Guida all'Architettura e al Codice Sorgente del Radar Target Simulator

Questo documento descrive l'architettura interna, le decisioni di progettazione e la struttura del codice sorgente dell'applicazione Radar Target Simulator. È destinato agli sviluppatori e agli ingegneri che necessitano di comprendere, manutenere o estendere il software.

1. Filosofia del Progetto e Principi Architetturali

L'architettura del software si basa su alcuni principi chiave volti a garantire manutenibilità, estendibilità e robustezza.

1.1. Separation of Concerns (SoC)

Il principio fondamentale è la netta separazione delle responsabilità tra i vari componenti del sistema:

- **Interfaccia Utente (GUI):** Responsabile solo della presentazione dei dati e della cattura dell'input utente. Non contiene logica di business.
- **Logica di Simulazione (Core/Engine):** Responsabile del calcolo della cinematica dei target. È completamente agnostica rispetto a come i dati vengono visualizzati o trasmessi.
- **Comunicazione (Communicators):** Responsabile della traduzione dei comandi di alto livello in messaggi specifici di un protocollo e della gestione della comunicazione a basso livello.

Questa separazione permette, ad esempio, di sostituire l'interfaccia basata su Tkinter con una basata su web senza modificare il motore di simulazione, o di aggiungere un nuovo protocollo di comunicazione senza toccare la GUI.

1.2. Thread Safety e Comunicazione tra Thread

L'applicazione è multi-threaded per garantire che l'interfaccia utente rimanga reattiva durante le operazioni potenzialmente lunghe (come la simulazione o l'attesa di pacchetti di rete).

- **GUI Thread (Main Thread):** Esegue il loop di Tkinter e gestisce tutti i widget.
- **Simulation Thread:** Esegue il loop del `SimulationEngine` per aggiornare lo stato dei target a intervalli regolari.
- **Communication Thread(s):** Componenti come `SfpTransport` avviano i propri thread per l'ascolto non bloccante di pacchetti UDP.

Per garantire una comunicazione sicura tra questi thread, l'architettura si affida a due meccanismi principali:

1. **`SimulationStateHub`:** Un contenitore dati centralizzato e thread-safe (protetto da `threading.Lock`). Funge da "single source of truth" per lo stato della simulazione. Tutti i thread scrivono e leggono da questo hub, che agisce come un intermediario sicuro, disaccoppiando i produttori di dati (motore di simulazione, ricevitore di rete) dai consumatori (GUI).
2. **`queue.Queue`:** Utilizzata per inviare notifiche o dati che richiedono un'elaborazione immediata dalla GUI (es. aggiornamenti per finestre di debug o notifiche di fine simulazione).

1.3. Modularità e Astrazione

Per facilitare l'estensione, il codice fa uso di astrazioni e interfacce. L'esempio più significativo è l'interfaccia `CommunicatorInterface`. Qualsiasi protocollo di comunicazione (Seriale, TFTP, SFP, o futuri) deve semplicemente implementare i metodi definiti da questa classe base (`connect`, `disconnect`, `send_scenario`, etc.). Il resto dell'applicazione interagisce solo con l'interfaccia, rendendo l'aggiunta di un nuovo protocollo un'operazione a basso impatto.

1.4. Standard e Convenzioni

Il codice aderisce alle seguenti convenzioni per garantire leggibilità e coerenza:

- **Stile del Codice:** Segue lo standard **PEP8**.
- **Linguaggio:** Nomi di funzioni, variabili, commenti e docstring sono scritti in **inglese**.
- **Tipizzazione:** Viene fatto uso di type hints (secondo PEP 484) per migliorare la robustezza e la comprensibilità del codice.

---

2. Struttura delle Cartelle del Progetto

Il codice sorgente è organizzato in un pacchetto Python (`target_simulator`) con una struttura modulare che riflette la separazione delle responsabilità.

```
├── doc/                    # Documentazione (manuali, guide come questa)
│   ├── manual/
│   └── user_guide/
├── scenarios/              # File JSON contenenti gli scenari predefiniti
├── target_simulator/       # Codice sorgente principale del pacchetto
│   ├── analysis/           # Moduli per l'analisi dati (es. PerformanceAnalyzer,
SimulationStateHub)
│   ├── core/               # Cuore della logica di business e simulazione
│   │   ├── models.py       # Definizioni dei dati (Target, Waypoint, Scenario)
│   │   ├── simulation_engine.py # Il motore di calcolo della simulazione
│   │   └── ... (moduli di comunicazione: sfp, serial, tftp)
│   ├── gui/                # Componenti dell'interfaccia utente (widget Tkinter)
│   │   ├── main_view.py    # Classe principale della finestra dell'applicazione
│   │   ├── ppi_display.py  # Widget per la visualizzazione radar
│   │   └── ... (altre finestre e widget personalizzati)
│   ├── simulation/         # Classi di alto livello per l'orchestrazione (es.
SimulationController)
│   ├── utils/              # Funzioni e classi di utilità trasversali
│   │   ├── config_manager.py # Gestione dei file di configurazione
│   │   ├── logger.py       # Configurazione del sistema di logging
│   │   └── ... (altre utilità)
│   ├── __init__.py         # Entry point del pacchetto
│   ├── __main__.py         # Entry point per l'esecuzione dell'applicazione
(`python -m target_simulator`)
│   └── config.py           # Configurazioni globali e costanti (es.
LOGGING_CONFIG)
├── README.md               # Readme del progetto
└── requirements.txt        # Dipendenze Python
```

3. Mappa dei Componenti Chiave e Flusso dei Dati

Comprendere come i dati fluiscono attraverso l'applicazione è fondamentale per capire il suo funzionamento. Questa sezione descrive le interazioni tra i componenti principali durante due operazioni chiave: l'**esecuzione di una simulazione** e la **ricezione di dati reali**.

3.1. Diagramma Architetturale e Flusso Dati

Il seguente diagramma illustra le relazioni e il flusso di informazioni tra i componenti chiave del sistema.

(Placeholder per l'immagine)

Descrizione dell'Immagine: `architettura_dettagliata.png` *(Questo è il diagramma descritto nel capitolo precedente del manuale tecnico, che possiamo riutilizzare e dettagliare qui. Mostra i blocchi* `MainView (GUI)`, `SimulationController`, `SimulationEngine`, `SimulationStateHub`, `CommunicatorManager` *e le loro interazioni).*

3.2. Flusso 1: Esecuzione di una Simulazione (Dati in Uscita)

Questo flusso descrive cosa accade dal momento in cui l'utente preme "Start Live" fino all'invio dei dati dei target.

1. **Azione Utente (GUI Thread):**
    - L'utente clicca il pulsante `Start Live` sulla `MainView`.
    - `MainView` delega la richiesta al `SimulationController`. Per evitare avvii multipli, imposta un flag (`_start_in_progress_main`) e aggiorna lo stato dei pulsanti.
2. **Orchestrazione (SimulationController - Nuovo Thread):**
    - Il `SimulationController` avvia un thread in background per non bloccare la GUI.
    - **Reset Radar:** Chiama il `CommunicatorManager` per inviare un comando di reset al device under test.
    - **Cattura Origine:** Legge lo stato corrente dell'ownship dallo `SimulationStateHub` e lo salva come "origine della simulazione" fissa chiamando `set_simulation_origin()`.
    - **Invio Scenario:** Chiama il `CommunicatorManager` per inviare lo stato iniziale di tutti i target.
    - **Avvio Motore:** Se i passaggi precedenti hanno successo, crea e avvia un'istanza del `SimulationEngine` in un nuovo thread.
3. **Ciclo di Simulazione (SimulationEngine - Thread di Simulazione):**
    - Il `SimulationEngine` entra nel suo loop principale, che si ripete a una frequenza fissa (es. 20 Hz).
    - Ad ogni "tick": a. Calcola il `delta_time` dall'ultimo tick. b. Chiama `scenario.update_state(delta_time)` per aggiornare la posizione di ogni target. c. Scrive il nuovo stato `(timestamp, x, y, z)` di ogni target nello `SimulationStateHub` chiamando `add_simulated_state()`.
    - A intervalli più lenti (es. 1 Hz), definiti dall'utente: a. Prepara una lista di comandi di aggiornamento (es. `tgtset` o payload JSON). b. Invia i comandi tramite l'interfaccia del `CommunicatorInterface`.

4. **Visualizzazione (GUI Thread):**
    - Il `_gui_refresh_loop` di `MainView` si esegue periodicamente (es. ogni 40 ms).
    - Chiama `build_display_data()` (in `ppi_adapter.py`).
    - `build_display_data()` legge dallo `SimulationStateHub`:
        - Gli stati più recenti dei target simulati.
        - L'origine della simulazione.
        - Lo stato corrente dell'ownship.
    - Esegue la trasformazione di coordinate (rotazione + traslazione) per calcolare la posizione dei target simulati relativa all'ownship corrente.
    - `MainView` passa i dati trasformati al `PPIDisplay`, che aggiorna il canvas.

3.3. Flusso 2: Ricezione di Dati Reali (Dati in Ingresso)

Questo flusso descrive come i dati inviati dal radar vengono ricevuti, processati e visualizzati.

1. **Ricezione di Rete (Thread di Comunicazione):**
    - Il `SfpTransport` (o un altro componente di comunicazione) ha un thread in background in ascolto su una porta UDP.
    - Quando riceve un pacchetto, lo processa (es. riassembla i frammenti SFP).
2. **Routing del Payload (Thread di Comunicazione):**
    - Una volta che un payload completo è disponibile, il `SfpTransport` lo passa al `DebugPayloadRouter` in base al suo Flow ID.
    - Il `DebugPayloadRouter` ispeziona il payload. Se è un messaggio di stato del radar (es. Flow ID 'R'): a. Decodifica il payload binario in una struttura dati (`SfpRisStatusPayload`). b. Estrae i dati dell'ownship (posizione, heading, etc.) e dei target reali.
3. **Aggiornamento dello Stato (Thread di Comunicazione):**
    - Il `DebugPayloadRouter` aggiorna lo `SimulationStateHub` (che è thread-safe):
        - Chiama `set_ownship_state()` con i nuovi dati di navigazione.
        - Per ogni target reale ricevuto, chiama `add_real_state()` con la sua posizione.
        - Chiama `set_antenna_azimuth()` per aggiornare la posizione dell'antenna.
4. **Visualizzazione (GUI Thread):**
    - Il `_gui_refresh_loop` di `MainView`, al suo ciclo successivo, legge i dati "reali" aggiornati dallo `SimulationStateHub`.
    - `build_display_data()` calcola la posizione dei target reali relativa all'ownship.
    - `MainView` aggiorna il `PPIDisplay`, che disegna i target reali (rossi), e aggiorna la posizione e l'orientamento dell'ownship e dell'antenna.

Questo doppio flusso di dati, che converge nello `SimulationStateHub` e viene poi letto dalla GUI, è il cuore del funzionamento in tempo reale dell'applicazione.

4. Analisi Dettagliata dei Moduli Principali

Questa sezione fornisce una scomposizione delle responsabilità e delle interazioni per i moduli e le classi più importanti del progetto.

4.1. Entry Point e Orchestrazione GUI (`__main__.py`, `gui/main_view.py`)

- **`target_simulator/__main__.py`**
  - **Responsabilità:** È l'entry point principale dell'applicazione. Il suo unico scopo è inizializzare il sistema di logging di base e creare un'istanza della classe `MainView`, per poi avviare il loop principale di Tkinter (`app.mainloop()`).
- **`target_simulator/gui/main_view.py` -> `MainView(tk.Tk)`**
  - **Responsabilità:** È la classe "Dio" della GUI, ma con responsabilità ben definite. Orchestra tutti i componenti principali, costruisce la finestra e gestisce il ciclo di aggiornamento dell'interfaccia.
  - **Input/Stato Interno:**
    - Possiede le istanze uniche di `ConfigManager`, `SimulationStateHub`, `CommunicatorManager` e `SimulationController`.
    - Mantiene lo `Scenario` corrente in memoria.
  - **Output/Side-effect:**
    - Disegna e aggiorna tutti i widget della GUI.
    - Delega le azioni complesse (start/stop) al `SimulationController`.
    - Salva le configurazioni tramite il `ConfigManager`.
  - **Interazioni Principali:**
    - **`_gui_refresh_loop`:** È il cuore pulsante della GUI. A intervalli regolari, legge i dati dallo `SimulationStateHub` tramite il `ppi_adapter` e aggiorna il `PPIDisplay` e altri widget.
    - **Callback degli Eventi:** I metodi `_on_*` (es. `_on_start_simulation`) rispondono agli input dell'utente e attivano la logica appropriata, solitamente delegando al controller.

4.2. Logica di Simulazione (`simulation/`, `core/simulation_engine.py`)

- **`target_simulator/simulation/simulation_controller.py` -> `SimulationController`**
  - **Responsabilità:** Agisce come un "regista" per le operazioni di simulazione. Gestisce la logica di alto livello per l'avvio e l'arresto, che coinvolge più componenti e deve avvenire in una sequenza precisa.
  - **Input:** Riceve richieste da `MainView`.
  - **Output/Side-effect:** Crea e distrugge istanze del `SimulationEngine`.
  - **Interazioni Principali:**
    - Usa il `CommunicatorManager` per inviare comandi di controllo (reset, invio scenario).
    - Usa lo `SimulationStateHub` per impostare l'origine della simulazione.
- **`target_simulator/core/simulation_engine.py` -> `SimulationEngine(threading.Thread)`**
  - **Responsabilità:** Eseguire i calcoli cinematici della simulazione in un thread separato per non bloccare la GUI. È il "motore" che fa avanzare il tempo per i target.
  - **Input:** Un oggetto `Scenario` da simulare e un `CommunicatorInterface` per inviare i dati.
  - **Output/Side-effect:** Scrive continuamente gli stati aggiornati dei target nello `SimulationStateHub` e invia comandi al `CommunicatorInterface`.

    - **Interazioni Principali:**
        - Nel suo `run()` loop:
            1. Chiama `target.update_state()` per ogni target.
            2. Chiama `simulation_hub.add_simulated_state()`.
            3. Chiama `communicator.send_commands()`.

4.3. Modelli dei Dati (`core/models.py`)

- `target_simulator/core/models.py`
    - **Responsabilità:** Definisce le strutture dati fondamentali dell'applicazione: `Scenario`, `Target`, `Waypoint`. Queste classi contengono non solo i dati, ma anche la logica per manipolarli.
    - `Target` **class:** Contiene la logica per calcolare la propria posizione (`update_state`) e per generare il percorso completo a partire dai waypoint (`_generate_path`). È il componente che implementa la cinematica.
    - `Scenario` **class:** È un contenitore di `Target` con metodi helper per gestire l'intero gruppo (es. `update_state` su tutti i target).
    - `Waypoint` **dataclass:** Una semplice struttura dati per descrivere una manovra.

4.4. Hub dei Dati (`analysis/simulation_state_hub.py`)

- `target_simulator/analysis/simulation_state_hub.py` -> `SimulationStateHub`
    - **Responsabilità:** Essere il contenitore dati centrale e thread-safe. Disaccoppia i produttori di dati (engine, comunicatori) dai consumatori (GUI). Mantiene una cronologia limitata degli stati per l'analisi e la visualizzazione delle tracce.
    - **Input:** Riceve dati tramite i suoi metodi `add_*_state()` e `set_*_state()`.
    - **Output:** Fornisce dati tramite i suoi metodi `get_*()`.
    - **Interazioni Principali:** È il componente più connesso. Praticamente tutti gli altri moduli principali interagiscono con esso. La sua natura thread-safe (tutti i metodi pubblici usano un `threading.Lock`) è cruciale per la stabilità dell'applicazione.

4.5. Comunicazione (`core/communicator_manager.py`, `core/*_communicator.py`)

- `target_simulator/core/communicator_interface.py` -> `CommunicatorInterface`
    - **Responsabilità:** Definire il contratto (interfaccia astratta) che tutti i moduli di comunicazione devono rispettare.
- `target_simulator/core/communicator_manager.py` -> `CommunicatorManager`
    - **Responsabilità:** Funge da "fabbrica" e facciata per i communicator. Crea l'istanza del communicator corretto (SFP, TFTP, etc.) in base alla configurazione e inoltra le chiamate (es. `connect`, `send_scenario`).
    - **Interazioni Principali:** Viene usato dal `SimulationController` per gestire le connessioni e inviare comandi di controllo.
- `SFPCommunicator`, `TFTPCommunicator`, `SerialCommunicator`
    - **Responsabilità:** Ognuna di queste classi implementa la logica specifica per un protocollo. `SFPCommunicator`, ad esempio, si affida a `SfpTransport` per la gestione a basso livello dei pacchetti UDP e dei frammenti.

4.6. Visualizzazione (`gui/ppi_display.py`, `gui/ppi_adapter.py`)

- `target_simulator/gui/ppi_display.py` -> `PPIDisplay`
  - **Responsabilità:** È un widget complesso e auto-contenuto che si occupa esclusivamente di disegnare lo scenario radar su un canvas Matplotlib. Gestisce la grafica di target, tracce, ownship, settore di scansione e antenna.
  - **Input:** Riceve liste di oggetti `Target` (già in coordinate relative) dai suoi metodi `update_simulated_targets` e `update_real_targets`. Riceve l'heading dell'ownship per orientare la vista.
  - **Output/Side-effect:** Aggiorna il canvas.
- `target_simulator/gui/ppi_adapter.py` -> `build_display_data`
  - **Responsabilità:** Funzione helper "pura" che funge da adattatore. Il suo unico scopo è leggere i dati grezzi dallo `SimulationStateHub` ed eseguire le trasformazioni di coordinate necessarie per preparare i dati nel formato atteso dal `PPIDisplay`.
  - **Input:** L'istanza dello `SimulationStateHub`.
  - **Output:** Un dizionario contenente le liste di `Target` pronti per essere disegnati.

4.7. Utilità (`utils/`)

- `target_simulator/utils/config_manager.py` -> `ConfigManager`
  - **Responsabilità:** Abstrae la lettura e la scrittura dei file di configurazione (`settings.json`) e degli scenari (`scenarios.json`). Gestisce il path dei file e la serializzazione/deserializzazione da/verso JSON.
- `target_simulator/utils/logger.py`
  - **Responsabilità:** Configura il sistema di logging centralizzato basato su `Queue`, permettendo ai thread in background di inviare log in modo sicuro che verranno poi scritti sulla console o sul widget della GUI dal thread principale.

5. Guida all'Estensione del Progetto

Questa sezione fornisce esempi pratici su come estendere l'applicazione con nuove funzionalità, seguendo i principi architetturali stabiliti.

5.1. Caso d'Uso: Aggiungere un Nuovo Tipo di Manovra

Supponiamo di voler aggiungere una nuova manovra, ad esempio "Orbita attorno a un punto".

1. **Modificare i Modelli (`core/models.py`):**
   - Aggiungere un nuovo valore all'enum `ManeuverType`:

     ```
     class ManeuverType(Enum):
         # ...
         ORBIT_POINT = "Orbit Point"
     ```

   - Aggiungere i campi necessari alla `dataclass Waypoint` per supportare la nuova manovra (es. `orbit_center_x`, `orbit_center_y`, `orbit_radius_nm`, `orbit_direction`).

2. **Aggiornare la Logica di Calcolo (`core/models.py`):**
    - Modificare il metodo statico `Target.generate_path_from_waypoints` per gestire il nuovo `ManeuverType.ORBIT_POINT`. Qui andrà implementata la logica matematica per generare i punti `(t, x, y, z)` che descrivono l'orbita.
3. **Aggiornare l'Interfaccia Utente (`gui/waypoint_editor_window.py`):**
    - Nella classe `WaypointEditorWindow`, creare un nuovo `ttk.Frame` contenente i widget (Spinbox, Combobox, etc.) per inserire i parametri della nuova manovra (raggio, centro, etc.).
    - Modificare il metodo `_on_maneuver_type_change` per mostrare questo nuovo frame quando l'utente seleziona "Orbit Point" dal menu a tendina.
    - Aggiornare il metodo `_on_ok` per leggere i valori dai nuovi widget e popolare correttamente l'oggetto `Waypoint` quando si salva.

5.2. Caso d'Uso: Aggiungere un Nuovo Protocollo di Comunicazione

Supponiamo di voler aggiungere il supporto per un nuovo protocollo basato su TCP.

1. **Creare la Classe del Communicator (`core/tcp_communicator.py`):**
    - Creare un nuovo file `tcp_communicator.py` nel modulo `core`.
    - Definire una nuova classe `TCPCommunicator` che eredita da `CommunicatorInterface` (`core/communicator_interface.py`).
    - Implementare tutti i metodi astratti richiesti dall'interfaccia:
        - `connect(self, config)`: Logica per stabilire la connessione TCP.
        - `disconnect(self)`: Logica per chiudere la connessione.
        - `is_open(self)`: Proprietà che restituisce lo stato della connessione.
        - `send_scenario(self, scenario)`: Logica per serializzare e inviare lo stato iniziale dello scenario su TCP.
        - `send_commands(self, commands)`: Logica per inviare gli aggiornamenti in tempo reale.
        - `test_connection(config)` e `list_available_ports()`: Metodi statici.
2. **Aggiornare il Gestore (`core/communicator_manager.py`):**
    - Nella classe `CommunicatorManager`, modificare il metodo `_setup_communicator` per riconoscere il nuovo tipo "tcp".

```
# In _setup_communicator
from target_simulator.core.tcp_communicator import TCPCommunicator
# ...
elif comm_type == "tcp":
    communicator = TCPCommunicator()
    config_data = config.get("tcp", {})
```

3. **Aggiornare l'Interfaccia Utente (`gui/connection_settings_window.py`):**
    - Aggiungere "TCP" alla lista dei `values` nel `ttk.Combobox` del tipo di connessione.
    - Creare un nuovo `ttk.Frame` (es. `tcp_frame`) con i widget per i parametri TCP (IP, porta).

- Aggiungere una nuova scheda al `ttk.Notebook` per ospitare il `tcp_frame`.
- Aggiornare i metodi `_load_settings` e `_on_save` per leggere e scrivere la nuova sezione di configurazione `tcp` nel dizionario `connection_config`.
- Aggiungere la logica per `_test_connection` per il tipo "TCP".

5.3. Flusso di Debug Consigliato

Quando si affronta un bug o si sviluppa una nuova funzionalità, si consiglia il seguente approccio:

1. **Isolare il Problema:** Il bug è nella visualizzazione (GUI), nel calcolo (Engine) o nella comunicazione (Communicator)? La separazione delle responsabilità dovrebbe aiutare a identificare il dominio del problema.
2. **Aumentare il Livello di Log:** Usare la finestra `Debug -> Logger Levels...` per impostare a `DEBUG` il livello di log del modulo sospetto. Ad esempio:
   - Problemi di visualizzazione? Abilita il debug per `target_simulator.gui.ppi_display` e `target_simulator.gui.ppi_adapter`.
   - Problemi di comunicazione? Abilita il debug per `target_simulator.core.sfp_communicator` e `target_simulator.core.sfp_transport`.
3. **Usare gli Strumenti di Debug:**
   - L'`SFP Packet Inspector` è fondamentale per problemi di comunicazione. Controlla la scheda `Raw` per vedere esattamente cosa viene ricevuto e la scheda `RIS` per vedere come viene interpretato.
   - Usa il `Simple Target Sender` per inviare comandi isolati e testare specifiche risposte del sistema radar senza eseguire uno scenario completo.
4. **Ispezionare lo `SimulationStateHub`:** Se c'è una discrepanza tra ciò che il motore dovrebbe calcolare e ciò che la GUI mostra, il problema è probabilmente nel `ppi_adapter`. Se i dati nello `Hub` stesso sono sbagliati, il problema è a monte (nel `SimulationEngine` o nel `DebugPayloadRouter`).

Seguendo questi principi e utilizzando gli strumenti forniti, la manutenzione e l'estensione dell'applicazione dovrebbero risultare più semplici e strutturate.