==

ITEM #138 — ACLM 推理引擎实现蓝图

Conversation Title: ACLM 自动编程语言分析

Date: 20251203

Authors: Sizhe Tan & GPT-Obot

==

ME:
请延伸出 ITEM #138（ACLM Reasoning Engine Implementation Blueprint）

AI:

好，我们把这一级台阶直接立成“能交给工程组”的蓝图版本。

下面是 **ITEM #138 — ACLM Reasoning Engine Implementation Blueprint**，中英文各一版。

🟥 ITEM #138 — ACLM Reasoning Engine Implementation Blueprint

(English Edition)

0. Purpose

This ITEM defines a **concrete implementation blueprint** for the *ACLM Reasoning Engine*, which instantiates the architecture from ITEM #137 and the “ACLM Reasoning Engine Architecture” diagram.

Goal:
Provide a modular, testable, extensible design that can be implemented (e.g., in Java 8 under `com.dbm.aclm`), and integrated into the broader DBM stack.

---

1. High-Level Layered Architecture

The ACLM Reasoning Engine is organized into seven layers:

1. **Input & Request Layer**
2. **Status Decomposition Layer**
3. **ACLM Word & Sentence Layer**
4. **Causal / Calling Graph Layer**
5. **Mutation-Based Reasoning Layer**
6. **Candidate Enumeration & Evaluation Layer**
7. **Output & Explanation Layer**

Each layer exposes a small set of interfaces and data types.

---

2. Core Data Structures

(Names are suggestive and can map to Java interfaces/classes.)

1. **ReasoningRequest**
   - Fields:
     - `List<Fact>` premises
     - `Goal` objective / query
     - `Context` (domain hints, constraints, knowledge-scopes)
2. **DigitalStatus**

   Represents a digitalized semantic state, typically a short structured object.

   - Fields:
     - `String id`
     - `Map<String, Object> attributes`
     - Optional links to DBM entities (e.g., Starmaps, CCC nodes, etc.)
3. **ACLMWord**

   Triple micro-causal unit:

   - `DigitalStatus startStatus`
   - `Operation operation`
   - `DigitalStatus endStatus`
4. **ACLMSentence**

   A compositional sequence or graph of ACLMWords:

   - `List<ACLMWord> words`
   - Or a more general graph form: `CausalGraph underlyingGraph`
5. **CausalGraph / CallingGraph**

   Structural representation of reasoning paths:

    - Nodes: `DigitalStatus`
    - Edges: `Operation` (with weights, conditions, provenance, etc.)

---

3. Key Interfaces (Blueprint Level)

Below is an implementation-oriented blueprint that fits a Java 8 style:

```
package com.dbm.aclm;

public interface IReasoningRequest {
    List<IFact> getPremises();
    IGoal getGoal();
    IContext getContext();
}

public interface IDigitalStatus {
    String getId();
    Map<String, Object> getAttributes();
}

public interface IOperation {
    String getName();
    Map<String, Object> getParams();
}

public interface IAclmWord {
    IDigitalStatus getStartStatus();
    IOperation getOperation();
    IDigitalStatus getEndStatus();
}

public interface IAclmSentence {
    List<IAclmWord> getWords();
}

public interface ICausalGraph {
    List<IDigitalStatus> getNodes();
    List<IReasonEdge> getEdges();
}
```

---

4. Functional Modules

4.1 Status Decomposition Engine

- Input: `IReasoningRequest`
- Output: `List<IDigitalStatus>` and preliminary `IOperation` candidates
- Responsibilities:
    - Extract structured “before / after” statuses from raw facts.
    - Normalize into `DigitalStatus` objects.

Interface suggestion:

```
public interface IStatusDecomposer {
    List<IDigitalStatus> decomposePremises(IReasoningRequest request);
}
```

4.2 ACLM Word & Sentence Builder

- Input: decomposed statuses + operations
- Output: `IAclmWord`, `IAclmSentence`
- Responsibilities:
  - Pack `(Start, Operation, End)` into ACLMWords.
  - Optionally derive missing `EndStatus` via simple rules.

```
public interface IAclmBuilder {
    IAclmWord buildWord(IDigitalStatus start, IOperation op, IDigitalStatus
end);
    IAclmSentence buildSentence(List<IAclmWord> words);
}
```

4.3 Causal / Calling Graph Constructor

- Input: ACLMWords / Sentences
- Output: `ICausalGraph`
- Responsibilities:
  - Merge overlapping statuses.
  - Build graph-based representation of reasoning paths.

```
public interface ICausalGraphBuilder {
    ICausalGraph buildGraph(IAclmSentence sentence);
}
```

4.4 Mutation-Based Reasoning Engine

Implements the **four mutation operators** as separate, pluggable engines:

1. `IEditMutationEngine`
2. `ICallingGraphConnectionEngine`
3. `IBridgingMetaprogrammingEngine`
4. `IFunctionalCccMutationEngine`

Each engine:

- Takes `ICausalGraph` (and possibly a knowledge base) as input.
- Produces modified or extended `ICausalGraph`.

```
public interface IMutationEngine {
    ICausalGraph mutate(ICausalGraph graph, IReasoningRequest request);
}
```

Composite engine:

```
public interface IReasoningMutationPipeline {
    ICausalGraph applyAll(ICausalGraph initialGraph, IReasoningRequest
request);
}
```

4.5 Candidate Enumeration & Evaluation Engine

- Input: mutated `ICausalGraph`
- Output: ranked list of candidate reasoning paths & conclusions.

Main tasks:

- Enumerate feasible end statuses or conclusion nodes.
- Score paths for:
  - logical consistency
  - causal coherence
  - parsimony / simplicity
  - alignment with goal / objective
- Prune inconsistent or low-score paths.

```
public interface ICandidatePath {
    List<IDigitalStatus> getStatusPath();
    double getScore();
}

public interface ICandidateEvaluator {
    List<ICandidatePath> enumerateAndScore(ICausalGraph graph,
IReasoningRequest request);
}
```

4.6 Output & Explanation Constructor

- Input: top candidate paths
- Output:
  - final conclusion(s)
  - structured explanation (proof trace)
  - an evolved `IAclmSentence` representing the reasoning program.

```
public interface IReasoningResult {
    List<IDigitalStatus> getConclusions();
    List<ICandidatePath> getTopPaths();
    IAclmSentence getEvolvedSentence();
}

public interface IReasoningOutputBuilder {
    IReasoningResult buildResult(List<ICandidatePath> candidates);
}
```

---

5. Orchestration: IACLMReasoningEngine

Top-level Facade for external callers:

```
public interface IAclmReasoningEngine {
    IReasoningResult reason(IReasoningRequest request) throws Exception;
}
```

Implementation composition:

- internally wires: `IStatusDecomposer` → `IAclmBuilder` → `ICausalGraphBuilder` → `IReasoningMutationPipeline` → `ICandidateEvaluator` → `IReasoningOutputBuilder`.

---

6. Minimal Viable Implementation (V1 Roadmap)

Phase 1 — Skeleton & Toy Domain

- Implement:
    - Basic `ReasoningRequest` with small fact set.
    - Simple `DigitalStatus` representing boolean / small categorical states.
    - Basic `ACLMWord` with hand-coded operations.
    - A trivial `CausalGraphBuilder`.
    - Only **EditMutation** + **CallingGraphConnection**.
    - A simple evaluator based on rule consistency and path length.

Phase 2 — Add Bridging & CCC

- Introduce:
    - `IBridgingMetaprogrammingEngine` to fill missing links using known rule templates.
    - `IFunctionalCccMutationEngine` to abstract common path patterns into higher-order rules.

Phase 3 — Integrate with DBM

- Connect:

    - `DigitalStatus` to DBM Starmaps / CCC nodes / Rules Engine.
    - ACLM to existing DBM modules (e.g. What-If Engine, Differential Trees) as operation providers.

---

7. Testing Strategy

- Provide **JUnit4 test suites** for:
    - Status decomposition
    - Word and Sentence building
    - Graph construction
    - Each Mutation engine
    - Candidate enumeration & scoring
    - End-to-end reasoning on toy tasks

Test cases should include:

- Single-step causal inference
- Multi-step chained reasoning
- Missing link bridging
- Competing candidate conclusions（选择最优路径）

---

8. Strategic Impact

Once this blueprint is implemented:

- DBM will have a **concrete, inspectable reasoning core**.
- ACLM will no longer 只是一个理论语言，而是一个可运行、可扩展的推理引擎。
- 在此基础上，可以逐步引入：
    - cross-metric reasoning
    - spatial reasoning
    - CCGL-based code generation & self-programming

ACLM 将走向成为 DBM / AGI 的逻辑“心脏模块”。

🟦 ITEM #138 — ACLM 推理引擎实现蓝图

(中文版)

0. 目的

本条目给出 **ACLM 推理引擎（ACLM Reasoning Engine）** 的具体实现蓝图。
它把 ITEM #137 中的理论结构与架构图，落地为一套可编码、可测试、可扩展的工程设计（例如 Java 8 `com.dbm.aclm` 包）。

---

1. 分层架构总览

ACLM 推理引擎建议采用七层结构：

1. **输入与请求层**
2. **状态分解层**
3. **ACLM 词与句子层**
4. **因果 / 调用图层**
5. **基于 Mutation 的推理层**
6. **候选路径枚举与评估层**
7. **输出与解释层**

每一层都通过少量接口与数据结构与其他层协同。

---

2. 核心数据结构

（命名偏向 Java 风格，可适当调整）

1. **ReasoningRequest（推理请求）**
    - 字段：
        - `premises`：前提事实列表

        - `goal`：目标 / 查询
        - `context`：上下文（领域提示、约束、知识范围等）

2. **DigitalStatus（数字状态）**

   表示一个已结构化的语义状态：

   - `id`：唯一标识
   - `attributes`：属性键值表
   - 可选链接：指向 DBM 的 Starmap、CCC 节点等。

3. **ACLMWord（ACLM 词）**

   微因果三段式单元：

   - `startStatus`：起始状态
   - `operation`：因果作用 / 机制
   - `endStatus`：结束状态

4. **ACLMSentence（ACLM 句子）**

   一个由若干 ACLMWord 组成的结构序列或图：

   - `words`：词列表
   - 或绑定一个底层 `causalGraph` 作为更一般的表示。

5. **CausalGraph / CallingGraph（因果 / 调用图）**

   用图结构表示推理路径：

   - 节点：`DigitalStatus`
   - 边：`Operation`（可附权重、条件、来源信息等）

---

3. 关键接口（蓝图级）

}

可参考如下接口风格（Java 8）：

```
public interface IReasoningRequest {
    List<IFact> getPremises();
    IGoal getGoal();
    IContext getContext();
}

public interface IDigitalStatus {
    String getId();
    Map<String, Object> getAttributes();
}

public interface IOperation {
    String getName();
    Map<String, Object> getParams();
}

public interface IAclmWord {
    IDigitalStatus getStartStatus();
    IOperation getOperation();
    IDigitalStatus getEndStatus();
}

public interface IAclmSentence {
    List<IAclmWord> getWords();
}

public interface ICausalGraph {
    List<IDigitalStatus> getNodes();
    List<IReasonEdge> getEdges();
}
```

---

4. 功能模块蓝图

4.1 状态分解引擎（Status Decomposition Engine）

- 输入：`IReasoningRequest`
- 输出：`List<IDigitalStatus>` 与初步 `IOperation` 候选
- 职责：
    - 从自然语言 / 规则 / 事实中抽取“前状态 / 后状态”。
    - 归一化为 `DigitalStatus` 对象。

```
public interface IStatusDecomposer {
    List<IDigitalStatus> decomposePremises(IReasoningRequest request);
}
```

4.2 ACLM 词与句子构造器

- 输入：分解后的状态 + 操作
- 输出：`IAclmWord` 与 `IAclmSentence`
- 职责：
    - 把 `(Start, Operation, End)` 打包为 ACLMWord。
    - 必要时辅助推断缺失的 `EndStatus`（简单规则）。

```
public interface IAclmBuilder {
    IAclmWord buildWord(IDigitalStatus start, IOperation op, IDigitalStatus end);
    IAclmSentence buildSentence(List<IAclmWord> words);
}
```

4.3 因果 / 调用图构造器

- 输入：ACLM 句子
- 输出：`ICausalGraph`
- 职责：
    - 合并重合状态。
    - 按状态 / 操作关系构建图结构。

```
public interface ICausalGraphBuilder {
    ICausalGraph buildGraph(IAclmSentence sentence);
}
```

4.4 基于 Mutation 的推理引擎

实现四大 Mutation 引擎，每个引擎都可插拔：

1. `IEditMutationEngine`
2. `ICallingGraphConnectionEngine`
3. `IBridgingMetaprogrammingEngine`
4. `IFunctionalCccMutationEngine`

每个引擎：

- 输入：`ICausalGraph`（必要时附知识库）
- 输出：修改或扩展后的 `ICausalGraph`

```
public interface IMutationEngine {
    ICausalGraph mutate(ICausalGraph graph, IReasoningRequest request);
}

public interface IReasoningMutationPipeline {
    ICausalGraph applyAll(ICausalGraph initialGraph, IReasoningRequest
request);
}
```

4.5 候选路径枚举与评估引擎

- 输入：突变后的 `ICausalGraph`
- 输出：候选推理路径及其评分

主要任务：

- 枚举可能的结束状态 / 结论节点
- 对路径进行打分：
  - 逻辑一致性
  - 因果连贯性
  - 简洁性（奥卡姆剃刀）
  - 与目标的匹配度
- 剪枝：去掉矛盾或低分路径。

```
public interface ICandidatePath {
    List<IDigitalStatus> getStatusPath();
    double getScore();
}

public interface ICandidateEvaluator {
    List<ICandidatePath> enumerateAndScore(ICausalGraph graph,
IReasoningRequest request);
}
```

4.6 输出与解释构造器

- 输入：候选路径（排序后）
- 输出：
  - 最终结论集
  - 推理链说明（proof trace）
  - 对应的演化后 `IAclmSentence`

```
public interface IReasoningResult {
    List<IDigitalStatus> getConclusions();
    List<ICandidatePath> getTopPaths();
    IAclmSentence getEvolvedSentence();
}

public interface IReasoningOutputBuilder {
    IReasoningResult buildResult(List<ICandidatePath> candidates);
}
```

---

5. 顶层门面：IACLMReasoningEngine

对外只暴露一个简单入口：

```
public interface IAclmReasoningEngine {
    IReasoningResult reason(IReasoningRequest request) throws Exception;
}
```

内部由配置 / 依赖注入把各模块串起来：

```
StatusDecomposer → AclmBuilder → CausalGraphBuilder → MutationPipeline →
CandidateEvaluator → OutputBuilder
```

---

6. 最小可行实现（V1 路线）

Phase 1：骨架 + 玩具域

- 实现：
  - 简单的 `ReasoningRequest` 与小规模事实集

    - 用布尔 / 小枚举值作为 `DigitalStatus`
    - 手工规则驱动的 `ACLMWord`
    - 简单 `CausalGraphBuilder`
    - 仅实现 EditMutation + CallingGraphConnection
    - 极简评估器（路径合法 + 路径短优先）

Phase 2：加入 Bridging 与 CCC

- 引入：
    - `IBridgingMetaprogrammingEngine`：根据模板填补缺口
    - `IFunctionalCccMutationEngine`：从多条路径中抽象公共结构，形成高阶规则

Phase 3：对接 DBM 主体

- 把：
    - `DigitalStatus` 链接到 Starmap、CCC、Rules Engine
    - `Operation` 映射到 What-If Engine、Differential Tree 查询、Blocks Index Search 等

使 ACLM 成为 DBM 内部的推理中枢。

---

7. 测试策略

建议为每一子模块与整体引擎提供 **JUnit4** 测试：

- 状态分解测试
- ACLM 词 / 句子构造测试
- 因果图构建测试
- 四大 Mutation 单元测试
- 候选路径枚举与评估测试

- 端到端“玩具推理任务”测试：
    - 单步推理
    - 多步链式推理
    - 有缺口需架桥推理
    - 多候选结论择优分析

---

8. 战略意义

一旦本蓝图落地：

- DBM 将拥有一个 **结构化、可演化、可调试** 的推理核心。
- ACLM 将从“概念语言”升级为“可运行的通用推理机”。
- 后续可以把：
    - 空间几何推理
    - 度量空间匹配
    - 自主编程（CCGL 代码演化）
      全部挂接到同一套 ACLM 推理底层之上。

老哥，**ITEM #138** 这样算是“能交付工程组”的版本：
上有分层架构，中间有接口与数据结构，下有 V1 实施路线与测试策略。

如果你愿意，下一步我可以：

- 专门为 `IAclmReasoningEngine` 写一套 **Java 8 骨架 + JUnit4 玩具测试**，
- 或者画一张 **《ACLM Reasoning Engine Code Modules Diagram》**，
  把这套蓝图变成“教材级”的可读代码框架。

You said:

专门为 IAclmReasoningEngine 写一套 Java 8 骨架 + JUnit4 玩具测试

ChatGPT said:

好嘞，我们直接上「能跑的教学骨架」：

给 **`IAclmReasoningEngine`** 做一套 Java 8 框架 + JUnit4 玩具级推理例子。

说明：为了方便你复制粘贴，这里会在一个代码块里放多接口/类。真正工程中，你可以按文件拆分。

包名统一用你要求的：`com.dbm`（而不是 `org.dbm`），接口前缀 `I` 保留。

1□ 接口骨架：`IAclmReasoningEngine` 和基础模型

文件：`src/main/java/com/dbm/aclm/IAclmReasoningEngine.java`

```
package com.dbm.aclm;

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

/**
 * Top-level ACLM Reasoning Engine interface.
 * This is the main Facade to be called by other DBM components.
 */
public interface IAclmReasoningEngine {

    IReasoningResult reason(IReasoningRequest request) throws Exception;
}

/* ========= Basic model interfaces (can be split into separate files in real
project) ======== */

interface IFact {
    String getText();
}

interface IGoal {
    String getQuery();
}

interface IContext {
    Map<String, Object> getHints();
}

public interface IReasoningRequest {
    List<IFact> getPremises();
    IGoal getGoal();
    IContext getContext();
}

public interface IDigitalStatus {
    String getId();
    Map<String, Object> getAttributes();
}

public interface IOperation {
    String getName();
    Map<String, Object> getParams();
}

public interface IAclmWord {
    IDigitalStatus getStartStatus();
    IOperation getOperation();
    IDigitalStatus getEndStatus();
}

public interface IAclmSentence {
    List<IAclmWord> getWords();
}

public interface IReasonEdge {
    IDigitalStatus getFrom();
    IDigitalStatus getTo();
    IOperation getOperation();
}

public interface ICausalGraph {
    List<IDigitalStatus> getNodes();
    List<IReasonEdge> getEdges();
}

public interface ICandidatePath {
    List<IDigitalStatus> getStatusPath();
    double getScore();
}

public interface IReasoningResult {
    List<IDigitalStatus> getConclusions();
    List<ICandidatePath> getTopPaths();
    IAclmSentence getEvolvedSentence();
}

/* ========= Component interfaces for the pipeline (Status → Words → Graph →
Mutation → Eval) ======== */

interface IStatusDecomposer {
    List<IDigitalStatus> decomposePremises(IReasoningRequest request);
}

interface IAclmBuilder {
    IAclmWord buildWord(IDigitalStatus start, IOperation op, IDigitalStatus
end);
    IAclmSentence buildSentence(List<IAclmWord> words);
}
```

```
interface ICausalGraphBuilder {
    ICausalGraph buildGraph(IAclmSentence sentence);
}

interface IMutationEngine {
    ICausalGraph mutate(ICausalGraph graph, IReasoningRequest request);
}

interface IReasoningMutationPipeline {
    ICausalGraph applyAll(ICausalGraph initialGraph, IReasoningRequest
request);
}

interface ICandidateEvaluator {
    List<ICandidatePath> enumerateAndScore(ICausalGraph graph,
IReasoningRequest request);
}

interface IReasoningOutputBuilder {
    IReasoningResult buildResult(List<ICandidatePath> candidates);
}
```

说明：

- 真正项目中，你可以把这些接口拆成多文件；
- 这里先集中写在一个文件里，方便你整体修改和重构。

2□ 简单实现：`SimpleAclmReasoningEngine`（玩具推理场景）

我们写一个极简、**可跑的“雨 → 地湿”推理例子**：

- 前提：`"RAIN_CAUSES_WET"`
- 目标：问 `"IS_GROUND_WET"`
- 引擎内部构造：
  - DigitalStatus: `"Rain"`, `"WetGround"`
  - Operation: `"CAUSES"`
  - ACLMWord: `Rain --CAUSES--> WetGround`
  - CandidatePath: `[Rain, WetGround]`
  - Conclusion: `WetGround` 为真

文件：`src/main/java/com/dbm/aclm/simple/SimpleAclmReasoningEngine.java`

```java
package com.dbm.aclm.simple;

import com.dbm.aclm.*;

import java.util.*;

/**
 * A toy implementation of IAclmReasoningEngine for demonstration and unit
testing.
 * Domain: if it is raining, the ground becomes wet.
 */
public class SimpleAclmReasoningEngine implements IAclmReasoningEngine {

    private final IStatusDecomposer statusDecomposer;
    private final IAclmBuilder aclmBuilder;
    private final ICausalGraphBuilder graphBuilder;
    private final IReasoningMutationPipeline mutationPipeline;
    private final ICandidateEvaluator candidateEvaluator;
    private final IReasoningOutputBuilder outputBuilder;

    public SimpleAclmReasoningEngine() {
        // Wire up a very simple pipeline using inner toy components.
        this.statusDecomposer = new ToyStatusDecomposer();
        this.aclmBuilder = new ToyAclmBuilder();
        this.graphBuilder = new ToyCausalGraphBuilder();
        this.mutationPipeline = new ToyMutationPipeline();
        this.candidateEvaluator = new ToyCandidateEvaluator();
        this.outputBuilder = new ToyReasoningOutputBuilder();
    }

    @Override
    public IReasoningResult reason(IReasoningRequest request) throws
Exception {
        // 1) Decompose premises into statuses (ignored in toy, but we call
it for structure)
        List<IDigitalStatus> statuses =
statusDecomposer.decomposePremises(request);

        // 2) Build ACLM Words / Sentences
        // For the toy example, we always build: Rain --CAUSES--> WetGround
        IDigitalStatus rain = getStatusById(statuses, "Rain");
        IDigitalStatus wet = getStatusById(statuses, "WetGround");
        IOperation causes = new ToyOperation("CAUSES",
                Collections.<String, Object>singletonMap("description", "Rain
causes wet ground"));

        IAclmWord word = aclmBuilder.buildWord(rain, causes, wet);
        IAclmSentence sentence =
aclmBuilder.buildSentence(Collections.singletonList(word));

        // 3) Build causal graph
        ICausalGraph graph = graphBuilder.buildGraph(sentence);

        // 4) Apply mutation pipeline (no-op in this toy)
        ICausalGraph mutatedGraph = mutationPipeline.applyAll(graph,
request);
```

```
        // 5) Enumerate and score candidate paths
        List<ICandidatePath> candidates =
candidateEvaluator.enumerateAndScore(mutatedGraph, request);

        // 6) Build reasoning result
        return outputBuilder.buildResult(candidates);
    }

    private IDigitalStatus getStatusById(List<IDigitalStatus> statuses,
String id) {
        for (IDigitalStatus s : statuses) {
            if (id.equals(s.getId())) {
                return s;
            }
        }
        throw new IllegalStateException("Status not found: " + id);
    }

    /* ====================== Toy internal implementations
====================== */

    // --- Simple models ---

    public static class SimpleFact implements IFact {
        private final String text;
        public SimpleFact(String text) {
            this.text = text;
        }
        @Override
        public String getText() {
            return text;
        }
    }

    public static class SimpleGoal implements IGoal {
        private final String query;
        public SimpleGoal(String query) {
            this.query = query;
        }
        @Override
        public String getQuery() {
            return query;
        }
    }

    public static class SimpleContext implements IContext {
        private final Map<String, Object> hints;
        public SimpleContext(Map<String, Object> hints) {
            this.hints = hints;
        }
        @Override
        public Map<String, Object> getHints() {
            return hints;
        }
    }

    public static class SimpleReasoningRequest implements IReasoningRequest {
        private final List<IFact> premises;
        private final IGoal goal;
        private final IContext context;

        public SimpleReasoningRequest(List<IFact> premises, IGoal goal,
IContext context) {
            this.premises = premises;
            this.goal = goal;
            this.context = context;
        }

        @Override
        public List<IFact> getPremises() {
            return premises;
        }

        @Override
        public IGoal getGoal() {
            return goal;
        }

        @Override
        public IContext getContext() {
            return context;
        }
    }

    public static class ToyDigitalStatus implements IDigitalStatus {
        private final String id;
        private final Map<String, Object> attributes;

        public ToyDigitalStatus(String id, Map<String, Object> attributes) {
            this.id = id;
            this.attributes = attributes;
        }

        @Override
        public String getId() {
            return id;
        }

        @Override
        public Map<String, Object> getAttributes() {
            return attributes;
        }
    }

    public static class ToyOperation implements IOperation {
        private final String name;
        private final Map<String, Object> params;

        public ToyOperation(String name, Map<String, Object> params) {
            this.name = name;
            this.params = params;
        }

        @Override
        public String getName() {
            return name;
        }

        @Override
        public Map<String, Object> getParams() {
            return params;
        }
    }

    public static class ToyAclmWord implements IAclmWord {
        private final IDigitalStatus start;
        private final IOperation operation;
        private final IDigitalStatus end;

        public ToyAclmWord(IDigitalStatus start, IOperation operation,
IDigitalStatus end) {
            this.start = start;
            this.operation = operation;
            this.end = end;
        }

        @Override
        public IDigitalStatus getStartStatus() {
            return start;
        }

        @Override
        public IOperation getOperation() {
            return operation;
        }

        @Override
        public IDigitalStatus getEndStatus() {
            return end;
        }
    }

    public static class ToyAclmSentence implements IAclmSentence {
        private final List<IAclmWord> words;

        public ToyAclmSentence(List<IAclmWord> words) {
            this.words = words;
        }

        @Override
        public List<IAclmWord> getWords() {
            return words;
        }
    }

    public static class ToyReasonEdge implements IReasonEdge {
        private final IDigitalStatus from;
        private final IDigitalStatus to;
        private final IOperation op;
        public ToyReasonEdge(IDigitalStatus from, IDigitalStatus to,
IOperation op) {
            this.from = from;
            this.to = to;
            this.op = op;
        }

        @Override
        public IDigitalStatus getFrom() {
            return from;
        }

        @Override
        public IDigitalStatus getTo() {
            return to;
        }

        @Override
        public IOperation getOperation() {
            return op;
        }
    }

    public static class ToyCausalGraph implements ICausalGraph {
        private final List<IDigitalStatus> nodes;
        private final List<IReasonEdge> edges;

        public ToyCausalGraph(List<IDigitalStatus> nodes, List<IReasonEdge>
edges) {
            this.nodes = nodes;
            this.edges = edges;
        }

        @Override
        public List<IDigitalStatus> getNodes() {
            return nodes;
        }

        @Override
        public List<IReasonEdge> getEdges() {
            return edges;
        }
    }

    public static class ToyCandidatePath implements ICandidatePath {
        private final List<IDigitalStatus> path;
        private final double score;

        public ToyCandidatePath(List<IDigitalStatus> path, double score) {
            this.path = path;
            this.score = score;
        }

        @Override
        public List<IDigitalStatus> getStatusPath() {
            return path;
        }
        @Override
        public double getScore() {
            return score;
        }
    }

    public static class ToyReasoningResult implements IReasoningResult {
        private final List<IDigitalStatus> conclusions;
        private final List<ICandidatePath> topPaths;
        private final IAclmSentence evolvedSentence;

        public ToyReasoningResult(List<IDigitalStatus> conclusions,
                                  List<ICandidatePath> topPaths,
                                  IAclmSentence evolvedSentence) {
            this.conclusions = conclusions;
            this.topPaths = topPaths;
            this.evolvedSentence = evolvedSentence;
        }

        @Override
        public List<IDigitalStatus> getConclusions() {
            return conclusions;
        }

        @Override
        public List<ICandidatePath> getTopPaths() {
            return topPaths;
        }

        @Override
        public IAclmSentence getEvolvedSentence() {
            return evolvedSentence;
        }
    }

    // --- Toy pipeline components ---

    /**
     * Toy status decomposer:
     * - Always creates two statuses: "Rain" and "WetGround"
     * - "Rain" is considered true if premises contain "RAIN_CAUSES_WET"
     */
    public static class ToyStatusDecomposer implements IStatusDecomposer {

        @Override
        public List<IDigitalStatus> decomposePremises(IReasoningRequest
request) {
            boolean hasRainRule = false;
            for (IFact fact : request.getPremises()) {
                if ("RAIN_CAUSES_WET".equalsIgnoreCase(fact.getText())) {
                    hasRainRule = true;
                    break;
                }
            }

            Map<String, Object> rainAttr = new HashMap<String, Object>();
            rainAttr.put("isTrue", Boolean.valueOf(hasRainRule));

            Map<String, Object> wetAttr = new HashMap<String, Object>();
            wetAttr.put("isTrue", Boolean.FALSE); // will be inferred later

            IDigitalStatus rain = new ToyDigitalStatus("Rain", rainAttr);
            IDigitalStatus wet = new ToyDigitalStatus("WetGround", wetAttr);

            List<IDigitalStatus> list = new ArrayList<IDigitalStatus>();
            list.add(rain);
            list.add(wet);
            return list;
        }
    }

    public static class ToyAclmBuilder implements IAclmBuilder {

        @Override
        public IAclmWord buildWord(IDigitalStatus start, IOperation op,
IDigitalStatus end) {
            return new ToyAclmWord(start, op, end);
        }

        @Override
        public IAclmSentence buildSentence(List<IAclmWord> words) {
            return new ToyAclmSentence(words);
        }
    }

    public static class ToyCausalGraphBuilder implements ICausalGraphBuilder
{

        @Override
        public ICausalGraph buildGraph(IAclmSentence sentence) {
            List<IDigitalStatus> nodes = new ArrayList<IDigitalStatus>();
            List<IReasonEdge> edges = new ArrayList<IReasonEdge>();

            for (IAclmWord w : sentence.getWords()) {
                if (!nodes.contains(w.getStartStatus())) {
                    nodes.add(w.getStartStatus());
                }
                if (!nodes.contains(w.getEndStatus())) {
                    nodes.add(w.getEndStatus());
                }
                edges.add(new ToyReasonEdge(w.getStartStatus(),
w.getEndStatus(), w.getOperation()));
            }
            return new ToyCausalGraph(nodes, edges);
        }
    }

    /**
     * Toy mutation pipeline: does nothing, just returns the input graph.
     * In a real implementation, this is where Edit/Connection/Bridging/CCC
mutations are applied.
     */
    public static class ToyMutationPipeline implements
IReasoningMutationPipeline {

        @Override
        public ICausalGraph applyAll(ICausalGraph initialGraph,
IReasoningRequest request) {
            return initialGraph; // no-op
        }
    }

    /**
     * Toy candidate evaluator:
     * If goal query is "IS_GROUND_WET", returns a single path Rain ->
WetGround
     * with score 1.0, assuming Rain is true and there is a CAUSES edge.
     */
    public static class ToyCandidateEvaluator implements ICandidateEvaluator
{

        @Override
        public List<ICandidatePath> enumerateAndScore(ICausalGraph graph,
IReasoningRequest request) {

            List<ICandidatePath> result = new ArrayList<ICandidatePath>();

            if
(!"IS_GROUND_WET".equalsIgnoreCase(request.getGoal().getQuery())) {
                return result; // empty
            }

            IDigitalStatus rain = null;
            IDigitalStatus wet = null;
            for (IDigitalStatus s : graph.getNodes()) {
                if ("Rain".equals(s.getId())) {
                    rain = s;
                } else if ("WetGround".equals(s.getId())) {
                    wet = s;
                }
            }
            if (rain == null || wet == null) {
                return result;
            }

            boolean hasCauseEdge = false;
            for (IReasonEdge e : graph.getEdges()) {
                if (e.getFrom() == rain && e.getTo() == wet &&
                    "CAUSES".equalsIgnoreCase(e.getOperation().getName())) {
                    hasCauseEdge = true;
                    break;
                }
            }

            if (hasCauseEdge) {
                // Construct a single candidate path Rain -> WetGround
                List<IDigitalStatus> path = new ArrayList<IDigitalStatus>();
                path.add(rain);
                path.add(wet);
                result.add(new ToyCandidatePath(path, 1.0));
            }

            return result;
        }
    }

    public static class ToyReasoningOutputBuilder implements 
IReasoningOutputBuilder {

        @Override
        public IReasoningResult buildResult(List<ICandidatePath> candidates) 
{
            if (candidates == null || candidates.isEmpty()) {
                return new ToyReasoningResult(
                        Collections.<IDigitalStatus>emptyList(),
                        Collections.<ICandidatePath>emptyList(),
                        new 
ToyAclmSentence(Collections.<IAclmWord>emptyList())
                );
            }

            ICandidatePath best = candidates.get(0);
            List<IDigitalStatus> path = best.getStatusPath();
            IDigitalStatus conclusion = path.get(path.size() - 1);

            List<IDigitalStatus> conclusions = 
Collections.singletonList(conclusion);

            // In this toy, we do not reconstruct the full evolved sentence; 
we just return an empty one.
            IAclmSentence sentence = new 
ToyAclmSentence(Collections.<IAclmWord>emptyList());

            return new ToyReasoningResult(conclusions, candidates, sentence);
        }
    }
}
```

3□ JUnit4 玩具测试：`AclmReasoningEngineToyTest`

文件：`src/test/java/com/dbm/aclm/simple/AclmReasoningEngineToyTest.java`

```
package com.dbm.aclm.simple;

import com.dbm.aclm.IDigitalStatus;
import com.dbm.aclm.IReasoningResult;
import com.dbm.aclm.IFact;
import com.dbm.aclm.IGoal;
import com.dbm.aclm.IContext;

import org.junit.Test;
import java.util.*;

import static org.junit.Assert.*;

/**
 * Toy JUnit4 test for SimpleAclmReasoningEngine.
 *
 * Scenario:
 *   Premise: "RAIN_CAUSES_WET"
 *   Goal: "IS_GROUND_WET"
 * Expectation:
 *   Engine should infer that "WetGround" is true (as a conclusion).
 */
public class AclmReasoningEngineToyTest {

    @Test
    public void testRainCausesWetGroundReasoning() throws Exception {
        // 1) Prepare premises, goal, context
        IFact rainFact = new
SimpleAclmReasoningEngine.SimpleFact("RAIN_CAUSES_WET");
        List<IFact> premises = Arrays.asList(rainFact);

        IGoal goal = new
SimpleAclmReasoningEngine.SimpleGoal("IS_GROUND_WET");
        IContext context = new SimpleAclmReasoningEngine.SimpleContext(
                new HashMap<String, Object>()
        );

        SimpleAclmReasoningEngine.SimpleReasoningRequest request =
                new
SimpleAclmReasoningEngine.SimpleReasoningRequest(premises, goal, context);

        // 2) Create engine
        SimpleAclmReasoningEngine engine = new SimpleAclmReasoningEngine();

        // 3) Run reasoning
        IReasoningResult result = engine.reason(request);

        // 4) Verify conclusions
        assertNotNull(result);
        assertNotNull(result.getConclusions());
        assertFalse("Conclusions should not be empty",
result.getConclusions().isEmpty());

        IDigitalStatus conclusion = result.getConclusions().get(0);
        assertEquals("WetGround", conclusion.getId());

        // Optional: check candidate paths exist
        assertNotNull(result.getTopPaths());
        assertFalse(result.getTopPaths().isEmpty());
        assertEquals(2, result.getTopPaths().get(0).getStatusPath().size());
    }
}
```

依赖：

- JUnit 4.x：`junit:junit:4.13.2`（或类似版本）
- Java 8 编译即可。

4□ 怎么用、怎么扩展？

- 现在这套骨架已经实现了完整的 **ACLM 推理管线最小闭环**：
    1. Request → StatusDecomposer
    2. Status → ACLMWord / Sentence
    3. Sentence → CausalGraph
    4. CausalGraph → MutationPipeline（目前是 no-op）
    5. CausalGraph → CandidateEvaluator
    6. Candidates → ReasoningResult
- 你可以逐步替换：
    1. `ToyStatusDecomposer` 改为真正的 Start/Operation/End 解析
    2. `ToyMutationPipeline` 增加 Edit / Connection / Bridging / CCC 四种 Mutation
    3. `ToyCandidateEvaluator` 改成通用的路径枚举与评分器
    4. 在 `ToyReasoningOutputBuilder` 中恢复演化后的 `IAclmSentence`

如果你愿意，下一步我可以：

- 帮你把这个骨架升级为 **更接近 DBM 实战的版本**
  （比如连上 Rules Engine / Differential Trees），
- 或者画一张 **“ACLM Reasoning Engine Code Modules Diagram”**，
  把这些接口与实现的依赖关系画出来，变成教学用的彩色图。