PROJECT REPORT: Lego Tiler (Java)

1. ARCHITECTURE OVERVIEW

The application follows a modular architecture designed to minimize coupling between the logic (Java), the data (SQL and Factory API), and the heavy calculation engine (C).

1.1 Core Components

- **App.java (Controller):** Acts as a Facade. It initializes the subsystems via a config file and executes the pipeline sequentially. It handles the "gluing" of the different modules.
- **FactoryClient (API Interface):** A dedicated client wrapping `java.net.HttpURLConnection`. It simplifies the use of the Factory API by adapting its return statements to the use cases.
- **InventoryManager (DB Interface):** Manages the connection to the MariaDB instance (currently still local). It implements a custom View Export logic to generate the `catalog.txt` file required by the C engine.
- **OrderManager (Restock logic):** Handles the business logic of parsing C-generated order requests, requesting quotes, and checking the factory for delivery completion.
- **Downscaler (Image compressor):** An interface that unifies image resizing methods. This allows us to implement multiple algorithms (`NearestNeighbour`, `BilinearInterpolator`, `BicubicInterpolator`) and switch between them seamlessly in the main application logic without changing the controller code.

1.2 The Payment System

For the payment system, I implemented a `PaymentMethod` interface. The `PoWMethod` implements this interface using a SHA-256 solver. This design choice allows us to easily swap the payment method (ex:, to a Credit Card implementation) without modifying the main application logic.

1.3 The Bridge with the C component

The connection to the C program is handled via Java's `ProcessBuilder`. We treat the C engine as a "Black Box" that consumes files and produces files, it outputs the result of two tiling algorithms as well as their respective order requests if there are any legos

bricks missing.

2. IMPLEMENTATION CHOICES

2.1 Custom Image Downscaling

In accordance to the requirements for this component of the project, I implemented 3 rescaling methods

- **Nearest Neighbor:** Fast but blocky.
- **Bilinear:** Smoother, averages 2x2 pixels.
- **Bicubic:** High quality, uses a 4x4 pixel neighborhood with polynomial interpolation.

This allows precise control over the input matrix given to the Lego tiler.

2.3 Gson for JSON

I used the Gson library because it is very versatile and allows for easier conversions between records and JsonObjects.

3. DIFFICULTIES ENCOUNTERED

3.1 Windows File Paths

The integration between Java (which prefers forward slashes) and the C executable (which expects Windows paths) caused `File Not Found` errors.

- **Solution:** I implemented a strict relative path system and ensured the `ProcessBuilder`'s working directory is set to the project root.

3.2 Factory API Null Handling

Despite what was said regarding the Delivery status, the delivery date is never returned by the Factory because it actually doesn't exist. The only way to know if an order is completed is by checking whether or not there are still pending bricks for a given order.

- **Solution:** I implemented verification in `OrderManager.java` using `JsonObject.has()` checks before accessing certain fields to prevent `NullPointerException`.

3.3 Database Initialization Latency

One major bottleneck is the initial population of the database. Since the catalog contains every permutation of brick shapes and colors (10 920 unique bricks), inserting them one by one into the local MariaDB instance can take significant time (several minutes) on the first run.

- **Trade-off:** Taking a bit of time for the first setup is necessary for the good functioning of the project, once the catalog is full, we can add bricks into the inventory table, which is the next longest thing to do.

3.4 Reactive vs. Proactive Ordering

Implementing a truly proactive ordering system (predicting future demand) proved complex within the timeframe.

- **Limitation:** The current system operates on a "Just-In-Time" (Reactive) model. It only triggers an order when the C algorithm explicitly flags bricks as missing in the generated invoice. This ensures correct stock levels but introduces a waiting period for the user during the print job.

4. LIMITATIONS & KNOWN BUGS

4.1 Local DB Dependency

The application currently requires a specific SQL dump to start. It lacks a "First Run" mode that creates the database schema from scratch if it doesn't exist.

Testing the code also demands some setup beforehand.

A more complete implementation that is accessible online will be implemented in the future (DB and frontend hosted on a personal server).

4.2 Single-Threaded Processing

The Bicubic interpolation and the C-Bridge execution happen on the main thread. For very large images (>4K resolution), this freezes the UI/Console until completion.

However, considering that a 1x1 lego brick is 8mm x 8mm, the max image size will probably be capped to around 256 x 256px (> 2m x 2m is unrealistic).

4.3 Hardcoded Executable Path

The code expects `C_tiler.exe` to be present in the root directory. It does not dynamically detect the OS to switch between `.exe` (Windows) and binary (Linux) formats.

5. FUTURE IMPROVEMENTS

5.1 FFM API Integration (Project Panama)

Currently, the communication between Java and C relies on file I/O and `ProcessBuilder`. While robust, this is slow due to disk latency. A state-of-the-art improvement would be to use the Foreign Function & Memory (FFM) API (introduced in Java 22). This would allow Java to allocate native memory arenas and pass pointers directly to the C engine, eliminating the need for intermediate text files (`hexmatrix.txt`) and significantly improving performance for high-frequency calls.

5.2 Proactive Stock Management

Currently, as said earlier, the only way to fill up our stock is by answering the order request sent by the C program, a better way to do this could be to check the current stock on a regular basis, and if we find that we don’t have enough of a specific brick, to order it.

Many possible solutions exist :

- analysing user patterns (most often ordered bricks) to buy more of certain types.
- Setting up an automatic Miner that constantly mines money to buy as much bricks as possible.
- Base ourselves on external factors, like the price of the bricks, to prioritize the most cost efficient ones.