Code Hierarchy

Code was grouped into packages, according to what they were part of. At the root level, code was split into seven 'main' packages: *Delegates* (Functional Interfaces), *Events* (Event Management), *Ext* (miscellaneous extension files), *Gameplay*, *Input*, *Rendering*, *UI*. Enemies were in `WizardTD.Gameplay.Enemies`, tiles in `WizardTD.Gameplay.Tiles`, UI elements in `WizardTD.UI.Elements`, etc.

In most cases, the code was structured to have mostly small 'data' classes, with larger (static) 'manager' classes overseeing those smaller classes. For example, all the game data was stored in `class GameData`, which would be loaded by `GameLoader`, and controlled by `GameManager`.

This led to quite a lot of *separation of concerns*, where each data-class/object knew as little information as possible, with the manager classes handling interactions between different sections of the code (via method parameters). E.g. the main `App` class would simply parse input events, and then pass them onto the `UiManager`, which would

When placing a tower, the `GameManager` would have no knowledge of the current state of the UI; instead the values of which upgrades to place would be passed as parameters to `public void placeOrUpgradeTower(game, tile, upgradeRange,...)`

Object-Oriented Design

Inheritance

Inheritance was used for enemies (`Enemy`), tiles (`Tile`), projectiles (`Projectile`), for the UI (`UiElement`), etc. Any behaviour/information that could be shared was placed into the base class, with the sub-classes providing the explicit implementations.

For example, `Enemy` contained the common fields for *health*, *speed*, *path following*, etc., with sub-classes `GremlinEnemy`, `BeetleEnemy`, `WormEnemy` overriding type-specific behaviour (in this case rendering).

Inheritance was used in places where not strictly necessary (such as for the *Mana Pool* spell), for future compatibility and good OOP design; Although there was only one spell implemented, it is possible to add more very easily, since the code is modular and flexibly designed.

Interfaces

Interface were also used, such as for objects that could be rendered (`Renderable`):

```java
public interface Renderable {
        RenderOrder getRenderOrder();
        void render(final App app, GameData gameData, UiState uiState);
}
```

This allowed for aggregation of objects from multiple sources that needed to be rendered extremely simple:

```
Streams.concat(game.enemies.stream(), game.projectiles.stream(), game.board.stream(),
ui.uiElements.stream())
```

```
        [...] // etc
```

The same applied to objects that could be ticked (`Tickable`).

Performance

In hot paths like the render and tick loop, some optimisations were made to be more performant and efficient. Collection pooling was used for both, to avoid large memory allocations every frame.

Extension

Debugging overlays

Multiple overlays were added, such as a line overlay for showing monster paths, an overlay for which tile is being hovered, and an `F3` menu for showing statistics (such as framerate and performance timings).

Multiple wizard houses

Multiple wizard houses could be placed on the map. This was achieved by storing mana information in the `GameData` class, and when pathfinding, iterating over all possible wizard houses as end points.

Fancy Fireballs

Fireballs would target the first 10 (configurable) enemies randomly, so that they wouldn't all target the same enemy and thus be wasteful. Also, if they W overkilled an enemy, they would continue to target more enemies until their damage was used up.

Word Count

496 words (excluding code blocks, word count, and frontmatter)