

# ***Advanced Software Engineering***

## **9. Sequence Sheets**

**Fall Semester 2024**



Marcus Kessel [marcus.kessel@uni-mannheim.de](mailto:marcus.kessel@uni-mannheim.de), Colin Atkinson [colin.atkinson@uni-mannheim.de](mailto:colin.atkinson@uni-mannheim.de)



# Describing Tests

- The final step is to create actual tests which fulfill the specified test requirements
  - can be conveniently described using [sequence sheets](#)
- Sequence sheets provide a complete and easy-to-understand description of tests
  - are directly executable
  - direct verification technique
  - notation independent from programming language
- In addition, sequence sheets
  - speed up the identification of errors
  - provide a simple and direct record of test execution traces
    - regression testing
- Software system is not acceptable until all sequence sheets passed



# Motivation for Sequence Sheets

- Testing is an integral part of software engineering
- Tests define the expected behaviour of system
  - Important part of documentation
- There are numerous stakeholders -
  - developers, domain-experts, sales and marketing personnel, customers, ...
- Current test definition technologies are not suitable for non-developers
- Abstracts away programming language details → ideal for testing education 😊

➡ understandable and usable test definition approach

# Look-and-Feel of Test Definitions (JUnit 5)

```
public class MinTestJUnit5 {
    private List<String> list; // Test fixture

    @BeforeEach // Set up - Called before every test method.
    public void setUp() {
        list = new ArrayList<>();
    }

    @AfterEach // Tear down - Called after every test method.
    public void tearDown() {
        list = null; // redundant in this example!
    }

    // old JUnit 4 style for testing exceptions
    @Test
    public void testForNullList() {
        list = null;
        try {
            Min.min(list);
        } catch (NullPointerException e) {
            return;
        }
        fail("NullPointerException expected");
    }

    // assertThrows is now the preferred mechanism for testing exceptions
    // note the lambda expression in the assertion
    @Test
    public void testForNullElement() {
        list.add(null);
        list.add("cat");
        assertThrows(NullPointerException.class, () -> Min.min(list));
    }
}
```

```
@Test
public void testForSoloNullElement() {
    list.add(null);
    assertThrows(NullPointerException.class, () -> Min.min(list));
}

@Test
public void testMutuallyIncomparable() {
    List list = new ArrayList();
    list.add("cat");
    list.add("dog");
    list.add(1);
    assertThrows(ClassCastException.class, () -> Min.min(list));
}

@Test
public void testEmptyList() {
    assertThrows(IllegalArgumentException.class, () -> Min.min(list));
}

@Test
public void testSingleElement() {
    list.add("cat");
    Object obj = Min.min(list);
    assertTrue(obj.equals("cat"), "Single Element List");
}

@Test
public void testDoubleElement() {
    list.add("dog");
    list.add("cat");
    Object obj = Min.min(list);
    assertTrue(obj.equals("cat"), "Double Element List");
}
}
```

*Class Under Test*

```
public class Min {
    /**
     * Returns the minimum element in a list
     *
     * @param list Comparable list of elements to search
     * @return the minimum element in the list
     * @throws NullPointerException if list is null or
     *                               if any list elements are null
     * @throws ClassCastException if list elements are not mutually
     *                             comparable
     * @throws IllegalArgumentException if list is empty
     */
    public static <T extends Comparable<? super T>> T min(List<? extends T> list) {
        if (list.size() == 0) {
            throw new IllegalArgumentException("Min.min");
        }

        Iterator<? extends T> itr = list.iterator();
        T result = itr.next();

        if (result == null) throw new NullPointerException("Min.min");

        while (itr.hasNext()) { // throws NPE, CCE as needed
            T comp = itr.next();
            if (comp.compareTo(result) < 0) {
                result = comp;
            }
        }
        return result;
    }
}
```

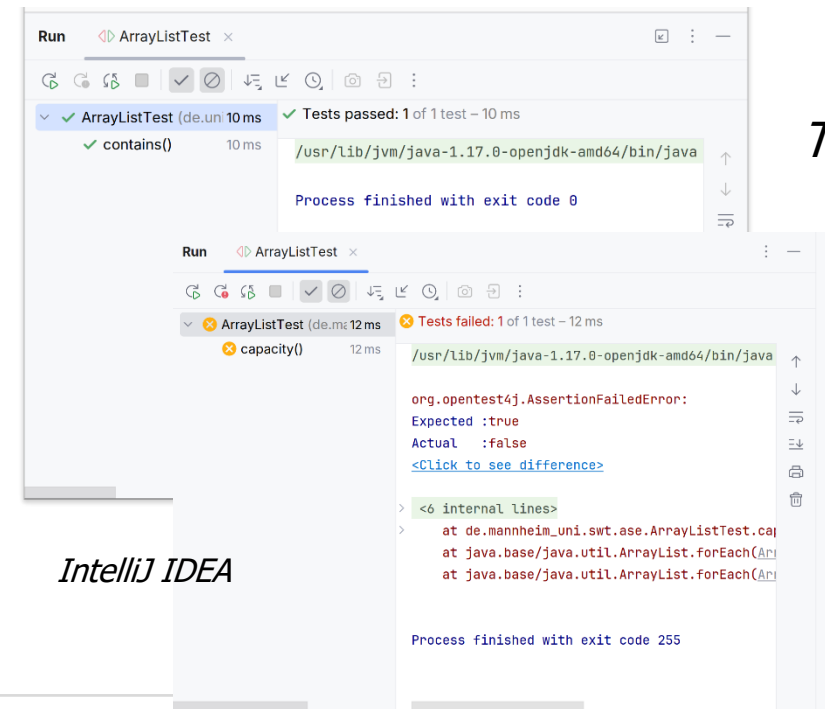


# Writing Executable Tests

- XUnit - main examples: {J | N | Q | ...}Unit
- In JUnit, tests are defined by writing annotated code
  - all features of programming language are supported
  - tests are typically structured into test methods/functions, part of a test class
  - test suites aggregate one or more test classes
- but not suitable for non-developers
- test results separated from test definition
- raised/thrown assertion exceptions for failing tests indicate comparison failures

```
public class ArrayListTest {  
  
    @Test  
    public void contains() {  
        List<Integer> fEmpty = new ArrayList<>();  
        List<Integer> fFull = new ArrayList<>();  
        fFull.add(1);  
        fFull.add(2);  
        fFull.add(3);  
  
        assertTrue(fFull.contains(1));  
        assertTrue(!fEmpty.contains(1));  
    }  
}
```

*Test Class*



*Test Results*

*IntelliJ IDEA*

# Test Fixtures in JUnit for Java

## Test Fixtures in JUnit 5

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (prefix values)
  - Reset values (postfix values)

→ consistency (i.e., controlled environment and state), isolation (i.e., test independence)

- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables
  - They should be initialized in a `@BeforeEach` method
  - Can be deallocated or reset in an `@AfterEach` method
- Others: `@BeforeAll`, `@AfterAll` ...

## Parameterized Tests

- e.g. `@ParameterizedTest` for parameterized tests (via junit params)

```
public class ParameterizedCalcTest {  
  
    @ParameterizedTest  
    @MethodSource("provideParameters") // alternative: @CsvSource(...)   
    public void testSum(int a, int b, int result) {  
        assertEquals(result, Calc.add(a, b), "Addition Test");  
    }  
  
    private static Stream<Arguments> provideParameters() {  
        return Stream.of(  
            Arguments.of(1, 1, 2),  
            Arguments.of(2, 3, 5)  
        );  
    }  
}
```

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests>



# Unit Testing in Python with *pytest*

- pytest documentation
  - <https://docs.pytest.org/en/latest/>
  - supports test fixtures (prefix and postfix values), parameterized tests (data-driven tests) etc.
- Other popular frameworks
  - unittest  
<https://docs.python.org/3/library/unittest.html>
  - doctest  
<https://docs.python.org/3/library/doctest.html#module-doctest>

```
import pytest

class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_raise(self):
        with pytest.raises(Exception):
            x = 1 / 0
```

*Test Class*

```
$ pytest -q test_class.py
```

```
===== test session starts =====
collecting ... collected 2 items

test_class.py::TestClass::test_one PASSED [ 50%]
test_class.py::TestClass::test_raise PASSED [100%]

===== 2 passed in 0.00s =====
```

*Test Results*



# Tabular Test Representation – FIT (Mugridge et al.)

- usable by domain experts and customers
- test developer bridges gap between tables and code
  - focused on acceptance and integration testing
- BUT
  - limited descriptive power
  - not semantically self-contained
- Need an easily understandable, yet executable, test description approach which combines input (i.e., stimulus) descriptions with output (i.e., response) descriptions

Payroll Fixtures WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	\$800
45	0	20	\$950
48	8	20	\$1360 <i>expected</i>
			\$1040 <i>actual</i>

```
public class WeeklyCompensation: ColumnFixture { Fixture
    public int StandardHours;
    public int HolidayHours;
    public Currency Wage;

    public Currency Pay() {
        WeeklyTimesheet timesheet = new WeeklyTimesheet(
            StandardHours, HolidayHours);
        return timesheet.CalculatePay(Wage);
    }
}
```

➡ Sequence Sheets

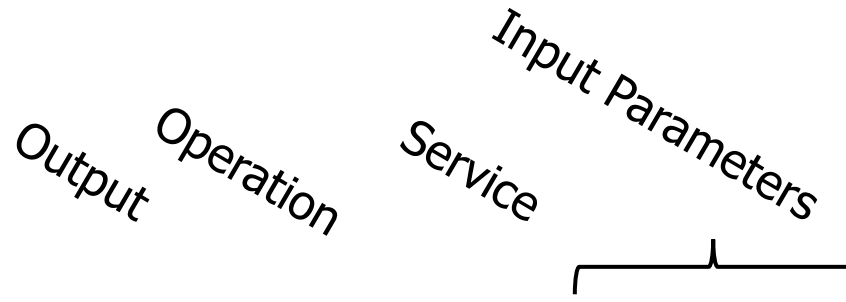


# Sequence Sheets

- FIT and xUnit represent extremes of the abstraction spectrum
  - Sequence Sheets (Kessel et al.): positioned in between
    - inspired by test sheets (Atkinson et al.)
    - descriptive power of xUnit
    - tabular definition of test data (FIT)
  - spreadsheet metaphor
  - automatic transformation into executable code
    - higher decoupling from actual programming language
    - limited set of language features
  - test results also displayed as sequence sheet
- ➡ provide a simple, concise, easy-to-understand yet **executable description** of what a **component under test** should do.

# Basic Sequence Sheet Layout

- Basic ingredients
  - output column
    - Observed response to operation invocation
  - operation column
    - Operation to be invoked, typically method invocations
  - service column
    - either class which is to be instantiated (can have input parameters like object-oriented constructors), or
    - instance (i.e., object) on which an operation is to be invoked
  - input parameter column(s)
    - zero or more **input parameter columns** for the input parameters of the operation



	A	B	C	D	...
1					
2					
3					
4					
5					
6					
7					

# Stimulus Sheet – Input

- each row represents separate **stimulus** to the system
- objects maintain state between invocations
- cells can be cross-referenced in normal spreadsheet style
- *create* – special command to create an instance from a class

Input Parameters  
Service  
Operation  
Output

	A	B	C	D
1		create	'Stack	
2		push	A1	23
3		push	A1	42
4	42	pop	A1	
5	D2	pop	A1	

```
public class Stack {  
    private ArrayList<Integer> list = new ArrayList();  
  
    public void push(int e) {  
        list.add(e);  
    }  
  
    public int pop() {  
        return list.remove(list.size() - 1);  
    }  
}
```

*Java implementation*

```
Stack {  
    Stack()  
    push(int)->void  
    pop()->int  
}
```

*Interface*

$m_1$	create	[Class]	→	[Stack]
$m_2$	push	[Stack,int]	→	[void]
$m_3$	push	[Stack,int]	→	[void]
$m_4$	pop	[Stack]	→	[int]
$m_5$	pop	[Stack]	→	[int]

*Sequence of  
Invocations (Types)*



# Actuation Sheet – Output

- In classic software testing of a single component under test (i.e., code unit), stimulus sheets typically define expected outputs in the output column (if none, cells are kept blank)
- The actuation sheet contains the actual outputs recorded at runtime, hence is produced by executing the stimulus sheet – i.e., combines stimulus sheet with test result records
- actual outputs are then compared to the expected outputs

	A	B	C	D
1		create	'Stack	
2		push	A1	23
3		push	A1	42
4	42	pop	A1	
5	D2	pop	A1	

Stimulus Sheet

*Expected outputs*



	A	B	C	D
1	<instance>	create	'Stack	
2		push	A1	23
3		push	A1	42
4	23	pop	A1	
5	D2	pop	A1	

Actuation Sheet

*Actual outputs*

# ArrayList Example

- Assume the following interface from Java's `java.util.ArrayList` implementation
- <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html>

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

*Expected outputs*

Stimulus Sheet

# Instance Creation

- Objects are created through initialization with the `create` command
  - similar to constructors in object-oriented languages
  - can have zero or more input parameter values

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	3
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

*initial capacity of the underlying array*

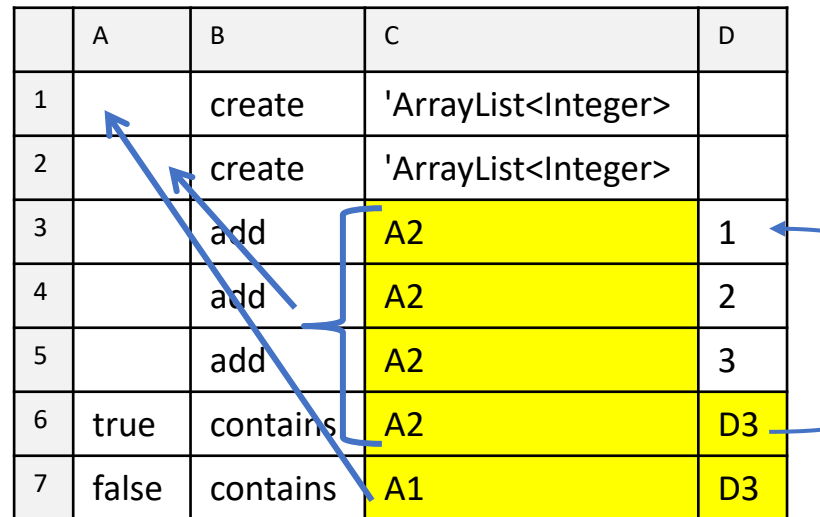
```
ArrayList {  
  ArrayList()  
  ArrayList(int)  
  ...  
}
```

*Interface*

Stimulus Sheet

# References, etc.

- Instances and values are referenceable using cell addresses
  - format: *ColumnRow*, e.g. *A1*



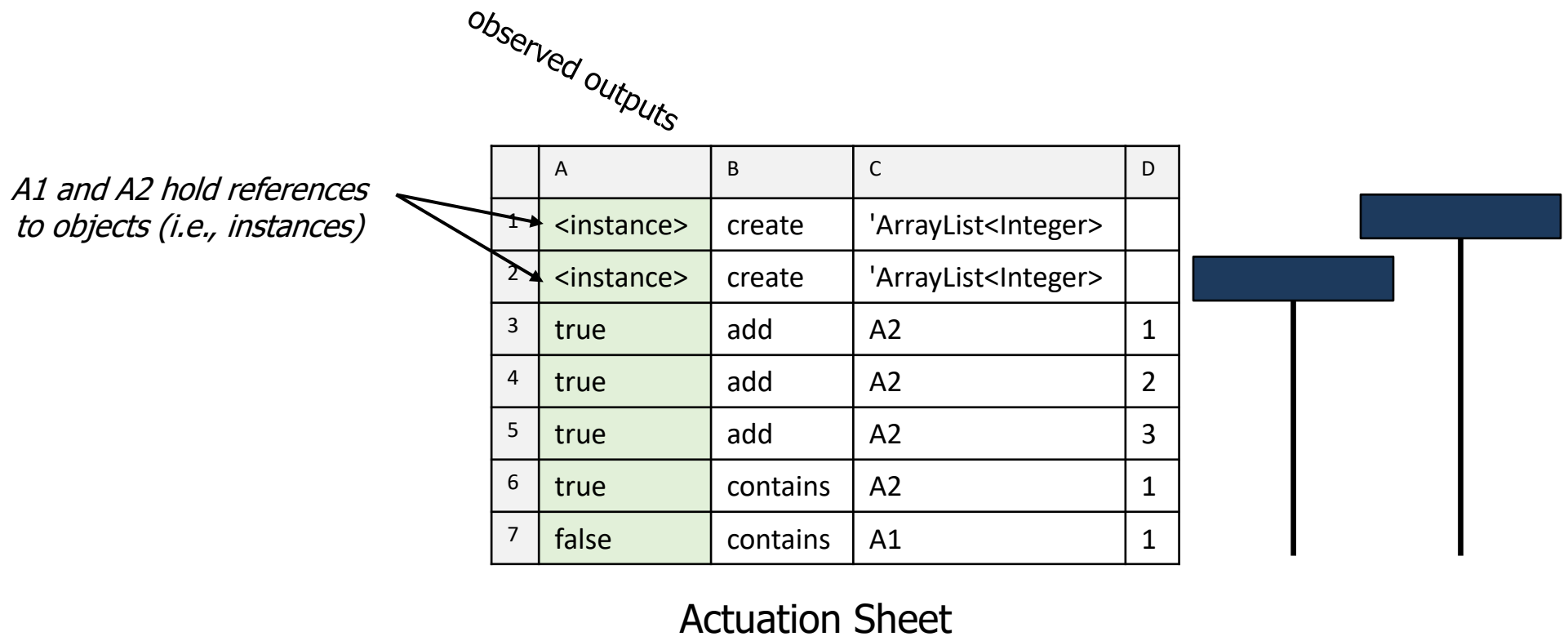
The diagram shows a table with 7 rows and 5 columns. The first row is a header with columns labeled A, B, C, and D. The subsequent rows contain data. Blue arrows indicate references: an arrow from cell B3 to cell A2, an arrow from cell B4 to cell A2, an arrow from cell B5 to cell A2, an arrow from cell B6 to cell A2, an arrow from cell B7 to cell A1, and a bracketed arrow from cell D3 to cell D6. The cells containing 'A2', 'A1', and 'D3' are highlighted in yellow.

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	D3
7	false	contains	A1	D3

Stimulus Sheet

# State

- The (intermediate) states of all involved components is stored in the output column of the actuation sheet (i.e., column A) that is created in response of executing the stimulus sheet





# Result Representation and Comparison

- output column contains a serialized representation of the output values (i.e., objects)

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

Stimulus Sheet



	A	B	C	D
1	<instance>	create	'ArrayList<Integer>	
2	<instance>	create	'ArrayList<Integer>	
3	true	add	A2	1
4	true	add	A2	2
5	true	add	A2	3
6	true	contains	A2	1
7	true	contains	A1	1

Actuation Sheet

*Compare expected outputs with actual, observed outputs*

*Let us assume the output observed is true!  
(which is wrong, of course)*

# Parameterized Sequence Sheets

- Parameterization of stimulus sheets
  - allow for clean and simple structures, and
  - avoids duplication – enables “reuse” of sequences

	A	B	C	D
1		create	'ArrayList<Integer>	
2		add	A1	1
3		add	A1	2
4		add	A1	3
5		add	A1	4
6		add	A1	5
7	3	get	A1	2

Stimulus Sheet

*Let's assume we want to test different elements  
(i.e., index)*

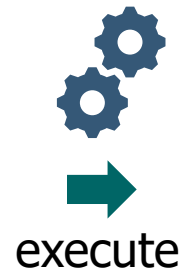
# Parameterized Sequence Sheets

- Sheets can be treated like classic methods, since they have well-defined, invocable signatures

	A	B	C	D
1		create	'ArrayList<Integer>	
2		add	A1	1
3		add	A1	2
4		add	A1	3
5		add	A1	4
6		add	A1	5
7	?p2	get	A1	?p1

Parameterized Stimulus Sheet

**test(p1: int, p2: int)**



**test(0, 1)**

**test(4, 5)**

## Actuation Sheets

	A	B	C	D
1	<instance>	create	'ArrayList<Integer>	
2	true	add	A1	1
3	true	add	A1	2
4	true	add	A1	3
5	true	add	A1	4
6	true	add	A1	5
7	1	get	A1	0

	A	B	C	D
1	<instance>	create	'ArrayList<Integer>	
2	true	add	A1	1
3	true	add	A1	2
4	true	add	A1	3
5	true	add	A1	4
6	true	add	A1	5
7	5	get	A1	4



# Parameterized Sequence Sheets

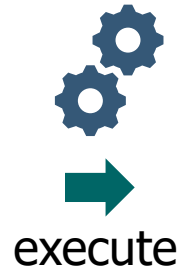
- Swapping classes under test
  - assume same interface

	A	B	C	D
1		create	?c	
2		add	A1	1
3		add	A1	2
4		add	A1	3
5		add	A1	4
6		add	A1	5
7	3	get	A1	2

Parameterized Stimulus Sheet

**test(c: Class)**

**test('ArrayList<Integer>')**



**test('LinkedList<Integer>')**

Actuation Sheets

	A	B	C	D
1	<instance>	create	'ArrayList<Integer>	
2	true	add	A1	1
3	true	add	A1	2
4	true	add	A1	3
5	true	add	A1	4
6	true	add	A1	5
7	3	get	A1	2

	A	B	C	D
1	<instance>	create	'LinkedList<Integer>	
2	true	add	A1	1
3	true	add	A1	2
4	true	add	A1	3
5	true	add	A1	4
6	true	add	A1	5
7	5	get	A1	4



# Multiple Invocations of Stimulus Sheets

- multiple invocations of a stimulus sheet lead to multiple actuation sheets (1:N mapping)
- useful for checking determinism, regression testing etc. (i.e., compare output columns of actuation sheets)

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

Stimulus Sheet

**test()**



execute

**test()**

**test()**

**test()**

...

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

Actuation Sheets



# JUnit List Example – List Interface (Excerpt)

- From Java JDK (java.util.List)
- See Java Documentation

```
public interface List<E> extends Collection<E> {  
  
    E get(int index);  
    E set(int index, E element);  
    boolean add(E element);  
    void add(int index, E element);  
    E remove(int index);  
    boolean addAll(int index, Collection<? extends E> c);  
  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
  
    List<E> subList(int from, int to);  
}
```



# Source Code of *ListTest* Test Class – Test Fixture

- assume JUnit 5

```
public class ListTest {  
    protected List<Integer> fEmpty;  
    protected List<Integer> fFull;  
  
    @BeforeEach  
    public void setUp() {  
        fEmpty = new ArrayList<>();  
        fFull = new ArrayList<>();  
        fFull.add(1);  
        fFull.add(2);  
        fFull.add(3);  
    }  
}
```



# Source Code of *ListTest* Test Class – Test Methods

```
@Test
public void testCopy() {
    List<Integer> copy = new ArrayList<>(fFull.size());
    copy.addAll(fFull);
    assertTrue(copy.size() == fFull.size());
    assertTrue(copy.contains(1));
}
```

```
@Test
public void contains() {
    assertTrue(fFull.contains(1));
    assertTrue(!fEmpty.contains(1));
}
```

```
@ParameterizedTest
@CsvSource({"0,1", "1,2", "2,3"})
public void elementAt(int index, int element) {
    int i = fFull.get(index);
    assertTrue(i == element);
}
```

```
@Test
public void elementAtFail() {
    assertThrows(IndexOutOfBoundsException.class, () ->
        fFull.get(fFull.size()));
}
```

```
@Test
public void removeAll() {
    fFull.removeAll(fFull);
    fEmpty.removeAll(fEmpty);
    assertTrue(fFull.isEmpty());
    assertTrue(fEmpty.isEmpty());
}
```

```
@Test
public void removeElement() {
    fFull.remove(Integer.valueOf(3));
    assertTrue(!fFull.contains(3));
}
```





# Stimulus Sheet for *contains()* Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}

@Test
public void contains() {
    assertTrue(fFull.contains(1));
    assertTrue(!fEmpty.contains(1));
}
```

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

# Stimulus Sheet for *contains()* Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}

@Test
public void contains() {
    assertTrue(fFull.contains(1));
    assertTrue(!fEmpty.contains(1));
}
```

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	true	contains	A2	1
7	false	contains	A1	1

*unequal, but why?*

# Stimulus Sheet for *removeAll()* Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}

@Test
public void removeAll() {
    fFull.removeAll(fFull);
    fEmpty.removeAll(fEmpty);
    assertTrue(fFull.isEmpty());
    assertTrue(fEmpty.isEmpty());
}
```

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6		removeAll	A2	
7		removeAll	A1	
8	true	isEmpty		
9	true	isEmpty		



# Stimulus Sheet for *testCopy()* Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}

@Test
public void testCopy() {
    List<Integer> copy = new ArrayList<>(fFull.size());
    copy.addAll(fFull);
    assertTrue(copy.size() == fFull.size());
    assertTrue(copy.contains(1));
}
```

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6		size	A2	
7		create	'ArrayList<Integer>	A6
8		addAll	A7	A2
9		size	A7	
10	A9	size	A2	
11	true	contains	A7	1

*Note: first row may be removed in practice, since it is irrelevant ...*

# Stimulus Sheet for *removeElement()* Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}

@Test
public void removeElement() {
    fFull.remove(Integer.valueOf(3));
    assertTrue(!fFull.contains(3));
}
```

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6		remove	A2	D5
7	false	contains	A2	

*Note: first row may be removed in practice, since it is irrelevant ...*

# Parameterized Stimulus Sheet for *elementAt()* Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}
```

*Note: JUnit 5 via @ParameterizedTest*

```
@ParameterizedTest
@CsvSource({"0,1", "1,2", "2,3"})
public void elementAt(int index, int element) {
    int i = fFull.get(index);
    assertTrue(i == element);
}
```

Parameterized Sheet Example

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6	?p2	get	A2	?p1

**test(0, 1)**

**test(1, 2), test(2, 3)**

*Note: first row may be removed in practice, since it is irrelevant ...*

# Stimulus Sheet for elementAtFail() Test

```
@BeforeEach
public void setUp() {
    fEmpty = new ArrayList<>();
    fFull = new ArrayList<>();
    fFull.add(1);
    fFull.add(2);
    fFull.add(3);
}

@Test
public void elementAtFail() {
    assertThrows(IndexOutOfBoundsException.class, () ->
        fFull.get(fFull.size()));
}
```

	A	B	C	D
1		create	'ArrayList<Integer>	
2		create	'ArrayList<Integer>	
3		add	A2	1
4		add	A2	2
5		add	A2	3
6		size	A2	
7	<exception>	get	A2	D6

actuation sheet stores (description of) exception thrown/raised

Failure handling in Junit 5 → assertThrows + Lambda expression

Note: first row may be removed in practice, since it is irrelevant ...

# Testability

- the purpose of a test is to check whether the system behaves in the way expected
  - since, by definition, changer operations affect the state of the system, the ability to fully check their correct behaviour depends on the ability to view the state of the system after their execution
  - but this might not always be possible
- ideally there should be operations to directly return or validate the value of all externally visible properties and states (inspector operations) – e.g. “getter” methods in Java
- if this is not the case, determining whether operations have had the correct internal effect on the system is much harder
  - correct internal changes can only be revealed by the behaviour of subsequently executed changer operations
- test sequences need to include appropriate “checking” invocations in addition to fulfilling test requirements





# Creating Sequence Sheets

1. identify sequences of operation invocations that form test paths covering the graph-based test requirements
  - not necessary to combine them all into one sequence sheet
  - direct touring is preferable, but sidetrips are often necessary
2. pick input values for the operations that cover as many logic-based and input based criteria as possible
  - you may also consider mutations of the input space of operations (cf. syntax-based)
3. if necessary, add more operation invocations to set up the conditions for more logic-based and input-based criteria to be covered
  - e.g. to move the system into a required precondition state
4. add additional inspector operations as appropriate to check whether an operation has had the expected effect
  - sometimes it is only possible to check the cumulative effect of a number of operations together



# Stack Example – Test Requirements

- From Graph Coverage Criteria

- [2,3,4], [2,3,5], [2,3,8], [2,3,9], [4,3,4], [4,3,5], [4,3,8], [4,3,9], [5,6,7], [7,3,4], [7,3,5], [7,3,8], [7,3,9], [8,3,4], [8,3,5], [8,3,8], [8,3,9], [9,1,2], [1,10]

- From Logic Coverage Criteria

push(): elems = max – 1	LC1 = True LC2 = False
pop(): elems = 1	LC3 = True LC4 = False

- From Input Partitioning Criteria

<b>push()</b>	IP1: (PriorState is partiallyOccupied, not null) IP2: (PriorState is partiallyOccupied, null) IP3: (PriorState is empty, not null)
<b>pop()</b>	IP4: (PriorState is partiallyOccupied) IP5: (PriorState is full)

*Note: We do syntax coverage in the next lecture ...*



# Stack Example – Sequence Sheet

- We assume that *max* is “large” (i.e., at least greater than 3)

	A	B	C	D		Transition/state	Logic	Data	elems
1		create	`Stack						
2	true	isEmpty	A1			1			0
3		push	A1	new Object		2	LC2	IP3	1
4	false	isEmpty	A1			3			
5	D3	pop	A1			9	LC3	IP4	0
6	true	isEmpty	A1			1			
7		push	A1	new Object		2	LC2	IP3	1
8	false	isEmpty	A1			3			
9	false	isFull	A1			3			
10		push	A1	new Object		4	LC2	IP1	2
11	false	isFull	A1			3			
12	D10	pop	A1			8	LC4	IP4	1
13	false	isEmpty	A1			3			
14		push	A1	null		4	LC2	IP2	1
15	false	isFull	A1			3			
16		push	A1	new Object		4	LC2	IP1	2
17	false	isFull	A1			3			
18	D16	pop	A1			8			1
19	false	isEmpty	A1			3			
20	D7	pop	A1			9	LC3	IP4	0
21	true	isEmpty	A1			1			
22		shutdown	A1			10			

*Note: new Object is used for simplification (i.e., push some element ...)*

## Executed Path –

[1, 2, 3, 9, 1, 2, 3, 4, 3, 8, 3, 4, 3, 4, 3, 8, 3, 9, 1, 10]

## Covered Subpaths -

Direct = [2,3,4], [2,3,9], [4,3,8], [4,3,4], [4,3,8], [8,3,4], [8,3,9], [9,1,2], [1,10]

## Not-covered Subpaths -

[2,3,5], [4,3,5], [5,6,7], [7,3,4], [7,3,5], [7,3,8], [7,3,9], [8,3,5]

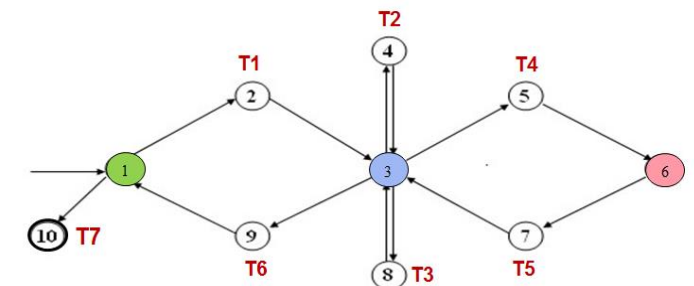
with sidetrip (but not def clear) = [2,3,8], [4,3,9], [8,3,8]

## Logic Conditions-

LC2, LC3, LC4

## Input Partitioning Criteria -

IP1, IP2, IP3, IP4



# Sequence Sheet with *getElems()*

- if it is possible to access attributes (e.g. via a *getElems()* operation), these can be used instead of (or in addition to) the other inspectors

	A	B	C	D	Transition/state	Logic	Data	elems
1		create	`Stack					
2	0	getElems	D1		1			0
3		push	D1	new Object	2	LC2	IP3	1
4	1	getElems	D1		3			
5	D3	pop	D1		9	LC3	IP4	0
6	0	getElems	D1		1			
7		push	D1	new Object	2	LC2	IP3	1
8	1	getElems	D1		3			
9	false	isFull	D1		3			
10		push	D1	new Object	4	LC2	IP1	2
11	2	getElems	D1		3			
12	D10	pop	D1		8	LC4	IP4	1
13	1	getElems	D1		3			
14		push	D1	null	4	LC2	IP2	1
15	1	getElems	D1		3			
16		push	D1	new Object	4	LC2	IP1	2
17	2	getElems	D1		3			
18	D16	pop	D1		8			1
19	1	getElems	D1		3			
20	D7	pop	D1		9	LC3	IP4	0
21	0	getElems	D1		1			
22		shutdown	D1		10			

## Executed Path –

[1, 2, 3, 9, 1, 2, 3, 3, 4, 3, 8, 3, 4, 3, 4, 3, 8, 3, 9, 1, 10]

## Covered Subpaths –

[4,3,8], [4,3,9], [4,3,4],  
[4,3,8], [8,3,4], [8,3,8], [8,3,9],  
[9,1,2], [1,10]

*Note: new Object is used for simplification (i.e., push some element ...)*



# Conclusion

- the expressiveness of sequence sheets makes it easy to write tests that concisely combine the three different kinds of test requirements
  - you may also consider mutations of the input space of operations (cf. syntax-based)
- final sequence sheets can provide a finely tuned balance between the demands of the different kinds of test requirements
- actuation sheets clearly show which test requirement has not been fulfilled and in what way
  - are always linear
  - show how the returned value deviates from the expected value
- the location of the problem is easier if it is uncovered as close as possible to the responsible operation
  - try to include at least one asserting operation after every changer operation
  - if this is not possible, try to minimize the length of assertion-free invocation sequences



# Literature

- Software Testing
  - Ammann, Paul, and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.
- FIT Tables
  - Mugridge, Rick, and Ward Cunningham. *Fit for developing software: framework for integrated tests*. Pearson Education, 2005.
- Test Sheets
  - C. Atkinson, F. Barth and D. Brenner, "Software Testing Using Test Sheets," *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, Paris, France, 2010, pp. 454-459, doi: 10.1109/ICSTW.2010.21.
- JUnit 5 - <https://junit.org/junit5/>
- Sequence Sheets
  - Publications
    - Marcus Kessel, Colin Atkinson, Promoting open science in test-driven software experiments, *Journal of Systems and Software*, Volume 212, 2024, 111971, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2024.111971>
    - Marcus Kessel , LASSO – an observatorium for the dynamic selection, analysis and comparison of software, Dissertation, 2023 - <https://madoc.bib.uni-mannheim.de/64107/>
  - LASSO Platform - <https://softwareobservatorium.github.io/>

