Dungeon Crawler RPG — Design, TODO breakdown, and Java/JavaFX scaffold

Project summary

Single-player, turn-based dungeon crawler in a fantasy setting with a minimal UI. Player navigates an N×M grid dungeon, fights monsters, collects items, gains XP, and levels up. UI will be window-in-window with a visual tile map, click-to-move/ability, and a small HUD.

Goals / TODO (derived from your list)

- Implement game map (N×M matrix)
- Character and Monster classes
- Basic commands: attack, move, use (items/abilities)
- Visual interface: JavaFX (primary) and notes for Swift front end
- Click-to-move on tiles
- Click-to-use ability buttons
- Window-in-window layout: map view + HUD + modal windows (inventory, ability tree)
- Turn-based combat system
- XP-based level up system (linear scale)
- Argumentum-based ability tree stored as a dependency graph

High-level architecture

- `model` — game logic: `Map`, `Tile`, `Entity`, `Character`, `Monster`, `Item`, `Ability`, `DependencyGraph`
- `engine` — turn & rules engine: `GameEngine`, `CombatSystem`, `AISystem`
- `ui` — JavaFX UI: `MainWindow`, `MapView`, `HudView`, `ModalWindow` (inventory/ability)
- `util` — helpers: `RandomUtil`, `Vector2`, `SaveLoad`

Communication: UI sends input events to `GameEngine`. `GameEngine` updates model, notifies UI to redraw.

Data structures & key decisions

Map

- Represent as `Tile[][] tiles` with width N and height M.
- Each `Tile` holds: `terrain`, optional `Entity` reference (character or monster or item stack), `isWalkable`, `isVisible` (for fog of war).
- Pathfinding: A* for click-to-move (grid with 8-direction optional).

Entities

- Common base class `Entity` with: `id`, `name`, `position (x,y)`, `hp`, `maxHp`, `atk`, `def`, `speed` (turn order), methods `takeDamage(int)`, `isDead()`.
- `Character extends Entity` with inventory, abilities, xp, level, stat growth formula, equipped items.
- `Monster extends Entity` with AI behavior, xpReward, loot table.

Items

- `Item` base with `id`, `name`, `stackable`, `onUse(Character)`.
- `Equipment extends Item` modifies character stats when equipped.

Abilities and Argumentum-based Tree

- `Ability` has `id`, `name`, `cost` (mana/action points), `cooldown`, `effect` (functional interface), `requiredArgs`.
- Ability tree stored as a directed acyclic graph (DAG) — `DependencyGraph<Ability>` where edges point from prerequisite to dependent ability.
- Unlocking abilities requires spending ability points and meeting dependency prerequisites.

XP & Leveling

- Use basic linear scale: XP required for level L: `xpFor(L) = baseXP + (L-1)*xpPerLevel`.
- On level up: grant stat points and ability points.

Turn system

- Turn-based by initiative (speed) or simple round where player acts then each monster acts (simpler to start).
- Suggested: player-turn then monsters-turn (easier); later evolve to priority queue by `speed`.

Algorithms

Movement

- Click tile -> compute A* path to tile, check walkability and visibility; queue steps.
- Each move consumes 1 action point. If moving into a tile with a monster, movement converts to melee attack.

Combat (basic)

- Attack calculation: `damage = max(1, attacker.atk - defender.def)` or use random variance: `damage = floor((atk * rand(0.85..1.15)) - def)` then clamp ≥ 1.
- Criticals/evade: later features.
- Turn flow example (simple):
- Player chooses action (move, attack, use item, use ability).
- Resolve action fully.
- All monsters take their actions (AI) in arbitrary order.
- Apply end-of-turn effects (dot, regen), check XP and loot.

Ability dependency graph

- Stored as adjacency list mapping `Ability -> Set<Ability>` of children.
- To check unlock eligibility: ensure all parents are unlocked.
- To display tree: perform topological layout (simple vertical/horizontal layering by depth).

XP formula (example)

```
baseXP = 100
xpPerLevel = 50
xpFor(level) = baseXP + (level-1) * xpPerLevel
```

Example: level 1 -> 100 XP to reach level 2; level 2 -> 150 more (total 250) to reach level 3, etc.

Minimal Java class skeletons (start)

model/Vector2.java

```java
package model;
public record Vector2(int x, int y) {}
```

model/Entity.java

```java
package model;
public abstract class Entity {
    protected final String id;
    protected String name;
    protected Vector2 pos;
    protected int hp, maxHp;
    protected int atk, def, speed;

    public Entity(String id, String name, Vector2 pos, int maxHp, int atk,
int def, int speed) {
        this.id = id; this.name = name; this.pos = pos; this.maxHp = maxHp;
this.hp = maxHp;
        this.atk = atk; this.def = def; this.speed = speed;
    }

    public void takeDamage(int d) { hp = Math.max(0, hp - d); }
    public boolean isDead() { return hp <= 0; }

    public Vector2 getPos(){ return pos; }
    public void setPos(Vector2 p){ pos = p; }
}
```

model/Character.java

```
package model;
import java.util.*;
public class Character extends Entity {
    private int xp, level;
    private List<Item> inventory = new ArrayList<>();
    private Set<String> unlockedAbilities = new HashSet<>();

    public Character(String id, String name, Vector2 pos){
        super(id, name, pos, 30, 5, 1, 5);
        this.xp = 0; this.level = 1;
    }

    public void addXp(int amount){ xp += amount; checkLevelUp(); }
    private void checkLevelUp(){
        while(xp >= GameMath.xpFor(level)) { xp -= GameMath.xpFor(level);
level++; onLevelUp(); }
    }
    private void onLevelUp(){ maxHp += 5; atk += 1; def += 1; hp = maxHp; }
}
```

model/Monster.java

```
package model;
public class Monster extends Entity {
    private int xpReward;
    public Monster(String id, String name, Vector2 pos, int maxHp, int atk,
int def, int speed, int xpReward){
        super(id, name, pos, maxHp, atk, def, speed);
        this.xpReward = xpReward;
    }
    public int getXpReward(){ return xpReward; }
}
```

engine/GameEngine.java (skeleton)

```
package engine;
import model.*;
public class GameEngine {
    private final GameMap map;
    private final Character player;

    public GameEngine(GameMap map, Character player){ this.map = map;
this.player = player; }

    public void playerMoveTo(int x, int y){ /* pathfinding + move resolution
*/ }
```

```
    public void playerAttack(Monster m){ int dmg = Math.max(1, player.atk -
m.def); m.takeDamage(dmg); if(m.isDead()){ player.addXp(m.getXpReward());
map.removeEntity(m); }}

    public void endPlayerTurn(){ /* AI moves, monster attacks, etc. */ }
}
```

model/GameMap.java

```
package model;
import java.util.*;
public class GameMap {
    private final int width, height;
    private final Tile[][] tiles;

    public GameMap(int w, int h){ width = w; height = h; tiles = new Tile[h]
[w]; /* init */ }

    public boolean isWalkable(int x, int y){ return inBounds(x,y) && tiles[y]
[x].isWalkable(); }
    public boolean inBounds(int x,int y){ return x>=0 && y>=0 && x<width &&
y<height; }
    public void removeEntity(Entity e){ /* find & remove */ }
}
```

util/GameMath.java

```
package util;
public class GameMath {
    public static final int BASE_XP = 100;
    public static final int XP_PER_LEVEL = 50;
    public static int xpFor(int level){ return BASE_XP +
(level-1)*XP_PER_LEVEL; }
}
```

JavaFX UI notes

- Use `Canvas` or `GridPane` for tile rendering. Canvas + manual draw loop gives performance control.
- Input: mouse clicks -> map coordinates -> call `GameEngine.playerMoveTo(x,y)` or show contextual menu if clicking on an enemy tile.
- Ability buttons in a toolbar; modal windows for inventory and ability tree.
- Use `AnimationTimer` to drive simple UI updates and transitions; the game logic can be event-driven (no continuous simulation needed).

Swift notes (for iOS/macOS)

- Similar MVVM architecture with SwiftUI or AppKit. Use `Grid` / `Canvas` for map view. Translate Java model classes into Swift structs/classes.

Next steps (practical incremental plan)

1. Implement `GameMap` and `Tile` classes with a small test map and spawn player and a monster.
2. Implement `Entity`, `Character`, `Monster` and a simple `GameEngine` with player move and attack.
3. Add simple JavaFX window rendering the map and allowing click-to-move (no pathfinding at first — direct neighbor moves).
4. Add A* pathfinder and integrate.
5. Implement turn sequence and XP awarding.
6. Implement ability graph and UI modal to view/unlock abilities.

Files to include in initial repo

- src/model/*.java
- src/engine/*.java
- src/ui/*.java
- resources/tiles/*.png
- docs/ability_graph.md
- build.gradle or pom.xml

If you'd like, I can: - generate working Java files for the skeleton above (ready-to-run minimal game loop + JavaFX) OR - provide a full A* implementation, pathfinding examples, and an ability-graph implementation, OR - make level/monster generation rules and loot tables.

Tell me which of these you'd like next and I'll expand with code you can run locally.