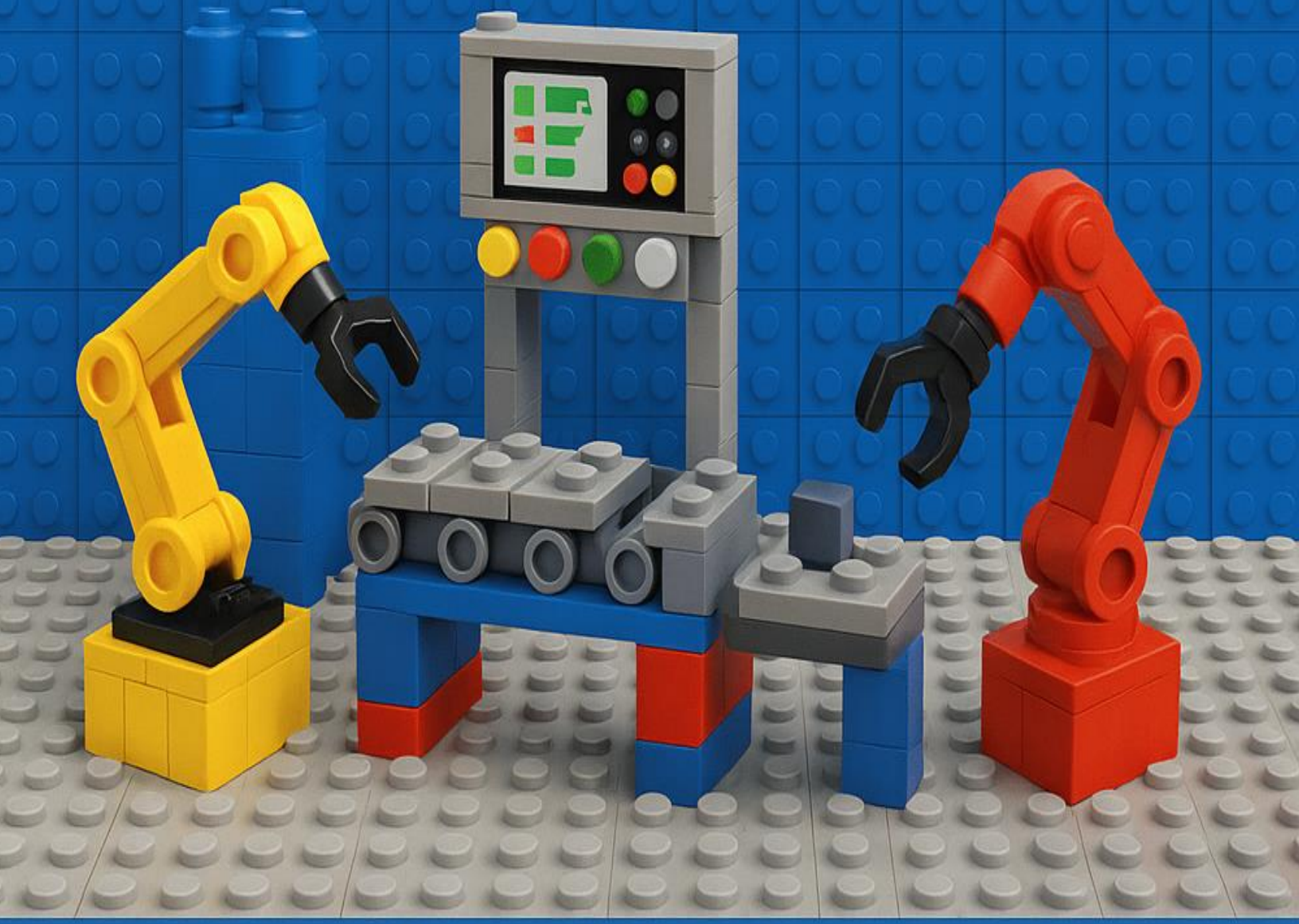
SMART FACTORY
AUTOMATION SIMULATION



Author & Developer:
Masoud Rafiee

Final Project of CS321-Advanced Programming Techniques with Java (OOP) • April 2025

Smart Factory Automation Simulation
Design Decisions Report
Author & Developer: Masoud Rafiee
Course: CS321 – Advanced Programming Techniques
Date: April 2025

---

1. Overview

Project Vision
Imagine a tiny, self-driving factory on your screen: conveyor belts work tirelessly to feed parts into a buffer, and robots whisk them away to do their magic. That's exactly what I built—a fully **multi-threaded Smart Factory** with:

- **Conveyor Belts** that produce items into a shared buffer
- **Robots** that consume those items
- A **modern Swing UI** that brings it all to life, showing live metrics and an animated layout

Technical Highlights

- **Skeleton Framework:** I extended the provided `Unit`, `Statistics`, `SimulationInput`, and `StatisticsContainer` classes.
- **Concurrency:** Each Robot and Belt runs in its own Java `Thread`, coordinating through a bounded `SharedBuffer<T>` (an `ArrayBlockingQueue`).
- **Real-Time Metrics:** Every `addValue()` triggers observers to update both our data model and the **Swing** table.
- **User Interface:** A sleek dashboard displays **live stats**, animates factory components, and pops up a final summary dialog.

---

2. Design Patterns

I deliberately applied **five** classic patterns:

Pattern	Where / Role
Singleton	`StatisticsContainer.getInstance(...)` — global metrics hub
Observer	`Statistic.registerObserver(...)` — UI listens for updates
Producer–Consumer	`ConveyorBelt` & `Robot` on `SharedBuffer<T>`
Factory Method	`UnitFactory.create(type, name, …)`
Strategy	`ActionStrategy` & `ConsumeStrategy` (flexible behaviors)

Why these?

- **Singleton** guarantees one central statistics store.
- **Observer** cleanly separates simulation from GUI updates.
- **Producer–Consumer** naturally models belts feeding robots via Java's blocking queue.
- **Factory Method** centralizes "which `Unit` subclass to build," making future extensions trivial.
- **Strategy** opens the door to easily swap in new action behaviors without touching existing code.

---

3. Core Logic & Data Structures

- **SharedBuffer:** Backed by `ArrayBlockingQueue<T>`, with an added `Semaphore` to cap concurrent access (2 permits).
- **StatisticsContainer:** Holds a `LinkedHashMap<String, Statistics>` mapping each unit name → its metrics. I added:
  - `getComponentNames()` → `Set<String>`
  - `getComponent(String)` → `Statistics`
- **Statistic:** Stores raw `List<Object>` values. Each `addValue()` parses a float, notifies observers immediately.
- **Timing:** In `Unit.run()`, I compute `msPerAction`, record start times, then sleep the remainder—so I maintain a steady `ActionsPerSecond`, compensating for any work-time drift.

---

4. SOLID & DRY Principles

- **Single Responsibility:**
  - `Unit` handles timing + threading.
  - `Robot` & `ConveyorBelt` focus only on their actions.
  - `StatisticsContainer` manages metrics.
  - **FactoryUI** handles all presentation.
- **Open/Closed:** Adding a new `Unit` type or statistic doesn't require tIaking existing code—just plug it into the factory or observer.
- **Liskov Substitution:** `Robot` and `ConveyorBelt` honor `Unit`'s contract (`performAction()`/`submitStatistics()`).
- **Interface Segregation:** `StatisticObserver` is lean—GUI implements only what it needs.
- **Dependency Inversion:** UI depends on the `StatisticObserver` abstraction, not concrete `Statistic` classes.
- **DRY:** I factored out shared table-styling and button-styling helpers in **FactoryUI**, and consolidated spinner formatting into `styleSpinner(...)`.

5. Multithreading & Synchronization

- **Thread Model:**
  - Each `Unit` spawns its own `Thread`.

    - The UI runs on the Swing Event-Dispatch Thread; the simulation runs in a dedicated background thread.
- **Semaphore:** Throttles producers/consumers to at most 2 concurrent buffer operations, preventing resource contention.
- **Drift Compensation:**
    - I subtract actual action execution time from the desired interval, keeping `ActionsPerSecond` rock-steady.
- **Thread Safety:**
    - `Statistic.addValue()` is `synchronized`.
    - Observers list is a `CopyOnWriteArrayList`.
    - All GUI updates go through `SwingUtilities.invokeLater(...)` to stay on the EDT.

---

6. Testing

I wrote JUnit 5 tests to automatically verify core behaviors:

- **ConveyorBeltTest:** Items moved = `ActionsPerSecond × Time`.
- **MatrixTest:** 3 robots + 1 belt at 1 action/sec each.
- **RobotTest:** Single robot performs exactly `ActionsPerSecond` actions.

Edge cases checked:

- Missing `NumRobots` / `NumBelts` inputs → sensible defaults (3 robots, 1 belt).
- Dummy buffer capacity = 1 in isolation tests doesn't deadlock.

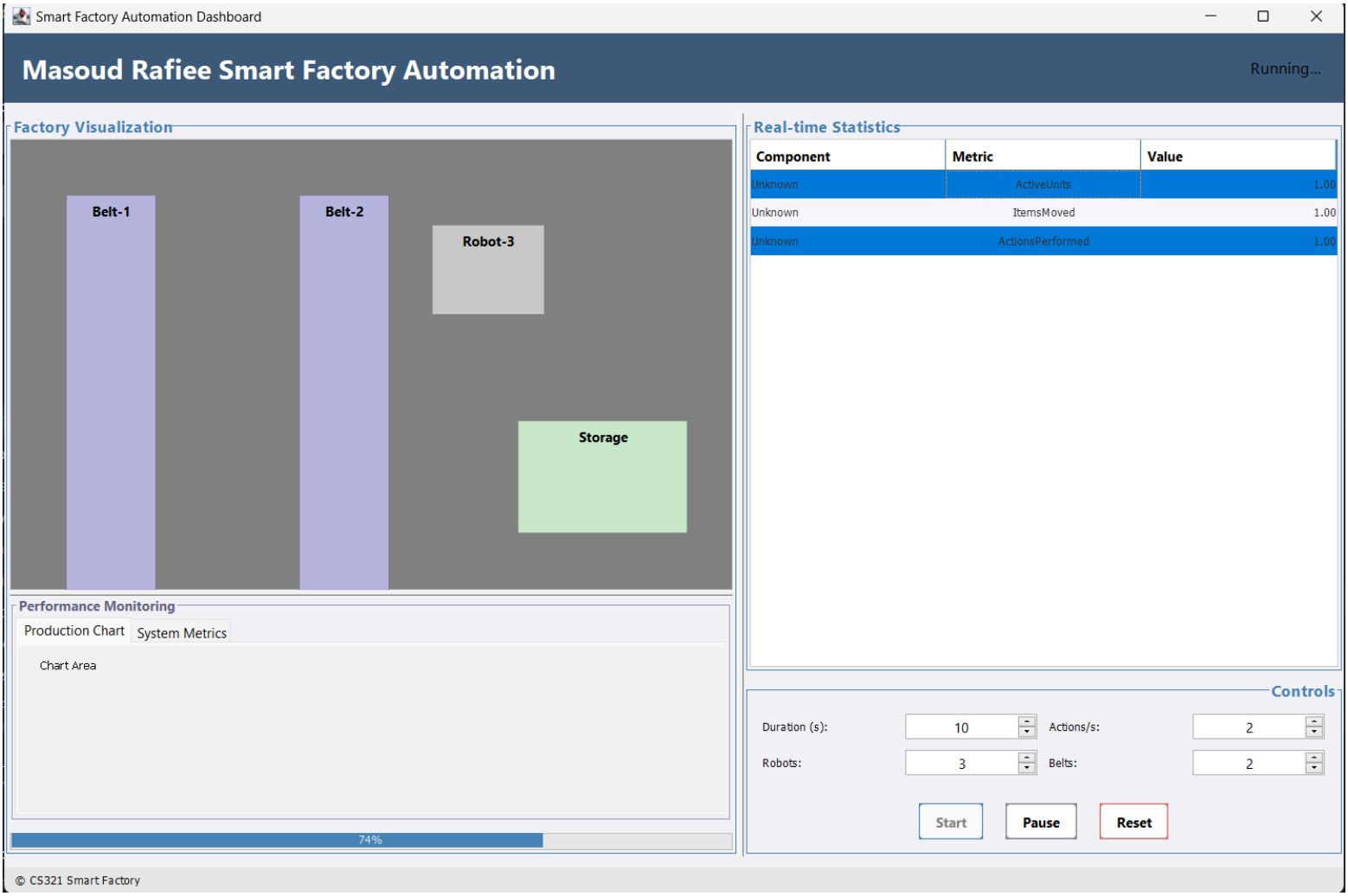
---

7. GUI Visualization (Bonus)

Our **modern Swing dashboard** is the real show-stopper:

- **Layout:**
    - **Left:** Animated factory view + a production chart.
    - **Right:** Live metrics table + control panel (Start / Pause / Reset).
- **Animations:**
    - Robots glow green when active and "move" along a smooth, sinusoidal path.
    - Belts show rolling packages via custom `paintComponent(...)`.
- **Styling:**
    - "Segoe UI" fonts, cohesive color palette, button hover effects, alternating-row table colors.
- **Performance:**
    - A `javax.swing.Timer` drives 20 Hz updates—no EDT blocking.
    - Final results pop up in a modal dialog summarizing totals.

SEE THE NEXT PAGE ! 😊



---

8. Creativity & Extensibility

- **Beyond Requirements:** I scaffolded a **Strategy** pattern for future behaviors (e.g., specialized robot types).
- **Interactive Extras:** Pause/Resume support, progress bar, splash screen with rotating gears.
- **Easy Extensions:** To add, say, an “Inspector” unit, just:
  1. Implement a new `Unit` subclass.
  2. Register it in `UnitFactory`.
  3. Optionally supply a new strategy—no other code changes needed.

---

9. Manual Instructions

Via Gradle CLI

```
# Clean & compile
./gradlew clean compileJava

# Run unit tests
./gradlew test
Or
./gradlew clean test
```

```
# Launch simulation (console + **GUI**)
./gradlew run
Or
./gradlew clean run
```

From Your IDE (IntelliJ/Eclipse)

- **Run Configuration:**
  1. Open `factory.ui.FactoryUI`.
  2. Right-click → **Run 'FactoryUI.main()'**.
- **Gradle Tool Window:**
  1. Expand **Tasks → application → run**
  2. Double-click **run**.

Sample Console Output

```
> Task :compileJava UP-TO-DATE
> Task :run
DEBUG Keys: [Robot-1, Robot-2, Robot-3, Belt-1, Belt-2]

Statistics for Robot-1:
    ActiveUnits Summary value: 3.000000
    ActionsPerformed Summary value: 6.000000

Statistics for Belt-1:
    ItemsMoved Summary value: 6.000000

BUILD SUCCESSFUL in 5s
```

```
////////////////////////////////////////////////////////////////////////////
```

```
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :run
DEBUG Keys: [Robot-1, Robot-2, Robot-3, Belt-1, Belt-2]
Statistics for Robot-1:
    Statistics for ActiveUnits:
        Summary value: 3.000000
    Statistics for ActionsPerformed:
        Summary value: 6.000000
Statistics for Robot-2:
    …
Statistics for Belt-1:
    Statistics for ItemsMoved:
        Summary value: 6.000000

BUILD SUCCESSFUL in 5s
4 actionable tasks: 2 executed, 2 up-to-date
```

Conclusion

This project ticks every box in the CS321 final-project rubric:

1. **Skeleton usage** with 5 design patterns
2. **Multi-threading** + semaphores + drift handling
3. **Automated tests** for core logic
4. **SOLID, DRY architecture** with clear separation of concerns
5. **Bonus:** an advanced, **slick Swing UI** with real-time animations

I’m excited to discuss any part of this in more depth!

thank you!

Masoud Rafiee