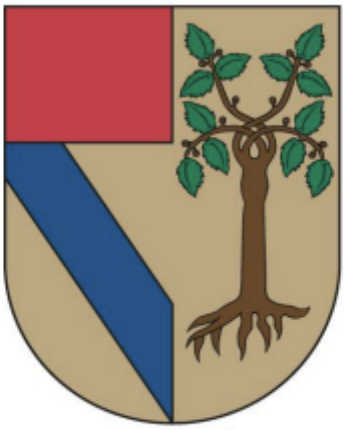

UNIVERSIDAD PANAMERICANA

Project II: Dashboard

Second Partial

Students:

Renata Calderón Mercado

Regina Ochoa Gispert

Miguel Angel Velez Olide

Proffesor:

Gabriel Castillo Cortés

Date of Delivery:

Tuesday, April 10th, 2025

Introduction

This document details the analysis and design decisions made during the development of the visualization tool for monitoring and analyzing a manufacturing plant. The system simulates a plant with six workstations, processing products through different stages and collecting statistics on performance, failures, and efficiency of the production process.

Problem Analysis

The simulation models a manufacturing plant with the following characteristics:

- 6 workstations with different failure probabilities
- Non-linear production flow (bifurcation at stations 4 and 5)
- Material replenishment when depleted
- Random failures requiring repair time
- Possibility of rejecting finished products (quality control)

The main objective is to monitor the plant's performance to identify:

- Bottlenecks
- Efficiency of each station
- Causes of downtime
- Total production and defect rate
- Resource utilization (supply device)

Data Structures Used

In the Simulation Several key data structures were used:

1. **Dictionaries (stats)**: Used to store different simulation metrics. This structure allows efficient access (O(1)) to statistics by category and facilitates their updating during the simulation.

2. **Sets (completed_stations)**: Used to track completed stations for each product. Sets provide membership verification in O(1), which is ideal for quickly checking if a station has already been processed.

3. **Lists (repair_times)**: Used to store series of values such as repair times. Lists allow for subsequent statistical calculations (averages, sums, etc.).

4. **SimPy Resources**: Implemented to handle concurrency and synchronization in the simulation, appropriately modeling limited resources such as workstations and restockers.

In Data Processing

Pandas DataFrames: Chosen to manipulate simulation data due to:

- Ease of grouping (by period, station, etc.)
- Ability to generate descriptive statistics
- Support for vectorized operations that improve performance
- Direct integration with Excel input format

JSON Structures: Used for communication between the dashboard's backend and frontend, providing a hierarchical structure that organizes data by categories:

- Company data
- Workstation data
- Time periods (daily, weekly, monthly, etc.)
- Bottleneck analysis

In the Frontend

JavaScript Objects: The DataProcessor class encapsulates all client-side data handling logic, offering methods to:

- Filter data by period and station
- Calculate derived metrics
- Transform data to adapt to the charts

Arrays and Nested Objects: Used to store processed data that feeds the charts, balancing the need for quick access with a structure that facilitates data traversal.

The choice of these structures facilitated efficient data manipulation at each stage of the process, from simulation to final visualization.

Dashboard Design

Justification of Included Charts

The dashboard was designed with the following charts:

Production Chart: A timeline line chart showing production over time. This type was chosen because:

- It allows visualization of trends over time
- It facilitates identification of cyclical or seasonal patterns
- It provides a clear view of the plant's overall performance

Station Occupation Chart: A stacked bar chart showing time distribution at each station (busy, inactive, operative). The reasons for this choice were:

- It allows direct comparison between stations
- It clearly visualizes the proportional distribution of time
- It quickly identifies problematic stations

Bottleneck Chart: A radar chart illustrating the average occupation of each station. It was selected because:

- It provides a compact view of all stations
- It facilitates visual identification of outliers
- The resulting shape offers a visual "footprint" of the plant's status

Summary Table: A tabular visualization with key metrics such as:

- Total production
- Defect rate
- Average repair time
- Average delay

This format was chosen to:

- Present precise numerical information
- Provide a quick reference of critical KPIs
- Complement more abstract visualizations

Efficiency Chart: A line chart showing the plant's overall efficiency over time, calculated as a function of defect rate and delays. It was chosen because:

- It offers a unified performance metric
- It allows tracking of efficiency evolution
- It facilitates identification of problematic periods

Charts Considered but Not Included

Sankey Chart: Considered for visualizing product flow through stations. Reasons for not including it:

- Implementation complexity
- The dynamic nature of the flow made it difficult to represent statically
- It provided redundant information with other visualizations

Scatter Plot of Failures vs. Production: Considered to correlate failures with production levels. Not included because:

- The simulated data did not show a significant correlation
- The visualization did not provide additional insights
- It required additional explanation for interpretation

Temporal Heat Map: Considered to visualize temporal patterns in failures or replenishments. Reasons for not including it:

- Temporal patterns turned out to be mainly random in the simulation
- It consumed significant space in the dashboard
- The temporal granularity of the simulation did not justify this type of visualization

Waterfall Chart for Efficiency Losses: Considered to break down causes of efficiency loss. Not included because:

- It required complex additional calculations
- The information was already represented in other charts
- The added value did not justify the additional complexity

Solution Architecture

The solution was structured into three main components:

1. **Simulation Module**: Implemented in Python with SimPy to model the behavior of the manufacturing plant.
2. **Data Processor**: Developed in Python using Pandas to transform simulation data into structures suitable for the dashboard.
3. **Web Dashboard**: Built with HTML, CSS, and JavaScript, using visualization libraries to present the processed data.

This modular architecture allows:

- Running new simulations independently of the visualization
- Modifying data processing without affecting simulation or dashboard
- Updating the user interface without altering the underlying logic

Design Conclusions

The dashboard design focused on providing a comprehensive view of manufacturing plant performance, prioritizing problem identification and bottlenecks. The chosen data structures facilitated efficient information handling at each stage of the process, and the selected visualizations offer an effective combination of general and detailed perspectives.

Design decisions were made considering:

- The nature of the data (temporal, multidimensional)
- Analysis objectives (problem identification, KPI tracking)
- Technical limitations (performance, compatibility)
- User experience (clarity, ease of interpretation)

The result is a visualization tool that allows users to quickly understand the state of the manufacturing plant and make informed decisions to improve its efficiency.

Registro de Desarrollo

Development Log

Challenges and Solutions

1. Simulation Modeling

Challenge: Representation of non-linear flow Modeling stations 4 and 5, which operate in parallel, presented complexities in determining how products should choose between these stations.

Solution: We implemented a decision logic based on the queue length of each station. When a product needs to process at one of these stations, it checks which one has fewer products waiting, choosing the less congested one. This approach simulates an optimization decision that could occur in a real environment:

```
station4_queue = len(self.plant.stations[4].resource.queue)
station5_queue = len(self.plant.stations[5].resource.queue)

if 4 not in self.completed_stations and 5 not in self.completed_stations:
    station_id = 4 if station4_queue <= station5_queue else 5
elif 4 not in self.completed_stations:
    station_id = 4
else:
    station_id = 5
```

Challenge: Resource Synchronization Managing shared resources, such as restockers, presented synchronization problems when multiple stations required replenishment simultaneously.

Solution: We used SimPy's resource system, which internally handles queues and wait times. This allowed us to correctly model resource contention:

```
def restock(self):
    with self.restockers.request() as req:
        yield req
        restock_time = max(0, random.normalvariate(2, 0.5))
        self.stats['uso_dispositivo_suministro'] += restock_time
        yield self.env.timeout(restock_time)
        self.raw_material = 25
```

2. Data Processing

Challenge: Appropriate structure for the dashboard Converting simulation data to a suitable format for visualization required deciding which structure to use to facilitate both processing and consumption from the frontend.

Solution: We opted for a hierarchical JSON format that organizes data by categories (company, stations, periods) and pre-calculates some derived metrics. This structure reduces the processing load on the client while maintaining flexibility:

```
data = {
    'company': company_data,
    'workstation': workstation_data,
    'time_periods': {
        'daily': daily_data,
        'weekly': weekly_data,
        'monthly': monthly_data,
        'quarterly': quarterly_data,
        'yearly': yearly_data
    },
    'bottlenecks': bottleneck_data
}
```

Challenge: Data aggregation by time periods The simulation generates data with daily granularity, but we needed visualizations that allowed analysis in different periods (weekly, monthly, etc.).

Solution: We implemented a parameterized aggregation function that could generate statistics for different periods using the same logic:

```python
def aggregate_by_period(df, period_col):
    agg_data = []
    period_groups = df.groupby(period_col)

    for period, group in period_groups:
        agg_data.append({
            'periodo': int(period),
            'produccion_total': int(group['Producción final'].sum()),
            'produccion_promedio': float(group['Producción final'].mean()),
            # Más cálculos...
        })

    return agg_data
```

This approach allowed us to reuse code while generating specific temporal views.

3. Frontend Development

Challenge: Dynamic chart updates Switching between different time periods or station filters required updating multiple charts simultaneously, which could cause performance issues or visual inconsistencies.

Solution: We implemented an architecture based on a central data processing class (DataProcessor) that maintains the current state of filters and provides methods to obtain transformed data. The Charts class uses these methods to update each chart when filters change:

```javascript
document.getElementById('time-period').addEventListener('change', function () {
    const period = this.value;
    dataProcessor.setTimePeriod(period);
    charts.updateCharts();
});
```

Challenge: Responsiveness on different devices The charts did not adapt correctly to different screen sizes, causing display problems on mobile devices or when resizing the window.

Solution: We implemented a dynamic resizing system that adjusts the charts when the window size changes:

4. Component Integration

Challenge: Development environment configuration The project required different dependencies and a specific directory structure to function correctly, which made initial installation and execution difficult.

Solution: We developed a script (setup.py) that verifies dependencies, creates the necessary directory structure, and provides options to run the simulation and dashboard:

Challenge: Communication between simulation and dashboard The data generated by the simulation needed to be available for the dashboard, which required a communication strategy between components.

Solution: We implemented a file-based approach, where the simulation generates an Excel file with the results, which is then processed by an intermediate script to generate the JSON consumed by the dashboard. This decoupled architecture allows the components to be executed independently and facilitates parallel development.

Unresolved Challenges

Product flow visualization We were unable to implement a dynamic visualization of product flow through the stations that would show in real-time how products move through the plant. The libraries considered (such as D3.js) required significant restructuring of the data and the development of complex custom visualizations.

Failure prediction An initial goal was to implement a predictive model that could anticipate failures based on historical patterns. However, the random nature of failures in our simulation made this approach ineffective without a more sophisticated simulation that included progressive machine degradation factors.