a basic choice block is chosen for each characteristic and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant (and using each non-base choice in each other characteristic).

- Example: function triang(): define as base A_1, B_1, C_1

BCC: number of tests

For the BCC criteria ("using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant "), the number of tests is one base test plus one test for each other block

$$1 + \Sigma_{i=1}^{Q}(\mathrm{Bl}_i - 1)$$

- Example: function triang() ($Q = 3$, $Bl_1 = Bl_2 = Bl_3 = 4$, thus $1 + 3 + 3 + 3$ tests

Example (BCC)

- Example: function triang() ($Q = 3$, $Bl_1 = Bl_2 = Bl_3 = 4$: thus $1 + 3 + 3 + 3$ tests
- Define as base A_1, B_1, C_1
- Test requirements for the subsequent tests

$$\begin{array}{ccc} A_1, B_1, C_2 & A_1, B_1, C_3 & A_1, B_1, C_4 \\ A_1, B_2, C_1 & A_1, B_3, C_1 & A_1, B_4, C_1 \\ A_2, B_1, C_1 & A_3, B_1, C_1 & A_4, B_1, C_1 \end{array}$$

- Test set {(2,2,2), (2,2,1), (2,2,0), (2,2,-1), (2,1,2), (2,0,2), (2,-1,2), (1,2,2), (0,2,2), (-1,2,2)}

In-class exercise

```
public static Set intersection (Set s1, Set s2)
/**
 * @param  s1, s2 : sets to compute intersection of
 * @return a (non null) set equal to the intersection of s1 and s2
 * @throws NullPointerException if s1 or s2 is null
 */
```

Multiple base choice (MBCC)

- The multiple base choice criterion can be applied if there is more than one logical base choice:

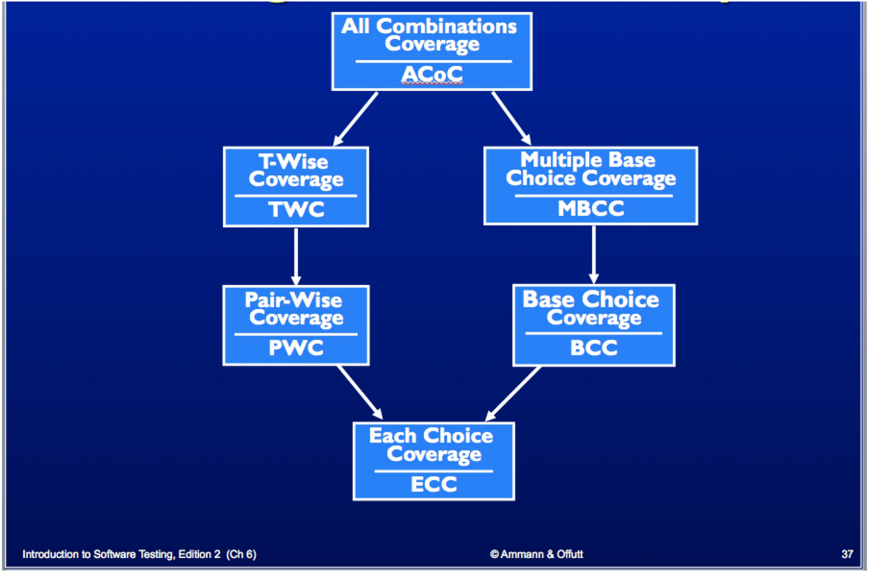
Definition

In the Multiple Base Choice Coverage (MBCCC) criterion, one or more basic choice blocks are chosen for each characteristic and base tests are formed by using the base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant (and using each non-base choice in each other characteristic).

- Let M be the number of base tests with m_i base choices for each characteristic. The number of tests is then

$$M + \Sigma_{i=1}^{Q}(M \cdot (\mathrm{Bl}_i - m_i))$$

Coverage criteria subsumption



Outline

Constraints (on characteristics)

- Certain combinations of blocks are infeasible
  - Ex. (triang): scalene and < 0 cannot hold at the same time
- Infeasible combinations are represented as constraints
  - Relations between blocks from different characteristics
- Kinds of constraints
  - Positive: "can only be combined with"
  - Negative: "cannot be combined with
- Handling of constraints
  - ACoC, PWC, TWC: drop infeasible pairs
  - BCC, MBCC: use another non-base choice

Example (handling constraints)

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//    else return true if element is in the list, false otherwise
```

Two characteristics c_1, c_2:

- c_1: "how long `List` is and whether it is sorted" (length/structure)
  c_2: "how often `element` matches"

Charact.	b_1	b_2	b_3	b_4
length /structure	A1: one elt	A2: >1 elt, unsorted	A3: >1 elt, strictly sorted	A4: >1, all equal
match	B1: elt not found	B2: elt found once	B3: elt found more than one	

- Invalid combinations: (A1,B3), (A4,B2)

Summary: input space partioning (ISP)

- Based on the input space only (not the implementation)
- Applicable at all levels of testing (unit, class, integration, system)
- Manual application possible

Outline

Case study: IDM

- Goal: JUnit tests for the iterator interface
- Input domain modeling based on the Javadoc API
  - We assume that the API defines all testable functions (step 1)

The API (before 1.8)

http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

Interface Iterator<E>

Type Parameters:

`E` - the type of elements returned by this iterator

All Known Subinterfaces:

ListIterator<E>, XMLEventReader

All Known Implementing Classes:

BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

```
public interface Iterator<E>
```

An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `ListIterator`, `Iterable`

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Step 2: identify parameters

- Functions, functional units, parameters, return types and return values, exceptional behavior

Method	Params	RetType	RetVal	Exc

Identify parameters (cont'd)

- Result:

Method	Params	RetType	RetVal	Exc	Characteristics
hasNext	state	boolean	true, false		
next	state	E	"E-val", null		
remove	state				

- Next: develop characteristics
  - Cover return types/values and exceptions

Step 3: determine characteristics

Method	Params	RetType	RetVal	Exc	Characteristic
hasNext	state	boolean	true, false	-	more values
next	state	E	E-val, null		returns non-null object
				NoSuch Element	
remove	state			Unsuppor-tedOp	remove supported
				Illegal State	remove constraint holds

Determine characteristics (cont'd)

- Associate each method with relevant characteristics

Characteristics	hasNext	next	remove
more values	x	x	x
returns non-null object		x	x
remove() supported			x
remove() constraint satisfied			x

- Design partitions

Determine characteristics (cont'd)

- Result:

Characteristics	hasNext	next	remove	partition
more values	x	x	x	{true,false}
returns non-null object		x	x	{true,false}
remove() supported			x	{true,false}
remove() constraint satisfied			x	{true,false}

- Model done!
- Next: define test requirements

Step 4: define test requirements

We introduce IDs for characteristics:

more values	C1
returns non-null object	C2
remove() supported	C3
remove() constraint satisfied	C4

- Task 1: pick coverage criterion
  - We pick BCC
- Task 2: choose base cases (if needed)
  - We pick the happy path (all true)
- Task 3: test requirements

Method	Characteristics	Test Requirements
hasNext	C1	
next	C1,C2	
remove	C1,C2,C3,C4	

Define test requirements (cont'd)

- Result task 3:

Method	Characteristics	Test Requirements
hasNext	C1	{T, F}
next	C1,C2	{TT, FT, TF }
remove	C1,C2,C3,C4	{TTTT, FTTT, TFTT, TTFT, TTTF}

- Task 4: infeasible TRs

Method	Characteristics	TRs	Infeasible TRs
hasNext	C1	{T, F}	
next	C1,C2	{TT, FT, TF }	
remove	C1,C2,C3,C4	{TTTT, FTTT, TFTT, TTFT, TTTF}	

Define test requirements (cont'd)

- Result (infeasible TRs):

Method	Characteristics	Test Requirements	Infeasible TRs
hasNext	C1	{T,F}	–
next	C1,C2	{TT, FT, TF}	FT
remove	C1,C2,C3,C4	{TTTT, FTTT, TFTT, TTFT, TTTF}	FTTT

- Task 5: revise infeasible test requirements

Method	Characteristics	Revised Test Requirements
hasNext	C1	{T,F}
next	C1,C2	{TT, FF, TF }
remove	C1,C2,C3,C4	{TTTT, FFTT, TFTT, TTFT, TTTF}

Test automation

- Since `iterator` is an interface, we need an implementation
  - We'll use `ArrayList`
- From the IDM, one can derive 10 test cases.
- The test class `IteratorTest` contains those test cases, plus the test fixture
  - Annotation @BeforeEach https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations
- One might additionally include tests of preconditions that do not manifest themselves in parameter values.

Test functions

```
// C1 T
@Test public void testHasNext_BaseCase()
// C1 F
@Test public void testHasNext_C1()()

// C1 T, C2 T
@Test public void testNext_BaseCase()
// C1 T, C2 F
@Test public void testNext_C2()
// C1 F, C2 F
@Test public void testNext_C1()

// C1 T, C2 T, C3 T, C4 T
@Test public void testRemove_BaseCase()
// C1 F, C2 F, C3 T, C4 T
@Test public void testRemove_C1()
// C1 T, C2 F, C3 T, C4 T
@Test  public void testRemove_C2() // ! CME. Additional test?
// C1 T, C2 T, C3 F, C4 T
@Test public void testRemove_C3()
 // C1 T, C2 T, C3 T, C4 F
@Test public void testRemove_C4()
```

Test class and test fixture

```
import java.util.*;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertFalse;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ItTest {
    private List<String> list;
    private Iterator<String> it;
    @BeforeEach
    public void setUp() {
        list = new ArrayList<String>();
        list.add("iphone"); list.add("galaxy"); list.add("nokia");
        it = list.iterator();
    }
    @Test public void testHasNext_BaseCase() {
        assertTrue(it.hasNext());
    }
    // C1 F
    @Test public void testHasNext_C1() {
        it.next(); it.next(); it.next();
        assertFalse(it.hasNext());
    }}}
```

Check list: IDM model

Steps:

1. Identify the following and document them in Table A: functional units, parameters, return types and values, exceptional behavior.
2. Develop characteristics for return types and exceptional behavior and document them in Table A. Include a column “Covered by.”
3. Create Table B that associates methods and relevant characteristics.
4. Add a column to Table B that contains a partition for each characteristic.

Check list: Test requirements

Steps:

1. Choose coverage criterion.
2. For BCC or MBCC: choose base cases and document them in Table B.
3. For each (method, characteristics) from Table B design complete test requirements and document them in Table C.
4. Identify infeasible test requirements and document them in Table C.
5. If needed, revise test requirements and document them in Table C.

References

- AO, Ch. 6.2-6.4