

ChatGPT 5.2 Thinking ▾

This is a copy of a conversation between ChatGPT & Anonymous.

[Report conversation](#)

Uploaded a file Uploaded a file

Analysiere die beiden Dokumente und erkläre sie mir.
Mache auch eine Beurteilung auf Basis deines INTERLIS-Wissens.

Thought for 1m 4s >

Ja — die zwei Dokumente passen sehr gut zusammen: Das eine beschreibt die **Vision/Architektur** von ConvConf, das andere zeigt die **Machbarkeit an konkreten Beispielen**. Beide sind als vorläufige Fassungen von Sepp Dorfschmid gekennzeichnet (Februar 2026). ConvConf_Perspektive ConvConf_Beispiele

Was ist ConvConf in einem Satz?

ConvConf ist ein Ansatz, um den Konfigurationsaufwand für Datenaustausch von einem **nxm-Problem** (für jede Sender-/Empfänger-Kombination eine eigene Konversion) auf ein **modellzentriertes Mapping** zu reduzieren:

Jedes System beschreibt nur sein eigenes **logisches Modell (LM)** in Bezug auf ein gemeinsames **konzeptionelles Modell (KM)** — idealerweise in INTERLIS — und ConvConf leitet daraus die Konversionskonfiguration zwischen zwei Systemen ab.

ConvConf_Perspektive

Dokument 1: „Neue Perspektive für (Geo)Daten-Interoperabilität“ — erklärt

1) Problemstellung (sehr treffend beschrieben)

Das Dokument startet mit einer Gegenüberstellung:

- **Ursprüngliche Idee mit INTERLIS:** Softwarehersteller kümmern sich generisch um die Abbildung von INTERLIS-Modellen auf ihre internen Datenmodelle.

□ ConvConf_Perspektive

- **Heutige Realität:** Oft machen das nicht die Hersteller, sondern Betreiber/Projekte/Kantone individuell — häufig pro Modell und pro Anwendungsfall. Ergebnis: hoher Aufwand und Fehlerpotenzial. □ ConvConf_Perspektive

Das ist aus CH-Sicht absolut realistisch: föderale Unterschiede + viele Fachmodelle + unterschiedliche DB/Applikationsstrukturen führen zu sehr vielen Spezialkonversionen.

2) Kernidee: Gemeinsames KM, systemindividuelle LMs

ConvConf trennt sauber:

- **KM (Konzeptionelles Modell)** = gemeinsames semantisches Verständnis (vorzugsweise INTERLIS2)
- **LM (Logisches Modell)** = systemspezifische Datenstruktur/Formatbeschreibung inkl. Zuordnung zum KM (über Views) □ ConvConf_Perspektive

Der große Punkt: Sender und Empfänger müssen einander nicht kennen. Beide „sprechen“ nur über ihr LM zum gleichen KM. ConvConf baut daraus die Brücke.

□ ConvConf_Perspektive

3) Was ConvConf konkret macht

Das Dokument beschreibt zwei Komponenten:

- **ConvConf (Konfigurator):** Nimmt KM + LM(s) + LM(e) und erzeugt eine Konfiguration **KK(s2e)** für die Konversion. □ ConvConf_Perspektive
- **Konverter:** Führt die Konversion mit dieser Konfiguration aus (unabhängig von ConvConf als Produktidee). □ ConvConf_Perspektive

Das ist architektonisch sauber: **Konfigurationslogik** und **Ausführungsengine** sind getrennt.

4) Nutzen / Vision

Sehr starke Punkte im Dokument:

- Betreiber müssen nur noch ihr **eigenes System** beschreiben (nicht jede Gegenstelle). □ ConvConf_Perspektive
- Auch **reduzierte/vereinfachte Modelle** (nur ein Ausschnitt des KM) sollen als LM möglich sein. □ ConvConf_Perspektive
- Perspektive eines **Konversionsdienstes** (Service), der KM/LMs verwaltet, prüft und

Konversionen automatisch ausführt. ConvConf_Perspektive

5) Reifegrad

Der Autor sagt selbst: **POC ist erbracht**, mit Prototyp und verschiedenen Beispielen getestet. Gleichzeitig nennt er offen Einschränkungen (Geometrie/Blackbox noch nicht implementiert, wenige Konversionsfunktionen, unvollständige Prüfung/Dokumentation). Das ist sehr transparent und glaubwürdig. ConvConf_Perspektive

Dokument 2: „Einstieg mit Beispielen“ — erklärt

Dieses Dokument ist die praktische Ergänzung: Es zeigt **an einem kleinen Vereinsmodell**, wie man verschiedene Datenstrukturen auf denselben Inhalt (KM)

abbilden kann.  ConvConf_Beispiele

1) Das Test-KM (konzeptionelles Modell)

Das Beispielmodell enthält u.a.:

- Mitglied (abstrakt) mit Spezialisierungen Person und Organisation
- Veranstaltung
- Teilnahme als Assoziation
- Kommentar / OrgKommentar als Strukturen
- Aufzählung JurPersonArt (inkl. verschachtelter Enum-Struktur)

 ConvConf_Beispiele

Das ist absichtlich „klein“, aber modellierungstechnisch gut gewählt, weil es typische INTERLIS-Themen enthält:

Vererbung, Referenzen, Assoziation, Strukturen, Enumerationen, Nested-Strukturdaten.  ConvConf_Beispiele

2) Beispiel-Daten

Die Daten werden in einem einfachen textuellen Zeilenformat gezeigt („Stil INTERLIS 1“ als lesbares Testformat), nicht als offizielles INTERLIS-Transferformat. Das dient der Verständlichkeit des Prototyps.  ConvConf_Beispiele

3) Die wichtigen LM-Beispiele (sehr anschaulich)

a) LM nahe bei INTERLIS 2

Startpunkt mit einer Strukturierung, die dem KM recht nahe ist (inkl. Nesting). Das dient als Referenzbasis für weitere Umformungen.  ConvConf_Beispiele

b) Relational „Tabelle pro Subklasse“

Mitglied -Gemeinsamkeiten in eigener Tabelle (Member), Spezifika in Person / Organisation .

Das ist ein klassisches ORM-/DB-Schema und zeigt, dass ConvConf Vererbungsabbildungen ausdrücken kann.  ConvConf_Beispiele

c) „Flaches“ relationales Modell

Alle Mitglieder in eine Tabelle mit Typ-Spalte + gemischten Attributen (FMember). Die konkrete Klasse wird über VALUEMAP (1=Person, 2=Organisation) auf Klassen gemappt. Das ist sehr praxisnah.  ConvConf_Beispiele

d) Keine Referenzattribute, nur Beziehungen

Anstatt Person.Vertretene_Organisation als Referenz wird eine separate Beziehungstabelle PersOrg verwendet.

Das ist ein häufiges Integrationsmuster (und aus INTERLIS-Sicht semantisch heikel, aber machbar). ConvConf adressiert das explizit. ConvConf_Beispiele

e) OID in separater Tabelle

Objektkennungen werden außerhalb der Hauptdatensätze geführt (0id -Tabelle) und per ANNEXE eingebunden.

Das ist ein starkes Beispiel für „attributweise ausgelagerte“ externe Datenstrukturen.

ConvConf_Beispiele

f) Strukturattribute ohne Nesting

Kommentare (Strukturen) werden in eigener Tabelle geführt, inklusive:

- Besitzerobjekt (\$PARENT)
- welches Strukturattribut gemeint ist (\$STRUCTATTR)
- Klassentyp (Kommentar / OrgKommentar) via VALUEMAP

Das ist eines der technisch interessantesten Beispiele, weil es echte Komplexität abbildet. ConvConf_Beispiele

4) Umgang mit Enumerationen

Das Dokument zeigt eine explizite VALUEMAP von Integer-Codes auf INTERLIS-Aufzählwerte (JurPersonArt).

Das ist sehr wichtig in der Praxis und sauber demonstriert. ConvConf_Beispiele

5) Output-/Abfrage-LMs

Die letzten Beispiele zeigen „nur Output“-Modelle:

- Personen mit zugeordneter Organisation (JOIN/Outer Join)
- Organisationen mit eingeschachtelten Repräsentanten (NESTING)

ConvConf_Beispiele

Das ist besonders spannend, weil ConvConf hier nicht nur als „Formatkonverter“, sondern auch als **semantischer Projektions-/Abfragegenerator** erscheint.

Wie die beiden Dokumente zusammenhängen

- **Dokument 1 (Perspektive)** = Warum das Problem existiert, warum ein KM/LM-Ansatz helfen kann, und wie ein Dienst aussehen könnte. ConvConf_Perspektive
- **Dokument 2 (Beispiele)** = Beweis, dass die LM-Sprache viele reale Strukturvarianten tatsächlich ausdrücken kann. ConvConf_Beispiele

Kurz: Das erste ist die **Strategie**, das zweite die **technische Demonstration**.

Beurteilung aus INTERLIS-Sicht

Was fachlich sehr stark ist

1) Sehr gute Übereinstimmung mit dem INTERLIS-Grundgedanken

INTERLIS trennt bewusst **konzeptionelle Semantik** von technischer Umsetzung. Genau das macht ConvConf zum Zentrum der Architektur:

KM (semantisch) \leftrightarrow LM (systemlogisch) \leftrightarrow Konversion.

Das ist aus INTERLIS-Sicht nicht nur sinnvoll, sondern eigentlich „im Geist der Sache“.

ConvConf_Perspektive

2) Realistische Antwort auf ein echtes CH-Problem

Die Dokumente adressieren präzise die föderale und systembedingte Heterogenität. INTERLIS alleine definiert ein Modell/Transfer, aber nicht automatisch die vollständige Brücke in jedes beliebige Zielschema. ConvConf setzt genau dort an.

ConvConf_Perspektive

3) Semantisch interessante Mapping-Fähigkeiten

Die Beispiele zeigen, dass nicht nur triviale Feld-zu-Feld-Mappings gemeint sind, sondern auch:

- Vererbungsabbildungen
- Referenz vs. Beziehung
- Struktur-Nesting vs. Tabellen
- Enum-Codierung
- Ausgabe-Projektionen / Joins / NESTING ConvConf_Beispiele

Das ist für Interoperabilität entscheidend.

4) Potenziell große Hebelwirkung

Wenn LM-Beschreibungen tatsächlich stabil und wiederverwendbar sind, sinkt der Integrationsaufwand massiv (weg von vielen individuellen Konvertern). Das ist ein starker Systemeffekt.

Wo ich aus INTERLIS-Sicht die größten Risiken / offenen Punkte sehe

1) Semantik-Erhält ist schwerer als Strukturabbildung

Die Beispiele zeigen primär Strukturumformungen. In produktiven Geo-/Fachdatenwelten kommt viel dazu:

- Kardinalitäten / Pflichtattribute
- Eindeutigkeit / Identitäten (TID/OID-Semantik)
- Domänenregeln
- Geometrien (inkl. CRS, Topologie, Kurven)
- Datum/Zeit, Einheiten, Formatnormalisierung
- Reihenfolge / Stabilität von IDs
- Objektlebenszyklus (Änderungen, Löschungen, inkrementelle Transfers)

Der Autor nennt selbst, dass Geometrie, Blackbox-Typen und weitere Konversionsfunktionen im Prototyp noch fehlen. Das ist aus INTERLIS-/Geodaten-Sicht der größte Schritt zur Produktionsreife. ☐ ConvConf_Perspektive

2) Prüfung der LM-Korrektheit ist zentral (und laut Dokument noch unvollständig)

Für einen verlässlichen Dienst braucht es starke statische Prüfungen:

- Typkompatibilität
- Vollständigkeit für benötigte Output-Felder
- Konsistenz der Klassen-/Attributpfade
- Eindeutigkeit/Join-Sicherheit
- Umgang mit optionalen Daten und Mehrdeutigkeiten

Das Dokument nennt die unvollständige Prüfung explizit — völlig richtig als prioritäres Thema. ☐ ConvConf_Perspektive

3) Reduzierte LMs sind praktisch, aber oft nicht invertierbar

Das ist kein Fehler, aber wichtig:

Ein vereinfachtes LM (z.B. nur Ausgabe/Abfrage) verliert häufig Information. Dann ist eine Rückkonversion nicht möglich oder nur eingeschränkt. Das sollte im System formal markiert werden (z.B. **read-only/output-only**, was das Beispieldokument bereits andeutet mit <- statt <->).  ConvConf_Beispiele

4) Versionierung und Evolution (KM/LM)

In der INTERLIS-Praxis ändern sich Modelle. Ein produktiver ConvConf-Dienst braucht deshalb:

- Modellversionen
- Kompatibilitätsregeln
- Migration von LMs
- Nachvollziehbarkeit, welche Konfiguration mit welcher Modellversion erzeugt wurde

Das ist im Visionsteil noch nicht tief beschrieben (was für ein Vorentwurf okay ist).

 ConvConf_Perspektive

5) Interaktion mit bestehender INTERLIS-Toolchain

Aus meiner Sicht sollte ConvConf **nicht als Ersatz**, sondern als **Ergänzung** zu bestehender INTERLIS-Toolchain gedacht werden (Compiler/Validator/DB-Generator/Transfer-Tools).

Der stärkste Weg wäre: ConvConf baut auf validierten INTERLIS-Modellen auf und nutzt vorhandene Werkzeuge, wo möglich.

Meine fachliche Gesamtbewertung

Sehr überzeugende Idee, fachlich sauber positioniert, und die Beispiele sind überraschend stark.

Vor allem die Beispiele zu Vererbung, Referenz/Beziehung, ausgelagerten OIDs und flach gespeicherten Strukturen zeigen, dass der Ansatz mehr ist als eine simple Mapping-Tabelle.  ConvConf_Beispiele

Wenn man es nüchtern bewertet:

- **Konzeptqualität:** hoch
 - **Praxisrelevanz (CH/INTERLIS):** sehr hoch
 - **POC-Glaubwürdigkeit:** gut (transparent dokumentierte Grenzen)
 - ConvConf_Perspektive
 - **Weg zur produktiven Lösung:** noch substanzial (v.a. Geometrie, Validierung, Robustheit, Governance)
-

Was ich als Nächstes empfehlen würde (aus INTERLIS-Sicht)

1. **Formale Spezifikation der LM-Sprache schärfen** (inkl. Typregeln, Fehlerfälle, Semantik von Join/Nesting/INSPECTION).
 2. **LM↔KM-Validierung massiv ausbauen** (präzise Fehlermeldungen).
 3. **Geometrie- und Datum/Zeit-Konversionen priorisieren** (sonst im Geo-Bereich limitiert).
 4. **Konformitäts-/Regressionstests** mit realen CH-Fachmodellen aufsetzen.
 5. **Rollenklärung** zu existierenden INTERLIS-Werkzeugen (Integration statt Konkurrenz).
 6. **Service-MVP definieren** (welche Funktionen zuerst: nur batch? nur read-only? nur bestimmte Formate?).
-

Wenn du willst, kann ich dir als Nächstes noch eine „**INTERLIS-Brille**“-Lesart der **LM-Syntax** machen (z.B. ANNEXE , VALUEMAP , INSPECTION , NESTING , WITH , <-> vs <-) und erklären, wie das semantisch zu INTERLIS-Konzepten passt.

Sources

Bräuchte es dann nicht auch Werkzeuge, um die Syntax der LM zu überprüfen? Und dass die Daten gemäss dem LM korrekt sind?

Thought for 25s >

Ja — **unbedingt**. Und genau das ist einer der entscheidenden Punkte für eine produktive ConvConf-Lösung.

Du triffst damit einen Kernpunkt: Es braucht **mindestens zwei Arten von Werkzeugen**:

1. **LM-Validator / „Compiler“** (prüft die *Beschreibung* des logischen Modells)
2. **Daten-Validator gegen LM** (prüft die *konkreten Daten* gemäss diesem LM)

Das ist auch in den Dokumenten bereits angelegt:

- Der geplante Dienst soll LMs **entgegennehmen und prüfen**.  ConvConf_Perspektive
- Gleichzeitig steht explizit drin, dass die **Überprüfung der LMs auf Korrektheit noch unvollständig** ist und Fehlermeldungen knapp sind.  ConvConf_Perspektive

Aus INTERLIS-Sicht: das Pendant zu „Compiler + Validator“

Bei INTERLIS ist es ja ähnlich getrennt:

- **Modell prüfen** (Syntax/Semantik des INTERLIS-Modells)
- **Daten prüfen** (Transferdaten gegen Modell + Constraints)

Für ConvConf/LM wäre die saubere Analogie:

- **LM-Compiler/Checker** (analog zu einem Modell-Compiler)
- **LM-Datenvalidator** (analog zu einem Datentransfer-Validator)
- optional zusätzlich: **KM-seitige Validierung nach Mapping** (sehr wichtig)

1) Was ein LM-Prüfwerkzeug prüfen müsste (statisch)

Nicht nur „Syntax korrekt?“, sondern auch **semantische Konsistenz** mit dem KM.

A. Reine Syntaxprüfung (Parser)

Beispiele:

- DATA ... END
- VALUemap , CONVERSION , ANNEXE , JOIN , INSPECTION , NESTING , WITH
- korrekte Klammern, Tokens, Operatoren (<-> , <-)

Das ist die Mindeststufe.

B. Semantische LM-Prüfung gegen KM (wichtiger)

Hier wird es spannend:

- Existieren referenzierte KM-Elemente wirklich?
 - Klassen, Attribute, Strukturen, Enums
- Sind Typen kompatibel?
 - INTEGER ↔ Enum nur mit VALUemap
 - Referenzen nur auf passende Zielklassen
- Ist die Verwendung von Spezialsymbolen korrekt?
 - \$CLASS, \$PARENT, \$STRUCTATTR, \$IDENT
- Sind ALIAS / WITH / ANNEXED konsistent?
- Ist VALUemap gültig?
 - doppelte Zuordnungen?
 - fehlende Werte?
 - ungültige Zielwerte?
- Ist ein LM als bidirektional (\leftrightarrow) wirklich plausibel oder nur Output (\leftarrow)?
 - Gerade bei JOIN/NESTING/Sichten oft **nicht invertierbar**. Das wird in den Beispielen ja schon angedeutet (\leftarrow für Output-only). ConvConf_Beispiele

C. „Konvertierbarkeitsprüfung“ (Input deckt Output ab)

Das ist im Perspektiven-Dokument explizit vorgesehen:

- Es soll geprüft werden, ob der Input die vom Output benötigten Daten enthält.
 ConvConf_Perspektive

Das ist aus meiner Sicht ein eigenes Werkzeug/Modul:

- **Abdeckungsanalyse**
- Welche KM-Attribute/Klassen braucht LMe?
- Welche kann LMs liefern?
- Wo fehlen Informationen / wo entstehen Ambiguitäten?

2) Was ein Datenprüfwerkzeug gegen LM prüfen müsste (Instanzdaten)

Das ist die zweite Hälfte deiner Frage — und genauso wichtig.

A. Formale Datenstruktur gemäss LM

- Gibt es nur deklarierte Datensatztypen (DATA X)?
- Stimmen Anzahl/Reihenfolge der Felder?
- Sind Pflichtwerte vorhanden (MANDATORY)?
- Stimmen Datentypen (Text, Integer, ObjID, Referenz)?

Gerade im Beispieldokument wird mit einem einfachen Zeilenformat gearbeitet; dort wäre das essenziell. ConvConf_Beispiele

B. Referenzielle Integrität im LM

- Zeigt TransferID T0 ... auf existierende Datensätze?
- Bei \$OWNER / ANNEXE : gibt es das Besitzerobjekt?
- Bei ausgelagerten OID-/Attributtabellen: passt die Zuordnung? (z. B. Beispiel mit separater Oid -Tabelle) ConvConf_Beispiele

C. Wertebereich / Codierungen

- VALUemap : kommt nur ein zulässiger Code vor?
- sind alle enum-codierten Werte abbildbar?
- ungültige Codes sauber melden (z. B. OrgForm=999)

D. Strukturen / Nesting / Spezialfälle

Bei Dingen wie:

- Strukturattribute als eigene Tabelle
- INSPECTION
- \$PARENT , \$STRUCTATTR , \$CLASS

braucht es zusätzliche Konsistenzprüfungen:

- gehört der Strukturdatensatz zu einem gültigen Parent?
- passt der Klassendiskriminator zur befüllten Feldmenge?
- ist das referenzierte Strukturattribut wirklich eines der erlaubten?

Die Beispiele zeigen genau solche komplexen Fälle (z. B. Comment mit INSPECTION , \$PARENT , \$STRUCTATTR , \$CLASS).

3) Aus INTERLIS-Sicht sogar sinnvoll: zweistufige Datenvalidierung

Ich würde das für ConvConf klar so aufbauen:

Stufe 1: Validierung der Rohdaten gegen LM

„Sind die gelieferten Systemdaten formal und logisch gemäss LM korrekt?“

Stufe 2: Validierung der transformierten Daten gegen KM (INTERLIS)

„Ergeben die gemappten Daten ein gültiges KM-konformes Datenbild?“

Das ist extrem wertvoll, weil man so Fehler sauber trennt:

- **LM-/Quelldatenfehler** (z. B. ungültiger Ref, falscher Code)
 - **Mapping-/Semantikfehler** (z. B. Pflichtattribut im KM bleibt leer)
 - **KM-/Fachregelverletzung** (z. B. INTERLIS-Constraints, Kardinalitäten)
-

4) Welche Werkzeuge ich konkret erwarten würde (Produktionsreife)

Wenn man es sauber machen will, bräuchte man eigentlich eine kleine Toolchain:

- **lmccheck**
Prüft LM-Syntax + statische Semantik gegen KM
 - **lmid-validate**
Prüft Daten gegen LM (Schema + Referenzen + Codierungen + Konsistenz)
 - **convconf-planchek**
Prüft LMs→LMe-Abdeckung / Konvertierbarkeit vor dem Lauf
 - **kmvalidate (INTERLIS-seitig)**
Prüft Ergebnisdaten gegen KM / INTERLIS-Regeln
 - **lint / Quality-Checks** (optional, aber praktisch)
 - unvollständige VALUEMAP
 - riskante JOINS
 - potenziell mehrdeutige Aliase
 - unnötig komplexe Konstrukte
-

5) Warum das nicht „nice to have“, sondern Pflicht ist

Ohne diese Werkzeuge wird ConvConf in der Praxis schnell schwierig, weil:

- LM-Sprache relativ mächtig ist (JOIN, NESTING, INSPECTION, ANNEXE etc.)

- Fehler oft **nicht syntaktisch**, sondern **semantisch** sind
- Integrationsfehler sonst erst bei der Konversion oder (schlimmer) in den Fachdaten auffallen

Und genau das würde den eigentlichen Nutzen (weniger Aufwand / weniger Fehlerrisiko) wieder untergraben.

Kurzantwort auf deine Frage

Ja, absolut.

Für ConvConf braucht es aus meiner Sicht zwingend:

- ein Werkzeug zur **LM-Prüfung (Syntax + Semantik gegen KM)**
- ein Werkzeug zur **Datenprüfung gemäss LM**
- idealerweise zusätzlich eine **KM-/INTERLIS-Validierung nach der Transformation**

Das passt auch vollständig zur in den Dokumenten beschriebenen Zielarchitektur eines Dienstes mit LM-Entgegennahme und Prüfung.

Wenn du willst, kann ich dir als Nächstes eine **konkrete Prüfliste („Muss-Regeln“)** für **ein LM-Check-Tool** skizzieren — z. B. nach Kategorien wie **DATA** , **VALUemap** , **ANNEXE** , **JOIN** , **NESTING** , **INSPECTION** .

Sources

Der erwähnte einfache PoC ist closed source. Wie gut könntest du anhand der Unterlagen einen solchen PoC selber bauen (mit Java)?

Thought for 45s >

Ja — einen vergleichbaren **einfachen PoC** könnte ich anhand der Unterlagen ziemlich gut entwerfen und mit Java umsetzen.

Aber: **nicht 1:1 den bestehenden closed-source PoC nachbauen**, weil dafür ein paar entscheidende Semantikdetails fehlen bzw. in den Unterlagen bewusst nicht vollständig

ausdefiniert sind.

Meine ehrliche Einschätzung

Was ich gut nachbauen könnte (PoC-Niveau)

Auf Basis der zwei PDFs ist ein Java-PoC für den dokumentierten Funktionsumfang realistisch, insbesondere weil:

- die Zielarchitektur klar beschrieben ist (KM + LMs → ConvConf → Konfig/Views → Konverter), inkl. Trennung von Konfigurator und Konverter. □ ConvConf_Perspektive
- die Beispiele sehr konkret sind und echte LM-Konstrukte zeigen (VALUEMAP , ANNEXE , JOIN , NESTING , INSPECTION , WITH , \$CLASS/\$PARENT/\$STRUCTATTR).
- der Autor die Beispielkonversionen inkl. konkreter (künstlicher) Daten und Resultate dokumentiert hat — das ist ideal als Testsuite/Referenzverhalten.
- der dokumentierte Test-Konverter absichtlich einfach gehalten ist (einfaches Zeilenformat, keine Performanceoptimierung).

Was ich nur mit Annahmen nachbauen könnte

Die Dokumente sind stark, aber sie sind **keine vollständige Sprach- und Ausführungsspezifikation**. Das Beispiel-Dokument sagt sogar explizit, dass es nicht detailliert auf Möglichkeiten/Bedeutungen eingeht. □ ConvConf_Beispiele

Außerdem nennt der Autor selbst Limitierungen des Prototyps:

- formale Sprache zwar definiert, aber noch nicht praktisch erhärtet,
- LM-Prüfung unvollständig,
- Dokumentation unvollständig.

Das heißt: Ich könnte einen PoC bauen, aber gewisse Randfälle/Fehlerfälle müsste ich **selbst definieren**.

Was ich konkret in Java bauen würde (PoC-Scope)

1) Parser + AST für einen LM-Teilumfang

Ein robuster Parser für die in den Beispielen sichtbare Syntax, z. B.:

- DATA -Definitionen (inkl. Typen, Referenzen, NESTING)

- LM↔KM-Zuordnungen (\leftrightarrow , \leftarrow)
- VALUemap
- CONVERSION
- ALIAS , WITH
- ANNEXE / ANNEXED
- JOIN (inkl. optionalem Join via ?)
- INSPECTION ... ALL

Das lässt sich in Java sauber mit einem Parser-Framework (z. B. ANTLR) + AST-Modell umsetzen.

2) Internes Modell für KM/LM/Views

Die Perspektive beschreibt explizit, dass der PoC KM/LM einliest und in interne Modellelemente umsetzt. ConvConf_Perspektive

Genau das wäre mein Kern:

- KM-Metaobjekte (Klassen, Attribute, Strukturen, Enums)
- LM-Datensätze und Felder
- Mapping-Regeln / View-Definitionen
- Konversionsplan (intermediate representation)

3) Einfache Ausführungsengine (Konverter für Testzwecke)

Angelehnt an den dokumentierten „simpel“ Konverter:

- Reader für das einfache Zeilenformat (im Stil INTERLIS 1)
- In-Memory-Ausführung (Listen/Maps statt DB)
- Writer für dasselbe Format
- Ausführung von:
 - Feldzuordnungen
 - VALUemap
 - Klassen-Diskriminierung via \$CLASS
 - Referenzen / Owner-Bezüge via \$PARENT
 - flache ↔ verschachtelte Strukturabbildungen (NESTING , INSPECTION)

Die Beispiele liefern dafür sehr gute Zielbilder.

4) Erste Validierungstools (wie du vorher angesprochen hast)

Mindestens:

- **LM-Syntaxcheck**
- **LM-semantischer Check gegen KM** (existiert Attribut/Klasse? Typ passt?)
- **Datencheck gegen LM** (Feldzahl, Typen, Referenzen, Enum-Codes)

Das wäre im Sinne des Ausblicks („LM entgegennehmen und prüfen“) und adressiert auch eine explizit genannte Schwäche des jetzigen Prototyps.

Wo die Unterlagen für einen Nachbau nicht ausreichen (ohne zusätzliche Festlegungen)

Die größten Lücken wären aus meiner Sicht:

A) Vollständige Sprachsemantik

Die Beispiele zeigen **wie** etwas aussieht, aber nicht immer die vollständige Regel:

- Prioritäten/Bindung in Ausdrücken
- genaue Fehlersemantik
- Mehrdeutigkeitsauflösung
- Invertierbarkeit bei \leftrightarrow in komplexen Fällen
- Verhalten bei Null/fehlenden Referenzen/Outer Joins im Detail

B) Vollständige Planungslogik im ConvConf-Kern

Das Perspektive-PDF beschreibt den Kern als Java-Code, der den nötigen Strukturumbau bestimmt, aber nicht die exakte Algorithmik.

Also: **Architektur klar, Algorithmus nur teilweise sichtbar.**

C) Produktionsnahe Features

Explizit noch nicht im Prototyp:

- Geometrie
- Blackbox-Typen
- viele Konversionsfunktionen (Datum/Zeit, Geometrie, String-Umformungen etc.)

 ConvConf_Perspektive

Einen *einfachen* PoC kann man darum gut bauen. Einen Geo-/Enterprise-tauglichen Nachbau: deutlich schwieriger.

Mein realistisches Fähigkeitsprofil für dieses Vorhaben (Java)

Wenn du mich fragst „wie gut“:

- Dokumentierten PoC-Teilumfang (Beispielwelt) in Java: hoch
- Generischer PoC mit sauberer Architektur + Testframework: hoch
- 1:1 Verhalten des closed-source PoC in allen Randfällen: mittel bis niedrig (ohne Reverse Engineering / zusätzliche Spezifikation)
- Produktionsreife ConvConf-Plattform: deutlich mehr Arbeit (v. a. Validierung, Fehlermeldungen, Geometrie, Governance)

Der wichtigste Punkt:

Die Unterlagen sind **gut genug** für einen belastbaren *Rebuild-PoC*, aber **nicht gut genug** für einen vollständig kompatiblen Re-Implementation-Claim.

Was ich konkret als ersten Java-PoC bauen würde (sinnvolle Minimalversion)

1. KM als vereinfachtes Meta-Modell laden (zunächst aus einer eigenen Zwischenrepräsentation; INTERLIS-Import optional später)
2. LM-Parser für den in den Beispielen verwendeten Sprachteil
3. Semantik-Checker (LM gegen KM)
4. In-Memory-Konverter für das einfache Zeilenformat
5. Golden Tests mit den Beispiel-Inputs/-Outputs aus dem PDF
6. Trace-Ausgabe, damit man sieht, wie ein Ziel-Feld entstanden ist (für Debugging extrem wertvoll)

Damit hätte man schnell einen PoC, der fachlich überzeugend ist und genau an den dokumentierten Beispielen messbar wäre. Die Beispiele sind dafür wirklich ein Geschenk.

Kurz gesagt

Ja, ziemlich gut — für einen eigenen, dokumentenbasierten Java-PoC.

Nein, nicht seriös 1:1 kompatibel ohne zusätzliche Spezifikation oder Zugriff auf den closed-source Code/Tests.

Wenn du willst, kann ich dir als Nächstes eine **konkrete Java-Architektur für so einen Rebuild-PoC** skizzieren (Pakete, Klassen, AST, Validator, Ausführungsengine, Teststrategie).

Sources

Ja, du kannst auch gerne Annahmen treffen, die uns weiter bringen. Bei Annahmen kannst du auch nachfragen, falls du meinst, dass vielleicht noch zusätzliches Wissen vorhanden ist oder wenn man sowieso entscheiden soll welche Variante am sinnvollsten ist.

Thought for 48s >

Perfekt — das ist genau die richtige Haltung für so ein Vorhaben.

Ja: Mit expliziten Annahmen kann ich einen belastbaren Java-Rebuild-PoC sehr konkret vorantreiben.

Und ich würde das sogar bewusst so machen: **Annahmen dokumentieren, testen, bei Bedarf nachschärfen** (statt auf eine nie vollständig werdende Spezifikation zu warten).

Die Unterlagen geben dafür eine starke Basis:

- Architekturidee und Rollen von KM / LM / ConvConf / Konverter sind klar beschrieben.  ConvConf_Perspektive
- Es gibt konkrete LM-Beispiele inkl. Daten und erwarteten Ergebnissen (ideal als Referenztests).
- Der bestehende PoC ist in Java und umfasst Parser/Modell/Kernel/Konfig/Konverter, aber mit klar benannten Einschränkungen (z. B. Geometrie, Validierung, Doku).
 ConvConf_Perspektive

So würde ich es angehen

Zielbild für unseren PoC (nicht 1:1-Replikat)

Ein Java-PoC, der:

1. KM + LM einliest
2. LM gegen KM statisch prüft
3. aus LMs/LMe einen Konversionsplan erzeugt
4. konkrete Daten im einfachen Zeilenformat konvertiert
5. die Beispiele aus dem PDF reproduzierbar besteht

Damit wären wir fachlich sehr nah an der dokumentierten Idee, ohne uns an unbekannte Interna des closed-source PoC zu binden.

Meine Annahmen (explizit und absichtlich pragmatisch)

Ich würde mit diesen Annahmen starten, weil sie uns schnell arbeitsfähig machen:

A1 — PoC-Scope = Beispielabdeckung zuerst

Wir priorisieren die im Beispiel-PDF sichtbaren Konstrukte:

- DATA
- Zuordnung <-> / <-
- VALUemap
- CONVERSION
- ALIAS
- WITH
- ANNEXE / ANNEXED
- JOIN (inkl. ? Outer-Join)
- NESTING
- INSPECTION ... ALL
- Spezialfelder wie \$CLASS , \$PARENT , \$STRUCTATTR

Diese kommen in den Beispielen real vor.

A2 — Kein Geo-Feature in v1

Geometrie / Blackbox-Typen / erweiterte Konversionsfunktionen bleiben zunächst

draußen (passt zur dokumentierten Einschränkung des bestehenden PoC).

ConvConf_Perspektive

A3 — Einfaches Zeilenformat als Test-IO

Für den PoC verwenden wir dasselbe „simple line format“-Prinzip wie im Beispiel-Dokument (lesbar, schnell testbar).

A4 — Semantik bei Unklarheiten: “strict + explizit dokumentiert”

Wenn die PDF-Semantik nicht vollständig ist:

- wir wählen eine klare Regel,
- markieren sie als Annahme,
- schreiben Tests dafür,
- machen sie später konfigurierbar falls nötig.

Das ist m. E. der schnellste Weg zu einem brauchbaren Ergebnis.

Konkrete Java-Architektur (PoC, aber sauber)

1) Module / Pakete

Ich würde das in klar getrennte Bausteine schneiden:

convconf-km

KM-Metamodell (INTERLIS-nah, aber PoC-intern)

- KmModel
- KmClass , KmStructure , KmAssociation
- KmAttribute
- KmEnumType , KmRefType , KmScalarType
- KmPath (für Pfade wie Veranstaltung.DirektKommentare)

convconf-lm-grammar

Parser + Lexer (z. B. ANTLR)

- Grammatik für den PoC-Sprachteil
- AST-Knoten für LM-Dateien

convconf-lm-ast

Syntaxbaum (nah an Text)

- LmFileAst
- DataDeclAst , FieldDeclAst
- MappingBlockAst
- JoinSourceAst , InspectionAst , NestedDeclAst
- ValueMapDeclAst , ConversionDeclAst

convconf-lm-semantic

AST → semantisches LM-Modell + Checks

- Symboltabellen
- Namensauflösung
- Typprüfung
- Sonderbedeutungen (\$CLASS , \$PARENT , \$STRUCTATTR)

convconf-plan

ConvConf-Kern (Planner)

- Input: KM + LM(s) + LM(e)
 - Output: ConversionPlan
 - plant:
 - benötigte Sichten/Views
 - Zwischenfelder
 - Reihenfolge (read → transform → write)
 - Konversionsparameter (z. B. Enum-Mappings)
- Das entspricht inhaltlich der dokumentierten Rolle von ConvConf + Konfig mit Views/Parametern/Ablaufbeschreibung.

convconf-runtime

Ausführungsengine

- In-Memory-Datenmodell
- Operatoren (JOIN , PROJECT , NEST , ANNEX , INSPECT)
- Evaluator für Feldzuweisungen
- Fehler-/Trace-Mechanismus

convconf-io-simpleline

Ein-/Ausgabe für PoC-Testformat

- Parser für DATA ... , NEST ... , ENDN
- Serializer
- Validierung gegen deklarierte DATA -Strukturen

convconf-cli

CLI für schnelle Iteration

- check-lm
- validate-data
- plan
- convert
- trace

convconf-tests

Golden Tests, Negativtests, Roundtrip-Tests

2) Internes Datenmodell für die Runtime (wichtig für PoC-Geschwindigkeit)

Ich würde **nicht** zuerst ein relationales Backend bauen, sondern ein simples In-Memory-Modell:

- DataSet = Sammlung von RecordTable
- RecordTable(name) = Liste von Record
- Record = Map <FieldName, Value> + optional recordId
- NestedRecords separat modelliert (z. B. Parent→Children Map)

Werte-Typen:

- TextValue
- IntValue
- RefValue (z. B. TransferID / Dataldent)
- NullMarker (für @ in Beispielen, falls wir das so interpretieren)
- EnumLiteralValue

- ClassLiteralValue
- StructAttrLiteralValue

Das reicht für die Beispielwelt völlig aus.

3) Planner-Ansatz (der "ConvConf-Kern" im Kleinen)

Statt direkt alles "magisch" zu konvertieren, würde ich einen **expliziten Plan** generieren.

Beispielhafte Plan-Knoten

- ReadTableStep(Event)
- ApplyValueMapStep(UFormMap)
- JoinStep(Person, Organisation, outer=true)
- AnnexStep(Member -> Person/Organisation)
- InspectStructureStep(Kommentar ALL)
- NestStep(Representatives by Vertretene_Organisation)
- WriteTableStep(PersonWithOrg)

Vorteil:

- leichter testbar
 - nachvollziehbar
 - später in SQL/Streaming überführbar
-

4) Validierung (dein Punkt — und ich würde das früh einbauen)

Ich würde schon im PoC zwei Validatoren liefern:

a) LMChecker (statisch)

Prüft u. a.:

- Syntax/Parserfehler
- unbekannte Klassen/Attribute/Strukturen im KM
- Typkompatibilität
- Gültigkeit von VALUEMAP
- korrekte Verwendung von \$CLASS , \$PARENT , \$STRUCTATTR
- ANNEXE / ANNEXED -Konsistenz

- Join-Referenzen plausibel
- <-> nur bei plausibel invertierbaren Fällen (zumindest Warnung)

Das ist auch direkt anschlussfähig an die im Perspektive-Dokument genannte Notwendigkeit der LM-Prüfung.

b) LMDataValidator (Instanzdaten gegen LM)

Prüft:

- Datensatztyp existiert?
 - Feldanzahl/-typen korrekt?
 - Pflichtattribute vorhanden?
 - Referenzen zeigen auf gültige Datensätze?
 - VALUEMAP deckt Werte ab?
 - NESTING-Struktur formal korrekt?
-

5) Wo ich bewusst Annahmen treffen würde (und wie)

Ein paar Dinge sind in den PDFs angedeutet, aber nicht formal ausdefiniert. Genau dort würde ich aktiv Annahmen setzen.

Annahmegruppe S — Semantik-Details

Beispiele:

- **S1: Nullrepräsentation (@)**
 - Annahme: @ = "kein Wert" / Null (auch bei Referenzen)
 - Alternative später: getrennte Marker für "null" vs "leer" vs "nicht gesetzt"
- **S2: ANNEXED Merge-Priorität**
 - Annahme: explizite Felder im Hauptmapping haben Vorrang vor annexierten Feldern
 - Kollision = Fehler (nicht still überschreiben)
- **S3: JOIN ?**
 - Annahme: ? markiert optionales Join-Teil (Left Outer Join vom bereits aufgebauten linken Kontext)
 - Null-Felder bei Nichttreffer
- **S4: <-> bei komplexen Sichten**

- Annahme: nur für als invertierbar markierte Muster erlaubt
- bei JOIN + Projektion + Verlust → Warnung/Fehler; stattdessen <- empfehlen
(passt gut zu den Output-only Beispielen im PDF).
- **S5: INSPECTION** Kommentar ALL
 - Annahme: liefert alle Strukturvorkommen von Typ Kommentar inkl. Subtypen + Metainfos (\$CLASS , \$PARENT , \$STRUCTATTR)
 - Reihenfolge zunächst nicht garantiert

Diese Annahmen würde ich als **ADR (Architecture Decision Records)** ablegen. Dann kann man sie später gezielt ändern.

6) Teststrategie (entscheidend bei unvollständiger Spezifikation)

Die PDFs sind dafür Gold wert, weil sie echte Beispiele mit Daten zeigen.

Testebenen

1. Parser-Tests

- einzelne LM-Fragmente aus den Beispielen parsen
- AST-Snapshots

2. Semantik-Tests

- gültige Beispiele müssen bestehen
- gezielte Fehlerfälle (falsches Attribut, ungültige Enum-Map, falscher Alias)

3. Golden Conversion Tests

Beispiel:

- Input = LM I2-nah
- Output = flach/Annexe/Join/Nesting
- Ergebnis = exakt erwartete Daten

4. Roundtrip-Tests (wo sinnvoll)

Nur bei invertierbaren LMs (<-> -Muster):

- A → B → A sollte stabil sein (oder definierte Äquivalenz)

5. Trace-Tests

Für ein Zielattribut zeigen:

- aus welchem Source-Feld / via welcher Conversion es kam
Das spart später sehr viel Zeit im Debugging.
-

7) Was ich *nicht* sofort bauen würde (bewusst)

Damit wir nicht verzetteln:

- keine Web-Oberfläche
- keine DB-Anbindung
- keine Geometrie
- kein vollständiger INTERLIS-Compiler-Ersatz
- keine Optimierung (Streaming, große Datenmengen)

Das ist auch konsistent mit dem dokumentierten "Test-Konverter"-Charakter des vorhandenen PoC.  ConvConf_Perspektive

8) Interlis-Anbindung: zwei sinnvolle Varianten

Variante A — KM als internes JSON/YAML-Meta-Modell (schnellster Start)

Wir modellieren das Vereins-KM (und später weitere KMs) zunächst in einem eigenen Metaformat.

Vorteil: maximal schnell

Nachteil: noch keine echte INTERLIS-Toolchain-Anbindung

Variante B — KM-Import aus INTERLIS (später)

Später Adapter auf bestehende INTERLIS-Werkzeuge / Compiler-Ausgabe.

Vorteil: realitätsnäher

Nachteil: mehr Integrationsaufwand sofort

Meine Empfehlung für den PoC: erst A, dann B.

9) Wie gut ich das praktisch liefern könnte (mit Annahmen)

Mit deinen erlaubten Annahmen kann ich sehr zielgerichtet arbeiten und würde die Erfolgschance so einschätzen:

- **Parser + AST für den Beispiel-Sprachteil:** sehr gut
- **Semantik-Checker (LM gegen KM):** gut bis sehr gut
- **In-Memory-Konverter für die Beispiele:** sehr gut
- **Reproduzierbare Testabdeckung aus PDFs:** sehr gut
- **Vollständige Sprachspezifikation / Produktionssemantik:** nur schrittweise, über Tests und echte Fälle

Das ist genau die Art Problem, bei der ein "**spec by examples + validator**"-Ansatz funktioniert.

Die 5 Entscheidungen, die ich als Nächstes von euch wissen möchte (optional, aber hilfreich)

Du hast explizit gesagt, ich darf nachfragen, wenn Entscheidungen nötig sind — das sind die wichtigsten:

1. PoC-Ziel zuerst:

Soll der PoC primär die PDF-Beispiele reproduzieren, oder schon ein reales kleines INTERLIS-Modell (aus eurem Umfeld) unterstützen?

2. KM-Input in Phase 1:

Lieber internes JSON/YAML (schnell) oder gleich INTERLIS-Import (realitätsnäher, langsamer)?

3. Priorität bei Validierung:

Lieber zuerst Konversion läuft, oder zuerst LM-Check + gute Fehlermeldungen? (Ich würde beides parallel machen, aber mit Schwerpunkt wählen.)

4. Zielloberfläche:

Reicht euch fürs Erste eine CLI (check , plan , convert , trace), oder wollt ihr früh eine kleine REST-API?

5. Semantik-Politik bei Unklarheiten:

Lieber strikt (Fehler früh) oder tolerant (Warnungen, weiterlaufen)?

Wenn du möchtest, gehe ich als Nächstes einen Schritt tiefer und mache dir eine **konkrete Java-Struktur mit Klassen-/Interface-Skeletten** (inkl. ConversionPlan , LmChecker , JoinStep , InspectionStep , TraceEvent) und eine **erste ANTLR-Grammatik für den sichtbaren LM-Teil.**

Sources

- 1) PDF-Beispiele; 2) INTERLIS-Import; 3) Lieber zuerst Konversion läuft; 4) CLI, REST-API können wirklich erst ganz am Schluss machen 5) strikt.

Thought for 31s ›

Perfekt — damit ist der Rahmen jetzt **klar genug, um wirklich zu bauen.**

Deine Entscheidungen ergeben ein sehr sinnvolles PoC-Profil:

- **Ziel:** zuerst die **PDF-Beispiele** reproduzieren
- **KM-Input:** **INTERLIS-Import**
- **Priorität:** **Konversion** läuft zuerst
- **Interface:** **CLI**
- **Semantikpolitik:** **strikt** (Fehler früh, nicht raten)

Das ist eine starke Kombination, weil sie fachlich sauber ist und trotzdem schnell zu einem sichtbaren Ergebnis führt.

Was ich daraus konkret ableite

Ziel von Phase 1

Ein Java-CLI-PoC, der:

1. ein INTERLIS-KM lädt (über Import),
2. ein LM parst,
3. eine Konversion für die PDF-Beispiele ausführt,

4. dabei in **strict mode** harte Fehler wirft,
5. und reproduzierbare Tests hat.

Wichtig: Weil du „Konversion läuft zuerst“ gewählt hast, bauen wir den LM-Checker **minimal aber strikt** (Gatekeeper), und erweitern ihn danach.

Architektur für deinen gewählten Pfad

1) Module (konkret, auf deinen Entscheid abgestimmt)

`convconf-cli`

CLI-Kommandos (`convert` , `check-lm` , `trace` , später `plan`)

`convconf-interlis-import`

KM-Import aus INTERLIS (Adapter-Schicht)

- liest INTERLIS-Modelle ein
- überführt sie in ein internes KM-Metamodell

`convconf-km`

Internes, tool-unabhängiges KM-Metamodell

- Klassen, Attribute, Strukturen, Enums, Referenzen, Vererbung

`convconf-lm-parser`

LM-Lexer/Parser + AST (ANTLR empfohlen)

`convconf-lm-semantic`

AST → semantisches LM-Modell

(minimale strikte Checks für Phase 1)

`convconf-runtime`

In-Memory-Konverter (Operatoren + Evaluator)

`convconf-io-simpleline`

Ein-/Ausgabe im einfachen PoC-Zeilenformat (für PDF-Beispiele)

convconf-tests

Golden Tests / Negativtests / Parser-Tests

2) Priorisierte Reihenfolge (damit „Konversion läuft zuerst“ wirklich klappt)

Phase A — Lauffähiger Konverter für einfache Beispiele

Zuerst implementieren:

- DATA -Definitionen
- einfache Feldzuordnung
- VALUEMAP
- <-> / <- (nur als Richtungs-Metadatum)
- einfaches Lesen/Schreiben der Testdaten

Damit bekommen wir sehr schnell erste End-to-End-Ergebnisse.

Phase B — Struktur/Relation-Komplexität

Dann:

- ANNEXE / ANNEXED
- JOIN (inkl. ?)
- NESTING
- INSPECTION ... ALL
- \$CLASS , \$PARENT , \$STRUCTATTR

Das deckt die spannendsten PDF-Fälle ab.

Phase C — Strikte Checks ausbauen

Danach gezielt:

- LM-semantische Prüfung
 - Datenvalidierung gegen LM
 - bessere Fehlermeldungen
 - Tracing
-

Strikte Semantikpolitik (dein Entscheid 5)

Das ist sehr gut. Ich würde im PoC standardmäßig so fahren:

Strict by default

- unbekanntes Attribut/Klasse → **Fehler**
- VALUemap unvollständig bei vorkommendem Wert → **Fehler**
- Join-Mehrdeutigkeit (wenn 1 erwartet, aber n gefunden) → **Fehler**
- Feldkollision bei ANNEXED → **Fehler**
- ungültige Referenz → **Fehler**
- nicht unterstütztes Sprachkonstrukt → **Fehler** (nicht still ignorieren)

Optional später:

- --lenient als Debug-/Explorationsmodus

Für einen PoC mit unvollständiger Spezifikation ist das genau richtig, weil sonst versteckte Fehlinterpretationen entstehen.

Java-Klassenskelett (konkret, startbar)

1) Internes KM-Metamodell

«» Java



```
import java.util.*;  
  
public final class KmModel {  
    private final String name;  
    private final Map<String, KmType> typesByName;  
  
    public KmModel(String name, Map<String, KmType> typesByName) {  
        this.name = Objects.requireNonNull(name);  
        this.typesByName = Map.copyOf(typesByName);  
    }  
  
    public String name() { return name; }  
    public Optional<KmType> findType(String qname) { return Optional.ofNull  
    public Map<String, KmType> types() { return typesByName; }  
}
```

<> Java



```
public sealed interface KmType permits KmClassType, KmStructureType, KmEnumType {
    String qname();
}
```

<> Java



```
import java.util.*;

public final class KmClassType implements KmType {
    private final String qname;
    private final boolean isAbstract;
    private final String superClassQName; // nullable
    private final Map<String, KmAttribute> attributes;

    public KmClassType(String qname, boolean isAbstract, String superClassQName) {
        this.qname = qname;
        this.isAbstract = isAbstract;
        this.superClassQName = superClassQName;
        this.attributes = Map.copyOf(attributes);
    }

    @Override public String qname() { return qname; }
    public boolean isAbstract() { return isAbstract; }
    public Optional<String> superClassQName() { return Optional.ofNullable(superClassQName); }
    public Map<String, KmAttribute> attributes() { return attributes; }
}
```

<> Java



```
public record KmAttribute(
    String name,
    KmValueType type,
    boolean mandatory
) {}
```

</> Java



```
package ch.example.convconf.km;

public sealed interface KmValueType permits KmScalarRef, KmEnumRef, KmClassRef, KmStructureRef {
    public record KmScalarRef(String scalarQName) implements KmValueType {}
    public record KmEnumRef(String enumQName) implements KmValueType {}
    public record KmClassRef(String classQName) implements KmValueType {}
    public record KmStructureRef(String structQName) implements KmValueType {}
```

2) INTERLIS-Import-Adapter (wichtig wegen Entscheid 2)

</> Java



```
import ch.example.convconf.km.KmModel;
import java.nio.file.Path;

public interface InterlisKmImporter {
    KmModel importModel(Path iliFileOrModelRepo, String modelName) throws
}
```

</> Java



```
public final class KmImportException extends Exception {
    public KmImportException(String message, Throwable cause) { super(message, cause); }
    public KmImportException(String message) { super(message); }
}
```

Pragmatische Annahme für Phase 1:

Wir kapseln die konkrete INTERLIS-Bibliothek vollständig in diesem Modul. Dann bleibt der Rest des PoC stabil, auch wenn sich die Import-Implementierung ändert.

3) LM-Parser und semantisches Modell

</> Java



```
package ch.example.convconf.lm;

import java.util.List;

public final class LmModel {
    private final String name;
    private final List<DataSetDecl> dataSets;
    private final List<ViewDecl> views;
    private final List<ValueMapDecl> valueMaps;
    private final List<ConversionDecl> conversions;

    public LmModel(String name,
                   List<DataSetDecl> dataSets,
                   List<ViewDecl> views,
                   List<ValueMapDecl> valueMaps,
                   List<ConversionDecl> conversions) {
        this.name = name;
        this.dataSets = List.copyOf(dataSets);
        this.views = List.copyOf(views);
        this.valueMaps = List.copyOf(valueMaps);
        this.conversions = List.copyOf(conversions);
    }

    public String name() { return name; }
    public List<DataSetDecl> dataSets() { return dataSets; }
    public List<ViewDecl> views() { return views; }
    public List<ValueMapDecl> valueMaps() { return valueMaps; }
    public List<ConversionDecl> conversions() { return conversions; }
}
```

</> Java



```
public enum MappingDirection {
    BIDIRECTIONAL, // <->
    OUTPUT_ONLY    // <-
}
```

</> Java



```
package ch.example.convconf.lm;

public record ValueMapEntry(String sourceLiteral, String targetLiteral) {}
```

« Java



```
import java.util.List;

public record ValueMapDecl(String name, List<ValueMapEntry> entries) {}
```

4) Minimaler strikter LM-Checker (Gatekeeper)

« Java



```
import ch.example.convconf.km.KmModel;
import ch.example.convconf.lm.LmModel;
import java.util.List;

public interface LmChecker {
    CheckReport check(KmModel km, LmModel lm);
}
```

« Java



```
package ch.example.convconf.check;

import java.util.List;

public final class CheckReport {
    private final List<Diagnostic> diagnostics;

    public CheckReport(List<Diagnostic> diagnostics) {
        this.diagnostics = List.copyOf(diagnostics);
    }

    public List<Diagnostic> diagnostics() { return diagnostics; }

    public boolean hasErrors() {
        return diagnostics.stream().anyMatch(d -> d.severity() == Severity);
    }

    public void throwIfErrors() {
        if (hasErrors()) throw new LmCheckException(this);
    }
}
```

« Java



```
public record Diagnostic(
    Severity severity,
    String code,
    String message,
    SourceSpan span
) {}
```

« Java



```
public enum Severity { ERROR, WARNING }
```

« Java



```
public record SourceSpan(String file, int line, int column) {}
```

Für deinen „strict“-Wunsch: In Phase 1 behandeln wir viele Dinge direkt als ERROR , auch wenn man sie später evtl. nur warnen würde.

5) Runtime / Konversion

«» Java



```
import ch.example.convconf.km.KmModel;
import ch.example.convconf.lm.LmModel;

public interface ConverterEngine {
    ConversionResult convert(KmModel km, LmModel sourceLm, LmModel targetLm
        throws ConversionException;
}
```

«» Java



```
public record ConvertOptions(
    boolean strictMode,
    boolean traceEnabled
) {
    public static ConvertOptions strict() { return new ConvertOptions(true);
}
```

«» Java



```
import java.util.List;
import java.util.Map;

public final class DataSet {
    private final Map<String, RecordTable> tables;
    public DataSet(Map<String, RecordTable> tables) { this.tables = Map.co
    public Map<String, RecordTable> tables() { return tables; }
}
```

```
«» Java
```



```
package ch.example.convconf.runtime;  
  
import java.util.List;  
  
public record RecordTable(String name, List<Record> records) {}
```

```
«» Java
```



```
import java.util.Map;  
  
public record Record(String typeName, Map<String, Value> fields) {}
```

```
«» Java
```



```
public sealed interface Value permits TextValue, IntValue, RefValue, NullValue  
public record TextValue(String value) implements Value {}  
public record IntValue(long value) implements Value {}  
public record RefValue(String targetType, String id) implements Value {}  
public record NullValue() implements Value {}  
public record MetaValue(String kind, String value) implements Value {}
```

6) Trace (extrem nützlich bei PoC)

```
«» Java
```



```
package ch.example.convconf.runtime.trace;

public record TraceEvent(
    String targetRecordType,
    String targetField,
    String targetRecordId,
    String operation,      // e.g. "VALUEMAP", "JOIN", "COPY"
    String sourceExpr,
    String sourceValue,
    String resultValue
) {}
```

Damit kannst du bei Fehlern sofort sehen:

- *woher* ein Wert kam,
- *welche Regel* angewandt wurde,
- *warum* er evtl. fehlt.

Gerade bei JOIN/NESTING/INSPECTION ist das Gold wert.

CLI-Design (dein Entscheid 4)

Ich würde **nur** CLI bauen, aber so, dass eine REST-API später einfach aufsetzen kann.

Kommandos v1

check-lm

Prüft LM gegen KM (strict)

«» Bash



```
--km model.ili --km-model Verein \
--lm source.lm
```

convert

Konvertiert Daten von LMs nach LMe

«» Bash



```
convconf convert \
--km model.ili --km-model Verein \
--lm-source source.lm \
--lm-target target.lm \
--in source-data.txt \
--out target-data.txt \
--strict
```

trace

Wie convert , aber mit Trace-Log

«» Bash



```
--km model.ili --km-model Verein \
--lm-source source.lm \
--lm-target target.lm \
--in source-data.txt \
--trace-out trace.json
```

(später) validate-data

Daten gegen LM prüfen

ANTLR-Grammatik-Startpunkt (sichtbarer LM-Teil)

Nicht vollständig, aber gut genug zum Loslegen:

«» antlr



```
grammar Lm;

lmFile      : statement* EOF ;

statement
: dataDecl
| valueMapDecl
| conversionDecl
| mappingDecl
;

dataDecl      : 'DATA' IDENT fieldDecl* 'END' ;
fieldDecl     : IDENT typeRef fieldModifier* ;
fieldModifier  : 'MANDATORY'
| 'NESTING'
;

mappingDecl   : path mappingOp viewExpr ;
mappingOp     : '<->' | '<-' ;

valueMapDecl  : 'VALUemap' IDENT valueMapEntry+ 'END' ;
valueMapEntry : literal '->' literal ;

conversionDecl : 'CONVERSION' IDENT '(' IDENT ')' '->' IDENT ; // PoC-ve

viewExpr
: sourceExpr withClause? joinClause* annexClause* inspectionClause? ne
;

sourceExpr    : IDENT ('.' IDENT)* ;
withClause    : 'WITH' IDENT ;
joinClause    : 'JOIN' ('?' )? sourceExpr 'ON' joinCond ;
joinCond      : pathRef '=' pathRef ;

annexClause   : 'ANNEXE' IDENT
| 'ANNEXED' IDENT
;

inspectionClause : 'INSPECTION' IDENT 'ALL' ;
nestingClause  : 'NESTING' IDENT ;

path          : IDENT ('.' IDENT)* ;
pathRef       : ('$' IDENT) | path ;

typeRef       : IDENT ('.' IDENT)* ;
literal        : STRING | INT | IDENT ;

IDENT         : [A-Za-z_][A-Za-z0-9_]* ;
```

```
INT          : [0-9]+ ;  
STRING       : '\'' (~['\r\n'] | '\'\'')* '\'' ;  
WS           : [\t\r\n]+ -> skip ;  
LINE_COMMENT : '//' ~[\r\n]* -> skip ;
```

Wichtige Anmerkung dazu

Die Grammatik oben ist **bewusst nur ein Startpunkt**.

Für die PDF-Beispiele werden wir sie beim Implementieren schrittweise erweitern (z. B. Spezialfelder, Aliasing, evtl. konkretere Mapping-Ausdrücke).

Wie wir deinen Wunsch „Konversion läuft zuerst“ praktisch umsetzen (ohne Chaos)

Ich würde Feature-by-feature entlang der PDF-Beispiele bauen.

Iterationsplan (sehr konkret)

Iteration 1 — „Hello Conversion“

- INTERLIS-KM importieren (Vereinsmodell)
- LM parsen (einfacher Fall)
- Daten lesen
- 1:1 Feldkopie + VALUemap
- Daten schreiben

Ergebnis: erstes End-to-End-Beispiel läuft

Iteration 2 — Vererbung / Klassenmapping

- \$CLASS
- flaches Modell → Klassen Person/Organisation
- strict checks auf Klassendiskriminierung

Ergebnis: das „flache Mitglieder“-Beispiel wird realistisch

Iteration 3 — ANNEXE / ausgelagerte Daten

- ausgelagerte OID-Tabelle
- Zuordnung via Owner/ID
- Kollisionen streng behandeln

Ergebnis: OID-/Annexe-Beispiel abgedeckt

Iteration 4 — JOIN / Output-only Views

- JOIN
- JOIN ? als Left Outer Join
- <- strikt als output-only markieren

Ergebnis: Personen+Organisation-Output Beispiel

Iteration 5 — NESTING + INSPECTION

- Struktur-/Nesting-Fälle
- \$PARENT , \$STRUCTATTR , \$CLASS
- Trace obligatorisch aktivierbar

Ergebnis: die technisch schwierigsten Beispiele aus dem PDF

Iteration 6 — härterer Checker + Negativtests

- LM-Checks ausbauen
- Datenvalidator gegen LM
- gezielte Fehlerfälle

Was ich von euch (optional) als Zusatzwissen sehr gut gebrauchen könnte

Du hast erlaubt, dass ich bei sinnvollen Varianten nachfrage. Für den Start ist schon genug entschieden, aber diese 3 Dinge würden stark helfen:

1. Welches INTERLIS-Import-Werkzeug / welche Bibliothek ist bei euch gesetzt?

(Falls schon ein Standard existiert, spare ich Integrationsentscheidungen.)

2. Wollt ihr Java 17 oder Java 21?

(Ich würde **Java 21** empfehlen, wenn möglich; ansonsten Java 17 ist völlig okay.)

3. Build-Tool: Maven oder Gradle?

(Ich würde für so ein Tooling-Projekt eher **Maven** nehmen, wenn nichts dagegen spricht.)

Wenn ihr dazu nichts vorgebt, kann ich pragmatisch annehmen:

- **Java 21**
 - **Maven**
 - **JUnit 5 + AssertJ**
 - **ANTLR 4**
-

Mein nächster sinnvoller Schritt

Jetzt, mit deinen Entscheidungen, würde ich als Nächstes liefern:

1. eine konkrete **Repo-Struktur** (Maven Multi-Module),
2. die ersten **Interfaces + Datenklassen** (wie oben, vollständiger),
3. eine erste ANTLR-Grammatik v0, die 1–2 PDF-Beispiele parsebar macht,
4. einen **End-to-End Test-Case** (convert über ein Minimalbeispiel).

Wenn du willst, mache ich im nächsten Schritt direkt ein **Maven-Projektgerüst mit Paketstruktur und Skelettcode** (so, dass ihr es praktisch sofort starten könnt).

Zwischenfrage: das logische Modell würde z.B. meinen Tabellen in einer PostgreSQL-Datenbank entsprechen?

Thought for 16s >

Ja — genau, oft wäre das so. 

Kurz gesagt:

Das **LM (logische Modell)** kann sehr gut den **Tabellen / Views / Beziehungen** deiner **PostgreSQL-Datenbank** entsprechen.

Wichtige Präzisierung

Es ist **nicht zwingend** einfach „1:1 das physische DB-Schema“, sondern eher:

- die **logische Sicht** auf deine Daten,
- also das, was für den Datenaustausch fachlich relevant ist.

Das kann beinhalten:

- Tabellen
- Foreign Keys / Beziehungstabellen
- Views (z. B. JOINs)
- Codierungen (z. B. 1 , 2 , 3 für Enum-Werte)
- aufgeteilte Strukturen (z. B. Kommentare in separater Tabelle)
- ggf. technische IDs / OIDs in separaten Tabellen

In deinem PostgreSQL-Beispiel heißt das praktisch

Wenn du z. B. hast:

- person
- organisation
- member
- person_org
- comment
- event

...dann wäre das ein typischer Kandidat für ein LM.

ConvConf würde dann (vereinfacht gesagt) beschreiben:

- wie diese Tabellen **zum KM passen**
- welche Tabelle welcher KM-Klasse entspricht
- wie Codes gemappt werden
- wie Beziehungen/Joins interpretiert werden
- wie verschachtelte Strukturen aus flachen Tabellen entstehen (oder umgekehrt)

Was **nicht** primär Teil des LM ist

Eher **nicht** (oder nur indirekt) LM-Thema sind typische physische DB-Details wie:

- Indexe

- Partitionierung
- Storage-Parameter
- Performance-Tuning
- Tabellenräume

Das sind eher **physisches Modell / Betriebsaspekte**.

Merksatz

- **KM** = fachliche Semantik (INTERLIS-Welt)
- **LM** = logische Struktur deines Systems (z. B. PostgreSQL-Schema/Views)
- **DB physisch** = technische Umsetzung/Optimierung

Wenn du willst, kann ich dir als Nächstes ein **Mini-Beispiel „PostgreSQL-Schema → mögliches LM → KM“** skizzieren (z. B. mit person , organisation , person_org und Enum-Codes).