**Ensuring Accuracy and Efficiency of Process Management**

**Team 15**

**5/2/19**

**Evan Brittain**

**Roselyn Duong**

**Nicholas Myers**

**California State University, Sacramento**

**1. Problem Statement and a solution**

The software we are testing is a Task Manager - an Android app where users can log and track the completion of personal tasks. The Task Manager uses a SQLite database to store tasks with multiple interfaces (screens) for user interaction. The Task Manager is able to store and plot historical task activity, and prompt for input and displays fields, lists, reports, etc.

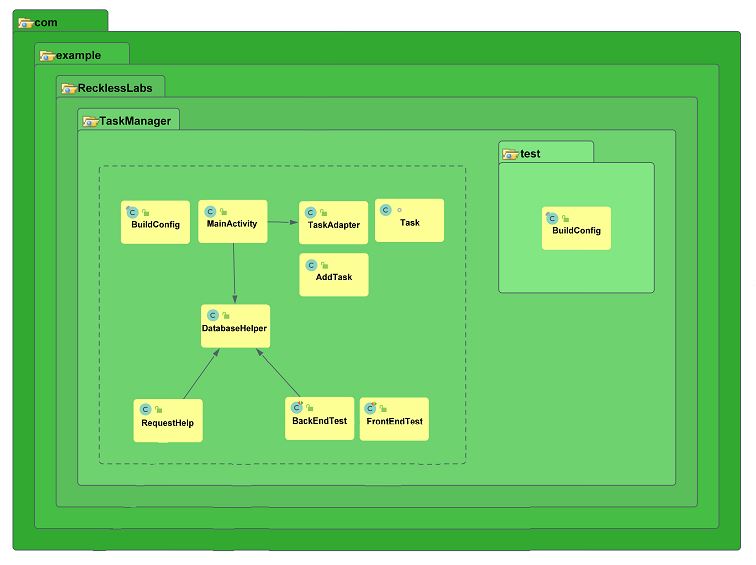
In our project, we will be attempting to resolve issues related to database transactions. Many mobile applications have functionality related to creating, retrieving, updating, and deleting information from databases. In many circumstances, these transactions can be corrupted, or other issues could arise. Say, for example, in a task scheduling application having to deal with payroll, invalid start and end times can result in employees being paid incorrectly for time worked. To do this, we will be validating field entries and database transactions to ensure values can not be entered into the databases correctly.

Our goal is to identify various functional and non-functional issues of the Task Manager app using unit, module, and system tests; as well as analyzing test coverage by testing both the front end and back end. We plan to apply white box testing to buttons and form fields as well as use state diagrams to show the transition between activities and alert dialog boxes. We also plan to apply white box testing for all of the database transactions and use control flow diagrams to test the implementation.

**2. System Description: System diagram with its components & data flows**

The following diagram outlines the applications nested packages, as well as the classes and data flows within the project. The Code Iris plugin for Android Studio was used to generate the diagram.

In the below diagram, each of the program’s packages are indicated by a green folder. Each class is indicated by a yellow box with a © icon, and the data flow FROM and TO is indicated by the arrows pointing from and to classes.



**3. High level Test Specification (Requirements)Function/Non-Functional requirements (Top 5 requirements only) – focus on most important ones)**

We decided to select the requirements to test by determining which functions and non-functions would impact the application most if we were to take that function away. In doing so, we have chosen the following as our requirements.

The system will have a function that allows it to add and remove a task.

The system will have a function that allows it to start and stop the timer for a task.

The system will allows users to view and edit task information.

The system will allow users to view historical task runtimes graphically.

The system will have accurate read and write operations on the database.

**4. Testing Approaches**

**Front end**

User interface testing ensures that the developers mobile application meets the functional requirements and is held to high standard of quality. We aimed to fully test our user interface to verify all front end components behaved correctly. To fully test our app, we chose to automate all front end tests using the Espresso API. Espresso allows developers to programmatically simulate user interactions such as pressing buttons, manipulating text fields, and performing gestures. The automation will perform all actions specified within a test and provide the developer with a summary to show which tests passed and failed within the test suite.

Since our team was familiar with the codebase we chose to use white box testing methods for our user interface tests. To model our tests we used state diagrams to illustrate key transitions the user will experience when using the application. From there we used our state diagrams and Espresso to thoroughly test all paths a user could experience.

**Back End**

Back end / database testing ensures that the developer’s mobile application meets the functional requirements and is held to high standard of quality. We aimed to test the back end to ensure all of the core database operations, including create, update, read, and delete, function wholly and data integrity can be maintained. To fully test our app, we chose to automate all back end tests using the Roboelectric testing API. Robolectric tests run inside a sandbox which allows allows the Android environment to be precisely configured to the desired conditions for each test, isolates each test from its neighbors, and extends the Android framework with test APIs which provide minute control over the Android framework’s behavior and visibility of state for assertions. This testing of the back end will allow us to determine how well the developer implemented the back end functionality, and will give us awareness into possible faults.

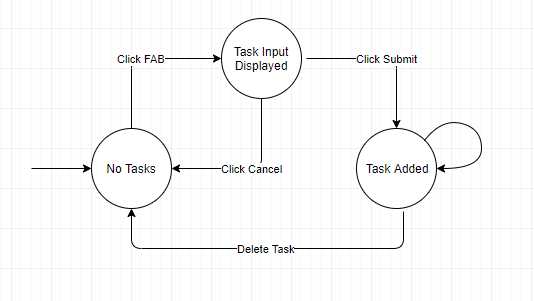
Similarly to our front end, the team was familiar with the codebase, so we chose to use white box testing methods for our database / backend tests. To model our tests, we used control flow modeling, along with respective control flow diagrams to outline the paths taken during program execution. After modeling, we used the Robolectric testing framework to construct our test runner and to carry out our context-based tests.

**5. Test Modeling (including Diagram)**

**Front end**

For our front end testing we chose to focus on four key interactions a user will experience when using the task manager application. These include adding and removing a task, starting and stopping a task, viewing and editing task data, and viewing statistical information for a task. The following diagrams show the transitions a user will experience while performing each.

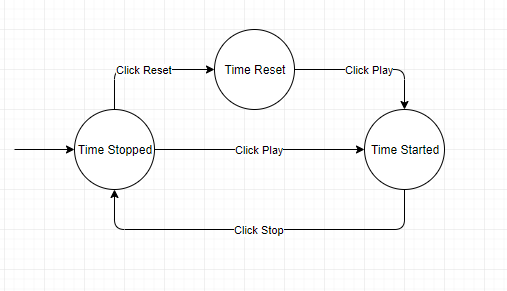
**Add/Remove**



Users are able to add remove tasks from the home screen. The process for adding a task include clicking a floating action button, inputting task information to be stored to the database, and clicking submit. A user can also remove a task by swiping the task left or right on the home screen.

|  |  |  |
| --- | --- | --- |
| **ID** | **Path** | **Expected** |
| 1 | a, b | Task Input Displayed |
| 2 | a, b, a | No Tasks |
| 3 | a, b, c | Task Added |
| 4 | a, b, c, a | No Tasks |

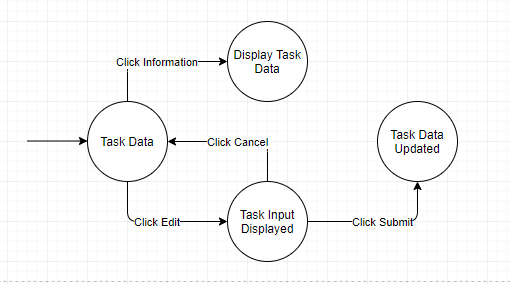
**Start/Stop/Restart**



Once a task is created a user can start, stop, and restart the timer for a task. This involves clicking a button to start and a button to stop. The system must ensure that once start has been clicked it can not be clicked again. Additionally, it must ensure that the time is stored and displayed correctly while running and after being stopped. To restart the timer the user must click the restart button. The system will zero out the time and reset the data stored in the database.

|  |  |  |
| --- | --- | --- |
| **ID** | **Path** | **Expected** |
| 5 | a, b | Time Reset |
| 6 | a, b, c | Time Started |
| 7 | a, c | Time Started |
| 8 | a, c, a | Time Stopped |

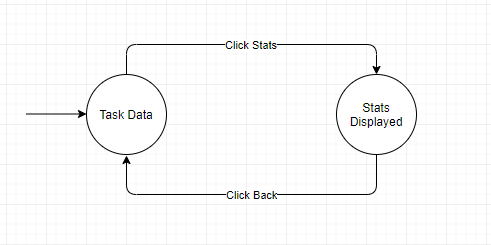
**View/Edit**



A user is able to view and edit the name and description of any task stored on the home screen. To view the task data the user must click the information button. The system should display a alert box containing the accurate name and description of the task. To edit data the user must click the edit button. The system should then display an alert box containing text fields for the user to enter a new name and new description. The new data should then be stored in the database.

|  |  |  |
| --- | --- | --- |
| **ID** | **Path** | **Expected** |
| 9 | a, b | Display Task Data |
| 10 | a, c | Task Input Displayed |
| 11 | a, c, a | Task Data |
| 12 | a, c, d | Task Data Updated |

**View Stats**

****

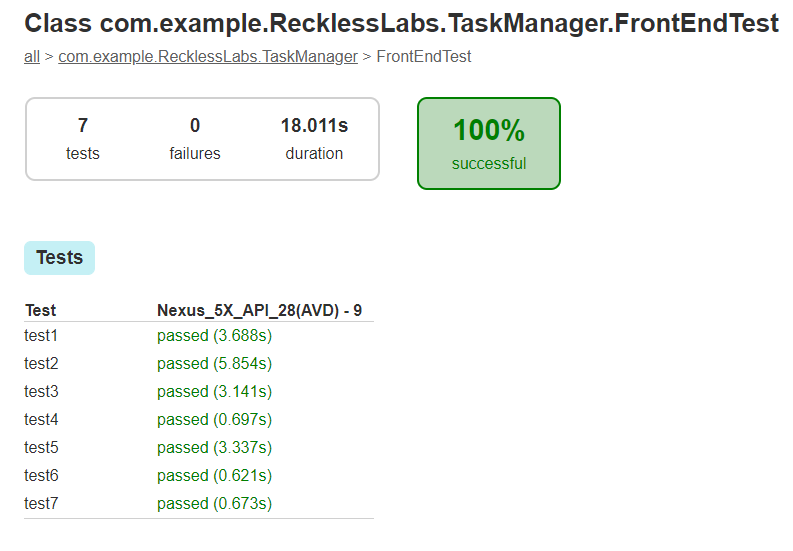
A user must be able to view the historical activity of the task. The system should display a new screen containing a line graph that shows the historic task running times. To enter this screen the user must click the statistics button associated with the task. The system should ensure a new screen appears that displays correct data pulled from the database. The system should also correctly return to the home screen after the user presses the back button.

|  |  |  |
| --- | --- | --- |
| **ID** | **Path** | **Expected** |
| 13 | a, b | Stats Displayed |
| 14 | a, b, a | Task Data |

**Results**

After eliminating redundant tests we were able to reduce our front end test suite to only seven tests. These seven tests all passed and provided ninety percent coverage.

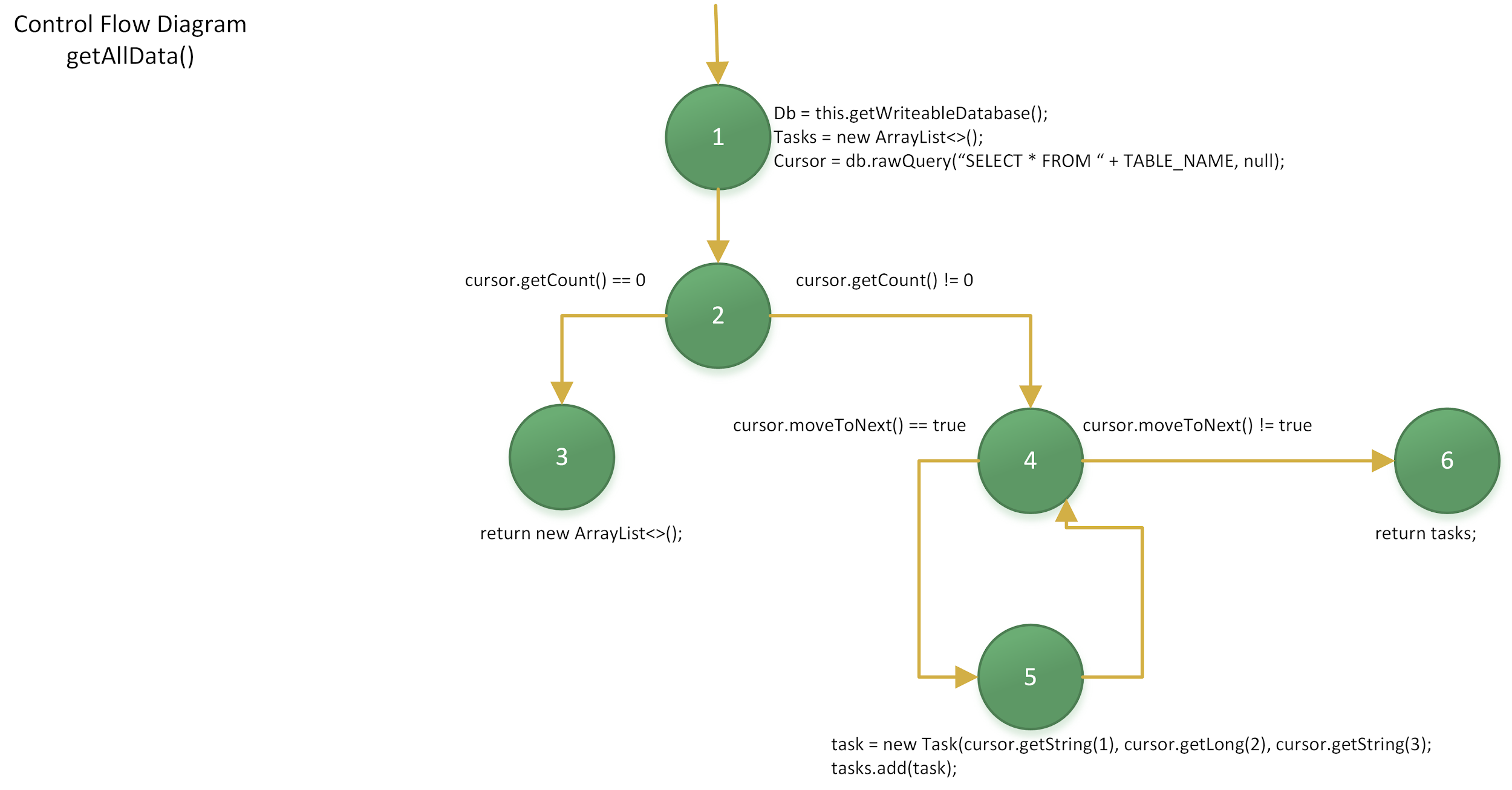
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Path** | **Expected** | **Actual** | **Results** |
| 2 | a, b, a | No Tasks | No Tasks | Pass |
| 4 | a, b, c, a | No Tasks | No Tasks | Pass |
| 5 | a, b | Time Reset | Time Reset | Pass |
| 8 | a, c, a | Time Stopped | Time Stopped | Pass |
| 9 | a, b | Display Task Data | Display Task Data | Pass |
| 11 | a, c, a | Task Data | Task Data | Pass |
| 14 | a, b, a | Task Data | Task Data | Pass |



**Back End**

For our back end testing, we chose to focus on the primary transactions that the application has with the database, and the most crucial functions that act as intermediaries between the user and the data itself. These include CRUD operations - creating, reading, updating, and deleting tasks from the application. The following diagrams show the transitions the system will experience while performing each.

**Get All Data**

****

public ArrayList<Task> getAllData(){  
 SQLiteDatabase db = this.getWritableDatabase();  
 ArrayList<Task> tasks = new ArrayList<>();  
 Cursor cursor = db.rawQuery("SELECT \* FROM "+ *TABLE\_NAME*, null);  
  
 if(cursor.getCount()==0)return new ArrayList<>();  
  
 while (cursor.moveToNext()){  
 Task task = new Task(  
 cursor.getString(1),  
 cursor.getLong(2),  
 cursor.getString(3));  
  
 tasks.add(task);  
 }  
  
 return tasks;  
}

**Basic Blocks**

1. db = this.getWriteableDatabase();

tasks = new ArrayList<>();

cursor = db.rawQuery(“SELECT \* FROM “ + TABLE\_NAME, null);

1. if(cursor.getCount() == 0)
2. return new ArrayList<>();
3. while(cursor.moveToNext())
4. task = new Task(cursor.getString(1), cursor.getLong(2), cursor.getString(3);

tasks.add(task);

1. return tasks;

**Entry Node(s)**

1

**Exit Node(s)**

3, 6

**Control Flow**

1->2

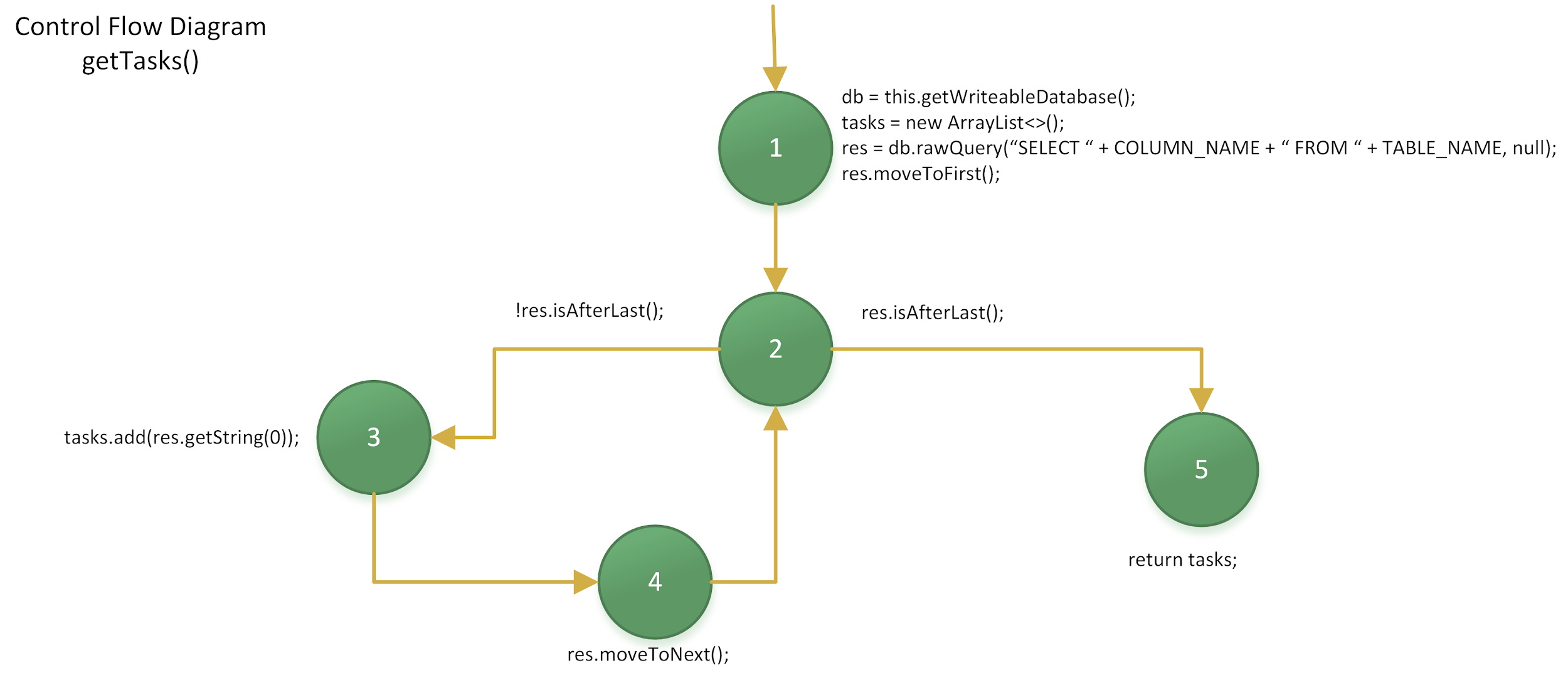
2->3, 2->4

4->5, 4->6

5->4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Input** | **Input Type** | **Expected Output** | **Actual Output** | **Result** |
| 1 | {TABLE\_NAME = “TASKLIST” } | Valid | ArrayList<Task> containing all of our tasks | ArrayList<[Task, Task]> | PASS |

**Get Tasks**

****

public ArrayList<String> getTasks(){  
 SQLiteDatabase db = this.getWritableDatabase();  
 ArrayList<String> tasks = new ArrayList<>();  
  
 Cursor res = db.rawQuery("SELECT "+ *COLUMN\_NAME*+" FROM "+ *TABLE\_NAME*, null);  
  
 for(res.moveToFirst();!res.isAfterLast();res.moveToNext()){  
 tasks.add(res.getString(0));  
 }  
  
 return tasks;  
}

**Basic Blocks**

1. db = this.getWriteableDatabase();

tasks = new ArrayList<>();

res = db.rawQuery(“SELECT “ + COLUMN\_NAME + “ FROM “ + TABLE\_NAME, null);

res.moveToFirst();

1. !res.isAfterLast();
2. tasks.add(res.getString(0));
3. res.moveToNext();
4. return tasks;

**Entry Node(s)**

1

**Exit Node(s)**

5

**Control Flow**

1->2

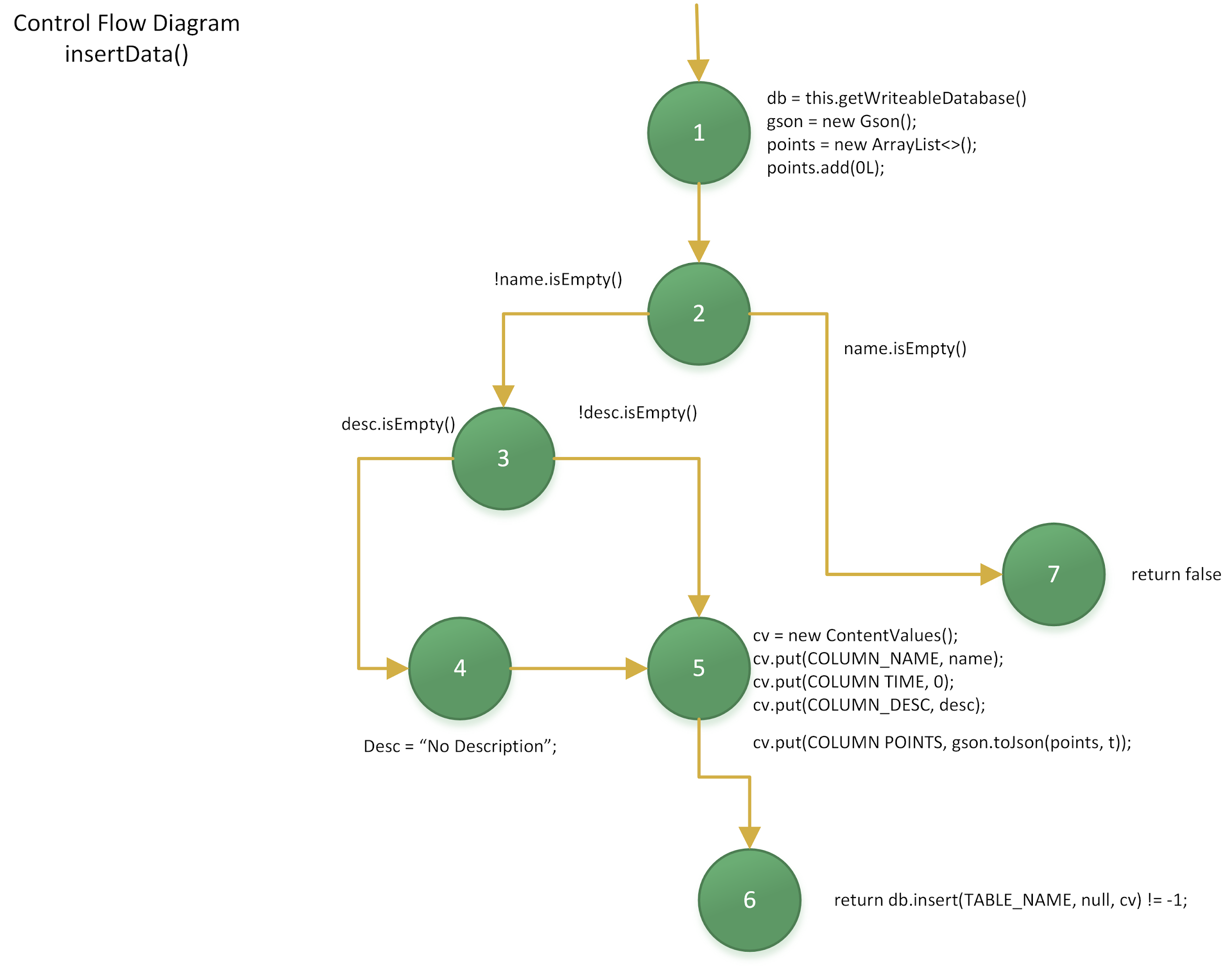
2->3, 2->4, 2->5

3->2

4->2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Input** | **Input Type** | **Expected Output** | **Actual Output** | **Result** |
| 2 | {TABLE\_NAME = “TASKLIST”, COLUMN\_NAME = “NAME”} | Valid | ArrayList <String> containing each of the tasks in the database | ArrayList<String> containing each task | PASS |

**Insert Data**

****

public boolean insertData(String name, String desc){  
 SQLiteDatabase db = this.getWritableDatabase();  
 Gson gson = new Gson();  
  
 ArrayList<Long> points = new ArrayList<>();  
 points.add(0L);  
  
 if(!name.isEmpty()) {  
 if (desc.isEmpty()) desc = "No Description";  
 ContentValues cv = new ContentValues();  
 cv.put(*COLUMN\_NAME*, name);  
 cv.put(*COLUMN\_TIME*, 0);  
 cv.put(*COLUMN\_DESC*, desc);  
 cv.put(*COLUMN\_POINTS*, gson.toJson(points, t));  
  
 return db.insert(*TABLE\_NAME*, null, cv) != -1;  
 }  
 else return false;  
}

**Basic Blocks (nodes)**

1. db = this.getWriteableDatabase();

gson = new Gson();

points = new ArrayList<>();

points.add(0L);

1. if(!name.isEmpty())
2. if(desc.isEmpty())
3. desc = “No Description”;
4. cv = new ContentValues();

cv.put(COLUMN\_NAME, name);

cv.put(COLUMN TIME, 0);

cv.put(COLUMN\_DESC, desc);

cv.put(COLUMN POINTS, gson.toJson(points, t));

1. return db.insert(TABLE\_NAME, null, cv) != -1;
2. else return false

**Entry Node(s)**

1

**Exit Node(s)**

6,7

**Control Flow**

1->2

2->3, 2->7

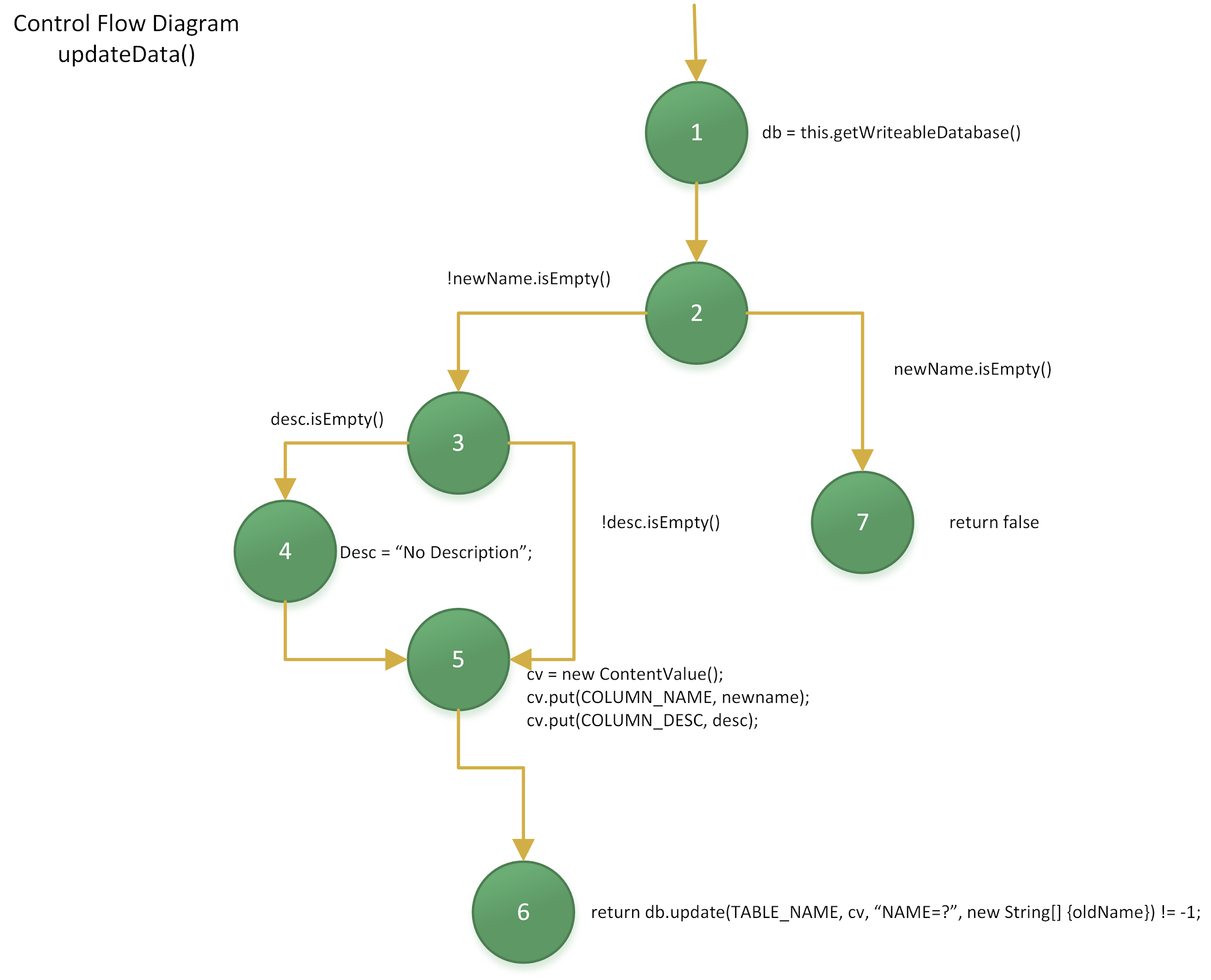
3->4, 3->5

4->5

5->6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Input** | **Input Type** | **Expected Output** | **Actual Output** | **Result** |
| 3 | {name = “task3”, desc = “testDesc3} | Valid task name | true | true | PASS |
| 4 | {name = 4, desc = “”} | Invalid parameter type for task name (task name is an integer) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |
| 5 | {name=”task5”, desc=5} | Invalid parameter type for task description (int, expected str) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |
| 6 | {name = “”, desc = “”} | Task name is empty, desc is empty | false | false | PASS |
| 7 | {name = “task7”, desc = “”} | Task name is valid, desc name is empty | true | true | PASS |
| 8 | {name = “”, desc = “testDesc8”} | Task name is null, desc name is valid | false | false | PASS |

**Update Data**

****

public boolean updateData(String oldName, String newName, String desc){  
 SQLiteDatabase db = this.getWritableDatabase();  
  
 if(!newName.isEmpty()) {  
 if(desc.isEmpty()) desc = "No Description";  
 ContentValues cv = new ContentValues();  
 cv.put(*COLUMN\_NAME*, newName);  
 cv.put(*COLUMN\_DESC*, desc);  
  
 return db.update(*TABLE\_NAME*, cv, "NAME = ?", new String[]{oldName}) != -1;  
 }  
 else return false;  
}

**Basic Blocks**

1. db = this.getWriteableDatabase();
2. if(!newName.isEmpty())
3. if(desc.isEmpty())
4. desc = “No Description”;
5. cv = new ContentValue();

cv.put(COLUMN\_NAME, newname);

cv.put(COLUMN\_DESC, desc);

1. return db.update(TABLE\_NAME, cv, “NAME = ?“, new String[]{oldName}) != -1;
2. return false

**Entry Node(s)**

1

**Exit Node(s)**

6,7

**Control Flow**

1->2

2->3, 2->7

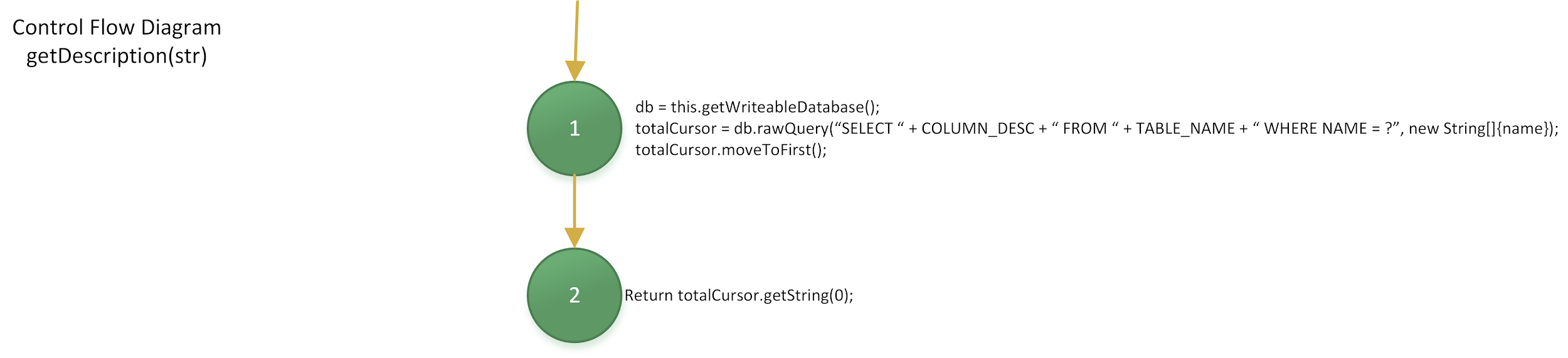
3->4, 3->5

4->5

5->6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Input** | **Input Type** | **Expected Output** | **Actual Output** | **Result** |
| 9 | {oldName = “task3”, newName = “task9”, desc = “testDesc9”} | Valid data | true | true | PASS |
| 10 | {oldName = “”, newName = “task10”, desc = “testDesc10”} | oldName is null | false | true | FAIL |
| 11 | {oldName = “task9”, newName = “”, desc = “testDesc11”} | valid oldName, null newName | false | false | PASS |
| 12 | {oldName = “task12”, newName = “newTask12”, desc = “testDesc12”} | oldName entered could not find existing task | false | true | FAIL |
| 13 | {oldName = “task9”, newName = “task13”, desc = “”} | Valid oldName, valid newName, empty desc | true | true | PASS |
| 14 | {oldName = “task13”, newName = “task14”, desc = 14} | Valid oldName, valid newName, desc is int (invalid param type) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |
| 15 | {oldName = 1, newName = “task15”, desc = “testDesc15”} | oldName is int (invalid param type) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |
| 16 | {oldName = “task13” , newName = 16, desc = “testDesc16”} | oldName is valid, newName is int (invalid param type) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |

**Get Description**



public String getDescription(String name){  
 SQLiteDatabase db = this.getWritableDatabase();  
  
 Cursor totalCursor = db.rawQuery("SELECT " + *COLUMN\_DESC* + " FROM " + *TABLE\_NAME* + " WHERE NAME = ?", new String[]{name});  
 totalCursor.moveToFirst();  
  
 return totalCursor.getString(0);  
}

**Basic Blocks**

1. db = this.getWriteableDatabase();

totalCursor = db.rawQuery(“SELECT “ + COLUMN\_DESC + “ FROM “ + TABLE\_NAME + “ WHERE NAME = ? “, new String[] {name});

totalCursor.moveToFirst();

1. return totalCursor.getString();

**Entry Node(s)**

1

**Exit Node(s)**

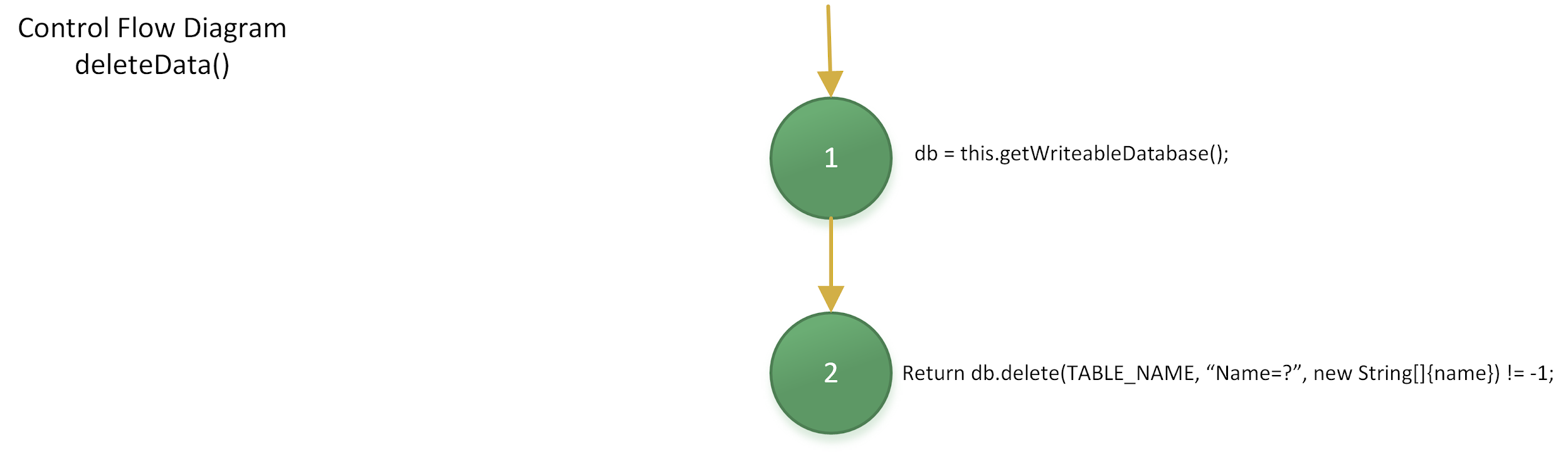
2

**Control Flow**

1->2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Input** | **Input Type** | **Expected Output** | **Actual Output** | **Result** |
| 17 | {name = “task7”} | Valid data | “No Description” | “No Description | PASS |
| 18 | {name = “task18” | No valid task | “No Description” | “No Description” | PASS |
| 19 | {name = 19} | Name is invalid type (str expected, given int) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |

**Delete Data**

****

public boolean deleteData(String name){  
 SQLiteDatabase db = this.getWritableDatabase();  
 return db.delete(*TABLE\_NAME*, "NAME = ?", new String[] {name}) !=-1;  
}

**Basic Blocks**

1. db = this.getWriteableDatabase();
2. return db.delete(TABLE\_NAME, “Name = ?”, new String[] {name}) != -1;

**Entry Node(s)**

1

**Exit Node(s)**

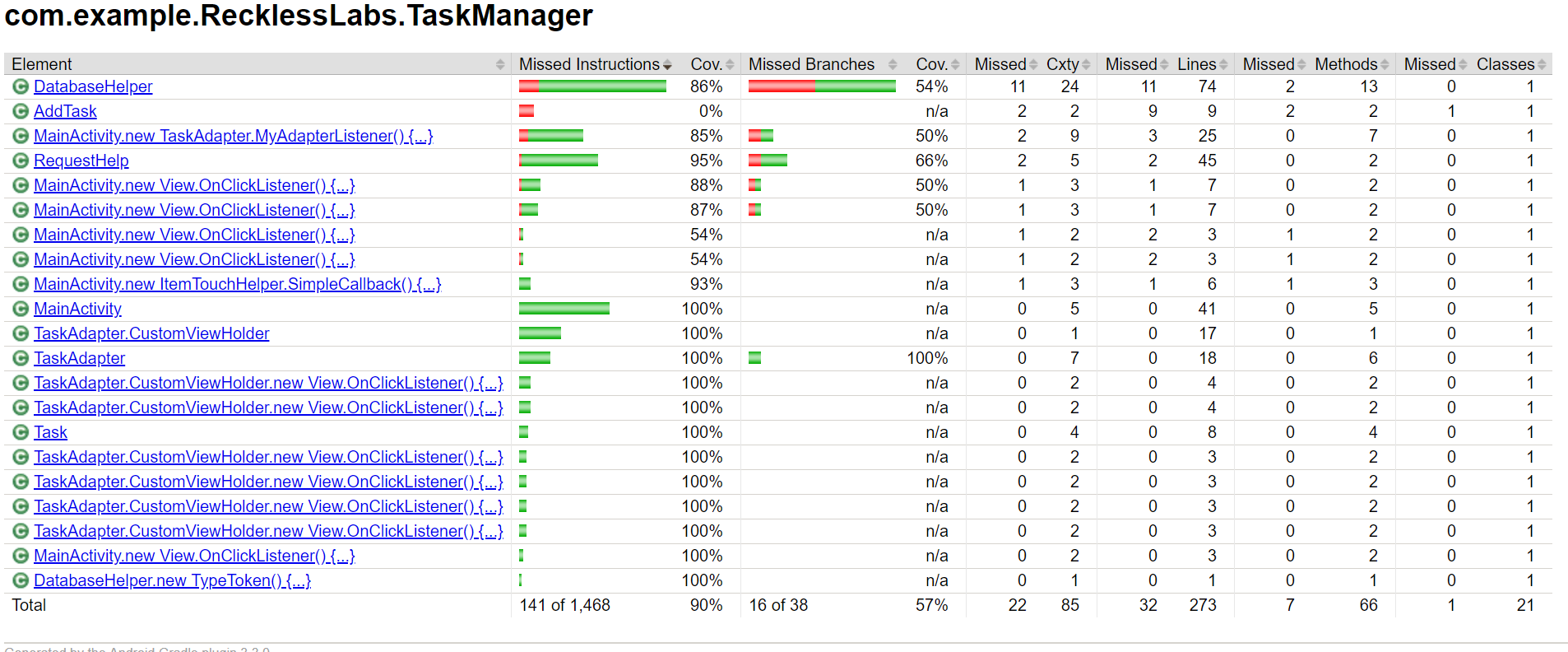
2

**Control Flow**

1->2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Input** | **Input Type** | **Expected Output** | **Actual Output** | **Result** |
| 20 | { name = “task13”} | Valid task name | true | true | PASS |
| 21 | {name = “task21”} | invalid task name/task does not exist | false | true | FAIL |
| 22 | {name = “”} | null task name | false | true | FAIL |
| 23 | {name = 23} | Name is invalid type (str expected, given int) | IncompatibleTypeError (compile fail) | IncompatibleTypeError (compile fail) | PASS |

After all of our tests completed and analyzed, after running our code coverage analysis, we found that we covered 90% of total instructions within our tests, and 86% in the DatabaseHelper class (back end) but only received 54% branch coverage. We only received 54% branch coverage, rather than much higher, because the remaining, untested back end functions were simple helper functions that 1) accepted no inputs and 2) had a single entry and/or exit node.



**7. Conclusion**

Through our endeavors with the final project, we have all been able to learn the importance of quality assurance by testing an application called Task Manager. We were able to apply the knowledge that we have gained in class to ensure that we were able to deliver a software that met project expectations. We first determined the important functional and non-functional requirements of the application, and then moved on to test the application based on the test cases derived from these requirements using white box testing techniques.

As a result of our project, we were able to conclude with a coverage of 90% by implementing the white box testing techniques, and models that we had learned over the course of this term. For this project in particular, we used State diagrams to show the transition between activities and alert dialog boxes, and Control Flow Diagrams to test the backend code. In doing so, we were able to shape our minds in situations of how to code in a safe manner for the future rather than simply worrying about testing after the coding is complete. Therefore, our next steps are to take the knowledge that we have learned from this class and apply it to software that we will create in the future to ensure safe, quality, and well-documented code.