Jasmine Lisondra

Chapter 13: Concurrency

Why Concurrency?

Concurrency is a way to handle multiple tasks at the same time in software. It helps programs run better by using resources more efficiently and improving performance. Concurrency separates what needs to be done from when it gets done, making applications faster and better organized.

Concurrency Defense Principles

There are four important principles to handle concurrency:

1. **Single Responsibility Principle**: Keep concurrency-related code separate from other code. This makes it easier to manage and maintain because concurrency has its own challenges.
2. **Limit the Scope of Data**: Prevent problems by restricting access to shared data. Use synchronization to protect critical sections and avoid errors.
3. **Use Copies of Data**: Instead of sharing data between threads, use copies. This reduces the need for synchronization and improves performance.
4. **Threads Should Be as Independent as Possible**: Write code so each thread works independently with minimal shared data. This reduces synchronization and makes the system more scalable and efficient.

Following these principles helps create robust and scalable systems.

Know Your Library

When working with concurrency, use thread-safe data structures and libraries provided by your programming language. These libraries have optimized tools for managing concurrent tasks. Use executor frameworks or task queues to manage thread pools and task execution. Consider nonblocking or lock-free algorithms to improve performance and avoid concurrency bugs.

Know Your Execution Models

Understand different execution models in concurrency, like mutual exclusion, starvation, deadlock, and solutions like Producer-Consumer and Readers-Writers problems. Mastering these helps you tackle complex concurrency challenges.

Testing Threaded Code

Testing threaded code is tricky due to concurrent execution and shared data. Write thorough tests for different scenarios and configurations. Treat unexpected failures as potential threading issues. Validate non-threaded code first, then test threaded implementations. Make threaded code pluggable and tunable to test it under different conditions. Test on different platforms to find platform-specific issues.

Chapter 14: Successive Refinement

Args Implementation

When starting to write code, it’s okay to start with a rough draft. This is a basic, unpolished version that you improve over time.

Args: The Rough Draft

A rough draft is the early version of your work. It serves as a starting point for revisions and improvements. Use Test-Driven Development (TDD) to guide your changes. Start with automated tests to make sure your code works. Then, make small changes to gradually refine and improve your code. Keep your system functional at all times, ensuring that each change doesn’t break the existing behavior. Automated tests help catch any issues.