Table of Contents

Iteration Introduction

Defining a direction for your database and describing how it will be used is a great first step, and this what you accomplished for Iteration 1, but this is just the beginning of the process. Your goal is a live database that supports the organization or application that uses it; there are several more components that need to be created to make this happen. You will create two such components in this iteration.

Your next step is to more formally design your database. Structural database rules are a great place to start, a useful tool to frame and guide your design. Structural database rules are written carefully to ensure that they define specific components and constraints for your database. You create these for your database in this iteration, then create entity-relationship diagram (ERD), a universally accepted method of visualizing database schemas. With your structural database rules and ERD, you are able to articulate and visualize the data and relationships that exists in your mind for your database. Structural database rules and ERDs are the foundation of your database design.

Let's again look at an outline of what you created in Iteration 1, and what you will be creating in this and future iterations.

Prior Iteration	Iteration 1	*Project Direction Overview* – You provide an overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it. *Use Cases and Fields* – You provide use cases that enumerate steps of how the database will be typically used, also identify significant database fields needed to support the use case. *Summary and Reflection* – You concisely summarize your project and the work you have completed thus far, and additionally record your questions, concerns, and observations, so that you and your facilitator or instructor are aware of them and can communicate about them.
Current Iteration	Iteration 2	*Structural Database Rules* – You define structural database rules which formally specify the entities, relationships, and constraints for your database design. *Conceptual Entity Relationship Diagram (ERD)* – You create an initial ERD, the universally accepted method of modeling and visualizing database designs, to visualize the entities and relationships defined by the structural database rules.
Future Iterations	Iteration 3	*Specialization-Generalization Relationships* – You add one or more specialization-generalization relationships, which allows one entity to specialize an abstract entity, to your structural database rules and ERD. *Initial DBMS Physical ERD* – You create an initial DBMS physical ERD, which is tied to a specific relational database vendor and version, with SQL-based constraints and datatypes.
	Iteration 4	*Full DBMS Physical ERD* – You define the attributes for your database design and add them to your DBMS Physical ERD. *Normalization* – You normalize your DBMS physical ERD to reduce or eliminate data redundancy. *Tables and Constraints* – You create your tables and constraints in SQL. *Index Placement and Creation* – To speed up performance, you identify columns needing indexes for your database, then create them in SQL.
	Iteration 5	*Reusable, Transaction-Oriented Store Procedures* – You create and execute reusable stored procedures that complete the steps of transactions necessary to add data to your database. *History Table* – You create a history table to track changes to values, and develop a trigger to maintain it. *Questions and Queries* – You define questions useful to the organization or application that will use your database, then write queries to address the questions.

Before you proceed to add more items to your design document, consider first revising what you completed in Iteration 1. Since you have learned more about database concepts, have had additional time to think about your database, and have received feedback from your facilitator or instructor, you may have noticed tweaks and enhancements that would benefit your project.

Organizational Analysis

Before we dive into the details of what structural database rules and how to create them, let us first provide some background. Every organization operates according to a set of rules. For example, courts processes cases in a certain way for everyone to ensure justice and equality, consulting companies negotiate contracts and pay consultants in a consistent manner, and international aid charities have processes in place to collect donations and provide care. These rules are not always written down or formalized, and it may be that no one person knows them all; knowledge of them may be spread across different departments and people. Nevertheless, such rules exist for every organization, to help govern what they do and how they do it.

When an application or I.T. system is being created for an organization, careful analysis of the organization and its processes results in a formal description of their operation. Business rules and domain models, amongst other artifacts, are commonly created during such analysis. We need this formality to help automate operations through I.T. systems, because while the organization may operate without these rules being formalized or written down, an I.T. system cannot be created without a formal knowledge of how they operate.

Domain models model high-level business objects and their relationships to each other, commonly using UML class diagrams; however, business rules are of more interest to us for purposes of modeling our database. Each business rule defines or constrains an aspect of an organization's structure and processes. The following examples of general business rules should help give you an idea of what they are about.

Business Rule Examples

Organization	Business Rule
Criminal Court	Each arrest results in one or more dockets being created for the court to process.
Consulting Company	Each consultant on a contract is given a specific per-hour rate.
International Aid Charity	Donations are accepted in-person, online using credit cards or PayPal, or with a check via regular mail.

Business rules, as the name suggests, are focused on the general operation of the business, but are not design or implementation details for the application or database. Rather, business rules along with other items are the foundation for creating rules and constraints for each application or I.T. system component. We consider this formal analysis of the organization one of the first steps involved in creating an application or I.T. system, but it is certainly not the last.

Each application component has its own set of concerns and technologies, and not surprisingly we need to define constraints and rules for each component individually. For example, security specialists create quite specific security requirements the application must implement to ensure the application is secure.

Application programmers create UML diagrams such as class diagrams and sequence diagrams to help describe how the application will be designed. Database designers create *structural database rules* and entity-relationship diagrams to describe how the database will be designed. Each component of the application is designed to support what the organization needs.

Structural Database Rules

Structural database rules are written carefully to ensure that they define specific components and constraints. Each rule specifies two entities, a relationship between the entities, participation constraints on both sides of the relationship, and plurality constraints on both sides of the relationship. Entities? Relationships? Constraints? What are you talking about? If this is what you are thinking right now, don't worry. We'll explain each in turn.

An entity is a blueprint for a data set. An entity defines the name, attributes, allowable values, and whether or not the attribute is required or optional for the dataset. Each item in the dataset contains only the attributes as defined by the entity. Technically, each item in the dataset is termed an *entity instance*, a term you will see in the textbook and online lectures; however, do not let the term itself confuse you. An entity instance is one item in the entity's dataset, and it has all of the attributes as defined by the entity.

Let's look at an example entity, Person. We can presume a Person entity would have attributes like first_name, last_name, and birth_date, data points that help distinguish one person from another and that databases typically store. A Person entity could be defined as illustrated below.

Person Entity Example

Person	
first_name	varchar(255)
last_name	varchar(255)
birth_date	date

The name attributes are given the ANSI standard varchar datatype to indicate they can contain characters, and birth_date is given the ANSI date datatype.

Some example items (entity instances) in the Person data set could look as follows.

Example Person Instances

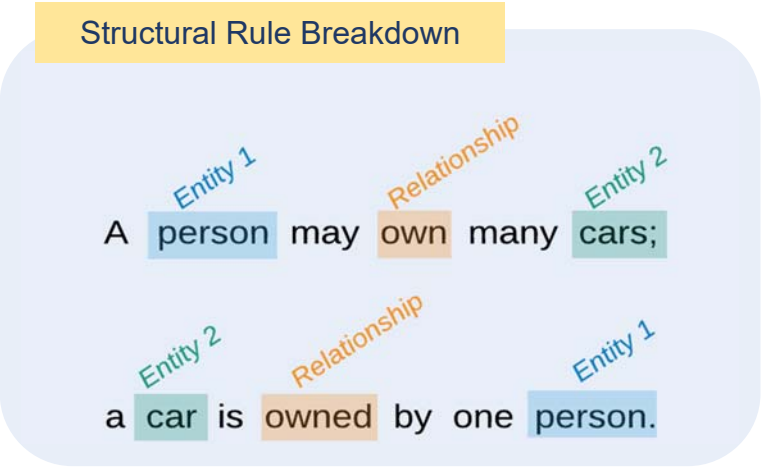
Person 1	Person 2	Person 3
first_name = John last_name = Glass birth_date = 10/13/1972	first_name = Aria last_name = Eleazer birth_date = 9/5/2000	first_name = Decker last_name = Gordon birth_date = 3/13/1950

Notice that each entity instance has only the attributes defined by the entity, and the values are only of the type defined by the entity (such as characters for the name fields, and dates for the birth_date field).

This is why entities are said to be blueprints, because they define the structure of all instances. Note that we have presumed the attributes for purpose is illustration, but when defining your design, you want to stick with attributes you know are required through your analysis. You don't want to presume anything for your design.

Now back to relationships. Relationships are commonplace in the real world. If we say "People have a mailing address", we are saying that each person is related to a mailing address, that is, "person" and "address" are associated to one another. Other examples are "smartphones have a manufacturer" and "houses have windows". We express these kinds or real-world relationships in the structural database rules by formally associating two entities. So, we can say that a database relationship is an association between two entities.

Let's look at an example structural database rule, "A person may own many cars; a car is owned by one person". The following diagram illustrates how this rule specifies the entities and relationship.

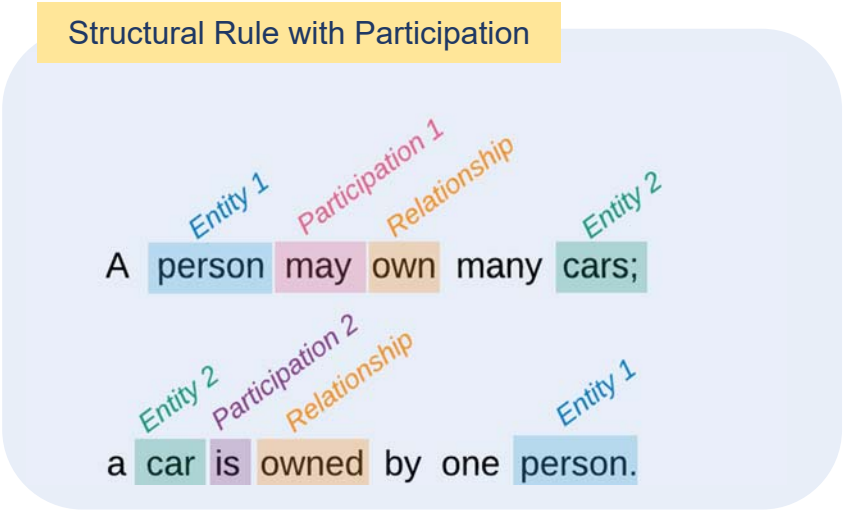


Notice that "person" has been identified as the first entity, and the "car" has been identified as the second. Further notice that "own" has been identified as the relationship. You might wonder why there are two statements rather than just one, since it seems to duplicate the same information, that a person is related to car through an owns relationship. The reason lies in the fact that we must also define the constraints in the relationship.

A participation constraint indicates whether or not each data time (entity instance) must participate in the relationship. If each one must, the relationship is said to be *mandatory*; otherwise it is said to be *optional*. When we look at the statement "a person *may* own many cars", the "may" tells us that the relationship is optional to person, that is, each person may or may not own a car. When we look at the statement "a car *is* owned by one person", the "is" word, along with the absence of any words like "may", "could", and the like, tells us that the relationship is mandatory to car. That is, each car must be owned by a person. We could get into philosophical arguments about whether this accurately reflects the real world, but what matters most is the way it is stored in our database. If the people stored in our database may or may not own cars, but each car in our database must be owned by a person, the rule holds true. For example, perhaps our database tracks car ownership, so its implicit that every car has an owner.

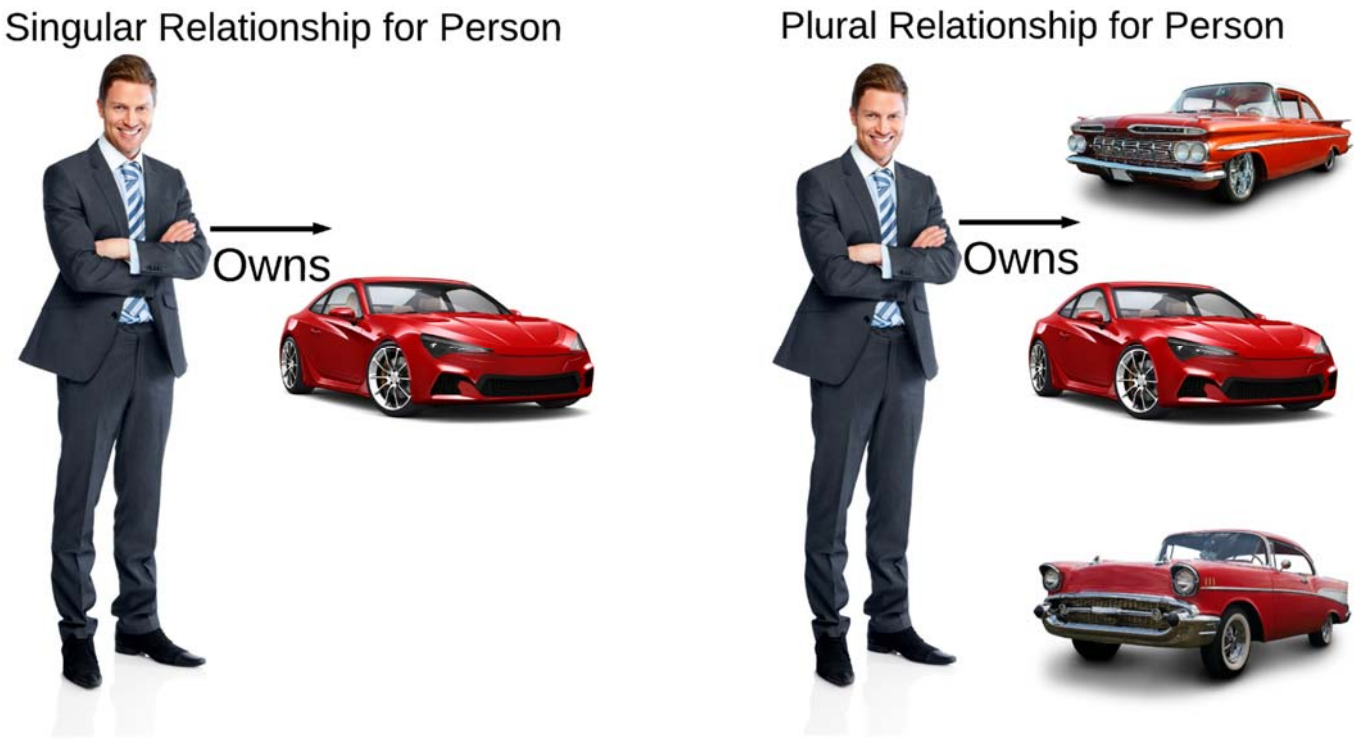
Further notice that since there are two entities, we must specify the participation constraint from the perspective of both entities. We need to know if the relationship is mandatory for Person *and* for Car, not just one or the other. This is why there are two statements in structural database rules rather than just one.

Now we can expand our diagram to include the participation constraint, as illustrated below.



In the diagram, you now see more words highlighted that are involved in the participation constraint – "may" and "is". "A person *may*…" tells us that a person does not have to own a car. "A car *is*…" tells us that a car is owned, not that a car may be owned. If we use "might" or "could" instead of "may" to indicate the relationship is optional, such as "a person *might* own many cars" or "a person *could* own many cars", that is fine. We could use "must be owned" or something similar instead of "is" to indicate the relationship is mandatory to Car, such as "a car must be owned by a one person". The precise wording used in structural database rules can vary somewhat.

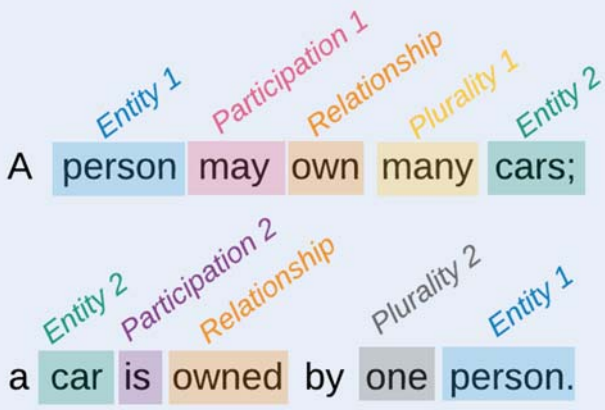
A plurality constraint indicates whether each data item (entity instance) can be associated to only one, or more than one, data items. Continuing with our example, one question is, if a person owns cars, can they own only one, or can they own more than one? If a person can own more than one, we say that the relationship is *plural* for Person. If a person can only own one, we say that the relationship is *singular* for Person. This concept is illustrated below.



When Person can own only one car, then it's singular; otherwise, it's plural. Just as with the participation constraint, we choose our words carefully to disambiguate the plurality constraint. We say "a person may own *many* cars" to indicate that the relationship is plural to Person (i.e. one person may own more than one car). We say "a car is owned by *one* person" to indicate the relationship is singular to Car (i.e. one car can only be owned by one person). One caveat here is that a plural relationship doesn't mean every person *must* own more than one car; it only means they *can* own many cars. On another note, this person owns some nice cars, which is probably why he's so happy!

We can now present a complete diagram with all relevant words highlighted.

Structural Rule, All Keywords Highlighted



Notice that the words “many” and “one”, indicating plurality, have been highlighted.

Take time to review the diagram above carefully to make sure you understand the significance of each of the highlighted words. If you master this structure, you will find it more straightforward to model your database.

One principle for structural database rules is, *we can phrase them in many different ways, but the entities, relationship, and constraints must be entirely unambiguous*. This principle of unambiguity instills many requirements for structural database rules. If part of a rule can be interpreted in different ways, then it is ambiguous and we need to correct that.

One subprinciple here is that first entity in each statement must be stated in the singular. Notice that we stated “a person…” and “a car…”, in the singular. Let’s look at an example if we had not done this, “people may own many cars”. For this statement, the plurality constraint ambiguous. Why? Because even though we see “people may own *many* cars” we don’t know if “many” applies to *one* person, or if that applies to many people collectively. That is, if we say “people may own many cars”, perhaps there are 10 people who each own one car, so people own many cars. In that case, each person only owns one car, but collectively they own many. Or, perhaps each individual person owns many cars. We just can’t tell from “people may own many cars”. But if we say “*a person* may own many cars”, this is unambiguous, because now we know that *each individual person* may own many cars.

Plurality is not the only constraint affected; participation can be affected as well. What if we simply stated “cars are owned by people”? While we do not see any words like “can”, “might”, or “may” to indicate optionality, participation is still ambiguous. Is *every* car owned by a person, or just some? Maybe it’s just a casual statement that cars are owned by people, or maybe it’s a strong statement that *every* car is owned by a person. We just can’t tell from that rule “cars are owned by people”. But if we state “*a car* is owned by one person” it’s very clear that *each individual car* is owned by one person, making the participation constraint clear.

Additional Structural Database Rules Examples

To ensure you have a solid understanding of structural database rules, let’s look at a few more examples. After all, what better way to learn than by example?

We’ll start by creating a structural database rule from one of our example criminal court business rules we looked at earlier, “Each arrest results in one or more dockets being created for the court to process.” From this business rule, we can spot three entities – Arrest, Docket, and Court. We can also see some relationships as well. Arrest is related to Docket through a “results” relationship, and from the business rule it looks like there is a one-to-many relationship between them. So we could create the structural database rule as follows.

> *Each arrest results in one or more dockets; Each docket was created from an arrest.*

If we further analyze this structural database rule, we can break down the participation constraints and plurality constraints as well.

> *From the perspective of Arrest, it must participate in the relationship (mandatory participation).*
> *From the perspective of Arrest, it may be associated to many Dockets (plural).*

From the perspective of Docket, it must participate in the relationship (mandatory participation).
From the perspective of Docket, it is associated to exactly one Arrest (singular).

Now for the Docket to Court relationship, we can see that Docket is associated to court through a "Process" relationship. The original business rule gives us some clue to the constraints as well. We create the structural database rule as follows:

Each docket is processed in a court; each court processes many dockets.

We can break down the constraints as follows.

From the perspective of Docket, it must participate in the relationship (mandatory participation).
From the perspective of Docket, it is associated to one Court (singular).
From the perspective of Court, it must participate in the relationship (mandatory participation).
From the perspective of Court, it is associated to many Dockets (plural).

Let's look at another business rule we previously looked at, "Each consultant on a contract is given a specific per-hour rate." From here, we can definitely spot the Consultant and Contract entities. "Rate" could be an entity, or it could be an attribute that is a part of the relationship between Consultant and Contract. We don't have enough information to know which, but for simplicity we'll model it as an entity.

The first structural database rule could be written as follows.

Each consultant is on a contract; each contract has consultants.

Now to be fair, from the original business rule alone, we don't know if a consultant could be on more than one contract or just one. However, we can use our knowledge of the real world to conjecture that a consultant can be on more than one contract over time. Also, it's possible that a consultant may not have ever taken a contract (perhaps they just started employment). So we would likely rewrite this rule as follows.

Each consultant may be on many contracts; each contract has consultants.

We can break down the constraints as follows.

From the perspective of Consultant, it may or may not participate in the relationship (optional participation).
From the perspective of Consultant, it may be associated to many Contracts (plural).
From the perspective of Contract, it must participate in the relationship (mandatory participation).
From the perspective of Contract, it may be associated to many Consultants (plural).

Structural Database Rules: Major Points

You have reviewed a lot of material in a short time. Whew! So you can keep it all straight, let's review the major points.

- Every organization operates according to a set of rules, and we start the process of creating our application by analyzing the organization and its processes.
- We formerly model the organization and its processes by creating business rules, each of which defines or constrains an aspect of the organization's structure or process, and other artifacts.
- We design each application component to support the business rules, which for the database means creating structural database rules.
- Each structural database rule defines two entities, a relationship, participation constraints from the perspective of both entities, and plurality constraints from the perspective of both entities.
- An entity is a blueprint for a data set which defines the attributes for each data item in the set.
- A relationship is an association between two entities.
- A participation constraint indicates whether or not each data time must participate in the relationship.
- A plurality constraint indicates whether each data item can be associated to only one, or more than one, data items.
- We can phrase structural database rules them in many different ways, but the entities, relationship, and constraints must be entirely unambiguous.

Structural Database Rules For Your Database

Now armed with the right knowledge and examples, you are ready to tackle creating your own structural database rules. The significance of these rules cannot be overstated. *The fundamentals you define in your structural database rules – significant entities, relationships, participation constraints, and plurality constraints – will determine the structure of your database throughout all phases of development.* The database design defined by the structural database rules have significant impact on the performance, implementation, and use of the database for as long as it is in use, even if it is used for the next 20 years! This is why it's critical to understand and correctly make use of these fundamentals.

Not surprisingly, you will create the structural database rules based upon use cases recall that a use case is a list of steps defining interactions between a person and the system. Although not a strict requirement, you may find it helpful to also create a list of business rules for the target organizations or clients for which you are creating the system and database. Business rules may help you define your system use cases, and may help you understand what significant entities and relationships should exist for your database. Although the focus of this course is not business or technical analysis, the truth is that any good database designer needs to know the business, or else they cannot create a useful database design to be used by the business.

Create a list of structural database rules for all significant entities and relationships, with the constraints defined.

Let's take a look at this process for TrackMyBuys.

TrackMyBuys Structural Rules

I'll start with the first use case.

Account Signup/Installation Use Case

1. The person visits TrackMyBuys' website or app store and installs the application.
2. The application asks them to create an account when its first run.
3. The user enters their information and the account is created in the database.

The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them.

There are a several components in play for this use case, the application, the person using the application, and the database. However, I am focused on what the database needs to store for this project so I am focusing on the structural database rules. From step #3, I see one entity – Account. In looking through the other steps, I do not see any other entity or relationship. I could attempt to break down Account into multiple entities with relationships, but from this use case alone I don't have enough information. Instead, I will keep in mind the Account entity is needed for the database, and move on to the next use case.

Automatic Purchase Tracking Use Case

1. The person visits an online retailer and makes a purchase.
2. The TrackMyBuys browser plugin detects that the purchase is made, and records the relevant information in the database such as the purchase date, price, product, store, etc …

From this use case, I see three significant data points, Product, Store, and Purchase. TrackMyBuys of course tracks which products are purchased, and for which stores. Though not explicitly mentioned, it stands to reason that each Purchase is associated with an Account, since we need to store who is actually making the purchase. Recall that we previously discovered the Account entity from the first use case, Account Signup/Installation. I now have enough information to create some structural database rules. I'll number them so that they can later be referred to by number.

1. Each purchase is associated with an account; each account may be associated with many purchases.

I create this structural rule because I infer from the use cases that each purchase is associated with an account, and of course, since each person can make many purchases, an account can be associated with many purchases. I indicate that account to purchase is optional (by stating "each account *may*…") to leave room for the fact that an account is created when the application is first run, before any purchases are made. It's well possible that an account exists in the database without any purchase being made, if someone installs the application but does not record any purchases.

Here's a second rule.

2. Each purchase is made from a store; each store has one to many purchases.

This rule indicates that every purchase is tied to a specific store, and that of course, each store could have many purchases associated to it, since a person can buy multiple things from the same store. I make store to purchase mandatory since a store won't be stored in my database unless a purchase is made from it. Note that future analysis could change store to purchase to optional, if for example we end up having a separate screen where people enter a list stores they might buy from before they make any purchase. But for now, it is mandatory.
Here's a third rule.

3. Each purchase has one or more products; each product is associated with one to many purchases.

This rule indicates that every purchase can have many products associated with it, since a person could purchase many things at once. And of course, each product can be associated with many purchases since the same product could be purchased multiple times. I make purchase to product mandatory since TrackMyBuys will not record a purchase unless it is made, and a purchase cannot be made without at least one product. I also make product to purchase mandatory since the application doesn't record products until they are purchased based upon the use case. Just as with rule #2, if future analysis gives us a separate screen where people can enter lists of products before any purchases are made, we might change this to optional.

So far, we've create three rules for four entities. I don't see any other structural database rules emerging from the first two use cases, so I'll take a look at the third use case.

Purchase Lookup Use Case
1. The person signs into TrackMyBuys.
2. The person selects the option to lookup past purchases.
3. TrackMyBuys pulls a short list of recent purchases from the database, and also gives the user the option to search.
4. The user searches based upon a date range, which causes a database search.
5. The application pulls all purchases matching the criteria from the database.
6. The user selects the purchase they are interested in.
7. TrackMyBuys displays all recorded information about the purchase.

This use case is interesting in particular because, although it describes functionality of the application, it does not describe any additional data that must be stored in the database. This search feature would search over the Product, Purchase, Account, and Store entities we defined from the other two use cases. We do not need additional entities in order to support this search feature.

So from the three use cases I have thus far, I have these three structural database rules.

1. Each purchase is associated with an account; each account may be associated with many purchases.
2. Each purchase is made from a store; each store has one to many purchases.
3. Each purchase has one or more products; each product is associated with one to many purchases.

You have now seen the process I went through in creating the structural database rules from the use cases, which should give you an idea of how to go through this process. It's worth mentioning several important points. First, *you need to focus on what the database must store, that is, what data must be durable*. Use cases will describe many components such as the people, the application, and the database, and you need to wade through that and find the durable data. You then describe that durable data in the form structural database rules with entities, relationships, and constraints. For example, for an application to display a screen, it would contain screen components such as textboxes, checkboxes, drop-down lists, and the like, and none of these need to be durably stored in the database. The data displayed in them may need to be durable, however, and that is what you would store in your database. As another example, a person may use the application to search for information, so the searchable data must be durable, but the process of searching itself, including how the person interacts with the application screen, is not durable.

Second, *the more you know about the system and how it will be used, the more accurately you can identify the entities and relationships.* As you observed in my process of working through the use cases, the decisions of what entities and relationships we need all center around what the system needs and what the people who use the system needs. We are constantly asking ourselves what they need in order to create useful entities and relationships. Some entities just emerge from the core business. For example, a human resources system must contain an employee entity since one major purpose of the system is to track employees. Some entities may be present as a "nice to have" feature of the system. For example, most any system may have user preferences such as what color of screen they like, or what font size they like, and these may need to be stored in the database. So we observe that the clients'/organizations' operations define the core entities and relationships, and system-specific features also define some entities and relationships.

Third, *there is not a single "correct" solution, but there are solutions that capture the data needed by the system better than others.* The process of creating structural database rules has us asking many questions and there are many tradeoffs in this process. Should we make this relationship optional or mandatory? What are the core and nice-to-have entities? How are they related? Given the same use cases and requirements, two different people skilled in database design would likely create two different database designs that both support the data and relationships well. They would not create identical solutions. In this course, you need to take off the "is it correct" hat and put on the "does it meet the needs well" hat.

Entity-Relationship Diagramming

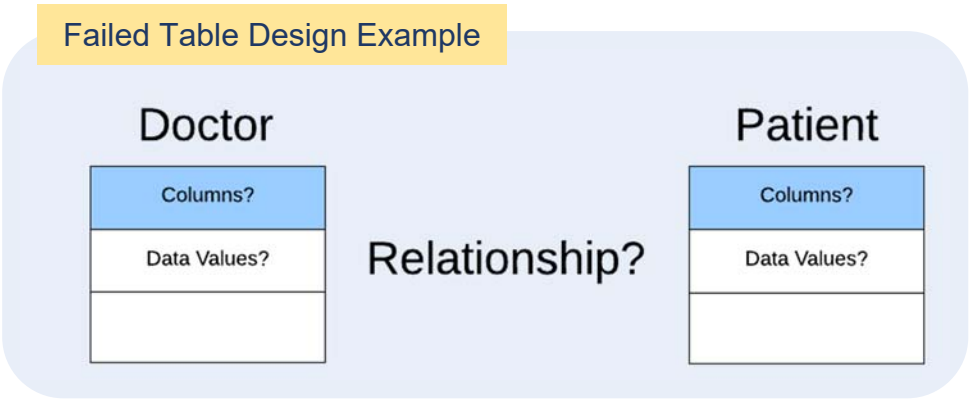
Structural database rules are powerful, but alone are not expressive enough to model real-world database schemas. You need to know more before you can design and implement your database with excellence. Entity-relationship diagramming which is the universally accepted method of visualizing

database schemas. Designing your database properly with these tools is critical to creating excellent, scalable, robust database designs.

While the relational model and relational databases have many strengths, one if their most significant and well-known weaknesses is the inability to convey real-world semantics (i.e. real-world meaning). It is critical that we are able to consistently map the entities and relationships that exist in our minds, the ones discovered during analysis of the organization and how the database will be used, into a solid database design. Unfortunately, it is too much of a leap to directly map our mental data model into the relational model.

The understructure of the relational model and relational databases is the table – a two-dimensional structure consisting of rows and columns. The structure of an individual table is relatively easy to understand and work with compared to many other data representation structures. However, data spread across many tables does not map particularly well to how we envision entities and relationships in our minds. For example, imagine during analysis we came up with the following structural database rule, "A doctor sees many patients; each patient may be seen by many doctors." We learned in a prior iteration that two entities are evident – Doctor and Patient – and that there is a "sees" relationship between them. How would we represent these as tables? We could create two tables, one for each entity, but that leaves us with two fundamental problems. The first is we want to initially represent these entities without describing their attributes and data values, and tables leave us no room to do so. The second is that we are not sure how we would structure the relationship between them exactly.

A failed attempt at using tables for the Doctor-Patient structural rule is below.



The two problems previously identified become immediately evident. We don't need a table with columns and values in it and this point in the design; rather, we need to represent just the entities themselves. But columnless tables do not exist. A problem tied to this is that tables are expected to contain data, but we don't have data at this point in the design. Second, tables do not natively support relationships, so how do we represent the "sees" relationship in the rule?
Entity-relationship diagrams (ERDs) were created to address these exact problems, and have become the universally accepted method of modeling and visualizing database designs. ERDs and the underlying entity-relationship model were first envisioned and documented by Peter Chen in the late 1970s. Recognizing the aforementioned issues, he defined ERDs to support multiple levels of data representation. The first level allows us to model entities and relationships as they exist in our mind without the need to define the attributes and data values. The higher levels allow us to model the attributes specifically for the relational model. He also defined the ERDs to support explicit presentation

of relationships with specialized symbols to describe the relationships' constraints. Chen envisioned that ERDs would be used to model the data, and then the relational database representation would be derived from the ERD (Chen, 1976). This approach has proven to be successful since this is the way we design our schemas today.

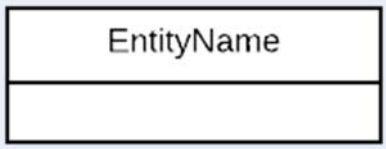
To be sure, there have been enhancements and adjustments to ERDs over the years by various people, so that ERDs we use today use almost all of Chen's concepts but look different than he originally envisioned. While Chen is credited for defining and popularizing the ERD concepts, Chen's visual representations require a lot of space and have some limitations; other ERD styles were created to help address this, including the important Crow's Foot Style (also known as Information Engineering Style), and modified UML class diagram representation. These other representations keep almost all of the same base concepts defined by Chen but use different visual representations. One exception is that Chen defined relationships to support attributes, while both Crow's Foot Style and modified UML class diagram representation do not allow relationships to support attributes. Relationships must be reified as entities in order to contain attributes. This works better for mapping ERDs to the relational model (this will become clearer to you when you start creating ERDs).

So how does all of this apply to your project? To design relational databases in general and to understand database articles and texts, it is important that you understand ERD concepts, as well as Crow's Foot Style and modified UML class diagram representation (many texts use Crow's Foot and many other texts use modified UML class diagram representation). Don't worry. We will describe the critical concepts you need to know in this document. In this and future iterations, I'll refer to these with shortened descriptions, Crow's Foot and UML, respectively, to reduce the cognitive noise produced by listing out their proper names.

Entity Representation

Recall that an entity is a blueprint for a data set, which defines the name, attributes, allowable values, and whether or not the attribute is required or optional for the dataset. You learned about entities in Term Project Iteration 2 in order to create structural database rules; now you know where the idea of an entity for data modeling comes from – Chen's and others' definitions of ERDs. An entity is represented diagrammatically as a rectangle with the entity's name inside of a subrectangle at the top, as illustrated in the following image.

Diagrammatic Representation of Entity



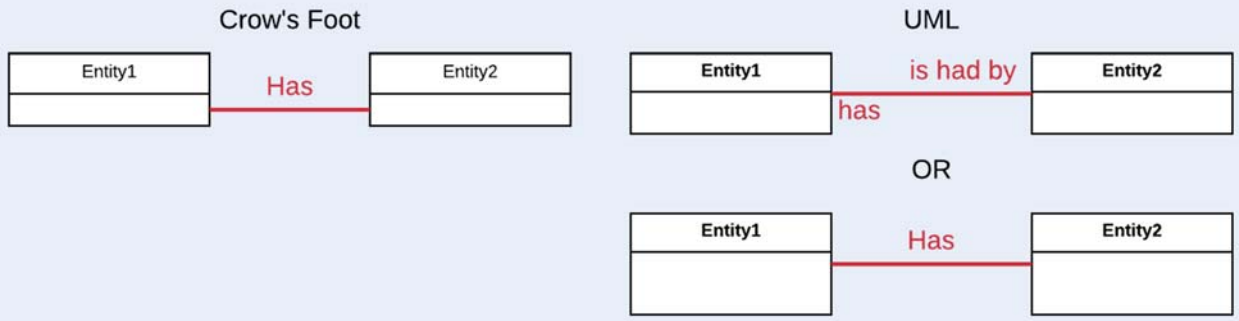
The basic entity with no attributes is diagrammed the same in both Crow's Foot and UML.

Relationship Representation

The existence of a relationship between two entities is diagrammed with a line between the two related entities. In Crow's Foot, the relationship is identified by a verb on the line, while in UML may be

identified in the same way (a single verb) or by two verbs – one for each entity's role. This is illustrated in the following image.

Diagrammatic Representation of Relationship Existence

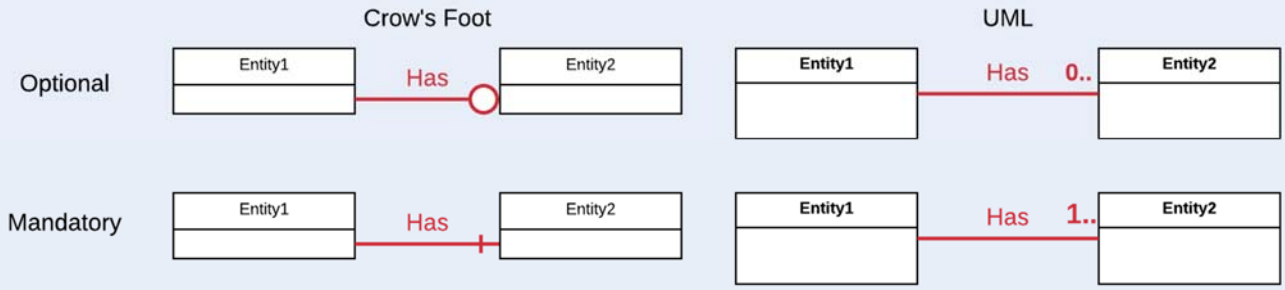


Notice that both Crow's Foot and UML have the option of using a single verb for the relationship, but only UML has the option of using two verbs based upon the relationship role.

Now you have seen diagrammatic indications that there is a relationship between two entities, which is important, but not yet complete. As you may recall, participation and plurality constraints are an integral part of each relationship. ERDs provide us with a diagrammatic means of representing each constraint.

Recall that a participation constraint indicates whether or not every item in the data set must participate in the relationship, and there are two options – optional and mandatory. If the relationship is optional to an entity, each data item (entity instance) may or may not participate in the relationship. If the relationship is mandatory, every data item must participate. The symbols for participation are illustrated in the following figure.

Diagrammatic Representation of Participation

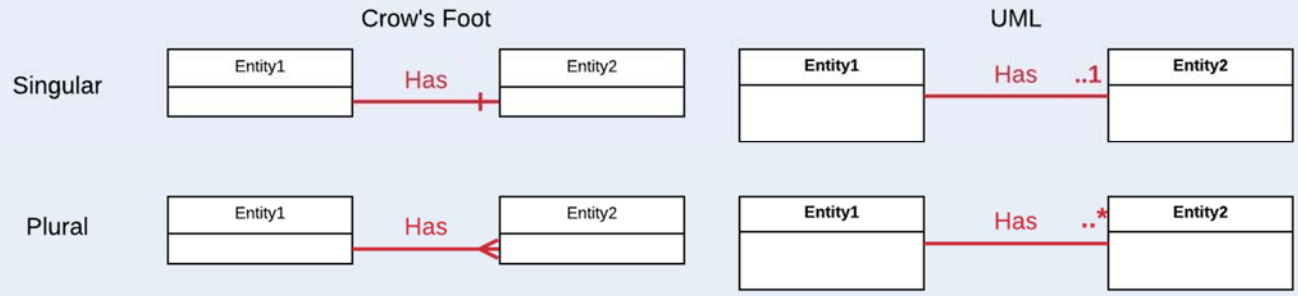


Crow's Foot makes use of a circle near the end of the line to indicate that the relationship is optional, and a single bar to indicate the relationship is mandatory. UML makes use of a 0 followed by two dots to indicate the relationship is optional, and a 1 to indicate it's mandatory. Of course, this is based upon perspective. The above diagram is indicating that the relationship is optional or mandatory to Entity1,

not Entity2. This may not be intuitive since the symbols are located close to Entity2, so please review this carefully to make sure you know which side of the line to put symbols on.

Recall that plurality indicates whether or not each data item can be associated with at most one, or more than one, other data items. There are two options – singular and plural – and singular means that each data item is associated with at most one, and plural means each data item can be associated with more than one. Plurality is represented as indicated in the following figure.

Diagrammatic Representation of Plurality



Crow's Foot make use of a single bar to indicate that the relationship is singular, and a crow's foot symbol (which looks similar to the end of a fork) to indicate the relationship is plural. UML uses a 1 on other side of the two dots to indicate the relationship is singular, and a "*" to indicate it's plural. For both Crow's Foot and UML, this diagram is indicating plurality from the perspective of Entity1.

You have now seen the symbols displayed individually and explained, which is useful for your understanding. However, a complete ERD displays the participation and plurality constraints from the perspective of both entities all in one diagram. Let's take a look at some examples.

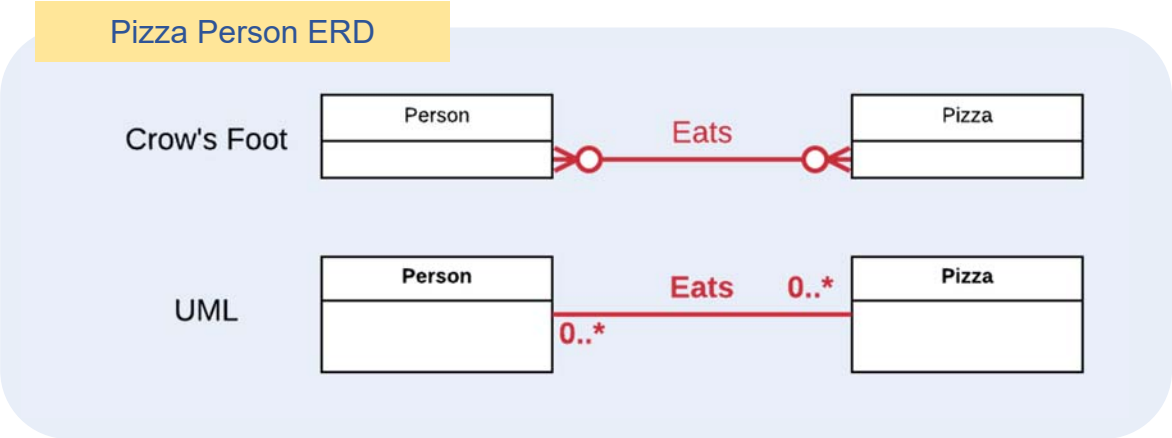
Let's start with the structural database rule "Each restaurant offers one or more menus; each menu is offered at a restaurant." We see two entities – Restaurant and Menu – and a relationship of "offers". We observe that the relationship is mandatory to both entities, because there is no indication that either is optional. The rule indicates every restaurant has a menu and every menu is associated to a restaurant. We also observe that the relationship is plural to Restaurant, because one *or more* menus may be offered at one restaurant. The relationship is singular to Menu, because each menu is offered at only one restaurant. Armed with this knowledge, we can now create the associated ERD, illustrated below.

Restaurant Menu ERD

Now you see a more complete picture, with the relationship line and constraint symbols on both sides pf the relationship. Notice that from the perspective of Restaurant, we use a bar followed by the Crow's Foot symbol to indicate the relationship is mandatory-plural to Restaurant. In UML we use "1..*" to indicate the same. From the perspective of Menu, we use a bar and another bar to indicate the relationship is mandatory-singular to Menu. In UML we use "1..1" to indicate the same. The ERD shown above is the diagrammatic representation of the structural database rule "Each restaurant offers one or more menus; each menu is offered at a restaurant."

Let's take a look at one more example, "A person may eat many pizzas; each pizza may be eaten by many people." We identify Person and Pizza as the entities, and Eat as the relationship. When we see "A person *may* eat", we know that the relationship is optional to person, that is, any particular person may or may not eat any pizza. When we see "A person may eat *many* pizzas", we know that the relationship is plural to person, that is, that each person may be associated to many pizzas (some people are pizzaholics and eat pizza multiple times a week). Likewise, when we see "Each pizza *may be* eaten", we know that the relationship is optional to pizza, that is, any particular pizza may or may not be eaten. For example, perhaps a pizza chain overestimated the number of pizzas and some were never eaten at the end of the day. When we see "Each pizza may be eaten by *many* people", we know that the relationship is plural to Pizza, that is, that one pizza can be associated to many people.

The following image is the ERD for this structural database rule.



From the perspective of Person, we use the circle combined with the Crow's Foot symbol to indicate that the relationship is optional and plural to Person. From the perspective we Pizza, we use the same circle and Crow's Foot combination to indicate that the relationship is optional and plural to Pizza. We use "0..*" on both sides of the relationship for UML to indicate the same, that both perspectives are optional and plural.

There is one more detail worth pointing out. For Crow's Foot style, the plurality constraint symbols are always the symbols immediately adjacent to the entities, while the participation constraints are on the outside. You'll notice that the Crow's Feet on both sides are immediately adjacent to Person and Pizza. The circle indicating the relationship is optional are on the inside, not immediately next to the entity.

Creating an Initial ERD

You have now learned the concepts and principles needed to create an ERD. You still need to know how use a tool to actually create the diagram, which is beyond the scope of this document. You will need to

rely on other resources such as your instructor, facilitator, live classrooms, and YouTube videos to learn how to use the tool. The tool I recommend is LucidChart (https://lucidchart.com) because it works from most any operating system, does not require an install, and allows you to create diagrams more quickly and with less effort than other tools I have seen. You can sign up for a free educational account using your BU email address to access all of its features.

However, *no diagramming tool is mandated for this course.* You are welcome to use Microsoft Visio Pro, OmniGraffle, or any of the other capable drawing tools available. My recommendation is, if you are already familiar with a drawing tool and it will support you creating ERDs, go ahead and use it; otherwise, use LucidChart. Your facilitator or instructor will *not* be evaluating your use of any particular tool; rather, they will be evaluating the ERDS produced by the tool as images embedded in your document.

Create an ERD from the structural database rules you have thus far in your project, making sure to correctly reflect the entities, relationships, and constraints.

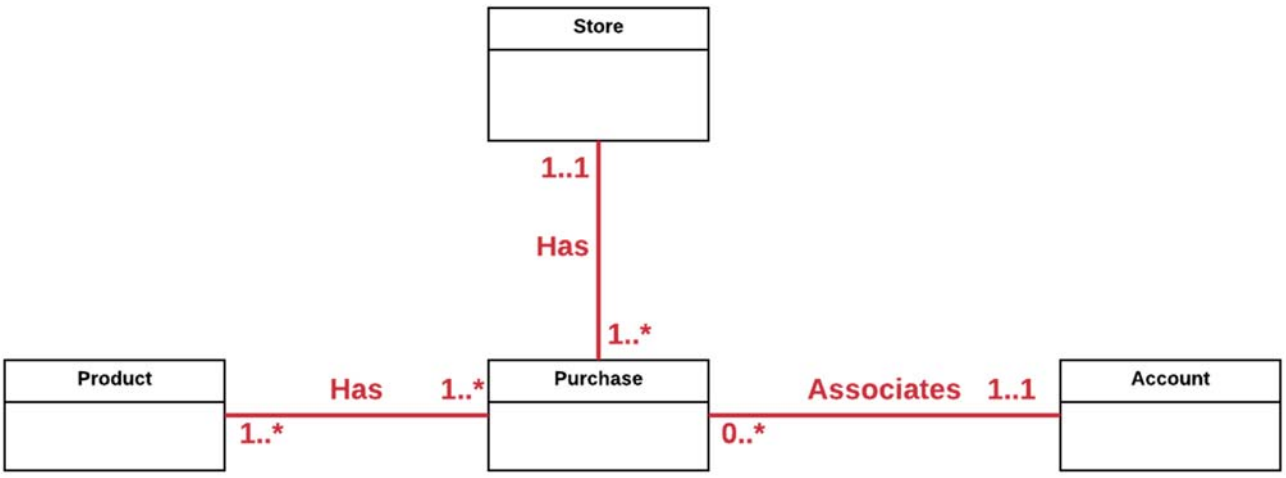
Let's take a look at how I map my TrackMyBuys structural database rules from Iteration 2 into an ERD using LucidChart.

Initial TrackMyBuys ERD

Here are the structural database rules I came up with in Iteration 2. I haven't changed or added to these at this point, so I will simply re-list them.

1. Each purchase is associated with an account; each account may be associated with many purchases.
2. Each purchase is made from a store; each store has one to many purchases.
3. Each purchase has one or more products; each product is associated with one to many purchases.

Here is the ERD I came up with for these rules, using UML since it's the most familiar to everyone, including those who do not know about database design.



The Purchase entity is associated with all three of the other entities, just as it was one of the entities in all three structural database rules. The participation and plurality constraints reflect what is in each of the structural database rules. For example, the "1..1" between Purchase and Account indicates that each Purchase must be associated to an account. Likewise, the "0..*" indicates that an Account may be associated with no purchases, or many purchases. This reflects the business rules

Hopefully, how structural database rules are mapped to ERDs, and what the ERD symbols mean, is becoming clearer. If not, please review the Restaurant Menu, Pizza Person, and TrackMyBuys examples again, as this foundational material is essential for your database design skills. If you understand how structural database rules map to ERDs, and you can create correctly formed structural database rules that are suitable for your database design, you have conquered the most difficult parts of database design.

Summary and Reflection

This week you have explored some additional aspects of your system and database, especially in a more formal way by creating structural database rules and an ERD. You and others can now more formally understand and visualize your database design. This is great! Update your project summary to reflect your new work.

Trying out these skills may have left you with some questions, concerns, or observations. Write these down so that you and your facilitator or instructor are aware of them and can communicate about them. If you feel confident things are headed in the right direction and don't have any questions, let us know that here as well.

Here is an updated summary as well as some observations I have about my progress on TrackMyBuys for this iteration.

TrackMyBuys Summary and Reflection

My database is for a mobile app named TrackMyBuys which records purchases made across all stores, making it the one stop for any purchase history. Typically, when a person purchases something, they can only see their purchase history with that same vendor, and TrackMyBuys seeks to provide a single interface for all purchases. The database must support a person entering, searching, and even analyzing their purchases across all stores.

The structural database rules and ERD for my database design contain the important entities of Store, Product, Purchase, and Account, as well as relationships between them.

I was surprised that though I had three decent size use cases, I only ended up three structural database rules. This exercise has shown me that since the use cases are focused on the system, the people using the system, and the database, even a large use case can result in a small number of entities to be stored in the database. I suspect that the reverse is also true, that a small use case could end up requiring many entities to be stored.

I am thankful that I did not need to create too many structural database rules this week, to give room for future growth without too much complexity.

Items to Submit

In summary, for this iteration, you update your design document by revising sections from iteration 1, and adding structural database rules and an initial ERD. Your design document will contain the following items, with items new to this iteration highlighted.

Items	Description
Project Direction Overview	Revise (if necessary) your overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it.
Use Cases and Fields	Revise (if necessary) your use cases that enumerate steps of how the database will be typically used, and the significant database fields needed to support the use case.
Structural Database Rules	Provide a list of structural database rules for all significant entities and relationships, with the constraints defined, based upon the use cases you defined.
Entity-relationship diagram	Provide an initial ERD from the structural database rules.
Summary and Reflection	Revise the concise summary of your project and the work you have completed thus far, and your questions, concerns, and observations.

Evaluation

Your iteration will be reviewed by your facilitator or instructor with the following criteria and grade breakdown.

Criterion	A	B	C	D	F	Letter Grade
Technical mastery (50%)	Evidence of excellent mastery throughout	Evidence of good mastery throughout	Evidence of basic mastery throughout or good mastery intermittently	Minimal mastery evidenced	Virtually no mastery evidenced	
Depth and thoroughness of coverage (25%)	Excellent depth and coverage of significant topics and issues	Good depth and coverage of significant topics and issues	Basic depth and coverage of significant topics and issues	Minimal depth and coverage of significant topics and issues	Virtually no depth and coverage of significant topics and issues	
Clarity in presentation (25%)	Ideas and designs are exceptionally clear and organized throughout	Ideas and designs are clear and organized throughout	Ideas and designs are somewhat clear and organized throughout	Ideas and designs are mostly obscure and disorganized	Ideas and designs are entirely obscure and disorganized	
					Assignment Grade:	#N/A
The resulting grade is calculated as a weighted average as listed using A+=100, A=96, A-=92, B+=88, B=85, B-=82 etc.						
To obtain an A grade for the course, your weighted average should be >=95, A- >=90, B+ >=87, B >= 82, B- >= 80 etc.						

Use the **Ask the Facilitators Discussion Forum** if you have any questions regarding how to approach this iteration. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.

Works Cited

Chen, P. (1976). The Entity Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems, 1.*