ENPM 611 Project Assignment

Deliverables

The deliverables for all tasks must be submitted by the project deadline. The tasks are as follows:

Category	Task	Deliverable	Points
Planning	GitHub Issue Tracking	A	20
Requirements	Prioritizations Bigraphs	B	20
	User Stories + Priorities	C	20
Testing	API Test Inputs	D	20
	Unit Tests	E	20

Do NOT zip the deliverable but submit them as separate files!

Make sure to study the information provided in the appendix as it provides crucial background to complete this assignment.

Project Context

The University of Higher Learning (UHL) has requested proposals for a system to automate some of the processes involved with giving lectures and tracking student performance. Several proposals were received and after a multi-step review process, your proposal was accepted. The UHL administrators have already identified the scenarios, the system should be supporting and documented them in their Statement of Work (SOW).

You are the lead software engineer on the project and tasked with overseeing the requirements development process. Since you are not familiar with the customer domain, you employ several methods to analyze requirements that are stated in the SOW. You can find the SOW in the Appendix “Statement of Work for the Smarter University System (SUS).”

Part 1: Planning

Planning is an important part of every software project. Before development can truly begin, tasks must be identified and organized. These tasks serve as guidance for team members through the project and as a progress indicator.

In this part of the project, you will plan the tasks associated with the project. You will utilize the GitHub task tracking capabilities to create different types of issues. As you progress through the project, you will update the status of issues. The creation of issues and their maintenance are split up in two separate tasks.

Note, that you will NOT create issues for the requirements analyzed in the next part of the project. The issues are meant to track your progress on the course project tasks.

Task 1: Create issues in GitHub

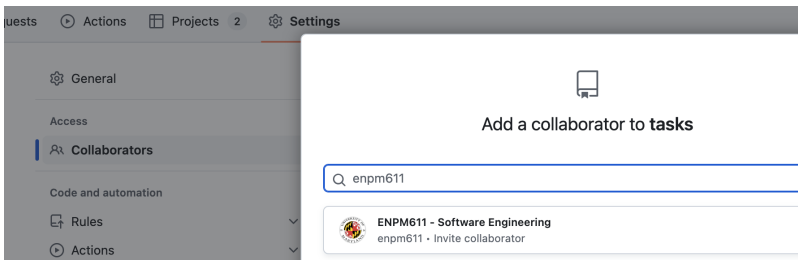
The first part of the planning process is to create issues that reflect the tasks that need to be accomplished as part of the course project. More specifically, you will create a hierarchy of issues that show the hierarchy and how they are derived. You will also establish traceability between parent and child issues.

Before you start creating issues, you will fork the following repository:

https://github.com/enpm611/smarter-university-system

Make sure that the repository is PRIVATE. You will use the repository for issue tracking as well as version control.

Add all members of your team. Go to your repository page. Then click on the “Settings” tab. Then click on “Collaborators” and “Add People”. In addition to all team member, you also need to add the “**enpm611**” and the “**Sameer-Arjun-S**” user:



Once you have set up the repository, you can start creating the following two types of issues:

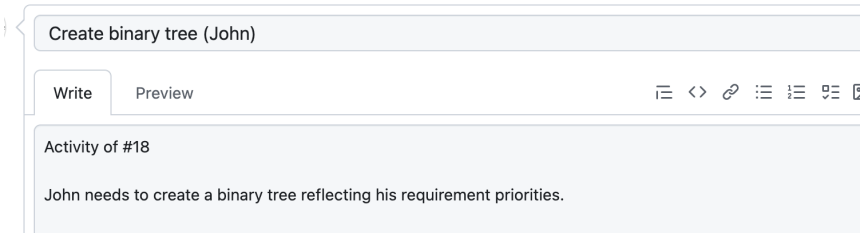
- **MVP**: This project assignment consists of multiple parts. Consider each part a separate MVP. You will create issues for each MVP and apply the “MVP” label.
- **Epic**: Each project task is considered its own epic as it groups a number of sub-tasks. You will create an issue for each task and label those issues “Epic”.

You can create these issues simply by following the structure of the group project assignment. Once you are completed with creating the MPV and Epic issues, you should discuss as a team what activities are involved in accomplishing the tasks of the group project. For each such activity you create the following type of issue:

- **Activity**: You will break down each task into multiple activities. Then, you will create issues for each activity and apply the “Activity” label. You will then assign the team member to the ticket who will complete the activity. You may only assign ONE team member to each activity. If an activity requires more than one team member, you will need to create two separate tickets.

Just as demonstrated in class, you will create labels named “MVP”, “Epic”, and “Activity” to indicate the different types of issues. Note that the labels here are different from the labels we have used in class. The reason is that you are tracking your progress on the group project rather than the development of a software system.

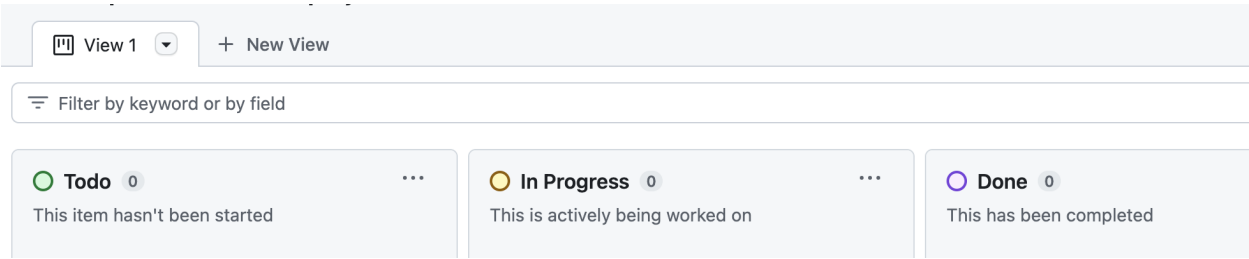
The different issue types (MVP, Epic, Activity) form a hierarchy. Make sure to link from child tickets back to the parent ticket. For instance, if you create an “Activity” issue, start the description of the issue with: “Activity for #13” where #13 is the number of the parent issue. That establishes traceability between parent and child issues.



As you are planning and creating issues, make sure to NOT only include the basic parts, tasks, and sub-tasks outlined in this project document. You also need to identify activities for reviewing artifacts, combining artifacts, and submitting artifacts. You will have to brainstorm early on to identify what these tasks will be. You can add tasks later if you didn’t think of them ahead of time. But the more thorough you are in creating issues in the beginning, the better.

Task 2: Create and maintain scrum board

Throughout working on the course project, team members are expected to update the status of issues to reflect the current state of the project. That status is indicated via an Agile board, which in GitHub is created as a “Project”. Set up one single GitHub project with the default columns to track your tickets:



Add all activities (only activities but no MVPs or Epics) to the project’s Todo column. As you start work on an activity, move the activity into the “In Progress” column. When you completed an activity, move it into the “Done” column and close the issue.

You will be graded on how well you create and maintain GitHub issues. Make sure that you always maintain a true reflection of the project’s status on GitHub. As a bonus, communicate through the issues by leaving comments and instructions to others on your team. After all, issue tracking is supposed to help you stay on task and communicate with the team.

Deliverable A: Send an email to chrisumd@umd.edu and ssarjun@umd.edu with the link to the repository you have forked. The subject of the email should be “Fork for Team <team number>” where <team number> is the number of your team. The body of the email should simply be the URL to the repository.

Part 2: Software Requirements

Now that you have planned your project, you can turn your attention to the stakeholder and the Smarter University System (SuS) they have hired your team to build. As we discussed in class, software requirements are a crucial part of any software development project. In this part of the project, you will create user stories from a requirements description and prioritize those user stories.

Task 3: Create user stories

In an effort to better understand what requirements the customer has raised in the SOW, you decide to rephrase them into user stories. User stories have the advantage that they are smaller, separate and, therefore, more manageable requirements. As a first step, identify epics that you want to organize the requirements by. Then, re-formulate the requirements stated in the SOW as user stories. The user stories must follow the exact format discussed in class. Each of the user stories must be assigned to an epic. Also, each user story must be assigned a unique identifier (e.g., numbers 1...n). Document the user story in the provided spreadsheet template and submit it as part of deliverable E.

Task 4: Prioritize user stories

Now that you have derived the user stories, you need to establish a priority order amongst them. The customer is currently unavailable but you have decided on a prioritization strategy and would like to do a practice run with the software engineers on your team. Apply the binary tree method discussed in class. All members of your team should apply the technique independently and you should combine the priorities of team members into an overall priority for each requirement. More specifically, every team member should take the list of requirements. Using the binary tree algorithm, consider each of the requirements. Starting with the first requirement as the root node, take any subsequent requirement and decide if it is more or less important than the requirements already in the tree. Follow the procedure discussed in class to create a binary tree (one for each team member).

Deliverable B: Initially, you can create the binary tree in any format you choose. However, you have to submit your diagram in Graphviz format. See the section *Creating binary trees in GraphViz format* below for a description of how to generate such a graph. The trees should be submitted in one document named Team_x_Bigraph.txt where “x” is to be replaced with the number of your team. Label the graph with the name of the team member. For instance, for John Doe, the first line of the graph should be “`digraph john_doe {`“.

After you constructed the bigraph, you determine the order of requirements for each of the team members by reading their respective bigraphs and noting the index of each requirement. The

requirement with index 1 is the most important requirement. This will tell you the requirement priorities for each team member. Now, aggregate the priority ranks that each team member gave each requirement into a total priority. That priority ranking reflects the order of requirements in order of importance. The development team can now take this input and take the top few requirements to be implemented.

Deliverable C: Take the spreadsheet with the user stories from Task 3 and add columns to capture the requirements rank for each of the team members as well as the total overall requirements rank. Submit the document in xlsx (Excel) format. The file must be named `Team_x_UserStories.xlsx` where `x` is to be replaced with the number of your team. Each requirement is represented as a row in the spreadsheet. The first column contains the requirements ID, the second column contains the name of the epic, and the third column contains the user story. A template for this spreadsheet is provided in the files for the assignment.

Example spreadsheet:

ID	Epic	User Story	Sam's Priorities	Jill's Priorities	Overall Priority
1	Check	As a truck driver, I want to check how long I have been driving on the current shift so that I know when I can switch drivers.	1	1	1
2	View	As a manager, I want to see what trucks are currently available so that I can assign drivers to them.	2	2	2

Part 3: Software Testing

Your team strongly believes in the test driven development strategy. That means that unit tests should be written before any code is implemented. Developers can then check for correctness of their implementation throughout the development process. Developers know they are done when their implementation satisfies the test cases you wrote.

In this task, you will be determining the inputs that should be provided to the Function under Test (FUT). You will use the category partition and boundary value analysis methods that were discussed in class to find the inputs that have the highest likelihood of uncovering any bugs. You will then implement unit tests to exercise the FUT and provide feedback about the FUT meeting the expected behavior.

Task 5: Create API test inputs

A crucial part of the quality assurance procedures is the testing of the system boundaries. More specifically, the Application Programming Interface (API) through which the user interface and other systems interact with the backend must be robust even when it receives requests that violate the API specification. Your task is to create test inputs for an API endpoint.

The endpoint exposes a new capability of the SUS that has come out of feedback from sprint reviews in which stakeholders expressed the desire to be able to show an activity stream in the sidebar of the SUS user interface. Lecturers and students alike can post to that activity stream as a means to provide status updates or to prompt others to complete tasks.

The primary component of an activity post is the message text, which is specified in the POST body. However, several additional parameters are specified in the URL to that endpoint as shown in this example:

```
POST /activity?flag=IMPORTANT&directed=TRUE&recipients=cdarwin
body: I'd like to discuss the project
```

The list of all input parameters is shown in the table below. Your task is to create test cases that are effective in uncovering defects using the category partition and boundary analysis methods discussed in class. The goal of your test cases is to mimic both nominal use and invalid use of the API endpoint. That is, you should specify test cases where the values of parameters adhere to the specified constraints. You should also specify test cases that slightly deviate from the format but may be reasonably considered valid from a common sense perspective. Finally, you should provide values that are clearly outside the specified format. After all, your goal is to expose any coding mistakes and make sure that when real users call the endpoints, they are not able to break the system.

Parameter Name	Required	Location	Description
text	required	body	The text of the post must have at least one character but may not exceed 100 characters.
directed	optional	url	If set to TRUE, the post is directed towards other users. If this is set TRUE, the set of users must be specified in the recipients parameter.
recipients	optional	url	Comma separated list of user ids where each user ID is a string. This value should not be set if the “directed” parameter is FALSE. The user names must be existing user names (see appendix)
post_time	optional	url	The date and time when the post should be made visible. This allows for delaying the posting to a specific time. The date and time must be specified in the ISO 8601 format with offset (https://en.wikipedia.org/wiki/ISO_8601)
due_time	optional	url	The date and time when the post is due. This can be used to alert viewers of an action that needs to be taken and by what time that action is due. The date and time must be specified in the ISO 8601 format with offset (https://en.wikipedia.org/wiki/ISO_8601)
flag	optional	url	Attaches an icon to the post that provides a quick indicator about the role of the post. Possible values are: “INFO”, “IMPORTANT”,”DUE_SOON”.
lat	optional	url	The latitude of the location this post was sent from. This is set when the user posts from a mobile device and location is enabled. The value must be inside the range -90.0 to +90.0 (North is positive) inclusive.
lon	optional	url	The longitude of the location this post was sent from. The valid ranges for longitude are -180.0 to +180.0 (East is positive) inclusive.

Remember, a test case means you call the API endpoint and you specify values for each of the parameters that are listed above. Example:

```
POST /activity?flag=IMPORTANT&directed=TRUE&recipients=cdarwin
body: I'd like to discuss the project
```

Hence, for a single test case, you must choose what value to provide for each of the input parameters. Of course, some parameters may not have a value at all for some test cases. In the example URL above, only some of the parameters have been specified. In that situation, you leave the value for that parameter empty.

Deliverable D: A spreadsheet where each row represents a test case and each column the value of the respective parameter. The submitted spreadsheet should be named `activities_team_<number>.xlsx`
Where `<number>` is the number of our team.

Note that a test case must specify a concrete value and cannot specify the equivalence class. For instance, the status value in a test case cannot be "> 5 characters" but it should be "I want ice cream!" (which is more than 5 characters). Any spaces in your input text will be interpreted as characters. So make sure to remove any leading or trailing spaces if you don't intend for them to be included.

Also make sure to create test cases that exploit the interaction between variables. There are several parameters that influence each other, such as "directed" and "recipients". Create test cases that adhere to the specified rules and test cases that violate the rules in order to verify the robustness of the API.

An example spreadsheet is provided. You must submit a copy of that example spreadsheet after you added the test cases you developed. Do NOT add additional columns to the spreadsheet or change the formatting. The spreadsheet will be parsed by a machine and changes in formatting may impact that process and result in deduction of points. Do NOT use Excel formulas. Any formulas will be ignored.

Task 6: Write unit tests

Development on the software has started and the first two sprints have been completed. Your team has been developing features at a very fast pace and is worried that the quality of the software may be suffering. You are put in charge of a specific part of the for review. Your task is to review the source code and find instances where the software would crash under specific circumstances. The current state of the application can be found in GitHub:

https://github.com/enpm611/smarter-university-system

You are to write unit tests to cause such an application failure. Your focus in the **quizzes module**. After the developers fix these issues, your test cases can be used to verify that the issues have indeed been resolved.

Your task is to write **3 unit tests** such that each unit test causes the quizzes part of the application to crash. You can find already existing test cases in the `test` directory of the repository. See the `discussions_test.py` for a complete test case. The `quizzes_test.py` has already been created. It has the first test function named `test_expose_failure_01()` already created. You need to implement that function and add two more functions.

Your test cases should cause the source code related to the quizzes capabilities to crash. That means the program will exit prematurely. Each of your test cases must make the program crash. No two test cases can cause a crash in the same line of code. In other words, the crash induced for every one of your test cases must be at a different line in the program.

In the comment of your test case, you must specify the file and the line number where the program is crashing.

You must specify at least one assertion on the expected output if the program did not crash.

Deliverable E: Push the updated test cases to your forked repository and create a Pull Request. You must commit the test cases by the due date of the project.

Appendix

A) Statement of Work for the Smarter University System (SUS)

The communication between instructors and students as well as across students is immensely important for our university. Therefore, we are seeking a software system that can support instructors in delivering their lectures more effectively and efficiently and provides students with the information they need to participate with as little overhead as possible. SUS should provide a host of capabilities that achieve that goal.

The management of classes is an important aspect. SUS should be able to let students sign up for classes if they are available (i.e., if the current enrollment has not met or exceeded the maximum number of students for that class). A class can have one or more named sections (e.g,. “Shady Grove Classroom”), based on the geographical location of the classroom. Some classrooms are located on the main campus while others are located in remote locations. Students in these classes will be able to participate in class via a remote video capability.

SUS should support the student in finding classes to attend. After the student has logged in, the student will be able to see all classes that he/she has not attended in the past. The student should be able to see what the identifier of the class is as well as the title of the class (e.g., CMSC101, Introduction to programming in C). It should also display the name of the lecturer with a link to the class syllabus page. That syllabus page should be maintained by the instructor and should contain the date of each lecture and the topic. Ideally, the instructor enters that information 2 months in advance of the semester starting but should have the option to modify the page during the semester to make updates.

When a student signs up for a course, the course will be updated immediately with the updated student count. An email is sent to the department administrators if a course is 80% or more full. Administrators are then able to check on the status of all courses. That is, they should be able to see an overview of all courses that includes the number of enrolled students and the maximum number of students. The administrators should be able to add and remove students from a class so that students may sign up after the maximum enrollment has been reached or to remove students who decided to drop the class.

Students who do not meet the prerequisites for the class can submit an enrollment request to SUS, which is then sent to both the administrator and the instructor. Both need to confirm that the student is allowed to take the class. Once both approvals have been submitted, the student automatically enrolled and the student is notified.

Administrators should also be able to find information about each of the instructors to include name, email, education (i.e., highest degree, school, and graduation date) as well as a list of courses that that instructor has taught. As the instructor plans lectures, he/she should be able to

access the list of students that are enrolled in the class and be able to view students' name and ID.

To support team assignments, the instructor or the teaching assistant should be able to assign students into groups and name the groups. Students should be able to see what team they are in for a specific class and who the other members of his/her team are.

SUS should also serve as the primary database for grades. That includes grades for exams, assignments and quizzes. After the instructor or teaching assistant has determined a grade, they can enter it into the system either one by one or in bulk (i.e., all grades at once). The instructor or teaching assistant can see the average grade for each assignment, the lowest grade, and the highest grade. They can also see all the grades for specific students over the course of the semester. Likewise, students can see all the grades they have received throughout the semester.

B) Creating binary trees in GraphViz format

Throughout the software development lifecycle, graphic depictions of aspects of the software system aid in communicating critical aspects as well as consensus building and the discovery of solutions. While a bigraph is unlikely to be used for that purpose, it is good practice in creating graphic software descriptions.

GraphViz is a tool for creating graphic depictions of graphs and trees via a specification notation (the dot notation). There are various online versions of GraphViz. You can choose any for this exercise but here are two of them:

- https://dreampuf.github.io/GraphvizOnline
- http://graphviz.it

For this exercise, only the most basic part of the notation is used. Here is an example for a simple tree along with the priorities expressed by that tree:

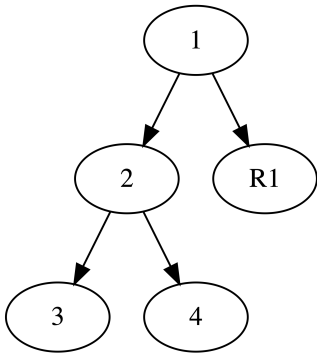
Input Requirements

ID
1
2
3
4

Bigraph

```
digraph john_doe
{
  1 -> 2
  1 -> R1
  2 -> 3
  2 -> 4
}
```

Graph



Ranked Requirements

ID	Rank
1	1
4	2
2	3
3	4

Note: Some nodes will only have one child. In that case, the child node would be centered under the parent node and one could not read the graph to derive the requirements priorities. To avoid that issue, you have to add a placeholder node for the other side. For instance, if node `1` does not have a child on its right, you should add a right node as follows:

```
1 -> R1
```

This adds a right-hand node for node `1` (thus "`R1`"). The order in which you specify nodes impacts where the node is attached (i.e., left or right) so you'll have to pay attention to that. A placeholder for the left-hand child of node `1` would be called "`L1`". The simple graph above contains such a placeholder child node for parent node `1`.

C) Valid user names for API testing

When creating API test inputs, the endpoint expects usernames of users that exist in the system. Valid usernames are the following:

- **`cdarwin`**
- **`tedison`**
- **`einstein`**

Any other username is considered an invalid username.