

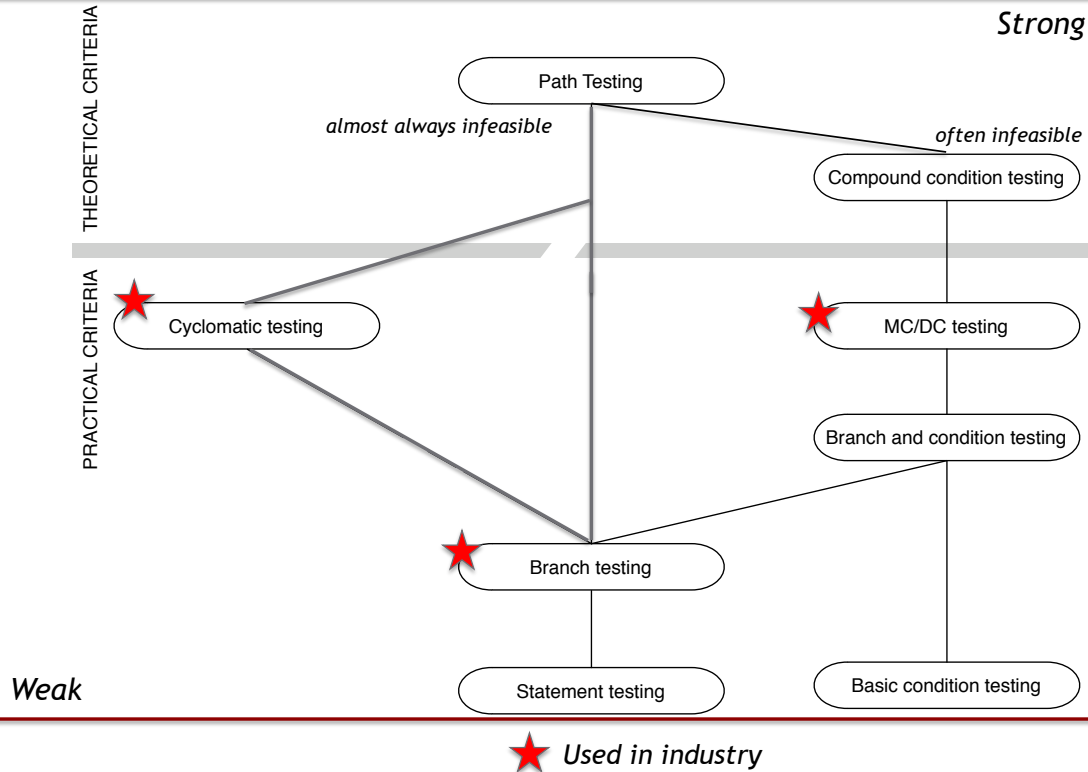
MIDTERM REVIEW

(CANVAS > MODULES > EXAMS)

EXAM

- ~110 minutes
- Be there about 5 minutes before class start time!
- Closed-book, closed-internet
- Test Lockdown Browser by trying the sample exam (there is also a Test Lockdown Browser quiz)
- Know the rules (really important!)
 - will not accept excuses along the lines "I didn't read instructions...", "I didn't know...", "I just assumed...", ...

CFG structural coverage criteria



Branch coverage

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage measure:


$$\frac{\# \text{ executed branches}}{\# \text{ branches}}$$

*Do you know how to calculate branch coverage given a set of test cases and a CFG? **Count the forks, not the joins!***

Compound condition coverage

Sometimes decisions in code are compound:

e.g., `(a > 0) || ((b > 0) && (z != null) && (a + b < 100) && (c != null))`

- All combinations of basic conditions within each decision
- Both outcomes of all decisions (all compound branches)
- *Exponential complexity!*

Worst-case: Without short-circuit evaluation, how many test cases?
 $2^5 = 32$ just for this one compound condition

MC/DC Example

trying to keep all basic conditions except one fixed,
and
flip the decision by changing the truth value of the remaining
basic condition

(not guaranteed to satisfy MC/DC for all basic conditions)

How many MC/DC test cases for this conditional?



$(a == b \ || \ a == c) \ \&\& \ (c > d)$ 3 basic conditions

How many test cases does this compound condition
require for MC/DC coverage? $3 + 1 = 4$

How many test cases does a method with two loops and
two if-then blocks require for full MC/DC coverage?
Don't know. Don't have the info on compound conditions.

MC/DC: a compromise condition coverage, and effective

A compound condition in a control-flow construct
with 4 basic conditions

$(a \ \&\& \ b) \ || \ (c \ \&\& \ d)$

For each compound
condition with N basic
conditions:
 $N + 1$ test cases

MC/DC Truth Table

#	a	b	c	d	Decision	covered
1						
2						
3						
4						
5						

Need 5 test cases

MC/DC test
case table

Do you know how to create an MC/DC truth table and derive test cases?

MC/DC example

$(a \ \&\& \ b) \ || \ (c \ \&\& \ d)$

#	a	b	c	d	Decision	covered
1	T	T	T	T	T	
2						
3						
4						
5						

Start with a random set of values

MC/DC example

↓
(**a** && b) || (c && d)

#	a	b	c	d	Decision	covered
1	T	T	T	T	T	
2	F	T	T	T	T	a???
3						
4						
5						

In #2: Try to cover **a** with #1
Doesn't work; try another set of values for #1

MC/DC example

 (a && b) || (c && d)

#	a	b	c	d	Decision	covered
1	T	T	F	T	T	
2						
3						
4						
5						

Start over: try another set of values for b, c, d that retains decision outcome (T)

MC/DC example

↓
(**a** && b) || (c && d)

#	a	b	c	d	Decision	covered
1	T	T	F	T	T	
2	F	T	F	T	F	a (1)
3						
4						
5						

In #2: Try to cover **a** with #1
Works!

MC/DC example

(a && **b**) || (c && d)



#	a	b	c	d	Decision	covered
1	T	T	F	T	T	
2	F	T	F	T	F	a (1)
3	T	F	F	T	F	b (1)
4						
5						

In #3: Try to cover **b** with #1 or #2
#2 doesn't work; #1 works

MC/DC example

(a && b) || (c && d)



#	a	b	c	d	Decision	covered
1	T	T	F	T	T	
2	F	T	F	T	F	a (1)
3	T	F	F	T	F	b (1)
4	T	F	T	T	T	c (3)
5						

In #4: Try to cover **c** with #1, #2, or #3:
#1 doesn't work; #3 and #2 work; let's pick #3

MC/DC example

(a && b) || (c && **d**)



#	a	b	c	d	Decision	covered
1	T	T	F	T	T	
2	F	T	F	T	F	a (1)
3	T	F	F	T	F	b (1)
4	T	F	T	T	T	c (3)
5	T	F	T	F	F	d (4)

In #5: Try to cover **d** with #1, #2, #3, or #4:
#4 works!

MC/DC example

$(a \ \&\& \ b) \ || \ (c \ \&\& \ d)$

#	a	b	c	d	Decision	covered
1	T	T	F	T	T	
2	F	T	F	T	F	a (1)
3	T	F	F	T	F	b (1)
4	T	F	T	T	T	c (3)
5	T	F	T	F	F	d (4)

If stuck, backtrack and try to cover a previous condition with a different row that works, or if that doesn't succeed, we may have to eventually try a different starting point

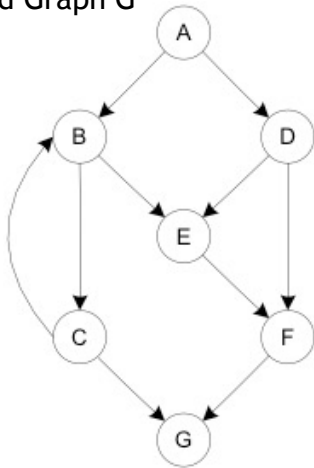
Path coverage is too ambitious

- All paths: includes all combinations of all decisions
 - Difference between condition and path coverage:
 - Condition coverage: no memory
 - Path coverage: has memory (remembers how you arrived at a decision!)
 - Too many paths in a CFG, especially if there are loops
 - *Exponential complexity!*
 - Cyclomatic coverage provides a compromise!
-

Cyclomatic Coverage Example

Cyclomatic complexity measures the control flow complexity of a piece of code

Connected Graph G



Find a basis for this CFG?

Cyclomatic complexity = $V(G)$

$$= e - n + 2 = 10 - 7 + 2 = 5$$

= number of linearly independent paths whose weighted superposition can cover all paths of G

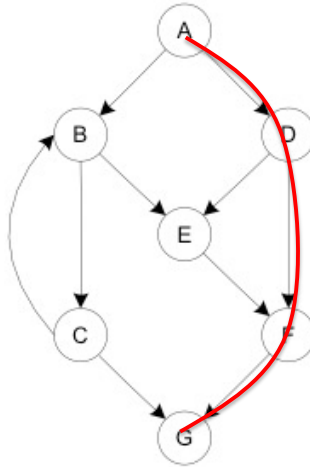
➔ Such a path set forms a **basis** for G

Special treatment for loops:

basis paths must both execute and skip each loop!

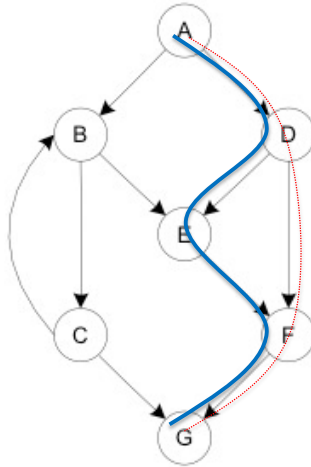
depth-first search heuristic *with* loop exit conditions prioritized over loop entry conditions

Constructing a basis



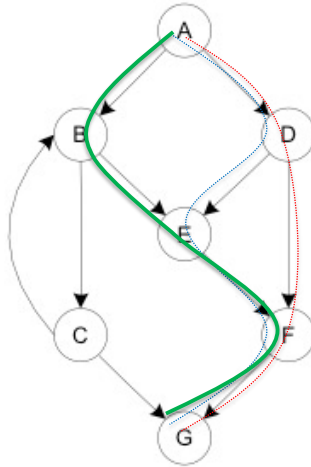
1
any nominal path

Constructing a basis



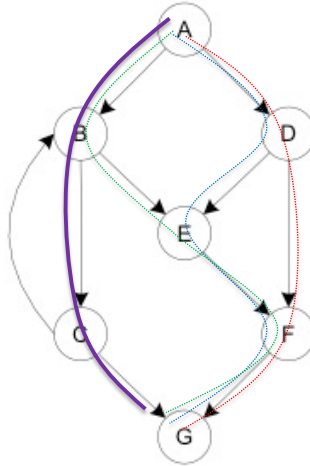
2
add new edge

Constructing a basis



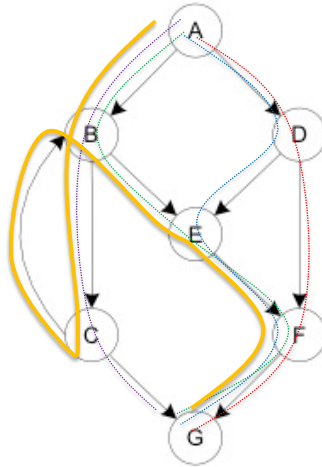
3
add new edge
exit loop first

Constructing a basis



4
add new edge

Constructing a basis



5
add final edge

Done!

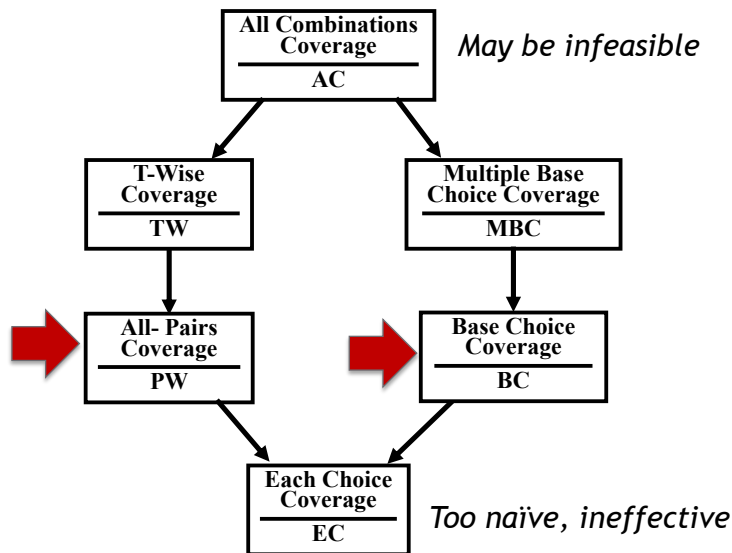
Spec-Based Testing

Combinatorial Strategies

All-Pairs Coverage Example

Spec-based combinatorial adequacy criteria

- Corresponding to different combinatorial testing strategies
- Answering the question “How well do the tests **cover** the input space or the spec?”



All-Pairs: 4 characteristics, all combinations feasible

- A: 4 blocks {a1, a2, a3, a4 }
- B: 3 blocks {b1, b2, b3 }
- C: 2 blocks {c1, c2 }

Possible ways of choosing characteristic pairs:

$$C(3, 2) = 3!/2!(3 - 2!) = 3$$

(A, B), (A, C), (B, C)

Pairwise combinations that need to be covered

Characteristic Pair	Possible Pairwise Combinations
(A, B)	$4 \times 3 = 12$
(A, C)	$4 \times 2 = 8$
(B, C)	$3 \times 2 = 6$
Total: 6 Pairs	26 Pairwise Combinations

- If each case covers 3 entirely new pairs...
 - Min # of test cases = $\text{Ceiling}(26/3) = 9$, but...
 - to cover largest attributes (A, B), must have at least $4 \times 3 = 12$ test cases
- If each case covers just 1 new pair (no overlaps):
 - Max # of test cases = **26 (but we will do better because of overlaps)**

Calculating all pairs coverage

Case #	A	B	C		# new	Cum. new
1	a1	b1	c1		3	3
2	a1	b2	c1		2	5
3	a1	b3	c2		3	8
4	a2	b1	c1		2	10
5	a2	b2	c1		1	11

Possible additional ones to reach AP

e.g.,	a3	b3	c1		3	

given test cases

Total pairwise combinations to be covered

= 26

Covered pairwise combinations = 11

All-Pairs Coverage
= $11/26 = 42\%$

Need more

Base-Choice Combinatorial Strategy

Make sure to review! It's easy but you must work it out yourself at least once! See sample question on Multiple Base-Choice (MBC)

SOLID principles support testability

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)
 - **Dependency Injection** ★

How do they support testability and testing?

Dependency Inversion Principle (DIP)

“Code should depend on abstractions, not on concretions.”

- Complex high-level classes should not directly depend on low-level classes that are likely to change: instead both should depend on abstractions
- DIP (a principle) *leads to* dependency injection (a ubiquitous mechanism that we use all the time to achieve DIP)

“a class shouldn’t instantiate its own collaborators but rather have their instances passed in with constructor/setter injection or as a parameter to a method that requires the collaborator”

Dependency Inversion Principle (DIP)

“Code should depend on abstractions, not on concretions.”

- Complex high-level modules should not depend on low-level modules: both should depend on abstractions. Instead of low-level modules depending on high-level modules, high-level modules depend on abstractions.
- DIP (a principle of software design) is a popular implementation of the DIP.

“a class should depend on an interface, not on a concrete class, and collaborators should be passed in”

**This is super important
for testing: why?
Implications for test
doubles!**

**Evident in Vocabulary
question!**

DI \neq DIP

Coding Question on Vocareum

Go to Canvas > Modules > Exams
> Sample Midterm Coding Question

Vocareum coding question

- Take 5 minutes to read the question
- Activate Vocareum from Canvas.
- Take 10 minutes to solve it.

Solution 1 (with Mockito, no stubbing)

@Test

```
public void salespersonNotifiedWhenOrderChanged() {
```

```
    SalesPerson hakan = new SalesPerson("Hakan");
```

```
    SalesPerson hakanSpy = spy(hakan);
```

```
    Order ord = new Order(23, 45, hakanSpy) {
```

```
        @Override public void process() {
```

```
            processed = true;
```

```
        }
```

```
    };
```

```
    ord.process();
```

```
    ord.change(5);
```

```
    verify(hakanSpy).orderChanged(ord);
```

```
}
```

Inject spy for collaborator

Override tested class to emulate order processing without actually processing

Workflow that production code normally follows: just reproduce it in test

Assertion: verify that collaborator is notified

Solution 2 (with Mockito, with stubbing)

```
@Test
public void salespersonNotifiedWhenOrderChangedAlternate() {
    SalesPerson hakan = new SalesPerson("Hakan");
    SalesPerson hakanSpy = spy(hakan);
    Order ord = new Order(23, 45, hakanSpy);
    Order ordSpy = spy(ord);
    doNothing().when(ordSpy).process(); // stub out process()
    ordSpy.process();
    ordSpy.processed = true; // need this to emulate its side effect
    ordSpy.change(5);
    verify(hakanSpy).orderChanged(ordSpy);
}
```

Solution 3 (not using Mockito, hand-made spy)

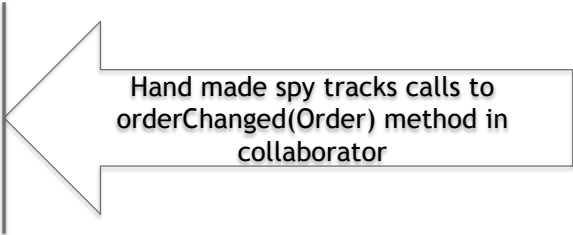
```
@Test
public void salespersonNotifiedWhenOrderChangedNoMockito() {

    class SalesPersonSpy extends SalesPerson {
        public boolean orderChanged = false;

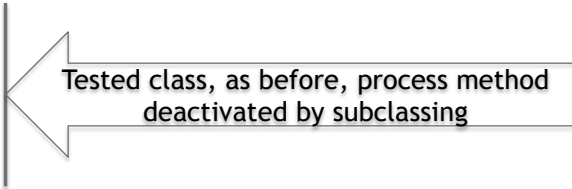
        SalesPersonSpy(String owner) {
            super(owner);
        }

        @Override public void orderChanged(Order order) {
            orderChanged = true;
        }
    };

    SalesPersonSpy hakan = new SalesPersonSpy("Hakan");
    Order ord = new Order(23, 45, hakan) {
        @Override public void process() {
            processed = true;
        }
    };
    ord.process();
    ord.change(5);
    assertTrue(hakan.orderChanged);
}
```



Hand made spy tracks calls to
orderChanged(Order) method in
collaborator



Tested class, as before, process method
deactivated by subclassing

Main Principles of SVT

- **Redundancy**: triangulating results
- **Partitioning**: divide and conquer
- **Approximation** (Restriction*): making the problem easier (or possible!)
- **Visibility**: making information accessible
- **Feedback**: providing actionable information
- **Repeatability**: better to fail every time than sometimes

Can you give an application examples of each principle?

Main Principles: examples both from the point of view of designing tests and designing testable production code

- **Redundancy**: *use of both spec-based and structural testing, which may generate overlapping test cases*
- **Partitioning**: *simplicity and single-purpose principles, modularity, dividing a big function into small focused functions that are easy to test*
- **Approximation** (Restriction*): *use of test doubles*
- **Visibility**: *avoiding testability inhibitors, design that doesn't hide side effects and behavior from tests, running all tests after each build on an integration server*
- **Feedback**: *failure messages that describe in detail why a test failed, using specific rather than generic assertions*
- **Repeatability**: *determinism, independence principles*

Good unit testing practices

- Simplicity
- Understandability
- Essentiality
- Single purpose
- Behavior first
- Maintainability
- Determinism
- Independence
- Failability
- Comprehensiveness
- Speed

Can you name them and explain them?

TDD

- Why TDD?
- What characterizes ideal TDD?

Is it strictly about writing tests first? Is it about integrating testing with development in a tight way, so that tests don't fall behind production code?

Mutation Testing:

Estimating code quality

Test suite:

- Killed 10 out of 100 mutants
- Plus found 5 native bugs

Test suite was 10% effective in finding mutants. If it were equally effective in finding native bugs, then...

$$\frac{\# \text{ Mutants Killed}}{\text{Total } \# \text{ Mutants}} = \frac{\# \text{ Native Bugs Found}}{\text{Total } \# \text{ Native Bugs}}$$

$10/100 = 5/N \rightarrow N = 50$ Total # Native Bugs
45 Native Bugs remaining (quality estimate)



Test Case Design Example

time permitting, otherwise you may want to review on your own for your project!

Login (18652 Project)

- Usernames are provided by users and should be at least 3 character long.
 - Usernames should be different from the list of banned usernames.
 - If Username already exists and password is correct, login is accepted.
 - If Username already exists, but password is incorrect, login is rejected.
 - If Username does not exist, and password is valid, new user is created and login is accepted.
 - If the Username does not exist, and password is invalid, login is rejected.
-

Potential attributes/characteristics



Highlight potential attributes/characteristics

Potential attributes



-
- Username should be at least 3 character long.
 - Username should not be in the list of banned usernames.
 - If Username already exists and password is correct, login is accepted.
 - If Username already exists, but password is incorrect, login is rejected.
 - If Username does not exist, and password is valid, new user is created and login is accepted.
 - If the Username does not exist, and password is invalid, login is rejected.
-

Attributes



-
- Username
 - Password
 - UserExists? (internal state of the system)
 - PCorrect? (internal state of the system)

These are the required inputs and starting states whose values will be needed for actual test cases

Potential characteristics



- Username should be **at least 3 character long**.
- Username should **not be in the list of banned usernames**.
- If Username already exists and password is correct, **login is accepted**.
- If Username already exists, but password is incorrect, login is **rejected**.
- If Username does not exist, and **password is valid**, **new user is created** and **login is accepted**.
- If the Username does not exist, and **password is invalid**, login is rejected.

Characteristics



Input-related

- ...

Output-related

- ...

Characteristics



Input

- ULength
 - UBanned?
 - UExists?
 - PCorrect?
 - PValid?
- } ~~UValid?~~
= ULength >= 3
& !UBanned

- I choose **ULength** and **UBanned?** (two small characteristics) over **UValid** (one big characteristic) because I prefer more characteristics with few blocks to few characteristic with more blocks

Output

- LoginAccepted?
- UCreated?

- Let's delay dealing with **LoginAccepted?** and **UCreated?** for now. We want all feasible combinations to be covered, but they may be automatically handled by the above. We can check later and add test cases if necessary.

Partitions



Characteristic	Partition	Constraints/Notes
ULength		
UBanned?		
UExists?		
PCorrect?		
PValid?		

Partitions



Characteristic	Partition	Constraints/Notes
ULength	0..1, 2, 3, > 3 (but !too-long), too-long	Boundary values; 0..1, 2 only if !Uexists Could add 4 as separate block
UBanned?	T, F	If ULength >= 3 but not too-long; T only with !Uexists
UExists?	T, F	If ULength >= 3 but not too-long & !UBanned
PCorrect?	T, F	If Uexists
PValid?	T, F	If !Uexists

Apply All Combinations subject to Constraints

Test Case Specs -- All Combinations subject to constraints



Case	ULength	UBanned?	UExists?	PCorrect?	PValid?	
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
...	...					

Test Case Specs -- All Combinations subject to constraints



Case	ULength	UBanned?	UExists?	PCorrect?	PValid?	LoginAccepted?
1	0..1	-	F	-	T	
2	0..1	-	F	-	F	
3	2	-	F	-	T	
4	2	-	F	-	F	
5	3	T	F	-	T	
6	3	T	F	-	F	
7	3	F	T	T	-	
8	3	F	T	F		
9	3	F	F	-	T	
10	3	F	F	-	F	
11-16	Repeat 5-10 for ULength > 3					

Test Case Specs with oracle



Case	ULength	UBanned?	UExists?	PCorrect?	PValid?	LoginAccepted?
1	0..1	-	F	-	T	F
2	0..1	-	F	-	F	F
3	2	-	F	-	T	F
4	2	-	F	-	F	F
5	3	T	F	-	T	F
6	3	T	F	-	F	F
7	3	F	T	T	-	T
8	3	F	T	F	-	F
9	3	F	F	-	T	T
10	3	F	F	-	F	F
11-16	Repeat 5-10 for ULength > 3					
17-18	Repeat 1-2 for ULength = too-long					

Test Cases with actual attributes and oracles (adding oracle for UCreated?)



Case	ULength	UBanned?	UExists?	PCorrect?	PValid?	Username	Password	LoginAccepted?	UCreated?
1	0	-	F	-	T	?	?	F	F
2	1	-	F	-	F	?	?	F	F
3	2	-	F	-	T	?	?	F	F
4	2	-	F	-	F	?	?	F	F
5	3	T	F	-	T	?	?	F	F
6	3	T	F	-	F	?	?	F	F
7	3	F	T	T	-	?	?	T	F
8	3	F	T	F	-	?	?	F	F
9	3	F	F	-	T	?	?	T	T
10	3	F	F	-	F	?	?	F	F
11-16	Repeat 5-10 for ULength > 3 (choose a different length in each)								
17-18	Repeat 1-2 for ULength = too-long (choose a different length in each)								

Check all feasible combinations of LoginAccepted? and UCreated? are covered! (F, T) is missing, but it's infeasible.