Progetto di Ingegneria del Software 2025/26

Università Ca' Foscari Venezia

Piano di testing

Versione 1.1

Colombo

26/01/2026

Document Informations

Green Drive	GD	
Deliverable	Documento dei requisiti	
Data di Consegna	18/11/2025	
Team Leader	Raffaele Cuniolo	903407@stud.unive.it
Team members	Alberto Barison	901146@stud.unive.it
	Riccardo Brollo	902485@stud.unive.it
	Gabriele Bute	895898@stud.unive.it
	Alessandro Derevytskyy	899454@stud.unive.it

Document History

Version	Issue Date	Stage	Changes	Contributors
1.0	18/11/2025	Draft	Presentazione iniziale	Alberto Barison Alessandro Derevytskyy Gabriele Bute
1.1	26/01/2026	Final	Sistemati valori requisiti funzionali	Gabriele Bute

Indice

1. Introduzione

Il seguente documento illustra gli approcci e le tecniche che verranno adottate per la fase di testing del progetto. Particolare attenzione viene posta sui criteri adottati per i singoli test e le modalità in cui andranno ad essere implementati per i singoli requisiti. Si conclude con una visione più generale legata ai vincoli e ai requisiti che possono influenzare le modalità di testing.

2. Glossario

- **Account**: Consiste in un insieme di dati critici (nome, cognome, indirizzo mail…) attraverso i quali è possibile usufruire dell'applicativo, sia il profilo del comune che del singolo cittadino. La gestione dell'insieme di dati è decentralizzata rispetto all'account.
- **Log in e Log out:** Procedure che consentono rispettivamente di entrare e uscire da un sistema informatico, permettendo al sistema di riconoscere l'utente e gestire la sua sessione in modo sicuro.
- **Token:** Elemento digitale utilizzato per l'autenticazione, che racchiude informazioni sull'identità dell'utente e sui permessi di accesso concessi dal sistema.
- **API (Application Programming Interface):** Insieme di regole e strumenti che permettono a diversi programmi o componenti software di comunicare e interagire tra loro.
- **Debugger:** Programma che consente di analizzare, individuare e correggere gli errori presenti nel codice di un software durante la fase di sviluppo
- **Database:** Sistema organizzato per raccogliere, conservare e gestire grandi quantità di dati, permettendone l'accesso, la modifica e la consultazione in modo efficiente e strutturato.
- **Frontend:** Parte visibile di un'applicazione o di un sito web, con cui l'utente interagisce direttamente; comprende l'interfaccia grafica e gli elementi che ne determinano l'aspetto e l'esperienza d'uso.
- **Backend:** Componente di un sistema informatico che gestisce la logica, l'elaborazione dei dati e la comunicazione con il database, rendendo disponibili al frontend le informazioni necessarie al funzionamento dell'applicazione.
- **Eco-score**: Punteggio attribuito all'utente a fronte di una certa condotta di guida, nell'osservazione o meno delle migliori pratiche orientate al risparmio dei consumi, alla riduzione dell'inquinamento. Il calcolo poggia su tecniche statistiche apposite
- **Copilot:** Assistente di programmazione basato sull'intelligenza artificiale che suggerisce in tempo reale frammenti di codice, funzioni o soluzioni, aiutando lo sviluppatore a scrivere software in modo più rapido ed efficiente.
- **GitHub Issues:** Strumento integrato in GitHub che consente di segnalare, discutere e monitorare problemi, richieste di funzionalità o attività di sviluppo all'interno di un progetto software collaborativo.
- **PostgreSQL (Postgres):** Sistema di gestione di database relazionali open source, noto per la sua affidabilità, estendibilità e conformità agli standard SQL, utilizzato per archiviare e gestire dati strutturati.

- **Prisma:** ORM (Object-Relational Mapping) moderno che facilita l'interazione tra un'applicazione e il database, offrendo un'interfaccia tipizzata e intuitiva per creare, leggere, aggiornare e cancellare dati in modo sicuro ed efficiente.
- **Flutter:** Framework open source sviluppato da Google che consente di creare applicazioni multipiattaforma (mobile, web e desktop) partendo da un unico codice sorgente, utilizzando il linguaggio Dart.
- **Angular:** Framework open source basato su TypeScript, sviluppato da Google, che permette di costruire applicazioni web dinamiche e strutturate, grazie a un'architettura modulare e a un solido sistema di componenti.

3. Tecniche di testing

3.1. Strategia principale

La strategia principale adottata dal team si fonda su un processo di verifica continuo e collaborativo, che combina tecniche complementari per garantire una copertura completa del sistema sia sul piano funzionale sia strutturale. Ogni fase dello sviluppo è accompagnata da un confronto costante tra sviluppatori e revisori, da un'analisi mirata del codice e da test periodici che vengono aggiornati all'evolvere della codebase.

L'approccio integra metodologie Black-box e White-box, insieme a revisioni formali e informali del codice, così da intercettare sia difetti logici sia anomalie comportamentali. La comunicazione interna svolge un ruolo centrale: feedback rapidi, documentazione condivisa e cicli di verifica ravvicinati permettono di individuare precocemente i problemi, ridurre il rischio di regressioni e consolidare un'applicazione robusta e affidabile nel tempo.

3.2. Test di regressione

Il test di regressione è quella fase dello sviluppo dove, a seguito di un aggiornamento o miglioria della codebase, è necessario verificare la stessa non sia stata pregiudicata dall'introduzione di nuove complicazioni o, peggio, dall'esasperazione di criticità già presenti, magari nemmeno rilevate in precedenza.

A tal fine, oltre l'applicazione delle tecniche di verifica, fortemente sfruttati sono i processi automatizzati. L'applicazione si svolge a livello del singolo e/o di gruppi ridotti di moduli, per testimoniare la corretta interoperabilità, assieme a pipeline predisposte al deploy dell'applicativo in chunk più grossi; l'infrastruttura docker è un esempio forte, poggiante oltretutto sul sistema di notifica e di log ad ogni step della build and run. Il test di regressione strutturato similmente significa un applicativo solido a livello microscopico e macroscopico.

3.3. Tecniche di verifica

L'approccio scelto dal team tutto consiste in una sovrapposizione, a coppie, di diverse tecniche di testing e di analisi mirata del software. Questa strategia così ricca è pensata per sfruttare i pregi di tutte le tecniche che, spesso, soddisfano dove le altre lasciano spazio a errori o a mancanza di convalidazione di integrità del codice. Il fine ultimo delle analisi non si ferma ad assicurare solamente il corretto funzionamento dell'applicativo, ma accertano la soddisfazione da parte dei suoi utilizzatori, cioè i comuni e gli utenti. Il testing vuole essere una procedura da reiterare nel corso del tempo, specie a braccetto con upgrade delle funzionalità o con qualsivoglia modifica minimamente strutturale dell'applicativo.

Per quanto concerne le coppie di tecniche sopra citate, consideriamo quelle *Black-box* e *White-box* la prima, *Introspection* e *Walkthrough* la seconda.

- **Black-box e White-box Testing**: La strategia B*lack-box* è fatta per testare il funzionamento dell'applicativo e di sottoparti sulla base del successo o meno nel fornire correttamente servizi. Non richiedendo specifiche conoscenze circa i meccanismi sottostanti le funzionalità offerte, tutto il team è coinvolto nei periodici test, basati su array di input e su output. Particolare attenzione cattura la Boundary Value Analysis, mentre il Pairwise testing procede con meno attenzione ma con senno.
  Dal lato *White-box* invece è centrale l'osservazione critica del codice sorgente assieme alla copertura sistematica dei test case, quindi verificando la correttezza di esecuzione al variare dei path intrapresi (rami if-else) nel rispetto delle precondizioni e delle postcondizioni (copertura delle decisioni e dei valori critici). In senso lato questo approccio va a beneficio del Black-box testing, seppure del White-box siano responsabili un sottoinsieme del team, in particolare i membri responsabili della stesura del codice.

- **Walkthrough e Inspection**: Entrambe le procedure hanno in comune il fatto di procedere pressoché con gli stessi mezzi: la supervisione dei membri più indicati della codebase. La differenza fondamentale sta nella vicinanza effettiva al codice poi scritto, quanto astrattamente sia concepito l'oggetto della nostra valutazione. Mentre l'introspection ruota attorno all'error guessing, circa quindi complicanze intrinseche dei linguaggi, dei framework con cui lavoriamo e che sono ipotizzabili, il walkthrough, più lento, consiste nella lettura ponderata del codice sorgente finché gli addetti a revisione non siano pienamente soddisfatti.
  I gruppi per simili ruoli sono generalmente misti e non potrebbero che richiedere una preparazione solida sul campo (conoscenza del framework, dimestichezza con le convenzioni di documentazione…)

4. Criteri di rilevazione dei malfunzionamenti

Come evidenziato nella letteratura di riferimento (Dijkstra, Goodenough & Gerhart), non esiste un singolo criterio di test che sia contemporaneamente totalmente affidabile e totalmente valido per programmi non banali . Pertanto, la strategia di Green Drive prevede l'utilizzo combinato di criteri complementari per massimizzare l'efficacia del testing.

4.1. Criteri affidabili

Per garantire l'affidabilità, utilizzeremo il Partizionamento in Classi di Equivalenza (Equivalence Partitioning). Questo criterio è stato scelto perché permette di raggruppare i dati di input (come i valori letti dal sensore OBD-II o i campi dei form di registrazione) in classi omogenee, riducendo la ridondanza dei test pur mantenendo una copertura logica completa .

- **Input OBD (RF-03):** Se un sensore invia i Giri Motore (RPM), divideremo i valori in classi valide (es. 0 < RPM < 8000) e classi non valide (es. RPM < 0 o RPM > 15000). Testare un valore qualsiasi all'interno della classe valida (es. 3000 RPM) è considerato affidabile e rappresentativo per l'intera classe.
- **Autenticazione (RF-01):** Le credenziali verranno divise in classi: "Utente registrato", "Utente non registrato", "Campi vuoti".

4.2. Criteri validi

Per massimizzare la validità (efficacia nello scovare bug), utilizzeremo una combinazione di approcci Black-box e White-box.

1. **Analisi dei Valori Limite (Boundary Value Analysis - BVA) [Black-box]** Poiché la maggior parte degli errori si verifica ai margini dei domini di input , questo criterio è fondamentale per il calcolo dell'Eco-Score (RF-03).

   Applicazione: Verificheremo il comportamento dell'algoritmo esattamente sulle soglie di cambio punteggio. Ad esempio, se una guida è considerata "virtuosa" sotto i 2000 RPM, testeremo input a 1999, 2000 e 2001 RPM per assicurarci che la logica di > o >= sia corretta.

2. **Copertura delle Decisioni (Decision Coverage) [White-box]** Per la logica interna, specialmente nella gestione della connessione OBD e sincronizzazione dati, useremo la copertura delle decisioni.

   Applicazione: Ogni struttura di controllo nel codice (es. if (connessione_persa)) dovrà essere testata sia nel ramo TRUE (connessione persa -> tentativo riconnessione come da RNF-03) sia nel ramo FALSE (connessione stabile). Questo garantisce che nessun ramo logico critico rimanga inesplorato, aumentando la validità del test strutturale.

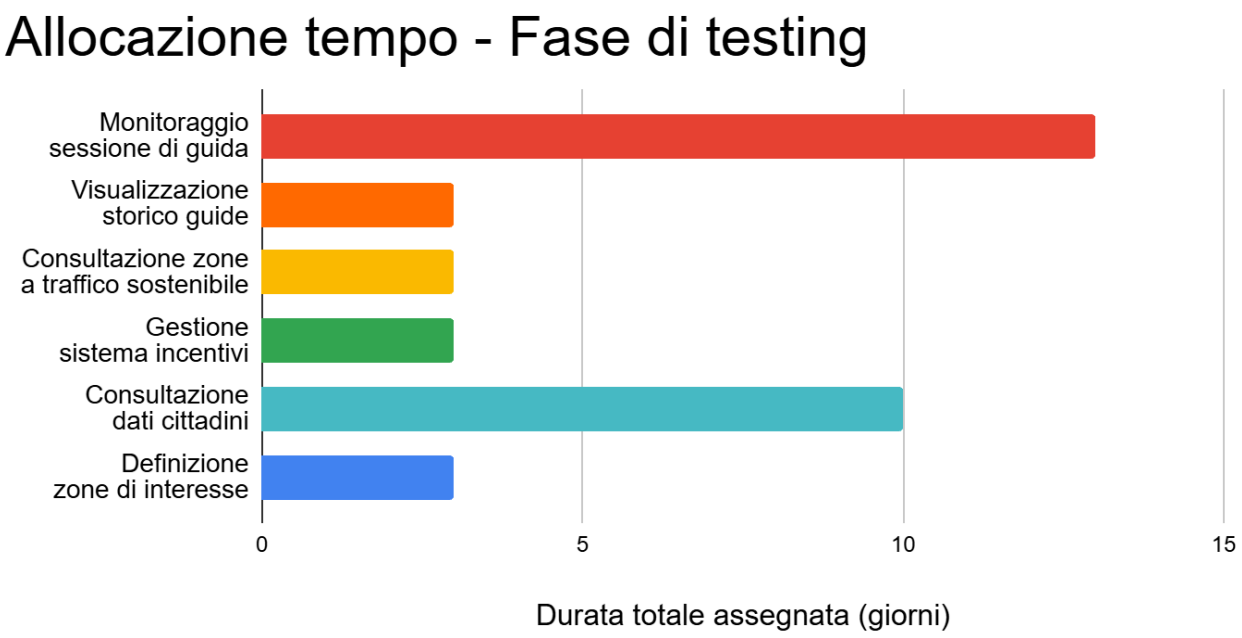
5. Modalità di verifica dei singoli requisiti

- **RF-03 (Monitoraggio sessione di guida)**
  - Verrà verificato che l'algoritmo per accumulare le rilevazioni produca approssimazioni il cui errore rientri in un intervallo del 10% rispetto al calcolo finale effettuato dal server
  - Verranno confrontati i dati prodotti con quelli visualizzati dall'interfaccia grafica, per stabilire se siano presenti errori di conversione e se elementi come il tachimetro visualizzino la velocità correttamente
- **RF-04 (Visualizzazione storico guide)**
  - Si esaminerà il punteggio complessivo con i punteggi cumulati dalle singole rilevazioni per stabilire se rientrano nella soglia minima del calcolo finale
- **RF-05 (Consultazione zone a traffico sostenibile)**
  - Si controllerà la corrispondenza tra le coordinate inserite dai singoli comuni con la relativa rappresentazione sulla mappa
- **RF-08 (Definizione zone di interesse)**
  - Verrà controllato che le zone di interesse siano delimitate dai contorni di un poligono
  - Seguiranno dei test di partizionamento per verificare se un particolare percorso attraversa la zona a traffico sostenibile
- **RF-07 (Consultazione dati cittadini)**
  - Poichè il punteggio relativo alle zone di interesse verrà calcolato in maniera dinamica, si presterà maggiore attenzione al codice sorgente tramite una strategia white-box
- **RF-06 (Gestione sistema incentivi)**
  - Verrà utilizzato un approccio esaustivo per verificare che le combinazioni degli elementi selezionabili nel form non generino eccezioni durante l'elaborazione

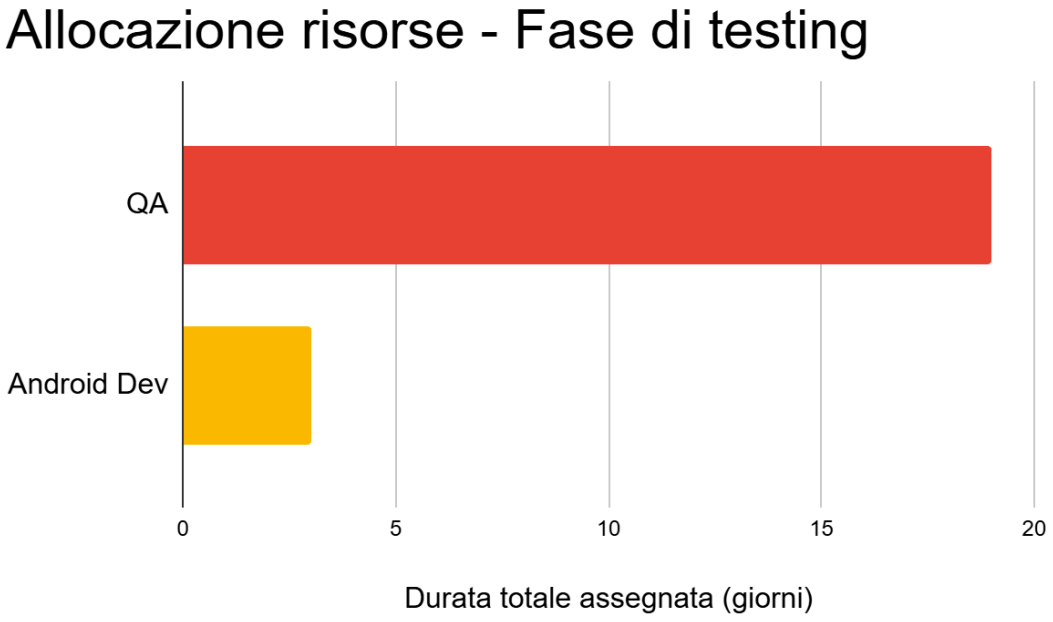
6. Pianificazione del testing

Il tempo e le risorse da dedicare ai singoli test vengono valutate rispetto alle scadenze definite nel piano di progetto, classificandole rispetto ad una delle sotto-attività del processo di testing.

6.1. Tempo allocato



6.2. Risorse allocate



7. Procedure di registrazione dei test

Il risultato dei vari test può essere classificato in due modalità:

Esito Positivo: durante il controllo sono emerse una o più criticità che richiedono attenzione o intervento.
Esito Negativo: il test non ha evidenziato problemi e il comportamento osservato è risultato conforme alle aspettative.

7.1. Monitoraggio dei test

Durante l'intera fase di testing verranno registrati con cura tutti gli esiti ottenuti. Ogni verifica effettuata contribuirà a costruire un quadro chiaro e aggiornato dello stato del sistema, permettendo al team di seguire l'evoluzione dei controlli in maniera precisa e ordinata.

7.2. Aggiornamento dei test

Al termine della fase di monitoraggio, è necessario aggiornare le tabelle dei casi di test e degli altri test, ovvero le tabelle presenti rispettivamente nel capitolo 4 e 8, con l'esito e le post-condizioni riscontrate. L'esito di ciascun test, che può essere positivo o negativo, dovrà essere riportato nelle rispettive voci delle tabelle dedicate.

7.3. Risoluzione dei problemi

Quando un test evidenzia una criticità, il team procede attraverso una serie di passaggi mirati, così da affrontare il problema in modo ordinato ed efficace:

1. **Condivisione dell'anomalia:** il team viene informato dell'errore rilevato, avviando un confronto sulle possibili origini e su come intervenire.
2. **Analisi della causa:** si approfondisce il problema esaminando con attenzione il codice, le logiche coinvolte o le specifiche collegate, per comprendere cosa lo abbia generato.
3. **Organizzazione degli interventi:** viene definito un piano di azione che stabilisce quali modifiche apportare e a chi affidare le attività.
4. **Applicazione delle correzioni:** si procede con la risoluzione vera e propria del problema, annotando le modifiche introdotte per mantenerne traccia.
5. **Verifica delle modifiche:** le correzioni vengono testate per assicurarsi che abbiano realmente eliminato l'anomalia senza crearne di nuove.
6. **Nuovo ciclo di monitoraggio:** la fase di monitoraggio viene ripetuta per confermare che il sistema si comporti correttamente dopo gli interventi.

8. Requisiti hardware e software

- **GitHub:** piattaforma per salvare, condividere e versionare il codice in modo collaborativo.
- **Discord:** ambiente di comunicazione online utile per coordinare il lavoro tramite chat e vocali.
- **WhatsApp:** strumento rapido di messaggistica per scambi veloci tra i membri del team.
- **VSCode:** editor di codice leggero e potente per sviluppare e gestire progetti software.
- **Flutter:** framework per creare applicazioni mobili, web e desktop da un unico codice sorgente.
- **Prisma:** ORM che semplifica la gestione e l'accesso ai dati nei database tramite codice tipizzato.
- **Postgres:** database relazionale affidabile usato per memorizzare e organizzare grandi quantità di dati.
- **Google Docs:** servizio per scrivere e modificare documenti online in modo condiviso.
- **Android Studio:** ambiente di sviluppo completo per progettare e testare app Android.
- **Angular:** framework per costruire applicazioni web strutturate basate su componenti.
- **Docker:** tecnologia che isola ed esegue applicazioni in contenitori portabili e riproducibili.
- **gRPC:** sistema di comunicazione ad alte prestazioni per far dialogare servizi diversi in rete.
- **Nginx:** server leggero e veloce per distribuire siti web e gestire traffico in entrata.

9. Vincoli che condizionano il testing

Tra i vincoli condizionanti seguono:

- **Problemi con l'infrastruttura**: La stragrande maggioranza dei test necessita di un'infrastruttura web predisposta correttamente. In particolare la configurazione corretta di un web server è indispensabile, non soltanto per attuare i test, ma anche per completezza dell'applicativo stesso (certificazione SSL, error logging, …). Il processo di configurazione è macchinoso e poco intuitivo, esponendo ad una complessità non indifferente; abbiamo un collo di bottiglia logistico, ovvero che può portare a ritardi importanti nella pianificazione di testing.

- **Vincoli di competenza**: L'insorgere di imprevisti quali strumenti applicativi o dettagliate condizioni di operatività del software sorti solo durante lo sviluppo può rappresentare una sfida delle proprie competenze. Normalmente l'approccio preferibile è rifarsi a documentazioni e a risorse telematiche che, giocoforza, occupano un tempo non indifferente sottratto al testing, con possibile rivalutazione della pianificazione.