ECSE 429 Software Validation
Term Project
Part B

Group 17
Weige Qian 260763075
Erdong Luo 260778475
Yudong Zhou 260721223
Hanwen Wang 260778557

1. Deliverable summary

In part B of this project, at first our team made a story test suite. Ten user stories are provided in the project manual. To correspond these stories, ten feature files written in Gherkin script were created in the story testing suite. Then, to implement the story test automation, the files containing step definitions are also created. All the step definitions for the test suite are implemented in a way such that the individual steps can be reused for different features.

Besides the coding job, this deliverable also contains a video. The video shows the code can run successfully no matter in which order the feature tests are executed, hence, all the test cases are independent of each other.

2. Structure of acceptance test suite description

For story test suites, as the previous section mentioned, ten feature files written in Gherkin script are provided corresponding to the ten user stories. In every feature file, there are at least three scenario outlines, including one normal flow, one alternate flow and one error flow as acceptance tests. Normal flow means the normal situation and the result that users will get when they interact with the system. Alternate flow means another method that the user can use and will provide the same result as normal flow. Error flow means that the user has taken some incorrect action, or the system is not in an expected state when the action starts, so the user will receive an error message from the server.

For example, if the user wants to modify the description of a task. The normal flow can be directly posting the new description with the id of the task. The alternate flow can be first deleting the existing task and then creating a new task with the new description while keeping other information the same as before. The error flow can happen when the requested task does not exist, so that it is impossible to modify the description.

For different features, they have different normal flows, alternate flows and error flows.

For feature 1 which is to categorize tasks as HIGH, MEDIUM or LOW priority, the normal flow is using the POST API. The alternative solution is that if the user happened to categorize one task with a wrong priority and they can still delete it and put a new category. The error flow will be trying to categorize a task that does not exist. In that case, an error message should be received. For each scenario, we test all the three categories with four different tasks.

For feature 2, to add a task to a course, we first create some tasks and some courses(projects). Then the normal flow will be just add a task to a project's todo list. The alternate flow will be create a new task and add the task to the course's todo list. The task doesn't exist before. The error flow will be trying to add a task to a project that doesn't exist.

For feature 3 which is to mark a task as done on my course todo list, we should first have a class and the class should have an undone task in the todo list. Then the normal flow will be using the post API call with task id to change the donestatus of the task. The alternative flow will be to change the donestatus as well as change the description of the task at the same time. The error flow will be entering the wrong task id. We should receive an error message.

For feature 4, to delete an unnecessary task from a course todo list, the normal flow is using the DELETE API /projects/id/task/id. But the user could also use PUT API /projects/id to delete the relation by changing the task field of the course todo list as an alternate solution. If the user uses the wrong id, there will be an error message from the server, so this is the error flow.

For feature 5 which is to create a course todo list, the normal flow is using POST API /projects with the course title in the data pass to the server. But the user could also write the course description in the data, this is the alternate flow. If the user specifies the project id in the data pass to the server, he/she will get an error message from the server, this is the error flow.

For feature 6, to remove a course todo list, the normal flow is using the DELETE API /projects/id. But this is for the situation that the user knows the id of the course todo list. If the user forgets the id, we will have to first use GET API /projects with the parameter as a filtration to find the projects with the course name and get the id. After that it could repeat the step of the normal flow. This the alternate flow. For error flow, it is the same as feature 4.

For feature 7 which is to query incomplete tasks of a course, for all three flows, first we create a project , a incomplete todo and a complete todo, then we link them together using the relation projects/:id/tasks. In the normal flow, we use GET API/projects/:id/tasks with params = {'doneStatus':'false'} to filter out completed todos and get incomplete todos. In the alternate flow, we still use the same thing as in the normal flow, the difference is that we can pass in the class name to find its corresponding id and then perform the same step. In the error flow, there should be an error message showing when we pass in an invalid class. However, the status

code returned by GET API is 200. This is an old bug discovered in the previous part and it continually affects the following parts.

For feature 8, to query incomplete tasks with HIGH priority of all courses, the logic is quite similar to the previous one. First we define three priority levels, an incomplete task and a complete task, both have HIGH priority. Then we build a relation between priority levels and todos. In the normal flow, we then use GET API categories/:Id/todos together with params={'doneStatus':'false'} to filter out those complete tasks and get incomplete tasks. In the alternative flow, the idea is quite the same as that in normal flow, the only difference is we use a category name to find its corresponding id first before performing the remaining steps. In the error flow, we should get an error when we use invalid category id. However, the status code returned is 200 instead of 404. This is also an old bug discovered in the previous part and affecting current parts.

For feature 9 which is to change priority of the task, the normal flow is using the Delete API to delete the old relation and using the POST API to create a new relation between the new priority and the task. The alternate flow is changing the order of deletion and creation of the new relation between the new priority and the task. There are more than one error flows. The first is about the wrong format on the id and the second is about wrong priorities.

For feature 10, to change the description of a task, the normal flow is using the POST API /todos/id to change the description directly. The other method is using the DELETE API /todos/id to delete the old task and using the POST API /todos to create a new task with a new description, with other information unchanged. The error flows are about non-existing tasks and the wrong id format.

For every scenario outline, the team also created several groups of data for testing, in the form of 'Examples' in Gherkin. Figure shows an example of the testing data.

```
Examples:
  | taskName |     newD |
  |    task1 |   newD1  |
  |    task2 |   newD2  |
  |    task3 |   newD3  |
```

Figure 1. Testing data

3. Source code repository description

Before implementing the story test automation, the team needs to map the variable in the user story to the business logic of the server. As a consequence, we choose to express priorities as different categories with the name of priorities, map the course todo list to projects with course names, and map the tasks as todos with task names.

The source code is written in python because of its strong abilities on sending/receiving requests and parsing json. To parse and execute the Gherkin scripts, the team uses the library Behave. Behave is a semi-official library of

Cucumber which is implemented in python. Since it is a third-party library, before running the test, please make sure you have installed it.

In the main folder. There are 3 types of files: a python script environment.py containing all the controllers needed to change the testing environment, all the feature files and a folder called steps which contains the python scripts of step definitions. Figure 2 shows the structure of the folders.

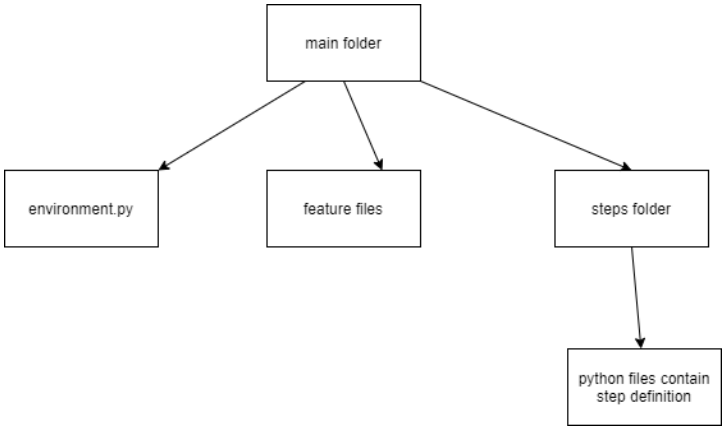

Figure 2: Logic tree of repository

For this part of the project, the environment.py is very important. It contains 2 functions, before_all, after_feature. The before_all function will be booted every time before the whole testing script starts. Its job is testing if the system is running, if not, the whole testing process will be turned off. The after_feature function will be booted every time a testing process for a feature is finished. Its job is to check if any object has been added to the server. If it is true, it will send a delete request to delete these objects. These steps could let all the test cases be independent of each other. Since all the actions we made are not on the original objects, the after_feature function does not check the original objects. Figure 3 shows the implementation of the code in environment.py.

```
HOOK-ERROR in before_all: ConnectionError: HTTPConnectionPool(host='127.0.0.1', port=4567): Max retries exceeded with url: / (Caused by NewConnectionError('<urllib3.connect
ion.HTTPConnection object at 0x0000021A045C1A90>: Failed to establish a new connection: [WinError 10061] No connection could be made because the target machine actively ref
used it'))

ABORTED: By user.
3 features passed, 0 failed, 0 skipped
0 scenarios passed, 0 failed, 0 skipped, 48 untested
0 steps passed, 0 failed, 0 skipped, 0 undefined, 224 untested
Took 0m0.000s
```

```
def before_all(context):
    try:
        r = requests.get('http://127.0.0.1:4567/')
    except ConnectionError as e:
        fail("conditions not met")
```

```
def after_feature(context, feature):
    r = requests.get('http://127.0.0.1:4567/todos') #check if there are any r
    r = r.json()['todos']
    for i in r:
        if not i['id'] == '1' or not i['id'] == '2':
            r = requests.delete('http://127.0.0.1:4567/todos/'+i['id'])
    r = requests.get('http://127.0.0.1:4567/projects') #check if there are ar
    r = r.json()['projects']
    for i in r:
        if not i['id'] == '1':
            r = requests.delete('http://127.0.0.1:4567/projects/'+i['id'])
    r = requests.get('http://127.0.0.1:4567/categories') #check if there are
    r = r.json()['categories']
    for i in r:
        if not i['id'] == '1' and not i['id'] == '2':
            r = requests.delete('http://127.0.0.1:4567/categories/'+i['id'])
```

Figure 3: Snippet of code of environment.py

Sometimes, there must be a common variable for different steps to communicate. For example, when one step creates an object in the server, and the subsequent steps need to know the id of the object. In this case, we choose the built-in variable 'context' in the Behave library. When a step wants to add a common variable, it can just do 'context.variablename = xxx'. Then when the subsequent steps want to access it, they can just call 'context.variablename'. When entering a new scenario, Behave will do a cleanup on context. Therefore, one scenario will not access the variable of another scenario. This also helps us to keep the scenarios independent.

4. Test finding

For this part, the team also met some challenges due to some small problems in the server.

The first finding is that, for some GET APIs like /projects/:id/tasks, when the user uses some non-existing id number, the server should respond an error message to indicate it like 'could find the instance with ***', but the server just sends all the objects of todos in this case. This is not an expected normal behavior. This case is shown in Figure 4 and Figure 5.

Figure 4: Screenshot of one old bug example

Figure 5: Screenshot of status code returned

The second finding is that, when posting a new relationship between a category and a task using the API '/categories/:id/todos' with a task id in the body of the message, the server will not return an error message when the task id does not exist. For example, initially, the system should contain only two original tasks (id = 1 and id = 2). If now we post a relationship between category 1 and a non-existing task 3, the system will not return any error. Then, if we try to get all tasks related to category 1, the result will be an empty list. Figure 6 shows the code we used to create the relationship, and Figure 7 is the response when getting all the tasks associated with category 1 (using GET with '/categories/1/todos').

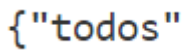

Figure 6. GET response

```
data = {
        'id': '3'
    }
url = 'http://localhost:4567/categories/1/todos'
post_response = requests.post(url, json = data)
```

Figure 7. POST relationship between category and task

Also, different from the unit testing, in the story testing, every test execution depends on several methods, so using global variables is very important. As mentioned in the previous section. The team decided to use the ‘context’ to store the global variables. Since all step definitions are stored in the different python files separately.

5. Test results

Integrated Test:

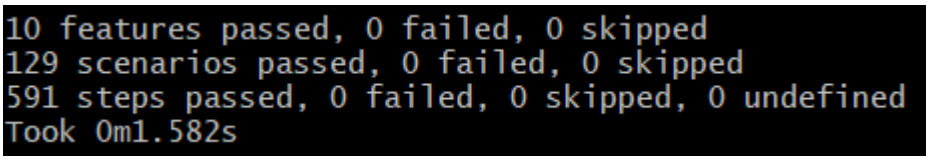

Figure 8: Integrated test results

Figure 8 shows the result when running all the tests together. As can be seen, all 10 features have passed and there is no test fail.