CSCD01 Deliverable 3

Team Number: 11
Team Name: Phantom Developers
Date: March 20th, 2020

Table of Contents

Feature List

Feature - Support formulary status for drugs

Description

This feature adds a “formulary” status to drugs, which are used to put drugs into a list. Each drug can be assigned to more than one formulary. Queries and management of formulary should also be supported. Formularies are lists of approved medication for prescription. They are used by hospitals and insurance companies to determine the eligibility of approving a claim, being prescribed, etc.

Code Modification

Add a new class “formulary” which will store a list of drugs. To support queries for the user, creating a new service and its respective DAO for formulary would be needed. We also need to add new tables in the database to support formularies.

Implementation Plan

In liquibase-schema-only.xml:

- Create a new table in the database called drug_formulary with columns (drug_formulary_id PRIMARY KEY, name VARCHAR, description VARCHAR)
- Create a new table in the database called drug_formulary_map with columns (PRIMARY KEY drug_formulary_map_id, FOREIGN KEY drug_formulary_id, FOREIGN KEY drug_id)

Create class FormularyDAO to query the DB.

- getAllFormularies(int limit, int page): List<Formulary>
    - Select all the formularies with limit (-1 for no limit) and pagination
- getDrugsFromFormulary(Integer formulary_id): List<Drug>
    - Returns all drugs in this formulary
- isDrugInFormulary(Integer drug_id, Integer formulary_id): Boolean
    - Check if given drug is in a particular formulary
- isDrugInFormularies(Integer drug_id, List<Integer> formulary_ids): Boolean
    - Check if given drug is in a list of specified formularies
- getFormulariesFromDrug(Integer drug_id): List<Formulary>
    - Get all formularies with this drug
- getFormulary(Integer formulary_id): Formulary
    - Select a formulary given formulary_id

- insertDrugToFormulary(Integer drugid, Integer formulary_id)
    - Insert a drug to a formulary
- removeDrugFromFormulary(Integer formulary_id, Integer drug_id): void
    - Remove a drug from a formulary
- deleteFormulary(Integer formulary_id): void
    - Deletes the formulary
- createFormulary(Formulary formulary): Integer
    - Creates the formulary
    - Returns the id of the formulary
- editFormulary(Formulary formulary): void
    - Updates the formulary

Create class FormularyService in the API

- Methods:
    - getAllFormularies(): List<Formulary>
    - getAllFormularies(int limit, int page): List<Formulary>
    - getDrugsFromFormulary(Integer formulary_id): List<Drug>
    - isDrugInFormulary(Integer drug_id, Integer formulary_id): Boolean
    - isDrugInFormularies(Integer drug_id, List<Integer> formulary_ids): Boolean
    - getFormulariesFromDrug(Integer drug_id): List<Formulary>
    - getFormulary(Integer formulary_id): Formulary
    - insertDrugToFormulary(Integer formulary_id, Integer drug_id): void
    - removeDrugFromFormulary(Integer formulary_id, Integer drug_id): void
    - deleteFormulary(Integer formulary_id): void
    - createFormulary(Formulary formulary): Integer
    - editFormulary(Formulary formulary): void

Create new Formulary Data type

- Attributes (Required): drugFormularyId, name, description
- Attributes (Optional): clinicalInformation, doses, interactions
- Methods:
    - getDrugFormularyId(): Integer id
    - getName(): String name
    - getDescription(): String
    - getClinicalInformation(): String
    - getDoses(): Integer doses
    - getInteractions(): String
    - setName(String name): void
    - setDescription(String description): void
    - setClinicalInformation(String clinicalInformation): void
    - setDoses(Double doses): void

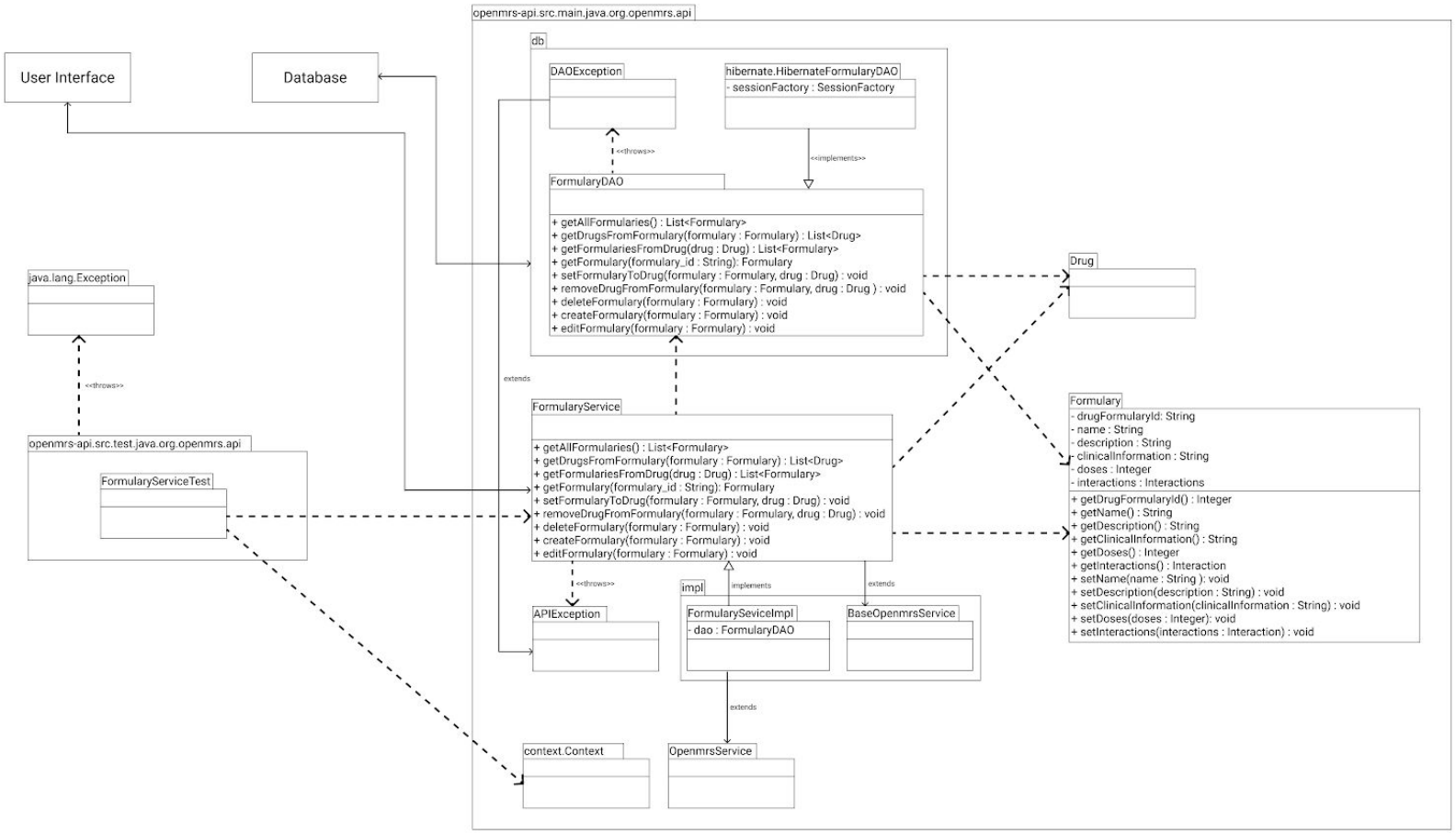
    - setInteractions(String interactions): void

Create a new test class FormularyServiceTest

- getAllFormularies:
    - should return a list of formularies
    - should return an empty list if none exist
- getDrugsFromFormulary:
    - should return a list of Drug objects given a formulary id
    - should return an empty list if given a formulary that has no drugs
    - should throw an exception if given a formulary that does not exist
- isDrugInFormulary:
    - Should return true iff drug is in the formulary
    - should throw an exception if given a formulary or drug that does not exist
- isDrugInFormularies
    - Should return true iff drug is in the formularies
    - should throw an exception if given a formulary or drug that does not exist
- getFormulariesFromDrug: List<Formulary>
    - Should return the formularies that have the drug
    - should throw an exception if given a drug that does not exist
- getFormulariesFromDrug:
    - should return the list of formulary data types given a drug id
    - should return an empty list for a drug that doesn't belong in any formulary
    - should throw an exception if the formulary does not exist
    - should throw an exception if the drug does not exist
- getFormulary:
    - should return a Formulary object when given an id for a Formulary
    - should throw an exception if a formulary does not exist
- setFormularyToDrug:
    - should add the drug to the formulary
    - should throw an error if the drug or formulary doesn't exist
- removeDrugFromFormulary:
    - should delete a drug from a formulary when given drug and formulary id
    - should throw exception when drug or formulary does not exist
- deleteFormulary:
    - should delete the given formulary when given a formulary id
    - should throw exception when formulary does not exist
- createFormulary:
    - should add formulary to the database
    - should throw exception when there is a duplicate name
- editFormulary:
    - should be able to change the description, name, etc. of a Formulary
    - should throw exception if there is another formulary with the same name

- should be able to update the database after making changes to the formulary

UML Diagram



Feature - Add module to change a clinical (or any) relationship for multiple patients at a time

Description

Currently, there is no way to change the relationships for multiple patients at the same time. For example, if multiple patients are assigned to the same physician or nurse, and that physician leaves and another takes over their cases, you have to either make this change in the database or one by one in the UI. We need to create an OpenMRS module that can change the relationship of a doctor to multiple patients.

Code Modifications

The new module would not modify the existing code base. When we create a new module, it creates a new package with its own files and structure. To connect to the database, we have to create a "Module Application Context File" (see OpenMRS wiki) in the module package which overrides or appends to the current spring application context. This allows our module to access the current Context, which is then used to get objects and directly modify the database.

Implementation Plan

- Create an OpenMRS Module
- Create a "Module Application Context File" called moduleApplicationContext.xml in the module directory so that it can access Context
- Create a new admin page section via an extension point
- Create a new spring controller and jsp page.
- Add java code to controller to accept params and do the relationshipping
- Add unit test for the controller
- Add jsp page according to mockup

Create Module Application Context and Config files

One will be named *moduleApplicationContext.xml*, and it is used to work with the main OpenMRS application by having access to the Context, which has access to objects and the database. The other will be config.xml, which will set up information about the module.

Create class ChangeMultipleRelationshipsActivator

This is a java class that is used to make the module work with the openMRS core API.

Create class ChangeMultipleRelationshipsDAO to query the DB

- replaceAllRelationshipsWithPerson(Integer oldPersonId, Integer newPersonId): void

    - Update all the relationships of a given person with new person
- replaceAllRelationshipsOfRelationshipTypeWithPerson(Integer oldPersonId, Integer newPersonId, Integer relationshipTypeId): void
    - Update all the relationships of a given person with new person and relationship type
- removeRelationshipsWithPerson(Integer personId): void
    - Remove all the relationships of a given person
- removeRelationshipsOfRelationshipTypeWithPerson(Integer personId, Integer relationshipTypeId): void
    - Remove all the relationships of a given person and relationship type

Create class ChangeMultipleRelationshipsService in the API

- Methods:
    - replaceAllRelationshipsWithPerson(Integer oldPersonId, Integer newPersonId): void
    - replaceAllRelationshipsOfRelationshipTypeWithPerson(Integer personId): void
    - removeRelationshipsWithPerson(Integer personId): void
    - removeRelationshipsOfRelationshipTypeWithPerson(Integer personId, Integer relationshipTypeId): void

Create a new test class ChangeMultipleRelationshipsServiceTest

- replaceAllRelationshipsWithPerson:
    - Should update all relationships of old person given the old person id and new person id
    - Should throw exception if old or new person id does not exist
- replaceAllRelationshipsOfRelationshipTypeWithPerson:
    - Should update all relationships of old person given the old person id, new person id, and the relationship type id
    - Should throw exception if old, new person or relationship type id does not exist
- removeRelationshipsWithPerson:
    - Should remove all relationships given a person
    - Should throw exception if person id does not exist
- removeRelationshipsOfRelationshipTypeWithPerson:
    - Should remove all relationships given a person and relationship type
    - Should throw exception if person or relationship type id does not exist

Create ChangeMultipleRelationshipsServiceExtension to add a new link to the admin page section

This is a java class that will provide a linked hash map that will be added onto the admin page, so that the user can go to the page that will change multiple relationships.

Create a JSP page ChangeMultipleRelationships to create a UI for changing a relationship for multiple patients

This is a jsp page that will provide the UI for changing multiple relationships. It should be made so that there will be a list of persons to choose from, and selecting a person will show their relationships. The user will be presented options of whether to remove all relationships relating to that person, or remove all relationships to that person and a relationship type to choose from. If they are replacing, it will be similar to removing, but have to select another person to replace that person with.

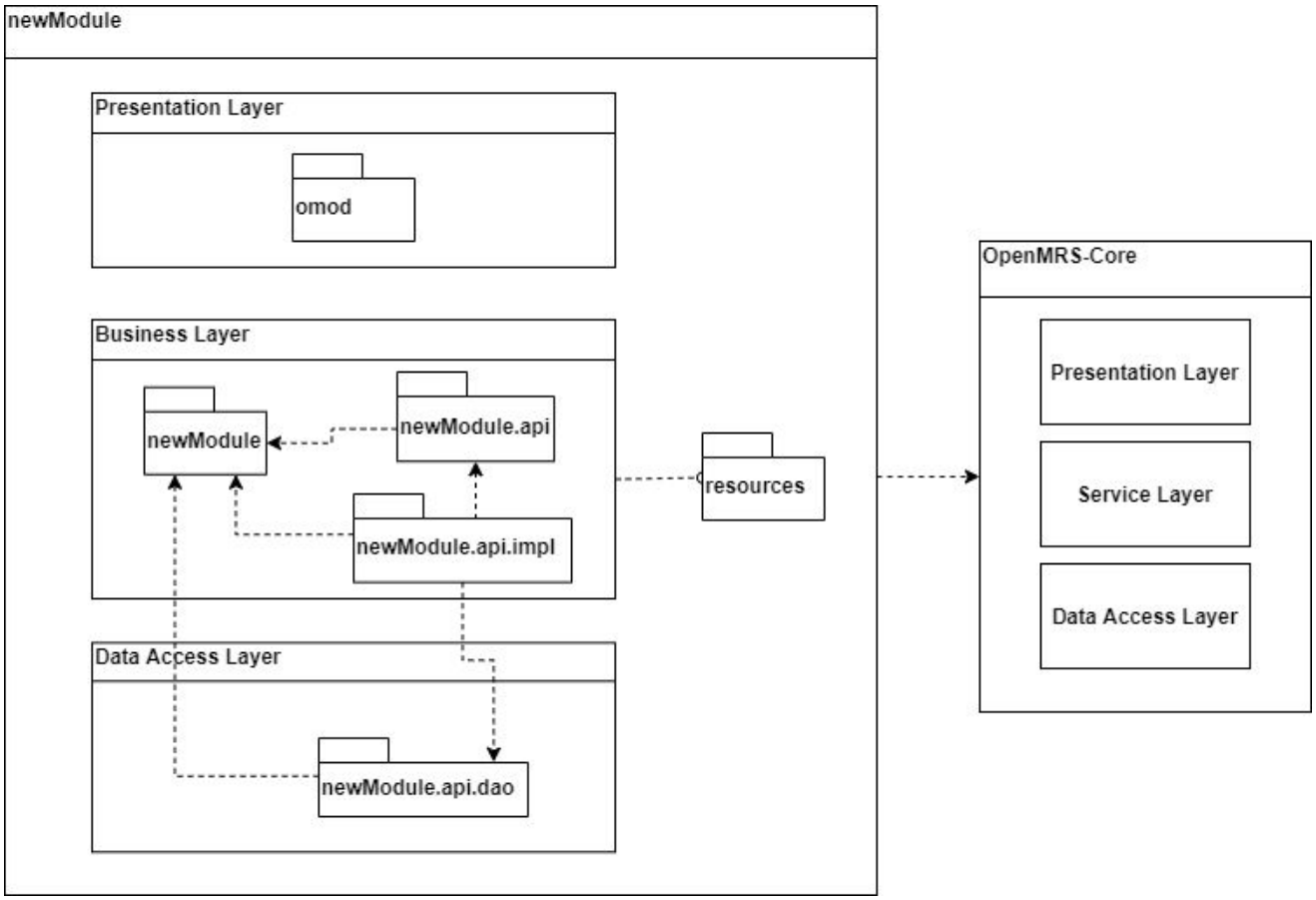
Create a controller ChangeMultipleRelationshipsController

This is a jsp page that would deal with managing the view for the form and also deal with user requests on it, such as getting a query for a list of relationships for a person and returning them.

Create a test class ChangeMultipleRelationshipsControllerTest

This is a java class that will test if the controller is validating the user's requests.

Design



The diagram above shows the structure of the packages for the new module. This is what gets created when you create a new module using the OpenMRS SDK. Refer to the new module as newModule for now. We'll use the same structure for our code if we decide to implement this feature. The newModule package will contain the module activator class and module config class. The activator class registers the module with the core API and config class sets the privileges for the module. The resource package contains the *moduleApplicationContext.xml* file that is used to access the current Context. With the Context class, we could call or modify objects by doing something like *Context.getPerson()*. newModule.api will contain the API interfaces for changing relationships, and *newModule.api.impl* will contain the implementations of it. The omod package will contain subpackages called webapp, java and resources. The JSP and HTML files will be in webapp, and controllers will be in the java package.

Module File Structure

If viewed as a single mavenmodule project:

- **.settings** - Eclipse specific folder containing preferences for your environment
- **api** - non web specific 'maven module' project
  - **src**
    - **main** - Java files in the module that are not web-specific. These will be compiled into a distributable mymodule.jar
    - **test** - contains the unit test java files for the generic java classes
  - **target** - folder built at runtime that will contain the distributable jar file for the module
- **omod**
  - **src**
    - **main**
      - **java** - web specific java files like controllers, servlets, and filters
      - **resources** -
        - config.xml
        - *.hbm.xml files
        - liquibase.xml (or the old sqldiff.xml)
        - messages_*.properties files
        - modulesApplicationContext.xml
        - log4j.xml - optional file to control logging in your module
      - **webapp** - jsp and html files included in the omod
        - portlets -
        - resources - image, js, and css files that your jsp files reference
        - **tags** -
        - **taglibs** -
    - **test** - contains java unit test classes that test the controllers in omod/src/main/java
  - **target** - Contains the distributable omod file
- **.classpath** - Eclipse specific file that points to the files necessary for building the omod and jar files on the fly
- **.project** - Eclipse specific file containing the name and properties of your eclipse project
- **pom.xml** - Maven build file. Delegates to pom.xml files in the *omod* and *api* project

From Creating Modules

Selected Feature

Feature Choice

We decided to implement the feature to support formulary status for drugs. We decided to work on this feature because we are already familiar with the DAO and the api from our last deliverable. Therefore it will be quicker for the team to understand which places in the existing code base needs additions or changes.
We chose this feature rather than the other feature in our list to examine further because it is a feature that adds core functionality to the system that is a nice to have for the clients rather than a quality of life addition that impacts a fewer number of people.

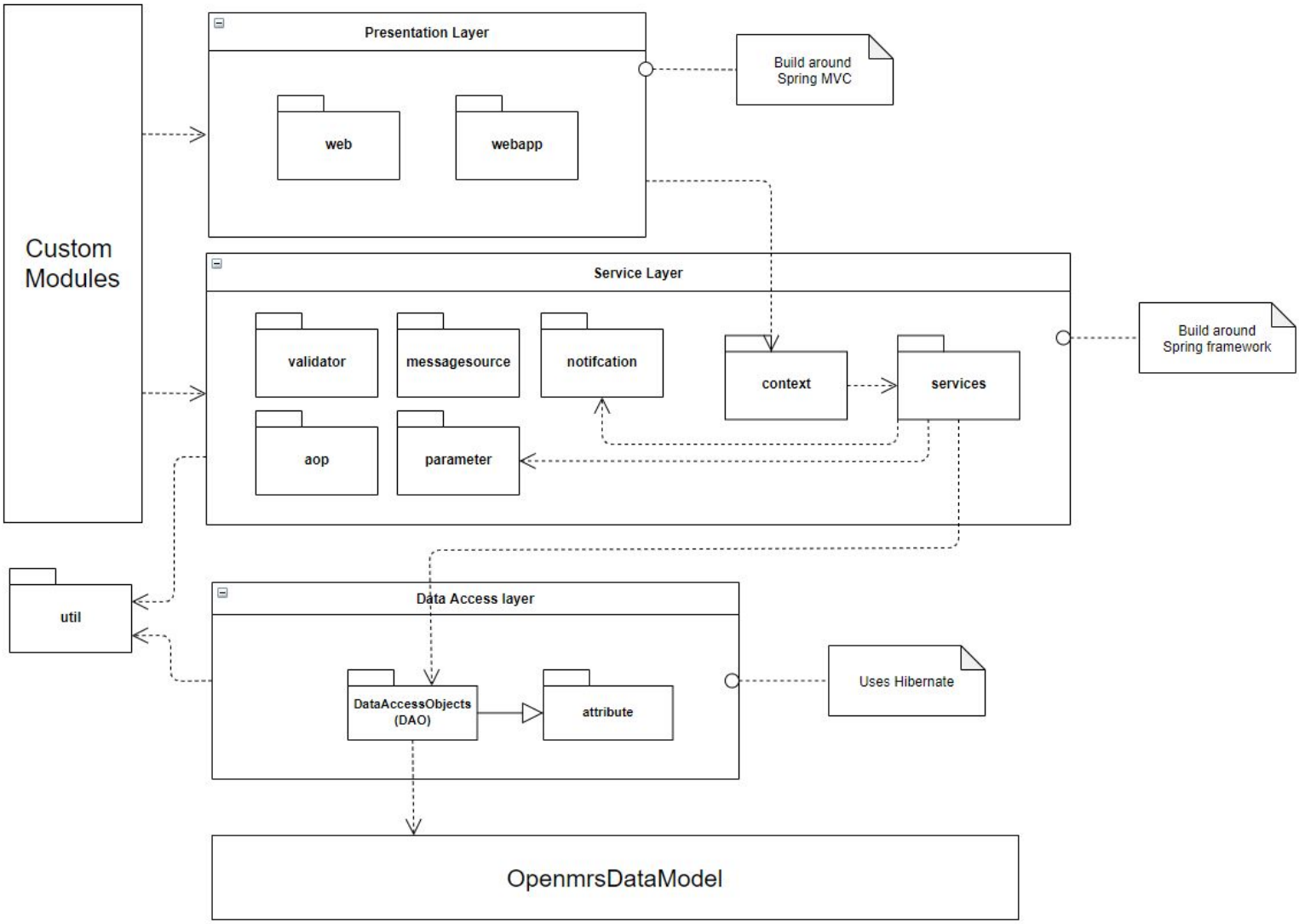
Acceptance Tests

Provided Acceptance Criteria for API:

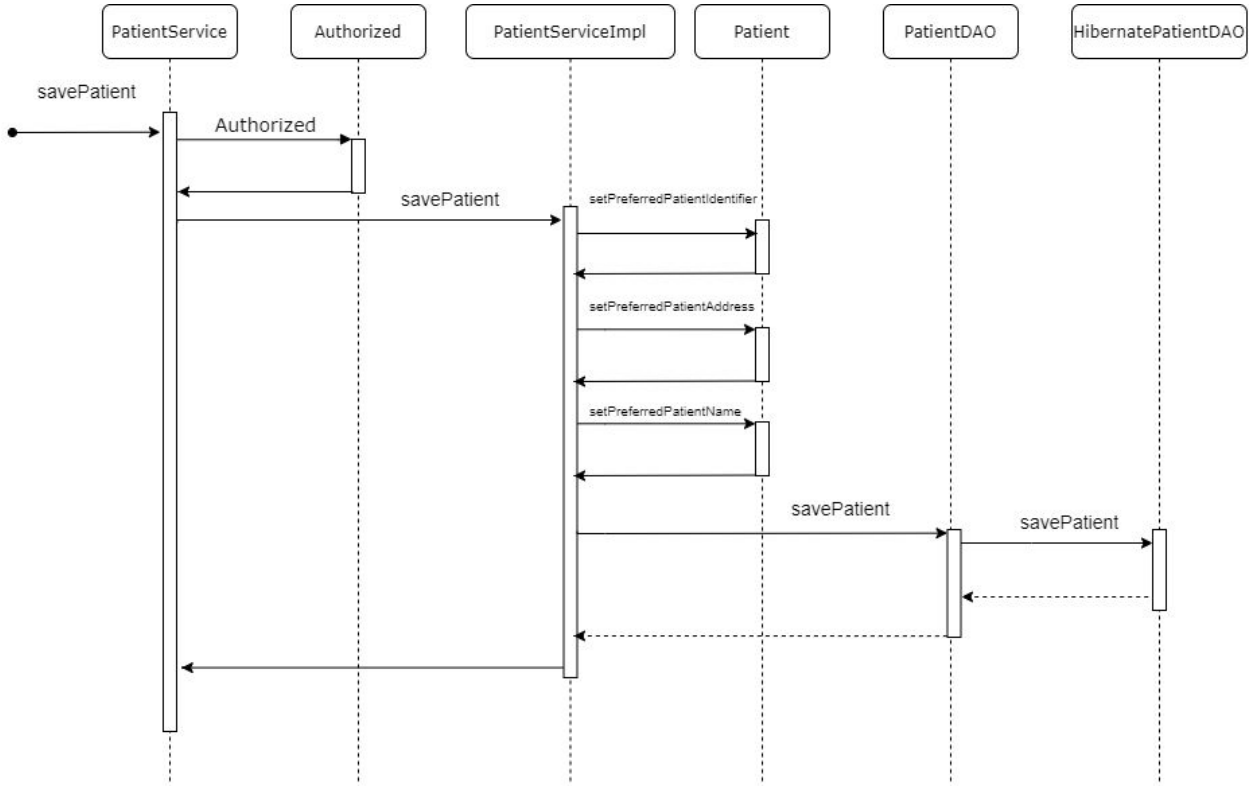
- Clients should be able to get a list of available formularies.
- Clients should be able to check if a drug is on a particular formulary.
- Clients should be able to get the list of formularies for a particular drug.
- Clients should be able to search for a drug within a specified list of formularies.
- Clients should be able to ask the API for a (paginated) list of all drugs within a given formulary.
- Privileged clients should be able to manage the list of formularies and assignment of drugs to formularies.

Project Architecture

Overall Design



Specific Use Case / Example



An example when the PatientService is used to save a new patient to the database.

Design Patterns

Layered Architecture

Presentation Layer

The presentation layer is the user interface for the application. This layer is built around Spring MVC, JSP, and JavaScript.

Service Layer

The service layer contains all the business logic. It uses Spring framework to work with the presentation layer. Each service is registered as a bean in the Spring context. The Context class uses a factory design pattern that has access to all the OpenMRS services. The Spring Aspect Oriented Programming is used to provide separate cross cutting functions. The Spring Dependency injection provides dependencies between components.

Data Access Layer

The data access layer contains all the database queries, with Data Access Object classes. It uses Hibernate to allow developers to describe the relationship between tables.

Modular Architecture

The custom module framework allows developers to extend the default functionality of the OpenMRS-core and modify them for any specific needs. All modules are also structured like OpenMRS-core (consist of user interface, data access and service layers.)
For instance, the user can create a user interface module that allows OpenMRS-core to have a different UI, or create a web service api module that allows others to send requests in accordance to their needs.

Builder Design Pattern

One interesting design pattern in openmrs is that in the *org.openmrs.parameter* package, where all the search criteria class locates. Each search criteria class also has a SearchCriteriaBuilder class that uses a builder design pattern that builds the search criteria. The SearchCriteriaBuilder often used in the services when a user wants to do a query with specific requirements. (The program will create a builder, set its properties to desired values and finally call createSearchCritiera() method to build a searchCritiera class for the query).

Factory Design Pattern

OpenMRS also uses a factory design pattern in the Context class. The Context class uses the ServiceContext class that will get one of many service classes for the Context class. For example, if the Context class wants to save a patient to the database, it uses the ServiceContext class to get the PatientService class, and the context can use that service class to save the patient to the database. In addition, all services are accessed in a static way from the *Context* object. Hence, we do not need to instantiate a new “Context” object nor a new service object every time when we want to have access to an openmrs service.