Autonomous fruit harvesting system using drone technology

Professor:

- Maxim Glaida

Group members:

- Lucas Catolino
- Santiago Lo Coco
- Joel Kudiyirickal

March 2024

Index

Overview **3**
Resilience patterns used **3**
- Circuit Breaker Pattern 3
- Retry Pattern 3
- Timeout Pattern 3
- Rate Limiting Pattern 3

Application Idea and Problem Statement **3**
Source Code and Implementation of Resilience Design Patterns **4**
- 1. Retry Pattern 4
- 2. Timeout Pattern 5
- 3. Rate Limiting Pattern 6
- 4. Circuit Breaker Pattern 6

Design Decisions and Insights **7**
Changes from the base code from the first delivery **8**
Class diagram **9**
GitHub repository **9**

Overview

In this project, we were asked to use complementary resilience patterns. We decided to work on the previous 'Library' project presented in the 'Implementing Design Patterns' assignment. The idea of reutilizing the previous project was to focus on the resilience patterns more than the design of the project itself. Additionally, it was a good opportunity to develop a project as it would be in real-life system development: after launching a version of the system, more features are requested, and as the system grows, it should also be improved.

Resilience patterns used

Circuit Breaker Pattern

The Circuit Breaker pattern helps handle failures gracefully by temporarily blocking requests to a service when it's deemed unavailable or experiencing a high failure rate, thereby preventing cascading failures and conserving resources.

Retry Pattern

The Retry pattern allows the system to automatically retry failed operations with the expectation that they might succeed on subsequent attempts.

Timeout Pattern

The Timeout pattern sets a maximum time for an operation to complete before it's considered unsuccessful, helping prevent long-running operations from causing delays or blocking resources indefinitely.

Rate Limiting Pattern

The Rate Limiting pattern restricts the number of requests a system can handle within a specified time frame to prevent overload and ensure fair resource allocation.

Application Idea and Problem Statement

The library system aims to provide a flexible and reliable platform for managing books and magazines within a library setting. The primary goal is to offer users a seamless experience for browsing, accessing, and managing library items while ensuring system stability and resilience against potential failures or disruptions.

The challenges addressed by the library system include:

1. Reliable Access: Users should be able to access library items without encountering system failures or downtime.
2. Performance: The system should maintain optimal performance even under varying loads and conditions.
3. Fault Tolerance: The system should gracefully handle errors and failures to prevent disruptions in service.

Source Code and Implementation of Resilience Design Patterns

The following resilience design patterns have been implemented in the library system:

1. Retry Pattern

Implemented to automatically retry failed operations, allowing for potential success on subsequent attempts. The retry logic is integrated into critical operations such as accessing external resources or performing network requests.

Source: The retry logic is embedded within methods that interact with external services or perform potentially unreliable operations.

We decided to demonstrate its functionality in methods responsible for adding library items as seen in **Figure 1**. When a library item is added, if the insertion fails, it will retry a certain number of times until it eventually throws an exception. The idea behind this is to simulate a real-life situation where a client tries to insert a book, but the library is full. The system will retry the insertion, and perhaps another client removes an item, creating a free spot where the book can now be inserted. This scenario is depicted in **Figure 2**.

```
// Retry method
private void performWithRetry(ExceptionRunnable action) throws Exception {
    int attempt = 0;
    while (attempt < retryAttempts) {
        System.out.println("Attempt: " + attempt);
        try {
            action.run();
            break;
        } catch (Exception e) {
            attempt++;
            if (attempt >= retryAttempts) {
                System.out.println("Attempts error");
                throw e;
            }
        }
    }
}
```

Figure 1: Retry pattern

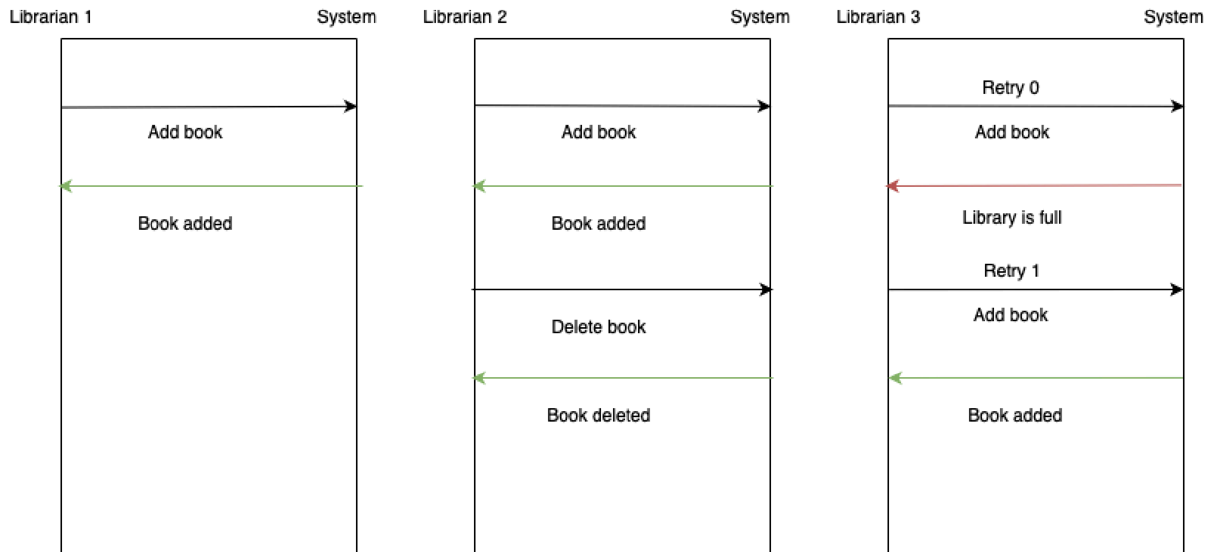

Figure 2: Retry real-life situation

2. Timeout Pattern

Applied to set a maximum time for operations to complete before considering them unsuccessful. This prevents long-running operations from causing delays or blocking resources indefinitely.

Source: Timeout mechanisms are integrated into methods that perform potentially time-consuming tasks.

We decided to demonstrate its functionality in methods responsible for removing library items as seen in **Figure 2**. When a library item is removed, a random time is calculated, and the thread will sleep for that duration. If the thread sleeps for longer than expected, a timeout exception is thrown. The idea behind this is to simulate a real-life situation where the server is slow for any reason but still capable of resolving the request, such as when using an SQL database and removing an item involves a lot of table manipulations due to constraints like "on delete cascade".

```
// Timeout method
private void performWithTimeout(Runnable action, long timeoutMillis) throws Exception {
    Thread thread = new Thread(action);
    thread.start();
    System.out.println("Thread state: " + thread.getState());
    thread.join(timeoutMillis);
    System.out.println("Thread state: " + thread.getState());
    if (thread.isAlive()) {
        thread.interrupt();
        throw new TimeoutException("Operation timed out");
    }
}
```

Figure 2: Timeout pattern

3. Rate Limiting Pattern

Implemented to restrict the number of requests the system can handle within a specified time frame. This prevents overload and ensures fair resource allocation, enhancing system stability and performance.

Source: Rate limiting logic is incorporated into methods that interact with external services or handle incoming requests.

We have integrated the rate limiter pattern using a leaky bucket mechanism as seen in **Figure 3**. This pattern is particularly useful for methods responsible for displaying library items. In a real-life scenario, these methods often need to aggregate items based on external services such as APIs. This helps maintain system stability and performance even during periods of high demand.

```java
// Display with rate limit method
public void displayLibraryItems() {
    long currentTime = System.currentTimeMillis();
    long timeElapsed = currentTime - lastAccessTime;

    tokens += (int) (timeElapsed / interval) * rateLimit;
    System.out.println("Tokens: " + tokens);
    System.out.println("Time elapsed: " + timeElapsed);
    tokens = Math.min(tokens, rateLimit);

    if (tokens > 0) {
        System.out.println("Items available in the library:");
        libraryItems.forEach(System.out::println);
        tokens--;
        lastAccessTime = currentTime;
    } else {
        System.out.println("Rate limit exceeded. Please try again later.");
    }
}
```

Figure 3: Rate Limiter pattern

4. Circuit Breaker Pattern

Employed to handle failures gracefully by temporarily blocking requests to a service experiencing high failure rates or unavailability. This helps prevent cascading failures and conserves resources during degraded states.

Source: Circuit breaker mechanisms are integrated into critical operations to monitor service availability and failure rates.

We decided to demonstrate its functionality by incorporating a random possibility where the circuit is closed as seen in **Figure 4**. When adding or removing items from the library, it

checks whether the circuit is open or not. The rationale behind this is to simulate a real-life scenario where the circuit might close due to errors or other issues.

```java
// Circuit Breaker
private boolean isLibraryOpen() {
    double rand = Math.random();
    if (rand > libraryOpenCondition) {
        System.out.println("Library open");
        return true;
    }
    System.out.println("Library closed");
    return false;
}
```

Figure 4: Circuit Breaker pattern

Design Decisions and Insights

The integration of resilience design patterns in the library system was a deliberate choice aimed at enhancing system reliability and fault tolerance. Through this exercise, several insights were gained:

1. Modular Design: Designing the system with modularity in mind allowed for easier integration of resilience patterns without tightly coupling components.
2. Failure Handling: Resilience patterns provided robust mechanisms for handling failures and mitigating potential disruptions, improving overall system stability.
3. Performance Impact: While resilience patterns add overhead in terms of complexity and processing, their benefits in terms of system reliability and fault tolerance outweigh the performance costs.
4. Configuration Flexibility: The ability to configure parameters such as retry attempts, timeouts, rate limits, and circuit breaker thresholds from external sources (e.g., configuration files or environment variables) adds flexibility to adapt to varying operational requirements.

We decided to implement the patterns ourselves instead of relying on an external library like resilience4j. We believed it was beneficial to familiarize ourselves with the functionality of the patterns and learn how to use them firsthand. When developers utilize frameworks or external libraries, they often abstract away many technical details, delegating them to a third party. While this approach is not inherently flawed and is encouraged in industry to utilize proven solutions and avoid reinventing the wheel, in an educational setting, relying on third-party libraries or frameworks can abstract away significant learning opportunities. That is why we chose to implement the patterns from scratch.

In conclusion, the integration of resilience design patterns in the library system not only improves its robustness and reliability but also provides valuable insights into building resilient software systems in real-world applications. These patterns serve as essential tools for ensuring system stability and fault tolerance in dynamic and challenging environments.

Changes from the base code from the first delivery

Several enhancements and refinements have been made to the base code of the library system to improve its functionality, maintainability, and test coverage. Here are the key changes:

1. Creation of Java Packages: The codebase has been organized into separate Java packages to enhance maintainability and code organization. Packages such as decorators, interfaces, exceptions, and items have been created to logically group related classes and components.
2. Method and Class Refactoring: Existing methods and classes have been refactored for clarity, readability, and improved functionality. Code duplication has been reduced, and more descriptive methods and variable names have been used to enhance code comprehensibility.
3. Implementation of Builders for Library Items: Builders have been implemented for creating instances of library items such as books and magazines. The builder pattern allows for the construction of complex objects step by step, providing a more flexible and readable way to create library items with various attributes.
4. Addition of More Methods: Additional methods have been added to enhance the functionality of the library system. These methods include operations for adding, removing, and updating library items, as well as retrieving information about the library's contents and capacity.
5. Expansion of Test Coverage: The test suite has been expanded to achieve over 90% test coverage, ensuring thorough validation of the system's functionality and behavior. Unit tests and pseudo-integration tests have been added to cover critical components and scenarios.

These changes contribute to the overall improvement and refinement of the library system, making it more robust, maintainable, and reliable. The adoption of best practices in software development, such as code organization, refactoring, and comprehensive testing, ensures the delivery of a high-quality and resilient application.

Class diagram

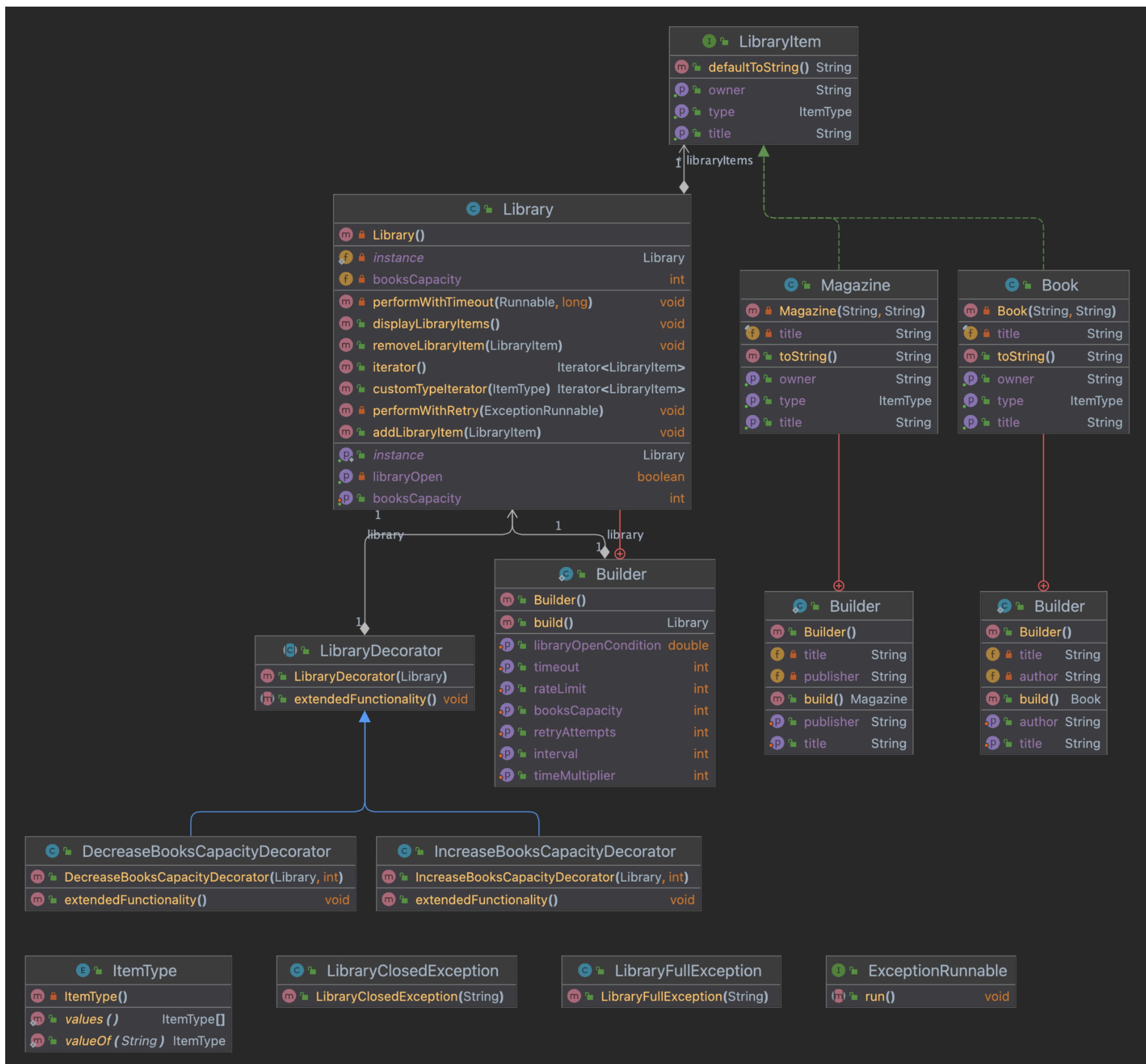

Figure 5: Class diagram

GitHub repository

https://github.com/slococo/adp-hw1