Part One: The Subscription Services

Highlighting Scheme:

- entities will be highlighted in pink
- attributes will be highlighted in lime
- relationships will be written in blue
- constraints will be highlighted in peach
- irrelevant information will be lightened

The proposed subscription model supports two disjoint types of membership accounts:

1. a personal account[1] held by a single private person (e.g., a community member interested in art); or
2. an organization account[2] (e.g., held by a local company or not-for-profit organization).

Organization accounts are managed by one-or-more representatives (with personal accounts) that can manage the organization account (e.g., borrow art pieces in the name of the organization). Furthermore, organization accounts have a type[3] (e.g., company, local government, not-for-profit, etc.). Members can log in to their account via their unique e-mail address and a password. According to the consultant, the password should be stored as a hashed value that incorporates a salt value.

Each account can hold a single subscription at any single time. That subscription can be a pre-paid subscription that allows the account to borrow a fixed number of art pieces or a long-term subscription. Long-term subscriptions belong to a tier that determines the subscription cost and how many art pieces can be borrowed simultaneously. The consultant noted that the diagram does not enforce that an account can only hold a single long-term subscription at any single time. For long-term subscriptions, a payment history covering each payment installment is maintained via the payment terms that have been paid (term one representing the first month of the subscription, term two the second month, and so on). For each payment term, the paying account[4] is also stored in case of a organization account (to keep track which representative paid). In addition, the system maintains a history of all subscriptions held by an account[5].

[1] ISA relationship with member
[2] ISA relationship with member
[3] potential check statement
[4] relationship between payment and personal account
[5] the history will be the table itself

I started with the **Member** entity because it is not a weak entity and does not partake in any one-to-many relationships. Translating this entity yields the following relational schema:

Member(mid: **INT**, email: **VARCHAR(200)**, passhash: **VARCHAR(200)**, name: **VARCHAR(100)**, address: **VARCHAR(200))**

The *email* and *passhash* attributes must be unique, and the *name* and *address* attributes cannot be null, which are all constraints. As such, I have chosen to use **VARCHAR** instead of **CLOB** because **CLOB** is not guaranteed to support these constraints. This is the same reasoning for the remaining **VARCHAR** variables. The *mid* attribute is an **INT** because it is most likely just going to be a unique generated integer ID. The **Member** schema translates to the following SQL statement:

```
CREATE TABLE member
    (
      mid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
      email VARCHAR(200) UNIQUE NOT NULL,
      passhash VARCHAR(200) UNIQUE NOT NULL,
      name VARCHAR(100) NOT NULL,
      address VARCHAR(200) NOT NULL
    );
```

Since the unique identifiers *mid* have no meaning outside of the database, I chose to always automatically generate the identifiers. This ensures that any rows that are added to the database will have a unique identifier. Next, I translated the **Personal** and **Organization** entity because they partake in an ISA relationship with the **Member** entity. In order to represent the different relationships that the **Personal** and **Organization** entities partake in, we need different tables, as such, the NULL method would not be as appropriate. Along with this, **Personal** does not have any additional attributes, so it would be hard to differentiate it with **Member** itself. From my understanding, a member can be a representative of an organizational account but will not have both a member account and an organizational account, which means that you would only have personal and organization as the possible combinations of entities. At this point, it would be easier to use the ER method. Translating the **Person** entity yields the following relational schema:

Person(pid: **INT**)

where the primary key *pid* is also a foreign key that refers to the primary key of **Member**. The *IsRep* relationship will later be implemented. This is because a person can be the representative of multiple organizations, but if we were to add a column *isRep* referencing the organization it is a representative of, it would become a one-to-one relationship instead of one-to-many. The **Person** schema translates to the following SQL statement:

```
CREATE TABLE person
    (
      pid INT NOT NULL PRIMARY KEY REFERENCES member(mid)
    );
```

Next, the **Organization** entity. I use the ER method for this ISA hierarchy for the same reason mentioned above. Translating this entity yields the following relational schema:

Organization(oid: **INT**, main_rep: **INT**, type: **VARCHAR(50)**)

where the primary key *oid* is also a foreign key that refers to the primary key of **Member**, and the *main_rep* attribute is a foreign key that refers to the primary key of **Person**. It should be noted that in order to represent the 'at least once' *IsRep* relationship, I added a main representative of each organization. This ensures that each organization has at least one representative (it will have the **NOT NULL** constraint in the SQL implementation). If the organization has one representative, then that representative will be the main one, and if there are many representatives then there will be one main one and the rest will be represented in the **IsRep** table. Along with this, there is a *type* attribute that describes the type of organization. Though I did not implement this, it should be noted that a check statement can be implemented so that the type of an organization is limited. The **Organization** schema then translates to the following SQL statement:

```
CREATE TABLE organization
    (
      oid INT NOT NULL PRIMARY KEY REFERENCES member(mid),
      main_rep INT NOT NULL REFERENCES person(pid),
      "type" VARCHAR(50) NOT NULL
    );
```

Next, we implement the **IsRep** relationship using a table. This is because we have a many-to-many relationship with the at-least-once constraint which we covered in our **Organization** table. Translating this entity yields the following relational schema:

IsRep(pid: **INT**, oid: **INT**)

where *pid* is a foreign key that refers to the primary key of **Person** and *oid* is a foreign key that refers to the primary key of **Organization**. The combination of *pid* and *oid* will be the primary key of **IsRep**. The **IsRep** schema then translates to the following SQL statement:

```
CREATE TABLE isRep
    (
      pid INT NOT NULL REFERENCES person(pid),
      oid INT NOT NULL REFERENCES organization(oid),
      PRIMARY KEY(pid, oid)
    );
```

Next, we implement the weak entity **Subscription** because the other entities rely on it. For this ISA hierarchy, using the NULL method would not be as appropriate because we need separate tables for each entity in order to implement the different relationships they partake in. I am assuming that a subscription can either be **PrePaid** or **LongTerm**, and not both. As such, though the OO method can be used, it would be easier to use the ER method. Translating the **Subscription** weak entity yields the following relational schema:

Subscription(mid: **INT**, start_year: **INT**, start_month: **VARCHAR(9)**)

where *mid* is a foreign key that refers to the primary key of **Member**. This is needed because **Subscription** is a weak entity to **Member**. It also has its partial keys *start_year* and *start_month*. *start_year* is an **INT** but will have a check that ensures it is greater than 0 (since years must be greater than 0). *start_month* is a **VARCHAR(9)** (9 characters because out of the 12 possible months, September is the longest with 9 characters). It will also have a check statement that checks for each possible month. The **Subscription** schema then translates to

the following SQL statement:

```
CREATE TABLE subscription
    (
      mid INT NOT NULL REFERENCES "MEMBER"(mid),
      start_year INT NOT NULL,
      CHECK (start_year > 0),
      start_month VARCHAR(9) NOT NULL,
      CHECK (start_month = 'January' OR start_month = 'February'
      OR start_month = 'March' OR start_month = 'April' OR start_month = 'May'
      OR start_month = 'June' OR start_month = 'July' OR start_month = 'August'
      OR start_month = 'September' OR start_month = 'October'
      OR start_month = 'November' OR start_month = 'December'),
      PRIMARY KEY (mid, start_year, start_month)
    );
```

It should be noted that *mid* is assumed to have the constraint **UNIQUE** since it is a primary key. Since it is unique, this means that a member can only have one subscription at a time, satisfying the 'exactly once' *PartOf* relationship. Next, we implement the entity **PrePaid** using the ER method for reasons stated above. Translating the **PrePaid** entity yields the following relational schema:

PrePaid(mid: **INT**, start_year: **INT** start_month: **VARCHAR(9)**,
charges: **VARCHAR(500)**)

where *mid*, *start_year* and *start_month* are all foreign keys that refers to the primary keys of **Subscription**. *charges* is a **VARCHAR(500)** so that it can store details regarding the payment (how much was paid, type of payment, card details etc). Once again, **CLOB** was not used because **NOT NULL** is a constraint that this attribute needs. The **PrePaid** schema then translates to the following SQL statement:

```
CREATE TABLE prepaid
    (
      mid INT NOT NULL,
      start_year INT NOT NULL,
      start_month VARCHAR(9) NOT NULL,
      charges VARCHAR(500) NOT NULL,
      FOREIGN KEY (mid, start_year, start_month)
      REFERENCES subscription,
      PRIMARY KEY (mid, start_year, start_month)
    );
```

Next, we implement the entity **LongTerm** using the ER method for reasons stated earlier. Translating the **LongTerm** entity yields the following relational schema:

LongTerm(mid: **INT**, start_year: **INT** start_month: **VARCHAR(9)**,
tier: **INT**)

where *mid*, *start_year* and *start_month* are all foreign keys that refers to the primary keys of **Subscription**. For *tier*, I am assuming that tiers will be ranked using an integer greater than

0. I am also assuming that every long term subscription is guaranteed to be paid. If it isn't then I would implement an attribute in **LongTerm** that directly references a member so that they can be held accountable. The **LongTerm** schema then translates to the following SQL statement:

```
CREATE TABLE longterm
    (
      mid INT NOT NULL,
      start_year INT NOT NULL,
      start_month VARCHAR(9) NOT NULL,
      tier INT NOT NULL,
      CHECK (tier > 0),
      FOREIGN KEY (mid, start_year, start_month)
      REFERENCES subscription,
      PRIMARY KEY (mid, start_year, start_month)
    );
```

Next, we implement the weak entity **Payment**. We use a table because we cannot simply use an attribute in **LongTerm** due to the fact that a single long term can have multiple payments (payment terms). Translating the **Payment** weak entity yields the following relational schema:

Payment(mid: **INT**, start_year: **INT** start_month: **VARCHAR(9)**,
term: **INT**, payment_terms: **VARCHAR(500)**, pay_by: **INT**)

where *mid*, *start_year* and *start_month* are all foreign keys that refers to the primary keys of **LongTerm**. I am referring to *term* as a specific integer that describes the specific term. *payment_terms* is a **VARCHAR(500)** that will hold the payment details (how much was paid, type of payment, card details etc). *pay_by* is also a foreign key that will refer to a *pid* in **Personal**. The foreign key *mid, start_year, start_month* will represent the *PayFor* relationship. The **Payment** schema then translates to the following SQL statement:

```
CREATE TABLE payment
    (
      mid INT NOT NULL,
      start_year INT NOT NULL,
      start_month VARCHAR(9) NOT NULL,
      term INT NOT NULL,
      CHECK (term > 0),
      payment_terms VARCHAR(500) NOT NULL,
      pay_by INT REFERENCES person(pid),
      FOREIGN KEY (mid, start_year, start_month),
      REFERENCES subscription,
      PRIMARY KEY (mid, start_year, start_month, term)
    );
```

It should be noted that term must be greater than 0 because in my opinion, term should always be positive. Along with this, since **Payment** partakes in a one-to-many *PayBy* relationship with **Personal**, *pay_by* does not have a **NOT NULL** constraint. If the arrow was curved, I would add **NOT NULL**.

Part Two: The Donation System

A *table* **DonorEvent**(*eid*, name, date, description, private):

We simply need to translate this schema into SQL. Since the description doesn't describe an identifier type for *eid*, we will automatically generate one. Since name and description do not seem to have constraints we can use **CLOB** instead of **VARCHAR**. We also assume that *private* is simply a boolean value that is true if it is private and false if it is public. We now have:

```
CREATE TABLE DonorEvent
    (
      eid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
      name CLOB NOT NULL,
      "DATE" DATE NOT NULL,
      description CLOB NOT NULL,
      private BOOLEAN NOT NULL
    );
```

A *table* **DonorInvite**(*eid*, *mid*, confirmed, amount, $\text{referredBy}_{\text{optional}}$)

Again, we translate this schema into SQL. I assume that *eid* references the *eid* from **DonorEvent**. Along with this, since *confirmed* was not defined, I will assume that it represents whether the member confirmed they are going, which can be represented using a **BOOLEAN**. I interpreted *referredBy* as a reference to another donor invite, however, following the schema, **DonorInvite** does not have its own separate identifier, so I will instead assume that *referredBy* refers to the member that referred them. Along with this, we have a check statement that essentially says, if you were referred, you cannot refer anyone (amount = 0) If you were not referred, by default, referredBy will be **NULL**. We now have:

```
CREATE TABLE DonorInvite
    (
      eid INT NOT NULL REFERENCES donorevent(eid),
      mid INT NOT NULL REFERENCES "MEMBER"(mid),
      confirmed BOOLEAN NOT NULL,
      amount INT NOT NULL,
      CHECK (amount >= 0),
      referredBy INT REFERENCES "MEMBER"(mid) DEFAULT NULL,
      CHECK ((referredBy IS NOT NULL AND amount = 0) OR (referredBy IS NULL))
    );
```

A *constraint* that the referrals for an event by a direct-invite member m do not exceed the amount of referrals allocated to m (as recorded in the **DonorInvite** table).

This constraint would need a multi-table constraint. We would need to do a select statement in **DonorInvite** to count how many referrals a member has, then use a check statement to check that it is less than there *amount* allowed to invite.
The consultant wants a table **donation** that stores, per donation, the event during which the donation was given, the member that donated, the amount donated, the person to which the donation is attributed (optional), and an attribution message (optional). The consultant had

difficulties coming up with a proper design for this table, but noted that people can make several donations during an event.

donation(eid: **INT**, donater: **INT** amount: **DECIMAL**,
donate_to: **INT**, message: **VARCHAR(500)**,)

It should be noted that assuming the donater donates a different amount, they can donate multiple times during one event.

```
CREATE TABLE donation
  (
   eid INT NOT NULL REFERENCES donorevent(eid),
   donater INT NOT NULL REFERENCES "MEMBER"(mid),
   amount DECIMAL NOT NULL,
   CHECK (amount >= 0),
   donate_to INT REFERENCES "MEMBER"(mid),
   message VARCHAR(500))
  );
```