IKE Coordinate System - Developer's Guide

Table of Contents

A comprehensive guide to understanding and using the IKE Coordinate *System (Integrated Knowledge Ecosystem) for managing temporal, versioned knowledge graphs. This guide covers all five coordinate types (STAMP, Language, Logic, Navigation, and Edit) and shows how to configure them for different use cases.*

Note: This coordinate system was originally published as part of the HL7 Terminology Knowledge Architecture (TINKAR) ballot. While TINKAR remains the historical and technical foundation, the open-source community now refers to this collaborative framework as the Integrated Knowledge Ecosystem (IKE).

1. Document Overview

This Getting Started Guide provides Java developers and users with practical knowledge of the IKE Coordinate System. The guide is organized into the following sections:

- **Introduction** - What coordinates are and why they matter
- **STAMP Coordinates** - Temporal versioning and provenance
- **Language Coordinates** - Multilingual description management
- **Logic Coordinates** - Description logic and reasoning
- **Navigation Coordinates** - Graph traversal and hierarchies
- **Edit Coordinates** - Change attribution and authoring
- **View Coordinates** - Unified coordinate access
- **Practical Examples** - Real-world configuration scenarios
- **Best Practices** - Tips for effective coordinate usage

2. Target Audience

This guide is designed for:

- **Java Developers** building applications on the IKE platform
- **Knowledge Engineers** configuring terminology systems
- **System Administrators** managing knowledge bases
- **Integration Specialists** connecting IKE to other systems

3. Prerequisites

To use this guide effectively, you should have:

- Basic Java programming knowledge
- Understanding of concepts like immutability and threading
- Familiarity with terminology and knowledge representation (helpful but not required)

4. How to Use This Guide

Each section can be read independently, but we recommend reading them in order for a complete understanding:

1. Start with the **Introduction** to understand the overall architecture
2. Read each coordinate type section based on your needs
3. Review the **Practical Examples** for real-world scenarios
4. Consult the **Best Practices** when implementing

> Code examples are provided throughout. You can copy and adapt them for your own use.

5. Getting Help

If you need assistance:

- Review the JavaDoc in the `dev.ikm.tinkar.coordinate` package
- Check the code examples in each section
- Consult the API reference for detailed method signatures
- Reach out to the IKE community for support

6. Document Sections

7. Introduction to Coordinates

7.1. What Are Coordinates?

In the IKE Coordinate System, a **coordinate** is a specification that describes **how** to access and interpret knowledge at a specific point in time, in a specific language, using specific reasoning rules, and through a specific navigational structure.

Think of coordinates like viewing settings for a knowledge graph:

- A STAMP coordinate is like setting a "point in time" filter
- A Language coordinate is like choosing your preferred language
- A Logic coordinate is like selecting which reasoning engine to use
- A Navigation coordinate is like choosing which hierarchy to browse
- An Edit coordinate is like setting your author identity

Together, these coordinates provide a complete context for accessing versioned, multilingual, logic-enabled knowledge.

7.2. About IKE and TINKAR

The **IKE Coordinate System** is the collaborative, open-source framework developed by our independent community for managing complex knowledge at scale. The architecture is based on the HL7 Terminology Knowledge Architecture (TINKAR), which was formally balloted by HL7. While **TINKAR** remains valid as the historical and technical foundation, we use **IKE (Integrated Knowledge Ecosystem)** to emphasize:

- **Collaborative**: Multiple organizations and individuals contributing together
- **Open-source**: Transparent development and freely available implementations
- **Ecosystem**: A comprehensive environment for knowledge curation at scale
- **Independent**: Community-driven rather than single-organization controlled

Throughout this documentation, you may see references to "Tinkar" in package names (`dev.ikm.tinkar.*`) and code—this reflects the historical naming and maintains backward compatibility.

7.3. Why Coordinates Matter

7.3.1. The Problem: Knowledge Changes Over Time

Medical terminologies, business vocabularies, and other knowledge bases are not static. They evolve:

- Concepts are added, modified, or deprecated
- Descriptions are refined or translated
- Relationships change as understanding improves
- Different teams work on different branches

Without coordinates, answering "What is the definition of Pneumonia?" is ambiguous:

- Which version? (from yesterday, last month, or today?)
- Which language? (English, Spanish, or French?)
- Which hierarchy? (stated by authors or inferred by classifier?)

- Which development branch? (production, staging, or development?)

With coordinates, the question becomes precise:

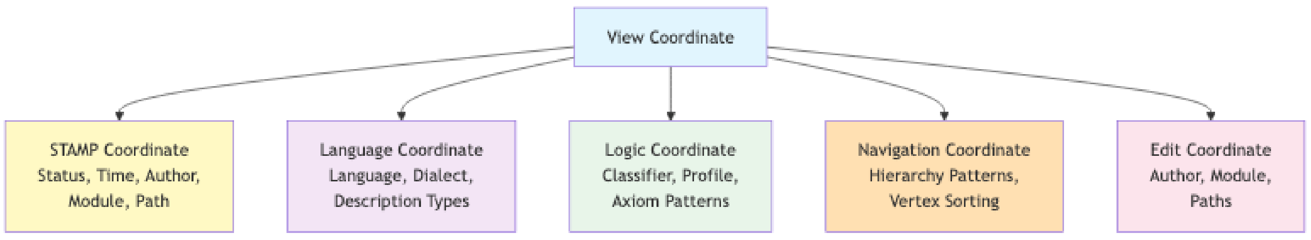
```
ViewCoordinateRecord view = ViewCoordinateRecord.make(
    Coordinates.Stamp.MasterLatestActiveOnly(),      // Production, latest, active
only
    Lists.immutable.of(
        Coordinates.Language.UsEnglishRegularName()  // US English, regular names
    ),
    Coordinates.Logic.ElPlusPlus(),                  // EL++ reasoning
    Coordinates.Navigation.inferred(),               // Inferred hierarchy
    Coordinates.Edit.Default()                       // Default edit context
);

ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);
String description = calc.getDescriptionText(pneumoniaNid);
// Returns: "Pneumonia" (as of latest master, in US English, active only)
```

7.3.2. The Solution: Declarative Coordinates

Coordinates provide a **declarative** way to specify your viewing context:

- **Consistency** - All operations use the same coordinate, ensuring coherent results
- **Reproducibility** - Same coordinates always produce same results
- **Flexibility** - Switch contexts by changing coordinates
- **Transparency** - Coordinates make viewing assumptions explicit

7.4. The Five Coordinate Types


View Coordinate
STAMP Coordinate Status, Time, Author, Module, Path
Language Coordinate Language, Dialect, Description Types
Logic Coordinate Classifier, Profile, Axiom Patterns
Navigation Coordinate Hierarchy Patterns, Vertex Sorting
Edit Coordinate Author, Module, Paths

7.4.1. STAMP Coordinates

Control **temporal versioning** and **provenance**:

Component	Purpose
Status	ACTIVE or INACTIVE (filter by lifecycle state)
Time	Timestamp (view knowledge as of specific date/time)
Author	Who made changes (not used in filtering, just metadata)

Component	Purpose
Module	Organizational unit (include/exclude specific modules)
Path	Development branch (production, staging, development)

Key Use Cases:

- "Show me active content only"
- "What did the terminology look like on January 1, 2024?"
- "Show me only content from the SNOMED CT core module"

7.4.2. Language Coordinates

Control **multilingual descriptions**:

Component	Purpose
Language	Natural language (English, Spanish, French, etc.)
Description Patterns	Which patterns define descriptions
Description Types	Prefer FQN, Regular Name, or Definition
Dialect Preferences	US English vs. GB English, etc.
Module Preferences	When multiple descriptions exist, which module wins

Key Use Cases:

- "Show me Spanish descriptions"
- "Prefer fully qualified names over regular names"
- "Use US English spellings when available, fall back to GB English"

7.4.3. Logic Coordinates

Control **description logic reasoning**:

Component	Purpose
Classifier	Reasoning engine (Snorocket, ELK, etc.)
Profile	Logic expressivity (EL++, ALC, SROIQ)
Axiom Patterns	Where to find stated and inferred logical definitions
Navigation Patterns	How axioms generate navigational hierarchies
Root Concept	Top of the taxonomy

Key Use Cases:

- "Classify concepts using EL++ description logic"
- "Get inferred parent-child relationships"

- "Check if concept A subsumes concept B"

7.4.4. Navigation Coordinates

Control **graph traversal** and **hierarchies**:

Component	Purpose
Navigation Patterns	Which graph(s) to use (inferred, stated, custom)
Vertex States	Include active, inactive, or both
Vertex Sorting	Sort children alphabetically or by custom order
Sort Patterns	Custom sorting criteria

Key Use Cases:

- "Navigate the inferred subsumption hierarchy"
- "Browse the stated (author-asserted) hierarchy"
- "Combine is-a and part-of relationships in one view"

7.4.5. Edit Coordinates

Control **change attribution**:

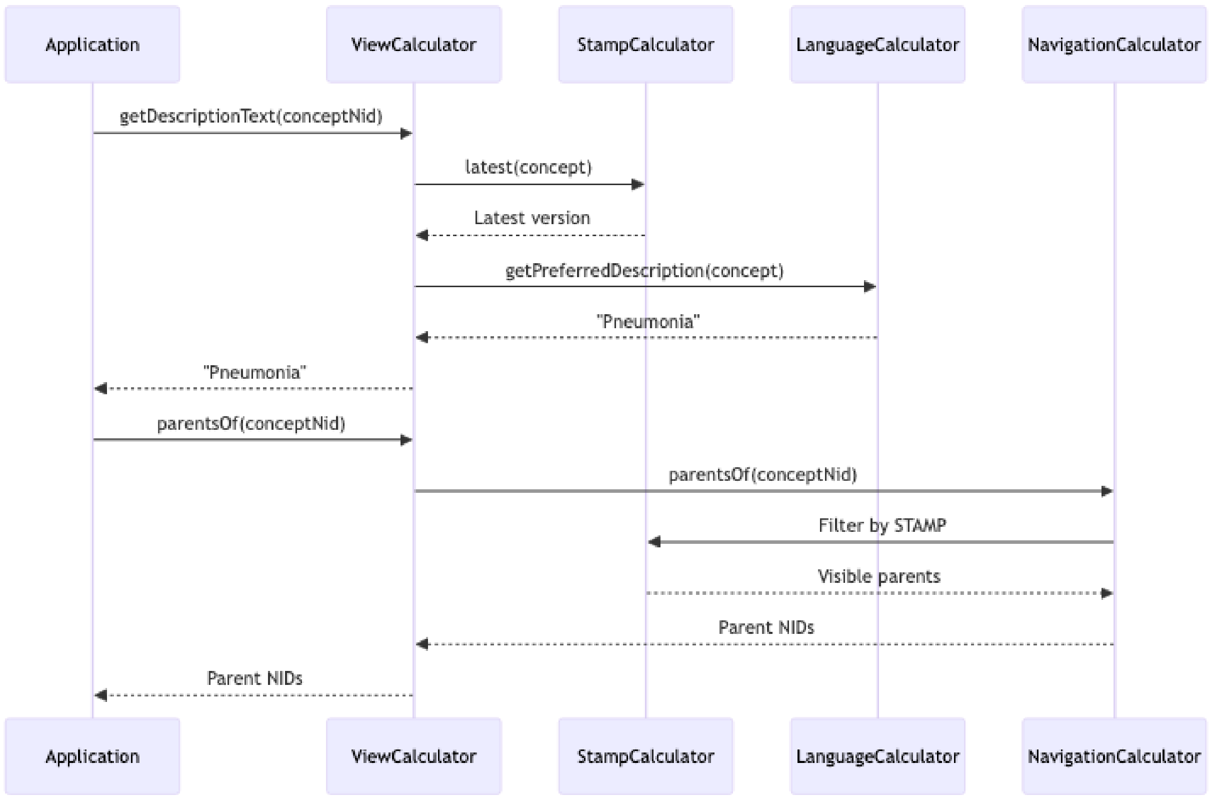
Component	Purpose
Author	Who is making changes
Default Module	Where new content goes
Destination Module	Target for modularization
Default Path	Where to create new content
Promotion Path	Target for content promotion

Key Use Cases:

- "Attribute changes to Dr. Smith"
- "Create new content in the Extension module"
- "Promote content from development to production"

7.5. How Coordinates Work Together

All five coordinate types combine in a **View Coordinate** to provide unified access:

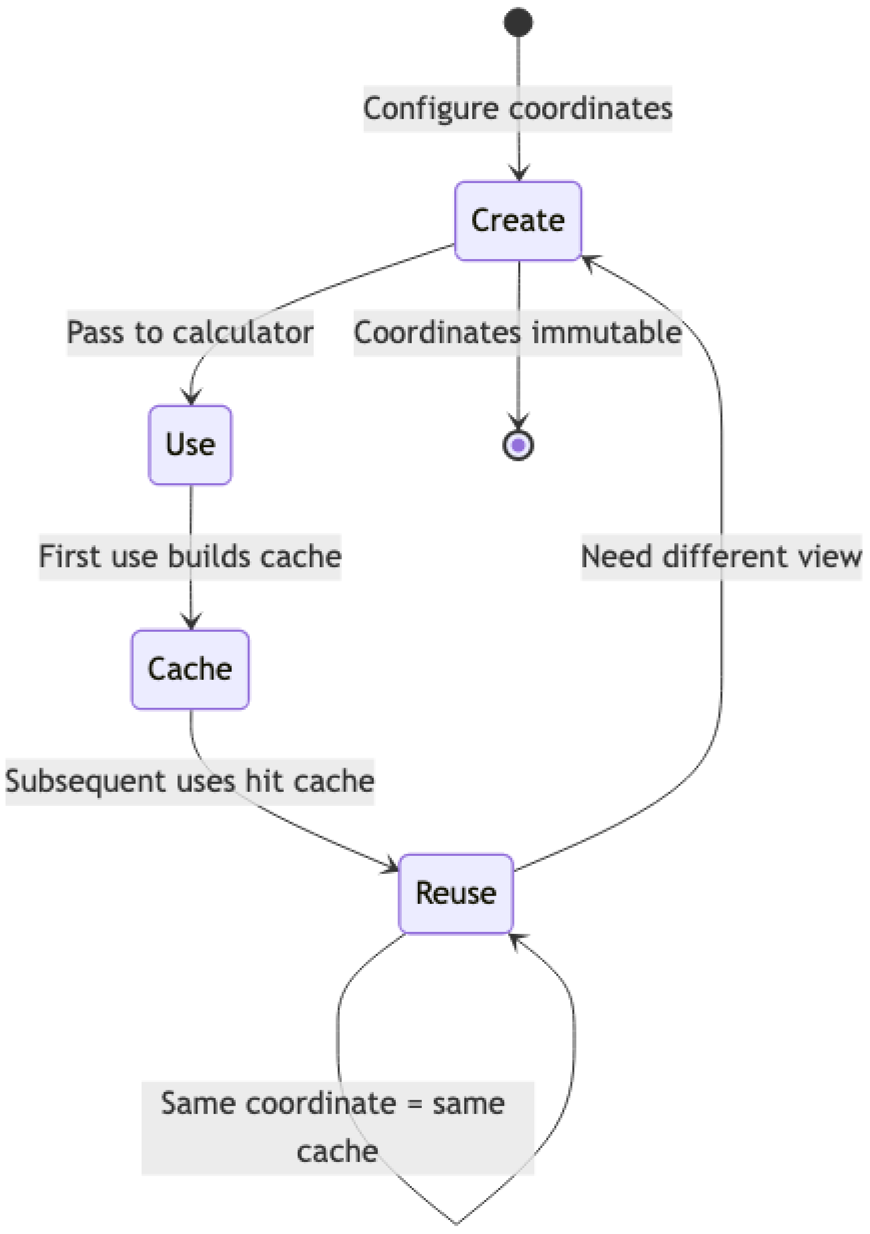


The View Coordinate ensures:

1. **Consistency** - Same STAMP coordinate used for all version resolution
2. **Integration** - Language, Logic, and Navigation work together seamlessly
3. **Simplicity** - Single point of configuration for entire application

7.6. Coordinate Lifecycle

Coordinates follow this lifecycle:



Key Points:

- Coordinates are **immutable** - Once created, they never change
- Calculators **cache** results per coordinate
- **Reusing** coordinates maximizes cache efficiency
- **Creating** new coordinates means new cache entries

7.7. Common Patterns

7.7.1. Pattern 1: Application Startup

```
// Create standard view at startup
public class MyApplication {
    // Reuse this coordinate throughout application
    private static final ViewCoordinateRecord STANDARD_VIEW =
        Coordinates.View.DefaultView();

    private static final ViewCalculator CALCULATOR =
        ViewCalculatorWithCache.getCalculator(STANDARD_VIEW);
}
```

7.7.2. Pattern 2: User Preferences

```
// Build view from user preferences
public ViewCoordinateRecord buildUserView(UserPreferences prefs) {
    return ViewCoordinateRecord.make(
        buildStampFromPrefs(prefs),
        buildLanguagesFromPrefs(prefs),
        Coordinates.Logic.ElPlusPlus(),
        Coordinates.Navigation.inferred(),
        Coordinates.Edit.Default()
    );
}
```

7.7.3. Pattern 3: Temporary Context Switch

```
// Temporarily view historical state
ViewCoordinateRecord currentView = Coordinates.View.DefaultView();
ViewCalculator currentCalc = ViewCalculatorWithCache.getCalculator(currentView);

// Switch to historical view
StampPositionRecord historicPosition = StampPositionRecord.make(
    historicTimestamp,
    TinkarTerm.MASTER_PATH
);
ViewCoordinateRecord historicView = currentView.withStampCoordinate(
    currentView.stampCoordinate().withStampPosition(historicPosition)
);
ViewCalculator historicCalc = ViewCalculatorWithCache.getCalculator(historicView);

// Compare
String currentDesc = currentCalc.getDescriptionText(conceptNid);
String historicDesc = historicCalc.getDescriptionText(conceptNid);
```

7.8. What’s Next?

Now that you understand what coordinates are and why they matter, the following sections will teach you how to configure each coordinate type for your specific needs.

Each section includes:

- Detailed explanation of the coordinate type

- Configuration options and parameters
- Code examples and use cases
- Best practices and tips

You can read sections in any order, but STAMP coordinates are foundational and recommended as the starting point.

8. STAMP Coordinates

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

8.1. Overview

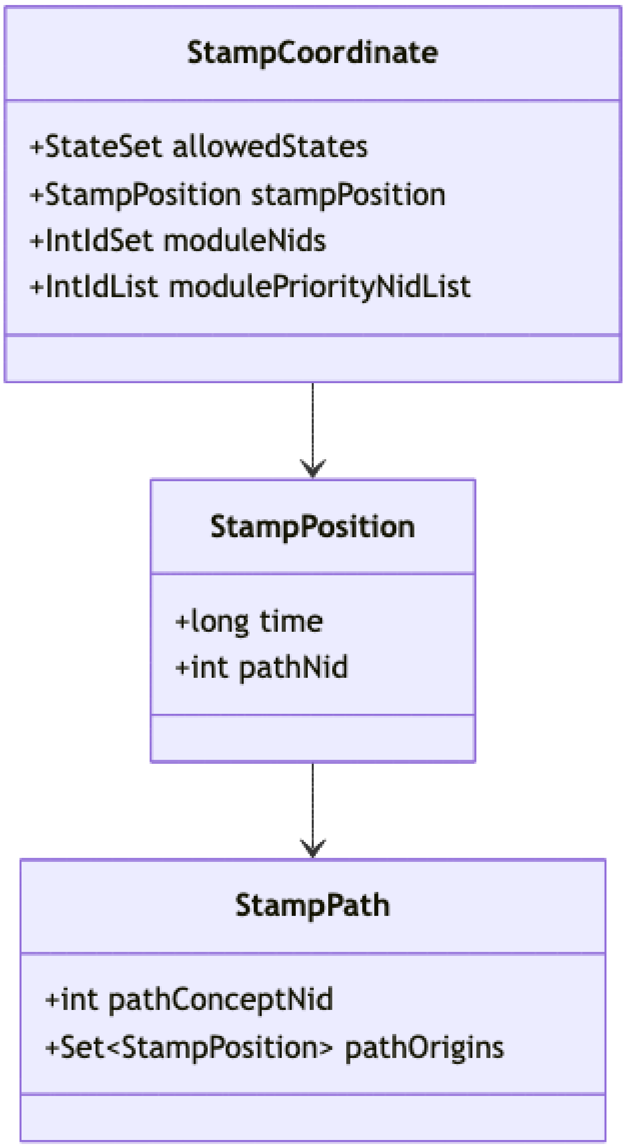
STAMP coordinates control **temporal versioning** and **provenance tracking** in the IKE Coordinate System. STAMP is an acronym for:

- *S*tatus - Lifecycle state (ACTIVE or INACTIVE)
- *T*ime - When the version was created
- *A*uthor - Who made the change
- *M*odule - Which organizational unit owns the content
- *P*ath - Which development branch contains the version

Every version of every entity in the IKE Coordinate System carries STAMP metadata. STAMP coordinates specify which versions you want to see.

8.2. The STAMP Hierarchy

STAMP coordinates are built from three levels:



8.2.1. StampPosition: Time and Path

A position specifies **when** on **which path**:

```
// Latest on development path
StampPositionRecord latest = StampPositionRecord.make(
    Long.MAX_VALUE,                          // Latest
    TinkarTerm.DEVELOPMENT_PATH
);

// Specific date on master path
StampPositionRecord historic = StampPositionRecord.make(
    Instant.parse("2024-01-01T00:00:00Z").toEpochMilli(),
    TinkarTerm.MASTER_PATH
);
```

8.2.2. StampPath: Development Branches

Paths represent development branches with their origins:

```
// Master path branches from primordial path
StampPathImmutable masterPath = StampPathImmutable.make(
    TinkarTerm.MASTER_PATH,
    Sets.immutable.of(
        StampPositionRecord.make(Long.MAX_VALUE, TinkarTerm.PRIMORDIAL_PATH.nid())
    )
);

// Development branches from master at specific time
StampPathImmutable devPath = StampPathImmutable.make(
    TinkarTerm.DEVELOPMENT_PATH,
    Sets.immutable.of(
        StampPositionRecord.make(branchTimestamp, TinkarTerm.MASTER_PATH.nid())
    )
);
```

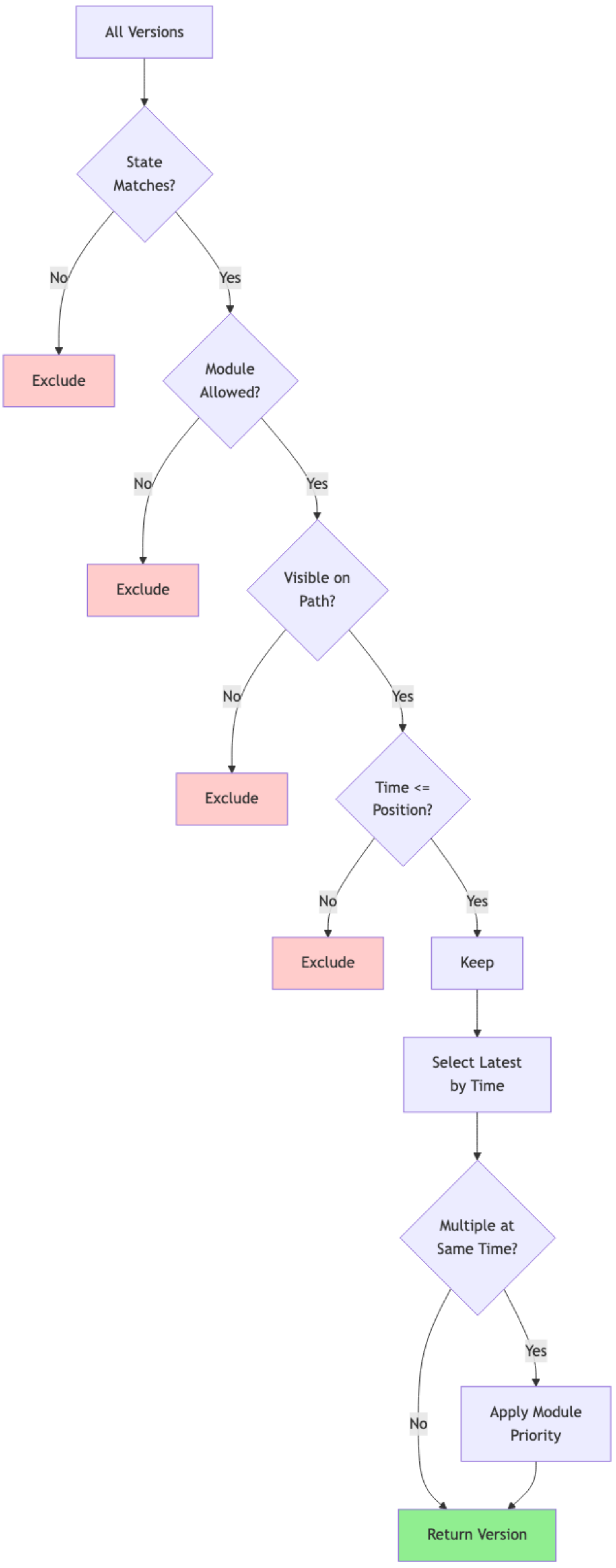
8.2.3. StampCoordinate: Complete Filter

A coordinate combines position with state and module filters:

```
StampCoordinateRecord coordinate = StampCoordinateRecord.make(
    StateSet.ACTIVE,                     // Only active content
    StampPositionRecord.make(            // Latest on master
        Long.MAX_VALUE,
        TinkarTerm.MASTER_PATH
    ),
    IntIds.set.empty()                   // All modules (wildcard)
);
```

8.3. Version Selection Algorithm

When you request the "latest" version, STAMP coordinates apply this algorithm:


All Versions
State Matches?
No
Yes
Exclude
Module Allowed?
No
Yes
Exclude
Visible on Path?
No
Yes
Exclude
Time <= Position?
No
Yes
Exclude
Keep
Select Latest by Time
Multiple at Same Time?
No
Yes
Apply Module Priority
Return Version

8.4. State Filtering

8.4.1. StateSet Options

```
// Active only (most common)
StateSet activeOnly = StateSet.ACTIVE;

// Inactive only (for retirement analysis)
StateSet inactiveOnly = StateSet.INACTIVE;

// Both (for administrative views)
StateSet both = StateSet.ACTIVE_AND_INACTIVE;
```

8.4.2. Use Cases

Table 1. When to Use Each StateSet

StateSet	**Use Case**	**Example**
ACTIVE	Normal application use	End-user interfaces, API responses
INACTIVE	Retirement analysis	Finding deprecated concepts, auditing deletions
ACTIVE_AND_INACTIVE	Administrative views	System administration, data analysis, quality assurance

8.5. Time-Based Queries

8.5.1. Latest (Most Common)

```
// Latest versions
StampPositionRecord latest = StampPositionRecord.make(
    Long.MAX_VALUE,
    TinkarTerm.DEVELOPMENT_PATH
);
```

8.5.2. Point-in-Time

```
// View as of January 1, 2024
long timestamp = Instant.parse("2024-01-01T00:00:00Z").toEpochMilli();
StampPositionRecord pointInTime = StampPositionRecord.make(
    timestamp,
    TinkarTerm.MASTER_PATH
);

// Use in coordinate
```

```
StampCoordinateRecord historic = StampCoordinateRecord.make(
    StateSet.ACTIVE,
    pointInTime,
    IntIds.set.empty()
);

// Query with historic coordinate
StampCalculator calc = StampCalculatorWithCache.getCalculator(historic);
Latest<ConceptVersion> historicVersion = calc.latest(concept);
```

8.5.3. Time Range Analysis

```
// Find what changed between two dates
StampPositionRecord start = StampPositionRecord.make(
    startTimestamp,
    TinkarTerm.MASTER_PATH
);
StampPositionRecord end = StampPositionRecord.make(
    endTimestamp,
    TinkarTerm.MASTER_PATH
);

// Get changes
StampCalculator calc = StampCalculatorWithCache.getCalculator(
    Coordinates.Stamp.MasterLatestActiveOnly()
);
ChangeChronology changes = calc.getChanges(concept, start, end);

if (!changes.versionChanges().isEmpty()) {
    System.out.println("Concept changed between dates");
}
```

8.6. Module Filtering

8.6.1. Module Inclusion

```
// Include specific modules only
IntIdSet allowedModules = IntIds.set.of(
    TinkarTerm.SOLOR_MODULE.nid(),
    TinkarTerm.SNOMED_CT_CORE_MODULE.nid()
);

StampCoordinateRecord filtered = coordinate.withModuleNids(allowedModules);
```

8.6.2. Module Exclusion

```
// Exclude deprecated modules
IntIdSet excludedModules = IntIds.set.of(
    TinkarTerm.DEPRECATED_MODULE.nid()
);

StampCoordinateRecord filtered = coordinate.withExcludedModuleNids(excludedModules);
```

8.6.3. Module Priority

When multiple versions exist with the same timestamp:

```
// Prefer versions from these modules (in order)
IntIdList modulePriority = IntIds.list.of(
    TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),  // 1st preference
    TinkarTerm.SOLOR_MODULE.nid()           // 2nd preference
);

StampCoordinateRecord prioritized =
coordinate.withModulePriorityNidList(modulePriority);
```

8.7. Path-Based Versioning

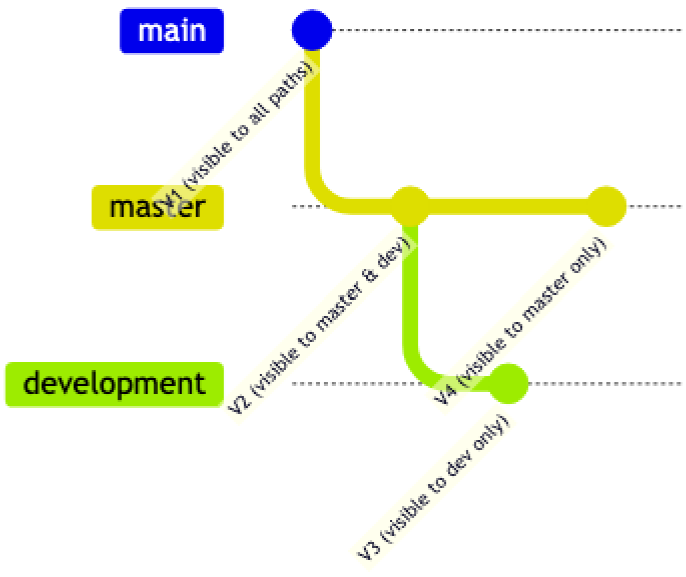
8.7.1. Common Paths

Path	Purpose
PRIMORDIAL_PATH	Initial import, root of all paths
MASTER_PATH	Production/released content
DEVELOPMENT_PATH	Active development
SANDBOX_PATH	Experimental changes

8.7.2. Path Visibility

Versions are visible on a path if:

1. They were created on that path, OR
2. They were created on a parent path before the branch point


main
master
development
V1 (visible to all paths)
V2 (visible to master & dev)
V4 (visible to master only)
V3 (visible to dev only)

8.8. Common STAMP Coordinate Patterns

8.8.1. Development Work

```
// Active and inactive, latest on development
StampCoordinateRecord devLatest = Coordinates.Stamp.DevelopmentLatest();

// Active only, latest on development
StampCoordinateRecord devActive = Coordinates.Stamp.DevelopmentLatestActiveOnly();
```

8.8.2. Production Queries

```
// Active only, latest on master (production)
StampCoordinateRecord production = Coordinates.Stamp.MasterLatestActiveOnly();
```

8.8.3. Historical Analysis

```
// View as of specific date
long historicDate = Instant.parse("2023-01-01T00:00:00Z").toEpochMilli();
StampCoordinateRecord historic = StampCoordinateRecord.make(
    StateSet.ACTIVE_AND_INACTIVE,  // See everything
    StampPositionRecord.make(historicDate, TinkarTerm.MASTER_PATH),
    IntIds.set.empty()
);
```

8.8.4. Module-Specific View

```
// Only SNOMED CT core content
```

```
StampCoordinateRecord snomedOnly = StampCoordinateRecord.make(
    StateSet.ACTIVE,
    StampPositionRecord.make(Long.MAX_VALUE, TinkarTerm.MASTER_PATH),
    IntIds.set.of(TinkarTerm.SNOMED_CT_CORE_MODULE.nid())
);
```

8.9. Using STAMP Calculators

8.9.1. Basic Version Resolution

```
StampCoordinateRecord stamp = Coordinates.Stamp.DevelopmentLatestActiveOnly();
StampCalculator calc = StampCalculatorWithCache.getCalculator(stamp);

// Get latest version
ConceptEntity concept = Entity.getConceptForNid(conceptNid);
Latest<ConceptVersion> latest = calc.latest(concept);

if (latest.isPresent()) {
    ConceptVersion version = latest.get();
    System.out.println("Found version: " + version.stampNid());
} else if (latest.isAbsent()) {
    System.out.println("No visible version");
} else if (latest.isContradiction()) {
    System.out.println("Multiple contradictory versions");
}
```

8.9.2. Checking Active Status

```
StampCalculator calc = StampCalculatorWithCache.getCalculator(stamp);

// Quick check if concept has active latest version
boolean isActive = calc.isLatestActive(conceptNid);
```

8.9.3. Relative Position

```
// Determine temporal ordering of two stamps
RelativePosition position = calc.relativePosition(stamp1Nid, stamp2Nid);

switch (position) {
    case BEFORE -> System.out.println("Stamp1 before Stamp2");
    case AFTER -> System.out.println("Stamp1 after Stamp2");
    case EQUAL -> System.out.println("Same stamp");
    case UNREACHABLE -> System.out.println("On unreachable paths");
}
```

8.10. Practical Examples

8.10.1. Example 1: Comparing Versions Across Paths

```
// What's different between development and production?
StampCalculator devCalc = StampCalculatorWithCache.getCalculator(
    Coordinates.Stamp.DevelopmentLatestActiveOnly()
);
StampCalculator prodCalc = StampCalculatorWithCache.getCalculator(
    Coordinates.Stamp.MasterLatestActiveOnly()
);

ConceptEntity concept = Entity.getConceptForNid(conceptNid);

Latest<ConceptVersion> devVersion = devCalc.latest(concept);
Latest<ConceptVersion> prodVersion = prodCalc.latest(concept);

if (devVersion.isPresent() && prodVersion.isPresent()) {
    if (!devVersion.get().equals(prodVersion.get())) {
        System.out.println("Development differs from production");
    }
}
```

8.10.2. Example 2: Audit Trail

```
// Get all versions with STAMP details
ConceptEntity concept = Entity.getConceptForNid(conceptNid);

System.out.println("Audit trail for concept: " + conceptNid);
for (ConceptVersion version : concept.versions()) {
    StampEntity stamp = Entity.getStamp(version.stampNid());

    System.out.printf("%s | %s | %s | %s%n",
        DateTimeUtil.format(stamp.time()),
        stamp.state(),
        Entity.getConceptForNid(stamp.authorNid()).description(),
        Entity.getConceptForNid(stamp.moduleNid()).description()
    );
}
```

8.10.3. Example 3: Finding Recent Changes

```
// Find concepts changed in last 7 days
long sevenDaysAgo = Instant.now()
    .minus(7, ChronoUnit.DAYS)
    .toEpochMilli();
```

```
StampPositionRecord weekAgo = StampPositionRecord.make(
    sevenDaysAgo,
    TinkarTerm.DEVELOPMENT_PATH
);
StampPositionRecord now = StampPositionRecord.make(
    Long.MAX_VALUE,
    TinkarTerm.DEVELOPMENT_PATH
);

List<Integer> changedConcepts = new ArrayList<>();
for (int conceptNid : allConceptNids) {
    ConceptEntity concept = Entity.getConceptForNid(conceptNid);
    ChangeChronology changes = calc.getChanges(concept, weekAgo, now);

    if (!changes.versionChanges().isEmpty()) {
        changedConcepts.add(conceptNid);
    }
}

System.out.println("Changed " + changedConcepts.size() + " concepts");
```

8.11. Best Practices

8.11.1. DO: Reuse STAMP Coordinates

```
// GOOD: Reuse coordinate across application
public class MyService {
    private static final StampCoordinateRecord STAMP =
        Coordinates.Stamp.MasterLatestActiveOnly();

    private static final StampCalculator CALC =
        StampCalculatorWithCache.getCalculator(STAMP);
}
```

8.11.2. DON'T: Create New Coordinates Repeatedly

```
// BAD: Creates new coordinate and calculator each time
public String getDescription(int nid) {
    StampCoordinateRecord stamp = Coordinates.Stamp.MasterLatestActiveOnly();
    StampCalculator calc = StampCalculatorWithCache.getCalculator(stamp);
    // ... loses cache benefits
}
```

8.11.3. DO: Use Appropriate State Sets

```
// GOOD: Active only for end users
if (isEndUserView) {
    stamp = stamp.withAllowedStates(StateSet.ACTIVE);
}

// GOOD: Both states for administrators
if (isAdminView) {
    stamp = stamp.withAllowedStates(StateSet.ACTIVE_AND_INACTIVE);
}
```

8.11.4. DO: Handle All Latest Cases

```
Latest<ConceptVersion> latest = calc.latest(concept);

if (latest.isPresent()) {
    // Process version
} else if (latest.isAbsent()) {
    // Handle no version (filtered out or doesn't exist)
} else if (latest.isContradiction()) {
    // Handle contradiction (multiple versions same time)
    List<ConceptVersion> contradictions = latest.contradictions();
}
```

8.12. Summary

STAMP coordinates provide precise control over temporal versioning:

- **State** - Filter by lifecycle (active/inactive)
- **Time** - Query at specific points in time or get latest
- **Module** - Include, exclude, or prioritize modules
- **Path** - Work on different development branches

Key points:

- Use `StampCalculatorWithCache` for efficient version resolution
- Reuse coordinates to maximize cache benefits
- Handle all `Latest` result cases (present, absent, contradiction)
- Choose appropriate state sets for your use case

9. Language Coordinates

This document describes the IKE Coordinate System (Integrated Knowledge

> Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

9.1. Overview

Language coordinates specify how to retrieve and display human-readable descriptions for concepts and other entities. They control:

- Which natural language (English, Spanish, etc.)
- Which description types (Fully Qualified Name, Regular Name, Definition)
- Dialect preferences (US English vs. GB English)
- Module preferences for description selection

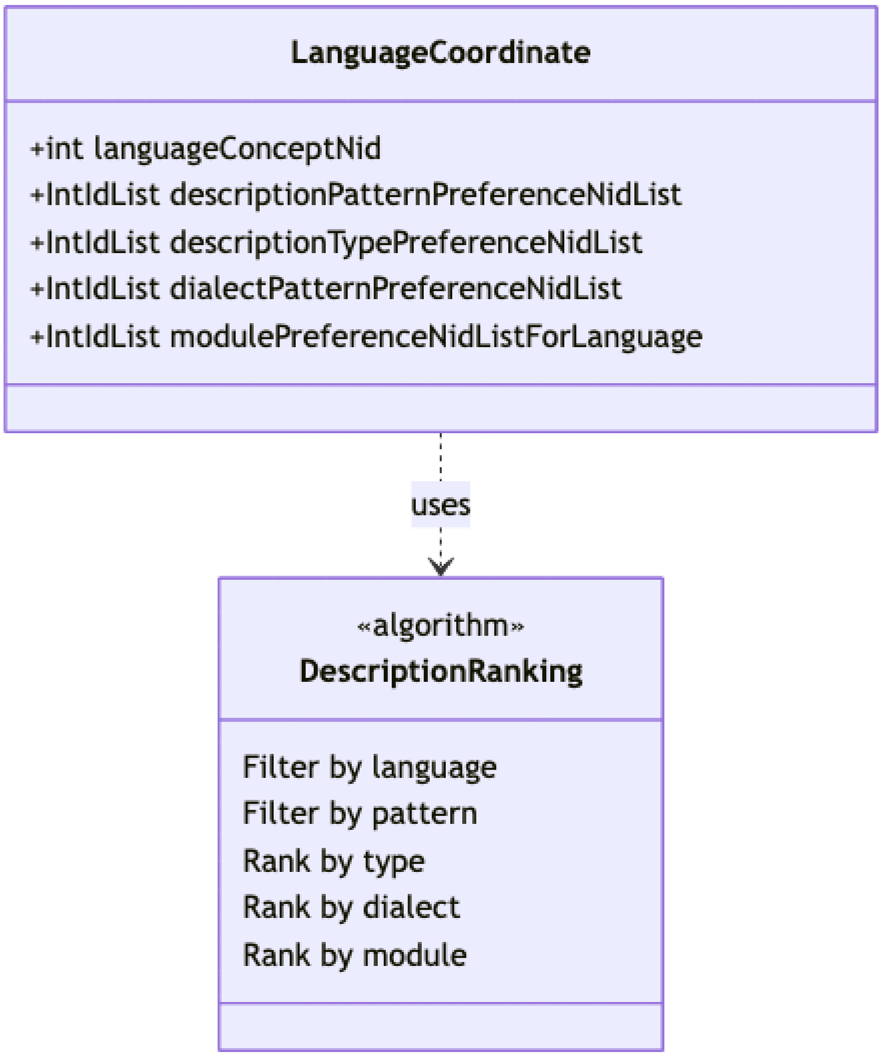
9.2. The Description Selection Problem

A single concept may have many descriptions:

- Multiple languages (English, Spanish, French)
- Multiple types (FQN, Regular Name, Definition, Synonym)
- Multiple dialects (US English: "Color", GB English: "Colour")
- Multiple modules (Core, Extension, Local)

Language coordinates solve this by providing a ranking algorithm.

9.3. Language Coordinate Components


LanguageCoordinate
+int languageConceptNid
+IntIdList descriptionPatternPreferenceNidList
+IntIdList descriptionTypePreferenceNidList
+IntIdList dialectPatternPreferenceNidList
+IntIdList modulePreferenceNidListForLanguage
uses
«algorithm»
DescriptionRanking
Filter by language
Filter by pattern
Rank by type
Rank by dialect
Rank by module

9.3.1. Language

The natural language for descriptions:

```
// English
int langNid = TinkarTerm.ENGLISH_LANGUAGE.nid();

// Spanish
int langNid = TinkarTerm.SPANISH_LANGUAGE.nid();

// Any language (wildcard)
int langNid = TinkarTerm.LANGUAGE.nid();
```

9.3.2. Description Patterns

Patterns that define description structure (usually DESCRIPTION_PATTERN):

```
IntIdList patterns = IntIds.list.of(
    TinkarTerm.DESCRIPTION_PATTERN.nid()
);
```

9.3.3. Description Types

Preferred types in priority order:

```
// Prefer regular names, fall back to FQN
IntIdList types = IntIds.list.of(
    TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
    TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
);

// Prefer FQN, fall back to regular name
IntIdList types = IntIds.list.of(
    TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid(),
    TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid()
);

// Definitions only
IntIdList types = IntIds.list.of(
    TinkarTerm.DEFINITION_DESCRIPTION_TYPE.nid()
);
```

9.3.4. Dialect Preferences

Language variants in priority order:

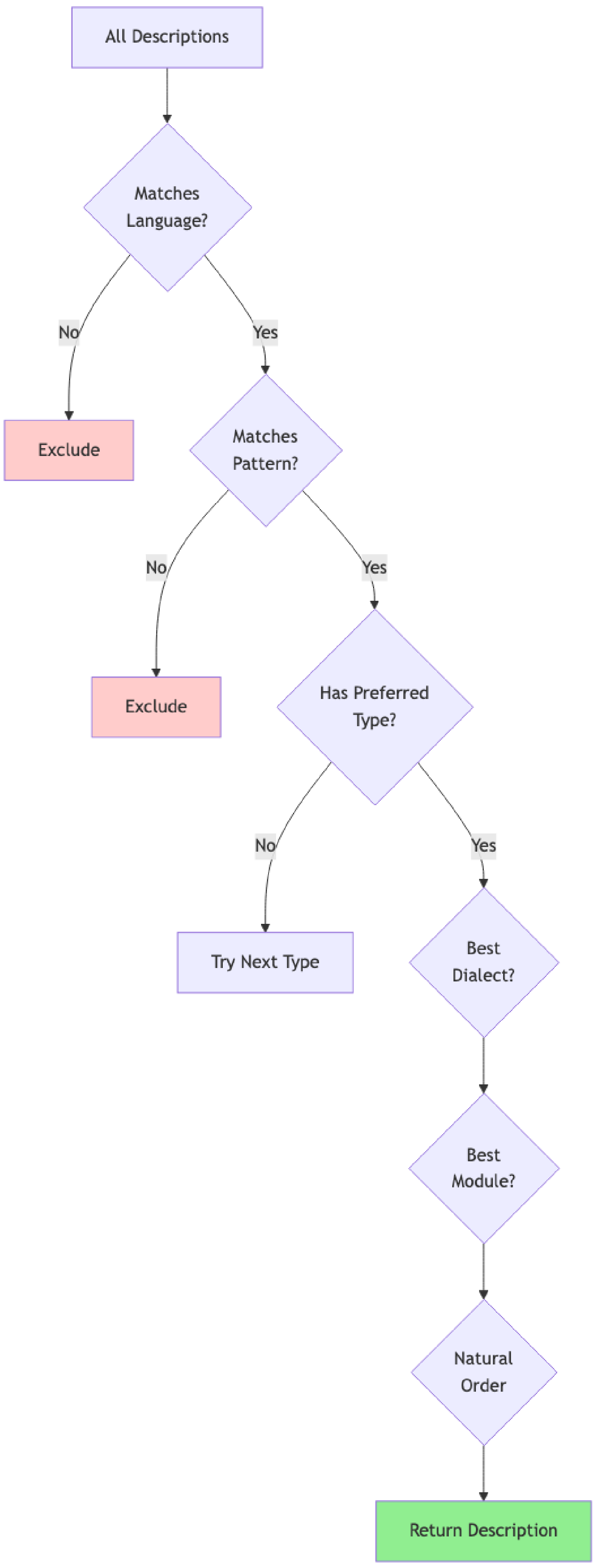
```
// Prefer US English, fall back to GB English
IntIdList dialects = IntIds.list.of(
    TinkarTerm.US_DIALECT_PATTERN.nid(),
    TinkarTerm.GB_DIALECT_PATTERN.nid()
);

// Prefer GB English
IntIdList dialects = IntIds.list.of(
    TinkarTerm.GB_DIALECT_PATTERN.nid(),
    TinkarTerm.US_DIALECT_PATTERN.nid()
);
```

9.3.5. Module Preferences

When multiple descriptions match all criteria:

```
// Prefer overlay module, fall back to core
IntIdList modules = IntIds.list.of(
    TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),
    TinkarTerm.SOLOR_MODULE.nid()
);
```

9.4. Description Ranking Algorithm


All Descriptions
Matches Language?
No
Yes
Exclude
Matches Pattern?
No
Yes
Exclude
Has Preferred Type?
No
Yes
Try Next Type
Best Dialect?
Best Module?
Natural Order
Return Description

9.5. Predefined Language Coordinates

9.5.1. English Variants

```
// US English, regular names preferred
LanguageCoordinateRecord usEnglish =
    Coordinates.Language.UsEnglishRegularName();

// US English, FQNs preferred
LanguageCoordinateRecord usEnglishFqn =
    Coordinates.Language.UsEnglishFullyQualifiedName();

// GB English, regular names
LanguageCoordinateRecord gbEnglish =
    Coordinates.Language.GbEnglishPreferredName();

// GB English, FQNs
LanguageCoordinateRecord gbEnglishFqn =
    Coordinates.Language.GbEnglishFullyQualifiedName();
```

9.5.2. Spanish

```
// Spanish, regular names
LanguageCoordinateRecord spanish =
    Coordinates.Language.SpanishPreferredName();

// Spanish, FQNs
LanguageCoordinateRecord spanishFqn =
    Coordinates.Language.SpanishFullyQualifiedName();
```

9.5.3. Language-Agnostic (Fallback)

```
// Any language, regular names
LanguageCoordinateRecord anyLang =
    Coordinates.Language.AnyLanguageRegularName();

// Any language, FQNs
LanguageCoordinateRecord anyLangFqn =
    Coordinates.Language.AnyLanguageFullyQualifiedName();

// Any language, definitions
LanguageCoordinateRecord anyDef =
    Coordinates.Language.AnyLanguageDefinition();
```

9.6. Custom Language Coordinates

```
// Custom: German, prefer definitions
LanguageCoordinateRecord germanDef = LanguageCoordinateRecord.make(
    TinkarTerm.GERMAN_LANGUAGE.nid(),
    IntIds.list.of(TinkarTerm.DESCRIPTION_PATTERN.nid()),
    IntIds.list.of(
        TinkarTerm.DEFINITION_DESCRIPTION_TYPE.nid(),
        TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
    ),
    IntIds.list.empty(),  // No dialect preferences
    IntIds.list.empty()   // No module preferences
);
```

9.7. Cascading Language Coordinates

ViewCoordinates support multiple language coordinates in priority order:

```
ViewCoordinateRecord multilingualView = ViewCoordinateRecord.make(
    stampCoord,
    Lists.immutable.of(
        Coordinates.Language.SpanishPreferredName(),      // Try Spanish first
        Coordinates.Language.UsEnglishRegularName(),      // Then English
        Coordinates.Language.AnyLanguageRegularName()     // Then any language
    ),
    logicCoord,
    navCoord,
    editCoord
);

ViewCalculator calc = ViewCalculatorWithCache.getCalculator(multilingualView);

// Will return Spanish if available, else English, else any language
String description = calc.getDescriptionText(conceptNid);
```

9.8. Using Language Calculators

9.8.1. Getting Descriptions

```
StampCoordinateRecord stamp = Coordinates.Stamp.DevelopmentLatestActiveOnly();
LanguageCoordinateRecord lang = Coordinates.Language.UsEnglishRegularName();

LanguageCalculator calc = LanguageCalculatorWithCache.getCalculator(
    stamp,
    Lists.immutable.of(lang)
);
```

```
// Get preferred text
String text = calc.getDescriptionText(conceptNid);

// Get FQN specifically
Optional<String> fqn = calc.getFullyQualifiedName(conceptNid);

// Get with fallback
String text = calc.getDescriptionTextOrDefault(conceptNid, "Unknown");
```

9.8.2. Getting All Descriptions

```
// Get all descriptions in preference order
ImmutableList<SemanticEntityVersion> descriptions =
    calc.getDescriptionsForComponent(conceptNid);

for (SemanticEntityVersion desc : descriptions) {
    String text = (String) desc.fieldValues().get(0);
    int typeNid = (Integer) desc.fieldValues().get(1);
    System.out.println(text + " (" +
        Entity.getConceptForNid(typeNid).description() + ")");
}
```

9.9. Practical Examples

9.9.1. Example 1: Multilingual Application

```
public class MultilingualService {
    private final Map<Locale, LanguageCoordinateRecord> langMap = Map.of(
        Locale.US, Coordinates.Language.UsEnglishRegularName(),
        Locale.UK, Coordinates.Language.GbEnglishPreferredName(),
        new Locale("es"), Coordinates.Language.SpanishPreferredName()
    );

    public String getDescription(int conceptNid, Locale locale) {
        LanguageCoordinateRecord lang = langMap.getOrDefault(
            locale,
            Coordinates.Language.UsEnglishRegularName()
        );

        LanguageCalculator calc = LanguageCalculatorWithCache.getCalculator(
            Coordinates.Stamp.MasterLatestActiveOnly(),
            Lists.immutable.of(lang)
        );

        return calc.getDescriptionText(conceptNid);
    }
}
```

9.9.2. Example 2: Description Type Comparison

```
// Get both regular name and FQN for comparison
LanguageCalculator calc = LanguageCalculatorWithCache.getCalculator(
    stamp,
    Lists.immutable.of(Coordinates.Language.UsEnglishRegularName())
);

Latest<SemanticEntityVersion> regularName = calc.getDescription(
    conceptNid,
    TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE
);

Latest<SemanticEntityVersion> fqn = calc.getDescription(
    conceptNid,
    TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE
);

if (regularName.isPresent() && fqn.isPresent()) {
    String regular = (String) regularName.get().fieldValues().get(0);
    String qualified = (String) fqn.get().fieldValues().get(0);

    if (!regular.equals(qualified)) {
        System.out.println("Regular: " + regular);
        System.out.println("FQN: " + qualified);
    }
}
```

9.9.3. Example 3: Finding Missing Translations

```
// Find concepts without Spanish descriptions
LanguageCalculator englishCalc = LanguageCalculatorWithCache.getCalculator(
    stamp,
    Lists.immutable.of(Coordinates.Language.UsEnglishRegularName())
);

LanguageCalculator spanishCalc = LanguageCalculatorWithCache.getCalculator(
    stamp,
    Lists.immutable.of(Coordinates.Language.SpanishPreferredName())
);

List<Integer> needsTranslation = new ArrayList<>();

for (int conceptNid : conceptsToCheck) {
    String english = englishCalc.getDescriptionText(conceptNid);
    String spanish = spanishCalc.getDescriptionText(conceptNid);
    // If Spanish returned English text, translation missing
    if (english.equals(spanish)) {
        needsTranslation.add(conceptNid);
    }
}

System.out.println(needsTranslation.size() + " concepts need Spanish translation");
```

9.10. Best Practices

9.10.1. DO: Use Cascading for Fallback

```
// GOOD: Graceful fallback
Lists.immutable.of(
    Coordinates.Language.SpanishPreferredName(),       // Primary
    Coordinates.Language.UsEnglishRegularName(),       // Fallback
    Coordinates.Language.AnyLanguageRegularName()      // Last resort
)
```

9.10.2. DO: Match Description Types to Use Case

```
// GOOD: FQN for technical interfaces
LanguageCoordinateRecord technical =
    Coordinates.Language.UsEnglishFullyQualifiedName();

// GOOD: Regular names for end users
LanguageCoordinateRecord endUser =
    Coordinates.Language.UsEnglishRegularName();

// GOOD: Definitions for help text
LanguageCoordinateRecord help =
    Coordinates.Language.AnyLanguageDefinition();
```

9.10.3. DON’T: Assume Descriptions Exist

```
// BAD: Assumes description exists
String text = calc.getDescriptionText(conceptNid);
// Could return empty string or concept NID

// GOOD: Provide fallback
String text = calc.getDescriptionTextOrDefault(
    conceptNid,
    "Concept " + conceptNid
);
```

9.11. Summary

Language coordinates control description retrieval through:

- **Language** - Natural language preference
- **Description Types** - FQN, Regular Name, Definition
- **Dialects** - Regional variants
- **Modules** - Module priorities

Key points:

- Use cascading coordinates for graceful fallback
- Match description types to your use case
- Provide defaults when descriptions might be missing
- Reuse calculators for cache efficiency

10. Logic Coordinates

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

10.1. Overview

Logic coordinates configure description logic reasoning and axiom processing. They specify:

- Which classifier to use (Snorocket, ELK, etc.)
- Which description logic profile (EL++, ALC, SROIQ)
- Where to find stated and inferred axioms
- How axioms generate navigation patterns

10.2. What is Description Logic?

Description Logic (DL) is a family of formal knowledge representation languages:

- Express concept definitions using logical operators
- Support automated reasoning and classification
- Enable subsumption testing (is-a relationships)
- Generate inferred taxonomies from stated axioms

Example concept definition in description logic:

```
Pneumonia ▯ Disease ▯ ▯finding_site.Lung ▯ ▯morphology.Inflammation
```

Translation: "Pneumonia is defined as a disease with finding site in the lung and morphology of inflammation."

10.3. Logic Coordinate Components

```
public interface LogicCoordinate {
    int classifierNid();              // Reasoning engine
    int descriptionLogicProfileNid(); // Logic expressivity
    int statedAxiomsPatternNid();     // Where stated axioms stored
    int inferredAxiomsPatternNid();   // Where inferred axioms stored
    int conceptMemberPatternNid();    // Which concepts to classify
    int statedNavigationPatternNid(); // Stated hierarchy pattern
    int inferredNavigationPatternNid();// Inferred hierarchy pattern
    int rootNid();                    // Root of taxonomy
}
```

10.4. The Standard: EL++ Logic

The most common configuration uses EL++ (a decidable DL profile):

```
LogicCoordinateRecord elPlusPlus = Coordinates.Logic.ElPlusPlus();

// Equivalent to:
LogicCoordinateRecord elPlusPlus = LogicCoordinateRecord.make(
    TinkarTerm.SNOROCKET_CLASSIFIER,              // Classifier
    TinkarTerm.EL_PLUS_PLUS_PROFILE,             // Profile
    TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN,
    TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN,
    TinkarTerm.SOLOR_CONCEPT_PATTERN,            // Concept membership
    TinkarTerm.STATED_NAVIGATION_PATTERN,
    TinkarTerm.INFERRED_NAVIGATION_PATTERN,
    TinkarTerm.NAVIGATION_VERTEX                 // Root
);
```

10.5. Classifiers

Table 2. Common Classifiers

Classifier	Profile Support	Use Case
Snorocket	EL++	Medical terminologies (fast, efficient)
ELK	EL++	Large terminologies (incremental classification)

Classifier	Profile Support	Use Case
HermiT	SROIQ (OWL 2 DL)	Complex ontologies (complete but slower)

10.6. Description Logic Profiles

Table 3. Profile Comparison

Profile	Expressivity	Classification Complexity
EL++	Existential restrictions, conjunctions, role hierarchies	Polynomial (fast)
ALC	Adds negation, disjunction, universal restrictions	ExpTime
SROIQ	Adds role composition, nominals, qualified cardinality	NExpTime

Use EL++ for medical terminologies - it provides the right balance of expressivity and performance.

10.7. Axiom Patterns

10.7.1. Stated vs. Inferred Axioms

Type	Source	Use Case
Stated	Author-asserted	Authoring, editing, quality assurance
Inferred	Classifier-generated	Queries, navigation, subsumption testing

10.7.2. Axiom Storage

Axioms are stored as DiTree graphs in semantic entities:

```
// Get stated axioms
LogicCoordinate logic = Coordinates.Logic.ElPlusPlus();
Latest<DiTreeEntity> statedAxioms = logic.getStatedAxiomsVersion(
    conceptNid,
    stampCoord
);

if (statedAxioms.isPresent()) {
    DiTreeEntity axiomTree = statedAxioms.get();
    // Process axiom tree
    processAxiomTree(axiomTree);
}
```

10.8. Premise Types

```
public enum PremiseType {
    STATED,   // Author-asserted axioms
    INFERRED  // Classifier-generated axioms
}

// Use premise type to select axioms
Latest<DiTreeEntity> axioms = logic.getAxiomsVersion(
    conceptNid,
    PremiseType.STATED,  // or INFERRED
    stampCoord
);
```

10.9. Using Logic Calculators

10.9.1. Getting Axioms

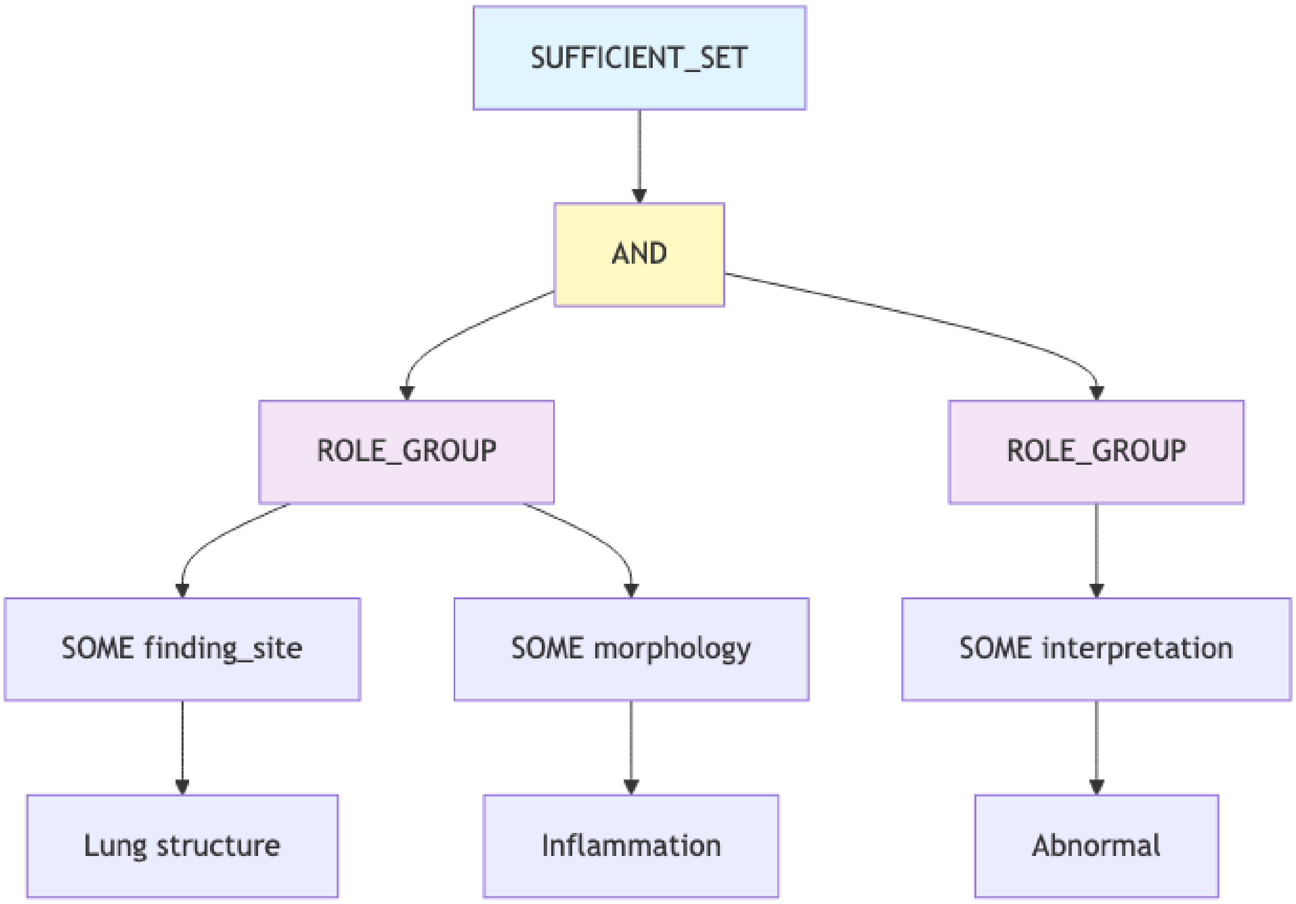
```
StampCoordinateRecord stamp = Coordinates.Stamp.DevelopmentLatestActiveOnly();
LogicCoordinateRecord logic = Coordinates.Logic.ElPlusPlus();

LogicCalculator calc = LogicCalculatorWithCache.getCalculator(stamp, logic);

// Get stated axioms
Latest<DiTreeEntity> statedAxioms = calc.getStatedAxiomTree(conceptNid);

// Get inferred axioms
Latest<DiTreeEntity> inferredAxioms = calc.getInferredAxiomTree(conceptNid);

// Get axioms by premise type
Latest<DiTreeEntity> axioms = calc.getAxiomTreeForEntity(
    conceptNid,
    PremiseType.STATED
);
```

10.9.2. Checking Definitions

```
// Check if concept has sufficient definition
Latest<DiTreeEntity> statedAxioms = calc.getStatedAxiomTree(conceptNid);

if (statedAxioms.isPresent()) {
    DiTreeEntity tree = statedAxioms.get();
    boolean isSufficientlyDefined =
        tree.containsVertexWithMeaning(TinkarTerm.SUFFICIENT_SET);

    if (isSufficientlyDefined) {
        System.out.println("Concept has sufficient definition");
    }
}
```

10.10. Axiom Tree Structure

Axioms are encoded as DiTree graphs:



10.11. Practical Examples

10.11.1. Example 1: Extracting Relationships from Axioms

```
public List<Relationship> extractRelationships(int conceptNid) {
    LogicCalculator calc = LogicCalculatorWithCache.getCalculator(
        stamp, logic
    );

    Latest<DiTreeEntity> axioms = calc.getStatedAxiomTree(conceptNid);
    if (!axioms.isPresent()) {
        return List.of();
    }

    List<Relationship> relationships = new ArrayList<>();
    DiTreeEntity tree = axioms.get();
    // Process tree to extract SOME restrictions
    processVertex(tree, tree.root().vertexIndex(), relationships);

    return relationships;
}

private void processVertex(DiTreeEntity tree, int vertexIndex,
                          List<Relationship> relationships) {
    EntityVertex vertex = tree.vertex(vertexIndex);

    if (vertex.getMeaningNid() == TinkarTerm.SOME.nid()) {
        // Existential restriction: SOME role.filler
        IntList successors = tree.successors(vertexIndex).toList();
        int roleNid = tree.vertex(successors.get(0)).getMeaningNid();
        int fillerNid = tree.vertex(successors.get(1)).getMeaningNid();

        relationships.add(new Relationship(roleNid, fillerNid));
    }

    // Recurse to children
    tree.successors(vertexIndex).forEach(childIndex ->
        processVertex(tree, childIndex, relationships)
    );
}
```

10.11.2. Example 2: Comparing Stated vs. Inferred

```
// Compare stated and inferred axioms
LogicCalculator calc = LogicCalculatorWithCache.getCalculator(stamp, logic);

Latest<DiTreeEntity> stated = calc.getStatedAxiomTree(conceptNid);
Latest<DiTreeEntity> inferred = calc.getInferredAxiomTree(conceptNid);

if (stated.isPresent() && inferred.isPresent()) {
    if (stated.get().equals(inferred.get())) {
        System.out.println("Stated and inferred match");
    } else {
        System.out.println("Classifier inferred additional relationships");
        // Could compare trees in detail
    }
}
```

10.11.3. Example 3: Finding Defined Concepts

```
// Find all sufficiently defined concepts
List<Integer> definedConcepts = new ArrayList<>();
```

```
for (int conceptNid : allConceptNids) {
    Latest<DiTreeEntity> axioms = calc.getStatedAxiomTree(conceptNid);

    if (axioms.isPresent()) {
        DiTreeEntity tree = axioms.get();
        if (tree.containsVertexWithMeaning(TinkarTerm.SUFFICIENT_SET)) {
            definedConcepts.add(conceptNid);
        }
    }
}

System.out.println("Found " + definedConcepts.size() +
    " sufficiently defined concepts");
```

10.12. Integration with Navigation

Logic coordinates generate navigation patterns:

```
// Stated navigation from stated axioms
int statedNavPattern = logic.statedNavigationPatternNid();

// Inferred navigation from classification
int inferredNavPattern = logic.inferredNavigationPatternNid();

// Use in navigation coordinate
NavigationCoordinateRecord nav = NavigationCoordinateRecord.make(
    IntIds.set.of(inferredNavPattern),  // Use inferred hierarchy
    StateSet.ACTIVE,
    IntIds.list.empty(),
    true
);
```

10.13. Best Practices

10.13.1. DO: Use EL++ for Medical Terminologies

```
// GOOD: Standard EL++ configuration
LogicCoordinateRecord logic = Coordinates.Logic.ElPlusPlus();
```

10.13.2. DO: Check for Axiom Presence

```
// GOOD: Handle absent axioms
Latest<DiTreeEntity> axioms = calc.getStatedAxiomTree(conceptNid);

if (axioms.isPresent()) {
    // Process axioms
} else {
    // Handle primitive concept (no stated axioms)
}
```

10.13.3. DO: Use Appropriate Premise Type

```
// GOOD: Stated for authoring
if (isAuthoringContext) {
    axioms = calc.getStatedAxiomTree(conceptNid);
}

// GOOD: Inferred for queries
if (isQueryContext) {
    axioms = calc.getInferredAxiomTree(conceptNid);
}
```

10.13.4. DON'T: Mix Premise Types Inconsistently

```
// BAD: Inconsistent premise types
Latest<DiTreeEntity> axioms1 = calc.getStatedAxiomTree(concept1Nid);
Latest<DiTreeEntity> axioms2 = calc.getInferredAxiomTree(concept2Nid);
// Comparing apples to oranges
```

10.14. Summary

Logic coordinates configure description logic reasoning:

- **Classifier** - Reasoning engine (Snorocket, ELK, HermiT)
- **Profile** - Logic expressivity (EL++, ALC, SROIQ)
- **Axiom Patterns** - Where to find stated and inferred definitions
- **Navigation Patterns** - How axioms generate hierarchies

Key points:

- Use EL++ for medical terminologies (fast and sufficient)
- Stated axioms for authoring, inferred for queries
- Check for axiom presence before processing
- Logic coordinates feed into navigation coordinates

11. Navigation Coordinates

This document describes the IKE Coordinate System (Integrated Knowledge

> Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

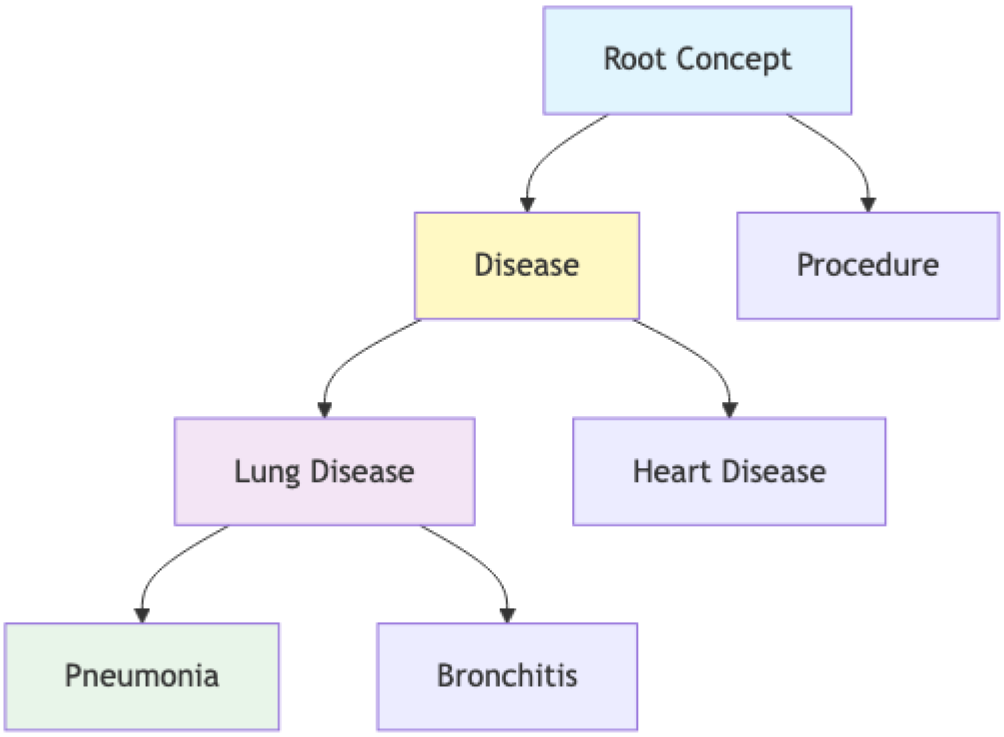
11.1. Overview

Navigation coordinates specify how to construct and traverse directed graphs over concepts. They control:

- Which navigation patterns to use (inferred, stated, custom)
- Which vertex states to include (active, inactive, both)
- Whether and how to sort children
- Custom sorting criteria

11.2. What is a Navigation Graph?

A navigation graph (digraph) represents parent-child relationships:



- **Vertices** = Concepts
- **Edges** = Parent-child relationships
- **Multiple patterns** can be combined

11.3. Navigation Coordinate Components

```
public interface NavigationCoordinate {
    IntIdSet navigationPatternNids();     // Which graphs to use
    StateSet vertexStates();              // Active/inactive filter
    IntIdList verticesSortPatternNidList();// Custom sort patterns
    boolean sortVertices();               // Enable sorting?
}
```

11.4. Navigation Patterns

11.4.1. Inferred Navigation

Based on classifier-generated taxonomy:

```
NavigationCoordinate inferred = Coordinates.Navigation.inferred();

// Equivalent to:
NavigationCoordinateRecord inferred = NavigationCoordinateRecord.make(
    IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
    StateSet.ACTIVE,
    IntIds.list.empty(),
    true  // Sort enabled
);
```

Use Cases:

- End-user browsing
- Semantic queries
- Subsumption testing
- Complete taxonomy with inferences

11.4.2. Stated Navigation

Based on author-asserted relationships:

```
NavigationCoordinate stated = Coordinates.Navigation.stated();

// Equivalent to:
NavigationCoordinateRecord stated = NavigationCoordinateRecord.make(
    IntIds.set.of(TinkarTerm.STATED_NAVIGATION.nid()),
    StateSet.ACTIVE,
    IntIds.list.empty(),
    true
);
```

Use Cases:

- Authoring workflows
- Quality assurance

- Viewing editorial structure
- Pre-classification browsing

11.4.3. Combining Multiple Patterns

```
// Combine inferred is-a with part-of relationships
NavigationCoordinateRecord combined = NavigationCoordinateRecord.make(
    IntIds.set.of(
        TinkarTerm.INFERRED_NAVIGATION.nid(),
        TinkarTerm.PART_OF_NAVIGATION.nid()
    ),
    StateSet.ACTIVE,
    IntIds.list.empty(),
    true
);

// Graph will include edges from both patterns
```

11.5. Vertex State Filtering

```
// Active only (most common)
StateSet activeOnly = StateSet.ACTIVE;

// Include inactive (for administration)
StateSet all = StateSet.ACTIVE_AND_INACTIVE;

// Inactive only (for retirement analysis)
StateSet inactiveOnly = StateSet.INACTIVE;

// Use in coordinate
NavigationCoordinateRecord nav = inferred.withVertexStates(activeOnly);
```

11.6. Vertex Sorting

11.6.1. Enabling/Disabling Sort

```
// Sorted alphabetically (default)
NavigationCoordinateRecord sorted = NavigationCoordinateRecord.makeInferred();
// sortVertices() returns true

// Unsorted (faster)
NavigationCoordinateRecord unsorted = NavigationCoordinateRecord.make(
    IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
    StateSet.ACTIVE,
    IntIds.list.empty(),
    false  // No sorting
);
```

11.6.2. Custom Sort Patterns

```
// Sort by custom patterns, then alphabetically
NavigationCoordinateRecord customSort = NavigationCoordinateRecord.make(
    IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
    StateSet.ACTIVE,
    IntIds.list.of(
        TinkarTerm.SEVERITY_SORT_PATTERN.nid(),  // Sort by severity first
        TinkarTerm.ALPHABETICAL_SORT_PATTERN.nid() // Then alphabetically
    ),
    true
);
```

11.7. Using Navigation Calculators

11.7.1. Basic Graph Operations

```
StampCoordinateRecord stamp = Coordinates.Stamp.DevelopmentLatestActiveOnly();
NavigationCoordinateRecord nav = Coordinates.Navigation.inferred();

NavigationCalculator calc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    nav
);

// Get immediate parents
IntIdSet parents = calc.parentsOf(conceptNid);

// Get immediate children (sorted if coordinate specifies)
IntIdSet children = calc.childrenOf(conceptNid);

// Check if child
boolean isChild = calc.isChildOf(childNid, parentNid);
```

11.7.2. Transitive Operations

```
// Get all ancestors (parents, grandparents, etc.)
IntIdSet ancestors = calc.ancestorsOf(conceptNid);

// Get all descendants (children, grandchildren, etc.)
IntIdSet descendants = calc.descendantsOf(conceptNid);

// Test subsumption
```

```
boolean subsumes = calc.isDescendentOf(specificNid, generalNid);
// True if specific is a descendant of general
```

11.7.3. Special Nodes

```
// Get root concepts (no parents)
IntIdSet roots = calc.roots();

// Get leaf concepts (no children)
IntIdSet leaves = calc.leaves();
```

11.8. Practical Examples

11.8.1. Example 1: Building a Tree View

```
public TreeNode buildTree(int rootNid, NavigationCalculator calc, int depth) {
    // Get description for display
    String name = langCalc.getDescriptionText(rootNid);
    TreeNode node = new TreeNode(rootNid, name);

    if (depth > 0) {
        // Get children (sorted by navigation coordinate)
        IntIdSet children = calc.childrenOf(rootNid);

        children.forEach(childNid -> {
            TreeNode childNode = buildTree(childNid, calc, depth - 1);
            node.addChild(childNode);
        });
    }

    return node;
}

// Use
NavigationCalculator calc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    Coordinates.Navigation.inferred()
);

IntIdSet roots = calc.roots();
roots.forEach(rootNid -> {
    TreeNode tree = buildTree(rootNid, calc, 3);  // 3 levels deep
    displayTree(tree);
});
```

11.8.2. Example 2: Finding Common Ancestors

```
// Find common ancestors of two concepts
public IntIdSet findCommonAncestors(int concept1Nid, int concept2Nid,
                                    NavigationCalculator calc) {
    IntIdSet ancestors1 = calc.ancestorsOf(concept1Nid);
    IntIdSet ancestors2 = calc.ancestorsOf(concept2Nid);

    // Intersection = common ancestors
    return ancestors1.intersect(ancestors2);
}

// Find most specific common ancestor
public int findMostSpecificCommonAncestor(int concept1Nid, int concept2Nid,
                                         NavigationCalculator calc) {
    IntIdSet commonAncestors = findCommonAncestors(
        concept1Nid, concept2Nid, calc
    );

    // Most specific = has no descendants in the common set
    for (int ancestorNid : commonAncestors.toArray()) {
        IntIdSet ancestorDescendants = calc.descendantsOf(ancestorNid);
        IntIdSet overlap = ancestorDescendants.intersect(commonAncestors);

        if (overlap.isEmpty()) {
            return ancestorNid;  // No descendants, most specific
        }
    }

    return -1;  // No common ancestor found
}
```

11.8.3. Example 3: Breadcrumb Navigation

```
// Build breadcrumb path from concept to root
public List<String> buildBreadcrumb(int conceptNid,
                                    NavigationCalculator navCalc,
                                    LanguageCalculator langCalc) {
    List<String> breadcrumb = new ArrayList<>();

    int current = conceptNid;
    while (current != -1) {
        // Add current to breadcrumb
        String name = langCalc.getDescriptionText(current);
        breadcrumb.add(0, name);  // Add to front

        // Move to parent (take first parent if multiple)
        IntIdSet parents = navCalc.parentsOf(current);
        if (parents.isEmpty()) {
```

```java
            break;
        }
        current = parents.iterator().next();
    }

    return breadcrumb;
}

// Use
List<String> breadcrumb = buildBreadcrumb(pneumoniaNid, navCalc, langCalc);
// Result: ["Root", "Disease", "Lung Disease", "Pneumonia"]
```

11.8.4. Example 4: Comparing Stated vs. Inferred Hierarchies

```java
// Find differences between stated and inferred parents
NavigationCalculator statedCalc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    Coordinates.Navigation.stated()
);

NavigationCalculator inferredCalc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    Coordinates.Navigation.inferred()
);

IntIdSet statedParents = statedCalc.parentsOf(conceptNid);
IntIdSet inferredParents = inferredCalc.parentsOf(conceptNid);

// Inferred but not stated = classifier added
IntIdSet added = inferredParents.difference(statedParents);

// Stated but not inferred = redundant
IntIdSet redundant = statedParents.difference(inferredParents);

if (!added.isEmpty()) {
    System.out.println("Classifier inferred additional parents:");
    added.forEach(nid ->
        System.out.println("  " + langCalc.getDescriptionText(nid))
    );
}

if (!redundant.isEmpty()) {
    System.out.println("Redundant stated relationships:");
    redundant.forEach(nid ->
        System.out.println("  " + langCalc.getDescriptionText(nid))
    );
}
```

11.9. Sorting Behavior

When sorting is enabled:

1. Apply custom sort patterns in order
2. Fall back to natural alphabetical order
3. Use language coordinate for description text

```
// Example: Children sorted alphabetically
NavigationCalculator calc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    Coordinates.Navigation.inferred()  // Sorting enabled
);

IntIdSet children = calc.childrenOf(diseaseNid);
// Returns: [Bronchitis, Pneumonia, ...]  (alphabetical)

// Without sorting
NavigationCalculator unsortedCalc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    NavigationCoordinateRecord.make(
        IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
        StateSet.ACTIVE,
        IntIds.list.empty(),
        false  // No sorting
    )
);

IntIdSet unsortedChildren = unsortedCalc.childrenOf(diseaseNid);
// Returns: [Pneumonia, Bronchitis, ...]  (arbitrary order, faster)
```

11.10. Best Practices

11.10.1. DO: Use Inferred Navigation for Queries

```
// GOOD: Inferred for semantic queries
NavigationCoordinate nav = Coordinates.Navigation.inferred();

if (calc.isDescendentOf(conceptNid, TinkarTerm.DISEASE.nid())) {
    System.out.println("Is a disease");
}
```

11.10.2. DO: Use Stated Navigation for Authoring

```
// GOOD: Stated for authoring views
```

```
if (isAuthoringContext) {
    NavigationCoordinate nav = Coordinates.Navigation.stated();
}
```

11.10.3. DO: Disable Sorting for Performance

```
// GOOD: Disable sorting when order doesn't matter
NavigationCoordinateRecord unsorted = NavigationCoordinateRecord.make(
    IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
    StateSet.ACTIVE,
    IntIds.list.empty(),
    false  // Faster when order doesn't matter
);
```

11.10.4. DO: Filter by Vertex State Appropriately

```
// GOOD: Active only for end users
if (isEndUserView) {
    nav = nav.withVertexStates(StateSet.ACTIVE);
}

// GOOD: Include inactive for administration
if (isAdminView) {
    nav = nav.withVertexStates(StateSet.ACTIVE_AND_INACTIVE);
}
```

11.10.5. DON'T: Mix Navigation Types Inconsistently

```
// BAD: Mixing stated and inferred
NavigationCalculator statedCalc = ...;
NavigationCalculator inferredCalc = ...;

IntIdSet statedParents = statedCalc.parentsOf(conceptNid);
IntIdSet inferredChildren = inferredCalc.childrenOf(conceptNid);
// Inconsistent hierarchy views
```

11.11. Summary

Navigation coordinates control graph traversal:

- **Navigation Patterns** - Which graphs to use (inferred, stated, custom)
- **Vertex States** - Include active, inactive, or both
- **Vertex Sorting** - Sort children or not
- **Sort Patterns** - Custom sorting criteria

Key points:

- Use inferred navigation for queries and browsing
- Use stated navigation for authoring
- Disable sorting when performance matters
- Filter by vertex state appropriately
- Combine patterns for rich navigation views

12. Edit Coordinates

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

12.1. Overview

Edit coordinates specify metadata for creating and modifying knowledge content. They control:

- Who is making changes (author)
- Where new content goes (default module)
- Where content is moved during modularization (destination module)
- Which path new content is created on (default path)
- Which path content is promoted to (promotion path)

12.2. Edit Coordinate Components

```
public interface EditCoordinate {
    int getAuthorNidForChanges();      // Who
    int getDefaultModuleNid();         // Where (new content)
    int getDestinationModuleNid();     // Where (modularizing)
    int getDefaultPathNid();           // Which path (new content)
    int getPromotionPathNid();         // Which path (promoting)
}
```

12.3. The Three Workflows

Edit coordinates support three distinct workflows:

12.3.1. 1. Developing

Creating new content or modifying existing content:

- New content goes to **default module**
- Created on **default path**
- Attributed to **author**
- Existing content retains its module

12.3.2. 2. Modularizing

Moving content between modules:

- Content moved to **destination module**
- Written to **default path**
- Module changes, other metadata preserved

12.3.3. 3. Promoting

Moving content across development paths:

- Content retains **current module**
- Copy written to **promotion path**
- Supports staged releases (dev → staging → production)

12.4. Standard Edit Coordinate

```
// Default edit coordinate
EditCoordinateRecord editCoord = Coordinates.Edit.Default();

// Equivalent to:
EditCoordinateRecord editCoord = EditCoordinateRecord.make(
    TinkarTerm.USER.nid(),                    // Author
    TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),    // Default module
    TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),    // Destination module
    TinkarTerm.DEVELOPMENT_PATH.nid(),        // Default path
    TinkarTerm.DEVELOPMENT_PATH.nid()         // Promotion path
);
```

12.5. Creating Custom Edit Coordinates

```
// Custom for specific user and module
EditCoordinateRecord customEdit = EditCoordinateRecord.make(
    drSmithNid,                              // Dr. Smith as author
    extensionModuleNid,                      // Extension module for new content
    coreModuleNid,                          // Core module for modularization
    developmentPathNid,                     // Create on development
    masterPathNid                           // Promote to master
);
```

12.6. Using Edit Coordinates

Edit coordinates are typically used when creating new versions:

```
EditCoordinateRecord editCoord = Coordinates.Edit.Default();

// Create STAMP for new version
StampEntity stamp = StampRecord.build(
    State.ACTIVE,
    System.currentTimeMillis(),
    editCoord.getAuthorNidForChanges(),      // From edit coordinate
    editCoord.getDefaultModuleNid(),         // From edit coordinate
    editCoord.getDefaultPathNid()            // From edit coordinate
);

// Create new concept version
ConceptVersionRecord newVersion = ConceptVersionRecord.build(
    conceptNid,
    stamp.nid()
);

// Commit to database
Entity.provider().putEntity(newVersion);
```

12.7. Practical Examples

12.7.1. Example 1: User-Specific Edit Context

```
public class UserEditContext {
    private final Map<String, EditCoordinateRecord> userEdits = new HashMap<>();

    public EditCoordinateRecord getEditCoordinateForUser(String userId) {
        return userEdits.computeIfAbsent(userId, id -> {
            int userNid = getUserNid(id);
            int moduleNid = getUserDefaultModule(id);

            return EditCoordinateRecord.make(
                userNid,
                moduleNid,
                moduleNid,
                TinkarTerm.DEVELOPMENT_PATH.nid(),
                TinkarTerm.DEVELOPMENT_PATH.nid()
            );
        });
    }
}
```

12.7.2. Example 2: Module-Specific Development

```
// Create edit coordinate for extension development
public EditCoordinateRecord createExtensionEditCoordinate(int authorNid) {
    return EditCoordinateRecord.make(
        authorNid,
        TinkarTerm.EXTENSION_MODULE.nid(),      // Extensions go here
        TinkarTerm.CORE_MODULE.nid(),           // Modularize to core
        TinkarTerm.DEVELOPMENT_PATH.nid(),
        TinkarTerm.MASTER_PATH.nid()
    );
}
```

12.7.3. Example 3: Promotion Workflow

```
// Promote content from development to master
public void promoteToMaster(List<Integer> conceptNids,
                           EditCoordinateRecord editCoord) {
    StampCoordinateRecord devStamp = Coordinates.Stamp.DevelopmentLatestActiveOnly();
    StampCalculator devCalc = StampCalculatorWithCache.getCalculator(devStamp);

    for (int conceptNid : conceptNids) {
        ConceptEntity concept = Entity.getConceptForNid(conceptNid);
        Latest<ConceptVersion> devVersion = devCalc.latest(concept);

        if (devVersion.isPresent()) {
            ConceptVersion version = devVersion.get();

            // Create STAMP on promotion path
            StampEntity promoStamp = StampRecord.build(
                version.state(),
                System.currentTimeMillis(),
                editCoord.getAuthorNidForChanges(),
                version.moduleNid(),                    // Keep original module
                editCoord.getPromotionPathNid()         // Promotion path
            );

            // Create version on master path
            ConceptVersionRecord promoVersion = ConceptVersionRecord.build(
                conceptNid,
                promoStamp.nid()
            );

            Entity.provider().putEntity(promoVersion);
        }
    }
}
```

12.8. Integration with View Coordinates

Edit coordinates are part of view coordinates:

```
ViewCoordinateRecord view = ViewCoordinateRecord.make(
    stampCoord,
    languageCoords,
    logicCoord,
    navCoord,
    editCoord  // Edit coordinate included
);

// Access via view
EditCoordinate editFromView = view.editCoordinate();
int authorNid = editFromView.getAuthorNidForChanges();
```

12.9. Best Practices

12.9.1. DO: Use Per-User Edit Coordinates

```
// GOOD: Each user has own edit coordinate
public EditCoordinateRecord getUserEditCoordinate(User user) {
    return EditCoordinateRecord.make(
        user.getConceptNid(),
        user.getDefaultModule(),
        user.getDefaultModule(),
        TinkarTerm.DEVELOPMENT_PATH.nid(),
        TinkarTerm.DEVELOPMENT_PATH.nid()
    );
}
```

12.9.2. DO: Match Module to Context

```
// GOOD: Core development
if (isDevelopingCore) {
    editCoord = EditCoordinateRecord.make(
        authorNid,
        TinkarTerm.CORE_MODULE.nid(),  // Core module
        ...
    );
}

// GOOD: Extension development
```

```
if (isDevelopingExtension) {
    editCoord = EditCoordinateRecord.make(
        authorNid,
        TinkarTerm.EXTENSION_MODULE.nid(),  // Extension module
        ...
    );
}
```

12.9.3. DON’T: Share Edit Coordinates Across Users

```
// BAD: All users share same author
public static final EditCoordinateRecord SHARED_EDIT =
    Coordinates.Edit.Default();

// All changes attributed to same author - can't track who did what
```

12.10. Summary

Edit coordinates control change attribution:

- **Author** - Who is making changes
- **Default Module** - Where new content goes
- **Destination Module** - Target for modularization
- **Default Path** - Where to create new content
- **Promotion Path** - Target for content promotion

Key points:

- Use per-user edit coordinates for proper attribution
- Match module to development context
- Support three workflows: developing, modularizing, promoting
- Integrate with STAMP for complete provenance

13. View Coordinates

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

13.1. Overview

View coordinates unify all five coordinate types into a single, cohesive specification. Instead of managing STAMP, Language, Logic, Navigation, and Edit coordinates separately, a view coordinate provides:

- **Unified Access** - Single point of configuration for all coordinate types
- **Simplified API** - One calculator that delegates to specialized calculators
- **Consistent Context** - All operations use the same coordinated view
- **Easy Switching** - Change entire context by switching view coordinates

```
public interface ViewCoordinate {
    StampCoordinate stampCoordinate();
    List<LanguageCoordinate> languageCoordinateList();
    LogicCoordinate logicCoordinate();
    NavigationCoordinate navigationCoordinate();
    EditCoordinate editCoordinate();
}
```

13.2. The View Calculator

The ViewCalculator provides unified access to all coordinate operations:

```
graph TD
    A[ViewCalculator] --> B[StampCalculator]
    A --> C[LanguageCalculator]
    A --> D[LogicCalculator]
    A --> E[NavigationCalculator]
    A --> F[EditCoordinate]

    B --> G[Version Selection]
    C --> H[Description Selection]
    D --> I[Axiom Retrieval]
    E --> J[Graph Traversal]
    F --> K[Change Attribution]

    style A fill:#e1f5ff,stroke:#0066cc,stroke-width:3px
    style B fill:#fff4e6,stroke:#ff9800
    style C fill:#f3e5f5,stroke:#9c27b0
    style D fill:#e8f5e9,stroke:#4caf50
    style E fill:#fff3e0,stroke:#ff9800
    style F fill:#fce4ec,stroke:#e91e63
```

13.3. Creating View Coordinates

13.3.1. Using Predefined Views

```
// Default development view
ViewCoordinateRecord view = Coordinates.View.DefaultView();
```

```
// What this includes:
// - STAMP: Development path, latest, active only
// - Language: English (US), regular name, FSN fallback
// - Logic: EL++ profile, stated axioms
// - Navigation: Inferred hierarchy, active only, sorted
// - Edit: User as author, SOLOR overlay module

// Get calculator
ViewCalculator calculator = ViewCalculatorWithCache.getCalculator(view);

// Use for all operations
String description = calculator.getDescriptionText(conceptNid);
Latest<ConceptVersion> latest = calculator.latest(conceptNid);
int[] parents = calculator.unsortedParentsOf(conceptNid);
```

13.3.2. Custom View Creation

```
// Create custom view for specific needs
ViewCoordinateRecord customView = ViewCoordinateRecord.make(
    // STAMP: Production path, point-in-time
    StampCoordinateRecord.make(
        StateSet.ACTIVE_ONLY,
        StampPositionRecord.make(
            releaseTimestamp,
            TinkarTerm.MASTER_PATH.nid()
        )
    ),

    // Language: Spanish with English fallback
    List.of(
        LanguageCoordinateRecord.make(
            TinkarTerm.SPANISH_LANGUAGE.nid(),
            TinkarTerm.SPANISH_DIALECT_ASSEMBLAGE.nid()
        ),
        LanguageCoordinateRecord.make(
            TinkarTerm.ENGLISH_LANGUAGE.nid(),
            TinkarTerm.US_DIALECT_ASSEMBLAGE.nid()
        )
    ),

    // Logic: EL++ with inferred axioms
    LogicCoordinateRecord.make(
        TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN.nid(),
        TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN.nid(),
        TinkarTerm.SNOROCKET_CLASSIFIER.nid(),
        TinkarTerm.STATED_PREMISE_TYPE.nid()
    ),
    // Navigation: Stated hierarchy, all states
    NavigationCoordinateRecord.make(
        IntIds.set.of(TinkarTerm.STATED_NAVIGATION.nid()),
        StateSet.ACTIVE_AND_INACTIVE,
        true,
        IntIds.list.empty()
    ),

    // Edit: Specific user and module
    EditCoordinateRecord.make(
        userNid,
        moduleNid,
        moduleNid,
        TinkarTerm.DEVELOPMENT_PATH.nid(),
        TinkarTerm.MASTER_PATH.nid()
    )
);
```

13.4. Common View Patterns

13.4.1. Pattern 1: Development View

For active development work:

```
public ViewCoordinateRecord createDevelopmentView(int authorNid, int moduleNid) {
    return ViewCoordinateRecord.make(
        // Latest on development path
        Coordinates.Stamp.DevelopmentLatestActiveOnly(),

        // English descriptions
        Coordinates.Language.UsEnglishRegularName(),

        // Stated axioms for editing
        Coordinates.Logic.ElPlusPlus(),

        // Inferred navigation for browsing
        Coordinates.Navigation.inferred(),

        // User-specific edit coordinate
        EditCoordinateRecord.make(
            authorNid,
            moduleNid,
            moduleNid,
            TinkarTerm.DEVELOPMENT_PATH.nid(),
            TinkarTerm.DEVELOPMENT_PATH.nid()
        )
    );
}
```

13.4.2. Pattern 2: Production View

For production/release access:

```
public ViewCoordinateRecord createProductionView(long releaseTime) {
    return ViewCoordinateRecord.make(
        // Fixed point in time on master path
        StampCoordinateRecord.make(
            StateSet.ACTIVE_ONLY,
            StampPositionRecord.make(releaseTime, TinkarTerm.MASTER_PATH.nid())
        ),

        // English descriptions
        Coordinates.Language.UsEnglishRegularName(),

        // Inferred axioms for querying
        LogicCoordinateRecord.make(
            TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN.nid(),
            TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN.nid(),
            TinkarTerm.SNOROCKET_CLASSIFIER.nid(),
            TinkarTerm.INFERRED_PREMISE_TYPE.nid()
        ),

        // Inferred navigation
        Coordinates.Navigation.inferred(),

        // Read-only (edit coordinate not used)
        Coordinates.Edit.Default()
    );
}
```

13.4.3. Pattern 3: Administrative View

For maintenance and content review:

```
public ViewCoordinateRecord createAdminView() {
    return ViewCoordinateRecord.make(
        // All states, latest on development
        StampCoordinateRecord.make(
            StateSet.ACTIVE_AND_INACTIVE,  // See inactive content
            StampPositionRecord.make(
                Long.MAX_VALUE,
                TinkarTerm.DEVELOPMENT_PATH.nid()
            )
        ),

        // English descriptions
        Coordinates.Language.UsEnglishRegularName(),
        // Both stated and inferred
        Coordinates.Logic.ElPlusPlus(),

        // Navigation with inactive vertices
        NavigationCoordinateRecord.make(
            IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
            StateSet.ACTIVE_AND_INACTIVE,  // See inactive in hierarchy
            true,
            IntIds.list.empty()
        ),

        // Admin user edit coordinate
        EditCoordinateRecord.make(
            TinkarTerm.USER.nid(),
            TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),
            TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),
            TinkarTerm.DEVELOPMENT_PATH.nid(),
            TinkarTerm.MASTER_PATH.nid()
        )
    );
}
```

13.4.4. Pattern 4: Multilingual View

For international applications:

```
public ViewCoordinateRecord createMultilingualView(List<Integer> languageNids) {
    // Create language coordinate list with priorities
    List<LanguageCoordinate> languageCoords = languageNids.stream()
        .map(langNid -> LanguageCoordinateRecord.make(
            langNid,
            getDialectNidForLanguage(langNid)
        ))
        .collect(Collectors.toList());

    return ViewCoordinateRecord.make(
        Coordinates.Stamp.DevelopmentLatestActiveOnly(),
        languageCoords,  // Multiple languages with fallback
        Coordinates.Logic.ElPlusPlus(),
        Coordinates.Navigation.inferred(),
        Coordinates.Edit.Default()
    );
}
```

13.5. Using View Calculators

13.5.1. Unified Operations

The ViewCalculator delegates to specialized calculators:

```
ViewCoordinateRecord view = Coordinates.View.DefaultView();
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);

// STAMP operations (delegates to StampCalculator)
Latest<ConceptVersion> latest = calc.latest(conceptNid);
boolean isLatest = calc.isLatestActive(versionNid);

// Language operations (delegates to LanguageCalculator)
String description = calc.getDescriptionText(conceptNid);
String fullName = calc.getFullyQualifiedDescriptionText(conceptNid);
Optional<String> preferredText = calc.getPreferredDescriptionText(conceptNid);

// Navigation operations (delegates to NavigationCalculator)
int[] parents = calc.unsortedParentsOf(conceptNid);
int[] children = calc.sortedChildrenOf(conceptNid);
IntIdSet ancestors = calc.ancestorsOf(conceptNid);

// Logic operations (delegates to LogicCalculator)
DiTreeEntity statedAxioms = calc.getStatedAxiomTree(conceptNid);
DiTreeEntity inferredAxioms = calc.getInferredAxiomTree(conceptNid);

// Edit operations (direct access)
EditCoordinate editCoord = calc.editCoordinate();
int authorNid = editCoord.getAuthorNidForChanges();
```

13.5.2. Switching Contexts

Change view to switch entire context:

```
// Start with development view
ViewCoordinateRecord devView = createDevelopmentView(userNid, moduleNid);
ViewCalculator devCalc = ViewCalculatorWithCache.getCalculator(devView);

// Work in development
String devDescription = devCalc.getDescriptionText(conceptNid);
int[] devParents = devCalc.unsortedParentsOf(conceptNid);

// Switch to production view
ViewCoordinateRecord prodView = createProductionView(releaseTimestamp);
ViewCalculator prodCalc = ViewCalculatorWithCache.getCalculator(prodView);

// Same operations, different results
String prodDescription = prodCalc.getDescriptionText(conceptNid);
int[] prodParents = prodCalc.unsortedParentsOf(conceptNid);
```

```
// Descriptions and hierarchies may differ between dev and production
```

13.6. View Coordinate Modification

Create modified views while preserving other coordinates:

```
ViewCoordinateRecord originalView = Coordinates.View.DefaultView();

// Change just the STAMP coordinate
ViewCoordinateRecord timeView = ViewCoordinateRecord.make(
    originalView.stampCoordinate().withStampPosition(
        StampPositionRecord.make(historicTime, pathNid)
    ),
    originalView.languageCoordinateList(),
    originalView.logicCoordinate(),
    originalView.navigationCoordinate(),
    originalView.editCoordinate()
);

// Change just the language coordinates
ViewCoordinateRecord spanishView = ViewCoordinateRecord.make(
    originalView.stampCoordinate(),
    List.of(Coordinates.Language.SpanishLanguage()),
    originalView.logicCoordinate(),
    originalView.navigationCoordinate(),
    originalView.editCoordinate()
);

// Change navigation to stated
ViewCoordinateRecord statedView = ViewCoordinateRecord.make(
    originalView.stampCoordinate(),
    originalView.languageCoordinateList(),
    originalView.logicCoordinate(),
    Coordinates.Navigation.stated(),
    originalView.editCoordinate()
);
```

13.7. Practical Examples

13.7.1. Example 1: Comparing Development vs Production

```
public void compareDevVsProd(int conceptNid, long releaseTime) {
    // Development view
    ViewCalculator devCalc = ViewCalculatorWithCache.getCalculator(
        createDevelopmentView(userNid, moduleNid)
    );
    // Production view
    ViewCalculator prodCalc = ViewCalculatorWithCache.getCalculator(
        createProductionView(releaseTime)
    );

    // Compare descriptions
    String devDesc = devCalc.getDescriptionText(conceptNid);
    String prodDesc = prodCalc.getDescriptionText(conceptNid);

    if (!devDesc.equals(prodDesc)) {
        System.out.println("Description changed since release:");
        System.out.println("  Production: " + prodDesc);
        System.out.println("  Development: " + devDesc);
    }

    // Compare hierarchies
    IntIdSet devAncestors = devCalc.ancestorsOf(conceptNid);
    IntIdSet prodAncestors = prodCalc.ancestorsOf(conceptNid);

    IntIdSet addedAncestors = devAncestors.difference(prodAncestors);
    IntIdSet removedAncestors = prodAncestors.difference(devAncestors);

    if (!addedAncestors.isEmpty()) {
        System.out.println("New ancestors in development:");
        addedAncestors.forEach(nid ->
            System.out.println("  " + devCalc.getDescriptionText(nid))
        );
    }
}
```

13.7.2. Example 2: User-Specific Views

```
public class ViewManager {
    private final Map<String, ViewCoordinateRecord> userViews = new HashMap<>();

    public ViewCalculator getViewForUser(String userId) {
        ViewCoordinateRecord view = userViews.computeIfAbsent(userId, id -> {
            User user = getUserDetails(id);

            return ViewCoordinateRecord.make(
                // User's preferred time/path
                StampCoordinateRecord.make(
                    user.showInactive() ?
                        StateSet.ACTIVE_AND_INACTIVE :
                        StateSet.ACTIVE_ONLY,
                    StampPositionRecord.make(
                        user.getViewTime(),
                        user.getPathNid()
                    )
                ),

                // User's language preferences
                user.getLanguagePreferences(),

                // Standard logic
                Coordinates.Logic.ElPlusPlus(),

                // User's navigation preference
                user.preferStated() ?
                    Coordinates.Navigation.stated() :
                    Coordinates.Navigation.inferred(),

                // User-specific edit coordinate
                EditCoordinateRecord.make(
                    user.getConceptNid(),
                    user.getDefaultModule(),
                    user.getDefaultModule(),
                    TinkarTerm.DEVELOPMENT_PATH.nid(),
                    TinkarTerm.MASTER_PATH.nid()
                )
            );
        });

        return ViewCalculatorWithCache.getCalculator(view);
    }
}
```

13.7.3. Example 3: Temporal Comparison

```
public void analyzeChangesOverTime(int conceptNid, long startTime, long endTime) {
    // View at start time
    ViewCalculator startCalc = ViewCalculatorWithCache.getCalculator(
        ViewCoordinateRecord.make(
            StampCoordinateRecord.make(
                StateSet.ACTIVE_AND_INACTIVE,
                StampPositionRecord.make(startTime, pathNid)
            ),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            Coordinates.Edit.Default()
        )
    );

    // View at end time
    ViewCalculator endCalc = ViewCalculatorWithCache.getCalculator(
        ViewCoordinateRecord.make(
            StampCoordinateRecord.make(
                StateSet.ACTIVE_AND_INACTIVE,
                StampPositionRecord.make(endTime, pathNid)
            ),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            Coordinates.Edit.Default()
        )
    );

    // Compare states
    Latest<ConceptVersion> startVersion = startCalc.latest(conceptNid);
    Latest<ConceptVersion> endVersion = endCalc.latest(conceptNid);

    if (startVersion.isPresent() && endVersion.isPresent()) {
        if (startVersion.get().state() != endVersion.get().state()) {
            System.out.println("State changed: " +
                startVersion.get().state() + " → " +
                endVersion.get().state());
        }
    }

    // Compare descriptions
    String startDesc = startCalc.getDescriptionText(conceptNid);
    String endDesc = endCalc.getDescriptionText(conceptNid);

    if (!startDesc.equals(endDesc)) {
        System.out.println("Description changed:");
        System.out.println("  Before: " + startDesc);
        System.out.println("  After: " + endDesc);
    }
}
```

13.8. Best Practices

13.8.1. DO: Use View Coordinates for Consistency

```
// GOOD: Single view for all operations
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);

String description = calc.getDescriptionText(conceptNid);
int[] parents = calc.unsortedParentsOf(conceptNid);
Latest<ConceptVersion> latest = calc.latest(conceptNid);

// All operations use same coordinated context
```

13.8.2. DO: Cache View Calculators

```
// GOOD: Reuse calculator
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);

for (int conceptNid : conceptNids) {
    String desc = calc.getDescriptionText(conceptNid);
    // ... process
}

// ViewCalculatorWithCache handles caching internally
```

13.8.3. DON’T: Mix Coordinate Types

```
// BAD: Mixing different coordinates
StampCalculator stampCalc1 = StampCalculatorWithCache.getCalculator(stamp1);
LanguageCalculator langCalc = LanguageCalculatorWithCache.getCalculator(
    stamp2,  // Different stamp!
    languageCoords
);

// Inconsistent context leads to confusing results
```

13.8.4. DO: Create Views for Specific Use Cases

```
// GOOD: Specific views for different needs
public class Views {
    public static ViewCoordinateRecord forDevelopment() { ... }
    public static ViewCoordinateRecord forProduction() { ... }
    public static ViewCoordinateRecord forAudit() { ... }
    public static ViewCoordinateRecord forUser(User user) { ... }
}

// Clear, reusable view definitions
```

13.9. Summary

View coordinates unify the five coordinate types:

- **Unified Access -** Single calculator for all operations
- **Consistent Context -** All operations use coordinated view
- **Easy Switching -** Change context by switching views
- **Delegation -** ViewCalculator delegates to specialized calculators

Key points:

- Use predefined views for common scenarios
- Create custom views for specific needs
- Cache view calculators for performance
- Switch views to change entire context
- Modify views while preserving other coordinates

View coordinates simplify coordinate management by providing a unified, consistent way to access all Tinkar functionality.

14. Practical Examples

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

14.1. Overview

This section provides real-world examples of coordinate configuration for common scenarios. Each example shows complete code and explains the rationale behind coordinate choices.

14.2. Example 1: Clinical Terminology Browser

Build a browser for SNOMED CT clinical terminology.

14.2.1. Requirements

- Display current active concepts only
- Show preferred English (US) descriptions
- Navigate inferred "IS-A" hierarchy
- Sort children alphabetically
- Support full-text search

14.2.2. Solution

```
public class ClinicalTerminologyBrowser {
    private final ViewCalculator viewCalc;

    public ClinicalTerminologyBrowser() {
        // Configure view for clinical browsing
        ViewCoordinateRecord view = ViewCoordinateRecord.make(
            // STAMP: Latest active on master path
            StampCoordinateRecord.make(
                StateSet.ACTIVE_ONLY,  // Only active concepts
                StampPositionRecord.make(
                    Long.MAX_VALUE,    // Latest
                    TinkarTerm.MASTER_PATH.nid()  // Production
                )
            ),

            // Language: US English preferred terms
            List.of(
                LanguageCoordinateRecord.make(
                    TinkarTerm.ENGLISH_LANGUAGE.nid(),
                    List.of(TinkarTerm.US_DIALECT_ASSEMBLAGE.nid()),
                    List.of(
                        TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
                        TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
                    )
                )
            ),

            // Logic: Inferred relationships
            LogicCoordinateRecord.make(
                TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN.nid(),
                TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN.nid(),
                TinkarTerm.SNOROCKET_CLASSIFIER.nid(),
                TinkarTerm.INFERRED_PREMISE_TYPE.nid()
            ),

            // Navigation: Inferred, sorted
            NavigationCoordinateRecord.make(
                IntIds.set.of(TinkarTerm.INFERRED_NAVIGATION.nid()),
                StateSet.ACTIVE_ONLY,
                true,  // Sort children
                IntIds.list.empty()  // By description text
            ),

            // Edit: Read-only
            Coordinates.Edit.Default()
        );

        this.viewCalc = ViewCalculatorWithCache.getCalculator(view);
    }

    public ConceptHierarchyNode getConceptHierarchy(int conceptNid) {
        String description = viewCalc.getDescriptionText(conceptNid);
        int[] children = viewCalc.sortedChildrenOf(conceptNid);

        return new ConceptHierarchyNode(
            conceptNid,
            description,
            Arrays.stream(children)
                .mapToObj(this::getConceptHierarchy)
                .collect(Collectors.toList())
        );
    }

    public List<SearchResult> search(String query) {
        // Search using view context
        return performSearch(query, viewCalc);
    }
}
```

14.2.3. Why These Coordinates?

- **STAMP**: Active only + master path ensures production-ready content
- **Language**: Regular names are clinically appropriate; FSN provides fallback
- **Logic**: Inferred relationships show computed subsumption hierarchy
- **Navigation**: Sorted children make browsing intuitive
- **Edit**: Not needed for read-only browsing

14.3. Example 2: Multi-Language Medical Dictionary

Support medical terminology in multiple languages with fallback.

14.3.1. Requirements

- Support Spanish, English, French (in priority order)
- Show active and inactive terms (for reference)
- Use latest development versions
- Display both stated and inferred relationships
- Enable editing by authenticated users

14.3.2. Solution

```
public class MultilingualMedicalDictionary {
    private final Map<String, ViewCalculator> userViews = new HashMap<>();

    public ViewCalculator getViewForUser(User user) {
        return userViews.computeIfAbsent(user.getId(), id -> {
            // Build cascading language preferences
            List<LanguageCoordinate> languagePrefs = new ArrayList<>();

            // Add user's preferred languages in order
            for (String langCode : user.getLanguagePreferences()) {
                languagePrefs.add(createLanguageCoordinate(langCode));
            }

            // Always add English as ultimate fallback
            if (!user.getLanguagePreferences().contains("en")) {
                languagePrefs.add(
                    LanguageCoordinateRecord.make(
                        TinkarTerm.ENGLISH_LANGUAGE.nid(),
                        List.of(TinkarTerm.US_DIALECT_ASSEMBLAGE.nid()),
                        List.of(
                            TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
                            TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
                        )
                    )
                );
            }

            ViewCoordinateRecord view = ViewCoordinateRecord.make(
                // STAMP: All states for reference
                StampCoordinateRecord.make(
                    StateSet.ACTIVE_AND_INACTIVE,
                    StampPositionRecord.make(
                        Long.MAX_VALUE,
                        TinkarTerm.DEVELOPMENT_PATH.nid()
                    )
                ),

                // Language: Cascading preferences
                languagePrefs,

                // Logic: Both stated and inferred
                Coordinates.Logic.ElPlusPlus(),

                // Navigation: Inferred hierarchy
                Coordinates.Navigation.inferred(),

                // Edit: User-specific
                EditCoordinateRecord.make(
                    user.getConceptNid(),
                    user.getDefaultModule(),
                    user.getDefaultModule(),
                    TinkarTerm.DEVELOPMENT_PATH.nid(),
                    TinkarTerm.DEVELOPMENT_PATH.nid()
                )
            );

            return ViewCalculatorWithCache.getCalculator(view);
        });
    }

    private LanguageCoordinate createLanguageCoordinate(String langCode) {
        return switch (langCode) {
            case "es" -> LanguageCoordinateRecord.make(
                TinkarTerm.SPANISH_LANGUAGE.nid(),
                List.of(TinkarTerm.SPANISH_DIALECT_ASSEMBLAGE.nid()),
                List.of(
                    TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
                    TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
                )
            );
            case "fr" -> LanguageCoordinateRecord.make(
                TinkarTerm.FRENCH_LANGUAGE.nid(),
                List.of(TinkarTerm.FRENCH_DIALECT_ASSEMBLAGE.nid()),
                List.of(
                    TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
                    TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
                )
            );
            default -> LanguageCoordinateRecord.make(
                TinkarTerm.ENGLISH_LANGUAGE.nid(),
                List.of(TinkarTerm.US_DIALECT_ASSEMBLAGE.nid()),
                List.of(
                    TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
                    TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
                )
            );
        };
    }

    public TranslatedConcept getTranslations(int conceptNid, User user) {
        ViewCalculator calc = getViewForUser(user);

        // Get description in preferred language (with fallback)
        String preferredDesc = calc.getDescriptionText(conceptNid);

        // Get descriptions in all available languages
        Map<String, String> translations = new HashMap<>();
        for (String langCode : List.of("en", "es", "fr")) {
            // Create temporary view for specific language
            ViewCoordinate langView = createSingleLanguageView(langCode);
            ViewCalculator langCalc = ViewCalculatorWithCache.getCalculator(
                langView.toViewCoordinateRecord()
            );

            String desc = langCalc.getDescriptionText(conceptNid);
            if (desc != null && !desc.isEmpty()) {
                translations.put(langCode, desc);
            }
        }

        return new TranslatedConcept(conceptNid, preferredDesc, translations);
    }
}
```

14.3.3. Why These Coordinates?

- **STAMP**: Include inactive for completeness in reference dictionary
- **Language**: Cascading with fallback ensures something always displays
- **Logic**: Both stated/inferred show complete relationships
- **Navigation**: Inferred for standard hierarchy
- **Edit**: User-specific for proper attribution

14.4. Example 3: Terminology Authoring Tool

Support content authors creating and modifying terminology.

14.4.1. Requirements

- See all content including work-in-progress
- View both stated and inferred relationships
- Navigate stated hierarchy for authoring
- Track changes by author
- Support module-based development

14.4.2. Solution

```
public class TerminologyAuthoringTool {
    private final Map<String, ViewCalculator> authorViews = new HashMap<>();

    public ViewCalculator getAuthorView(Author author) {
        return authorViews.computeIfAbsent(author.getId(), id -> {
            ViewCoordinateRecord view = ViewCoordinateRecord.make(
                // STAMP: All states, latest on dev path
                StampCoordinateRecord.make(
                    StateSet.ACTIVE_AND_INACTIVE,  // See all content
                    StampPositionRecord.make(
                        Long.MAX_VALUE,            // Latest
                        TinkarTerm.DEVELOPMENT_PATH.nid()
                    )
                ),

                // Language: Author's preference with fallback
                createLanguageCoordinates(author),

                // Logic: Stated axioms for editing
                LogicCoordinateRecord.make(
                    TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN.nid(),
                    TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN.nid(),
                    TinkarTerm.SNOROCKET_CLASSIFIER.nid(),
                    TinkarTerm.STATED_PREMISE_TYPE.nid()  // Stated!
                ),

                // Navigation: Stated hierarchy with all states
                NavigationCoordinateRecord.make(
                    IntIds.set.of(TinkarTerm.STATED_NAVIGATION.nid()),
                    StateSet.ACTIVE_AND_INACTIVE,  // See inactive
                    true,                          // Sorted
                    IntIds.list.empty()
                ),

                // Edit: Author-specific
                EditCoordinateRecord.make(
                    author.getConceptNid(),
                    author.getDefaultModule(),
                    author.getDefaultModule(),
                    TinkarTerm.DEVELOPMENT_PATH.nid(),
                    TinkarTerm.DEVELOPMENT_PATH.nid()
                )
            );

            return ViewCalculatorWithCache.getCalculator(view);
        });
    }

    public void createNewConcept(
        String description,
        int parentNid,
        Author author
    ) {
        ViewCalculator calc = getAuthorView(author);
        EditCoordinate editCoord = calc.editCoordinate();

        // Create STAMP for new version
        StampEntity stamp = StampRecord.build(
            State.ACTIVE,
            System.currentTimeMillis(),
            editCoord.getAuthorNidForChanges(),
            editCoord.getDefaultModuleNid(),
            editCoord.getDefaultPathNid()
        );

        // Create concept
        ConceptEntity concept = ConceptRecord.build(
            stamp.nid()
        );

        // Add description
        SemanticEntity description = createDescription(
            concept.nid(),
            description,
            stamp,
            editCoord
        );

        // Add stated parent relationship
        SemanticEntity relationship = createIsARelationship(
            concept.nid(),
            parentNid,
            stamp,
            editCoord
        );

        // Commit all
        Entity.provider().putEntity(concept);
        Entity.provider().putEntity(description);
        Entity.provider().putEntity(relationship);
    }

    public void compareStatedVsInferred(int conceptNid, Author author) {
        ViewCalculator authorCalc = getAuthorView(author);

        // Get stated parents (what author created)
        ViewCoordinateRecord statedView = authorCalc.viewCoordinate()
            .toViewCoordinateRecord();
        ViewCalculator statedCalc = ViewCalculatorWithCache.getCalculator(
            statedView
        );
        int[] statedParents = statedCalc.unsortedParentsOf(conceptNid);

        // Get inferred parents (what classifier computed)
        ViewCoordinateRecord inferredView = ViewCoordinateRecord.make(
            statedView.stampCoordinate(),
            statedView.languageCoordinateList(),
            LogicCoordinateRecord.make(
                TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN.nid(),
                TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN.nid(),
                TinkarTerm.SNOROCKET_CLASSIFIER.nid(),
                TinkarTerm.INFERRED_PREMISE_TYPE.nid()  // Inferred
            ),
            Coordinates.Navigation.inferred(),
            statedView.editCoordinate()
        );
        ViewCalculator inferredCalc = ViewCalculatorWithCache.getCalculator(
            inferredView
        );
        int[] inferredParents = inferredCalc.unsortedParentsOf(conceptNid);

        // Compare
        IntIdSet statedSet = IntIds.set.of(statedParents);
        IntIdSet inferredSet = IntIds.set.of(inferredParents);

        IntIdSet onlyStated = statedSet.difference(inferredSet);
        IntIdSet onlyInferred = inferredSet.difference(statedSet);

        System.out.println("Stated but not inferred:");
        onlyStated.forEach(nid ->
            System.out.println("  " + authorCalc.getDescriptionText(nid))
        );

        System.out.println("\nInferred but not stated:");
        onlyInferred.forEach(nid ->
            System.out.println("  " + authorCalc.getDescriptionText(nid))
        );
    }
}
```

14.4.3. Why These Coordinates?

- **STAMP**: All states shows work-in-progress and inactive content
- **Language**: Author's preference improves UX
- **Logic**: Stated axioms for editing; compare with inferred
- **Navigation**: Stated hierarchy for authoring structure
- **Edit**: Author-specific for proper change attribution

14.5. Example 4: Version Comparison Tool

Compare terminology versions across time or paths.

14.5.1. Requirements

- Compare same concept at different time points
- Compare development vs production versions
- Show what changed (descriptions, relationships, status)
- Support different paths

14.5.2. Solution

```
public class VersionComparisonTool {

    public ConceptComparison compare(
        int conceptNid,
        StampPosition position1,
        StampPosition position2
    ) {
        // Create view for first position
        ViewCalculator calc1 = createViewForPosition(position1);
        // Create view for second position
        ViewCalculator calc2 = createViewForPosition(position2);

        return new ConceptComparison(
            compareDescriptions(conceptNid, calc1, calc2),
            compareHierarchy(conceptNid, calc1, calc2),
            compareStatus(conceptNid, calc1, calc2),
            compareModule(conceptNid, calc1, calc2)
        );
    }

    private ViewCalculator createViewForPosition(StampPosition position) {
        ViewCoordinateRecord view = ViewCoordinateRecord.make(
            StampCoordinateRecord.make(
                StateSet.ACTIVE_AND_INACTIVE,  // See all states
                position
            ),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            Coordinates.Edit.Default()
        );

        return ViewCalculatorWithCache.getCalculator(view);
    }

    private DescriptionChanges compareDescriptions(
        int conceptNid,
        ViewCalculator calc1,
        ViewCalculator calc2
    ) {
        String desc1 = calc1.getDescriptionText(conceptNid);
        String desc2 = calc2.getDescriptionText(conceptNid);

        return new DescriptionChanges(
            desc1,
            desc2,
            !desc1.equals(desc2)
        );
    }

    private HierarchyChanges compareHierarchy(
        int conceptNid,
        ViewCalculator calc1,
        ViewCalculator calc2
    ) {
        IntIdSet parents1 = IntIds.set.of(calc1.unsortedParentsOf(conceptNid));
        IntIdSet parents2 = IntIds.set.of(calc2.unsortedParentsOf(conceptNid));

        IntIdSet added = parents2.difference(parents1);
        IntIdSet removed = parents1.difference(parents2);
        return new HierarchyChanges(
            parents1,
            parents2,
            added,
            removed
        );
    }

    private StatusChange compareStatus(
        int conceptNid,
        ViewCalculator calc1,
        ViewCalculator calc2
    ) {
        Latest<ConceptVersion> latest1 = calc1.latest(conceptNid);
        Latest<ConceptVersion> latest2 = calc2.latest(conceptNid);

        State state1 = latest1.isPresent() ?
            latest1.get().state() : null;
        State state2 = latest2.isPresent() ?
            latest2.get().state() : null;

        return new StatusChange(state1, state2);
    }

    private ModuleChange compareModule(
        int conceptNid,
        ViewCalculator calc1,
        ViewCalculator calc2
    ) {
        Latest<ConceptVersion> latest1 = calc1.latest(conceptNid);
        Latest<ConceptVersion> latest2 = calc2.latest(conceptNid);

        int module1 = latest1.isPresent() ?
            latest1.get().moduleNid() : 0;
        int module2 = latest2.isPresent() ?
            latest2.get().moduleNid() : 0;

        return new ModuleChange(module1, module2);
    }

    // Usage examples
    public void compareDevelopmentVsProduction(
        int conceptNid,
        long releaseTime
    ) {
        StampPosition devPos = StampPositionRecord.make(
            Long.MAX_VALUE,
            TinkarTerm.DEVELOPMENT_PATH.nid()
        );
        StampPosition prodPos = StampPositionRecord.make(
            releaseTime,
            TinkarTerm.MASTER_PATH.nid()
        );

        ConceptComparison comparison = compare(conceptNid, devPos, prodPos);

        System.out.println("Changes since release:");
        if (comparison.descriptions().hasChanges()) {
            System.out.println("  Description: " +
                comparison.descriptions().oldDescription() + " → " +
                comparison.descriptions().newDescription());
        }

        if (!comparison.hierarchy().added().isEmpty()) {
            System.out.println("  Added parents: " +
                comparison.hierarchy().added().size());
        }
    }

    public void compareAcrossTime(
        int conceptNid,
        long startTime,
        long endTime,
        int pathNid
    ) {
        StampPosition startPos = StampPositionRecord.make(startTime, pathNid);
        StampPosition endPos = StampPositionRecord.make(endTime, pathNid);

        ConceptComparison comparison = compare(conceptNid, startPos, endPos);

        System.out.println("Changes over time:");
        if (comparison.status().hasChanged()) {
            System.out.println("  Status: " +
                comparison.status().oldState() + " → " +
                comparison.status().newState());
        }
    }
}
```

14.5.3. Why These Coordinates?

- **STAMP**: Different positions enable temporal comparison
- **STAMP**: All states shows status transitions
- **Language**: Consistent for fair comparison
- **Logic**: Standard for both views
- **Navigation**: Standard for hierarchy comparison

14.6. Example 5: Module-Based Development

Manage content across multiple modules with dependencies.

14.6.1. Requirements

- Core module with base terminology
- Extension modules depend on core
- Filter content by module
- Priority when modules conflict
- Promote between modules

14.6.2. Solution

```
public class ModularDevelopment {
    private final int coreModuleNid;
    private final int extensionModuleNid;

    public ModularDevelopment(int coreModuleNid, int extensionModuleNid) {
        this.coreModuleNid = coreModuleNid;
        this.extensionModuleNid = extensionModuleNid;
    }

    // View showing only core module content
    public ViewCalculator getCoreView() {
        ViewCoordinateRecord view = ViewCoordinateRecord.make(
            StampCoordinateRecord.make(
                StateSet.ACTIVE_ONLY,
                StampPositionRecord.make(
                    Long.MAX_VALUE,
                    TinkarTerm.DEVELOPMENT_PATH.nid()
                )
            ).withModuleNids(
                IntIds.set.of(coreModuleNid)  // Only core
            ),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            Coordinates.Edit.Default()
        );

        return ViewCalculatorWithCache.getCalculator(view);
    }

    // View showing core + extension with priority
    public ViewCalculator getExtensionView() {
        ViewCoordinateRecord view = ViewCoordinateRecord.make(
            StampCoordinateRecord.make(
                StateSet.ACTIVE_ONLY,
                StampPositionRecord.make(
                    Long.MAX_VALUE,
                    TinkarTerm.DEVELOPMENT_PATH.nid()
                )
            ).withModuleNids(
                IntIds.set.of(coreModuleNid, extensionModuleNid)
            ).withModulePriorityNidList(
                IntIds.list.of(
                    extensionModuleNid,  // Extension first
                    coreModuleNid        // Core fallback
                )
            ),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            EditCoordinateRecord.make(
                TinkarTerm.USER.nid(),
                extensionModuleNid,      // New content to extension
                extensionModuleNid,
                TinkarTerm.DEVELOPMENT_PATH.nid(),
                TinkarTerm.DEVELOPMENT_PATH.nid()
            )
        );

        return ViewCalculatorWithCache.getCalculator(view);
    }

    // Create content in extension that extends core
    public void createExtensionConcept(
        String description,
        int coreParentNid,
        Author author
    ) {
        ViewCalculator extensionCalc = getExtensionView();
        EditCoordinate editCoord = extensionCalc.editCoordinate();

        // Verify parent is in core module
        ViewCalculator coreCalc = getCoreView();
        Latest<ConceptVersion> parent = coreCalc.latest(coreParentNid);

        if (!parent.isPresent()) {
            throw new IllegalArgumentException(
                "Parent must be in core module"
            );
        }

        // Create concept in extension module
        StampEntity stamp = StampRecord.build(
            State.ACTIVE,
            System.currentTimeMillis(),
            author.getConceptNid(),
            extensionModuleNid,  // Extension module!
            editCoord.getDefaultPathNid()
        );

        ConceptEntity concept = ConceptRecord.build(stamp.nid());

        // Add content...
        Entity.provider().putEntity(concept);
    }

    // Promote extension content to core
    public void promoteToCore(int conceptNid) {
        ViewCalculator extensionCalc = getExtensionView();

        // Get current version from extension
        Latest<ConceptVersion> extensionVersion =
            extensionCalc.latest(conceptNid);

        if (!extensionVersion.isPresent()) {
            throw new IllegalArgumentException("Concept not found");
        }

        if (extensionVersion.get().moduleNid() != extensionModuleNid) {
            throw new IllegalArgumentException(
                "Concept not in extension module"
            );
        }

        // Create new version in core module
        StampEntity coreStamp = StampRecord.build(
            extensionVersion.get().state(),
            System.currentTimeMillis(),
            extensionVersion.get().authorNid(),
            coreModuleNid,  // Core module!
            extensionVersion.get().pathNid()
        );

        ConceptVersionRecord coreVersion = ConceptVersionRecord.build(
            conceptNid,
            coreStamp.nid()
        );

        Entity.provider().putEntity(coreVersion);

        // Inactivate in extension
        StampEntity inactiveStamp = StampRecord.build(
            State.INACTIVE,
            System.currentTimeMillis(),
            extensionVersion.get().authorNid(),
            extensionModuleNid,
            extensionVersion.get().pathNid()
        );

        ConceptVersionRecord inactiveVersion = ConceptVersionRecord.build(
            conceptNid,
            inactiveStamp.nid()
        );

        Entity.provider().putEntity(inactiveVersion);
    }
}
```

14.6.3. Why These Coordinates?

- **STAMP**: Module filtering isolates content by module
- **STAMP**: Module priority resolves conflicts
- **Edit**: Default module directs new content
- **Edit**: Destination module supports promotion

14.7. Summary

These examples demonstrate:

- **Browser**: Active-only, production path, inferred navigation
- **Multilingual**: Cascading language fallback, all states
- **Authoring**: All states, stated navigation, author attribution
- **Comparison**: Temporal views, comprehensive change detection
- **Modular**: Module filtering, priority, and promotion

Key patterns:

- Choose coordinates based on use case requirements
- Use predefined coordinates when possible
- Cache calculators for performance
- Create user-specific views for personalization
- Compare views to detect changes

15. Best Practices

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

15.1. Overview

This section provides guidance on effectively using coordinates in IKE applications. Following these practices will help you build robust, performant, and maintainable systems.

15.2. Coordinate Creation and Management

15.2.1. Use Predefined Coordinates

```
// GOOD: Use factory methods
StampCoordinateRecord stamp = Coordinates.Stamp.DevelopmentLatestActiveOnly();
LanguageCoordinate lang = Coordinates.Language.UsEnglishRegularName();

// BAD: Manual construction when factory exists
StampCoordinateRecord stamp = StampCoordinateRecord.make(
    StateSet.ACTIVE_ONLY,
    StampPositionRecord.make(Long.MAX_VALUE, developmentPathNid)
);
```

Why? Predefined coordinates are tested, documented, and communicate intent clearly.

15.2.2. Create Reusable Coordinate Factories

```
// GOOD: Centralized coordinate definitions
public class AppCoordinates {
    public static ViewCoordinateRecord production() {
        return ViewCoordinateRecord.make(
            Coordinates.Stamp.DevelopmentLatestActiveOnly()
                .withPath(TinkarTerm.MASTER_PATH),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            Coordinates.Edit.Default()
        );
    }

    public static ViewCoordinateRecord development(int authorNid) {
        return ViewCoordinateRecord.make(
            Coordinates.Stamp.DevelopmentLatestActiveOnly(),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.stated(),
            EditCoordinateRecord.make(
                authorNid,
                TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),
                TinkarTerm.SOLOR_OVERLAY_MODULE.nid(),
                TinkarTerm.DEVELOPMENT_PATH.nid(),
                TinkarTerm.DEVELOPMENT_PATH.nid()
            )
        );
    }
}

// Usage
ViewCalculator prodCalc = ViewCalculatorWithCache.getCalculator(
    AppCoordinates.production()
);
```

Why? Centralization ensures consistency and simplifies maintenance.

15.2.3. Document Custom Coordinates

```
// GOOD: Clear documentation
/**
 * Creates a view coordinate for clinical content review.
 * <p>
 * Configuration:
 * - Shows both active and inactive content (for review)
 * - Latest versions on development path
 * - English descriptions with FSN fallback
 * - Inferred navigation for clinical hierarchy
 * - Read-only (no editing)
 *
 * @return view coordinate for clinical review
 */
public static ViewCoordinateRecord clinicalReview() {
    return ViewCoordinateRecord.make(
        StampCoordinateRecord.make(
            StateSet.ACTIVE_AND_INACTIVE,  // See inactive for review
            StampPositionRecord.make(
                Long.MAX_VALUE,
                TinkarTerm.DEVELOPMENT_PATH.nid()
            )
        ),
        // ... rest of configuration
    );
}
```

Why? Documents rationale for coordinate choices and expected behavior.

15.3. Calculator Usage

15.3.1. Cache Calculators, Not Coordinates

```
// GOOD: Cache calculators
private final ViewCalculator calculator;

public MyService(ViewCoordinateRecord view) {
    this.calculator = ViewCalculatorWithCache.getCalculator(view);
}

public void process(List<Integer> conceptNids) {
    for (int nid : conceptNids) {
        String desc = calculator.getDescriptionText(nid);
        // Process...
    }
}

// BAD: Recreate calculator repeatedly
public void process(List<Integer> conceptNids) {
    for (int nid : conceptNids) {
        ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);
        String desc = calc.getDescriptionText(nid);
    }
}
```

Why? Calculator creation has overhead; caching improves performance significantly.

15.3.2. Use Appropriate Calculator Type

```
// GOOD: Use specific calculator when only one coordinate type needed
StampCalculator stampCalc = StampCalculatorWithCache.getCalculator(stampCoord);
Latest<ConceptVersion> latest = stampCalc.latest(conceptNid);

// AVOID: Using ViewCalculator when only STAMP operations needed
ViewCalculator viewCalc = ViewCalculatorWithCache.getCalculator(view);
Latest<ConceptVersion> latest = viewCalc.latest(conceptNid);
```

Why? Specific calculators have less overhead when you don’t need full view functionality.

15.3.3. Don’t Mix Calculator Contexts

```
// BAD: Using results from different contexts
StampCalculator stampCalc1 = StampCalculatorWithCache.getCalculator(stamp1);
LanguageCalculator langCalc2 = LanguageCalculatorWithCache.getCalculator(
    stamp2,  // Different STAMP!
    languageCoords
);

Latest<ConceptVersion> latest = stampCalc1.latest(conceptNid);
String description = langCalc2.getDescriptionText(conceptNid);
```

```
// Description may be for different version!

// GOOD: Consistent context
ViewCalculator viewCalc = ViewCalculatorWithCache.getCalculator(view);
Latest<ConceptVersion> latest = viewCalc.latest(conceptNid);
String description = viewCalc.getDescriptionText(conceptNid);
// Description matches version
```

Why? Mixing contexts leads to inconsistent, confusing results.

15.4. STAMP Coordinates

15.4.1. Choose Appropriate State Filter

```
// Production/user-facing: Active only
StampCoordinateRecord userStamp = StampCoordinateRecord.make(
    StateSet.ACTIVE_ONLY,
    position
);

// Administrative/review: Include inactive
StampCoordinateRecord adminStamp = StampCoordinateRecord.make(
    StateSet.ACTIVE_AND_INACTIVE,
    position
);

// Specific state needs: Use StateSet methods
StateSet customStates = StateSet.ACTIVE_ONLY
    .withState(State.INACTIVE);
```

Why? State filtering prevents inappropriate content from reaching users.

15.4.2. Use Appropriate Time Position

```
// Latest versions
StampPositionRecord latest = StampPositionRecord.make(
    Long.MAX_VALUE,
    pathNid
);

// Point-in-time (release, audit, comparison)
StampPositionRecord release = StampPositionRecord.make(
    releaseTimestamp,
    pathNid
);

// Specific date range queries
```

```
long startTime = LocalDate.of(2024, 1, 1)
    .atStartOfDay(ZoneId.systemDefault())
    .toInstant()
    .toEpochMilli();
StampPositionRecord start = StampPositionRecord.make(startTime, pathNid);
```

Why? Time position controls which versions are visible.

15.4.3. Use Module Filtering Appropriately

```
// Include specific modules
StampCoordinateRecord filtered = stamp.withModuleNids(
    IntIds.set.of(module1Nid, module2Nid)
);

// Exclude specific modules
StampCoordinateRecord excluded = stamp.withExcludedModuleNids(
    IntIds.set.of(testModuleNid)
);

// Module priority (when versions conflict)
StampCoordinateRecord prioritized = stamp.withModulePriorityNidList(
    IntIds.list.of(
        extensionModuleNid,  // Prefer extension
        coreModuleNid        // Fall back to core
    )
);
```

Why? Module filtering enables modular terminology management.

15.5. Language Coordinates

15.5.1. Provide Appropriate Fallback

```
// GOOD: Cascading fallback
List<LanguageCoordinate> languages = List.of(
    Coordinates.Language.SpanishLanguage(),  // Preferred
    Coordinates.Language.UsEnglishRegularName()  // Fallback
);

// AVOID: No fallback (may return empty)
List<LanguageCoordinate> languages = List.of(
    Coordinates.Language.SpanishLanguage()
);
```

Why? Fallback ensures descriptions are always available.

15.5.2. Order Description Types Appropriately

```
// Clinical display: Regular name first
LanguageCoordinate clinical = LanguageCoordinateRecord.make(
    TinkarTerm.ENGLISH_LANGUAGE.nid(),
    List.of(TinkarTerm.US_DIALECT_ASSEMBLAGE.nid()),
    List.of(
        TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid(),
        TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid()
    )
);

// Technical/debugging: FSN first
LanguageCoordinate technical = LanguageCoordinateRecord.make(
    TinkarTerm.ENGLISH_LANGUAGE.nid(),
    List.of(TinkarTerm.US_DIALECT_ASSEMBLAGE.nid()),
    List.of(
        TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE.nid(),
        TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE.nid()
    )
);
```

Why? Description type order affects which descriptions are selected.

15.5.3. Use Language-Agnostic for Identifiers

```
// GOOD: Language-agnostic for UUIDs, codes
LanguageCoordinate identifiers =
    Coordinates.Language.LanguageAgnostic();

// Use for code lookups
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(
    view.withLanguageCoordinates(List.of(identifiers))
);
```

Why? Identifiers don’t depend on language preferences.

15.6. Logic Coordinates

15.6.1. Match Premise Type to Use Case

```
// Authoring/editing: Stated axioms
LogicCoordinate authoring = LogicCoordinateRecord.make(
    statedPatternNid,
    inferredPatternNid,
    classifierNid,
    TinkarTerm.STATED_PREMISE_TYPE.nid()  // What author created
);

// Querying/browsing: Inferred axioms
LogicCoordinate querying = LogicCoordinateRecord.make(
    statedPatternNid,
    inferredPatternNid,
    classifierNid,
    TinkarTerm.INFERRED_PREMISE_TYPE.nid()  // What classifier computed
);
```

Why? Stated shows authored content; inferred shows computed subsumption.

15.6.2. Understand Classifier Differences

```
// EL++: Fast, suitable for most clinical terminologies
LogicCoordinate elPlusPlus = Coordinates.Logic.ElPlusPlus();

// Full OWL: More expressive but slower
// Use only when EL++ is insufficient
```

Why? Classifier choice affects reasoning capabilities and performance.

15.7. Navigation Coordinates

15.7.1. Match Navigation Pattern to Task

```
// Browsing: Inferred (shows computed hierarchy)
NavigationCoordinate browsing = Coordinates.Navigation.inferred();

// Authoring: Stated (shows authored hierarchy)
NavigationCoordinate authoring = Coordinates.Navigation.stated();

// Comparison: Both
NavigationCoordinate both = NavigationCoordinateRecord.make(
    IntIds.set.of(
        TinkarTerm.INFERRED_NAVIGATION.nid(),
        TinkarTerm.STATED_NAVIGATION.nid()
    ),
    StateSet.ACTIVE_ONLY,
    true,
    IntIds.list.empty()
);
```

Why? Navigation pattern affects which relationships are used.

15.7.2. Use Vertex State Filtering

```
// Production: Active vertices only
NavigationCoordinate production = NavigationCoordinateRecord.make(
    navigationPatternNids,
    StateSet.ACTIVE_ONLY,  // Only active in hierarchy
    true,
    IntIds.list.empty()
);

// Review: Include inactive
NavigationCoordinate review = NavigationCoordinateRecord.make(
    navigationPatternNids,
    StateSet.ACTIVE_AND_INACTIVE,  // See deprecated concepts
    true,
    IntIds.list.empty()
);
```

Why? Vertex state filtering controls hierarchy visibility.

15.7.3. Enable Sorting for User Display

```
// User-facing: Sorted
NavigationCoordinate userNav = NavigationCoordinateRecord.make(
    navigationPatternNids,
    StateSet.ACTIVE_ONLY,
    true,  // Sort for readability
    IntIds.list.empty()
);

// Programmatic: Unsorted (when order doesn't matter)
NavigationCoordinate progNav = NavigationCoordinateRecord.make(
    navigationPatternNids,
    StateSet.ACTIVE_ONLY,
    false,  // Don't sort (faster)
    IntIds.list.empty()
);
```

Why? Sorting improves UX but has performance cost.

15.8. Edit Coordinates

15.8.1. Use Per-User Edit Coordinates

```
// GOOD: Each user has own edit coordinate
public EditCoordinateRecord getEditCoordinateForUser(User user) {
    return EditCoordinateRecord.make(
        user.getConceptNid(),     // Proper attribution
        user.getDefaultModule(),
        user.getDefaultModule(),
        TinkarTerm.DEVELOPMENT_PATH.nid(),
        TinkarTerm.DEVELOPMENT_PATH.nid()
    );
}

// BAD: Shared edit coordinate
public static final EditCoordinateRecord SHARED_EDIT =
    Coordinates.Edit.Default();
// Can't track who made changes!
```

Why? Per-user coordinates enable proper change attribution.

15.8.2. Match Default Module to Context

```
// Core development
EditCoordinateRecord coreEdit = EditCoordinateRecord.make(
    authorNid,
    TinkarTerm.CORE_MODULE.nid(),  // New content goes to core
    TinkarTerm.CORE_MODULE.nid(),
    TinkarTerm.DEVELOPMENT_PATH.nid(),
    TinkarTerm.DEVELOPMENT_PATH.nid()
);

// Extension development
EditCoordinateRecord extEdit = EditCoordinateRecord.make(
    authorNid,
    extensionModuleNid,  // New content goes to extension
    coreModuleNid,       // Can modularize to core
    TinkarTerm.DEVELOPMENT_PATH.nid(),
    TinkarTerm.MASTER_PATH.nid()
);
```

Why? Default module controls where new content is created.

15.8.3. Configure Paths for Workflow

```
// Development workflow
EditCoordinateRecord devEdit = EditCoordinateRecord.make(
    authorNid,
    moduleNid,
    moduleNid,
    TinkarTerm.DEVELOPMENT_PATH.nid(),  // Create on dev
    TinkarTerm.DEVELOPMENT_PATH.nid()   // No promotion
);
```

```java
// Release workflow
EditCoordinateRecord releaseEdit = EditCoordinateRecord.make(
    authorNid,
    moduleNid,
    moduleNid,
    TinkarTerm.DEVELOPMENT_PATH.nid(),  // Create on dev
    TinkarTerm.MASTER_PATH.nid()        // Promote to master
);
```

Why? Path configuration supports development workflows.

15.9. View Coordinates

15.9.1. Use View Coordinates for Consistency

```java
// GOOD: Single view for all operations
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);

String description = calc.getDescriptionText(conceptNid);
int[] parents = calc.unsortedParentsOf(conceptNid);
Latest<ConceptVersion> latest = calc.latest(conceptNid);
// All operations use consistent context

// AVOID: Mixing different coordinate types
StampCalculator stampCalc = StampCalculatorWithCache.getCalculator(stamp1);
LanguageCalculator langCalc = LanguageCalculatorWithCache.getCalculator(
    stamp2,
    languageCoords
);
// May have inconsistent results
```

Why? View coordinates ensure consistency across all operations.

15.9.2. Create Views for Specific Use Cases

```java
public class Views {
    public static ViewCoordinateRecord forDevelopment() {
        // Development configuration
    }

    public static ViewCoordinateRecord forProduction() {
        // Production configuration
    }

    public static ViewCoordinateRecord forAudit() {
        // Audit configuration
    }
    public static ViewCoordinateRecord forUser(User user) {
        // User-specific configuration
    }
}
```

Why? Named views communicate intent and ensure correct configuration.

15.10. Performance

15.10.1. Reuse Calculators

```
// GOOD: Reuse calculator instance
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);

for (int conceptNid : conceptNids) {
    process(conceptNid, calc);
}

// BAD: Recreate calculator
for (int conceptNid : conceptNids) {
    ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);
    process(conceptNid, calc);
}
```

Why? Calculator creation and cache initialization have overhead.

15.10.2. Use Batch Operations

```
// GOOD: Process in batch with single calculator
public void processConceptsBatch(IntIdSet conceptNids, ViewCalculator calc) {
    conceptNids.forEach(nid -> {
        String desc = calc.getDescriptionText(nid);
        int[] parents = calc.unsortedParentsOf(nid);
        process(desc, parents);
    });
}

// AVOID: Creating new calculator for each concept
public void processConcepts(IntIdSet conceptNids) {
    conceptNids.forEach(nid -> {
        ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);
        String desc = calc.getDescriptionText(nid);
    });
}
```

Why? Batch operations amortize calculator overhead.

15.10.3. Leverage Built-in Caching

```
// Calculators with caching
ViewCalculator viewCalc = ViewCalculatorWithCache.getCalculator(view);
StampCalculator stampCalc = StampCalculatorWithCache.getCalculator(stamp);
LanguageCalculator langCalc = LanguageCalculatorWithCache.getCalculator(
    stamp,
    languageCoords
);

// Cache handles repeated queries efficiently
String desc1 = viewCalc.getDescriptionText(conceptNid);
String desc2 = viewCalc.getDescriptionText(conceptNid);  // Cached!
```

Why? Built-in caching dramatically improves repeated query performance.

15.11. Thread Safety

15.11.1. Coordinates Are Immutable and Thread-Safe

```
// GOOD: Share coordinates across threads
private final ViewCoordinateRecord view = Coordinates.View.DefaultView();

public void processInParallel(List<Integer> conceptNids) {
    conceptNids.parallelStream().forEach(nid -> {
        ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);
        process(nid, calc);
    });
}
```

Why? Immutable coordinates can be safely shared between threads.

15.11.2. Calculators Are Thread-Safe

```
// GOOD: Share calculator across threads
private final ViewCalculator calculator =
    ViewCalculatorWithCache.getCalculator(view);

public void processInParallel(List<Integer> conceptNids) {
    conceptNids.parallelStream().forEach(nid -> {
        String desc = calculator.getDescriptionText(nid);
        process(nid, desc);
    });
}
```

Why? Calculators are thread-safe and can be shared.

15.12. Testing

15.12.1. Create Test-Specific Coordinates

```
public class TestCoordinates {
    public static ViewCoordinateRecord allStatesAllModules() {
        return ViewCoordinateRecord.make(
            StampCoordinateRecord.make(
                StateSet.ACTIVE_AND_INACTIVE,
                StampPositionRecord.make(
                    Long.MAX_VALUE,
                    TinkarTerm.DEVELOPMENT_PATH.nid()
                )
            ),
            Coordinates.Language.UsEnglishRegularName(),
            Coordinates.Logic.ElPlusPlus(),
            Coordinates.Navigation.inferred(),
            Coordinates.Edit.Default()
        );
    }
}

@Test
void testInactiveConceptHandling() {
    ViewCalculator calc = ViewCalculatorWithCache.getCalculator(
        TestCoordinates.allStatesAllModules()
    );
    // Test can see inactive concepts
}
```

Why? Test-specific coordinates enable comprehensive testing.

15.12.2. Test with Multiple Coordinate Configurations

```
@ParameterizedTest
@MethodSource("coordinateProvider")
void testWithDifferentCoordinates(ViewCoordinateRecord view) {
    ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);
    // Test behavior with different coordinate configurations
}

static Stream<ViewCoordinateRecord> coordinateProvider() {
    return Stream.of(
        Coordinates.View.DefaultView(),
        TestCoordinates.allStatesAllModules(),
        createProductionView()
    );
}
```

Why? Testing with multiple configurations catches coordinate-specific issues.

15.13. Summary

Key best practices:

- **Creation**: Use predefined coordinates; create reusable factories
- **Calculators**: Cache and reuse; don't mix contexts
- **STAMP**: Choose appropriate state, time, and module filters
- **Language**: Provide fallback; order types appropriately
- **Logic**: Match premise type to use case
- **Navigation**: Match pattern to task; use appropriate sorting
- **Edit**: Per-user coordinates; match module to context
- **View**: Use for consistency; create named views
- **Performance**: Reuse calculators; use batch operations; leverage caching
- **Thread Safety**: Share immutable coordinates and thread-safe calculators
- **Testing**: Create test-specific coordinates; test multiple configurations

Following these practices will help you build robust, performant Tinkar applications.

16. Quick Reference

This document describes the IKE Coordinate System (Integrated Knowledge Ecosystem), based on the HL7 TINKAR specification. Technical references use "Tinkar" in code for backward compatibility.

16.1. Overview

This reference provides quick lookup for common coordinate operations, factory methods, and key interfaces.

16.2. Coordinate Factory Methods

16.2.1. STAMP Coordinates

```
// Predefined STAMP coordinates
Coordinates.Stamp.DevelopmentLatest()
Coordinates.Stamp.DevelopmentLatestActiveOnly()
```

```
// Custom STAMP coordinate
StampCoordinateRecord.make(
    StateSet allowedStates,
    StampPositionRecord position
)

// STAMP position
StampPositionRecord.make(long time, int pathNid)

// State sets
StateSet.ACTIVE_ONLY
StateSet.ACTIVE_AND_INACTIVE
StateSet.INACTIVE_ONLY
```

16.2.2. Language Coordinates

```
// Predefined language coordinates
Coordinates.Language.UsEnglishRegularName()
Coordinates.Language.UsEnglishFullyQualifiedName()
Coordinates.Language.GbEnglishRegularName()
Coordinates.Language.SpanishLanguage()
Coordinates.Language.LanguageAgnostic()

// Custom language coordinate
LanguageCoordinateRecord.make(
    int languageConceptNid,
    List<Integer> dialectPatternNids,
    List<Integer> descriptionTypePreferenceNids
)
```

16.2.3. Logic Coordinates

```
// Predefined logic coordinates
Coordinates.Logic.ElPlusPlus()

// Custom logic coordinate
LogicCoordinateRecord.make(
    int statedAxiomsPatternNid,
    int inferredAxiomsPatternNid,
    int classifierNid,
    int inferredPremiseTypeNid
)
```

16.2.4. Navigation Coordinates

```
// Predefined navigation coordinates
```

```
Coordinates.Navigation.inferred()
Coordinates.Navigation.stated()

// Custom navigation coordinate
NavigationCoordinateRecord.make(
    IntIdSet navigationPatternNids,
    StateSet vertexStates,
    boolean sortVertices,
    IntIdList verticesSortPatternNidList
)
```

16.2.5. Edit Coordinates

```
// Predefined edit coordinates
Coordinates.Edit.Default()

// Custom edit coordinate
EditCoordinateRecord.make(
    int authorNid,
    int defaultModuleNid,
    int destinationModuleNid,
    int defaultPathNid,
    int promotionPathNid
)
```

16.2.6. View Coordinates

```
// Predefined view coordinates
Coordinates.View.DefaultView()

// Custom view coordinate
ViewCoordinateRecord.make(
    StampCoordinate stampCoordinate,
    List<LanguageCoordinate> languageCoordinateList,
    LogicCoordinate logicCoordinate,
    NavigationCoordinate navigationCoordinate,
    EditCoordinate editCoordinate
)
```

16.3. Calculator Creation

16.3.1. Obtaining Calculators

```
// STAMP calculator
StampCalculator stampCalc = StampCalculatorWithCache.getCalculator(
    stampCoordinate
);

// Language calculator
LanguageCalculator langCalc = LanguageCalculatorWithCache.getCalculator(
    stampCoordinate,
    languageCoordinateList
);

// Logic calculator
LogicCalculator logicCalc = LogicCalculatorWithCache.getCalculator(
    stampCoordinate,
    logicCoordinate
);

// Navigation calculator
NavigationCalculator navCalc = NavigationCalculatorWithCache.getCalculator(
    stampCoordinate,
    navigationCoordinate
);

// View calculator (unified)
ViewCalculator viewCalc = ViewCalculatorWithCache.getCalculator(
    viewCoordinate
);
```

16.4. Common Operations

16.4.1. STAMP Operations

```
StampCalculator calc = StampCalculatorWithCache.getCalculator(stamp);

// Get latest version
Latest<ConceptVersion> latest = calc.latest(conceptNid);
Latest<SemanticVersion> latestSemantic = calc.latest(semanticNid);

// Check if version is latest
boolean isLatest = calc.isLatestActive(versionNid);

// Get specific version
Optional<ConceptVersion> version = calc.getConceptVersion(
    conceptNid,
    versionStampNid
);

// Get all versions
List<ConceptVersion> versions = calc.getAllVersions(conceptNid);
```

16.4.2. Language Operations

```
LanguageCalculator calc = LanguageCalculatorWithCache.getCalculator(
    stamp,
    languageCoords
);

// Get description text
String description = calc.getDescriptionText(conceptNid);

// Get fully qualified name
String fqn = calc.getFullyQualifiedDescriptionText(conceptNid);

// Get preferred description
Optional<String> preferred = calc.getPreferredDescriptionText(conceptNid);

// Get description with details
Optional<DescriptionVersion> descVersion =
    calc.getDescription(conceptNid);
```

16.4.3. Logic Operations

```
LogicCalculator calc = LogicCalculatorWithCache.getCalculator(
    stamp,
    logicCoord
);

// Get stated axioms
DiTreeEntity statedAxioms = calc.getStatedAxiomTree(conceptNid);

// Get inferred axioms
DiTreeEntity inferredAxioms = calc.getInferredAxiomTree(conceptNid);

// Get axiom DiTree
Optional<DiTreeEntity> axiomTree = calc.getAxiomTreeForEntity(
    conceptNid,
    logicCoord.statedAxiomsPatternNid()
);
```

16.4.4. Navigation Operations

```
NavigationCalculator calc = NavigationCalculatorWithCache.getCalculator(
    stamp,
    navCoord
);

// Get parents
```

```
int[] parents = calc.unsortedParentsOf(conceptNid);
int[] sortedParents = calc.sortedParentsOf(conceptNid);

// Get children
int[] children = calc.unsortedChildrenOf(conceptNid);
int[] sortedChildren = calc.sortedChildrenOf(conceptNid);

// Get ancestors (transitive closure)
IntIdSet ancestors = calc.ancestorsOf(conceptNid);

// Get descendants (transitive closure)
IntIdSet descendants = calc.descendantsOf(conceptNid);

// Check relationships
boolean isChildOf = calc.isChildOf(childNid, parentNid);
boolean isDescendantOf = calc.isDescendantOf(descendantNid, ancestorNid);

// Get roots
IntIdSet roots = calc.roots();
```

16.4.5. View Operations (Unified)

```
ViewCalculator calc = ViewCalculatorWithCache.getCalculator(view);

// All STAMP operations
Latest<ConceptVersion> latest = calc.latest(conceptNid);

// All Language operations
String description = calc.getDescriptionText(conceptNid);

// All Logic operations
DiTreeEntity statedAxioms = calc.getStatedAxiomTree(conceptNid);

// All Navigation operations
int[] parents = calc.unsortedParentsOf(conceptNid);
IntIdSet ancestors = calc.ancestorsOf(conceptNid);

// Access to edit coordinate
EditCoordinate editCoord = calc.editCoordinate();
```

16.5. Coordinate Modification

16.5.1. STAMP Coordinate Modifications

```
StampCoordinateRecord modified = original
    .withAllowedStates(StateSet.ACTIVE_AND_INACTIVE)
    .withStampPosition(newPosition)
    .withModuleNids(moduleNidSet)
    .withExcludedModuleNids(excludedModuleNidSet)
    .withModulePriorityNidList(modulePriorityList)
    .withPath(pathConcept);
```

16.5.2. View Coordinate Modifications

```
// Modify individual coordinate
ViewCoordinateRecord modified = ViewCoordinateRecord.make(
    original.stampCoordinate().withAllowedStates(StateSet.ACTIVE_AND_INACTIVE),
    original.languageCoordinateList(),
    original.logicCoordinate(),
    original.navigationCoordinate(),
    original.editCoordinate()
);
```

16.6. Key Interfaces

16.6.1. StampCoordinate Interface

```
public interface StampCoordinate {
    StateSet allowedStates();
    StampPosition stampPosition();
    IntIdSet moduleNids();
    IntIdSet excludedModuleNids();
    IntIdList modulePriorityNidList();
    int pathNidForFilter();

    StampCoordinate withAllowedStates(StateSet states);
    StampCoordinate withStampPosition(StampPositionRecord position);
    StampCoordinate withModuleNids(IntIdSet moduleNids);
    StampCoordinate withExcludedModuleNids(IntIdSet excludedModuleNids);
    StampCoordinate withModulePriorityNidList(IntIdList priorityList);

    long time();
    String toUserString();
}
```

16.6.2. LanguageCoordinate Interface

```
public interface LanguageCoordinate {
    int languageConceptNid();
    IntIdList dialectPatternPreferenceNidList();
    IntIdList descriptionTypePreferenceNidList();

    String toUserString();
}
```

16.6.3. LogicCoordinate Interface

```
public interface LogicCoordinate {
    int statedAxiomsPatternNid();
    int inferredAxiomsPatternNid();
    int classifierNid();
    int inferredPremiseTypeNid();

    String toUserString();
}
```

16.6.4. NavigationCoordinate Interface

```
public interface NavigationCoordinate {
    IntIdSet navigationPatternNids();
    StateSet vertexStates();
    boolean sortVertices();
    IntIdList verticesSortPatternNidList();

    String toUserString();
}
```

16.6.5. EditCoordinate Interface

```
public interface EditCoordinate {
    int getAuthorNidForChanges();
    int getDefaultModuleNid();
    int getDestinationModuleNid();
    int getDefaultPathNid();
    int getPromotionPathNid();
}
```

16.6.6. ViewCoordinate Interface

```
public interface ViewCoordinate {
    StampCoordinate stampCoordinate();
    <T extends LanguageCoordinate> Iterable<T> languageCoordinateIterable();
    LogicCoordinate logicCoordinate();
    NavigationCoordinate navigationCoordinate();
    EditCoordinate editCoordinate();

    String toUserString();
}
```

16.7. Common TinkarTerm Constants

16.7.1. States

```
State.ACTIVE
State.INACTIVE
State.PRIMORDIAL
State.CANCELED
State.WITHDRAWN
```

16.7.2. Paths

```
TinkarTerm.DEVELOPMENT_PATH
TinkarTerm.MASTER_PATH
TinkarTerm.PRIMORDIAL_PATH
```

16.7.3. Modules

```
TinkarTerm.SOLOR_OVERLAY_MODULE
TinkarTerm.CORE_MODULE
TinkarTerm.EXTENSION_MODULE
```

16.7.4. Languages

```
TinkarTerm.ENGLISH_LANGUAGE
TinkarTerm.SPANISH_LANGUAGE
TinkarTerm.FRENCH_LANGUAGE
TinkarTerm.LANGUAGE_CONCEPT_NID_FOR_DESCRIPTION
```

16.7.5. Dialect Patterns

```
TinkarTerm.US_DIALECT_ASSEMBLAGE
TinkarTerm.GB_DIALECT_ASSEMBLAGE
TinkarTerm.SPANISH_DIALECT_ASSEMBLAGE
```

16.7.6. Description Types

```
TinkarTerm.REGULAR_NAME_DESCRIPTION_TYPE
TinkarTerm.FULLY_QUALIFIED_NAME_DESCRIPTION_TYPE
TinkarTerm.DEFINITION_DESCRIPTION_TYPE
```

16.7.7. Navigation Patterns

```
TinkarTerm.INFERRED_NAVIGATION
TinkarTerm.STATED_NAVIGATION
```

16.7.8. Logic Patterns

```
TinkarTerm.EL_PLUS_PLUS_STATED_AXIOMS_PATTERN
TinkarTerm.EL_PLUS_PLUS_INFERRED_AXIOMS_PATTERN
```

16.7.9. Classifiers

```
TinkarTerm.SNOROCKET_CLASSIFIER
```

16.7.10. Premise Types

```
TinkarTerm.STATED_PREMISE_TYPE
TinkarTerm.INFERRED_PREMISE_TYPE
```

16.7.11. Users

```
TinkarTerm.USER
```

16.8. Entity Operations

16.8.1. Getting Entities

```
// Get entity by NID
Entity entity = Entity.getFast(nid);
ConceptEntity concept = Entity.getConceptForNid(nid);
SemanticEntity semantic = Entity.getSemanticForNid(nid);
PatternEntity pattern = Entity.getPatternForNid(nid);
StampEntity stamp = Entity.getStampForNid(nid);

// Get entity by public ID
Entity entity = Entity.get(publicId);
```

16.8.2. Creating Versions

```
// Create STAMP
StampEntity stamp = StampRecord.build(
    state,
    time,
    authorNid,
    moduleNid,
    pathNid
);

// Create concept version
ConceptVersionRecord conceptVersion = ConceptVersionRecord.build(
    conceptNid,
    stampNid
);

// Create semantic version
SemanticVersionRecord semanticVersion = SemanticVersionRecord.build(
    semanticNid,
    stampNid,
    fieldValues
);

// Commit
Entity.provider().putEntity(stamp);
Entity.provider().putEntity(conceptVersion);
Entity.provider().putEntity(semanticVersion);
```

16.9. Working with Latest

16.9.1. Latest Pattern

```
Latest<ConceptVersion> latest = calculator.latest(conceptNid);

if (latest.isPresent()) {
    ConceptVersion version = latest.get();
    State state = version.state();
    int moduleNid = version.moduleNid();
    // ... use version
} else {
    // No version matches coordinate
}

// Or with ifPresent
latest.ifPresent(version -> {
    // Process version
});
```

```
// Or with orElse
ConceptVersion version = latest.orElse(defaultVersion);
```

16.9.2. Absence Reasons

```
Latest<ConceptVersion> latest = calculator.latest(conceptNid);

if (latest.isAbsent()) {
    System.out.println("No latest version because: " +
        latest.absenceReason());
}
```

16.10. IntId Collections

16.10.1. Creating IntId Collections

```
// IntIdSet
IntIdSet set = IntIds.set.empty();
IntIdSet set = IntIds.set.of(nid1, nid2, nid3);
IntIdSet set = IntIds.set.of(intArray);
IntIdSet set = IntIds.set.of(collection, Entity::nid);

// IntIdList
IntIdList list = IntIds.list.empty();
IntIdList list = IntIds.list.of(nid1, nid2, nid3);
IntIdList list = IntIds.list.of(intArray);
IntIdList list = IntIds.list.of(collection, Entity::nid);
```

16.10.2. Using IntId Collections

```
IntIdSet set = IntIds.set.of(nid1, nid2);

// Add/remove
IntIdSet modified = set.with(nid3);
IntIdSet modified = set.without(nid2);

// Operations
IntIdSet union = set1.union(set2);
IntIdSet intersection = set1.intersect(set2);
IntIdSet difference = set1.difference(set2);

// Query
boolean contains = set.contains(nid);
int size = set.size();
boolean isEmpty = set.isEmpty();
```

```
// Iterate
set.forEach(nid -> process(nid));
int[] array = set.toArray();
```

16.11. Common Patterns

16.11.1. Building a Hierarchy

```
public TreeNode buildHierarchy(int rootNid, ViewCalculator calc) {
    String description = calc.getDescriptionText(rootNid);
    int[] children = calc.sortedChildrenOf(rootNid);

    List<TreeNode> childNodes = Arrays.stream(children)
        .mapToObj(childNid -> buildHierarchy(childNid, calc))
        .collect(Collectors.toList());

    return new TreeNode(rootNid, description, childNodes);
}
```

16.11.2. Getting All Ancestors

```
public IntIdSet getAllAncestors(int conceptNid, ViewCalculator calc) {
    return calc.ancestorsOf(conceptNid);
}
```

16.11.3. Finding Common Ancestor

```
public Optional<Integer> findCommonAncestor(
    int concept1Nid,
    int concept2Nid,
    ViewCalculator calc
) {
    IntIdSet ancestors1 = calc.ancestorsOf(concept1Nid);
    IntIdSet ancestors2 = calc.ancestorsOf(concept2Nid);

    IntIdSet common = ancestors1.intersect(ancestors2);

    if (common.isEmpty()) {
        return Optional.empty();
    }

    // Return closest common ancestor (implementation depends on needs)
    return Optional.of(common.intIterator().next());
}
```

16.11.4. Checking Subsumption

```
public boolean isKindOf(
    int specificNid,
    int generalNid,
    ViewCalculator calc
) {
    if (specificNid == generalNid) {
        return true;
    }

    IntIdSet ancestors = calc.ancestorsOf(specificNid);
    return ancestors.contains(generalNid);
}
```

16.11.5. Creating New Version

```
public void createNewVersion(
    int conceptNid,
    EditCoordinate editCoord
) {
    // Create STAMP
    StampEntity stamp = StampRecord.build(
        State.ACTIVE,
        System.currentTimeMillis(),
        editCoord.getAuthorNidForChanges(),
        editCoord.getDefaultModuleNid(),
        editCoord.getDefaultPathNid()
    );

    // Create version
    ConceptVersionRecord version = ConceptVersionRecord.build(
        conceptNid,
        stamp.nid()
    );

    // Commit
    Entity.provider().putEntity(stamp);
    Entity.provider().putEntity(version);
}
```

16.12. Debugging and Logging

16.12.1. User-Friendly Output

```
// All coordinates have toUserString()
```

```
String stampInfo = stampCoord.toUserString();
String langInfo = langCoord.toUserString();
String logicInfo = logicCoord.toUserString();
String navInfo = navCoord.toUserString();
String viewInfo = viewCoord.toUserString();

System.out.println("Current view:");
System.out.println(viewInfo);
```

16.12.2. Getting Text for NIDs

```
// Convert NID to description
String text = PrimitiveData.text(nid);

// Convert multiple NIDs to text list
String textList = PrimitiveData.textList(nidArray);
```

16.13. Package Structure

Package	Purpose
`dev.ikm.tinkar.coordinate`	Root package with Coordinates factory
`dev.ikm.tinkar.coordinate.stamp`	STAMP coordinates and records
`dev.ikm.tinkar.coordinate.stamp.calculator`	STAMP calculator implementations
`dev.ikm.tinkar.coordinate.stamp.change`	Change tracking for STAMP
`dev.ikm.tinkar.coordinate.language`	Language coordinates and records
`dev.ikm.tinkar.coordinate.language.calculator`	Language calculator implementations
`dev.ikm.tinkar.coordinate.logic`	Logic coordinates and records
`dev.ikm.tinkar.coordinate.logic.calculator`	Logic calculator implementations
`dev.ikm.tinkar.coordinate.navigation`	Navigation coordinates and records
`dev.ikm.tinkar.coordinate.navigation.calculator`	Navigation calculator implementations
`dev.ikm.tinkar.coordinate.edit`	Edit coordinates and records
`dev.ikm.tinkar.coordinate.view`	View coordinates combining all types
`dev.ikm.tinkar.coordinate.view.calculator`	View calculator implementations

16.14. Additional Resources

16.14.1. JavaDoc

Complete JavaDoc is available for all packages:

- `dev.ikm.tinkar.coordinate` - Package documentation
- Individual interface and class JavaDoc
- Method-level documentation with examples

16.14.2. Source Code

Reference implementations in:

- `Coordinates.java` - Factory methods
- `*Record.java` - Immutable record implementations
- `*WithCache.java` - Cached calculator implementations

16.14.3. Related Packages

- `dev.ikm.tinkar.entity` - Entity system
- `dev.ikm.tinkar.terms` - Standard terminology constants
- `dev.ikm.tinkar.common` - Common utilities

16.15. Summary

This reference provides quick access to:

- Coordinate factory methods
- Calculator creation
- Common operations
- Coordinate modification
- Key interfaces
- TinkarTerm constants
- Entity operations
- IntId collections
- Common patterns
- Package structure

For detailed explanations and examples, refer to the main guide sections.