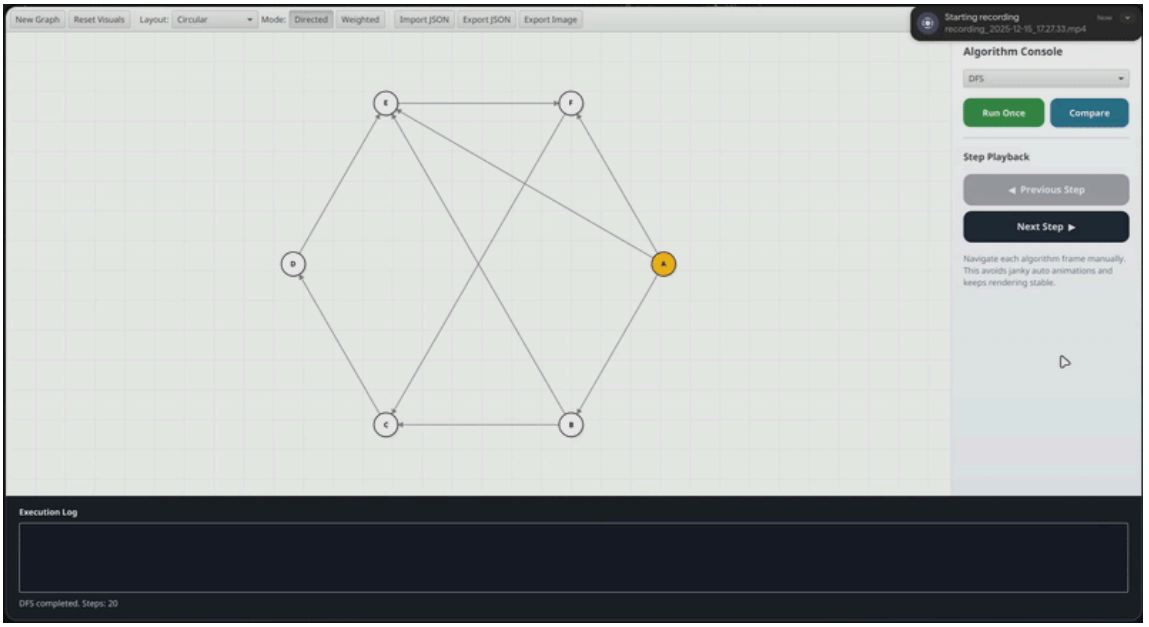
GraphVizFX Code Review Report

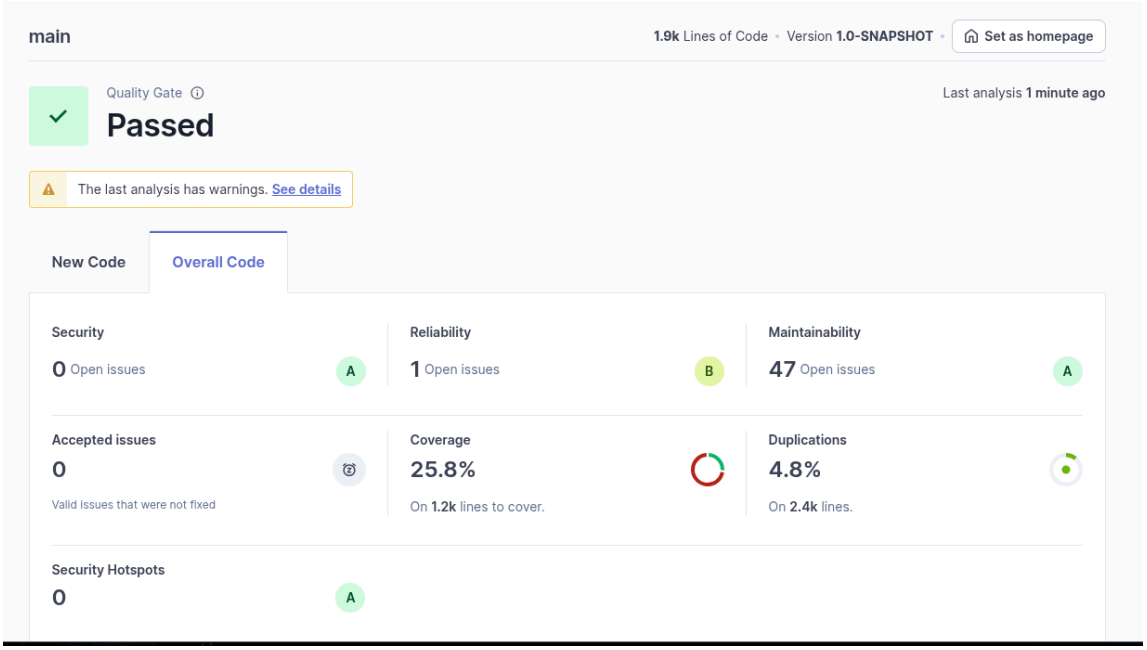
1. Executive Summary



- JavaFX-based graph visualizer with interactive canvas, multiple layouts, and animated algorithm traces; packages are cleanly separated and interfaces enable extension.
- Strengths: clear MVC-ish layering, reusable `GraphAlgorithm` / `LayoutAlgorithm` strategies, and a meaningful JUnit suite that exercises algorithms, layouts, and JSON I/O.
- Critical weaknesses: weak domain encapsulation (mutable collections, identity-only edge checks), regex-driven parsers that risk data loss/corruption, and algorithm efficiency/validation gaps (MST correctness, neighbor scanning, shallow visual-state copying).

2. Code Quality Audit

2.1 Static Analysis (SonarQube)



- Coverage at **25.8%** directly reinforces the Executive Summary's critical weakness around insufficient testing.
- Despite an **A** in Maintainability, the **B** in Reliability signals at least one non-trivial bug that should be triaged and fixed before release.

3. Architecture & Design Patterns

- **MVC assessment:**
  - `app/GraphVizFX` (View bootstrap) wires `GraphController` (Controller) and `GraphModel` (Model) with `GraphCanvas` (View). Separation is decent for algorithm execution and layout, but the View performs model mutation (node/edge creation/deletion) directly via controller without validation in the model itself, weakening encapsulation.
  - `GraphCanvas` mixes rendering and input handling, causing UI logic to leak into the View layer; `GraphController` is thin and delegates almost everything to `GraphModel` with no domain guards.
- **Patterns in use:**
  - **Strategy:** `GraphAlgorithm` and `LayoutAlgorithm` interfaces with concrete BFS/DFS/Dijkstra/A*/Prim/Kruskal and Grid/Circular/Force-Directed implementations. Good extensibility, but algorithms share neighbor-scanning inefficiencies and lack shared adjacency helpers.
  - **Factory-ish:** `AlgorithmEngine.createAlgorithm` acts as a simple factory switch; no error recovery or registration mechanism, so open/closed is limited.
  - **Singleton:** none. No global state—good.
  - **Observer/Event:** not used; UI polling and redraws are manual.
- **Interface vs concrete use:** Interfaces are present but concrete classes are manipulated directly (e.g., `new GraphController()`), and `GraphModel` exposes mutability, so substitution is limited. No DI; testing stubs would require refactoring.

4. Algorithmic Analysis

- **Shared inefficiency:** All algorithms iterate `graph.getEdges()` to find neighbors each step. Time blows up to O(VE) even for sparse graphs; space remains minimal but repeated map mutation/copying in `VisualState` can be costly.
- **Complexities (current implementation):**
  - **BFS (`BFSAlgorithm`)**: Time O(V·E) due to global edge scans; Space O(V+E) for queue, visited, and color maps.
  - **DFS (`DFSAlgorithm`)**: Time O(V·E); Space O(V+E) for stack, visited, and colors.
  - **Dijkstra (`DijkstraAlgorithm`)**: Time O(V·E + V log V) (PQ plus edge scans); Space O(V+E) for distances, PQ, settled, colors.
  - *A (`AStarAlgorithm`)**: Time O(V·E + V log V); Space O(V+E) for g-scores, parents, open/closed, colors.
  - **Prim (`PrimAlgorithm`)**: Time O(E log E + V·E) because of re-adding adjacent edges and global scans; Space O(E) for PQ plus color maps.
  - **Kruskal (`KruskalAlgorithm`)**: Time O(E log E + α(V)) for sorting and union-find, but still iterates all edges; Space O(E+V) for sorted edges and parent map.
- **Critique:** Building adjacency lists once per run would reduce BFS/DFS to O(V+E), Dijkstra/A* to O((V+E) log V), and clean up edge-color mutation. MSTs should stop after `|V|-1` edges and reject directed/disconnected graphs.

5. Code Quality & SOLID

- **Single Responsibility:**
  - `GraphCanvas` handles rendering, hit-testing, node/edge creation/deletion, and context menus—multiple responsibilities.
  - `GraphVizFX` combines UI composition, state management, status/log control, and algorithm orchestration.
- **Open/Closed:**
  - Algorithms/layouts can be added via new classes, but `AlgorithmEngine` switch violates OCP; adding types requires edits there.
  - Parsers (`JSONImporter`, `OSMImporter`) are rigid regex implementations; swapping parsers requires rewriting classes.
- **Liskov Substitution:** Interfaces are simple and respected; however, `GraphModel`'s exposed lists allow invariants to be broken, so substitutable implementations would need tighter contracts.
- **God classes / coupling:** `GraphCanvas` is UI-heavy and stateful; `GraphVizFX` is a large orchestrator. Model, controller, and view coupling is high because the model exposes internals and the view mutates them directly.
- **Standards / hygiene:** Naming matches Java conventions. Magic numbers abound (canvas padding, radii, colors, force-directed constants). Minimal Javadoc on methods. No nullability annotations.

6. Robustness & Security

- **JSONImporter:**
  - Regex-based; clears graph before validation; admits duplicates and malformed numeric fields; no size limits; errors can leave users with emptied graphs.
  - No UTF-8 enforcement or streaming—large files could be memory-heavy.
- **OSMImporter:**
  - Regex on XML-like data; silently skips invalid coordinates; no schema validation; accepts duplicate node IDs; lacks safeguards against huge files.
- **GraphModel encapsulation:**

    - `getNodes()` / `getEdges()` return mutable lists; `hasEdge` uses object identity and direction only, so undirected duplicates slip in; no invariants on unique IDs or self-loops.
- **Error handling:** Broad `catch (Exception e)` in `GraphVizFX` hides root causes; IO errors lack logging.

7. JavaFX Specifics

- **UI thread safety:** All UI interactions occur on FX thread (Application `start`). Long-running layout/algorithms execute synchronously and can freeze UI; no background tasks are used.
- **Rendering efficiency:** `GraphCanvas.draw()` redraws entire scene every call and recomputes grid/background each time. No dirty-region tracking or cached backgrounds. Edge hit-testing recomputes distances for all edges on each click.
- **Input UX:** Right-click deletion has no confirmation/undo; silent failures on invalid edge creation. Weight editing swallows `NumberFormatException` silently.

8. Testing

- **Coverage:** JUnit 5 tests cover BFS/DFS/Dijkstra behaviors, layout bounds, model edge/clear, JSON round-trip, and empty-file failure. Missing coverage for A*, Prim, Kruskal, OSM import, directed graphs, negative weights, disconnected MSTs, and concurrency/UI freezing.
- **Quality:** Tests are readable and deterministic using `TestGraphFactory`. Edge cases for null start nodes are checked; disconnected nodes in Dijkstra are verified. MST and A* paths are untested; importers lack malformed/large input tests.

9. Detailed Issue Log

Severity	File/Method	Description	Suggested Fix
High	`io/JSONImporter.importGraph`	Regex parsing, clears graph up-front, no uniqueness/schema validation; malformed JSON can wipe state or admit duplicates.	Use Jackson/Gson, validate into a temp model, enforce unique IDs, and swap on success. Add max size guard.
High	`model/GraphModel`	Mutable list exposure; `hasEdge` identity-only; duplicates and dangling edges possible; no self-loop or direction checks.	Encapsulate add/remove, return unmodifiable views, make `hasEdge` symmetric for undirected graphs, and enforce IDs/loops centrally.
Medium	`algorithms/*` neighbor scans	BFS/DFS/Dijkstra/A*/Prim/Kruskal scan all edges per step (O(VE)); `VisualState` maps mutate across steps.	Precompute adjacency maps and emit immutable snapshots per step.

Medium	`PrimAlgorithm` / `KruskalAlgorithm`	No validation for directed/disconnected graphs; may emit partial MSTs.	Guard directed graphs, stop after `
Low	`app/GraphVizFX` error handling	Broad `catch (Exception e)` hides causes; no logging.	Catch specific exceptions, log stack traces, and surface actionable messages.
Low	`io/OSMImporter`	Regex on XML; skips invalid coords silently; no duplicate detection or size limits.	Use XML/OSM parser, validate refs, enforce limits, and add tests.
Low	`view/GraphCanvas` UX	Destructive actions lack confirmation/undo; silent ignores on invalid edges; duplicate IDs allowed.	Validate in model/controller, provide feedback, and add undo/confirm flows.

10. Actionable Recommendations (with snippets)

- **Encapsulate `GraphModel` and fix edge checks:**

```
public final class GraphModel {
        private final Map<String, GNode> nodes = new HashMap<>();
        private final Set<GEdge> edges = new HashSet<>();

        public Collection<GNode> getNodes() { return
Collections.unmodifiableCollection(nodes.values()); }
        public Collection<GEdge> getEdges() { return
Collections.unmodifiableCollection(edges); }

        public void addNode(GNode n) {
                if (nodes.containsKey(n.getId())) throw new
IllegalArgumentException("Duplicate node id " + n.getId());
                nodes.put(n.getId(), n);
        }

        public void addEdge(GNode u, GNode v, int w, boolean directed) {
                if (u == v) throw new IllegalArgumentException("Self-loops not
allowed");
                GEdge e = new GEdge(u, v, w);
                if (!directed && edges.stream().anyMatch(x -> x.connects(u, v))) {
                        throw new IllegalArgumentException("Duplicate undirected
edge");
                }
                edges.add(e);
```

```
        }
}
```

- **Adjacency map for algorithms (reduces to O(V+E)):**

```
private Map<GNode, List<GEdge>> buildAdj(GraphModel g) {
        Map<GNode, List<GEdge>> adj = new HashMap<>();
        for (GEdge e : g.getEdges()) {
                adj.computeIfAbsent(e.getSource(), k -> new ArrayList<>()).add(e);
                if (!g.isDirected()) adj.computeIfAbsent(e.getTarget(), k -> new
ArrayList<>()).add(e);
        }
        return adj;
}

for (GEdge edge : adj.getOrDefault(current, List.of())) {
        GNode neighbor = edge.other(current);
        // relax / visit
}
```

- **Immutable VisualState snapshots to avoid cross-step mutation:**

```
public VisualState snapshot() {
        return new VisualState(
                Map.copyOf(nodeColors),
                Map.copyOf(edgeColors),
                Map.copyOf(distances),
                logLine
        );
}
```

- **Safe JSON import using Jackson:**

```
record EdgeDto(String from, String to, int weight) {}
record GraphDto(boolean directed, boolean weighted, List<GNodeDto> nodes,
List<EdgeDto> edges) {}

public static void importGraph(GraphModel graph, File file) throws IOException {
        GraphDto dto = new ObjectMapper().readValue(file, GraphDto.class);
        GraphModel tmp = new GraphModel();
        tmp.setDirected(dto.directed());
        tmp.setWeighted(dto.weighted());
        for (GNodeDto n : dto.nodes()) tmp.addNode(new GNode(n.id(), n.x(), n.y()));
        for (EdgeDto e : dto.edges()) tmp.addEdge(tmp.getNode(e.from()),
tmp.getNode(e.to()), e.weight(), tmp.isDirected());
        graph.replaceWith(tmp); // atomic swap
}
```

- **Background execution for UI responsiveness:** wrap algorithm/layout runs in `Task<List<VisualState>>` and update UI via `Platform.runLater` to avoid freezing the FX thread.

11. Grading

Estimated score: **86/100** — solid structure and partial coverage, but needs model encapsulation, parser hardening, performance improvements, and MST validation to reach production robustness.