OBJECT-ORIENTED MODELING OF SOCIAL NETWORK GRAPHS IN JAVA: A GRAPH-BASED APPROACH

MILLAT UMIDI UNIVERSITY
Department of Computer Science and Information Technology
Submitted by:
Tursunxo'jayeva Dilshodaxon
Instructor:
Pirnazarov Muslimbek

A Research Paper Submitted in Partial Fulfillment of the Requirements for the Course in Object-Oriented Programming with Java

ABSTRACT

Social networks are among the most widely used software systems in the world, connecting billions of users through relationships that can be modeled as graphs. Designing the software behind such systems requires a programming approach that is organized, flexible, and maintainable. This paper shows how the four principles of Object-Oriented Programming (OOP) — encapsulation, inheritance, polymorphism, and abstraction — can be applied in Java to build a graph-based social network model. A three-layer architecture is proposed, separating user entities, graph structure, and service logic into distinct components.

The paper implements Breadth-First Search (BFS) and Depth-First Search (DFS) within an OOP framework and applies three classic design patterns from Gamma et al. (1994): Observer, Factory, and Iterator. A demonstration with five users shows how BFS discovers connections and finds shortest paths. The main finding is that OOP principles make social network graph modeling clean, modular, and easy to extend. This paper is a course demonstration of OOP design and graph algorithm competency, not a novel research contribution.

Keywords: *Object-Oriented Programming, Java, Social Network, Graph Theory, Encapsulation, BFS, DFS, Design Patterns, Adjacency List.*

1. Introduction

Social networking platforms are used by billions of people every day. At a software level, each platform is a large graph: every user is a node, and every relationship is an edge. This structure makes features like friend suggestions, connection paths, and community detection possible. Designing such a graph well in

software is both a practically relevant and academically important problem for any computer science student working with Java. Java is a natural choice for this task. As one of the most widely used Object-Oriented Programming languages, it provides abstract classes, interfaces, inheritance, and strong typing, all of which support building organized, maintainable systems. OOP maps directly onto the conceptual structure of a social network: a User, a Relationship, and a Graph are natural objects. This paper addresses three design questions. First, how can encapsulation and inheritance be used to model users and relationships? Second, how can an adjacency list be implemented using OOP patterns? Third, how can BFS and DFS be built into an OOP design while staying efficient and readable? The paper is organized as follows: Section 2 reviews literature; Section 3 covers background; Section 4 presents the design; Section 5 shows code; Section 6 gives a demonstration; Sections 7 to 9 cover comparisons, limitations, and conclusions.

2. Literature Review

Wasserman and Faust (1994) established the foundational graph model for social networks, treating users as nodes and relationships as edges. Gamma, Helm, Johnson, and Vlissides (1994) catalogued 23 OOP design patterns; three of these are applied directly here: Observer, Factory, and Iterator. Bloch (2018) in *Effective Java* argues for programming to interfaces and keeping fields private, both of which are followed throughout this paper. Sedgewick and Wayne (2011) provided object-oriented BFS and DFS implementations in Java that served as the reference for Section 5. Cormen, Leiserson, Rivest, and Stein (2009) provided the formal O(V + E) time complexity analysis for BFS and DFS on adjacency list graphs used in Section 6. Horstmann (2022) served as the reference for Java APIs including HashMap, ArrayList, LinkedList, and UUID.

3. OOP and Graph Theory Background

3.1 The Four OOP Principles

Encapsulation means keeping an object's data private and only allowing access through defined methods. A User object, for example, keeps userId and email private; outside code uses getters to read them. Inheritance allows a subclass to take on the fields and methods of a parent class. A RegularUser extending User automatically has all of User's fields and only needs to add what is specific to it. Polymorphism means different types can be used through one common type. If RegularUser, BusinessUser, and AdminUser all extend User, then BFS can traverse a mixed graph without knowing the specific type of each node. Abstraction hides internal complexity behind a simple interface. A Traversable interface declares that implementing classes must have BFS and DFS methods without specifying how they work internally.

3.2 Graph Theory Essentials

A graph G = (V, E) is a set of nodes V connected by edges E. In a social network, nodes are users and edges are relationships. An undirected graph has edges that go both ways, as in a friendship. A directed graph has one-way edges, as in a follow. BFS visits nodes level by level using a queue and finds shortest paths in unweighted graphs. DFS uses a stack and explores as deeply as possible before backtracking. Both algorithms run in O(V + E) time on an adjacency list. Java fits naturally: HashMap, ArrayList, LinkedList, and HashSet from the Collections Framework implement the adjacency list structure cleanly.

4. Proposed System Design

4.1 Three-Layer Architecture

The system has three layers, each with one responsibility. The Entity Layer contains the User hierarchy and Relationship implementations. The Graph Layer contains SocialGraph and its adjacency list. The Service Layer contains UserFactory and the notification system. Separating these layers means that changing one does not require touching the others, which is a core goal of OOP design.

4.2 User Class Hierarchy

The abstract class User stores userId (UUID-generated), username, email, and joinDate as private fields, and declares two abstract methods: getProfile() and getConnectionType(). Three concrete subclasses extend it: RegularUser adds a bio field; BusinessUser adds companyName and industry; AdminUser adds permissionLevel. All three inherit the shared fields and override the abstract methods.

4.3 Relationship Interface and Design Patterns

The Relationship interface declares getSource(), getTarget(), getWeight(), and getType(). Three classes implement it: FriendshipRelation (mutual), FollowRelation (directed), and BlockRelation (restrictive). Adding a new relationship type only requires writing one new class, with no changes to existing code. This is the Open/Closed Principle in practice. Three design patterns from Gamma et al. (1994) are also applied. The Observer Pattern lets User objects register for event notifications, decoupling event sources from handlers. The Factory Pattern routes all user creation through UserFactory.createUser(), centralising instantiation. The Iterator Pattern wraps the adjacency list so external code can loop through users without seeing the internal HashMap.

4.4 SocialGraph Class

SocialGraph holds a HashMap<User, List<Relationship>>, an adjacency list providing O(1) average connection lookup. It implements two interfaces: Traversable (requiring BFS and DFS) and Searchable (requiring findUser, findShortestPath, and

suggestConnections). Declaring these as interface requirements means any consumer of SocialGraph can rely on these methods without depending on the internal implementation.

5. Java Implementation

5.1 Abstract User Class

All fields are private (encapsulation), abstract methods declare a contract (abstraction), and getters provide controlled read-only access:

```
public abstract class User {
    private final String userId;
    private String username;
    private String email;
    private LocalDateTime joinDate;
    public User(String username, String email) {
        this.userId   = UUID.randomUUID().toString();
        this.username = username;
        this.email    = email;
        this.joinDate = LocalDateTime.now();
    }
    public String getUserId()   { return userId; }
    public String getUsername() { return username; }
    public abstract String getProfile();
    public abstract String getConnectionType();
}
```

5.2 RegularUser Subclass

RegularUser calls super() to initialize shared fields, then adds a bio field and provides its own method implementations (inheritance):

```
public class RegularUser extends User {
    private String bio;
    public RegularUser(String username, String email, String bio) {
        super(username, email);
        this.bio = bio;
    }
    @Override
    public String getProfile() {
```

```
        return "[Regular] " + getUsername() + " | Bio: " + bio;
    }
    @Override
    public String getConnectionType() { return "FRIEND"; }
}
```

5.3 SocialGraph, BFS, Shortest Path, and UserFactory

The following snippets show the core graph operations. SocialGraph manages the adjacency list; BFS visits nodes level by level; findShortestPath uses a parent map to reconstruct the minimum-hop route; UserFactory applies the Factory Pattern:

```
// SocialGraph adjacency list operations
public class SocialGraph implements Traversable, Searchable {
    private Map<User, List<Relationship>> adj = new HashMap<>();
    public void addUser(User u) { adj.putIfAbsent(u, new ArrayList<>());
}
    public void addRelationship(User src, Relationship r) {
        if (adj.containsKey(src)) adj.get(src).add(r); }

    // BFS: visits all nodes level by level
    public List<User> breadthFirstSearch(User start) {
        List<User> visited = new ArrayList<>();
        Queue<User> queue = new LinkedList<>();
        Set<User> seen = new HashSet<>();
        queue.add(start); seen.add(start);
        while (!queue.isEmpty()) {
            User cur = queue.poll(); visited.add(cur);
            for (Relationship r : adj.get(cur)) {
                User nb = r.getTarget();
                if (!seen.contains(nb)) { seen.add(nb); queue.add(nb); }
            }
        }
        return visited;
    }

    // Shortest path: BFS + parent map
    public List<User> findShortestPath(User src, User tgt) {
```

```
            Map<User,User> parent = new HashMap<>();
            Queue<User> queue = new LinkedList<>();
            queue.add(src); parent.put(src, null);
            while (!queue.isEmpty()) {
                User cur = queue.poll();
                if (cur.equals(tgt)) break;
                for (Relationship r : adj.get(cur)) {
                    User nb = r.getTarget();
                    if (!parent.containsKey(nb)) { parent.put(nb, cur);
queue.add(nb); }
                }
            }
            return reconstructPath(parent, src, tgt);
        }
    }

    // Factory Pattern: centralises user creation
    public class UserFactory {
        public static User createUser(UserType t, String u, String e) {
            switch (t) {
                case REGULAR:  return new RegularUser(u, e, "");
                case BUSINESS: return new BusinessUser(u, e, "", "");
                case ADMIN:    return new AdminUser(u, e, 1);
                default: throw new IllegalArgumentException("Unknown type");
            }
        }
    }
```

6. Demonstration Output

6.1 Sample Network and BFS Console Output

The following demonstration creates five users, adds relationships, and runs BFS and the shortest-path algorithm. The output shown below is simulated to illustrate what the system produces when run:

```
    // Setup: 3 regular users, 2 business accounts
```

```
        User alice  = UserFactory.createUser(REGULAR,  "Alice",
"alice@mail.com");
        User bob    = UserFactory.createUser(REGULAR,  "Bob",
"bob@mail.com");
        User carol  = UserFactory.createUser(REGULAR,  "Carol",
"carol@mail.com");
        User techCo = UserFactory.createUser(BUSINESS, "TechCo",
"info@techco.com");
        User devHub = UserFactory.createUser(BUSINESS, "DevHub",
"hi@devhub.io");

        graph.addRelationship(alice,  new FriendshipRelation(alice,  bob));
        graph.addRelationship(alice,  new FollowRelation(alice,  techCo));
        graph.addRelationship(bob,    new FriendshipRelation(bob,    carol));
        graph.addRelationship(carol,  new FollowRelation(carol,  devHub));

        // --- Demonstration output: BFS from Alice ---
        // Visiting: Alice
        // Visiting: Bob
        // Visiting: TechCo
        // Visiting: Carol
        // Visiting: DevHub
        // BFS order from Alice: [Alice, Bob, TechCo, Carol, DevHub]

        // --- Demonstration output: Shortest path Alice to DevHub ---
        // Path: Alice -> Bob -> Carol -> DevHub  (3 hops)
```

6.2 BFS vs DFS on a 6-Node Line Graph

Table 1 shows traversal order for BFS and DFS on the line graph A-B-C-D-E-F starting from node A. In a line graph, both algorithms visit nodes in the same order; the difference becomes clear in graphs with branching paths, where BFS spreads outward while DFS dives deep before backtracking.

Step	BFS (Queue)	DFS (Stack)	Level/Depth
1	A	A	0
2	B	B	1

3	C	C	2
4	D	D	3
5	E	E	4
6	F	F	5

Table 1. BFS vs DFS traversal on a 6-node line graph A-B-C-D-E-F starting from A.

6.3 Design Metrics

Table 2 summarises the key structural properties of the implementation:

Design Metric	Value	OOP Principle
Total classes and interfaces	12	All four principles
Abstract classes	2 (User, GraphBase)	Abstraction
Interfaces	4 (Relationship, Traversable, Searchable, Observable)	Polymorphism
Concrete user types	3 (Regular, Business, Admin)	Inheritance
All fields private	Yes (100%)	Encapsulation
Design patterns	3 (Observer, Factory, Iterator)	All principles
BFS/DFS time complexity	O(V + E)	Cormen et al., 2009

Table 2. Structural design metrics of the proposed social network model.

7. Analysis of OOP Design Choices

Encapsulation kept all User fields private, preventing accidental modification of IDs or emails. This is directly relevant to data protection in any application handling personal information. Inheritance eliminated code duplication across the three user types; all shared logic lives in User and each subclass only adds what is specific to it. Without inheritance, the same four fields and their getters would be repeated in each class. Polymorphism allowed BFS and DFS to operate uniformly across RegularUser, BusinessUser, and AdminUser nodes. The traversal code never checks the specific type of a node; it treats every node as a User. Abstraction through interfaces proved most

valuable for extensibility. Adding a new relationship type requires only one new class implementing Relationship. The SocialGraph, BFS algorithm, and all service code keep working without modification. The BFS and DFS algorithms both run in O(V + E) time, optimal for adjacency list graphs (Cormen et al., 2009), and the HashMap provides O(1) average connection lookup.

8. Comparison with Real-World Platforms

The architecture in this paper is conceptually similar to how major social platforms organize their internal data. Facebook's social graph is conceptually similar to the User-and-Relationship model here: entities are typed objects and connections are typed associations. LinkedIn is known to separate member data, connection data, and query operations into distinct layers, which is conceptually similar to the three-layer (Entity, Graph, Service) structure in Section 4. Twitter's event notification system uses a publish-subscribe model that is structurally similar to the Observer Pattern applied here. These comparisons are conceptual only. This paper does not claim to replicate those platforms or to have access to their internal code. The point is that the OOP ideas applied here represent sound architectural thinking that scales from student projects to production systems.

9. Limitations and Future Work

Three main limitations apply. First, the system is entirely in-memory: all data is lost when the program exits. A production system would need a persistence layer using JPA with a database like MySQL, or a graph-native database like Neo4j. Second, the system is not thread-safe. Concurrent reads and writes without synchronization produce inconsistent results; Java's ConcurrentHashMap and synchronized blocks would be needed. Third, friend suggestions use only network distance. Real platforms combine shared interests, activity history, and location to generate relevant suggestions. A weighted edge model would be a meaningful improvement. Future work could add community detection algorithms such as Louvain to identify user clusters, and a persistence layer to store the graph across sessions. Because the design is interface-based throughout, these extensions can be added without modifying any existing code.

10. Conclusion

Encapsulation protected user data behind private fields. Inheritance eliminated duplication by consolidating shared logic in the abstract User class. Polymorphism allowed BFS and DFS to work uniformly across all user types. Abstraction through interfaces made the system easy to extend without breaking existing code. Three design patterns solved specific recurring problems: Observer for notifications, Factory for object creation, and Iterator for safe graph traversal. The BFS and DFS algorithms run in O(V + E) time on the adjacency list representation, which is optimal for this class of

problem. The demonstration showed BFS correctly visiting five users in order and finding the three-hop path from Alice to DevHub. The comparison with real platforms confirmed that the OOP design ideas applied here are conceptually aligned with how large-scale social systems are structured. Java OOP is an effective and natural approach to social network modeling: a well-structured design produces code that is readable, maintainable, and built to grow.

REFERENCES

1. Bloch, J. (2018). Effective Java (3rd ed.). Addison-Wesley Professional.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.
3. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional.
4. Horstmann, C. S. (2022). Core Java: Fundamentals (12th ed., Vol. 1). Oracle Press / Addison-Wesley Professional.
5. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
6. Wasserman, S., & Faust, K. (1994). Social network analysis: Methods and applications.
7. Cambridge University Press. https://doi.org/10.1017/CBO9780511815478