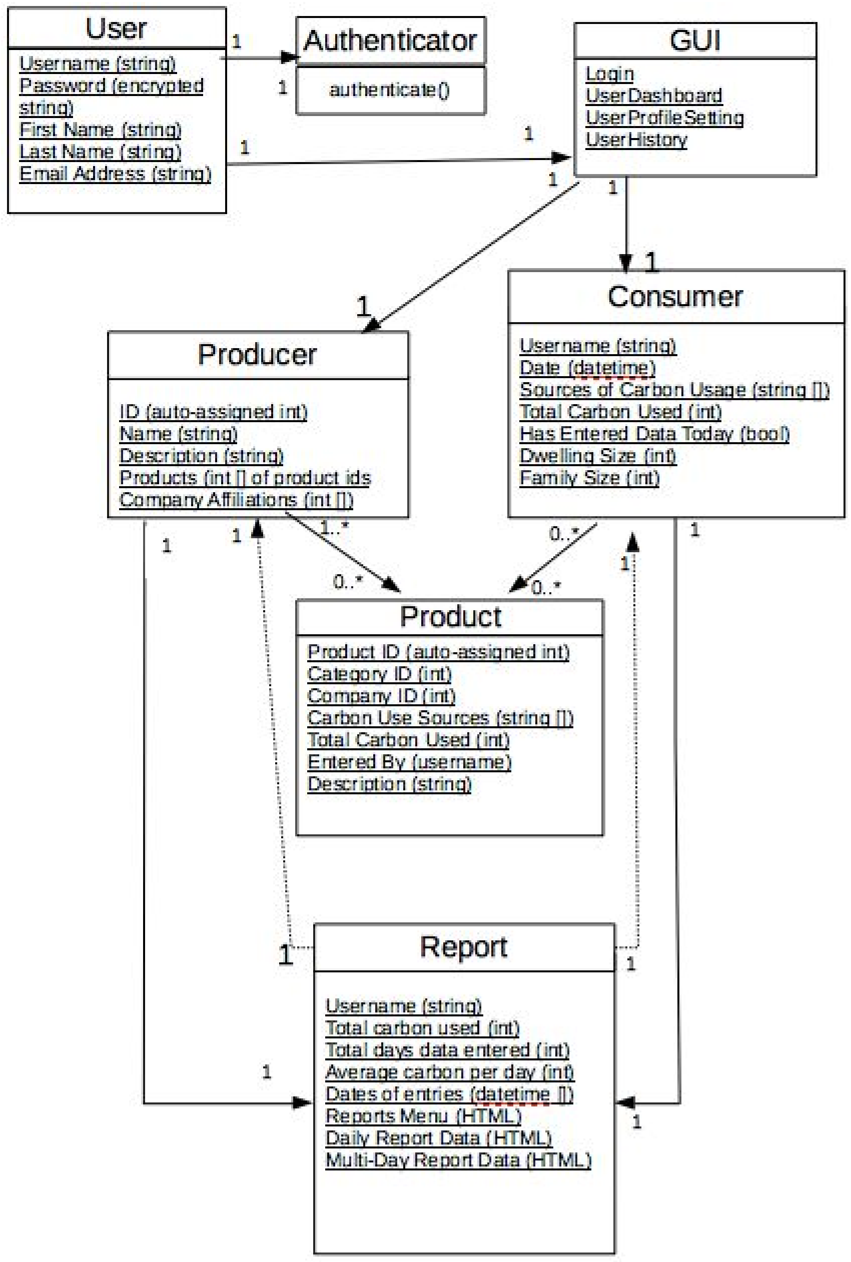
Carbon Footprint Web Portal / Application - Implementation Design - HW4

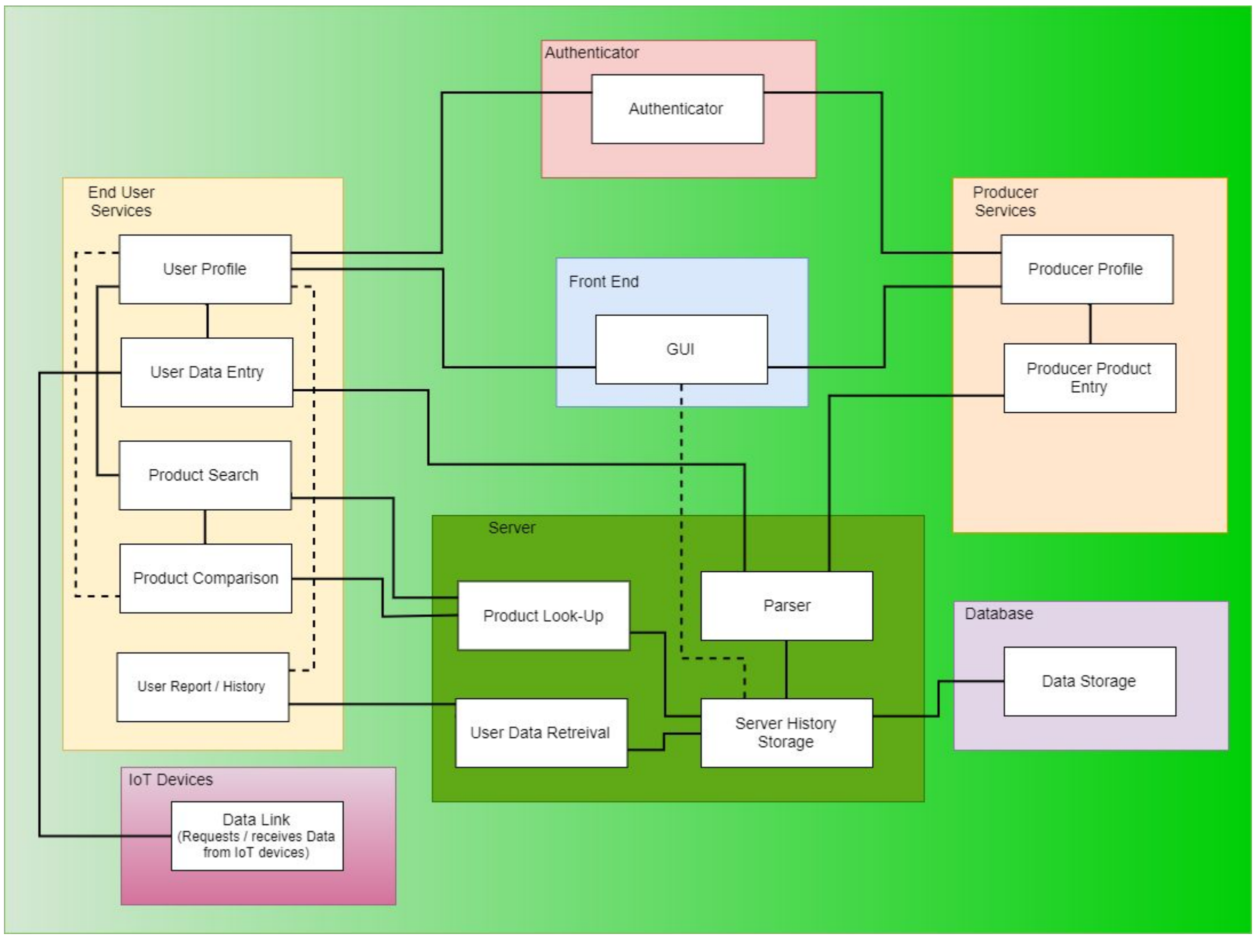
CS 361 Group 2 Project
11/10/19

- Alex Densmore
- Anousha Farshid
- Adam Wright
- Ken Wyckoff

UML Class Diagram



Packaging the Implementation Diagram



Coupling

Content Coupling

- ❖ Authenticator modifies User Profile access.
- ❖ Authenticator modifies Producer Profile access.

Common Coupling

- ❖ Product Search and Product Comparison both call Product Look-Up
- ❖ Product Look-Up and User Data Retrieval both call Server History Storage

Control Coupling

- Product Search can call Product Comparison
- User Report / History calls User Data Retrieval
- Product Search calls Product Look-Up
- Product Comparison calls Product Look-Up
- User Profile calls User Data Entry
- User Profile calls Product Search
- User Profile calls Product Comparison
- User Profile calls User Report / History
- Producer Profile calls Producer Product Entry
- User Data Entry calls Data Link

Stamp Coupling

- User Profile provides structured data to GUI
- Producer Profile provides structured data to GUI
- Parser provides structured data to Server History Storage
- Server History Storage provides structured data to Data Storage
- Server History Storage provides structured data to GUI

Data Coupling

- User Data Entry provides unstructured data to Parser
- Producer Product Entry provides unstructured data to Parser

Cohesion

Functional/Informational Cohesion

- Authenticator provides security to the system.
- GUI provides a system interface to users.
- End User Services provides consumer-type functionality to primary consumers.
- Producer Services provides minimal back-end functionality to helpful producers.
- IoT Devices provides End User data to alleviate mundane data entry tasks.
- Database records, updates, and transmits data for the system to use.

Communicational Cohesion

- Server manipulates data for system usage. Server parses the received data to be stored, sends parsed data to Database, and transmits requested data back to End User or Producer.

Iterative Versus Incremental Design Analysis

Our design will best support iterative development based on the requirements. There is no one component of our system that we consider the keymost component which would be of much value in and of itself. The consumer and producer client-side interfaces and back-end servers as well as the database will all be necessary for the system to be of value. The consumer client-side interface is needed so that individual consumers are able to enter data about their carbon usage (either confirming data retrieved from IoT devices or entering data manually) and so that consumers can view data about their carbon usage. The consumer back-end server is needed to process data that consumers input or request. The database is then needed for long-term storage of that data. Similarly, the producer client-side interface is needed so that producers can input data about carbon used in the production of their products (either by uploading a preformatted document or entering it manually) and so they can view reports of data previously entered. The producer back-end server is then needed to facilitate communications between the producer client-side interface and the database. Both the consumer and producer interfaces and servers are needed so that both consumers and producers get maximum value out of the software. Without the ability for producers to enter data about their products, consumers will not be able to easily track carbon used by individual products they use. Similarly, without the consumer interface, producers will not have as much reason to enter information about carbon used by their products since the whole reason for them entering it is to allow consumers to track their own carbon usage.

Since we have identified that an iterative approach is best for developing our system where we would get all components working fairly well, we would then go back through and update components as needed. For example, as new IoT devices are developed which could provide relevant data, we could incorporate those into the auto-import feature of the consumer client-side interface. Similarly, we could update data validation as the nature of products changes. For instance, perhaps someone driving a car for 2 hours and traveling 500 miles within that time would be flagged as invalid data by the system when it is first implemented. However, if, in the future, cars are developed that can travel much faster, the upper bound of how many miles per hour would be considered possible could be increased.

Design Patterns Analysis

The following design patterns presented in Lectures 5.2-5.3 would be helpful when implementing our system:

Builder: The builder design pattern would be helpful since objects containing data about consumer daily entries and products entered by producers would have many attributes. This design pattern allows for the construction of complex objects where much data needs to be processed to instantiate them, so it would be very helpful when creating such objects.

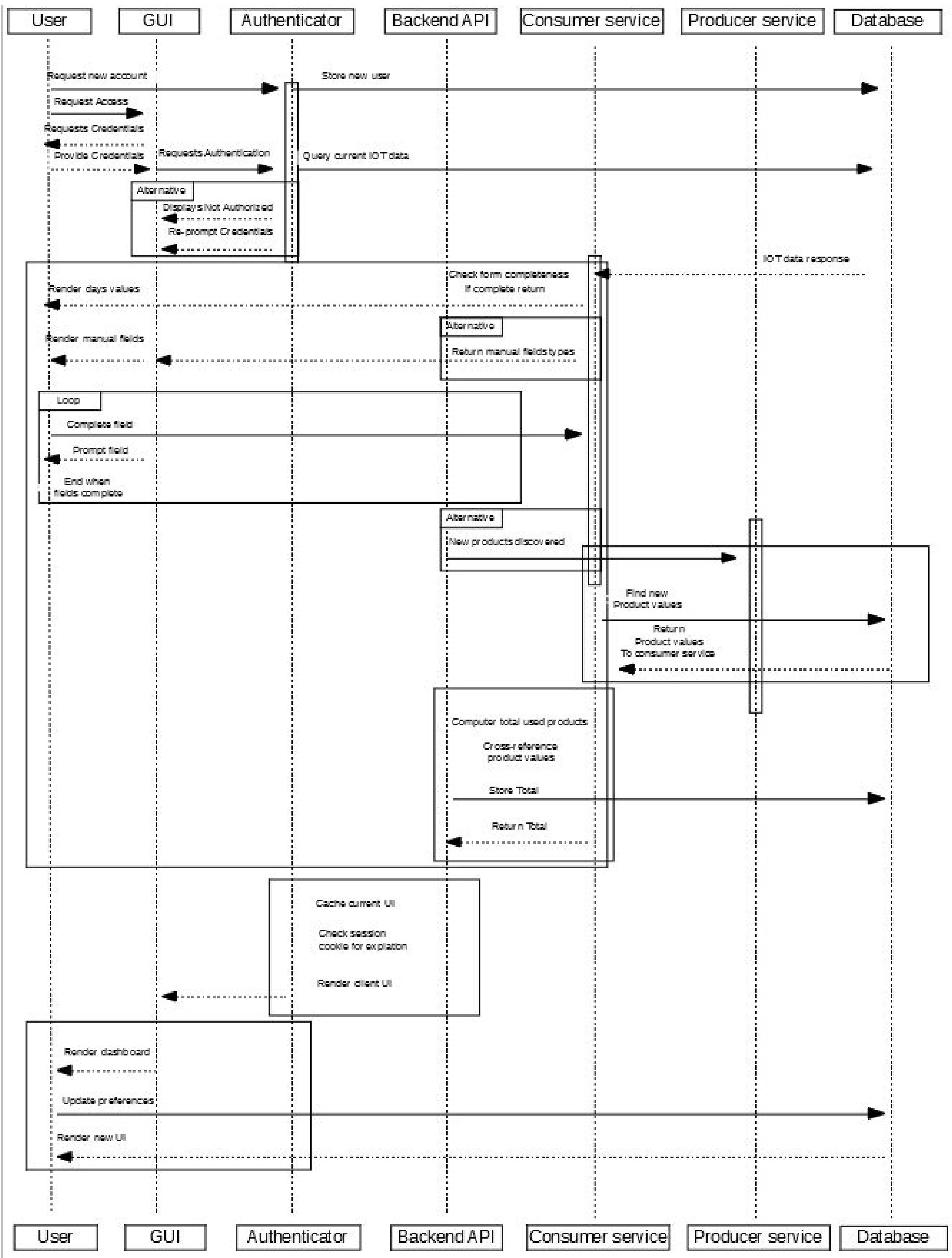
Adapter: The adapter design pattern would be helpful since the various IoT devices which will provide data to the system might not all be compatible with one another and might not all provide data in the same format. This design pattern would allow for creating a wrapper that overcomes incompatibilities by converting data and loading it into consumer daily entry objects in a consistent manner.

Facade: The facade design pattern would be helpful since system calls to pull and parse information from IoT devices may be very complicated. The facade design pattern would nicely implement these complex calls within a more unified, easy-to-use, high-level interface, allowing software developers to more seamlessly work with such system calls.

Template Method: The template method design pattern could be useful for performing various tasks that have shared steps. For example, although consumers and producers are two different types of users, there are some attributes that all types of users have in common. Therefore, the template method could be used so that child classes can override anything they need to from parent classes.

Composite: The composite design pattern would allow for the grouping of related objects together as a single object. For example, the various types of report objects would all be included within a general “Reports” object, which would reduce complexity and allow code reuse when dealing with specific types of reports.

Visitor: The visitor design pattern would be useful when data gathered by IoT devices should be kept separate from the algorithms processing it. IoT devices will likely collect more data than our system will read in from those IoT devices. For example, perhaps there will be certain personally identifiable information registered in IoT devices that is not kept by our system and, therefore, has no need to be accessed by our system. The visitor design pattern would allow our system to process the data it needs which the visiting IoT device allows it to access without exposing unnecessary data in the process.


User
GUI
Authenticator
Backend API
Consumer service
Producer service
Database
Request new account
Store new user
Request Access
Requests Credentials
Provide Credentials
Requests Authentication
Query current IOT data
Alternative
Displays Not Authorized
Re-prompt Credentials
IOT data response
Check form completeness
If complete return
Render days values
Alternative
Render manual fields
Return manual fields types
Loop
Complete field
Prompt field
End when
fields complete
Alternative
New products discovered
Find new
Product values
Return
Product values
To consumer service
Computer total used products
Cross-reference
product values
Store Total
Return Total
Cache current UI
Check session
cookie for expiation
Render client UI
Render dashboard
Update preferences
Render new UI
User
GUI
Authenticator
Backend API
Consumer service
Producer service
Database

Interface Contracts

Below are descriptions of some key interfaces and their contracts, including what input is needed (preconditions) to produce the desired output (postconditions).

Consumer Daily Entry Input Interface: Input: Data about various sources of carbon used (whether entered by IoT devices or by the consumer directly); consumer's confirmation that all data entered is correct. Output: A confirmation of the data entered which is then archived for later viewing in a Daily Report View and which is used in calculating aggregate values for reports covering periods of time.

Producer Product Input Interface: Input: Various pieces of data about carbon used in the creation and, when applicable, by the use of a product (data is either uploaded in a preformatted document or manually entered by the consumer); producer's confirmation that all data is entered correctly. Output: A confirmation of the data entered which is then archived for use when any user looks up a certain product's statistics or reports their use of the product.

Report View Interface: Input: the type of report desired (such as report of a consumer's carbon usage or report of how many consumers registered with the site have reported use of a certain product) and the date range of the desired report when applicable (such as a single-day or multi-day consumer carbon usage report). Output: A report containing the desired data. If the search returned no results (such as a user searching for a carbon usage report for a day on which they did not report any data), an informative message is displayed accordingly.

Single Product Search Interface: Input: The name of the desired product. Output: The data entered about that product by its producer. If more than one product matched the search criteria, all matching results are displayed. If no product matched the search criteria, an informative message is displayed to inform the user.

Product Comparison Interface: Input: The search terms (such as a category of product, the names of multiple specific products to compare, the name of a single company's products to compare, or the names of multiple companies' products to compare). Output: A report showing at least 5 products matching the search criteria (or as many such products as exist if less than 5 exist in the system), allowing the user to see each product's individual carbon data and compare the carbon used by all the products graphically. If no products match the search, an informative message is displayed accordingly.

Exception Descriptions

Below are descriptions of some exceptions that are likely to occur and the exception handlers that would process them.

IoT Device Connection Exception: This exception happens when the system is unable to connect to one or more IoT devices the user has registered for collecting their carbon usage data. After the software tries multiple times to attempt to connect to a given device when the user has selected the option to auto-import data, the user will be notified and advised to manually enter the data if it is available to them.

Product Data Autoimport Exception: This exception happens when a producer attempts to upload product data in a preformatted document but the system is unable to parse the data. The producer will be informed that the error has occurred and be reminded of the accepted formats of preformatted documents. They will then be given the option to try uploading the document again or enter data manually. This way, if they are able to make a simple correction to the document on their end and then reupload it, they have the opportunity to still autoimport their data in this manner even if the first attempt failed.

Data Validation Exception: This occurs when a user enters data that is invalid according to the expected format or numeric ranges. These will be handled on the client-side in all cases so that invalid data is not stored in the database. For example, the client-side HTML manual entry forms will be set up in such a way that they only allow the format of data expected to be entered. For instance, all numeric data will be collected via form fields that only accept numbers. In addition, data the user has entered will be validated, and they will be asked to fix invalid data and then resubmit the form before their form is processed. For example, if a user enters a completely impossible seeming number of miles they have driven in a day given the current car technology, they will be told what data is invalid and how to fix it. A similar process will be used for any missing data that is needed in order to process a report.

Server or Database Connection Exception: This occurs when some components of the system are online and functioning properly (such as the client-side interfaces), but one or more other systems are down (such as one of the back-end servers or the database). This exception type will be handled in a manner that allows the users to still do as much as possible with the site. For example, if the producer back-end server is down but the consumer back-end server is up and the client-side interfaces and database are all functioning, consumers will still be able to enter and view their own carbon usage data since all of the tools involved in their tasks will still be working. On the other hand, producers who attempt to upload data will be advised that the server is currently down and to please try entering product data again later.

Team Member Contributions

The following list indicates team member contributions during the preparation of HW4:

- All - Direct team meetings and continued communication via Google Hangouts and email.
- Alex Densmore - Wrote incremental versus iterative design analysis, design patterns analysis, interface contracts, and exception / exception handler descriptions.
- Anousha Farshid - UML Class Diagram
- Adam Wright - Message flow diagram and blog post
- Ken Wyckoff - Drew Packaging Implementation diagram and wrote Coupling / Cohesion assessment.

Each team member was also involved in creating this document from the various parts completed individually, along with reviewing said documents for accuracy and assurance.