CS 548—Fall 2022
Enterprise Software Architecture and Design
Assignment Four—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:

- `clinic-webapp0`: A dummy Web app for the project.
- `clinic-domain`: The domain model for the app.
- `clinic-init0`: Initializes the app during deployment.
- `clinic-root`: A Maven parent project for the modules above.

You will have to import these projects into Eclipse as Maven projects. Use the root project to compile these projects and generate the WAR file, `clinic-webapp.war`, for the Web app deployed from this project. We are using Maven to manage dependencies, between projects and with external software, so you should not manage the build path in Eclipse[1]. The Web app currently does not do anything except execute the initialization logic in `clinic-init0`, which in turn executes the logic in the domain model.

The persistence descriptor for the domain project, `persistence.xml` in `clinic-domain`, is configured to create or extend the database tables each time the app is deployed. You will need to edit it to list the entity classes. Besides the classes for the main domain, there are entity classes for other applications (billing and research) that will be used in a later assignment.

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). These DAOs should be defined as CDI beans, which are then injected where required in managed beans (See the initialization logic in `clinic-init0`):

```
@RequestScoped
public class PatientDao implements IPatientDao {

    @Inject @ClinicDomain
    private EntityManager em;
```

The producer of the entity manager is also defined as a CDI bean:

```
@RequestScoped
@Transactional
public class ClinicDomainProducer {
```

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

```
    @PersistenceContext(unitName="clinic-domain")
    EntityManager em;

    @Produces @ClinicDomain
    public EntityManager clinicDomainProducer() {
       return em;
    }
```

This is the only place where you should use the `PersistenceContext` annotation. The protocol for adding a treatment is that it is always added by a provider entity:

```
public void addTreatment (Patient p, Treatment t) {
    treatmentDao.addTreatment(t);
    p.addTreatment(t);
    ...
```

The provider in turn calls the patient to add the treatment to its own list of treatments, setting forward and backward pointers (with an operation that should only be called by the provider):

```
void addTreatment (Treatment t) {
    treatments.add(t);
    if (t.getPatient() != this) {
        t.setPatient(this);
    }
}
```

Note that a follow-up treatment is also between a patient and provider (though not necessarily the same provider as for the original treatment). 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.

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 {

    @Inject
    private ITreatmentDao treatmentDao;

    @Override
    public Patient getPatient(UUID id) {
        Patient p = ...
        p.setTreatmentDao(this.treatmentDao);
        return p;
    }
```

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 four methods, one for each of the forms of treatment:

```
public interface ITreatmentExporter<T> {
    public T exportDrugTreatment (UUI tid, ... String drug, ...);
    public T exportSurgery (UUID tid, ... LocalDate date, ...);
    public T exportRadiology
                  (UUID tid, ... List<LocalDate> dates, ......);
    public T exportPhysiotherapy
                  (UUID tid, ... List<LocalDate> dates, ......);
}
```

We will use this pattern in the next assignment for extracting information from the domain model without violating the aggregate pattern. For now, the initialization bean in this assignment implements `ITreatmentExporter<Void>`; it does not extract data but is used to log the contents of the database.

Testing

Since the application does not yet have an external interface, 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 implements ITreatmentExporter<Void> {

    @Inject
    private IPatientDao patientDao;

    @Inject
    private IProviderDao providerDao;
    @PostConstruct
    private void init() {
        info("Initializing the database.");
        ...
    }
}
```

This bean injects patient and provider DAOs, to persist entities of these types. Treatment entities are persisted and retrieved internally in the patient and provider aggregates , using their internal treatment DAOs.

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. Use the `docker logs` command to view the logs in your running server.

You will need to do the following with the app server to deploy the app:

1. You will need to set up the JDBC connection pools and resources in the application server for the database, as before. Use a resource name of `jdbc/cs548` for the JNDI name of the connection pool for the app database.
2. You will have to initialize the database with its tables. The persistence descriptor persistence.xml in `clinic-domain` is set up to automatically create or extend the database tables each the app is deployed.

Your solutions should be developed for Jakarta EE 8. You should use Payara 5.2022. You should use Java 17 with Eclipse. In addition, record short mpeg 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: `clinic-webapp0`, `clinic-root`, `clinic-domain` and `clinic-init0`. 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 WAR file (`clinic-webapp.war`) that you used to deploy your application. You should also provide a completed rubric.