Project OpenDSA Log Support
Final Report

Victoria Suwardiman

Anand Swaminathan

Shiyi Wei

{vsuward, anand90, wei}@vt.edu

CS5604 Fall 2012

Department of Computer Science

Virginia Tech

12/10/2012

ABSTRACT

The OpenDSA project is an online eTextbook project that includes not only literature but other dynamic content to be used in Data Structures and Algorithms courses. OpenDSA contains exercises of various types to go along with the literature in order to provide automated self-assessment for students. What the research team seeks to do is to collect and log data regarding student interactions with these exercises, logging both the students' performance, such as scores, as well as their interaction with the system, such as timestamps for button clicks. What we did to extend the current OpenDSA project is provide visualizations of the log data in meaningful ways as to be helpful to all users of the system. The OpenDSA Log Support Project, as we have called it, is designed to analyze the log data and provide views for the instructors who teach the course, the students who take the course, as well as for the developers who designed and are continually working on improving the system.

Taking the various forms of log data collected from the students in the DSA course of the Fall 2012 semester, we developed three views: the teacher view, student view, and developer view. Each view displays information that is most useful to its user; for example, a comprehensive table of all students, their scores, and their status in each exercise is the most important data that a teacher will be interested in seeing. We developed our views using the Django web framework that the OpenDSA research team is currently using, pulling our data from the database that all of the data gets logged to. Using this data, we then created online views accessible to those with accounts, namely the instructor, students, and developers.

Some challenges we ran into include the display of and performance of displaying the data in our views. This came up because of the amount of data logged, proving difficult to find efficient and readable ways to analyze and display the data. Though some solutions have been found, because this project is ongoing, future work include optimizing each view, improving the display of each view, as well as adding additional views for each user.

1. Project Overview

OpenDSA [1, 2. 3] is an eTextbook project that seeks to create online tutorial content for Data Structures and Algorithms courses. A major component of OpenDSA is a rich variety of exercises of various types that incorporate automated assessment. Log data are collected regarding student interactions with the exercises, including timestamps for various actions (button clicks, etc.) and performance results (such as score reporting from the assessment system).

The OpenDSA Log Support Project is designed to analyze the log data and present the data interpretation through visualization. The project is to assist the instructors who teach the course, the students who take the class, and the developers who designed the OpenDSA eTextbook.

1.1 Project Effort Summary

There are three views namely student's view, teacher's view, and developer's view. Our team consisted of three members and individual views were worked on in parallel by each team member. Each member provided designs for each view and implemented it. During implementation, we held weekly meetings with our clients: Dr. Cliff Shaffer, Eric Fouh, and Daniel Breakiron to discuss the progress and get feedback. We then performed weekly fixes based on their feedback.

The code repository is available at GitHub: https://github.com/cashaffer/Aalto--. Access to the repository can be requested from one of the OpenDSA research team members, listed above as clients. The OpenDSA eTextbook is deployed online at http://opendsa.cc.vt.edu/.

1.2 Challenges

Some challenges we found in our work on the project come in many forms. Below are a few listed:

- The data in the database was not in the proper format to begin with for processing.
- There was comma separated values in certain fields of the database, which meant that inner joins could not be performed.

- There are an extensive number of exercises to analyze and process.
- The main table of interest, UserExercise, had entries of an exercise for each user stored separately.
- There was redundant testing data in the live database.
- The exercises had to be merged into each module for display, instead of being stored in the database that way.
- The performance of the overall OpenDSA log analysis system is slow.
- The instructor's view has an extensive number of rows and columns for users and exercises, scrolling can cause lack of readability.
- Storage: having an online view is accessible, but not useful for an instructor's recordkeeping.
- More than 20 million button events in one table: document-load, document-unload, forward, backward, etc.

1.3 Contributions

We designed and implemented the OpenDSA log support system which solves the challenges in 1.2. The contributions of our work are:

- We instantiated the OpenDSA log support system that assists students, instructors and developers.
- We improved some performance of the log analysis and visualization of the system.
- Various visualization techniques are applied to better support the data interpretation.

The rest of the report is organized as follows: Section 2 provides the user's manual of the student, teacher and developer views. Section 3 illustrates the developer's manual including architecture, installation.

2. User's Manual

2.1 Student View

2.1.1 Functionality

- A way for students to view their progress and grades in the course.

- The list of modules they are proficient in.
- The list of exercises they are proficient in.
- Student’s score based on the exercises they are proficient in.
- The list of exercises the student needs to be proficient in, to be proficient in a module.
- A list of non-proficient and uncompleted exercises.

2.1.2 Visualization

Module Summary

The view shows the module summary - containing a list of modules where color indicates if the student is proficient, non- proficient or have not started the module. Each module can be expanded to view the exercises it contains. This will help the students view the exercises they are proficient in, in each module and the exercises the student needs to take up to be proficient in the module. The module summary view also displays the total points scored by the user.

Module Summary | Proficient Exercises | Exercises to be taken

Module Summary

Total 16.00

- Sorting
  - SortIntroSumm
- InsertionSort
  - InssortPRO
  - InssortSumm
    Covers : sorting
    Author : OpenDSA
    Description : Insertion Sort Review Questions
    Streak : 10
  - InssortCON1 0.20
    Exercise Details
    Covers :
    Author :
    Description :
    Score Required : 0
    Exercise Type : ss
    User Performance
    User Score : 2
    Total Correct : 1
    Total Done : 1
  - InssortCON2 0.20
  - InssortCON3 0.20

Proficient Exercises

This view displays the list of exercises the student is proficient in. This will help him view the points scored by him in each exercise. This view also shows the details of the exercise and the student’s performance in it.

Module Summary | Proficient Exercises | Exercises to be taken

Total 16.00
- BubsortCON1 0.20
- BubsortCON2 0.20
- mergeImplCON2

Exercise Details

Covers : dsa
Author :
Description :
Score Required : 21 0.20
Exercise Type : ss

User Performance

User Score : 22
Total Correct : 2
Total Done : 2

- mergesortProficiency

Exercise Details

Covers : sorting
Author : OpenDSA
Description : Mergesort Proficiency Exercise
Proficiency Percentage : 90 2.00
Exercise Type : pe

Exercise to be taken

This view shows the exercises the user is not proficient in and the exercises the user has not started. For non proficient exercises, the user can see his performance in those exercises with the details of the exercise. For exercises the user has not started, he/she can view the exercise details by clicking on the exercise.

Module Summary | Proficient Exercises | Exercises to be taken

- SelectionSortElement
- Hash_displayTable

Exercise Details

Covers : hashing
Author : OpenDSA
Description : Hashing JSAV demo: outside
Score Required : 3

User Performance

User Score : 3
Total Correct : 4
Total Done : 6

- ShellsortProficiency
- HashPrinciplesSumm

- edit-KA2
- QuicksortPivotPRO

Exercise Details

Covers : sorting
Author : OpenDSA
Description : Quicksort Find Pivot Exercise
Score Required : 3

- BubbleSortElement
- SelectionSortSumm
- InssortPRO

2.1.3 Technical Details

The controller (in Model View Controller) code for the student view is present in the file Aalto/aaltoplus/opendsa/views.py. ‘module_list’ is the function that renders the student view. The data for the controller comes from file models.py which contains a

set of classes for each table in the database. Each class has a set of member functions for performing certain utility operations (example: to check if the user is proficient in a exercise). Since the model fetches all the data from the corresponding tables, filtering of data at the controller level (views.py) proves to be a costly affair. For example, to extract all the entries for a particular user from table 'user_exercise', initially we used the filter function *userExercise.objects.filter* which does the filtering after the data is returned from the database. To improve the performance the controller calls static member functions of respective classes in models.py. These methods use cursor to execute appropriate stored procedure and return the result. o execute a query we use django.db.connection.connection.cursor() to get a cursor object. Then, we execute the query through statement cursor.execute(sql, [params]). Static methods inside the models and procedures are not the same. Static methods inside the class in models will call the stored procedure available in the database.

The use of stored procedures instead of using data from the models directly helped in reducing the time taken for the page to load. The page now loads in about a second as compared to 20 seconds that it took when data was used from the django models.

The classes used in **views.py** are

- Useruiexec (Fields : exercise, prof, points)
  - This class renders the exercises under the modules.
  - exercise : Contains the exercise object
  - prof : indicates the proficiency - if the person is proficient or not.
  - points : indicates points scored by the user. If a student is proficient in an exercise then based on the exercise type, value is set to this field.
- Profuiexec (Fields : userexercise, points)
  - This class renders the proficiency exercises.
  - userexercise : contains the userexercise object which has all the data from the table user_exercise.
  - points : indicates points scored by the user. If a student is proficient in a exercise then based on the exercise type, value is set to this field.

- Useroutputmodule (Fields : name, covers, authors, prof, countexec, prerequisites, userexecs)
    - This class renders the modules in the 'Module Summary' tab. It contains information about the module along with the exercises under it.
    - name: name of the module
    - covers: concepts which the module covers
    - authors: various authors of this module.
    - prof: this field is used to indicate if the module is proficient or not.
    - prerequisites: contains details of the modules which must be completed before starting the current module.
    - userexecs: a array of exercises which the module contains.
    - countexec: contains the count of number of exercises inside a module.

Function module_list(request):

The function module_list takes the request object as a parameter, reads the session id from the cookies and uses the session id to get the information about the logged in user. The cookies field in the request object is a dictionary. So request.cookies["sessionid"] contains the session id. With the user id the function fetches the data from the database. The function first fetches the list of all modules and the list of all module entries for the logged in user and maps them to decide the proficiency of the user in various modules. It then fetches all the exercises and inserts apporpriate exercise inside each 'Useroutputmodule' object. The function then constructs objects for proficient, nonproficient exercises, and exercises to be taken. These objects are then written out to the response stream and passed to the template. Thus the response stream now has an array of 'Useroutputmodule' objects which corresponds to the tab 'Module Summary', a array of 'Profuiexec' objects corresponding to proficient exercises, and an array of 'userexercise' and 'exercise' objects for non-proficient and proficient exercises respectively.

The template file: **module_list.html**

This file contains the html code for the student view. It reads the entries from the objects written out to the response and constructs the DOM elements. The interactions inside the page such as switching tabs or expanding modules/exercises are achieved through jQuery.

2.2 Teacher View

The teacher view is intended for instructors of Data Structure & Algorithms courses to view the progress and information relevant to their students in the course. Users of the teacher view then will typically be instructors or other administrative staff of a particular DSA course section.

The live site for all of the views developed in this project for the OpenDSA research team is accessible at: http://opendsa.cs.vt.edu/. Users are required to have a login to be able to access the teacher view as the pages contain sensitive information such as students' usernames and grades. Logins should be provided with the OpenDSA e-Book. From there, each view corresponds to a set of pages. The available teacher views are:

- /exercise_summary/
- /export_csv/
- /progress_summary/

2.2.1 Visualization

Exercise Summary

The exercise summary page displays a comprehensive table of all students and all exercises in the OpenDSA e-Book. The first column contains the username of students, not actual student names, followed by a column of scores for each student. Follow columns are color-coded based on status per exercise: green for proficient, red for not yet proficient, and gray for not yet started. The headers of both the rows and columns are fixed, allowing for easy scrolling and readability. The table is dynamically populated from information in the database so each instructor's teacher view will be their own students' information.

The exercise summary is intended for instructors to easily view their students' scores

as well as how they're doing in each exercise. It allows instructors to easily see which students are struggling and also overall, how students are doing.

Below is a sample screenshot of the exercise summary page from a test database:

Exercise Summary

Export to CSV

Username	Score	SortIntroSumm	InssortPRO	InssortSumm	ShellSortSublist	ShellsortSumm	ShellsortSeries	ShellsortProficiency	ShellsortSublist	BubsortSumm	B
student1	10										
student2	5										
student3	5										
teacher1	1										
student5	5										
student6	5										
student7	5										
student8	5										
student9	5										
student10	5										
student11	5										
student12	5										
student13	5										
student14	5										
student15	5										
student16	5										
student17	5										

Username Score SortIntroSumm InssortPRO InssortSumm ShellSortSublist ShellsortSumm ShellsortSeries ShellsortProficiency ShellsortSublist BubsortSumm B

Export CSV

The exercise summary page also links to the export csv page, which allows for users to export the exercise summary table to a CSV file. Upon clicking the link, the user is prompted to download the CSV as an attachment. The CSV file displays the same information as the exercise summary page except instead of color coding, the cells display 0, -1, 1 for not started, not proficient, and proficient statuses, respectively.

The export to csv functionality is intended to allow instructors to more easily store information relevant to their students. Also, with a CSV file, they are able to filter and manipulate the data for their individual analysis.

Below is a screenshot of a CSV file corresponding to the previous screenshot's display:

Username	Score	SortIntroS	InssortPR(	InssortSur	ShellSortS	ShellsortS	ShellsortS	ShellsortF	ShellsortS	BubsortSu	BubsortPF	SortCompareSumm
student1	10	0	0	0	0	0	0	0	0	1	0	1
student2	5	0	0	1	0	0	0	0	0	0	0	0
student3	5	0	1	0	0	0	0	0	0	0	0	0
teacher1	1	0	0	0	0	0	0	0	0	1	0	0
student5	5	0	0	0	0	0	0	0	0	1	0	0
student6	5	0	1	0	0	0	0	0	0	0	0	0
student7	5	0	0	0	0	0	0	0	0	0	0	1
student8	5	0	0	0	0	0	0	0	0	1	0	0
student9	5	0	0	0	0	0	0	0	0	0	0	0
student10	5	0	1	0	0	0	0	0	0	0	0	0
student11	5	0	0	0	0	0	0	0	0	1	0	0
student12	5	0	0	0	0	0	0	0	0	0	0	0
student13	5	0	0	0	0	0	0	0	0	0	0	0
student14	5	0	0	0	0	0	0	0	0	0	0	1
student15	5	0	0	0	0	0	0	0	0	0	0	1
student16	5	0	0	0	0	0	0	0	0	1	0	0
student17	5	0	1	0	0	0	0	0	0	0	0	0
student18	5	0	0	0	0	0	0	0	0	0	0	1
student19	5	0	1	0	0	0	0	0	0	0	0	0
student20	5	0	0	0	0	0	0	0	0	1	0	0

Progress Summary

The progress summary page displays similar information but in a different way. The table displays the students who fall in each of the three statuses for each exercise by their username.

The progress summary page is intended to allow instructors to more easily see how each exercise in the OpenDSA e-Book is fairing with the students. It might also allow instructors to gear their lectures based on these results, as well as to notify instructors of which students need to finish or get started on a particular exercise.

Below is a screenshot of the progress summary page from a test database:

Progress Summary

Exercise	Proficient	Inproficient	Not Started
SortIntroSumm			student1 student2
InssortPRO			student1 student2
InssortSumm	student2		student1
ShellSortSublist			student1 student2
ShellsortSumm			student1 student2
ShellsortSeries			student1 student2
ShellsortProficiency			student2 student1
ShellsortSublist			student1 student2
BubsortSumm	student1		student2
BubsortPRO			student1 student2
SortCompareSumm	student1		student2

2.2.2 Technical Details

Using the Django web framework for the development of views for the OpenDSA project, each view uses the Model-View-Controller architecture. In the repository: /Aalto--/aaltoplus/, the OpenDSA data is housed in /opendsa/.

The models used for the teacher view are defined in models.py. The model classes used by the teacher view are:

- Exercise
- BookModuleExercise
- UserData
- UserSummary

The views for the teacher view are defined in /Aalto--/aaltoplus/templates/teacher_view/. The views are html files that the controller uses to render data from the models. The views included for the teacher view are:

- exercise_summary.html
- progress_summary.html

These pages display the dynamic data from the models in different ways. The

exercise_summary page is of more detail, using Javascript and a JQuery plugin stored in /Aalto--/aaltoplus/assets/ to display an advanced table showing each student's score and status in each exercise. The progresss_summary page simply displays a table with columns for the three statuses for each exercise, along with which students fall in which category.

The controller for the teacher view is defined back in /opendsa/ in views.py. The relevant classes for the teacher view are:

- userValue
  - holds user's name, user's points, and list of values corresponding to user's statuses in the same order as exercises
- exerciseProgress
  - holds exercise's name and three lists of users in each status: proficient, not yet proficient, and not yet started

While the view definitions for the teacher view are:

exercise_summary

defines the Exercise Summary view. Uses the BookModuleExercise model to build a list of exercises in the order they appear in the book. Uses the UserData and UserExercise models to build a list of userValues that hold unique users, each with corresponding values for each user's statuses. Renders exercise_summary.html view using list of userValues to display comprehensive table.

export_csv

defines the Export CSV view. Uses same models as exercise_summary and renders information in the same way, except instead of rendering the exercise_summary.html view, it writes to a CSV file. This function is invoked upon clicking the link to "Export to CSV" from the exercise_summary.html page.

progress_summary

defines the Progress Summary view. Uses the BookModuleExercise model to build a list of exercises in the order they appear in the book. Uses the UserExercise model to build a list of exerciseProgresses that hold the exercise name and then the lists of

users that fall in each status. Renders the progress_summary.html view using list of exerciseProgresses to display table.

2.3 Developer View

2.3.1 View Summary

The developer view provides graph visualization of the OpenDSA system usage patterns by individual students. The summaries of each student's behavior on modules and exercises are visualized in different levels of details including exercises proficiency distribution, module loading distribution, exercise time distribution, exercise steps details, etc.

The developer view is mainly intended for two audiences: (1) the developers who designed and implemented the system use the developer view to find bugs or improve the functionalities; (2) the researchers who are interested in studying the usage of an interactive online e-book system use the developer view to summarize the patterns of how the system is being used by the students.

The developer view is accessible at: http://opendsa.cc.vt.edu/developer_view/.

2.3.2 Flow of Developer View

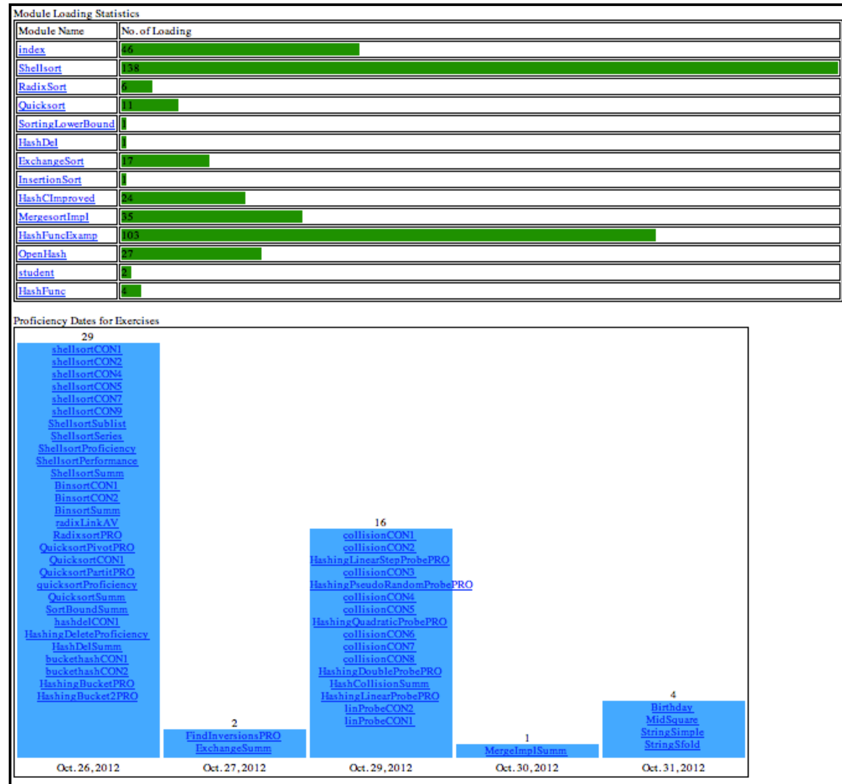
The page /developer_view/ lists all the students taking the Data Structure & Algorithms course:

Student List

id	user name	email
2	monday	ericfouh@gmail.com
3	monday1	ericfouh@gmail.com
4	test_user2	test_user2@algoviz.org
5	shaffer	shaffer@vt.edu
6	s1	shaffer@vt.edu
7	test_user	test_user@algoviz.org
8	s2	sh@vt.edu
9	test_user3	test_user3@algoviz.org
10	fabricemarcelin	fabricemarcelin@vt.edu
11	test_user4	test_user4@algoviz.org
12	tuesday	root@opendsa.com

Clicking on the user name directs the users to the student's activity summary page

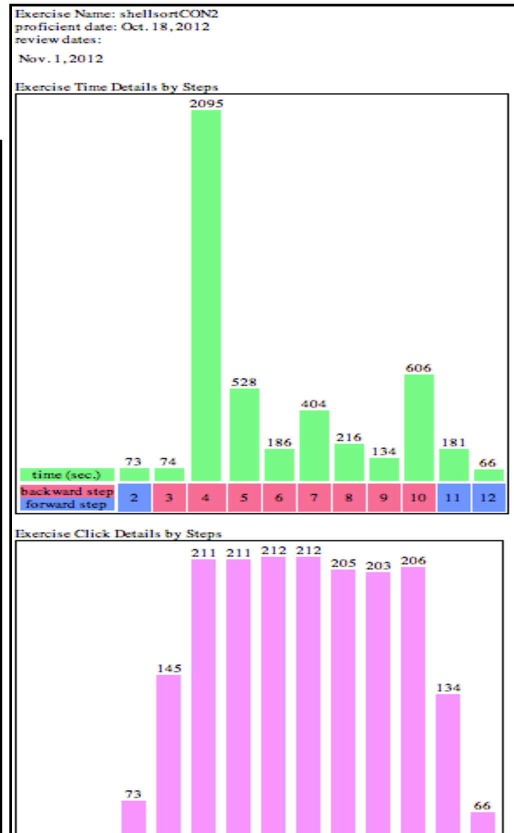
/developer_view/student-exercise/(student_id)/ which includes module loading statistics and exercises proficiency dates distribution graphs (details in the following section):



The links in the module loading statistics lead to the exercise list page /developer_view/exercise_list/(student_id)/(module_id)/ and the links in the proficiency dates distribution graph direct to the exercise summary & details page /developer_view/exercise_list/(student_id)/(exercise_id)/, respectively:

Exercise List

- ShellsortProficiency
- shellsortAV
- shellsortCON1
- shellsortCON2
- shellsortCON4
- shellsortCON5
- shellsortCON7
- shellsortCON9
- ShellsortPerformance
- shellsortCON3
- shellsortCON6
- shellsortCON8
- ssperform



The links on the exercises in the exercise list page also direct to theexercise summary & details page /developer_view/exercise_list/(student_id)/(exercise_id)/.

2.3.3 Visualization

Module Loading Statistics

The student's activity summary page contains a bar graph showing the student's module loading activity statistics. This view lists all the modules in the course and, for each module, shows the total number of loads (page opens) the student performed. The bar graph is formed according to the number of loads of each module.

Developers to determine which modules are more interactive or not can use this module loading statistics view. Modules loaded more often (except for the *index* module) suggest the student wants to get information from the module pages more frequently, thus other modules not frequently loaded should be improve to contain more useful information.

Below is the screenshot of a sample student's module loading statistics view:

Module Loading Statistics

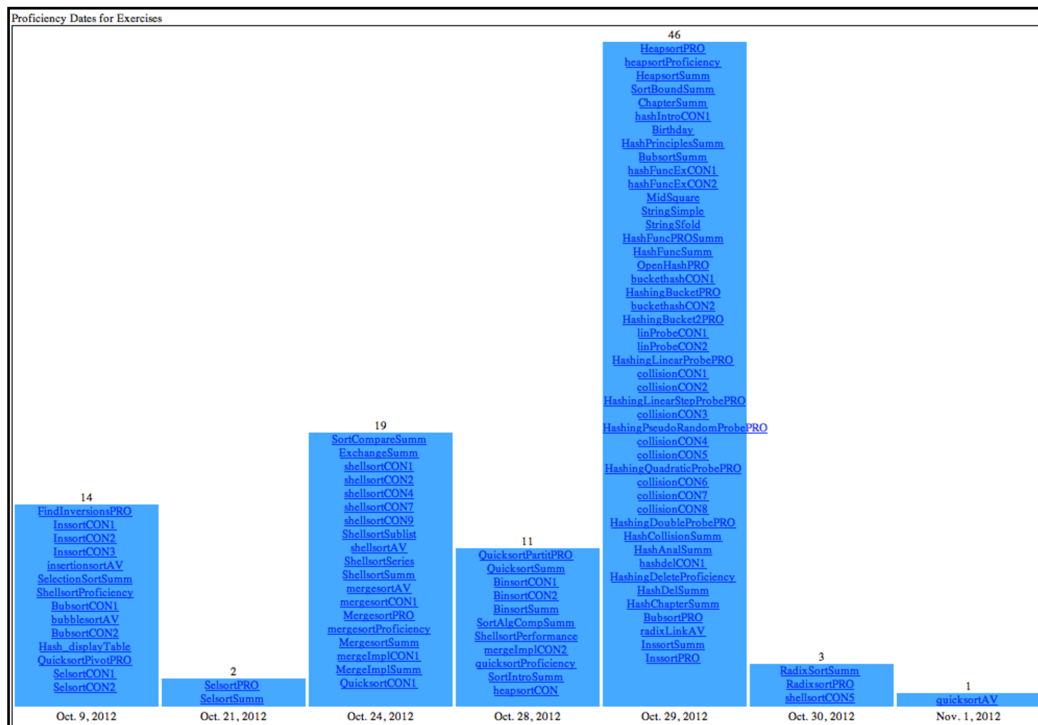
Module Name	No. of Loading
Sorting	25
InsertionSort	43
BubbleSort	91
index	169
SelectionSort	19
Quicksort	158
SortCompare	4
Shellsort	151
Mergesort	54
MergesortImpl	15
RadixSort	149
SortingEmpirical	27
SortingLowerBound	11
Heapsort	61
InsertOpt	43
Bibliography	4
SortSumm	6
HashIntro	4
HashFunc	25
HashFuncExamp	54
student	44
OpenHash	8
BucketHash	8
HashCSimple	4
HashCImproved	24
HashAnal	4
HashDel	12
HashSumm	4
ExchangeSort	2
BinSort	16

Exercises Proficiency Dates Distribution

The student's activity summary page contains another bar graph showing the student's proficiency dates of the exercises. This view shows all the exercises the student achieved proficiency and the bar graph visualizes the distribution of the dates these exercises become proficient. The exercises shown in the bars are ordered ascendant in proficient time.

The researchers/developers can go over this view to understand the distribution of dates the student finishing exercises. A graph that spreads widely suggests the student do the exercises all over the semester; while a graph that concentrates on particular dates (e.g., dates close to deadline) suggest the student intend to finish the exercises right before deadline. The developers use this view to summarize the student's behavior on finishing the exercises and, by comparing to the final grades, guidance can be applied in the future to force better patterns on finishing the exercises.

Below is the screenshot of a sample student's exercises proficiency dates distribution view:



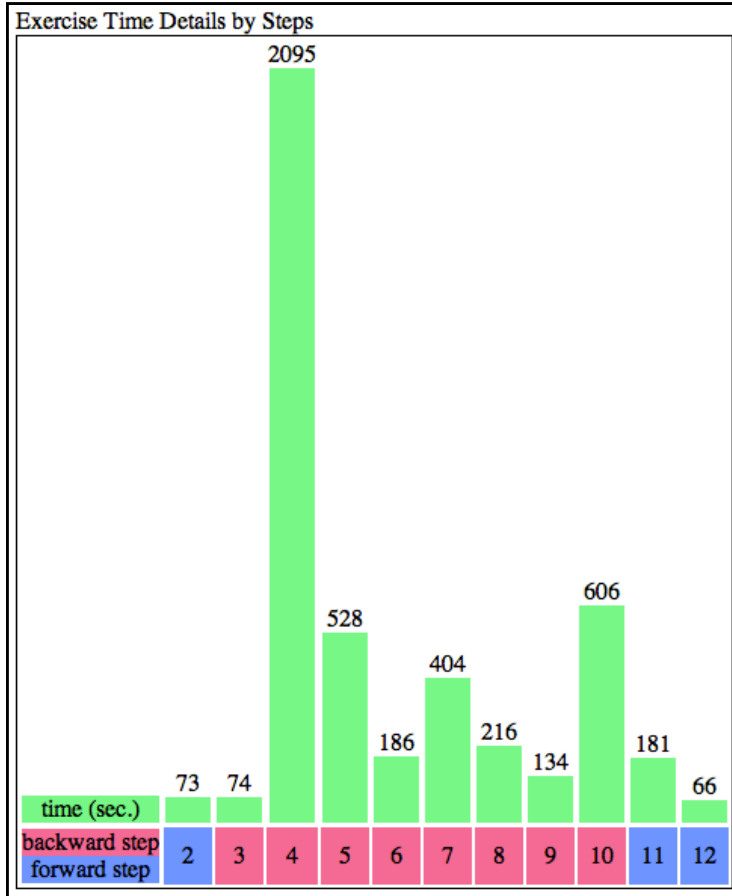
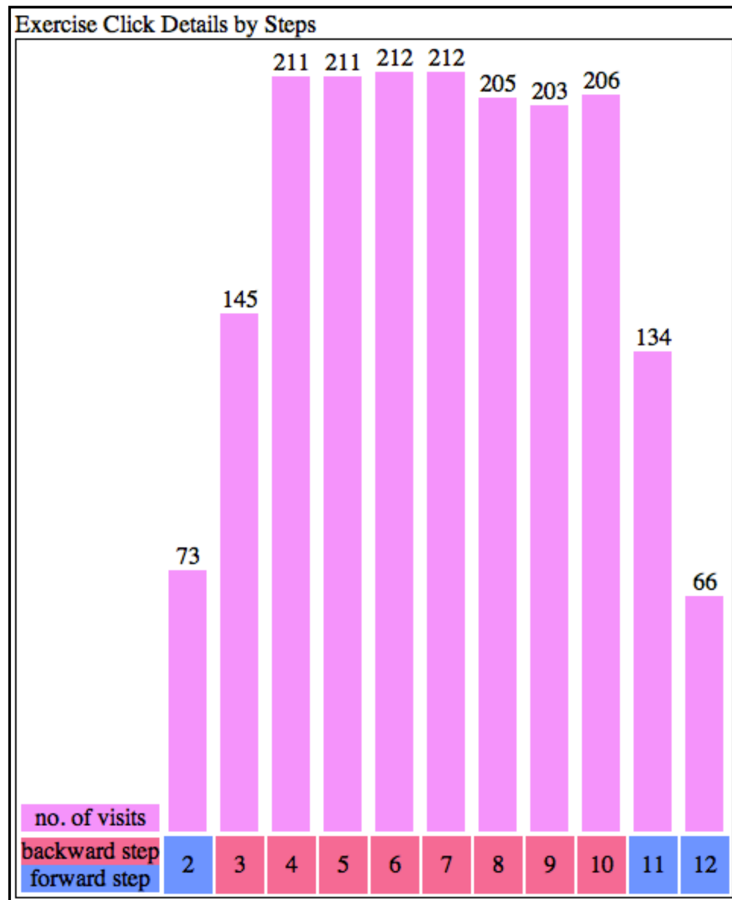
Exercise Steps Time& Visits Distribution

The student's exercise summary & details page contains a bar graph showing the time the student spent on each step of the exercise. This view shows the exercise step numbers and the total time (in sec.) spent on each step. The bar graph is formed according to the time and the step numbers in red indicating the steps were once visited backward (via. the backward click button).

The student's exercise summary & details page also contains a bar graph showing the number of times the student visited each step of the exercise. This view shows the exercise step numbers and the total number of visits on each step. The bar graph is formed according to the number of visits.

The developers use these two graphs combined to summarize the student's behavior on the exercise. Steps with more time spent and/or visits suggest that these are the difficult part of the exercise so that more instructions can be provided in the system or in class. Depending on the time and visits of the exercise, the researchers can categorize if the student does the exercise for grade (less time and visits) or for information (more time and visits).

Below are the screenshots of sample exercise steps time & visits distribution views:

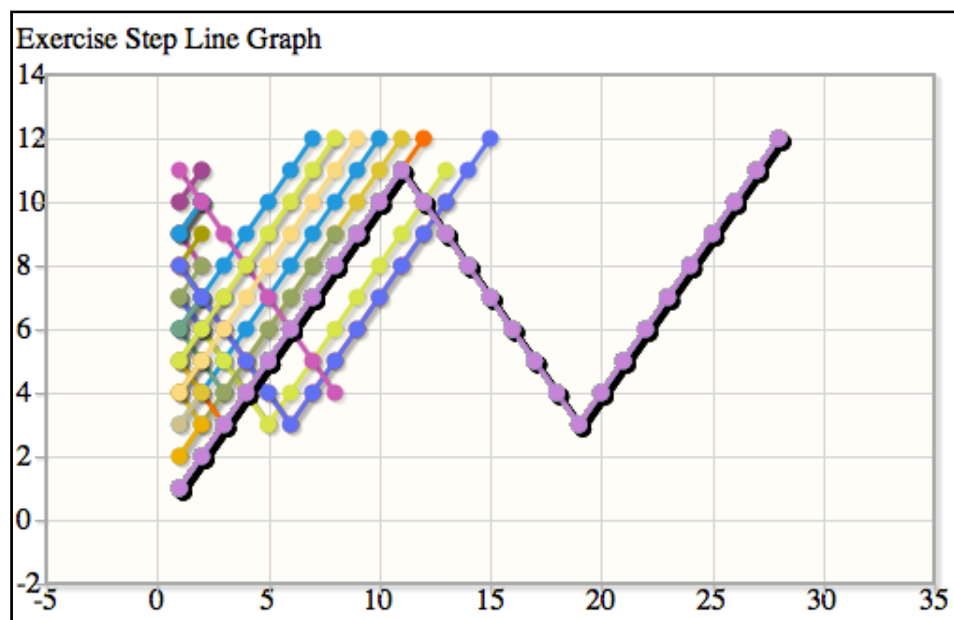

Exercise Time Details by Steps
2095
528
606
404
186
216
134
181
73
74
66
time (sec.)
backward step
forward step
2
3
4
5
6
7
8
9
10
11
12

Exercise Click Details by Steps
211
211
212
212
205
203
206
145
134
73
66
no. of visits
backward step
forward step
2
3
4
5
6
7
8
9
10
11
12

Exercise Detailed Steps Line Graph

The student's exercise summary & details page contains a line graph showing the sequence of steps when the student did the exercise. This view shows the exercise step numbers (y-axis) and the line graph forms the sequence of visits. Separate lines mean different scenarios from separate instances.

The developer uses this line graph view to simulate the student's actual behavior. The developer can extract the information from this view to summarize the patterns of the student doing the exercise.

Below is the screenshot of a sample exercise detailed steps line graph view:



2.3.4 Technical Details

The model classes by the developer view are:

- UserProfile
- UserButton
- UserExercise

The views for the developer view are defined in /templates/developer_view/. The views included for the developer view are:

- student_list.html
- student_exercise.html
- exercise_list.html
- exercise_detail.html

The tables and bar graphs in these pages are implemented using html and CSS. The line graph is implemented using JavaScript.

The controller for the developer view is defined in /opendsa/developerview.py. The classes in the developer view are:

- document_ready_activity
  - The module loading activity counting class. module: module object; activity_num: number of loading.
- proficient_exercises
  - date: the proficiency date in Data format; exercises: a list of exercise object.
- exercise_step
  - The exercise detail step class. step_num: step number; time: time spent; click_num: number of clicks; is_backward: boolean field if the step is backward step

The view definitions for the developer view are:

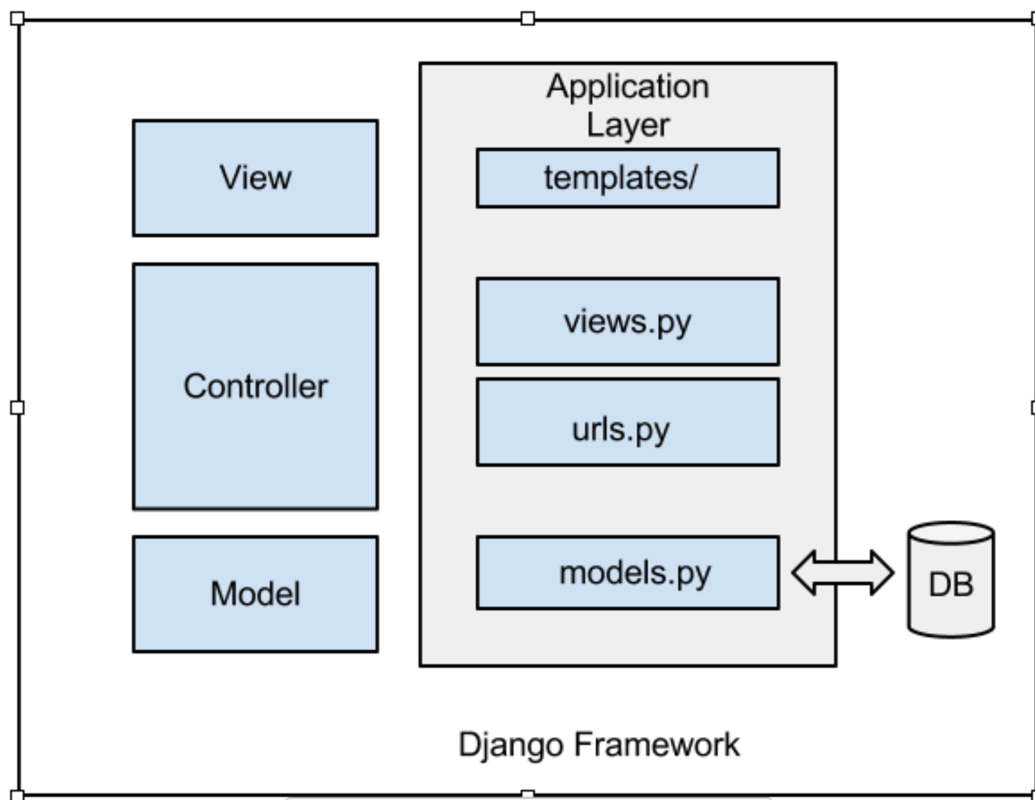
- student_list
  - Defines the student list view. Uses the UserProfile model to build the list of student objects.
- student_exercise
  - Uses UserButton model to fetch the activities of loading modules. The activity name is “document-ready”. Uses UserExercise model the build the list of proficient_exercises.
- exercise_list
  - Defines the exercise list view. Uses the UserButton model to fetch the exercises that are performed activities in the module.
- exercise_detail
  - Defines the exercise summary & details views. Uses UserButton to fetch the activities performed for the exercise. The activities named “jsav-forward” and “jsav-backward” are recorded and handled. Uses

UserExercise to fetch the proficiency date of the exercise and uses UserButton to list the dates the student reviews the exercise.

3. Developer's Manual

3.1 Architecture

Django is a high-level Python Web framework that provides a platform for rapid development of web applications. It follows a Model View Controller approach(MVC). The architecture of Django is as follows:



3.1.1 The Model Layer

Django provides an abstraction layer (the "models") for structuring and manipulating the data of the application. This layer is also known as the data access layer of Django framework.

- Models: A model is the single and definitive source of data. Generally, each model maps to a single database table. Each model is a Python class that subclasses django.db.models.Model. Each attribute of the model represents a database field.

●Model methods: Model methods define custom methods on a model to add custom “row-level" functionality to the model objects. This technique helps in keeping business logic in one place.

●Field types: Each field in your model should be an instance of the appropriate Field class. Django uses the field class types to determine a few things:

The database column type (e.g. INTEGER, VARCHAR).

The default widget to use when rendering a form field.

3.1.2 The View Layer

The view layer encapsulates the logic responsible for processing a user’s request and returning the response. A view function, or *view* is a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image.

●URL dispatcher: The url dispatcher provides a clean, elegant URL scheme for web applications. There are no framework limitations in designing urls.

●Request and response objects: The request and response objects are used to pass state of the web application through the system. When a page is requested, Django creates an HttpRequest object that contains metadata about the request. Then Django loads the appropriate view, passing the HttpRequest as the first argument to the view function. Each view is responsible for returning an HttpResponse object.

3.1.3 The Template Layer

Django’s template engine provides a powerful mini-language for defining the user-facing layer of your application, encouraging a clean separation of application and presentation logic. It provides a designer-friendly syntax for rendering the information to the user.

A template is a text document, or a normal Python string, that is marked-up using the Django template language. A template can contain block tags or variables.

A block tag is a symbol within a template that can output content, serve as a control structure (an “if” statement or “for” loop), grab content from a database or enable access to other template tags.

3.2 Data Flow

The flow of data in Django goes as follows:

●Visitor's browser requests for a URL.

●Django matches the request against its urls.py files.

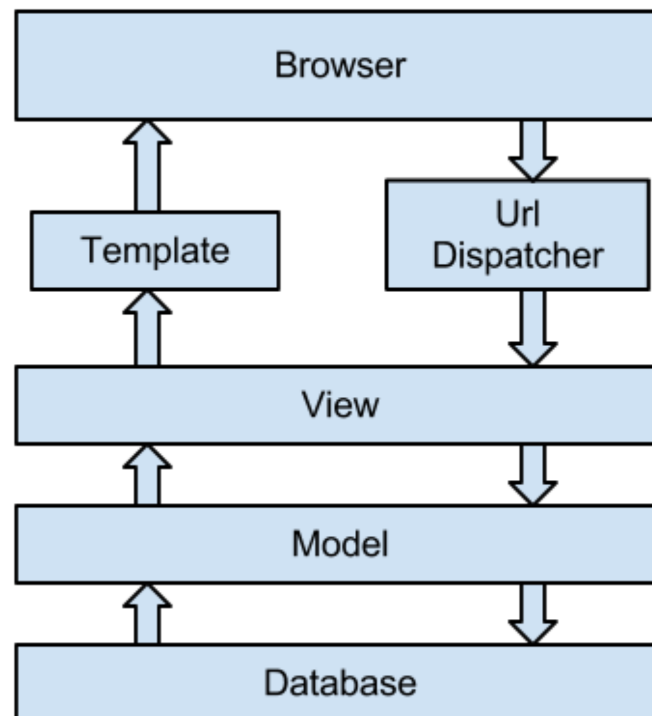
●If a match is found, Django moves on to the view that's associated with the URL. Views are generally found inside each application in the views.py file.

●The view generally handles all the database manipulation. It grabs data from the the database using the classes in the models.

●The model layer provides an abstraction for executing queries through Django framework.

●After the data is received from the models, the view constructs the response object and passes it to the template.

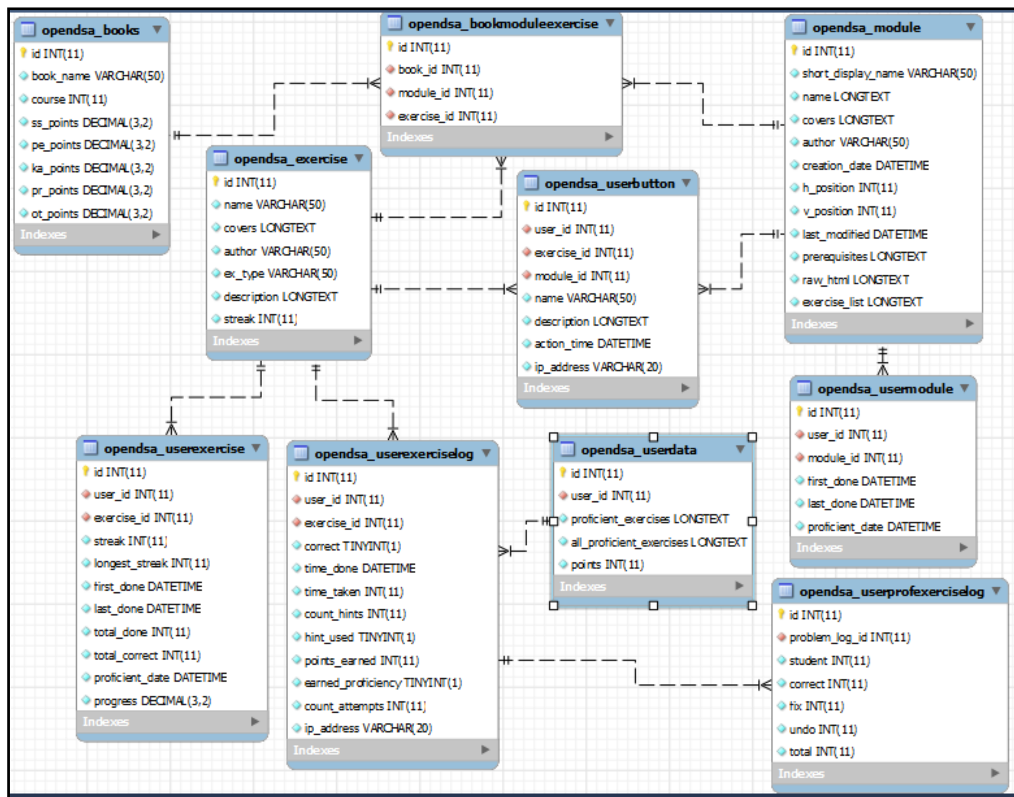
●The template (specified in the view) then displays that data.



3.3 Data Collection

The data used in this project was the logged data captured by the OpenDSA research

team. Using a working copy of the OpenDSA e-Book, they were able to use it in the DSA course of the Fall 2012 semester as a trial run. In the students' interaction with the e-Book, many processes and states were captured. Using this data, our project created visualizations for the different people of interest. Each view uses different or different subsets of the data. Below is only a part of the database schema.



Student Activity

The first type of data captured from the students' interaction with the e-Book came in the form of completion of exercises, activities, etc. Information such as how well a student scored, how many exercises, activities, etc. were correct, as well as when the student started and became proficient in an exercise all are necessary for creation of a grade book or tracking type of tool. This particular data was used in the student and teacher views. The most important models in the database then were userexercise, userdata, and usermodule. The data in these tables were then processed and rendered to create the student and teacher views.

Interface Activity

The second type of data captured was the actual interface activity of the students.

Things such as button pushes, time spent on particular exercises or parts of activities, attempts on certain exercises, etc. were captured and stored in the database. This data is most useful for developers of the OpenDSA e-Book, as this feedback can help them better the interface for the students. The more important models in the database for this case are userexerciselog, userprofexerciselog, and userbutton, to name a few. This data then was processed and used to render different views for the developer.

3.4 Data Processing

Data processing for each view was similar, only the data used and the way the data was rendered differed.

Student View

For the student view, rendering of the data was to show students how they're doing and what more they need to do. The view shows their overall progress in modules (which modules they are proficient in and which ones they are not) as well as at a deeper level, the exercises they need to become proficient in a particular module.

The data was processed for this view by pulling from the table: user information, module information, exercise information, and then also from the usermodule, userexercise, and userdata tables. By grabbing a particular student's information, namely their user ID, from the session information, the student activity data specific to that student could then be pulled from usermodule and userexercise tables. This data was used to generate the student view detailed in the user's manual: module_list. Below is some sample code that shows how the data from the database was pulled and then filtered for the student view.

```
@login_required
def module_list(request):
        book = Books.objects.all();
        userName= request.META.get('USER')

        currentUserVals = User.objects.filter(username = userName);
        modules = Module.objects.all();
        usermodules = UserModule.objects.all();
        userId = 0

        if 'sessionid' in request.COOKIES:
            s = Session.objects.get(pk=request.COOKIES['sessionid'])
            if '_auth_user_id' in s.get_decoded():
                u = User.objects.get(id=s.get_decoded()['_auth_user_id'])
                userId = u.id


        userExercs = UserExercise.GetUserExerciseByUserId(userId)
        userScore = 0;
        userDataObjects = UserData.GetUserDataByUserId(userId)

        for userdata in userDataObjects:
                if userdata.user.id == userId:
                        userScore = userdata.points;
                        break;

        UserModuleDict = dict()
        UserModules = UserModule.GetUserModuleByUserId(userId);
        for usermod in UserModules:
                UserModuleDict[usermod.module] =  usermod;
```

```
        userOutputModules = []
        for module in modules:
                if module.exercise_list.split(',').count > 0:
                        userOptMod = userOutputModule(module,userExercs,book)
                        if userOptMod.countExec > 0:
                                moduleexist = False;
                                if module in UserModuleDict.keys():
                                        if UserModuleDict[module].is_proficient_at():
                                                userOptMod.setprof(1)
                                        else:
                                                userOptMod.setprof(-1)
                                userOutputModules.append(userOptMod);
        takenExercs = []
        profExecs = [];
        nontakenExercs = []
        profUIExecs = []
        nonProfExecs = [];
        for userexerc in userExercs:
                if userexerc.is_proficient():
                        profExecs.append(userexerc);
                        profUIExecs.append(profUIExec(userexerc,book));
                takenExercs.append(userexerc.exercise);


        for userexerc in userExercs:
                if userexerc not in profExecs:
                        nonProfExecs.append(useruiExec(userexerc, False,book));
        for exercise in Exercise.objects.all():
                if exercise not in takenExercs:
                        nontakenExercs.append(exercise);

        return render_to_response("student_view/module_list.html",
                                  {'modules' : userOutputModules ,
                                   'profExecs' : profUIExecs,
                                   'nonProfExecs' : nonProfExecs,
                                   'nontakenExercs':nontakenExercs,'total':userScore });
```

Teacher View

For the teachers of the course, the functionality of the view was for the teachers to be able to see all of their students' activity data, in order to record scores as well as see each student's status in each exercise. The teacher view renders a comprehensive table of all students and all exercises, displaying each student's score and status in an exercise.

The data was processed for this view by pulling user, exercise, userdata, and userexercise tables from the database. Because this view contains all students and

all exercises and the correlating information, using the user and exercise information, lists were built for each user to contain their score and statuses for each exercise. The table in the exercise_summary view, detailed in the user's manual for the teacher view, renders this information. Below is sample code showing the pulling from each table and how the lists were built.

```
@login_required
def exercise_summary(request):

    exercises = []
    BookModExercises = BookModuleExercise.objects.select_related().order_by('book', 'module').all()
    exercise_table = {}

    for bookmodex in BookModExercises:
            exercises.append(bookmodex.exercise)
    #remove duplicates
    exercises = list(OrderedDict.fromkeys(exercises))
    userData = UserData.objects.select_related().order_by('user').all()
    userExercises = UserExercise.objects.select_related().all()
    users = []
    for userdata in userData:
            userVal = userValue(userdata.user, userdata.points)
            #creates an array the size of number of exercises. initilized at 0
            userVal.values = array.array('i',(0,)*len(exercises))
            for userExercise in userExercises:
                    if userdata.user == userExercise.user:
                            if userExercise.is_proficient():
                                    #updates the value in exercise array
                                    if userExercise.exercise in exercises:
                                        userVal.values[exercises.index(userExercise.exercise)]= 1
                            else:
                                    if userExercise.exercise in exercises:
                                        userVal.values[exercises.index(userExercise.exercise)]= -1
            users.append(userVal)

    context = RequestContext(request, {'users': users, 'exercises':exercises})

    return render_to_response("teacher_view/exercise_summary.html", context)
```

Developer View

The developer view is intended for both developers of the OpenDSA e-Book, to find bugs and improvements of the system, as well as for researchers interested in how online e-Book systems are used. The view provides graph visualizations of the system's usage patterns, including exercises proficiency distribution, module loading distribution, exercise time distribution, exercise steps details, etc. for an individual student.

The data processed for this view pulls from the userprofile, userbutton, and userexercise tables in the database. Using userprofile, lists of all students were built to expand into the other views detailed in the user's manual for the developer view. For each student, data is filtered from the userexercise table to render the student_exercise view, showing proficient exercises and other relevant information. Data from userbutton was processed to render the exercise_list view, in order to display which exercises were performed activities within a module. Lastly, the exercise_detail view is rendered using both the userbutton and userexercise tables to

build lists of proficiency dates as well as when the student returned to review the exercises. Below is sample code showing how the student_exercise view was rendered.

```
def student_exercise(request, student):
    userButtons = UserButton.objects.filter(user=student)

    activities = []
    max = 0

    for userButton in userButtons:
        flag = 0
        if userButton.name == 'document-ready':
            for activity in activities:
                if activity.module == userButton.module:
                    activity.update_activity()
                    flag = 1
                    break
            if flag == 0:
                doc_activity = document_ready_activity(userButton.module, 1)
                activities.append(doc_activity)
        #else:
         #   if userButton.name == 'jsav-forward' or userButton.name == 'jsav-backward':
         #       doc_activity.update_activity()

    #number = len(activities)

    userExercises = UserExercise.objects.filter(user=student).order_by('proficient_date')

    exercises = []

    for userExercise in userExercises:
        if userExercise.is_proficient():
            flag = 0
            date = userExercise.proficient_date.date()
            for exercise in exercises:
                if date == exercise.date:
                    exercise.add_exercise(userExercise.exercise)
                    flag = 1
                    break
            if flag == 0:
                temp = [userExercise.exercise]
                exercises.append(proficient_exercises(date, temp, 1))

    return render_to_response("developer_view/student_exercise.html", {'activities': activities
```

3.5 Installation and Maintenance

The OpenDSALog project's implementation is embedded in OpenDSA project so that the installation process is the same. If you are using the OpenDSA e-book as a tool, the project is deployed online. Go to http://opendsa.cc.vt.edu/ and select the course in the course archive. Log in is required for accessing the course contents. If you are a developer of OpenDSA project, the project installation process is as follows: (For more detailed installation document, go to the installation and setup file at https://github.com/cashaffer/Aalto--/blob/master/Aalto%2B_installation_and_setup.txt)

●Install Python:

sudo apt-get install python

●Download and install Django:

tar xzvf Django-z.y.z.tar.gz

cd Django-x.y.z

python setup.py install

●easy_install:

sudo apt-get install python-setuptools

●Install MySQL Server:

sudo apt-install mysql-server

●Start a mysql prompt

mysql –u root –p [ENTER]

Enter your password when prompted

CREATE DATABASE <database_name>;

exit

●MySQL-python:

easy_install MySQL-python

●Dependencies installation:

sudo easy_install oauth2

sudo easy_install simplejson

sudo easy_install feedparser

sudo easy_install icalendar

sudo easy_install mimeparse

sudo easy_install python-deteutil

sudo easy_install django-tastypie

●Get the project code:

Git clone http://YOURGITHUBID@github.com/cashaffer/Aalto--.git Aalto

●Setup the project:

cd Aalto/aaltoplus

Change values in settings.py (see the file for details)

python manage,py syncdb

python manage.py loaddata opendsa/fixtures/modules.json

python manage.py runserver 0.0.0.0:8000

●Access the project, go to http://127.0.0.1:8000/admin/

●The project is maintained on GitHub: https://github.com/cashaffer/Aalto--. The

write access of the project should be obtained from the manager of the OpenDSA project.

3.6 Enhancements and Extensions

The OpenDSALog project is implemented for OpenDSA e-book applied in CS3114. We list the possible enhancements and extensions:

Scalability

The data set kept in the database has an influence on the performance of OpenDSALog project. The scalability of the project shall be improved since the data set keeps growing.

Deployment

The current implementation of OpenDSALog has not been deployed in class for real use. We are planning on deploying it for use for classes in the upcoming semesters. The student and teacher views require appropriate authentications to access.

Visualization

The visualization techniques can be improved to better support the users. Considering the performance of the project, we are implementing the tables and graphs in relatively simple html, CSS and JavaScript techniques. More advanced techniques and libraries can be used in the future to improve the user experience.

Functionality

The log analysis is quite limited to a portion of data set. The analyses on the other data can provide more useful information.

References

[1] E. Fouh, M. Sun, and C.A. Shaffer, OpenDSA: A Creative Commons Active-eBook, a poster presented at SIGCSE 2012, Raleigh, NC, March 2012.

[2] C.A. Shaffer, V. Karavirta, A. Korhonen and T.L. Naps, OpenDSA: Beginning a Community Hypertextbook Project in *Proceedings of 11th Koli Calling International Conference on Computing Education Research*, November 17-20, 2011, Koli National Park, Finland, 112--117.

[3] C.A. Shaffer, T.L. Naps, and E. Fouh, Interactive Textbooks for Computer Science Education in *Proceedings of the Sixth Program Visualization Workshop*, June 30, 2011, Darmstadt, Germany, 97-103.