

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

Cirilli Davide<sup>1</sup> and Fontana Emanuele<sup>2</sup>

<sup>1,2</sup>Department of Computer Science, Università degli Studi di Bari

April 11, 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Class Transformations</b>	<b>3</b>
2.1	Attachment Class . . . . .	3
2.2	Tag Abstract Class . . . . .	3
2.3	DomainTag Abstract Class . . . . .	4
2.4	Attribute Class . . . . .	4
2.5	Author Class . . . . .	5
2.6	Axiom Class . . . . .	5
2.7	UType Class . . . . .	5
2.8	HallUser and HallComparator Classes . . . . .	5
2.9	Instance Class . . . . .	6
2.10	Reference Class . . . . .	6
2.11	Entity Class . . . . .	6
2.12	Relationship Class . . . . .	7
2.13	RelationshipSite Class . . . . .	8
2.14	RelationTriple Class . . . . .	9
2.15	Union Class . . . . .	9
2.16	DomainData Class . . . . .	9
2.17	RecordData Class . . . . .	11
2.18	TranslatorAPIProlog Class . . . . .	13

<b>3</b>	<b>Additional classes</b>	<b>16</b>
3.1	TreeNode Class . . . . .	16
3.2	DefaultTreeNode Class . . . . .	17

# 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

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 private fields in Java were mapped to public attributes in Python.

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 inherits from `DomainTag`, preserving the original class hierarchy. Attributes of the Java class were mapped to Python attributes with the same type and name.

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.

## 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 meth-

ods `__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`).

## 2.9 Instance Class

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 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.

## 2.12 Relationship Class

The `Relationship` class, which extends `Entity`, presented several interesting challenges during the transformation from Java to Python. This class represents connections between entities, forming a central part of GraphBrain’s semantic network model.

In the Python implementation, the inheritance relationship with `Entity` was preserved, while the specialized fields for relationship management were adapted appropriately. The private Java fields `inverse`, `references`, `children`,

`parent`, and `symmetric` were mapped to attributes with underscore prefixes in Python, following the convention for protected status.

A significant aspect of the transformation was managing the multiple constructors in Java. These were consolidated into a single `__init__` method in Python using optional parameters:

```
def __init__(self, name: str, domain: Optional[str] = None, inverse: Optional[str] = None,
             parent: Optional[Relationship] = None, symmetric: bool = False,
             attributes: Optional[List[Attribute]] = None):
    super().__init__(name, domain)
    self._inverse: Optional[str] = inverse
    self._references: List[Reference] = []
    self._parent: Optional[Relationship] = None
    self._symmetric: bool = symmetric
```

The transformation required careful type hinting to address potential circular import issues, employing the `TYPE_CHECKING` flag from the typing module to avoid runtime import errors while maintaining code clarity.

Managing the hierarchical relationship structure was particularly challenging. Methods like `setParentRelationship()`, `addChildrenRelationship()`, and `isTopRelationship()` were implemented with special attention to maintaining proper bidirectional parent-child references, which required overriding some `Entity` behaviors.

References management methods, including `addReference()`, `getReference()`, and relationship operations like `getSubj_Objjs()` and `getObj_Subjs()`, were implemented with the same functionality as the original. The Python implementation employed set comprehensions and list operations to provide equivalent collection handling to the Java original.

The transformation of the `Relationship` class demonstrates the complexities involved when mapping classes with both inheritance and composition relationships, requiring careful consideration of type safety and object relationship integrity across the language boundary.

## 2.13 RelationshipSite Class

In the Python implementation, the Java private fields `name`, `inverse`, `symmetric`, `attributes`, and `relationships` were mapped to public attributes, following Python's convention of relying on naming conventions rather than access modifiers. The constructor logic was preserved in the `__init__` method, initializing all attributes with their corresponding parameters.

The relationship retrieval methods (`getSubjects()`, `getObjects()`, `getSubj_Objjs()`, and `getObj_Subjs()`) were implemented with equivalent functionality as



their Java counterparts. The Python implementation maintained the same sorting logic for collections and list operations to provide consistent behavior.

## 2.14 RelationTriple Class

The `RelationTriple` class represents a fundamental semantic structure in GraphBrain's knowledge representation - a triple consisting of subject, relation, and object instances. This straightforward class was transformed from Java to Python with additions that enhance code quality and maintainability.

In the Python implementation, the Java private fields `subject`, `relation`, and `object` were mapped to public attributes, as is conventional in Python.

## 2.15 Union Class

The `Union` class represents a specialized domain tag that contains a set of values. This class extends `DomainTag` in both Java and Python implementations, preserving the inheritance hierarchy of the original model.

In the Python implementation, the private field `values` from Java was mapped to a protected attribute `_values` in Python, using the underscore prefix to indicate protected status according to Python conventions. Both implementations use a set data structure.

Java's `equals()` and `hashCode()` methods were transformed into Python's special methods `__eq__()` and `__hash__()`, maintaining the same comparison logic based on the `name` attribute. This ensures consistent behavior in collections and equality operations:

## 2.16 DomainData Class

The transformation of the `DomainData` class represents one of the most complex mapping challenges due to its central role in the GraphBrain domain model and extensive functionality. This class is responsible for loading and parsing *.gbs* files, as well as storing and manipulating domain information including entities, relationships, attributes, unions, and axioms.

In transforming this critical class from Java to Python, careful attention was given to preserving its extensive data structures and processing logic while adapting to Python's idioms and type system. The class required considerable refactoring to maintain functional equivalence while leveraging Python's strengths.

The complex field structure of the Java implementation was mapped to typed attributes in Python, with special attention to collection types:

```
# Java: private Vector<Attribute> types = new Vector<>();
types: List[Attribute]

# Java: private HashSet<Union> unions = new HashSet<>();
unions: Set[Union]

# Java: private Map<String,Vector<String>> subjRels = new HashMap<>();
subjRels: Dict[str, List[str]]
```

Java's overloaded constructors were consolidated into a single, flexible `__init__` method in Python that mimics all the original constructor variants through optional parameters and type checking:

```
def __init__(self,
              path_or_bytearray: str | bytes | None = None,
              webInfFolder: str | None = None,
              domainName: str | None = None,
              file: str | Path | None = None):
    """
    Initializes the DomainData object. Mimics Java overloaded constructors.
    """
```

One of the most significant challenges was adapting Java's XML parsing logic, which uses DOM (Document Object Model), to Python's `ElementTree` approach. The Python implementation carefully reconstructs the validation and traversal logic:

```
def validateTag(self, tag: ET.Element, validTags: List[str]) -> None:
    """
    Validates a tag against a list of valid tags.
    Raises ValueError if the tag is not found in the list of valid tags.
    """
    if tag.tag not in validTags:
        raise ValueError(
            f"Invalid tag <{tag.tag}> found where one of {validTags} was expected"
        )
```

The complex recursive parsing methods, such as `parseEntities`, `parseRelationships`, and `parseReferences`, were transformed with particular attention to maintaining exact equivalence in tree construction and inheritance logic. These methods handle the hierarchical relationships between domain elements:

```
def parseEntities(self, parentNode: ET.Element, root: Entity, domainName: str) -
    """
    Recursively parses entities from the given parent node and adds them to the
    """
```

Python's type hints were extensively used throughout the class to improve code clarity and maintainability, a significant enhancement over the Java version. This is particularly evident in helper methods:

```
def addValue(self, map_dict: Dict[str, List[str]], key: str, value: str) -> None
    """
    Adds a value to a list within a dictionary, ensuring no duplicates in the li
    Sorts the list.
    """
```

A notable enhancement in the Python implementation is improved error handling, with more specific error messages and graceful fallbacks when operations can't be completed:

```
except AttributeError:
    print(f"Warning: Relationship class missing 'getSubjects'. Cannot check "
          f"subject involvement for '{entity_name}' in '{rel.getName()}'.")
```

## 2.17 RecordData Class

The transformation of the `RecordData` class represents one of the most comprehensive examples of language mapping in the project, as this class serves as a central data repository for domain information. The class is responsible for loading, parsing, and managing domain metadata including entities, relationships, attributes, axioms, and relationship triples.

In the Python implementation, the extensive field structure of the Java class was carefully mapped using Python's type annotations to preserve both type safety and semantic meaning:

```
# Java: private Vector<String> subjects = new Vector<String>();
subjects: List[str]

# Java: private HashSet<Union> unions = new HashSet<>();
unions: Set[Union]

# Java: private Map<String,Vector<String>> subjRel_Objs = new HashMap<>();
subjRel_Objs: Dict[str, List[str]]
```

A notable improvement in the Python implementation was the use of more appropriate Python data structures. For instance, Java's `HashMap` with vectors was transformed into Python's `defaultdict(list)`, which automatically initializes new keys with empty lists:

```
# Initialize collections with defaultdict for automatic creation of empty lists
self.subjRels = defaultdict(list)
self.relSubjs = defaultdict(list)
self.objRels = defaultdict(list)
```

The Java class's complex XML parsing logic, which used the DOM API, was reimplemented using Python's `ElementTree`, requiring significant adaptation of the traversal and validation mechanisms:

```
def _parseFile(self, file_path: Path) -> ET.ElementTree:
    """Parses the given file path and returns an ElementTree object."""
    try:
        tree = ET.parse(file_path)
        return tree
    except ET.ParseError as e:
        print(f"XML Parse Error in file {file_path}: {e}")
        raise
```

Java's method overloading was consolidated into single methods with optional parameters, leveraging Python's more flexible parameter handling:

```
def __init__(self,
              path_or_bytearray: Optional[Union[str, bytes]] = None,
              webInfFolder: Optional[str] = None,
              domainName: Optional[str] = None,
              file: Optional[Union[str, Path]] = None):
    """
    Initializes the RecordData object. Mimics Java overloaded constructors.
    """
```

The recursive traversal methods for building entity and relationship hierarchies were reimplemented with particular attention to maintaining the same inheritance and composition relationships, while adapting to Python's iteration patterns:

```
def _parseEntities(self, parentNode: ET.Element, root: Entity, domainName: str)
    """
    Recursively parses entities from the given parent node and adds them to the
    """
```

Error handling was significantly enhanced in the Python implementation, providing more specific exception types and descriptive error messages:

```
try:
    domain_name_attr = root_element.get("name")
    if domain_name_attr is None:
        raise ValueError("Domain tag missing required name attribute")
except ValueError as e:
    print(f"Error validating domain structure: {e}")
    raise
```

The Python version also includes thorough documentation through doc-strings, which weren't present in the original Java code:

```
def _addValue(self, map_dict: Dict[str, List[str]], key: str, value: str) -> Non
"""
    Adds a value to a list within a dictionary, ensuring no duplicates in the li
    Sorts the list after addition.

    Args:
        map_dict: Dictionary mapping strings to lists of strings
        key: Dictionary key to add the value to
        value: String value to add to the list
"""
```

Collection operations that used Java's sorting and filtering methods were reimplemented using Python's list comprehensions and set operations, providing more concise and readable code while maintaining the same functionality:

```
def getSubjsFromRel(self, relationship_name: str) -> Set[str]:
    """Returns the set of subjects participating in the given relationship."""
    try:
        return set(self.relSubjs.get(relationship_name, []))
    except Exception as e:
        print(f"Warning: Error getting subjects for '{relationship_name}': {e}")
        return set()
```

## 2.18 TranslatorAPIProlog Class

The transformation of the `TranslatorAPIProlog` class represents an interesting case study in adapting specialized domain-specific functionality from

Java to Python. This utility class, responsible for converting domain objects into Prolog facts, demonstrates the language-agnostic nature of knowledge representation transformations.

In the Python implementation, the Java class's private fields (`attributes`, `attributesRel`, `domainName`, and `recur`) were mapped to class attributes with appropriate type annotations:

```
serialVersionUID: int = 1 # Placeholder for Java's serialVersionUID
attributes: Optional[Dict[str, List[Attribute]]] = None
attributesRel: Optional[Dict[str, List[Attribute]]] = None
domainName: Optional[str] = None
recur: str = ""
```

The Java constructor was transformed into a Python `__init__` method, maintaining the same functionality of generating Prolog facts from a `DomainData` object. However, the Python implementation adds stronger error checking:

```
def __init__(self, domain: DomainData):
    """
    Initializes the TranslatorAPIProlog instance and generates Prolog facts.
    Mirrors the Java constructor.
    """
    if domain.getDomain() is None:
        raise ValueError("Domain name cannot be None")
    # ...
```

A notable adaptation was the transformation of Java's static methods to Python's `@staticmethod` decorated functions. This preserved the original design pattern while adapting to Python's syntax:

```
@staticmethod
def writeFactsWithId(content: str, id_prefix: str) -> str:
    """
    Adds a unique ID to each fact string.
    Mirrors the Java static method.
    """
    # Implementation
```

The core string processing methods (`createEntities`, `writeEntity`, and `createRelationships`) underwent significant refactoring to adapt Java's string concatenation approach to Python's formatting capabilities, while maintaining identical output formats:

```
# Java: values += "entity(" + entity.getDomain().toLowerCase() + ", " +
#           entity.getName().toLowerCase() + ").\n";
```

```
# Python:
```

```
values += f"entity({domain_lower}, {name_lower}).\n"
```

The Python implementation adds comprehensive error handling and null-safety throughout the string generation process:

```
domain_lower = entity.getDomain().lower() if entity.getDomain() else "unknown_domain"
name_lower = entity.getName().lower() if entity.getName() else "unknown_entity"
```

Methods working with tree structures (`getClassi`, `recursiveTree`, and `recursiveTreeTax`) were preserved but required adaptation to Python's different approach to collections and iteration:

```
def recursiveTree(self, node: 'TreeNode') -> List[str]:
    """
    Recursively collects the string representation of all nodes in a subtree.
    Requires a TreeNode implementation.
    """
    classi: List[str] = []
    children = node.getChildren()
    if children:
        for child_node in children:
            classi.append(str(child_node))
            classi.extend(self.recursiveTree(child_node))
    return classi
```

The private helper method `createList` in Java was transformed into a Python private method using the underscore prefix convention (`_createList`), maintaining Python's naming conventions while preserving the method's utility function status.

A significant improvement in the Python implementation is the addition of comprehensive docstrings, which enhance code readability and provide clear documentation for each method:

```
def getDomainName(self) -> Optional[str]:
    """Returns the domain name for this translator."""
    return self.domainName
```

## 3 Additional classes

*TreeNode* and *DefaultTreeNode* classes don't have a direct equivalent in the Java code, so we created them in Python from scratch.

### 3.1 **TreeNode Class**

The **TreeNode** class was created from scratch in Python, inspired by `org.primefaces.model.Tree` from the Java PrimeFaces library. Since GraphBrain's Java implementation utilized this external library class without modifications, we needed to implement an equivalent functionality in our Python conversion.

Our Python implementation maintains all the essential functionality of the original Java class, including methods for managing parent-child relationships, node data, and display state. The class provides a comprehensive API that includes:

- Data management: `getData()`, `setData()`
- Parent-child relationship management: `getParent()`, `setParent()`, `getChildren()`, `addChild()`, `removeChild()`
- State management: `isExpanded()`, `setExpanded()`, `isSelected()`, `setSelected()`
- Tree navigation: `getChild_count()`, `isLeaf()`
- Type information: `getType()`, `setType()`

Additionally, the Python implementation includes Pythonic enhancements that weren't present in the original Java class:

- String representation methods (`__str__` and `__repr__`) for easier debugging
- Bidirectional relationship management in `setParent()` to ensure consistency between parent and child nodes
- A `clearParent()` helper method to simplify relationship management



## 3.2 DefaultTreeNode Class

The `DefaultTreeNode` class extends `TreeNode` and provides a concrete implementation with enhanced constructor flexibility. This class was created to mirror `org.primefaces.model.DefaultTreeNode` from the Java PrimeFaces library, which was used in the original GraphBrain implementation.

- `DefaultTreeNode()` - Creates an empty node
- `DefaultTreeNode(data)` - Creates a node with specified data
- `DefaultTreeNode(data, parent)` - Creates a node with specified data and parent
- `DefaultTreeNode(type, data, parent)` - Creates a node with specified type, data, and parent

The constructor implementation uses Python's optional parameters and type checking to determine which Java constructor variant is being emulated:

```
def __init__(self, type_or_data=None, data_or_parent=None, parent=None):
    """
    Initializes a DefaultTreeNode. Mimics PrimeFaces constructors.
    """
    # Logic to determine which constructor variant to emulate
```

The class also maintains automatic parent-child relationship management, ensuring that when a node is created with a parent, it is automatically added to that parent's children collection - preserving the behavior of the original Java implementation.