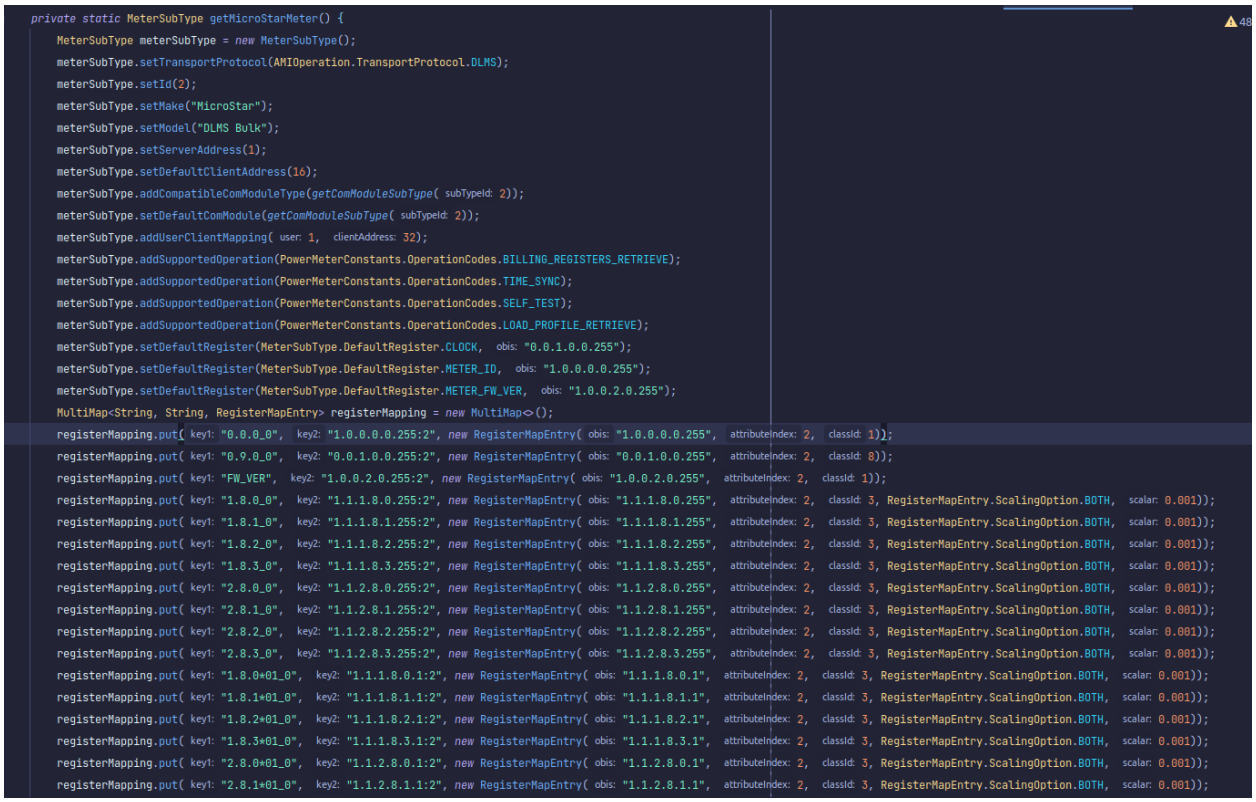
Implement Subtype Management Component

What is the Requirement?

Currently Power meter has a fixed set of meter subtypes,sim module subtypes & com module subtypes.(All of them are hard coded).
For example,In this Image we can see how we are currently storing the meter subtype by hard coding in power meter.We are storing all subtypes' data in the code in this manner.



```
private static MeterSubType getMicroStarMeter() {
    MeterSubType meterSubType = new MeterSubType();
    meterSubType.setTransportProtocol(AMIOperation.TransportProtocol.DLMS);
    meterSubType.setId(2);
    meterSubType.setMake("MicroStar");
    meterSubType.setModel("DLMS Bulk");
    meterSubType.setServerAddress(1);
    meterSubType.setDefaultClientAddress(16);
    meterSubType.addCompatibleComModuleType(getComModuleSubType( subTypeId: 2));
    meterSubType.setDefaultComModule(getComModuleSubType( subTypeId: 2));
    meterSubType.addUserClientMapping( user: 1,  clientAddress: 32);
    meterSubType.addSupportedOperation(PowerMeterConstants.OperationCodes.BILLING_REGISTERS_RETRIEVE);
    meterSubType.addSupportedOperation(PowerMeterConstants.OperationCodes.TIME_SYNC);
    meterSubType.addSupportedOperation(PowerMeterConstants.OperationCodes.SELF_TEST);
    meterSubType.addSupportedOperation(PowerMeterConstants.OperationCodes.LOAD_PROFILE_RETRIEVE);
    meterSubType.setDefaultRegister(MeterSubType.DefaultRegister.CLOCK,  obis: "0.0.1.0.0.255");
    meterSubType.setDefaultRegister(MeterSubType.DefaultRegister.METER_ID,  obis: "1.0.0.0.0.255");
    meterSubType.setDefaultRegister(MeterSubType.DefaultRegister.METER_FW_VER,  obis: "1.0.0.2.0.255");
    MultiMap<String, String, RegisterMapEntry> registerMapping = new MultiMap<>();
    registerMapping.put( key1: "0.0.0_0",  key2: "1.0.0.0.0.255:2", new RegisterMapEntry( obis: "1.0.0.0.0.255",  attributeIndex: 2,  classId: 1));
    registerMapping.put( key1: "0.9.0_0",  key2: "0.0.1.0.0.255:2", new RegisterMapEntry( obis: "0.0.1.0.0.255",  attributeIndex: 2,  classId: 8));
    registerMapping.put( key1: "FW_VER",  key2: "1.0.0.2.0.255:2", new RegisterMapEntry( obis: "1.0.0.2.0.255",  attributeIndex: 2,  classId: 1));
    registerMapping.put( key1: "1.8.0_0",  key2: "1.1.1.8.0.255:2", new RegisterMapEntry( obis: "1.1.1.8.0.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.1_0",  key2: "1.1.1.8.1.255:2", new RegisterMapEntry( obis: "1.1.1.8.1.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.2_0",  key2: "1.1.1.8.2.255:2", new RegisterMapEntry( obis: "1.1.1.8.2.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.3_0",  key2: "1.1.1.8.3.255:2", new RegisterMapEntry( obis: "1.1.1.8.3.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "2.8.0_0",  key2: "1.1.2.8.0.255:2", new RegisterMapEntry( obis: "1.1.2.8.0.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "2.8.1_0",  key2: "1.1.2.8.1.255:2", new RegisterMapEntry( obis: "1.1.2.8.1.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "2.8.2_0",  key2: "1.1.2.8.2.255:2", new RegisterMapEntry( obis: "1.1.2.8.2.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "2.8.3_0",  key2: "1.1.2.8.3.255:2", new RegisterMapEntry( obis: "1.1.2.8.3.255",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.0*01_0",  key2: "1.1.1.8.0.1:2", new RegisterMapEntry( obis: "1.1.1.8.0.1",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.1*01_0",  key2: "1.1.1.8.1.1:2", new RegisterMapEntry( obis: "1.1.1.8.1.1",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.2*01_0",  key2: "1.1.1.8.2.1:2", new RegisterMapEntry( obis: "1.1.1.8.2.1",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "1.8.3*01_0",  key2: "1.1.1.8.3.1:2", new RegisterMapEntry( obis: "1.1.1.8.3.1",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "2.8.0*01_0",  key2: "1.1.2.8.0.1:2", new RegisterMapEntry( obis: "1.1.2.8.0.1",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
    registerMapping.put( key1: "2.8.1*01_0",  key2: "1.1.2.8.1.1:2", new RegisterMapEntry( obis: "1.1.2.8.1.1",  attributeIndex: 2,  classId: 3, RegisterMapEntry.ScalingOption.BOTH,  scalar: 0.001));
```

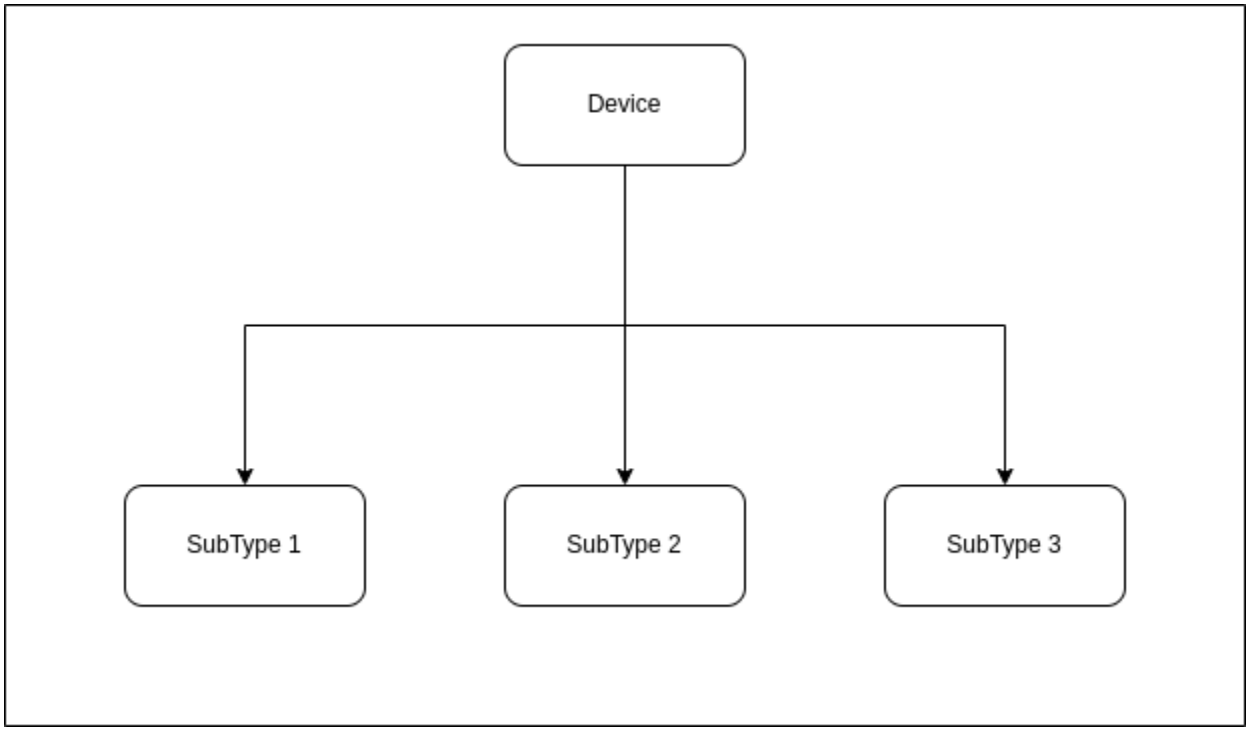
–Drawback: It will take longer if we add or modify a subtype because we have to change the hard-coded lines and redeploy it.

In order to do that, we need to implement dynamic meter subtypes, sim module subtypes & com module subtypes.That means It is required to have APIs to add / modify /retrieve device subtypes dynamically without requiring code changes.

–Advantage: when we want to add/modify/retrieve subtype data, we don’t need to change the lines that are hard-coded.For do that we will have API’s.

Why do we need Subtypes?

What are the subtypes?
We can classify specific devices into subtypes based on the device's properties and part of their operations.for example,



PowerMeter - In power meter, we have 3 main supported devices as supported Meter Types,supported modem types & supported sim types.

- Supported meter types (Meter Subtypes)
  We can classify meters into different meter subtypes according to the different communication protocol. for example meter subtypes,
  - Ante Single Phase
  - Ante Three Phase
  - Microstar p2000
  - Iskra IEC meters

- Supported communication modem types (Com Module Subtypes)
  We can classify communication modems into different communication modem subtypes as com module subtypes according to the variations on each modem types.for example com module subtypes,
  - ATx-Mega SIM800
  - Micostar M590

- Supported sim types (Sim Module Subtypes)
  We can classify sim types into sim module subtypes according to the differentiations of each sim providers.for example sim module subtypes,
    - Dialog
    - Mobitel

Switchgear - In Switchgear, we have other supported devices as Control Box device types,Operating Chamber device types.

- Control Box
  for example control box subtypes,
    - NOJA
    - ENSTRO
    - BH
    - PNC

- Operating Chamber
  for example operating chamber subtypes,
    - NOJA
    - ENSTRO
    - BH
    - PNC

Note: we can add this subtype mechanism for Android or IOS
(for example AndroidTv & Android Mobile are subtypes of Android devices because they have the same operating systems but there are some variances between them So we can add subtype for Android)

What is the Design Approach?

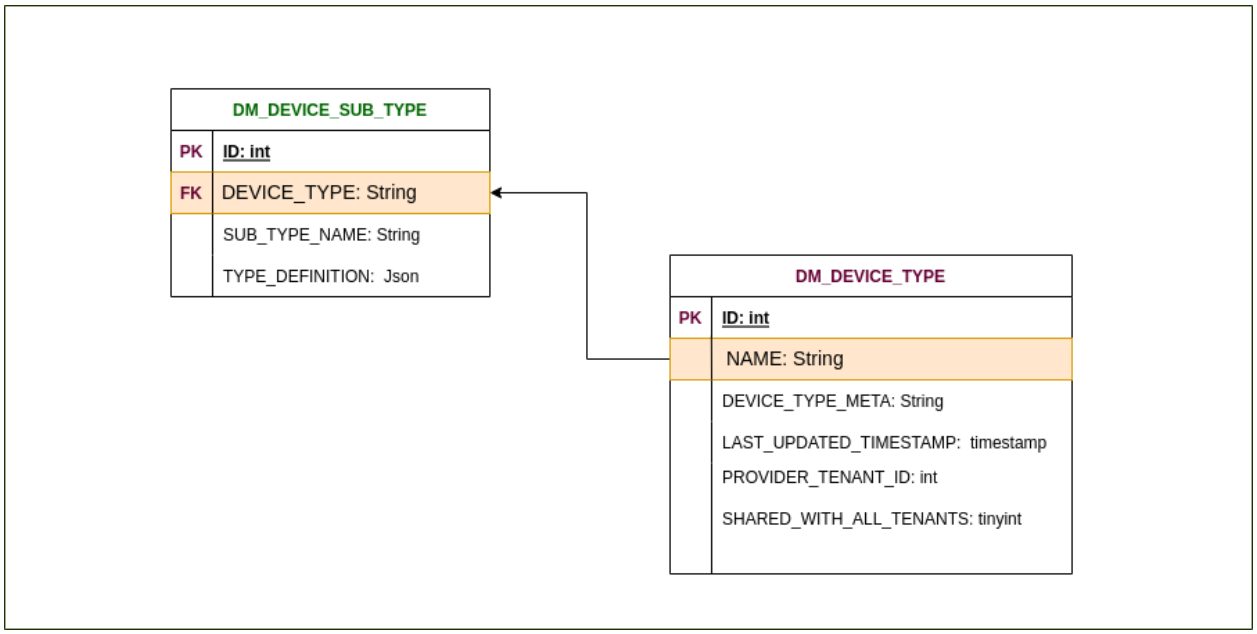
In Power Meter,

DM DEVICE table and DM DEVICE TYPE table are currently available, and they indicate if the device type is com/android/sim/meter. We must keep the device subtype separate per our criteria. However, we lack a subtype table.

We have three variants in our power meter use case: Sim module subtypes, Com module subtypes, and meter subtypes. These 3 subtypes have common fields as ID & TYPE_DEFINITION =>
(The TYPE DEFINITION field stores each subtype's details, features, and operations as a JSON String. Each subtype's TYPE DEFINITION is distinct from the others.)

Not only the power meter ,In this scenario we can use the TYPE_DEFINITION field for storing any subtype's specific data as JSON String.

Then we can create a DM_DEVICE_SUB_TYPE table for storing all subtypes data along with attributes like devicetype, subtype name, and type definition. It will come up below. This table can be used for device-mgt-core, switchgear, and not just power meters.

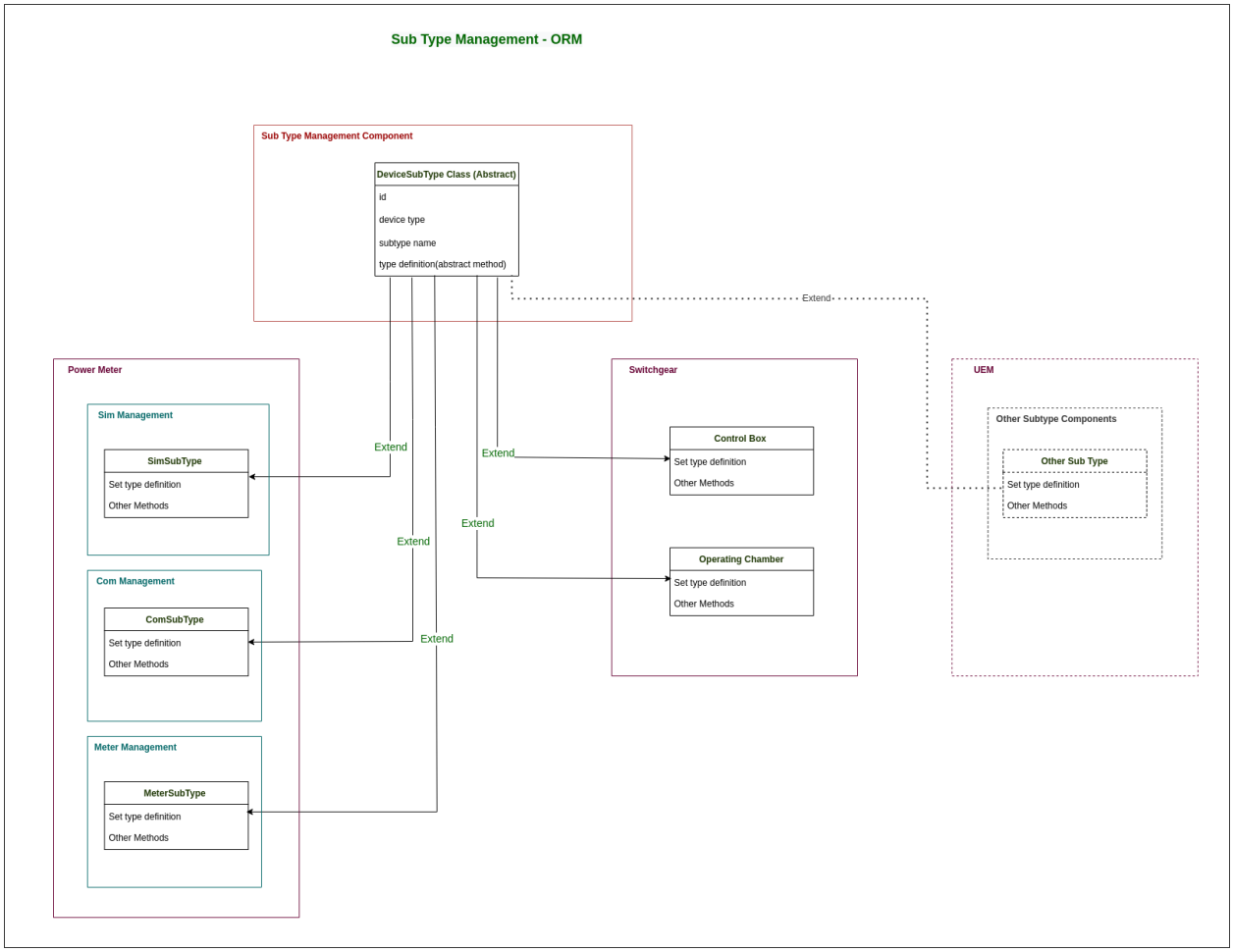


ORM:

Since subtypes have common attributes we can push these common attributes into the parent class. Let's introduce our parent class as **DeviceSubType.**It should be an abstract class.

So we can implement these generic fields with methods in DeviceSubType parent class.then if we want to implement API's for each subtype we can extend the parent DeviceSubType class & set TYPE_DEFINITION according to the specific subtype.

There are main generic things therefore we can separate out a component as a **subtype management component**.This component can be reused at platform level. We can move this into device-mgt-core. This is shown in the diagram below. Please refer to it.



Outcome:

Finally if we want to retrieve specific device subtypes (if it is sim /com /meter /switchgear-subtypes/Android etc) via the Subtype's API, It will give us by loading the DM_DEVICE_SUB_TYPE table.Not only retrieving method It also can applying for adding or modifying methods of subtypes.
This will be optimum by loading data from cacheloader. we will have to put it to use.