Gestione del Progetto: Sala Giochi Virtuale

Progetto: Sala Giochi Virtuale
Data: 09 Gennaio 2026
Team di Progetto: Team UNI
Corso: Ingegneria del Software – Università degli Studi di Bergamo

Contenuto:

- Software Life Cycle
- Configuration Management
- People Management and Team Organization

1. Software Life Cycle (Ciclo di Vita del Software)

1.1 Processo di Sviluppo

Il team ha adottato un processo di sviluppo **iterativo e incrementale**, strutturato in cicli brevi della durata di circa 2 settimane ciascuno. Questa scelta metodologica è giustificata dalla necessità di gestire la complessità del progetto attraverso rilasci progressivi di funzionalità, permettendo al contempo di tornare sulle fasi precedenti per affinamenti (iterazioni) basati sul feedback continuo.

Confronto Project Plan vs. Esecuzione Reale: **:** Rispetto a quanto previsto nel Project Plan iniziale, il processo ha subito un leggero adattamento:

- **Pianificato:** Iterazioni rigide di 2 settimane esatte.
- **Effettivo:** Le iterazioni dedicate allo sviluppo del *Gameplay* (in particolare *BattleShip*) si sono estese leggermente oltre il timebox previsto a causa della complessità della classe. Per compensare, la fase di *Refinement* è stata compressa, parallelizzando i test con la rifinitura della GUI.

Organizzazione Temporale (Sprint Log):

- **Sprint 1 (Analisi e Setup):** Definizione requisiti, Setup GitHub/Maven.
- **Sprint 2 (Core):** Database H2, Login, Classe Room.
- **Sprint 3 (Gameplay A):** Tris e DiceGuesser.

- **Sprint 4 (Gameplay B):** BattleShip e Roulette.
- **Sprint 5 (Chiusura):** Classifiche e Bug Fixing.

1.2 Gestione delle Modifiche (Change Request Process)

Le richieste di modifica (es. aggiunta di una funzionalità non prevista o modifica di un requisito) sono state gestite seguendo un flusso formalizzato per valutare l'impatto sull'architettura.

Di seguito il **Diagramma di Attività UML** che formalizza il processo di gestione di una *Change Request*:

1.3 Approccio MDA e SPL

- **Approccio MDA (Model Driven Architecture):** È stato adottato un approccio MDA utilizzando **Papyrus UML** per la modellazione delle entità e delle classi principali. Sono seguite poi integrazioni manuali per far fronte a problemi scatuturiti dalla generazione dei modelli e per poi progettare nel concreto i vari metodi di ciascuna classe.
- **Software Product Line (SPL):** Il progetto è stato strutturato come una linea di prodotti software. La classe astratta `VideoGames` funge da *Core Asset* (piattaforma comune), mentre i singoli giochi (`Tris`, `Battleship`, ecc.) rappresentano i *prodotti* derivati che condividono le caratteristiche comuni (punteggio, associazione utente) variando solo per la logica di gameplay specifica.

Iterazioni:

- **Core & Infrastructure:** Focalizzata sulla stabilità del sistema. Include lo sviluppo dello schema relazionale del database H2 e dei meccanismi di persistenza dei dati, oltre alla creazione dell'architettura di navigazione e alla mappatura dei *Layer* dell’applicazione.

- **UML Definition & Model Mapping:** I diagrammi delle classi prodotti in fase di analisi sono stati tradotti in codice Java. È stato fondamentale delineare le relazioni UML (composizione, aggregazione, ereditarietà) per garantire la coerenza tra design e implementazione.
- **Gameplay Implementation:** Sviluppo verticale delle logiche dei quattro giochi. Ogni gioco è stato implementato come modulo indipendente, con l'uso di una **classe astratta comune** che ha permesso di applicare il principio di polimorfismo, facilitando l'estendibilità e la manutenzione del codice.
- **Refinement & Finalization:** Fase di consolidamento. Oltre alla rifinitura estetica della *GUI* con *JavaFX*, è stata data priorità alla risoluzione dei bug emersi dai test *JUnit* e all'ottimizzazione delle query-SQL per le classifiche.

2. Configuration Management (Gestione della Configurazione)

2.1 Sistema di Versionamento

La gestione della configurazione è stata affidata a **Git**, con hosting remoto su **GitHub**. I cambiamenti sono stati costantemente salvati online sul branch principale, in modo che tutti i componenti del team potessero rimanere aggiornati in tempo reale sull'ultima versione del progetto.

2.2 Workflow e Strumenti (Issues, Branches, Pull Request)

Il team ha adottato un approccio snello, riducendo al minimo gli strumenti burocratici per massimizzare il tempo di sviluppo, in linea con i principi Agile.

Gestione del Repository (Trunk-Based):

- **Strategia Branching:** Lo sviluppo è avvenuto principalmente sul branch `main` (Trunk). La stabilità del codice è stata garantita "a monte" dalla metodologia *Pair Programming*: il codice veniva committato solo quando entrambi i programmatori (Driver e Navigator) confermano il funzionamento.
- **Integrazione Continua:** I *commit* frequenti hanno sostituito le Pull Requests formali, evitando conflitti complessi tipici dei branch a lunga vita.

Gestione delle Attività (Task Tracking):

- **Gestione Sincrona (Alternative to GitHub Issues):** Data la dimensione ridotta del team (3 persone) e la modalità di lavoro co-locata e sincrona, l'assegnamento degli issues è stato fatto in "tempo-reale".
- **Giustificazione:** La tracciabilità dei task e la risoluzione dei bug sono state gestite efficacemente tramite comunicazione diretta durante i meeting di allineamento e l'uso di *backlog* semplificati (es. liste condivise), garantendo una velocità di reazione immediata.

3. People Management and Team Organization

3.1 Struttura Organizzativa

L'organizzazione si basa su un modello **ibrido** che coniuga lo schema del *Chief Programmer Team*, adattato alle dimensioni ridotte del gruppo e integrato con metodologie Agile.

- **Chief Programmer** (Andrea Mondino):
    - *Responsabilità*: architettura di alto livello, definizione delle interfacce chiave, gestione del repository.
- **Programmers** (Matteo Megna, Matilde Scioscia):
    - *Responsabilità*: sviluppo di funzionalità specifiche, stesura e revisione critica dei documenti tecnici (Document Reviewer).

3.2 Metodologia di Lavoro (Pair Programming)

La gestione operativa delle persone si è basata sulla tecnica del **Pair Programming**, che è stata utilizzata come strumento di *knowledge sharing* e *real-time code review*. Le risorse non hanno lavorato in isolamento, ma in coppie.

1. **Driver:** La persona alla tastiera che scrive il codice.
2. **Navigator:** La persona che osserva, controlla errori in tempo reale e pensa alla strategia di design.

- ➢ **PP Verticale:** Utile per il trasferimento di competenze dal leader ai membri del team su componenti infrastrutturali complesse.

- ➢ **PP Orizzontale:** Applicato, ad esempio, allo sviluppo dei giochi per massimizzare la creatività algoritmica e ridurre gli errori logici attraverso il confronto costante tra *driver* (chi scrive) e *navigator* (chi osserva e pianifica).

Questa organizzazione ha permesso di:

- Livellare le competenze *(Knowledge Sharing*): la rotazione delle coppie ha favorito una rapida condivisione delle conoscenze tecniche, anche dal punto di vista della responsabilità dei Layer.
- Mitigazione del *Bus Factor*: nessuna parte di codice è di conoscenza esclusiva di un singolo sviluppatore.
- Qualità del codice: il pair programming ha agito come meccanismo di revisione sincrona, mantenendo alta la qualità senza necessitare di lunghe sessioni di *review* asincrone.