QuizASSIST: Mobile Application for ASSISTments

An Interactive Qualifying Project
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfilment of the requirements for the
Degree of Bachelor of Science

by
Yiren Wang

Date:
October 13, 2016

Submitted to:

Professor Neil Heffernan and Cristina Heffernan
Worcester Polytechnic Institute

ABSTRACT

The goal of this IQP project is to develop the initial version of an iOS mobile application called QuizASSIST. This app is an successor to the ASSISTments website, and allows the users to take quizzes for different problem sets. This report describes the development process and talks about the design of the app in detail.

ACKNOWLEDGEMENTS

First of all, I would like to thank Professor Neil Heffernan and Cristina Heffernan for providing this IQP opportunity for me. The large amount of advice and support coming along the way has been very helpful in the development of this project.

I would also like to thank David Magid and Christopher Donnelly, who work as software engineers in the ASSISTments lab, for answering my questions and discussing many implementation problems with me. I could not have done this project without their instructions and suggestions for fields that I am not familiar with.

Finally, I would like to thank Emily Hao for working on the user interface design of this project, and Fangming Ning for his previous work on another ASSISTments mobile application, which has inspired me for my project.

TABLE OF CONTENTS

INTRODUCTION

ASSISTments is a web-based tutoring system developed at WPI and has been put into use since 2004. ASSISTments assists students in many different subjects, mostly math and science. By working on the problems assigned in class, students are supposed to master the materials faster. Meanwhile, a large amount of research has been conducted based on students' behaviors and their feedback with respect to the problems. The outcomes of these research studies can help the teachers teach better.

The new idea of QuizASSIST, a mobile application and successor to ASSISTments, helps the students enjoy working on problems. Instead of asking teachers to give out assignments, QuizASSIST generates questions randomly under a topic selected by the student, and sends them out in the form of a quiz. The quiz is usually fast-paced and students can practice on it as many times as they want. They will be able to see the score they get on every quiz, thus knowing whether they have achieved any progress after practicing many times.

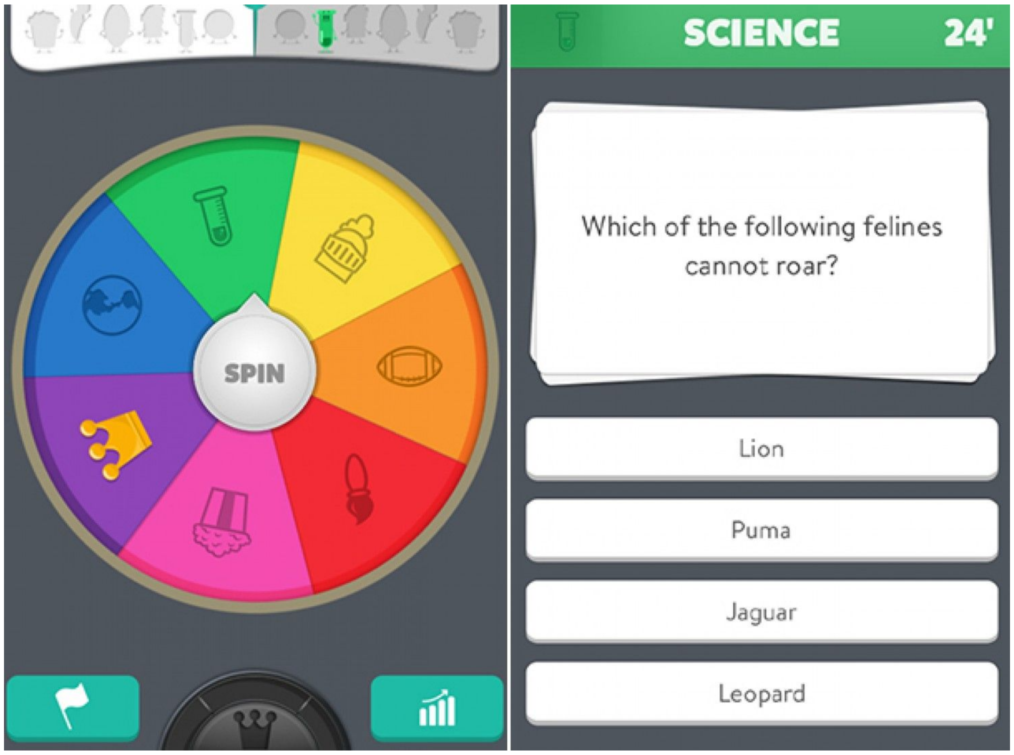
The goal of this IQP is to build the first version of QuizASSIST, which allows students to practice the quizzes by themselves. Another team will keep working on the app and add more functions to it, such as implementing a friend system where students can challenge each other by taking the same quiz and comparing scores, working again on all the problems they did wrong before, etc. The app is planned to be published onto the app store by the end of B term 2016.

BACKGROUND

Before starting to design QuizASSIST, I have researched other mobile applications with similar functions and goals. The one called Trivia Crack is the most popular among them.

Figure 1. Trivia Crack Icon

The first thing that attracted me was the user interfaces, pretty icons (seen above in Figure 1), and drawings. The whole design is very user-friendly and makes the quiz look fun to play with. This will also be one of the key problems we need to consider for QuizASSIST: how can we make the app attractive?



Which of the following felines cannot roar?

Figure 2. Trivia Crack Game View

In Trivia Crack, the user can start a new game to play with his friends or strangers, as shown in Figure 2. The classic mode takes at most three days to complete, and the challenge mode is always completed in a few minutes. The problems given are multiple choice and cover all kinds of knowledge.

Since we want to keep QuizASSIST fast-paced, we would like the user to always finish a quiz within several minutes. For now the user can only do the quiz by himself, but we are looking forward to adding a friend system that will make the quiz more interesting.

The idea of giving out problems randomly works for Trivia Crack because the goal of the game is to test the user's knowledge over everything. However, we want our users to master some skills by using

QuizASSIST, so we will ask the users to choose a problem set that they want to practice, and generate random problems from it to create a quiz.

Trivia Crack has many other functions such as selecting a different language for the problems, chatting with a friend before starting the game, etc. Many parts of its design are worth learning from, and have helped my design with QuizASSIST.

METHODOLOGY

Development Platform

Our goal is to develop QuizASSIST both on the Apple iOS platform and Android platform. We are focusing on the iOS platform right now because many K-12 schools nowadays have programs to provide students with Apple iPads for educational purpose. Therefore a large number of students will be able to try out our app for free and provide feedbacks for us to improve the app.

The app is developed using Objective-C, but will switch to Swift someday in the future because Swift will supplant Objective-C as it has a higher performance compiler.

The server side of the application is written in Java and uses Spring Framework to deal with HTTP requests sent from the app. In order to connect with ASSISTments and achieve related problems from it, the server code also implements the software development kit (sdk) provided by ASSISTments.

Decisions for Design

1. User Account

Since QuizASSIST is a product coming from ASSISTments, at the very beginning we would like users in ASSISTments to be able to use QuizASSIST with the same accounts, so they can login to QuizASSIST directly. However, if there are people who do not know about ASSISTments at all and are willing to try QuizASSIST, they will have to create a new account using the ASSISTments website, which is inconvenient for them.

Therefore, we decided to separate QuizASSIST users from ASSISTments users by introducing a different user database for QuizASSIST. Everyone who needs access to QuizASSIST should create a new user account with a unique email address. This simplifies the control of user accounts for us developers, and makes QuizASSIST an independent application from ASSISTments.

If researchers are interested in the relation of the two user groups in the future, we can still compare the email addresses and names from both databases. Once there are two records that match, it is highly possible that they belong to the same user and we can connect his performances in these applications.

2. Problem Source

When we first came up with the idea of QuizASSIST, we wanted to distinguish it from ASSISTments by not having teachers involved in it. That means the only users of QuizASSIST will be the students. Although we will import some existing problems created by teachers from ASSISTments, most of the problems should be created by the students.

Then we have to deal with another concern: who is going to verify those problems created by students? The program itself can help us check if the problem formatting is right and if all the information needed is filled in, but we still need humans to check whether the problem is meaningful, the correct answer given is convincing, etc. We need a verification process faster than the speed of problems coming in, because otherwise we will start to lose users.

One way to improve the situation is to let the users decide whether the problem is useful or not. For example, every time they finish working on a problem, they can “like” or “dislike” the problem. Once we receive enough feedbacks, we will be able to tell if this problem is reliable, and decide whether we need a human to look through it. We can also give a problem with many “like”s a higher chance to show up in the quiz.

There are also other concerns that need to be solved in order for students to build problems. Many problem descriptions need to have images and formulas, so the problem builder on the phone will be complicated to implement. The problems built by students should not be stored with the ASSISTments ones because they only apply to QuizASSIST. We also need to think about how to create a new quiz with questions both from these problems and ASSISTments verified problems if they are in two different databases.

After several discussions during the IQP, we decided for now not to let the students build problems. Instead, we will use the existing ASSISTments problems only to create a quiz. However, the idea of letting students build problems can still be realized in later versions of this app.

Meanwhile, another IQP team has started to add more Chemistry problems into ASSISTments in A term 2016. These problems are mostly for the Chemistry 1010 class at WPI in C term 2017. After the app is published in B term 2016, those problems will be our main focus for testing when the app is used by Chemistry 1010 class students.

3. Problem Details

On the ASSISTments website, teachers are allowed to build problems with these types: multiple choice, check all that apply, numeric and algebraic expressions, exact match (case sensitive or not), ordering and open response. We will implement all of them on QuizASSIST except ordering, which is rarely used, and open response, which cannot be graded.

The existing problems in ASSISTments usually have hints provided, and the students will lose some points if they choose to see the hints before trying the problem. The students can also provide feedback regarding whether they think the hints are useful. We will not implement hints and feedback on the first version of QuizASSIST, but they can be added in future versions.

APPLICATION

Icon, Logo and Launcher Image

The icon, logo and launcher image for QuizASSIST were developed using Adobe Illustrator. For the QuizASSIST logo in Figure 3, I decided to use a smiling face inside a “Q” as the theme, and arranged the word “QuizASSIST” with “Q” and “A” standing out. In order to make a connection with ASSISTments, I put a right tick above the “i” like the ASSISTments logo shown in Figure 5.

The icon in Figure 4 is especially for a mobile app, and it will be shown on the home screen of the mobile device. The picture for the icon is taken from the “Q” in the QuizASSIST logo.

The launcher image in Figure 6 is shown as soon as the user clicks the app icon on his mobile device. It will stay there for about three seconds and then disappear. The picture has a very similar design with the logo, but I picked a different color theme for it.

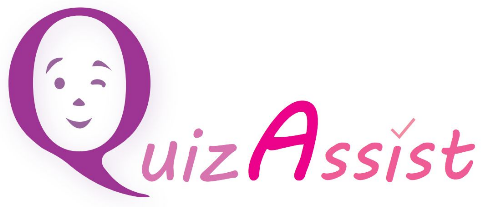


Figure 3. QuizASSIST Logo

Figure 4. QuizASSIST Icon

ASSISTments

Figure 5. ASSISTments Logo

Figure 6. QuizASSIST Launcher Image

Program Outline

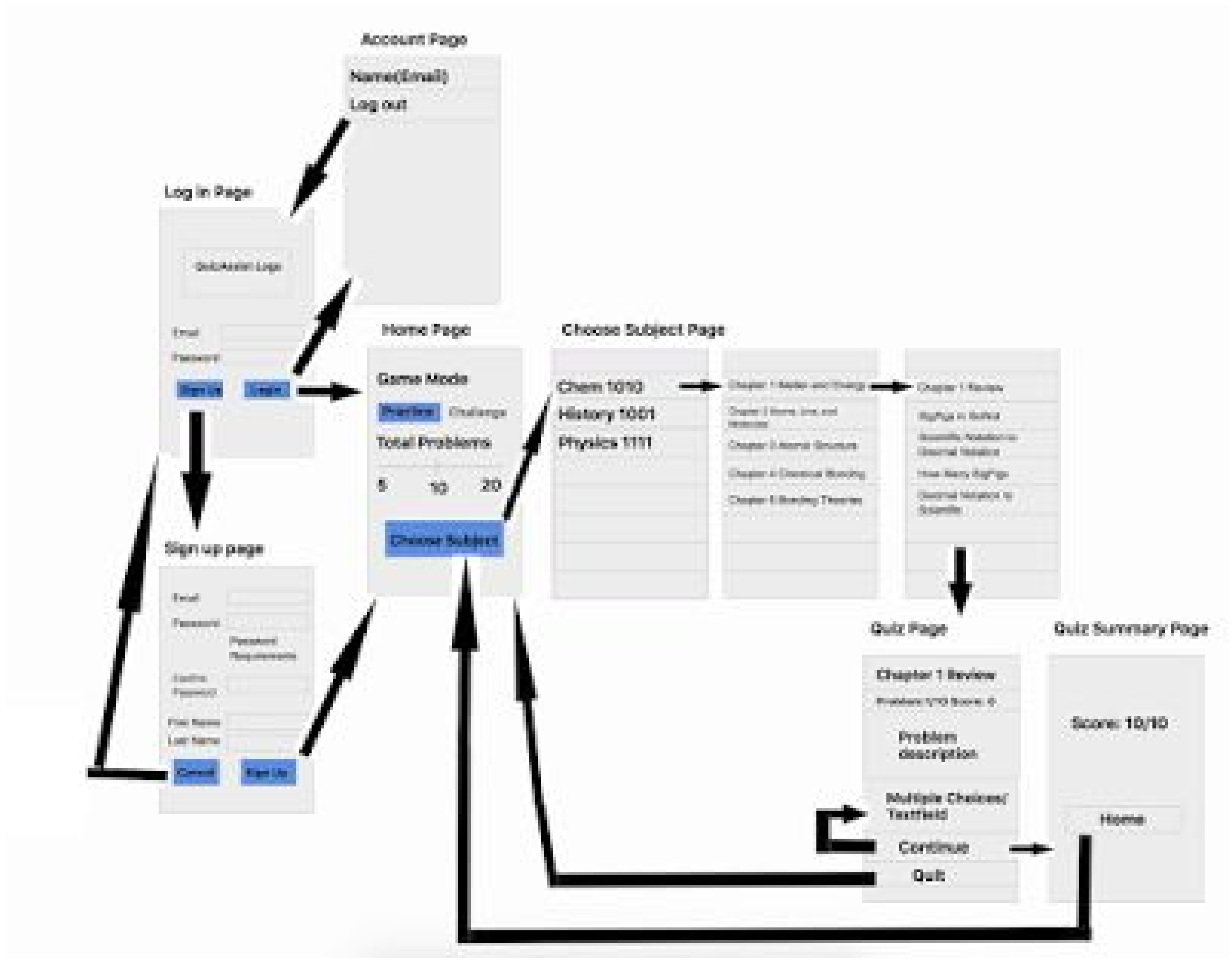


Figure 7. QuizASSIST Application Flow Chart (created by Emily Hao)

The flowchart of the first version of the app is shown in Figure 7. We will discuss more about each page in the following sections.

1. Log in Page

As the user opens the app, after the launcher image disappears, he will be directed to the login page. He can login with an email address and a password, or click on “Sign up” and go to the sign up page.

2. Sign up Page

The user will use this page to create a QuizASSIST account. The email address he uses must be unique among the QuizASSIST users.

There is a strict requirement for the password he can use. The password has to be at least 6 characters long, contain at least one uppercase letter, one lowercase letter, a number and a special character. This requirement comes from the software development kit used by ASSISTments.

The user has to fill in all the information in order to create his account. If the account is created successfully, the user will be automatically directed to the home page.

3. Home Page

The user will create a new quiz starting from here. The only game mode available for the first version is “Practice”, which means the user practices the quiz by himself. The total problem count varies from 5 to 20, and the user can pick a value using the slider. Then he can continue to the next page by clicking “Choose Subject”.

4. Choose Subject Page

The user will choose the problem set as he navigates through the pages. For example, under the class “CH 1010” there are seven chapters, and under each chapter there are different numbers of problem sets. The user will end up choosing one problem set and start the quiz with it, and he can also go back and forth during this process.

5. Quiz Page

The quiz will start by giving out one problem at a time. The problem set name, current progress of the game and problem description will show up. If it is a “multiple choice” or “check all that apply” problem, four options will be listed for the user to choose. Note that some problems in our database may contain more than four choices, but we will only display four of them and must include the correct ones. For a “fill in the answer” problem, we will only show a text field and wait for the user’s input. After the user puts in his answer, a “Continue” button will pop up. The design of the “Continue” button makes sure that the user has to answer every problem before moving on to the next one.

After the user selects an answer for a “multiple choice” problem, we will show immediately whether his answer is correct. If it is, the answer he selects will be marked as green, otherwise we will mark his answer as red and show the correct one in green. Then a “Continue” button will show up for the user to go to the next problem.

For a “check all that apply” problem, there can be more than one correct answer. If the first answer the user selects is correct, we will mark it as green and wait for his next answer without showing the “Continue” button. “Continue” is only shown once the user has selected all the correct answers or one wrong answer. In the latter case we will display all the correct answers to him.

With a “fill in the answer” problem, the user will put in his answer and click “Submit” on his keyboard. We will let the server decide whether his answer is correct, and display the correct answer if it is wrong.

At the end of the quiz page, there is a “Quit Game” choice, and the user can give up the quiz at any time during the quiz. He cannot come back to the quiz later once he gives it up, so we do not need to think about how to save his progress.

6. Quiz Summary Page

As the user completes all the problems for the quiz, we will show a summary page, and from here the user can go back to the home page. For now we only show the user’s score and total problem count. The summary page will need to be redesigned after implementing the friend system in the B term 2016.

7. Account Page

The user account page can be accessed from the home page by choosing the tab at the bottom of the page. This page shows the user’s name and email address, and allows the user to log out.

Plan for Version 2

1. Friend System

As we plan to publish the app to the app store by the end of B term 2016, we hope to have the friend system implemented. The “Challenge” game mode will be available, and the user will be able to choose to do the quiz with a friend or random QuizASSIST user.

First of all the user will be able to add a new friend and accept friend requests from others. This has actually been done in the first version, and we also allow the user to delete a friend.

When the user starts a new quiz, the user interface will change to allow him to choose a friend from a list to play with. A notification for this quiz request will be sent to his friend. Then we will need another new user interface for his friend to deal with this request.

These two users will not need to play the quiz at the same time, but the server will be able to record their progress for the quiz. When one user has started or finished the quiz, the server will have a way to display this change to the other user. By the time both users finish the quiz, their final scores will be shown to each other.

2. Logging

Another important thing we will do for Version 2 of QuizASSIST is to log the user’s actions. We are mostly interested in the users’ performance on each quiz, and we would like to record their answer on each problem in particular. We will show these records to the user on a new “Achievements” page so that they will be able to check their performance on every quiz and problem set as well. This way they can improve skills by practicing on those problem sets that they have not got a high score on.

Meanwhile, the data will be very useful for future research. The researchers in the ASSISTments lab can do different analyses on the data, and come up with some advice about how we can modify the app to have better user experience.

Moreover, after we build a connection between the ASSISTments users and QuizASSIST users, we will have an idea of how many students using ASSISTments are involved in QuizASSIST. While the students are practicing on QuizASSIST for a class, we can generate reports based on their performance and send them to the teacher. This will also help the teacher to teach better, such as focusing more on those parts where most of the students got low scores.

The logging will be implemented on the server side only and will use some sdk from ASSISTments, so that the information logged will be consistent with the ASSISTments logging formatting.

3. Additional Databases

The ASSISTments sdk handles most of the databases for the first version, such as the user’s information, problems and the problem sets. The only thing that the server will do is to make queries on those databases, and send the organized information back to the app. However, the friend system and logging will require additional databases particularly for the QuizASSIST application.

In order to exchange information between two users, first of all we need to record the tokens of the mobile devices they are using. The token strings are stored in the database called “mobile_devices”, and the database should be updated everytime a user logs in. The server will communicate with a certain device by using ApnsService to create a package with this string and send it to the device. Then the mobile device will receive a notification from the Control Center and then handle the incoming package.

The Android app to be built in the future will need a similar database for the friend system. Therefore, we have an attribute “device_type” in the “mobile_devices” database, so we can store its tokens in this database as well by using a different “device_type” number. For now we only use iOS devices, so we put a “1” for every “device_type”.

After a user sends a friend request, we need to record this request in our database “friends”, and update the information if the other user accepts or deletes this request. When we want to get the list of friends for a user, we also go to this database and do a query based on the user id.

When we are logging for the QuizASSIST application, we may need a separate logging database from the ASSISTments ones, though it will use the same structure as the ASSISTments databases do. This may need further discussion and will depend on how we want to use the logging information.

CONCLUSION

Building the first version for QuizASSIST will be done successfully by the end of A term. I designed the icon, logo and launcher image; worked with Emily to come up with the user interfaces and the flow of the app; programmed most of the codes both on the client and server side. I have also met up with the other IQP team in A term 2016 a lot, introducing the app to them and helping them to create problems needed by the app.

During this process I have also thought about many other functions. Since the time for my IQP is limited, I cannot implement all of them in the first version. The design of QuizASSIST has also changed many times and we gave up some functions which previously worked due to many reasons. These potential functions to be done in the future all need further discussions before implementation, and I will leave them here as a reference for the other teams. More of these can be found in the Appendix A.

Future Potential Features

1. Selecting Multiple Problem Sets

Our plan for the first and second versions is that the user starts a quiz with only one problem set at a time, but sometimes the user may want to do a review for several problem sets. In the future we can allow the user to choose multiple problem sets at a time, such as implementing a check box in front of every problem set.

2. Multiple Players for “Challenge” Mode

The “Challenge” mode of the quiz to be implemented in B term is only for two users. If we have more users joining QuizASSIST later, having multiple players doing the same quiz will be more fun for the users, since they can value their skills among more people.

3. “Rough Match” Problem Type

Our “fill in the answer” type problem uses “exact match” when comparing answers in plain text. This means if the user misspelled an answer even by one letter, the answer will be marked as wrong. Since QuizASSIST will be a fast-paced game, it is highly possible that the user types in the answer too fast to make it correct.

Therefore, we would like the problem to be more user-friendly and the user only gets it wrong for not knowing the answer. A “rough match” problem should allow the user to misspell several letters in the answer, and still mark him correct as he gets the general idea right.

This will need extra implementation in the ASSISTments sdk and need more discussion for details.

APPENDIX A: Project Outline Draft

A term - Version 1

User should be able to

- Start a new game for practice only
- Add and delete friends

Tasks for coding group

- Make the app support these problem types: multiple choice, check all that apply, exact match text answer (ignore case or not), numeric and algebraic type question (all kinds)
  (*according to assistments question types)
- Make sure html codes and images can be displayed properly
- New icons, logos and launcher image work on different screen sizes
- Remove redundant codes and databases and saved for future
- Publish the app for testing by random users
- User should be able to leave a game at any time - should not need to finish a game to leave

Tasks for content group

- Generate some organized problem sets into assistments database
- Generate at least 20 questions for some problem sets
- For each problem type, generate some questions for testing on the app
  (MC & CA : >=4 answers for each problem, only text in answers)
- Images and specified text styles are allowed
- Hints and feedback can be put in and used in the future

B term - Version 2

User should be able to

- Start a new game with a friend or stranger

Tasks for coding group

- Add challenge mode for game: user selects to play with a friend (choose a friend from friend list) or stranger; server side sends the same problems to both players
- Show notification when receiving a challenge; challenge has to be completed in some time; show whether opponent completed the challenge; when both players complete the challenge, update the status and show scores
- Show the score history (exact numbers or percentages, different game modes) to user
- Publish the app to app store
- Save progress on a particular practice quiz if the user left the game early

Tasks for content group

- Generate enough questions to be ready for use in C term

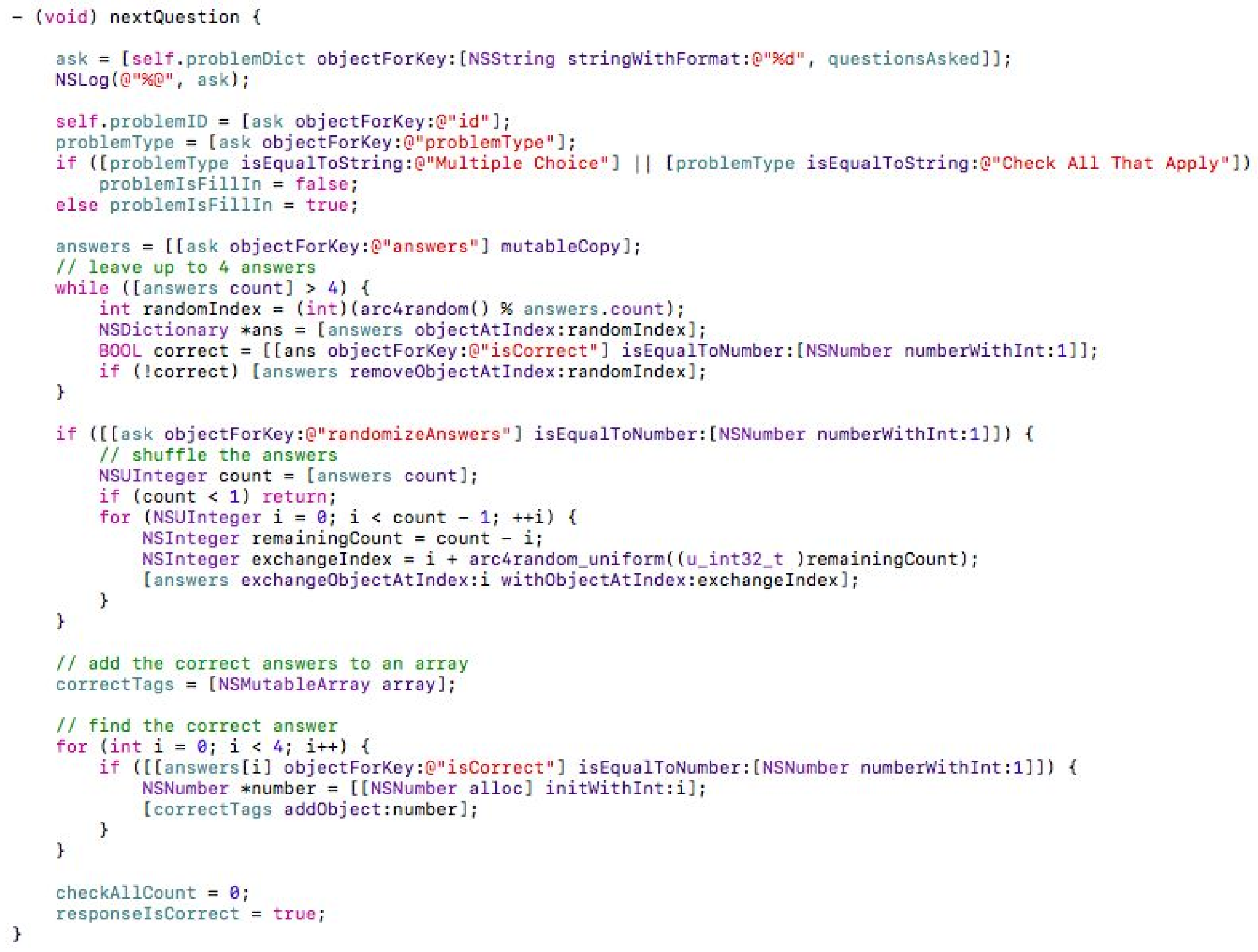
Potential tasks for future

- Implement "Like" and "Dislike" for each problem; "popularity" of the problem shown to users or only shown to teachers; more "popular" problem shows up in quiz more frequently
- User select multiple problem sets to start a game
- User start a game with "all the problems he did wrong before"
- User be able to start a game with multiple players (friends or strangers)
- User can exit the game and play it later (different from our current design)
- Logging user actions on server side; be able to generate reports for teachers
- Display hints and receive feedback
- User be able to create problems (most codes have been done, but think about a more complicated problem builder (etc. images), where to store the new problems, who to verify the problems are valid, when starting a game these problems can be reached)

- A new question type “Rough match”: judge answer is correct if misspelled
- Allow users to change names/password
- Make a new storyboard for ipad
- Switch from objective-c to swift
- On login page, user chooses to remember login or not; if remember login then do not show login page, go to home page directly
- User has ability to change themes (color, pic, etc.)
- User should be able to skip a problem and come back to it later in the quiz

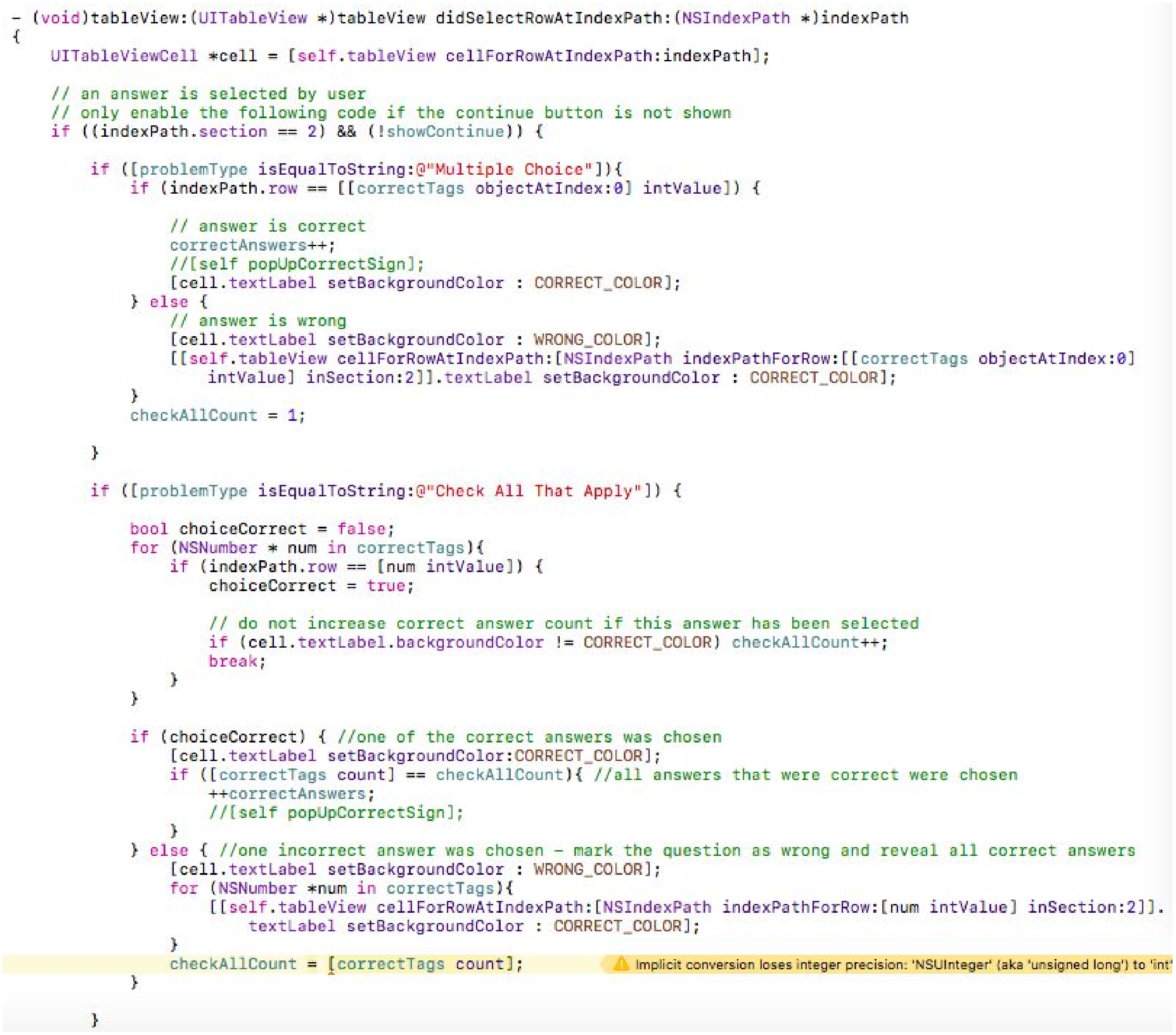
APPENDIX B: Sample Source Codes

There are about 30 code files for the iOS app and 20 files for the server side. They can be accessed on the fusion website using SVN. Contact the ASSISTments lab for more details about how to achieve them. Here I will show screenshots for some main codes in GameTableViewController.m for iOS and some server side codes.



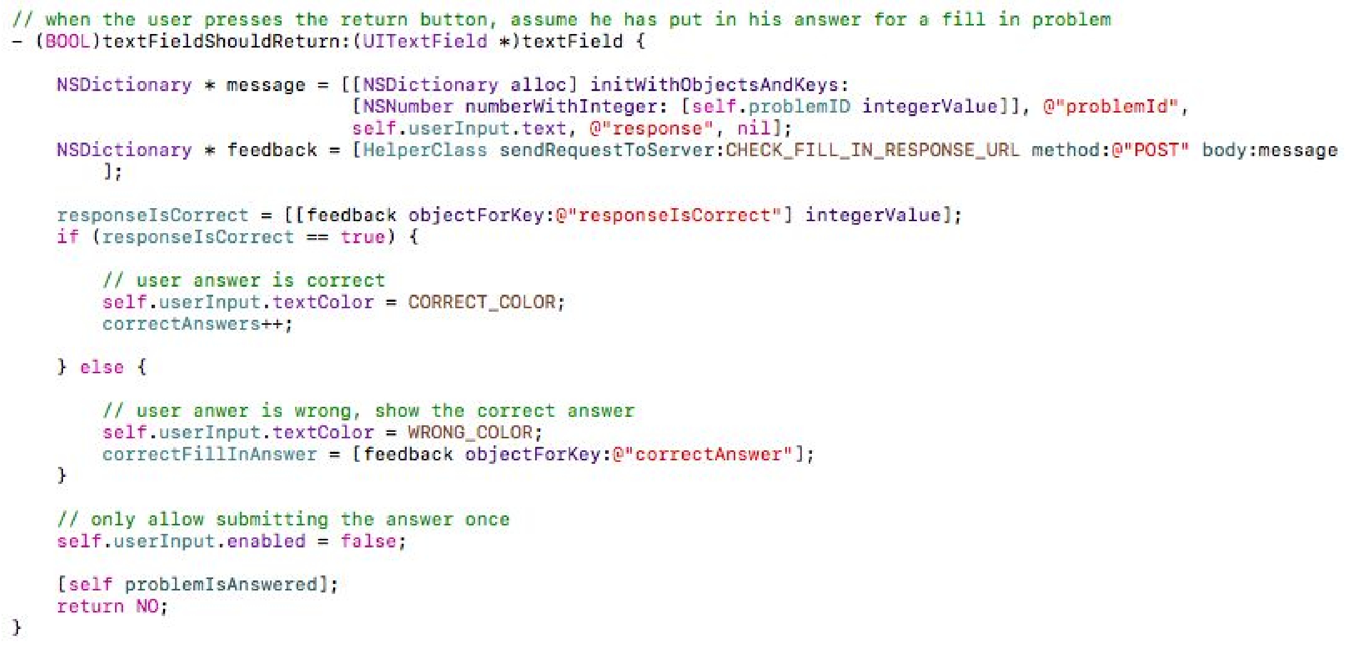
```
- (void) nextQuestion {

    ask = [self.problemDict objectForKey:[NSString stringWithFormat:@"%d", questionsAsked]];
    NSLog(@"%@", ask);

    self.problemID = [ask objectForKey:@"id"];
    problemType = [ask objectForKey:@"problemType"];
    if ([problemType isEqualToString:@"Multiple Choice"] || [problemType isEqualToString:@"Check All That Apply"])
        problemIsFillIn = false;
    else problemIsFillIn = true;

    answers = [[ask objectForKey:@"answers"] mutableCopy];
    // leave up to 4 answers
    while ([answers count] > 4) {
        int randomIndex = (int)(arc4random() % answers.count);
        NSDictionary *ans = [answers objectAtIndex:randomIndex];
        BOOL correct = [[ans objectForKey:@"isCorrect"] isEqualToNumber:[NSNumber numberWithInt:1]];
        if (!correct) [answers removeObjectAtIndex:randomIndex];
    }

    if ([[ask objectForKey:@"randomizeAnswers"] isEqualToNumber:[NSNumber numberWithInt:1]]) {
        // shuffle the answers
        NSUInteger count = [answers count];
        if (count < 1) return;
        for (NSUInteger i = 0; i < count - 1; ++i) {
            NSInteger remainingCount = count - i;
            NSInteger exchangeIndex = i + arc4random_uniform((u_int32_t )remainingCount);
            [answers exchangeObjectAtIndex:i withObjectAtIndex:exchangeIndex];
        }
    }

    // add the correct answers to an array
    correctTags = [NSMutableArray array];

    // find the correct answer
    for (int i = 0; i < 4; i++) {
        if ([[answers[i] objectForKey:@"isCorrect"] isEqualToNumber:[NSNumber numberWithInt:1]]) {
            NSNumber *number = [[NSNumber alloc] initWithInt:i];
            [correctTags addObject:number];
        }
    }

    checkAllCount = 0;
    responseIsCorrect = true;
}
```

Figure A. Preparation before displaying the next problem



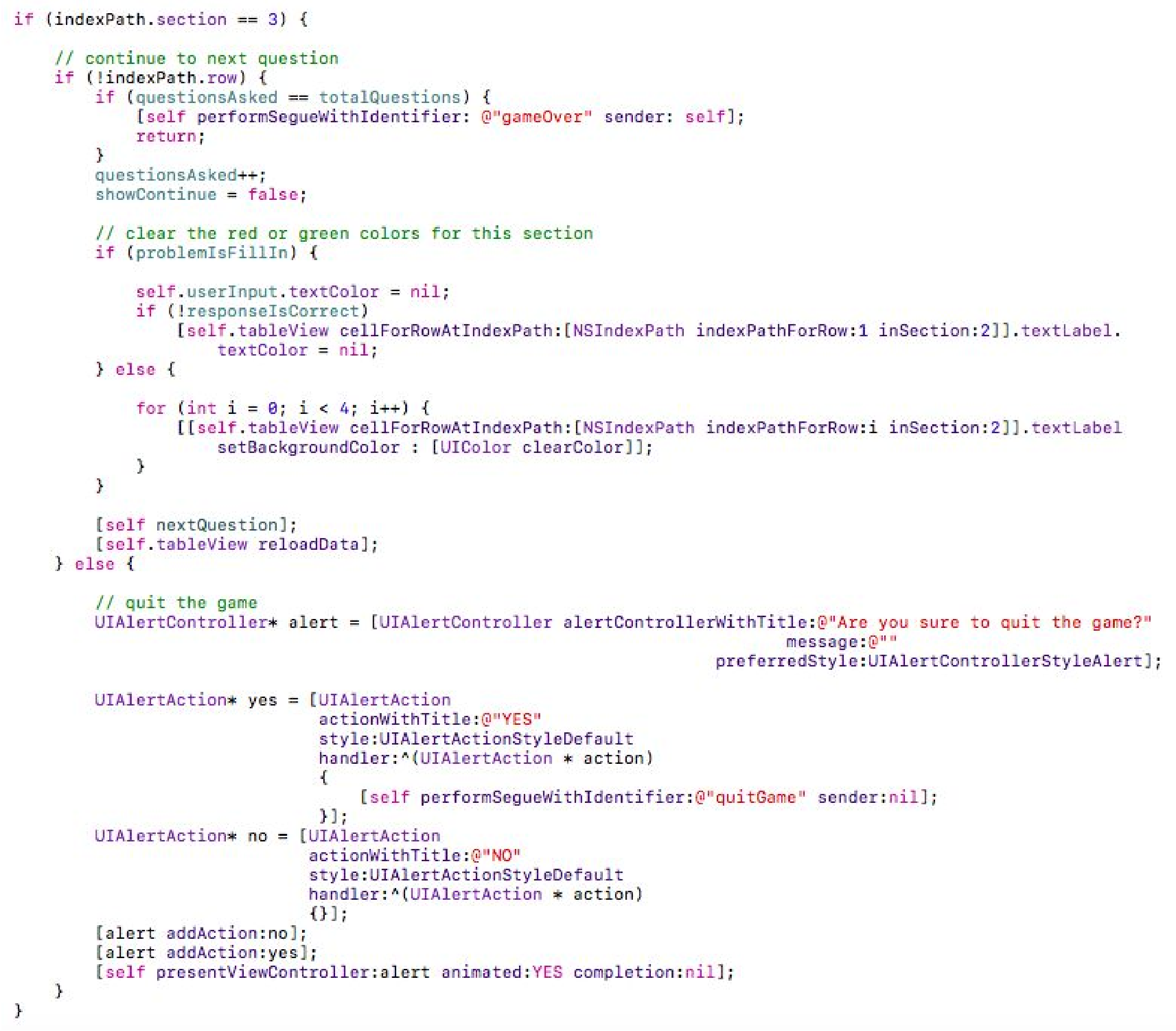
```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:indexPath];

    // an answer is selected by user
    // only enable the following code if the continue button is not shown
    if ((indexPath.section == 2) && (!showContinue)) {

        if ([problemType isEqualToString:@"Multiple Choice"]){
            if (indexPath.row == [[correctTags objectAtIndex:0] intValue]) {

                // answer is correct
                correctAnswers++;
                //[self popUpCorrectSign];
                [cell.textLabel setBackgroundColor : CORRECT_COLOR];
            } else {
                // answer is wrong
                [cell.textLabel setBackgroundColor : WRONG_COLOR];
                [[self.tableView cellForRowAtIndexPath:[NSIndexPath indexPathForRow:[[correctTags objectAtIndex:0]
                    intValue] inSection:2]].textLabel setBackgroundColor : CORRECT_COLOR];
            }
            checkAllCount = 1;

        }

        if ([problemType isEqualToString:@"Check All That Apply"]) {

            bool choiceCorrect = false;
            for (NSNumber * num in correctTags){
                if (indexPath.row == [num intValue]) {
                    choiceCorrect = true;

                    // do not increase correct answer count if this answer has been selected
                    if (cell.textLabel.backgroundColor != CORRECT_COLOR) checkAllCount++;
                    break;
                }
            }

            if (choiceCorrect) { //one of the correct answers was chosen
                [cell.textLabel setBackgroundColor:CORRECT_COLOR];
                if ([correctTags count] == checkAllCount){ //all answers that were correct were chosen
                    ++correctAnswers;
                    //[self popUpCorrectSign];
                }
            } else { //one incorrect answer was chosen - mark the question as wrong and reveal all correct answers
                [cell.textLabel setBackgroundColor : WRONG_COLOR];
                for (NSNumber *num in correctTags){
                    [[self.tableView cellForRowAtIndexPath:[NSIndexPath indexPathForRow:[num intValue] inSection:2]].
                        textLabel setBackgroundColor : CORRECT_COLOR];
                }
                checkAllCount = [correctTags count];          ⚠ Implicit conversion loses integer precision: 'NSUInteger' (aka 'unsigned long') to 'int'
            }

        }
```

Figure B. Adjusting the user interface after the user selects an answer



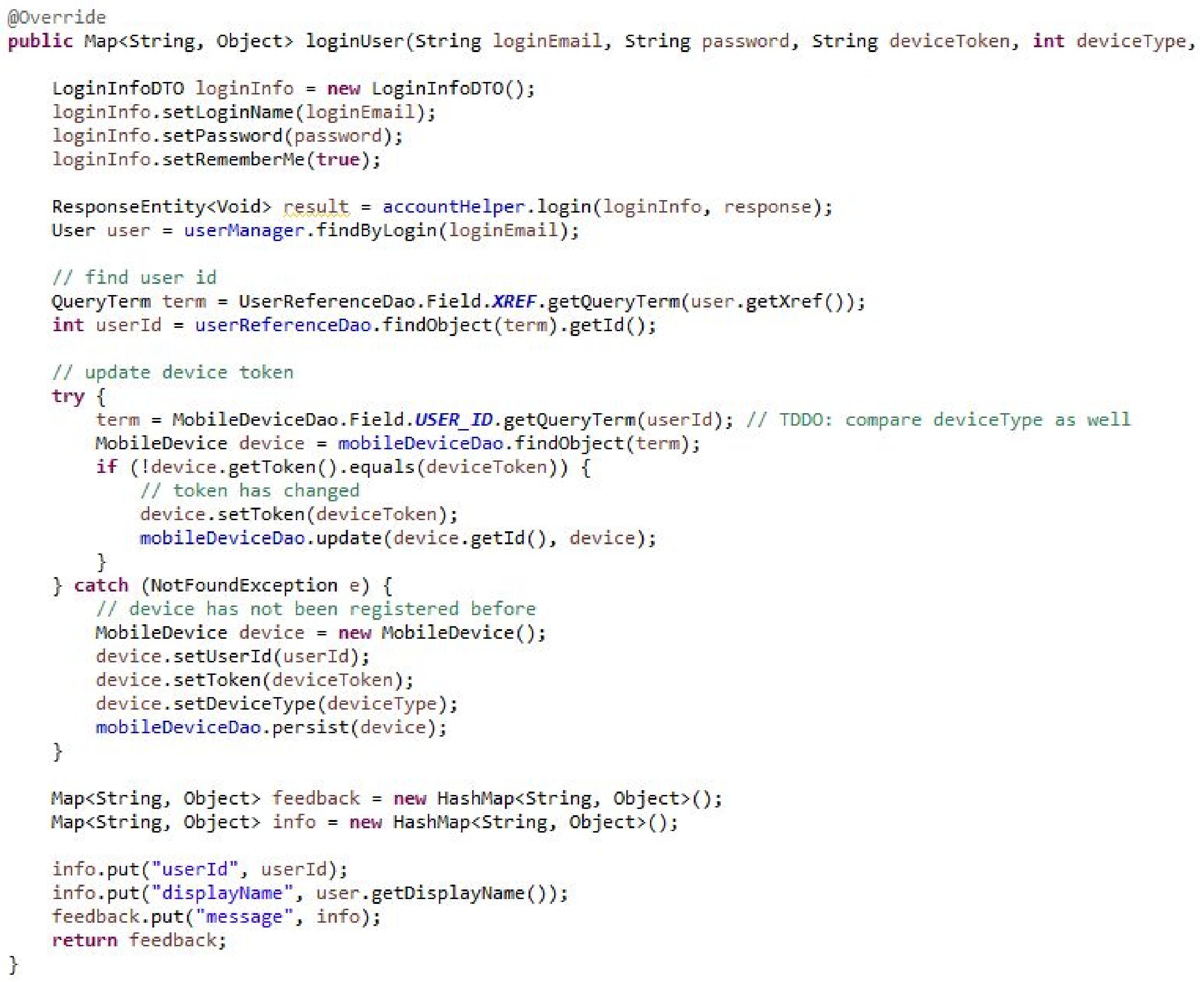
```
// when the user presses the return button, assume he has put in his answer for a fill in problem
- (BOOL)textFieldShouldReturn:(UITextField *)textField {

    NSDictionary * message = [[NSDictionary alloc] initWithObjectsAndKeys:
                              [NSNumber numberWithInteger: [self.problemID integerValue]], @"problemId",
                              self.userInput.text, @"response", nil];
    NSDictionary * feedback = [HelperClass sendRequestToServer:CHECK_FILL_IN_RESPONSE_URL method:@"POST" body:message
        ];

    responseIsCorrect = [[feedback objectForKey:@"responseIsCorrect"] integerValue];
    if (responseIsCorrect == true) {

        // user answer is correct
        self.userInput.textColor = CORRECT_COLOR;
        correctAnswers++;

    } else {

        // user anwer is wrong, show the correct answer
        self.userInput.textColor = WRONG_COLOR;
        correctFillInAnswer = [feedback objectForKey:@"correctAnswer"];
    }

    // only allow submitting the answer once
    self.userInput.enabled = false;

    [self problemIsAnswered];
    return NO;
}
```

Figure C. Adjusting the user interface after the user types in an answer



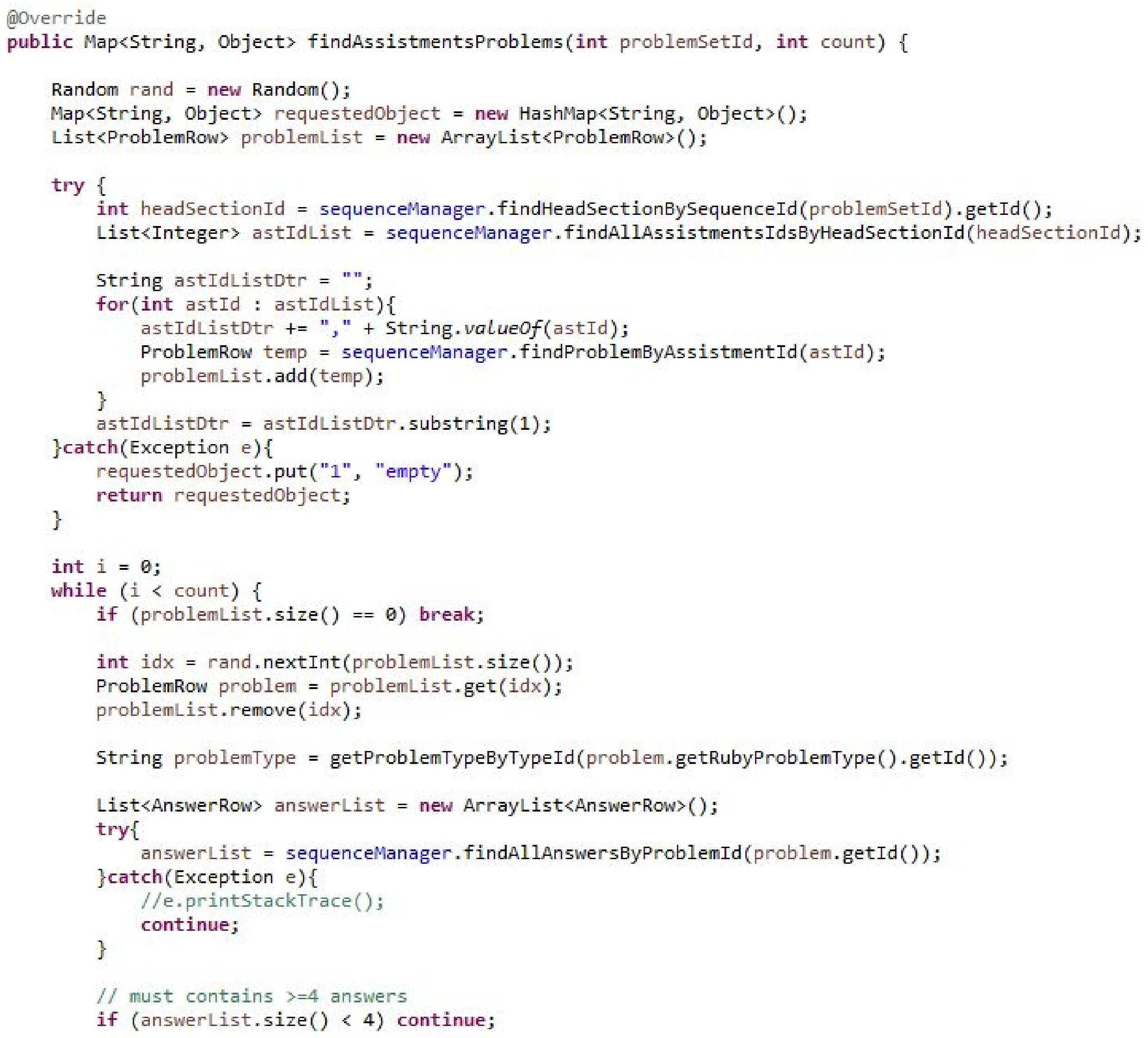
```
if (indexPath.section == 3) {

    // continue to next question
    if (!indexPath.row) {
        if (questionsAsked == totalQuestions) {
            [self performSegueWithIdentifier: @"gameOver" sender: self];
            return;
        }
        questionsAsked++;
        showContinue = false;

        // clear the red or green colors for this section
        if (problemIsFillIn) {

            self.userInput.textColor = nil;
            if (!responseIsCorrect)
                [self.tableView cellForRowAtIndexPath:[NSIndexPath indexPathForRow:1 inSection:2]].textLabel.
                    textColor = nil;
        } else {

            for (int i = 0; i < 4; i++) {
                [[self.tableView cellForRowAtIndexPath:[NSIndexPath indexPathForRow:i inSection:2]].textLabel
                    setBackgroundColor : [UIColor clearColor]];
            }
        }

        [self nextQuestion];
        [self.tableView reloadData];
    } else {

        // quit the game
        UIAlertController* alert = [UIAlertController alertControllerWithTitle:@"Are you sure to quit the game?"
                                                                        message:@""
                                                                 preferredStyle:UIAlertControllerStyleAlert];

        UIAlertAction* yes = [UIAlertAction
                              actionWithTitle:@"YES"
                              style:UIAlertActionStyleDefault
                              handler:^(UIAlertAction * action)
                              {
                                  [self performSegueWithIdentifier:@"quitGame" sender:nil];
                              }];
        UIAlertAction* no = [UIAlertAction
                             actionWithTitle:@"NO"
                             style:UIAlertActionStyleDefault
                             handler:^(UIAlertAction * action)
                             {}];
        [alert addAction:no];
        [alert addAction:yes];
        [self presentViewController:alert animated:YES completion:nil];
    }
}
```

Figure D. Handling the situation where the “Continue” or “Quit Game” button is pressed



```
@Override
public Map<String, Object> loginUser(String loginEmail, String password, String deviceToken, int deviceType,

    LoginInfoDTO loginInfo = new LoginInfoDTO();
    loginInfo.setLoginName(loginEmail);
    loginInfo.setPassword(password);
    loginInfo.setRememberMe(true);

    ResponseEntity<Void> result = accountHelper.login(loginInfo, response);
    User user = userManager.findByLogin(loginEmail);

    // find user id
    QueryTerm term = UserReferenceDao.Field.XREF.getQueryTerm(user.getXref());
    int userId = userReferenceDao.findObject(term).getId();

    // update device token
    try {
        term = MobileDeviceDao.Field.USER_ID.getQueryTerm(userId); // TDDO: compare deviceType as well
        MobileDevice device = mobileDeviceDao.findObject(term);
        if (!device.getToken().equals(deviceToken)) {
            // token has changed
            device.setToken(deviceToken);
            mobileDeviceDao.update(device.getId(), device);
        }
    } catch (NotFoundException e) {
        // device has not been registered before
        MobileDevice device = new MobileDevice();
        device.setUserId(userId);
        device.setToken(deviceToken);
        device.setDeviceType(deviceType);
        mobileDeviceDao.persist(device);
    }

    Map<String, Object> feedback = new HashMap<String, Object>();
    Map<String, Object> info = new HashMap<String, Object>();

    info.put("userId", userId);
    info.put("displayName", user.getDisplayName());
    feedback.put("message", info);
    return feedback;
}
```

Figure E. Server side code dealing with user login



```java
@Override
public Map<String, Object> findAssistmentsProblems(int problemSetId, int count) {

    Random rand = new Random();
    Map<String, Object> requestedObject = new HashMap<String, Object>();
    List<ProblemRow> problemList = new ArrayList<ProblemRow>();

    try {
        int headSectionId = sequenceManager.findHeadSectionBySequenceId(problemSetId).getId();
        List<Integer> astIdList = sequenceManager.findAllAssistmentsIdsByHeadSectionId(headSectionId);

        String astIdListDtr = "";
        for(int astId : astIdList){
            astIdListDtr += "," + String.valueOf(astId);
            ProblemRow temp = sequenceManager.findProblemByAssistmentId(astId);
            problemList.add(temp);
        }
        astIdListDtr = astIdListDtr.substring(1);
    }catch(Exception e){
        requestedObject.put("1", "empty");
        return requestedObject;
    }

    int i = 0;
    while (i < count) {
        if (problemList.size() == 0) break;

        int idx = rand.nextInt(problemList.size());
        ProblemRow problem = problemList.get(idx);
        problemList.remove(idx);

        String problemType = getProblemTypeByTypeId(problem.getRubyProblemType().getId());

        List<AnswerRow> answerList = new ArrayList<AnswerRow>();
        try{
            answerList = sequenceManager.findAllAnswersByProblemId(problem.getId());
        }catch(Exception e){
            //e.printStackTrace();
            continue;
        }

        // must contains >=4 answers
        if (answerList.size() < 4) continue;
```

Figure F. Server side code achieving random problems for a given problem set