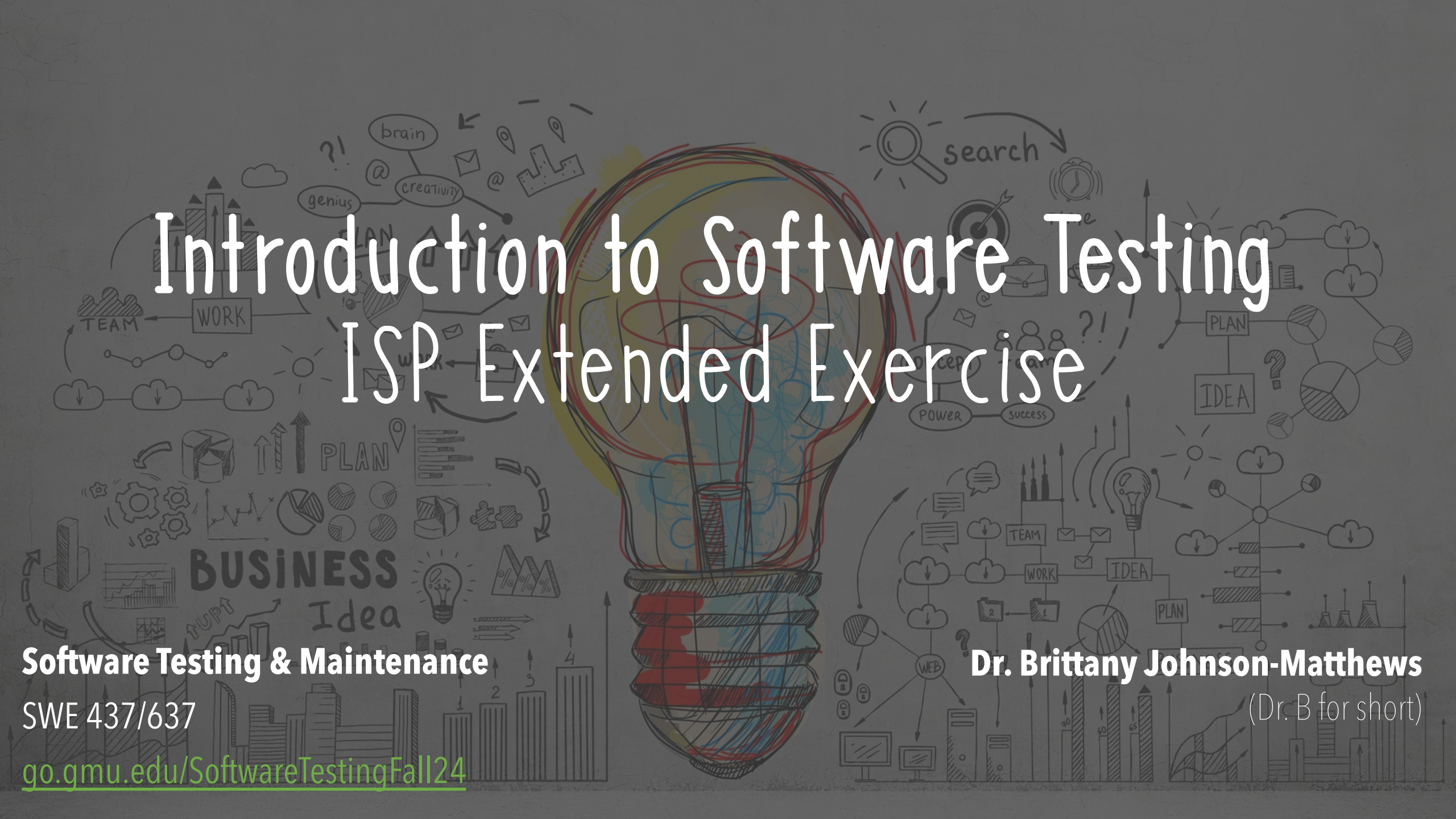


Introduction to Software Testing
ISP Extended Exercise

Software Testing & Maintenance

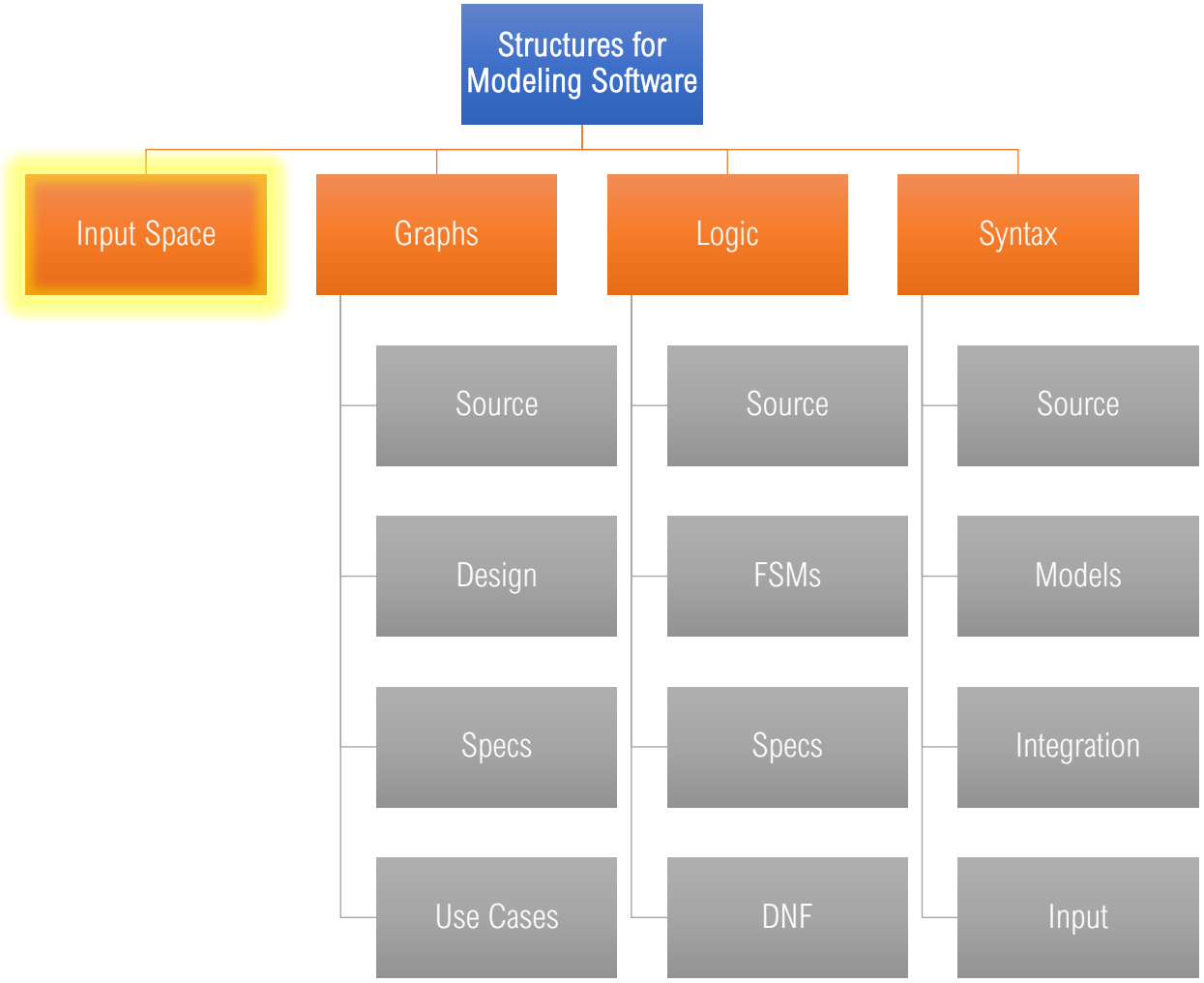
SWE 437/637

go.gmu.edu/SoftwareTestingFall24

Dr. Brittany Johnson-Matthews

(Dr. B for short)

Input Space Coverage



Input domains

Input domain: all possible inputs to a program

- Most input domains are effectively **infinite**

Input parameters define the input domain

- Parameter values to a method
- Data from a file
- Global variables
- User inputs

We **partition** input domains into *regions* (called ***blocks***)

Choose at least **one value** from each block

Input domain: Alphabetic letters

Partitioning characteristic: Case of letter

- **Block 1:** upper case
- **Block 2:** lower case

Partitioning input domains

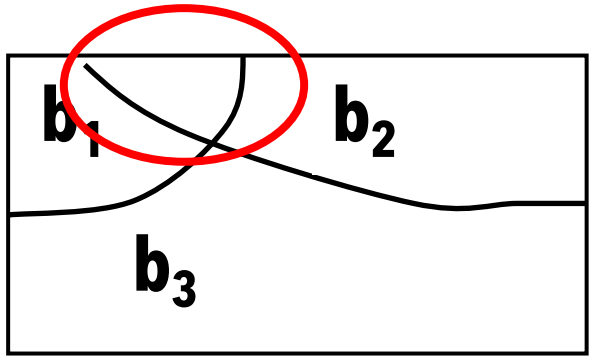
Domain ***D***

Partition scheme ***q*** *of* ***D***

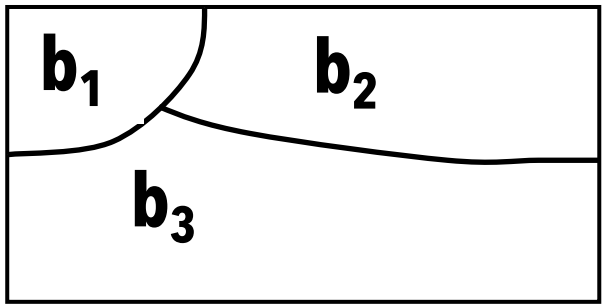
The partition ***q*** defines a set of blocks, $\boldsymbol{Bq = b_1, b_2, \ldots, b_q}$

The partition must satisfy two **properties**:

1. Blocks must be ***pairwise disjoint*** (no overlap)



2. Together the blocks ***cover*** the domain ***D*** (complete)



Input Characteristics

A feature or quality belonging typically to a person, place, or thing and serving to identify it.

Input: *people*

Abstraction

A = [a1, a2, a3, a4, a5]
B = [b1, b2, b3, b4, b5, b6]

Concrete

Characteristics: hair color, major
Blocks:
A = (1) red, (2) black, (3) brown, (4) blonde, (5) other
B = (1) cs, (2) swe, (3) ce, (4) math, (5) ist, (6) other

Creating an Input Domain Model (IDM)

Step 1: Identify testable functions

Step 2: Find all **inputs, parameters, & characteristics**

Step 3: Model the **input domain**

Step 4: Apply a test **criterion** to choose **combinations** of values (6.2)

Step 5: Refine combinations of blocks into **test inputs**

Step 1 – Identify Testable Functions

Individual ***methods*** have one testable function

Methods in a ***class*** often have the same characteristics

Programs have more complicated characteristics, modeling documents like UML can be used to design characteristics

Systems of integrated hardware and software components can have many testable functions – devices, operating systems, hardware platforms, browsers, etc.

Step 2 – Find all parameters

Often straightforward or mechanical

- Preconditions and postconditions
- Relationships among variables
- Special values (zero, null, etc.)

Do not use program source code, **characteristics should be based on the *input domain***

Methods: parameters and state variables
Components: parameters to methods and state variables
Systems: all inputs, including files and databases

Step 3 – Model the input domain

The domain is scoped by the *parameters*

The structure is defined by *characteristics*

Each characteristic is partitioned into *sets of blocks*

Each block represents a *set of values*

This is the most creative design step in ISP

- Better to have **more characteristics and fewer blocks**; leads to fewer tests
- Strategies include valid/invalid/special values, boundary values, "normal" values

Step 4 – Choose values

Apply a *test criterion* to choose *combinations* of values.

A test input has *one value* for each parameter

There is *one block* for each characteristic

Choosing *all combinations* is usually infeasible

- **Coverage criteria** allow subsets to be chosen

Step 5 – Translate into inputs

Refine combinations of blocks into *test inputs*.

Choose *appropriate values* for each block

Combinatorial test optimization tools can help

- These tools dramatically reduce the number of tests

Choosing Values (6.2)

After partitioning characteristics into blocks, testers design tests by combining blocks from different characteristics

- 3 characteristics: A, B, C
- Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

A test starts by combining one block from each characteristic

- Then values are chosen to satisfy the combinations

We use **criteria** to choose **effective** combinations

Choosing Values (6.2)

DEFINITION

All Combinations Coverage (ACoC) – all combinations of blocks from all characteristics must be covered

DEFINITION

Each Choice Coverage (ECC) – one value from each characteristic must be used in at least one test

DEFINITION

Base Choice Coverage (BCC) – a base 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.

Choosing values with BCC

Use *domain knowledge* of the program to identify important values

DEFINITION

Base Choice Coverage (BCC) – a base 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.

Choosing values with BCC

The base test must be **feasible**, that is, all values in the base choice must be compatible

Base choices can be:

- The most likely or most common values
- The simplest values
- The smallest values
- The first values in some logical ordering

Happy path tests make good base choices

The base choice is a *crucial design decision*

- Test designers should document why the base choice was selected
- A poor base choice can result in many infeasible combinations

BCC Example

Base choices

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

TR = { **(a1, b1, c1)**, Base test

(a2, b1, c1),
(**a3**, b1, c1), } Variations on A

(a1, **b2**, c1), Variation on B

(a1, b1, **c2**) } Variation on C

Choosing values with MBCC

There can sometimes be more than one logical base choice for each characteristic.

DEFINITION

Multiple Base Choice Coverage (MBCC) – at least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each 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.

MBCC Example

Multiple base choices

Characteristic	Blocks		
A	a1	a2	a3
B	b1	b2	--
C	c1	c2	--

TR = { **(a1, b1, c1),** Base choice #1

(a2, b1, c1), Variations on A

~~(**a3**, b1, c1),~~

(a1, **b2**, c1), Variation on B

~~(a1, b1, **c2**),~~ Variation on C

(a3, b1, c2), Base choice #2

~~(**a1**, b1, c2),~~ Variations on A

(**a2**, b1, c2),

(a3, **b2**, c2), Variation on B

~~(a3, b1, **c1**)~~ } Variation on C

Substituting a3 in place of a1 is not necessary because a3 is also a base choice and will show up in a later TR

Characteristics Constraints

Some combinations are **infeasible**

- Can't have "less than zero" and "scalene"

This is represented as **constraints**

Two general types of constraints

- A block from one characteristic *cannot be* combined with a specific block from another
- A block from one characteristic *can only be* combined with a specific block from another

Handling constraints depends on the criterion used

- ACC, PWC, TWC – drop the infeasible pairs
- BCC, MBCC – change a value to another non-base choice to find a feasible combination

Constraints Example

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

Characteristic	b_1	b_2	b_3	b_4	b_5	b_6
A: size and contents	list=null	size=0	size=1	size>1 varied unsorted	size>1 varied sorted	size>1 all same
B: match	Element not found	Element found once	Element found more than once	--	--	--
Infeasible combinations: (Ab_1, Bb_2), (Ab_1, Bb_3), (Ab_2, Bb_2), (Ab_2, Bb_3), (Ab_3, Bb_3), (Ab_6,Bb_2)						

Element cannot be in a null list once (or more than once)

Element cannot be in a 0-element list once (or more than once)

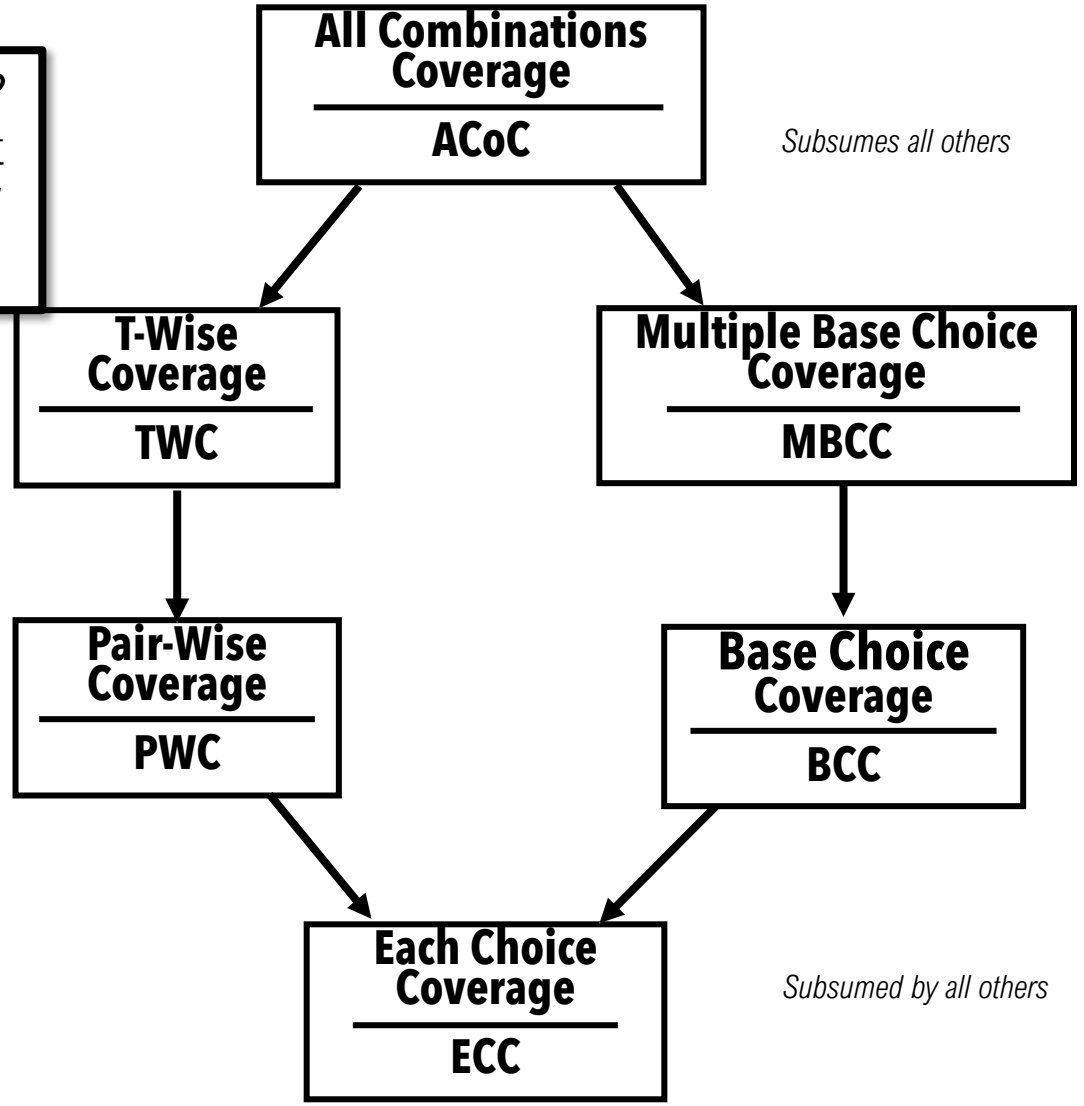
Element cannot be in a 1-element list more than once

If a list has many of the same element, we can't find it just once

ISP Coverage Criteria Subsumption

DEFINITION

A test criterion *C1* subsumes *C2* if and only if every set of test cases that satisfies criterion *C1* also satisfies *C2*



Input Space Partitioning Summary

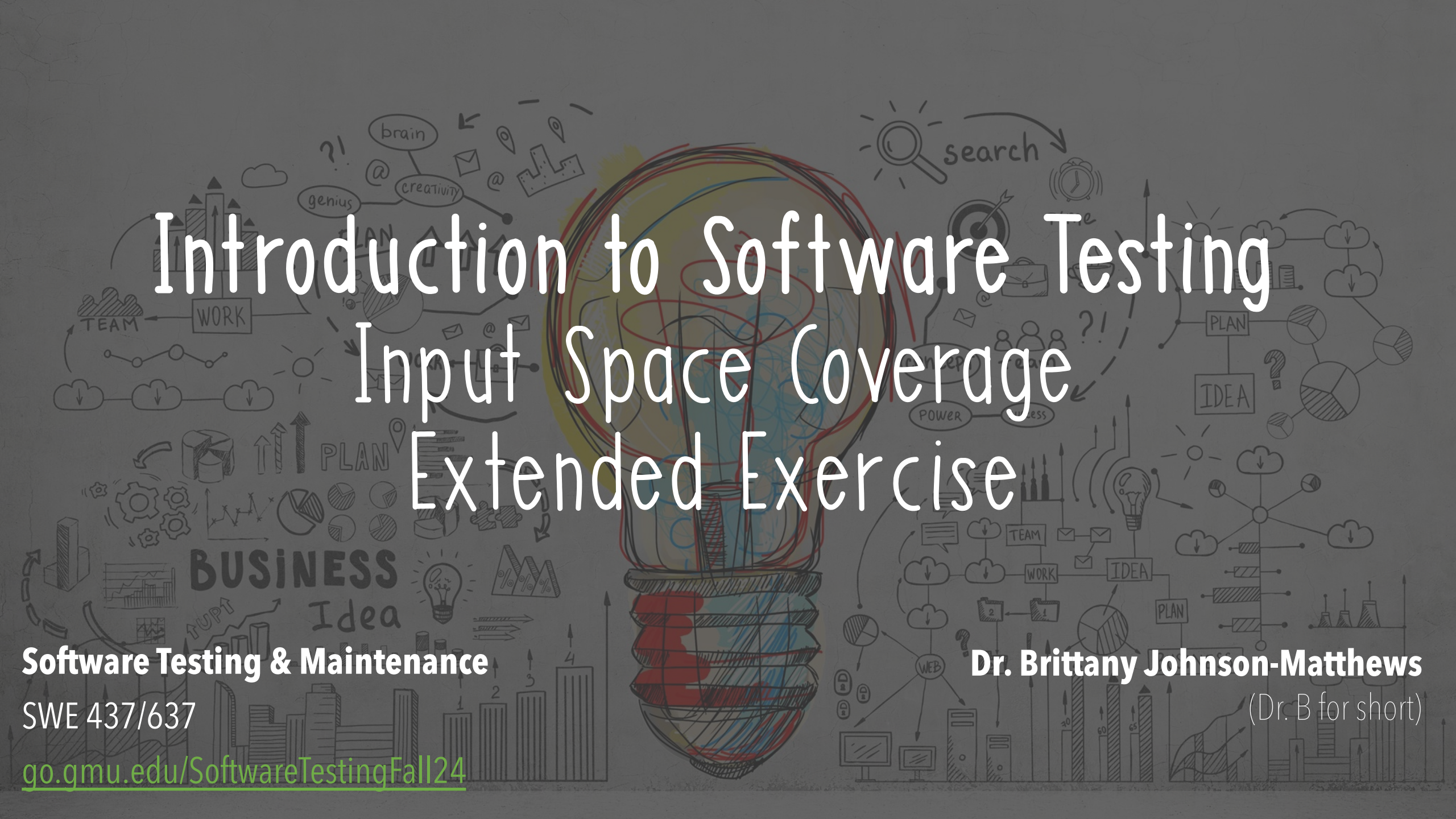
Fairly easy to apply, even with **no automation**

Convenient ways to **add more or less** testing

Equally applicable to **all levels** of testing – unit, class, integration, system, etc.

Based only on the **input space** of the program, not the implementation

Simple, straight-forward, effective, and widely used.



Introduction to Software Testing
Input Space Coverage
Extended Exercise

Software Testing & Maintenance

SWE 437/637

go.gmu.edu/SoftwareTestingFall24

Dr. Brittany Johnson-Matthews

(Dr. B for short)

Today's Exercise

Textbook chapter 6.4

Design an input domain model (IDM) for the Java 7 Iterator interface

https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html has the full version

Note that there may be some differences in the way I solve this exercise as compared to the textbook – **input domain modeling is a creative exercise!**

Java 7 Iterator

```
public interface Iterator<E> {
   /**
    * Returns true if the iteration has more elements. (In other words,
    *  returns true if next() would return an element rather than throwing
    *  an exception.)
    * @return true if the iteration has more elements
    */
    boolean hasNext();

   /**
    * Returns the next element in the iteration.
    * @return the next element in the iteration
    * @throws NoSuchElementException - if the iteration has no more elements
    */
    E next();

   /**
    * Removes from the underlying collection the last element returned by
    *  this iterator (optional operation). This method can be called only once
    *  per call to next(). The behavior of an iterator is unspecified if the
    *  underlying collection is modified while the iteration is in progress in
    *  any way other than by calling this method.
    * @throws UnsupportedOperationException - if the remove operation is not
    *  supported by this iterator
    * @throws IllegalStateException - if the next method has not yet been
    *  called, or the remove method has already been called after the last call
    *  to the next method
    */
    void remove();
}
```

Task 1 – Determine Characteristics

Step 1 – Identify characteristics in **Table A**

Step 2 – Develop characteristics

Step 3 – Associate methods and characteristics in **Table B**

Step 4 – Design a partitioning

Step 1 – Identify characteristics

Identify characteristics by considering

- Functional units
- Parameters
- Return types and values
- Exceptional behavior

Table A							
Method	Params	Returns	Values	Exception	Characteristic	ID	Covered by

Identify Characteristics

`hasNext()` – returns true if collection has more elements

`E next()` – returns next element

Exception – `NoSuchElementException` if at end

`void remove()` – removes the most recent element returned by the iterator

Exception – `UnsupportedOperationException`

Exception – `IllegalStateException`

Note that the void return challenges us to verify the behavior indirectly

Parameters – internal state of the iterator

Internal state changes with `next()` and `remove()`

Modifying the underlying collection directly also changes the iterator state

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext							

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	**state**						

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	**state**	**boolean**	**true, false**				

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	**state**	**boolean**	**true, false**	**--**			

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	**state**	**boolean**	**true, false**	**--**	**has more values**	**C1**	**--**

Identify Characteristics

`hasNext()` – returns true if collection has more elements

`E next()` – returns next element

Exception – `NoSuchElementException` if at end

`void remove()` – removes the most recent element returned by the iterator

Exception – `UnsupportedOperationException`

Exception – `IllegalStateException`

Note that the void return challenges us to verify the behavior indirectly

Parameters – internal state of the iterator

Internal state changes with `next()` and `remove()`

Modifying the underlying collection directly also changes the iterator state

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next							

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next	**state**						

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next	**state**	**E**	**E, null**	**?**	**?**	**?**	**?**

Let's leave this to your groups…

Identify Characteristics

`hasNext()` – returns true if collection has more elements

`E next()` – returns next element

- Exception – `NoSuchElementException` if at end

`void remove()` – removes the most recent element returned by the iterator

- Exception – `UnsupportedOperationException`
- Exception – `IllegalStateException`
- *Note that the void return challenges us to verify the behavior indirectly*

Parameters – internal state of the iterator

- Internal state changes with `next()` and `remove()`
- Modifying the underlying collection directly also changes the iterator state

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next	state	E	E, null	?	?	?	?
remove							

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next	state	E	E, null	?	?	?	?
remove	**state**						

Document in Table A

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next	state	E	E, null	?	?	?	?
remove	**state**	**--**	**--**	**?**	**?**	**?**	**?**

Let's leave this to your groups…

Step 2 – Develop characteristics

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
hasNext	state	boolean	true, false	--	has more values	C1	--
next	state	E	E, null	?	?	?	?
remove	state	--	--	?	?	?	?

Hint – think about both normal and exceptional conditions; each method can have *more than one row* for Exception, Characteristic, ID, and Covered By:

Table A							
Method	*Params*	*Returns*	*Values*	*Exception*	*Characteristic*	*ID*	*Covered by*
Method	Params	Returns	Values	Normal	…	…	…
				Ex1	…	…	…
				Ex2	…	…	…

Step 3 – Associate characteristics

Which characteristics are relevant for which methods?

Table B					
ID	Characteristic	hasNext()	next()	remove()	Partition
C1	**Has more values**				

Add or remove rows to the table as needed

Step 3 – Associate characteristics

How can we partition each characteristic?

Table B					
ID	Characteristic	hasNext()	next()	remove()	Partition
C1	**Has more values**				

Add or remove rows to the table as needed

Exercise 1

20 minutes to work

- Develop characteristics
- Associate characteristics with methods
- Partition characteristics into blocks

15 minutes for debrief and discussion

Task 2 – Define Test Requirements

Step 1 – Select a coverage criterion, we'll use **base choice** (BCC)

Step 2 – Identify a *happy-path* test for the base case in **Table C**

Step 3 – Identify test requirements (TRs)

Step 4 – Identify infeasible TRs

Step 5 – Refine TRs to remove infeasible cases

Refining Infeasible Requirements

Characteristic	b_1	b_2	b_3
Protein	Chicken	Fish	Lamb
Vegetable	Asparagus	Eggplant	Squash
Starch	Bread	Rice	Potato

Applying base choice coverage, we might select a base test
{ **C**hicken, **S**quash, **R**ice }

BCC requires that we vary each characteristic:
{**F**,S,R}, {**L**,S,R}, {C,**A**,R}, {C,**E**,R}, {C,S,**B**}, {C,S,**P**}

Assume that {**F**,S,R} is infeasible – BCC requires that we have a test with Fish, so keep Fish and try changing one (or both) of the other characteristics – is {**F**,A,R} feasible? Is {**F**,S,P}? Maybe {**F**,S,B}?

If we can't find *any* feasible combination that includes Fish, then we discard the TR

Exercise 2

15 minutes to work

- Create a happy-path base test
- Build a set of base choice tests
- Identify infeasible test requirements
- Develop replacement test requirements for any infeasible test requirements

10 minutes for debrief and discussion

Task 3 – Automate Tests

We need an *implementation* of Iterator because Iterator is merely an interface

- `ArrayList` implements Iterator, so we can use `ArrayList` for our testing

Create a test fixture with two variables

- List of strings
- Iterator for strings

`@Before setup()`

- Creates a list with two strings
- Initializes an iterator

Task 3 – Automate Tests

Example implementation framework

```
public class IteratorTest {

   private List<String> list;              // test fixture list
   private Iterator<String> itr;           // test fixture iterator

   @Before public void setUp()             // set up test fixture
   {
      list = new ArrayList<String>();      // create new ArrayList
      list.add ("cat");                    // append “cat”
      list.add ("dog");                    // append “dog”
      itr = list.iterator();               // initialize the iterator
   }

   ...    // test implementations to be defined on upcoming slides
}
```

Exercise 3

10 minutes to work

- Write tests for `hasNext()`
- Write tests for `next()`
- Write tests for `remove()`

No debrief, but answers will be posted

Automate Tests

Write tests for `remove()`

5 test cases (1-3 shown)

```
// Test 1 of remove(): testRemove_BaseCase(): C1=T, C2=T, C3=T, C4=T
@Test public void testRemove_BaseCase()
{
   itr.next(); // consume “cat”
   itr.remove(); // remove “cat”
   assertFalse (list.contains ("cat")); // verify list does not contain “cat”
}

// Test 2 of remove(): testRemove_C1(): C1=F, C2=F, C3=T, C4=T
@Test public void testRemove_C1()
{
   itr.next(); // consume “cat”
   itr.next(); // consume “dog”
   itr.remove(); // remove “dog”
   assertFalse (list.contains ("dog")); // verify list does not contain “dog”
}

// Test 3 of remove(): testRemove_C2(): C1=T, C2=F, C3=T, C4=T
@Test public void testRemove_C2()
{
   list.add (null); // append a null object to the list
   list.add ("elephant"); // append “elephant” to the list
   itr = list.iterator(); // reinitialize the iterator
   itr.next(); // consume “cat”
   itr.next(); // consume “dog”
   itr.next(); // consume null; iterator not empty
   itr.remove(); // remove null from list
   assertFalse (list.contains (null)); // verify list does not contain null
}
```