CS 548—Spring 2019
Enterprise Software Architecture and Design
Assignment Five—Domain Driven Design

Provide a domain-driven design for a clinical information system. You will use the data model you designed in the previous assignment. You should add domain-specific logic to the Java classes of the last assignment. Follow the principles of domain-driven architecture in designing your domain model. You are provided with several Maven projects that you should complete for the assignment:

- ClinicApp: The umbrella enterprise application project.
- ClinicDomain: The domain model for the app.
- ClinicInit0: Initializes the app during deployment.
- ClinicRoot: A Maven parent project for the modules above.

You will have to import these projects into Eclipse as Maven projects. There are several ways to generate an enterprise archive file (ClinicApp.ear) from these projects:

1. From Eclipse, right-click on ClinicApp and select `Export | EAR File`.
2. From Eclipse, right-click on ClinicRoot and select `Run | Maven clean` and then `Run | Maven install`.
3. From the command line in the root directory of ClinicRoot, type "`mvn clean install`".

Some of the later assignments require the use of Maven to generate artifacts (such as DTOs). We are using Maven to manage dependencies, between projects and with external software, so you do not need to manage the build path in Eclipse[1]. Because of this, you should always do (2) or (3) above before doing (1).

The persistence descriptor for the domain project, `persistence.xml` in ClinicDomain, is configured to drop and recreate the database tables each time the app is deployed. The names of the SQL scripts are specified there. The Maven POM file declares the `resources` subfolder of the main folder to be one that may contain resources. You should put the scripts that you generated from the last assignment there. Maven will copy them to the jar file when building the application for deployment.

The Maven POM file for the ClinicApp module lists the components in the enterprise app that is deployed. For now, the only enterprise-specific artifact will be the EJB module for initializing the application, so the POM file has an entry of the form:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
```

[1] If you find that some classes are not recognized in dependent classes, then right-click and select `Maven | Update Project`.

[2] In the next assignment, we will define a service project with a producer class for entity managers, and use

```
   <artifactId>maven-ear-plugin</artifactId>
   <configuration>
      <defaultLibBundleDir>lib</defaultLibBundleDir>
      <modules>
         <ejbModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicInit</artifactId>
         </ejbModule>
      </modules>
   </configuration>
</plugin>
```

Factory and Repository (DAO) Pattern

Use the *factory pattern*. Define factory objects for creating patient, treatment and provider entities. Define *repository (DAO) objects* that encapsulate the use of the entity manager. The entity manager should not be accessed outside a DAO. There should be one DAO per entity type that is persisted (patient, provider and treatment).

A patient or provider entity will need a treatment DAO to access and create treatment entities. How do they obtain such an object, since like all JPA objects they are POJOs rather than managed beans? One pattern to achieve this is to add a transient field for the DAO in these entity objects, and then provide a setter method to set the DAO in the entity when it is persisted or retrieved. For example:

```
@Entity
public class Patient {
      …
      @Transient
      private TreatmentDAO treatmentDAO;
      public void setTreatmentDAO (TreatmentDAO td) {
            this.treatmentDAO = td;
      }
}

public class PatientDAO {
      …
      private EntityManager em;
      public void addPatient(Patient patient) {
            em.persist(patient);
            patient.setTreatmentDAO(new TreatmentDAO(em));
      }
      public Patient getPatient(long id) {
            Patient patient = em.find(Patient.class, id);
            If (patient != null)
                  patient.setTreatmentDAO(new TreatmentDAO(em));
            return patient;
      }
```

```
}
```

Aggregate and Visitor Pattern

Architect patient and provider *aggregate objects* for accessing treatment information. These aggregates encapsulate the underlying domain entity objects, and provide the logic for accessing treatment information without returning treatment objects. To support this, a treatment entity supports the *visitor pattern*: A "visit" operation that takes a callback object with three methods, one for each of the forms of treatment:

```
public interface ITreatmentExporter<T> {
      public T exportDrugTreatment
                  (long tid, ... String drug, float dosage);
      public T exportSurgery (long tid, ... Date date);
      public T exportRadiology
                  (long tid, ... List<Date> dates);
}
```

Provide the following domain logic in your model:

1. Add a new patient to the clinic. The patient entity is created with a factory method. The inputs for this operation should be the patient name and optionally patient identifier. Since the patient identifier is optional (we will still treat a patient, even if they don't remember their patient identifier), the primary key for patient entities will be a separate auto-generated field. The patient DAO then persists the patient entity to the database. If a patient identifier is provided, and a patient with that identifier already exists in the system, then an exception should be raised and the addition aborted without adding that patient again to the database. If successful, the DAO operation should return the primary key for the new patient object in the database.
2. Retrieve a patient aggregate from the system, given the primary key for the patient. This operation is provided by the patient DAO. An exception should be thrown if there is no patient record for the input key.
3. Retrieve a patient aggregate from the system, given the patient identifier for the patient. Again this is a DAO operation. Use a JPQL query to search for patients matching the provided patient identifier. Raise an exception if there is not exactly one patient in the database with that patient identifier.
4. Add a provider to a clinic. As above, this is provided by a provider factory and a provider DAO. We will assume that a provider's National Provider Identifier (NPI) is provided with their name and their specialization, when they are added to the clinic. It is an error to attempt to add a provider to a clinic if a provider with that NPI is already defined for the clinic.
5. Retrieve a provider aggregate from the clinic. As with patients, there are two operations:
    a. Return the provider for a primary key.
    b. Return a provider aggregate given a NPI

6. Add a treatment for a patient. This operation takes a treatment entity object and inserts it into the database. Use the following logic:
    a. Use the treatment factory to create the entity.
    b. Define an insertion operation in the provider aggregate for inserting a treatment into the database. This operation should take a treatment entity and an associated patient entity.
    c. Define an insertion operation in the patient aggregate that takes the treatment entity and inserts it into the database, using a treatment DAO. This should be called from the provider operation. The insertion operation should return the primary key for the new treatment object in the database. You will need to sync with the database to generate the new primary key.
    d. The patient operation sets forward and backward pointers between the patient and treatment entities, while the provider operation sets forward and backward pointers between provider and treatment entities.
7. A patient aggregate does not return a set of treatment entities directly to a client, since this would violate the encapsulation of the aggregate pattern. Instead a patient aggregate provides access to its treatment entities with these operations:
    a. One operation returns a list of treatment identifiers for that patient's treatments.
    b. A second operation allows a particular treatment, identified by a treatment identifier, to be visited, taking a visitor object as one of its parameters. In this case, check that the specified treatment is in fact associated with the current patient. If it is not, raise an exception and log the attempted access to the treatment.
8. Similarly, the provider aggregate should provide these operations for accessing treatments:
    a. Return a list of all treatments (treatment identifiers) associated with that provider.
    b. Visit a particular treatment specified by a treatment identifier (primary key). This last operation should check that the provider is in fact supervising the treatment specified by the treatment identifier.

Make sure to include the logic for programmatically maintaining the relationships that exist between patients, providers and treatments, as entities are inserted. If you do not do this, you will get an exception when your program tries to flush your object graph to the database, due to violation of database integrity constraints.

One of the issues to be dealt with is that you cannot rely on dependency injection to inject resources, including references to entity managers, into entity objects. Dependency injection is only available in service objects, which we consider in the next assignment. That is why the patient and provider aggregates rely on an additional operation for setting a treatment DAO, but how do we obtain these DAOs, since they rely on the provision of an entity manager? For now, you are provided with a service object, `InitBean`, for initializing the database at app deployment. This is defined in the `ClinicInit0` project. Since this is a managed bean, you can use

resource injection to inject an entity manager for the application persistence context[2]:

```
@PersistenceContext(unitName=”ClinicDomain”)
EntityManager em;
```

The initialization logic in the bean can use this entity manager to instantiate the DAO classes, using them to persist and retrieve patient and provider entities. Treatment entities are persisted and retrieved internally in the patient and provider aggregates , using their internal treatment DAOs.

Since the application does not have an external interface yet, do your testing in the initialization bean, displaying the results of testing in the log in the application server. Testing your code now will avoid postponing all your testing until a later assignment, when you deploy with an external interface. A sketch of the definition of the initialization bean is provided to you below. You should fill this in with your own initialization logic in the `init()` method.

```
@Singleton
@LocalBean
@Startup
public class InitBean {
	private static Logger logger =
		Logger.getLogger(InitBean.class.getCanonicalName());
	private static void info(String m) {
		logger.info(m);
	}

	public InitBean() { }

	@PersistenceContext(unitName = "ClinicDomain")
	private EntityManager em;

	@PostConstruct
	public void init() {
		info("Initializing the user database.");

		IPatientDAO patientDAO = new PatientDAO(em);
		patientDAO.deleteAll();

		// Insert code to test the functions of your domain model
		// Use info(“...”) to display the results in the log
	}
}
```

[2] In the next assignment, we will define a service project with a producer class for entity managers, and use dependency injection to inject the persistence context.

To test your code, export the `ClinicApp` as an enterprise archive. The best way to do this is using Maven. Right-click the `ClinicRoot` project and execute `Run | Maven install`. This will produce a file called `ClinicApp.ear` in a subdirectory `cs548` of your home directory. Use the administration console in the application server to deploy your application, as you did with the "Hello, World" application for the first assignment. The application server will instantiate the `InitBean` class to create a tester bean, and call its `init()` method once it is created. The output of the tests will be printed to the server log, which you can view from the Admin Server tab in the administration console (click on "`View Log Files`", although you may be better advised to view the raw logs since the log viewer is unreliable and may not show you everything).

Your solutions should be developed for Java EE 8. You should use Payara (Glassfish 5.0) and Eclipselink 2.5.2. You should use Java 8 with Eclipse Oxygen. In addition, record short mpeg or Quicktime videos of a demonstration of your assignment working (deploy the app and view the server raw log file in enough detail to see the output of your testing). Make sure that your name appears at the beginning of the video. *Do not provide private information such as your email or cwid in the video*. You can upload this video to the Canvas classroom.

Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have four Eclipse Maven projects: ClinicRoot, ClinicApp, ClinicDomain and ClinicInit. You should also provide videos demonstrating the working of your assignment. This involves showing your project being successfully deployed, and viewing the log files to see the output of successful testing. Finally the root folder should contain the enterprise archive file (ClinicApp.ear) that you used to deploy your application. You should also provided a completed rubric.