Loyola University Chicago

Department of Computer Science

Android Timer Application

COMP 313/413 – Intermediate Object-Oriented Design

Dr. Robert Yacobellis

Submitted By:

Halah, Ahmad
Alkhalil, Mahran
Kofira, Zach
Zulaik, Che Alyssa
Goldberg-Finkelstein, Jonah
Serek, Kassidy

December 6, 2025

Overview

This project transformed a stopwatch Android application into a countdown timer by adapting its state machine architecture. The timer allows users to set a countdown duration (1-99 seconds) by pressing a button, automatically starts after 3 seconds of inactivity, counts down to zero, and triggers an alarm. Around 70% of the stopwatch codebase successfully reused (MVA architecture, TimeModel, ClockModel) while implementing new timer-specific states, with time-based guards and audio feedback . The implementation demonstrates effective application of design patterns (State pattern, dependency injection, Facade pattern) and proper separation of concerns between model logic and UI presentation.

Development journey

We started with a state diagram to guide implementation, then built the timer by reusing stopwatch architecture and adding new states with time-based guards. Main challenges included implementing countdown with an increment-only interface, managing clock lifecycle, and switching to MediaPlayer for audio. The State pattern and dependency injection made the code testable and maintainable, while the model-first approach helped coordinate our team and verify completeness.

Model vs. Code

The Timer Project, started by drawing a state diagram (the "model") showing how the timer should work, then writing Java code to implement it. The following sections reflects on how the diagram and code relate to each other, whether starting with the diagram was helpful, and what we'd change about our diagram now that the code is done.

What are the similarities and differences between model and code?

The state diagram and code align very well structurally. Both have the same four states (Stopped, Incrementing, Running, Alarming) with identical transitions. The guards from the diagram translate directly to code: [clicks < 99] becomes if (currentTime < MAX_TIME), and [elapsedTime == 3] becomes if (ticksSinceStart >= TIMEOUT_TICKS). This one-to-one correspondence made implementation straightforward. However, the code contains crucial details absent from the diagram. The biggest difference is decrement implementation. The diagram simply shows clicks--, but since TimeModel only supports incrementing (inherited from stopwatch), the code decrement by resetting to zero then incrementing to n-1. The diagram also doesn't show dependency injection - how TimeModel and ClockModel are created and injected through constructors - or where audio is handled (Activity layer using MediaPlayer with getApplicationContext). Clock lifecycle management (when to start/stop, preventing multiple timers) required careful coding but isn't represented in the diagram at all.

Model-First vs. Code-First: What Worked Better?

For this project, model-first was definitely the right approach. The diagram gives a clear roadmap before writing code, avoiding getting lost in implementation details. It made team coordination easier - we could point at states and divide work clearly. Most importantly, it served as a checklist to verify we'd implemented all transitions and guards.

Model-first worked because we had clear, specific requirements that fit the state pattern naturally. For more exploratory projects with vague requirements, code-first might reveal edge cases faster. But for state-based behavior like our timer, the diagram was invaluable.

Possible future changes to Model

Based on our coding experience, we would add key improvements to our state diagram. First, annotate each state with clock status (e.g., "Incrementing [clock: ON]") to clarify when the timer is running. Second, rename "clicks" to "remainingTime" since it represents seconds, not button presses. Third, use different arrow styles - solid for user-triggered transitions (button press) and dashed for automatic ones (timeout, reaching zero). Fourth, document that audio plays in the Activity layer using MediaPlayer, not in the state machine, to clarify architectural decisions.

Conclusion

The state diagram successfully guided our implementation and helped verify completeness. While it abstracted away important details like dependency injection and clock management, this was beneficial, it let us focus on logic first, then tackle implementation challenges. For state-based projects with clear requirements, model-first is the best approach.