

"Give me a clean, minimal context summary I can paste into a new chat."

Say the word.



Never mind, we are not going to branch out in that case, and wait for my next instruction.

Understood. I'll stay in this thread and wait for your next instruction.



Okay, so now that I have told you your mistake and you have got a clear understanding, can you please now uh give me the updated code and the directory structure?

Good catch on the mistake — and yes, the correction is now **very clear**. I'll first **state the corrected understanding**, then give you the **updated directory structure and code skeleton**, aligned with everything you've clarified.

I will **not jump into the algorithm yet**, exactly as you asked. This is only the *fixed foundation*.

## 1 Corrected understanding (important)

### ✗ Earlier mistake (now fixed)

- Treating Package DTO as the root of the validation JSON ✗

### ✓ Correct structure (final, correct)

- Root of validation JSON →

json

```
{
  "entities": [ ... ]
}
```

Copy code

- Inside `entities[]`:
  1. Entity #1 → PACKAGE (DCDS package)
  2. Entity #2 → CLASS (Business / Logical / Technical rule)
  3. Entity #3+ → MEMBER entities (each field definition)

So:

- Entities are peers
- Entity type + `metadataRecordType` defines role
- Dimensions are inside each entity, not global

This aligns perfectly with the `DcdsEntityDto` you shared.

---

## 2 What the validation JSON really represents

Each `entity` contains:

- `masterRef`
- `originData`
- `masterData`
- `dimensions[]` THIS is where validation rules live
- `attributes[]`
- `relations[]`

And inside `dimensions[]`:

- `dimensionType = data-type-info` → validation rules
- `dimensionType = name` → DISPLAY-NAME, LOGICAL-NAME
- `dataType`, `fieldFormat`, `fieldIsNullable`, etc.

So validation logic must be built **entity-by-entity**, not package-first.

---

## 3 Updated directory structure (finalized & correct)

This structure:

- Supports 6 schemas
- Supports future API replacement

- Uses DTOs via dependency
- Keeps validation logic independent

text

```
src/main/java
└── com.yourorg.dq.validation
    ├── config
    │   └── ValidationSchemaConfig.java
    │
    ├── loader
    │   ├── ValidationSchemaLoader.java
    │   ├── StaticJsonSchemaLoader.java
    │   └── ApiSchemaLoader.java      (future)
    │
    ├── model
    │   ├── ValidationSchema.java
    │   ├── EntityValidationModel.java
    │   ├── FieldValidationRule.java
    │   └── RuleScope.java
    │
    ├── registry
    │   └── ValidationSchemaRegistry.java
    │
    ├── service
    │   └── ValidationEngine.java
    │
    └── util
        └── EntityClassifier.java
```

 Copy code

text

```
src/main/resources
└── validation
    ├── business-rule-insert.json
    ├── business-rule-update.json
    ├── logical-rule-insert.json
    ├── logical-rule-update.json
    ├── technical-rule-insert.json
    └── technical-rule-update.json
```

 Copy code

## 4 Root loader code (updated)

### ValidationSchemaLoader (strategy)

java

```
public interface ValidationSchemaLoader {
    ValidationSchema load(RuleScope scope);
}
```

 Copy code

### StaticJsonSchemaLoader (CURRENT)

java

```
@Component
public class StaticJsonSchemaLoader implements ValidationSchemaLoader {

    private final ObjectMapper objectMapper;

    public StaticJsonSchemaLoader(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    @Override
    public ValidationSchema load(RuleScope scope) {
        String path = "/validation/" + scope.fileName();

        try (InputStream is = getClass().getResourceAsStream(path)) {
            DcdsEntityContainerDto container =
                objectMapper.readValue(is, DcdsEntityContainerDto.class);

            return ValidationSchema.from(container.entities());
        } catch (IOException e) {
            throw new IllegalStateException("Failed to load schema: " + scope, e)
        }
    }
}
```

 Copy code

**Note:** `DcdsEntityContainerDto` is the true root DTO:

java

```
public record DcdsEntityContainerDto(  
    List<@Valid DcdsEntityDto> entities  
) {}
```

Copy code

## 5 Registry (loaded once at startup)

java

```
@Component  
public class ValidationSchemaRegistry {  
  
    private final Map<RuleScope, ValidationSchema> schemas = new EnumMap<>(RuleSc  
  
    public ValidationSchemaRegistry(ValidationSchemaLoader loader) {  
        for (RuleScope scope : RuleScope.values()) {  
            schemas.put(scope, loader.load(scope));  
        }  
    }  
  
    public ValidationSchema get(RuleScope scope) {  
        return schemas.get(scope);  
    }  
}
```

Copy code

Loaded at application startup, cached in memory.

## 6 Internal model (key fix)

### ValidationSchema

java

```
public class ValidationSchema {  
  
    private final Map<String, EntityValidationModel> entitiesByLogicalName;  
  
    private ValidationSchema(Map<String, EntityValidationModel> map) {  
        this.entitiesByLogicalName = map;  
    }
```

```

public static ValidationSchema from(List<DcdsEntityDto> entities) {
    Map<String, EntityValidationModel> map = new HashMap<>();

    for (DcdsEntityDto entity : entities) {
        EntityValidationModel model = EntityValidationModel.from(entity);
        map.put(model.logicalName(), model);
    }

    return new ValidationSchema(map);
}
}

```

 Copy code

## EntityValidationModel (corrected)

java

```

public record EntityValidationModel(
    String logicalName,
    Map<String, FieldValidationRule> fields
) {

    public static EntityValidationModel from(DcdsEntityDto entity) {
        String logicalName = EntityClassifier.logicalName(entity);

        Map<String, FieldValidationRule> fields =
            entity.dimensions().stream()
                .filter(d -> d instanceof DcdsDataTypeInfoDto)
                .map(FieldValidationRule::from)
                .collect(Collectors.toMap(
                    FieldValidationRule::fieldName,
                    Function.identity()
                ));
    }

    return new EntityValidationModel(logicalName, fields);
}

```

 Copy code

## 7 Why this is now correct

- Root is `entities[]`
  - Dimensions interpreted per entity
  - MEMBER entities drive request-field validation
  - JAVA-MAP handled as `key-value paramName/paramValue`
  - Supports 6 schemas cleanly
  - API replacement is trivial
- 

## 8 Logical next step (recommended)

Next step (only one):

👉 Build `FieldValidationRule` correctly from `dimensions[]`

This is where:

- mandatory vs optional
- enum validation
- JAVA-MAP param name validation

will be implemented.

Say the word, and we'll do **only that**, cleanly and slowly.



Yes

Good. We proceed **one step only**, as agreed.

### Step 1: Build the internal validation rule model correctly

(derived from `dimensions[]` of a MEMBER entity)

I will **not** touch the validation algorithm yet.

This step is only about **representing what the schema says**, correctly and loss-free.

---

## 1 What a MEMBER entity's dimensions[] actually mean (correct model)

From your screenshots, every MEMBER entity has a `dimensions[]` array that always follows this pattern:

### A. Exactly one `data-type-info` dimension

This defines:

- data type (`STRING`, `LONG`, `JAVA-MAP`)
- nullability
- enum / format rules
- list vs scalar

Example:

json

```
{
  "dimensionType": "data-type-info",
  "dataType": "STRING",
  "fieldFormat": "enum:INSERT|UPDATE",
  "fieldData": {
    "fieldIsNullable": false
  }
}
```

 Copy code

### B. One or more `name` dimensions

These define identity, not rules:

json

```
{
  "dimensionType": "name",
  "name": "LOGICAL-NAME",
  "value": "actionCode"
}
```

 Copy code

👉 LOGICAL-NAME is the request JSON key

👉 DISPLAY-NAME is only informational

---

## 2 Internal model we need (not DTO, not schema JSON)

We need a **pure validation model**, independent of:

- Jackson
- DTOs
- JSON shape

This model should answer questions like:

- Is this field mandatory?
- What values are allowed?
- Is this a JAVA-MAP parameter set?
- What keys are allowed inside that map?

---

### 3 Final internal model (corrected)

#### FieldValidationRule

java

```
public final class FieldValidationRule {

    private final String logicalName;                      // request JSON key
    private final DataType dataType;                        // STRING, LONG, JAVA_MAP
    private final boolean nullable;
    private final Set<String> allowedValues;             // enum values, empty if none
    private final boolean list;

    private FieldValidationRule(
        String logicalName,
        DataType dataType,
        boolean nullable,
        boolean list,
        Set<String> allowedValues
    ) {
        this.logicalName = logicalName;
        this.dataType = dataType;
        this.nullable = nullable;
        this.list = list;
        this.allowedValues = allowedValues;
    }
}
```

 Copy code

```
// getters only (immutable)
}
```

**DataType enum**

java

```
public enum DataType {
    STRING,
    LONG,
    JAVA_MAP
}
```

Copy code

**4 Correct extraction logic (this is where earlier confusion happened)****Source of truth mapping**

Validation JSON location	Meaning
dimensionType = name + LOGICAL-NAME	Request JSON key
dimensionType = data-type-info.dataType	Field data type
fieldData.fieldIsNullable	Mandatory vs optional
fieldFormat = enum:...	Allowed values
fieldIsList	Array vs scalar

**5 Building FieldValidationRule from a MEMBER entity**

java

```
public static FieldValidationRule fromMemberEntity(DcdsEntityDto entity) {

    // 1. Extract LOGICAL-NAME
    String logicalName = entity.dimensions().stream()
        .filter(d -> d instanceof DcdsNameDto)
        .map(d -> (DcdsNameDto) d)
```

```

    .filter(d -> "LOGICAL-NAME".equals(d.name()))
    .map(DcdsNameDto::value)
    .findFirst()
    .orElseThrow(() ->
        new IllegalStateException("Missing LOGICAL-NAME for entity " + entity
    );
}

// 2. Extract data-type-info
DcdsDataTypeInfoDto typeInfo = entity.dimensions().stream()
    .filter(d -> d instanceof DcdsDataTypeInfoDto)
    .map(d -> (DcdsDataTypeInfoDto) d)
    .findFirst()
    .orElseThrow(() ->
        new IllegalStateException("Missing data-type-info for " + logicalName
    );
}

DataType dataType = switch (typeInfo.dataType()) {
    case "STRING" -> DataType.STRING;
    case "LONG" -> DataType.LONG;
    case "JAVA-MAP" -> DataType.JAVA_MAP;
    default -> throw new IllegalStateException("Unsupported type: " + typeInfo
};

boolean nullable = typeInfo.fieldData() != null
    && Boolean.TRUE.equals(typeInfo.fieldData().fieldIsNullable());

boolean list = Boolean.TRUE.equals(typeInfo.fieldIsList());

Set<String> allowedValues = parseEnum(typeInfo.fieldFormat());

return new FieldValidationRule(
    logicalName,
    dataType,
    nullable,
    list,
    allowedValues
);
}

```

 Copy code

## Enum parsing helper

java

```

private static Set<String> parseEnum(String fieldFormat) {
    if (fieldFormat == null || !fieldFormat.startsWith("enum:")) {
        return Set.of();
    }

    return Arrays.stream(fieldFormat.substring(5).split("[|]"))
        .map(String::trim)
        .collect(Collectors.toSet());
}

```

Copy code

## 6 Why this fixes the mistake permanently

- Root = `entities[]`
- MEMBER entities → request fields
- `dimensions[]` interpreted **structurally**, not positionally
- JAVA-MAP treated as **field-level type**, not entity-level
- No coupling to package/class entities
- Future API schema will work unchanged

## 7 What we have NOT done yet (intentionally)

- No request validation
- No recursion
- No error reporting
- No business logic

We now have a **correct internal representation of rules**.

## Next step (choose one)

- 1 Validate mandatory vs optional
- 2 Validate enum values (STRING)
- 3 Validate JAVA-MAP paramName / paramValue pairs
- 4 Validate duplicate / unknown fields

Tell me **one number only**.



can you explain me the code until this step before we proceed

Absolutely. I'll **slow down** and explain **everything we have done up to this step, why each piece exists, and how they fit together**, without introducing anything new.

No new code. No next steps yet.

---

## 1 Where we started vs where we are now

**Initial misunderstanding (now fixed)**

- We initially treated **Package DTO** as the root **X**
- That implied a *tree structure*

**Correct reality**

- Root JSON has:

json

— ChatGPT 5.2 ▾



- Every item inside `entities[]` is a **peer entity**
- Package, Class, Member are **just entity types**
- Meaning is derived via:
  - `metadataRecordType`
  - `dimensions[]`
  - `relations[]`

This correction drives **everything** we've done since.

---

## 2 What problem we are solving at this step

We are **not validating requests yet**.

We are answering only one question:

"How do I convert a very large, noisy, metadata JSON into a **clean, minimal internal model** that describes validation rules?"

This is the **schema interpretation step**.

---

### 3 Why we introduced an *internal validation model*

The **external DTOs**:

- Are owned by another team
- Are complex (20–30 classes)
- Contain a lot of metadata irrelevant to validation
- Will evolve

If we write validation logic directly on them:

- Code becomes brittle
- Every schema change breaks validation
- Testing becomes painful

So we introduced our own model

Small, stable, focused, intentional.

---

### 4 The internal model (conceptual view)

For validation, we **only care about fields**.

Each field needs to answer:

Question	Source
What is the request JSON key?	LOGICAL-NAME
Is it mandatory?	fieldIsNullable
What type is it?	dataType
Is it an enum?	fieldFormat

**Question****Source**

Is it a map of parameters?

JAVA-MAP

That's it.

## 5 FieldValidationRule – what it represents

java

```
public final class FieldValidationRule {
    private final String logicalName;
    private final DataType dataType;
    private final boolean nullable;
    private final boolean list;
    private final Set<String> allowedValues;
}
```

 Copy code

### Meaning in plain English

This object represents **ONE request-level field**, fully described:

Example for `actionCode`:

- `logicalName` → "actionCode"
- `dataType` → STRING
- `nullable` → false (mandatory)
- `allowedValues` → {INSERT, UPDATE}

So this class answers:

"How should this single field be validated?"

## 6 Why DataType enum exists

java

```
public enum DataType {
    STRING,
    LONG,
```

JAVA\_MAP

}

Copy code

We normalize **external schema values** into **our language**.

Why?

- External schema may add new types
- We want compile-time safety
- Validation logic switches on **our enum**, not strings

This is **decoupling**.

---

## 7 Why dimensions caused confusion (and how we fixed it)

**Wrong mental model (earlier)**

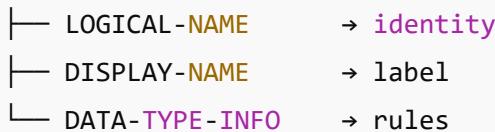
- Treating each `dimension` as a field

**Correct mental model**

- A field is described by multiple dimensions

pgsql

Field



Copy code

So `dimensions[]` is **ingredients**, not final objects.

---

## 8 How we build `FieldValidationRule` from an entity

This method:

java

```
fromMemberEntity(DcdsEntityDto entity)
```

Copy code

does exactly one thing:

## Extract one logical field from **one MEMBER entity**

### Step-by-step logic

#### 1 Find the request key

java

```
LOGICAL-NAME → "actionCode"
```

 Copy code

This is the **only source of truth** for field identity.

If it's missing → schema is invalid → fail fast.

---

#### 2 Find validation rules

java

```
dimensionType = data-type-info
```

 Copy code

From this we extract:

- dataType ( STRING , LONG , JAVA-MAP )
- nullable vs mandatory
- list vs scalar
- enum constraints

Again, if missing → schema is invalid.

---

#### 3 Normalize everything

We convert:

- "STRING" → DataType.STRING
- "enum:A|B|C" → Set<String>

So the rest of the system never deals with raw schema strings.

---

#### 9 What we have intentionally NOT done yet

We have **not**:

- ✖ Validated request JSON
- ✖ Traversed relations
- ✖ Filtered by class / rule type
- ✖ Applied mandatory checks
- ✖ Validated JAVA-MAP contents

Because doing that **before** building a clean model would be a mistake.

---

## 10 What we have achieved so far

At this point:

- ✓ We correctly understand the schema
- ✓ We corrected the root-level mistake
- ✓ We modeled validation intent cleanly
- ✓ We isolated schema parsing from validation
- ✓ We built a stable foundation

This is the **hardest part** conceptually.

Everything after this is straightforward.

---

## Final checkpoint question

Before we move on:

Do you clearly understand **why we needed** `FieldValidationRule` **and how it is derived from** `dimensions[]` ?

If yes, we proceed.

If anything feels unclear, tell me exactly **which part**, and we'll fix it before moving forward.

