RETROSPECTIVA DEL PROYECTO - BAD DOPO CREAM – STIVEN ESNEIDER PARDO GUTIERREZ

1. ¿Cuál fue el tiempo total invertido en el proyecto? (Horas)

El tiempo total invertido se estima en aproximadamente 20 horas de trabajo individual. La fase de análisis y diseño tomó entre 2 y 3 horas para definir requisitos y arquitectura inicial. La implementación de la capa de presentación requirió entre 5 y 6 horas desarrollando interfaz, menús y animaciones básicas. La capa de dominio consumió entre 4 y 5 horas en entidades, lógica del juego y colisiones. La integración inicial tomó aproximadamente 2 horas. La refactorización para aplicar patrones de diseño y principios SOLID requirió entre 3 y 4 horas. La implementación de inteligencia artificial demandó alrededor de 2 horas. Las pruebas y depuración consumieron entre 1 y 2 horas, y la documentación tomó aproximadamente 1 hora.

2. ¿Cuál es el estado actual del proyecto? ¿Por qué?

El proyecto está completado y funcional al cien por ciento. Se superaron los requisitos mínimos implementando tres personajes cuando se solicitaban dos, tres tipos de enemigos en lugar de dos, y cuatro tipos de frutas superando el mínimo de dos. Se desarrolló una arquitectura limpia con separación total entre presentación y dominio mediante el patrón Facade, aplicación completa de principios SOLID, múltiples patrones de diseño como Strategy, Builder y DTO, cero violaciones de encapsulamiento y type-safety completo. Se implementaron tres niveles funcionales con tres modos de juego: un jugador, dos jugadores y machine versus machine. El código alcanzó calidad profesional con JavaDoc completo, diagramas actualizados y un sistema de inteligencia artificial funcional con estrategias diferenciadas por nivel.

3. Considerando las prácticas XP del proyecto, ¿cuál fue la más útil? ¿Por qué?

La práctica más útil fue el refactoring continuo. El proyecto pasó por dos refactorizaciones mayores que transformaron código funcional pero acoplado en una arquitectura profesional. Inicialmente la presentación accedía directamente a entidades del dominio con diecisiete violaciones de arquitectura detectadas. Tras el refactoring se implementó el

patrón Facade como único punto de acceso, se crearon DTOs mediante Snapshots para transferencia segura de datos, se extrajo comportamiento a Strategy Pattern y se separaron responsabilidades en componentes especializados. Los resultados fueron una reducción del treinta y cinco por ciento en líneas de código, cero violaciones de encapsulamiento, cien por ciento de type-safety y mejora significativa en testabilidad y mantenibilidad. Esta práctica permitió detectar deficiencias arquitectónicas tempranamente, facilitar nuevas funcionalidades sin romper código existente y reducir deuda técnica antes de que se volviera inmanejable.

4. ¿Cuál consideran fue el mayor logro? ¿Por qué?

El mayor logro fue alcanzar una arquitectura limpia con separación total de responsabilidades. Antes de la refactorización GamePanel accedía directamente a objetos del dominio violando encapsulamiento. Después, solo utiliza DTOs tipo Snapshot que proporcionan únicamente datos garantizando type-safety. Se alcanzó encapsulamiento perfecto donde la presentación nunca toca el dominio directamente, testabilidad mejorada porque cada capa puede probarse independientemente, escalabilidad para agregar funcionalidades sin modificar múltiples capas, reutilización de lógica del dominio en diferentes contextos y profesionalismo siguiendo estándares de la industria. Este logro demuestra comprensión profunda de principios de diseño más allá de código funcional, es la base que permite todas las demás funcionalidades y facilita mantenimiento futuro del proyecto.

5. ¿Cuál consideran que fue el mayor problema técnico? ¿Qué hicieron para resolverlo?

El mayor problema fue la sincronización de animaciones con lógica de juego en modo machine versus machine. La inteligencia artificial mueve entidades desde el dominio pero las animaciones suaves se manejan en presentación, causando teletransportación visual, desincronización y glitches. La solución se basó en separación clara de responsabilidades donde el dominio maneja posición lógica en grid y la presentación maneja posición visual en píxeles. Se implementó un método específico de sincronización que compara posiciones y activa interpolación cuando difieren. Se diseñó un timer dual que actualiza dominio, animaciones visuales y sincronización cada dieciseis milisegundos manteniendo

sesenta cuadros por segundo. El resultado fue animaciones completamente suaves sin saltos visuales y sincronización perfecta entre estado lógico y visual.

6. ¿Qué hicieron bien como equipo? ¿Qué se comprometen a hacer para mejorar los resultados?

Se aplicaron principios de buenas prácticas exitosamente: planificación incremental desde funcionalidad básica agregando complejidad gradualmente, documentación continua con JavaDoc desde el inicio, refactorización proactiva detectando code smells tempranamente, testing exhaustivo de cada funcionalidad y casos extremos, y cumplimiento riguroso de estándares Java y principios SOLID.

Los compromisos de mejora incluyen implementar pruebas automatizadas con JUnit buscando ochenta por ciento de cobertura, añadir integración continua mediante GitHub Actions para ejecutar tests automáticamente, mejorar gestión de configuración externalizando constantes a archivos JSON o YAML, implementar logging profesional con frameworks como SLF4J reemplazando System.out.println, agregar persistencia para guardar partidas y puntajes, y optimizar recursos mediante lazy loading y object pooling.

7. ¿Qué referencias usaron? ¿Cuál fue la más útil? Incluyan citas con estándares adecuados.

Las referencias principales fueron Gamma et al. en Design Patterns (1994) para patrones Strategy, Builder y Facade; Martin en Clean Architecture (2017) para separación de capas y DTOs; Beck en Extreme Programming Explained (2004) para refactoring continuo; Bloch en Effective Java (2018) para Builder pattern y encapsulamiento; Freeman y Freeman en Head First Design Patterns (2020) para Strategy y Template Method; documentación Oracle de Java SE Swing para timers y animaciones; y Fowler en Refactoring (2018) para técnicas de refactorización.

La referencia más útil fue Clean Architecture de Robert C. Martin. Fue fundamental para entender separación de capas aplicando "The Dependency Rule: Source code

dependencies must point only inward" haciendo que GamePanel solo conociera GameFacade. Para DTOs aplicó que "Data Transfer Objects are simple data structures that are used to transfer data across boundaries" diseñando todos los Snapshots. El diseño de Facade siguió que "The Facade pattern provides a unified interface to a set of interfaces in a subsystem" como único punto de acceso al dominio. La inversión de dependencias aplicó "High-level policy should not depend upon low-level details" haciendo GameLogic dependiente de la interfaz MovementBehavior. La cita clave que guió la refactorización fue "The best structure is the one that allows the high-level policy to remain unchanged when the details change" permitiendo que cambios en interfaz gráfica no afectaran lógica del juego y viceversa.