Language Transformation Analysis: Java to Python Conversion of GraphBrain Domain Package

Cirilli Davide[1] and Fontana Emanuele[2]

[1,2]Department of Computer Science, Università degli Studi di Bari

April 10, 2025

Contents

1 Introduction

This assignment focuses on the implementation of a parser for the Java package *domain* of GraphBrain, recreating its functionality in Python. The domain package constitutes the core model of GraphBrain, containing classes that represent Entities and Relationships, as well as functionality to extract these from *.gbs* files. This language transformation exercise required careful consideration of both language-specific features and object-oriented design principles to ensure functional equivalence between the original Java implementation and the resulting Python code.

The primary objectives of this work were to:

- Accurately translate the class hierarchy and inheritance relationships
- Preserve method functionality and interface contracts
- Adapt Java-specific constructs to idiomatic Python patterns
- Maintain the original semantic model of the domain

2 Class Transformations

2.1 Attachment Class

The transformation of the `Attachment` class from Java to Python required mapping private fields to public attributes while maintaining the intended encapsulation through method interfaces. The Java class's private fields `progr`, `extension`, `description`, and `fileName` were represented as public attributes in Python, following Python's convention of relying on naming conventions rather than access modifiers.

The constructor logic was preserved in the Python `__init__` method, initializing all attributes with their corresponding parameters. Additionally, getter methods were implemented with identical names and functionality, with particular attention to the `getFilename()` method, which replicates the string concatenation logic from the original Java implementation.

2.2 Tag Abstract Class

The abstract `Tag` class presented an interesting challenge in the transformation process. Java's abstract class concept was mapped to Python using the `abc.ABC` base class to enforce abstractness. The protected fields `name`, `description`, and `notes` from the Java implementation were represented as public attributes in Python, initialized to `None` in the constructor.

All getter and setter methods were faithfully recreated in Python, maintaining the same method signatures and functionality. This approach preserves the original API contract while adapting to Python's conventions regarding attribute visibility.

2.3 DomainTag Abstract Class

Building upon the `Tag` class, the abstract `DomainTag` class extends the base functionality with domain-specific features. In the Python implementation, `DomainTag` inherits from both `Tag` and `abc.ABC`, preserving the inheritance hierarchy and abstract nature of the class.

The protected field `domain` from Java was mapped to the attribute `_domain` in Python, employing the underscore prefix as a conventional indication of protected status. Corresponding getter and setter methods were implemented to maintain the original interface while adapting to Python's attribute access patterns.

2.4 Attribute Class

The `Attribute` class represents one of the more complex transformations due to its numerous fields, multiple constructors, and diverse methods. Inheriting from `Tag` in both languages, the Python implementation preserves all public fields (`mandatory`, `distinguishing`, `display`) as public attributes with equivalent names.

The Java class's private collection field `values` (of type `List<String>`) was implemented in Python as a typed list (`List[str]`), leveraging Python's type hints for improved code clarity. Similarly, the `dataType` field was mapped to `data_type` with an `Optional[str]` type hint, acknowledging that this field might be uninitialized in some contexts.

Java's overloaded constructors were consolidated into a single `__init__` method in Python, utilizing optional parameters and type checking to replicate the functionality of the multiple Java constructors. This approach maintains the flexibility of the original design while adapting to Python's constructor paradigm.

The comprehensive set of methods in the Java class, including getters, setters, and utility functions, were faithfully recreated in Python with equivalent functionality. Special attention was given to the `clone()` method, which replicates the deep copying behavior of the Java original.

2.5 Author Class

The `Author` class transformation demonstrates the adaptation of naming conventions between Java and Python. The private fields in Java were mapped to public attributes in Python, with camelCase identifiers converted to snake_case where appropriate (e.g., `attributeKey` to `attribute_key`), following Python's style guidelines.

A notable aspect of this transformation was the handling of Java's `Timestamp` type for the `creationDate` field, which was mapped to Python's `datetime` type, providing equivalent functionality while using Python's standard library.

The Java class's implicit default constructor was represented by an `__init__` method in Python that initializes all attributes to `None`, preserving the original initialization behavior. Getter and setter methods were implemented with identical names, maintaining the original API contract.

2.6 Axiom Class

The `Axiom` class, which extends `DomainTag`, demonstrates the propagation of inheritance patterns across languages. In Python, the class inherits from `DomainTag`, preserving the original class hierarchy.

Private fields `formalism` and `expression` were mapped to conventionally private attributes `_formalism` and `_expression` in Python, utilizing the underscore prefix naming convention. The constructor logic was preserved in the `__init__` method, which calls the superclass constructor before initializing the class's specific attributes.

Particular attention was given to the implementation of Java's `equals()` and `hashCode()` methods, which were mapped to Python's special methods `__eq__()` and `__hash__()`. These methods maintain the original equality and hashing behavior based on the `name` attribute, ensuring that collections and comparison operations behave consistently across both languages.

2.7 UType Class

The `UType` class presents an interesting case of inheritance without additional fields or methods. In both Java and Python implementations, `UType` inherits directly from `Attribute` without extending the functionality, effectively serving as a specialized type marker.

This transformation demonstrates the concept of preserving class structure even when the derived class does not add functional elements, an important aspect of maintaining the semantic model of the domain.

2.8 HallUser and HallComparator Classes

The transformation of `HallUser` and `HallComparator` illustrates the adaptation of Java's comparator pattern to Python's comparison protocol. The `HallUser` class was implemented in Python with equivalent fields and methods, preserving the original data structure and functionality.

The Java `HallComparator` class, which implements the `Comparator<HallUser>` interface to define comparison logic, was transformed by integrating its functionality directly into the Python `HallUser` class through the special methods `__lt__()` and `__eq__()`. This approach leverages Python's rich comparison protocol, allowing instances to be naturally sorted according to the original comparison rules (descending order by `usageStatistic`, then by `trustIndex`).

This transformation demonstrates how Java's separate comparator pattern can be elegantly mapped to Python's object-oriented comparison protocol, eliminating the need for a distinct comparator class while preserving the original sorting behavior.

2.9 Instance Class

The `Instance` class transformation involved mapping complex data structures and comparison logic from Java to Python. Private fields `type`, `selectedInstanceId`, `attributeValues` (of type `Map<String,String>`), and `shortDescription` were implemented as public attributes in Python, with `attributeValues` specifically mapped to a typed dictionary (`Dict[str, str]`).

The constructor logic, which involves building a short description based on attribute values, was faithfully recreated in the Python `__init__` method. The private Java method `buildShortDescription` was implemented as a public method in Python, reflecting Python's more relaxed approach to method visibility while preserving the original functionality.

The implementation of Java's `equals()` method as Python's `__eq__()` ensures that instance equality is determined by the `selectedInstanceId` attribute, as in the original Java code, maintaining consistent behavior in collections and comparison operations.

2.10 Reference Class

The final class examined, `Reference`, demonstrates the adaptation of Java's collection types to their Python equivalents. The private field `attributes` of type `Vector<Attribute>` in Java was mapped to `Optional[List[Attribute]]` in Python, acknowledging both the change in collection type and the possibility of null values.

Java's overloaded constructors were consolidated into a single `__init__` method with an optional parameter for `attributes`, defaulting to `None`. This approach preserves the flexibility of the original design while adapting to Python's constructor paradigm.

The getter and setter methods were implemented with identical names, maintaining the original API contract, and the `toString()` method was mapped to `__str__()`, providing a consistent string representation across both languages.

2.11 Entity Class

The `Entity` class, which extends `DomainTag`, represents one of the central elements in GraphBrain's domain model. Its transformation from Java to Python required particular attention due to the class's complexity, characterized by numerous fields, methods, and hierarchical relationships.

In Python, the class maintains the same inheritance relationship, extending `DomainTag`. Java's private fields (`values`, `graphBrainID`, `attributes`, `children`, `parent`, `_abstract`) were mapped to attributes with underscore prefixes in Python, following the convention for indicating protected or private attributes.

A significant aspect of the transformation was adapting the type system: Python type annotations were utilized to improve code readability and maintainability. For example:

```
from typing import List, Optional, TYPE_CHECKING

self._values: List[str] = []
self._parent: Optional[Entity] = None
```

Managing hierarchical relationships between entities required special attention. Methods such as `getAllAttributes()`, `getClassPath()`, and `getSubclassesTree()` were implemented maintaining the same recursive logic as the Java original, but adapted to Python conventions.

Another challenge was implementing the entity comparison system: Java's `equals()` method was mapped to Python's `__eq__()` special method, while `toString()` was mapped to `__str__()`. This approach ensures that comparison and string representation operations work consistently across both languages.

The hierarchy manipulation methods, including `addChild()`, `detach()`, and `removeAllAttributes()`, were implemented with the same behavior as the original, preserving the consistency of entity relationships during structural modification operations.

The transformation of the `Entity` class demonstrates the importance of deep understanding of both source and target languages, as differences in programming paradigms require informed decisions to maintain the original semantics while adopting the conventions of the new language.

3 Conclusion

The transformation of GraphBrain's domain package from Java to Python required a systematic approach to mapping classes, fields, methods, and inheritance relationships while preserving the original semantic model.