Test and Validation of Software

Nuno Ribeiro
`ist199293`

Rúben Nobre
`ist199321`

Francisco Abreu
`ist1110946`

2nd Semester

1 Testing Class Question

In the initial phase of our testing strategy, it is imperative to determine whether the *Question* class is **Modal** or **Non-Modal**. A Modal Test class involves strict sequencing in method calls and a defined transition of states, whereas a Non-Modal class is characterized by minimal constraints on the order of method invocations.

Given the **Non-Modal** nature of the *Question* class, our test strategy will focus on understanding and effectively covering the class's state space and complex interface. We need to ensure that all legal sequences of method calls are tested, along with various combinations of method parameters.

Due to the trivial nature of the state control model, state-based testing, which is usually reserved for Modal classes where specific sequences of state transitions are critical, is not suitable for this class. Instead, testing should focus on interface functionality and interaction between methods under various conditions, rather than on state transitions.

We start by analyzing the class invariant:

$$0 < \text{weight} \leq 15 \land \quad \text{topic.length} \geq 6 \quad \land \quad 1 < \#\text{topics\_list} \leq 5 \quad \land \quad 2 \leq \text{choices.size} \leq 8 \quad \land$$

$$\forall_{t1,t2 \in topic\_list} \implies t_1 \,!= t_2 \quad \text{and body != null and correctChoice < \#choices}$$

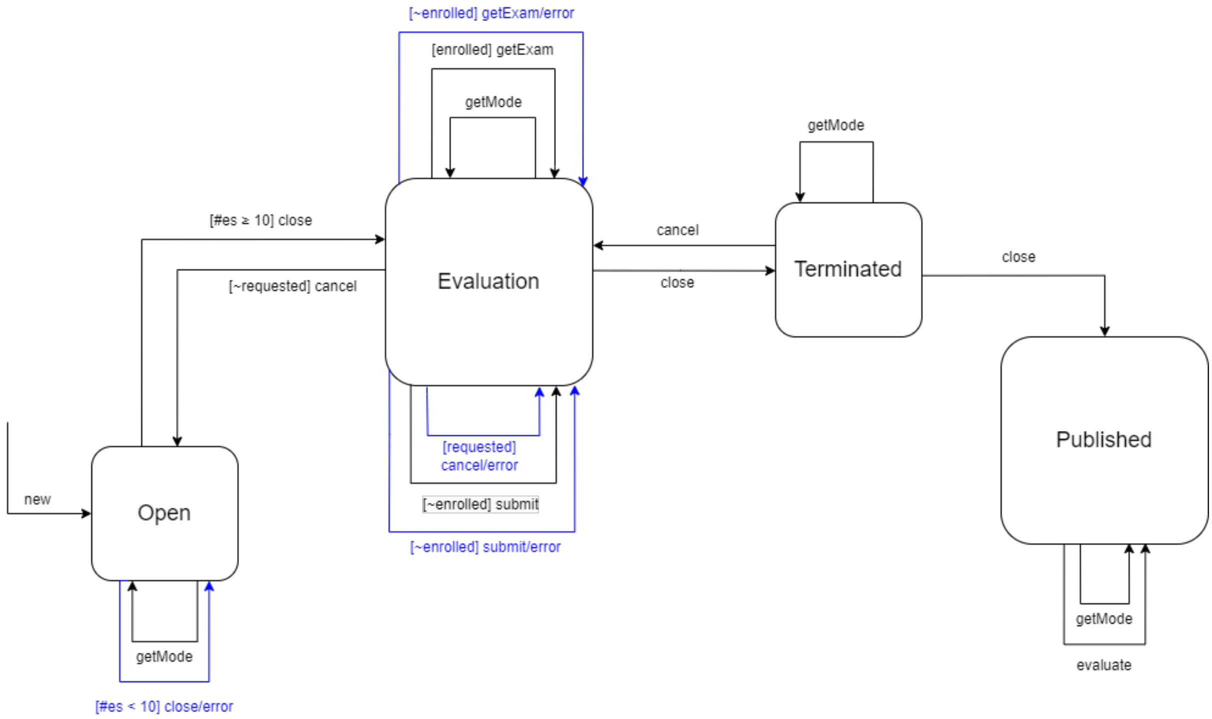
With the *invariant* defined, the next step is doing the *Domain Matrix*:

var	cond		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
weigth	>0	On	0																			
		Off		1																		
	<=15	On			15																	
		Off				16																
	Typ	In					2	3	4	5	6	7	8	9	10	11	12	13	14	2	3	4
topic.length	>= 6	On					6															
		Off						5														
	Typ	In	7	8	9	10			11	12	13	14	15	16	17	18	19	20	35	40	45	50
#topic_list	<= 5	On							5													
		Off								6												
	>= 1	On									1											
		Off										0										
	Typ	In	2	3	4	2	3	4					2	3	4	2	3	4	2	2	3	4
#choices	>= 2	On											2									
		Off												1								
	<= 8	On													8							
		Off														9						
	Type	In	3	4	5	6	7	3	4	5	6	7					3	4	5	5	5	5
unique_topics	== true	On															true					
		Off																false				
	Typ	In	true	true	true	true	true	true	true	true	true	true	true	true	true	true			true	true	true	true
body	!= null	On																	null			
		Off																		10		
	Typ	In	32	32	20	39	19	19	32	29	15	18	19	25	26	30	45	9			40	21
correctChoice	<#choices	On																			5	
		Off																				4
	Typ	In	0	0	3	5	3	0	1	4	2	4	1	0	6	5	1	3	2	3		
Expected Result			×	✓	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	×	✓	×	✓

We have implemented a total of **8** test cases in a *Java* file, which is included in the accompanying ZIP file. These test cases are specifically designed to evaluate the functionality of the Question class at the class scope level. The suite comprises four success test cases and four failure test cases to ensure comprehensive coverage and robustness of the class's features.

2 Class-scope test ExamManager

The testing of the class ExamManager requires the most elaborate of techniques used by us since this class is a **modal** class. This means that there are specific constraints to what messages we can receive, depending on the state in which the class finds itself in.
Below is the state diagram for the ExamManager class.



After the initial state diagram is sketched, we must enumerate all possible conditions including the ones that are not mentioned explicitly in the problem. The proposition *requested* is used for when we mean to say that a *student has requested the exam* and the proposition *enrolled* means that the student is enrolled in the course related to the ExamManager instance. For this, we build a table containing each condition and its corresponding negation, adding it to the initial state diagram in blue.

State	Message	Condition	Next State
Open	close	Pre: $\#e.s. \geq 10$	Evaluation
Open	close/error	Pre: $\#e.s. < 10$	Open
Evaluation	getExam	Pre: enrolled == 1	Evaluation
Evaluation	getExam/error	Pre: enrolled == 0	Evaluation
Evaluation	submit	Pre: enrolled == 1	Evaluation
Evaluation	submit/error	Pre: enrolled == 0	Evaluation
Evaluation	cancel	Pre: requested == 0	Open
Evaluation	cancel/error	Pre: requested == 1	Evaluation

Table 1: Conditional Transition Variants, class ExamManager

Next, we draw the new transition/conformance tree for the ExamManager class, now including the aforementioned additional condition sets:

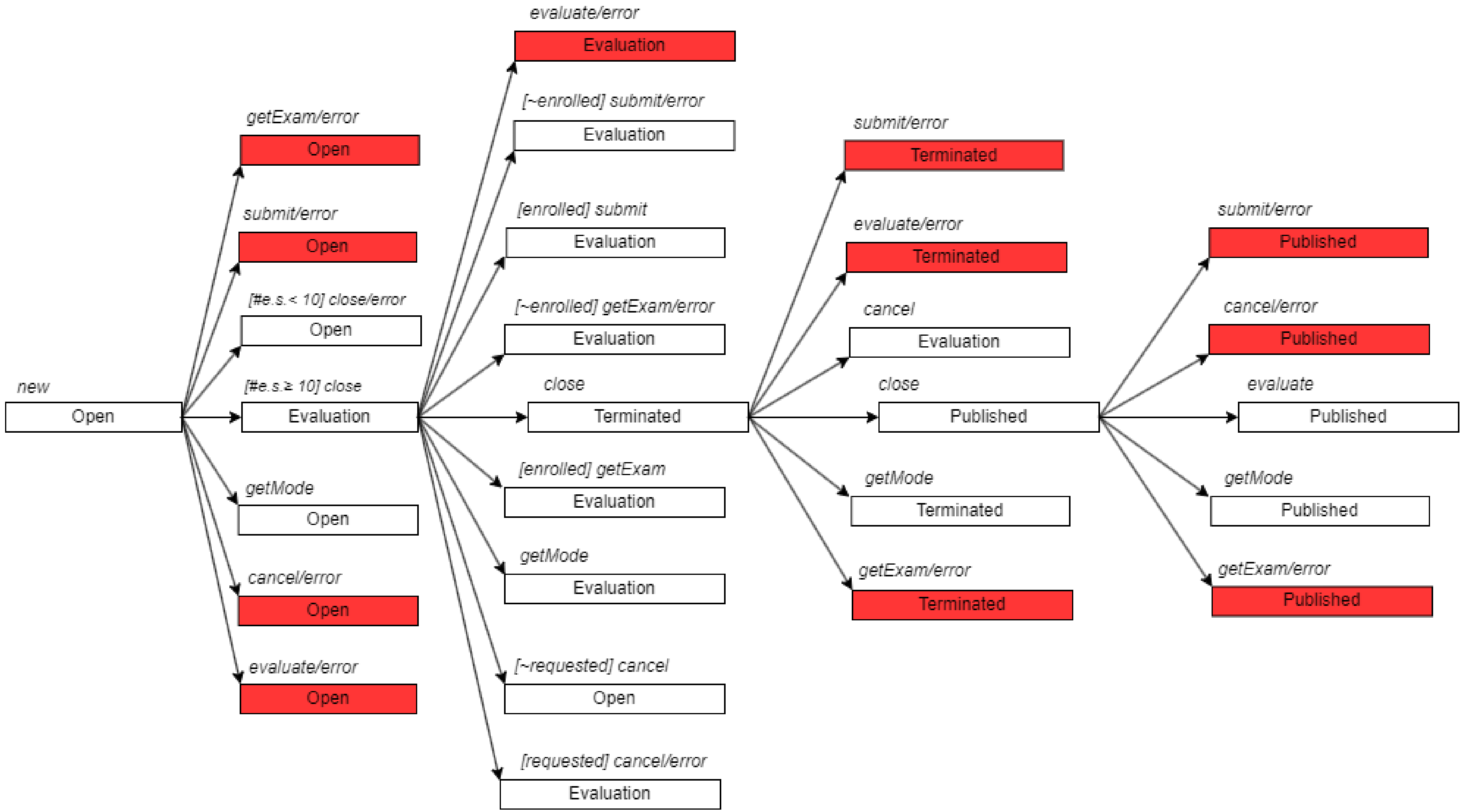


Figure 1: Expanded conformance tree for class ExamManager

Now we must generate the test data for each possible, valid path; since the only scalar/numeric parameter in the class is the *number of enrolled students* (#e.s.), we get:

close in Open		
Condition	On Point	Off Point
[#e.s. $\geq$ 10]	10 ✓	9
[#e.s. $<$ 10]	10	9 ✓

Table 2: Test data using Invariant Boundaries

We are now ready to develop a conformance test suite for the class ExamManager:

Run	Test Run/Event Path					Expected State	Throws Exception
	Level 1	Level 2	Level 3	Level 4	Level 5		
1	new					Open	✗
2	new	[#e.s. $<$ 10] close				Open	✓
3	new	getMode				Open	✗
4	new	[#e.s. $\geq$ 10] close				Evaluation	✗
5	new	[#e.s. $\geq$ 10] close	[¬enrolled] submit/error			Evaluation	✓
6	new	[#e.s. $\geq$ 10] close	getExam			Evaluation	✗
7	new	[#e.s. $\geq$ 10] close	getMode			Evaluation	✗
8	new	[#e.s. $\geq$ 10] close	[enrolled] submit			Evaluation	✗
9	new	[#e.s. $\geq$ 10] close	[¬requested] cancel			Open	✗
10	new	[#e.s. $\geq$ 10] close	[requested] cancel/error			Evaluation	✓
11	new	[#e.s. $\geq$ 10] close	close			Terminated	✗
12	new	[#e.s. $\geq$ 10] close	close	getMode		Terminated	✗
13	new	[#e.s. $\geq$ 10] close	close	cancel		Evaluation	✗
14	new	[#e.s. $\geq$ 10] close	close	close		Published	✗
15	new	[#e.s. $\geq$ 10] close	close	close	getMode	Published	✗
16	new	[#e.s. $\geq$ 10] close	close	close	evaluate	Published	✗

Table 3: Conformance Test Suite for class ExamManager

However, the conformance tree does not guarantee full implementation of the expected behaviour from the CUT. To achieve this, we must list all *possible sneak paths* (PSPs). These indicate that an illegal message (otherwise valid that should not be accepted given the current state of the class) was received. For each specified state we register a possible sneak path for each message not accepted in that state.

Events	States			
	Open	Evaluation	Terminated	Published
close	✓	✓	✓	PSP
getMode	✓	✓	✓	✓
getExam	PSP	✓	PSP	PSP
submit	PSP	✓	PSP	PSP
cancel	PSP	✓	✓	PSP
evaluate	PSP	PSP	PSP	✓

Table 4: Possible sneak paths in state/event matrix, class ExamManager

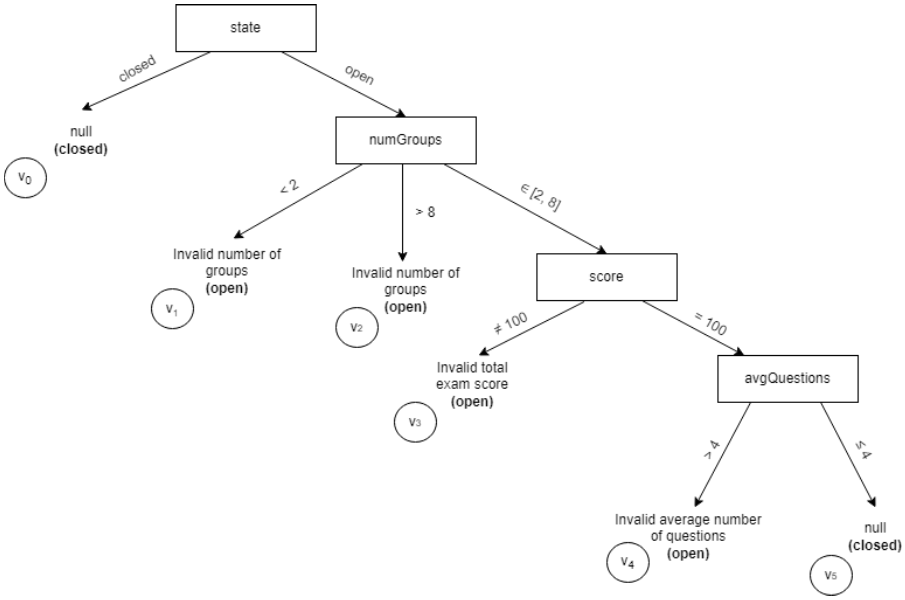
Having all sneak paths registered means we should add these situations to our original test suite. Keep in mind that any series of events that is not described in the project statement is supposed to throw the *InvalidOperationException*.

Run	Test Run/Event Path					Expected State	Throws Exception
	Level 1	Level 2	Level 3	Level 4	Level 5		
17	new	getExam				Open	✓
18	new	submit				Open	✓
19	new	cancel				Open	✓
20	new	evaluate				Evaluation	✓
21	new	[#e.s. $\geq$ 10] close	evaluate			Evaluation	✓
22	new	[#e.s. $\geq$ 10] close	close	getExam		Evaluation	✓
23	new	[#e.s. $\geq$ 10] close	close	submit		Evaluation	✓
24	new	[#e.s. $\geq$ 10] close	close	evaluate		Evaluation	✓
25	new	[#e.s. $\geq$ 10] close	close	close	getExam	Open	✓
26	new	[#e.s. $\geq$ 10] close	close	close	submit	Evaluation	✓
27	new	[#e.s. $\geq$ 10] close	close	close	cancel	Terminated	✓

Table 5: Conformance Test Suite Expansion with PSPs for class ExamManager

3 Method-scope test *validate()*

The *validate()* method belongs to the ExamModel class. It features some if-else logic, for which the method scope testing pattern of ***Combinational Function*** *Test* is the most appropriate.
First, we start by drawing the tree.



The decision tree allows to determine that there will be six variants in total, and their respective expression:

Variant	Expression	Output
v_0	$state = \text{closed}$	null ref, **closed**
v_1	$state = \text{open} \wedge numGroups < 2$	Invalid number of groups, **open**
v_2	$state = \text{open} \wedge numGroups > 8$	Invalid number of groups, **open**
v_3	$state = \text{open} \wedge numGroups \in [2, 8] \wedge score \neq 100$	Invalid total exam score, **open**
v_4	$state = \text{open} \wedge numGroups \in [2, 8] \wedge score = 100 \wedge avgQuestions > 4$	Invalid average number of questions, **open**
v_5	$state = \text{open} \wedge numGroups \in [2, 8] \wedge score = 100 \wedge avgQuestions \leq 4$	null ref, **closed**

With the variants well defined, we can reach the desired test suite by drawing the Domain Test Matrix of each of our variants. Below we show these, for variants v_0 to v_5 (v_3, v_4, v_5 are displayed in landscape mode for easier reading). The strings mentioned in the project statmement were shortened in a logical manner:

- Invalid number of groups → I.N.G.
- Invalid total exam score → I.T.E.S.
- Invalid average number of questions → I.A.N.Q.

Variant 0 Boundaries			Test cases	
			1	
state	== closed	On	closed	
		Off		open
	Typical	In		
numGroups	Typical	In	3	4
score	Typical	In	99	100
avgQuestions	Typical	In	2	3
Expected Result			Accept	V3
Return Value			null	
State			closed	

Table 6: Domain Test Matrix for Variant 0

Variant 1 Boundaries			Test cases			
			1			2
state	== open	On	open			
		Off		closed		
	Typical	In			open	open
numGroups	< 2	On			2	
		Off				1
	Typical	In	0	0		
score	Typical	In	96	97	98	99
avgQuestions	Typical	In	1	2	3	4
Expected Result			Accept	V0	V3	Accept
Return Value			I.N.G.			I.N.G.
State			open			open

Table 7: Domain Test Matrix for Variant 1

Variant 2 Boundaries			Test cases			
			1			2
state	== open	On	open			
		Off		closed		
	Typical	In			open	open
numGroups	> 8	On			8	
		Off				9
	Typical	In	10	15		
score	Typical	In	96	97	98	99
avgQuestions	Typical	In	1	2	3	4
Expected Result			Accept	V0	V5	Accept
Return Value			I.N.G.			I.N.G.
State			open			open

Table 8: Domain Test Matrix for Variant 2

Variant 3 Boundaries			Test cases							
			1		2		3			4
state	== open	On	open							
		Off		closed						
	Typical	In			open	open	open	open	open	open
numGroups	≥ 2	On			2					
		Off				1				
	≤ 8	On					8			
		Off						9		
	Typical	In	3	4					5	6
score	!= 100	On							100	
		Off								99
	Typical	In	52	45	18	72	26	83		
avgQuestions	Typical	In	1	2	3	4	1	2	3	4
Expected Result			Accept	V0	Accept	V1	Accept	V2	V5	Accept
Return Value			I.T.E.S.		I.T.E.S.		I.T.E.S.			I.T.E.S.
State			open		open		open			open

Table 9: Domain Test Matrix for Variant 3

Variant 4 Boundaries			Test cases										
			1		2		3		4			5	
state	== open	On	open										
		Off		closed									
	Typical	In			open	open	open	open	open	open	open	open	open
numGroups	≥ 2	On			2								
		Off				1							
	≤ 8	On					8						
		Off						9					
	Typical	In	3	4					5	6	7	3	4
score	== 100	On							100				
		Off (under)								99			
		Off (above)									101		
	Typical	In	100	100	100	100	100	100				100	100
avgQuestions	> 4	On										4	
		Off											3
	Typical	In	5	10	20	30	40	50	60	70			
Expected Result			Accept	V0	Accept	V1	Accept	V2	Accept	V3	V3	Accept	V5
Return Value			I.A.N.Q.		I.A.N.Q.		I.A.N.Q.		I.A.N.Q.			I.A.N.Q.	
State			open		open		open		open			open	

Table 10: Domain Test Matrix for Variant 4

Variant 5 Boundaries			Test cases 1		2		3		4			5	6
state	== open	On	open										
		Off		closed									
	Typical	In			open	open	open	open	open	open	open	open	open
numGroups	≥ 2	On			2								
		Off				1							
	≤ 8	On					8						
		Off						9					
	Typical	In	3	4					5	6	7	3	4
score	== 100	On							100				
		Off (under)								99			
		Off (above)									101		
	Typical	In	100	100	100	100	100	100				100	100
avgQuestions	≤ 4	On										4	
		Off											5
	Typical	In	1	2	3	4	1	2	3	4	1		
Expected Result			Accept	V0	Accept	V1	Accept	V2	Accept	V3	V3	Accept	Accept
Return Value			null		null		null		null			null	I.A.N.Q.
State			closed		closed		closed		closed			closed	open

Table 11: Domain Test Matrix for Variant 5

4 Method-scope test *addQuestion()*

For the testing of the method *addQuestion()* of the ExamModel class, we decided to use the **Category-Partition** test design pattern. This pattern focuses on building a test suite around the input/output categories of the method.

To start, we must delineate the functions of the method. As specified in the statement, the method's main function is to **add a question to a specified group**. As secondary functions, we can list:

- Making sure the current model's state is still open.
- Making sure that the *groupId* is a valid group in the current model.
- Making sure that one of the aforementioned question's topics match with the group's topic.

With this we can elaborate the table of parameters, categories and choices needed for the application of the Category-Partition Test model. We know that the functions arguments will be used as parameters - *Question q* and *groupId*. Also, the state of the model will play a factor in the execution of the method since a question can't be added to a model if it's closed. Then, we need to account for the topic of the group with *id groupId* - which we will call $t_{groupId}$. We will categorize this parameter in two: one for when this topic agrees with the question's topics ($t_{groupId} \in T_q$); and another for when it doesn't ($t_{groupId} \notin T_q$). Finally, we must also take into consideration the list of current questions that make up groud *groupId* - $Q_{groupId}$ - , since the behaviour of the method depends on if this list is full or not.

Parameter	Categories	Choices
Question q	$q \in Q_{groupId}$	Q_d
	$q \notin Q_{groupId}$	Q_x
groupId	nth element	$n, n \leq G_{\text{Max}}$
	Incorrect value	$n, n > G_{\text{Max}}$
$t_{groupId}$	$t_{groupId} \in T_q$	$T = \{t_1, ..., t_n\}, \exists_{t \in T}, t = t_q$
	$t_{groupId} \notin T_q$	$T = \{t_1, ..., t_n\}, \nexists_{t \in T}, t = t_q$
$Q_{groupId}$	m-elements	$m =$ some $x <$ Max
	Full	Full
state	open	open
	closed	closed

Table 12: Categories and choices for addQuestion()

Table **12** shows choices for the addQuestion() parameters. By performing the cross-product of all the choices we can arrive at a test-suite composed of **32** tests. Below we show the first **30** of these, as requested in the project statement.

	Function Parameters/Choices					Expected Result
	Question	Group Id	Topics of group	Questions of group	Model State	Returned
1	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
2	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
3	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
4	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
5	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
6	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
7	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
8	Q_d	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
9	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
10	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
11	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
12	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
13	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
14	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
15	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
16	Q_d	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
17	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	**true**
18	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
19	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
20	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
21	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
22	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
23	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
24	Q_x	$n = \text{rand}(x),\ n < \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
25	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
26	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
27	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
28	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \in T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false
29	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	open	false
30	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m < \text{Max}$	closed	false
31	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	open	false
32	Q_x	$n = \text{rand}(x),\ n > \#G$	$t_{groupId} \notin T_q$	$m = \text{rand}(x),\ m > \text{Max}$	closed	false

Table 13: Test cases for *addQuestion()*

The table correctly reflects that only one of the 32 cases can result in a question being added to group *groupId*.