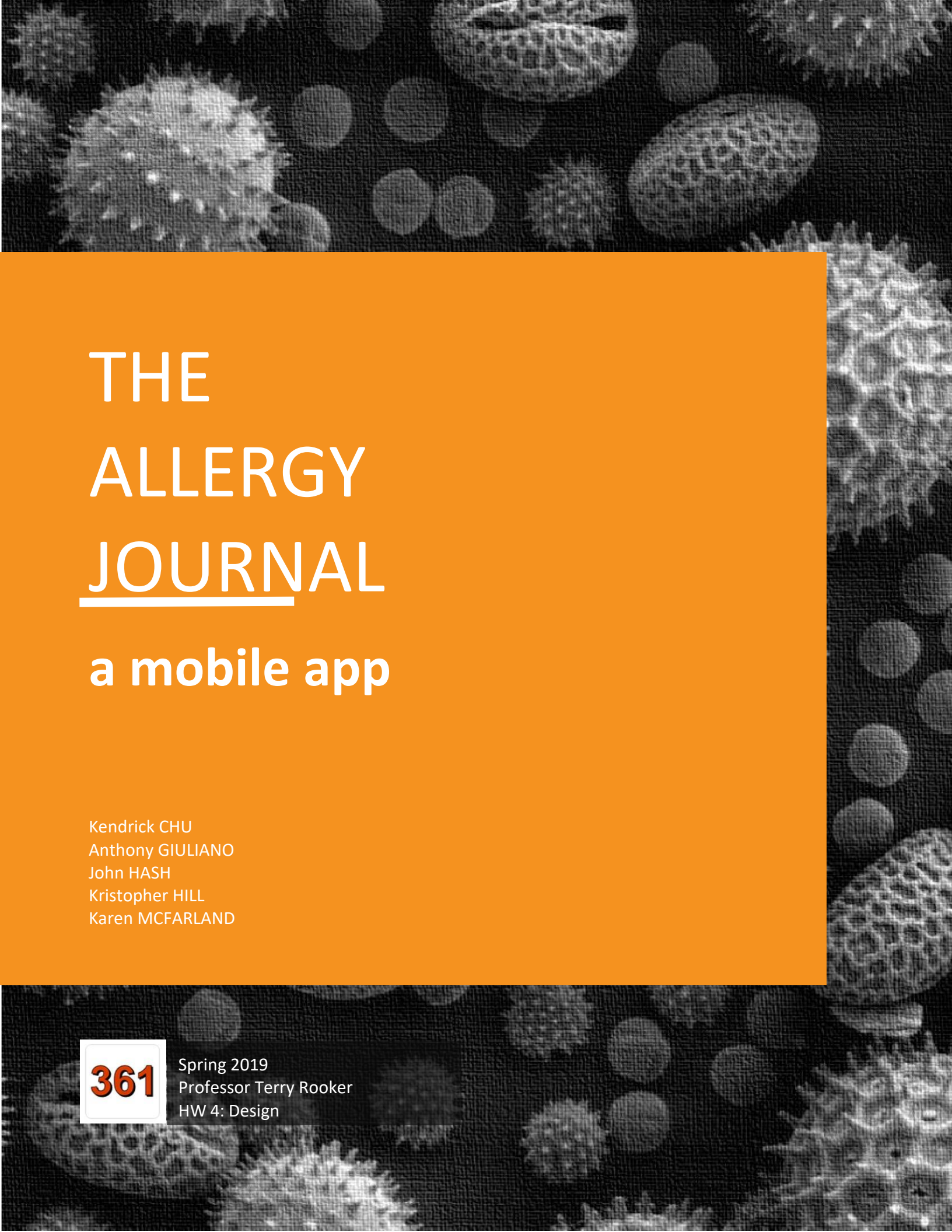

THE ALLERGY JOURNAL

a mobile app

Kendrick CHU
Anthony GIULIANO
John HASH
Kristopher HILL
Karen MCFARLAND

361

Spring 2019
Professor Terry Rooker
HW 4: Design

UML Class Diagram

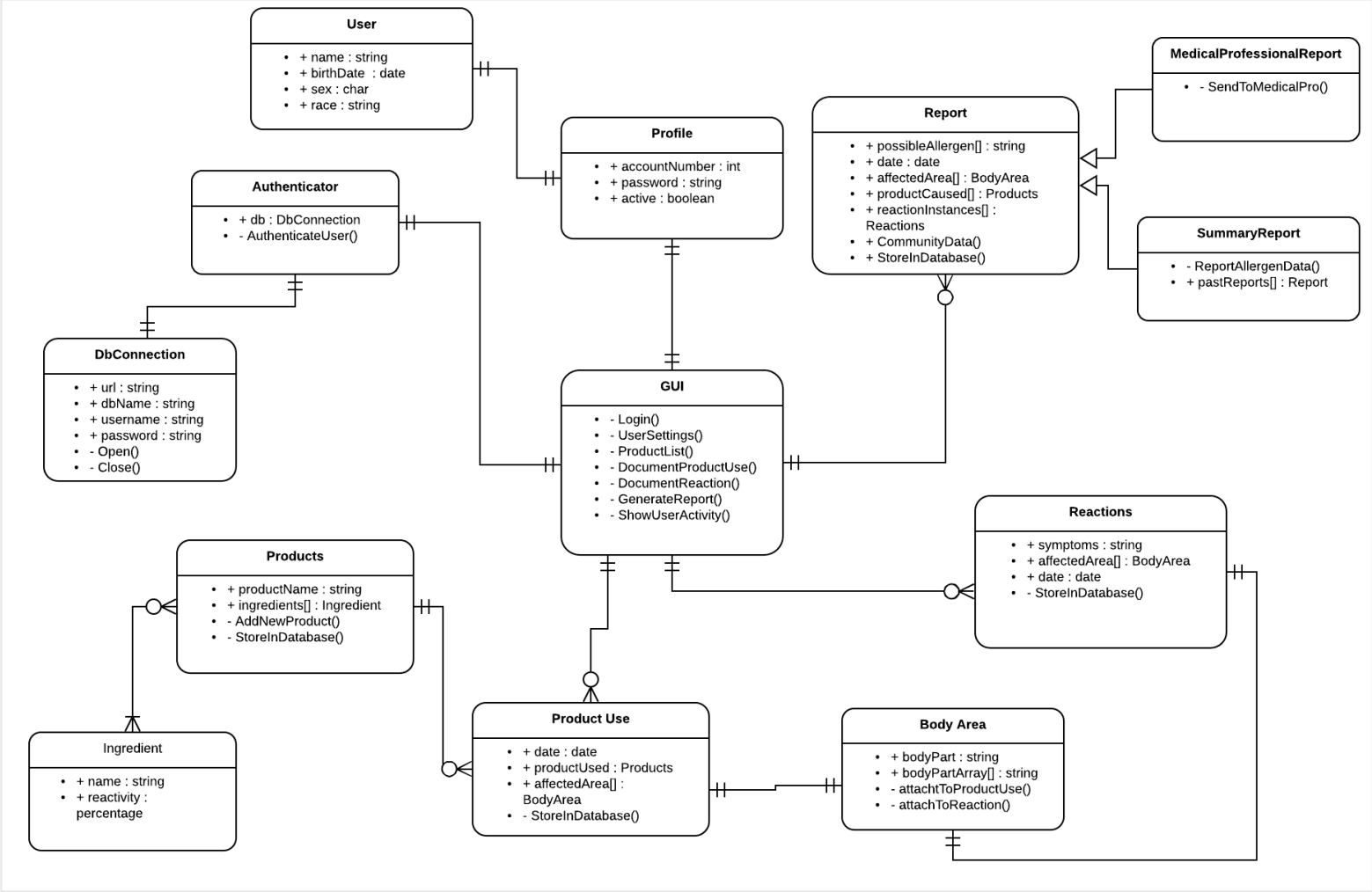

Figure 1. UML class diagram showing the key OO entities and their attributes

Coupling

Our design exhibits loose decoupling among the major modules that make up the system. To increase the amount of reusable code, make system changes easier and find faults, we are minimizing common and content coupling where possible.

Data Coupling

- There is no data coupling in our design.

Stamp Coupling

- *Body Area* supplies structured data to *Reactions*.
- *Body Area* supplies structured data to *Product Use*.
- *Ingredients* supplies structured data to *Products*.
- *Products* supplies structured data to *Reports*.
- *Reactions* supplies structured data to *Reports*.

Control Coupling

- *Report* handler will control the type of report that is generated and returned.

Common Coupling

- Both *Reactions* and *Product Use* could have access to the same *Body Area* data, and some protocols might have to be implemented where either *Reactions* or *Products* have sole control over the data at any one time.

Content Coupling

- *Authenticator* will be able to modify the *Profile*.

Cohesion

Other than the modules for generating reports and predicting allergens, the modules in our system are fairly cohesive in that each is essentially used for a single purpose and contains elements that are used for a single function.

Functional Cohesion

- *GUI* gives the user an interface to interact with the system.
- *Authenticator* provides security to the system.
- *Products* handle and store data solely used for *Product Use*.
- *Reactions* handle and store data solely for *Reports*.

Communicational Cohesion

- *Reactions* and *Product Use* use the same data from *Body Area*.

Procedural Cohesion

- Generating a Report has data pulled from *Reactions*, and then this data is combined with documented *Product Use* to calculate suggestions on allergens.

Design Assessment

Realistically, a mix of iterative and incremental system development could be employed for this system. Initially primitive versions of the primary system modules (e.g. Documenting Reactions, Documenting Product Use, and Reports) could be released. Subsequent to the initial release, updates to those modules, such as improved product lookup (via an iterative approach) could take place at the same time new modules (via an incremental approach) are released. However, the preferred overall development strategy for our system is iterative because the user needs a relatively complete and functioning system before value can be extracted.

Our requirements for the allergy application are clearly circumscribed in a way that the likelihood of adding new modules is low. This means that improvement to the system will happen primarily in an iterative fashion. For example, *Reports* functionality will by definition improve iteratively because the increasing amount of data from the user and from the community means that the set of questions that can be answered will increase. This will force changes to be made to the algorithms that process these data to make predictions about allergens.

Another example of iterative development that will inevitably take place is the *Add Product* module. Improved camera resolution, more reliable internet connections, and increased size of the products database will mean that the module can be updated over time to leverage these improvements.

The iterative process lends itself to code reusability because new modules are not being continually added or replaced. On each iteration, we are making small improvements to existing code. As an aside, the Waterfall process in general may lead to less code reusability than an Agile process because of the top down approach that can lock some aspects of the software into place early on in development.

Design Patterns

Façade

Because our application is intended for users to have a simple way to interact with the system, the façade pattern would be useful. The user does not need to interact with various classes and their functions and member data when trying to Document Product Use, for instance. The user needs to only tap the submit button after the appropriate information has been entered in the fields. Our GUI will remove the user from the details of the underlying system.

Memento

Users will be adding data to forms that are interacting with multiple classes. Due to the high volume of data and the uncertainty that users will sometimes have while entering data, they will need to be able to go back to previous states to change their inputs through functionality such as undo. The memento pattern will allow objects to be restored to their previous states, if needed.

Builder

Report objects will be relatively complex. We have two representations of them between the type displayed to the user and the type returned to medical professionals.

Observer

It may be suitable to make reports observable, so that users or medical professionals may register to be notified when a report is newly available or a potential allergen is identified based on community data.

Strategy

Strategy pattern may be useful in terms of identifying potential allergens. The algorithm we use to match reactions and ingredients will vary based on information only available at runtime. For example, evaluation of community data will change and improve over time and affect the evaluation of a users' reaction and product-use instances.

Use Case Sequence Diagram: Documenting an Allergic Reaction

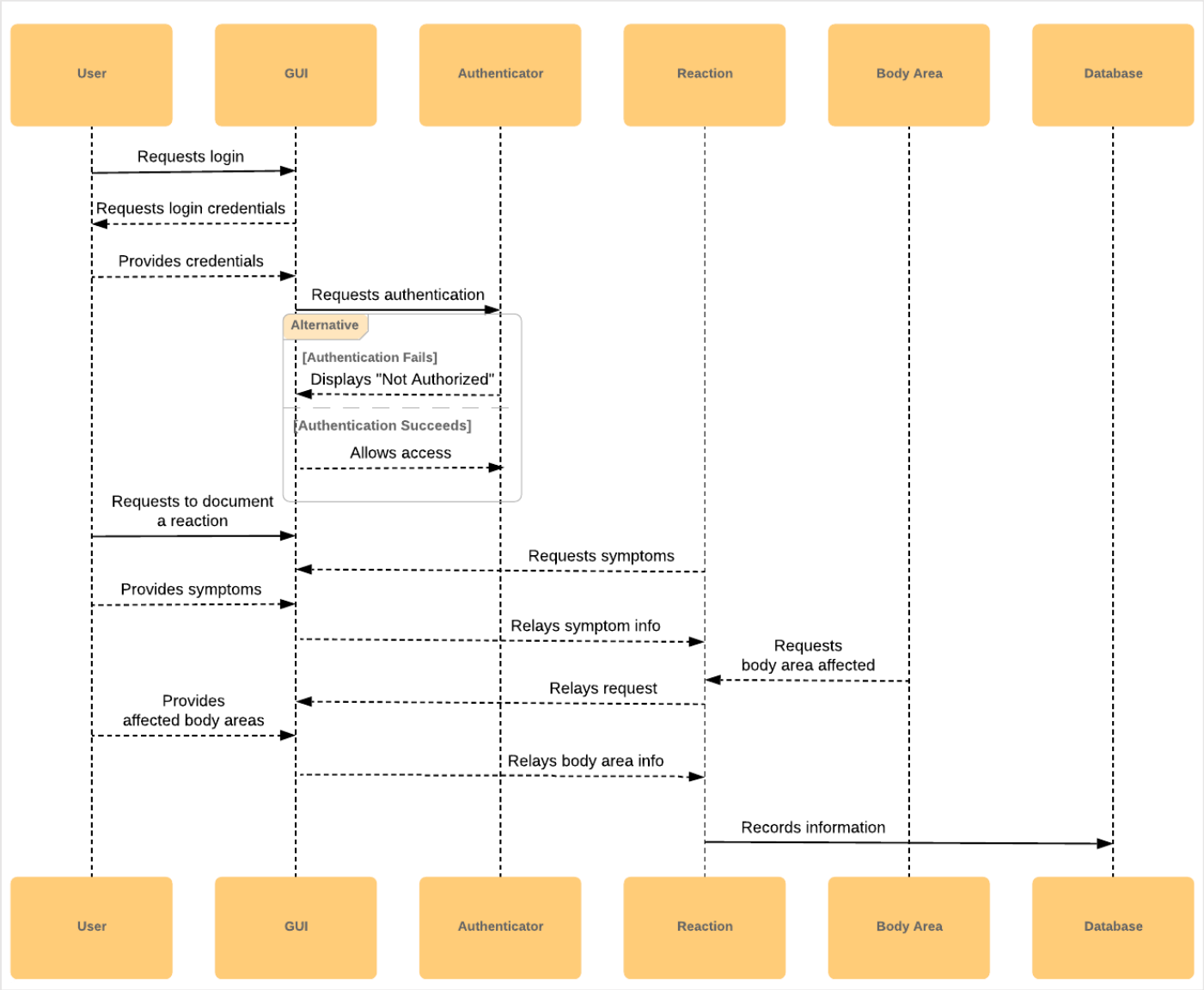

Figure 2. Use Case Sequence Diagram: Documenting an Allergic Reaction

Interfaces

Product Search API

Our application would benefit from an interface for performing product searches. For now, we're only worrying about integrating with one external search API, but that could easily change or grow going forward. Creating a consistent interface for performing these searches will make it easy to add to or change the search API that our app is using just by implementing `search` in different ways but still always returning an instance of `Product`.

Database

Our database connection class will provide an interface between the database and any classes that need to access it. With this interface we will be able to easily switch to a different underlying database or add support for additional databases just by creating new classes that implement this interface. This interface will force implementations of `openConnection`, `closeConnection` and `initiateTransaction`.

Doctor API

Another access point for our system will be the API that provides reaction data to medical professionals. It will accept as input a string containing identifying information about the user and will return a JSON representation of `Report`.

GUI

The GUI is really the main interface used both by the user and by many of the components in the system; all of the application's actual functionality is exposed through it, and it is responsible for displaying output. Requests to log in, add products, record product usage and reactions will all be received as messages by the GUI and relayed to the appropriate module server-side.

Exceptions

Exceptions and error handling is divided into two main classes: 1) user-initiated errors and 2) system-initiated errors. In each case, passive fault detection will be the primary way errors are caught. However, active fault detection methods may be needed for periodically checking the database to ensure data integrity and that personally identifiable data are secure. Some examples are provided below.

User-Initiated Errors

- Spell-checking
    - Much of the input functionality in the user interface will rely on pre-populated drop-down menus and other standardized choices. However, some fields will require manual input, such as the feature for manually looking up and adding a product. In these cases, the system will first try to match the user's input verbatim. If it fails, the system will send back similarly spelled choices to the user for confirmation.

- Other Invalid Input
    - Some values cannot be null. For example, in the body part class, the user will not be allowed to submit a product use instance or document a reaction if all body part variables are null.

  - Data types must be enforced. Before any data can be added to the database under the user's profile, all user data will be validated. If incorrect data types are found, the user will be prompted to re-enter their data.

- Smart Phone Settings
  - Users will either need to be able to connect to Wi-Fi or use cellular data for documenting new reactions, documenting new instances of product use, or generating new reports. If no connection is detected at login, the user will be prompted to allow cellular data or connect to WIFI until the app connection is established or the app is closed.
  - Quick product lookup via the barcode scanner will require access to the user's phone. The user will be prompted to either allow camera access or input product data manually.

System-Initiated Errors

- Database connection: In the case of a database connection error, the system will make several attempts to make a connection, and if the connection error persists, then the handler will alert the user that there was no connection and identify the type of connection error that occurred.

- Service not available: Similar to the database connection error, if any other part of the service is unavailable, the system will make several attempts to connect to the service. If that doesn't succeed, then the handler will alert the user that the particular service they are trying to use is unavailable.

- Sending data to doctor: Again, like database connection errors and service not available errors, when the system cannot send data to the doctor, it will attempt several times before alerting the user that the data could not be sent to the doctor and notify them that they should try again later.

- Trying to find product (product doesn't exist): If the user attempts to find a product that, in fact, does not exist in the database, the error handler will alert the user to this fact and then ask the user if they would like to put in a request to the administrators to add the product. If the user does indeed want to put in a request to the admin to add said product, the user will be sent to the interface for requesting an admin to add a product.

Contributions of team members

	Kendrick CHU	Anthony GIULIANO	John HASH	Kristopher HILL	Karen MCFARLAND
UML class diagram	X			X	X
Coupling and Cohesion Discussion	X		X		
Incremental vs Iterative Discussion			X		
Design Patterns Explanation	X	X	X		
USE CASE Sequence Diagram					X
Identify interfaces and contracts	X	X			
Exception handlers	X		X	X	
Stitch together document					X
Review final draft	X	X	X	X	X
Submit document					X