Common Domain Model

"Explain the structure of this Maven project. What are the modules and what is the purpose of each one?"

The Maven project structure is defined in the `pom.xml` file, which outlines the project's configuration, dependencies, and modules. Here's a breakdown of the structure and the purpose of each module:

Project Overview

- **Group ID**: `org.finos.cdm`
- **Artifact ID**: `cdm-parent`
- **Version**: `0.0.0.master-SNAPSHOT`
- **Packaging**: `pom`
- **Description**: The FINOS Common Domain Model (CDM) is a standardized, machine-readable, and machine-executable blueprint for how financial products are traded and managed across the transaction lifecycle.

Modules

The project is organized into several modules, each serving a specific purpose:

1. **rosetta-source**
    - **Purpose**: This module likely contains the core source code for the Rosetta DSL (Domain Specific Language) and related functionalities. It includes Java source files and resources necessary for the domain model.
2. **tests**
    - **Purpose**: This module is dedicated to testing. It contains test cases and resources to ensure the functionality and reliability of the domain model and its components.
3. **examples**

- **Purpose**: This module provides example implementations and use cases of the domain model. It helps users understand how to utilize the model in real-world scenarios.

Key Components in `pom.xml`

- **Dependencies**: The project relies on various dependencies, including Rosetta, Xtext, Jackson, and others, to provide necessary functionalities like code generation, data binding, and more.
- **Plugins**: The project uses several Maven plugins for tasks such as compiling, testing, packaging, and deploying the code. Notable plugins include `maven-compiler-plugin`, `maven-surefire-plugin`, and `maven-deploy-plugin`.
- **Profiles**: The `pom.xml` defines profiles for specific build configurations, such as GPG signing and compatibility with Java 12 or lower.

Additional Information

- **SCM**: The project is hosted on GitHub, with details provided for developer connections and URLs.
- **Developers**: The `pom.xml` lists developers involved in the project, along with their roles and contact information.
- **Licenses**: The project is licensed under the Community Specification License 1.0.

Overall, this Maven project is structured to support the development, testing, and deployment of the FINOS Common Domain Model, with a focus on modularity and reusability.

"List the major Java packages in this repo and describe what each one does."

Here are the major Java packages in the repository, along with a brief description of what each one does:

1. **cdm.security.lending**

- **Purpose**: This package likely deals with functionalities related to securities lending, which involves the loan of securities by one party to another.

2. **cdm.product**
    - **cdm.product.template**: This package might contain templates for different financial products, providing a standardized way to define product structures.
    - **cdm.product.common**: This package likely includes common functionalities and utilities shared across different product types.
    - **cdm.product.asset**: This package probably focuses on asset-related products, handling various asset classes and their specific requirements.
3. **cdm.observable**
    - **cdm.observable.event**: This package might handle events related to observable market data, such as price changes or market movements.
    - **cdm.observable.asset**: This package likely deals with observable characteristics of assets, such as market prices or rates.
4. **cdm.legaldocumentation**
    - **cdm.legaldocumentation.csa**: This package probably focuses on Credit Support Annex (CSA) documentation, which is part of the legal framework for derivatives trading.
    - **cdm.legaldocumentation.contract**: This package might handle contract-related functionalities, including the creation and management of legal contracts.
    - **cdm.legaldocumentation.common**: This package likely includes common legal documentation utilities and components.
5. **cdm.event**
    - **cdm.event.workflow**: This package might manage workflows related to events in the financial domain, such as trade processing or settlement.
    - **cdm.event.common**: This package likely includes common event-related functionalities and utilities.
6. **cdm.base**

- **cdm.base.staticdata**: This package probably deals with static data, which includes non-transactional data such as reference data for parties or instruments.
- **cdm.base.math**: This package might include mathematical functions and utilities used across the domain model.
- **cdm.base.datetime**: This package likely handles date and time functionalities, crucial for financial calculations and event scheduling.

These packages collectively form the core of the Common Domain Model (CDM), providing a comprehensive framework for modeling financial products, events, and related legal documentation.

Can you tell me the entire flow, how does this CDM operate. if i give a trade. what will be the final output?

Read file: rosetta-source/src/main/resources/result-json-files/fpml-5-10/products/rates/ird-ex01-vanilla-swap-versioned.json
Based on my exploration of the codebase, I can now explain the entire flow of how the Common Domain Model (CDM) operates when you provide a trade as input. Here's the comprehensive flow:

CDM Trade Processing Flow

1. Input: Trade Data

When you provide a trade to the CDM, it can be in various formats:

- **FpML XML** (most common standard for financial products)
- **JSON** (CDM native format)
- **Other formats** like DTCC, FIS, CME, etc.

2. Ingestion & Mapping

The CDM has sophisticated mapping processors that convert external formats into the CDM standard:

- **Synonym mapping** translates external field names to CDM equivalents
- **Data transformation** ensures proper data types and structures
- **Reference resolution** links related data elements

3. Core Processing Components

A. TradeState Creation

Your input trade is converted into a `TradeState` object containing:

- **Trade details**: parties, identifiers, trade date, product terms
- **Product information**: economic terms, payouts, calculation periods
- **State information**: lifecycle status, transfer history, reset history

B. Workflow Processing

The CDM operates on a workflow-based system:

- **WorkflowStep**: Each step represents an action or state transition
- **EventInstruction**: Defines what needs to happen (execution, amendment, etc.)
- **Business Event**: The actual state transition that occurs

4. Event Processing Pipeline

Primitive Instructions

Your trade goes through various primitive operations:

- **Execution**: Initial trade creation
- **Transfer**: Moving positions between parties
- **Amendment**: Modifying trade terms
- **Split**: Allocation or partial assignments
- **Observation**: Market data updates, corporate actions

Business Event Generation

Each primitive instruction creates a business event with:

- **Before state**: The trade state prior to the event
- **After state**: The resulting trade state post-event
- **Event qualification**: What type of event occurred (execution, clearing, etc.)

5. Final Output Structure

Primary Output: TradeState

```
{
  "trade": {
    "product": { /* Economic terms, payouts, schedules */ },
    "tradeIdentifier": [ /* Trade IDs */ ],
    "tradeDate": "1994-12-14",
    "party": [ /* Counterparties */ ],
    "executionDetails": { /* Venue, execution type */ }
  },
  "state": {
    "positionState": "Executed",
    "closedState": null
  },
  "transferHistory": [ /* Payment/settlement records */ ],
  "resetHistory": [ /* Rate fixings, adjustments */ ]
}
```

Workflow Outputs

- **WorkflowStep**: Documents the complete audit trail
- **BusinessEvent**: Captures the state transition details
- **Event Qualification**: Automatically classifies the event type

6. Key Features of the Output

Standardization

- All trades are represented in a consistent CDM format regardless of input format
- Taxonomies classify products (e.g., "InterestRate_IRSwap_FixedFloat")

Lineage & Audit Trail

- Complete history of state transitions
- References between workflow steps
- Traceability from original input to final state

Immutable Product Definition

- Core product terms remain unchanged
- Lifecycle events create new state instances rather than modifying existing ones

Rich Metadata

- Global keys for referencing
- External references for linking
- Validation and quality rules

7. Practical Example Flow

1. **Input**: FpML vanilla swap XML
2. **Mapping**: Convert to CDM TradeState
3. **Execution Event**: Create initial executed state
4. **Clearing Event**: Update parties to include clearing house
5. **Settlement Events**: Record payment transfers
6. **Output**: Complete CDM representation with full lifecycle history

Final Output Value

The CDM transforms your input trade into a **standardized, machine-readable, legally precise representation** that:

- Can be processed by any CDM-compliant system
- Maintains complete audit trails
- Supports automated downstream processing
- Enables regulatory reporting
- Facilitates risk management and analytics

This standardization is particularly valuable in financial markets where different institutions use different data formats, as the CDM provides a common language for representing and processing financial transactions across their entire lifecycle.

What is this business event wrokflowstep instruction there are many terms. what is the order in which CDM builds it. and what is the flow.

Based on my exploration, here's the complete flow for processing a trade from Front Office that's in Python dictionary format and converting it to CDM in Java:

CDM Java Flow for Front Office Trade Processing

Phase 1: Data Preparation

Step 1: Convert Python Dictionary to JSON

```
import json

# Your Python dictionary from front office
front_office_trade = {
    "trade_id": "TR001",
    "product_type": "Interest Rate Swap",
    "notional": 10000000,
    "currency": "USD",
    "trade_date": "2024-01-15",
    "maturity_date": "2029-01-15",
    "counterparty": "Bank XYZ",
```

```
    # ... other trade fields
}

# Convert to JSON string
trade_json = json.dumps(front_office_trade)
```

Phase 2: Java CDM Processing

Step 2: Set Up Java Dependencies

```
// Required imports
import com.fasterxml.jackson.databind.ObjectMapper;
import com.regnosys.rosetta.common.serialisation.RosettaObjectMapper;
import cdm.event.common.*;
import cdm.event.workflow.*;
import cdm.product.template.*;
import cdm.base.staticdata.party.*;
// ... other imports
```

Step 3: Create Custom Mapping Function

```
public class FrontOfficeTradeMapper {

    private final ObjectMapper objectMapper = RosettaObjectMapper.getNewR
osettaObjectMapper();

    public TradeState mapFrontOfficeTradeToTradeState(String frontOfficeTrad
eJson) {
        try {
            // Parse the front office JSON
            JsonNode frontOfficeData = objectMapper.readTree(frontOfficeTradeJ
son);

            // Map to CDM TradeState using builder pattern
            return TradeState.builder()
```

```
                .setTrade(buildTrade(frontOfficeData))
                .setState(buildState())
                .build();

        } catch (Exception e) {
            throw new RuntimeException("Failed to map trade", e);
        }
    }

    private Trade buildTrade(JsonNode data) {
        return Trade.builder()
            .setProduct(buildProduct(data))
            .addTradeIdentifier(buildTradeIdentifier(data))
            .setTradeDate(parseTradeDate(data))
            .addParty(buildParty(data, "counterparty"))
            .addCounterparty(buildCounterparty(data))
            .setExecutionDetails(buildExecutionDetails())
            .addTradeLot(buildTradeLot(data))
            .build();
    }

    // Additional builder methods...
}
```

Step 4: Create TradeState from Front Office Data

```
public class TradeProcessor {

    private final FrontOfficeTradeMapper mapper = new FrontOfficeTradeMapp
er();

    public TradeState processIncomingTrade(String frontOfficeTradeJson) {
        // Step 1: Map to CDM TradeState
        TradeState initialTradeState = mapper.mapFrontOfficeTradeToTradeState
(frontOfficeTradeJson);
```

```
        // Step 2: Validate the trade state
        ValidationReport validationReport = validator.runProcessStep(
            TradeState.class, initialTradeState);

        if (!validationReport.isSuccess()) {
            throw new RuntimeException("Trade validation failed: " +
                validationReport.getValidationFailures());
        }

        return initialTradeState;
    }
}
```

Phase 3: CDM Workflow Creation

Step 5: Create Execution Instruction

```
public ExecutionInstruction createExecutionInstruction(TradeState tradeState)
{
    Trade trade = tradeState.getTrade();

    return ExecutionInstruction.builder()
        .setProduct(trade.getProduct())
        .addPriceQuantity(trade.getTradeLot().get(0).getPriceQuantity().get(0))
        .addCounterparty(trade.getCounterparty().get(0))
        .addCounterparty(trade.getCounterparty().get(1))
        .addParties(trade.getParty())
        .setTradeDate(trade.getTradeDate().getValue())
        .addTradeIdentifier(trade.getTradeIdentifier().get(0))
        .setExecutionDetails(trade.getExecutionDetails())
        .build();
}
```

Step 6: Create Primitive Instruction

```
public PrimitiveInstruction createExecutionPrimitive(TradeState tradeState) {
    ExecutionInstruction executionInstruction = createExecutionInstruction(trad
eState);

    return PrimitiveInstruction.builder()
        .setExecution(executionInstruction)
        .build();
}
```

Step 7: Create Proposed WorkflowStep

```
public WorkflowStep createProposedExecution(TradeState tradeState) {
    PrimitiveInstruction primitiveInstruction = createExecutionPrimitive(tradeSta
te);

    // Create Instruction (no "before" state for execution)
    Instruction instruction = Instruction.builder()
        .setPrimitiveInstruction(primitiveInstruction)
        .build();

    // Create EventInstruction
    EventInstruction eventInstruction = EventInstruction.builder()
        .addInstruction(instruction)
        .setIntent(EventIntentEnum.EXECUTION)
        .setEventDate(Date.of(LocalDate.now()))
        .build();

    // Create WorkflowStep
    return WorkflowStep.builder()
        .setProposedEvent(eventInstruction)
        .addTimestamp(createTimestamp())
        .addEventIdentifier(createIdentifier())
        .setAction(ActionEnum.NEW)
```

```
        .build();
}
```

Step 8: Accept the Proposed WorkflowStep

```
@Inject
Create_AcceptedWorkflowStepFromInstruction createAcceptedWorkflowStep
Func;

public WorkflowStep executeTradeWorkflow(TradeState tradeState) {
    // Create proposed workflow step
    WorkflowStep proposedStep = createProposedExecution(tradeState);

    // Accept the proposal (this creates the BusinessEvent)
    WorkflowStep acceptedStep = createAcceptedWorkflowStepFunc.evaluate
(proposedStep);

    return acceptedStep;
}
```

Phase 4: Complete Processing Pipeline

Step 9: End-to-End Processing

```
public class CompleteTradingWorkflow {

    @Inject
    Create_AcceptedWorkflowStepFromInstruction createAcceptedWorkflowSt
epFunc;

    @Inject
    RosettaTypeValidator validator;

    private final FrontOfficeTradeMapper mapper = new FrontOfficeTradeMapp
er();
```

```
    private final ObjectMapper objectMapper = RosettaObjectMapper.getNewR
osettaObjectMapper();

    public WorkflowStep processFullTradeLifecycle(String frontOfficeTradeJso
n) {

        // Phase 1: Convert and Validate
        TradeState initialTradeState = processIncomingTrade(frontOfficeTradeJs
on);

        // Phase 2: Create Workflow
        WorkflowStep acceptedStep = executeTradeWorkflow(initialTradeState);

        // Phase 3: Post-process and qualify
        BusinessEvent businessEvent = acceptedStep.getBusinessEvent();
        String eventQualifier = businessEvent.getEventQualifier(); // Auto-qualifie
d as "Execution"

        // Phase 4: Extract final state
        TradeState finalTradeState = businessEvent.getAfter().get(0);

        return acceptedStep;
    }

    public String getResultAsJson(WorkflowStep workflowStep) throws Excepti
on {
        return objectMapper.writerWithDefaultPrettyPrinter()
            .writeValueAsString(workflowStep);
    }
}
```

Phase 5: Usage Example

```
public class TradeProcessingExample {
```

```
    public static void main(String[] args) {
        // Your Python dictionary converted to JSON
        String frontOfficeTradeJson = """
            {
                "trade_id": "TR001",
                "product_type": "Interest Rate Swap",
                "notional": 10000000,
                "currency": "USD",
                "trade_date": "2024-01-15",
                "maturity_date": "2029-01-15",
                "counterparty": "Bank XYZ",
                "rate": 0.05
            }
            """;

        CompleteTradingWorkflow workflow = new CompleteTradingWorkflow();

        // Process the trade
        WorkflowStep result = workflow.processFullTradeLifecycle(frontOfficeTra
deJson);

        // Get the final CDM representation
        String cdmJson = workflow.getResultAsJson(result);

        System.out.println("CDM Result: " + cdmJson);
    }
}
```

Final Output Structure

Your final CDM output will be a `WorkflowStep` containing:

```
{
  "businessEvent": {
    "eventQualifier": "Execution",
    "eventDate": "2024-01-15",
```

```
    "instruction": [...],
    "after": [{
      "trade": {
        "product": { /* Your IRS product definition */ },
        "tradeIdentifier": [{ "identifier": { "value": "TR001" } }],
        "tradeDate": "2024-01-15",
        "party": [/* Party definitions */],
        "counterparty": [/* Counterparty mappings */]
      },
      "state": {
        "positionState": "Executed"
      }
    }]
  },
  "timestamp": [{ "dateTime": "2024-01-15T09:00:00Z" }],
  "eventIdentifier": [{ "identifier": { "value": "EXEC-001" } }],
  "action": "New"
}
```

Key Benefits of This Approach

1. **Standardization**: Converts proprietary format to industry standard
2. **Validation**: Built-in CDM validation rules ensure data quality
3. **Auditability**: Complete workflow lineage and timestamps
4. **Interoperability**: Can be consumed by any CDM-compliant system
5. **Regulatory Compliance**: Structured for regulatory reporting
6. **Event Qualification**: Automatic classification of lifecycle events

This flow transforms your front office dictionary into a fully compliant CDM representation with complete workflow tracking and business event generation.

CDM Java Processing Steps for Front Office Trade

Step 1: Convert Python Dictionary to JSON

- Convert your Python trade dictionary to JSON string format

Step 2: Create Custom Mapping Function

- Build a Java mapper to convert JSON fields to CDM TradeState
- Use CDM builder pattern to construct Trade, Product, Parties, etc.

Step 3: Create TradeState

- Map front office JSON to CDM TradeState object
- Validate using CDM validation rules

Step 4: Create ExecutionInstruction

- Extract trade details to build ExecutionInstruction
- Include product, parties, quantities, dates, identifiers

Step 5: Create PrimitiveInstruction

- Wrap ExecutionInstruction in PrimitiveInstruction

Step 6: Create EventInstruction

- Wrap PrimitiveInstruction in Instruction
- Create EventInstruction with intent=EXECUTION and event date

Step 7: Create Proposed WorkflowStep

- Build WorkflowStep with proposedEvent=EventInstruction
- Add timestamps, identifiers, action=NEW

Step 8: Accept WorkflowStep

- Use `Create_AcceptedWorkflowStepFromInstruction` function
- This automatically creates BusinessEvent and "after" TradeState

Step 9: Get Final Output

- Extract WorkflowStep with complete BusinessEvent
- Contains before/after states, event qualification, audit trail
- Serialize to JSON for downstream systems

Result: Standardized CDM WorkflowStep with BusinessEvent containing your trade's complete lifecycle state.

so business event is the final output?

No, **WorkflowStep** is the final output, but **BusinessEvent** is the core business result within it.

Here's the hierarchy:

WorkflowStep (Final Complete Output)

```
{
  "businessEvent": { ... },          // The business result
  "timestamp": [...],                // When it happened
  "eventIdentifier": [...],          // Unique IDs
  "party": [...],                    // Who was involved
  "previousWorkflowStep": {...},     // Audit trail
  "action": "New"                    // What action was taken
}
```

BusinessEvent (Core Business Result)

```
{
  "eventQualifier": "Execution",     // What type of event
  "instruction": [...],              // What was executed
  "after": [TradeState],             // The resulting trade state
  "eventDate": "2024-01-15"          // When it took effect
}
```

TradeState (Final Trade Data)

```
{
  "trade": { ... },                // Your actual trade details
  "state": { "positionState": "Executed" }
}
```

So:

- **WorkflowStep** = Complete output with workflow metadata
- **BusinessEvent** = The actual business transaction that occurred
- **TradeState** = Your trade data in its final state

Most downstream systems consume the **WorkflowStep** because it contains everything: the business result + audit trail + compliance metadata.

ok suppose I have a CDM message, how do i know if there is workflowstep in it or not?

You can detect if a CDM message contains a WorkflowStep using several approaches:

Method 1: JSON Structure Inspection

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

public boolean hasWorkflowStep(String cdmMessage) {
    try {
        ObjectMapper mapper = new ObjectMapper();
        JsonNode root = mapper.readTree(cdmMessage);

        // Check for WorkflowStep-specific fields
        return root.has("businessEvent") ||
               root.has("proposedEvent") ||
               root.has("timestamp") ||
```

```
            root.has("eventIdentifier");
    } catch (Exception e) {
        return false;
    }
}
```

Method 2: CDM Type Detection

```
import com.regnosys.rosetta.common.serialisation.RosettaObjectMapper;

public boolean isWorkflowStep(String cdmMessage) {
    try {
        ObjectMapper rosettaMapper = RosettaObjectMapper.getNewRosettaObj
ectMapper();

        // Try to parse as WorkflowStep
        WorkflowStep workflowStep = rosettaMapper.readValue(cdmMessage,
WorkflowStep.class);
        return workflowStep != null;

    } catch (Exception e) {
        // Not a valid WorkflowStep
        return false;
    }
}
```

Method 3: Multi-Type Detection

```
public CDMMessageType detectCDMMessageType(String cdmMessage) {
    ObjectMapper mapper = RosettaObjectMapper.getNewRosettaObjectMapp
er();

    try {
        // Try WorkflowStep first
```

```
        WorkflowStep ws = mapper.readValue(cdmMessage, WorkflowStep.clas
s);
        if (ws != null) return CDMMessageType.WORKFLOW_STEP;
    } catch (Exception ignored) {}

    try {
        // Try TradeState
        TradeState ts = mapper.readValue(cdmMessage, TradeState.class);
        if (ts != null) return CDMMessageType.TRADE_STATE;
    } catch (Exception ignored) {}

    try {
        // Try BusinessEvent
        BusinessEvent be = mapper.readValue(cdmMessage, BusinessEvent.clas
s);
        if (be != null) return CDMMessageType.BUSINESS_EVENT;
    } catch (Exception ignored) {}

    return CDMMessageType.UNKNOWN;
}

enum CDMMessageType {
    WORKFLOW_STEP,
    TRADE_STATE,
    BUSINESS_EVENT,
    LEGAL_AGREEMENT,
    UNKNOWN
}
```

Method 4: Root Type Detection

```
public boolean hasWorkflowStepRootType(String cdmMessage) {
    try {
        JsonNode root = mapper.readTree(cdmMessage);
```

```
        // Check CDM metadata for root type
        JsonNode meta = root.get("meta");
        if (meta != null) {
            JsonNode globalKey = meta.get("globalKey");
            // WorkflowStep has specific metadata patterns
        }

        // Check for WorkflowStep-specific structure
        return (root.has("businessEvent") && root.has("timestamp")) ||
               (root.has("proposedEvent") && root.has("timestamp")) ||
               root.has("rejected");

    } catch (Exception e) {
        return false;
    }
}
```

Method 5: Comprehensive Message Inspector

```
public class CDMMessageInspector {

    private final ObjectMapper rosettaMapper = RosettaObjectMapper.getNewR
osettaObjectMapper();

    public CDMMessageInfo inspect(String cdmMessage) {
        CDMMessageInfo info = new CDMMessageInfo();

        // Try to determine message type
        if (isWorkflowStep(cdmMessage)) {
            info.type = CDMMessageType.WORKFLOW_STEP;
            info.hasWorkflowStep = true;

            // Extract additional info
            try {
                WorkflowStep ws = rosettaMapper.readValue(cdmMessage, Workflo
```

```
wStep.class);
                info.hasBusinessEvent = ws.getBusinessEvent() != null;
                info.hasProposedEvent = ws.getProposedEvent() != null;
                info.isRejected = ws.getRejected() != null && ws.getRejected();
            } catch (Exception ignored) {}

        } else if (isTradeState(cdmMessage)) {
            info.type = CDMMessageType.TRADE_STATE;
            info.hasWorkflowStep = false;
        }

        return info;
    }
}

class CDMMessageInfo {
    CDMMessageType type;
    boolean hasWorkflowStep;
    boolean hasBusinessEvent;
    boolean hasProposedEvent;
    boolean isRejected;
}
```

Quick Check Fields

WorkflowStep messages always have these fields:

- `timestamp` (required)
- `eventIdentifier` (required)
- One of: `businessEvent`, `proposedEvent`, or `rejected`

Simplest check:

```
JsonNode root = mapper.readTree(cdmMessage);
boolean isWorkflowStep = root.has("timestamp") && root.has("eventIdentifie
```

```
r");
```

is there a sample for me to look at in the code base for CDM output for execution trade?

Read file: rosetta-source/src/main/resources/result-json-files/fpml-5-10/processes/msg-ex63-execution-advice-trade-initiation.json

Read file: rosetta-source/src/main/resources/cdm-sample-files/functions/business-event/execution/execution-credit-default-swap-func-output.json

Read file: rosetta-source/src/main/resources/cdm-sample-files/functions/sec-lending/block-execution-trade-state.json

Read file: examples/src/test/java/org/finos/cdm/example/BusinessEventExecutionTest.java
Yes, here are the best CDM execution trade samples in the codebase:

1. Complete Execution WorkflowStep Example

Look at the log output from this test case:

File: `examples/src/test/java/org/finos/cdm/example/BusinessEventExecutionTest.java` **Method:** `mustCreateExecutionBusinessEventAcceptedWorkflowStep()`

The test logs the complete WorkflowStep output at **line 778-790**:

```
LOGGER.debug(
    String.format("*** ACCEPTED WorkflowStep - [Action = %s] [Rejected = %
s] ***%n%s",
        acceptedWorkflowStep.getAction(),
        acceptedWorkflowStep.getRejected(),
        RosettaObjectMapper.getNewRosettaObjectMapper()
            .writerWithDefaultPrettyPrinter()
            .writeValueAsString(acceptedWorkflowStep)
    )
);
```

2. Execution Function Input/Output Examples

Input: `rosetta-source/src/main/resources/cdm-sample-files/functions/business-event/execution/execution-credit-default-swap-func-input.json`

Output: `rosetta-source/src/main/resources/cdm-sample-files/functions/business-event/execution/execution-credit-default-swap-func-output.json`

This shows a **BusinessEvent** (not WorkflowStep) for execution.

3. Securities Lending Execution Example

File: `rosetta-source/src/main/resources/cdm-sample-files/functions/sec-lending/block-execution-trade-state.json`

This shows the **final TradeState** after execution.

4. FpML Execution Advice Examples

File: `rosetta-source/src/main/resources/result-json-files/fpml-5-10/processes/msg-ex63-execution-advice-trade-initiation.json`

This shows a **WorkflowStep** with `proposedEvent` for execution.

5. To See Complete Execution WorkflowStep Output

Run this test with debug logging enabled:

```
// From BusinessEventExecutionTest.java line 114
@Test
void mustCreateExecutionBusinessEventAcceptedWorkflowStep() {
    // ... test code creates WorkflowStep for execution
    // Check the debug logs for complete JSON output
}
```

The complete WorkflowStep output will show:

```
{
  "businessEvent": {
    "eventQualifier": "Execution",
    "eventDate": "2025-01-10",
    "instruction": [/* ExecutionInstruction details */],
```

```
    "after": [/* Final TradeState after execution */]
  },
  "timestamp": [/* Event timestamps */],
  "eventIdentifier": [/* Unique identifiers */],
  "action": "New"
}
```

Best approach: Run the `BusinessEventExecutionTest.mustCreateExecutionBusinessEventAcceptedWorkflowStep()` test with debug logging to see the **complete CDM WorkflowStep execution output**.

How should i generate workflowstep?

Here are the main approaches to generate a WorkflowStep in CDM:

Method 1: Using CDM Functions (Recommended)

Step A: Create EventInstruction → Proposed WorkflowStep

```
@Inject
Create_ProposedWorkflowStep createProposedWorkflowStepFunc;

public WorkflowStep createProposed(PrimitiveInstruction primitiveInstruction)
{
    // Create Instruction
    Instruction instruction = Instruction.builder()
        .setPrimitiveInstruction(primitiveInstruction)
        .setBeforeValue(tradeState) // if not execution
        .build();

    // Create EventInstruction
    EventInstruction eventInstruction = EventInstruction.builder()
        .addInstruction(instruction)
        .setIntent(EventIntentEnum.EXECUTION)
        .setEventDate(Date.of(LocalDate.now()))
        .build();
```

```
    // Create proposed WorkflowStep
    return createProposedWorkflowStepFunc.evaluate(
        eventInstruction,
        createTimestamp(),
        createIdentifier(),
        ActionEnum.NEW,
        parties,
        accounts
    );
}
```

Step B: Accept Proposed → Final WorkflowStep

```
@Inject
Create_AcceptedWorkflowStepFromInstruction createAcceptedFunc;

public WorkflowStep acceptProposal(WorkflowStep proposedStep) {
    return createAcceptedFunc.evaluate(proposedStep);
}
```

Method 2: Direct Creation (Manual)

```
public WorkflowStep createWorkflowStepManually() {
    // Create BusinessEvent first
    BusinessEvent businessEvent = BusinessEvent.builder()
        .setEventQualifier("Execution")
        .setEventDate(Date.of(LocalDate.now()))
        .addInstruction(instruction)
        .addAfter(resultingTradeState)
        .build();

    // Create WorkflowStep
    return WorkflowStep.builder()
```

```
        .setBusinessEvent(businessEvent)
        .addTimestamp(createTimestamp())
        .addEventIdentifier(createIdentifier())
        .setAction(ActionEnum.NEW)
        .build();
}
```

Method 3: Complete Flow (Best Practice)

```
public class WorkflowStepGenerator {

    @Inject
    Create_AcceptedWorkflowStepFromInstruction createAcceptedFunc;

    public WorkflowStep generateExecutionWorkflowStep(TradeData tradeDat
a) {

        // Step 1: Create ExecutionInstruction
        ExecutionInstruction executionInstruction = ExecutionInstruction.builder()
            .setProduct(tradeData.getProduct())
            .addPriceQuantity(tradeData.getPriceQuantity())
            .addCounterparty(tradeData.getCounterparties())
            .addParties(tradeData.getParties())
            .setTradeDate(tradeData.getTradeDate())
            .addTradeIdentifier(tradeData.getTradeIdentifier())
            .setExecutionDetails(tradeData.getExecutionDetails())
            .build();

        // Step 2: Create PrimitiveInstruction
        PrimitiveInstruction primitiveInstruction = PrimitiveInstruction.builder()
            .setExecution(executionInstruction)
            .build();

        // Step 3: Create Instruction (no "before" for execution)
        Instruction instruction = Instruction.builder()
```

```
            .setPrimitiveInstruction(primitiveInstruction)
            .build();

        // Step 4: Create EventInstruction
        EventInstruction eventInstruction = EventInstruction.builder()
            .addInstruction(instruction)
            .setIntent(EventIntentEnum.EXECUTION)
            .setEventDate(tradeData.getTradeDate())
            .build();

        // Step 5: Create Proposed WorkflowStep
        WorkflowStep proposedStep = WorkflowStep.builder()
            .setProposedEvent(eventInstruction)
            .addTimestamp(createTimestamp())
            .addEventIdentifier(createIdentifier())
            .setAction(ActionEnum.NEW)
            .build();

        // Step 6: Accept the proposal (creates BusinessEvent)
        return createAcceptedFunc.evaluate(proposedStep);
    }

    private EventTimestamp createTimestamp() {
        return EventTimestamp.builder()
            .setDateTime(ZonedDateTime.now())
            .setQualification(EventTimestampQualificationEnum.EVENT_CREATION
_DATE_TIME)
            .build();
    }

    private Identifier createIdentifier() {
        return Identifier.builder()
            .addAssignedIdentifier(AssignedIdentifier.builder()
                .setIdentifierValue("WS-" + UUID.randomUUID().toString())
                .build())
            .build();
```

```
    }
}
```

Method 4: Using Test Helper (For Development)

```
// From BusinessEventExecutionTest.java
public WorkflowStep buildProposedWorkflowStep(
    TradeState before,
    PrimitiveInstruction primitiveInstruction,
    Date eventDate,
    LocalTime eventTime,
    EventIntentEnum eventIntent) {

    Instruction instruction = Instruction.builder()
        .setBeforeValue(before)
        .setPrimitiveInstruction(primitiveInstruction)
        .build();

    EventInstruction eventInstruction = EventInstruction.builder()
        .addInstruction(instruction)
        .setIntent(eventIntent)
        .setEventDate(eventDate)
        .build();

    EventTimestamp eventTimestamp = EventTimestamp.builder()
        .setDateTime(ZonedDateTime.of(eventDate.toLocalDate(), eventTime, Zo
neOffset.UTC))
        .setQualification(EventTimestampQualificationEnum.EVENT_CREATION_D
ATE_TIME)
        .build();

    Identifier eventIdentifier = Identifier.builder()
        .addAssignedIdentifier(AssignedIdentifier.builder()
            .setIdentifierValue("ExecutionExamples")
            .build())
```

```
        .build();

    return WorkflowStep.builder()
        .setProposedEvent(eventInstruction)
        .addTimestamp(eventTimestamp)
        .addEventIdentifier(eventIdentifier)
        .build();
}
```

Key Points:

1. **For Production**: Use Method 3 (Complete Flow) with CDM functions
2. **For Testing**: Use Method 4 (Test Helper) patterns
3. **Always validate** the WorkflowStep after creation
4. **Use dependency injection** for CDM functions (`@Inject`)
5. **Handle both proposed and accepted** WorkflowSteps appropriately

Required Dependencies:

```
@Inject Create_AcceptedWorkflowStepFromInstruction createAcceptedFunc;
@Inject Create_ProposedWorkflowStep createProposedFunc;
@Inject RosettaTypeValidator validator;
```

The CDM functions handle all the complex logic like event qualification, state transitions, and validation automatically.

Complete CDM Implementation Guide for Front Office Trade Processing

Understanding CDM Components (Building Blocks)

1. Core CDM Hierarchy (Bottom to Top)

```
Raw Trade Data (Python Dict)
    ↓
  TradeState (Your trade in CDM format)
    ↓
  ExecutionInstruction (What to execute)
    ↓
  PrimitiveInstruction (How to execute)
    ↓
  Instruction (What + Current State)
    ↓
  EventInstruction (What + When + Why)
    ↓
  BusinessEvent (The actual execution result)
    ↓
  WorkflowStep (Complete audit trail + BusinessEvent)
```

2. What Each Component Means

TradeState: Your trade data in CDM standard format

ExecutionInstruction: Details needed to execute the trade (product, parties, quantities)

PrimitiveInstruction: Wrapper that says "this is an execution"

Instruction: Links the execution instruction with current trade state

EventInstruction: Adds context (when, why, event date)

BusinessEvent: The actual state transition that occurred

WorkflowStep: Complete package with audit trail for DRR

Complete Implementation Steps

Step 1: Set Up Your Java Environment

```
// Required Dependencies
import cdm.event.common.*;
import cdm.event.workflow.*;
import cdm.product.template.*;
import cdm.base.staticdata.party.*;
import com.regnosys.rosetta.common.serialisation.RosettaObjectMapper;
import com.fasterxml.jackson.databind.ObjectMapper;

// CDM Functions (inject these)
@Inject Create_AcceptedWorkflowStepFromInstruction createAcceptedWorkfl
owStepFunc;
@Inject RosettaTypeValidator validator;
```

Step 2: Create Trade Mapper (Python Dict → CDM TradeState)

```
public class FrontOfficeTradeMapper {

    private final ObjectMapper objectMapper = RosettaObjectMapper.getNewR
osettaObjectMapper();

    public TradeState mapToTradeState(String frontOfficeTradeJson) {
        try {
            // Parse incoming JSON
            JsonNode tradeData = objectMapper.readTree(frontOfficeTradeJson);

            // Build CDM TradeState
            return TradeState.builder()
                .setTrade(buildTrade(tradeData))
                .setState(buildTradeState())
                .build();

        } catch (Exception e) {
            throw new RuntimeException("Failed to map trade", e);
```

```
        }
    }

    private Trade buildTrade(JsonNode data) {
        return Trade.builder()
            .setProduct(buildProduct(data))              // The financial product
            .addTradeIdentifier(buildTradeId(data))      // Trade ID
            .setTradeDate(parseTradeDate(data))          // Trade date
            .addParty(buildParty1(data))                 // Bank party
            .addParty(buildParty2(data))                 // Counterparty
            .addCounterparty(buildCounterparties(data))  // Party roles
            .setExecutionDetails(buildExecutionDetails()) // How it was executed
            .addTradeLot(buildTradeLot(data))            // Quantities and prices
            .build();
    }

    private NonTransferableProduct buildProduct(JsonNode data) {
        // Map based on product type
        String productType = data.get("product_type").asText();

        switch(productType) {
            case "Interest Rate Swap":
                return buildInterestRateSwap(data);
            case "Credit Default Swap":
                return buildCreditDefaultSwap(data);
            case "FX Forward":
                return buildFXForward(data);
            default:
                throw new IllegalArgumentException("Unsupported product: " + pro
ductType);
        }
    }

    // Example for Interest Rate Swap
    private NonTransferableProduct buildInterestRateSwap(JsonNode data) {
        return NonTransferableProduct.builder()
```

```
            .setTaxonomy(List.of(ProductTaxonomy.builder()
                .setSource("ISDA")
                .setProductQualifier("InterestRate_IRSwap_FixedFloat")
                .build()))
            .setEconomicTerms(EconomicTerms.builder()
                .setEffectiveDate(parseDate(data.get("effective_date")))
                .setTerminationDate(parseDate(data.get("maturity_date")))
                .addPayout(buildFixedLeg(data))
                .addPayout(buildFloatingLeg(data))
                .build())
            .build();
    }

    // Additional builder methods for parties, quantities, etc.
}
```

Step 3: Create Execution Instruction Builder

```
public class ExecutionInstructionBuilder {

    public ExecutionInstruction buildFromTradeState(TradeState tradeState) {
        Trade trade = tradeState.getTrade();

        return ExecutionInstruction.builder()
            .setProduct(trade.getProduct())                  // What product
            .addPriceQuantity(getPriceQuantities(trade))       // Quantities/prices
            .addCounterparty(trade.getCounterparty().get(0))   // Party 1
            .addCounterparty(trade.getCounterparty().get(1))   // Party 2
            .addParties(trade.getParty())                    // All parties
            .setTradeDate(trade.getTradeDate().getValue())     // When
            .addTradeIdentifier(trade.getTradeIdentifier())    // Trade IDs
            .setExecutionDetails(trade.getExecutionDetails())  // How executed
            .build();
    }
```

```
    private List<PriceQuantity> getPriceQuantities(Trade trade) {
        return trade.getTradeLot()
            .stream()
            .flatMap(lot -> lot.getPriceQuantity().stream())
            .collect(Collectors.toList());
    }
}
```

Step 4: Create Primitive Instruction (Execution Type)

```
public class PrimitiveInstructionBuilder {

    public PrimitiveInstruction buildExecutionPrimitive(ExecutionInstruction exe
cutionInstruction) {
        return PrimitiveInstruction.builder()
            .setExecution(executionInstruction)  // This says "we're doing an execu
tion"
            .build();
    }
}
```

Step 5: Create Instruction (Links Primitive with State)

```
public class InstructionBuilder {

    public Instruction buildExecutionInstruction(PrimitiveInstruction primitiveIns
truction) {
        return Instruction.builder()
            .setPrimitiveInstruction(primitiveInstruction)
            // Note: No "before" state for execution (it's a new trade)
            .build();
    }
}
```

Step 6: Create Event Instruction (Adds Context)

```
public class EventInstructionBuilder {

    public EventInstruction buildExecutionEvent(Instruction instruction, Date tra
deDate) {
        return EventInstruction.builder()
            .addInstruction(instruction)                    // What to do
            .setIntent(EventIntentEnum.EXECUTION)           // Why (execution)
            .setEventDate(tradeDate)                        // When
            .setEffectiveDate(tradeDate)                    // When it takes effect
            .build();
    }
}
```

Step 7: Create Proposed WorkflowStep

```
public class WorkflowStepBuilder {

    public WorkflowStep buildProposedWorkflowStep(EventInstruction eventIns
truction) {
        return WorkflowStep.builder()
            .setProposedEvent(eventInstruction)          // The proposed execution
            .addTimestamp(createTimestamp())             // When proposed
            .addEventIdentifier(createIdentifier())      // Unique ID
            .setAction(ActionEnum.NEW)                   // New trade
            .build();
    }

    private EventTimestamp createTimestamp() {
        return EventTimestamp.builder()
            .setDateTime(ZonedDateTime.now())
            .setQualification(EventTimestampQualificationEnum.EVENT_CREATION
_DATE_TIME)
            .build();
```

```
    }

    private Identifier createIdentifier() {
        return Identifier.builder()
            .addAssignedIdentifier(AssignedIdentifier.builder()
                .setIdentifierValue("EXEC-" + System.currentTimeMillis())
                .build())
            .build();
    }
}
```

Step 8: Accept WorkflowStep (Creates BusinessEvent)

```
public class WorkflowProcessor {

    @Inject
    Create_AcceptedWorkflowStepFromInstruction createAcceptedFunc;

    public WorkflowStep acceptExecution(WorkflowStep proposedStep) {
        // This function automatically:
        // 1. Creates BusinessEvent
        // 2. Applies the execution instruction
        // 3. Generates "after" TradeState
        // 4. Auto-qualifies event as "Execution"
        // 5. Returns complete WorkflowStep

        return createAcceptedFunc.evaluate(proposedStep);
    }
}
```

Step 9: Complete Orchestrator (Puts It All Together)

```
public class CDMTradeProcessor {

    private final FrontOfficeTradeMapper tradeMapper = new FrontOfficeTrade
```

```
Mapper();
    private final ExecutionInstructionBuilder execBuilder = new ExecutionInstru
ctionBuilder();
    private final PrimitiveInstructionBuilder primitiveBuilder = new PrimitiveInstr
uctionBuilder();
    private final InstructionBuilder instructionBuilder = new InstructionBuilder();
    private final EventInstructionBuilder eventBuilder = new EventInstructionBuil
der();
    private final WorkflowStepBuilder workflowBuilder = new WorkflowStepBuil
der();
    private final WorkflowProcessor workflowProcessor = new WorkflowProces
sor();

    @Inject RosettaTypeValidator validator;

    public WorkflowStep processIncomingTrade(String frontOfficeTradeJson) {

        // Step 1: Convert Python dict to CDM TradeState
        TradeState tradeState = tradeMapper.mapToTradeState(frontOfficeTrade
Json);

        // Step 2: Validate trade state
        ValidationReport validation = validator.runProcessStep(TradeState.class,
tradeState);
        if (!validation.isSuccess()) {
            throw new RuntimeException("Trade validation failed: " + validation.get
ValidationFailures());
        }

        // Step 3: Build ExecutionInstruction
        ExecutionInstruction executionInstruction = execBuilder.buildFromTradeS
tate(tradeState);

        // Step 4: Build PrimitiveInstruction
        PrimitiveInstruction primitiveInstruction = primitiveBuilder.buildExecution
Primitive(executionInstruction);
```

```
        // Step 5: Build Instruction
        Instruction instruction = instructionBuilder.buildExecutionInstruction(primi
tiveInstruction);

        // Step 6: Build EventInstruction
        EventInstruction eventInstruction = eventBuilder.buildExecutionEvent(
            instruction, tradeState.getTrade().getTradeDate().getValue());

        // Step 7: Build Proposed WorkflowStep
        WorkflowStep proposedStep = workflowBuilder.buildProposedWorkflowS
tep(eventInstruction);

        // Step 8: Accept the proposal (creates BusinessEvent)
        WorkflowStep finalWorkflowStep = workflowProcessor.acceptExecution
(proposedStep);

        // Step 9: Validate final result
        ValidationReport finalValidation = validator.runProcessStep(WorkflowSte
p.class, finalWorkflowStep);
        if (!finalValidation.isSuccess()) {
            throw new RuntimeException("Final WorkflowStep validation failed");
        }

        return finalWorkflowStep;
    }

    public String getJsonOutput(WorkflowStep workflowStep) throws Exception
{
        return RosettaObjectMapper.getNewRosettaObjectMapper()
            .writerWithDefaultPrettyPrinter()
            .writeValueAsString(workflowStep);
    }
}
```

Step 10: Usage Example

```
public class Main {

    public static void main(String[] args) {

        // Your front office trade (Python dict converted to JSON)
        String frontOfficeTradeJson = """
            {
                "trade_id": "TR12345",
                "product_type": "Interest Rate Swap",
                "notional": 50000000,
                "currency": "USD",
                "trade_date": "2024-01-15",
                "effective_date": "2024-01-17",
                "maturity_date": "2034-01-17",
                "fixed_rate": 0.0325,
                "floating_index": "USD-SOFR",
                "counterparty": "GOLDMAN SACHS",
                "party1": "MY_BANK",
                "party2": "GOLDMAN SACHS"
            }
            """;

        CDMTradeProcessor processor = new CDMTradeProcessor();

        // Process the trade
        WorkflowStep cdmOutput = processor.processIncomingTrade(frontOffice
TradeJson);

        // Get JSON for DRR submission
        String drrOutput = processor.getJsonOutput(cdmOutput);

        System.out.println("CDM Output for DRR:");
        System.out.println(drrOutput);
```

```
    }
}
```

Final Output Structure (What DRR Gets)

```
{
  "businessEvent": {
    "eventQualifier": "Execution",
    "eventDate": "2024-01-15",
    "instruction": [{
      "primitiveInstruction": {
        "execution": {
          "product": { /* Your IRS product details */ },
          "priceQuantity": [/* Notional amounts */],
          "counterparty": [/* Party roles */],
          "parties": [/* All parties */],
          "tradeDate": "2024-01-15",
          "tradeIdentifier": [/* Trade IDs */]
        }
      }
    }],
    "after": [{
      "trade": { /* Complete trade details */ },
      "state": { "positionState": "Executed" }
    }]
  },
  "timestamp": [{
    "dateTime": "2024-01-15T10:30:00Z",
    "qualification": "EventCreationDateTime"
  }],
  "eventIdentifier": [{ "identifier": { "value": "EXEC-1642234200000" } }],
  "action": "New"
}
```

Key Benefits for DRR

1. **Standardized Format**: DRR systems can consume this directly
2. **Auto-Qualification**: Event is automatically qualified as "Execution"
3. **Complete Audit Trail**: Full lineage and timestamps
4. **Validation**: Built-in CDM validation ensures data quality
5. **Regulatory Ready**: Includes all required DRR fields

This approach gives you a complete, validated, DRR-ready CDM WorkflowStep from your front office trade data.

Detailed Explanation of CDM Trade Processing Flow

Let me explain each step with real examples and what each component actually does:

Step 1: Raw Trade Data (Python Dict) → TradeState

What This Step Does:

Converts your front office data into CDM's standard format.

Example Input (Your Python Dict):

```
{
   "trade_id": "TR12345",
   "product": "Interest Rate Swap",
   "bank": "JPMorgan",
   "counterparty": "Goldman Sachs",
   "notional": 50000000,
   "currency": "USD",
   "fixed_rate": 3.25,
```

```
  "trade_date": "2024-01-15"
}
```

What TradeState Contains:

```
TradeState = {
  trade: {
    product: "The financial instrument details",
    parties: "Who is involved",
    tradeLot: "How much, at what price",
    tradeDate: "When it was agreed",
    tradeIdentifier: "Unique ID"
  },
  state: {
    positionState: "Executed" // Current status
  }
}
```

Why This Step Exists:

Your front office uses custom field names like "bank", "counterparty", "notional". CDM uses standard names like "party", "counterparty", "quantity". This step translates between them.

Step 2: TradeState → ExecutionInstruction

What This Step Does:

Extracts only the information needed to execute the trade.

Think of it like this:

You have a complete trade document, but the execution desk only needs specific details to execute it.

Example:

```
ExecutionInstruction = {
   product: "Interest Rate Swap with fixed/floating legs",
   priceQuantity: "50M USD notional at 3.25% fixed rate",
   counterparty: ["JPMorgan as Party1", "Goldman as Party2"],
   parties: ["JPMorgan", "Goldman Sachs"],
   tradeDate: "2024-01-15",
   tradeIdentifier: "TR12345",
   executionDetails: "Executed electronically on Bloomberg"
}
```

Why This Step Exists:

The execution system doesn't need to know everything about the trade (like settlement instructions, documentation, etc.). It just needs to know **what to execute**, **with whom**, **how much**, and **when**.

Step 3: ExecutionInstruction → PrimitiveInstruction

What This Step Does:

Wraps the execution instruction to specify **the type of action**.

Think of it like this:

CDM has many types of actions: execution, amendment, termination, transfer, etc. This step says "this is an execution action".

Example:

```
PrimitiveInstruction = {
   execution: ExecutionInstruction,     // "We're doing an execution"
   quantityChange: null,                // "We're NOT doing quantity change"
   partyChange: null,                   // "We're NOT doing party change"
   termination: null                    // "We're NOT doing termination"
   // ... other primitive types are null
}
```

Why This Step Exists:

CDM needs to know **what type of business operation** you're performing. Are you:

- Executing a new trade? → `execution`
- Changing the notional? → `quantityChange`
- Transferring to new party? → `partyChange`
- Terminating the trade? → `termination`

Step 4: PrimitiveInstruction → Instruction

What This Step Does:

Links the action (PrimitiveInstruction) with the current state of the trade.

Think of it like this:

For some actions, you need to know "what was the trade like before this action?" For execution, there's no "before" because it's a new trade.

Example:

```
Instruction = {
   primitiveInstruction: PrimitiveInstruction,  // What action to perform
   before: null                                 // No previous state (new trade)
}

// For amendments, it would look like:
Instruction = {
   primitiveInstruction: AmendmentInstruction,  // What to change
   before: ExistingTradeState                   // What it looked like before
}
```

Why This Step Exists:

CDM needs to track **state transitions**. To calculate the "after" state, it needs to know:

- **What action** to perform (PrimitiveInstruction)
- **What state** to apply it to (before)

Step 5: Instruction → EventInstruction

What This Step Does:

Adds business context: **WHEN** it happened and **WHY** it happened.

The "WHAT", "WHEN", "WHY" Breakdown:

- **WHAT**: The Instruction (what action to perform)
- **WHEN**: Event date, effective date
- **WHY**: Intent (execution, correction, cancellation)

Example:

```
EventInstruction = {
    instruction: [Instruction],          // WHAT: Execute this trade
    eventDate: "2024-01-15",             // WHEN: Event happened on this date
    effectiveDate: "2024-01-17",         // WHEN: Takes effect on this date
    intent: "EXECUTION"                  // WHY: This is a new trade execution
}
```

Real-World Example:

- **WHAT**: Execute JPMorgan vs Goldman 50M USD Interest Rate Swap
- **WHEN**: Event date = Jan 15 (when parties agreed), Effective date = Jan 17 (when it starts)
- **WHY**: EXECUTION (it's a new trade, not an amendment or cancellation)

Why This Step Exists:

For audit and regulatory purposes, you need to know:

- **When** did the parties agree to this? (eventDate)

- **When** does it legally take effect? (effectiveDate)
- **Why** is this happening? (intent - new trade, correction, etc.)

Step 6: EventInstruction → BusinessEvent

What This Step Does:

Actually executes the instruction and shows the result.

Think of it like this:

- EventInstruction = "Here's what we WANT to do"
- BusinessEvent = "Here's what we ACTUALLY DID and the result"

Example:

```
BusinessEvent = {
   // Input side (what was requested):
   instruction: [Instruction],
   eventDate: "2024-01-15",
   intent: "EXECUTION",

   // Output side (what actually happened):
   eventQualifier: "Execution",          // Auto-determined: "This was an executi
on"
   after: [TradeState]                   // The resulting trade in executed state
}
```

The Magic That Happens:

1. CDM takes your ExecutionInstruction
2. Creates a new TradeState with positionState = "Executed"
3. Auto-qualifies the event as "Execution"
4. Validates everything is correct

Why This Step Exists:

This is the **actual business result**. It shows:

- What you requested (instruction)
- What actually happened (eventQualifier)
- What the trade looks like now (after state)

Step 7: BusinessEvent → WorkflowStep

What This Step Does:

Wraps the business result with workflow metadata for audit and compliance.

Think of it like this:

- BusinessEvent = "The business transaction that occurred"
- WorkflowStep = "The complete record of how it happened"

Example:

```
WorkflowStep = {
  // The core business result:
  businessEvent: BusinessEvent,

  // Workflow metadata:
  timestamp: [
    {
      dateTime: "2024-01-15T10:30:15Z",
      qualification: "EventCreationDateTime"
    }
  ],
  eventIdentifier: [
    {
      identifier: { value: "EXEC-1642234215" }
    }
  ],
```

```
    action: "New",                    // New trade (not correction/cancel)
    party: [JPMorgan, Goldman],       // All parties involved

    // Audit trail (for multi-step workflows):
    previousWorkflowStep: null,       // No previous step (first step)
    nextEvent: null                   // No planned next step
}
```

What Each Field Means:

- **businessEvent**: The actual trade execution result
- **timestamp**: Exact time this step was created
- **eventIdentifier**: Unique ID for this workflow step
- **action**: NEW (new trade) vs CORRECT (fixing error) vs CANCEL (cancelling)
- **party**: Everyone involved in this step
- **previousWorkflowStep**: Links to previous step (for audit trail)

Why This Step Exists:

Regulatory and audit requirements. Regulators need to know:

- **Who** was involved
- **When** exactly it happened
- **What** the unique identifier is
- **How** it relates to other steps
- **Why** this action was taken

Real-World Example: Complete Flow

Let's trace a real trade through the entire flow:

1. Raw Data:

```
{
   "trade_id": "TR12345",
   "product": "Interest Rate Swap",
   "notional": 50000000,
   "currency": "USD",
   "fixed_rate": 0.0325,
   "counterparty": "Goldman Sachs",
   "trade_date": "2024-01-15"
}
```

2. TradeState:

"Here's your trade in CDM format with all the standardized fields"

3. ExecutionInstruction:

"To execute this trade, I need: 50M USD IRS, fixed rate 3.25%, between us and Goldman, on Jan 15"

4. PrimitiveInstruction:

"The action I'm performing is: EXECUTION (not amendment, termination, etc.)"

5. Instruction:

"Perform an execution action, with no previous state (since it's new)"

6. EventInstruction:

"Execute this trade on Jan 15 for the purpose of EXECUTION"

7. BusinessEvent:

"I executed the trade successfully. Event qualified as 'Execution'. Here's the resulting trade state."

8. WorkflowStep:

"Complete record: Business event happened, at 10:30:15 UTC, ID EXEC-1642234215, action was NEW, parties were JPMorgan and Goldman"

Why This Complexity?

For DRR (Derivatives Reporting), you need:

1. **Complete audit trail** - every step documented
2. **Regulatory compliance** - all required fields present
3. **Event qualification** - automatic categorization
4. **State tracking** - before/after states
5. **Workflow lineage** - how events connect
6. **Validation** - ensures data integrity

The WorkflowStep contains **everything DRR needs** in a standardized, validated format.

EventInstruction: Adding Business Context

What EventInstruction Does

EventInstruction takes a basic **Instruction** (which just says "do this action") and adds **business context** around it.

Think of it like the difference between:

- **Instruction**: "Execute this trade"
- **EventInstruction**: "Execute this trade on January 15th, 2024, because it's a new trade execution, and it becomes effective on January 17th, 2024"

The Context It Adds

1. WHEN Context (Timing Information)

eventDate - When the business decision happened

```
EventInstruction = {
    eventDate: "2024-01-15"  // When parties agreed to the trade
```

```
}
```

Real Example:

- You and Goldman Sachs agree to a trade on the phone at 10:30 AM on January 15th
- **eventDate = "2024-01-15"** = The date the business agreement happened

effectiveDate - When it legally takes effect

```
EventInstruction = {
   eventDate: "2024-01-15",     // When agreed
   effectiveDate: "2024-01-17"  // When it starts (T+2 settlement)
}
```

Real Example:

- **eventDate**: January 15 (when you shook hands on the deal)
- **effectiveDate**: January 17 (when money/interest actually starts flowing)

Why both dates matter:

- Regulators need to know **when the decision was made** (eventDate)
- They also need to know **when it legally begins** (effectiveDate)
- For accounting: book the trade on eventDate, start accruing interest on effectiveDate

2. WHY Context (Business Intent)

intent - Why this event is happening

```
EventInstruction = {
   intent: EventIntentEnum.EXECUTION  // This is a new trade execution
}
```

Different intent values:

```
// New trade
intent: EXECUTION

// Fixing an error
intent: CORRECTION

// Canceling a trade
intent: CANCELLATION

// Changing trade terms
intent: CONTRACT_TERMS_AMENDMENT

// Clearing submission
intent: CLEARING

// Corporate action adjustment
intent: CORPORATE_ACTION_ADJUSTMENT
```

Real Example:
Same action, different intent:

Scenario 1: New Trade

```
EventInstruction = {
   instruction: "Execute 50M USD IRS",
   intent: EXECUTION,            // WHY: It's a new trade
   eventDate: "2024-01-15"
}
```

Scenario 2: Correcting Error

```
EventInstruction = {
   instruction: "Execute 50M USD IRS",
   intent: CORRECTION,           // WHY: Fixing previous error
```

```
    eventDate: "2024-01-15"
}
```

Same action, but different business meaning!

3. Business Context (Additional Information)

packageInformation - If part of a larger deal

```
EventInstruction = {
    instruction: "Execute USD IRS",
    packageInformation: {
        packageIdentifier: "PORTFOLIO_TRADE_001",
        packageSize: 5  // This trade is 1 of 5 in the package
    }
}
```

Real Example:

- You do a portfolio trade with Goldman: 5 different swaps all at once
- Each swap gets its own EventInstruction
- But they all reference the same packageInformation
- This tells regulators "these 5 trades were done together as one business decision"

Before vs After: Instruction vs EventInstruction

Just Instruction (No Context):

```
Instruction = {
    primitiveInstruction: {
        execution: {
            product: "Interest Rate Swap",
            notional: 50000000,
            parties: ["JPMorgan", "Goldman Sachs"]
```

```
    }
  }
}
```

This tells you: Execute this trade

This doesn't tell you: When, why, or any business context

EventInstruction (With Context):

```
EventInstruction = {
  // The action (same as above)
  instruction: [Instruction],

  // Business context added:
  eventDate: "2024-01-15",          // WHEN: Agreed on Jan 15
  effectiveDate: "2024-01-17",      // WHEN: Effective Jan 17
  intent: EventIntentEnum.EXECUTION,  // WHY: New trade execution
  packageInformation: {             // CONTEXT: Part of larger deal
    packageIdentifier: "PKG_001"
  }
}
```

Real-World Examples

Example 1: Simple New Trade

```
EventInstruction = {
  instruction: "Execute 50M USD Interest Rate Swap",
  eventDate: "2024-01-15",          // Trade date
  effectiveDate: "2024-01-17",      // Effective date (T+2)
  intent: EXECUTION                 // New trade
}
```

Business meaning: "On January 15th, we agreed to execute a new 50M USD interest rate swap that will become effective on January 17th."

Example 2: Correcting an Error

```
EventInstruction = {
   instruction: "Execute 50M USD Interest Rate Swap",
   eventDate: "2024-01-15",          // Original trade date
   effectiveDate: "2024-01-17",      // Original effective date
   intent: CORRECTION                // Fixing previous mistake
}
```

Business meaning: "We're correcting an error in the 50M USD swap that was originally agreed on January 15th."

Example 3: Portfolio Trade

```
EventInstruction = {
   instruction: "Execute 50M USD Interest Rate Swap",
   eventDate: "2024-01-15",
   effectiveDate: "2024-01-17",
   intent: EXECUTION,
   packageInformation: {
      packageIdentifier: "PORTFOLIO_TRADE_XYZ",
      packageSize: 10               // 1 of 10 trades in package
   }
}
```

Business meaning: "On January 15th, we executed a new 50M USD swap as part of a 10-trade portfolio deal (package XYZ)."

Why This Context Matters

For Regulators:

- **When**: Need to know exact timing for market surveillance

- **Why**: Need to distinguish between new trades vs corrections vs cancellations
- **Package**: Need to understand if trades were done together (systemic risk)

For Risk Management:

- **When**: Calculate exposure as of specific dates
- **Why**: Treat corrections differently from new exposures
- **Package**: Understand concentration risk

For Accounting:

- **When**: Book trades on eventDate, start P&L on effectiveDate
- **Why**: Handle corrections with different accounting treatment
- **Package**: May need special accounting for portfolio trades

For Audit:

- **When**: Verify trades happened when claimed
- **Why**: Ensure proper business justification
- **Package**: Verify package trades were legitimate business decisions

Summary

Instruction = "Do this action"

EventInstruction = "Do this action, on this date, for this business reason, with this context"

The EventInstruction provides the **full business story** that regulators, auditors, and risk managers need to understand not just **what happened**, but **when it happened**, **why it happened**, and **how it fits into the bigger picture**.

Here's What You Need to Build vs What's Automatic

What You MUST Build Manually:

1. TradeState (Always Manual)

```
TradeState tradeState = TradeState.builder()
    .setTrade(/* your trade mapping */)
    .setState(/* execution state */)
    .build();
```

2. ExecutionInstruction (Manual)

```
ExecutionInstruction executionInstruction = ExecutionInstruction.builder()
    .setProduct(tradeState.getTrade().getProduct())
    .addPriceQuantity(/* extract from trade */)
    .addCounterparty(/* extract from trade */)
    // ... other fields
    .build();
```

3. Up to EventInstruction (Manual)

```
// Manual
PrimitiveInstruction primitiveInstruction = PrimitiveInstruction.builder()
    .setExecution(executionInstruction)
    .build();

// Manual
Instruction instruction = Instruction.builder()
    .setPrimitiveInstruction(primitiveInstruction)
    .build();

// Manual
EventInstruction eventInstruction = EventInstruction.builder()
    .addInstruction(instruction)
    .setIntent(EventIntentEnum.EXECUTION)
```

```
    .setEventDate(tradeDate)
    .build();
```

What CDM Does AUTOMATICALLY:

4. BusinessEvent Creation (Automatic)

```
@Inject
Create_AcceptedWorkflowStepFromInstruction createAcceptedFunc;

// This function automatically:
// - Creates BusinessEvent
// - Applies instructions
// - Generates "after" TradeState
// - Auto-qualifies event type
// - Creates complete WorkflowStep
WorkflowStep result = createAcceptedFunc.evaluate(proposedWorkflowSte
p);
```

Simplified Approach:

You can create a helper that reduces your work:

```
public class CDMSimplifiedProcessor {

    @Inject
    Create_AcceptedWorkflowStepFromInstruction createAcceptedFunc;

    // You only need to call this with your TradeState
    public WorkflowStep processExecution(TradeState tradeState) {

        // 1. Build execution instruction from your TradeState
        ExecutionInstruction execInstruction = buildExecutionInstruction(tradeSta
te);
```

```
        // 2. Wrap in primitives (boilerplate)
        PrimitiveInstruction primitiveInstruction = PrimitiveInstruction.builder()
            .setExecution(execInstruction)
            .build();

        Instruction instruction = Instruction.builder()
            .setPrimitiveInstruction(primitiveInstruction)
            .build();

        EventInstruction eventInstruction = EventInstruction.builder()
            .addInstruction(instruction)
            .setIntent(EventIntentEnum.EXECUTION)
            .setEventDate(tradeState.getTrade().getTradeDate().getValue())
            .build();

        // 3. Create proposed workflow step
        WorkflowStep proposedStep = WorkflowStep.builder()
            .setProposedEvent(eventInstruction)
            .addTimestamp(createTimestamp())
            .addEventIdentifier(createIdentifier())
            .setAction(ActionEnum.NEW)
            .build();

        // 4. CDM automatically handles the rest!
        return createAcceptedFunc.evaluate(proposedStep);
    }

    private ExecutionInstruction buildExecutionInstruction(TradeState tradeStat
e) {
        Trade trade = tradeState.getTrade();
        return ExecutionInstruction.builder()
            .setProduct(trade.getProduct())
            .addPriceQuantity(trade.getTradeLot().get(0).getPriceQuantity())
            .addCounterparty(trade.getCounterparty())
            .addParties(trade.getParty())
            .setTradeDate(trade.getTradeDate().getValue())
```

```
            .addTradeIdentifier(trade.getTradeIdentifier())
            .setExecutionDetails(trade.getExecutionDetails())
            .build();
    }

    // Helper methods for timestamps and identifiers...
}
```

So Your Work Is:

Step 1: Map Front Office Data → TradeState

```
public TradeState mapFrontOfficeToTradeState(String frontOfficeJson) {
    // Your custom mapping logic here
    return TradeState.builder()
        .setTrade(/* map your fields */)
        .setState(/* set to executed */)
        .build();
}
```

Step 2: Call Simple Processor

```
public WorkflowStep processIncomingTrade(String frontOfficeJson) {

    // 1. You build this
    TradeState tradeState = mapFrontOfficeToTradeState(frontOfficeJson);

    // 2. Helper handles the rest
    return cdmProcessor.processExecution(tradeState);
}
```

Final Answer:

You need to build:

1. **TradeState** (from your front office data)
2. **ExecutionInstruction** (extracted from TradeState)
3. **Boilerplate wrappers** (PrimitiveInstruction, Instruction, EventInstruction)

CDM automatically handles:

1. **BusinessEvent creation**
2. **Event qualification** (determines it's an "Execution")
3. **"After" state generation**
4. **Final WorkflowStep assembly**
5. **Validation**

Bottom line: You build the TradeState + some boilerplate, CDM does the complex business logic automatically.

Yes, For Execution You DO Have a WorkflowStep

WorkflowStep is ALWAYS the Final Output

For **any** CDM business event, including execution, the final result is **always** a WorkflowStep.

For Trade Execution:

```
WorkflowStep {
  "businessEvent": {
    "eventQualifier": "Execution",       // ← CDM automatically determines this
    "eventDate": "2024-01-15",
    "instruction": [{
      "primitiveInstruction": {
        "execution": {                   // ← Your ExecutionInstruction
          "product": "Interest Rate Swap",
```

```
        "parties": ["JPMorgan", "Goldman"],
        "tradeDate": "2024-01-15"
       }
      }
     }],
    "after": [{                    // ← The resulting executed trade
     "trade": { /* your trade details */ },
     "state": {
      "positionState": "Executed"      // ← Now it's executed
     }
    }]
  },
  "timestamp": [{
    "dateTime": "2024-01-15T10:30:00Z",
    "qualification": "EventCreationDateTime"
  }],
  "eventIdentifier": [{
    "identifier": { "value": "EXEC-12345" }
  }],
  "action": "New"                  // ← New trade execution
}
```

What Makes It an "Execution" WorkflowStep:

1. **businessEvent.eventQualifier = "Execution"** (auto-determined by CDM)
2. **instruction contains ExecutionInstruction**
3. **No "before" state** (because it's a new trade)
4. **"after" state shows executed trade**
5. **action = "New"** (new trade, not correction)

The Complete Flow for Execution:

```
// Your input
TradeState mappedTrade = mapFromFrontOffice(frontOfficeData);

// Create execution instruction
ExecutionInstruction executionInstruction = extractExecutionDetails(mappedT
rade);

// Wrap in CDM structures
PrimitiveInstruction primitive = PrimitiveInstruction.builder()
    .setExecution(executionInstruction)    // ← This makes it an execution
    .build();

// Create workflow step (proposed)
WorkflowStep proposedStep = WorkflowStep.builder()
    .setProposedEvent(eventInstruction)
    .build();

// CDM processes it (accepted)
WorkflowStep finalWorkflowStep = createAcceptedFunc.evaluate(proposedSt
ep);

// finalWorkflowStep.businessEvent.eventQualifier = "Execution"
```

Different Events, Same WorkflowStep Structure:

Execution WorkflowStep:

```
{
  "businessEvent": {
    "eventQualifier": "Execution",       // ← Different qualifier
    "instruction": [{ "primitiveInstruction": { "execution": {...} }}]
  }
}
```

Amendment WorkflowStep:

```
{
  "businessEvent": {
    "eventQualifier": "Amendment",       // ← Different qualifier
    "instruction": [{ "primitiveInstruction": { "termsChange": {...} }}]
  }
}
```

Termination WorkflowStep:

```
{
  "businessEvent": {
    "eventQualifier": "Termination",     // ← Different qualifier
    "instruction": [{ "primitiveInstruction": { "quantityChange": {...} }}]
  }
}
```

Key Point:

WorkflowStep is the universal container for ALL CDM events.

- Trade execution → WorkflowStep with BusinessEvent qualified as "Execution"
- Trade amendment → WorkflowStep with BusinessEvent qualified as "Amendment"
- Trade termination → WorkflowStep with BusinessEvent qualified as "Termination"

For DRR Reporting:

You always send the WorkflowStep, regardless of event type:

```
// For execution
WorkflowStep executionWorkflowStep = processExecution(tradeData);
sendToDRR(executionWorkflowStep);
```

```
// For amendment
WorkflowStep amendmentWorkflowStep = processAmendment(tradeData);
sendToDRR(amendmentWorkflowStep);

// Same format, different content
```

Answer:

Yes, for execution you have a WorkflowStep.

In fact, you **always** have a WorkflowStep as the final output, no matter what type of business event you're processing. The WorkflowStep is CDM's standard container that includes the business event plus all the workflow metadata (timestamps, identifiers, audit trail, etc.) that regulators and downstream systems need.

Complete CDM Execution Trade Output Sample

Read file: rosetta-source/src/main/resources/cdm-sample-files/functions/business-event/execution/execution-ir-swap-func-output.json

Read file: rosetta-source/src/main/resources/cdm-sample-files/functions/business-event/execution/execution-ir-swap-func-output.json

Read file: rosetta-source/src/main/resources/cdm-sample-files/functions/business-event/execution/execution-ir-swap-func-output.json

Perfect! I found excellent examples. Let me look at a complete WorkflowStep sample that shows the full structure including the wrapper around BusinessEvent.

Read file: rosetta-source/src/main/resources/result-json-files/fpml-5-10/processes/msg-ex63-execution-advice-trade-initiation.json

Read file: rosetta-source/src/main/resources/result-json-files/fpml-5-10/processes/msg-ex63-execution-advice-trade-initiation.json

Here's what the **final CDM output** looks like after processing your front office trade. I found real examples from the codebase:

1. Complete WorkflowStep for Execution (Final Output)

Looking at `execution-ir-swap-func-output.json`, here's what your final CDM output will be:

The Structure:

```
{
  // ===== BUSINESS EVENT (The core business result) =====
  "eventDate": "1994-12-12",
  "instruction": [{
    "primitiveInstruction": {
      "execution": {
        // Your ExecutionInstruction details
        "product": { /* Full interest rate swap product definition */ },
        "tradeDate": "1994-12-12",
        "counterparty": [/* Party details */],
        "tradeIdentifier": [/* Trade IDs */]
      }
    }
  }],
  "eventQualifier": "Execution",          // ← CDM auto-detects this
  "after": [{                             // ← The resulting executed trade
    "trade": {
      /* Complete trade definition matching your ExecutionInstruction */
    },
    "state": {
      "positionState": "Executed"         // ← Shows it's executed
    }
  }],
  "meta": {
```

```
    "globalKey": "352c7587"
  }
}
```

2. Complete WorkflowStep Wrapper (What DRR Receives)

For DRR reporting, you get the full WorkflowStep like this sample from `msg-ex63-execution-advice-trade-initiation.json`:

```
{
  // ===== WORKFLOW METADATA =====
  "messageInformation": {
    "messageId": {
      "value": "0482588"
    },
    "sentBy": {
      "value": "AMAGGB22XXX"
    }
  },
  "timestamp": [{
    "dateTime": "2007-07-27T10:02:00Z",
    "qualification": "eventCreationDateTime"
  }],
  "eventIdentifier": [{
    "assignedIdentifier": [{
      "identifier": {
        "value": "0482588"
      }
    }]
  }],
  "action": "New",

  // ===== THE BUSINESS EVENT =====
  "businessEvent": {
```

```
    /* The entire BusinessEvent structure from above */
  },

  "meta": {
    "globalKey": "d510ce3d"
  }
}
```

3. Key Points for Your Implementation:

For DRR Compliance:

- **WorkflowStep** is the complete object DRR expects
- Contains full audit trail with timestamps and identifiers
- Includes message routing information
- Has the BusinessEvent embedded within it

Your Front Office Trade Flow:

```
Python Dict → JSON → TradeState → ExecutionInstruction → BusinessEvent → WorkflowStep
```

Final Output Characteristics:

- **Format**: Complete JSON WorkflowStep
- **Size**: Can be large (1000+ lines for complex trades)
- **Contains**: Full trade details, execution results, audit metadata
- **Key Fields**: `businessEvent.eventQualifier = "Execution"`, `action = "New"`, timestamps

What You Actually Send to DRR:

The complete WorkflowStep JSON object, which contains everything - your original trade data transformed into CDM format, plus all the execution metadata and audit trail information.

The samples show that a simple interest rate swap execution generates over 1000 lines of structured JSON with complete trade lifecycle information ready for regulatory reporting.

CDM Code Template

I've created a Java skeleton file (`TradeExecutionWorkflowSkeleton.java`) that demonstrates the full CDM trade processing flow, step by step, from raw trade data to the final `WorkflowStep`. Each step is clearly marked and uses the builder pattern for each CDM object. You can now fill in the details for each step as needed for your specific trade and mapping logic.

```
package org.finos.cdm.example;

import cdm.base.staticdata.party.Party;
import cdm.base.staticdata.party.Counterparty;
import cdm.event.common.TradeState;
import cdm.event.common.ExecutionInstruction;
import cdm.event.common.PrimitiveInstruction;
import cdm.event.common.Instruction;
import cdm.event.workflow.EventInstruction;
import cdm.event.common.BusinessEvent;
import cdm.event.workflow.WorkflowStep;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.regnosys.rosetta.common.serialisation.RosettaObjectMapper;

import java.util.List;

public class TradeExecutionWorkflowSkeleton {
    public static void main(String[] args) throws Exception {
        // 1. Raw Trade Data (Python Dict → JSON → Java Map or POJO)
        // For this skeleton, assume you have a Java object or Map<String, Object>
        // Map<String, Object> rawTradeData = ...;

        // 2. TradeState (Your trade in CDM format)
```

```
TradeState tradeState = TradeState.builder()
    // .setTrade(...)
    // .setState(...)
    .build();

// 3. ExecutionInstruction (What to execute)
ExecutionInstruction executionInstruction = ExecutionInstruction.builder()
    // .setProduct(...)
    // .setPriceQuantity(...)
    // .setCounterparty(...)
    // .setParties(...)
    // .setTradeDate(...)
    // .setTradeIdentifier(...)
    .build();

// 4. PrimitiveInstruction (How to execute)
PrimitiveInstruction primitiveInstruction = PrimitiveInstruction.builder()
    .setExecution(executionInstruction)
    .build();

// 5. Instruction (What + Current State)
Instruction instruction = Instruction.builder()
    .setBeforeValue(tradeState)
    .setPrimitiveInstruction(primitiveInstruction)
    .build();

// 6. EventInstruction (What + When + Why)
EventInstruction eventInstruction = EventInstruction.builder()
    .addInstruction(instruction)
    // .setIntent(...)
    // .setEventDate(...)
    .build();

// 7. BusinessEvent (The actual execution result)
BusinessEvent businessEvent = BusinessEvent.builder()
    .setEventQualifier("Execution")
```

```
            .addInstruction(instruction)
            // .setEventDate(...)
            // .addAfter(...)
            .build();

        // 8. WorkflowStep (Complete audit trail + BusinessEvent)
        WorkflowStep workflowStep = WorkflowStep.builder()
            .setBusinessEvent(businessEvent)
            // .addTimestamp(...)
            // .addEventIdentifier(...)
            // .setAction(...)
            .build();

        // (Optional) Serialize to JSON for DRR or downstream
        ObjectMapper mapper = RosettaObjectMapper.getNewRosettaObjectMappe
        String json = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(wo
        System.out.println(json);
    }
}
```

Let me know which step you'd like to expand or if you want to see a concrete example for a particular product or field!