Laurea Magistrale in informatica-Università di Salerno
Corso di *Gestione dei Progetti Software*- Prof.ssa F.Ferrucci

Object Design Document

RoadGuardian

Riferimento	C07_ODD
Versione	1.0.0
Data	25/11/2025
Destinatario	Docenti di Ingegneria del Software
Presentato da	C07-Angela Setola, Simone Domenico Avitabile, Mattia D'Auria, Raffaele Cimino, Giovanna Massa, Ciro Navarra, Lorenzo Olivola, Davide Pio Lazzarini, Sabato Iaquino, Carlo Mancusi

Revision History

Data	Versione	Descrizione	Autori
25/11/2025	0.0.1	Introduzione	Mattia D'Auria
25/11/2025	0.0.2	Aggiunta Object Design Goals e Trade-Offs	Ciro Navarra, Sabato Iaquino, Davide Pio Lazzarini, Angela Setola
25/11/2025	0.0.3	Aggiunta definizioni, acronimi e abbreviazioni	Carlo Mancusi
26/11/2025	0.0.4	Aggiunta Design Patterns e Linee Guida di Implementazione	Raffaele Cimino, Lorenzo Olivola, Giovanna Massa, Simone Domenico Avitabile
28/11/2025	0.0.5	Modifica di Design Pattern e Linee guida di Implementazione	Raffaele Cimino
29/11/2025	0.0.6	Inserimento Component Off-the-shelf	Davide Pio Lazzarini, Ciro Navarra, Sabato Iaquino, Angela Setola
29/11/2025	0.0.7	Rimozione Object Design Goals e Trade-Offs	Ciro Navarra, Sabato Iaquino, Davide Pio Lazzarini, Angela Setola
30/11/2025	0.0.8	Inserimento Design Pattern e linee guida di implementazione	Raffaele Cimino, Lorenzo Olivola, Simone Domenico

			Avitabile, Giovanna Massa
02/12/2025	0.0.9	Aggiunta riferimenti	Carlo Mancusi
09/12/2025	0.0.10	Modifica delle Linee Guida di Implementazione	Raffaele Cimino
11/12/2025	0.0.11	Modifica dei Component Off-the-shelf e revisione generale documento	Davide Pio Lazzarini, Lorenzo Olivola
12/12/2025	0.0.12	Revisione design patterns Facade adoperati	Raffaele Cimino, Lorenzo Olivola
14/12/2025	1.0.0	Revisione finale	Tutti i TM

Indice

1. Object Design Goals, Trade-Off e Linee Guida

1.1. Introduzione

Il presente Object Design Document (ODD) prosegue l'analisi e la progettazione del sistema RoadGuardian, basandosi sul Requirements Analysis Document (RAD) e sul System Design Document (SDD).

RoadGuardian è un'applicazione mobile sviluppata per:

- Consentire agli utenti l'invio di segnalazioni (manuali o veloci) e geolocalizzate.
- Fornire notifiche push agli automobilisti in prossimità di un incidente (entro 3 km).

1.2. Definizioni e Acronimi

Definizioni

Back-end	La parte del sistema che implementa la logica di business, gestisce l'accesso ai dati e coordina le operazioni sul database.
COTS (Component Off-theShelf)	Componenti software o hardware forniti da terze parti, pronti all'uso, integrabili in un sistema per ridurre tempi di sviluppo e complessità progettuale.
Design Patterns	Soluzioni progettuali generali e riutilizzabili a problemi ricorrenti nell'ingegneria del software, che forniscono uno schema concettuale per organizzare il codice.

Facade	Design Pattern strutturale che introduce un oggetto frontale che incapsula e coordina l'accesso a componenti complessi, fornendo un'API coerente e di alto livello per i client.
Front-end	Componente dell'applicazione responsabile della presentazione e dell'interazione con l'utente: realizza l'interfaccia utente, gestisce eventi e input, e coordina la visualizzazione e la validazione dei dati prima della comunicazione con il livello server.
Object Design Goals	Insieme di principi e linee guida che indirizzano la progettazione degli oggetti in un sistema software, con l'obiettivo di massimizzare modularità, riusabilità, manutenibilità e coesione, riducendo al contempo il coupling e il debito tecnico.

Acronimi

Acronimo	Definizione
ODD	Object Design Document
ODG	Object Design Goals
DP	Design Patterns
API	Application Programming Interface

COTS	Component Off-the-Shelf

1.3. Riferimenti

Documentazioni ed altro:

- 2025_C07_SOW;
- 2025_C07_RAD;
- 2025_C07_SDD;
- Slide del corso di Ingegneria del Software.

1.4. Component Off-the-Shelf

COTS che verranno utilizzati per il front-end sono:

- **Flutter**: Framework open-source per lo sviluppo di applicazioni mobile, web e desktop, utilizzato per creare interfacce utente moderne e dinamiche grazie alla sua architettura basata su Dart che permette facile implementazione di widget, APIs e al supporto integrato per la gestione dello stato e del rendering reattivo.
  - Versione: 3.38.0

- **API Open Street Map:** Insieme di interfacce di programmazione di applicazioni e software development kit che consentono agli sviluppatori di integrare le funzionalità e i dati di google maps nelle proprie applicazione web, mobile e backend
  - Versione: 0.7.62.4

- **FCM:** Permette agli sviluppatori di inviare messaggi (chiamati anche notifiche push) in modo affidabile da un server, o da un ambiente serverless come Cloud Functions, alle applicazioni client.
  - Versione: 25.0.1

- **JavaScript:** È un linguaggio di programmazione di alto livello, leggero e interpretato (o Just-In-Time compilato), noto principalmente come il linguaggio di scripting per il Web.
  - Versione: ES2025

COTS che verranno utilizzati per la back-end sono:

- **MongoDB – DBMS:** Database NoSQL orientato ai documenti, usato dall'applicazione perché è flessibile, intuitivo e sincronizza in tempo reale i dati; supporta scalabilità, replica e query flessibili. L'accesso da Python avviene tramite PyMongo.

- Versione: 1.48.2-1

- **PyMongo**: La distribuzione PyMongo contiene tools per interagire con il database MongoDB da Python. PyMongo è un driver nativo di Python per l'implementazione di MongoDB che offre API sincrone e asincrone.
  - Versione: 1.17.0-1

Gli strumenti di supporto sono:

- **dart format:** strumento di formattazione ufficiale del linguaggio Dart, usato da Flutter. Applica uno stile di formattazione coerente e non configurabile, il che elimina dibattiti sullo stile del codice.
  - Versione: 3.10.2

- **Black:** È un formattatore di codice "senza compromessi" . La sua filosofia è quella di essere non configurabile. Applica uno stile coerente e standard.
  - Versione: 25.11.0

- **Postman:** È un'applicazione utilizzata per testare le chiamate API lato server.
  - Versione: 11.74.5

1.5. Design Patterns

Facade

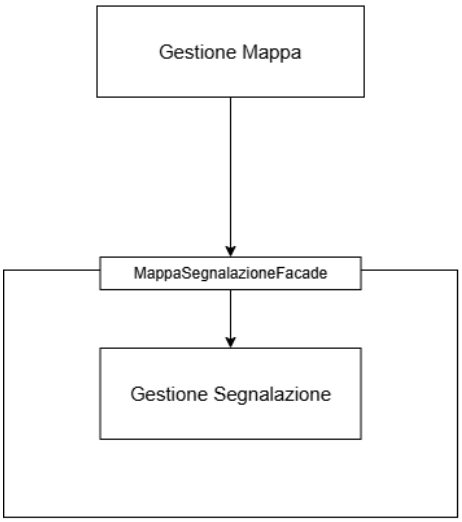
Il pattern **Facade** è uno dei pattern strutturali, il cui scopo principale è fornire una **singola interfaccia semplificata** per un insieme complesso di classi, sottosistemi, librerie o framework.
Meccanismo e struttura del design pattern:

- **Facciata (Facade):** È la singola classe che espone le funzionalità richieste dal cliente (client). Non implementa la logica del sottosistema, ma **delega** le richieste appropriate agli oggetti interni.
- **Sottosistema (Subsystem Classes):** Sono le classi complesse che svolgono il lavoro effettivo. Il cliente (client) non le conosce e non le utilizza direttamente; la Facciata le gestisce internamente.
- **Cliente (Client):** È l'oggetto che utilizza la Facciata. Interagisce solo con la Facciata, ignorando la complessità interna.

Useremo questo design pattern per costituire l'interfaccia **MappaSegnalazioneFacade** tra il sottosistema **Gestione Mappa** e **Gestione Segnalazione**. Questo permetterà una gestione facilitata delle interazione tra i sottosistemi in questione visto che vedranno soltanto ciò che è messo a disposizione dall'interfaccia riducendo quindi la probabilità che un cambiamento di un metodo che implementa un servizio offerto dall'interfaccia porti ad una modifica anche nel client.

L'uso di questo design pattern porterà:

- **maggiore indipendenza** tra le altre classi (basso accoppiamento):
    - le classi client (che usano la Facciata) non avranno più bisogno di conoscere o istanziare le numerose classi interne dei sottosistemi (Gestione Mappa, Gestione Segnalazione), ma si occuperanno soltanto di richiedere il servizio che necessitano all'interfaccia.
- **alta manutenibilità**:
    - La Facciata introduce un punto di controllo centralizzato per tutte le interazioni verso il sottosistema, ciò permette di individuare rapidamente un possibile bug, inoltre la logica complessa di orchestrazione (quindi l'esecuzione di più passaggi tra diversi sottosistemi per ottenere un risultato) è incapsulata all'interno dei metodi della Facciata.
- **facilità di modifica** nelle classi di implementazione:
    - se ad esempio l'implementazione del sottosistema Gestione Segnalazione cambia, solo la Facciata deve essere aggiornata. Gestione Mappa (client) rimarrà invariato. Ciò riduce drasticamente l'impatto delle modifiche.
- **minor efficienza** nello scambio di informazioni tra i sottosistemi:
    - sebbene i benefici strutturali siano molti, l'introduzione di un livello di astrazione aggiuntivo comporta un leggero costo in termini di prestazioni e, a volte, di flessibilità, infatti il client è costretto a utilizzare solo le operazioni predefinite e semplificate esposte dalla Facciata.



Adapter

L'**Adapter** è un pattern strutturale che permette a due interfacce incompatibili di collaborare tra loro. L'Adapter agisce come un "traduttore" che converte l'interfaccia di una classe in un'altra interfaccia prevista dal cliente (client). Questo pattern è fondamentale quando si integrano componenti di terze parti o **COTS,** poiché questi sistemi raramente espongono un'interfaccia che combacia perfettamente con le esigenze dell'applicazione ospite.

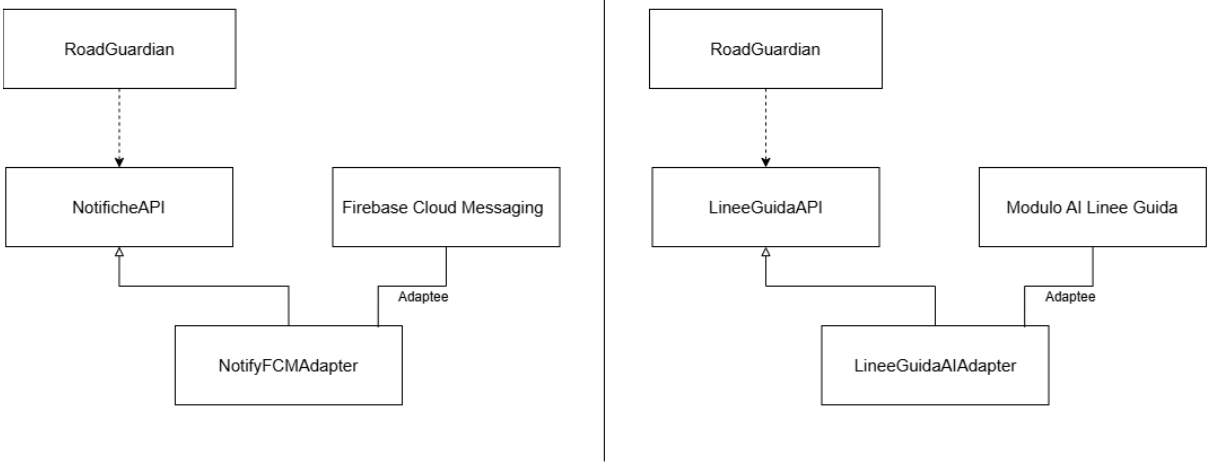
Meccanismo e struttura del design pattern:

- **Target (Interfaccia Desiderata):** L'interfaccia che il client utilizza e conosce.
- **Adaptee (Classe Incompatibile):** La classe esterna o COTS (come FCM o Modulo AI) con l'interfaccia che deve essere adattata.
- **Adapter (Adattatore):** La classe che implementa l'interfaccia Target e al suo interno contiene (o eredita) un oggetto Adaptee. L'Adapter riceve le chiamate dal client e le traduce in chiamate compatibili per l'Adaptee

Per la nostra applicazione il design pattern verrà usato come traduttore con Firebase Cloud Messaging (FCM) e verrà usato anche per far dialogare l'applicazione RoadGuardian con il Modulo Linee Guida AI (il modulo AI e relativo adapter verranno implementati nel prossimo incremento previsto per il progetto di FIA).

L'uso di questo design pattern porterà:

- un'attività di testing semplificata (grazie al principio di **separazione delle responsabilità**.):
    - **Isolamento del COTS:** L'Adapter incapsula l'interazione diretta con il COTS (FCM o Modulo AI). Quando si testa la logica di **RoadGuardian**, si interagisce solo con l'interfaccia **Target** (quella attesa dall'applicazione).
    - **Mocking Facile:** Durante i test unitari di RoadGuardian, non è necessario connettersi realmente a Firebase o al Modulo AI. È sufficiente creare un **Mock** o uno **Stub** dell'interfaccia **Target** che l'Adapter implementa. In questo modo si verifica che l'applicazione chiami correttamente i metodi dell'interfaccia, senza doversi preoccupare della complessa configurazione e dipendenza esterna del COTS.
    - Il **codice COTS viene testato separatamente**, garantendo che le integrazioni siano testate in modo mirato e isolato.
- **separazione tra il sistema e le componenti esterne** (COTS) minimizzando l'impatto dei nuovi software sul nostro progetto:
    - **Interfaccia Consistente:** L'applicazione RoadGuardian vede un'unica interfaccia (**NotificheAPI**). L'Adapter, **NotifyFCMAdapter**, si occupa di tradurre la chiamata da parte del client attraverso NotificheAPI, nel formato specifico richiesto dalla libreria FCM (es. FCMClient.sendPayload(messaggio, token)).
    - **Minimizzazione dell'Impatto:** Se in futuro si decidesse di sostituire FCM con un altro servizio di messaggistica, non si dovrà modificare l'intero codice per le notifiche, ma semplicemente creare un nuovo Adapter che implementi la stessa interfaccia Target (NotificheAPI) riducendo l'impatto sul progetto al minimo.
- **maggiore complessità del sistema**: Per l'introduzione di classi Adapter che aggiungono inevitabilmente un livello di complessità strutturale maggiore:
    - **Aumento del Contenuto del Codice:** Per ogni componente esterno che si integra, è necessario creare almeno una nuova classe: l'**Adapter**. Questo aumenta il numero totale di file, classi e relazioni nel progetto, rendendolo più grande e potenzialmente più difficile da navigare per i nuovi sviluppatori.
    - **Indirezione:** Ogni chiamata funzionale deve passare attraverso un intermediario (l'Adapter). Questo crea un livello di **indirezione** che non solo introduce un lieve **overhead** di performance (solitamente trascurabile), ma può anche rendere il **debugging** più oneroso, poiché si passa per la classe Adapter.



1.6 Linee Guida di Implementazione

Questa sezione contiene delle convenzioni da seguire per l'implementazione e la documentazione dell'applicazione, per aiutare gli sviluppatori a garantire coerenza, chiarezza e facilità di manutenzione degli artefatti.

1. Convenzioni di identificazione

- **Chiarezza e Coerenza:** Adottare nomi descrittivi e univoci per interfacce e metodi.
- **Pattern Standard:** Utilizzare il formato **verbo_oggetto** (es. **get_posizione_GPS()**) per garantire l'immediata comprensione dell'azione svolta.
- **Leggibilità:** Evitare abbreviazioni ambigue privilegiando sempre l'espressività del nome.

2. Standard per la descrizione tramite commenti Ogni classe deve avere una descrizione delle funzionalità offerte dalla stessa di massimo 150 caratteri, inoltre ogni metodo deve essere corredato da descrizione esaustiva che includa:

- **Scopo:** Descrizione sintetica ma completa della funzionalità offerte dal metodo.
- **Parametri:** Il tipo e il significato di ciascun parametro di input.
- **Valore di ritorno:** Il tipo di dato restituito e la semantica del valore (cosa rappresenta).
- **Eccezioni:** Elenco delle eccezioni potenzialmente sollevate durante l'esecuzione.

3. Specifiche di Input/Output Definizione chiara relativa a:

- **Tipizzazione e Formato:** Specificare i tipi di dato e formati di interscambio di dati (es. String, Integer, JSON, ecc), i pattern richiesti dai dati trattati e i limiti di lunghezza.
- **Vincoli:** Esplicitare validazioni obbligatorie, range di valori ammessi.

4. Gestione delle Eccezioni

- **Individuazione:** Identificare chiaramente le eccezioni previste per ogni operazione.