Gene Ontology Analysis System

Yelizaveta Manko, Anna Maria Bobita, Ingyin Aung

Advanced Programming

Table of Contents

I. Introduction

Executive Summary

The Gene Ontology Analysis System is a modular, object-oriented software solution designed to parse, model, and analyze complex biological data. With the exponential growth of genomic data, researchers require robust tools to interpret the functions of genes and gene products. This project addresses that need by providing a programmatic interface to the Gene Ontology (GO)—a directed acyclic graph (DAG) representing biological knowledge—and the Human Gene Annotation File (GAF), which links specific human genes to these ontological terms.

The system is built using Python, leveraging Pandas for high-performance data ingestion and NumPy for numerical matrix operations. The core architecture strictly adheres to Object-Oriented Programming (OOP) principles—specifically Encapsulation, Inheritance, Abstraction, and Polymorphism—to ensure the software is extensible, maintainable, and reusable.

Project Specification & Scope

The project operates on two primary datasets provided by the Gene Ontology Consortium:

- **Gene Ontology (OBO)**: A controlled vocabulary describing gene product characteristics across three namespaces: Biological Process, Molecular Function, and Cellular Component. The data is structured hierarchically, where terms are nodes and relationships (e.g., is_a) are edges, modeled internally as a GOgraph.

- **Gene Annotations (GAF)**: A dataset associating human gene products with GO terms, supported by various evidence codes (e.g., Experimental, Computational).

System Objectives

The primary goal is to transform these static files into dynamic, queryable software objects. The system is specified to deliver:

- **Parsing Layer**: Efficient extraction of data from .obo and .gaf files into structured Python objects and Pandas DataFrames.
- **OOP Core**: A rich class hierarchy representing GOterm entities, Annotation types, and the GOgraph structure.
- **Analytical Engine**: A suite of tools for calculating semantic similarity, analyzing graph neighborhoods (parents/children/siblings), and generating statistical summaries using matrix algebra.
- **User Interface**: A web-based platform built with Flask for interactive data exploration and visualization.

II. Software Structure

The software was designed as a modular and extensible software system for exploring, modelling, and analysing the Gene Ontology and human gene annotations, and to satisfy the requirements of modularity, extensibility, reusability, and interoperability.

The system is organised into five main layers: **parsing**, **ontology representation**, **analysis** (comparative), **system integration** and **user interface**. Each layer interacts with the others through clearly defined interfaces.

The project structure is organised as follows:

- Parsing (parser.py)
- Ontology (OOP/ontology.py, OOP/annotations.py)
- Numerical analysis (OOP/analysis.py)
- Analytical modules (comparative/neighborhood.py, comparative/similarity.py, comparative/statistics.py)
- System integration (main.py)
- User interface (app.py, templates/)

This separation ensures that changes in one part of the system do not propagate unnecessarily to others, improving maintainability and enabling future extensions.

Parsing

The parsing layer is responsible exclusively for reading and interpreting the input datasets provided by the Gene Ontology Consortium. The **parser.py** module handles both the Gene Ontology file (OBO format) and the Gene Annotation File (GAF format).

The parsing layer improves interoperability, as alternative parsers or additional data sources could be integrated without modifying the rest of the system.

Ontology

The core conceptual entities of the system are implemented in the domain model layer, located in the **OOP** directory. This layer captures the semantics of the Gene Ontology and gene annotations using object-oriented programming concepts.

By isolating the ontology model from the other layers, the system ensures reusability: the same ontology and annotation representations can be reused.

Analytical Layer

Analytical functionality is grouped in the **comparative** directory, which contains modules for neighbourhood exploration, similarity computation, and statistical summarisation. It is intentionally separated from the core domain model to maintain a clear distinction between data representation and data analysis.

This modular organisation allows new analytical methods to be added without modifying existing classes, supporting extensibility and experimentation.

Numerical Analysis

Numerical representations of the data are handled by the AnnotationMatrixBuilder class in the **OOP/analysis.py** module. This component transforms annotation data into matrix-based representations using NumPy, enabling quantitative analysis.

System Integration

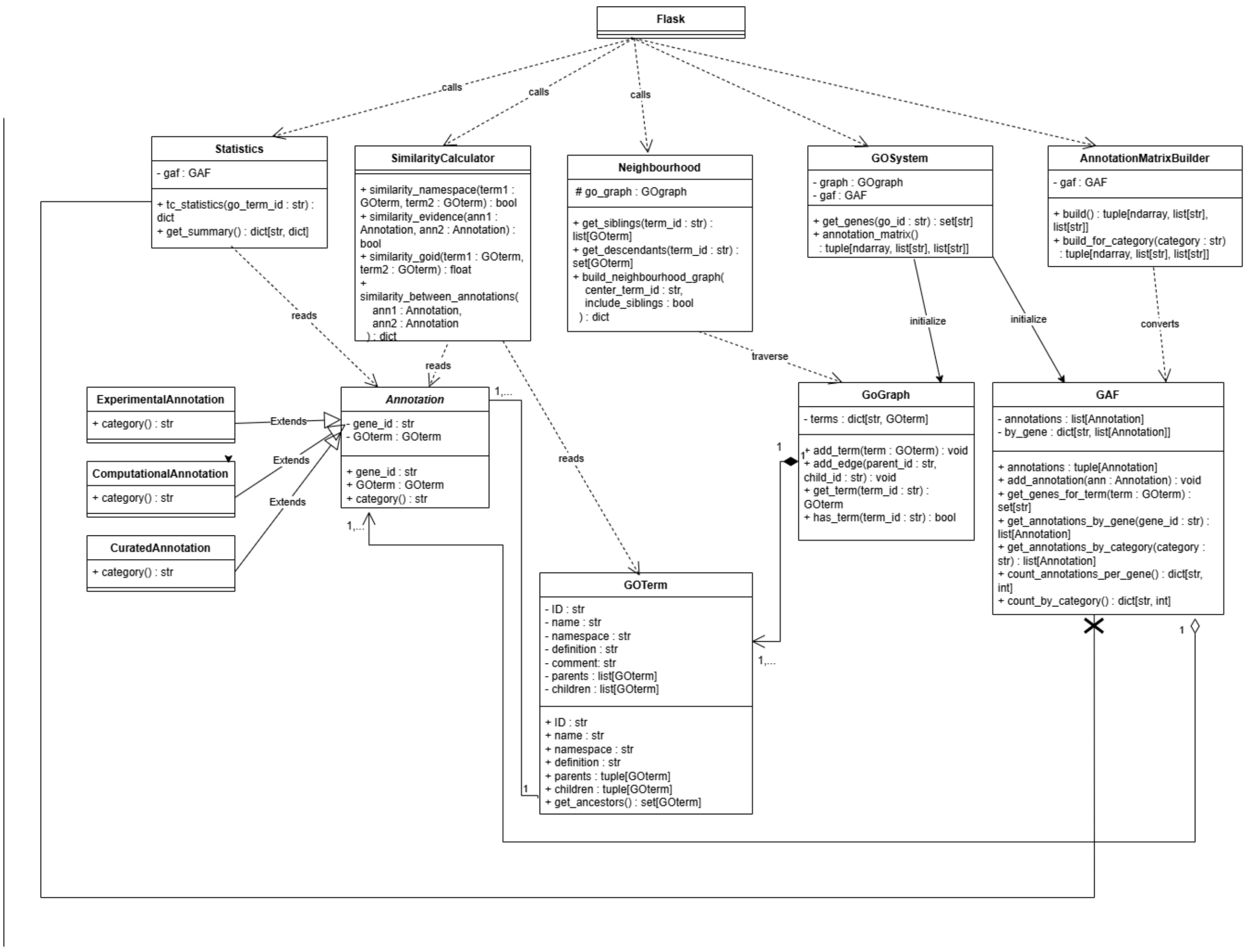
The GOSystem class, defined in **main.py**, coordinates the interaction between the ontology graph, the annotation dataset, and the analytical components. This class provides high-level methods for common operations, such as retrieving genes associated with a GO term or generating annotation matrices.

User Interface

The user interface is implemented as a web application using Flask and is located in **app.py** and the templates directory. The interface enables users to explore GO terms, compare GO concepts, see statistical computations, and analyse ontology neighbourhoods.

The web layer interacts with the system integration module and does not directly access parsing or domain logic. This design allows for reuse with alternative interfaces.

The UML diagram is presented below:


Flask
calls
calls
calls
Statistics
- gaf : GAF
+ tc_statistics(go_term_id : str) : dict
+ get_summary() : dict[str, dict]
SimilarityCalculator
+ similarity_namespace(term1 : GOterm, term2 : GOterm) : bool
+ similarity_evidence(ann1 : Annotation, ann2 : Annotation) : bool
+ similarity_goid(term1 : GOterm, term2 : GOterm) : float
+ similarity_between_annotations(ann1 : Annotation, ann2 : Annotation) : dict
Neighbourhood
# go_graph : GOgraph
+ get_siblings(term_id : str) : list[GOterm]
+ get_descendants(term_id : str) : set[GOterm]
+ build_neighbourhood_graph(center_term_id : str, include_siblings : bool) : dict
GOSystem
- graph : GOgraph
- gaf : GAF
+ get_genes(go_id : str) : set[str]
+ annotation_matrix() : tuple[ndarray, list[str], list[str]]
AnnotationMatrixBuilder
- gaf : GAF
+ build() : tuple[ndarray, list[str], list[str]]
+ build_for_category(category : str) : tuple[ndarray, list[str], list[str]]
reads
reads
reads
traverse
initialize
initialize
converts
ExperimentalAnnotation
+ category() : str
ComputationalAnnotation
+ category() : str
CuratedAnnotation
+ category() : str
Extends
Extends
Extends
Annotation
- gene_id : str
- GOterm : GOterm
+ gene_id : str
+ GOterm : GOterm
+ category() : str
1,...
1,...
GoGraph
- terms : dict[str, GOterm]
+ add_term(term : GOterm) : void
+ add_edge(parent_id : str, child_id : str) : void
+ get_term(term_id : str) : GOterm
+ has_term(term_id : str) : bool
1
1
1,...
GAF
- annotations : list[Annotation]
- by_gene : dict[str, list[Annotation]]
+ annotations : tuple[Annotation]
+ add_annotation(ann : Annotation) : void
+ get_genes_for_term(term : GOterm) : set[str]
+ get_annotations_by_gene(gene_id : str) : list[Annotation]
+ get_annotations_by_category(category : str) : list[Annotation]
+ count_annotations_per_gene() : dict[str, int]
+ count_by_category() : dict[str, int]
1
GOTerm
- ID : str
- name : str
- namespace : str
- definition : str
- comment: str
- parents : list[GOterm]
- children : list[GOterm]
+ ID : str
+ name : str
+ namespace : str
+ definition : str
+ parents : tuple[GOterm]
+ children : tuple[GOterm]
+ get_ancestors() : set[GOterm]
1

III. Parsing and Data Handling

The parsing and data handling responsibilities are encapsulated in a dedicated parsing module, whose purpose is to transform raw Gene Ontology (OBO) and Gene Annotation (GAF) files into structured, object-oriented representations that can be reused by all other components of the system.

This design ensures a clear separation of concerns:

- Parsing logic is isolated from ontology navigation, annotation analysis, statistics, and user interface code.
- Parsed data is exposed through well-defined domain objects rather than raw text or tables.

The parsing module consists of two main functions:

- **parse_obo()** for ontology parsing
- **parse_gaf()** for annotation parsing

Both functions produce fully initialised domain objects.

Parsing Responsibilities Distribution

a) OBO Parsing (**parse_obo**)

Responsibility

The **parse_obo** function is responsible for:

- Reading the GO ontology file line by line
- Extracting term attributes (ID, name, namespace, definition)
- Capturing hierarchical relationships (is_a)
- Constructing a complete ontology graph

Design Choice

Instead of storing ontology data in plain dictionaries or DataFrames, each GO term is instantiated as a **GOterm** object and stored inside a **GOgraph** object.

Process

1. The parser scans the file sequentially.
2. When a [Term] block is encountered, attributes are temporarily stored in a dictionary.
3. A GOterm object is created once all required fields are available.
4. Parent GO IDs are stored temporarily on the term (**GOterm_parents**) to allow edge creation after all terms are loaded.
5. After parsing all terms, the graph structure is built by connecting parent and child terms using **GOgraph.add_edge()**.

Encapsulation

- The internal structure of the ontology (parents, children) is hidden inside GOterm.
- External components can only access relationships via methods like parents, children, and **get_ancestors()**.

b) GAF Parsing (**parse_gaf**)
Responsibility
The **parse_gaf** function is responsible for:

- Reading the GAF file using Pandas
- Filtering and extracting relevant columns
- Mapping annotations to existing GO terms
- Creating appropriate annotation objects based on evidence codes

Use of Pandas

Pandas is used exclusively for:

- Efficient parsing of large tabular data
- Column selection and iteration

Once parsed, the DataFrame is not exposed to the rest of the system. This ensures that numerical libraries do not leak into the domain model.

Data Encapsulation Strategy

The system strictly avoids sharing raw data structures between components.

Data Type	**Encapsulated In**	**Purpose**
GO terms	GOterm	Ontology node
Ontology graph	GOgraph	DAG structure
Annotations	Annotation subclasses	Gene–term relationships
Annotation collection	GAF	Querying and summarisation

This design ensures:

- Strong encapsulation
- Controlled access to internal state
- Easy extensibility

For example:

- External components cannot directly modify parent–child relationships.

Annotation evidence categories are abstracted behind a polymorphic interface.

Application of OOP Principles

Encapsulation

- Internal data (parents, children, annotation lists) is private.
- Access is controlled through methods and properties.

Abstraction

- **Annotation** defines a common interface.
- Parsing functions hide file format complexity.

Inheritance

- Evidence types inherit from **Annotation**.

Polymorphism

- Evidence categories are handled via **category()** without conditional logic elsewhere.
- The same interface is used regardless of annotation type.

This architecture was chosen to:

- Separate parsing logic from domain logic
- Support extensibility (new evidence types, new analyses)
- Allow safe reuse across statistical analysis, similarity computation, and web UI
- Align with object-oriented best practices required by the project specification

The parsing component acts as the foundation of the system, enabling all higher-level functionality without exposing low-level file handling details.

IV. OOP

The project is centered around the structured biological concepts - Gene Ontology terms, their relationships, and gene annotations - which can be represented by objects with certain attributes and behaviors. For this reason, an object-oriented approach was chosen for the structure of the system.

The OOP design shows:

- **Encapsulation** of biological concepts (GO terms, annotations);

- Clear **abstraction** boundaries between ontology structure, annotations, and numerical analysis;

- **Inheritance**, used in the annotation hierarchy, where concrete annotation classes extend a common base class Annotation;

- **Polymorphism**, according to which the analytical modules operate uniformly on different annotation subclasses, without requiring changes to the code.

The OOP components are contained in the **OOP/ directory**.

Ontology

The Gene Ontology is modeled using two core classes:

- GOterm, representing a single ontology term;

- GOgraph, representing the ontology as a directed acyclic graph (DAG)

Each GOTerm stores:

- a unique identifier (__ID)
  a name (__name)
- a namespace (__namespace, e.g. biological_process, molecular_function)
- a definition (__definition)
- references to parent and child terms (__parents, __children)

Access to these attributes is provided through read-only properties (ID, name, namespace, definition, parents, children), enforcing encapsulation. The parents and children properties can be extended by the private **__add_parent(parent)** and **__add_child(child)** methods.

The class also defines the **get_ancestors()** method, which performs a graph analysis over the parent relationships, computing the whole set of ancestor terms.

The term GOgraph manages the global structure of ontology and the relationships between the terms. It uses a dictionary mapping GO ids to GOterm objects.

Key methods include:
- **add_term(term)**, which registers new ontology terms;
- **add_edge(parent_id, child_id)**, which creates parent–child relationships between existing terms;
- **get_term(term_id)**, which retrieves a term by its identifier;
- **has_term(term_id)**, which checks for term existence.

By isolating the graph-related information inside GOgraph, we avoid duplicating the logic inside individual terms and maintain a single representation of the ontology structure.

Annotation

Annotations are represented using a class hierarchy, representing the distinction between different annotation evidences while maintaining a shared interface.

- The abstract base Annotation class defines common attributes (**__gene_id**, **__GO term**);
- Specialized subclasses (**ExperimentalAnnotation**, **ComputationalAnnotation**, **CuratedAnnotation**) capture the semantic differences without code duplication.

The Annotation class attributes are exposed via read-only properties (**gene_id**, **GOterm**) to preserve data integrity. It also contains an abstract method **category()**.

Each subclass implements the category() method to return a different evidence-based string. This demonstrates inheritance and polymorphism, allowing analytical components to interact with annotations through the common Annotation interface without knowing their concrete type.

The GAF class acts as a container and indexing structure for annotations. It stores:
- a private list of all annotations (**__annotations**);
- a dictionary indexing annotations by gene identifier (**__by_gene**).

The class provides multiple query and summarisation methods, including:
- get_annotations_by_gene(gene_id)
- get_annotations_by_category(category)
- count_annotations_per_gene()
- count_by_category()

In addition, the method **get_genes_for_term(term)** retrieves all genes annotated to a given GO term or any of its ancestor terms, combining annotation data with ontology structure.

This design allows analytical modules to operate polymorphically on annotations, enabling evidence-based filtering or statistics.

Numerical and Matrix-Based Analysis

The class **AnnotationMatrixBuilder** transforms object-based annotations into NumPy-based matrix representations, which are required for statistical and comparative analysis.

This class receives a GAF object and transforms its annotation data into a matrix, where:

- rows correspond to genes;
- columns correspond to GO terms;
- matrix entries indicate the presence or absence of an annotation.

The **build()** method constructs the matrix. The method **build_for_category(category)** filters annotations based on their polymorphic category() method before constructing the matrix.

This class represents the relationship between OOP and numerical computation.

CRC cards for OOP

Class Name: GOterm	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Maintain parent and child relationships	GOgraph	
Store GO term metadata (ID, name, namespace)	Other GOterms	
Provide access to ancestors and descendants		

Class Name: GOgraph	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Store all GO terms in a unified graph structure	GOterm	
Manage parent–child relationships between terms	Parser	
Serve as a representation of the GO DAG	Analytical modules	

Class Name: Annotation (Base class)	**Superclass:** None	**Subclass:** ExperimentalAnnotation ComputationalAnnotation CuratedAnnotation
Responsibilities	**Collaborators**	
Store core annotation attributes (gene ID, GO term ID, evidence code)	GOterm	
Provide a shared interface for all annotation types	GAF	
Represent gene–term associations with evidence data		

Class Name: ExperimentalAnnotation	**Superclass:** Annotation	**Subclass:** None
Responsibilities	**Collaborators**	
Represent experimentally validated annotations	Annotation	
	GAF	

Class Name: ComputationalAnnotation	**Superclass:** Annotation	**Subclass:** None
Responsibilities	**Collaborators**	
Represent computationally validated annotations	Annotation	
	GAF	

Class Name: CuratedAnnotation	**Superclass:** Annotation	**Subclass:** None

Responsibilities	**Collaborators**
Represent manually validated annotations	Annotation
	GAF

Class Name: GAF	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Store collections of annotations	Annotation	
Index annotations by gene ID	Parser	
Act as an interface between parsed data and OOP analysis	Analytical modules	

Class Name: AnnotationMatrixBuilder	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Convert annotation objects into numerical matrix representations	Annotation	
Interface with NumPy-based analysis	GAF	
Separate biological modeling from numerical computation	GOgraph	

V. Analysis

The Analysis component builds upon the OOP foundation to extract biological insights from the ontology and annotation data. By separating analytical logic from data storage, the system ensures that algorithms remain modular and easy to test.
The software supports analytical operations combining ontology structure, annotations, and numerical representations. These capabilities include:

- **Similarity computations:** Comparing terms and annotations based on graph position or evidence.
- **Ontology neighbourhood analyses:** Exploring local graph structures such as siblings and descendants.
- **Statistical summaries:** Aggregating annotation data into numerical metrics.

Similarity computation(polymorphism) in Analysis

A key feature of the analytical module is the clear demonstration of polymorphism. The SimilarityCalculator class compares annotations using the similarity_evidence method.

- It accepts any object inheriting from the base Annotation class.
- It calls the .category() method, which is dynamically resolved at runtime depending on the specific subclass (e.g., ExperimentalAnnotation vs. ComputationalAnnotation).
- This allows the analysis logic to remain unchanged even if new annotation types are added.

The analysis components are implemented in neighborhood.py, statistics.py, and similarity.py.

Graph Neighborhood Analysis

The Neighbourhood class performs ontology neighbourhood analyses by exploring the local context of a specific Gene Ontology term.

- It identifies **siblings** (terms sharing a parent) to find related concepts.
- It retrieves **descendants** recursively to understand term specificity.
- It constructs a subgraph dictionary representing the term's immediate topology.

Statistical Analysis

The **Statistics** class provides statistical summaries by aggregating data from the GAF file.

- It computes term-centric metrics, such as the number of genes and the category breakdown.
- It utilizes **NumPy** to calculate the mean and standard deviation of annotations per gene, bridging object-oriented data with numerical analysis.

Similarity and Comparison
The SimilarityCalculator class handles similarity computations between biological entities.
- **Semantic Similarity:** It implements the Jaccard Index on term ancestors (similarity_goid) to measure how closely related two terms are in the graph.
- **Evidence Similarity:** It uses the polymorphic comparison of evidence codes.
- **Namespace Similarity:** It validates if terms belong to the same biological domain.

CRC Cards for analysis

Below are the CRC (Class–Responsibility–Collaborator) cards for all classes in the Analysis module.

Class Name: Neighbourhood	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Traverse the GO graph to retrieve siblings	GOgraph	
Recursive retrieval of all descendants for a given term	GOterm	
Construct a dictionary-based graph representation of a term's local neighborhood		

Class Name: Statistics	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Filter annotations by GO term ID	GAF	
Aggregate gene counts and category summaries	Annotation (and subclasses)	
Calculate mean and standard deviation of annotations per gene using NumPy	GOterm	
Generate a summary report for all terms in a GAF file		

Class Name: Similarity	**Superclass:** None	**Subclass:** None
Responsibilities	**Collaborators**	
Calculate Semantic Similarity between GO terms (Jaccard Index of ancestors)	GOterm	
Demonstrate Polymorphism: Compare Annotation objects based on evidence category	Annotation (and subclasses)	
Compare GO terms based on namespace		

VI. **System Integration and User Interface**

Purpose of main.py

The file **main.py** defines the **GOSystem** class, which acts as the central integration component of the software system. Its primary responsibility is to coordinate and connect all major subsystems, including ontology parsing, annotation handling, and analytical modules.

Rather than performing domain-specific computations itself, **GOSystem** serves as a high-level façade, providing a simplified and unified interface for accessing the system's core functionality.

In summary, **main.py**:

- does not implement biological logic or numerical analysis directly;
- instead, it orchestrates and exposes the system's capabilities;
- improves modularity, readability, and architectural clarity.

Responsibilities of the GOSystem Class

The **GOSystem** class is responsible for:

1. **System Initialization**

- Loading and parsing the Gene Ontology (OBO) file
- Loading and parsing the Gene Annotation File (GAF)
- Constructing and storing the ontology graph and annotation dataset

2. **Component Integration**

- Connecting the ontology representation (**GOgraph**) with gene annotations (GAF)
- Ensuring that annotations reference valid ontology terms
- Providing a single access point for other components (e.g. analysis modules and the web interface)

3. **High-Level Query Operations**

- Retrieving genes annotated to a given GO term, including inherited annotations via ontology hierarchy
- Creating numerical representations (annotation matrices) by delegating to analytical components

4. **Decoupling User Interface from Domain Logic**

- The Flask web application interacts only with **GOSystem**

- Internal implementation details (parsing, data structures, NumPy matrices) remain hidden

Application of Object-Oriented Principles

Encapsulation

GOSystem encapsulates:

- the ontology graph (**GOgraph**);
- the annotation dataset (GAF);
- the logic required to initialise and connect these components.

External components access this functionality only through public methods such as **get_genes()** and **annotation_matrix()**.

Abstraction

GOSystem abstracts away:

- file formats (OBO, GAF);
- parsing strategies;
- internal data representations.

Consumers of the class do not need to understand how data is loaded or stored, only *what* operations are available.

Polymorphism

Although **GOSystem** does not directly implement inheritance, it operates on polymorphic objects indirectly:

- it stores annotations that may be **ExperimentalAnnotation**, **ComputationalAnnotation**, or **CuratedAnnotation**;
- analytical methods work uniformly on these objects via the abstract **Annotation** interface.

Inheritance

Inheritance is not used directly in **GOSystem**, by design. Instead, it coordinates inherited components (such as the **Annotation** hierarchy), respecting the separation of concerns.

Position in the Overall Architecture

Within the system architecture:

- **main.py** belongs to the integration layer
- it sits above the parsing, domain model, and analytical layers
- it acts as a single entry point for both command-line execution and the web-based interface

This design ensures that future extensions (e.g. additional analyses or alternative UIs) can reuse the same core system without modification.

CRC Card for GOSystem

Class	Responsibilities	Collaborators
GOSystem	• Initialise and coordinate ontology and annotation parsing • Load and store the GO graph and GAF annotation data • Provide a unified access point to the system's core functionality • Support high-level queries (e.g. retrieve genes for a GO term) • Delegate numerical and matrix-based analyses to specialised components • Encapsulate internal system structure from external users	• **parse_obo** • **parse_gaf** • **GOgraph** • **GAF** • **AnnotationMatrixBuilder** • Flask Web Interface

The decision to introduce a **GOSystem** class was driven by the need to:

- avoid tight coupling between the user interface and internal data structures;
- centralise system configuration and initialisation logic;
- improve maintainability and extensibility.

By grouping system-level responsibilities into a single class, the architecture becomes easier to understand and modify.

Purpose of the User Interface

The user interface (UI) provides an interactive, web-based access layer to the Gene Ontology Analysis System. Its goal is to expose the underlying ontology representation, annotation analysis, statistical modelling, similarity computation, and neighbourhood exploration functionalities to end users in an intuitive and structured way.

The UI is implemented as a thin presentation layer that delegates all domain logic to the existing object-oriented backend. This ensures that the web interface remains modular, extensible, and consistent with the overall system architecture.

Architectural Overview

The UI is built using a Model–View–Controller (MVC)-inspired architecture:

- **Model**: Domain classes (**GOterm**, **GOgraph**, **GAF**, **Statistics**, **SimilarityCalculator**, **Neighbourhood**)
- **Controller**: Flask routes in **app.py**
- **View**: HTML templates rendered with Jinja2

Flask acts as the controller layer, coordinating user requests, invoking domain logic, and passing structured results to HTML templates.

User → HTML Form → Flask Route → Domain Objects → Flask → HTML Template → User

Backend Responsibilities (Flask – app.py)

The Flask backend is responsible for:

- Handling HTTP requests (GET/POST)
- Validating user input (e.g. GO term IDs)
- Calling domain-level services
- Passing results to templates for rendering

Key Design Decisions

- **Single system instance** (**GOSystem**) is created at startup to avoid repeated parsing of large OBO/GAF files.
- Flask routes never manipulate raw data structures directly.
- All computation is delegated to domain classes （**Statistics，SimilarityCalculator，Neighbourhood**）.

This preserves separation of concerns and avoids logic duplication.

Implemented UI Functionalities

1. Home Page (index.html)

Purpose

- Acts as the navigation hub for the application.

Features

- GO term search
- GO term comparison

- Access to statistics
- Access to ontology neighbourhood exploration

Backend Route

```
@app.route("/")

def index():

    return render_template("index.html")
```

2. GO Term Exploration (/term)

Purpose

- Display detailed information about a single GO term.

Displayed Information

- GO ID
- Name
- Namespace
- Definition
- List of annotated genes

Backend Logic

- Retrieves a **GOterm** object from **GOgraph**
- Retrieves associated genes via **GAF.get_genes_for_term**

This ensures that ontology structure and annotation propagation (via ancestors) are respected.

3. GO Term Statistics (Single-Term Analysis) (/stats)

Purpose

- Provide statistical summaries for a single GO term, rather than the entire dataset.

Metrics Displayed

- Number of annotated genes
- Mean annotations per gene
- Standard deviation
- Annotation category breakdown (experimental, computational, curated)

Key Design Improvement

- Statistics are computed on demand for a single GO term using:

Statistics.tc_statistics(go_id)

This avoids unnecessary global computations and aligns with user-driven exploration.

4. GO Term Comparison (/compare)

Purpose

- Compare two GO terms using multiple similarity measures.

Similarity Dimensions

- Namespace similarity
- Semantic similarity (Jaccard index over ancestors)
- Evidence similarity (via polymorphic annotation categories)

Design Choice

- Similarity logic is entirely encapsulated in **SimilarityCalculator**.
- Flask only coordinates input/output.

This allows future extension (e.g. new similarity metrics) without changing the UI.

5. Ontology Neighbourhood Exploration (/neighbourhood)

Purpose

- Allow users to explore the local ontology structure around a GO term.

Displayed Relationships

- Parents
- Children
- Siblings

Visualization Strategy

- **Text-based hierarchical representation**
- Chosen deliberately to:
    - Keep UI simple and interpretable
    - Focus on conceptual understanding of the DAG

This approach satisfies the project requirements without introducing unnecessary visualization libraries.

HTML Templates

Each HTML template corresponds to one specific responsibility, following the Single Responsibility Principle.

Template	**Responsibility**
index.html	Navigation and entry point
term.html	GO term details and genes
stats.html	GO term statistics input
matrix_single_term.html	Statistical result rendering
compare.html	GO term similarity results
neighbourhood.html	Ontology neighbourhood display

Templates contain only presentation logic using Jinja2.

Application of OOP Principles in the UI

Encapsulation

- UI never accesses raw data structures directly.
- All data access occurs via methods (**tc_statistics, get_siblings, similarity_goid**).

Abstraction

- Flask routes operate on abstract concepts (terms, statistics, similarity), not file formats or DataFrames.

Polymorphism

- Evidence similarity relies on polymorphic **category()** methods in annotation subclasses.

Inheritance

- Annotation subclasses (**ExperimentalAnnotation**, **ComputationalAnnotation**, **CuratedAnnotation**) enable uniform handling in the UI.

The user interface layer is a clean, modular, and extensible presentation layer that:

- Integrates seamlessly with the object-oriented backend
- Demonstrates clear separation of concerns
- Supports ontology exploration, statistical analysis, similarity computation, and neighbourhood navigation
- Applies OOP principles consistently at the architectural level

This design ensures that the system can be extended with new analyses or visualisations without restructuring the UI or backend.