**Overview of Testing**

The activity of designing your own database system is not just entertaining, but also gives you a chance to innovate and learn by doing. While it’s easy to design the system and write code to implement it, it’s also easy to miss connections and details that would make the system more robust. This is where the importance of testing comes in. While going through the testing requirements, we discovered some issues with our design, which when addressed would make our system more powerful. While some issues were design related, there were others that simply required implementation of triggers to ensure that the data in our system is what we would want it to be.

Our system does not guarantee that an underage Patient has a Parent because there is no way to force the user to add a parent when adding an underage patient. The user must manually add a Parent, and then add an entry to the PatientHasParent table to tie the Parent and Patient together. The design seems appropriate and supports many-to-many relations between Patient and Parent. To guarantee a Parent is created for underage Patients, we could create a trigger that checks the age of the Patient upon insertion and forces creation of Parent before Patient can be created if the age of the Patient is under 18 years.

Any Patient record being entered requires a manual insertion of a PatientHasParent record. Since we don’t manually insert a PatientHasParent record for a Patient over the age of 18 being inserted, the test passes in this case. However, if the age of an already existing Patient record is updated (which could be a Patient under the age of 18), the Parent record associated with the Patient isn’t deleted. Having a trigger that gets called on any updates to the age of a Patient would solve this problem by checking if the age of the Patient is over 18, and if it is, then the associated Parent record can be deleted.

Yes, the Has\_insurance attribute documents when a patient does not have insurance. The system also allows a patient that does not have an Insurance entity linked to it. A flaw is that the Has\_insurance attribute must be manually set. If there is a Patient with insurance and their insurance expires, the flag is not updated automatically. We need a trigger to update the Has\_insurance attribute when the End\_date attribute of Insurance is changed. Similarly, if a user doesn’t have insurance and gets insurance, the Has\_insurance attribute must be set to true. Our design retains old insurance information, and it allows users to access the current insurance by checking if the End\_date attribute of Insurance is NULL.

Since each visit has a unique id associated with it, this ensures that a Patient can only be seen by one ServiceProvider in a visit. The primary key constraint as well as a constraint on the domain of the ServiceProvider id works well to ensure the success of the test that checks that only one ServiceProvider can be associated with a Visit.

We can have multiple Diagnosis associated with the same Visit. But the system does not require having at least one Diagnosis. We can update the design to always have one Diagnosis associated with the Visit (which could be some sort of default value of “No Diagnosis”). We can create a trigger that calls a function to create a Diagnosis for the Patient if one is given by the ServiceProvider and this would overwrite the default value.

The Diagnosis for a Visit can be made by anyone. Our test checks and ensures that a ServiceProvider can successfully make a Diagnosis. However, there was an oversight in our design. We clearly intended to have a foreign key constraint on the service provider id being stored in our VisitHasDiagnosis table, but somehow missed it which allows anyone to make a Diagnosis for the Patient whether they even are an Employee or not.

The system fails to document the followup tests/procedures with proper coding. While running this test, we found that random codes could be entered for the ICD-10-PCS-Code. This requires a check that would ensure that whenever a Treatment record is being inserted, the ICD-10-PCS-Code being entered is valid. This could be done by comparing the code being entered against a regular expression that contains what the proper coding should look like.

The system successfully documents which IntakeClerk collected the insurance and copay information for a Visit. A Visit entity cannot be created unless an intake clerk id is provided and this ensures that there’s always an intake clerk associated with the Visit. Having a foregin key constraint on the intake clerk id ensures that the intake clerk being assigned to collect the insurance information for a visit exists in the system and is valid.

InitialAssessment is a weak entity that depends on the Visit entity for its existence. The system does allow multiple InitialAssessments to exist at any time. This was checked by inserting a couple of Visits and InitialAssessments and ensuring that it could be done successfully. The system also ensures that one Visit can only have one InitialAssessment associated with it.

Each Visit that a Patient makes has a unique id associated with it which is its primary key. Hence, the system allows a Patient to visit the urgent care facility more than once and the test passes successfully.

Initial assessments have the nurse’s information associated with it and this information can be retrieved using the Nurse and Employee table. However, since our design does not have a foreign key relation to the Nurse table, we could simply add in a Nurse ID to the initial assessment that isn’t valid. This is a weakness of our design and could have been taken care of by a simple foreign key constraint.

A nurse can do assessments for multiple patients. In the test, we created two patients and one nurse. After the nurse finished assessments for both the patients, we created a SQL query to get all the patients that the nurse did the assessment for, and the result was the two patients.

The initial assessment can be completed by only one nurse. We created a Visit and a Nurse, and the nurse completed the assessment successfully. This aligns with the design we have since each InitialAssessment can have only one Nurse associated with it.

The queries to review the vitals of patient from a given visit is doable because InitialAssessment, InitialAssessmentHasMedicalConditions, and InitialAssessmentHasMedication are all tied to the visit’s id. We simply need the visit id to retrieve all the vitals of the Patient from the InitialAssessment.

The classification of an Employee into Hourly or Salaried fails because of a design flaw. Even though the specialization exists for classifying each Employee as either Hourly or Salaried, the system does not require such a classification to be done on insertion of an Employee record. We could have avoided this specialization altogether. The design update that makes more sense here is to have an attribute associated with the Employee record that stores their pay information and to make this attribute NOT NULL. This would ensure that whenever an Employee record is created, their salary information is captured by the system.

Our systems allows a Doctor to request more than one Treatment per Visit. A Visit is tied to exactly one Doctor, and the Visit entity has a many to many relationship with the Treatment entity. We inserted several treatments and tied them to the Visit through the VisitHasTreatments table to verify this requirement.

Based on the design that we have, a Patient can have no tests/procedures during a Visit. Treatments and Visits are tied together through the VisitHasTreatment table. If an entry is not inserted in this table for a Visit, it does not have any treatments. In the test, we created a visit without tests/procedures. This makes sense since it is not necessary for each Visit to end up with a Diagnosis that requires a Treatment.

We found that there is no way to enforce a minimum constraint in the many-to-many relationship between a Patient and EmergencyContact. This is not a design flaw because we can’t force users to add a row to the PatientHasEmergencyContact table. A trigger could add an EmergencyContact after a Patient is added. Maximum constraints are supported because we don’t limit the number of entries in the PatientHasEmergencyContact table.

We enforced maximum and minimum constraints on the many-to-one relationship between Employee and Department. Employees must have a department. Since we placed a NOT NULL constraint on then DId forgien key in Employee, a department id must be provided when inserting an employee. Otherwise, the insert statement will fail. Also, there cannot be more than one department associated with an employee because there is only one attribute for the department id in the Employee table. Departments can have more than one employee working under it since the DId attribute can be repeated for different employees.

The primary keys for Treatment and Patient ensures that each row in the table can be uniquely identified. For each table, a primary key is required when inserting a row. In testing we found that the system did not allow duplication of primary keys or null values as primary keys. Our primary keys worked as expected.

Testing revealed a few flaws in the design of our system. Most of these oversights can simply be resolved by adding additional foreign keys and other types of constraints. However, the designation of hourly and salaried employees requires a more significant change to the design. Many of the issues we discovered cannot be resolved purely with an improved design. They require additional code to be written, such as triggers, checks, and assertions. For the most part, our system was robust.