HAW HAMBURG

MASTER THESIS
Cedric Stolze

Detecting Design Patterns in Source Code using Neural Subgraph Matching: A Language-Independent Approach

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Cedric Stolze

Detecting Design Patterns in Source Code using Neural Subgraph Matching: A Language-Independent Approach

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Olaf Zukunft
Supervisor: Prof. Dr. Stefan Sarstedt

Submitted on: 17. April 2025

Cedric Stolze

Thema der Arbeit

Erkennung von Design Patterns im Quellcode durch neuronales Subgraph Matching: Ein sprachunabhängiger Ansatz

Stichworte

Statische Codeanalyse, Entwurfsmuster, Code Property Graph, Graph Neural Network, Subgraph Matching

Kurzzusammenfassung

Entwurfsmuster sind essenziell um die Softwarequalität, Wartbarkeit und Skalierbarkeit zu verbessern. Die Erkennung von Entwurfsmustern ist aufgrund unterschiedlicher Implementierungen, sprachspezifischer Merkmale und komplexer Codestrukturen eine Herausforderung. Ansätze im Bereich maschinelles Lernen und graphbasierte Methoden haben vielversprechende Ergebnisse gezeigt, basieren jedoch häufig auf sprachspezifischen Vorlagen. Diese Arbeit erforscht einen sprachunabhängigen Ansatz zur Erkennung von Entwurfsmustern mittels Neural Subgraph Matching. Das vorgestellte Konzept abstrahiert Quellcode in einen sprachunabhängigen Code Property Graph (CPG), welcher Syntax, Semantik und Verhalten vereinheitlicht. Der CPG wird anschließend in einen Record Interaction Graph (RIG) umgewandelt, welcher zentrale Interaktionen zwischen Code-Elementen zusammenfasst. Ein Graph Learnable Multi-hop Attention Network (GLeMA Net) wird verwendet, um Neural Subgraph Matching auf Beispielen von Entwurfsmustern statt auf vordefinierten Regeln durchzuführen. Auswertungen mit realen Softwareprojekten demonstrieren die Flexibilität des Ansatzes, verdeutlichen jedoch auch eine hohe Rate an False Positive Erkennungen. Im Vergleich zu bestehenden Ansätzen mit sprachspezifischen Vorlagen zeigt der verwendete Ansatz Potential für weitere Betrachtung.

Cedric Stolze

Title of Thesis

Detecting Design Patterns in Source Code using Neural Subgraph Matching: A Language-Independent Approach

Keywords

Static Code Analysis, Design Pattern Detection, Code Property Graph, Graph Neural Network, Subgraph Matching

Abstract

Design patterns are essential for improving software quality, maintainability, and scalability. Detecting design patterns is challenging due to varied implementations, language-specific features, and complex code structures. Existing methods, including machine learning and graph-based approaches, show promising results but rely on language-specific templates. This thesis researches a language-independent approach for detecting design patterns using neural subgraph matching. The proposed concept abstracts source code into a language-independent Code Property Graph (CPG) that unifies syntax, semantics, and behavior. The CPG is transformed into a Record Interaction Graph (RIG), capturing crucial interactions between code entities. A Graph Learnable Multi-hop Attention Network (GLeMA Net) is used to perform subgraph matching on design pattern examples rather than handcrafted rules. Evaluations using real-world software demonstrate the flexibility of this approach but also highlight the high false-positive rate. When compared to existing machine learning approaches with language-specific templates, the proposed approach shows potential for further research.

Contents

List of Figures

List of Tables

Abbreviations

AST Abstract Syntax Tree.

CFG Control Flow Graph.

CNN Convolutional Neural Network.

CPG Code Property Graph.

DFG Data Flow Graph.

DSL Domain Specific Language.

GAT Graph Attention Network.

GCN Graph Convolutional Network.

GLeMA Net Graph Learnable Multi-hop Attention Network.

GNN Graph Neural Network.

GoF Gang of Four.

ML Machine Learning.

OOP Object-Oriented Programming.

PDG Program Dependence Graph.

RIG Record Interaction Graph.

SSSP Single Source Shortest Path.

1 Introduction

Software design patterns are widely recognized as essential components in object-oriented software engineering, contributing significantly to software maintainability, reusability, and comprehension [81]. The effective utilization of design patterns allows developers to solve common and recurring problems by employing well-established and proven solutions, preventing inconsistencies and possible errors [1]. Identifying instances of design patterns within existing software systems further promotes consistency across projects, facilitates automated documentation, and enhances the scalability of software maintenance efforts [5]. Software systems are under constant evolution and may lack comprehensive documentation, making it challenging to maintain existing design pattern implementations.

Detecting design patterns is therefore a critical task that can help to improve software quality. Existing approaches to design pattern detection primarily use heuristic-based, graph-based, and machine learning methods [78]. Heuristic methods involve manually defined rules that rely heavily on specific characteristics of patterns and are often limited by their rigidity and inability to generalize across variations in pattern implementations. Graph-based methods address some of these limitations by abstracting source code into structural representations, enabling pattern detection through a wide range of graph algorithms. Recent advancements in machine learning have introduced opportunities for more flexible and generalized pattern detection by learning directly from data rather than relying on explicit rule definitions [78]. Hybrid approaches combining graph-based and machine learning techniques have shown promising results in detecting design patterns across diverse software systems [60].

1.1 Problem Statement

Despite these advances, design pattern detection continues to face significant challenges. One major difficulty arises from implementation variability, where patterns may have

diverse representations even within the same software system [78]. Existing language-specific detection approaches often require handcrafted templates, limiting their applicability across different programming languages [7, 4]. Additionally, many pattern-specific approaches lack the ability to generalize to other patterns, restricting their broader utility [20]. Moreover, current detection methods frequently depend on compilable code, which constrains their applicability to incomplete or dynamically evolving software systems.

1.2 Research Goals

To address these challenges, this thesis researches a language-independent approach for detecting design patterns using neural subgraph matching techniques on Code Property Graphs (CPGs). The proposed concept capitalizes on the versatility of CPGs, which represents source code in a unified abstraction, independent of any specific programming language. This abstraction captures both structural and behavioral characteristics advantageous for effective pattern detection. By employing a Graph Neural Network (GNN) for subgraph matching, the proposed approach avoids dependency on handcrafted templates or rigid rule-based definitions, instead relying on example-based queries to generalize across multiple patterns and languages.

The general goal of this thesis is to investigate the effectiveness of combining language-independent source code abstractions with neural subgraph matching techniques for detecting design patterns. For this, the following research questions are proposed:

RQ 1. How can Code Property Graphs be effectively abstracted into a language-independent representation that captures the structural and behavioral characteristics of design patterns and is suitable for neural subgraph matching?

RQ 2. What techniques enable robust detection of design patterns that handle implementation variations without relying on handcrafted templates or rule-based definitions?

RQ 4. To what extent can a language-independent approach for design pattern detection achieve comparable accuracy to existing language-specific machine learning approaches?

To answer these questions, the research includes the development of a prototype implementation for abstracting source code into CPGs, training a GNN model for subgraph

matching, evaluating the approach on real-world software projects, and comparing the results against existing pattern detection methods. This approach aims to demonstrate the potential of language-independent source code abstractions in the context of a complex pattern detection task.

1.3 Structure of Content

To address the outlined research goal and questions, this thesis starts by introducing foundational knowledge and principles essential for understanding the subsequent research in chapter 2, including topics like software design, code analysis techniques, graph theory, and graph neural networks. Chapter 3 provides an overview of existing research in design pattern detection, highlighting proven methods and identifying limitations that this thesis aims to address. After establishing the theoretical foundations and related works, chapter 4 presents the central concepts of the proposed approach, including the CPG abstraction and neural subgraph matching techniques. Chapter 5 subsequently outlines the implementation details, describing the system architecture, graph generation methods, and procedures for model training. The experiment results and evaluation of the implementation are described in chapter 6. Finally, chapter 7 summarizes the key findings, discusses their implications and limitations, and suggests possible directions for future research.

2 Background

In order to comprehend the context of this thesis, a basic understanding of the underlying theoretical concepts is essential. For this, the chapter provides an overview of each topic and begins with an introduction to software design and its fundamental principles. After this, various design patterns and their importance in software development are described. Next, the chapter explains methods for analyzing code and describing the difference between static and dynamic approaches. It then provides a basic understanding of graph theory, including a general graph definition and essential properties that are central to the subsequent discussions of the thesis. Finally, the chapter examines Graph Neural Networks, highlighting recent developments in this area.

2.1 Software Design

The design aspect of software development is one of the most critical phases in the software development lifecycle (fig. 2.1). This phase is crucial because it lays the foundation for the entire software system [18]. Effective software design ensures that the software can handle changing requirements and scale with increasing demands [39]. In addition to this, software design is a highly individual process that can vary significantly depending on the project, the team, and the requirements. For this, software architects and developers follow a set of principles and blueprints to ensure that the software reaches the desired quality and functionality [55]. The scope can be a global system architecture or a local design pattern for specific problems. Software architecture refers to the overall structure of a system, including its components, their interactions, and the guiding design policies. On the other hand, at a more localized level, design patterns represent proven solutions to common software development problems. These patterns encapsulate best practices and provide reusable templates that can be applied to similar situations across different projects. In all cases, effective software design choices can leverage the common design principles for software development [53].

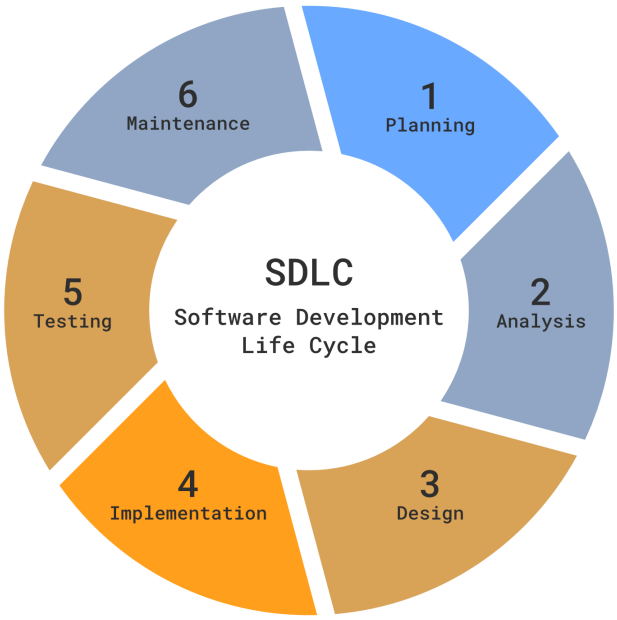


Figure 2.1: A common software development life cycle (SDLC) with 6 phases [57].

2.1.1 Design Quality and Principles

An effective software design is based on a set of principles and quality requirements [41]. In general, a software product should meet specific quality attributes through its design decisions. Common quality attributes like the *ISO 25010* standard [27, 6] are shown in fig. 2.2.

Functional Suitability	Performance Efficiency	Compatibility	Interaction Capability	Reliability	Security	Maintainability	Flexibility	Safety
• Functional Completeness • Functional Correctness • Functional Appropriateness	• Time Behaviour • Resource Utilization • Capacity	• Co-Existence • Interoperability	• Appropriateness Recognizability • Learnability • Operability • User Error Protection • User Engagement • Inclusivity • User Assistance	• Faultlessness • Availability • Fault Tolerance • Recoverability	• Confidentiality • Integrity • Non-Repudiation • Accountability • Authenticity • Resistance	• Modularity • Reusability • Analysability • Modifiability • Testability	• Adaptability • Scalability • Installability • Replaceability	• Operational Constraint • Risk Identification • Fail Safe • Hazard Warning • Safe Integration

Figure 2.2: Quality attributes of a software product by the standard *ISO 25010* [27].

Most design principles are focused on improving the maintainability, readability, and scalability of the software. For this, modularity and encapsulation are key concepts in software design [1]. Modularity refers to the division of a software system into smaller, independent components that can be developed, tested, and maintained separately. Encapsulation, on the other hand, refers to the bundling of data and methods into a single

unit. This can be measured by the coupling and cohesion of the software components. Coupling refers to the degree of interdependence between software modules, while cohesion refers to the degree to which the elements inside a module belong together. High cohesion and low coupling are desirable in software design because they make the software easier to understand, maintain, and extend [59]. Those concepts are central to Object-Oriented Programming (OOP), which is a common programming paradigm that is based on the concept of encapsulating data in objects [11, 10]. Objects are instances of classes, which are user-defined types that contain data and methods. The most common design principles for OOP are the SOLID principles [61], which help developers reach the described software quality attributes. The SOLID principles are:

- **Single Responsibility Principle (SRP):** A class should have only one reason to change.
- **Open/Closed Principle (OCP):** Software entities should be open for extension but closed for modification.
- **Liskov Substitution Principle (LSP):** Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.
- **Interface Segregation Principle (ISP):** A client should not be forced to implement an interface that it does not use.
- **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Those principles are proven to be effective in object-oriented programming, but they can be applied to other paradigms as well. The SOLID principles are the foundation of many design patterns, which are discussed in the next section [81].

2.1.2 Design Patterns

In contrast to software architecture patterns that are focused on the overall structure of a system, design patterns are focused on solving specific problems that arise during software development. Design patterns are reusable solutions to common problems [43]. They are not finished designs that can be transformed directly into code. Instead, they provide a

template for how to solve a problem that can be used in many different situations. Design patterns can speed up the development process by providing tested, proven development paradigms [81]. Reusing design patterns helps to prevent issues that can cause major problems and improves code readability for developers and architects.

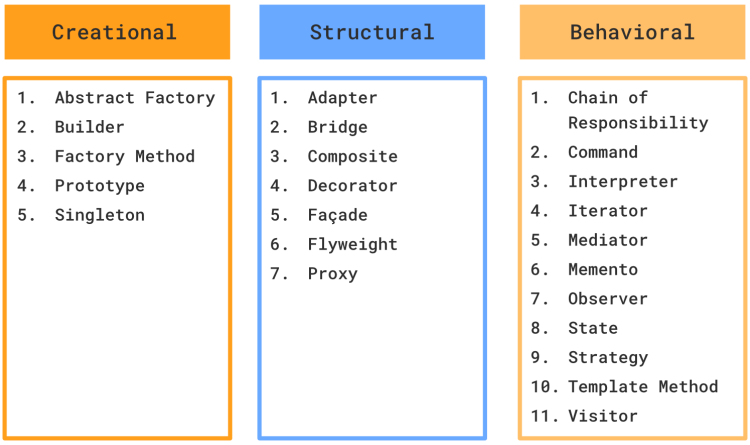


Figure 2.3: Categorized design patterns by the Gang of Four [19].

The most common design patterns are the Gang of Four (GoF) patterns [19] and are shown in fig. 2.3. According to the GoF, design patterns can be classified into three main categories:

- **Creational Patterns:** These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by controlling the object creation process.
- **Structural Patterns:** These patterns deal with object composition. They describe how objects and classes can be combined to form larger structures. Structural design patterns simplify the structure by identifying and abstracting the dependencies.
- **Behavioral Patterns:** These patterns are focused on interactions between objects. They describe how objects interact and communicate with each other. Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

The GoF patterns are all formulated as general reusable solutions, expressed in terms of objects and interfaces and focusing on the OOP paradigm.

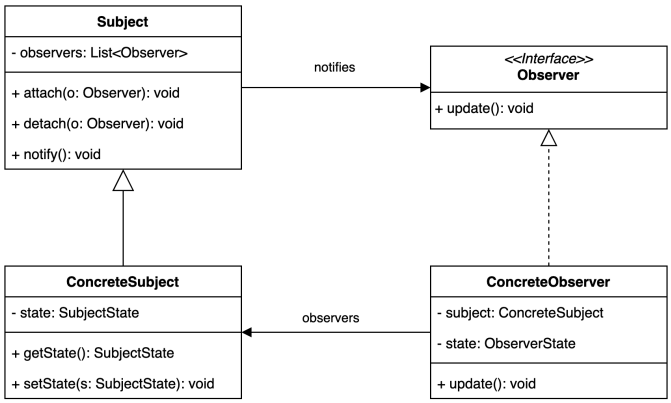


Figure 2.4: The *Observer* patterns participants and interactions, defined by the GoF [19].

A design pattern systematically declares and explains a general design that addresses a recurring design problem in object-oriented systems and is formulated as language-independent definitions. An example of a design pattern is the *Observer* pattern, which is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The design pattern also describes the participants involved in the pattern and the interactions between them (fig. 2.4). In the case of the *Observer* pattern, the participants are the subject, the observer, the concrete subject, and the concrete observer. The subject is the object that is being observed, and the observer is the object that observes the subject. The concrete subject and concrete observer are the implementations of the subject and observer. With the help of this pattern, the developer can design a solution to a common object interaction problem by implementing a loosely coupled system that can be extended and maintained.

2.2 Code Analysis

Code analysis is the systematic examination of software artifacts to derive insights, detect defects, and ensure adherence to quality standards. In modern software development, it plays a pivotal role in addressing the growing complexity of systems, accelerating development cycles, and mitigating risks associated with security vulnerabilities, perfor-

mance degradation, and maintenance costs [52]. Applications of code analysis are used in multiple domains:

- **Bug prevention** Identifying logic errors, type mismatches, and resource leaks early in development.
- **Security** Detecting vulnerabilities like buffer overflows, injection flaws, or insecure dependencies.
- **Compliance** Enforcing coding standards or licensing requirements [38].
- **Maintainability** Assessing technical debt and code smells (e.g., duplicated code and high coupling).
- **Optimization** Profiling performance-critical paths and memory usage patterns.

The importance of code analysis has grown continuously with the rise of large-scale distributed systems, open-source dependencies, and regulatory demands [71]. Research in recent years has focused on enhancing precision (e.g., reducing false positives with machine learning), scaling analysis to huge codebases, and integrating analysis into DevOps pipelines (e.g., shift-left testing [67]). Emerging techniques, such as AI-powered code review and hybrid approaches combining symbolic execution with dynamic fuzzing, have further expanded the capabilities of modern tools.

2.2.1 Static and Dynamic

Code analysis techniques are broadly categorized into static and dynamic analysis, differentiated by whether they examine code *without* execution (static) or *during* execution (dynamic). Both approaches have distinct advantages and limitations, making them complementary in practice [52].

Static analysis inspects source code, bytecode, or binaries without running the program. Tools like linters (e.g., *ESLint*, *Pylint*), type checkers (e.g., *MyPy*), and security scanners (e.g., *SonarQube*) parse code structure to identify syntax and style violations, potential bugs (e.g., null dereferences or resource leaks), security vulnerabilities (e.g., SQL injection patterns), and compliance with coding standards (e.g., *MISRA-C*) [32, 25]. This can help to detect early errors in development and scale to large codebases [49]. However, static analysis tools may produce false positives and cannot analyze runtime behavior.

Dynamic analysis evaluates code during execution, often using instrumentation like debuggers, profilers, or test cases [52, 47]. Tools like *Valgrind* (memory debugging), coverage tools (e.g., *JaCoCo*), and fuzz testers (e.g., *AFL*) can help to detect memory leaks, performance bottlenecks, and input validation flaws [26]. This approach captures real-world behavior but requires representative test inputs and includes runtime overhead. In addition to this, it cannot guarantee full path coverage and may miss latent defects in unexecuted code.

2.2.2 Code Property Graphs

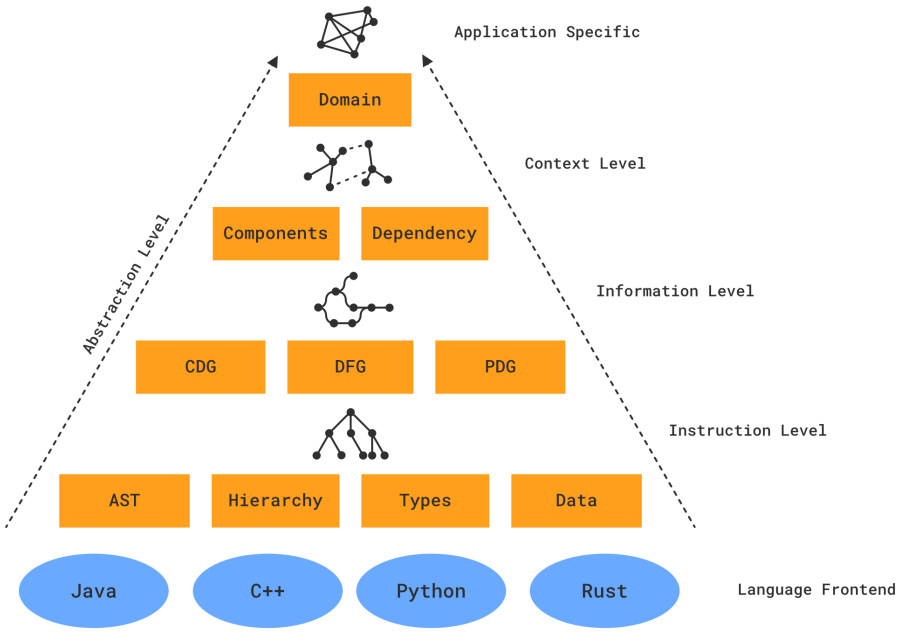


Figure 2.5: The different levels of abstraction and granularity a CPG can represent [72].

A Code Property Graph (CPG) is a unified, graph-based representation of software code that integrates multiple abstract code structures into a single interconnected model. This approach addresses the fragmentation of traditional code analysis, which often operates on isolated representations like syntax trees or control flow graphs. A CPG combines syntactic, semantic, and behavioral properties into a single graph [72]. This enables analysts to traverse and query code across multiple layers of abstraction, making them particularly effective for tasks requiring cross-cutting reasoning, such as vulnerability detection and program comprehension [75, 76, 82]. For this, a CPG merges four fundamental code representations into a single graph:

- **Abstract Syntax Tree (AST):** Captures the syntactic structure of code, with nodes representing language constructs (e.g., functions, loops, variables) and edges denoting syntactic relationships (e.g., parent-child dependencies).
- **Control Flow Graph (CFG):** Models the order of execution between code blocks, with edges representing conditional branches, loops, and function calls.
- **Data Flow Graph (DFG):** Tracks how data propagates through variables, function arguments, and return values, highlighting dependencies between operations.
- **Program Dependence Graph (PDG):** Combines control and data flow dependencies to represent conditions under which data is computed or used.

By unifying these layers, the CPG allows queries to span syntax, control flow, and data flow simultaneously. For example, a taint analysis query for a sensitive function call might trace user input from a syntax-level variable declaration through control flow branches to a data flow sink [84].

The strength of a CPG lies in its ability to abstract away language-specific details while preserving semantic relationships. For individual use cases, the CPG schema enables aggregation to higher-level abstractions for domain-specific analysis (fig. 2.5). A CPG is a multi-graph, with nodes representing code entities (e.g., classes, functions, or variables), and edges encode detailed relationships such as *calls*, *extends*, or *defines*. This abstraction enables:

- **Language-independent analysis:** Queries can be written once and applied to codebases in multiple programming languages, provided the CPG schema normalizes language-specific constructs.
- **Context-aware reasoning:** Combining control and data flow layers allows precise identification of vulnerabilities (e.g., detecting if untrusted data reaches a security-critical operation without proper sanitization).
- **Scalable exploration:** Graph traversal algorithms efficiently navigate complex interactions, such as data leaks or race conditions.

Currently, CPGs are widely used as a static code analysis representation for vulnerability detection, code similarity detection, code comprehension, and automated refactoring [9, 14]. Despite their versatility, the limitations of CPGs include the complexity of construction. The generation of a CPG requires parsing and merging multiple code

representations, which can be resource-intensive for large codebases. Furthermore, normalizing diverse languages into a unified schema may obscure language-specific details (e.g., dynamic and static types), which still is a major challenge in the field [48].

2.3 Graph Theory

As a fundamental area within discrete mathematics, graph theory provides a versatile framework for representing and analyzing a wide range of systems. Its utility in computer science is expressed by the natural way in which graphs model entities and the relationships among them, thereby facilitating the design and analysis of algorithms in areas such as networking, optimization, and data mining [12]. The abstraction offered by graphs allows complex real-world problems to be encapsulated in a structure compatible with both theoretical analysis and practical computation.

2.3.1 Definition

In the context of graph theory, a simple graph G is denoted by the ordered pair $G = (V, E)$, where V is a non-empty set of nodes (or vertices) and E is a set of edges [45]. For graphs without self-loops, each edge is represented as an unordered pair of nodes:

$$E \subseteq \big\{\{u, v\} \text{ (or } (u, v) \text{ for a directed graph)} \mid u, v \in V,\, u \neq v\big\} \tag{2.1}$$

In this formulation, the edge $\{u, v\}$ denotes a bidirectional relationship between the nodes u and v. For directed graphs, the set of edges is a subset of the Cartesian product of V with itself. An edge (u, v) in this context represents a relationship that is oriented from node u to node v.

Expanding the expressiveness of graphs, a multigraph permits multiple edges between the same pair of nodes [45]. Furthermore, many applications necessitate the inclusion of weights to capture quantitative properties of the relationships. A weighted graph augments the basic graph structure with a weight function. In the case of a weighted undirected graph, the graph is defined as the triple $G = (V, E, w)$, where w is the weight function $w : E \to \mathbb{R}$, assigning a real number as the weight to each edge. For weighted directed graphs, the same definition applies with $E \subseteq V \times V$ and the weight function w assigning values to ordered pairs accordingly [45].

A widely used representation of graph relationships is the adjacency matrix [45]. Given a graph $G = (V, E)$ with $|V| = n$ nodes, the adjacency matrix A is an $n \times n$ matrix where each entry A_{ij} encodes the presence or weight of an edge between node v_i and node v_j. For a simple weighted graph, the adjacency matrix is defined as:

$$A_{ij} = \begin{cases} w(v_i, v_j), & \text{if } \{v_i, v_j\} \in E \text{ (or } (v_i, v_j) \in E \text{ for a directed graph)} \\ 0 \text{ or } \infty, & \text{if no edge exists.} \end{cases} \tag{2.2}$$

Here, a missing edge may be represented either by zero or by infinity (if no direct connection exists). If G is unweighted, the adjacency matrix is binary, with $A_{ij} = 1$ if an edge exists between v_i and v_j, and $A_{ij} = 0$ otherwise. The adjacency matrix provides an efficient way to perform matrix-based computations on graphs, such as spectral graph analysis and shortest path calculations. However, for sparse graphs, where $|E| \ll |V|^2$, adjacency lists are often preferred due to their lower space complexity [45].

2.3.2 Properties

A graph has multiple properties that can be used to characterize its structure and behavior [45]. Each of these properties provides valuable insights into the nature of the graph and its relationships (fig. 2.6).

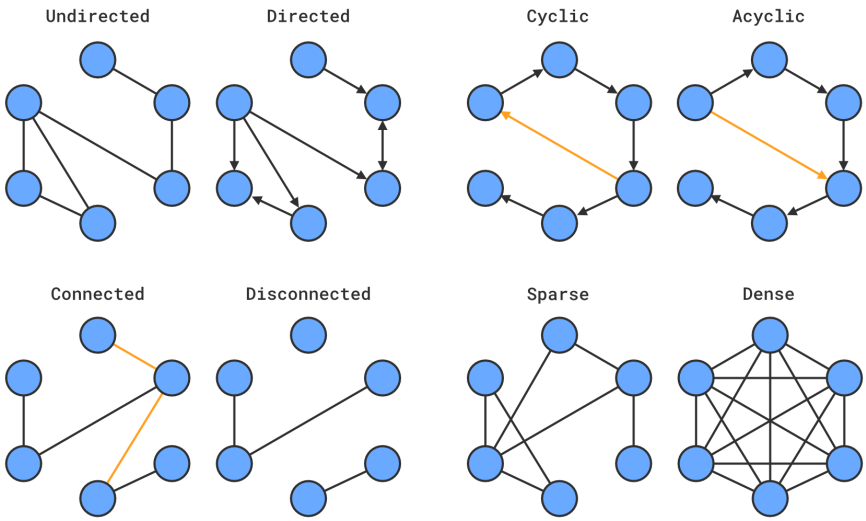


Figure 2.6: Common structural properties of a graph [45].

Connectivity Let $G = (V, E)$ be an undirected graph. A graph G is said to be *connected* if for every pair of distinct nodes $u, v \in V$ there exists a sequence of nodes (a *path*) $(u = v_0, v_1, \ldots, v_k = v)$ such that $\{v_i, v_{i+1}\} \in E \quad \text{for } i = 0, 1, \ldots, k-1$. If no such path exists for at least one pair of nodes, the graph is *disconnected*. In a directed graph, the concept of connectivity is extended to *strongly connected* and *weakly connected*. A directed graph is *strongly connected* if there exists a directed path from every node to every other node. Conversely, a directed graph is *weakly connected* if replacing all directed edges with undirected edges results in a connected graph.

Density The *density* of a graph is a measure of how many edges are present compared to the maximum number of edges possible. For a simple undirected graph $G = (V, E)$ with $|V| = n$ nodes, the density D is defined as

$$D = \frac{2|E|}{n(n-1)} \tag{2.3}$$

since the maximum number of edges in an undirected graph without loops is $(n(n-1))/2$. A graph is often described as *sparse* if D is close to 0 and *dense* if D is close to 1.

Cyclicity A *cycle* in an undirected graph $G = (V, E)$ is defined as a path $(v_0, v_1, \ldots, v_k)$ with $k \geq 3$ such that $v_0 = v_k$ and the nodes $v_0, v_1, \ldots, v_{k-1}$ are distinct. A graph is called *cyclic* if it contains at least one cycle. Conversely, if a graph contains no cycles, it is termed *acyclic*. In the context of directed graphs, a *directed cycle* is defined analogously, with the additional requirement that each consecutive pair of nodes (v_i, v_{i+1}) (and (v_k, v_1) for closure) respects the orientation of the edge.

Isomorphism Graphs are said to be *isomorphic* if they have exactly the same structure, even if their representations differ. This means that there exists a one-to-one mapping between the vertices of the two graphs such that the edges are preserved. Let $G = (V, E)$ and $G' = (V', E')$ be two graphs (either both directed or both undirected). These graphs are *isomorphic* if there exists a bijective function $f : V \to V'$ such that for all $u, v \in V$:

$$\{u, v\} \text{ (or } (u, v) \text{ for a directed graph)} \in E \quad \text{if} \quad \{f(u), f(v)\} \in E' \tag{2.4}$$

This bijection f preserves the adjacency relation, ensuring that the two graphs share the same structure.

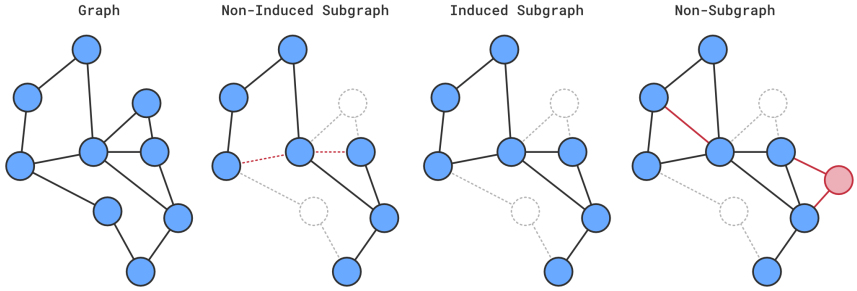


Figure 2.7: Comparison of different types of subgraphs [54].

Subgraph Given a graph $G = (V, E)$, a graph $H = (V_H, E_H)$ is called a *subgraph* of G if

$$V_H \subseteq V \quad \text{and} \quad E_H \subseteq E \tag{2.5}$$

with the additional requirement that for every edge $e \in E_H$, both endpoints of e belong to V_H. An *induced subgraph* is a special type of subgraph that is formed by selecting a subset of nodes from the original graph and including all edges that connect those nodes, thus preserving the original connectivity. Formally, $\forall e \in E$, if $e = (u, v)$ with $u, v \in V_H$, then $e \in E_H$ must hold. In the case of undirected graphs, the same definition applies, but the edges are undirected.

2.3.3 Algorithms

In addition to their structural properties, graphs serve as the foundation for a wide range of algorithms that tackle diverse computational challenges. For instance, the **Connected Components** (CC) algorithm is designed to identify distinct clusters within a graph by partitioning the node set into subsets where every pair of nodes within a subset is connected by some path. This is particularly useful in analyzing social networks or clustering data [65].

Another graph algorithm is **Single Source Shortest Path (SSSP)**, which focuses on finding the shortest paths from a designated source node to all other nodes. Techniques such as the *Dijkstra* algorithm [16] or the *Bellman-Ford* algorithm are widely applied in routing, navigation, and network optimization scenarios.

PageRank (PR) represents a different paradigm, where the goal is to assess the relative importance of nodes within a directed graph. By iteratively updating node scores based on the structure of incoming links, the algorithm effectively quantifies the influence of nodes, a method that has proven instrumental in ranking web pages and analyzing complex networks [23].

Subgraph Matching (SM) addresses the challenge of identifying specific patterns within larger graphs and computes their subgraph isomorphism property. This problem is critical in various domains, including pattern recognition, bioinformatics, and social network analysis, where the ability to detect and extract meaningful substructures can yield significant insights [54].

2.4 Graph Neural Networks

Neural networks have demonstrated remarkable success in various domains, particularly where data can be naturally represented in a Euclidean space [83]. However, many real-world systems involve entities and relationships that are best described as graphs, where the structure and the relationships themselves hold crucial information. Traditional neural networks, such as Convolutional Neural Networks (CNNs) [31], are not inherently designed to capture these relational structures. Graph Neural Networks (GNNs) extend deep learning methodologies to graph-structured data, enabling models to learn from both the node features and the topology of the graph in various domains [33, 74].

2.4.1 Concept

A GNN operates by iteratively updating node representations through a process known as *message-passing* [85]. At each layer, nodes aggregate information from their neighbors to refine their embeddings. The general formulation of message-passing at layer t for all nodes $v \in V$ is given by

$$h_v^{(t)} = \sigma\left(f_{Agg}^{(t)}\left(h_v^{(t-1)},\ \{h_u^{(t-1)} \mid u \in \mathcal{N}(v)\}\right)\right) \tag{2.6}$$

where $h_v^{(t)}$ is the embedding of the node v at the layer t, $\mathcal{N}(v)$ denotes the set of neighbors of v, $f_{Agg}^{(t)}$ is the aggregation function, and σ is the node update function. The choice of the aggregation function (e.g., mean, sum, max pooling) affects model expressiveness

[68]. Graph learning tasks can be categorized into three levels: *node-level* tasks, such as node classification and node clustering, *edge-level* tasks, including link prediction and edge classification, and *graph-level* tasks, such as graph classification and regression.

2.4.2 Approaches

In recent years, various GNN architectures have been proposed, each introducing different mechanisms to enhance expressivity and efficiency. A **Graph Convolutional Network (GCN)** [8, 28] is based on spectral graph theory and employs a simplified layer-wise propagation rule given by

$$H^{(t)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(t-1)}W^{(t)}\right) \tag{2.7}$$

where $\tilde{A} = A + I$ is the adjacency matrix with self-loops, $\tilde{D}$ is its degree matrix, $H^{(t)}$ is the node feature matrix at layer t, and $W^{(t)}$ is the learnable weight matrix. Another influential architecture is the **Graph Attention Network (GAT)** [62, 36], which incorporates attention mechanisms to dynamically weight the importance of neighbors [63]. The importance for each edge e to the neighbors v_j of a node v_i is expressed as

$$e(h_i, h_j) = \text{LeakyReLU}(a^T[Wh_i||Wh_j]) \quad \text{with} \quad \text{LeakyReLU}(x) = \max(\alpha x, x) \tag{2.8}$$

where h_i and h_j are the feature vectors of the nodes v_i and v_j, W is a learnable weight matrix, $||$ denotes vector concatenation, and a is a learnable attention vector. The final attention coefficient is defined as

$$\alpha_{ij} = \text{softmax}_j(e(h_i, h_j)) = \frac{\exp(e(h_i, h_j))}{\sum_{v_{j'} \in \mathcal{N}(i)} \exp(e(h_i, h_{j'}))} \tag{2.9}$$

which normalizes the attention scores across all neighbors $\mathcal{N}(i)$ of the node v_i using *softmax*. By allowing different neighbors to contribute unequally to a node's representation, a GAT enhances the expressiveness of GNNs.

3 Related Work

This chapter reviews key research areas relevant to this thesis. It summarizes foundational concepts, categorizes existing approaches, and highlights important contributions. The chapter outlines the research in the area of design pattern detection (section 3.1), followed by approaches to subgraph matching (section 3.2) and code property graphs (section 3.3) that are influential to the thesis methodology.

3.1 Design Pattern Detection

Design pattern detection is essential in software engineering for identifying recurring structures in object-oriented systems. These patterns offer solutions to common design challenges, enhancing readability, maintenance, and reverse engineering by recovering lost design knowledge and promoting code reusability [20]. Various rule-based and machine learning-based approaches have been proposed [51].

Rule-based approaches Rule-based approaches rely on predefined rules to detect design patterns in software systems. These rules are typically derived from the formal definition of the design pattern, specifying the structural constraints that must be satisfied by the pattern instances. Several rule-based approaches have been developed over the years. Some approaches use pattern-specific heuristics to identify pattern-specific properties [22, 3, 15], while others leverage structural analysis techniques [58, 60]. One early example is a similarity-scoring-based approach [66], which represents software systems as graphs and applies an iterative similarity scoring between graph vertices to identify design pattern instances. To handle large-scale applications, the approach reduces the computational complexity by partitioning systems into inheritance hierarchies. *PatternScout* [50] is another approach and utilizes SPARQL queries, generated by UML representations of design patterns. For the detection of the patterns, the queries are used on a Resource Description Framework (RDF) representation of the source code. For describing design

Table 3.1: Overview of design pattern detection approaches.

Approach	Year	Category	Methodology
Similarity Scoring [66]	2006	Rule-based	Graphs (adjacency matrices)
PINOT [58]	2006	Rule-based	Structural & control-flow graphs
DeMIMA [22]	2008	Rule-based	Binary class relationships & model constraints
Sempatrec [3]	2014	Rule-based	Ontology-based model (OWL/SWRL)
DSL-driven Graph Matching [7]	2014	Rule-based	Attributed graphs (DSL meta-model)
FINDER [15]	2015	Rule-based	QL-scripts
Predicting Architectural DPs [29]	2022	ML-based	Various ML-models
Ex-DPDFE [30]	2022	Rule-based	Feature-based
DPDF [44]	2022	ML-based	Word2Vec
PatternScout [50]	2022	Rule-based	UML & SPARQL queries
DPDT [60]	2022	Rule-based	Graph matching & static analysis
Neural Sub-graph Matching [4]	2022	ML-based	Subgraph matching with DSL-patterns

patterns, Domain Specific Languages (DSLs) have been implemented, where design patterns are modeled as attributed graphs, and detection is performed through a subgraph matching algorithm [7].

Despite their effectiveness, rule-based methods have several limitations. One significant drawback is their limited adaptability to variations in pattern implementations. Since these methods depend on predefined structural rules, they often fail to detect non-standard or modified implementations of design patterns, which may deviate from canonical structures due to software evolution or different coding practices [20]. Additionally, the increase in computational overhead when parsing large codebases poses scalability challenges [78]. Another challenge is the significant manual effort required for fine-tuning rules. Because rule-based approaches rely on handcrafted heuristics, developers often need to refine and adjust them for different programming languages and software domains, leading to additional maintenance and adaptability concerns [51].

ML-based approaches Approaches based on machine learning leverage the potential of ML models and algorithms to effectively learn the design patterns. These approaches often extract structural and behavioral features from source code and train classifiers to recognize design pattern instances or utilize Graph Neural Networks. For architectural design patterns like the *Model View Controller*, various machine learning models were evaluated to detect these patterns in Java-based Android projects [29]. The evaluated models include *Support Vector Machines*, *Random Forest*, and *Naive Bayes*. Key findings indicate that source code metrics like class methods, inheritance depth, and coupling are strong predictors of architectural patterns. The approach outperforms prior research, offering a more automated and accurate method for architecture detection. However, it is limited to Java and only two patterns. *DPDF* [44] is another approach that uses machine learning to detect design patterns in Java source code. The approach combines structural and lexical code features to construct a semantic representation of the source code. This representation is then processed using *Word2Vec* to generate a word-space model, capturing relationships between classes, methods, and design patterns. A supervised machine learning classifier is trained on a labeled dataset to identify a wide range of GoF design patterns. The approach highlights the importance of lexical features in pattern detection and suggests that combining structural and semantic analysis significantly enhances accuracy. One of the most recent approaches combines a GNN and subgraph matching [4]. This approach models both software systems and design patterns as graphs and generates embeddings using a *Order Embedding* GNN [40] to perform subgraph matching efficiently instead of relying on heuristics. The detection process involves extracting source code structures, transforming them into graphs, and matching them with design pattern templates in an embedding space. The design pattern templates are implemented as language-specific DSL representations.

Despite their advancements, ML-based approaches face several challenges. Dataset imbalance causes biased performance as underrepresented design patterns lead to inconsistent detection rates [44]. Labeling errors from subjective annotators further affect model accuracy, even when multiple annotators are involved [44]. Additionally, while some models achieve high precision for frequently occurring patterns, they often struggle with rare or complex cases, leading to overfitting and poor generalization [29]. Furthermore, these detection methods typically lack explainability. This affects especially deep learning models, which act as black boxes and obscure the reasoning behind pattern detection [29]. Finally, accurately representing intricate class relationships through graph-based methods is computationally expensive and prone to errors, reducing overall detection reliability [4].

3.2 Subgraph Matching

Subgraph matching is a core problem in graph theory that aims to identify occurrences of a smaller query graph within a larger target graph. It is widely employed in domains such as pattern recognition and large-scale data analysis, but its NP-completeness makes it computationally demanding [56]. Various exact and approximate approaches have been proposed to handle its complexity.

Before the advent of machine learning, research centered on exact methods (e.g., clique-based search, backtracking, and dynamic programming) and optimizations for subgraph isomorphism detection. Notable among these were *Maximum Common Subgraph* (MCS) approaches [54], the memory-efficient *VF2* algorithm [13], and distributed systems like *STwig* [64] that used query decomposition for large-scale graphs. Methods also extended into frequent subgraph mining and pattern detection. For instance, *GraMi* [17] formulated frequency evaluation as a constraint satisfaction problem to avoid exhaustive enumeration. Recently, machine learning methods have transformed subgraph matching by learning effective node and edge representations. Graph Convolutional Networks (GCNs) with *Dual Message Passing* [35] enhanced matching and counting accuracy, especially on heterogeneous graphs. Further progress involved reinforcement learning for query optimization, as exemplified by *RL-QVO* [70], which used Graph Neural Networks and *Markov Decision Processes* to learn optimal vertex orderings and reduce search costs.

NeuroMatch One of the most promising approaches to subgraph matching is *NeuroMatch* [56]. *NeuroMatch* leverages a scalable and efficient Graph Neural Network framework to learn an embedding space in which subgraph matching translates directly into comparing graph embeddings, thereby circumventing the need for expensive combinatorial searches. A key insight of this method is its use of *Partial Order Embeddings* [40] to enforce geometric constraints that reflect subgraph relations. By embedding subgraphs in a way that respects partial ordering, *NeuroMatch* can effectively capture structural hierarchies and preserve the relational information crucial for accurate matching. In addition, NeuroMatch incorporates the concept from Identity-Aware Graph Neural Networks (ID-GNNs) [80]. Whereas traditional GNNs often struggle to distinguish between structurally similar nodes, ID-GNNs include the node identity information in the message-passing process. By integrating this identity-aware mechanism, *NeuroMatch* enhances its expressiveness to differentiate nodes that share similar structural roles but play distinct parts in their respective subgraphs.

xNeuSM Another promising approximative machine learning approach is *xNeuSM* [46], which builds on prior multi-hop attention-based GNN architectures to subgraph isomorphism detection and explanation. Unlike earlier methods that depend on fixed attention decay factors, *xNeuSM* introduces a Graph Learnable Multi-hop Attention Network (GLeMA Net), allowing the model to adaptively learn node-specific attention decay parameters. This flexibility enables *xNeuSM* to capture relational dependencies across multiple hops more effectively, leading to significant gains in subgraph matching accuracy and interpretability. Traditional subgraph matching techniques typically rely on exact combinatorial methods, which are prohibitively expensive for large graphs, or on approximate methods that often lack interpretability. To address this challenge, *xNeuSM* adopts a multi-task learning framework that simultaneously optimizes for subgraph detection and explicit node correspondence prediction. This joint objective strikes a balance between efficiency and explainability, distinguishing *xNeuSM* from earlier neural-based approaches focused primarily on classification. The foundation of *xNeuSM* is based on the Multi-hop Attention Graph Neural Network (MAGNA) [69], which first introduced multi-hop attention diffusion to widen the receptive field of GNNs beyond direct neighbors. MAGNA showed that leveraging multi-hop attention can capture long-range dependencies without over-smoothing. Building on this insight, *xNeuSM* replaces the globally fixed decay parameter of MAGNA with node-specific learnable attention decay, further enhancing its ability to model the structural details of different graphs. This node-level adaptability not only improves subgraph matching performance but also provides more transparent insights into how various parts of the graph contribute to the matching process.

3.3 Code Property Graphs

The concept of Code Property Graphs (CPGs) was introduced as a method to improve vulnerability detection and program analysis by integrating multiple traditional program representations into a unified graph-based structure. There are various implementations of CPGs, each with its unique features and applications.

Joern The work that initially proposed CPGs [76] and is implemented in the open-source project *Joern* presented a CPG as a combination of Abstract Syntax Trees (ASTs),

Control Flow Graphs (CFGs), and Program Dependence Graphs (PDGs). This integration allows for a more comprehensive representation of source code, enabling security analysts to identify vulnerabilities more effectively. By using graph traversal techniques, CPGs facilitate the discovery of security flaws such as buffer overflows, integer overflows, and memory disclosures. The approach was implemented using a graph database, allowing efficient querying of large codebases.

Fraunhofer Building upon this foundational work, the approach of *Fraunhofer AISEC* [72] extended the CPG approach to a language-independent analysis platform. Their platform adapts CPGs to support multiple programming languages, making it suitable for heterogeneous software environments. A key advancement introduced in this work is the Evaluation Order Graph (EOG), which refines control flow modeling by capturing execution order at a finer granularity than traditional CFGs. This enhancement enables a more precise understanding of program semantics, facilitating data flow analysis across different programming paradigms. Furthermore, the platform incorporates fuzzy parsing, allowing it to analyze incomplete or non-compilable code, which is particularly useful in early development stages or during security audits of partial source code.

The evolution of CPGs from a targeted vulnerability detection tool to a versatile analysis framework demonstrates their growing relevance in software security and compliance checking. Building on these foundational CPG concepts, a wide range of research has employed CPGs to tackle various software engineering and security challenges. Several works focus on vulnerability detection and secure code analysis. For instance, deep learning-based approaches integrate AST, CFG, and data-flow representations to automatically detect security flaws [75]. Others employ subgraph isomorphism to identify vulnerable code clones at scale, pruning large CPGs for efficiency [73]. Extending these ideas further, cross-project vulnerability detection frameworks leverage domain adaptation and graph attention mechanisms to transfer knowledge across different codebases [82]. Beyond vulnerability detection, CPGs are also used for bug predictions and refactoring recommendations [14]. Some approaches leverage source code graph structures to capture incremental code change complexities, enabling just-in-time bug prediction [42]. Others integrate deep semantic features from CPGs with attention-based models to localize defects more accurately than traditional methods [24]. By modeling dependencies, control flow, and syntactic structures together, these techniques achieve higher precision and recall. For concurrency challenges, specific graph extensions capture thread synchronization and inter-thread dependencies [9].

4 Concepts

This chapter provides the core concepts for the proposed approach to design pattern detection. It combines techniques and approaches from the discussed related works (chapter 3) and extends them to create a language-independent mechanism for recognizing design patterns. The subsequent sections explain how source code is represented as a graph-based abstraction, how neural subgraph matching identifies patterns, and how a pattern voting method aggregates pattern predictions across multiple examples.

4.1 Approach

The proposed approach aims to meet the following requirements for the design pattern detection to extend and improve on previous works.

- **General Pattern Detection:** The approach should be able to detect design patterns without relying on handcrafted pattern definitions or templates. Instead, it should extract and capture the inherent structural properties of design patterns from existing examples.
- **Language-Independence:** The approach should be language-independent, meaning the underlying representation should remain consistent and valid across different programming languages. Specific properties must be abstracted to a more general representation.
- **Code Resilience:** The approach should exhibit code resilience and remain functional with incomplete or non-compilable source code. It should handle errors or syntax anomalies gracefully, without breaking the detection process.

To meet those requirements, the approach generates a high-level, graph-based representation of the source code. For this, a Code Property Graph (CPG) is used. The CPG will then be aggregated into a domain-specific Record Interaction Graph (RIG), which

captures relevant information on how various code entities (or *records*) interact together (section 4.2). Once this abstraction is obtained, the approach uses subgraph matching that leverages a Graph Neural Network to efficiently identify structures in the RIG corresponding to a given query pattern (section 4.3). Finally, a pattern voting algorithm is used to decide whether a pattern is present or not. For the queried patterns, the approach extracts and normalizes common design pattern graphs from a set of examples. This allows an expandable approach for other patterns (section 4.4).

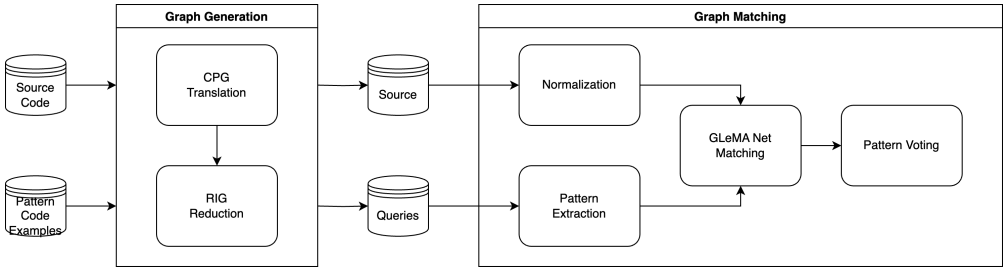


Figure 4.1: Overview of the proposed approach.

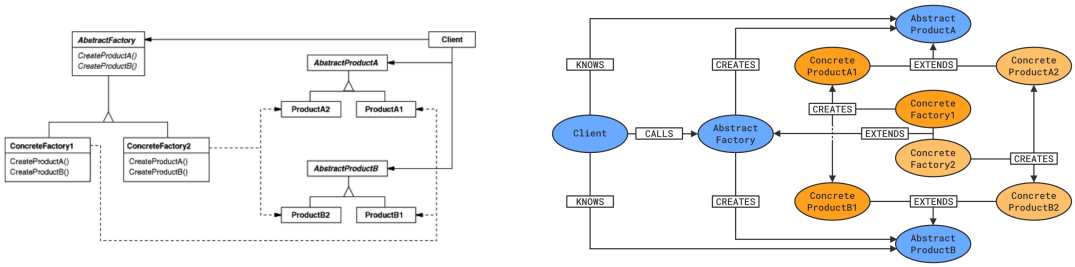
The reasoning behind the used concepts of the proposed approach follows from prior research in code property graphs and graph-based pattern matching. CPGs offer a powerful, language-independent abstraction enriched with multiple aspects of source code, such as control flow, data flow, and internal dependencies. Some existing CPG implementations are tolerant of compilation errors and partial code, which supports the code resilience requirement. This is the reason why the approach of *Fraunhofer AISEC* [72] is used for this problem. While theoretically NP-complete [56], subgraph matching can be handled efficiently in many practical contexts by using specialized Graph Neural Network architectures. For this, the *GLeMA Net* will be used [46], which has shown a high accuracy in subgraph matching tasks and can be easily adapted to different graph structures and node features. By combining these techniques with a pattern voting mechanism, the approach uses the ability of machine learning to discover common structural motifs and unify them into robust design pattern signatures.

4.2 Record Interaction Graph

To detect design patterns with a Graph Neural Network, the source code has to be abstracted into a graph-based representation. For this, the proposed approach translates the source code into a Code Property Graph. This representation captures a wide range

of information by combining multiple aspects of the code (section 2.2.2). However, the CPG can quickly grow to large sizes and contain irrelevant information for the purpose of design pattern detection. To reduce the complexity and focus on the relevant information, the CPG has to be further processed and aggregated to a more concentrated graph abstraction the neural network can work with. For this purpose, a domain-specific abstraction is introduced, the Record Interaction Graph (RIG). The RIG captures the interactions between code entities, focusing on aspects like entity dependencies, visibility, inheritance, and behavior. The assumption for this abstraction is that the interactions between code entities are the identifying features of design patterns.

4.2.1 Record Interactions



(a) GoF [19] (b) RIG

Figure 4.2: Comparison between the GoF definition and the RIG of an *Abstract Factory*.

A *record* in an object-oriented context commonly refers to a code construct such as a class, struct, entity, object, or instance that can be instantiated and holds data as well as optional behavior. This broad definition emphasizes any self-contained unit within the source code that describes a distinct set of attributes and methods. By encapsulating these attributes and methods, a record promotes modularity and clarity in the overall design, enabling the concept to serve as a fundamental building block for object-oriented systems. Design patterns in object-oriented systems often revolve around the interactions between these records. Typically, a design pattern aims to improve maintainability, reusability, and extensibility by reducing coupling and increasing cohesion. This is achieved by defining clear visibility rules and communication channels between records, ensuring that other records do not depend on concrete implementations that are not necessary for their functionality. To achieve this, design patterns often introduce intermediary abstractions, such as interfaces or abstract classes, to decouple records from specific implementations and enforce modularity.

In the proposed approach, a *record interaction* is defined as any direct operation involving another record, such as:

- **Extending a class:** Extending a class indicates an inheritance relationship, where the subclass gains access to the properties and methods of the superclass. This relationship is important for recognizing hierarchical connections in design patterns, as inheritance can be a fundamental structural feature.
- **Creating an instance:** Instantiating another record establishes a direct dependency because one record explicitly controls the creation of the other. This process is often used in patterns that revolve around object creation, such as the Factory Method or Abstract Factory.
- **Returning an instance:** Methods that return instances create a dependency from the caller's perspective, tying the caller to that returned record. This interaction can indicate roles in patterns that involve transferring or sharing objects, underscoring how records circulate in the system.
- **Accessing instance properties:** Accessing a record's properties typically means reading its state, which implies a less invasive form of interaction than direct modification. Nevertheless, it remains a crucial indicator of how records depend on one another's data, especially in patterns where data sharing or observation plays a central role.
- **Calling methods on an instance:** Invoking another record's methods signifies a functional dependency between the caller and the callee. It represents a direct link in the control flow and is often central in patterns that coordinate behavior across multiple collaborating records.
- **Referencing a record:** Holding a reference to another record (e.g., storing it in a field) establishes a structural connection between the two. This reference-based link is instrumental for identifying composition or association in design patterns, clarifying how records relate to one another within the larger system.

It is crucial to distinguish direct interactions from indirect ones, such as the use of an interface or a superclass without referencing a specific implementation. In those cases, the interaction is tied to the abstraction rather than to a particular record instance. For the purposes of the proposed concept, only direct interactions contribute to the Record Interaction Graph, given that the central assumption is that concrete interactions

among records are key to identifying design pattern roles and responsibilities. The GoF patterns are defined by the relationships between the records and the specific roles of the records. The roles and relationships can be captured by the direct interactions between the records. For example, in the Abstract Factory pattern, the record's role *ConcreteFactory* extends the superclass with the role *AbstractFactory*, which creates instances of the role *AbstractProduct*. The role *ConcreteProduct* creates instances of the role *ConcreteProduct*. Only the *AbstractFactory* is visible and can be called by the record with the role *Client*. With this description, the roles for each record are implicitly defined by the interactions between the records (fig. 4.2).

4.2.2 Code Property Graph Abstraction

In the proposed approach, the Record Interaction Graph is a high-level abstraction layer of a Code Property Graph. For this purpose, the CPG is filtered and aggregated to depict the interactions between records, capturing all edge information in the graph. This includes the AST, CFG, DFG, and PDG edges that are relevant for the interactions between records.

Given a CPG $G = (V, E, \ell_V, \ell_E)$, where:

- V is a finite set of *nodes*
- $E \subseteq V \times V$ is a set of *edges*
- $\ell_V : V \to \Sigma_V$ is a *node labeling function* that assigns a label from some set Σ_V to each node
- $\ell_E : E \to \Sigma_E$ is an *edge labeling function* that assigns a label from some set Σ_E to each edge

Assuming there exists a label $RECORD \in \Sigma_V$, which defines a record node, the set of all records $R \subseteq V$ is defined as $R = \{r \in V \mid \ell_V(r) = RECORD\}$. The RIG is a directed multi-graph $G' = (V', E', \ell_{E'})$ and captures the aggregated paths between all records in R, so that V' is defined as $V' \subseteq R$. The record interactions are captured by the edges $E' \subseteq V' \times V'$, where each edge $(r_1, r_2) \in E'$ represents a direct interaction between two records $r_1, r_2 \in V'$ and the interaction is labelled by the edge labeling function $\ell_{E'}$.

In order to capture all relevant interactions between records in G, all other nodes $v \notin R$ have to be assigned to the scope of any record $r \in R$. This is because the RIG

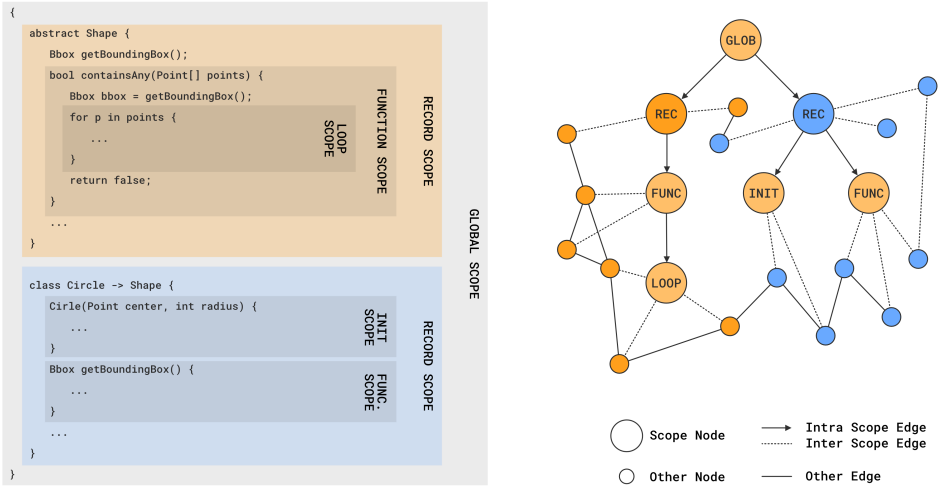
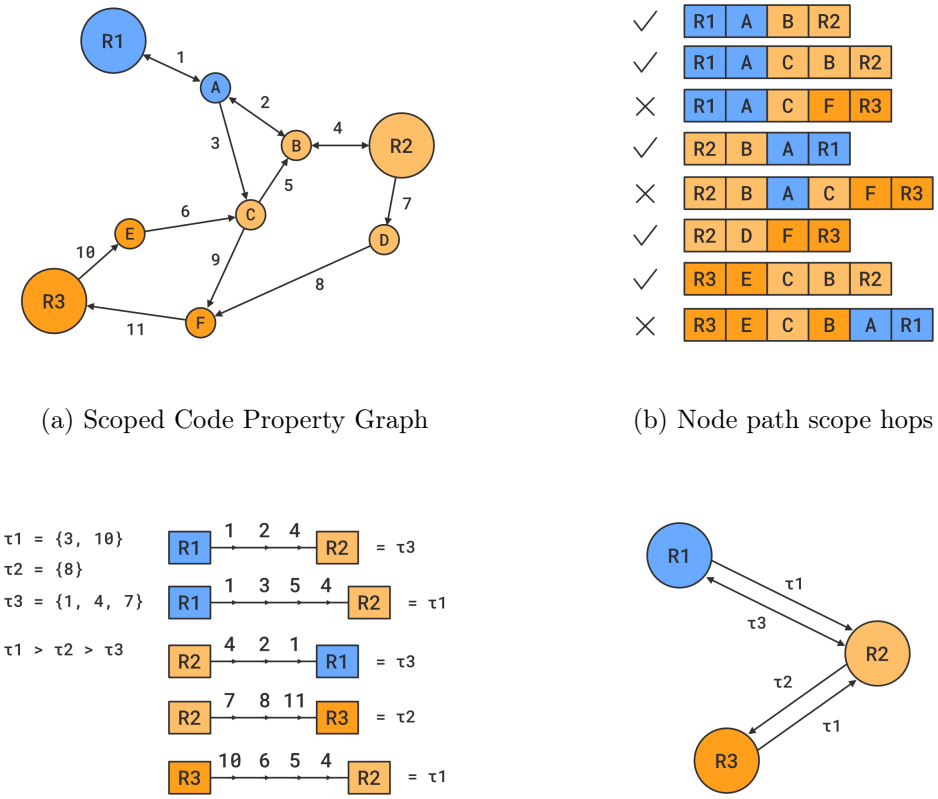


Figure 4.3: Illustration of the scope tree contained in a CPG and the corresponding color-coded scope associations for the nodes.

captures the direct interactions between the records, which means that any path between records should only change the record scope once to capture the interaction between directly reachable neighbor records. For this, the CPG definition of *Fraunhofer AISEC* [72] includes the concept of scopes. Assuming there exists a set of node scope labels $\Sigma_S \subseteq \Sigma_V$, then for every node $v \in G$, there exists a directed edge $e = (v, v_S)$, where $\ell_V(v_S) \in \Sigma_S$. The scopes in the given CPG definition include a wide range of types, for example, *GLOBAL SCOPE*, *RECORD SCOPE*, *FUNCTION SCOPE*, *BLOCK SCOPE*, and *LOOP SCOPE*. All the scope types have a hierarchical order, where scopes like *GLOBAL SCOPE* are the highest scope, and the *LOOP SCOPE* is further down. This scope structure can be expressed as a directed tree graph. For the RIG, the record scope for every node must be determined, which is a transitive and recursive relation in the scope tree of G. To determine the record scope of a node $v \in G$, there exists a recursive function

$$\ell_{RS}(v) = \begin{cases} \ell_V(v_S) & \text{if } \ell_V(v_S) = \textit{RECORD SCOPE}, \\ \ell_{RS}(v_S) & \text{otherwise}, \end{cases} \tag{4.1}$$

where v_S is the directly associated scope node of v in G. This function recursively traverses the scope tree of G until it reaches a node with the label *RECORD SCOPE*. According to the CPG definition, it is guaranteed that there exists a record scope declaration for every node in G by traversing the graph in this way.



(a) Scoped Code Property Graph

(b) Node path scope hops

(c) Edge path interactions

(d) Final Record Interaction Graph

Figure 4.4: Steps to construct the Record Interaction Graph from a Code Property Graph.

Given the record scopes for each node in G, the direct paths between records can be determined. For this, a path P_V is defined as a finite sequence of nodes $P_V = (v_1, v_2, \ldots, v_n)$ with $n \geq 2$. The start node v_1 and the end node v_n are both records, *i.e.*, $v_1, v_n \in R$, while any node strictly between v_1 and v_n is not a record, *i.e.*, $\forall\, 2 \leq i \leq n-1 : v_i \notin R$. For all in-between nodes v_i, the record scope is constrained to one of the record scopes of the start and end nodes, so that $\ell_{RS}(v_i) \in \{\ell_{RS}(v_1), \ell_{RS}(v_n)\}$ holds. Moreover, for each consecutive pair of nodes in P_V, $(v_i, v_{i+1}) \in E$ holds for all $1 \leq i < n$. Note that the start node and end node are part of R, with any node in between not belonging to R, and all nodes in P_V (except the start) are connected to the previous node by a directed

edge in E. This implies the following properties for an aggregated path P_V and its start and end nodes $v_1, v_n \in P_V$:

- v_1 has a directed connection to v_n
- v_1 has an *n-hop* distance to v_n
- v_n is one of the next reachable records for v_1
- The record scope in P changes only once

If such an aggregated path P_V exists with $v_1, v_n \in R$, then the proposed concept includes an edge $\big(v_1, v_n\big)$ in E'. In this way, every existing *n-hop* connection between all records and every neighboring record is captured in the RIG.

To determine the type of interaction of a path P_V, there exists a mapping function

$$\mu : (v_1, v_2, \ldots, v_n) \mapsto \big((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\big) \tag{4.2}$$

which transforms the node path P_V into an edge path $P_E = (e_1, e_2, \ldots, e_{n-1})$, where each $e_i = (v_i, v_{i+1})$. The path P_E can now be reduced to a single interaction type, which is determined by the edge labeling function ℓ'_E. Let $\mathcal{T}$ be the set of all possible interaction types, and for each $\tau \in \mathcal{T}$, let $\Sigma_\tau \subseteq \Sigma_E$ be the *identity labels* that characterize τ. For a given edge path $P_E = (e_1, e_2, \ldots, e_{n-1})$, there exists at least one edge e_i whose label $\ell_E(e_i)$ belongs to the identity labels of a certain interaction type τ, then the aggregated edge $e \in E'$ is assigned the interaction type τ. The edge labeling function $\ell_{E'}$ is defined as:

$$\ell_{E'}(P_E) := \chi\Big(\big\{\tau \in \mathcal{T} \;\big|\; \exists\, e \in P_E : \ell_E(e) \in \Sigma_\tau\big\}\Big) \tag{4.3}$$

The χ function in this case picks the relevant interaction type τ from the set of all possible types for the given path by applying a predefined priority rule for each $\tau \in \mathcal{T}$. This rule is defined by the domain-specific requirements of the Record Interaction Graph, which can be adjusted to emphasize certain interactions over others. For example, the interaction type *CREATION* might be prioritized over *ACCESS* in a pattern detection context, as the former is more expressive for object creation roles.

4.3 Neural Subgraph Matching

Neural subgraph matching serves as the primary technique for identifying design patterns within the proposed Record Interaction Graph. The proposed approach adapts the *xNeuSM* model architecture, which builds on prior multi-hop attention-based Graph Neural Network designs [69]. *xNeuSM* leverages a Graph Learnable Multi-hop Attention Network (GLeMA Net) to tackle subgraph isomorphism detection and explanation [46]. Unlike earlier methods that used fixed attention decay factors, the *xNeuSM* model introduces node-specific learnable attention decay parameters. This adaptive mechanism captures relational dependencies across multiple hops more effectively, resulting in significant improvements in both matching accuracy and interpretability. The architecture employs a multi-task learning framework that optimizes for subgraph detection and explicit node correspondence prediction in parallel.

In addition to the base model architecture, the proposed approach incorporates a modification that anchors subgraphs at a specific node. This anchoring technique associates the identity of a detected subgraph with a particular record in the graph and enhances the interpretability of the matching process. By design, the concept not only identifies a matching substructure but also ties it to a central node, which serves as a reference point for the subgraph. This modification is mandatory for the subsequent pattern voting mechanism, which relies on the identification of pattern occurrences and their anchoring to specific records in the Record Interaction Graph.

4.3.1 Model Architecture

The proposed approach uses the GLeMA Net model. Unlike standard Graph Neural Networks that focus on immediate neighbors, it employs a multistep attention mechanism to capture deeper structural relationships and improve pattern recognition. The model takes as input node features and two types of adjacency matrices. Node features represent the properties of graph nodes and also indicate whether a node belongs to the pattern or the target graph. Each node is represented by a vector with two parts, one for the query and one for the target graph, forming a combined vector with dimension $2T_V$. The first adjacency matrix is called the **intra-graph** matrix and represents connections between nodes within the same graph. An entry in this matrix is set to 1 if two nodes are directly or indirectly connected within the same graph and 0 otherwise. The second adjacency matrix is the proxy cross-graph matrix, or **inter-graph** matrix, and includes

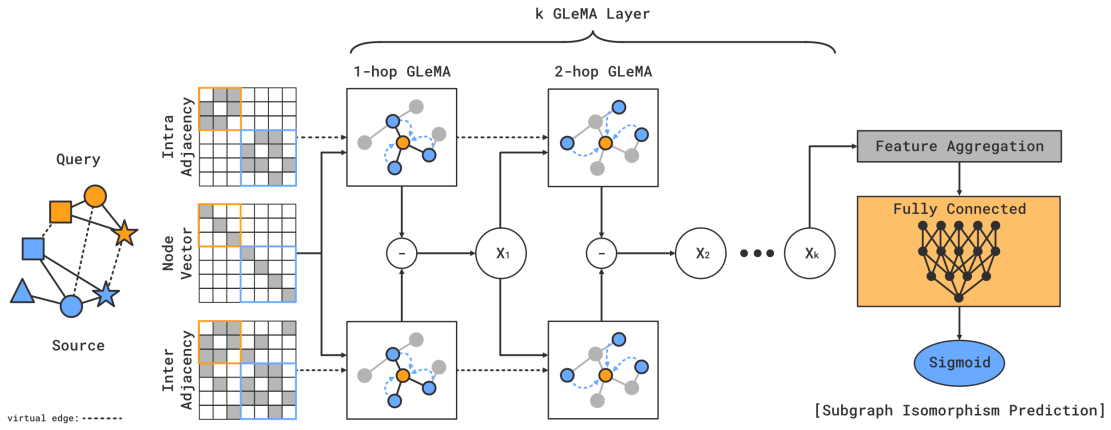


Figure 4.5: Overview of the GLeMA Net architecture [46].

all connections from the intra-graph matrix but additionally connects nodes from the other graph if they have matching labels. These virtual edge connections allow the model to directly compare nodes from the query and the target graph (fig. 4.5).

Initially, node features are transformed into a more informative embedding space using a simple linear transformation. Following this, the model calculates attention scores between node pairs using a method similar to *Luong's* attention, which weights the relevance of nodes based on their embeddings. These scores indicate how strongly each node pair influences each other and are then normalized to ensure that only immediate neighbors affect a node's representation. To incorporate information from nodes located multiple hops away, the model uses a multistep diffusion of attention. This diffusion aggregates information progressively from nodes farther away in the graph, weighting their contributions according to learned decay factors. The attention diffusion matrix A aggregates multiple steps of attention as follows:

$$A = \sum_{k=0}^{\infty} \theta_k (A^{(1)})^k \quad \text{with} \quad \theta_k = \alpha(1-\alpha)^k \tag{4.4}$$

Here, $A^{(1)}$ represents the normalized attention matrix computed at a single step (1-hop), and the term $(A^{(1)})^k$ reflects the attention propagated across k steps in the graph. The decay factor θ_k determines how much influence nodes at distance k have, where α is the teleport probability controlling the rate of decay. In practice, rather than summing infinitely many steps, the diffusion is approximated through iterative computations to ensure computational feasibility while still capturing distant relationships effectively.

A significant innovation in the proposed approach is learning a distinct decay factor for each node, rather than using a fixed decay factor for all nodes. This allows the model to adaptively decide the importance of distant nodes based on local graph structure. The GLeMA Net architecture applies multiple such layers sequentially, processing both intra-graph and inter-graph information separately in parallel. Afterward, it explicitly calculates the difference between these two perspectives to highlight discrepancies, which aids in identifying accurate matches between the subgraph queries and the target graph. These refined node embeddings are aggregated into a single representation that the model uses for two tasks: predicting if a subgraph match exists and explaining this prediction through node-level alignment. Both tasks are optimized simultaneously, guided by a loss function that combines prediction accuracy with alignment precision.

4.3.2 Record Anchoring

The core idea behind anchoring in the proposed approach is to resolve the ambiguity inherent in subgraph representations by explicitly marking a designated *anchor* node within each extracted subgraph. In standard Graph Neural Network formulations, nodes with similar local topologies are often assigned nearly identical embeddings, which can lead to indistinguishability in tasks such as design pattern detection. By augmenting the one-hot node features with an additional anchor indicator, the approach enforces a structural asymmetry that preserves the identity of the central node. For instance, if the original one-hot feature of node v is denoted by x_v, the augmented feature $\tilde{x}_v$ is defined as

$$\tilde{x}_v = \begin{cases} [x_v; 1] & \text{if } v \text{ is the anchor,} \\ [x_v; 0] & \text{otherwise.} \end{cases} \tag{4.5}$$

This modification ensures that during the message-passing process, the anchor node is treated differently from its neighbors.

In the adapted GLeMA Net architecture, subgraph matching is carried out by embedding a k-hop neighborhood G_v around each node v in the target graph. The network computes node embeddings via an iterative message-passing scheme. However, by designating one node as the anchor, the message functions can be made sensitive to its special role.

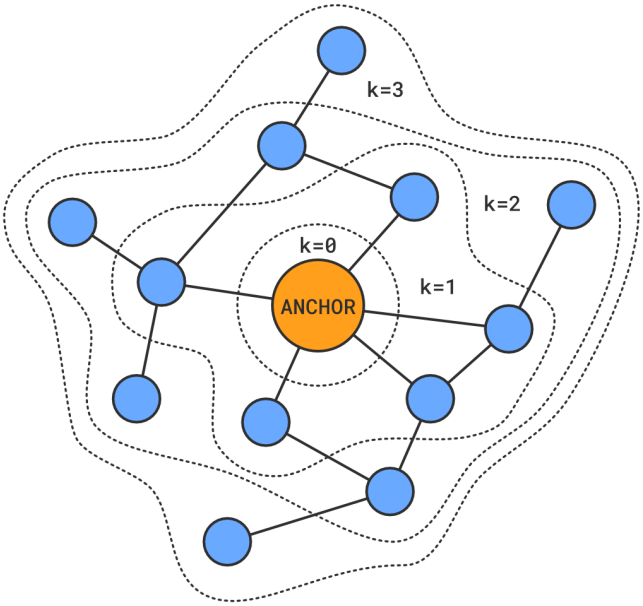


Figure 4.6: An anchored node with its k-hop neighborhood.

In particular, if $h_v^{(k-1)}$ denotes the node embedding at layer $k-1$ and $\mathcal{N}(u)$ is the neighborhood of node u, the message update can be formulated as

$$h_u^{(k)} = f_{Agg}^{(k)}\big(\{m_v^{(k)} : v \in \mathcal{N}(u)\}\big) \quad \text{with} \quad m_u^{(k)} = \sum_{v \in \mathcal{N}(u)} f_{Msg}^{(k)}\big(\tilde{h}_v^{(k-1)}\big) \tag{4.6}$$

where $\tilde{h}_v^{(k-1)}$ is computed using the augmented features $\tilde{x}_v$. By incorporating the augmented features, the GNN is provided with explicit information about which node in a subgraph is the anchor. This differentiation allows the network to learn distinct transformation parameters for the anchor compared to its neighbors, thereby improving the ability to identify and compare the structure of subgraphs.

In practical terms, anchoring enables the following improvements:

- **Disambiguation of Subgraph Structure:** By marking one node as the anchor, the model can resolve ambiguities that arise when multiple nodes share similar local topologies.

- **Enhanced Interpretability:** Each subgraph's representation becomes tied to a specific record, which is critical for downstream tasks like pattern voting in design pattern detection.
- **Improved Matching Accuracy:** In subgraph matching, the query graph is decomposed around an anchor node. When comparing a query subgraph G_q (anchored at node q) with a candidate target subgraph G_u (anchored at node u), the model focuses on matching the context relative to these anchor nodes.

Notably, while *NeuroMatch* [56] employs anchoring to structure its subgraph matching routine, the proposed approach adopts only the anchoring idea without incorporating order embeddings. Instead, the focus is on leveraging the anchoring signal to condition the message-passing process.

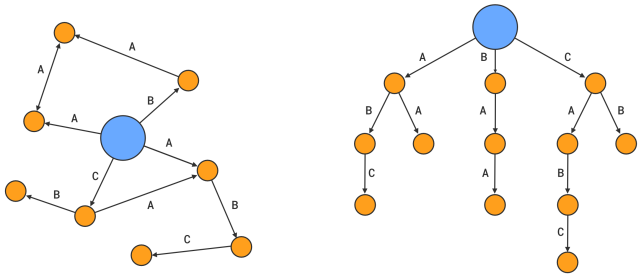
4.4 Pattern Matching

The pattern detection concept is a key component of the proposed approach for design pattern identification in source code. This section introduces the detection mechanism based on neural subgraph matching within the Record Interaction Graph (RIG). In this context, the source graph is defined as a subgraph of the RIG, anchored at a record node, while the query graph is a subgraph representing a design pattern example. Using subgraph matching, the proposed concept verifies whether the query graph is an induced subgraph of the source graph. A positive match indicates the detection of the design pattern.

The detection process faces several challenges. First, the extracted subgraph may be noisy. Different instances of the same design pattern in the source graphs can exhibit varying degrees of relationships, some of which may not be present in the query graph. This requires a normalization step to align the representations (section 4.4.1). Second, although design patterns embody specific structures, they may be implemented in diverse ways. Because of this, extracting the most important components and preserving the various structural shapes of design pattern examples is essential (section 4.4.2). Third, when multiple instances of the same design pattern are used as queries, the accuracy of detection may decrease, which requires the integration of a voting mechanism to identify the most likely pattern (section 4.4.3).

4.4.1 Graph Normalization

The extracted subgraphs from the Record Interaction Graph (RIG) may exhibit noise due to the variability in the number and configuration of interaction relationships among records. For instance, a design pattern example in the RIG might include four inheritance interactions between two records, while an actual instance in the source graph may only manifest three such interactions. Such discrepancies can prevent the query graph from being recognized as an induced subgraph of the source graph, leading to false negatives.



(a) source (b) normalized

Figure 4.7: Illustration of the normalization process for a RIG subgraph.

The proposed concept approaches this issue through a normalization process that aggregates interaction relationships. Let a RIG subgraph be defined as $G = (V, E)$, where V denotes the set of record nodes and E the set of directed edges representing interactions. The normalized subgraph $G' = (V', E')$ is derived from G by applying an aggregation operator $\mathcal{A}$ that groups edges based on their interaction type. The normalization process is defined as follows:

- **Anchor Node Initialization:** The process begins at an anchor node $r_a \in V$, which serves as the root of the normalized subgraph. The anchor node is included in V'.
- **Edge Grouping and Aggregation:** For each node $r \in V$ that is within the 1-hop neighborhood of the anchor, the set of outgoing edges $\{e \in E \mid e = (r_a, r), \ell_E(e) = \tau\}$ is aggregated into a single representative edge (r_a, r') in E' with the interaction type τ. Here, ℓ_E is the edge labeling function, and τ is an element of the set of interaction types $\mathcal{T}$. This step essentially performs an *edge contraction* on groups of edges sharing the same label.

- **Iterative Aggregation:** The same aggregation procedure is then applied iteratively. For any aggregated node r' that represents a group of original nodes, the outgoing edges from all nodes represented by r' are grouped by interaction type, and the aggregated edges are added to G'. This recursion continues until an n-hop neighborhood around the anchor is completely processed or all nodes in G have been visited.

This normalization process can be seen as a graph reduction technique that maps the original graph G onto a more abstract representation G' via a function $f : G \to G'$ that preserves the structural essence of the interactions while ignoring minor variations in edge multiplicity. The key assumption is that the characteristic features of a design pattern are captured by the topology and types of interactions in an n-hop neighborhood, not by the precise counts of these interactions. Thus, the aggregation operator $\mathcal{A}$ effectively implements a many-to-one mapping:

$$\mathcal{A} : \{e \in E \mid \ell_E(e) = \tau\} \to \{e' \in E' \mid \ell_E(e') = \tau\}. \tag{4.7}$$

The resulting normalized graph G' has a tree-like structure with the anchor node as its root, and it preserves the essential connectivity and interaction patterns of the original RIG subgraph while filtering out noise (fig. 4.7). This refined representation enables the neural subgraph matching component of the proposed approach to focus on the intrinsic structural properties of design patterns and improves the comparability of subgraphs during the detection process.

4.4.2 Pattern Extraction

In order to capture the essential structural characteristics of design patterns while accommodating implementation variability, the proposed concept employs an iterative common subgraph extraction method, which is based on the idea of identifying commonalities among multiple subgraph instances. This process is designed to distill the most representative features of a design pattern from a set of normalized RIG subgraph examples. Let $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$ denote the set of normalized RIG subgraph examples corresponding to a particular design pattern, where each $G_i = (V_i, E_i)$ is a directed graph with V_i representing the set of record nodes and E_i the set of interaction edges.

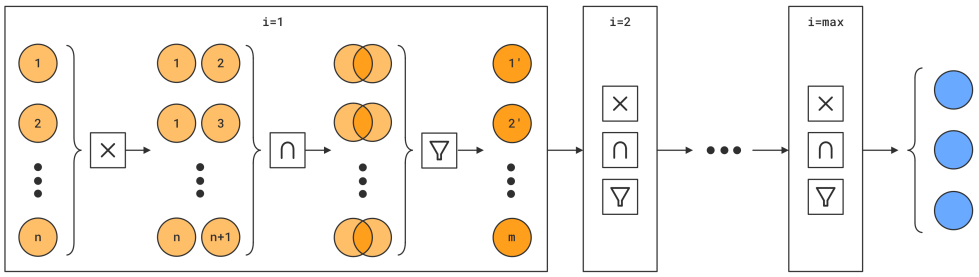


Figure 4.8: The iterative common subgraph extraction process for design pattern detection.

The extraction process is built upon two fundamental functions:

1. An *intersection function* $\cap_N$ that computes the common subgraph of two normalized graphs G_i and G_j starting from their respective anchor nodes. For any pair (G_i, G_j), the common subgraph is defined as $G_{ij} = G_i \cap_N G_j$.

2. An *equivalence function* $\equiv_N$ that determines whether two normalized subgraphs are equal in structure and edge labeling, i.e., $G \equiv_N G'$ if and only if G and G' are isomorphic with respect to the labels.

The algorithm initiates by setting the candidate set of common patterns as $\mathcal{C}^{(0)} = \mathcal{G}$. In each iteration i, the process considers all unordered pairs (G_a, G_b) from the current candidate set $\mathcal{C}^{(i)}$ and computes their common subgraph:

$$\mathcal{C}^{(i+1)} = \{G_a \cap_N G_b \mid G_a, G_b \in \mathcal{C}^{(i)}\}. \tag{4.8}$$

To avoid redundancy, the set $\mathcal{C}^{(i+1)}$ is filtered by discarding any subgraph G for which there exists another subgraph G' such that $G \equiv_N G'$. Moreover, only those subgraphs satisfying the node threshold $|V(G)| \geq n_{min}$ are retained, where n_{min} is a predefined minimum node count to ensure that only substantial substructures are considered.

This iterative extraction continues until the candidate subgraphs are reduced to at most g_{max} or until the iteration count reaches i_{max}, stopping when $|\mathcal{C}^{(i)}| \leq g_{max}$ or $i \geq i_{max}$. The final output $\mathcal{C}^{(i)}$ represents the distilled set of common subgraphs that encapsulate the key interactions inherent in the design pattern. By iteratively extracting pairwise common subgraphs and eliminating duplicates as well as subgraphs below a significance node threshold, the proposed concept efficiently isolates the most common structural motifs of a design pattern.

4.4.3 Pattern Voting

The detection of design patterns using multiple query instances can lead to inconsistent matching results, especially when variations in implementation and noise in the source graph are present. The proposed approach addresses this challenge by introducing a voting mechanism that consolidates evidence from several pattern queries to determine whether a source graph represents an instance of a specific design pattern.

The core idea is to utilize a set of representative design pattern examples as queries. Each query subgraph is matched against a source graph using neural subgraph matching, resulting in a prediction value $p \in [0, 1]$ that indicates the likelihood of the query being a subgraph of the source. To ensure valid comparisons, the approach first aligns the query with the source: if a query subgraph is larger than the source subgraph, the query is reduced in size by incrementally including nodes that are closer to the anchor until the node count of the source is reached.

In addition, to account for the diverse structural manifestations of design pattern instances, the query is sampled at varying distances from the anchor node. A distance offset interval is defined such that subqueries of the original query are generated. The maximum distance of the original query is denoted by $d_{\max}(G_q)$. In the iteration i, a subquery with maximum distance $d_{\max}(G_{q_i}) = d_{\max}(G_q) - i$ is extracted, with the process terminating once a minimum distance $d_{\min}$ is reached. Since reducing the query distance also reduces its descriptive significance, a weight w_i is assigned to each subquery

$$w_i = \frac{|V_{q_i}|}{|V_q|} \tag{4.9}$$

ensuring that the influence of each query is proportional to its node size $|V_{q_i}|$ relative to the original query node size $|V_q|$. Subsequently, pairs consisting of the source graph and each sampled query are formed, and the neural model computes a subgraph matching prediction p for each pair. Each prediction is then weighted by the corresponding query weight, yielding a weighted prediction p_w defined as $p_w = p \cdot w_i$.

Let $\mathcal{P}_\mathcal{T} = \{p_{w_1}, p_{w_2}, \ldots, p_{w_n}\}$ denote the set of weighted predictions for a given design pattern type $\mathcal{T}$. In the final voting stage, for each design pattern type, the quantile $Q_\mathcal{T}$ of the set $\mathcal{P}_\mathcal{T}$ is computed using a high quantile value q (e.g., $q = 0.9$), so that only the top percentage of the predictions contribute as positive votes.

Let $\{p_{(0)}, p_{(1)}, \ldots, p_{(n)}\}$ be the elements of $\mathcal{P}_\mathcal{T}$ arranged in increasing order. For any quantile level $q \in [0, 1]$, the distribution index is defined as $h = q \cdot (n - 1)$. Let $i = \lfloor h \rfloor$ be the integer part and $\delta = h - i$ be the fractional part for interpolation. Then the q-th quantile for $\mathcal{P}_\mathcal{T}$ is given by:

$$Q_\mathcal{T} = Q_q(\mathcal{P}_\mathcal{T}) = p_{(i)} + \delta \left(p_{(i+1)} - p_{(i)}\right) \tag{4.10}$$

$Q_\mathcal{T}$ represents the interpolated threshold value such that approximately $q \cdot 100\%$ of the predictions in $\mathcal{P}_\mathcal{T}$ fall below it. If $Q_\mathcal{T}$ exceeds a predefined threshold (e.g., 0.5), the source graph is declared as an instance of the design pattern. This voting mechanism enhances the robustness of the detection process by aggregating evidence across multiple, variably sampled queries. Consequently, the proposed approach is designed to detect design patterns even in noisy source graphs and when design pattern examples exhibit implementation variability.

5 Design and Implementation

This chapter details the concrete realization of the proposed approach for design pattern detection through the use of CPGs and neural subgraph matching. It presents an in-depth view of the technologies and design decisions used for the implementation, highlighting how language-independent code abstractions are transformed and processed into graph representations for effective pattern matching. The chapter describes the implementation of the concepts introduced earlier (chapter 4), providing insight into technical challenges encountered and the respective solutions. The following sections outline the system architecture (section 5.1), detail the stages of graph generation (section 5.2) and model training (section 5.3), and discuss the testing strategies applied to ensure robustness and correctness (section 5.4). The implementation represents a prototype that demonstrates the feasibility of the proposed concepts and serves as a foundation for future research and development. Therefore, the focus lies primarily on the functional correctness and maintainability of the system, rather than on performance or scalability.

5.1 System Architecture

The proposed approach is built as a three-layer architecture using containerized components that separate the overall processes. The components handle different stages of the design pattern detection pipeline, including graph generation and pattern matching. Both components are managed and orchestrated by the virtual container environment. A lightweight command-line interface (CLI) tool provides a set of interactions to the user by handling the communication and order of execution with systems underlying components. Between the components, a robust graph database as a persistence layer is provided. The persistence layer is used for storing and sharing the generated graph data between the components. In addition, the graph database can be used for in-depth analysis of the graph data with graph query languages and graph visualization tools provided by the specific database implementation.

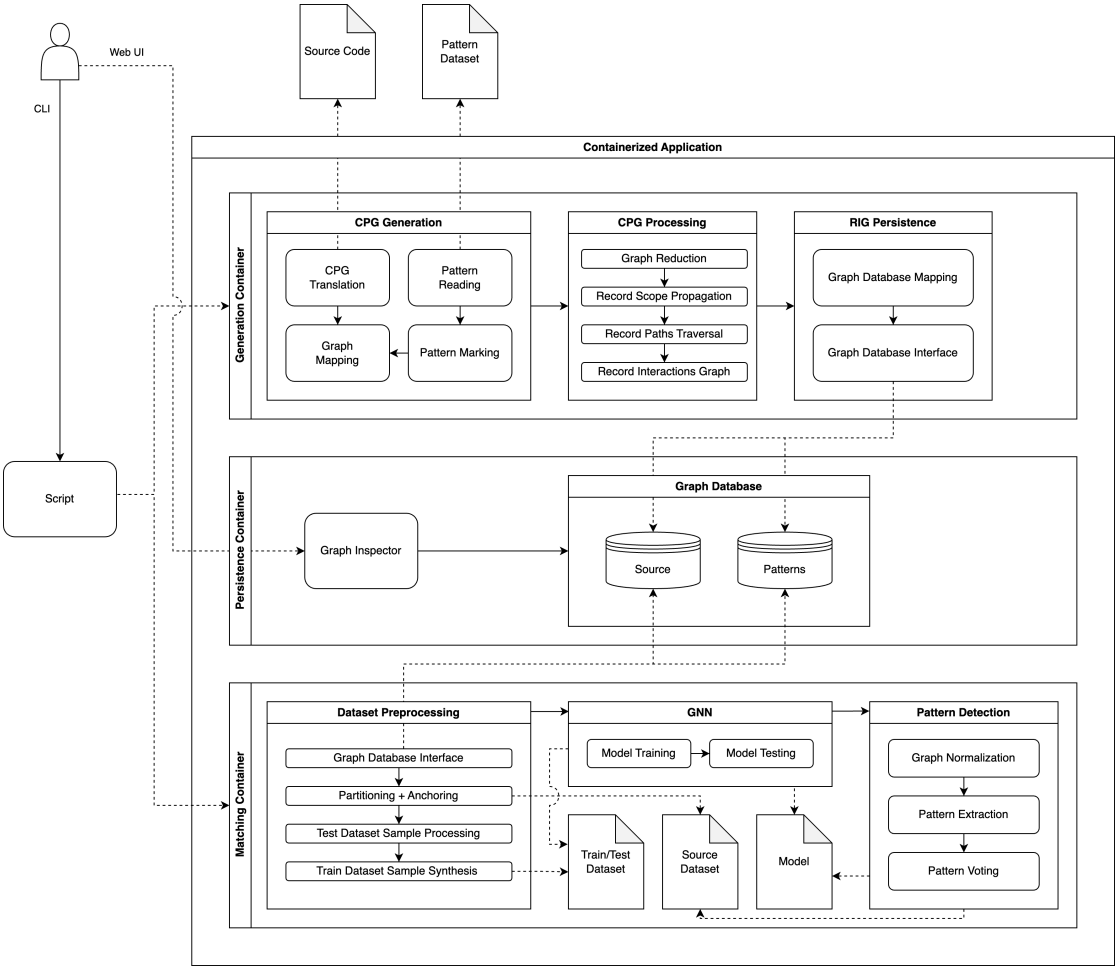


Figure 5.1: An overview of the system architecture and components.

The graph generation component has the purpose of translating and processing source code into a structured graph representation. For this, the component is divided into three tasks. First, the source code is translated into an external Code Property Graph model defined by the external translation library. This model is then mapped to the internal CPG representation and is optionally marked with design pattern annotations for training and testing purposes. In the second task, the CPG model is transformed into the RIG representation using a specific processing pipeline (section 5.2.3). The last task maps the RIG to a graph database model and uses a dedicated persistence interface to write the graph into the database.

After the graph generation component has completed the processing of the source code, the pattern matching component is responsible for reading the persisted graph datasets from the database and preprocessing them for training and testing. The preprocessing phase includes the partitioning of the datasets into smaller graph samples, generating both positive and negative subgraphs for each sample, and synthesizing additional training data. Using the preprocessed data, the GNN model is trained and tested on the generated graph samples. The pattern detection task performs graph normalization steps on the graph data and extracts pattern examples for the final pattern voting mechanism.

5.2 Graph Generation

The graph generation component is implemented as a Java 17 application and converts raw source code into a structured graph representation. The process begins by translating the source code into a Code Property Graph using the *Fraunhofer AISEC* CPG project (version 8.3.0). This process leverages the comprehensive and extensible design of the CPG, which is divided into core and language-specific modules. The library is designed to be easily extensible and is still under active development, currently providing matured support for languages like Java and C++ and experimental support for languages like Ruby and JavaScript. Once the CPG is constructed, it is processed into a Record Interaction Graph via a custom pipeline that employs *GraphStream* (version 2.0) for efficient in-memory graph manipulation. For this, core frameworks are implemented to manage the processing steps, including a custom pipeline and a graph traversal algorithm. After processing, the RIGs are persisted in a *Neo4j* graph database using the *Neo4j OGM driver* (version 4.0.10), which ensures reliable storage and facilitates subsequent retrieval for pattern matching.

5.2.1 Core Frameworks

The processing of the Code Property Graph relies on two core frameworks: a custom pipeline and a node-centric graph traversal algorithm. These frameworks form the foundation of the proposed approach by managing the majority of the processing steps required for constructing the design pattern detection architecture. Both frameworks are designed with a strong emphasis on modularity, extensibility, and maintainability.

Pipeline The further transformation and processing task of the CPG is realized through a custom pipeline that orchestrates the various processing steps required to construct the RIG. The pipeline is designed with the primary goals of maintainability, clear separation of concerns, and extensibility. The implementation organizes the required steps through a sequence of processing modules. Each module in the pipeline is responsible for a distinct transformation, ensuring that the logic is encapsulated and isolated within well-defined boundaries.

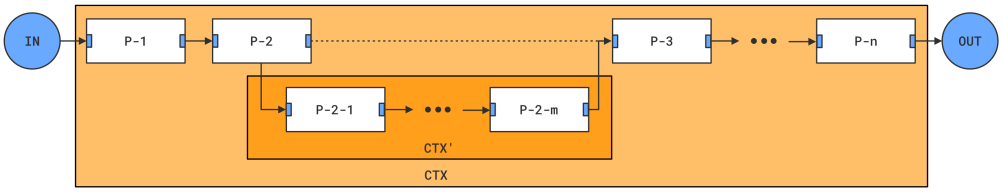


Figure 5.2: The pipeline implementation including the context handling by subprocesses.

The modules in the pipeline are constructed using a *Builder* pattern, which allows for the dynamic assembly of the processing chain. Each processing module has a reference to the next module in the sequence, enabling the sequential execution of the pipeline. For this, each module uses the output of the previous module as input. In addition, a pipeline context is maintained throughout the execution, which includes essential metadata such as configuration arguments, processing statistics, and custom context information. The context can be accessed and modified by each module, ensuring that the execution state is consistently maintained across all processing stages. An abstract class is used to define the execution management and context handling, as well as performance tracking. Each concrete module implements this class and defines the input and output types of the module. There is a single method that must be implemented by each module, which is responsible for the actual processing logic. This modular structure not only enables easier debugging and testing but also promotes maintainability by allowing individual modules to be modified or replaced without impacting the overall system. By defining pipelines in the modules themselves, processes can be easily grouped to combine subprocesses. This generic implementation of a pipeline is used as the foundational architecture for processing the CPGs.

Graph Process Traversal The second core framework implemented for processing the CPG is a node-centric graph traversal algorithm, which serves as the central abstraction for processing at a global graph level. This framework processes the graph on a node

level by employing message-passing through the edges and is encapsulated within an abstract class that handles the traversal logic and manages visited edges. The class also determines when to halt the traversal, either via a custom stop signal or upon reaching a predefined maximum depth.

Algorithm 1 Graph Process Traversal

```
1: procedure TRAVERSE(startNode, maxDepth)
2:     queue ← an empty deque
3:     push (startNode, null, null, null, 0) into queue
4:     while queue is not empty do
5:         processData ← pop from queue
6:         node ← processData.node
7:         inputData ← processData.message
8:         ctx ← (processData.edge, processData.parent, processData.depth)
9:         if maxDepth ≥ 0 and ctx.depth ≥ maxDepth then
10:            continue          ▷ Skip further processing if maximum depth is reached.
11:        end if
12:        output ← PROCESS(node, inputData, ctx)
13:        if ctx.edge ≠ null then
14:            mark ctx.edge as visited
15:        end if
16:        if output.proceed = false then
17:            continue                    ▷ Do not traverse further from this node.
18:        end if
19:        for all edge in NEXT(node) do
20:            if edge is not visited then
21:                nextNode ← edge.opposite(node)
22:                message ← output.message
23:                depth ← ctx.depth+1
24:                push (nextNode, node, edge, message, depth) into queue
25:            end if
26:        end for
27:    end while
28: end procedure
```

Traversal processes that implement the abstract class must provide concrete implementations of only two methods. The `Process` method is responsible for processing a node by applying custom logic. It accepts as input the current node, an incoming message from the calling edge, and a context containing additional information such as the parent node, the incoming edge, and the current traversal depth. Based on this input, the `Process` method produces a process output that consists of the process output mes-

sage and a flag indicating whether the traversal should continue from the current node. In contrast, the `Next` method returns a list of edges to be queued for the subsequent traversal step relative to the current node, thereby defining the custom traversal strategy. This clear separation between the processing logic and the traversal strategy facilitates flexible adaptation of the approach to various processing requirements.

5.2.2 Translation

The translation module is responsible for converting raw source code into an intermediate Code Property Graph representation using the *Fraunhofer AISEC* project. The library is divided into core functionalities and language-specific components, thereby supporting multiple programming languages. This structure decouples the CPG logic from language-specific features by using a dedicated translation layer that abstracts concrete language constructs into a unified CPG representation. The translation supports multiple CPG passes, each designed to extract different aspects of the source code. For instance:

- **Data Flow Pass:** Tracks the flow of data through variables and functions to identify potential dependencies.
- **Control Dependency Pass:** Captures the control flow dependencies between various code constructs.
- **Program Dependence Pass:** Combines control and data dependencies to provide a comprehensive view of program structure.
- **Evaluation Order Pass:** Describes the order in which expressions are evaluated within a program.
- **Type Hierarchy Pass:** Analyzes and resolves relationships between types, supporting inheritance and interface implementations.

In the implemented processing module, a *Builder* pattern is employed to configure the translation process. The *Builder* is used to register the desired passes, specify the target language, define source code files, attempt dynamic code inference, and load any required dependencies. Once the configuration is complete, the translation process loads the source files, computes the registered passes, and constructs the internal CPG representation.

For further processing, the internal CPG must be converted into a graph representation using *GraphStream*. Although *GraphStream* was originally designed for dynamic graph processing, it is utilized in this context for its robust support for static graph manipulation, its extensive built-in algorithms (e.g., BFS, DFS, SSSP, CC), and its compatibility with multi-graphs. The *Fraunhofer* library provides a mapping to a generic JSON representation of the CPG graph, which includes a list of all nodes and edges along with their properties[1]. This JSON is then iterated over to construct a *GraphStream* multi-graph, which is subsequently used for further processing within the pipeline.

5.2.3 Processing

The processing of the CPG and the transformation into the RIG are implemented as a series of modules within the pipeline. Each module is responsible for a specific transformation step, such as record scope propagation, record path computation, and record interaction computation. The processing modules are designed to be modular and extensible, allowing for easy integration of additional processing steps or modifications to existing ones. The most important processing modules are the following:

Record Scope Module The Record Scope Module is responsible for assigning the record scope attribute to every node in the CPG in accordance with the hierarchical scope structure defined in the concept section 4.2.2. In the proposed approach, the module first iterates over all nodes and distinguishes between record declaration nodes and those associated with a record scope. For nodes identified as record declarations, the module extracts the full record scope attribute and immediately propagates it to all directly connected nodes via incoming edges of type `RECORD_DECLARATION`. For nodes that have the label `SCOPE_RECORD` and have not yet been propagated, the module employs a graph traversal process to determine the correct record scope. It invokes an implementation of the node-centric graph processing abstraction. This traversal follows a message-passing paradigm where each node updates its record scope attribute by checking if it directly represents a record scope and, if so, extracting the corresponding attribute. The propagation then continues along the entering edges to update adjacent nodes. The traversal stops upon reaching another record scope node. By following this strategy, the

[1]The JSON mapping for the *Fraunhofer* CPG is experimental and is not fully optimized. In cases where the source code input is large, memory constraints might necessitate reducing the depth of CPG traversal or employing batched database queries to store the CPG in smaller chunks.

approach ensures that every node in the CPG is assigned the appropriate record scope, thus preserving the integrity of the Record Interaction Graph construction.

Record Path Module The Record Path Module computes the direct paths between record nodes in the CPG in accordance with the proposed approach. The module first filters the graph to obtain nodes that are declared as records by checking for the corresponding record label. For every record node, a record neighbor subgraph is extracted by traversing the CPG up to a defined maximum path distance. This subgraph provides the local context for the record and serves as the basis for computing the record interaction paths. The module then employs the shortest path algorithm based on *Dijkstra's algorithm* [16] to compute the shortest paths from the current record node to all other record nodes within the subgraph. In each iteration, the module calculates the shortest path and removes the last edge of the found path from a copied instance of the subgraph. This iterative process identifies multiple path variations for every record node and ensures that alternative paths are explored up to a predefined maximum number of variations. In an optimal scenario, the module would generate every possible unique path between the record nodes, using k-shortest path algorithms like *Yen's algorithm* [79]. However, due to the complexity of the CPG and the computational overhead, the module is designed to generate a limited number of path variations to balance performance and accuracy. Each computed path is represented as a sequence of edges that connects the start and end record nodes with intermediate nodes that do not belong to the record set. In this way the module respects the constraint that the record scope changes only once along the path. All computed record paths are aggregated in a dedicated container, and the collection is then attached to the processing context to be used by subsequent stages in the pipeline.

Record Interaction Module The proposed approach implements the Record Interaction Module as a processing step that computes and aggregates interactions between record nodes. The module retrieves the collection of record paths from the processing context and verifies each path to ensure it represents a direct interaction between record nodes and that the record scope in the path changes only once. After validation, the module determines the interaction type by analyzing the edge types along the path. If the path contains specific edge types such as `INSTANTIATES`, `SUPER_TYPE_DECLARATIONS`, or `RETURN_TYPES`, the corresponding interaction type is assigned. Otherwise, the default interaction type is used. The computed interaction is aggregated and added

to the graph by either reusing an existing interaction or creating a new one. An edge is then created between the record and interaction node with attributes capturing the interaction type.

Pattern Marking Module The Pattern Marking Module is responsible for annotating the Code Property Graph with design pattern labels by processing each node and applying the corresponding design pattern marker to generate the train and test datasets for the GNN model. The module retrieves the dataset of design patterns from the processing context and initializes a statistics container for tracking the number of markings per design pattern. The module iterates over all nodes in the graph and focuses on nodes that represent declaration records. For each relevant node, the module extracts the full class name from its attributes and identifies potential design patterns by matching the class name with the design pattern declarations contained in the dataset. When a match is detected, the module marks the node by adding a label that corresponds to the detected design pattern type. The marking process is combined with an update of the statistical counters that record the occurrence of each design pattern and a total count. In addition, the module computes ground truth statistics by traversing the dataset and then compares them with the observed markings.

5.3 Model Training

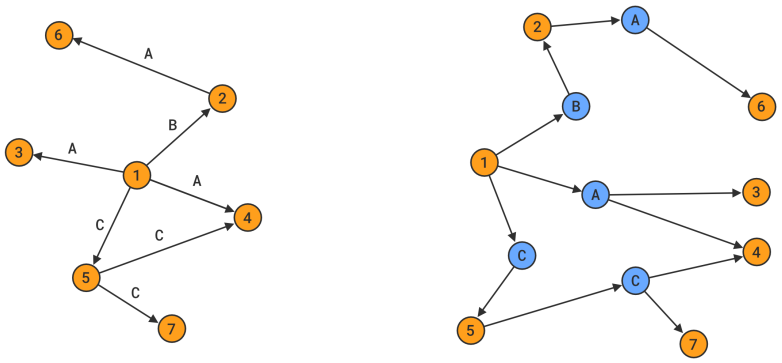
The graph matching component is realized as a Python 3.9 application and focuses on the training of the GNN model and the actual pattern detection task. For this, the component utilizes the *Neo4j* driver (version 5.26.0) for data access and relies on *NetworkX* (version 3.2.1) for the manipulation and transformation of the graph structures. The implementation partitions the persisted graph data into smaller subgraphs for each record. For each sample, the anchor node is defined, and both positive and negative subgraph samples are generated. Additional training samples are synthesized by analyzing the structural properties of the graph. For the GNN, the implementation uses the GLeMA Net model that is configured to use case-specific hyperparameters. The training process is implemented with *PyTorch* (version 2.4.1) and optimized through a curriculum training strategy that starts with smaller subgraphs and gradually increases their size over the training epochs.

5.3.1 Dataset Preprocessing

The data preprocessing of the proposed approach is performed in four steps: loading datasets, source graph generation, test sample augmentation, and train sample synthesis. First, the datasets are read from the *Neo4j* database and transformed for model compatibility. The dataset is then partitioned into smaller subgraphs with an anchor record node set for each sample. Based on these anchored source graphs, test samples are generated by augmenting them with multiple positive subgraph query samples that guarantee subgraph isomorphism. Negative subgraph query samples are generated by random graph modifications to ensure non-isomorphism. Finally, synthesized training samples are created by analyzing structural properties such as the average and standard deviation of source sizes and node degrees, generating connected graphs that closely resemble the original source graphs and amounting to four times the number of test samples. Steps 3 and 4 are skipped for the inference data, as only the training of the model requires artificial query samples.

Loading Datasets Datasets are read from the *Neo4j* database. The RIG graph data is then transformed for compatibility with the GLeMA Net model. Since the model cannot handle edge features, each edge representing an interaction between records is converted into a node with the original edge label as its node label. This interaction node is connected with a single edge to the source record and with one or more edges to the target record nodes that share the same interaction. In this way, the transformation preserves the full informational content of the RIG while ensuring compatibility with the GNN model (fig. 5.3).

Source Graph Generation The implementation in this step partitions the dataset into smaller subgraph samples centered around each record node by extracting a k-hop neighborhood. For each node identified as a valid record and candidate for anchoring, an ego graph is computed in an undirected manner using an initial radius defined by the import parameters. This is necessary to extract subgraphs of a mostly homogeneous size around the anchor node. If the resulting subgraph exceeds the maximum allowed nodes, the radius is iteratively decreased. If it contains fewer nodes than the minimum required, the radius is increased. This dynamic adjustment is performed up to a predefined maximum number of retries to obtain a subgraph that meets the size constraints. Once a valid subgraph is extracted, it is converted to a directed graph and merged into



(a) Original RIG (b) Transformed RIG

Figure 5.3: Comparison of the original RIG (left) and the transformed RIG (right) for compatibility with the GNN model.

a global target graph by assigning each node a new sequential identifier. In this merging process, the anchor node is explicitly mapped and preserved while essential attributes such as label indices, design pattern types, and record scope are maintained. All edges between nodes are subsequently added to the target graph provided both endpoints have been successfully mapped. This systematic extraction and integration ensure that each subgraph sample accurately represents the local structure of the original graph and is suitable for further augmentation and model training.

Test Sample Augmentation To generate the test samples the *GLeMA Net* model will be tested on while training, all source graphs must be augmented with a set of query subgraphs. For this, the implementation of the proposed *GLeMA Net* model provides a process step, which uniformly generates positive (isomorphic) and negative (non-isomorphic) subgraph samples for each anchored source graph. In the case of positive samples, a subset of nodes is selected based on a probability derived from the desired subgraph size relative to the total number of nodes in the source graph. The anchor node is always retained while the remaining nodes are removed stochastically to yield a connected subgraph that is isomorphic to the source graph. For negative samples, a similar node selection is followed by a series of random modifications such as node label changes and the addition or removal of nodes and edges. These operations are iteratively applied while checking for subgraph isomorphism to ensure that the resulting subgraph is not isomorphic to the source graph. The process is executed in parallel over multiple source graphs with progress tracked through a multiprocessing queue.

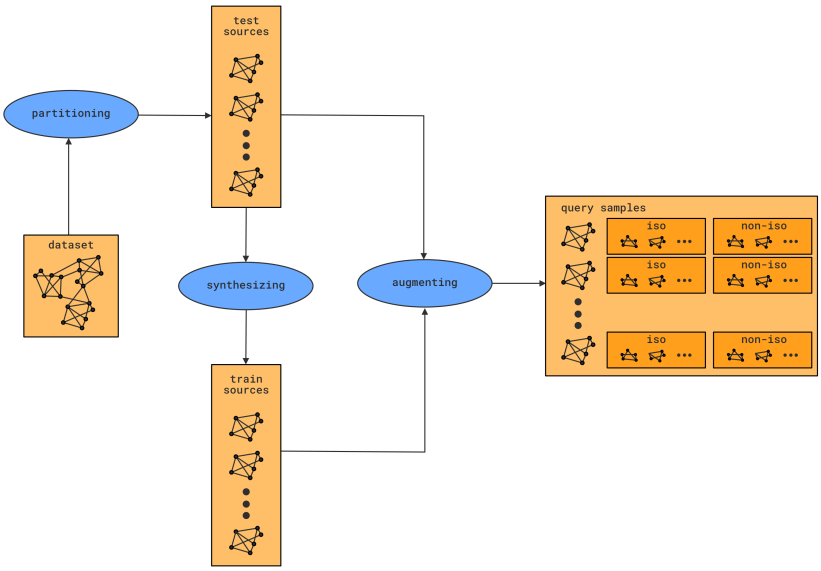


Figure 5.4: Dataset preprocessing for the GLeMA Net model training.

Train Sample Synthesis To generate a training dataset that is multiple times larger than the test dataset, the implementation synthesizes the training samples by analyzing the structural properties of the source graphs and emulating their structural properties. This process is also included from the original implementation of the *GLeMA Net* model, which is marginally modified to handle the structure of the RIG. In this implementation, a connected synthetic graph is created using the *Erdős–Rényi* graph generator included in the *NetworkX* library, where the number of nodes is sampled from a normal distribution defined by the average source size and its standard deviation, and the probability of edge creation is determined by the average degree in the dataset source graphs. Once generated, the synthetic graph is augmented with additional features by first assigning an anchor node based on a top *PageRank* score and then performing a depth-first traversal from the anchor to assign node labels. Even-depth nodes are consistently labeled as record nodes, while odd-depth nodes receive random labels, representing the interaction types. This labeling process mirrors the structure of the transformed RIG (fig. 5.3). Subsequently, the same query generation algorithm is used for the test samples, which generates multiple positive and negative subgraph isomorphism examples. This entire synthesis process is executed in parallel and configured to produce a training dataset that is multiple times larger than the test dataset, thereby providing an optimal training-to-test data distribution.

5.3.2 Network Setup

The implementation described in this section is adopted directly from the original paper. The *GLeMA Net* model is built upon a series of *GLeMA* layers that leverage a learnable multi-hop attention mechanism. In each *GLeMA* layer, the input node features are first projected into a higher-dimensional space via a linear transformation. The projected features are then reshaped into an attention representation where a learnable edge parameter is used to compute pairwise interactions. A key aspect of this layer is the derivation of node-specific attention decay factors through an additional linear mapping; these factors are computed via a sigmoid function and enable each node to modulate the contributions from its multi-hop neighbors dynamically.

The GLeMA Net module stacks four such graph attention layers. In each layer, the node features are mapped to a 140-dimensional space. The multi-hop attention mechanism in these layers is configured using the *jump tactic*. With this tactic, the number of hops is increased in a non-linear fashion, following the rule that the effective number of hops is $2i+1$, where i is the index of the layer. This design allows higher layers to capture broader contextual information from distant nodes without incurring the computational cost of a fixed large number of hops. An additional interesting detail is the dual branch design incorporated in *GLeMA Net*. The network processes the input graph with two parallel branches. One branch is dedicated to intra-graph feature extraction using an adjacency matrix that represents direct connections among nodes. The other branch processes inter-graph relationships via a proxy adjacency matrix that encodes "virtual" links based on node label similarities between the query (pattern) and target graphs. The outputs from these branches are then combined by taking their difference, effectively highlighting the discrepancies between intra-graph structures and cross-graph correspondences. This branching strategy is critical for robust subgraph matching and matching explanation.

Prior to entering the graph convolutional layers, the initial node features are embedded into an n-dimensional space using a separate linear mapping. The number of the embedded node features equals the number of node labels plus a feature for the anchor node. After the stacked *GLeMA* layers, the aggregated node embeddings are passed through a fully connected network comprising four layers, where the first layer expands the representation to 128 dimensions. The final output is generated using a sigmoid activation function, returning a probability score for subgraph matching.

5.3.3 Curriculum Training

The proposed approach integrates a number of optimization strategies to improve the training process of the GNN model. For the general training loop, the training and test datasets are preprocessed into collections of pickle files that are indexed by unique identifiers and corresponding metadata that includes metrics about the data samples, like the number of nodes and edges. These identifiers serve to load individual samples representing a source graph and either positive or negative query samples. While training, the data loader retrieves the samples from the pickle files and processes them in batches. The data loader is designed to handle the large graph samples efficiently by loading only the required data into memory, thereby reducing the memory footprint and ensuring that the training process is not hampered by memory constraints.

One of the key strategies is curriculum training, which is implemented to optimize the training process when working with large graph samples [56]. For this, the curriculum training approach is employed to gradually increase the complexity of the training samples over the course of the training epochs. The assumption is that the model can learn more effectively from smaller subgraphs at the beginning and that this enables the model to generalize better to larger subgraphs. Let $\mathcal{D}$ denote the complete dataset and let $C(x)$ be a function that quantifies the complexity of a sample $x \in \mathcal{D}$. The implementation uses the $C(x) = |V_x|$ as the complexity measure, where V_x represents the nodes of x. The dataset is partitioned into subsets based on complexity ranges. A complexity range is defined as

$$\mathcal{D}_i = \{x \in \mathcal{D} \mid a_i \leq C(x) < b_i\} \tag{5.1}$$

where the interval $[a_i, b_i)$ specifies the complexity range for subset $\mathcal{D}_i$. The curriculum training strategy begins with training on the lowest complexity range $\mathcal{D}_1$, which corresponds to the smallest graph samples. During training, samples are selected from the union

$$\mathcal{D}_{\leq k} = \bigcup_{i=1}^{k} \mathcal{D}_i \tag{5.2}$$

where k is the current complexity threshold. Initially, k is set to 1. After every T epochs, the threshold is increased by one unit, thereby incorporating samples of higher complexity into the training process. When k exceeds the maximum available level, the complexity constraint is removed, and training is conducted on the entire dataset $\mathcal{D}$. This approach enables the model to build a strong foundation by first learning from simpler subgraphs before gradually adapting to more complex structures [56].

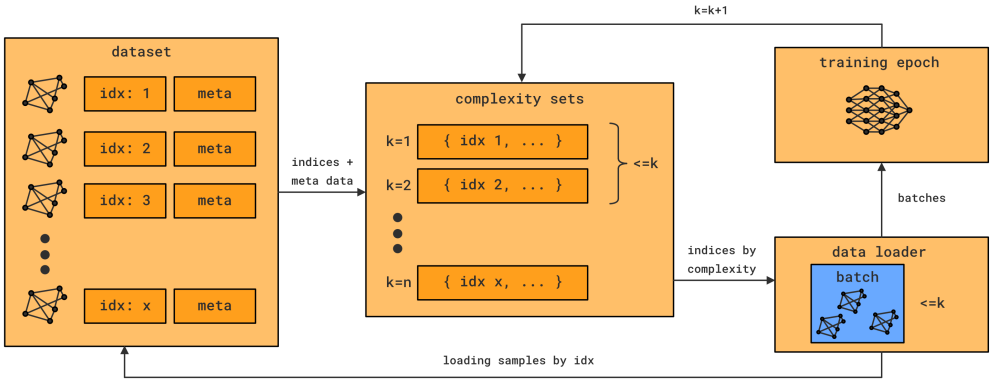


Figure 5.5: Overview of the curriculum training process.

In addition to the curriculum training, the training loop includes a mechanism for early stopping. The training process is monitored for performance metrics such as the *Area Under Receiver Operating Characteristic Curve* (ROC AUC) score (section 6.2). An improvement in this metric triggers a reset of the early stopping counter. Conversely, if the performance does not improve and training is beyond the initial stage, the counter increments until a predefined limit is reached, leading to early stopping. This prevents overfitting and ensures that the model generalizes well to unseen data. Another technique to prevent overfitting to the training data is to balance the data samples. Each epoch, the model should be trained with an equal number of positive and negative query samples. This is achieved by dynamically adjusting the training samples based on the number of available samples for each class, especially in consideration of the used curriculum training approach.

5.4 Tests

Testing is crucial to guarantee the correctness, robustness, and reliability of the implementation by verifying that each component functions accurately both independently and within the integrated system. The implementation uses a test-driven design approach and focuses on unit tests. Integration tests are also included but are less emphasized.

Tests within this implementation utilize well-established testing frameworks in Java and Python. Specifically, *JUnit* is employed for unit testing the Java-based components, while Python-based modules utilize *Pytest* for both unit and integration testing. Table 5.1 provides an overview of the test coverage across the implementation and is generated

Table 5.1: Test coverage of the implementation.

	Files Covered (%)	**Lines Covered (%)**
Graph Generation		
Code Translation	75	66
CPG Processing	81	73
RIG Persistence	71	52
Utils	93	87
Graph Matching		
Dataset Preparation	50	26
Graph Embedding	75	32
Pattern Detection	75	65
Utils	73	84

using the test coverage tools provided by the used IDEs (*IntelliJ* and *PyCharm*). The coverage presents a high percentage across the different implementation scopes. Lower test coverage is present in files that use external libraries, like the CPG generation, the *Neo4j* database, and the training of the GLeMA Net model. Those external libraries are not covered by the tests, since they are not part of the implementation. To test the modules using these libraries, mock frameworks like *Mockito* are used to simulate the behavior of the external libraries.

Unit tests were used to validate individual functionalities, including the transformation from CPG to RIG, ensuring accurate capturing of record interactions. Further unit tests assess the graph normalization routines, verifying correct graph aggregation and abstraction. Integration tests are used for the Code Property Graphs pipeline and the pattern voting process. Automated system tests for the complete detection task are not utilized but were manually evaluated.

6 Evaluation

This chapter presents a comprehensive evaluation of the proposed approach for detecting design patterns in source code using neural subgraph matching. The evaluation examines the effectiveness and efficiency of the concept by discussing the experimental setup, the choice of datasets, the definition of metrics, the obtained results, and the subsequent analysis. The analysis addresses generalization capabilities, language independence, and the robustness of the concept, answering the research questions stated in section 1.2.

6.1 Datasets

The evaluation of the proposed approach relies on two datasets that serve as robust benchmark corpora in the field of design pattern detection. The availability of reliable benchmark datasets is a well-known challenge in this domain. The chosen datasets provide a wide range of Java projects that enable a comprehensive assessment of the approach.

Table 6.1: Overview of projects included in the P-MARt dataset [4].

Project	**Version**	**Size (KB)**	**Classes**	**Methods**	**Pattern-Instances**
QuickUML	2.1	632	142	1264	46
Lexi	0.1	355	23	601	18
JRefactory	2.6	2800	556	4690	201
Netbeans	1.0	26000	2238	25446	255
JUnit	3.7	469	69	856	49
JHotDraw	5.1	639	136	1393	165
MapperXML	1.9	1800	195	2307	95
Nutch	0.4	1300	149	1832	19
PMD	1.8	1300	423	3752	50

P-MARt The first dataset is *P-MARt* [21]. This dataset is widely used in design pattern detection benchmarks and includes 9 Java projects. The numbers of instances per design pattern have a high degree of variance, resulting in an unbalanced dataset. Because of this and for comparison reasons, *P-MARt* is only used as a benchmark dataset.

DPDf The second dataset is the *DPDf* corpus [44]. The dataset contains labeled design patterns of the Github Java Corpus [2], which contains over 14,000 Java projects. In contrast to *P-MARt*, the *DPDf* corpus offers a balanced dataset with an average of 90 instances for each design pattern. For this reason, the dataset is primarily used as design pattern examples for the queries.

Table 6.2: Design pattern instances included in the datasets [44].

Pattern	**P-MARt**	**DPDf**	**Pattern**	**P-MARt**	**DPDf**
Abstract Factory	210	89	Decorator	59	89
Observer	104	89	Proxy	3	96
Adapter	189	94	Factory Method	96	96
Memento	11	86	Singleton	13	91
Builder	35	97	Facade	11	99
Prototype	26	85	Visitor	141	91

The representation of design patterns differs between the two datasets. *P-MARt* provides detailed listings that include all roles and relations for each design pattern instance across multiple classes as specified by the GoF. The *DPDf* corpus represents each design pattern instance only by a single class name without additional role or relation details. To support both datasets, the detailed design pattern instances in *P-MARt* are aligned with the *DPDf* representation by discarding the role information. In addition to this, the evaluation focuses only on a subset of the design pattern types included in both datasets. This selection is made for comparison against other approaches in the field.

6.2 Metrics

The evaluation uses several metrics to quantify the performance of the proposed approach for design pattern detection. The metrics have been chosen for their ability to capture various aspects of the task. They provide insight into the effectiveness of neural subgraph matching in identifying design patterns in source code. The metrics are calculated based

on the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) obtained from the matching and detection process.

Precision Precision measures the ability of the approach to correctly identify design patterns while minimizing false detections. This metric is particularly important in design pattern detection because it reflects the exactness of the identified patterns.

$$\mathrm{P} = \frac{TP}{TP + FP} \tag{6.1}$$

Recall Recall indicates the ability of the approach to retrieve all relevant instances of design patterns from the source code. This metric is essential for design pattern detection because it reflects the completeness of the recognition process.

$$\mathrm{R} = \frac{TP}{TP + FN} \tag{6.2}$$

F1-Score The F1-Score is the harmonic mean of precision and recall. It provides a balanced measure that combines both precision and recall. It serves as a comprehensive metric that summarizes the performance of the approach in scenarios with uneven class distributions in design pattern detection.

$$F = 2 \cdot \frac{\mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}} \tag{6.3}$$

Accuracy Accuracy is defined as the ratio of correctly identified instances to the total number of instances. It offers a general measure of the overall performance of the approach. In the context of design pattern detection, this metric evaluates the effectiveness of the approach in distinguishing between the presence and absence of design patterns.

$$\mathrm{A} = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.4}$$

ROC AUC The *Area Under Receiver Operating Characteristic Curve* (ROC AUC) measures the ability of the approach to discriminate between design patterns and non-design pattern instances at various threshold settings. It is defined as the area under the curve that plots the true positive rate (TPR) against the false positive rate (FPR). A

higher ROC AUC value indicates that the approach is effective in ranking true design pattern instances higher than false ones.

PR AUC The *Area Under Precision Recall Curve* (PR AUC) quantifies the relationship between precision and recall for different threshold values used by the approach. It is computed as the area under the curve that plots precision against recall. A higher PR AUC value demonstrates that the approach effectively balances the accuracy of the detections with the completeness of the detection process.

The metrics are widely used in the field of machine learning and pattern detection. They provide a comprehensive overview of the performance of the proposed approach in detecting design patterns in source code and leverage the comparability of the results with other approaches.

6.3 Experiments

The evaluation includes both qualitative and quantitative results with a primary focus on detection quality rather than runtime efficiency. The analysis addresses the entire process, starting from the graph generation through to the neural subgraph matching training and the final design pattern detection. The experiments were conducted on a system equipped with an Apple Silicon M3 Pro CPU and 18GB of RAM.

6.3.1 Quantitative Results

The runtime benchmarks for the graph generation are shown in fig. 6.1 and fig. 6.2. The first plot compares runtimes for data fetching, source code translation to the CPG, CPG processing, and graph persistence, using 100 projects from the *DPDf* dataset. The second plot illustrates how runtime scales with source code size, normalized for direct comparison across processes. It is evident that the translation and processing steps have the greatest impact on the overall runtime as they scale with the size of the source code. In contrast, the data-fetching step remains mostly constant and of low variance with only a minor influence on the overall runtime. The graph persistence step becomes increasingly costly when dealing with large graphs.

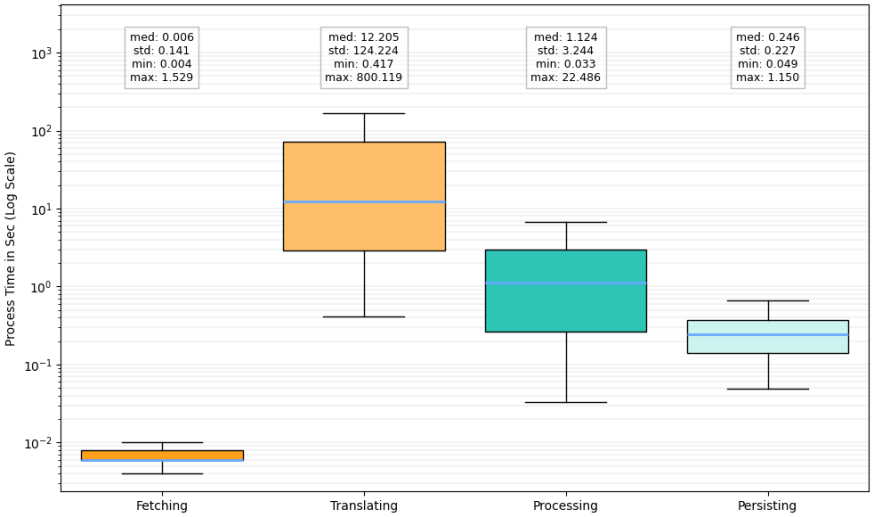


Figure 6.1: Runtime comparison for the graph generation processes.

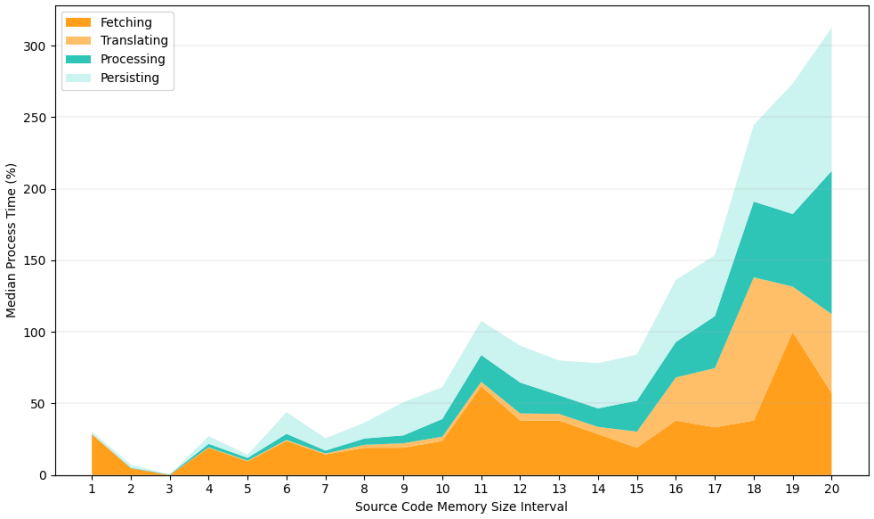
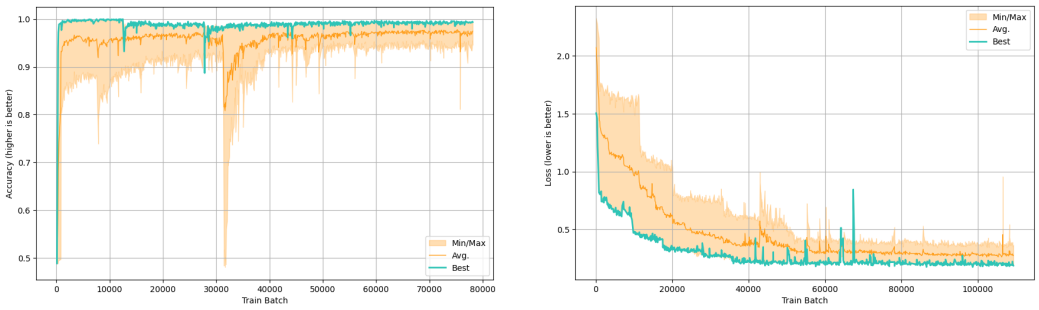


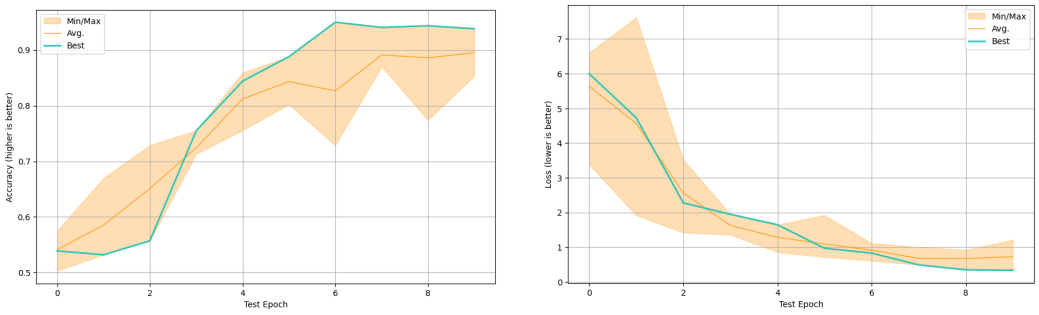
Figure 6.2: Runtime scaling for the graph generation processes.

The training of the *GLeMA Net* model is presented in fig. 6.3 and fig. 6.4. Multiple training runs were executed, and the training curves reveal the distinct stages of the curriculum training approach. An increase in the complexity limit initially causes a noticeable drop in accuracy. Especially in the middle of the training process, when the highest complexity is reached. At this point, the model experiences a significant decrease in accuracy but can adapt rapidly. The training experiments show that the best performance is achieved when the curriculum training strategy is applied. Convergence



(a) Train Accuracy (b) Train Loss

Figure 6.3: GLeMA Net train curves.

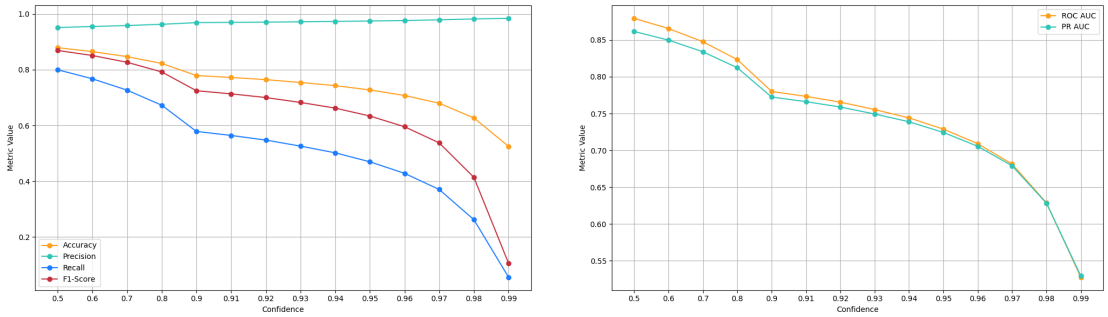


(a) Test Accuracy (b) Test Loss

Figure 6.4: GLeMA Net test curves.

is reached after around 10 epochs, and the early stopping mechanism is then activated to prevent overfitting. Results on the test data show that the model achieves high accuracy above 90% and increasingly low loss values, indicating that the model is able to generalize well to unseen data.

The performance metrics for the neural subgraph matching model are detailed in fig. 6.5, which represents the binary classification task of subgraph matching. The metrics are presented in relation to the prediction confidence. Only the predictions above or equal to the confidence threshold are considered as a potential positive match. The model achieves sufficiently high values across every metric on the test dataset. As the prediction confidence increases, there is a decrease in accuracy, recall, and F1-Score while precision increases marginally. This trend indicates that higher confidence levels lead to fewer false positives, although at the expense of a reduction in true positives. The ROC AUC and PR AUC values are high, indicating that the model is effective in distinguishing between positive and negative instances.



(a) Acc., Prec., Rec., F1-S. (b) ROC AUC, PR AUC

Figure 6.5: GLeMA Net metric curves by confidence.

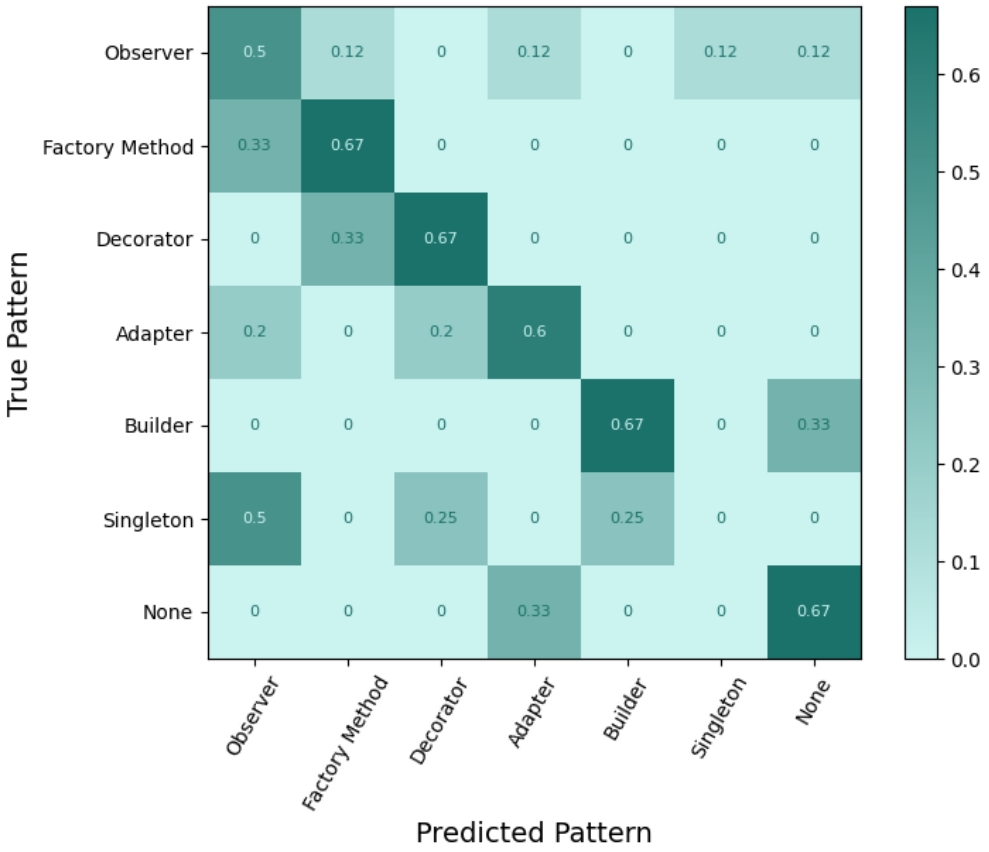
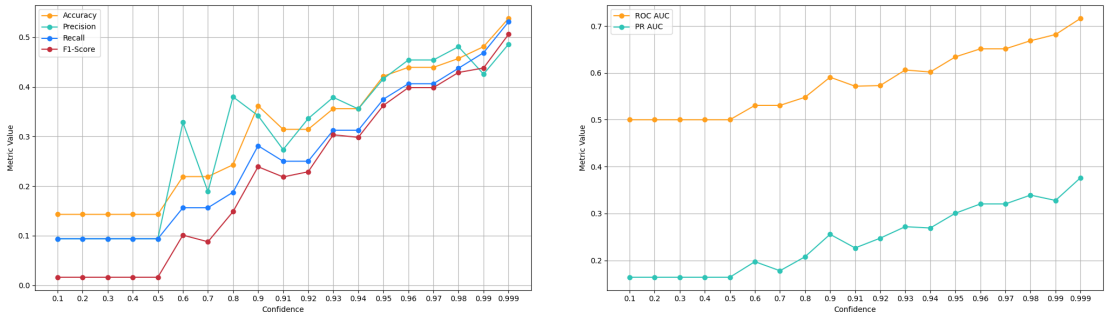


Figure 6.6: Confusion matrix of the pattern detection.

The actual design pattern detection results on the *P-MARt* dataset are presented in fig. 6.7 and table 6.3. The confusion matrix in fig. 6.6 is normalized by each pattern type, and the detection metrics, including accuracy, precision, recall, F1-Score, ROC



(a) Acc., Prec., Rec., F1-S.

(b) ROC AUC, PR AUC

Figure 6.7: Pattern detection metric curves by confidence.

Table 6.3: Design pattern detection results.

Pattern	Accuracy	Precision	Recall	F1-Score	ROC	PR
Observer	0.72	0.44	0.50	0.47	0.65	0.35
Factory Method	0.88	0.67	0.67	0.67	0.80	0.51
Decorator	0.91	0.50	0.67	0.57	0.80	0.36
Builder	0.94	0.67	0.67	0.67	0.82	0.48
Singleton	0.84	0.00	0.00	0.00	0.48	0.12
Adapter	0.88	0.60	0.60	0.60	0.76	0.42
Overall	0.86	0.48	0.52	0.50	0.72	0.37

AUC, and PR AUC, are also shown in relation to the prediction confidence. The results are summarized for each pattern and further compared with two other approaches using machine learning in table 6.4. The *DPDF* [44] approach uses Word2Vec embeddings for pattern detection, while the *DPF-GNN* [4] approach also uses a subgraph matching GNN but without the use of CPGs.

Table 6.4: Design pattern detection comparison.

	Own			*DPDF*			*DPF-GNN*		
Pattern	**Recall**	**Precision**	**F1-Score**	**Recall**	**Precision**	**F1-Score**	**Recall**	**Precision**	**F1-Score**
Adapter	0.60	0.60	0.60	**0.92**	**0.87**	**0.89**	0.89	0.82	0.85
Builder	0.67	0.67	0.67	**0.78**	**0.80**	**0.79**	-	-	-
Decorator	**0.67**	0.50	**0.57**	**0.60**	0.37	0.46	-	-	-
Factory Method	0.67	**0.67**	0.67	0.57	0.63	0.60	0.66	**0.73**	**0.69**
Observer	0.50	0.44	0.47	0.68	0.77	0.72	**0.91**	**0.89**	**0.90**
Singleton	0.00	0.00	0.00	0.43	0.40	0.42	**0.83**	**0.92**	**0.87**
Overall	0.52	0.48	0.50	0.66	0.64	0.64	**0.82**	**0.84**	**0.83**

The results demonstrate that the design patterns are detected with overall high accuracy. The detection has an overall high *ROC AUC* score, indicating that the approach is effective in distinguishing between the design pattern types. The approach was unable to detect a single instance of the *Singleton* pattern, often predicting it as a *Observer* pattern. In comparison to the other approaches, the proposed approach achieves lower metric scores but can achieve comparable results in some instances, like the *Factory Method* and *Decorator* patterns.

6.3.2 Qualitative Results

This section presents qualitative insights into the performance of the proposed approach by evaluating the generated RIGs, the node predictions of the GLeMA Net model, and concrete examples of pattern matching.

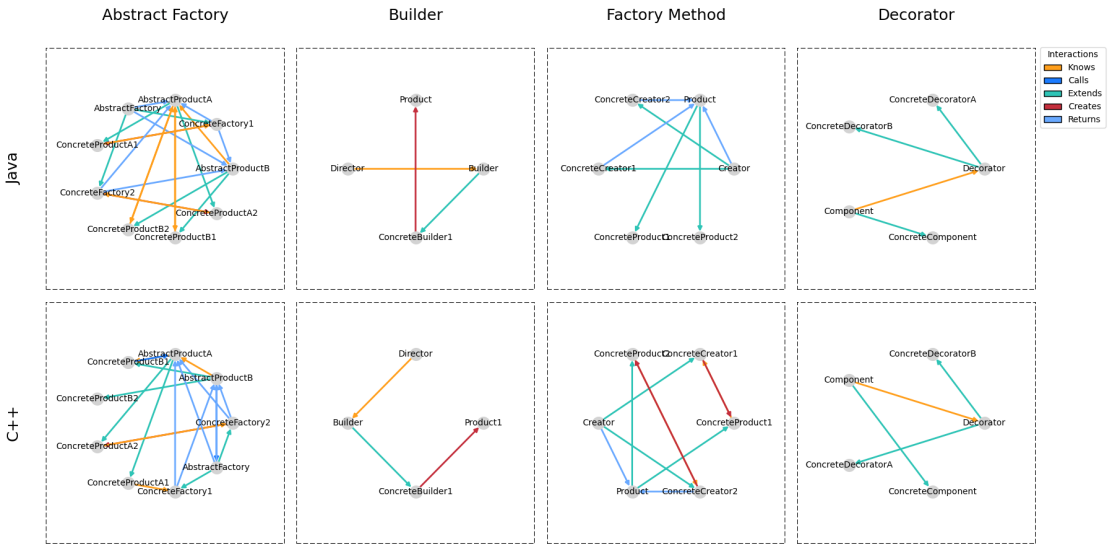


Figure 6.8: Language comparison of generated design pattern RIGs.

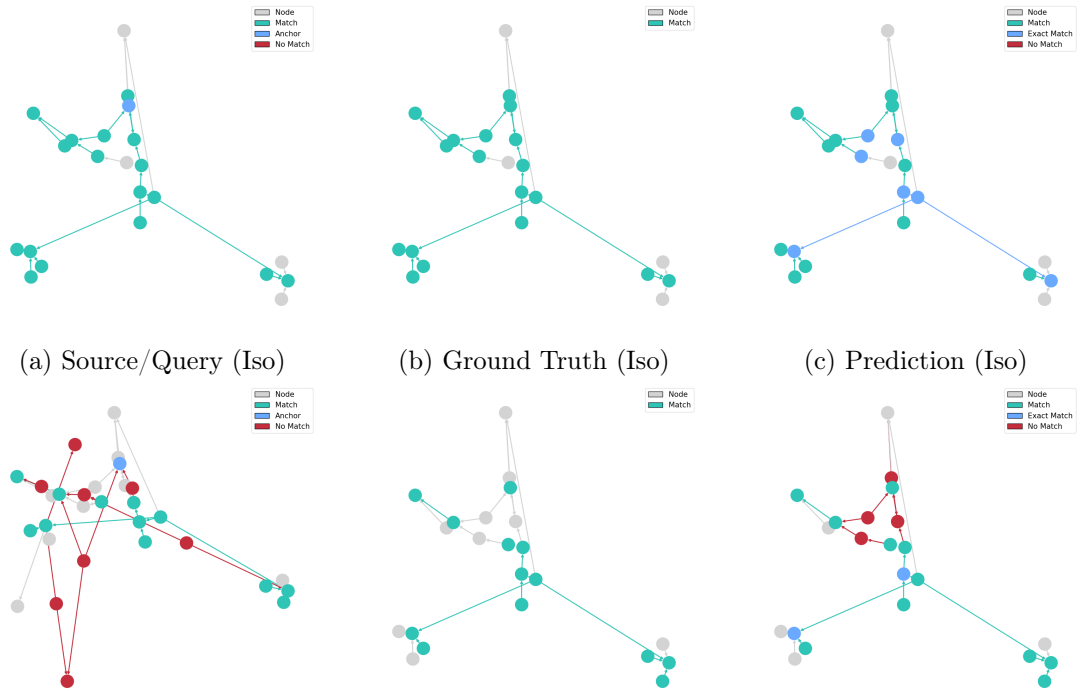
Figure 6.8 shows a comparison of the generated RIGs, including a selection of pattern examples written in *Java* and *C++*[1]. The results are showing that the generated RIGs are mostly equal in their general structure, and all records are included. Some differences appear in the interaction types, and some record relationships are missing. The interaction comparisons are detailed in table 6.5. The *Abstract Factory* pattern shows

[1]The design pattern examples are copied from *Refactoring Guru* (https://refactoring.guru/design-patterns/examples, Accessed 3. March 2025) and are translated into the respective languages.

Table 6.5: Language differences of generated design pattern RIGs.

	Records	Interactions	Knows	Calls	Extends	Creates	Returns
Abstract Factory							
C++	9	19	4	2	6	1	6
Java	9	21	7	0	6	2	6
$\lvert\Delta x\rvert$	0	2	3	2	0	1	0
Builder							
C++	4	3	1	0	1	1	0
Java	4	3	1	0	1	1	0
$\lvert\Delta x\rvert$	0	0	0	0	0	0	0
Factory Method							
C++	6	10	2	0	4	2	2
Java	6	7	0	0	4	0	3
$\lvert\Delta x\rvert$	0	3	2	0	0	2	1
Decorator							
C++	5	4	1	0	3	0	0
Java	5	4	1	0	3	0	0
$\lvert\Delta x\rvert$	0	0	0	0	0	0	0

the most differences. Additionally, some RIGs are less complex than others. This lower complexity makes them potentially harder to distinguish from non-pattern code.



(a) Source/Query (Iso) (b) Ground Truth (Iso) (c) Prediction (Iso)

(d) Source/Query (Non Iso) (e) Ground Truth (Non Iso) (f) Prediction (Non Iso)

Figure 6.9: Node predictions for queries in a source graph with the GLeMA Net model.

The node predictions of the GLeMA Net model are visualized to illustrate the mapping of nodes in a source graph to the query graph. Figure 6.9 shows an example with two queries that include isomorphic and non-isomorphic mappings on the same source graph. The first figure displays the source graph with the query graph and highlights their overlapping nodes and edges. The second figure shows the ground truth of the query graph in the source graph. Only query nodes and edges that are part of the source graph are included. The third figure shows the prediction for each query node. Exact matches are shown when a query node can be mapped to a node in the source graph. Matches are also displayed when a node can be mapped to any node in the source graph. In the case of the isomorphic query example, the model is able to map the query nodes in the source graph. Non-isomorphic queries are detected when some nodes in the query cannot be mapped to the source graph.

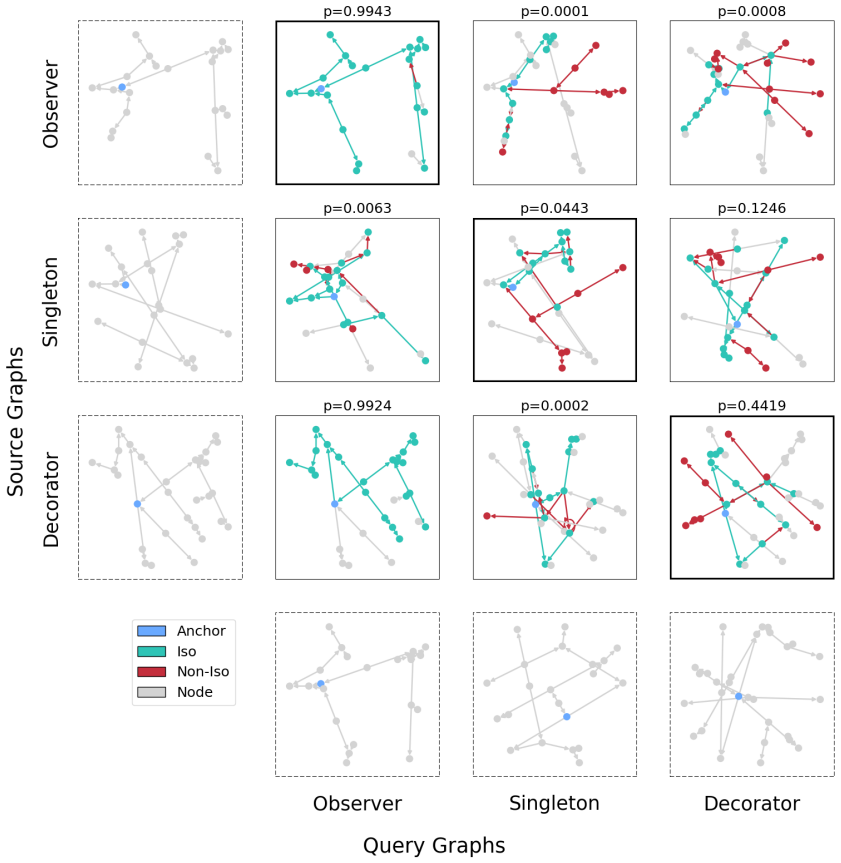


Figure 6.10: Matching examples of the *Observer*, *Singleton*, and *Decorator* patterns.

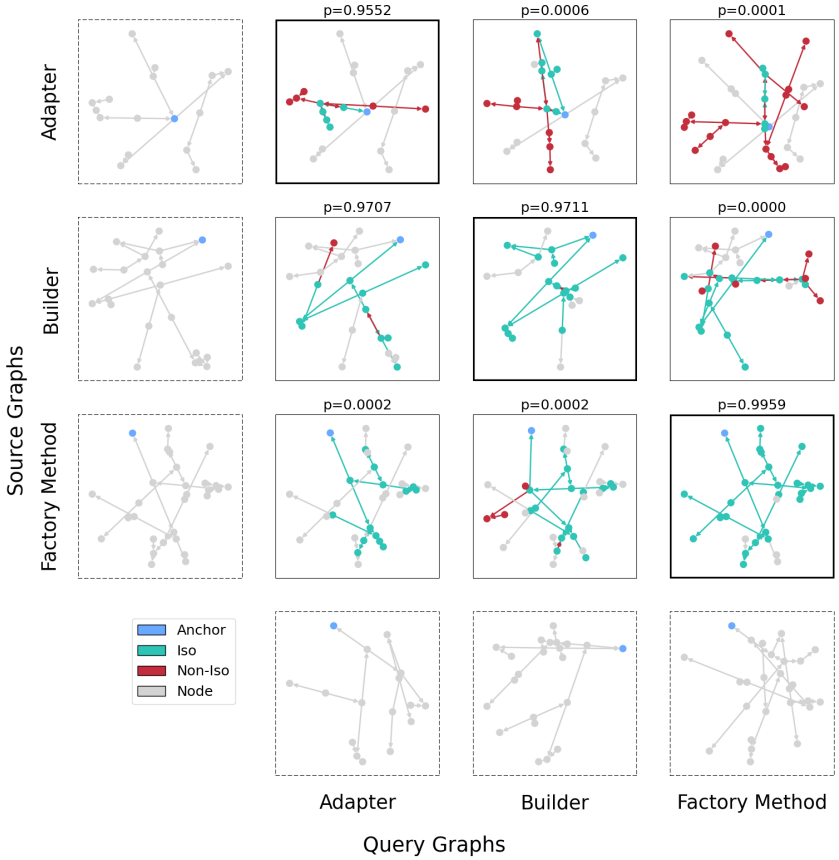


Figure 6.11: Matching examples of the *Adapter*, *Builder*, and *Factory Method* patterns.

To demonstrate the design pattern matching and voting process, figs. 6.10 and 6.11 show each pattern overlaid with all other patterns in a confusion matrix that displays the possible matches. In addition, the model prediction for each pair is presented. The matrix shows a high matching potential for most pairs with the same pattern type, and the model has a high prediction for most of those pairs. Some patterns of different types also exhibit a high matching potential. Examples include the *Builder* and *Adapter* as well as *Decorator* and *Observer* patterns. The model sometimes has a high subgraph prediction for non-matching pattern examples. Examples of this behavior include the *Adapter* pattern with itself or *Singleton Method* and *Decorator* patterns. An example of a false negative is the *Adapter* with the *Factory Method* pattern. In general, there are more false positives than false negatives in this set of examples.

6.4 Analysis

This section provides an analysis of the results obtained from the evaluation of the experiments for design pattern detection. The analysis focuses on the capabilities and limitations of the approach and addresses the research questions stated in section 1.2.

Table 6.6: Observations on the capabilities and limitations of the approach.

Capabilities	Limitations
• **Language-Independent Abstraction:** Translates source code into a language-independent representation using CPGs and reduces them into a RIG, capturing the characteristics of design patterns. • **Robustness & Generalization:** Tested on *Java* and *C++* examples showing similar structural properties, with normalization techniques that reduce language-specific variations. • **Accurate Subgraph Matching:** The GLeMA Net model achieves high accuracy and precision in detecting subgraphs. Training utilizes curriculum training and data synthesis, reaching over 90% recall and precision. • **Flexible Pattern Detection:** Detection relies on pattern examples rather than handcrafted templates, tested on a wide range of design patterns with an ROC AUC score of 72%.	• **Detection Accuracy Issues:** Exhibits a high false positive rate, where patterns are sometimes detected even when absent, and certain patterns (e.g., *Singleton* and *Adapter*) are frequently misclassified. For those patterns, the RIG model is too simple to capture their structural and behavioral characteristics. • **Model Sensitivity:** Accuracy is sensitive to the chosen confidence, leading to a tradeoff between precision and recall and difficulty in finding a suitable threshold. • **Scalability Challenges:** The generation of the CPGs is expensive in terms of runtime and memory usage, leading to scalability issues for larger projects. • **Consistency Challenges:** Minor inconsistencies in the generated RIGs across different programming languages may reduce detection precision in cross-language scenarios.

The proposed approach demonstrates promising capabilities in detecting design patterns with a language-independent source code representation and by enabling flexible detec-

tion with examples without the need for handcrafted templates or rule-based methods. However, it also faces significant challenges that need to be addressed. Based on the evaluation results, the following research questions are answered:

RQ 1. How can Code Property Graphs be effectively abstracted into a language-independent representation that captures the structural and behavioral characteristics of design patterns and is suitable for neural subgraph matching? The proposed approach abstracts source code into a language-independent representation by converting CPGs into a Record Interaction Graph. This transformation retains the essential structural and behavioral features that are critical for design pattern detection by tracing the shortest direct paths between records and aggregating them to predefined interaction types. The interaction types have to be modeled carefully to not compromise the expressiveness for especially small design patterns. The experiments show that the current RIG model introduces some ambiguities in those cases.

RQ 2. What techniques enable robust detection of design patterns that handle implementation variations without relying on handcrafted templates or rule-based definitions? The use of CPG representations abstracts away language-specific features, while the normalizing techniques of the RIGs reduce the effects of source code variations, resulting in a more robust detection process that accounts for differences in design pattern implementations. Aggregating multiple pattern examples for the neural subgraph matching, the approach generalizes and isolates the key structural features of design patterns, removing the need for handcrafted templates or rule-based definitions. The experiments indicate that this flexible method depends on high-quality, consistent pattern examples for the highest accuracy.

RQ 3. To what extent can a language-independent approach for design pattern detection achieve comparable accuracy to existing language-specific machine learning approaches? The evaluation demonstrates that the proposed approach achieves comparable accuracy for several design patterns, such as the *Factory Method* and *Decorator* patterns. However, the overall performance is affected by the high false positive rate and inconsistent pattern matching observed for simpler patterns like *Singleton*. These results indicate that while the approach shows promising potential, it requires further refinement to fully match the accuracy of language-specific machine learning approaches.

7 Conclusion

This chapter concludes the thesis by summarizing the research on detecting design patterns in source code using a language-independent approach. The chapter also discusses the challenges and limitations encountered. Finally, promising directions for improvements are described, and compatible applications for the approach are presented.

7.1 Summary

This thesis represents research on language-independent design pattern detection in source code. The work addresses a critical issue in software engineering, where existing methods on this problem rely on language-specific rules and handcrafted templates for detecting design patterns. Such limitations hinder the applicability and flexibility of design pattern detection for large-scale software projects and diverse code structures.

The proposed approach addresses these challenges by employing a language-independent abstraction of source code. For this, a Code Property Graph is used that abstracts the syntactic, semantic, and behavioral aspects of different programming languages into a uniform graph representation. The CPG is then processed and reduced into a Record Interaction Graph that captures the characteristic interactions between code entities. Neural subgraph matching is applied to query pattern instances, while a pattern voting mechanism combines the detection results from multiple pattern examples to improve the robustness of the detection process without the need for handcrafted templates.

The evaluation on real-world software projects demonstrates promising results for the prototype implementation and the capability of CPGs. The experiments show that the approach has the potential to generalize across multiple programming languages and varied design pattern implementations. At the same time, the results highlight a high rate of false positive detections and the need for a less reduced CPG abstraction to distinguish between design patterns and other code structures.

7.2 Discussion

To address the problem of language independence, the approach demonstrates that the usage of CPGs is effective in abstracting language-specific features and multiple source code aspects into a single graph representation. The problem with CPGs is the complexity of the generated graphs, which can be very large and expensive to compute. The approach resolves this issue by reducing the complexity of the CPG into a RIG, which captures the essential interactions between code entities while preserving the characteristics of design patterns. This reduction process allows for a more efficient computation and successfully enables the detection of design patterns across different programming languages.

The detection process includes the use of neural subgraph matching, performed with the GLeMA Net model. Multiple pattern examples are queried to improve the robustness of the detection process. The pattern voting mechanism combines the results from different pattern examples, which allows for a flexible and adaptable detection process. This is particularly beneficial for unseen patterns, as the approach does not rely on handcrafted templates or rule-based definitions. A more accurate approach for detecting specific design patterns would be to use a classification model, but this would limit the flexibility and adaptability of the approach because this kind of model does not generalize to unseen patterns. In contrast, the use of subgraph matching allows for a more general detection process. Assuming that there are enough pattern examples available and that the graph model is expressive enough, the approach can adapt to different design patterns and implementations.

While the approach enables the detection of design patterns in a language-independent manner, the evaluation reveals several limitations that must be considered. The graph model is not yet expressive enough to capture the characteristics of all design patterns. The experiments illustrate a high accuracy and precision in the subgraph matching task, but at the same time, the pattern detection exhibits a high false positive rate. Some design patterns, such as *Singleton* and *Adapter*, are frequently misclassified because the RIG may be too simple to differentiate these patterns from other code structures. In those cases, it is apparent that the chosen graph model is not expressive enough and has to be extended to capture the characteristics of the design patterns more precisely and in more detail.

The scalability of the graph generation process is another challenge. The conversion of source code into a CPG requires considerable runtime and memory resources. In comparison, the CPG processing and the detection process scale much better. This makes the approach less feasible for large source code projects and has to be improved. In addition, the translation process is not fully consistent across different programming languages. Minor differences in the generated RIGs can lead to less precise detection in cross-language scenarios.

The research on this problem demonstrates the capabilities of the proposed approach on language-independent design patterns detection. The implemented prototype achieves results that are not yet competitive with existing language-specific machine learning approaches but outlines a promising starting point for further improvements. The potential of the approach is evident in its flexibility and adaptability to different programming languages and design patterns. The combination of a language-independent abstraction of source code and neural subgraph matching techniques is a suitable approach to design patterns detection if the described challenges can be resolved.

7.3 Outlook

The approach can be improved in several ways. The use of an exact subgraph matching technique instead of the current approximative method would enhance the accuracy and precision of the detection process [77]. This would allow for a more reliable identification of design patterns and reduce the false positive rate. In addition, employing a more expressive and detailed graph model with less reduction of the original CPG may capture the details of design patterns with greater fidelity. To address the scalability issue, an interesting addition would be to partially generate and update the CPG instead of generating the complete graph from scratch.

The general approach enables the potential use in complex tasks and applications beyond design pattern detection. The approach may be applied to the reversed case, which is the detection of antipatterns or code smells [34, 37]. The concept can also be extended to architecture detection in source code [29], where the goal is to identify global architectural patterns and structures within software systems. The flexibility and adaptability of the approach make it potentially suitable for a wide variety of tasks that can be formulated as graph pattern detection problems.

Bibliography

[1] Mawal Ali and Mahmoud O. Elish. 2013. A Comparative Literature Survey of Design Patterns Impact on Software Quality. In *2013 International Conference on Information Science and Applications (ICISA)*. 1–7. https://doi.org/10.1109/ICISA.2013.6579460

[2] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 207–216. https://doi.org/10.1109/MSR.2013.6624029

[3] Awny Alnusair, Tian Zhao, and Gongjun Yan. 2014. Rule-Based Detection of Design Patterns in Program Code. *International Journal on Software Tools for Technology Transfer* 16, 3 (2014). https://doi.org/10.1007/s10009-013-0292-z

[4] Pasquale Ardimento, Lerina Aversano, Mario Luca Bernardi, and Marta Cimitile. 2022. Design Patterns Mining Using Neural Sub-Graph Matching. In *Proceedings of the ACM Symposium on Applied Computing*. https://doi.org/10.1145/3477314.3507073

[5] Jameleh Asaad and Elena Avksentieva. 2024. A Review of Approaches to Detecting Software Design Patterns. In *2024 35th Conference of Open Innovations Association (FRUCT)*. 142–148. https://doi.org/10.23919/FRUCT61870.2024.10516345

[6] Henri Basson, Mourad Bouneffa, Michiko Matsuda, Adeel Ahmad, Dukki Chung, and Eiji Arai. 2016. Qualitative Evaluation of Manufacturing Software Units Interoperability Using ISO 25000 Quality Model. In *Enterprise Interoperability VII*, Kai Mertins, Ricardo Jardim-Gonçalves, Keith Popplewell, and João P. Mendonça (Eds.). Springer International Publishing, Cham, 199–209. https://doi.org/10.1007/978-3-319-30957-6_16

[7] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Di Lucca. 2014. Design Pattern Detection Using a DSL-driven Graph Matching Approach. *Journal of Software: Evolution and Process* 26, 12 (2014). https://doi.org/10.1002/smr.1674

[8] Uzair Aslam Bhatti, Hao Tang, Guilu Wu, Shah Marjan, and Aamir Hussain. 2023. Deep Learning with Graph Convolutional Networks: An Overview and Latest Applications in Computational Intelligence. *International Journal of Intelligent Systems* 2023, 1 (2023), 8342104. https://doi.org/10.1155/2023/8342104

[9] Yang Cao and Yunwei Dong. 2023. Modeling and Discovering Data Race with Concurrent Code Property Graphs. In *Proceedings - 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2023*. Institute of Electrical and Electronics Engineers Inc., 646–653. https://doi.org/10.1109/QRS-C60940.2023.00074

[10] John M. Chambers. 2014. Object-Oriented Programming, Functional Programming and R. *Statist. Sci.* 29, 2 (May 2014), 167–180. https://doi.org/10.1214/13-STS452

[11] S.R. Chidamber and C.F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493. https://doi.org/10.1109/32.295895

[12] Miguel E. Coimbra, Alexandre P. Francisco, and Luís Veiga. 2021. An Analysis of the Graph Processing Landscape. *Journal of Big Data* 8, 1 (2021). https://doi.org/10.1186/s40537-021-00443-9

[13] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (Oct. 2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[14] Di Cui, Qiangqiang Wang, Siqi Wang, Jianlei Chi, Jianan Li, Lu Wang, and Qingshan Li. 2023. REMS: Recommending Extract Method Refactoring Opportunities via Multi-view Representation of Code Property Graph. In *IEEE International Conference on Program Comprehension*, Vol. 2023-May. https://doi.org/10.1109/ICPC58990.2023.00034

[15] Haneen Dabain, Ayesha Manzer, and Vassilios Tzerpos. 2015. Design Pattern Detection Using FINDER. In *Proceedings of the ACM Symposium on Applied Computing*, Vol. 13-17-April-2015. https://doi.org/10.1145/2695664.2695900

[16] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[17] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.* 7, 7 (March 2014), 517–528. https://doi.org/10.14778/2732286.2732289

[18] Davide Falessi, Giovanni Cantone, Rick Kazman, and Philippe Kruchten. 2011. Decision-Making Techniques for Software Architecture Design: A Comparative Survey. *ACM Comput. Surv.* 43, 4 (Oct. 2011), 33:1–33:28. https://doi.org/10.1145/1978802.1978812

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[20] Mohammed Ghazi Al-Obeidallah, Miltos Petridis, Stelios Kapetanakis, Mohammed Ghazi Al-Obeidallah MAl-Obeidallah, Miltos Petridis MPetridis, and Stelios Kapetanakis SKapetanakis. 2016. A Survey on Design Pattern Detection Approaches. *International Journal of Software Engineering (IJSE)* 7 (2016).

[21] Yann-Gaël Guéhéneuc. 2007. P-MARt: Pattern-like Micro Architecture Repository. *1st EuroPLoP Focus Group on Pattern Repositories* (Jan. 2007).

[22] Yann Gaël Guéhéneuc and Giuliano Antoniol. 2008. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Transactions on Software Engineering* 34, 5 (2008). https://doi.org/10.1109/TSE.2008.48

[23] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel Personalized Pagerank on Dynamic Graphs. *Proceedings of the VLDB Endowment* 11, 1 (2017). https://doi.org/10.14778/3151113.3151121

[24] Jiaxuan Han, Cheng Huang, Siqi Sun, Zhonglin Liu, and Jiayong Liu. 2023. bjXnet: An Improved Bug Localization Model Based on Code Property Graph and Attention Mechanism. *Automated Software Engineering* 30, 1 (March 2023), 12. https://doi.org/10.1007/s10515-023-00379-9

[25] Les Hatton. 2004. Safer Language Subsets: An Overview and a Case History, MISRA C. *Information and Software Technology* 46, 7 (June 2004), 465–472. https://doi.org/10.1016/j.infsof.2003.09.016

[26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. In *SIGMETRICS 2021 - Abstract Proceedings of the 2021 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*. https://doi.org/10.1145/3410220.3456276

[27] ISO/IEC 25010. 2011. ISO/IEC 25010:2011, Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models.

[28] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. https://doi.org/10.48550/arXiv.1609.02907 arXiv:1609.02907 [cs]

[29] Sirojiddin Komolov, Gcinizwe Dlamini, Swati Megha, and Manuel Mazzara. 2022. Towards Predicting Architectural Design Patterns: A Machine Learning Approach. *Computers* 11, 10 (Oct. 2022), 151. https://doi.org/10.3390/computers11100151

[30] Mariam Kouli and Abbas Rasoolzadegan. 2022. A Feature-Based Method for Detecting Design Patterns in Source Code. *Symmetry* 14, 7 (2022). https://doi.org/10.3390/sym14071491

[31] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. 2022. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems* 33, 12 (Dec. 2022), 6999–7019. https://doi.org/10.1109/TNNLS.2021.3084827

[32] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*. https://doi.org/10.1145/3460120.3484541

[33] Fan Liang, Cheng Qian, Wei Yu, David Griffith, and Nada Golmie. 2022. Survey of Graph Neural Networks and Applications. *Wireless Communications and Mobile Computing* 2022, 1 (2022), 9261537. https://doi.org/10.1155/2022/9261537

[34] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2021. Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering* 47, 9 (Sept. 2021), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

[35] Xin Liu and Yangqiu Song. 2021. Graph Convolutional Networks with Dual Message Passing for Subgraph Isomorphism Counting and Matching. https://doi.org/10.48550/arXiv.2112.08764 arXiv:2112.08764 [cs]

[36] Zhiyuan Liu and Jie Zhou. 2020. Graph Attention Networks. In *Introduction to Graph Neural Networks*, Zhiyuan Liu and Jie Zhou (Eds.). Springer International Publishing, Cham, 39–41. https://doi.org/10.1007/978-3-031-01587-8_7

[37] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. 2012. Support Vector Machines for Anti-Pattern Detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*. Association for Computing Machinery, New York, NY, USA, 278–281. https://doi.org/10.1145/2351676.2351723

[38] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. Style-Analyzer: Fixing Code Style Inconsistencies with Interpretable Unsupervised Algorithms. In *IEEE International Working Conference on Mining Software Repositories*, Vol. 2019-May. https://doi.org/10.1109/MSR.2019.00073

[39] Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, USA.

[40] Brian McFee and Gert Lanckriet. 2009. Partial Order Embedding with Multiple Kernels. 91. https://doi.org/10.1145/1553374.1553467

[41] Jose P. Miguel, David Mauricio, and Glen Rodriguez. 2014. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications* 5, 6 (Nov. 2014), 31–53. https://doi.org/10.5121/ijsea.2014.5603 arXiv:1412.2977 [cs]

[42] Md Nadim, Debajyoti Mondal, and Chanchal K. Roy. 2022. Leveraging Structural Properties of Source Code Graphs for Just-In-Time Bug Prediction. https://doi.org/10.48550/arXiv.2201.10137 arXiv:2201.10137 [cs]

[43] Ameneh Naghdipour, Seyed Mohammad Hossein Hasheminejad, and Roghayeh Leila Barmaki. 2023. Software Design Pattern Selection Approaches: A Systematic Literature Review. *Software: Practice and Experience* 53, 4 (2023), 1091–1122. https://doi.org/10.1002/spe.3176

[44] Najam Nazar, Aldeida Aleti, and Yaokun Zheng. 2022. Feature-Based Software Design Pattern Detection. *Journal of Systems and Software* 185 (2022). https://doi.org/10.1016/j.jss.2021.111179

[45] Mark Newman. 2010. *Networks: An Introduction*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199206650.001.0001

[46] Duc Q. Nguyen, Thanh Toan Nguyen, and Tho quan. 2023. xNeuSM: Explainable Neural Subgraph Matching with Graph Learnable Multi-hop Attention Networks. https://doi.org/10.48550/arXiv.2312.01612 arXiv:2312.01612 [cs]

[47] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. 2019. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.* 52, 5 (Sept. 2019), 88:1–88:48. https://doi.org/10.1145/3329786

[48] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2024. Comparing Semantic Graph Representations of Source Code: The Case of Automatic Feedback on Programming Assignments. *Computer Science and Information Systems* 21, 1 (2024). https://doi.org/10.2298/CSIS230615004P

[49] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2021. A Survey of Parametric Static Analysis. *ACM Comput. Surv.* 54, 7 (July 2021), 149:1–149:37. https://doi.org/10.1145/3464457

[50] Jeffy Jahfar Poozhithara, Hazeline U. Asuncion, and Brent Lagesse. 2022. Towards Lightweight Detection of Design Patterns in Source Code. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*. https://doi.org/10.18293/SEKE2022-167

[51] Mafizur Rahman, Md Showkat Hossain Chy, and Swapnil Saha. 2023. A Systematic Review on Software Design Patterns in Today's Perspective. In *2023 IEEE 11th*

International Conference on Serious Games and Applications for Health, SeGAH 2023. https://doi.org/10.1109/SeGAH57547.2023.10253758

[52] Md Mostafizer Rahman, Yutaka Watanobe, Atsushi Shirafuji, and Mohamed Hamada. 2023. Exploring Automated Code Evaluation Systems and Resources for Code Analysis: A Comprehensive Survey. https://doi.org/10.48550/arXiv.2307.08705 arXiv:2307.08705 [cs]

[53] Muhammad Ehsan Rana and Eddy Khonica. 2021. Impact of Design Principles and Patterns on Software Flexibility: An Experimental Evaluation Using Flexible Point (FXP). *Journal of Computer Science* 17, 7 (July 2021), 624–638. https://doi.org/10.3844/jcssp.2021.624.638

[54] John W. Raymond and Peter Willett. 2002. Maximum Common Subgraph Isomorphism Algorithms for the Matching of Chemical Structures. *Journal of Computer-Aided Molecular Design* 16, 7 (July 2002), 521–533. https://doi.org/10.1023/A:1021271615909

[55] Renita Raymond and S Margret Anouncia Savarimuthu. 2022. Software Design Patterns and Architecture Patterns –A Study Explored. In *2022 5th International Conference on Contemporary Computing and Informatics (IC3I)*. 1998–2006. https://doi.org/10.1109/IC3I56241.2022.10073279

[56] Rex, Ying, Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, and Jure Leskovec. 2020. Neural Subgraph Matching.

[57] Shylesh S. 2017. A Study of Software Development Life Cycle Process Models. https://doi.org/10.2139/ssrn.2988291 arXiv:2988291

[58] Nija Shi and Ronald A. Olsson. 2006. Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*. https://doi.org/10.1109/ASE.2006.57

[59] Samira Silva, Adiel Tuyishime, Tiziano Santilli, Patrizio Pelliccione, and Ludovico Iovino. 2023. Quality Metrics in Software Architecture. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 58–69. https://doi.org/10.1109/ICSA56044.2023.00014

[60] Jyoti Singh, Sripriya Roy Chowdhuri, Gosala Bethany, and Manjari Gupta. 2022. Detecting Design Patterns: A Hybrid Approach Based on Graph Matching and

Static Analysis. *Information Technology and Management* 23, 3 (2022). https://doi.org/10.1007/s10799-021-00339-3

[61] Syed Harmeet Singh and Imtiyaz Hassan. 2015. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment.

[62] Guangxin Su, Hanchen Wang, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2024. Simple and Deep Graph Attention Networks. *Knowledge-Based Systems* 293 (June 2024), 111649. https://doi.org/10.1016/j.knosys.2024.111649

[63] Chengcheng Sun, Chenhao Li, Xiang Lin, Tianji Zheng, Fanrong Meng, Xiaobin Rui, and Zhixiao Wang. 2023. Attention-Based Graph Neural Networks: A Survey. *Artificial Intelligence Review* 56, 2 (Nov. 2023), 2263–2310. https://doi.org/10.1007/s10462-023-10577-2

[64] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (May 2012), 788–799. https://doi.org/10.14778/2311906.2311907

[65] V. A. Traag, L. Waltman, and N. J. van Eck. 2019. From Louvain to Leiden: Guaranteeing Well-Connected Communities. *Scientific Reports* 9, 1 (2019). https://doi.org/10.1038/s41598-019-41695-z

[66] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. 2006. Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering* 32, 11 (2006). https://doi.org/10.1109/TSE.2006.112

[67] Srinivas Aditya Vaddadi, Ramya Thatikonda, Adithya Padthe, and Pandu Ranga Rao Arnepalli. 2023. Shift Left Testing Paradigm Process Implementation for Quality of Software Based on Fuzzy. *Soft Computing* (July 2023). https://doi.org/10.1007/s00500-023-08741-5

[68] Lilapati Waikhom and Ripon Patgiri. 2023. A Survey of Graph Neural Networks in Various Learning Paradigms: Methods, Applications, and Challenges. *Artificial Intelligence Review* 56, 7 (July 2023), 6295–6364. https://doi.org/10.1007/s10462-022-10321-2

[69] Guangtao Wang, Rex Ying, Jing Huang, and Jure Leskovec. 2021. Multi-Hop Attention Graph Neural Network. https://doi.org/10.48550/arXiv.2009.14332 arXiv:2009.14332 [cs]

[70] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement Learning Based Query Vertex Ordering Model for Subgraph Matching. https://doi.org/10.48550/arXiv.2201.11251 arXiv:2201.11251 [cs]

[71] Wentao Wang, Faryn Dumont, Nan Niu, and Glen Horton. 2022. Detecting Software Security Vulnerabilities Via Requirements Dependency Analysis. *IEEE Transactions on Software Engineering* 48, 5 (May 2022), 1665–1675. https://doi.org/10.1109/TSE.2020.3030745

[72] Konrad Weiss and Christian Banse. 2022. A Language-Independent Analysis Platform for Source Code. https://doi.org/10.48550/arXiv.2203.08424 arXiv:2203.08424

[73] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs. In *Proceedings of the ACM Web Conference 2022 (WWW '22)*. Association for Computing Machinery, New York, NY, USA, 755–766. https://doi.org/10.1145/3485447.3512235

[74] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (Jan. 2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[75] Wang Xiaomeng, Zhang Tao, Wu Runpu, Xin Wei, and Hou Changyu. 2018. CPGVA: Code Property Graph Based Vulnerability Analysis by Deep Learning. In *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*. 184–188. https://doi.org/10.1109/ICAIT.2018.8686548

[76] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings - IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2014.44

[77] Bin Yang, Zhaonian Zou, and Jianxiong Ye. 2025. GNN-based Anchor Embedding for Exact Subgraph Matching. https://doi.org/10.48550/arXiv.2502.00031 arXiv:2502.00031 [cs]

[78] Hadis Yarahmadi and Seyed Mohammad Hossein Hasheminejad. 2020. Design Pattern Detection Approaches: A Systematic Review of the Literature. *Artificial Intelligence Review* 53, 8 (Dec. 2020), 5789–5846. https://doi.org/10.1007/s10462-020-09834-5

[79] Jin Y. Yen. 1970. An Algorithm for Finding Shortest Routes from All Source Nodes to a given Destination in General Networks. *Quart. Appl. Math.* 27, 4 (1970), 526–530. https://doi.org/10.1090/qam/253822

[80] Jiaxuan You, Jonathan Gomes-Selman, Rex Ying, and Jure Leskovec. 2021. Identity-Aware Graph Neural Networks. https://doi.org/10.48550/arXiv.2101.10320 arXiv:2101.10320 [cs]

[81] Cheng Zhang and David Budgen. 2012. What Do We Know about the Effectiveness of Software Design Patterns? *IEEE Transactions on Software Engineering* 38, 5 (Sept. 2012), 1213–1231. https://doi.org/10.1109/TSE.2011.79

[82] Chunyong Zhang, Bin Liu, Yang Xin, and Liangwei Yao. 2023. CPVD: Cross Project Vulnerability Detection Based on Graph Attention Network and Domain Adaptation. *IEEE Transactions on Software Engineering* 49, 8 (Aug. 2023), 4152–4168. https://doi.org/10.1109/TSE.2023.3285910

[83] G.P. Zhang. 2000. Neural Networks for Classification: A Survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 30, 4 (Nov. 2000), 451–462. https://doi.org/10.1109/5326.897072

[84] Chunhui Zhao, Tengfei Tu, Cheng Wang, and Sujuan Qin. 2023. VulPathsFinder: A Static Method for Finding Vulnerable Paths in PHP Applications Based on CPG. *Applied Sciences* 13, 16 (Jan. 2023), 9240. https://doi.org/10.3390/app13169240

[85] Huaisheng Zhu, Guoji Fu, Zhimeng Guo, Zhiwei Zhang, Teng Xiao, and Suhang Wang. 2023. Fairness-Aware Message Passing for Graph Neural Networks. https://doi.org/10.48550/arXiv.2306.11132 arXiv:2306.11132 [cs]

A Appendix

A.1 Additonal Evaluation Results

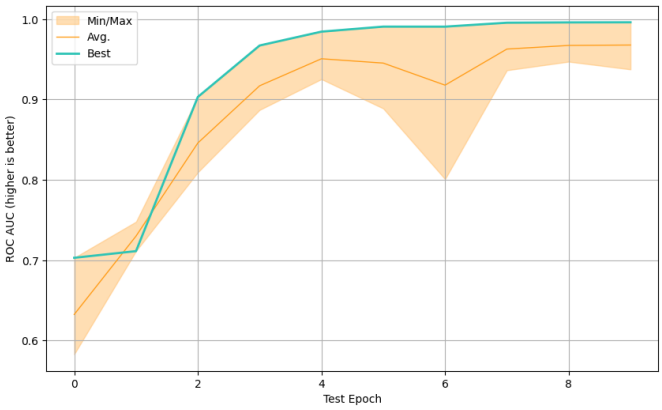


Figure A.1: GLeMA Net test ROC AUC curve.

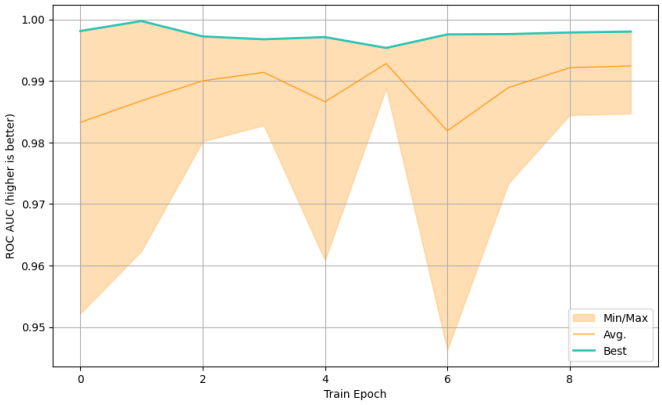


Figure A.2: GLeMA Net train ROC AUC curve.

Figure A.3: Design pattern matching examples.

A.2 Tools and Software

The tools used in addressing the subject of this thesis are listed in table A.1.

Table A.1: Overview of the used tools.

Tool	Usage
LaTeX	Typesetting and layout tool used for creating this document.
Zotero	Reference management software used for managing the bibliography.
VS Code	Text editor used for writing this document.
IntelliJ IDEA	Integrated development environment used for writing the Java code.
PyCharm	Integrated development environment used for writing the Python code.
Git	Version control system used for managing the source code revisions.
Neo4j Bloom	Visualization tool used for exploring the Neo4j graph database.
Docker	Containerization platform used for managing the application environment.

A.3 Content of the Electronic Appendix

The enclosed CD contains the following files and directories:

- thesis.pdf
- source
  - datasets
  - generation
  - matching
  - docker-compose.yml
  - README.md
  - run.py
- results
  - generation
  - matching

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original