# Trello Fellows

# Iteration 2 Summary Document

**Current Implementation Status:**

- Login and Register UC:
  - Fleshed out and fully-developed
  - Username and password formatting validation
  - Checks the database to validate that new information is unique or to validate the correlation of username and password
- Add Exercise UC:
  - Fleshed out and well-developed
  - Allows user to add exercise with name/description
  - Records that exercise in the user's exercise list in the database and locally
  - Plans to have a few premade exercises that users can choose from alongside creating their own
  - Would like to filter and sort the exercise list as well
- Record Weight UC:
  - Implemented but with plans to expand
  - Currently can change the user's weight (on the profile page)
  - Would like to be able to see the history of weight over a period of time
  - Not yet connected to database
- Other UI:
  - Homescreen (before log-in): ability to log-in and register
  - User Menu Screen (after log-in): welcome screen with menus to get to other screens (profile, exercise log, class list)
  - Profile Screen: will contain the dashboard of current stats with the ability to see histories of those stats and upcoming class
  - Class List Screen: will contain the classes that the user is registered for and the ability to search and register for more classes
- Database:
  - Registering and adding an exercise add information to the database
  - Remote access on Admin's machines

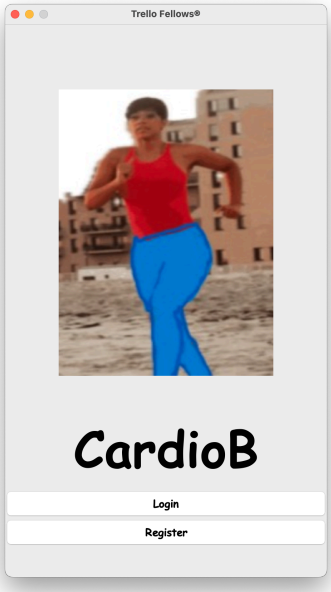
**Planned Scope:**

- User would be able to see history and graphs of logged statistics such as sleep, water intake, weight, caloric intake, and
- User would be able to search, register for, and attend courses and exercise plans
- Trainer would be able to create (and host) courses and exercise plans
- Admin would be able to add new users in their view of the application
- Would like to optimize the user experience with more clear instructions for using the app

**Main Roadblocks:**

- We spent a lot of time discussing SOLID methods which took away from our coding time, making hopefully a more well-developed structure behind the scenes but a less clean visualization.
- Our database setup requires connection to a server which makes testing trickier to implement.
- We also have sometimes found it difficult to maintain the same vision for the direction of the project and are also unsure how to implement coding standards that make sense for everyone.

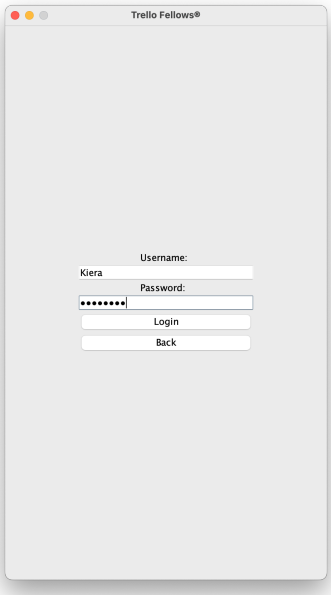
**UI's**

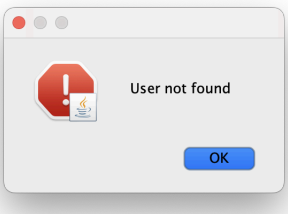


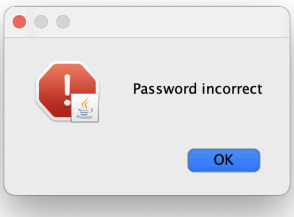

Welcome Screen: No particular use case

Created by: Josh and Noah

Connected: Yes



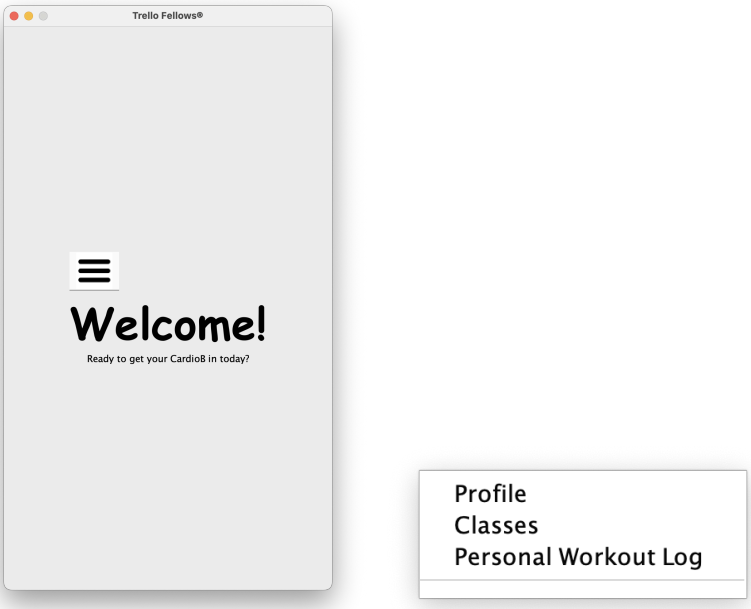




Login Screen: UC-U.ACCT-03, Josh

Created by: Josh and Kiera

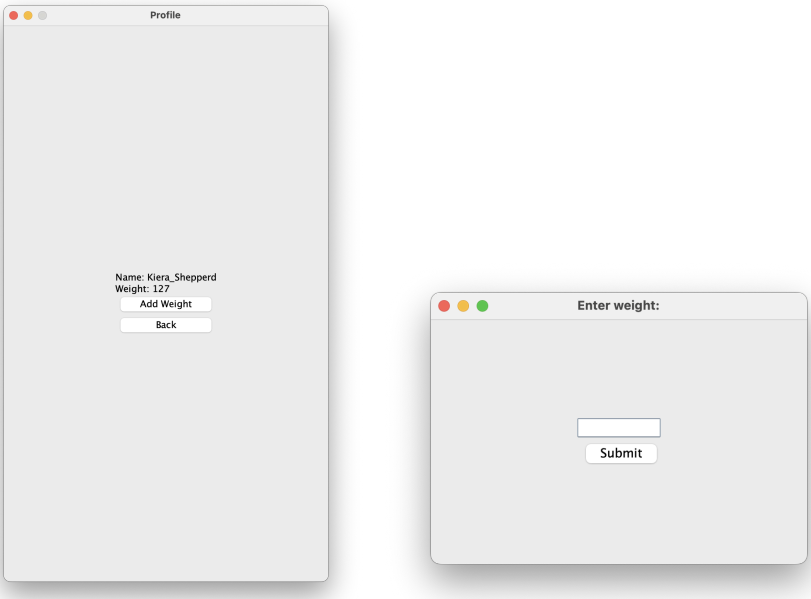
Connected: Yes




(Post-login) Homescreen: no particular use case

Created by: Emily and Lawson

Connected: Yes




Profile Screen: UC-TRACK-02, Kiera

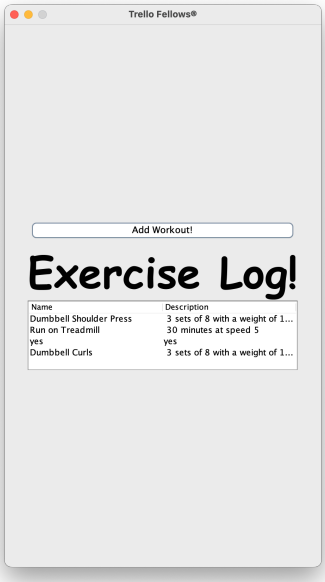
Created by: Kiera

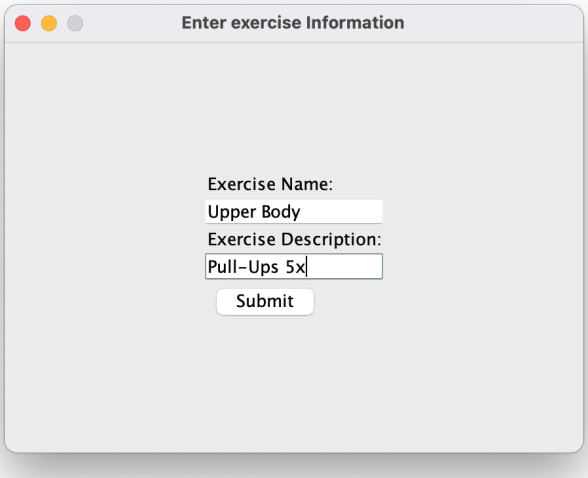
Connected: Yes


Class List Screen: UC-U.CLASS-02, Lawson

Created by: Emily and Lawson

Connected: No

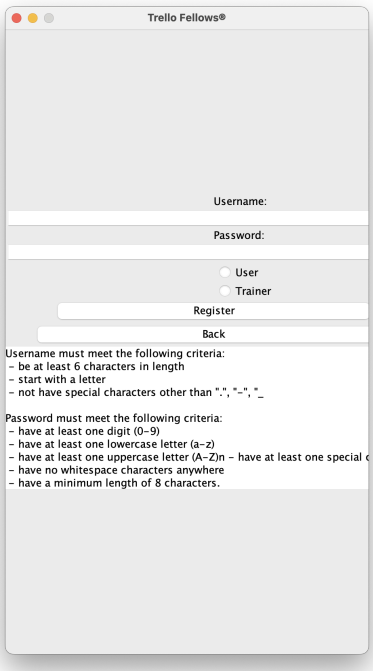


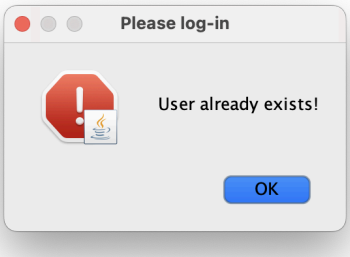



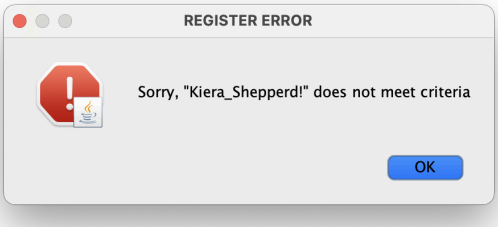

Exercise Log Screen: UC-TRACK-01, Emily

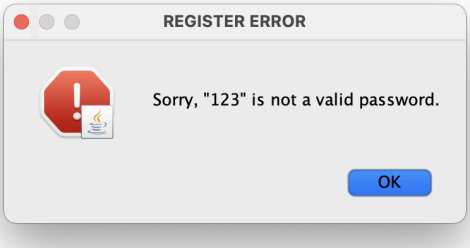
Created by: Lawson, Noah, and Emily

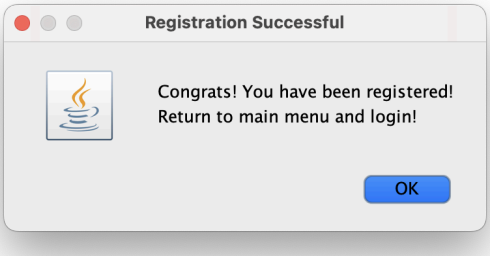
Connected: Yes



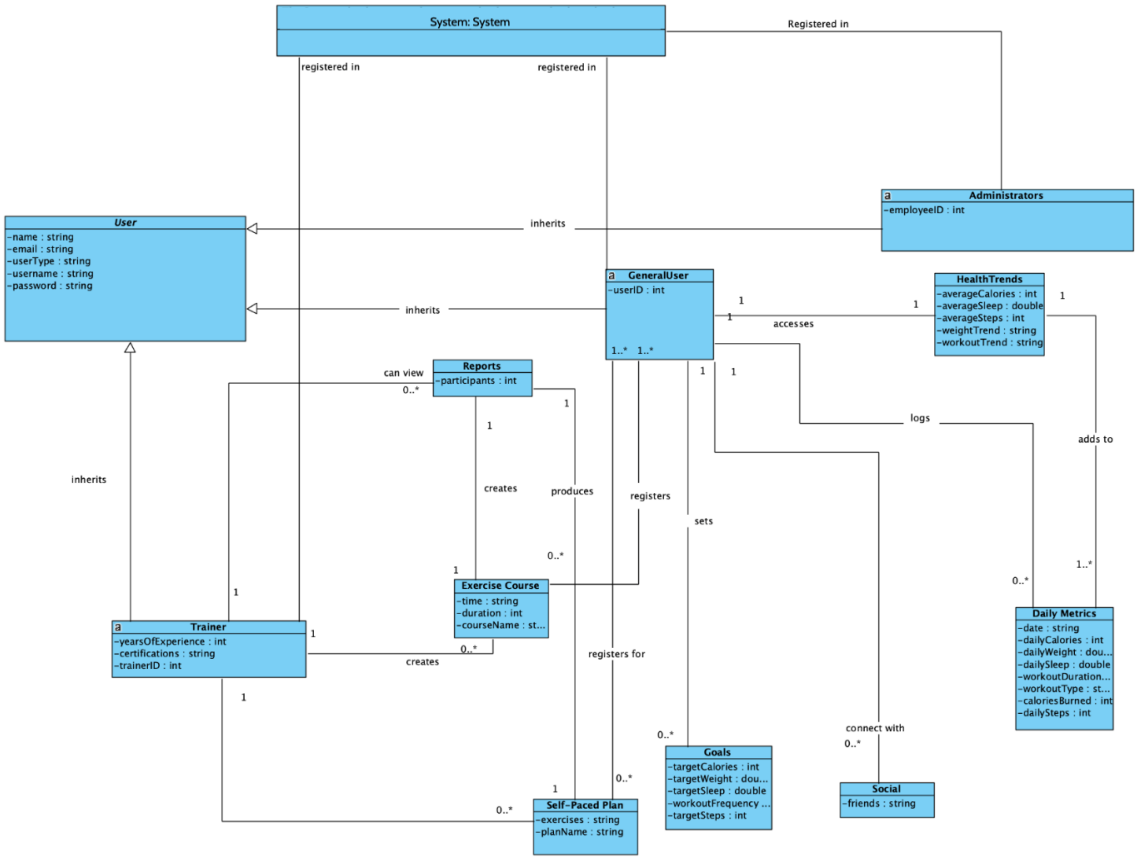




Register Screen (with success and failure dialogs): UC-U.ACCT-01, Noah

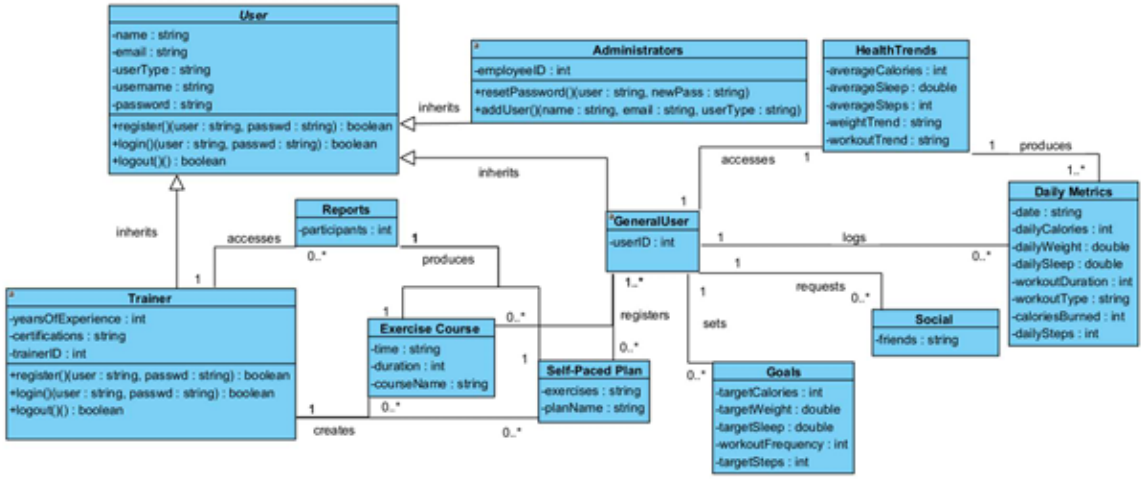
Created by: Josh, Kiera, and Carter

Connected: Yes

Domain Model




Design Class Diagram



**Author:** Noah Mathew
**ID:** UC-U.ACCT-01
**Name:** Create User Account
**Scope:** App Account Creation System
**Level:** User Goal
**Primary Actor:** User

**Stakeholders and interests:**

- Customer
    - User of the health app who is interested in tracking their health data, joining classes, and setting goals

**Precondition:**

- The user has downloaded the application
- The user has an internet connection
- The user has a valid email address

**Postcondition:**

- A new username and password is stored in the system
- System stores their preferences/personal health information that was filled out during account setup
- The user is able to login to their account again using the same credentials
- The user is able to reset their account password with the linked email address

**Minimal Guarantee:**

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must finish set up to create an account
- The system should provide an error message explaining what went wrong

**Main success scenario:**

1. The app should display a login page, with the option to create an account
2. After clicking the create account button, app will display fields for an email and password
3. The user enters information into the prompted fields
4. The system should evaluate whether the email is valid
5. The system should evaluate whether the password meets security requirements

6. The system should check whether the email is already in the database
7. If everything looks good, the system should send an email to the user's email address to verify it is correct
8. The user is able to click the verification email and be led back to the app
9. The user only then be able to login with that email address/username entered
10. Once validated login, the user fills out personal health information and other data or goals
11. The system is able to save the data the user fills out
12. The user is able to log out and log back into the same or different account

**Alternate paths**:

3a. The user doesn't fill in all the fields during the account creation process
   i. The system will prompt the user to enter an email and password

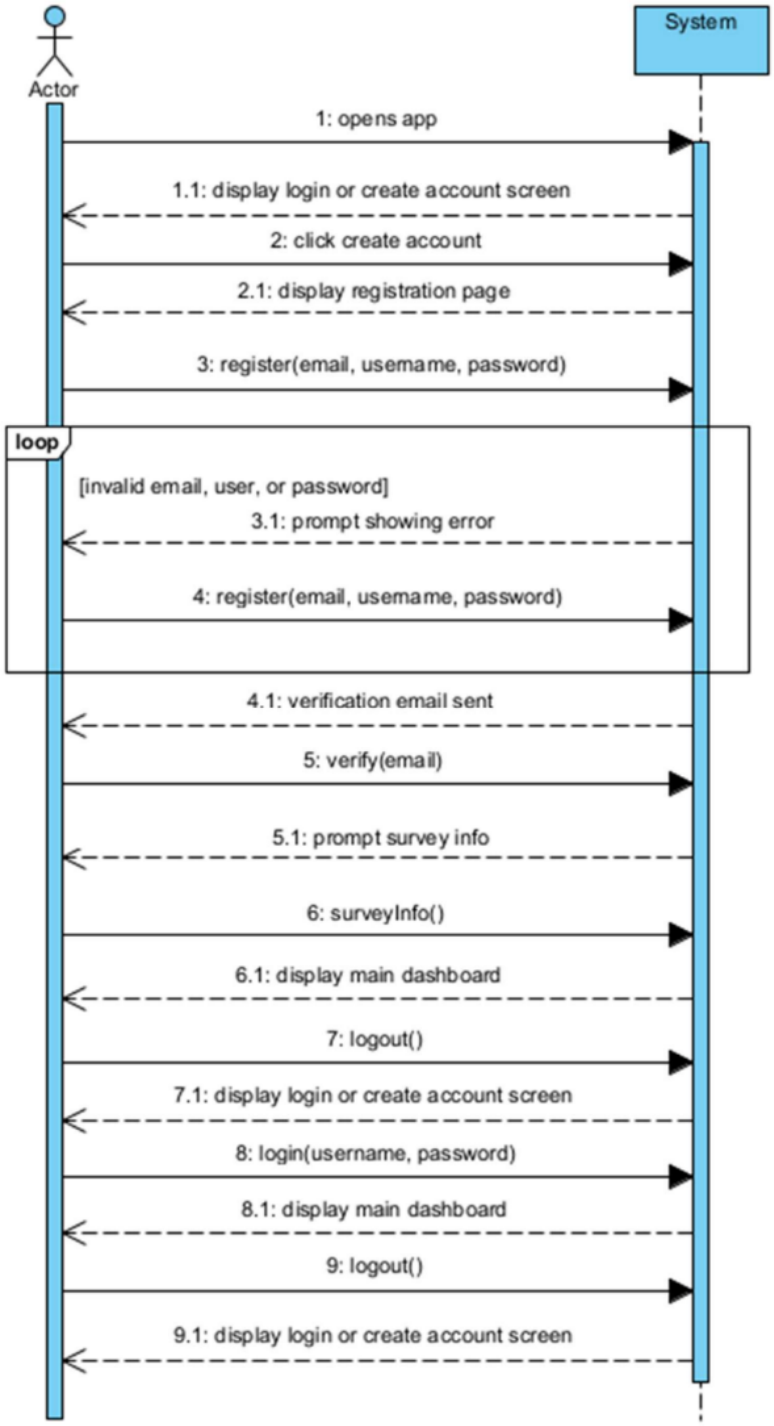
4a. Invalid email format is inputted
   i. The system will prompt the user to re-enter the email address

4b. The email entered is already in the system
   ii. The system will prompt the log-in scree

5a. The password entered isn't secure
   i. The system will prompt the user to re-enter a password that meets the security requirements, which will also be listed

8a. The user does not receive a verification email
   i. The user is able to request another be sent
   ii. The user can create a new account

10a. The user enters no data during the survey process
   i. The user can fill out the information later


Actor
System
1: opens app
1.1: display login or create account screen
2: click create account
2.1: display registration page
3: register(email, username, password)
loop
[invalid email, user, or password]
3.1: prompt showing error
4: register(email, username, password)
4.1: verification email sent
5: verify(email)
5.1: prompt survey info
6: surveyInfo()
6.1: display main dashboard
7: logout()
7.1: display login or create account screen
8: login(username, password)
8.1: display main dashboard
9: logout()
9.1: display login or create account screen

**Operation: register(email, username, password)**

Cross References

- Use Case: User Account Creation

Preconditions:

- none

Postconditions

- User instance *user* was created (object creation)
- *user.username* was set to username (attribute modified)
- *user.email* was set to email (attribute modified)
- *user.password* was set to password (attribute modified)
- *user.emailVerified* was set to false (attribute modified)

Exceptions

- None

**Operation: verify(email)**

Cross References

- Use Case: User Account Creation

Preconditions:

- none

Postconditions

- *user.emailVerified* was set to true (attribute modified)

Exceptions

- None

**Operation: surveyInfo()**

Cross References

- Use Case: User Account Creation

Preconditions:

- A User exists

Postconditions

- The User's profile was updated with survey information (attributes modified)

Exceptions

- None

**Operation: logout()**

Cross References

- Use Case: User Account Creation

Preconditions:

- User is logged in

Postconditions

- The User's session was terminated (association broken)
- *user.loggedIn* was set to false (attribute modified)

Exceptions

- None

**Operation: login(username, password)**

Cross References

- Use Case: User Account Creation

Preconditions:

- none

Postconditions

- A new session is created and associated with the User
- *user.loggedIn* was set to true (attribute modification)

Exceptions

- None

**Author:** Noah Mathew
**ID:** UC-U.ACCT-02
**Name:** Reset Password
**Scope:** User Actions
**Level:** User Goal
**Primary Actor:** User

**Stakeholders and interests:**

- User
    - User of the health app who is interested in gaining access to their account and being able to use its features
- System Admin
    - Employees interested in making sure users have support when trying to reset their password and running into issues

**Precondition**

- The user has an existing account with the app
- The user has a stable internet connection
- The user has access to the email address they registered their account with

**Postcondition**

- The user's password is reset to a new password they choose successfully
- The user is able to login to their account with the new password
- The system maintains the user's old data with the new password stored
- The user gets an email that their password has been reset

**Minimal Guarantee**

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must finish set up to create an account
- The system should provide an error message explaining what went wrong

**Main success scenario**

1. The user opens the app and navigates to the login page
2. The user clicks on Forgot Password button
3. The system prompts the user to provide the email they registered with
4. The system verifies that it is a registered email within the app
5. The system generates a reset password link and email that is sent to the corresponding email address
6. The user opens the email and clicks the reset password link
7. The link brings the user back to the app within a page to enter a new password
8. The user enters a new password that meets security requirements
9. The system updates the password in the database
10. The system emails the user that their password was reset
11. The user returns to the login page and logs in with their new password

**Alternate paths**

4a. The email address entered on the forgot password page is not associated with an account,

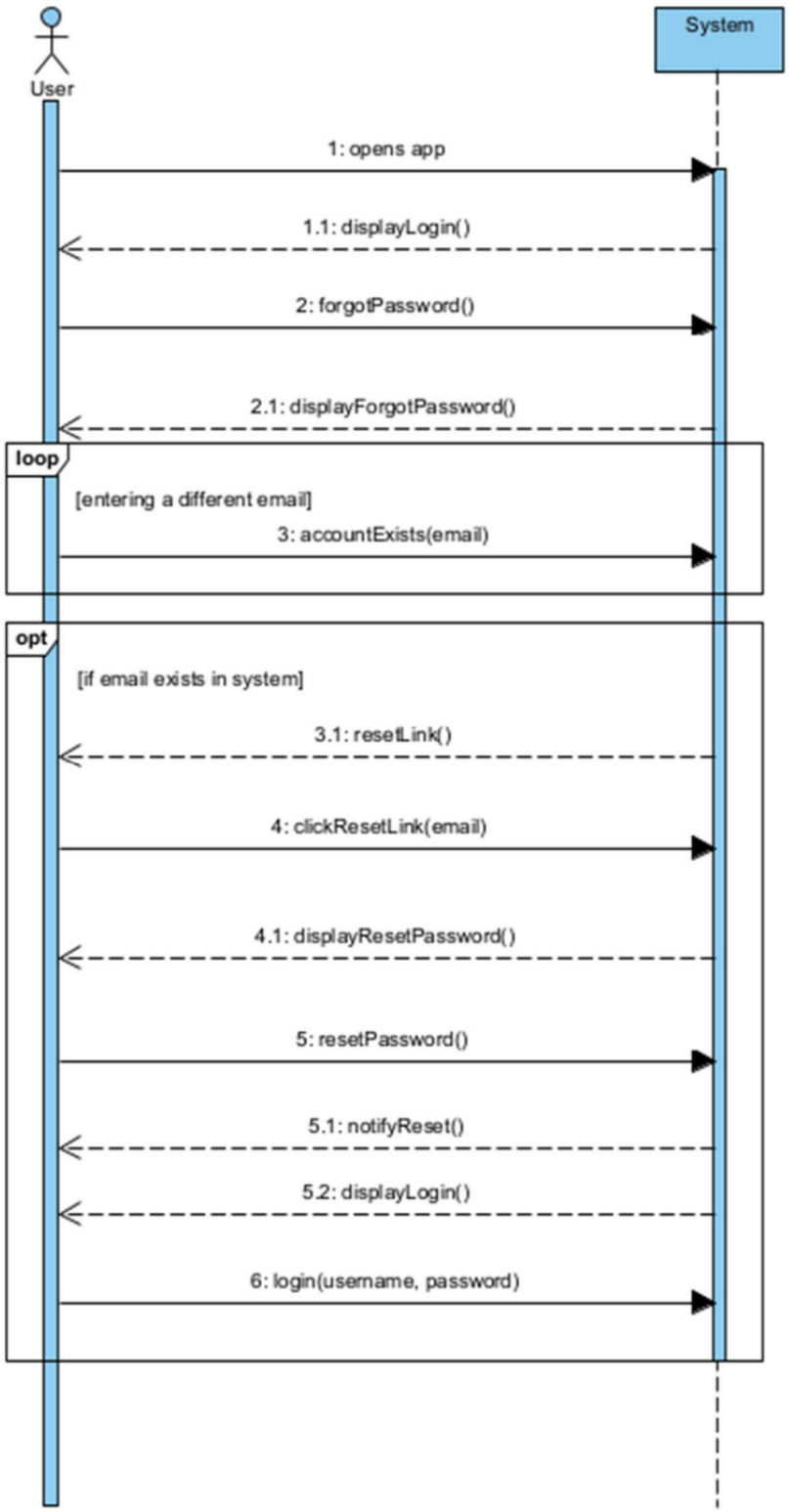
i. the system will re-prompt the user to enter a different email address

6a. The user does not receive the reset password email,

i. they can re-enter their email address

8a. The user's password does not meet security requirements

i. the system will prompt the user with the password requirements


User
System
1: opens app
1.1: displayLogin()
2: forgotPassword()
2.1: displayForgotPassword()
loop
[entering a different email]
3: accountExists(email)
opt
[if email exists in system]
3.1: resetLink()
4: clickResetLink(email)
4.1: displayResetPassword()
5: resetPassword()
5.1: notifyReset()
5.2: displayLogin()
6: login(username, password)

**Operation: accountExists(email: String)**

Cross References

- Use Case: User Reset Password

Preconditions:

- User provides an email address

Postconditions

- None

Exceptions

- Invalid email format

**Operation: sendResetLink(email: String)**

Cross References

- Use Case: User Reset Password

Preconditions:

- System has a registered user with that email

Postconditions

- A reset link is created and associated with user
- A reset request timestamp is associated with user

Exceptions

- None

**Operation: validateNewPassword(password: String)**

Cross References

- Use Case: User Reset Password

Preconditions:

- User has entered a new password

Postconditions

- *user.password* was changed to password (attribute modification)

Exceptions

- None

**Operation: notifyReset(email: String)**

Cross References

- Use Case: User Reset Password

Preconditions:

- User successfully reset password

Postconditions

- None

Exceptions

- None

**Operation: validateLogin(email: String, user: String, password: String)**

Cross References

- Use Case: User Reset Password

Preconditions:

- User has successfully reset their password

Postconditions

- A session is created and associated with the User

Exceptions

- None

**Author:** Josh Fulton
**ID:** UC-U.ACCT-03
**Name:** User Log-In
**Scope:** App Security System
**Level:** User Goal
**Primary Actor:** User

**Stakeholders and interests:**

- User
    - Wants to access account that was created using a username and password

**Precondition**:

- The account is created

**Postcondition:**

- The user gains access to the account

**Minimal Guarantee:**

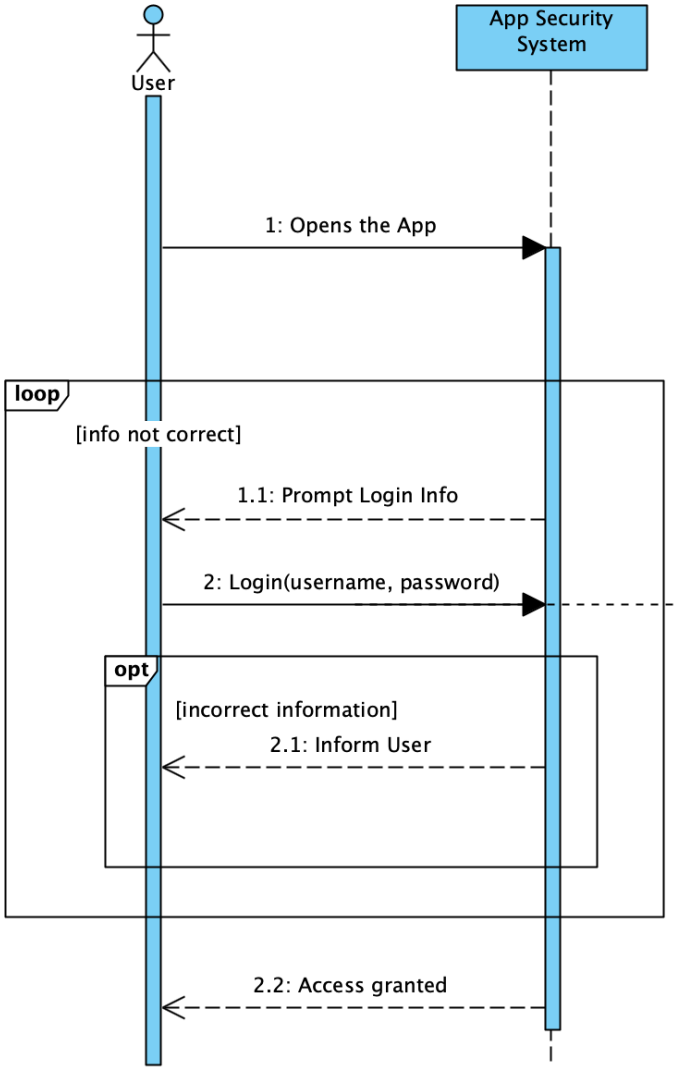
- The account is still existent
- The user can attempt to login after
- The login information for the account has not changed

**Main Success Scenario:**

1. The app prompts the user to enter a username and password
2. The user enters the correct username and password
3. The system verifies that the information provided is attached to an account and is correct
4. The user gains access to the account
5. The user logs out of the account

**Alternate Paths**:

2a. The user enters the incorrect information.
    i. The system notifies the user
    ii. The user is able to attempt again
3a. The system finds that the username is correct but the password is not
    i. The system notifies the user of this error
    ii. The user is able to attempt again




**Operation: Login(username: string, password: string)**

Cross-References:

- User Creates an Account
- Admin Resets Password

Preconditions:

- Account exists
- Username and password are correct

Postconditions:

- A new session is created and associated with the user

Exceptions:

- IncorrectPassword if the password is incorrect

**Author:** Carter Lewis
**ID:** UC-T.ACCT-01
**Name:** Create Trainer Account
**Scope:** Website System
**Level:** User Goal
**Primary Actor:** Trainer

**Stakeholders and interests:**

- Trainer
    - Create an account with affiliated username and password
    - Have a secure password, which is hashed upon entry (SHA-256)

**Precondition**

- System is fully functional with the ability to add user profiles and accounts
- Trainer has valid email address
- The trainer has opened the application

**Postcondition**

- Trainer is logged into his/her account, able to start training routines for users who follow them

**Minimal Guarantee**

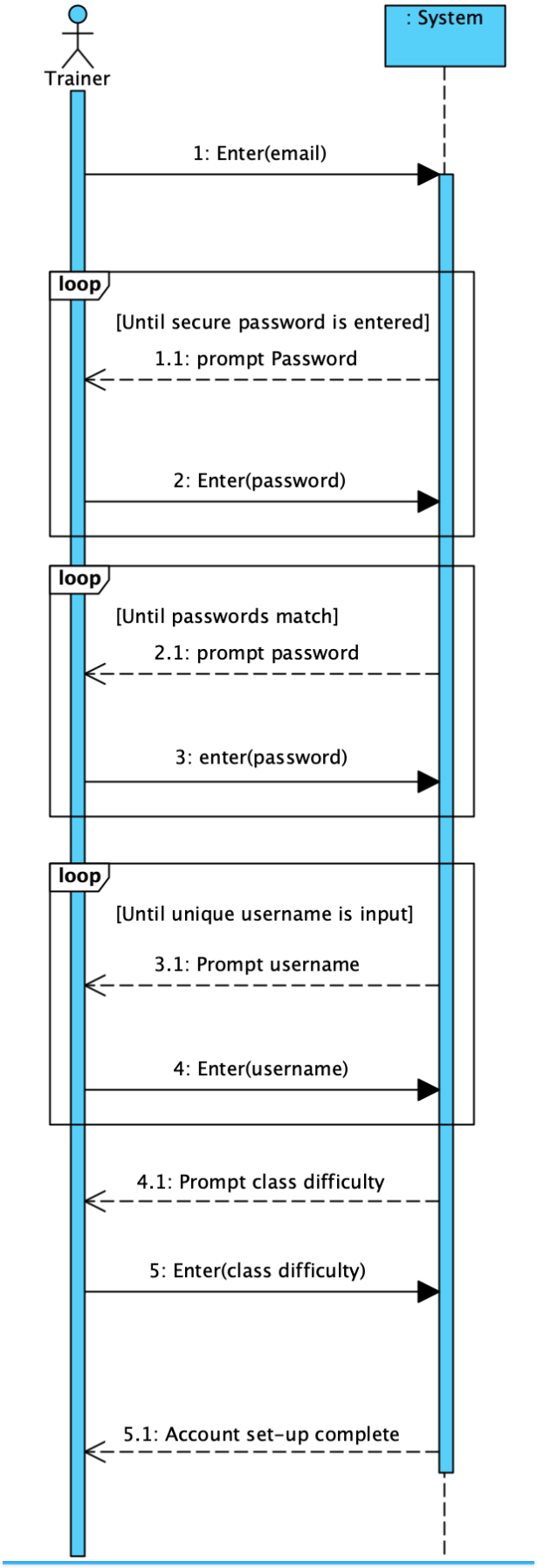
- The security of existing user/trainer accounts will not be compromised

**Main success scenario**

1. New trainer signs up for account using unique email
2. New trainer sets a password
3. Password is deemed secure by our password-checking algorithm
4. Password is hashed using SHA-256 hashing algorithm
5. Password is stored in database and affiliated with the new user's email
6. User is brought to an "account set-up" page to enter basic details about themself
7. These details are stored in their account
8. Trainer is brought to the home view of the website

**Alternate paths**

1a. New trainer enters existing email
    i. Trainer is given the option to log in to existing account or use a different email
3a. Password is NOT deemed secure
    ii. Trainer is asked to enter a new password until all security requirements are met


Trainer
: System
1: Enter(email)
loop
[Until secure password is entered]
1.1: prompt Password
2: Enter(password)
loop
[Until passwords match]
2.1: prompt password
3: enter(password)
loop
[Until unique username is input]
3.1: Prompt username
4: Enter(username)
4.1: Prompt class difficulty
5: Enter(class difficulty)
5.1: Account set-up complete

**Operation: Enter(email)**

Cross References

- Use Case: Trainer Account Creation

Preconditions:

- none

Postconditions

- new trainer instance *trainer* was created (instane created)
- *trainer.email* was set to valid email (attribute modification)

Exceptions

- None

**Operation: Enter(Password)**

Cross References

- Use Case: Trainer Account Creation

Preconditions

- Valid email is in the system

Postconditions

- *trainer.password* was set to hashed password (attribue modification)

Exceptions

- none

**Operation: Enter(Username)**

Cross References

- Use Case: Trainer Account Creation

Preconditions

- Valid password is saved in the system

Postconditions

- *trainer.username* was set to hashed password (attribue modification)

Exception

- None

**Operation: Enter(Class difficulty)**

Cross References

- Use Case: Trainer Account Creation

Preconditions

- Valid email, password, and username entered into the systems

Postconditions:

- *trainer.classDiff* was set to class difficulty (attribute modification)

Exception

- None

**Author:** Josh Fulton
**ID:** UC-A.ACCT-01
**Name:** Create and Set Up an Admin Account
**Scope**: App Account Creation System
**Level:** Admin Goal
**Primary Actor:** Admin

**Stakeholders and interests:**

- Admin
    - Admin wants to create an account

**Precondition:**

- The admin has downloaded the application.
- The admin has an internet connection.
- The admin has an email that is part of the list of valid emails that are able to create admin accounts.

**Postcondition:**

- The admin's account is created.
- The admin is able to view all general user and trainer accounts from the admin's account.

**Minimal Guarantee**

- The accounts previously created remain within the system.
- The admin can attempt to create an account again

**Main Success Scenario:**

1. The admin selects create an account.
2. The admin is shown three account types: general user, trainer, and admin.
3. The admin selects "admin".
4. The system provides fields in which the admin must provide information (username, email address, and password).
5. The system checks to see if the account can be an admin account my validating email with a register of emails that are acceptable.
6. The system verifies that the individual can be an admin.
7. The system creates the admin account.
8. The admin is able to login using the information they provided.
9. The admin logs out and exits the application.

**Alternate Paths:**

2a. The admin selects the wrong type of account.

i. They back out of the page they are on and reselect correct option

5a. The admin provides an invalid email. T

i. The system notifies the admin and requests to enter a different email.

8a. The admin misremembers their password.

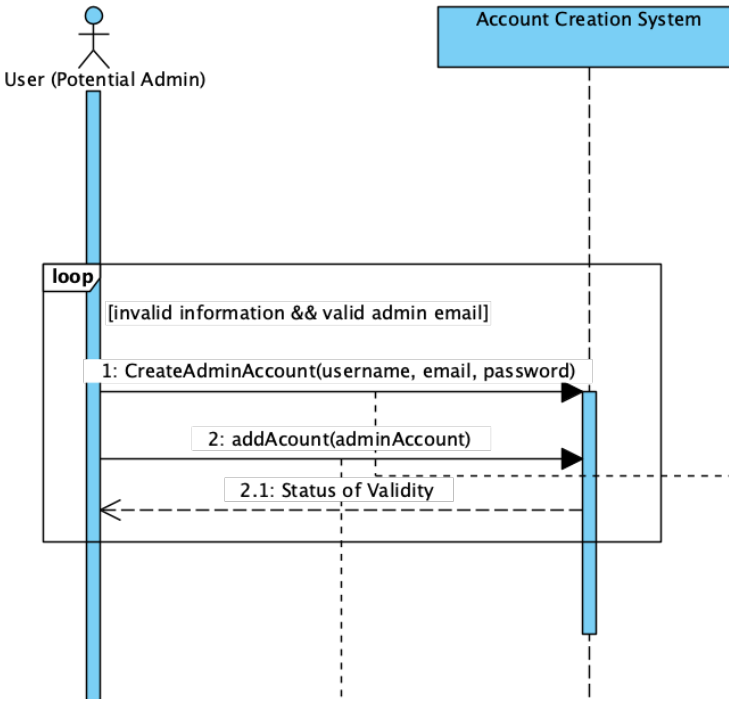
i. They request to reset their password.

ii. After it is reset, the admin logs in

8b. The admin mistypes their email.

i. They are notified that the username is incorrect

ii. The admin is prompted with the log-in screen again


User (Potential Admin)
Account Creation System
loop
[invalid information && valid admin email]
1: CreateAdminAccount(username, email, password)
2: addAcount(adminAccount)
2.1: Status of Validity

**Operation: CreateAdminAccount(username: string, email: string, password: string)**

Cross-references:

- Use case- Create Admin Account

Preconditions:

- none

Postconditions:

- Admin account *admin* was created (object creation)
- *admin.username* was set to username (attribute modified)
- *admin.password* was set to password (attribute modified)

Exceptions: none

**Operation: addAccount(adminAccount: adminAccount)**

Cross references:

- Use case- create Admin Account

Preconditions:

- Admin account exists

Postconditions:

- *admin* is added to the system (association formed)

Exceptions: none

**Author:** Josh Fulton
**ID:** UC-A.ACCT-02
**Name:** Add New User
**Scope**: App Account Creation System
**Level:** Admin Goal
**Primary Actor:** Admin

**Stakeholders and interests:**

- Admin
    - Wants to create an account for a user
- User
    - Wants an account to be created by admin

**Precondition:**

- Admin is logged into an admin account
- User notifies admin of force creation of account wanted

**Postcondition:**

- The admin creates the user account
- The user is able to access the new account

**Minimal Guarantee**

- The accounts previously created remain within the system.
- The admin can attempt to create an account again
- The admin account still exists

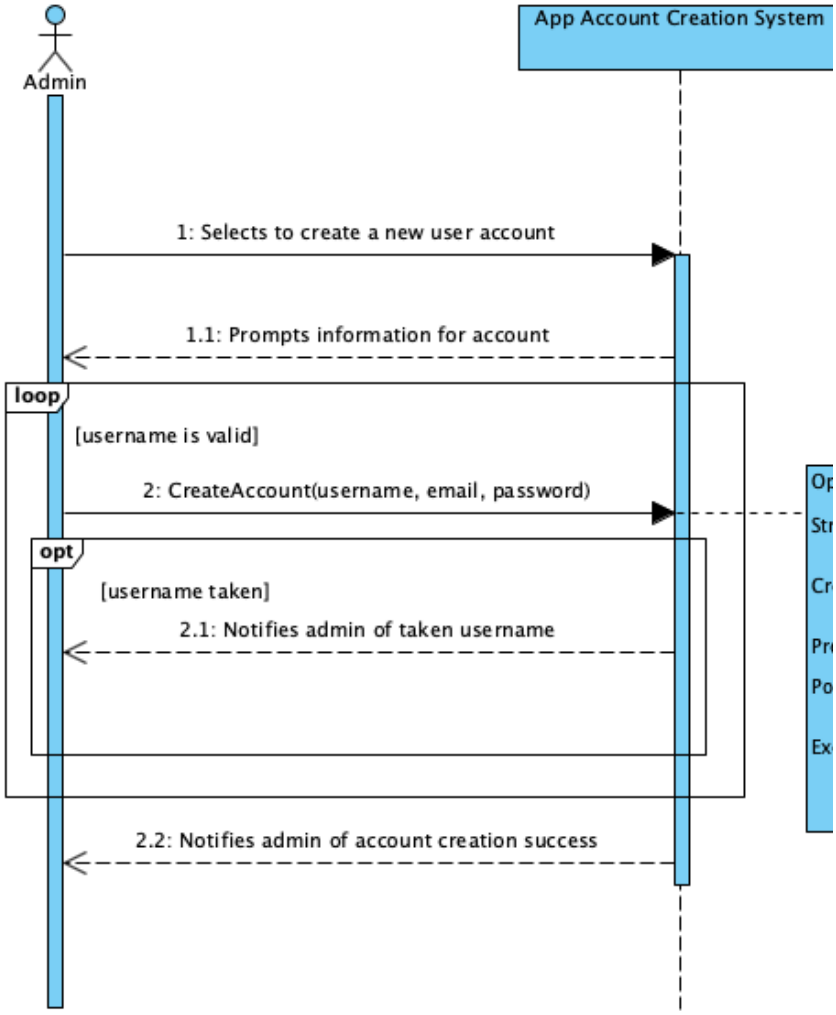
**Main Success Scenario:**

1. The admin selects “create user account”
2. The admin fills in login information for the user account
3. The admin selects “create”
4. The system notifies admin of success
5. The user is able to use the login information to access new account

**Alternate Paths:**

3a. The username the admin chooses is already in use
    i. The admin is able to enter another username
    ii. Repeats process until useable username is entered

4a. Due to unforeseen circumstances, the system says there is a failure and the account could not be created.
    i. The admin can attempt to create the account again


Admin
App Account Creation System
1: Selects to create a new user account
1.1: Prompts information for account
loop
[username is valid]
2: CreateAccount(username, email, password)
opt
[username taken]
2.1: Notifies admin of taken username
2.2: Notifies admin of account creation success

Operation Contract

**CreateAccount(username: String, email: String, password: String)**

Cross References:

- User Creates Account
- Admin Creates User Account

Preconditions:

- Admin account exists

Postconditions:

- User account *user* was created (object creation)
- *user.username* was set to username (attribute modified)
- *user.email* was set to email (attribute modified)
- *user.password* was set to password (attribute modified)

Exceptions:

- UsernameTaken if username is already taken

**Author:** Josh Fulton
**ID:** UC-A.ACCT-03
**Name:** Admin Changes User's Password
**Scope:** App Administrative System
**Level:** Admin Goal
**Primary Actor:** Admin

**Stakeholders and interests:**

- Customer
  - User wants to have their password changed
- System Admin
  - Employee wants to ensure the user has access to their account and updates password to allow user access.

**Precondition**:

- The admin is logged in
- The user has an account
- The user requests for a password change

**Postcondition:**

- The user's password is changed by the Admin
- The user is notified
- The user is able to login after the password reset

**Minimal Guarantee:**

- The account is still able to be access by the admin
- The user's account is not deleted
- The user has not entered eternal rest

**Main Success Scenario:**

1. The administrator navigates to the account of the user who requested the password reset
2. The administrator successfully resets the password
3. The user is notified of the password reset
4. The user is able to login to their account by using the new password

**Alternate Paths**:

1a. The admin is not notified for an unidentified reason and the user asks for another reset.

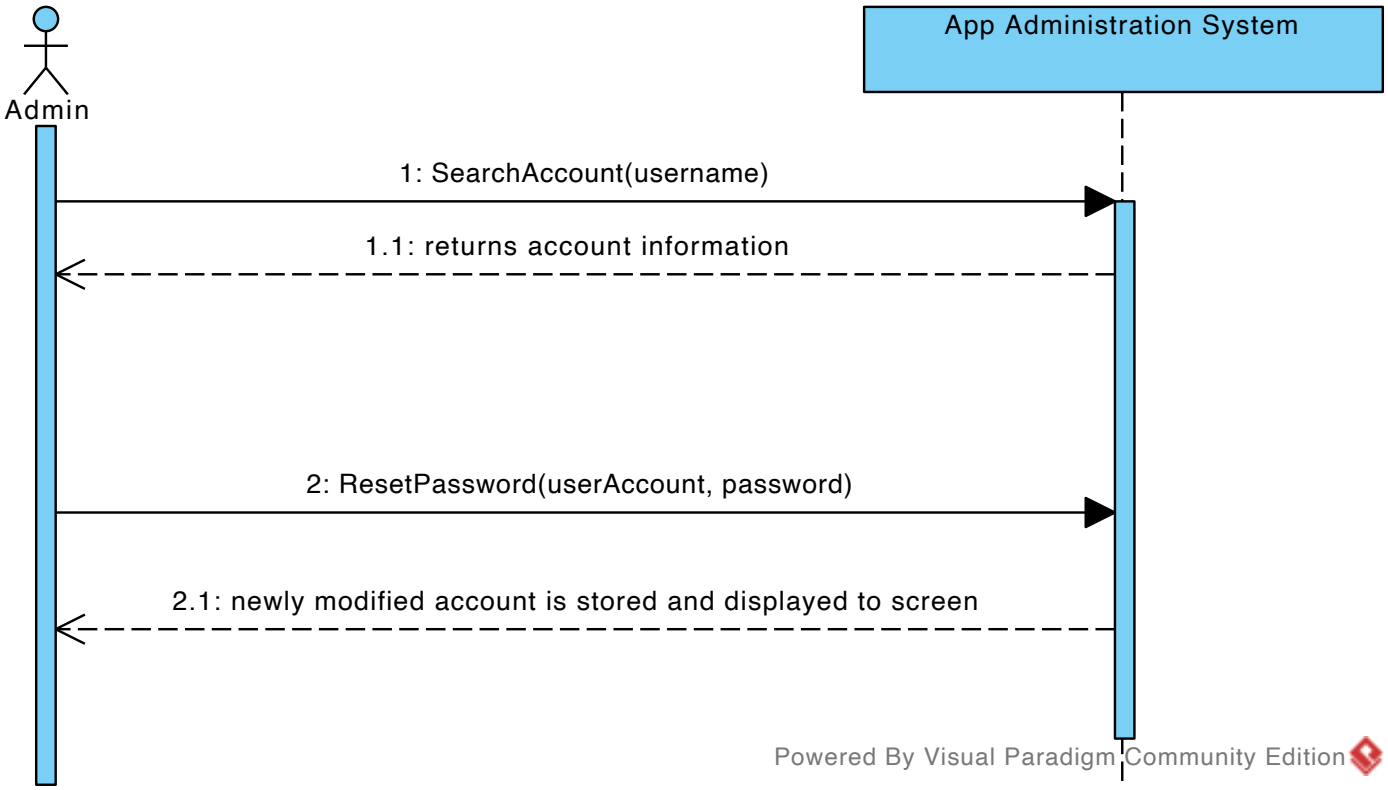
2a. The admin changes the wrong user's password and resets the password of another

user that was not intended.

3a. The admin changes the user's password but it is not the same as told to the user, causing confusion.

3b. The user is not notified and therefore asks for another reset.

4a. The user forgets the new password and asks for another password reset


Admin
App Administration System
1: SearchAccount(username)
1.1: returns account information
2: ResetPassword(userAccount, password)
2.1: newly modified account is stored and displayed to screen
Powered By Visual Paradigm Community Edition

**Operation: SearchAccount(username: string)**

Cross-References:

- User Case - Admin changes user's password

Preconditions:

- The userAccount exists
- The adminAccount exists

Postconditions:

- none

Exceptions:

- InvalidUsername if an account does not exist with the input username

**Operation: RegisterPassword(userAccount: userAccount, password: string)**

Cross-references:

- Use Case - Admin Changes User's password

Preconditions:

- userAccount exists

Postconditions:

- *user.password* was changed to password (attribute modification)

Exceptions:

- SamePassword if input password is same as current password

**Author:** Emily Wokoek
**ID:** UC-TRACK-01
**Name:** UC Add Workout
**Scope:** App Health Data Records System
**Level:** User Goal
**Primary Actor:** User

**Stakeholders and interests:**

- Customer
    - User of the health app (NOT A TRAINER) who is interested in tracking their health data, joining classes, and setting goals
- System Admin
    - Employee interested in making sure users have support when running into issues with account creation

**Precondition:**

- The user has downloaded the application
- The user is logged in

**Postcondition:**

- New user-inputted information has been saved in their personal workout log
- System maintains workout log information that isn't edited
- The user is able to access the newly logged workout

**Minimal Guarantee:**

- The same amount of users is still stored in the system as before
- No corrupted or incorrect data is stored in the system
- If failure occurs, the systems produces an error message

**Main success scenario:**

1. The system presents action options about data and classes
2. The user wishes to record a new workout in their personal workout log and is able to do so
3. The system should evaluate whether the specified inputted values are valid
4. If values are sound, system presents a save option
5. The system saves the information and presents the action options again
6. The user is able to view updated report of their workouts

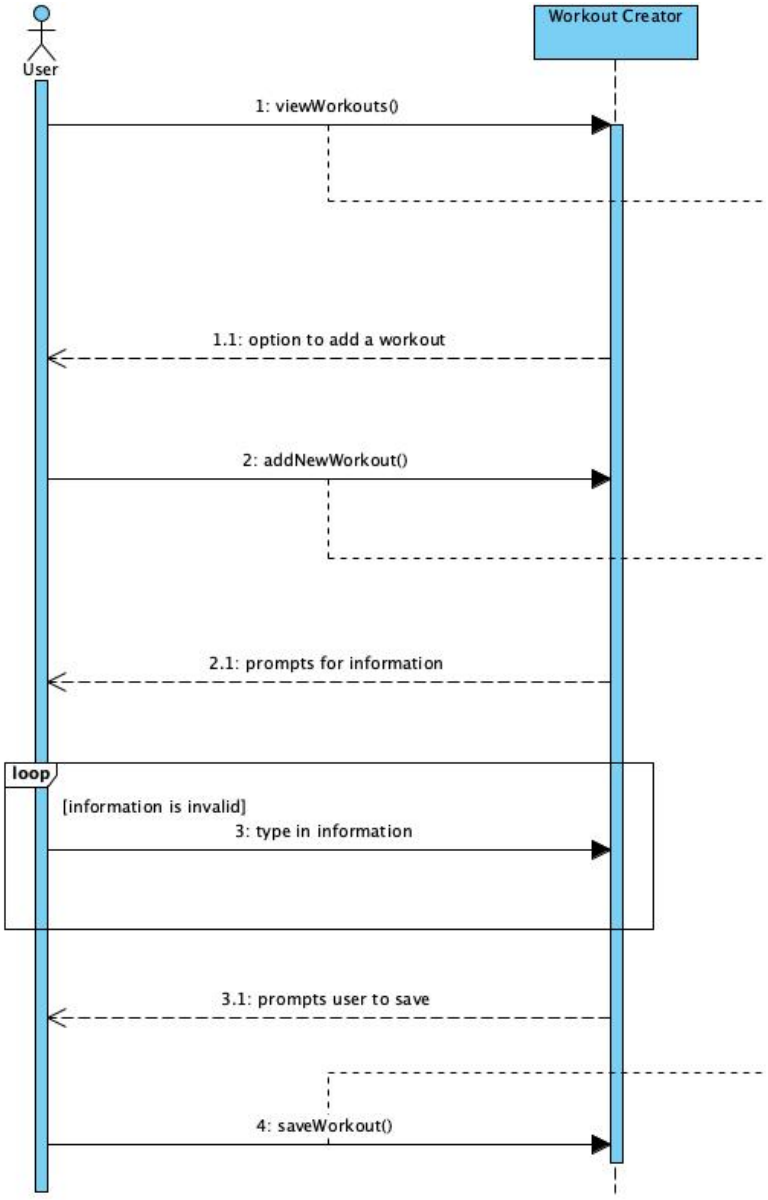
**Alternate paths** :

3a. There is an invalid value inputted

i. The system will prompt the trainer to re-enter the information

3b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


User
Workout Creator
1: viewWorkouts()
1.1: option to add a workout
2: addNewWorkout()
2.1: prompts for information
loop
[information is invalid]
3: type in information
3.1: prompts user to save
4: saveWorkout()

**Operation: viewWorkouts()**

Cross-References:

- Use Case - Add Workout

Preconditions:

- App is opened to home page

Postconditions:

- none

Exceptions:

- none

**Operation: addNewWorkout()**

Cross-References:

- Use Case - Add Workout

Preconditions:

- App is opened to workout log

Postconditions:

- Workout instance *workout* was created (instance creation)

Exceptions:

- none

**Operation: saveWorkout()**

Cross-References:

- Use Case - Add Workout

Preconditions:

- Valid information entered into all relevant fields

Postconditions:

- *workout* attributes are modified was created (attributes modified)
- *workout* associated with *user* (associations created)

Exceptions:

- none

**Author:** Kiera Shepperd
**Name:** UC-TRACK-02 Log Information
**ID:** UC-TRACK-02
**Scope:** App Health Data Records System
**Level:** User Goal
**Primary Actor:** User

**Stakeholders and interests:**

- User
    - User wants to track their health data (such as water consumption, sleep, caloric intake, and weight)

**Precondition:**

- The user is logged in

**Postcondition:**

- New user-inputted information has been saved in their personal log
- System maintains logged information that isn't edited

**Minimal Guarantee:**

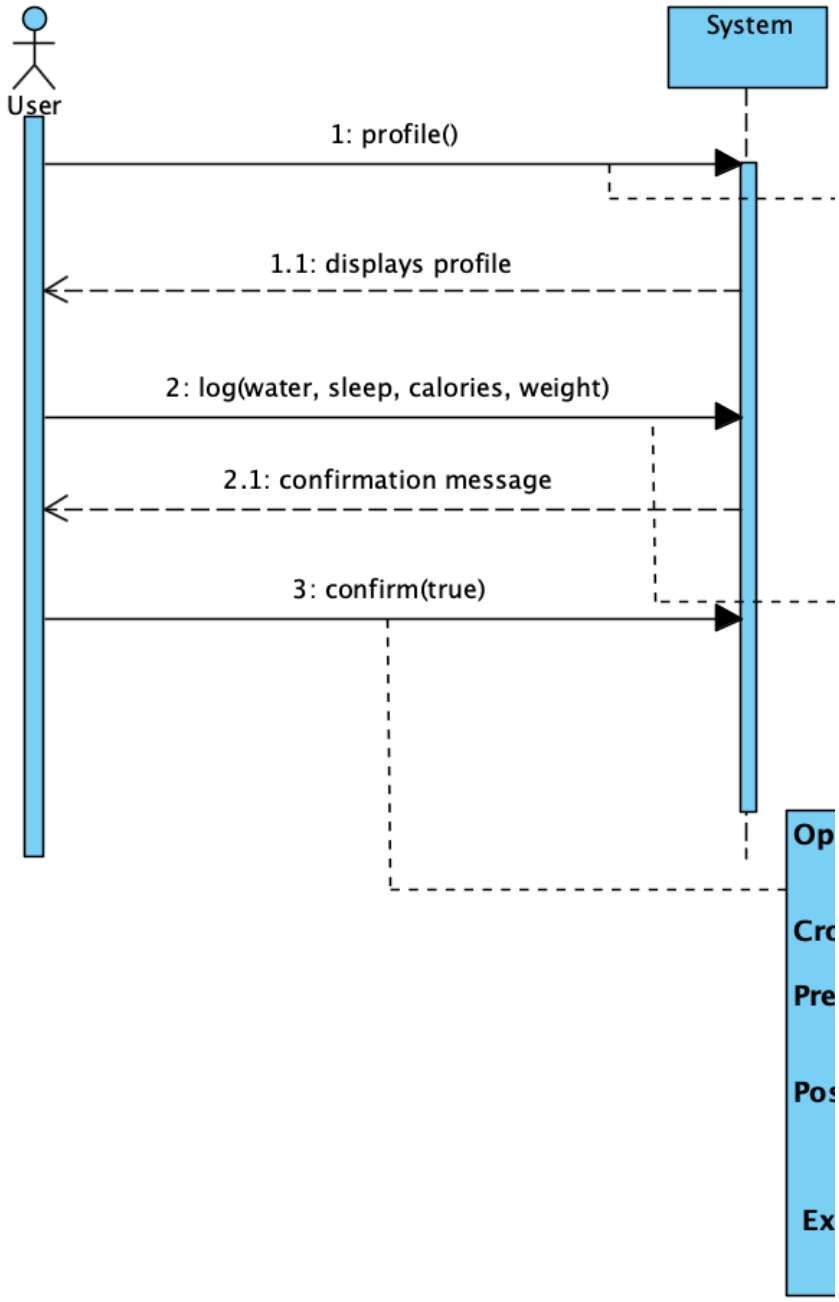
- The same amount of users is still stored in the system as before
- No corrupted or incorrect data is stored in the system
- The system will provide error messaging if something goes wrong

**Main success scenario:**

1. User navigates to their profile
2. User opens the information logging page
3. User inputs information as desired
4. The system should evaluate whether the specified inputted values are valid
5. If values are sound, system presents a confirmation message and a save option
6. The system saves the information and presents the action options again

**Alternate paths:**

5a. A field is left blank
    i. The system will automatically fill the field with a -1 to indicate that the field was left blank
    ii. The system will the store the inputted information as well as any blank fields
5b. There is an invalid value inputted
    i. The system will prompt the user to re-enter the information


User
System
1: profile()
1.1: displays profile
2: log(water, sleep, calories, weight)
2.1: confirmation message
3: confirm(true)
Op
Cr
Pre
Pos
Ex

**Operation: profile()**

Cross-References:

- Use Case - View User Report
- Use Case - Log Information

Preconditions:

- *user* is logged in

Postconditions:

- none

Exceptions:

- none

**Operation: log(double water, double sleep, int calories, double weight)**

Cross-References:

- Use Case - Log Information

Preconditions:

- *user* is logged in

Postconditions:

- A DailyMetrics instance *dm* was created (instance creation)
- *dm* attributes were set to inputted quantities (attribute modification)

Exceptions:

- none

**Operation: confirm(boolean true)**

Cross-References:

- Use Case - Log Information

Preconditions:

- *user* is logged in
- *dm* has information store

Postconditions:

- *dm* was associated with *user* (association formed)

Exceptions:

- none

**Author:** Lawson Hale
**ID:** UC-U.CLASS-01
**Name:** Register for self-paced exercise plan
**Scope:** Self-paced Exercise Plan Choice System
**Level:** User Goal
**Primary Actor:** General User

**Stakeholders and interests:**

- User
  - Wants to register for a self-paced exercise plan a trainer made
- Trainer
  - Wants to create and monitor plans that users use

**Precondition**

- The user is logged in
- There must be plans and data for available plans

**Postcondition**

- The user is registered for a self-paced exercise plan

**Minimal Guarantee**

- The user's information is safe
- The self-paced plan still exists
- The trainer can access the self-paced plan
- No one's personal information is leaked
- No data about the class is corrupted
- The system does not crash

**Main success scenario**

1. The user navigates to the self-paced plan tab
2. The user chooses a self-paced exercise plan from the available list
3. The user views the overview of the plan
4. The user chooses register
5. The system validates that the plan exists
6. The system adds the user to the list of current users in the plan
7. The user now has a view of the plan that has completed dates and specific workout information for all of the plan
8. The trainer can see the users currently using the plan

**Alternate paths**

2a. There are no self-paced plans available

i. The user cannot choose a self-paced plan

3a. The user views a self-paced plan and chooses not to register for it

i. The user can see other plans OR exits the app

6a. The user does not want to be in the self-paced plan anymore

i. The user deregisters

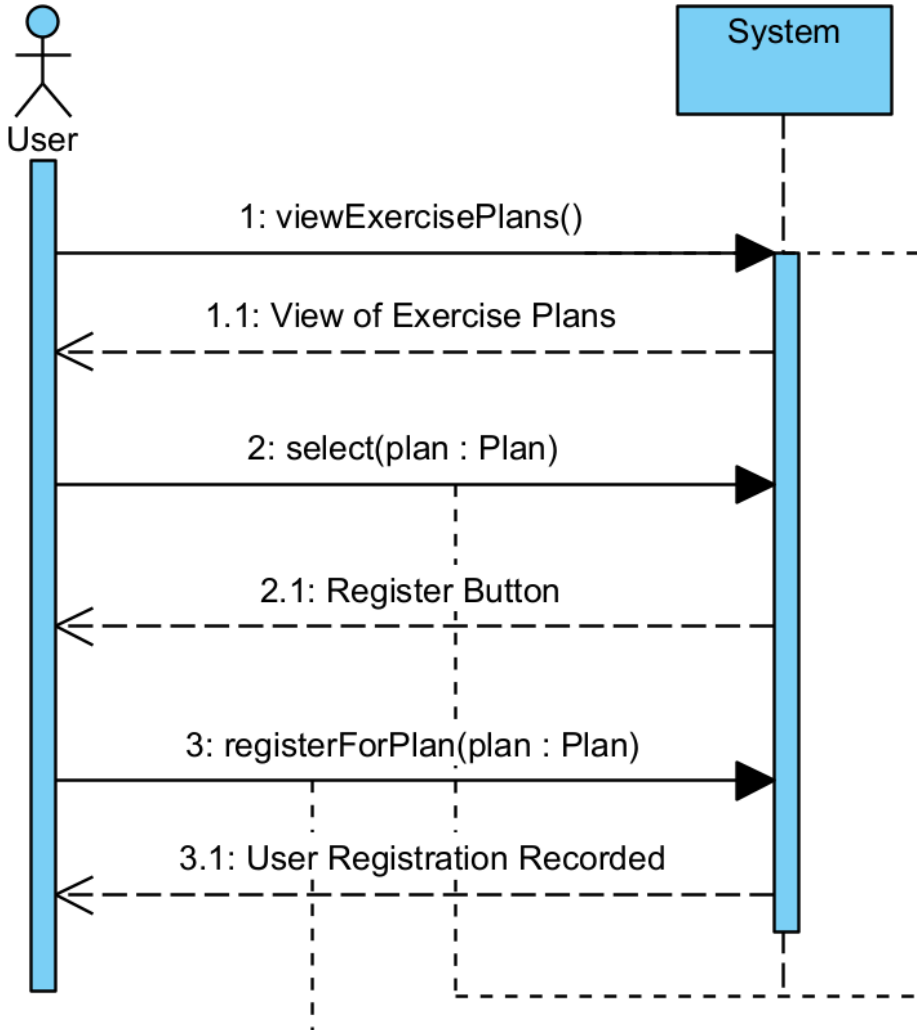
ii. The user is removed from the class list

5a. The user reviews the plan

i. The user can leave 1-5 stars for the self-paced exercise plan

ii. The plan has reviews that average to give it a score

SP Exercise Plan SSD



**Operation: viewExercisePlans()**

Cross-References:

- Use Case - Register for Self-Paced Workout Plan

Preconditions:

- *user* is logged in

Postconditions:

- none

Exceptions:

- NoPlans if there are no plans

**Operation: select()**

Cross-References:

- Use Case - Register for Self-Paced Workout Plan

Preconditions:

- *user* is logged in
- *plan* exists

Postconditions:

- none

Exceptions:

- none

**Operation: registerForPlan(plan: Plan)**

Cross-References:

- Use Case - Register for Self-Paced Workout Plan

Preconditions:

- *user* is logged in
- *plan* exists

Postconditions:

- *plan* was added to the list *user.plans* (association formed)
- *user* was added to the list *plan.roster* (association formed)

Exceptions:

- none

**Author:** Lawson Hale
**ID:** UC-U.CLASS-02
**Name:** Register for Class
**Scope:** Workout Class Choice System
**Level:** User Goal
**Primary Actor:** General User

**Stakeholders and interests:**

- User
    - Wants to register for and mark attendance for a workout class
- Trainer
    - Wants to make sure their class is available and ready

**Precondition**

- The user is logged in
- There must be classes and data for available classes

**Postcondition**

- The user is registered for a class and is on the class roster
- The trainer can see the new registration
- The class can become full if there are no more spaces left

**Minimal Guarantee**

- The user's information is safe
- The workout class still exists
- The trainer can access the workout class
- No one's personal information is leaked
- No data about the class is corrupted

**Main success scenario**

1. The user navigates to the workout class tab
2. The user chooses a workout class from the list available
3. The user hits the register button
4. The system validates that the class has available spots in it
5. The system enters the user into the class roster
6. The user can see the class roster with them in it

**Alternate paths**

2a. There are no workouts available

i. The user cannot choose a workout

4a. The workout class is full

i. The user cannot register for it

ii. The system sends an error message saying the class is full

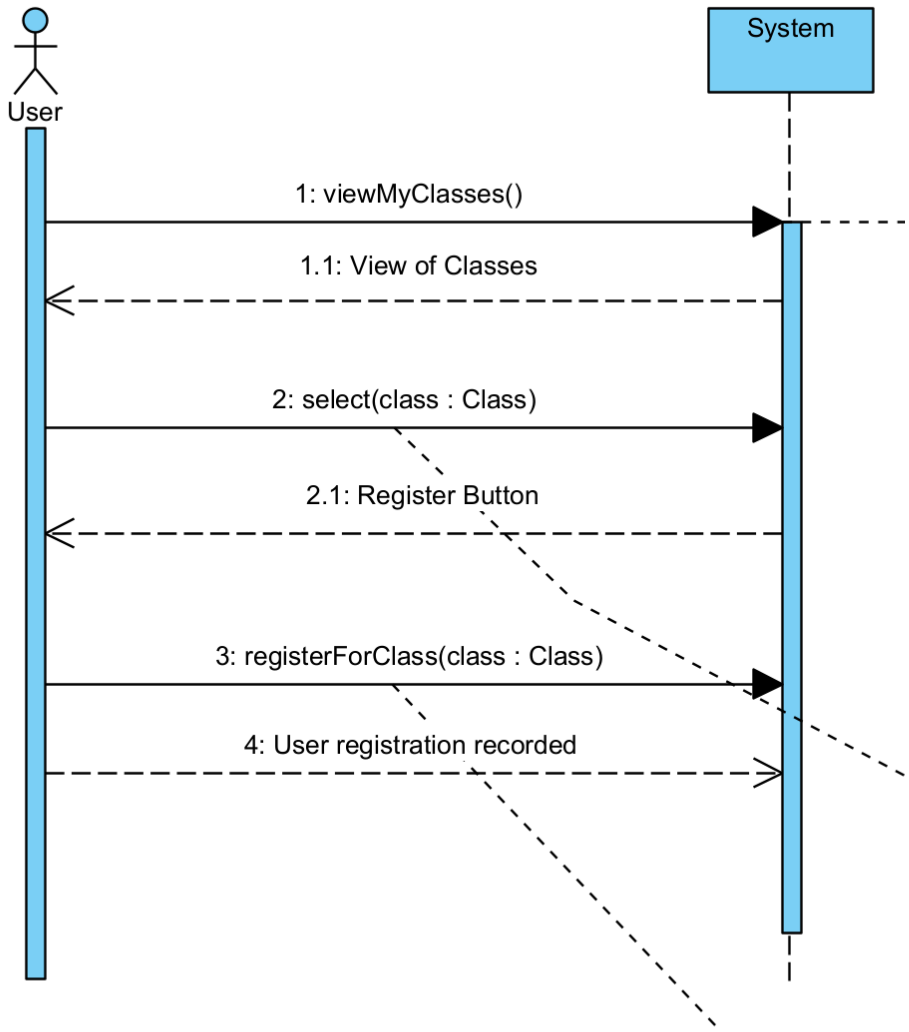
2a. The user views a workout class and chooses not to register for it

i. The user exits the app

6a. The user does not want to be in a class anymore

i. The user deregisters

ii. The user is removed from the class list


User
System
1: viewMyClasses()
1.1: View of Classes
2: select(class : Class)
2.1: Register Button
3: registerForClass(class : Class)
4: User registration recorded

**Operation: viewMyClasses()**

Cross-References:

- Use Case - Register for Workout Class

Preconditions:

- *user* is logged in

Postconditions:

- none

Exceptions:

- NoClasses if there are no classes

**Operation: select()**

Cross-References:

- Use Case - Register for Workout Class

Preconditions:

- *user* is logged in
- *class* exists

Postconditions:

- none

Exceptions:

- none

**Operation: registerForClass(class: Class)**

Cross-References:

- Use Case - Register for Workout Class

Preconditions:

- *user* is logged in
- *class* exists

Postconditions:

- *class* was added to the list *user.classes* (association formed)
- *user* was added to the list *class.roster* (association formed)

Exceptions:

- FullClass if the class is full

**Author:** Lawson Hale
**ID:** UC-U.CLASS-03
**Name:** Attend Workout Class
**Scope:** Workout Class Choice System
**Level:** User Goal
**Primary Actor:** General User

**Stakeholders and interests:**

- User
  - Wants to attend a workout class
- Trainer
  - Interested in making sure their class is available and ready, with a fleshed-out plan and attendees

**Precondition**

- The user is logged in
- The user must be registered for the class

**Postcondition**

- The user has been marked as present for the class

**Minimal Guarantee**

- The user's information is safe
- The workout class still exists
- The trainer can access the workout class
- No one's personal information is leaked
- No data about the class is corrupted
- All previous attendance records are still there

**Main success scenario**

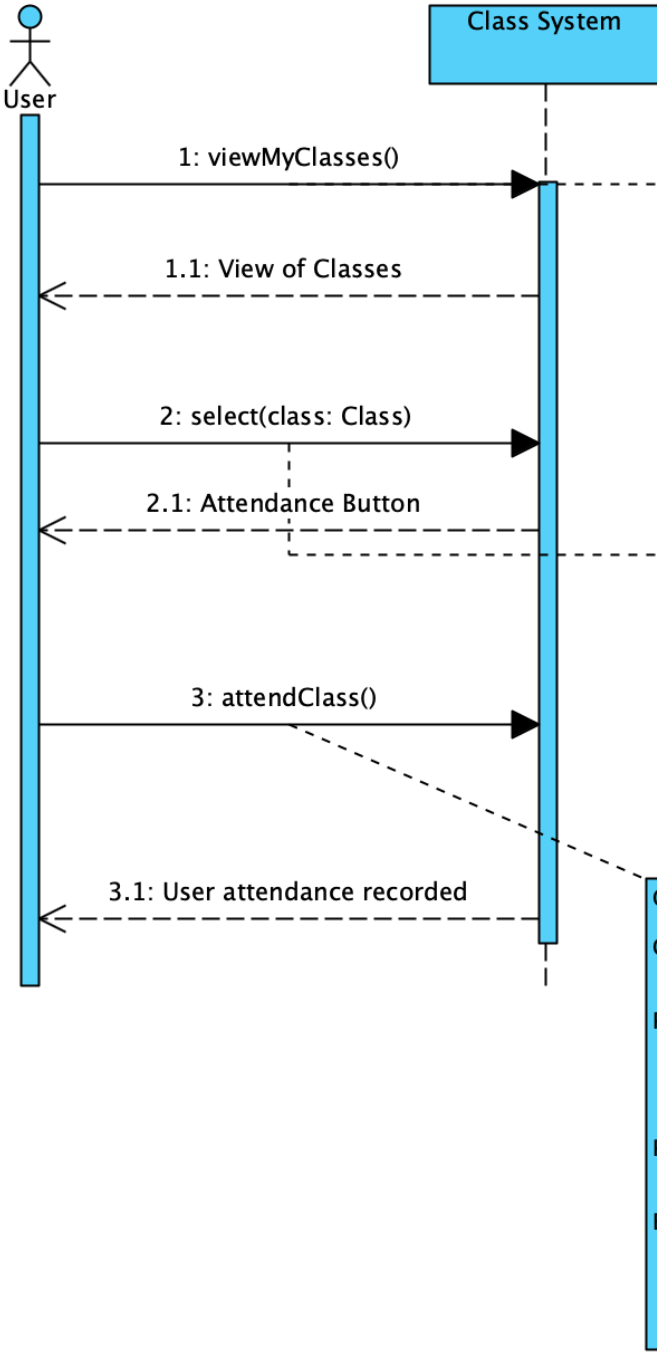
1. The user navigates to their profile
2. The user chooses a workout class from their registered workout classes
3. The system provides them an option to select "attend"
4. The user hits the "Attend" button
5. The system validates that the user is registered for the appropriate class
6. The system marks the user as present
7. The trainer can view that the user is present

**Alternate paths**

3a. If the user attempts to attend the same class twice

i. The system sends an error message saying the user is already present
ii. The user is not marked present twice

Attend Workout Class



**Operation: viewMyClasses()**

Cross References:

- Use Case - Attend Workout Class

Preconditions:

- *user.classes* is not empty

Postconditions:

- none

Exceptions:

- NoClasses if *user.classes* is empty

**Operation: select(class : Class)**

Cross References:

- Use Case - Attend Workout Class

Preconditions:

- *class* exists

Postconditions:

- none

Exceptions:

- none

**Operation: attendClass()**

Cross References:

- Use Case - Attend Workout Class

Preconditions:

- *class.started* is true

Postconditions:

- *class* is added to the list *user.attendedClasses* (attribution modified)
- *user* is associated with attending the class (association formed)

Exceptions:

- ClassNotStarted if *class.started* is false

**Author:** Emily Wokoek
**ID:** UC-T.CLASS-01
**Name:** UC Create Self-Paced Plan
**Scope:** App Self-Paced Plan Creation System
**Level:** User Goal
**Primary Actor:** Trainer

**Stakeholders and interests:**

- Trainer
    - Interested in creating a self-paced plan for users to do

**Precondition**

- Trainer is logged in

**Postcondition**

- New self-paced plan is created and is stored in the system
- System maintains previously created self-paced plans
- The trainer is able to access the newly added plan

**Minimal Guarantee**

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

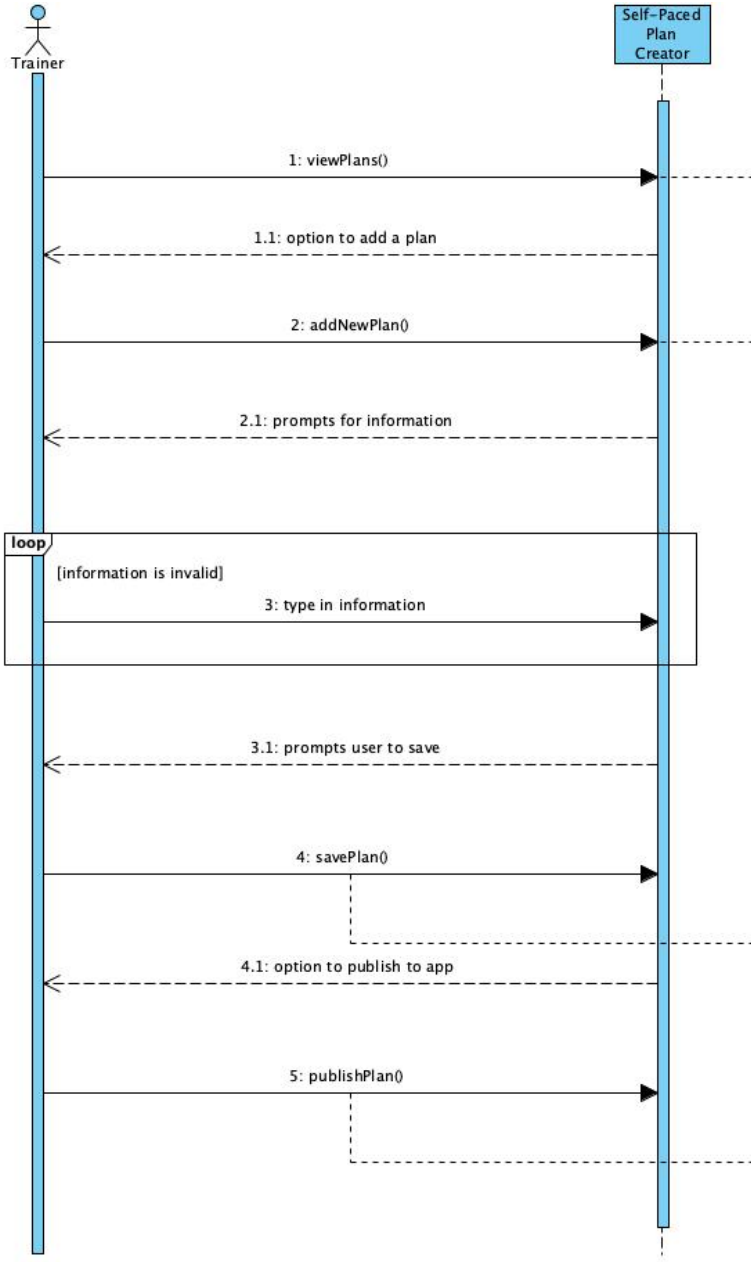
**Main success scenario**

1. The system presents action options about classes and self-paced plans
2. The trainer wishes to create a new self-paced plan and is able to do so
3. The system prompts trainer for information about class
4. The user inputs desired traits for class
5. The system should evaluate whether the specified inputted value is valid
6. If value is sound, system presents a create and save option to save the draft plan
7. The system saves the information and presents the option to publish the self-paced plan to the public
8. The trainer is able to preview what the plan looks like to users of the app

**Alternate paths**

5a. There is an invalid value inputted
    i. The system will prompt the trainer to re-enter the information
5b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
Self-Paced Plan Creator
1: viewPlans()
1.1: option to add a plan
2: addNewPlan()
2.1: prompts for information
loop
[information is invalid]
3: type in information
3.1: prompts user to save
4: savePlan()
4.1: option to publish to app
5: publishPlan()

**Operation: viewPlans()**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- App is opened to home page

Postconditions:

- none

Exceptions:

- none

**Operation: addNewPlan()**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- App is opened to plans page

Postconditions:

- Plan instance *plan* is created (object creation)

Exceptions:

- none

**Operation: savePlan(information: String)**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- Information in fields is valid

Postconditions:

- *plan* attributes were set to values passed in (attribute modification)

Exceptions:

- none

**Operation: publishPlan()**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- *plan* is created and instantiated

Postconditions:

- *plan* is added to *publicPlans* list (association formed)

Exceptions:

- none

**Author:** Carter Lewis
**ID:** UC-T.CLASS-02
**Name:** UC Create Class
**Scope:** App Self-Paced Plan Creation System
**Level:** User Goal
**Primary Actor:** Trainer

**Stakeholders and interests:**

- Trainer
    - Interested in creating an asynchronous class for users to do

**Precondition**

- Trainer is logged in

**Postcondition**

- New asynchronous class is created and is stored in the system
- System maintains previously created classes
- The trainer is able to access and start the newly added class

**Minimal Guarantee**

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

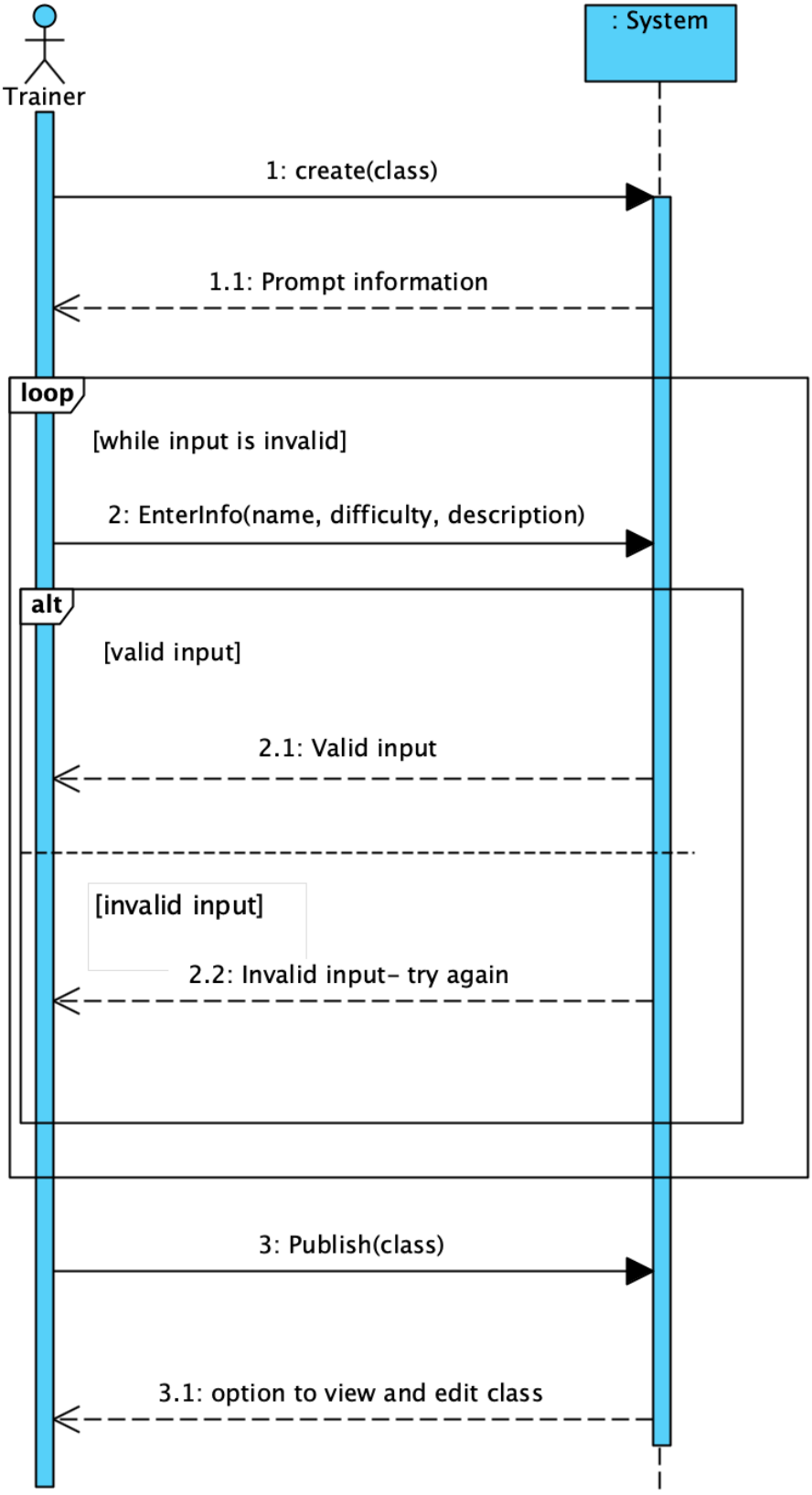
**Main success scenario**

1. The system presents action options about classes
2. The trainer wishes to create a new asynchronous class and is able to do so
3. The system prompts trainer for information (name, description, difficulty) about class
4. The user inputs desired traits for class
5. The system should evaluate whether the specified inputted value is valid
6. If value is sound, system presents a create and save option to save the draft class
7. The system saves the information and presents the option to publish the class to the public
8. The trainer is able to preview what the class looks like to users of the app

**Alternate paths**

5a. There is an invalid value inputted
    i. The system will prompt the trainer to re-enter the information
5b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
: System
1: create(class)
1.1: Prompt information
loop
[while input is invalid]
2: EnterInfo(name, difficulty, description)
alt
[valid input]
2.1: Valid input
[invalid input]
2.2: Invalid input- try again
3: Publish(class)
3.1: option to view and edit class

**Operation: create(class: class)**

Cross References

- Use Case: Trainer Class Creation

Preconditions:

- *trainer* is logged in

Postconditions

- Fitness course instance *course* was created (Object creation)

Exceptions

- None

**Operation: enterInfo(String name, int difficulty, Type: type, String description)**

Cross References

- Use Case: Trainer Class Creation

Preconditions:

- *trainer* is logged in
- *course* is created

Postconditions

- *class* attributes were updated with inputted information (Attribute modification)

Exceptions

- None

**Operation: submit(class: class)**

Cross References

- Use Case: Trainer Class Creation

Preconditions:

- *trainer* is logged in
- *course* has been instantiated

Postconditions

- Association between *trainer* and *class* (Association formed)

Exceptions

- None

**Author:** Emily Wokoek
**ID:** UC-T.CLASS-03
**Name:** UC Modify Plan
**Scope:** App Self-Paced Plan System
**Level:** User Goal
**Primary Actor:** Trainer

**Stakeholders and interests:**

- Trainer
    - Interested in modified a pre-existing self-paced plan

**Precondition**

- The trainer is logged in

**Postcondition**

- Self-paced plan is modified and is stored in the system
- System maintains previously created self-paced plans that are not edited
- The trainer is able to access the newly added or edited plan

**Minimal Guarantee**

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

**Main success scenario**

1. The system presents action options about classes and self-paced plans
2. The trainer selects the option modify a new self-paced plan
3. The trainer selects one of their self-paced plans
4. The system displays the plan information to the trainer
5. The trainer edits desired fields for class
6. The system should evaluate whether the specified inputted value is valid
7. If value is sound, system saves the information and publishes the self-paced plan to the public
8. The trainer is able to preview what the plan looks like to users of the app

**Alternate paths**

4a. Trainer enters the edit mode and then decides that they no longer wish to edit any of the information

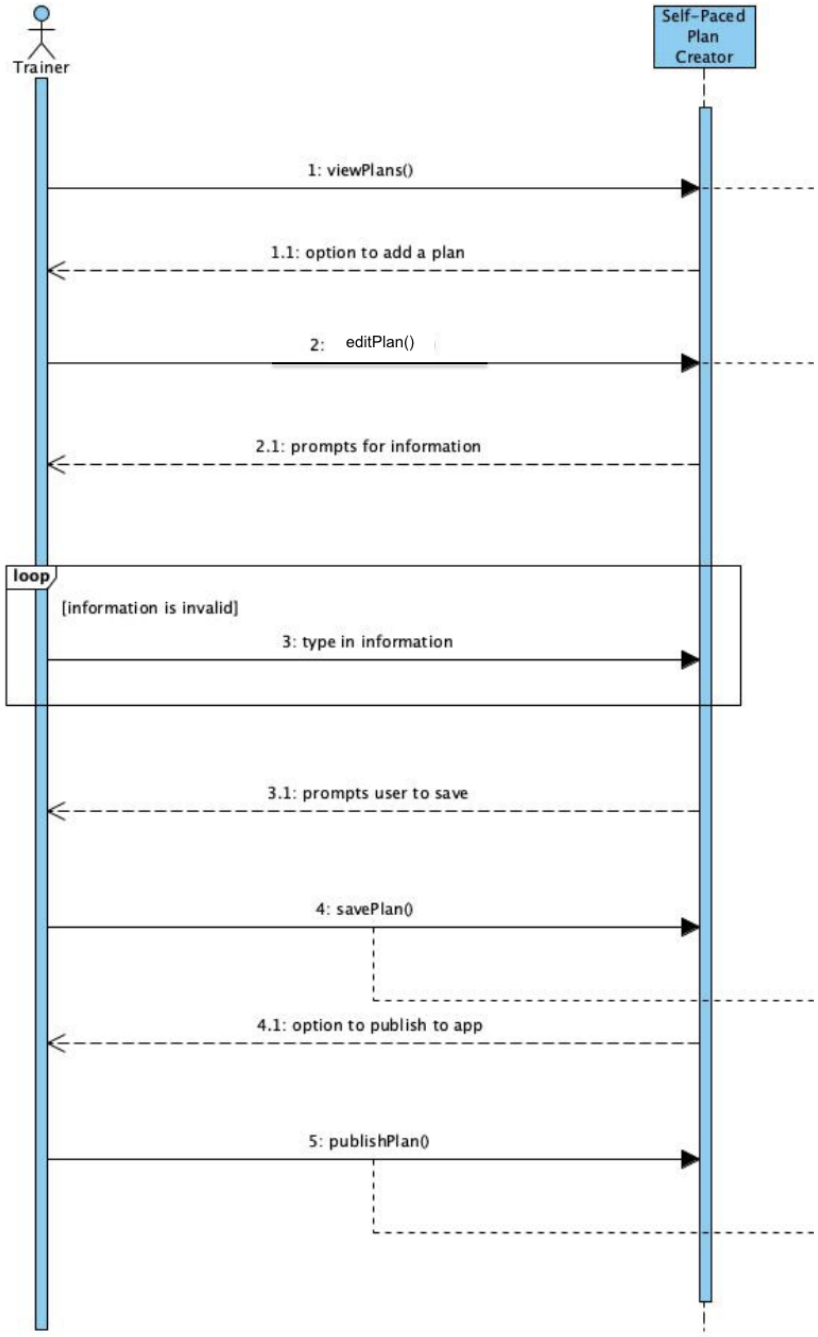
i. The system presents the option to cancel this operation

6a. There is an invalid value inputted

i. The system will prompt the trainer to re-enter the information

6b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
Self-Paced Plan Creator
1: viewPlans()
1.1: option to add a plan
2: editPlan()
2.1: prompts for information
loop
[information is invalid]
3: type in information
3.1: prompts user to save
4: savePlan()
4.1: option to publish to app
5: publishPlan()

**Operation: viewPlans()**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- App is opened to home page

Postconditions:

- none

Exceptions:

- none

**Operation: editPlan()**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- *plan* exists

Postconditions:

- none

Exceptions:

- none

**Operation: savePlan(information: String)**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- Information in fields is valid

Postconditions:

- *plan* attributes were set to values passed in (attribute modification)

Exceptions:

- none

**Operation: publishPlan()**

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- *plan* is created and instantiated

Postconditions:

- none

Exceptions:

- none

**Author:** Carter Lewis
**ID:** UC-T.CLASS-04
**Name:** UC Edit existing class
**Scope:** App Self-Paced Plan Creation System
**Level:** User Goal
**Primary Actor:** Trainer

**Stakeholders and interests:**

- Trainer
    - Interested in editing attributes of existing class

**Precondition**

- Trainer is logged in

**Postcondition**

- New attributes of class are saved and visible by users (Object creation)
- System maintains previously created asynchronous classes
- The trainer is able to access the newly edited class, and edit again if desired

**Minimal Guarantee**

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

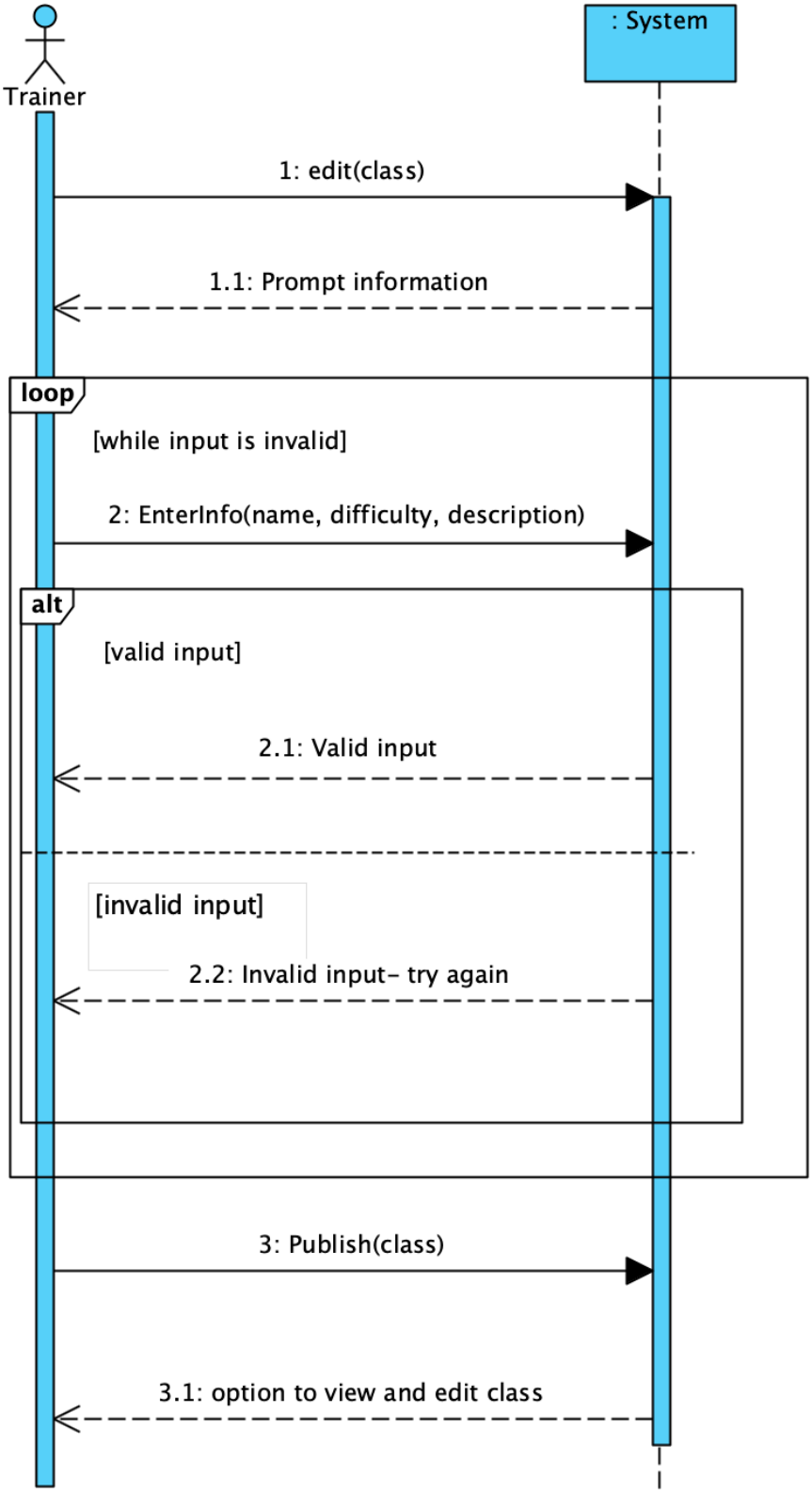
**Main success scenario**

1. The system presents action options about classes
2. The system prompts trainer for information about class
3. The user inputs desired traits for class
4. The system should evaluate whether the specified inputted value is valid
5. If values are sound, system presents a create and save option to save the draft plan
6. The system saves the information and updates all information for users to see
    a. Notifies members of a class when edits are made via email or push notification?
        i. Not in current design, but could be implemented in future iterations
7. The trainer is able to preview what the class looks like to users of the app

**Alternate paths**

5a. There is an invalid value inputted
    i. The system will prompt the trainer to re-enter the information
5b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
: System
1: edit(class)
1.1: Prompt information
loop
[while input is invalid]
2: EnterInfo(name, difficulty, description)
alt
[valid input]
2.1: Valid input
[invalid input]
2.2: Invalid input- try again
3: Publish(class)
3.1: option to view and edit class

**Operation: edit(class: class)**

Cross References

- Use Case: Trainer Edit Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class* is removed from the *publicClasses* list (association broken)

Exceptions

- None

**Operation: enterInfo(String name, int difficulty, Type: type, String description)**

Cross References

- Use Case: Trainer Edit Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class* attributes were updated with new information (Attribute modification)

Exceptions

- None

**Operation: publish(class: class)**

Cross References

- Use Case: Trainer Edit Class

Preconditions:

- Trainer is logged in
- Valid class information has been input

Postconditions

- *class* is added to the *publicClasses* list (Association Formed)

Exceptions

- None

**Author:** Kiera Shepperd
**Use Case:** UC-T.CLASS-05 Host Class
**ID:** UC-T.CLASS-05
**Scope:** Trainer Account
**Level:** Trainer Goal
**Primary Actor:** Trainer

**Stakeholders and Interests**

- Trainer
    - Wants to start a preexisting class

**Pre-Conditions**

- Trainer is logged into the app
- Class is created

**Post-Condition**

- Class statistics are recorded
- Class is archived
- Class roster is visible to the trainer

**Minimal Guarantee**

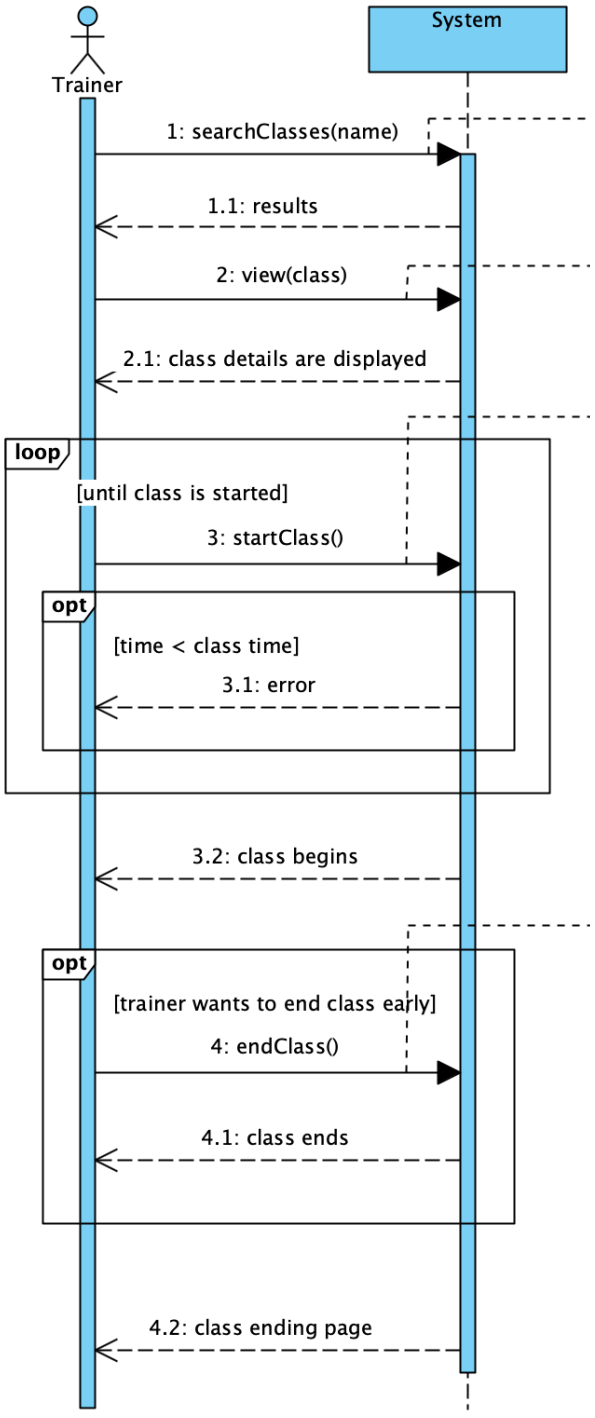
- The system will provide error messaging if something goes wrong
- Class will not be deleted

**Main Success Scenario**

1. Trainer searches for class
2. Trainer chooses desired class that they've created
3. At the time the class is scheduled to start, the trainer starts the class
4. The system keeps track of who attends the class
5. After the length of the class, the class ends

**Alternative Paths**

3a. The trainer tries to start the class before it is scheduled to start
    i. An error message appears and tells the trainer to wait until the posted class start time
3b. The class is not started 5 minutes after it's scheduled start time
    i. The class is rescheduled for 15 minutes later than the previously recorded time
4a. The trainer wishes to end the class early
    i. The trainer is selects the end class button


Trainer
System
1: searchClasses(name)
1.1: results
2: view(class)
2.1: class details are displayed
loop
[until class is started]
3: startClass()
opt
[time < class time]
3.1: error
3.2: class begins
opt
[trainer wants to end class early]
4: endClass()
4.1: class ends
4.2: class ending page

**Operation: searchClasses(String name)**

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in

Postconditions

- none

Exceptions

- None

**Operation: view(class: class)**

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- none

Exceptions

- None

**Operation: startClass(class: class)**

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class.started* is set to true (attribute modification)

Exceptions

- TooEarly if *class.canStart* is false

**Operation: endClass(class: class)**

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class.archived* is set to true (attribute modification)

- *class* is removed from *publicClasses* (association broken)

Exceptions

- TooEarly if *class.canStart* is false

**Author:** Noah Matthew
**Use Case:** UC-REPORT-01 View User Report
**ID:** UC-REPORT-01
**Scope:** User Account
**Level:** User Goal
**Primary Actor:** User

**Stakeholders and Interests**

- User
    - Wants to view a report of their recent activity

**Pre-Conditions**

- User is logged into the app

**Post-Condition**

- All information remains stored in the user's profile

**Minimal Guarantee**

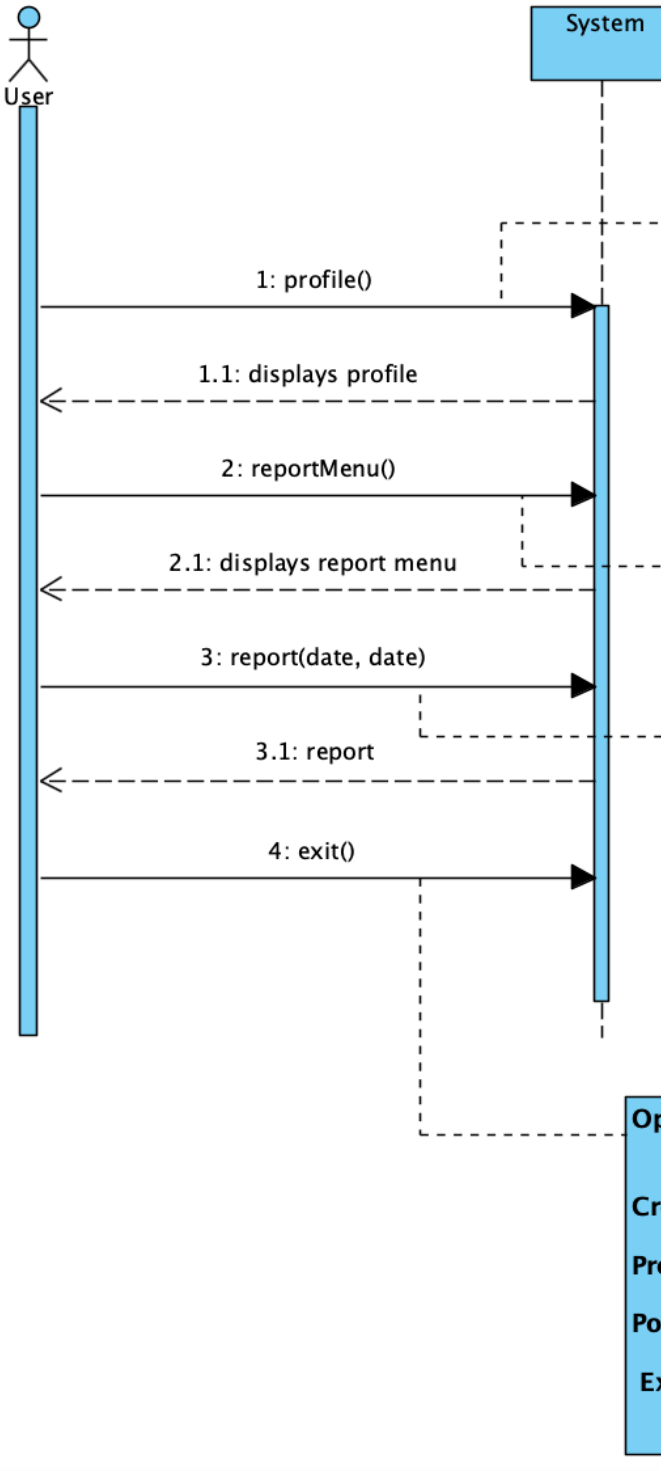
- The system will provide error messaging if something goes wrong
- User's data will not be deleted

**Main Success Scenario**

1. User navigates to their profile information
2. User clicks report
3. User selects a date range, ranging from when they created their account to the present
4. System averages the information logged during that time
5. System graphs the information logged during that time in comparison to the user's goals
6. System generates a report the information for the user to view
7. User exits the screen

**Alternative Paths**

3a. User no longer wants a report
    i. User exits the screen
5a. User's goals changed during the time frame selected for the report
    i. System graphs the current goal onto the entire time frame


User
System
1: profile()
1.1: displays profile
2: reportMenu()
2.1: displays report menu
3: report(date, date)
3.1: report
4: exit()

**Operation: profile()**

Cross References

- Use Case: View User Report
- Use Case: Log Information

Preconditions:

- *user* is logged in

Postconditions

- None

Exceptions

- None

**Operation: reportMenu()**

Cross References

- Use Case: View User Report

Preconditions:

- *user* is logged in

Postconditions

- None

Exceptions

- The User has no data

**Operation: report(Date date, Date date)**

Cross References

- Use Case: View User Report

Preconditions:

- *user* is logged in
- Report page is displayed

Postconditions

- Report instance *report* is created (object creation)
- *report* is added to *user.reports* list (association formed)

Exceptions

- NoData is thrown if there are no DailyMetrics between the dates provided

**Operation: exit()**

Cross References

- Use Case: View User Report

Preconditions:

- *user* is logged in
- Report page is displayed

Postconditions

- None

Exceptions

- None

**Author:** Kiera Shepperd
**Use Case:** UC-REPORT-02 View Class Report
**ID:** UC-REPORT-02
**Scope:** Trainer Account
**Level:** Trainer Goal
**Primary Actor:** Trainer

**Stakeholders and Interests**

- Trainer
    - Wants to view a report of those who attended the class

**Pre-Conditions**

- Trainer is logged into the app
- Class has started and ended
- Class statistics are recorded

**Post-Condition**

- All information remains stored in the archive with the class

**Minimal Guarantee**

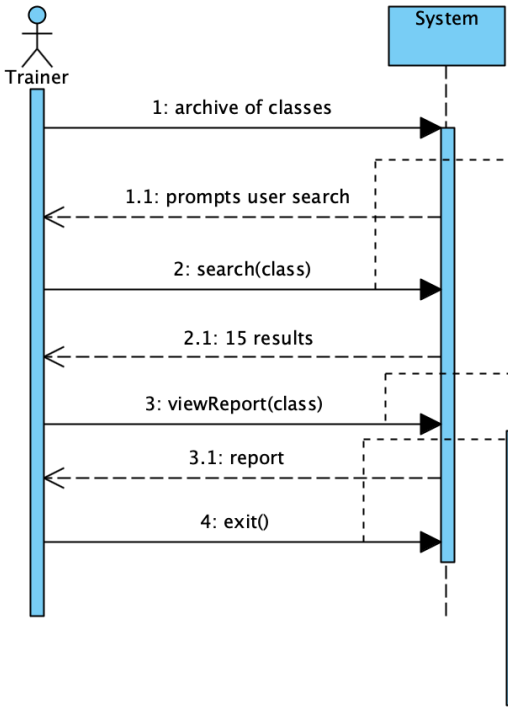
- The system will provide error messaging if something goes wrong
- Class will not be deleted

**Main Success Scenario**

1. Trainer navigates to report menu
2. Trainer searches for archived class
3. Trainer chooses desired class that they've hosted
4. Trainer clicks report
5. System generates a report of the attendees of the class and the average calories burned across the attendees
6. Trainer exists the screen

**Alternative Paths**

4a. Trainer no longer wants a report
    i. Trainer exits the screen
3a. Trainer wants to see reports that aggregate multiple classes
    i. Trainer selects multiple classes that they've hosted


Trainer
System
1: archive of classes
1.1: prompts user search
2: search(class)
2.1: 15 results
3: viewReport(class)
3.1: report
4: exit()

**Operation: search(class: class)**

Cross References

- Use Case: View Class Report

Preconditions:

- *trainer* is logged in
- *class* exist

Postconditions

- None

Exceptions

- NoClasses thrown if *trainer* has no classes

**Operation: viewReport(class: class)**

Cross References

- Use Case: View Class Report

Preconditions:

- *trainer* is logged in
- Report menu is displayed
- *class* exist

Postconditions

- Report instance created *classReport* (object creation)
- *classReport* added to *trainer.classReports* list (association formed)

Exceptions

- None

**Operation: exit()**

Cross References

- Use Case: View Class Report

Preconditions:

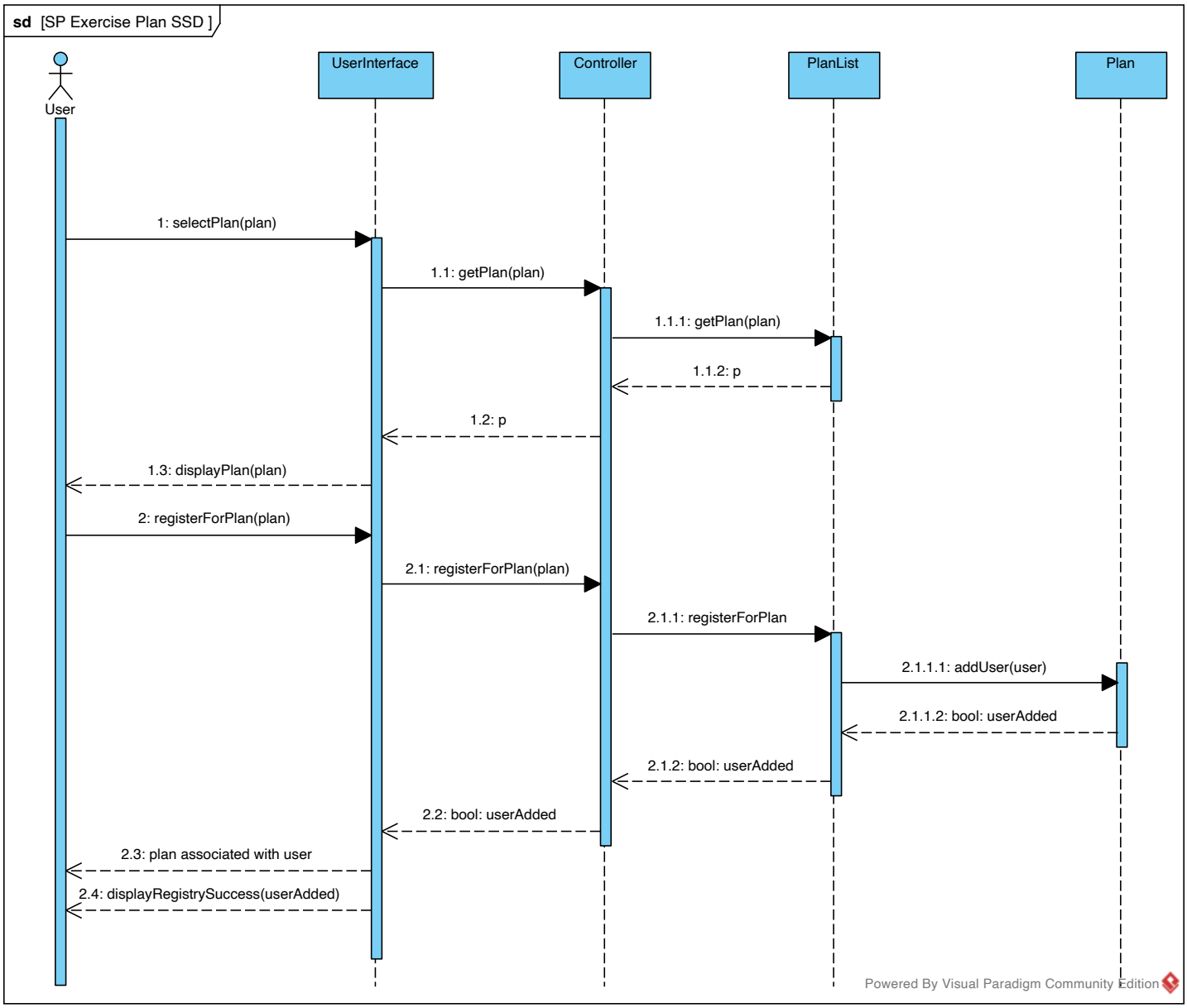
- *trainer* is logged in

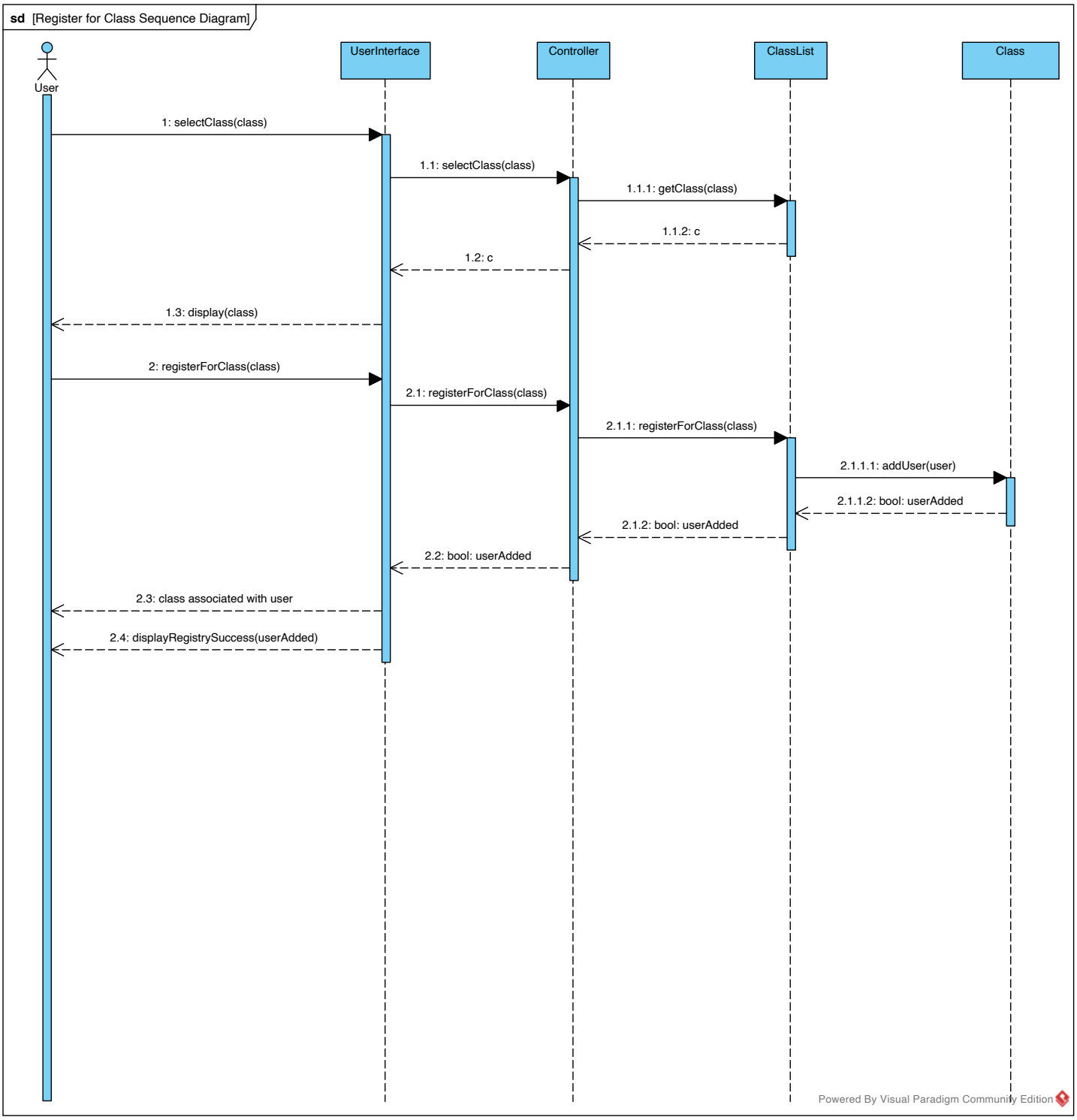
Postconditions

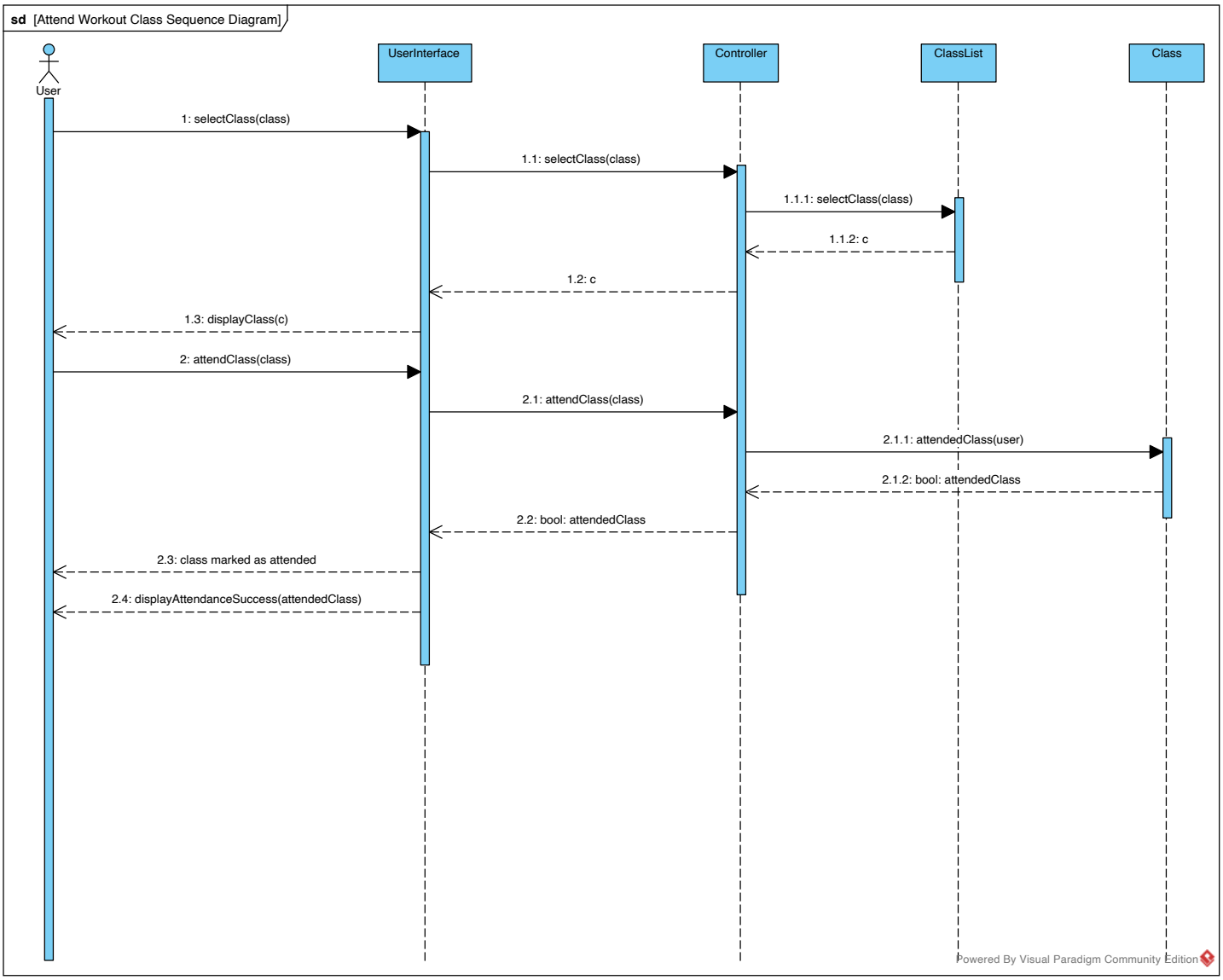
- None

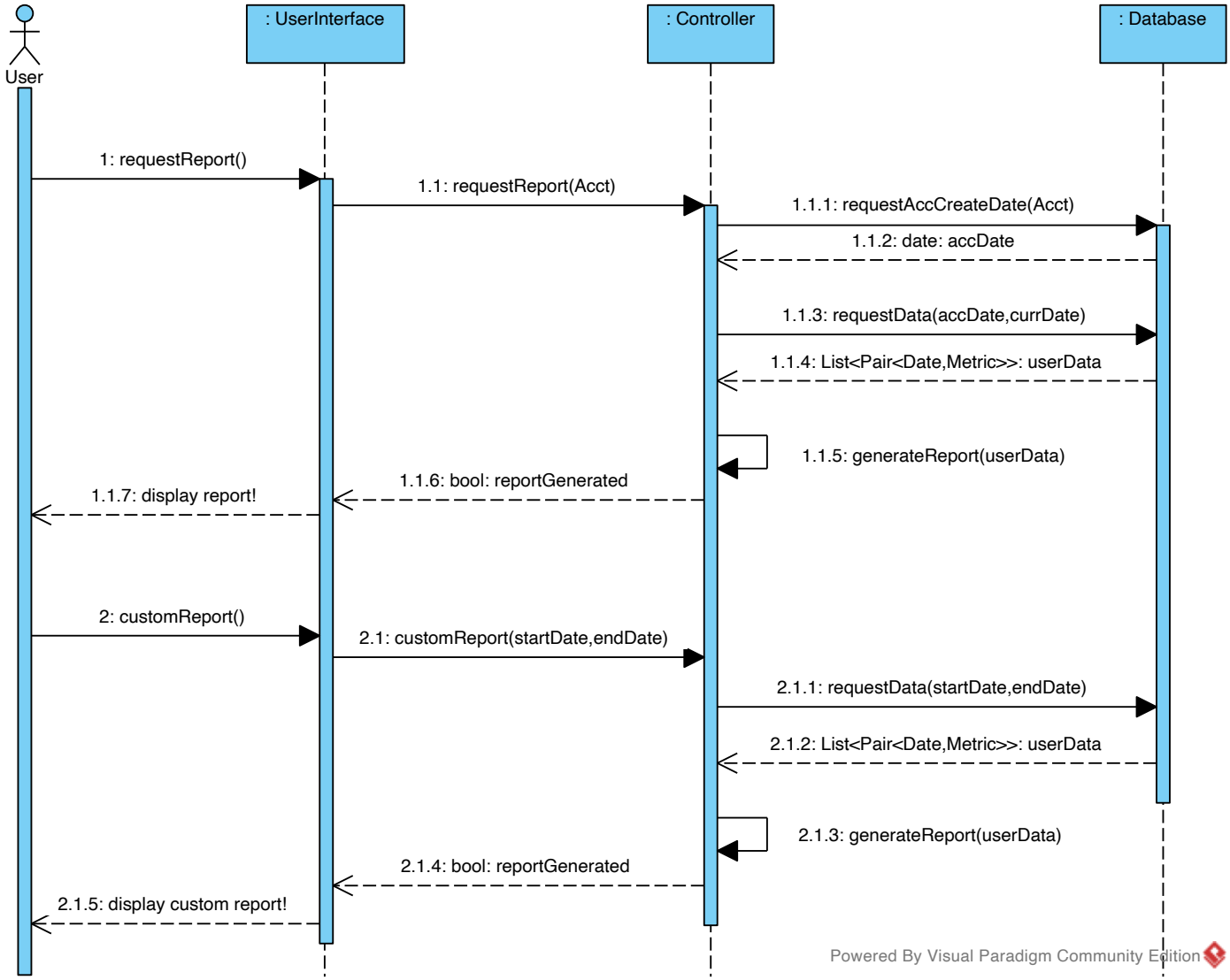
Exceptions

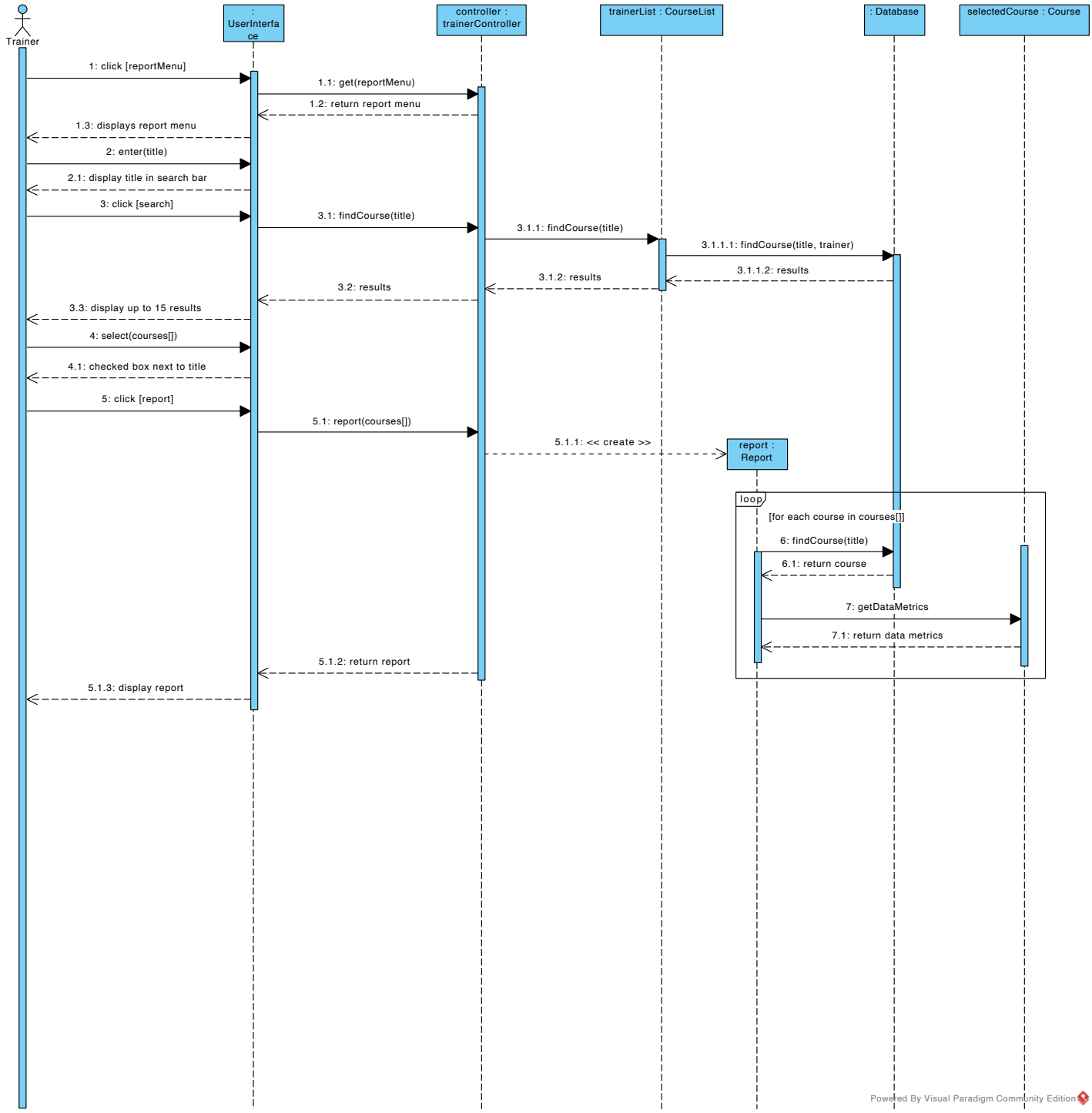
- None

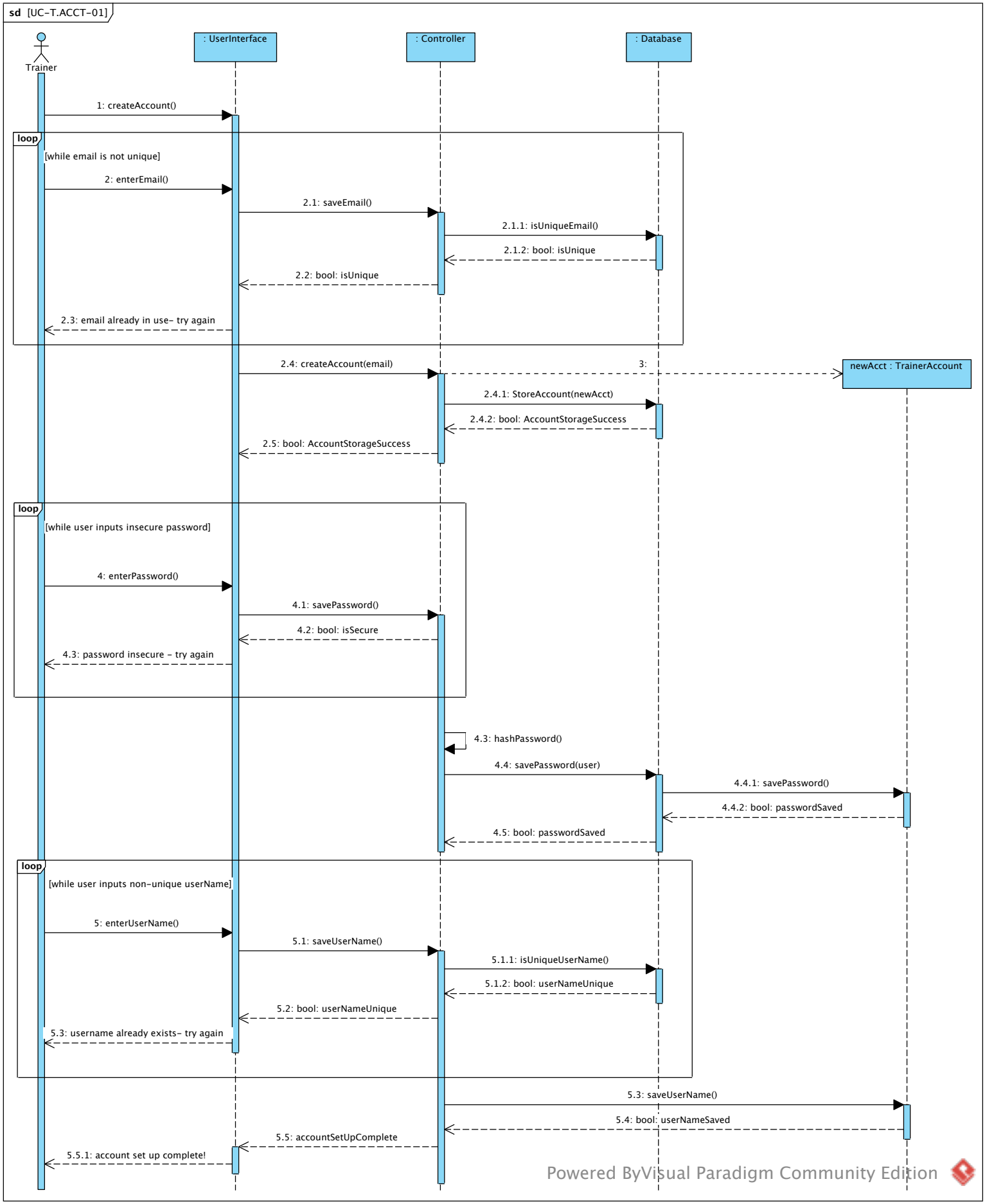

sd [SP Exercise Plan SSD ]
User
UserInterface
Controller
PlanList
Plan
1: selectPlan(plan)
1.1: getPlan(plan)
1.1.1: getPlan(plan)
1.1.2: p
1.2: p
1.3: displayPlan(plan)
2: registerForPlan(plan)
2.1: registerForPlan(plan)
2.1.1: registerForPlan
2.1.1.1: addUser(user)
2.1.1.2: bool: userAdded
2.1.2: bool: userAdded
2.2: bool: userAdded
2.3: plan associated with user
2.4: displayRegistrySuccess(userAdded)
Powered By Visual Paradigm Community Edition

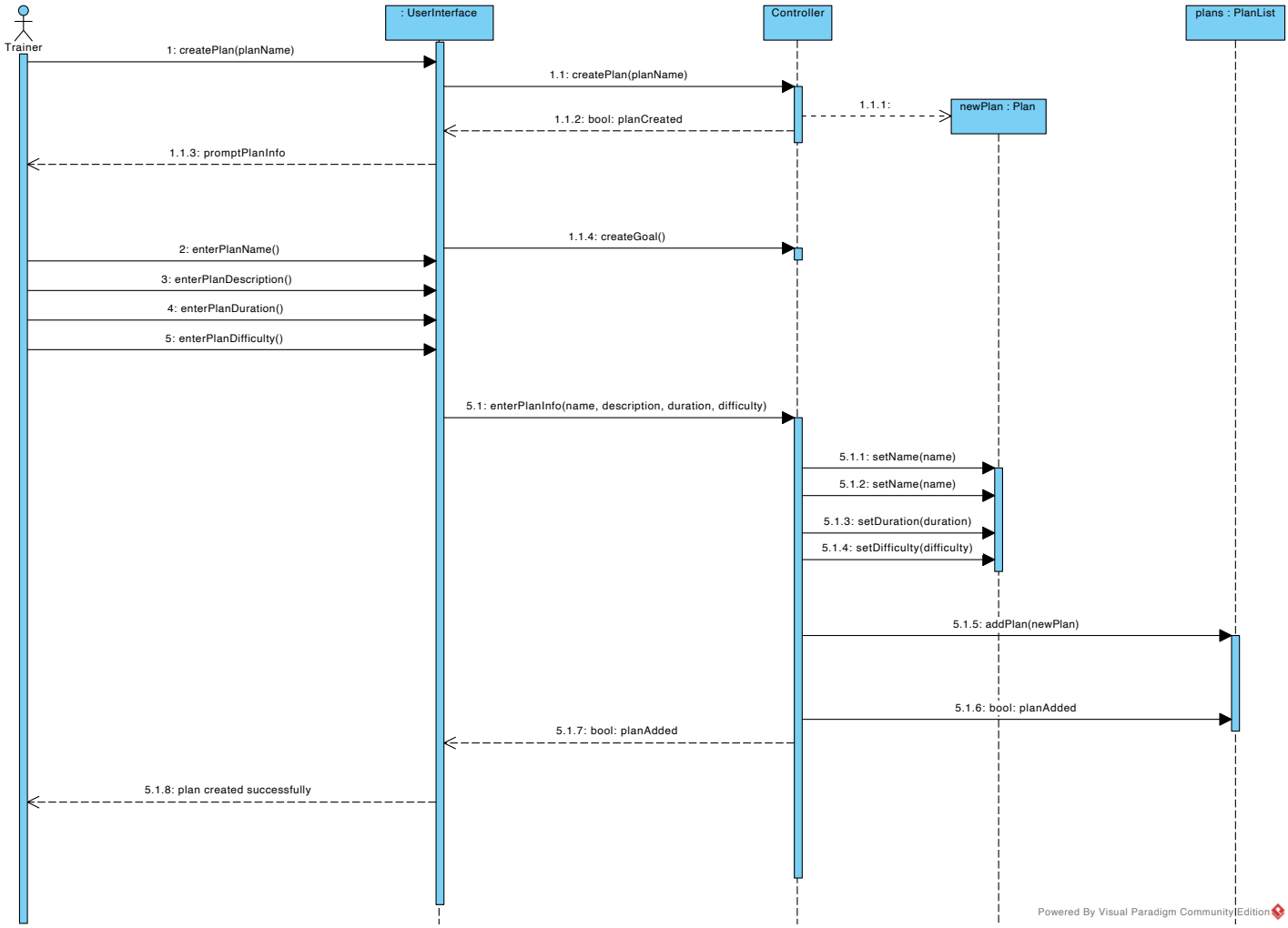

sd [Register for Class Sequence Diagram]
User
UserInterface
Controller
ClassList
Class
1: selectClass(class)
1.1: selectClass(class)
1.1.1: getClass(class)
1.1.2: c
1.2: c
1.3: display(class)
2: registerForClass(class)
2.1: registerForClass(class)
2.1.1: registerForClass(class)
2.1.1.1: addUser(user)
2.1.1.2: bool: userAdded
2.1.2: bool: userAdded
2.2: bool: userAdded
2.3: class associated with user
2.4: displayRegistrySuccess(userAdded)
Powered By Visual Paradigm Community Edition

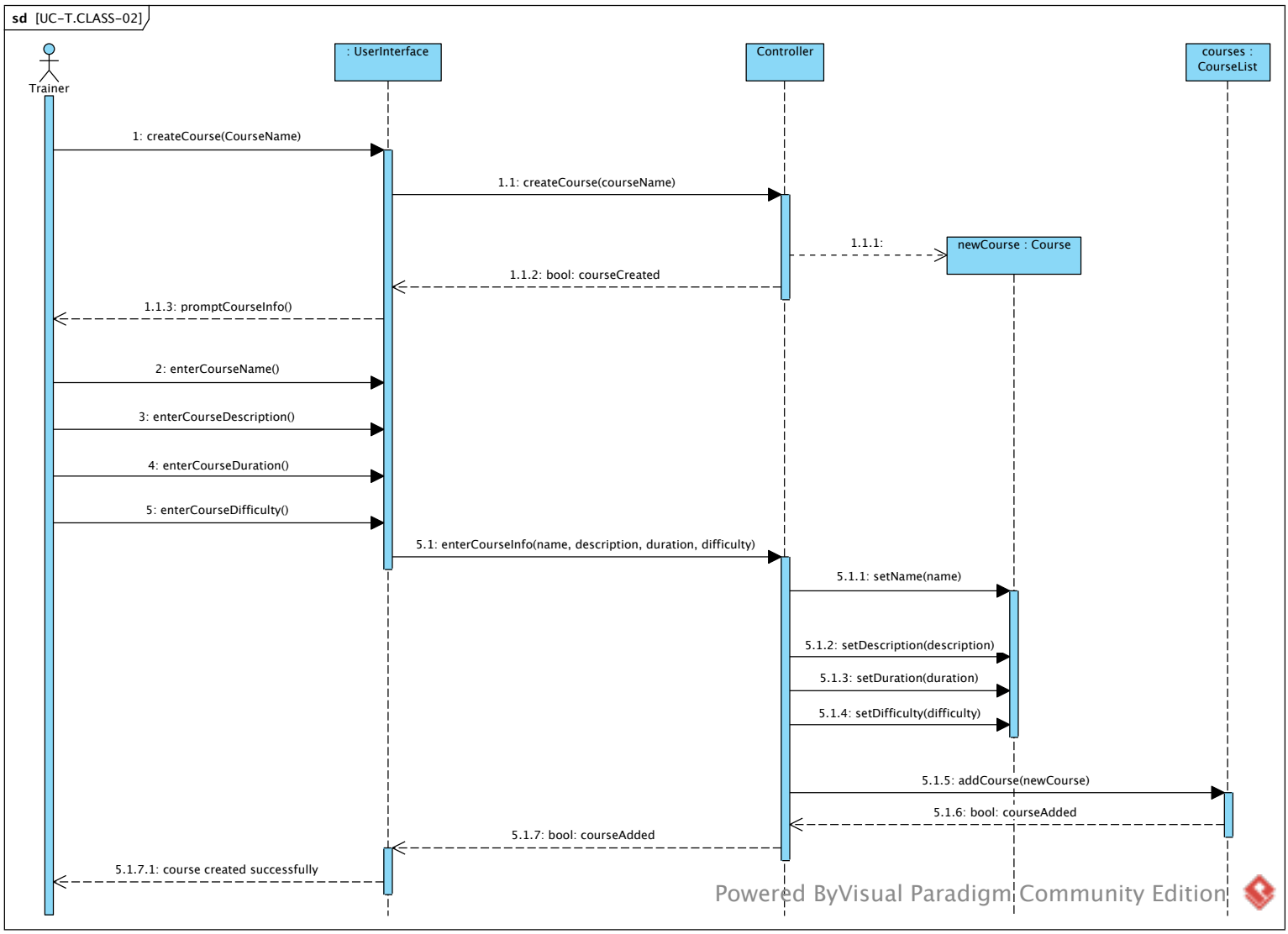

sd [Attend Workout Class Sequence Diagram]
User
UserInterface
Controller
ClassList
Class
1: selectClass(class)
1.1: selectClass(class)
1.1.1: selectClass(class)
1.1.2: c
1.2: c
1.3: displayClass(c)
2: attendClass(class)
2.1: attendClass(class)
2.1.1: attendedClass(user)
2.1.2: bool: attendedClass
2.2: bool: attendedClass
2.3: class marked as attended
2.4: displayAttendanceSuccess(attendedClass)
Powered By Visual Paradigm Community Edition

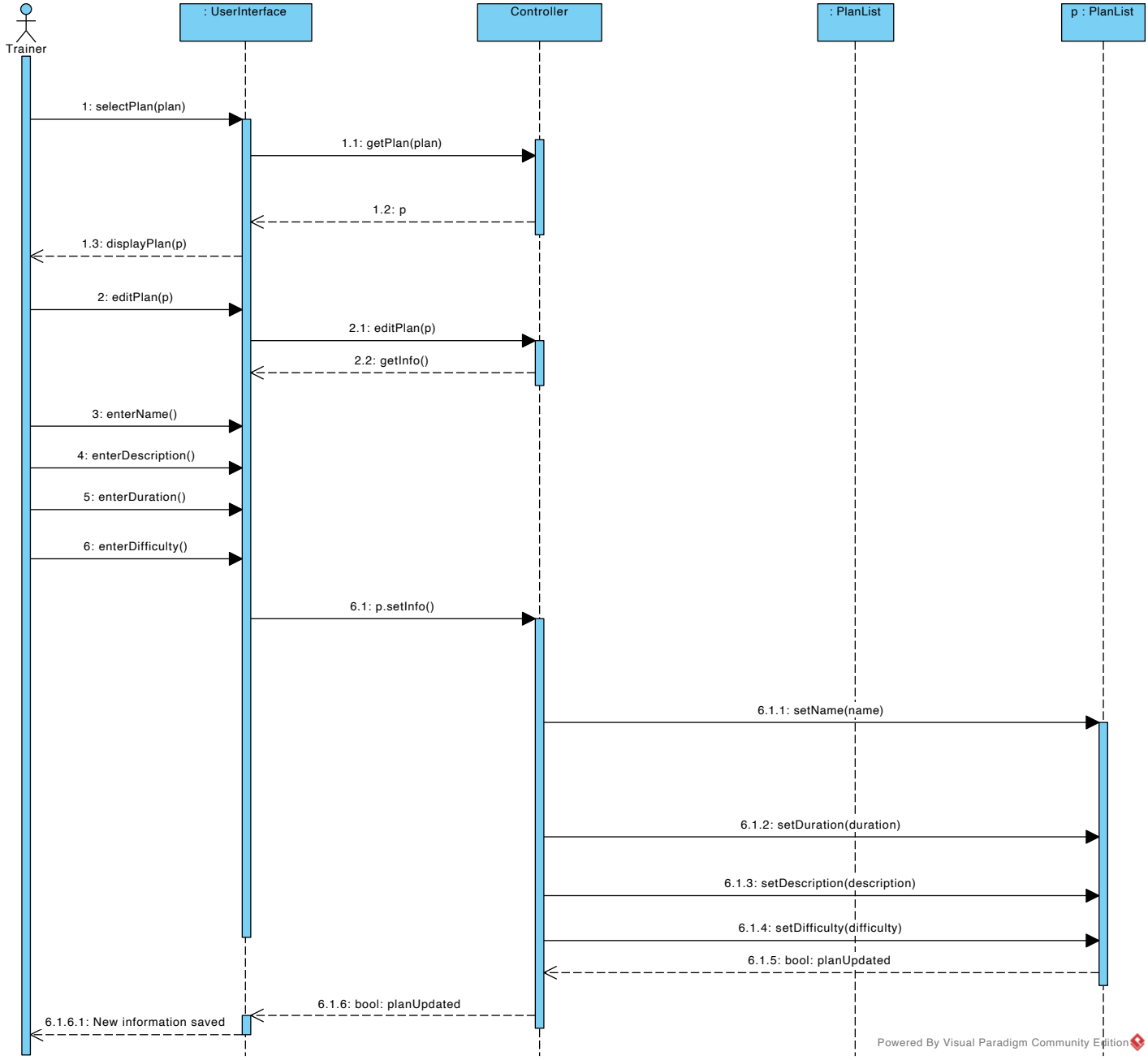

User
: UserInterface
: Controller
: Database
1: requestReport()
1.1: requestReport(Acct)
1.1.1: requestAccCreateDate(Acct)
1.1.2: date: accDate
1.1.3: requestData(accDate,currDate)
1.1.4: List<Pair<Date,Metric>>: userData
1.1.5: generateReport(userData)
1.1.6: bool: reportGenerated
1.1.7: display report!
2: customReport()
2.1: customReport(startDate,endDate)
2.1.1: requestData(startDate,endDate)
2.1.2: List<Pair<Date,Metric>>: userData
2.1.3: generateReport(userData)
2.1.4: bool: reportGenerated
2.1.5: display custom report!
Powered By Visual Paradigm Community Edition

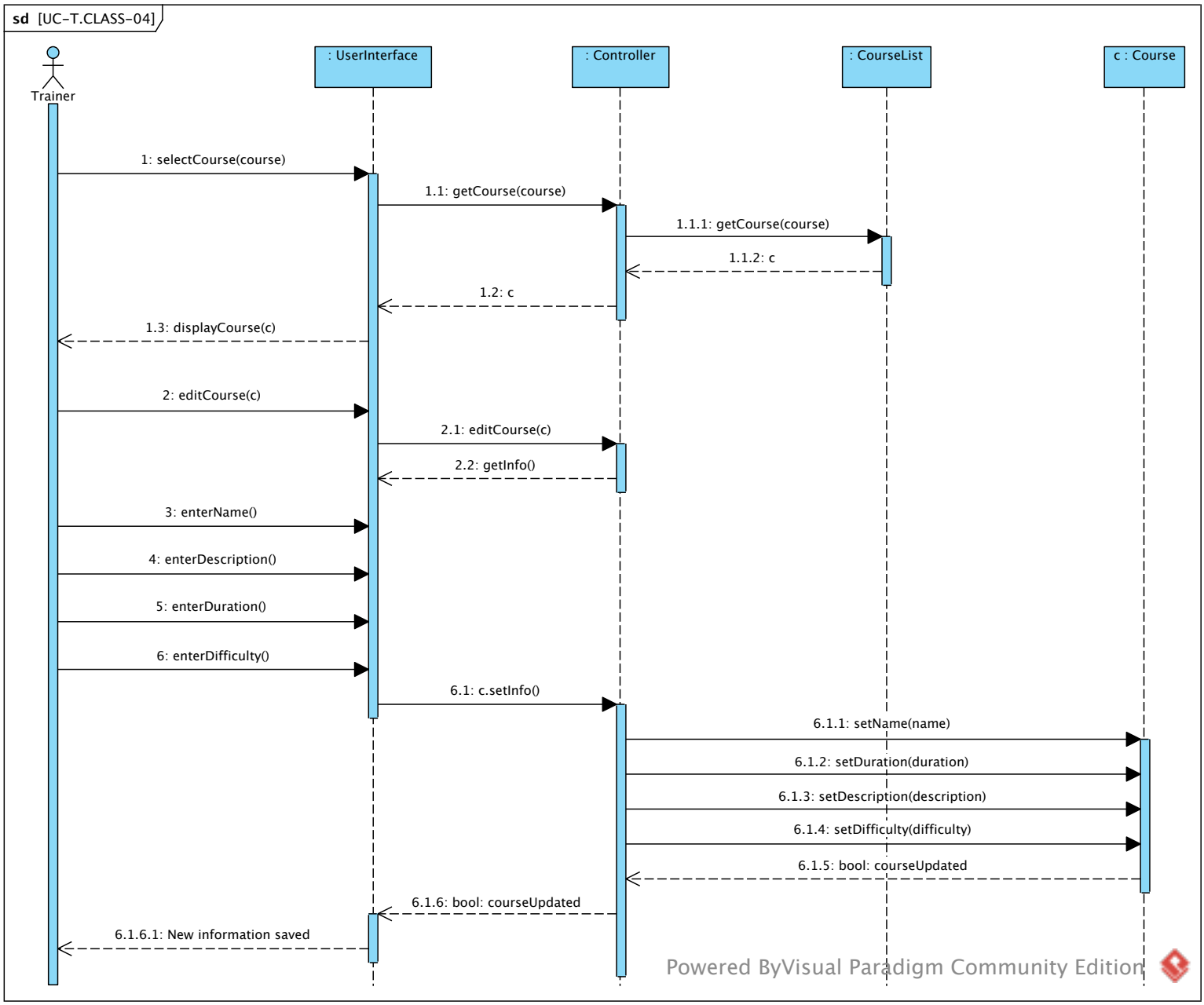

Trainer
: UserInterface
controller : trainerController
trainerList : CourseList
: Database
selectedCourse : Course
1: click [reportMenu]
1.1: get(reportMenu)
1.2: return report menu
1.3: displays report menu
2: enter(title)
2.1: display title in search bar
3: click [search]
3.1: findCourse(title)
3.1.1: findCourse(title)
3.1.1.1: findCourse(title, trainer)
3.1.1.2: results
3.1.2: results
3.2: results
3.3: display up to 15 results
4: select(courses[])
4.1: checked box next to title
5: click [report]
5.1: report(courses[])
5.1.1: << create >>
report : Report
loop
[for each course in courses[]]
6: findCourse(title)
6.1: return course
7: getDataMetrics
7.1: return data metrics
5.1.2: return report
5.1.3: display report
Powered By Visual Paradigm Community Edition

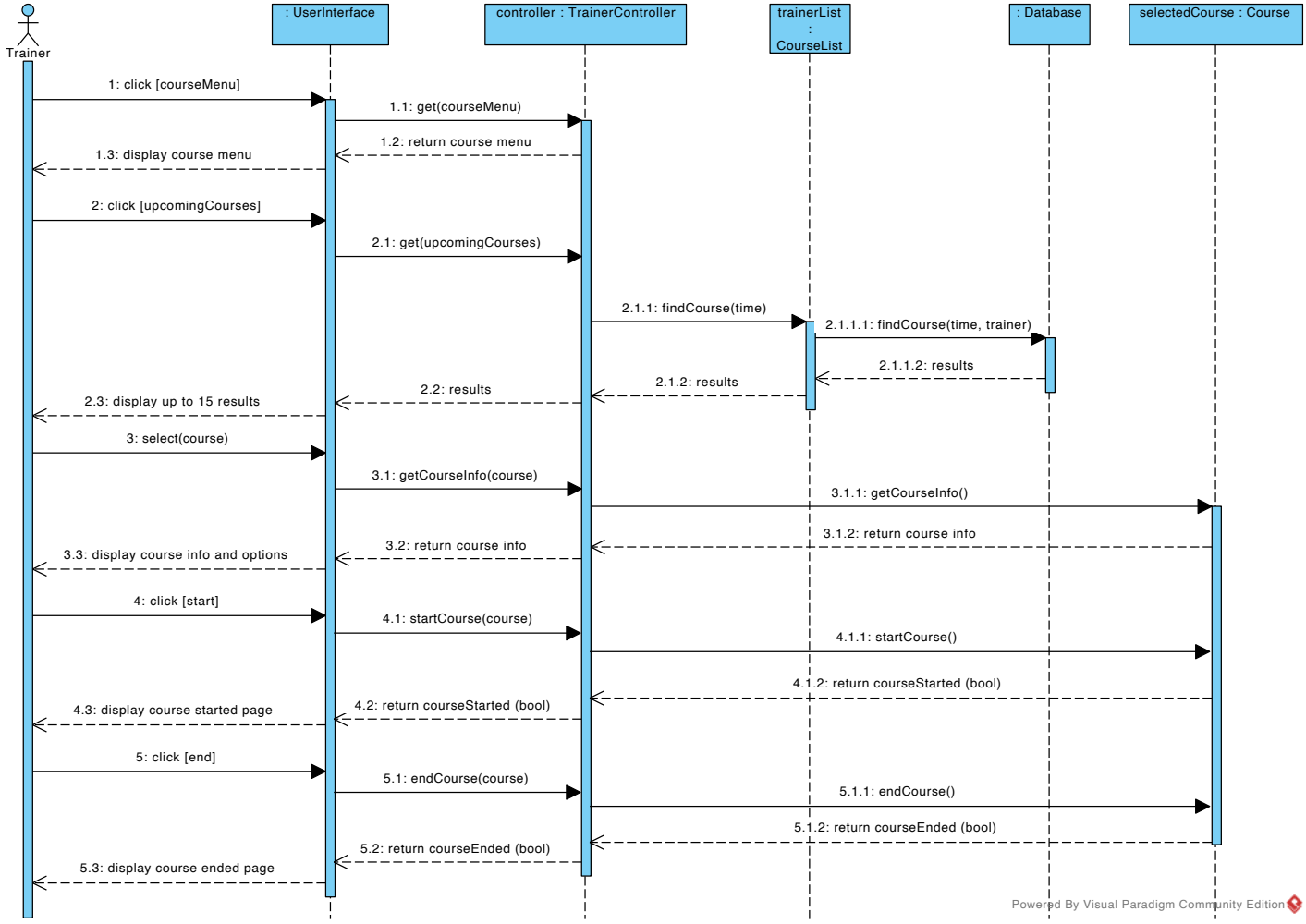

sd [UC-T.ACCT-01]
Trainer
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email already in use- try again
2.4: createAccount(email)
3:
newAcct : TrainerAccount
2.4.1: StoreAccount(newAcct)
2.4.2: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while user inputs insecure password]
4: enterPassword()
4.1: savePassword()
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: savePassword(user)
4.4.1: savePassword()
4.4.2: bool: passwordSaved
4.5: bool: passwordSaved
loop
[while user inputs non-unique userName]
5: enterUserName()
5.1: saveUserName()
5.1.1: isUniqueUserName()
5.1.2: bool: userNameUnique
5.2: bool: userNameUnique
5.3: username already exists- try again
5.3: saveUserName()
5.4: bool: userNameSaved
5.5: accountSetUpComplete
5.5.1: account set up complete!
Powered ByVisual Paradigm Community Edition

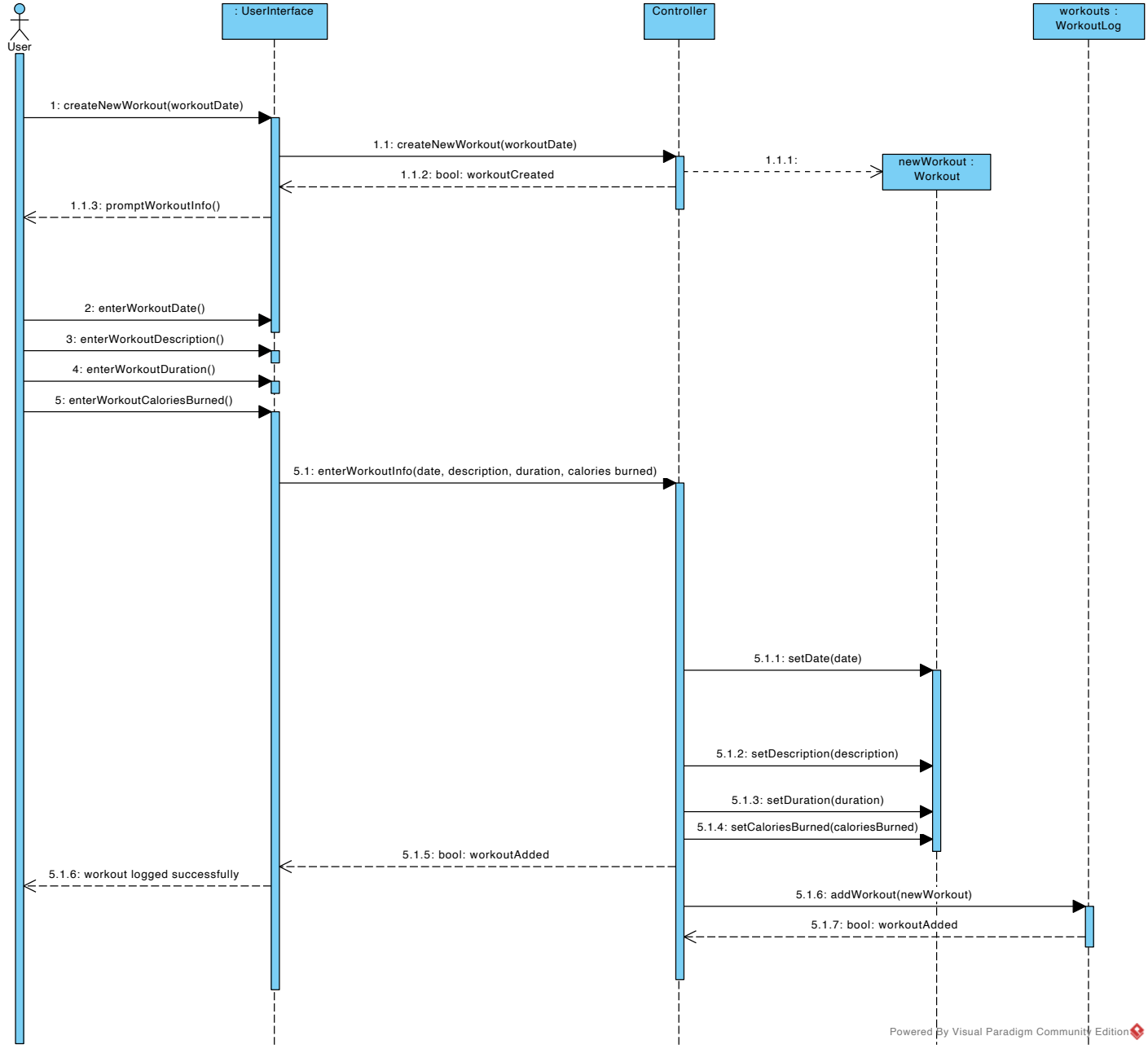

Trainer
: UserInterface
Controller
plans : PlanList
newPlan : Plan
1: createPlan(planName)
1.1: createPlan(planName)
1.1.1:
1.1.2: bool: planCreated
1.1.3: promptPlanInfo
1.1.4: createGoal()
2: enterPlanName()
3: enterPlanDescription()
4: enterPlanDuration()
5: enterPlanDifficulty()
5.1: enterPlanInfo(name, description, duration, difficulty)
5.1.1: setName(name)
5.1.2: setName(name)
5.1.3: setDuration(duration)
5.1.4: setDifficulty(difficulty)
5.1.5: addPlan(newPlan)
5.1.6: bool: planAdded
5.1.7: bool: planAdded
5.1.8: plan created successfully
Powered By Visual Paradigm Community Edition

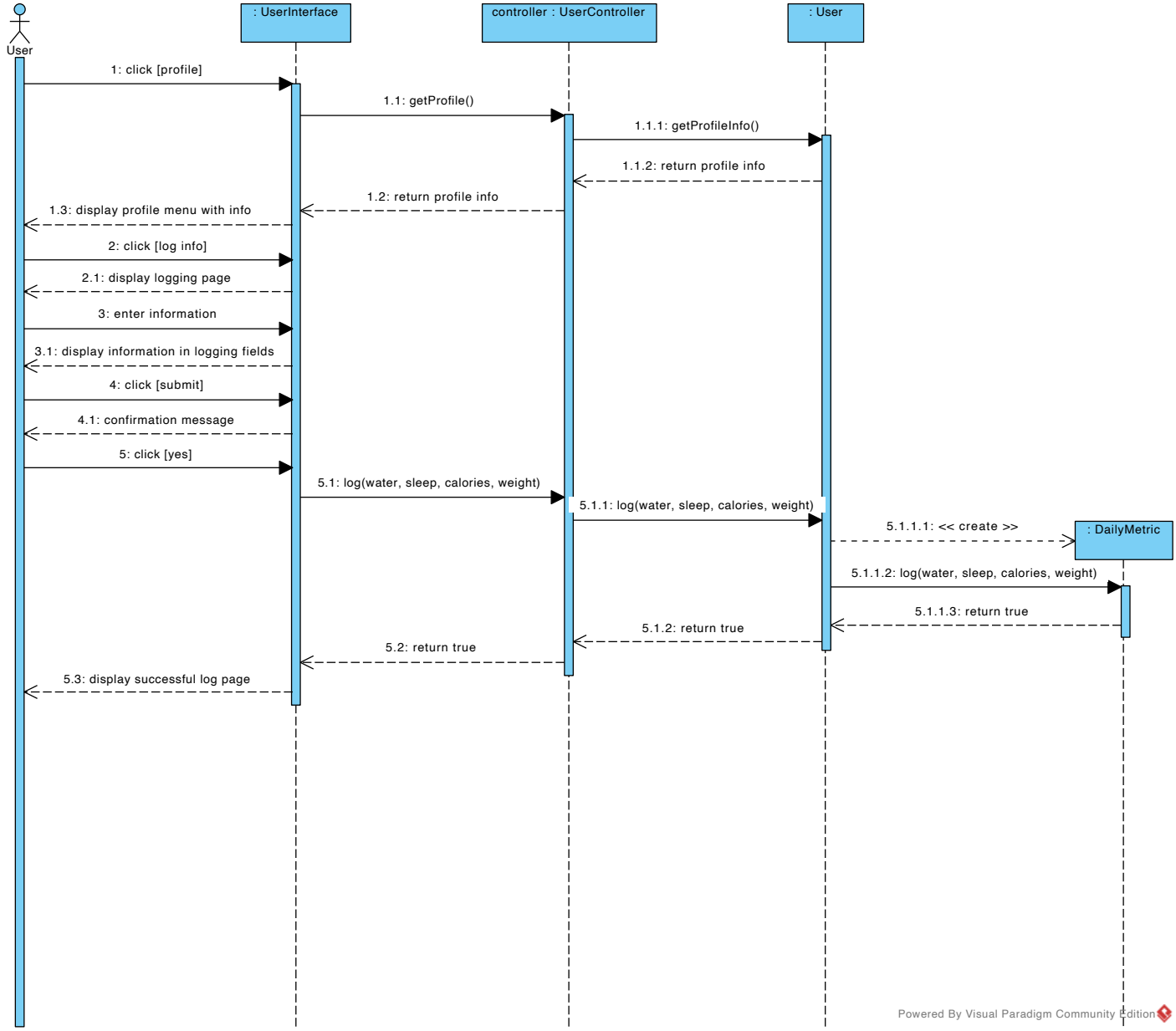

sd [UC-T.CLASS-02]
Trainer
: UserInterface
Controller
courses : CourseList
1: createCourse(CourseName)
1.1: createCourse(courseName)
1.1.1:
newCourse : Course
1.1.2: bool: courseCreated
1.1.3: promptCourseInfo()
2: enterCourseName()
3: enterCourseDescription()
4: enterCourseDuration()
5: enterCourseDifficulty()
5.1: enterCourseInfo(name, description, duration, difficulty)
5.1.1: setName(name)
5.1.2: setDescription(description)
5.1.3: setDuration(duration)
5.1.4: setDifficulty(difficulty)
5.1.5: addCourse(newCourse)
5.1.6: bool: courseAdded
5.1.7: bool: courseAdded
5.1.7.1: course created successfully
Powered ByVisual Paradigm Community Edition

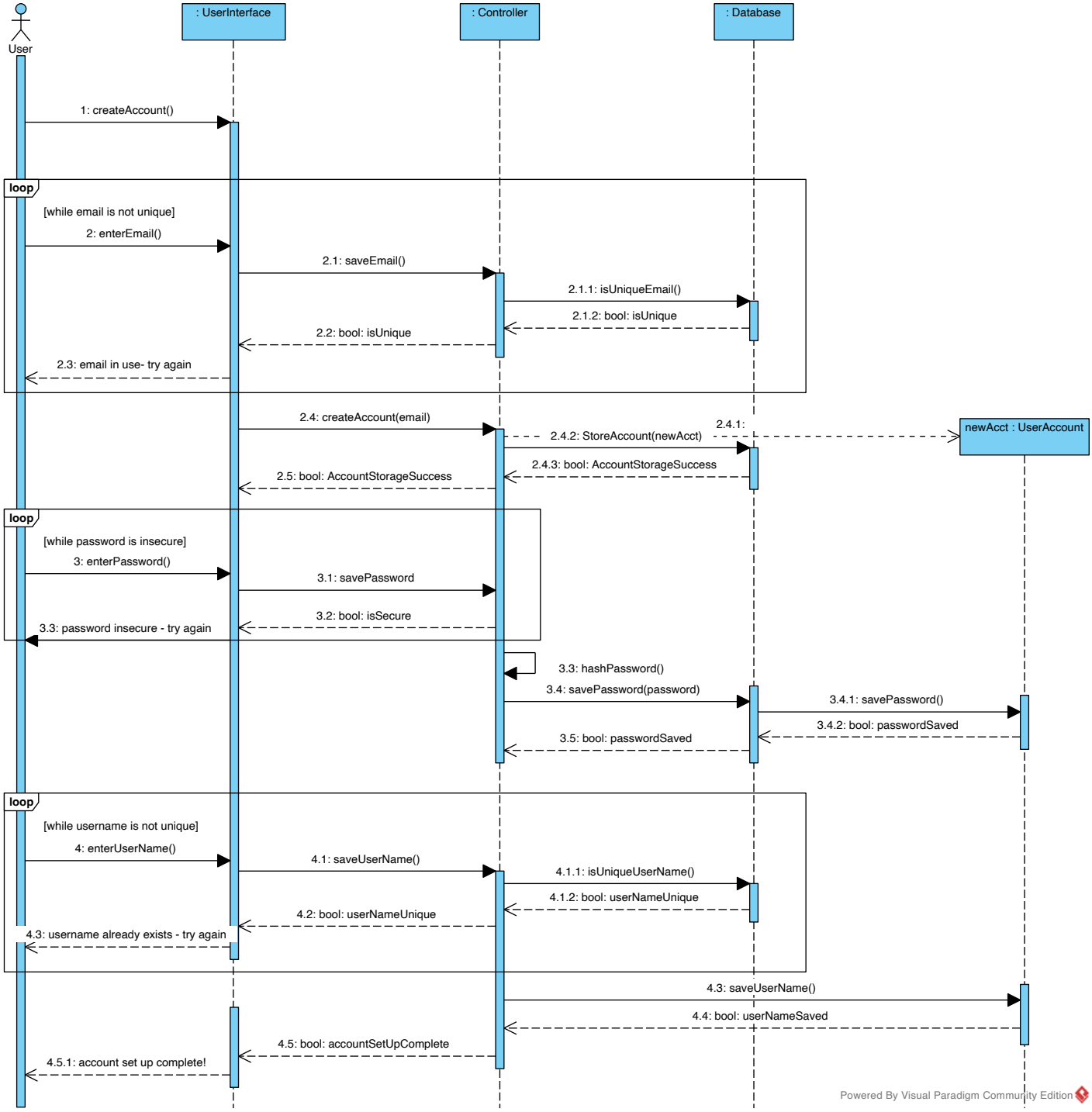

Trainer
: UserInterface
Controller
: PlanList
p : PlanList
1: selectPlan(plan)
1.1: getPlan(plan)
1.2: p
1.3: displayPlan(p)
2: editPlan(p)
2.1: editPlan(p)
2.2: getInfo()
3: enterName()
4: enterDescription()
5: enterDuration()
6: enterDifficulty()
6.1: p.setInfo()
6.1.1: setName(name)
6.1.2: setDuration(duration)
6.1.3: setDescription(description)
6.1.4: setDifficulty(difficulty)
6.1.5: bool: planUpdated
6.1.6: bool: planUpdated
6.1.6.1: New information saved
Powered By Visual Paradigm Community Edition

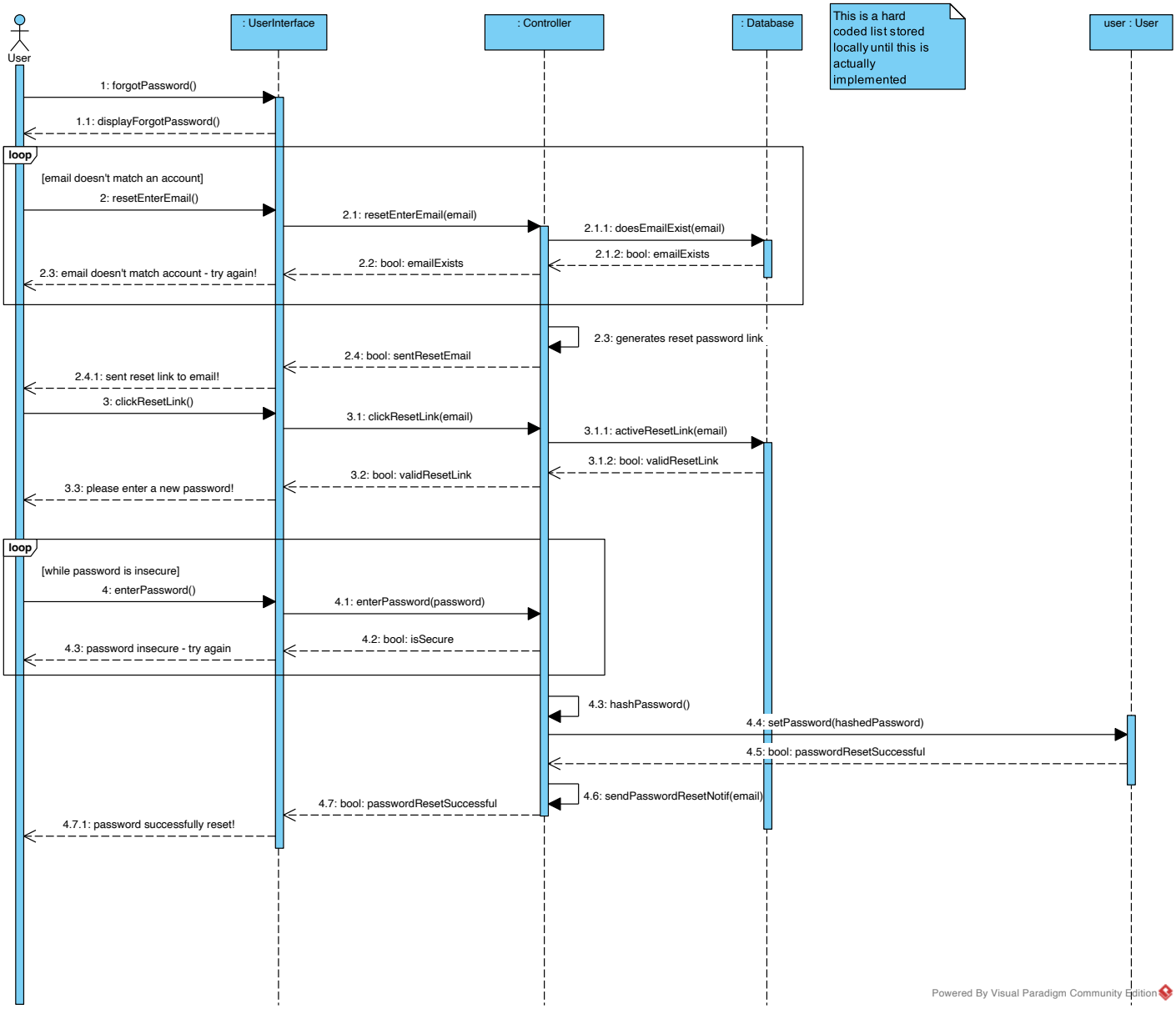

sd [UC-T.CLASS-04]
Trainer
: UserInterface
: Controller
: CourseList
c : Course
1: selectCourse(course)
1.1: getCourse(course)
1.1.1: getCourse(course)
1.1.2: c
1.2: c
1.3: displayCourse(c)
2: editCourse(c)
2.1: editCourse(c)
2.2: getInfo()
3: enterName()
4: enterDescription()
5: enterDuration()
6: enterDifficulty()
6.1: c.setInfo()
6.1.1: setName(name)
6.1.2: setDuration(duration)
6.1.3: setDescription(description)
6.1.4: setDifficulty(difficulty)
6.1.5: bool: courseUpdated
6.1.6: bool: courseUpdated
6.1.6.1: New information saved
Powered ByVisual Paradigm Community Edition

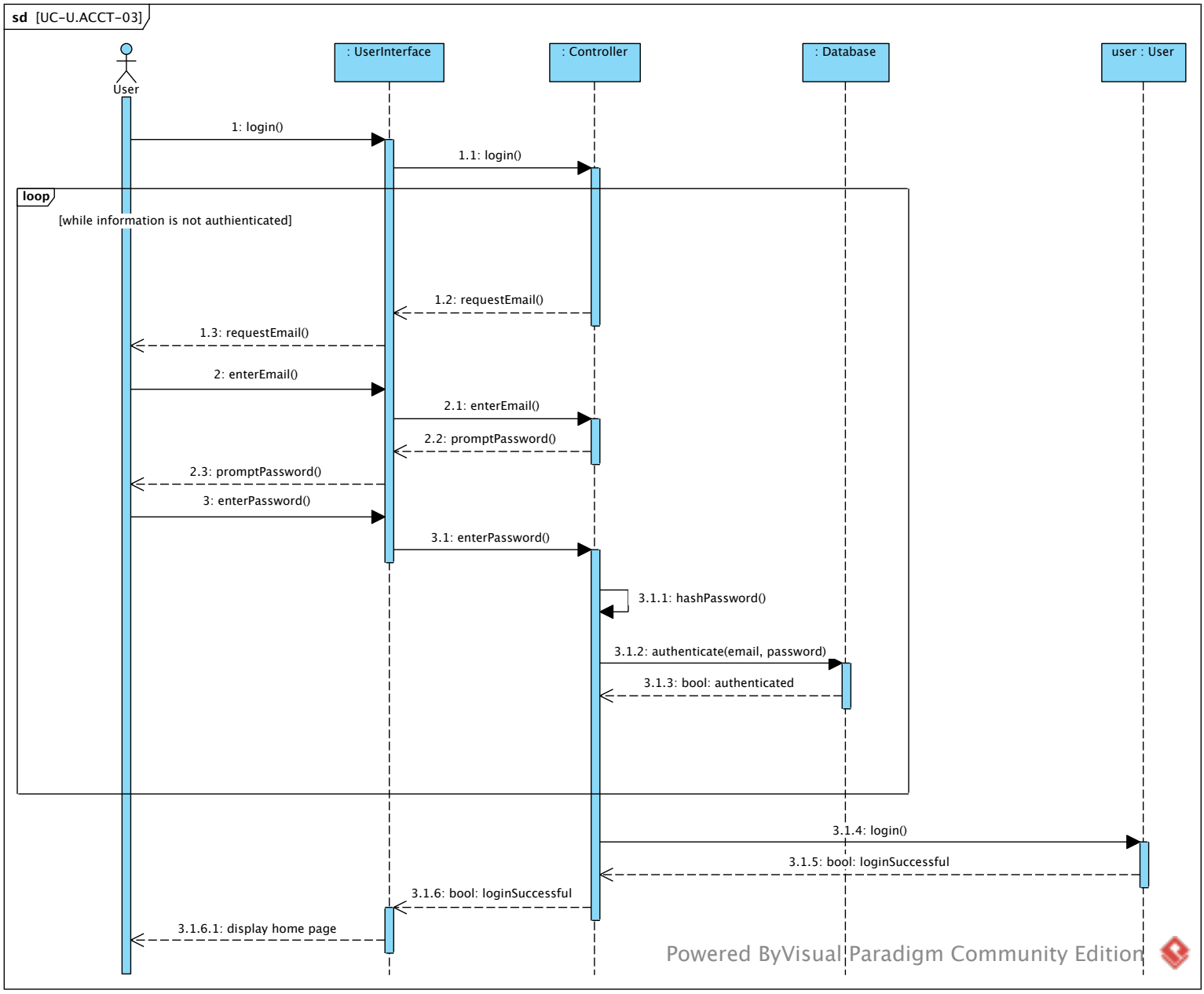

Trainer
: UserInterface
controller : TrainerController
trainerList : CourseList
: Database
selectedCourse : Course
1: click [courseMenu]
1.1: get(courseMenu)
1.2: return course menu
1.3: display course menu
2: click [upcomingCourses]
2.1: get(upcomingCourses)
2.1.1: findCourse(time)
2.1.1.1: findCourse(time, trainer)
2.1.1.2: results
2.1.2: results
2.2: results
2.3: display up to 15 results
3: select(course)
3.1: getCourseInfo(course)
3.1.1: getCourseInfo()
3.1.2: return course info
3.2: return course info
3.3: display course info and options
4: click [start]
4.1: startCourse(course)
4.1.1: startCourse()
4.1.2: return courseStarted (bool)
4.2: return courseStarted (bool)
4.3: display course started page
5: click [end]
5.1: endCourse(course)
5.1.1: endCourse()
5.1.2: return courseEnded (bool)
5.2: return courseEnded (bool)
5.3: display course ended page
Powered By Visual Paradigm Community Edition

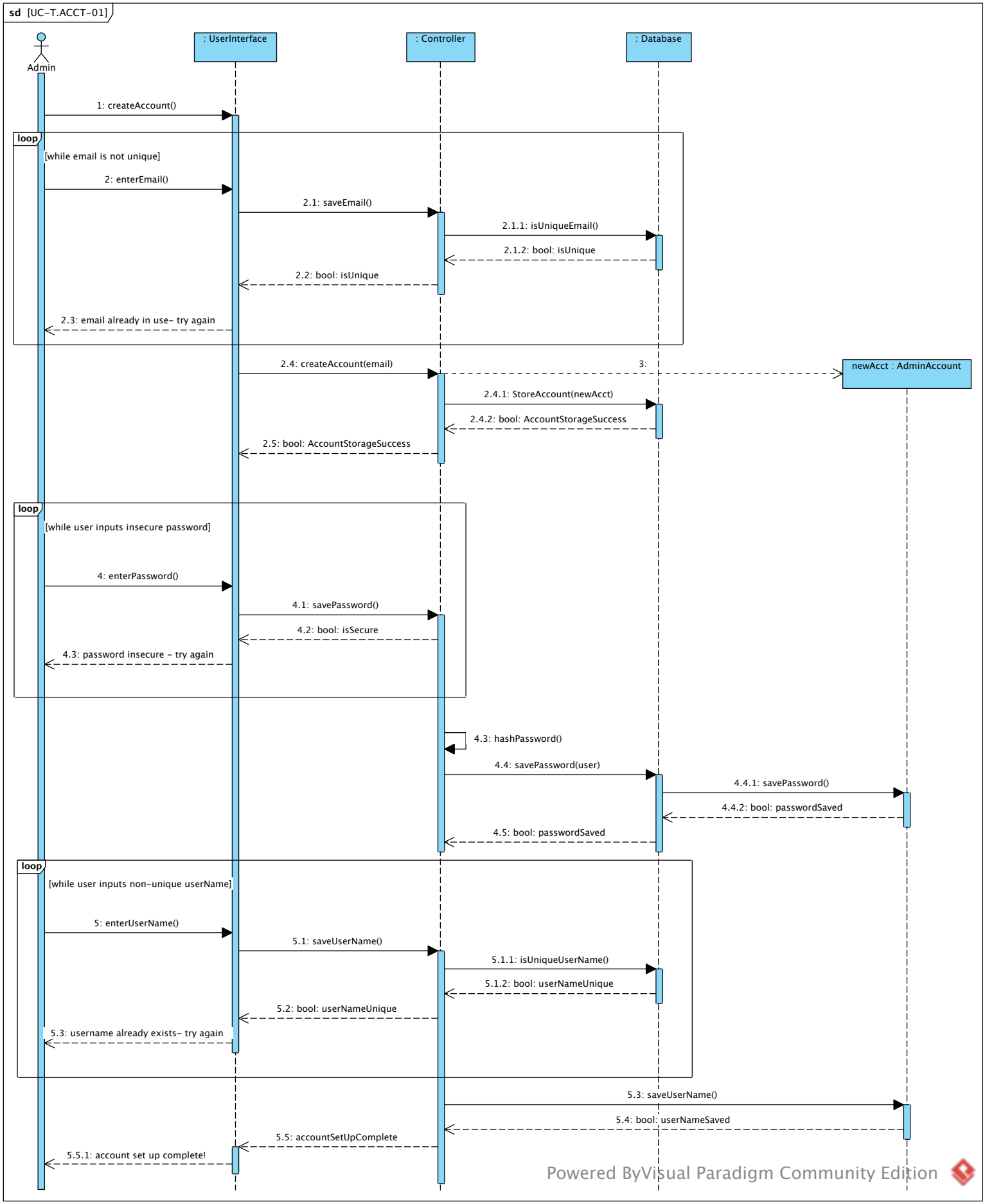

User
: UserInterface
Controller
workouts : WorkoutLog
1: createNewWorkout(workoutDate)
1.1: createNewWorkout(workoutDate)
1.1.1:
newWorkout : Workout
1.1.2: bool: workoutCreated
1.1.3: promptWorkoutInfo()
2: enterWorkoutDate()
3: enterWorkoutDescription()
4: enterWorkoutDuration()
5: enterWorkoutCaloriesBurned()
5.1: enterWorkoutInfo(date, description, duration, calories burned)
5.1.1: setDate(date)
5.1.2: setDescription(description)
5.1.3: setDuration(duration)
5.1.4: setCaloriesBurned(caloriesBurned)
5.1.5: bool: workoutAdded
5.1.6: workout logged successfully
5.1.6: addWorkout(newWorkout)
5.1.7: bool: workoutAdded
Powered By Visual Paradigm Community Edition

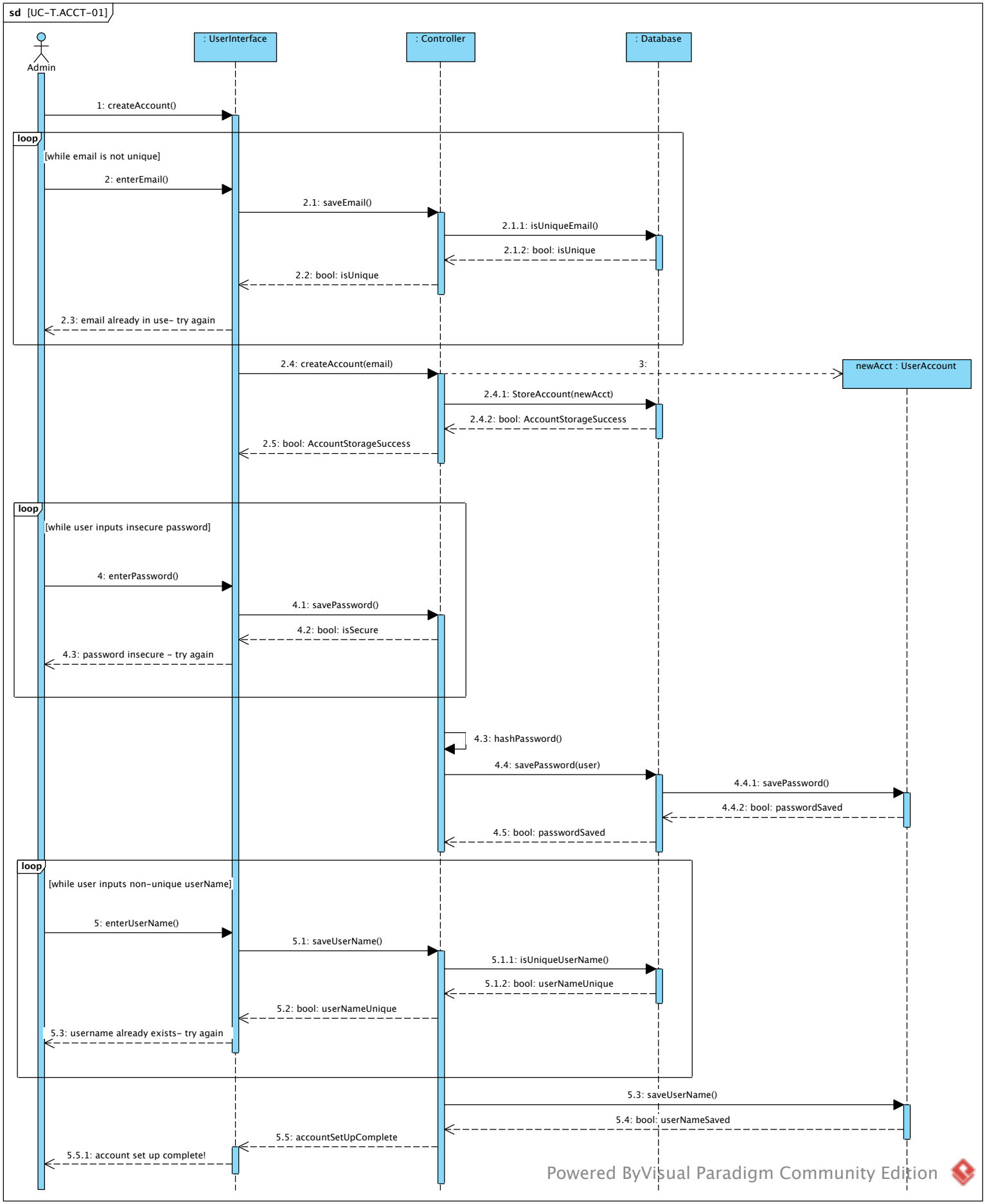

User
: UserInterface
controller : UserController
: User
: DailyMetric
1: click [profile]
1.1: getProfile()
1.1.1: getProfileInfo()
1.1.2: return profile info
1.2: return profile info
1.3: display profile menu with info
2: click [log info]
2.1: display logging page
3: enter information
3.1: display information in logging fields
4: click [submit]
4.1: confirmation message
5: click [yes]
5.1: log(water, sleep, calories, weight)
5.1.1: log(water, sleep, calories, weight)
5.1.1.1: << create >>
5.1.1.2: log(water, sleep, calories, weight)
5.1.1.3: return true
5.1.2: return true
5.2: return true
5.3: display successful log page
Powered By Visual Paradigm Community Edition

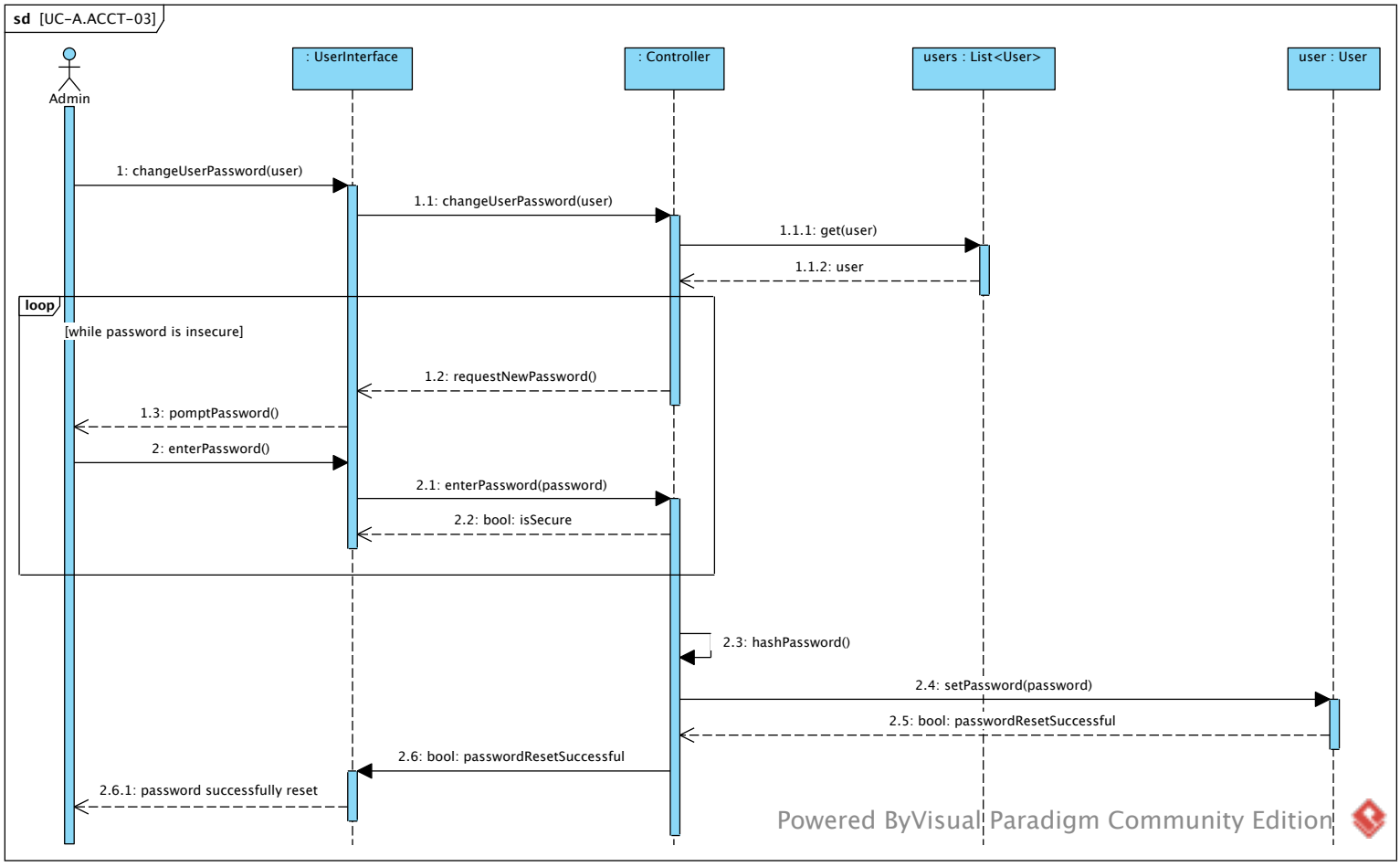

User
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email in use- try again
2.4: createAccount(email)
2.4.1:
newAcct : UserAccount
2.4.2: StoreAccount(newAcct)
2.4.3: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while password is insecure]
3: enterPassword()
3.1: savePassword
3.2: bool: isSecure
3.3: password insecure - try again
3.3: hashPassword()
3.4: savePassword(password)
3.4.1: savePassword()
3.4.2: bool: passwordSaved
3.5: bool: passwordSaved
loop
[while username is not unique]
4: enterUserName()
4.1: saveUserName()
4.1.1: isUniqueUserName()
4.1.2: bool: userNameUnique
4.2: bool: userNameUnique
4.3: username already exists - try again
4.3: saveUserName()
4.4: bool: userNameSaved
4.5: bool: accountSetUpComplete
4.5.1: account set up complete!
Powered By Visual Paradigm Community Edition

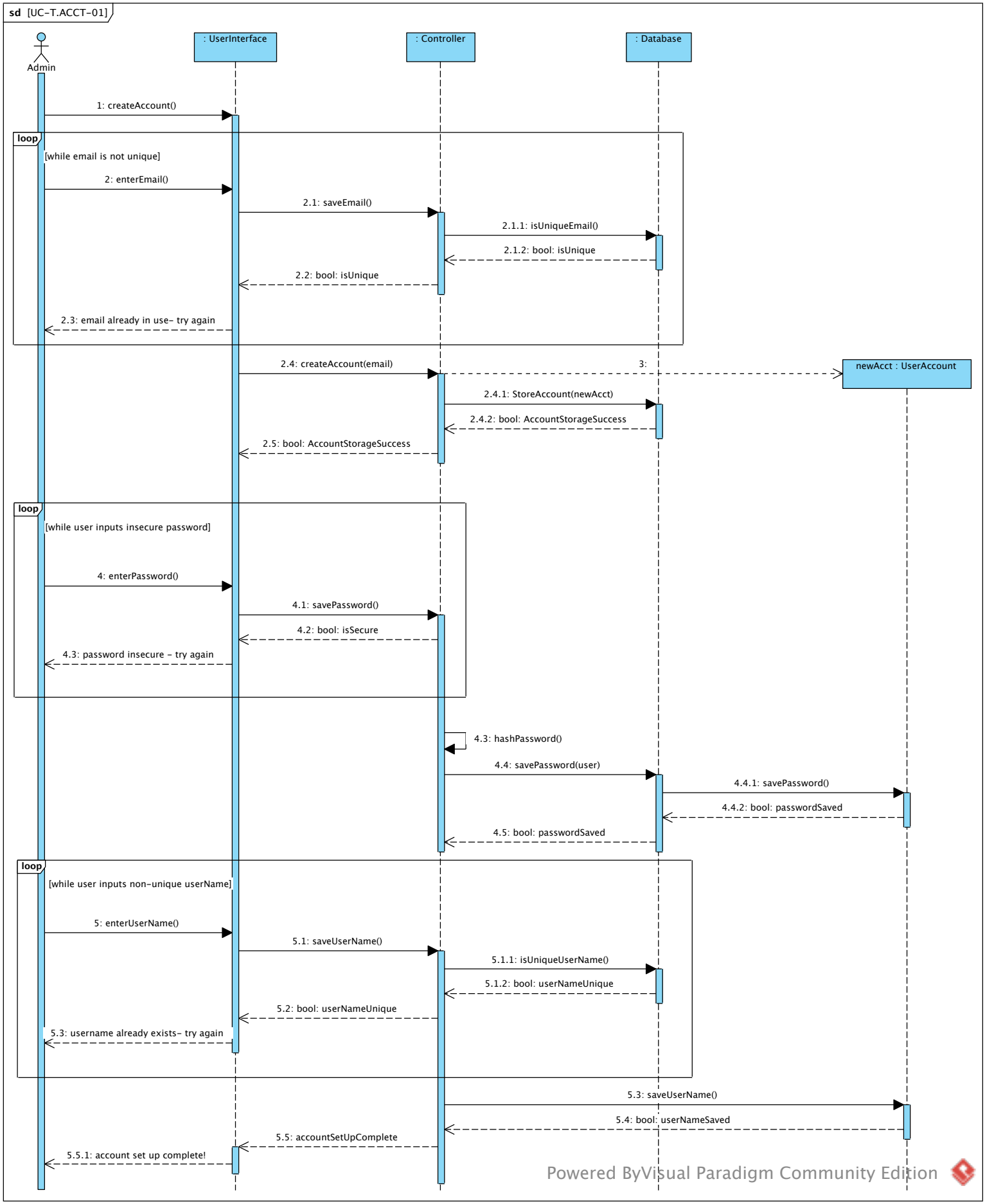

User
: UserInterface
: Controller
: Database
This is a hard coded list stored locally until this is actually implemented
user : User
1: forgotPassword()
1.1: displayForgotPassword()
loop
[email doesn't match an account]
2: resetEnterEmail()
2.1: resetEnterEmail(email)
2.1.1: doesEmailExist(email)
2.1.2: bool: emailExists
2.2: bool: emailExists
2.3: email doesn't match account - try again!
2.3: generates reset password link
2.4: bool: sentResetEmail
2.4.1: sent reset link to email!
3: clickResetLink()
3.1: clickResetLink(email)
3.1.1: activeResetLink(email)
3.1.2: bool: validResetLink
3.2: bool: validResetLink
3.3: please enter a new password!
loop
[while password is insecure]
4: enterPassword()
4.1: enterPassword(password)
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: setPassword(hashedPassword)
4.5: bool: passwordResetSuccessful
4.6: sendPasswordResetNotif(email)
4.7: bool: passwordResetSuccessful
4.7.1: password successfully reset!
Powered By Visual Paradigm Community Edition

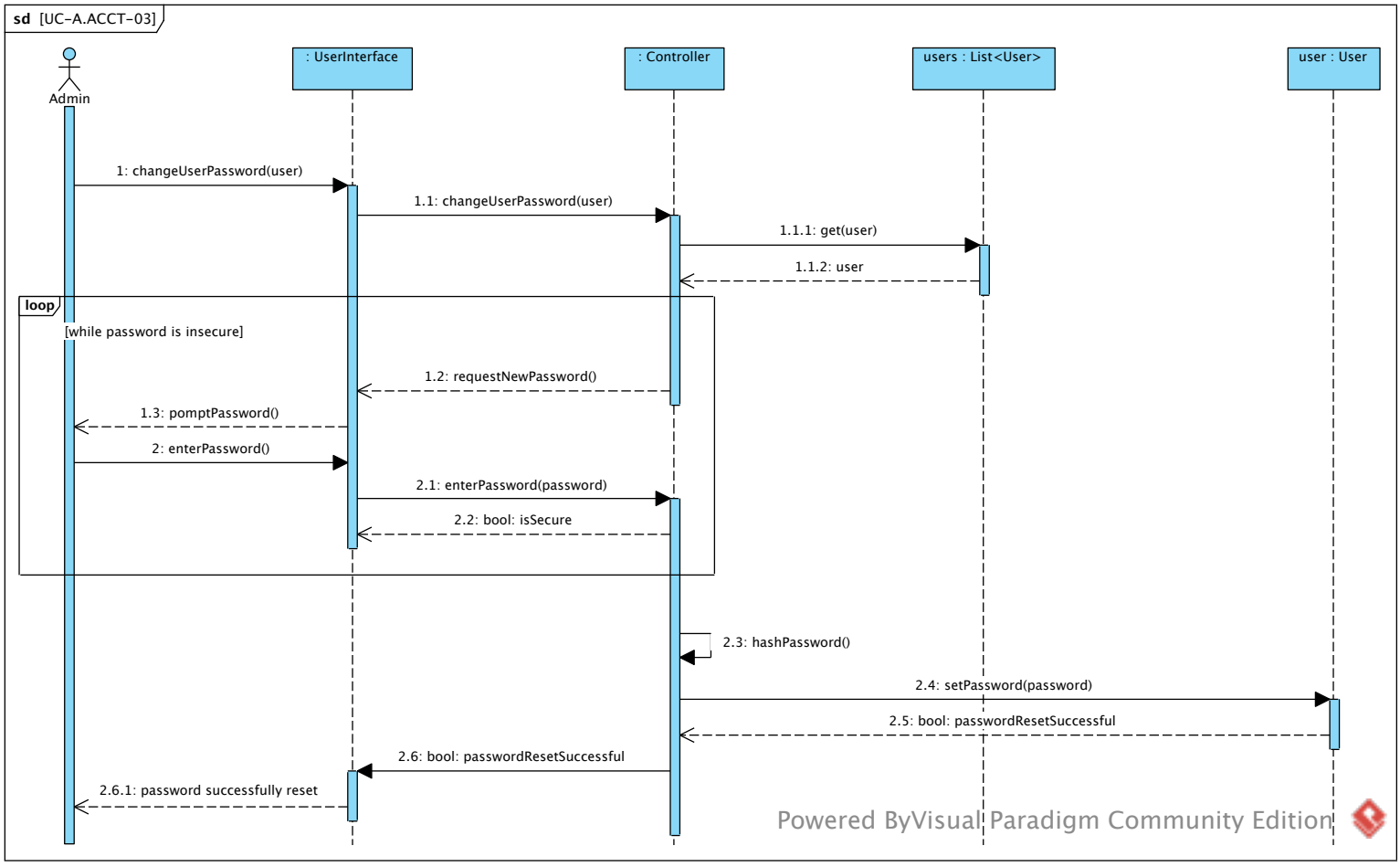

sd [UC-U.ACCT-03]
User
: UserInterface
: Controller
: Database
user : User
1: login()
1.1: login()
loop
[while information is not authienticated]
1.2: requestEmail()
1.3: requestEmail()
2: enterEmail()
2.1: enterEmail()
2.2: promptPassword()
2.3: promptPassword()
3: enterPassword()
3.1: enterPassword()
3.1.1: hashPassword()
3.1.2: authenticate(email, password)
3.1.3: bool: authenticated
3.1.4: login()
3.1.5: bool: loginSuccessful
3.1.6: bool: loginSuccessful
3.1.6.1: display home page
Powered ByVisual Paradigm Community Edition

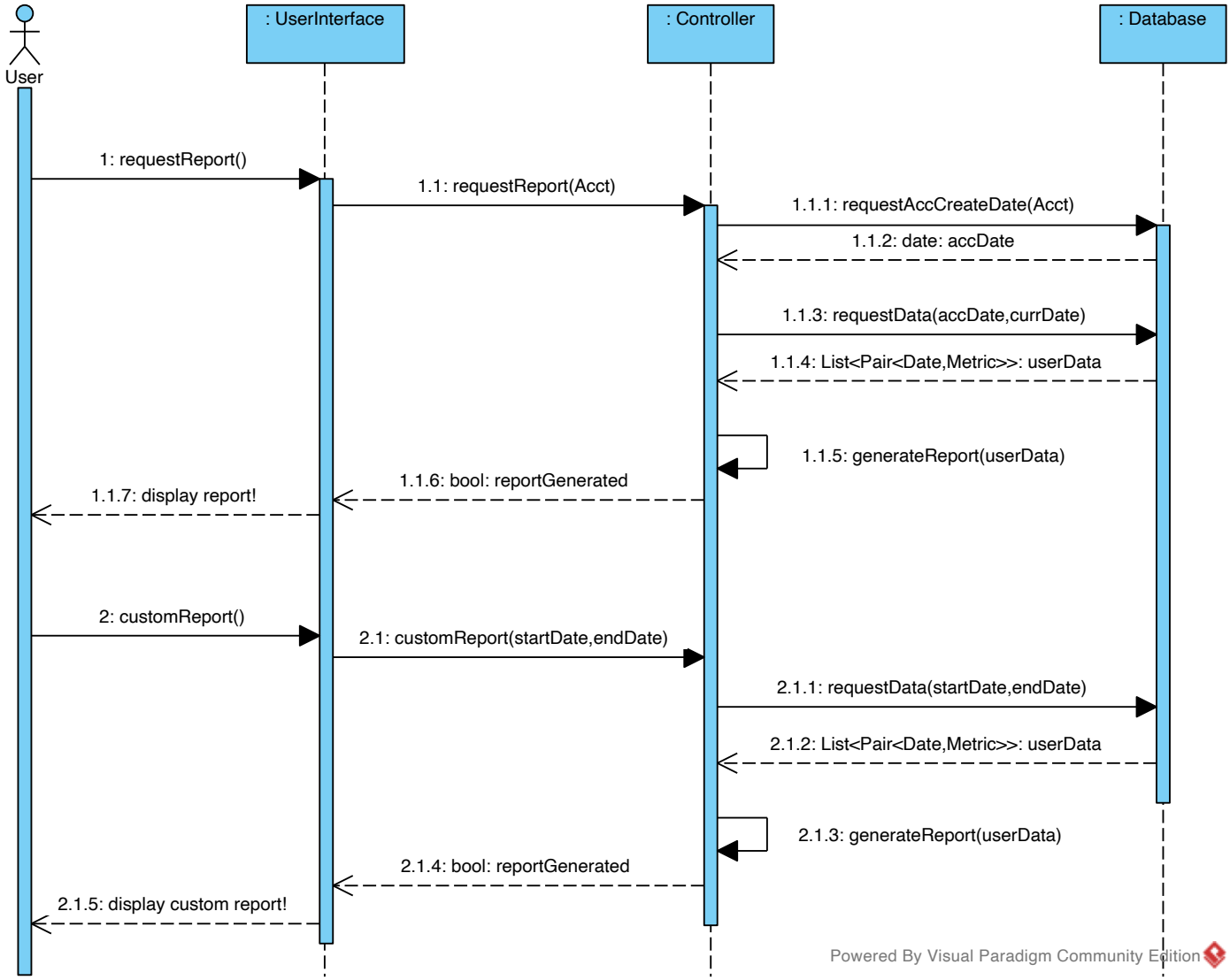

sd [UC-T.ACCT-01]
Admin
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email already in use- try again
2.4: createAccount(email)
3:
newAcct : AdminAccount
2.4.1: StoreAccount(newAcct)
2.4.2: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while user inputs insecure password]
4: enterPassword()
4.1: savePassword()
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: savePassword(user)
4.4.1: savePassword()
4.4.2: bool: passwordSaved
4.5: bool: passwordSaved
loop
[while user inputs non-unique userName]
5: enterUserName()
5.1: saveUserName()
5.1.1: isUniqueUserName()
5.1.2: bool: userNameUnique
5.2: bool: userNameUnique
5.3: username already exists- try again
5.3: saveUserName()
5.4: bool: userNameSaved
5.5: accountSetUpComplete
5.5.1: account set up complete!
Powered ByVisual Paradigm Community Edition

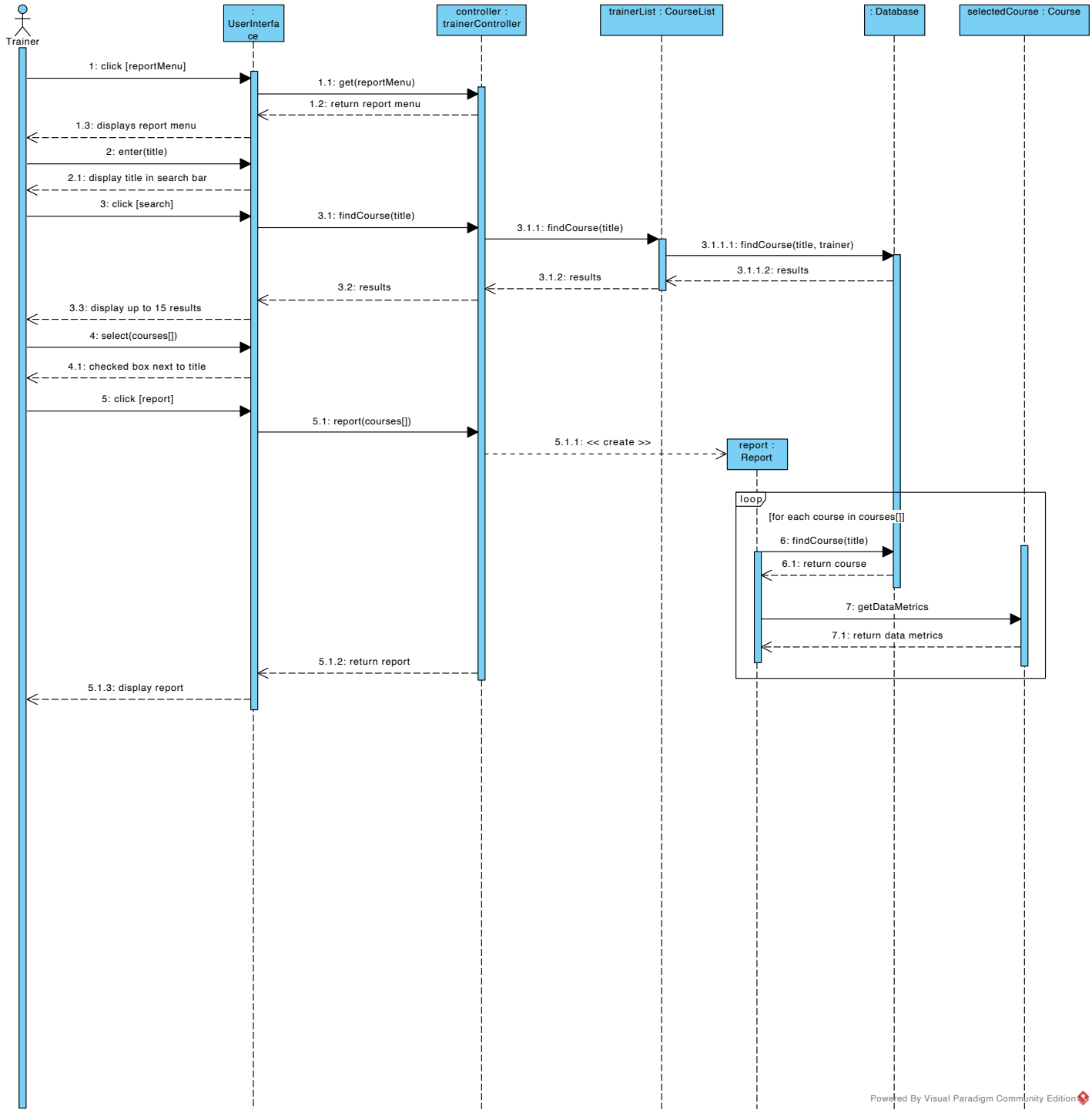

sd [UC-T.ACCT-01]
Admin
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email already in use- try again
2.4: createAccount(email)
3:
newAcct : UserAccount
2.4.1: StoreAccount(newAcct)
2.4.2: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while user inputs insecure password]
4: enterPassword()
4.1: savePassword()
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: savePassword(user)
4.4.1: savePassword()
4.4.2: bool: passwordSaved
4.5: bool: passwordSaved
loop
[while user inputs non-unique userName]
5: enterUserName()
5.1: saveUserName()
5.1.1: isUniqueUserName()
5.1.2: bool: userNameUnique
5.2: bool: userNameUnique
5.3: username already exists- try again
5.3: saveUserName()
5.4: bool: userNameSaved
5.5: accountSetUpComplete
5.5.1: account set up complete!
Powered ByVisual Paradigm Community Edition

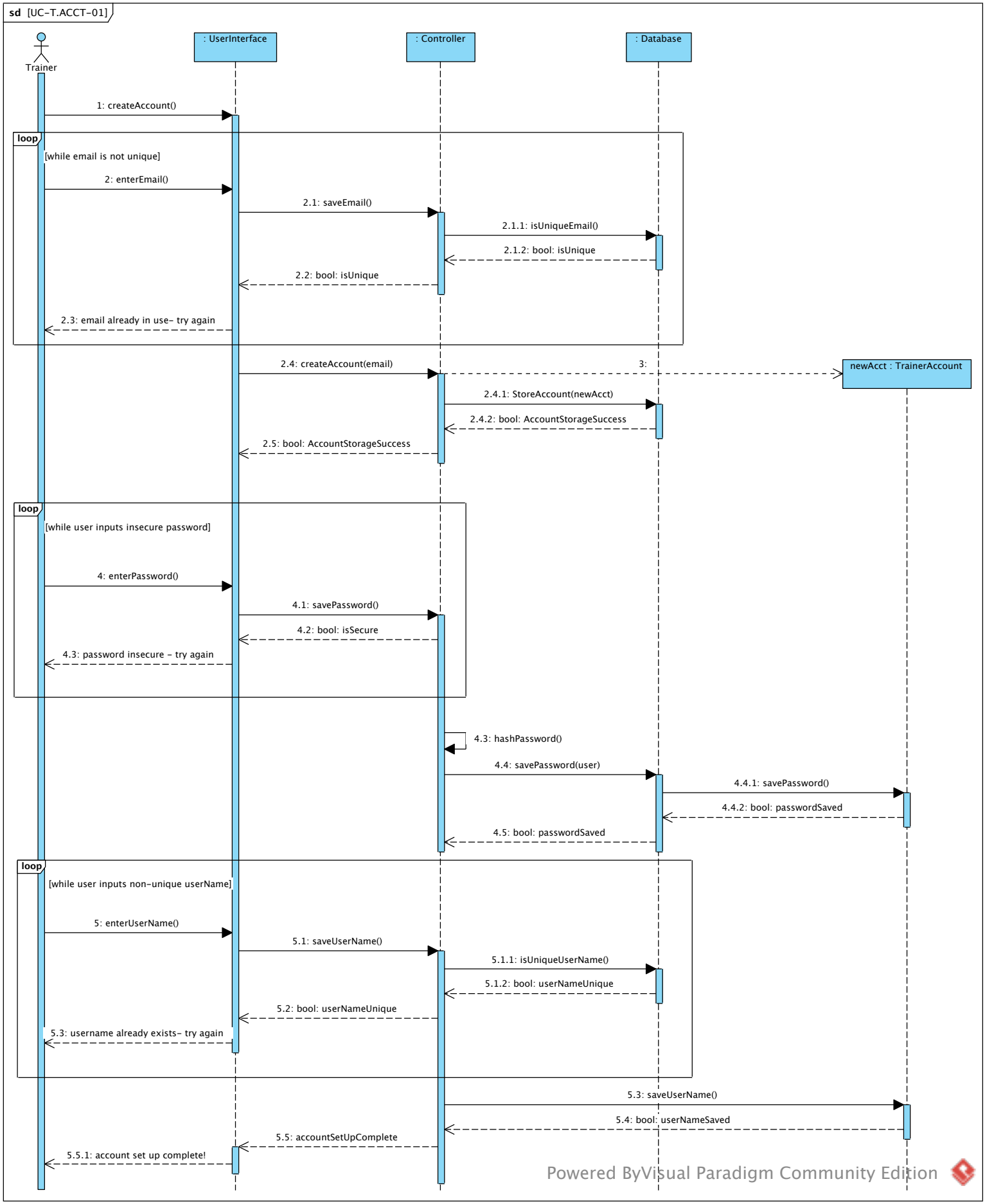

sd [UC-A.ACCT-03]
Admin
: UserInterface
: Controller
users : List<User>
user : User
1: changeUserPassword(user)
1.1: changeUserPassword(user)
1.1.1: get(user)
1.1.2: user
loop
[while password is insecure]
1.2: requestNewPassword()
1.3: pomptPassword()
2: enterPassword()
2.1: enterPassword(password)
2.2: bool: isSecure
2.3: hashPassword()
2.4: setPassword(password)
2.5: bool: passwordResetSuccessful
2.6: bool: passwordResetSuccessful
2.6.1: password successfully reset
Powered ByVisual Paradigm Community Edition

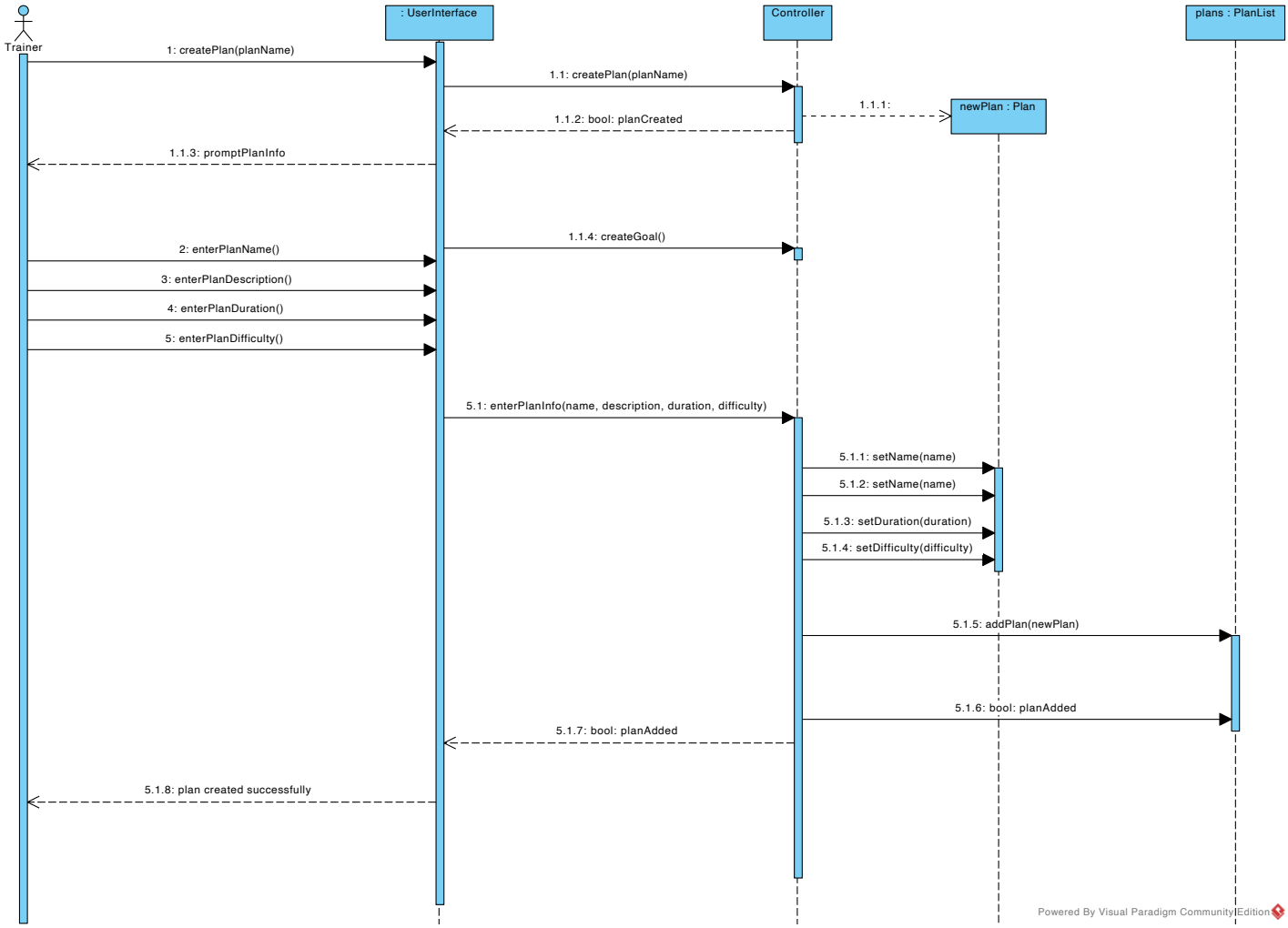

sd [UC-T.ACCT-01]
Admin
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email already in use- try again
2.4: createAccount(email)
3:
newAcct : UserAccount
2.4.1: StoreAccount(newAcct)
2.4.2: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while user inputs insecure password]
4: enterPassword()
4.1: savePassword()
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: savePassword(user)
4.4.1: savePassword()
4.4.2: bool: passwordSaved
4.5: bool: passwordSaved
loop
[while user inputs non-unique userName]
5: enterUserName()
5.1: saveUserName()
5.1.1: isUniqueUserName()
5.1.2: bool: userNameUnique
5.2: bool: userNameUnique
5.3: username already exists- try again
5.3: saveUserName()
5.4: bool: userNameSaved
5.5: accountSetUpComplete
5.5.1: account set up complete!
Powered ByVisual Paradigm Community Edition

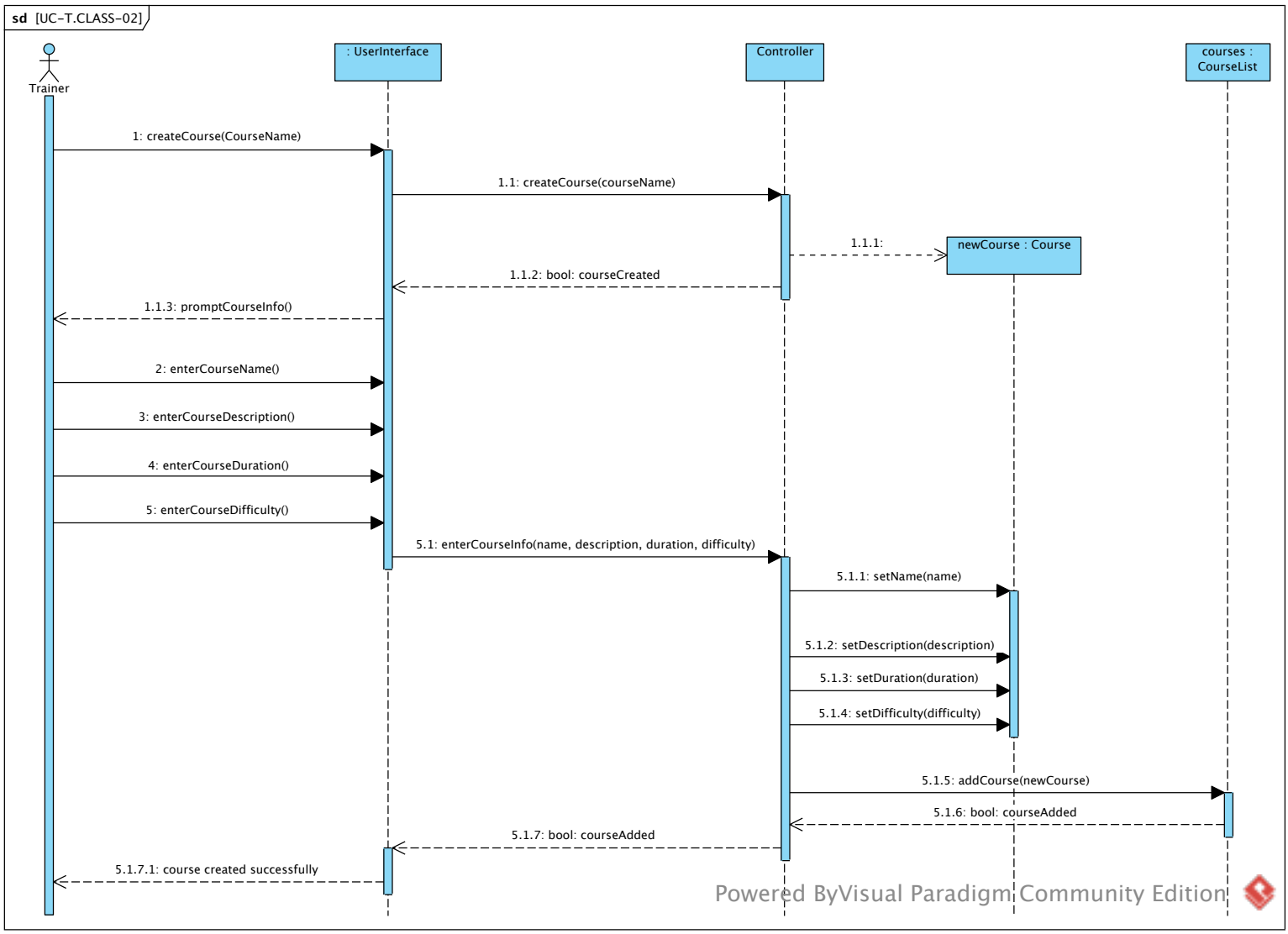

sd [UC-A.ACCT-03]
Admin
: UserInterface
: Controller
users : List<User>
user : User
1: changeUserPassword(user)
1.1: changeUserPassword(user)
1.1.1: get(user)
1.1.2: user
loop
[while password is insecure]
1.2: requestNewPassword()
1.3: pomptPassword()
2: enterPassword()
2.1: enterPassword(password)
2.2: bool: isSecure
2.3: hashPassword()
2.4: setPassword(password)
2.5: bool: passwordResetSuccessful
2.6: bool: passwordResetSuccessful
2.6.1: password successfully reset
Powered ByVisual Paradigm Community Edition

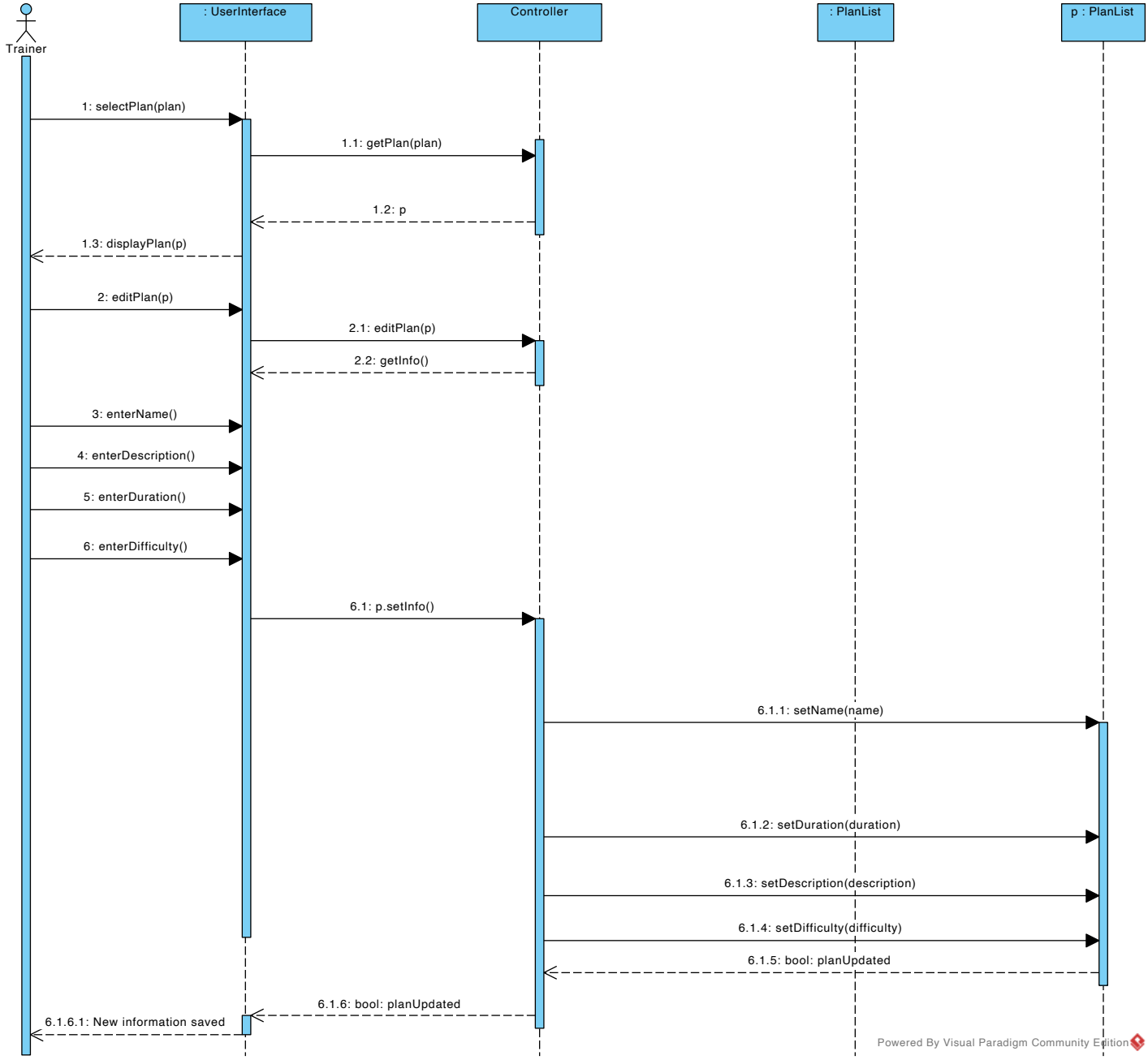

User
: UserInterface
: Controller
: Database
1: requestReport()
1.1: requestReport(Acct)
1.1.1: requestAccCreateDate(Acct)
1.1.2: date: accDate
1.1.3: requestData(accDate,currDate)
1.1.4: List<Pair<Date,Metric>>: userData
1.1.5: generateReport(userData)
1.1.6: bool: reportGenerated
1.1.7: display report!
2: customReport()
2.1: customReport(startDate,endDate)
2.1.1: requestData(startDate,endDate)
2.1.2: List<Pair<Date,Metric>>: userData
2.1.3: generateReport(userData)
2.1.4: bool: reportGenerated
2.1.5: display custom report!
Powered By Visual Paradigm Community Edition

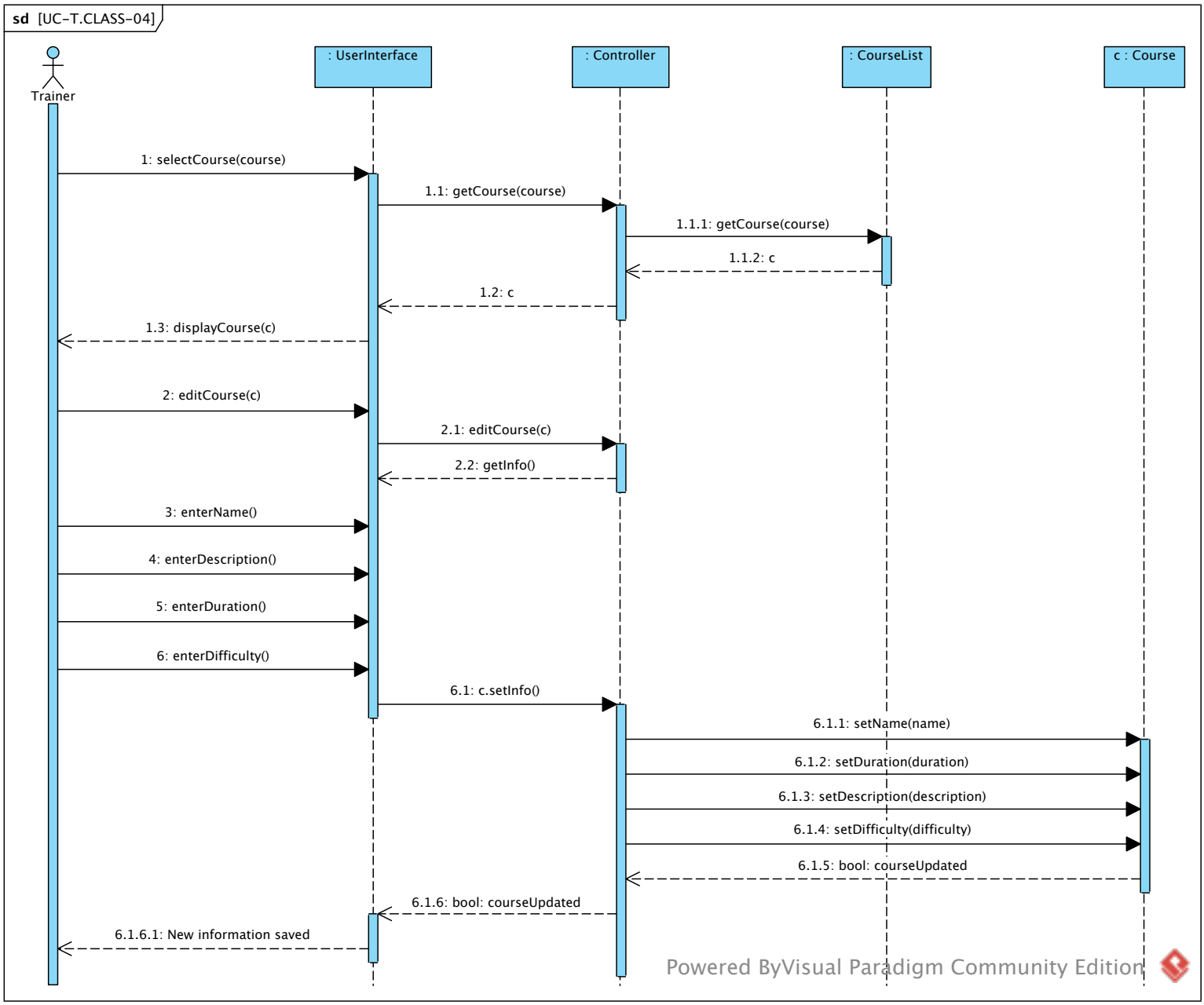

Trainer
: UserInterface
controller : trainerController
trainerList : CourseList
: Database
selectedCourse : Course
1: click [reportMenu]
1.1: get(reportMenu)
1.2: return report menu
1.3: displays report menu
2: enter(title)
2.1: display title in search bar
3: click [search]
3.1: findCourse(title)
3.1.1: findCourse(title)
3.1.1.1: findCourse(title, trainer)
3.1.1.2: results
3.1.2: results
3.2: results
3.3: display up to 15 results
4: select(courses[])
4.1: checked box next to title
5: click [report]
5.1: report(courses[])
5.1.1: << create >>
report : Report
loop
[for each course in courses[]]
6: findCourse(title)
6.1: return course
7: getDataMetrics
7.1: return data metrics
5.1.2: return report
5.1.3: display report
Powered By Visual Paradigm Community Edition

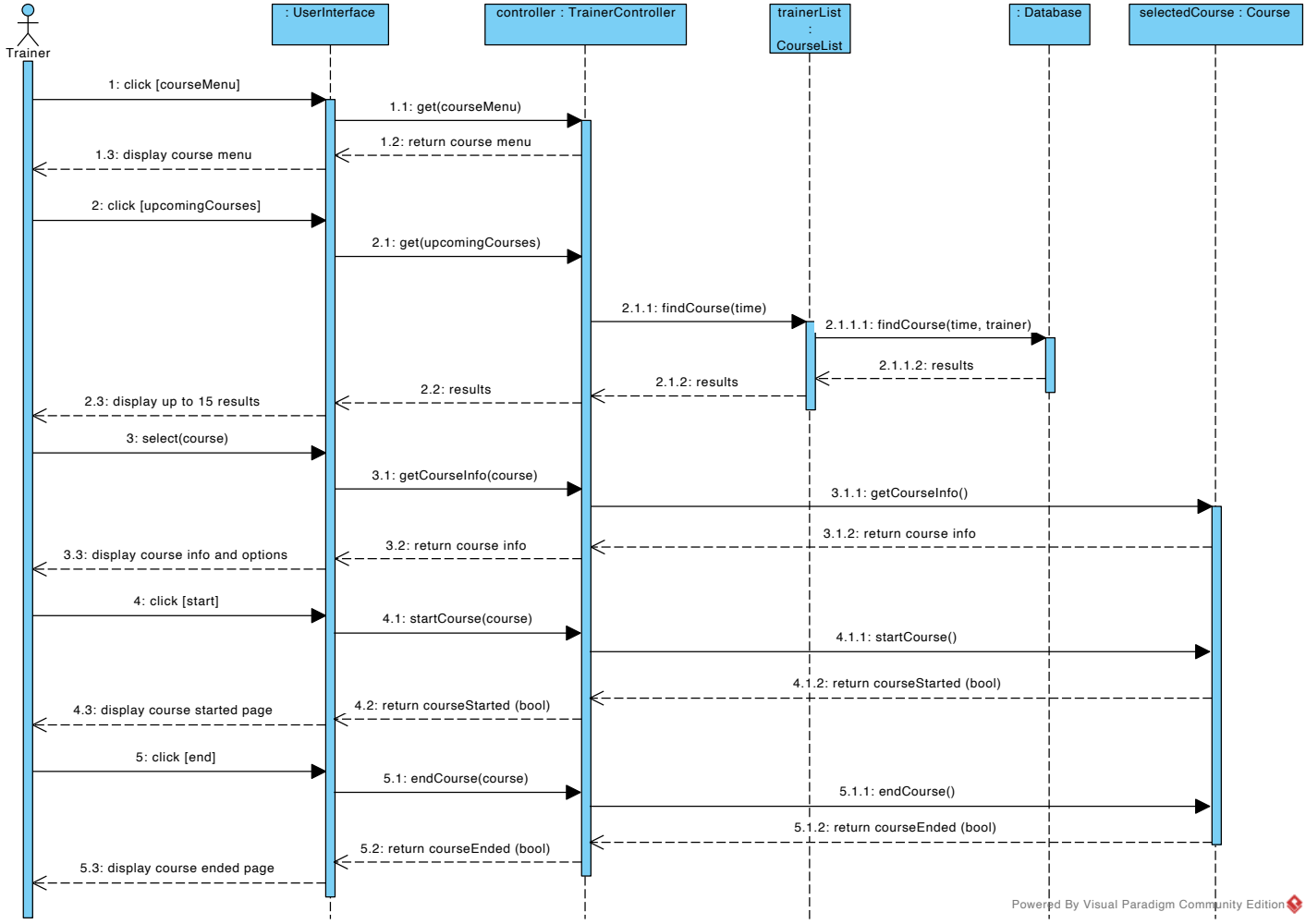

sd [UC-T.ACCT-01]
Trainer
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email already in use- try again
2.4: createAccount(email)
3:
newAcct : TrainerAccount
2.4.1: StoreAccount(newAcct)
2.4.2: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while user inputs insecure password]
4: enterPassword()
4.1: savePassword()
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: savePassword(user)
4.4.1: savePassword()
4.4.2: bool: passwordSaved
4.5: bool: passwordSaved
loop
[while user inputs non-unique userName]
5: enterUserName()
5.1: saveUserName()
5.1.1: isUniqueUserName()
5.1.2: bool: userNameUnique
5.2: bool: userNameUnique
5.3: username already exists- try again
5.3: saveUserName()
5.4: bool: userNameSaved
5.5: accountSetUpComplete
5.5.1: account set up complete!
Powered ByVisual Paradigm Community Edition

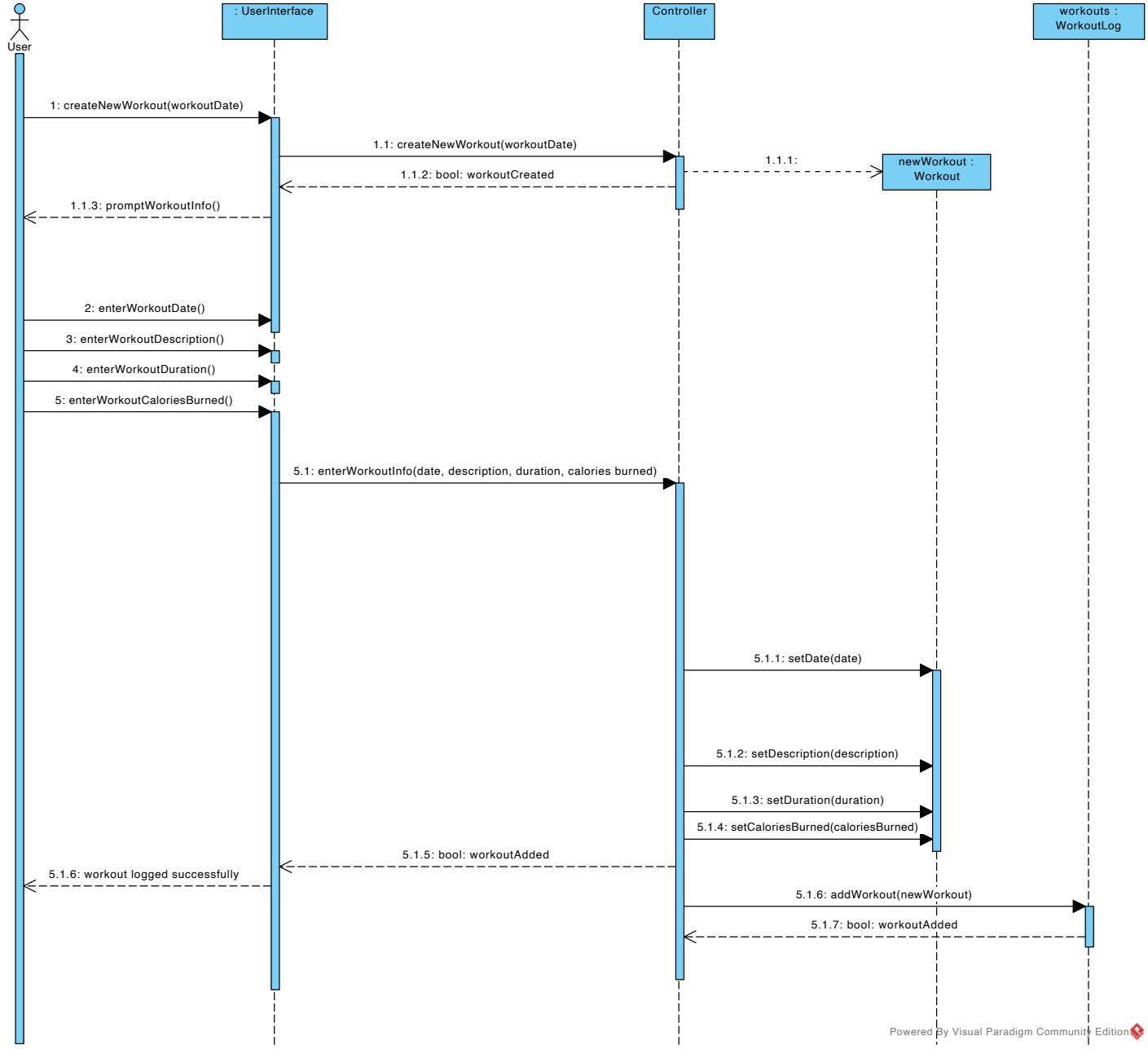

Trainer
: UserInterface
Controller
plans : PlanList
newPlan : Plan
1: createPlan(planName)
1.1: createPlan(planName)
1.1.1:
1.1.2: bool: planCreated
1.1.3: promptPlanInfo
1.1.4: createGoal()
2: enterPlanName()
3: enterPlanDescription()
4: enterPlanDuration()
5: enterPlanDifficulty()
5.1: enterPlanInfo(name, description, duration, difficulty)
5.1.1: setName(name)
5.1.2: setName(name)
5.1.3: setDuration(duration)
5.1.4: setDifficulty(difficulty)
5.1.5: addPlan(newPlan)
5.1.6: bool: planAdded
5.1.7: bool: planAdded
5.1.8: plan created successfully
Powered By Visual Paradigm Community Edition

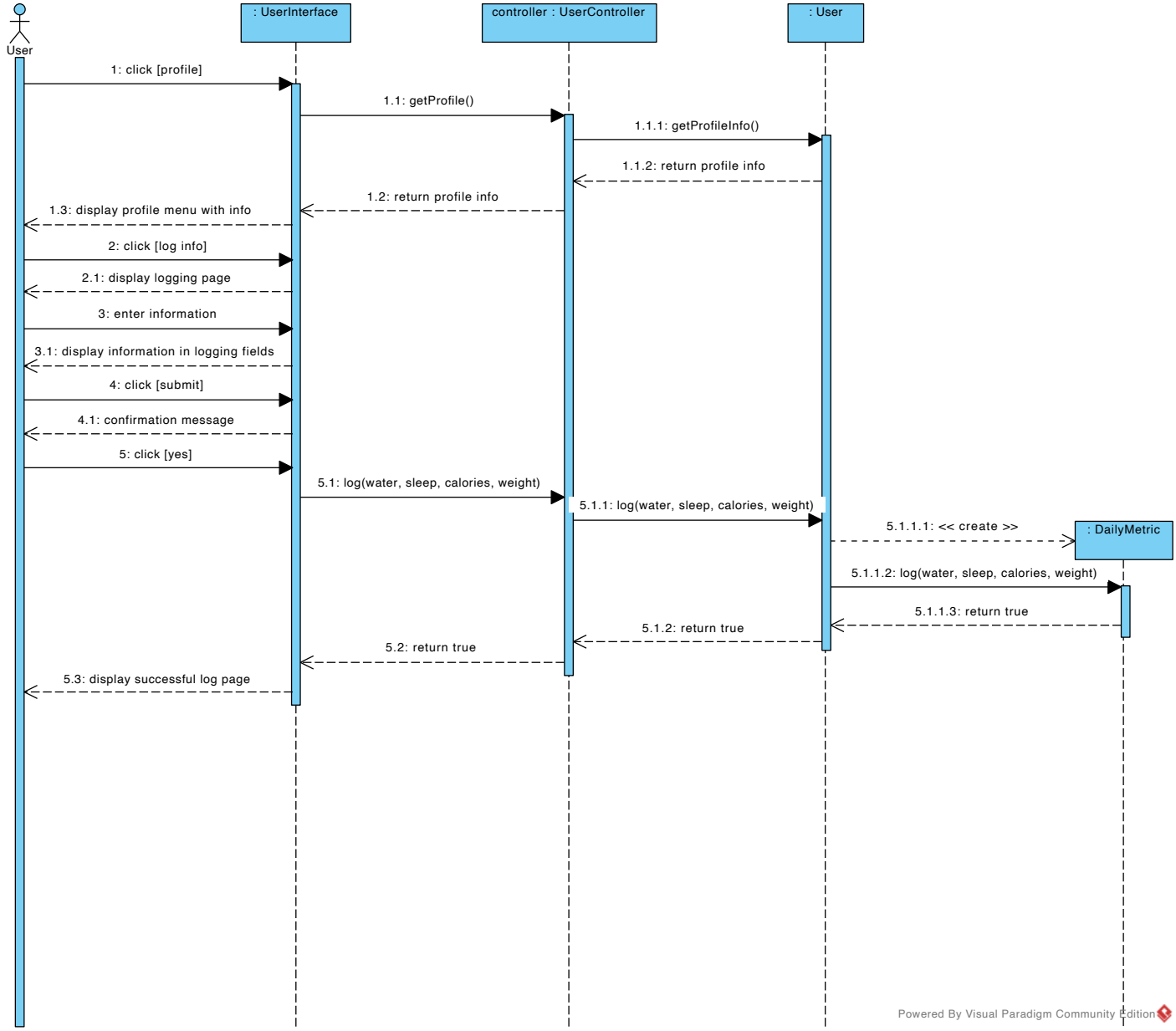

sd [UC-T.CLASS-02]
Trainer
: UserInterface
Controller
courses : CourseList
1: createCourse(CourseName)
1.1: createCourse(courseName)
1.1.1:
newCourse : Course
1.1.2: bool: courseCreated
1.1.3: promptCourseInfo()
2: enterCourseName()
3: enterCourseDescription()
4: enterCourseDuration()
5: enterCourseDifficulty()
5.1: enterCourseInfo(name, description, duration, difficulty)
5.1.1: setName(name)
5.1.2: setDescription(description)
5.1.3: setDuration(duration)
5.1.4: setDifficulty(difficulty)
5.1.5: addCourse(newCourse)
5.1.6: bool: courseAdded
5.1.7: bool: courseAdded
5.1.7.1: course created successfully
Powered ByVisual Paradigm Community Edition

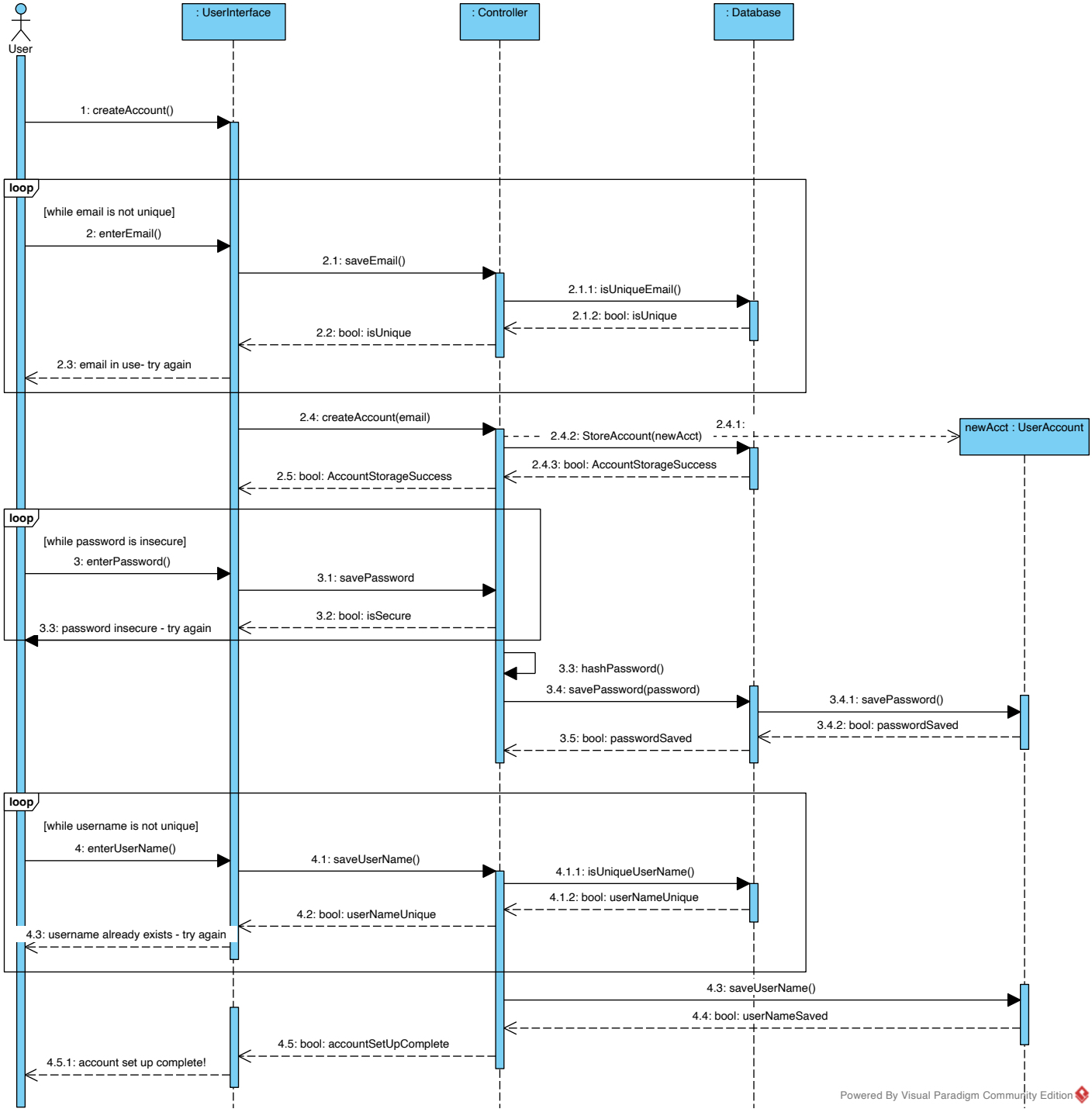

Trainer
: UserInterface
Controller
: PlanList
p : PlanList
1: selectPlan(plan)
1.1: getPlan(plan)
1.2: p
1.3: displayPlan(p)
2: editPlan(p)
2.1: editPlan(p)
2.2: getInfo()
3: enterName()
4: enterDescription()
5: enterDuration()
6: enterDifficulty()
6.1: p.setInfo()
6.1.1: setName(name)
6.1.2: setDuration(duration)
6.1.3: setDescription(description)
6.1.4: setDifficulty(difficulty)
6.1.5: bool: planUpdated
6.1.6: bool: planUpdated
6.1.6.1: New information saved
Powered By Visual Paradigm Community Edition

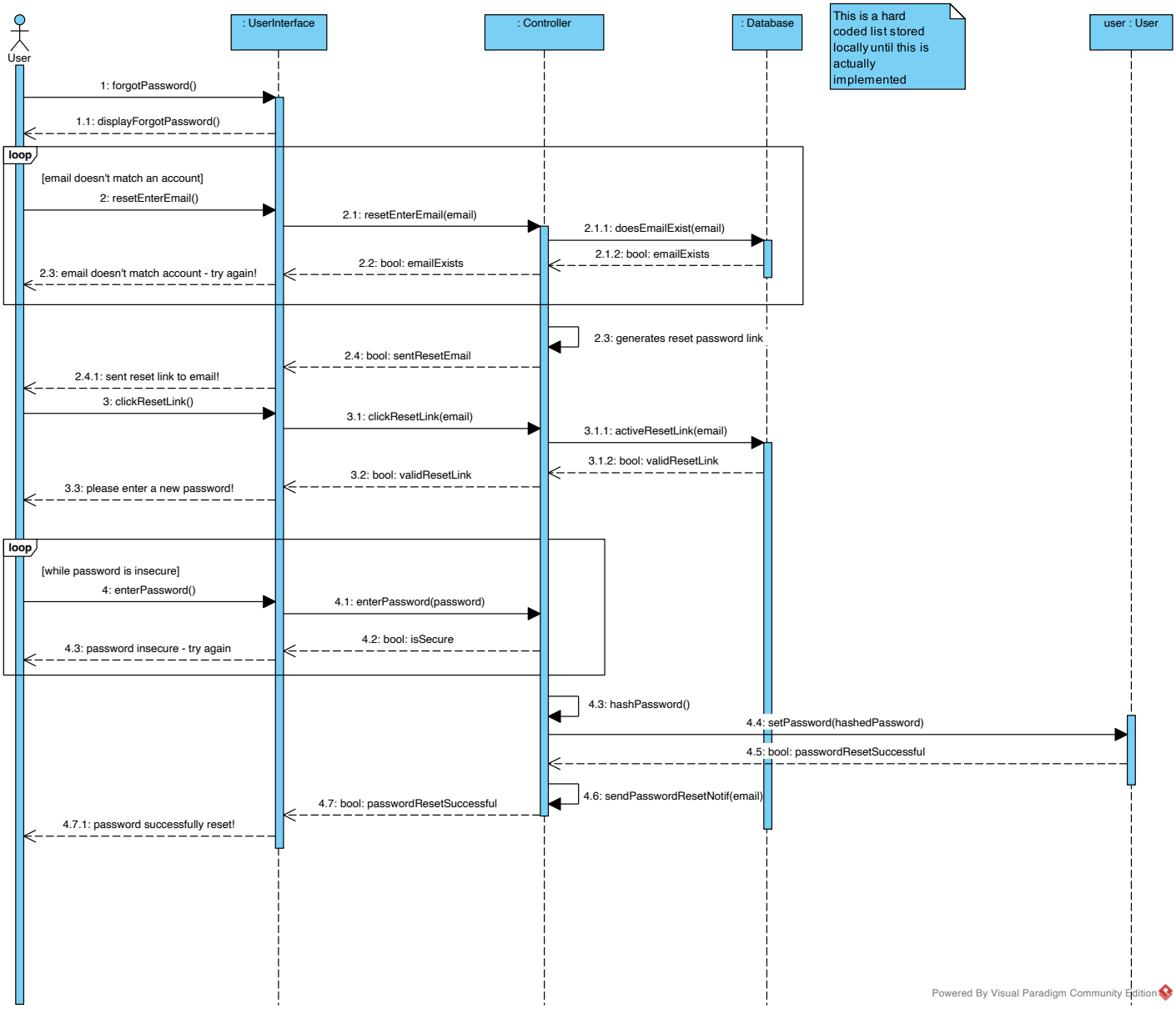

sd [UC-T.CLASS-04]
Trainer
: UserInterface
: Controller
: CourseList
c : Course
1: selectCourse(course)
1.1: getCourse(course)
1.1.1: getCourse(course)
1.1.2: c
1.2: c
1.3: displayCourse(c)
2: editCourse(c)
2.1: editCourse(c)
2.2: getInfo()
3: enterName()
4: enterDescription()
5: enterDuration()
6: enterDifficulty()
6.1: c.setInfo()
6.1.1: setName(name)
6.1.2: setDuration(duration)
6.1.3: setDescription(description)
6.1.4: setDifficulty(difficulty)
6.1.5: bool: courseUpdated
6.1.6: bool: courseUpdated
6.1.6.1: New information saved
Powered ByVisual Paradigm Community Edition

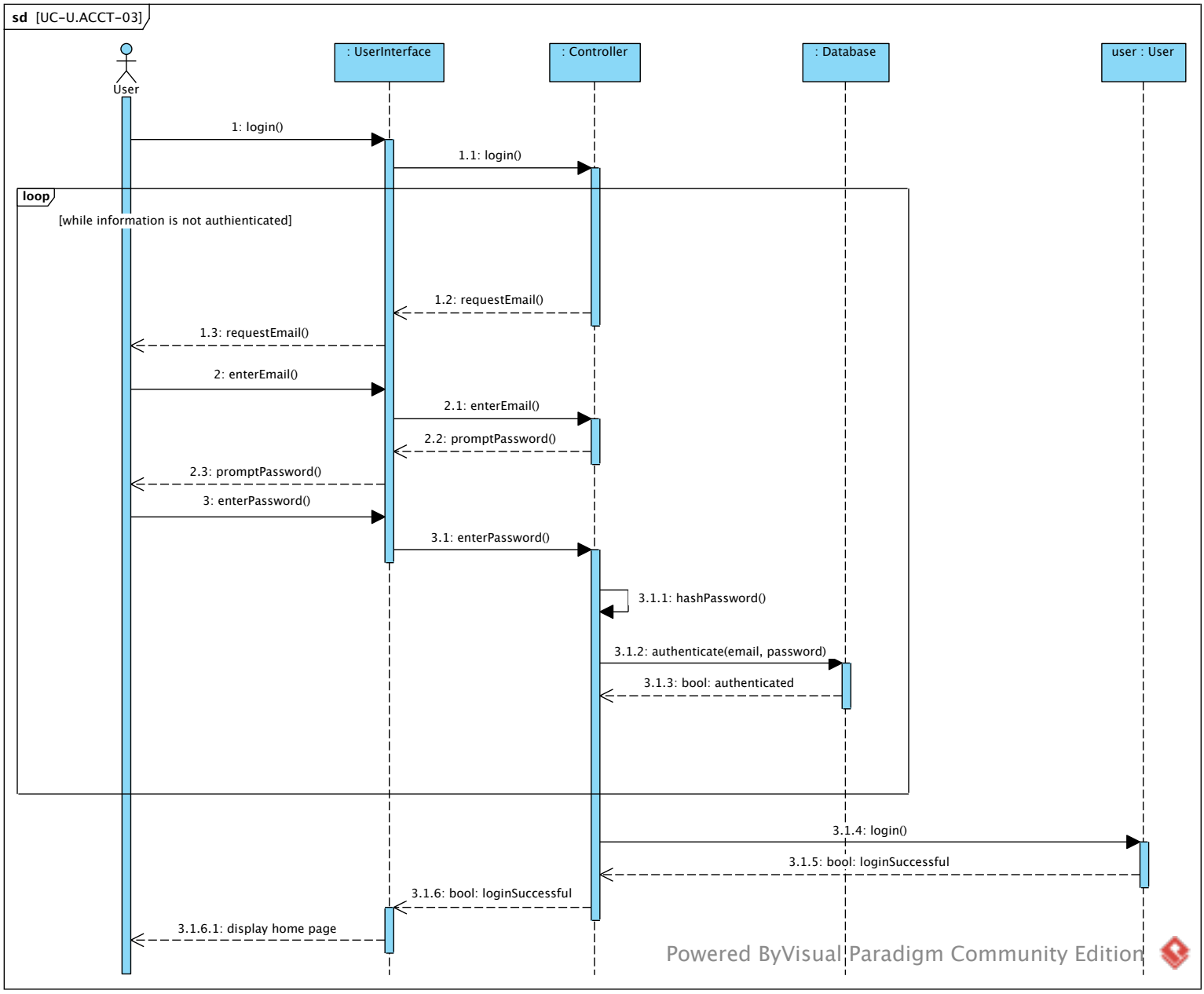

Trainer
: UserInterface
controller : TrainerController
trainerList : CourseList
: Database
selectedCourse : Course
1: click [courseMenu]
1.1: get(courseMenu)
1.2: return course menu
1.3: display course menu
2: click [upcomingCourses]
2.1: get(upcomingCourses)
2.1.1: findCourse(time)
2.1.1.1: findCourse(time, trainer)
2.1.1.2: results
2.1.2: results
2.2: results
2.3: display up to 15 results
3: select(course)
3.1: getCourseInfo(course)
3.1.1: getCourseInfo()
3.1.2: return course info
3.2: return course info
3.3: display course info and options
4: click [start]
4.1: startCourse(course)
4.1.1: startCourse()
4.1.2: return courseStarted (bool)
4.2: return courseStarted (bool)
4.3: display course started page
5: click [end]
5.1: endCourse(course)
5.1.1: endCourse()
5.1.2: return courseEnded (bool)
5.2: return courseEnded (bool)
5.3: display course ended page
Powered By Visual Paradigm Community Edition

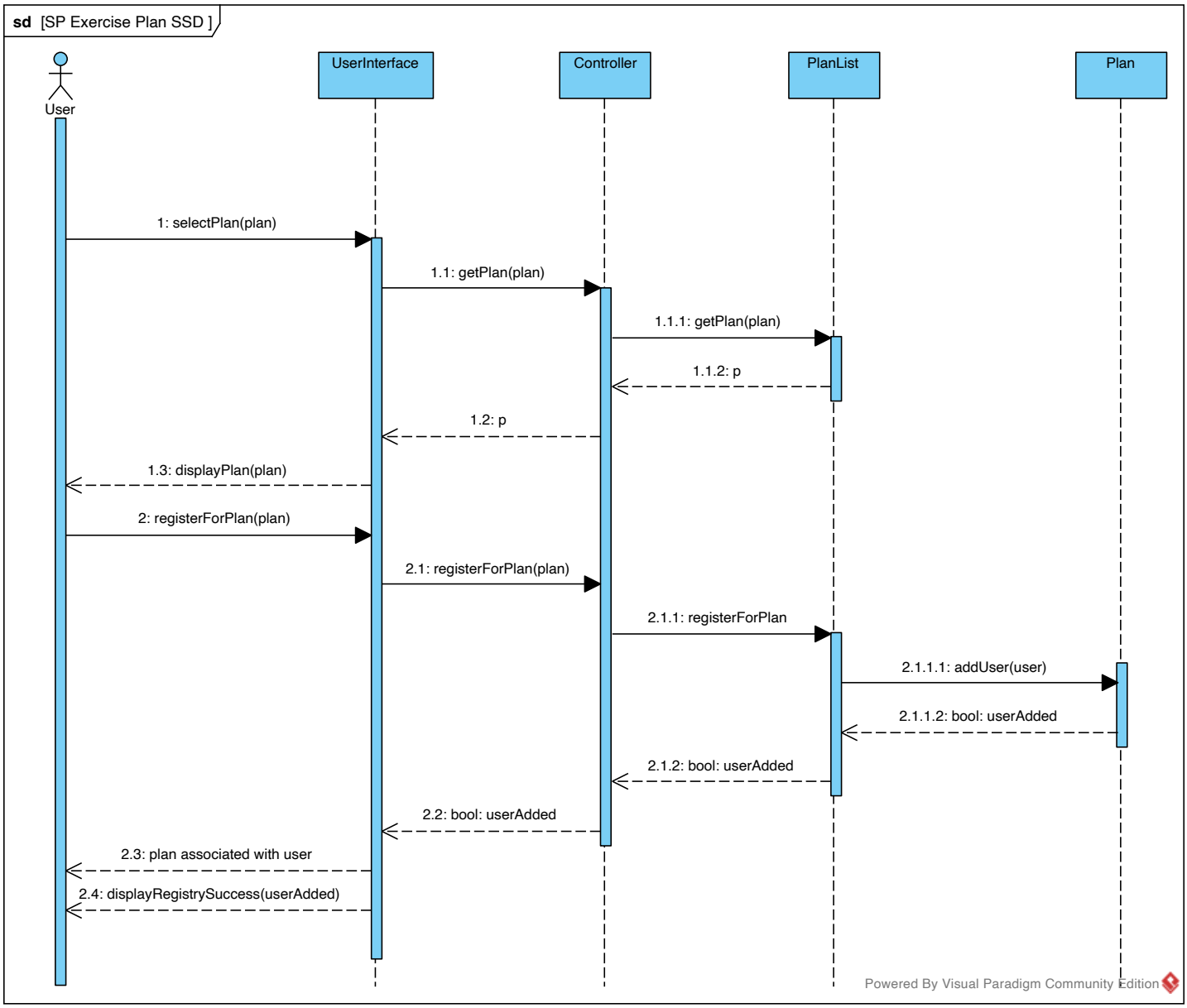

User
: UserInterface
Controller
workouts : WorkoutLog
1: createNewWorkout(workoutDate)
1.1: createNewWorkout(workoutDate)
1.1.1:
newWorkout : Workout
1.1.2: bool: workoutCreated
1.1.3: promptWorkoutInfo()
2: enterWorkoutDate()
3: enterWorkoutDescription()
4: enterWorkoutDuration()
5: enterWorkoutCaloriesBurned()
5.1: enterWorkoutInfo(date, description, duration, calories burned)
5.1.1: setDate(date)
5.1.2: setDescription(description)
5.1.3: setDuration(duration)
5.1.4: setCaloriesBurned(caloriesBurned)
5.1.5: bool: workoutAdded
5.1.6: workout logged successfully
5.1.6: addWorkout(newWorkout)
5.1.7: bool: workoutAdded
Powered By Visual Paradigm Community Edition

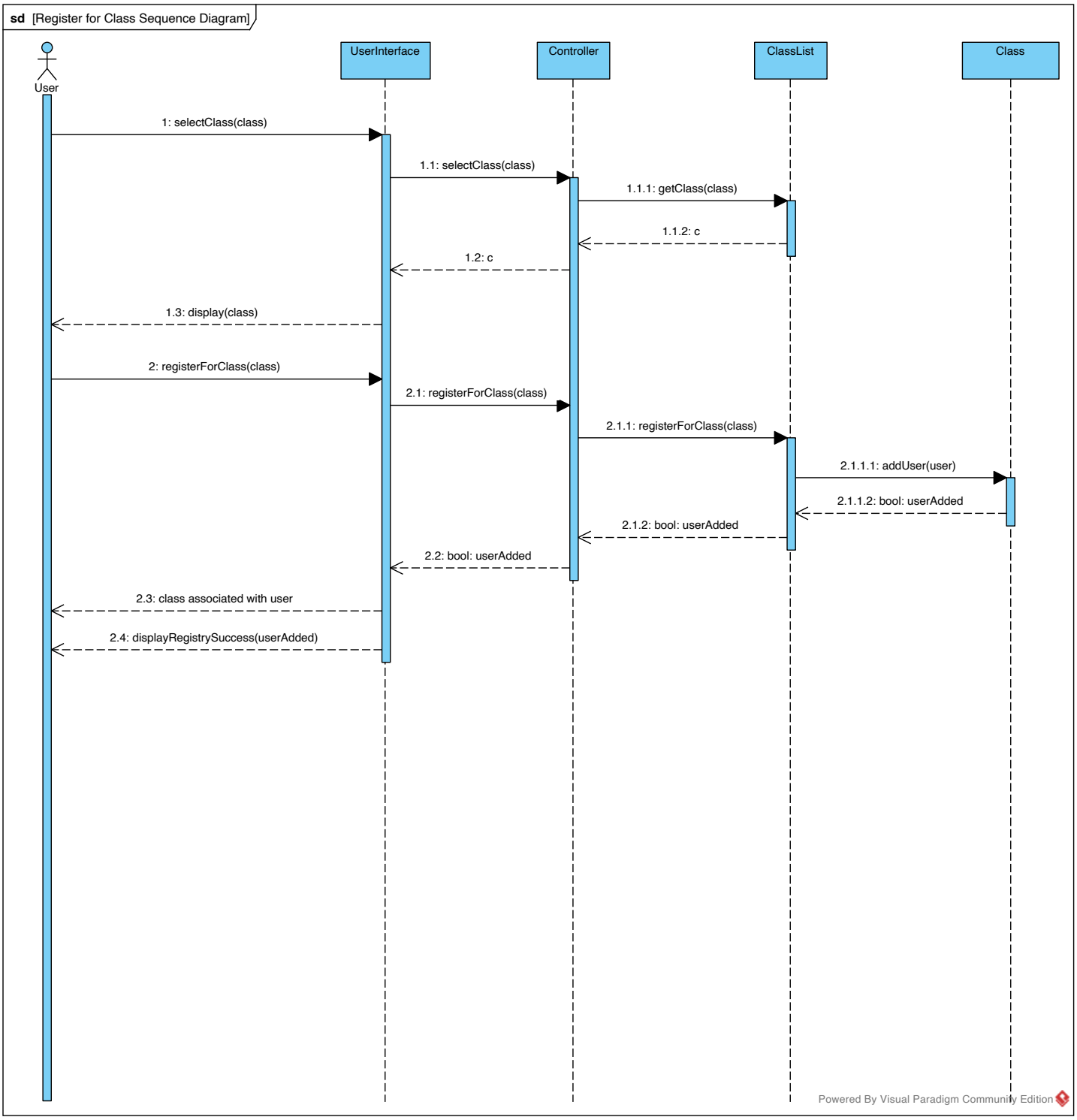

User
: UserInterface
controller : UserController
: User
: DailyMetric
1: click [profile]
1.1: getProfile()
1.1.1: getProfileInfo()
1.1.2: return profile info
1.2: return profile info
1.3: display profile menu with info
2: click [log info]
2.1: display logging page
3: enter information
3.1: display information in logging fields
4: click [submit]
4.1: confirmation message
5: click [yes]
5.1: log(water, sleep, calories, weight)
5.1.1: log(water, sleep, calories, weight)
5.1.1.1: << create >>
5.1.1.2: log(water, sleep, calories, weight)
5.1.1.3: return true
5.1.2: return true
5.2: return true
5.3: display successful log page
Powered By Visual Paradigm Community Edition

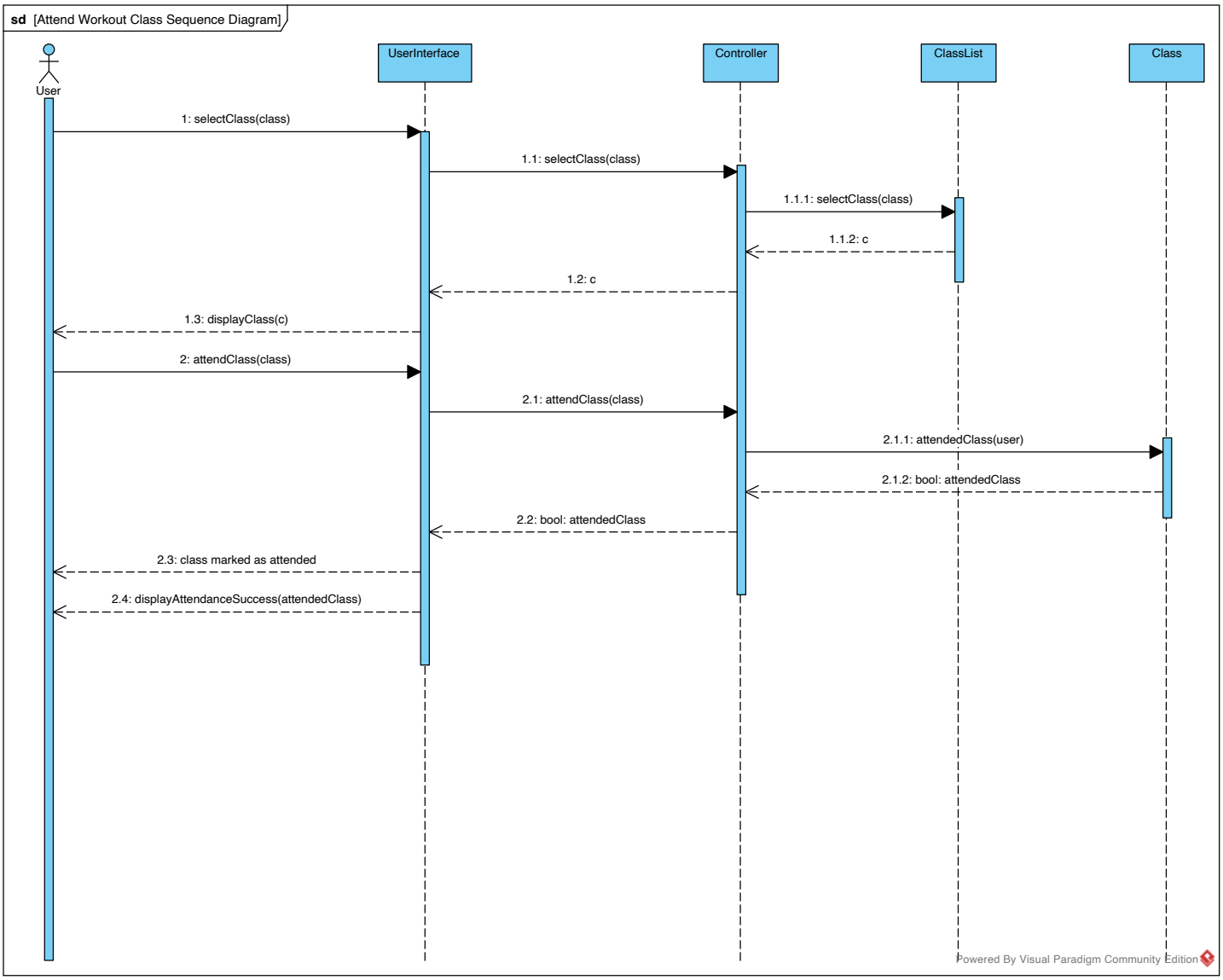

User
: UserInterface
: Controller
: Database
1: createAccount()
loop
[while email is not unique]
2: enterEmail()
2.1: saveEmail()
2.1.1: isUniqueEmail()
2.1.2: bool: isUnique
2.2: bool: isUnique
2.3: email in use- try again
2.4: createAccount(email)
2.4.1:
newAcct : UserAccount
2.4.2: StoreAccount(newAcct)
2.4.3: bool: AccountStorageSuccess
2.5: bool: AccountStorageSuccess
loop
[while password is insecure]
3: enterPassword()
3.1: savePassword
3.2: bool: isSecure
3.3: password insecure - try again
3.3: hashPassword()
3.4: savePassword(password)
3.4.1: savePassword()
3.4.2: bool: passwordSaved
3.5: bool: passwordSaved
loop
[while username is not unique]
4: enterUserName()
4.1: saveUserName()
4.1.1: isUniqueUserName()
4.1.2: bool: userNameUnique
4.2: bool: userNameUnique
4.3: username already exists - try again
4.3: saveUserName()
4.4: bool: userNameSaved
4.5: bool: accountSetUpComplete
4.5.1: account set up complete!
Powered By Visual Paradigm Community Edition


User
: UserInterface
: Controller
: Database
This is a hard coded list stored locally until this is actually implemented
user : User
1: forgotPassword()
1.1: displayForgotPassword()
loop
[email doesn't match an account]
2: resetEnterEmail()
2.1: resetEnterEmail(email)
2.1.1: doesEmailExist(email)
2.1.2: bool: emailExists
2.2: bool: emailExists
2.3: email doesn't match account - try again!
2.3: generates reset password link
2.4: bool: sentResetEmail
2.4.1: sent reset link to email!
3: clickResetLink()
3.1: clickResetLink(email)
3.1.1: activeResetLink(email)
3.1.2: bool: validResetLink
3.2: bool: validResetLink
3.3: please enter a new password!
loop
[while password is insecure]
4: enterPassword()
4.1: enterPassword(password)
4.2: bool: isSecure
4.3: password insecure - try again
4.3: hashPassword()
4.4: setPassword(hashedPassword)
4.5: bool: passwordResetSuccessful
4.6: sendPasswordResetNotif(email)
4.7: bool: passwordResetSuccessful
4.7.1: password successfully reset!
Powered By Visual Paradigm Community Edition


sd [UC-U.ACCT-03]
User
: UserInterface
: Controller
: Database
user : User
1: login()
1.1: login()
loop
[while information is not authienticated]
1.2: requestEmail()
1.3: requestEmail()
2: enterEmail()
2.1: enterEmail()
2.2: promptPassword()
2.3: promptPassword()
3: enterPassword()
3.1: enterPassword()
3.1.1: hashPassword()
3.1.2: authenticate(email, password)
3.1.3: bool: authenticated
3.1.4: login()
3.1.5: bool: loginSuccessful
3.1.6: bool: loginSuccessful
3.1.6.1: display home page
Powered ByVisual Paradigm Community Edition


sd [SP Exercise Plan SSD ]
User
UserInterface
Controller
PlanList
Plan
1: selectPlan(plan)
1.1: getPlan(plan)
1.1.1: getPlan(plan)
1.1.2: p
1.2: p
1.3: displayPlan(plan)
2: registerForPlan(plan)
2.1: registerForPlan(plan)
2.1.1: registerForPlan
2.1.1.1: addUser(user)
2.1.1.2: bool: userAdded
2.1.2: bool: userAdded
2.2: bool: userAdded
2.3: plan associated with user
2.4: displayRegistrySuccess(userAdded)
Powered By Visual Paradigm Community Edition


sd [Register for Class Sequence Diagram]
User
UserInterface
Controller
ClassList
Class
1: selectClass(class)
1.1: selectClass(class)
1.1.1: getClass(class)
1.1.2: c
1.2: c
1.3: display(class)
2: registerForClass(class)
2.1: registerForClass(class)
2.1.1: registerForClass(class)
2.1.1.1: addUser(user)
2.1.1.2: bool: userAdded
2.1.2: bool: userAdded
2.2: bool: userAdded
2.3: class associated with user
2.4: displayRegistrySuccess(userAdded)
Powered By Visual Paradigm Community Edition


sd [Attend Workout Class Sequence Diagram]
User
UserInterface
Controller
ClassList
Class
1: selectClass(class)
1.1: selectClass(class)
1.1.1: selectClass(class)
1.1.2: c
1.2: c
1.3: displayClass(c)
2: attendClass(class)
2.1: attendClass(class)
2.1.1: attendedClass(user)
2.1.2: bool: attendedClass
2.2: bool: attendedClass
2.3: class marked as attended
2.4: displayAttendanceSuccess(attendedClass)
Powered By Visual Paradigm Community Edition