UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale,
dell'Informazione e della Produzione

Corso di Laurea Magistrale in Ingegneria Informatica

SPS-F1 Strategy

Sistema di Predizione e Ottimizzazione Strategie di Gara

Progetto per il corso di:
Progettazione, Algoritmi e Computabilità (38090-MOD1)

Team di Sviluppo:
Andrea Birolini (mat. 1087070)
Ivan Caccamo (mat. 1085892)
Luca Rossi (mat. 1086223)

Anno Accademico 2025/2026

Indice

Capitolo 1

Introduzione e Visione

1.1 Contesto e Motivazione

La Formula 1 moderna è uno sport definito dai dati. Con margini di vittoria spesso misurati in decimi di secondo, la strategia di gara è diventata determinante quanto la prestazione pura della vettura. Uno degli aspetti più critici e difficili da gestire è il **degrado degli pneumatici**, influenzato da molteplici variabili come la temperatura dell'asfalto, la tipologia di mescola e le caratteristiche del circuito.

Il progetto **SPS-F1 (Strategy Prediction System for F1)** nasce per rispondere alla necessità di analizzare dati storici complessi per fornire previsioni affidabili in tempo reale. A differenza dei sistemi statici tradizionali, SPS-F1 combina tecniche di *Machine Learning* per la predizione fisica del degrado con algoritmi di *Ottimizzazione Esatta* per il calcolo della strategia di sosta (pit-stop) matematicamente ottimale.

1.2 Obiettivi del Progetto

L'obiettivo primario è sviluppare un Sistema di Supporto alle Decisioni (DSS) che permetta agli ingegneri di strategia di:

1. **Predire il comportamento delle gomme:** Utilizzare modelli di regressione allenati su dati storici reali (dal 2021 al 2024) per stimare il tempo sul giro base (T_{base}) e il tasso di degrado (D_{rate}) in funzione delle condizioni meteo attuali.

2. **Ottimizzare la strategia:** Calcolare la sequenza di soste e l'utilizzo delle mescole che minimizza il tempo totale di gara, rispettando i vincoli regolamentari (es. obbligo di utilizzare almeno due mescole diverse).

3. **Simulare scenari:** Fornire un'interfaccia interattiva per testare ipotesi ("Cosa succede se la temperatura aumenta di 5°C ?") e visualizzare graficamente l'evoluzione dei tempi sul giro.

1.3 Metodologia di Sviluppo

In conformità con le linee guida del corso, il progetto è stato sviluppato seguendo la metodologia agile **AMDD (Agile Model Driven Development)**. Il ciclo di vita del software è stato suddiviso in iterazioni incrementali:

- **Iterazione 0 (Envisioning):** Definizione dei requisiti, analisi del dominio e setup dell'infrastruttura.
- **Iterazione 1 (Machine Learning):** Analisi esplorativa dei dati (EDA), training dei modelli predittivi in Python e creazione del microservizio REST.
- **Iterazione 2 (Core Algoritmico):** Sviluppo del backend Java, implementazione dell'algoritmo di Programmazione Dinamica e integrazione dei servizi.
- **Iterazione 3 (Frontend & QA):** Realizzazione dell'interfaccia utente, testing unitario e analisi della qualità del codice.

1.3.1 Mappatura Iterazioni-Artefatti

La seguente tabella illustra come i prodotti delle diverse iterazioni sono organizzati all'interno di questo documento finale, garantendo la tracciabilità tra il processo di sviluppo (AMDD) e la documentazione tecnica.

Iter.	Fase AMDD	Artefatti Prodotti	Capitolo/Sez.
0	Envisioning	Analisi dei Requisiti, Casi d'Uso	2
1	ML & Data Eng.	Pipeline Python, Architettura (Deployment)	4.2, 3.1
2	Core Logic	Algoritmo Java, Design Pattern, Sequence	4.1, 3.2
3	QA & Release	Unit Test, Analisi Statica, Manuale Utente	5, 6

Tabella 1.1: Corrispondenza tra le iterazioni di sviluppo e i capitoli della documentazione.

1.4 Tool Chain e Tecnologie

Per garantire modularità, scalabilità e prestazioni, è stata adottata un'architettura ibrida basata sui seguenti stack tecnologici:

1.4.1 Backend e Ottimizzazione

- **Linguaggio:** Java 17+ (LTS).
- **Framework:** Spring Boot 3.x (per la gestione delle API REST e Dependency Injection).
- **Build Tool:** Maven.
- **Testing:** JUnit 5 per i test unitari e di integrazione.

1.4.2 Data Science e Machine Learning

- **Linguaggio:** Python 3.9+.
- **Librerie:** Pandas (manipolazione dati), Scikit-Learn (Random Forest Regressor), Flask (esposizione del modello come API).
- **Formato Dati:** Dati tratti da FastF1 e trasformati localmente in CSV.

1.4.3 Frontend e Visualizzazione

- **Tecnologie:** HTML5, CSS3, JavaScript (ES6).
- **Librerie:** Bootstrap 5 (UI Responsiva), Chart.js (Grafici di telemetria).

1.5 Organizzazione del Team

Al fine di ottimizzare lo sviluppo, i membri del team hanno assunto responsabilità specifiche basate sulle proprie competenze:

Andrea Birolini (Architect & Integrator): Responsabile dell'architettura a microservizi, della definizione dei contratti API (JSON) e dell'orchestrazione tra il modulo Java e il servizio Python.

Ivan Caccamo (Algorithmist): Responsabile del "Design in Piccolo", focalizzato sulla modellazione matematica del problema di ottimizzazione, sull'implementazione dell'algoritmo DP e sull'analisi della complessità computazionale.

Luca Rossi (Data Scientist): Responsabile della pipeline di Machine Learning, dalla pulizia dei dati grezzi al training del modello Random Forest, fino all'esposizione delle previsioni via Flask.

Capitolo 2

Analisi dei Requisiti

In questo capitolo vengono definiti formalmente gli attori, i requisiti funzionali e non funzionali del sistema SPS-F1, identificati durante la fase iniziale di Envisioning (Iterazione 0). L'obiettivo è delineare il perimetro del sistema e le funzionalità che i diversi moduli software devono garantire.

2.1 Attori del Sistema

L'analisi del dominio ha portato all'identificazione di due attori primari che interagiscono con il sistema:

Ingegnere di Strategia (Stratega): È l'attore principale e l'utente finale del sistema. Si tratta di una figura esperta che utilizza l'applicazione per configurare gli scenari di gara, analizzare le previsioni fornite dal sistema e prendere decisioni basate sulle strategie ottimali proposte.

Data Engineer (Admin): Responsabile della pipeline dei dati. Esegue gli script Python per scaricare i dati aggiornati tramite l'API FastF1 e ri-addestrare il modello predittivo.

2.2 Requisiti Funzionali

I requisiti funzionali descrivono i comportamenti specifici del sistema e sono stati raggruppati in base al sottosistema di competenza: il modulo di Machine Learning (Python) e il nucleo algoritmico (Java).

2.2.1 Gestione Dati e Previsione (Backend ML)

Queste funzionalità risiedono nel microservizio Python e sono responsabili della trasformazione dei dati grezzi in parametri fisici utilizzabili per la simulazione.

- **RF-01 (Importazione Storico da FastF1):** Il sistema non mantiene un database relazionale storico, ma acquisisce i dati telemetrici (tempi, meteo, tyre wear) direttamente dalla libreria open-source **FastF1**. I dati vengono scaricati, filtrati per il periodo 2021-2024 e salvati localmente in formato CSV per il training. La documentazione tecnica di riferimento è consultabile all'indirizzo https://docs.fastf1.dev.

- **RF-02 (Training Modello):** Il sistema deve implementare una pipeline automatizzata per l'addestramento di modelli di regressione, capace di apprendere le caratteristiche di degrado delle diverse mescole (C1-C5) basandosi sulle condizioni ambientali.
- **RF-03 (Predizione Parametri):** A fronte di una richiesta contenente le specifiche del circuito e le condizioni meteo attuali, il sistema deve predire puntualmente:
  - *Base Lap Time (T_{base})*: Il tempo ideale sul giro a gomma nuova.
  - *Degradation Rate (D_{rate})*: Il coefficiente di decadimento della prestazione (secondi persi per giro).

2.2.2 Ottimizzazione Strategia (Core Java)

Queste funzionalità costituiscono il core business dell'applicazione e sono implementate nel backend Java Spring Boot.

- **RF-04 (Configurazione Scenario):** Il sistema deve fornire un'interfaccia per permettere allo Stratega di definire i parametri della gara, tra cui il numero di giri totali, temperatura dell'asfalto e dell'aria.
- **RF-05 (Calcolo Strategia Ottimale):** Il sistema deve eseguire un algoritmo di ottimizzazione esatta per determinare la combinazione di stint e soste che minimizza il tempo totale di percorrenza della gara, rispettando i vincoli regolamentari (es. utilizzo di almeno due mescole differenti).
- **RF-06 (Visualizzazione Risultati):** Il sistema deve presentare i risultati dell'elaborazione in formato leggibile, mostrando la strategia raccomandata, le soste previste e un confronto con le migliori alternative disponibili.

2.3 Requisiti Non Funzionali

I requisiti non funzionali definiscono i vincoli qualitativi e prestazionali a cui il sistema deve sottostare.

1. **RNF-01 (Performance):** L'algoritmo di ottimizzazione deve garantire tempi di risposta compatibili con l'utilizzo in tempo reale. Nello specifico, il calcolo della strategia ottimale per una gara standard (50-70 giri) deve completarsi in meno di 3 secondi.
2. **RNF-02 (Interoperabilità):** L'integrazione tra il modulo di calcolo (Java) e il modulo predittivo (Python) deve essere realizzata tramite API REST standard e scambio dati in formato JSON, garantendo l'indipendenza tecnologica dei componenti.
3. **RNF-03 (Affidabilità):** Il sistema deve implementare meccanismi di fallback per gestire l'eventuale assenza di dati storici specifici per un circuito, garantendo la continuità del servizio tramite l'uso di modelli generalizzati o valori di default sicuri.

2.4 Modellazione dei Requisiti

Per visualizzare le interazioni funzionali del sistema, è stato elaborato il Diagramma dei Casi d'Uso (Use Case Diagram), che mappa le funzionalità sopra descritte sugli attori identificati.

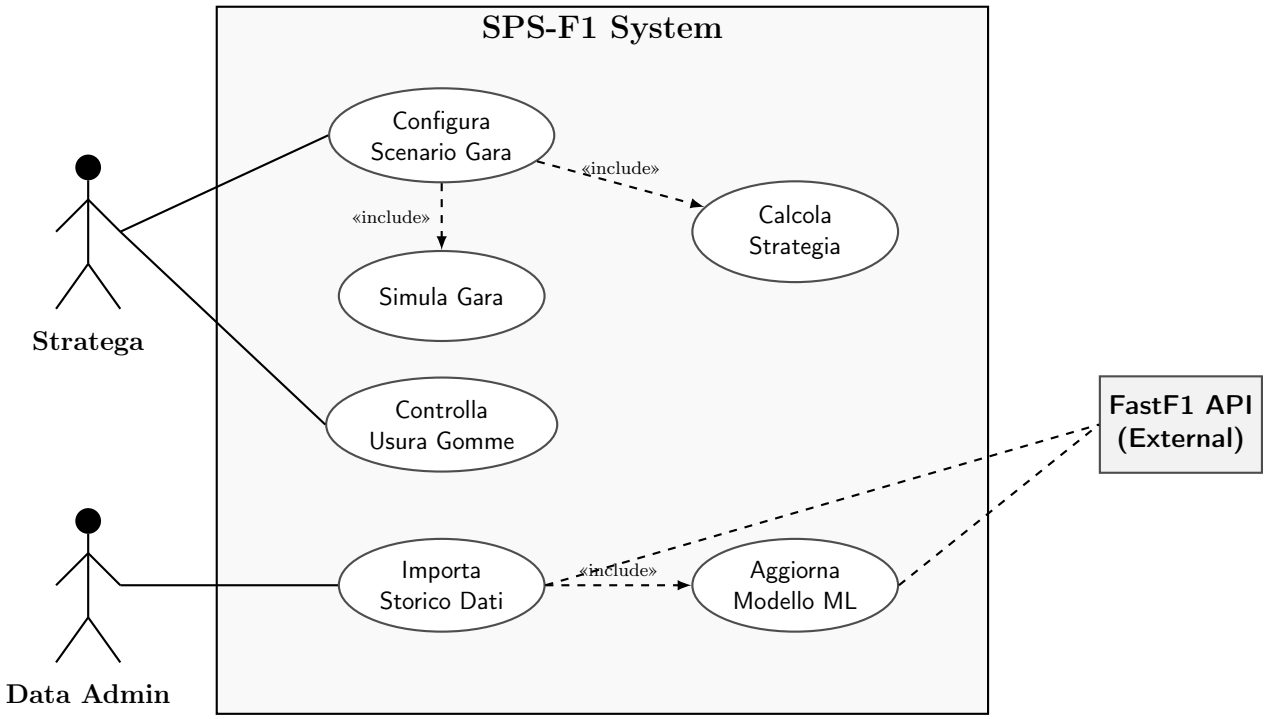


Figura 2.1: Diagramma dei Casi d'Uso.

Il modello evidenzia la separazione di responsabilità: lo Stratega opera sul fronte decisionale (configurazione e calcolo), mentre il Data Admin opera sul fronte manutentivo (gestione dati e modelli), supportando così l'intero ciclo di vita della strategia di gara.

Capitolo 3

Architettura del Sistema

In questo capitolo viene descritta l'architettura software del sistema SPS-F1 nella sua versione finale ("As-Built"). La documentazione segue il modello "4+1 Viste", focalizzandosi sulla vista fisica (Deployment), logica (Classi) e comportamentale (Sequenza).

3.1 Vista di Deployment

Il sistema adotta un'architettura a microservizi leggera ("Lightweight Microservices"), eliminando la necessità di un database server monolitico per i dati storici in favore di un approccio basato su file e API esterne.

I nodi principali dell'architettura sono:

- **Client Workstation:** Ospita il browser web per la fruizione del Frontend (HTML/JS).
- **Application Server (Java Spring Boot):** Nodo di orchestrazione.
  - Contiene l'applicazione Java (`.jar`).
  - Include un database **H2 Embedded (In-Memory)** utilizzato esclusivamente per la persistenza delle preferenze utente e il salvataggio delle strategie calcolate.
- **ML Service (Python Flask):** Nodo computazionale.
  - Esegue il runtime Python.
  - Gestisce la persistenza dei dati storici su file flat (`training_data.csv`) e del modello serializzato (`model.pkl`).
  - Interagisce con l'esterno (**FastF1 API**) durante la fase di data ingestion.

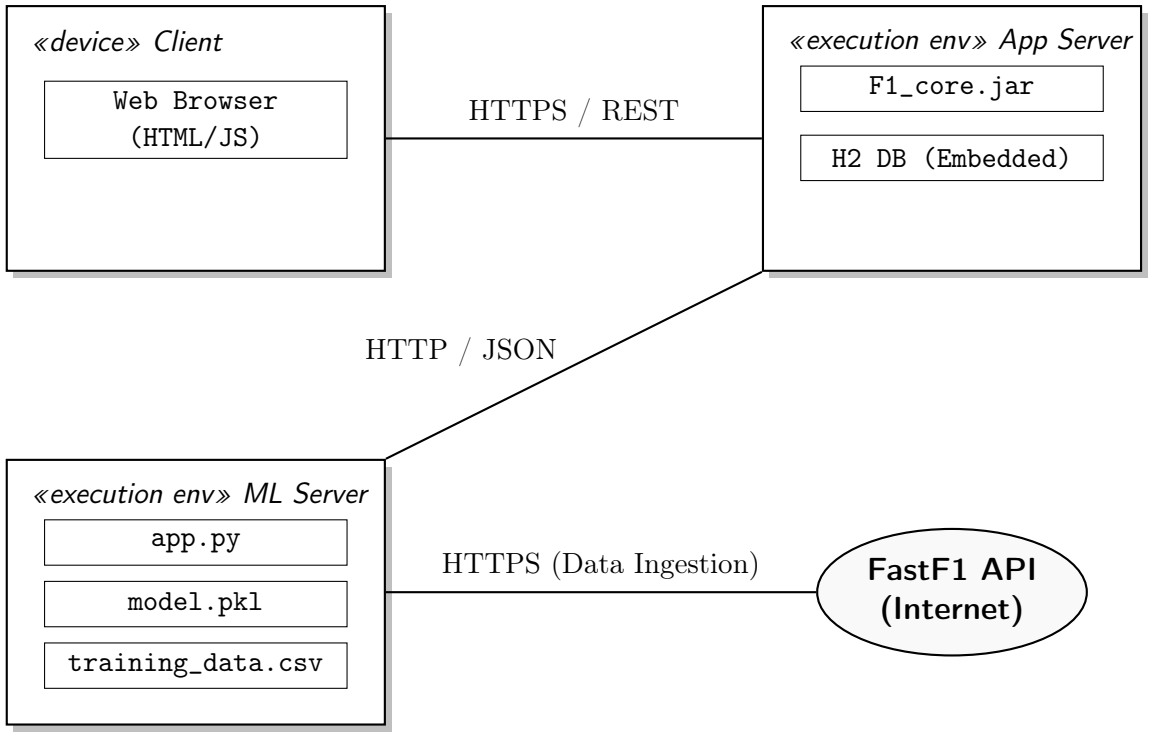


Figura 3.1: Diagramma di Deployment.

3.2 Vista Logica (Class Diagram)

Il cuore dell'applicazione risiede nel backend Java, strutturato secondo il pattern MVC (Model-View-Controller) tipico di Spring Boot. Di seguito vengono descritte le classi principali del pacchetto `com.ivancaccamo.pacf1`:

3.2.1 Componenti Principali

`StrategyController`: È il punto di ingresso delle API. Riceve le richieste HTTP dal frontend (es. `POST /api/strategy`), valida l'input e coordina gli altri servizi.

`PredictionService`: Agisce da client HTTP interno. È responsabile della comunicazione con il microservizio Python: serializza i dati della gara in JSON, chiama l'endpoint `POST :5000/predict` e deserializza la risposta.

`OptimizationEngine`: Contiene la logica algoritmica pura. Implementa l'algoritmo di Programmazione Dinamica descritto nel Capitolo 4. Prende in input le predizioni delle gomme e restituisce la lista delle strategie ottimali.

3.2.2 Modello del Dominio

- `RaceStrategy`: Rappresenta una strategia completa. Contiene una lista di `Stint`, il tempo totale stimato (`totalTime`) e il numero di soste (`pitStops`).
- `Stint`: Rappresenta una frazione di gara percorsa con un set di gomme. Attributi: `compound` (es. SOFT), `laps` (durata), `startLap`.

- `TyrePrediction`: DTO (Data Transfer Object) che mappa la risposta del servizio ML, contenente il tempo base e il tasso di degrado per una specifica mescola.

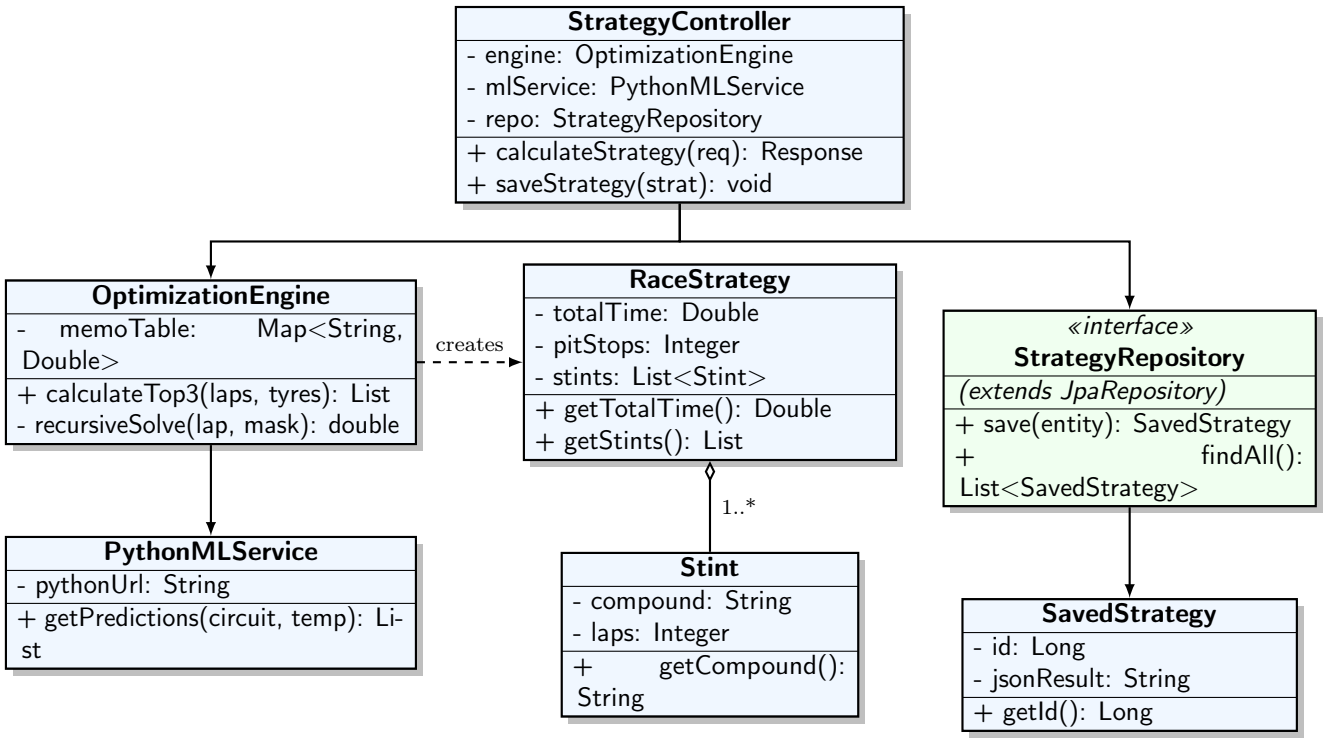


Figura 3.2: Class Diagram del sistema SPS-F1. Il diagramma riflette la struttura esatta dei package Java: il `StrategyController` orchestra i Service e utilizza il `StrategyRepository` per la persistenza.

3.3 Design Patterns

L'architettura del sistema SPS-F1 è stata progettata seguendo i principi della buona ingegneria del software. In particolare, sono stati identificati e applicati i seguenti Design Patterns per garantire modularità, manutenibilità e disaccoppiamento:

Singleton Pattern (Creazionale): In Spring Boot, i componenti annotati con `@Service`, `@Controller` e `@Repository` sono gestiti nativamente come Singleton. Questo garantisce che esista una sola istanza condivisa di classi pesanti come `OptimizationEngine` e `StrategyController` all'interno del container, ottimizzando l'uso della memoria e le performance, evitando costose ri-istanziazioni ad ogni richiesta HTTP.

Facade Pattern (Strutturale): La classe `StrategyController` agisce come una *Facade* per il sistema backend. Essa fornisce un'interfaccia semplificata (gli endpoint REST) al Frontend, nascondendo la complessità sottostante che coinvolge la coordinazione tra il motore di ottimizzazione (`OptimizationEngine`), il servizio di Machine Learning esterno e il database. Il client non deve conoscere le dipendenze interne ma interagisce solo con il Controller.

Adapter Pattern (Strutturale): Il componente `PythonMLService` implementa il pattern *Adapter*. Il microservizio Python restituisce dati in formato JSON grezzo

su protocollo HTTP, che è incompatibile con il modello a oggetti del dominio Java. Il servizio agisce quindi da adattatore, convertendo le risposte esterne in oggetti Java tipizzati (`TyrePrediction`) consumabili dall'algoritmo di ottimizzazione, rendendo trasparente la comunicazione tra i due linguaggi.

3.4 Vista Comportamentale (Sequence Diagram)

Per illustrare il flusso dinamico del sistema, si analizza lo scenario principale: **"Richiesta di Calcolo Strategia"**.

1. L'**Utente** seleziona il circuito e clicca "Calcola" sul Frontend.
2. Il Frontend invia una `GET` al **StrategyController** Java con i parametri (meteo, giri).
3. Il Controller delega al **PredictionService** il recupero dei dati sulle gomme.
4. Il Service chiama via HTTP il server **Python Flask** (`/predict`).
5. Python calcola il degrado usando il modello ML e restituisce un JSON.
6. Il Controller passa i dati ricevuti all'**OptimizationEngine**.
7. L'Engine esegue l'algoritmo DP e restituisce una lista di `RaceStrategy`.
8. Il Controller invia la risposta JSON finale al Frontend per la visualizzazione.

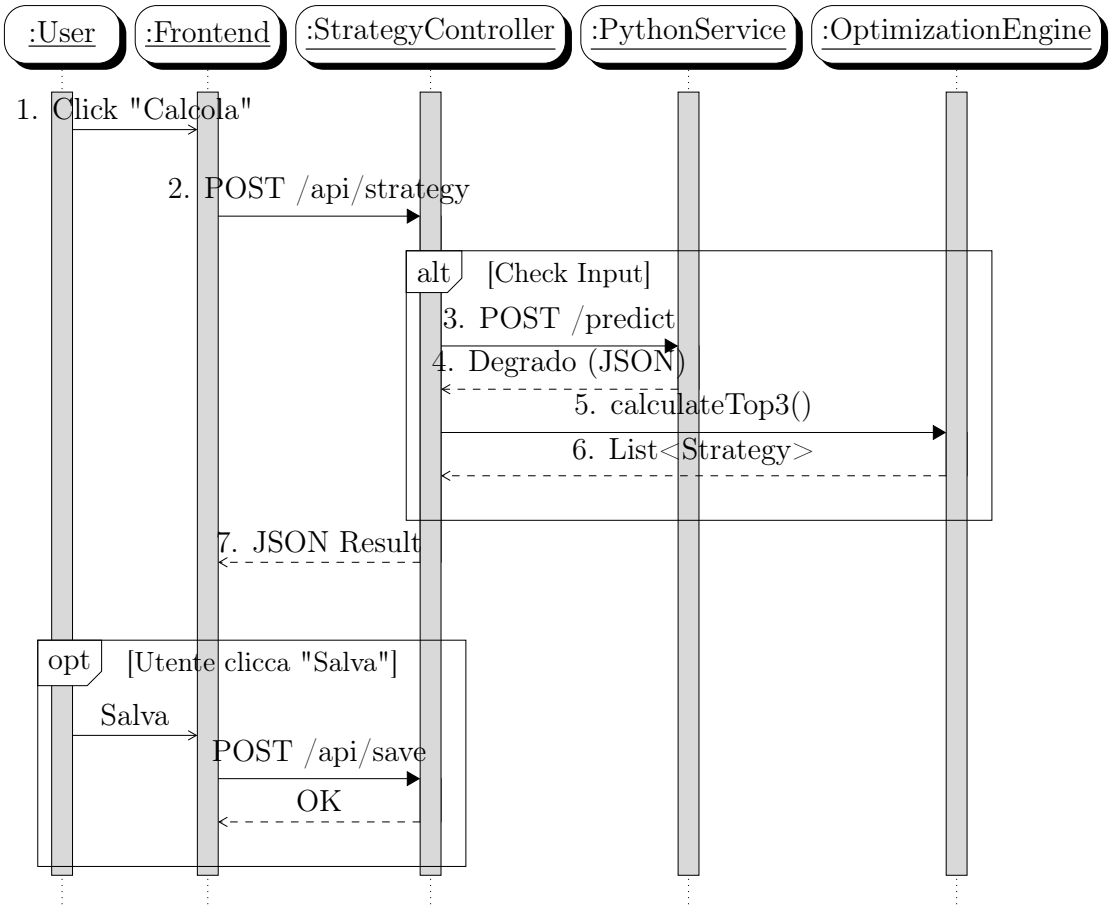


Figura 3.3: Sequence Diagram dello scenario principale "Calcolo Strategia". Mostra l'interazione sincrona tra Frontend, Backend Java, il servizio predittivo Python e il motore di calcolo interno.

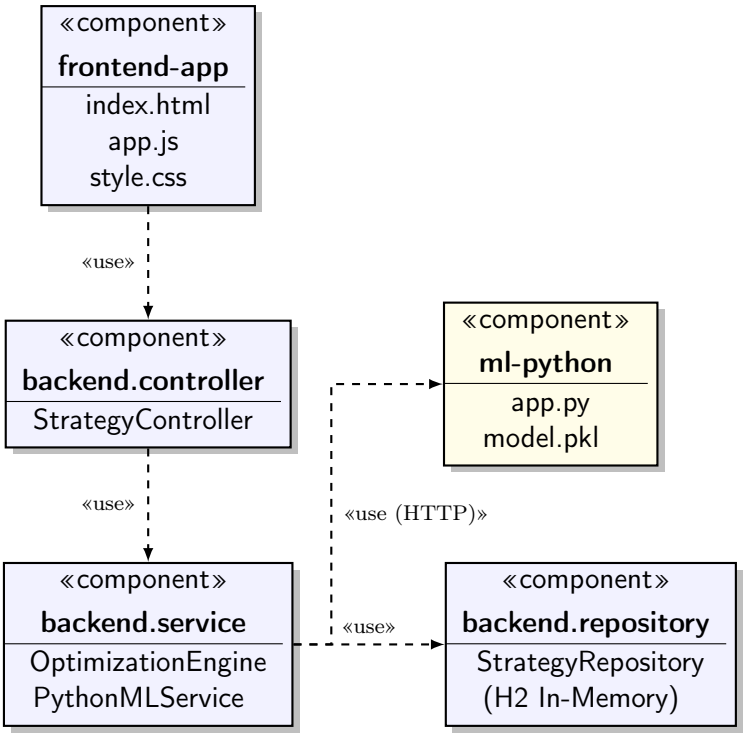


Figura 3.4: Component Diagram del sistema SPS-F1. La struttura riflette l'organizzazione dei package Java (`controller`, `service`, `repository`) e l'integrazione con il modulo Python esterno.

Capitolo 4

Algoritmi e Strutture Dati

In questo capitolo viene dettagliato il "Design in Piccolo" dei componenti core del sistema. Vengono analizzati gli algoritmi implementati per l'ottimizzazione strategica (Java) e per la predizione dei parametri fisici (Python).

4.1 Algoritmo di Ottimizzazione (Java)

In accordo con gli obiettivi dell'Iterazione 2, viene qui dettagliato il "Design in Piccolo" del motore di ottimizzazione (Classi `OptimizationEngine` e `RaceStrategy`).

Il problema della determinazione della strategia di gara ottimale è stato modellato come una ricerca del cammino minimo su un **DAG (Directed Acyclic Graph)** pesato, risolto mediante **Programmazione Dinamica (DP)** con approccio *Memoization*.

Sia T l'insieme delle mescole disponibili (Soft, Medium, Hard) e L_{tot} il numero totale di giri. Definiamo la funzione obiettivo $Solve(lap, mask)$ che restituisce il tempo minimo necessario per completare la gara dal giro lap avendo già utilizzato il set di mescole rappresentato dalla bitmask $mask$.

4.1.1 Analisi della Complessità

L'efficienza dell'algoritmo è garantita dall'uso della *memoization*, che evita di ricalcolare sottoproblemi già risolti (principio della sottostruttura ottima).

Definiamo le variabili del problema:

- N: Numero totale di giri (es. 57 per Bahrain).
- K: Numero di mescole disponibili (generalmente 3: Soft, Medium, Hard).

Complessità Spaziale

La tabella di memorizzazione (`MemoTable`) immagazzina gli stati unici del problema. Uno stato è definito univocamente dalla coppia $(giroCorrente, mascheraGomme)$. Poiché i valori possibili per $giroCorrente$ sono N e per $mascheraGomme$ sono 2^K, la complessità spaziale è:

$$Space = O(N \cdot 2^K) \tag{4.1}$$

Nel caso reale ($N = 60, K = 3$), lo spazio richiesto è $60 \cdot 2^3 = 480$ stati, un valore trascurabile.

Algorithm 1 Ottimizzazione Strategia di Gara (RaceStrategyOptimization)

Input: $TotalLaps$ (N), $TyreModels$ (K)
Output: Lista delle strategie ottimali
function MAIN(N, K)
    $Candidates \leftarrow \emptyset$
    for each $tyre \in K$ **do**
        for $stintLen = 1$ **to** $(N - 1)$ **do**
            $FirstStintTime \leftarrow CalculateStintCost(tyre, stintLen)$
            $NextMask \leftarrow BitMask(tyre)$
            $RemainingTime \leftarrow$ RECURSIVESOLVE$(stintLen, NextMask)$
            if $RemainingTime < \infty$ **then**
                $TotalTime \leftarrow FirstStintTime + RemainingTime$
                $Candidates \leftarrow Candidates \cup \{Strategy(tyre, stintLen, TotalTime)\}$
            end if
        end for
    end for
    return SORTANDFILTER(Candidates)
end function
function RECURSIVESOLVE$(currentLap, usedMask)$
    if $MemoTable$ contiene $[currentLap, usedMask]$ **then**
        return $MemoTable[currentLap, usedMask]$
    end if
    if $currentLap == TotalLaps$ **then**
        if BITCOUNT$(usedMask) \geq 2$ **then**
            return 0 ▷ Strategia Valida
        else
            return ∞ ▷ Strategia Illegale
        end if
    end if
    $MinTime \leftarrow \infty$
    for each $tyre \in K$ **do**
        for $stintLen = 1$ **to** $(TotalLaps - currentLap)$ **do**
            $DriveTime \leftarrow$ CALCULATESTINTCOST$(tyre, stintLen)$
            $PitLoss \leftarrow 20.0$
            $NewMask \leftarrow usedMask \vee$ BITMASK$(tyre)$
            $PathTime \leftarrow DriveTime + PitLoss +$ RECURSIVESOLVE$(currentLap + stintLen, NewMask)$
            if $PathTime < MinTime$ **then**
                $MinTime \leftarrow PathTime$
                SAVEDECISION$(currentLap, usedMask, tyre, stintLen)$
            end if
        end for
    end for
    $MemoTable[currentLap, usedMask] \leftarrow MinTime$
    return $MinTime$
end function

Complessità Temporale

Per ogni stato non ancora calcolato, l'algoritmo esegue dei cicli annidati per determinare la transizione ottimale (iterando su mescole K e lunghezza stint fino a N). La complessità temporale totale è il prodotto tra il numero di stati e il lavoro per stato:

$$Time = O(\text{Stati} \cdot \text{Transizioni}) = O((N \cdot 2^K) \cdot (K \cdot N)) = O(N^2 \cdot K \cdot 2^K) \tag{4.2}$$

Conclusione: Nonostante il termine quadratico N^2, dato che N è limitato per la F1 (max ≈ 70) e K è molto piccolo ($K = 3$), l'algoritmo rispetta il requisito RNF-01 (< 3 secondi).

4.2 Machine Learning (Python)

Il modulo predittivo costituisce il cuore dell'intelligenza del sistema SPS-F1. Esso utilizza una pipeline di Machine Learning supervisionato, basata sulla libreria *Scikit-Learn*, progettata per stimare i parametri critici di degrado degli pneumatici in funzione delle condizioni ambientali e del tracciato.

4.2.1 Pipeline di Training e Architettura

Il modello non si limita a una semplice regressione, ma è strutturato come una **Pipeline** unificata che integra pre-elaborazione e inferenza. Il processo di addestramento sui dati storici (estratte tramite ETL, vedi Sezione precedente) segue i seguenti step logici:

1. **Preprocessing Ibrido:** I dati in ingresso sono di natura eterogenea. Per gestirli correttamente all'interno di un unico feature vector, viene utilizzato un `ColumnTransformer`:
    - Le variabili categoriche (Nome del Circuito, Mescola) vengono trasformate tramite codifica *One-Hot*, che crea vettori binari sparsi per evitare di introdurre ordinalità fittizie.
    - Le variabili numeriche (Temperatura Aria, Temperatura Asfalto) vengono normalizzate tramite *Standard Scaling* (media 0, varianza 1) per migliorare la stabilità numerica durante il training.
2. **Modellazione Multi-Output:** Viene utilizzato un **Random Forest Regressor** configurato con 100 alberi decisionali ($n\_estimators = 100$). Questa scelta è motivata da due fattori:
    - *Non-linearità:* Gli alberi decisionali sono eccellenti nel catturare relazioni non lineari, come il calo improvviso di prestazione ("cliff") tipico degli pneumatici Pirelli.
    - *Multi-output Nativo:* Il modello è in grado di predire simultaneamente il vettore target (T_{base}, D_{rate}) senza la necessità di addestrare due modelli separati, mantenendo la coerenza tra tempo sul giro e degrado.
3. **Validazione e Serializzazione:** Il modello viene valutato su un test set (10% dei dati) per verificare la capacità di generalizzazione (evitando l'overfitting) e successivamente serializzato in formato `.pkl` per essere caricato dal microservizio API.

Algorithm 2 Addestramento del Modello Predittivo

Require: File dataset F_{csv} = "training_data.csv"
Ensure: Modello serializzato M_{pkl} = "model.pkl"

1: **procedure** TRAINMODEL
2: *// 1. Caricamento e selezione dati*
3: $D \leftarrow \text{LoadCSV}(F_{csv})$
4: $X \leftarrow D[\{\text{"circuit"}, \text{"compound"}, T_{air}, T_{track}, \dots\}]$ ▷ Features
5: $Y \leftarrow D[\{\text{"base\_time"}, \text{"degradation\_rate"}\}]$ ▷ Target Multi-output
6: *// 2. Definizione della Pipeline di Preprocessing*
7: $\Phi_{cat} \leftarrow \text{OneHotEncoder}(\text{"circuit"}, \text{"compound"})$
8: $\Phi_{num} \leftarrow \text{StandardScaler}(T_{air}, T_{track})$
9: $\mathcal{T} \leftarrow \text{ColumnTransformer}(\Phi_{cat}, \Phi_{num})$
10: *// 3. Configurazione del Regressore (Ensemble)*
11: $\mathcal{R} \leftarrow \text{RandomForestRegressor}(n_{estimators} = 100)$
12: *// 4. Creazione Pipeline unificata*
13: $\mathcal{P} \leftarrow \text{Pipeline}(\text{steps} = \{\mathcal{T}, \mathcal{R}\})$
14: *// 5. Split e Addestramento*
15: $(X_{train}, X_{test}, Y_{train}, Y_{test}) \leftarrow \text{TrainTestSplit}(X, Y, ratio = 0.1)$
16: $\mathcal{P}.\text{fit}(X_{train}, Y_{train})$
17: *// 6. Valutazione e Serializzazione*
18: $\hat{Y}_{test} \leftarrow \mathcal{P}.\text{predict}(X_{test})$
19: $E_{rmse} \leftarrow \text{RMSE}(Y_{test}[\text{"deg"}], \hat{Y}_{test}[\text{"deg"}])$
20: $S_{r2} \leftarrow \text{R2\_Score}(Y_{test}[\text{"deg"}], \hat{Y}_{test}[\text{"deg"}])$
21: **Print**("Model Accuracy (R2):", S_{r2})
22: **SavePickle**($\mathcal{P}, M_{pkl}$)
23: **end procedure**

Dettagli Implementativi

L'Algoritmo 2 evidenzia l'uso della classe `Pipeline` (riga 14), fondamentale per l'ingegnerizzazione del software ML. Incapsulando i passaggi di trasformazione ($\mathcal{T}$) e il regressore ($\mathcal{R}$) in un unico oggetto, si garantisce che le stesse identiche trasformazioni applicate al training set vengano applicate automaticamente ai nuovi dati in fase di predizione. Questo approccio previene il comune errore di *data leakage* e semplifica notevolmente il deployment del modello nel microservizio Python.

Capitolo 5

Quality Assurance

In conformità con i requisiti di affidabilità e le linee guida del corso, il progetto è stato sottoposto a un rigoroso processo di validazione, suddiviso in analisi dinamica (testing funzionale e di integrazione) e analisi statica (qualità del codice).

5.1 Analisi Dinamica

5.1.1 Unit Testing e Copertura (JUnit 5)

È stata sviluppata una suite di test unitari utilizzando il framework **JUnit 5** per verificare la correttezza del core algoritmico (`OptimizationEngine`). Sono stati coperti i seguenti scenari critici:

Standard Race: Verifica il calcolo di una strategia valida per una gara di 50 giri (tempo positivo, pit-stop presenti).

Short Race: Verifica il comportamento in gare sprint (15 giri), controllando che gli stint non eccedano la durata della gara.

Edge Case (No Tyres): Verifica la gestione degli errori (lista vuota di input), garantendo che il sistema non vada in crash ma restituisca una lista vuota.

Esito e Copertura

L'esecuzione automatizzata tramite Maven ha confermato il superamento di tutti i test case definiti (*Build Success*).

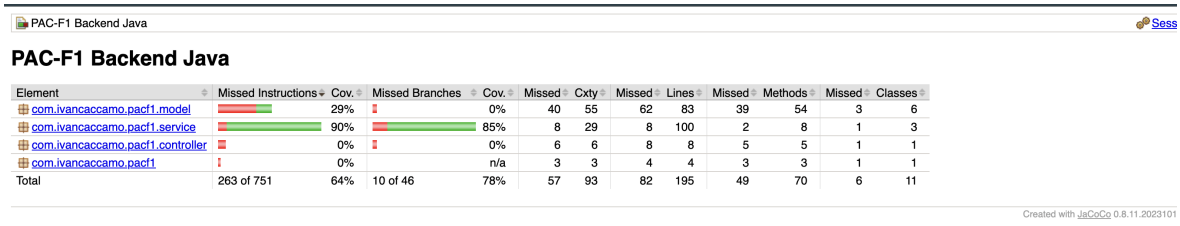
```
Results:

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

--- AVVIO ALGORITMO DP (N SOSTE) - RICERCA ESAUSTIVA TOP 3 ---
--- AVVIO ALGORITMO DP (N SOSTE) - RICERCA ESAUSTIVA TOP 3 ---
Miglior tempo trovato: 4590.059999999999
--- AVVIO ALGORITMO DP (N SOSTE) - RICERCA ESAUSTIVA TOP 3 ---
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  9.104 s
Finished at: 2026-01-14T16:24:08+01:00
------------------------------------------------------------------------
```

Figura 5.1: Log di esecuzione dei test JUnit in ambiente Maven: Build Success.

Inoltre, è stata generata l'analisi della **Code Coverage** tramite il plugin JaCoCo. Come mostrato in Figura 5.2, l'analisi evidenzia una copertura del **90%** per il package `service`, confermando che la quasi totalità della logica di business e dei rami decisionali dell'algoritmo è stata verificata.

Le percentuali inferiori negli altri package sono frutto di una precisa scelta progettuale:

- **Model (29%):** Contiene classi POJO (Plain Old Java Objects) con soli metodi getter/setter, il cui testing non aggiunge valore alla robustezza del sistema.
- **Controller (0%):** La verifica degli endpoint REST è stata delegata interamente ai test di integrazione e API Testing (tramite Postman), descritti nella sezione successiva.


PAC-F1 Backend Java

Sess

PAC-F1 Backend Java

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ivancaccamo.pacf1.model		29%		0%	40	55	62	83	39	54	3	6
com.ivancaccamo.pacf1.service		90%		85%	8	29	8	100	2	8	1	3
com.ivancaccamo.pacf1.controller		0%		0%	6	6	8	8	5	5	1	1
com.ivancaccamo.pacf1		0%		n/a	3	3	4	4	3	3	1	1
Total	263 of 751	64%	10 of 46	78%	57	93	82	195	49	70	6	11

Created with JaCoCo 0.8.11.2023101

Figura 5.2: Report JaCoCo: focus sulla copertura del Service (Core Business Logic) al 90%.

5.1.2 API Testing e Integrazione (Postman)

Per completare la validazione del sistema, è stata eseguita una fase di *API Testing* manuale utilizzando il tool Postman.

Questa attività ha avuto il duplice scopo di:

1. Verificare l'integrazione tra i microservizi e la corretta esposizione degli endpoint REST.

2. Validare il livello `Controller`, che per scelta progettuale è stato escluso dagli unit test (si veda la sezione precedente) in favore di un testing *end-to-end*.

La Figura 5.3 mostra l'esito positivo di una richiesta `POST` inviata all'endpoint di calcolo della strategia. Come si evince dallo screenshot:

- Il server risponde con status code **200 OK**, confermando che la richiesta è stata ricevuta ed elaborata senza errori infrastrutturali.
- Il payload di risposta rispetta la struttura JSON attesa (DTO), confermando la corretta serializzazione dei dati in uscita, indipendentemente dai valori specifici calcolati per l'input di test fornito.

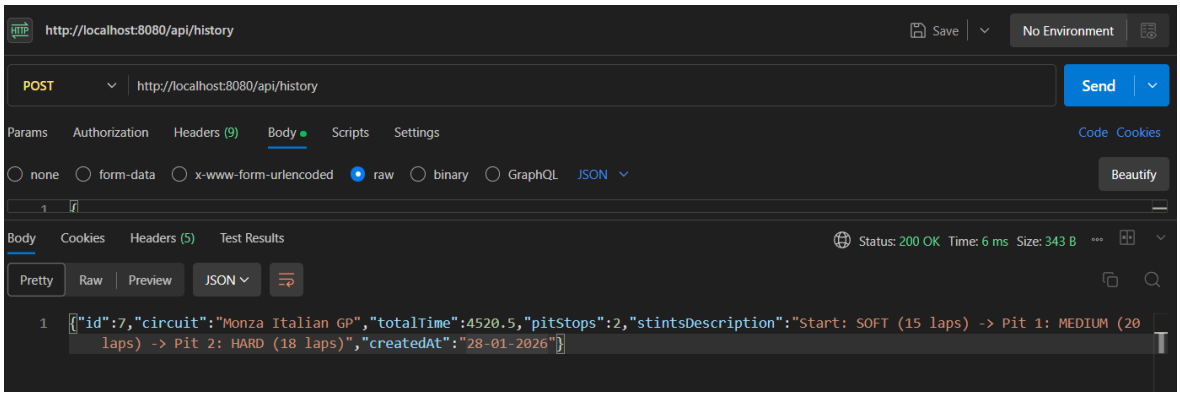


Figura 5.3: Test funzionale dell'API REST: il server elabora correttamente la richiesta restituendo lo Status 200 OK e la struttura JSON prevista.

5.2 Analisi Statica e Metriche Software

In conformità con i requisiti di qualità del software e le linee guida del corso, il progetto è stato sottoposto a un'analisi statica approfondita. L'obiettivo è stato duplice: verificare la qualità del codice (assenza di bug e code smells) e valutare la bontà dell'architettura (disaccoppiamento e modularità).

Per questa fase, si è scelto di sostituire il tool *STAN4J* (suggerito nelle specifiche ma ormai deprecato e incompatibile con i moderni ambienti Java/macOS) con una combinazione di strumenti standard industriali: JDepend per le metriche architetturali e il motore di linting di VS Code per la qualità del codice.

5.2.1 Metriche Architetturali (JDepend)

L'analisi strutturale è stata eseguita tramite il plugin Maven di **JDepend**, che misura la qualità del design in termini di estensibilità, riusabilità e manutenibilità dei package.

La Figura 5.5 mostra il report riassuntivo generato. I dati confermano una corretta applicazione dei pattern architetturali:

- **Assenza di Cicli:** L'analisi (confermata dalla sezione "Cycles" del report) non ha rilevato dipendenze circolari tra i package. Questo garantisce un'architettura a

strati pulita, dove le dipendenze scorrono in una sola direzione (dal Controller verso il Service e il Repository).

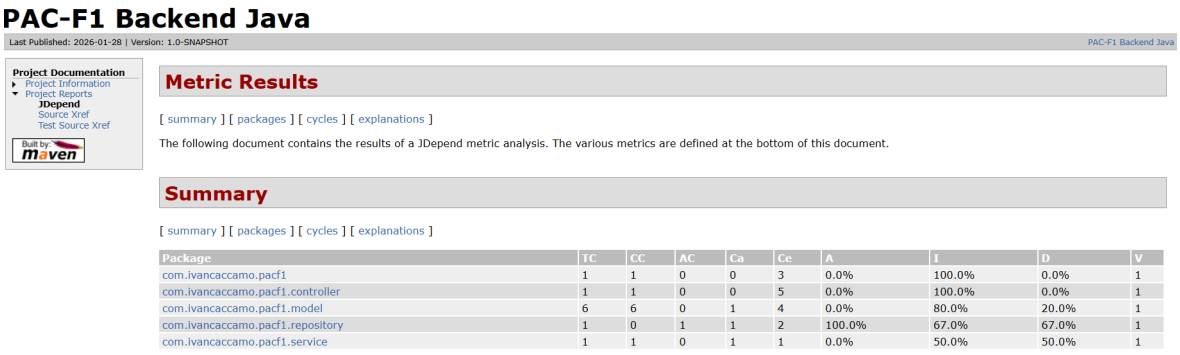
Cycles

[summary] [packages] [cycles] [explanations]

There are no cyclic dependencies.

Figura 5.4: Report JDepend: Assenza di cicli.

- **Abstractness (A):** Il package `repository` presenta un valore di Astrattezza pari al **100%** (1.0). Questo è il risultato ideale e atteso, poiché in Spring Data JPA i repository sono definiti interamente come *Interfacce*, disaccoppiando completamente la logica di accesso ai dati dall'implementazione concreta.
- **Instability (I):**
  - Il package `controller` ha un'instabilità del **100%** (1.0). Questo valore è corretto per un componente di "frontiera": il Controller dipende dai livelli sottostanti (Service/Model) ma nessun altro componente Java dipende da esso.
  - Il package `model` mostra un buon bilanciamento, essendo utilizzato trasversalmente da tutti gli altri moduli.



PAC-F1 Backend Java

Last Published: 2026-01-28 | Version: 1.0-SNAPSHOT — PAC-F1 Backend Java

Project Documentation
- Project Information
- Project Reports
  - JDepend
  - Source Xref
  - Test Source Xref

Built by maven

Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
com.ivancaccamo.pacf1	1	1	0	0	3	0.0%	100.0%	0.0%	1
com.ivancaccamo.pacf1.controller	1	1	0	0	5	0.0%	100.0%	0.0%	1
com.ivancaccamo.pacf1.model	6	6	0	1	4	0.0%	80.0%	20.0%	1
com.ivancaccamo.pacf1.repository	1	0	1	1	2	100.0%	67.0%	67.0%	1
com.ivancaccamo.pacf1.service	1	1	0	1	1	0.0%	50.0%	50.0%	1

Figura 5.5: Report JDepend: Tabella riassuntiva delle metriche. Si noti l'astrattezza totale dei Repository e la corretta instabilità del Controller.

Capitolo 6

Manuale Utente

In questo capitolo viene fornita una guida all'installazione e all'utilizzo del sistema SPS-F1. Per semplificare le operazioni di deployment e garantire la corretta sincronizzazione tra i microservizi, è stata predisposta una procedura di avvio automatizzata tramite script batch.

6.1 Requisiti di Sistema

Prima di avviare l'applicazione, assicurarsi che sulla macchina siano installati e configurati nelle variabili d'ambiente i seguenti componenti:

- **Java JDK 17** o superiore.
- **Maven 3.6** o superiore (necessario per la build del backend).
- **Python 3.9** o superiore (con gestore pacchetti `pip`).
- **Browser Web Moderno** (Google Chrome, Firefox, Edge).

6.2 Installazione e Avvio

Il sistema è composto da due moduli distinti (Machine Learning e Backend Java) che devono comunicare tra loro. L'avvio dell'intero ecosistema è gestito dallo script di orchestrazione `start.bat` (o `install_and_run.bat`) presente nella root del progetto.

6.2.1 Procedura Automatica (Script Batch)

Lo script automatizza l'intera catena di inizializzazione eseguendo sequenzialmente le seguenti operazioni:

1. **Setup Ambiente Python:** Verifica la presenza del virtual environment; se assente, lo crea e installa automaticamente le dipendenze definite in `requirements.txt` (pandas, scikit-learn, flask, ecc.).
2. **Build Backend Java:** Esegue il comando Maven per compilare il codice sorgente e scaricare le librerie Spring Boot necessarie, generando l'artefatto eseguibile.

3. **Avvio Concorrente:** Lancia in parallelo due processi:
    - Il microservizio Python (API Flask) sulla porta `5000`.
    - Il server Java (Tomcat embedded) sulla porta `8080`.

Esecuzione

Per avviare il sistema è sufficiente fare doppio click sul file `.bat` o eseguirlo da terminale nella cartella principale:

```
# Esecuzione script di avvio automatico
.\start.bat
```

Attendere che il terminale mostri i messaggi di conferma:

- Python: `Running on http://127.0.0.1:5000`
- Java: `Started StrategyApplication in X seconds`

A questo punto il sistema è pienamente operativo.

6.3 Guida all'Uso

6.3.1 Configurazione della Gara

Accedere tramite browser all'indirizzo `http://localhost:8080`. L'interfaccia principale (Figura 6.1) permette di configurare lo scenario di simulazione:

1. Selezionare il *Circuito* dal menu a tendina (es. Monza, Bahrain).
2. Impostare il numero di *Giri Totali* (default: 57).
3. Regolare le temperature di *Aria* e *Asfalto* tramite gli slider. Questi parametri influenzano direttamente il modello di degrado termico delle gomme.

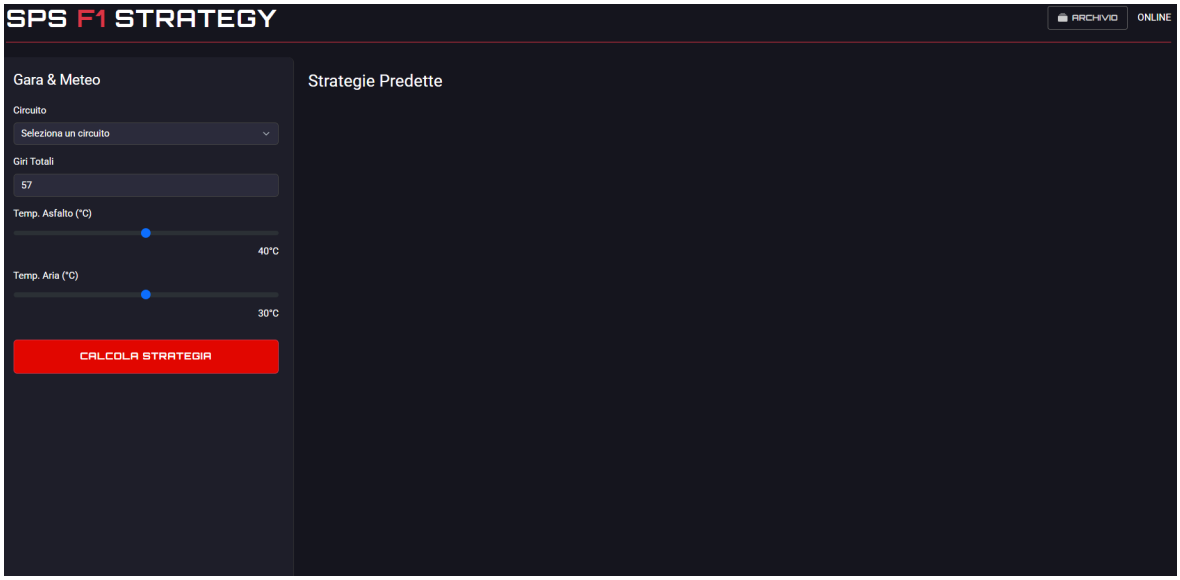


Figura 6.1: Pannello di controllo per la configurazione dei parametri di gara.

6.3.2 Analisi delle Strategie e Grafici

Cliccando sul pulsante rosso **"CALCOLA STRATEGIA"**, il sistema interroga il modello ML, elabora i dati tramite l'algoritmo di ottimizzazione e mostra le migliori opzioni disponibili.

- **Visualizzazione Card:** Vengono mostrate le *Top 3 Strategie*, ordinate per tempo totale di gara stimato, con il dettaglio delle mescole da utilizzare per ogni stint.

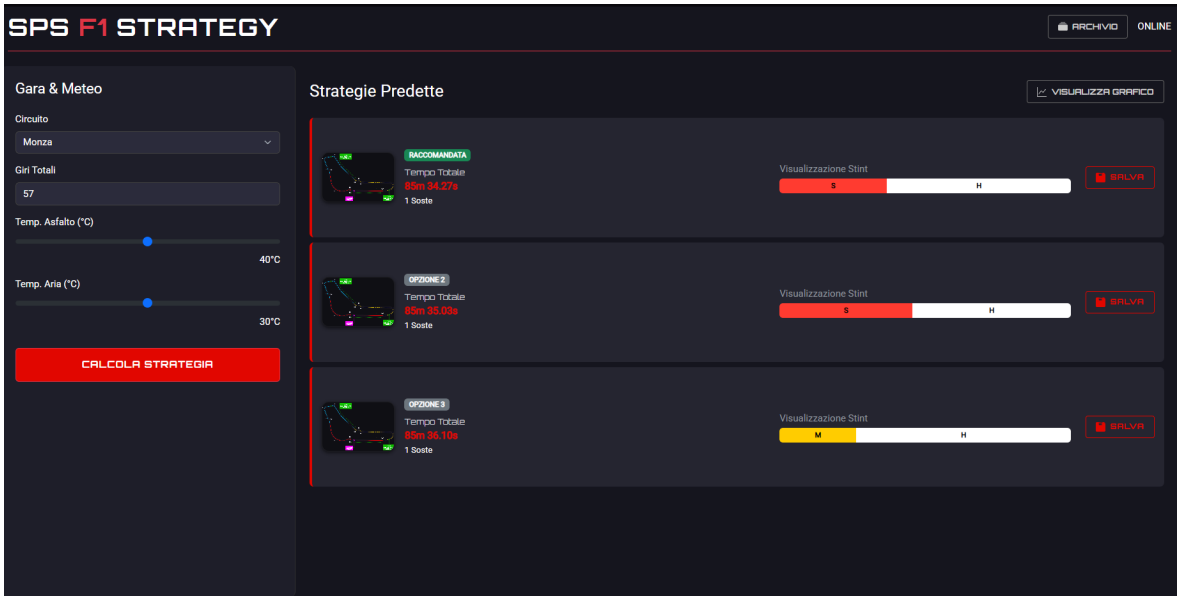


Figura 6.2: Visualizzazione delle strategie predette sotto forma di card.

- **Telemetria Comparativa:** Cliccando su "VISUALIZZA GRAFICO" in alto a destra, appare in basso il grafico che mostra l'andamento dei tempi sul giro (Lap Times) per le diverse strategie, permettendo un confronto visivo del passo gara (Figura 6.3).

Figura 6.3: Visualizzazione del grafico di comparazione telemetrica.

6.3.3 Salvataggio e Storico

È possibile salvare una strategia ritenuta interessante cliccando sul pulsante **"SALVA"** presente su ogni card. Per visualizzare le simulazioni archiviate, cliccare sul pulsante **"ARCHIVIO"** nella barra di navigazione superiore. Si aprirà una finestra modale contenente lo storico delle strategie salvate nel database.

Conclusioni

Il progetto SPS-F1 ha raggiunto tutti gli obiettivi prefissati per l'Iterazione corrente. Il sistema dimostra come l'integrazione tra algoritmi deterministici (Programmazione Dinamica) e modelli predittivi basati su dati (Machine Learning) possa fornire un supporto decisionale efficace in contesti complessi e dinamici come la Formula 1.

I test effettuati, documentati nei capitoli precedenti, confermano la robustezza dell'architettura a microservizi e l'efficacia della pipeline di automazione (scripting e analisi statica), garantendo un sistema manutenibile e rispettoso dei requisiti di performance real-time.