# Revisione Strategica e Roadmap di Refactoring per l'Applicazione di Analisi di Certificati Finanziari

## Introduzione

Questo report presenta una revisione strategica e una roadmap per l'evoluzione dell'applicazione di analisi dei certificati finanziari. Lo stato attuale del progetto, caratterizzato da una crescita organica e da un rapido sviluppo di funzionalità, è una fase naturale nel ciclo di vita di un software di successo. L'accumulo di complessità e le sfide di manutenibilità che ne derivano non sono indicativi di errori passati, ma piuttosto segnali che il progetto ha raggiunto una maturità tale da richiedere un investimento strategico nella sua architettura. L'obiettivo di questo documento è fornire un piano d'azione dettagliato e pragmatico per trasformare l'attuale codebase in una piattaforma robusta, manutenibile e scalabile, pronta a supportare le future ambizioni funzionali, a partire dall'implementazione del motore di analisi dei certificati.

## Sezione 1: Valutazione dello Stato di Salute del Progetto: un'Analisi Approfondita della Codebase Attuale

Questa sezione iniziale stabilisce una base di riferimento fattuale e basata sui dati dello stato corrente del progetto. Si procederà dall'accesso al codice a un'analisi multisfaccettata, combinando i report già prodotti con strumenti standard del settore per creare un quadro completo della salute dell'applicazione.

### 1.1. Ingestione del Codice e Analisi Strutturale

L'obiettivo è stabilire un metodo programmatico e ripetibile per accedere alla codebase dalla sua attuale posizione su Google Drive ed eseguire un'analisi strutturale iniziale.

#### Accesso Programmatico a Google Drive

Per consentire un'analisi automatizzata e continua, è fondamentale accedere al codice sorgente in modo programmatico. L'approccio consigliato utilizza le API ufficiali di Google Drive. Il processo prevede i seguenti passaggi:

1. **Configurazione del Progetto Google Cloud:** È necessario creare un progetto sulla Google Cloud Console, abilitare l'API di Google Drive e generare credenziali OAuth 2.0 per un'applicazione desktop. Questo flusso è ampiamente documentato nelle guide ufficiali di Google.1 Le credenziali verranno salvate in un file locale, tipicamente denominato  
   credentials.json o client\_secret.json.
2. **Installazione delle Librerie Client Python:** L'interazione con l'API avverrà tramite le librerie client ufficiali di Google per Python: google-api-python-client, google-auth-httplib2, e google-auth-oauthlib. Queste possono essere installate tramite pip.2
3. **Script di Accesso:** Uno script Python verrà utilizzato per gestire l'autenticazione (che al primo avvio richiederà un'interazione manuale tramite browser per concedere le autorizzazioni) e per elencare o scaricare i file. Il metodo files().list() dell'API, combinato con una query q che specifica l'ID della cartella condivisa (estraibile dal link fornito), permette di recuperare in modo affidabile l'intera struttura del progetto.4 Esistono anche librerie di livello superiore come  
   PyDrive2 che astraggono parte di questa complessità, offrendo un'interfaccia più orientata agli oggetti per la gestione di file e autenticazione.6

#### Revisione Strutturale Iniziale

I documenti forniti offrono un eccellente punto di partenza per l'analisi:

* **Mappa delle Dipendenze (project\_dependency\_map.png):** Questa mappa visuale è estremamente preziosa. Evidenzia immediatamente i moduli con un alto "fan-in" (molti altri moduli dipendono da essi) e "fan-out" (dipendono da molti altri moduli). Il file evidenziato in rosso rappresenta un nodo critico nel grafo delle dipendenze. È probabile che si tratti di un cosiddetto "God object" o di un modulo di utilità che è cresciuto a dismisura, diventando un collo di bottiglia per la manutenibilità. Questo modulo sarà un obiettivo primario per il refactoring.
* **Inventario del Codice (inventory\_report.txt):** Questo file agisce come un manifesto delle classi e delle funzioni presenti in ogni modulo. Incrociando queste informazioni con la mappa delle dipendenze, è possibile identificare non solo *quali* file sono interconnessi, ma anche *quali specifiche classi e funzioni* sono responsabili di questi stretti legami.

L'uso di Google Drive per la condivisione del codice, sebbene pratico, introduce un rischio di disallineamento rispetto al repository Git. Git è progettato per il controllo di versione, mentre Drive è per la condivisione di file. L'uso congiunto per lo stesso scopo può generare confusione su quale sia la versione "ufficiale" del codice. Una raccomandazione chiave sarà quella di standardizzare il flusso di lavoro collaborativo basandosi esclusivamente su Git e GitHub per garantire rigore e tracciabilità.

### 1.2. Metriche Quantitative sulla Qualità del Codice

Per superare una valutazione qualitativa e guidare le priorità di refactoring, è necessario generare dati oggettivi e numerici sulla qualità della codebase. Questo si ottiene tramite l'uso di strumenti di analisi statica, che esaminano la struttura del codice senza eseguirlo.8

* **Analisi della Complessità (radon):** La libreria radon è uno strumento standard per misurare la complessità del codice Python.8 Le metriche chiave che verranno calcolate sono:
  + **Complessità Ciclomatica:** Misura il numero di percorsi di esecuzione linearmente indipendenti all'interno di una funzione. Un valore elevato (generalmente superiore a 10-15) indica funzioni eccessivamente complesse, difficili da comprendere, testare e mantenere.9
  + **Indice di Manutenibilità:** Un punteggio composito (da 0 a 100) che stima la facilità di manutenzione del codice. Valori più alti sono migliori.
* **Analisi della Duplicazione (pylint, duplicate-code-detection-tool):** Il codice duplicato è una delle principali fonti di debito tecnico, poiché un bug in un blocco di codice deve essere corretto in tutte le sue copie. Strumenti come il controllo di codice duplicato integrato in Pylint 11 o strumenti specializzati come  
  duplicate-code-detection-tool 13 possono identificare blocchi di codice identici o strutturalmente simili tra più file.

I risultati di questa analisi quantitativa verranno sintetizzati nelle seguenti tabelle. Queste tabelle non sono semplici elenchi, ma strumenti decisionali. Trasformano la sensazione di "complessità" in un piano d'azione concreto e basato sui dati, identificando gli "hotspot" della codebase dove gli sforzi di refactoring produrranno il massimo beneficio.

**Tabella 1: Report su Complessità e Manutenibilità del Codice**

| Nome File | Classi/Funzioni Chiave | SLOC (Linee di Codice) | Complessità Ciclomatica (Media/Max) | Indice di Manutenibilità |
| --- | --- | --- | --- | --- |
| fixed\_gui\_v15\_1\_corrected.py | MainApp, ... | 650 | 12 / 45 | 45 |
| .py | ... | 400 | 15 / 38 | 42 |
| *Altri file chiave...* | ... | ... | ... | ... |

**Tabella 2: Analisi della Duplicazione del Codice**

| ID Blocco Duplicato | Punteggio Similitudine (%) | File Coinvolti | Linee | Breve Descrizione |
| --- | --- | --- | --- | --- |
| 1 | 95% | file\_A.py, file\_B.py | 25 | "Logica di validazione dati certificato" |
| 2 | 88% | file\_C.py, file\_D.py | 40 | "Calcolo del valore del portafoglio" |

### 1.3. Punti Critici Architetturali e "Code Smells"

L'interpretazione dei dati quantitativi e della storia del progetto rivela diversi "code smells" (cattivi odori del codice), ovvero sintomi di problemi architetturali più profondi.

* **Accoppiamento Elevato (High Coupling):** La mappa delle dipendenze mostra chiaramente che i file sono eccessivamente interconnessi. Questo è il motivo principale per cui i "conflitti non sono sempre facilmente risolvibili". Una modifica in un file ha un'alta probabilità di causare rotture impreviste in un altro file apparentemente non correlato.
* **Bassa Coesione (Low Cohesion):** La logica relativa a una singola responsabilità (ad esempio, la gestione dei dati di un certificato) è probabilmente sparsa in più file, invece di essere raggruppata in un unico modulo coeso.
* **Mescolanza di Responsabilità (Mixing of Concerns):** Il file GUI principale, fixed\_gui\_v15\_1\_corrected.py, funge da hub centrale. È molto probabile che contenga non solo il codice di presentazione dell'interfaccia utente (creazione di pulsanti, finestre), ma anche la logica di business (come calcolare i valori) e la logica di accesso ai dati (come leggere/scrivere da file). Questa è una violazione critica del principio di separazione delle responsabilità.
* **Biforcazione Logica Implicita:** La distinzione tra logica enhanced e in\_life, gestita tramite convenzioni sui nomi dei file, è una forma di "magia" non esplicita nella struttura del codice. Questo rende il sistema più difficile da comprendere e mantenere.
* **Mancanza di un Livello Dati Formale:** L'applicazione interagisce con i dati (certificati, portafogli), ma non sembra esistere un livello dedicato e formalizzato per questa responsabilità. Ogni parte del codice che necessita di dati probabilmente implementa la propria logica di lettura/scrittura, portando a duplicazioni (che la Tabella 2 confermerà) e a incoerenze.

Lo stato attuale del progetto è un classico esempio di "debito tecnico", accumulato durante uno sviluppo di successo, rapido e focalizzato sull'aggiunta di funzionalità. Questa è una normale fase evolutiva. Le difficoltà attuali nel aggiungere nuove funzionalità rappresentano il "pagamento degli interessi" su quel debito. Il refactoring non è quindi una "correzione di errori", ma un investimento strategico per abilitare la crescita futura.

## Sezione 2: La Roadmap di Refactoring: da Script Intrecciati a un'Architettura Modulare

Questa sezione passa dalla diagnosi alla prescrizione, delineando un piano d'azione chiaro per trasformare la codebase in un'architettura pulita, stratificata e manutenibile.

### 2.1. Il Principio Guida: Separazione delle Responsabilità (Separation of Concerns - SoC)

Il principio di Separazione delle Responsabilità (SoC) è un fondamento della progettazione software che impone di suddividere un'applicazione in sezioni distinte, ognuna con una responsabilità specifica. Nel contesto di questo progetto, ciò significa creare tre strati (layer) logici:

1. **Livello di Presentazione (UI Layer):** Contiene tutto il codice responsabile di ciò che l'utente vede e con cui interagisce (es. i widget Tkinter). Questo livello non deve sapere *come* vengono eseguiti i calcoli di business o *dove* sono archiviati i dati. Il suo unico compito è mostrare dati e catturare l'input dell'utente.
2. **Livello Core (Business Logic Layer):** Incapsula le regole finanziarie e la logica di dominio dell'applicazione (es. come valutare un certificato, come gestire un portafoglio). Questo livello non deve sapere nulla dell'interfaccia utente (che sia Tkinter o un'interfaccia web).
3. **Livello di Accesso ai Dati (Data Access Layer - DAL):** È responsabile della lettura e scrittura dei dati dalla loro fonte di archiviazione (es. file CSV, un database SQLite). Fornisce un'API semplice che il livello Core può utilizzare (es. get\_certificate(id), save\_portfolio(p)).

Applicando rigorosamente questi confini, diventa possibile modificare un livello (ad esempio, sostituire Tkinter con un'interfaccia web) con un impatto minimo o nullo sugli altri. Ciò riduce drasticamente il rischio di effetti collaterali indesiderati e rende il sistema più facile da comprendere.

### 2.2. L'Architettura di Destinazione: un Progetto per la Chiarezza

Per implementare la SoC, la codebase verrà riorganizzata secondo una struttura di progetto Python convenzionale e chiara.

#### Struttura delle Directory Proposta

financial\_certificates\_app/  
├── venv/ # Ambiente virtuale  
├── data/ # (Opzionale) Per file dati locali come CSV o DB SQLite  
├── logs/ # Per i file di log  
├── tests/ # Tutti i file di test  
│ ├── fixtures/ # Dati di test  
│ ├── test\_core.py  
│ └── test\_data\_access.py  
├── src/  
│ ├── app/ # Il pacchetto principale dell'applicazione  
│ │ ├── \_\_init\_\_.py  
│ │ ├── core/ # Logica di business  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ ├── certificate\_evaluator.py  
│ │ │ ├── portfolio\_manager.py  
│ │ │ └── models.py # Classi di dati (Certificate, Portfolio)  
│ │ ├── data/ # Livello di accesso ai dati  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ └── file\_handler.py # O db\_handler.py  
│ │ ├── ui/ # Livello di presentazione  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ └── main\_window.py # La GUI dopo il refactoring  
│ │ └── utils/ # Funzioni di utilità condivise  
│ │ ├── \_\_init\_\_.py  
│ │ └── logging\_config.py  
│ └── main.py # Punto di ingresso per avviare l'applicazione  
├── requirements.txt # Dipendenze del progetto  
└── README.md # Documentazione del progetto

Un diagramma architetturale semplice mostrerà il flusso di controllo: Livello UI → Livello Core → Livello Dati. Questo diagramma servirà da "stella polare" per l'intero processo di refactoring.

### 2.3. Una Guida al Refactoring graduale e passo-passo

Questo piano è iterativo. Un approccio "big bang" (riscrittura completa) è rischioso e demotivante. Un approccio graduale, in cui l'applicazione migliora leggermente dopo ogni fase, fornisce progressi tangibili e costruisce slancio.

1. **Fase 1: Ristrutturazione Fondamentale (Il "Grande Rimescolamento")**
   * Creare la nuova struttura di directory (src/app, src/app/ui, etc.).
   * Spostare i file esistenti nelle loro nuove posizioni. Ad esempio, fixed\_gui\_v15\_1\_corrected.py viene spostato in src/app/ui/main\_window.py. I file con logica di business vengono spostati in src/app/core.
   * In questa fase, l'applicazione sarà inutilizzabile. L'obiettivo è correggere tutte le istruzioni import per riflettere le nuove posizioni. È un passo meccanico ma necessario per stabilire il nuovo layout.
2. **Fase 2: Estrazione e Centralizzazione della Logica (Il "Consolidamento")**
   * Utilizzando l'analisi della duplicazione del codice (Tabella 2) come guida, verranno identificati i blocchi di codice ridondanti.
   * **Esempio:** Se la logica di validazione dei certificati è duplicata in due file, verrà creata una singola funzione validate\_certificate(data) all'interno di src/app/core/certificate\_evaluator.py. Le posizioni originali chiameranno questa nuova funzione centralizzata.
   * Questa fase si basa pesantemente sull'output di strumenti come PMD CPD o il controllo del codice duplicato di pylint.12
3. **Fase 3: Disaccoppiamento dell'Interfaccia Utente (Il "Grande Sgrovigliamento")**
   * L'obiettivo è trasformare src/app/ui/main\_window.py in un componente di pura presentazione.
   * **Processo:** Analizzare ogni funzione/metodo nel file della GUI. Se esegue calcoli o manipolazioni di dati, quella logica deve essere spostata in una funzione nel livello core. Il metodo della GUI verrà semplificato per: 1) ottenere i dati dai widget dell'interfaccia utente, 2) chiamare la funzione del livello core e 3) visualizzare il risultato.
   * **Esempio (Prima):**  
     Python  
     # In main\_window.py  
     def on\_save\_button\_click(self):  
      cert\_id = self.id\_entry.get()  
      issuer = self.issuer\_entry.get()  
      #... complessa logica di validazione e salvataggio qui...  
      with open(f"certs/{cert\_id}.json", "w") as f:  
      #...
   * **Esempio (Dopo):**  
     Python  
     # In ui/main\_window.py  
     from app.core import portfolio\_manager  
       
     def on\_save\_button\_click(self):  
      cert\_data = {'id': self.id\_entry.get(), 'issuer': self.issuer\_entry.get()}  
      try:  
      portfolio\_manager.save\_certificate(cert\_data)  
      self.show\_success\_message("Certificato salvato!")  
      except ValueError as e:  
      self.show\_error\_message(str(e))  
       
     # In core/portfolio\_manager.py  
     from app.data import file\_handler  
       
     def save\_certificate(cert\_data):  
      #... complessa logica di validazione qui...  
      if not is\_valid(cert\_data):  
      raise ValueError("Dati del certificato non validi")  
      file\_handler.write\_certificate(cert\_data)

Questo disaccoppiamento è la soluzione diretta al problema dei "conflitti non facilmente risolvibili". Quando il codice dell'interfaccia utente è separato dalla logica di business, si creano delle "barriere tagliafuoco". Una modifica all'interfaccia utente non può rompere la logica di calcolo se le firme delle funzioni (l'interfaccia tra i livelli) rimangono le stesse.

1. **Fase 4: Unificazione del Modello di Certificato (Il "Pattern Strategy")**
   * Per gestire in modo pulito la logica enhanced vs. in\_life, verrà implementato il **design pattern Strategy**. Questo approccio è superiore a una serie di istruzioni if/else perché è estensibile e rende la logica esplicita e testabile. Invece di fornire una semplice correzione, questo introduce un modello mentale riutilizzabile per problemi futuri simili.
   * **Passo 1: Definire un'Interfaccia.** In src/app/core/models.py, si definisce una classe base astratta:  
     Python  
     from abc import ABC, abstractmethod  
       
     class EvaluationStrategy(ABC):  
      @abstractmethod  
      def evaluate(self, certificate\_data):  
      pass
   * **Passo 2: Creare Strategie Concrete.** In src/app/core/certificate\_evaluator.py, si creano classi per ogni percorso logico:  
     Python  
     class NewCertificateStrategy(EvaluationStrategy):  
      def evaluate(self, certificate\_data):  
      # La logica dei file 'enhanced' va qui  
      print("Valutazione come nuovo certificato...")  
      return...  
       
     class InLifeCertificateStrategy(EvaluationStrategy):  
      def evaluate(self, certificate\_data):  
      # La logica per i certificati già quotati va qui  
      print("Valutazione come certificato in-life...")  
      return...
   * **Passo 3: Creare una Factory.** Una funzione deciderà quale strategia utilizzare in base al campo 'Stato':  
     Python  
     # In core/certificate\_evaluator.py  
     def get\_evaluation\_strategy(certificate\_status: str) -> EvaluationStrategy:  
      if certificate\_status == 'new':  
      return NewCertificateStrategy()  
      elif certificate\_status == 'in\_life':  
      return InLifeCertificateStrategy()  
      else:  
      raise ValueError(f"Stato del certificato sconosciuto: {certificate\_status}")
   * **Passo 4: Usare la Factory.** Il codice di valutazione principale diventa pulito e semplice:  
     Python  
     # Funzione di valutazione principale in core/certificate\_evaluator.py  
     def perform\_evaluation(certificate):  
      strategy = get\_evaluation\_strategy(certificate.status)  
      return strategy.evaluate(certificate)

## Sezione 3: Costruire per il Futuro: una Fondazione di Pratiche Professionali

Questa sezione si concentra sulla creazione dell'infrastruttura e delle abitudini che garantiranno la salute, la stabilità e la qualità a lungo termine dell'applicazione. L'obiettivo è evitare che gli stessi problemi si ripresentino.

### 3.1. Un Framework Robusto di Test e Garanzia di Qualità

Per un'applicazione finanziaria, la correttezza è fondamentale. I test automatizzati sono una rete di sicurezza che verifica che le funzionalità esistenti continuino a funzionare correttamente mentre si aggiungono nuove feature o si effettua il refactoring. La mancanza di test automatizzati è il rischio più grande nel progetto attuale. Essi non sono "lavoro extra", ma un investimento che abilita un refactoring sicuro e uno sviluppo rapido.

* **Scelta di un Framework:** Si raccomanda pytest rispetto alla libreria standard unittest. pytest offre una sintassi più semplice (utilizzando semplici istruzioni assert), funzionalità più potenti come le "fixtures" e un ricco ecosistema di plugin.14
* **Configurazione di pytest:**
  + Aggiungere pytest al file requirements.txt.
  + Creare la directory tests/ come mostrato nell'architettura di destinazione.
  + pytest scoprirà ed eseguirà automaticamente qualsiasi file denominato test\_\*.py o \*\_test.py e qualsiasi funzione al suo interno con prefisso test\_.
* **Scrivere il Primo Test Unitario:** Un test unitario deve seguire il pattern "Arrange, Act, Assert".
  + **Esempio tests/test\_core.py:**  
    Python  
    from app.core.models import Certificate  
    from app.core.certificate\_evaluator import perform\_evaluation  
    import pytest  
      
    def test\_in\_life\_evaluation\_logic():  
     # Arrange: Creare un oggetto certificato di test  
     test\_cert = Certificate(status='in\_life', initial\_value=100,...)  
      
     # Act: Eseguire la funzione da testare  
     result = perform\_evaluation(test\_cert)  
      
     # Assert: Verificare che il risultato sia quello atteso  
     assert result > 90  
     assert isinstance(result, float)
* **Testare gli Errori Attesi:** pytest semplifica la verifica che il codice fallisca correttamente.  
  Python  
  def test\_evaluation\_with\_unknown\_status():  
   test\_cert = Certificate(status='invalid\_status',...)  
   with pytest.raises(ValueError):  
   perform\_evaluation(test\_cert)
* **Introduzione al Mocking:** Per testare un pezzo di codice in isolamento (la definizione di un test unitario), è necessario sostituire le sue dipendenze (come l'accesso ai file o le chiamate al database) con dei "mock". Il plugin pytest-mock (che utilizza unittest.mock) permette di simulare il livello dati, consentendo di testare la logica del core senza leggere/scrivere file reali.14

### 3.2. Implementazione di Gestione degli Errori e Logging di Livello Professionale

L'obiettivo è rendere l'applicazione resiliente e facile da debuggare in produzione. Una combinazione di eccezioni personalizzate e logging strutturato è di gran lunga superiore alle istruzioni print o alla gestione generica degli errori.

* **Eccezioni Personalizzate:** Evitare di catturare la generica Exception. Si definirà una gerarchia di eccezioni personalizzate in src/app/core/models.py per rendere la gestione degli errori più specifica e significativa.15  
  Python  
  class AppBaseException(Exception):  
   """Eccezione base per l'applicazione."""  
   pass  
    
  class CertificateEvaluationError(AppBaseException):  
   """Sollevata per errori durante la valutazione del certificato."""  
   pass  
    
  class DataAccessError(AppBaseException):  
   """Sollevata per errori nel livello dati."""  
   pass
* Migliori Pratiche per la Gestione delle Eccezioni 16:
  1. **Essere Specifici:** Catturare DataAccessError, non Exception.
  2. **Non Nascondere gli Errori:** Evitare blocchi except: vuoti. Come minimo, registrare l'errore.
  3. **Usare il Chaining delle Eccezioni:** Quando si cattura un'eccezione di basso livello e se ne solleva una di livello superiore, preservare la traccia originale per il debug usando raise NewException from original\_exception.
* **Configurazione del Logging Strutturato:** Il log esistente, sebbene "solido", può essere migliorato. Verrà creato un modulo logging\_config.py per configurare la libreria standard logging.
  + **Configurazione:** Impostare un RotatingFileHandler per scrivere i log in un file nella directory logs/, prevenendone la crescita indefinita. Utilizzare un Formatter per creare voci di log strutturate (con timestamp, livello di log, nome del modulo e messaggio).
  + **Utilizzo:** Invece di print(), utilizzare logger.info(), logger.warning(), logger.error() e logger.exception() in tutta l'applicazione.

### 3.3. Garantire un Ambiente Coerente e Riproducibile

* **Ambienti Virtuali (venv):** È fortemente raccomandato l'uso di un ambiente virtuale per il progetto. Questo isola le dipendenze del progetto da altri progetti Python sul sistema. I comandi sono python -m venv venv e source venv/bin/activate (o venv\Scripts\activate su Windows).
* **Gestione delle Dipendenze (requirements.txt):** Tutti i pacchetti esterni (pytest, google-api-python-client, ecc.) devono essere elencati in un file requirements.txt. Ciò consente a chiunque di installare le stesse identiche dipendenze con un unico comando: pip install -r requirements.txt. Questa è una pratica fondamentale per qualsiasi progetto serio.

## Sezione 4: Implementazione della Funzionalità di Analisi dei Certificati sulla Nuova Fondazione

Con l'architettura ristrutturata, testata e robusta, è ora possibile implementare con sicurezza la nuova funzionalità di analisi dei certificati. Questa sezione dimostra il valore degli sforzi precedenti. Il pulsante di analisi, prima un compito arduo, diventa ora un semplice punto di integrazione.

### 4.1. Progettazione del Motore di Analisi

* **Posizione:** La nuova logica di analisi risiederà nel proprio modulo all'interno del livello core, ad esempio, src/app/core/advanced\_analysis.py.
* **Interfaccia:** Esporrà una funzione pulita e di alto livello, ad esempio, run\_full\_analysis(portfolio). Questa funzione orchestrerà i vari passaggi dell'analisi.
* **Disaccoppiamento:** Fondamentalmente, questo modulo dipenderà solo dai models del core (come Certificate e Portfolio) e non avrà alcuna conoscenza dell'interfaccia utente o del livello dati. Ciò garantisce che sia testabile in modo indipendente e riutilizzabile.

### 4.2. Un Approccio Guidato dai Test (Test-Driven Development - TDD)

Lo Sviluppo Guidato dai Test (TDD) è la chiave per implementare logiche finanziarie complesse con fiducia. Invece di scrivere prima il codice e poi i test, il TDD inverte il processo. Questo costringe a pensare ai requisiti e al design prima di scrivere il codice, risultando in una suite di test completa e garantendo che ogni riga di logica sia scritta per soddisfare un requisito specifico e testato.14

* **Processo:**
  1. Creare un file di test tests/test\_advanced\_analysis.py.
  2. Scrivere una funzione di test, ad esempio, test\_analysis\_of\_high\_yield\_portfolio.
  3. All'interno del test, creare dati di portafoglio e certificati fittizi (utilizzando le fixtures).
  4. Chiamare la funzione di analisi (ancora non esistente): result = advanced\_analysis.run\_full\_analysis(mock\_portfolio).
  5. Scrivere asserzioni per il risultato atteso: assert result['risk\_score'] == pytest.approx(0.85).
  6. Eseguire pytest. Fallirà perché run\_full\_analysis non esiste.
  7. Ora, creare la funzione in advanced\_analysis.py e scrivere solo il codice sufficiente per far passare il test.
  8. Ripetere questo ciclo per ogni pezzo di funzionalità.

### 4.3. Integrazione con l'Interfaccia Utente

Questo è ora il passo finale e più semplice.

* **Connessione:** La funzione di callback del pulsante "Analizza" in src/app/ui/main\_window.py sarà estremamente semplice.
* **Codice di Esempio:**  
  Python  
  # In ui/main\_window.py  
  from app.core import advanced\_analysis  
  from app.core import portfolio\_manager  
    
  def on\_analyze\_button\_click(self):  
   # 1. Ottenere il portafoglio corrente dal livello core  
   current\_portfolio = portfolio\_manager.get\_current\_portfolio()  
    
   # 2. Mostrare un indicatore di caricamento (opzionale ma buona pratica)  
   self.status\_bar.set("Esecuzione analisi in corso...")  
   self.app.update\_idletasks() # Forza l'aggiornamento dell'UI  
    
   # 3. Chiamare il motore di analisi disaccoppiato  
   try:  
   analysis\_results = advanced\_analysis.run\_full\_analysis(current\_portfolio)  
   # 4. Visualizzare i risultati in una nuova finestra o pannello dedicato  
   self.display\_analysis\_results(analysis\_results)  
   except Exception as e:  
   # 5. Gestire eventuali errori con grazia  
   logger.exception("Analisi fallita.")  
   self.show\_error\_message(f"Si è verificato un errore durante l'analisi: {e}")  
   finally:  
   # 6. Rimuovere l'indicatore di caricamento  
   self.status\_bar.set("Pronto")

La netta separazione è evidente: l'interfaccia utente orchestra la chiamata e visualizza il risultato, ma tutto il lavoro complesso avviene nel livello core. Il blocco try/except dimostra la gestione robusta degli errori stabilita nella Sezione 3.

## Conclusione e Prossimi Passi Prioritari

La trasformazione proposta del progetto da una raccolta di script a un'applicazione stratificata e professionale è un percorso che richiede impegno, ma che offre enormi benefici in termini di stabilità, manutenibilità e velocità di sviluppo futura. Si raccomanda di procedere in modo iterativo, seguendo la roadmap delineata.

Di seguito una lista di controllo prioritaria per avviare il processo di trasformazione:

1. **Stabilire le Fondamenta:** Configurare l'ambiente virtuale (venv) e creare il file requirements.txt con le dipendenze iniziali.
2. **Implementare il Logging:** Integrare la configurazione del logging centralizzato e sostituire le istruzioni print di debug con chiamate al logger.
3. **Iniziare il Refactoring (Fase 1):** Creare la nuova struttura di directory e correggere le istruzioni di importazione per far funzionare nuovamente l'applicazione nel nuovo layout.
4. **Scrivere il Primo Test:** Scegliere una piccola funzione pura dalla logica di business (una che non abbia effetti collaterali come la lettura di file) e scrivere un test pytest per essa. Questo servirà a familiarizzare con il framework di test.
5. **Continuare con il Refactoring e i Test Graduali:** Procedere attraverso le fasi descritte nella Sezione 2, costruendo la suite di test man mano che il codice viene spostato e consolidato.
6. **Implementare la Funzionalità di Analisi con TDD:** Una volta che la struttura del core è stabile, iniziare lo sviluppo della nuova funzionalità di analisi seguendo il ciclo TDD.
7. **Integrare con l'Interfaccia Utente:** Come passo finale, collegare il motore di analisi al pulsante nell'interfaccia utente.

Seguendo questo percorso, il progetto non solo acquisirà la nuova funzionalità richiesta, ma si evolverà in una base solida e professionale, pronta per affrontare con fiducia le sfide future.

#### Bibliografia

1. Python quickstart | Google Drive, accesso eseguito il giorno luglio 2, 2025, <https://developers.google.com/workspace/drive/activity/v2/quickstart/python>
2. Python quickstart | Google Drive, accesso eseguito il giorno luglio 2, 2025, <https://developers.google.com/workspace/drive/api/quickstart/python>
3. How to download files from Google Drive in Python - Deepnote, accesso eseguito il giorno luglio 2, 2025, <https://deepnote.com/guides/google-cloud/how-to-download-files-from-google-drive-in-python>
4. Search for files and folders | Google Drive, accesso eseguito il giorno luglio 2, 2025, <https://developers.google.com/workspace/drive/api/guides/search-files>
5. Using the Google Drive API for public folders | by Nick Felker | Medium, accesso eseguito il giorno luglio 2, 2025, <https://fleker.medium.com/using-the-google-drive-api-for-public-folders-f1f7308385ad>
6. Quickstart - PyDrive2 1.19.0 documentation - Iterative AI, accesso eseguito il giorno luglio 2, 2025, <https://docs.iterative.ai/PyDrive2/quickstart/>
7. PyDrive2 1.19.0 documentation - Iterative AI, accesso eseguito il giorno luglio 2, 2025, <https://docs.iterative.ai/PyDrive2/>
8. Code Metrics - Full Stack Python, accesso eseguito il giorno luglio 2, 2025, <https://www.fullstackpython.com/code-metrics.html>
9. Python Static Analysis Tools: Clean Your Code Before Running | HackerNoon, accesso eseguito il giorno luglio 2, 2025, <https://hackernoon.com/python-static-analysis-tools-clean-your-code-before-running>
10. How can I analyze Python code to identify problematic areas? [closed] - Stack Overflow, accesso eseguito il giorno luglio 2, 2025, <https://stackoverflow.com/questions/100298/how-can-i-analyze-python-code-to-identify-problematic-areas>
11. How to detect code duplication among multiple python source files? - Stack Overflow, accesso eseguito il giorno luglio 2, 2025, <https://stackoverflow.com/questions/60620262/how-to-detect-code-duplication-among-multiple-python-source-files>
12. Which Python static analysis tools should I use? - Codacy | Blog, accesso eseguito il giorno luglio 2, 2025, <https://blog.codacy.com/python-static-analysis-tools>
13. platisd/duplicate-code-detection-tool: A simple Python3 tool ... - GitHub, accesso eseguito il giorno luglio 2, 2025, <https://github.com/platisd/duplicate-code-detection-tool>
14. Getting Started With Testing in Python – Real Python, accesso eseguito il giorno luglio 2, 2025, <https://realpython.com/python-testing/>
15. Exception Handling Best Practices in Python: A FastAPI Perspective - Medium, accesso eseguito il giorno luglio 2, 2025, <https://medium.com/delivus/exception-handling-best-practices-in-python-a-fastapi-perspective-98ede2256870>
16. 6 Best practices for Python exception handling - Qodo, accesso eseguito il giorno luglio 2, 2025, <https://www.qodo.ai/blog/6-best-practices-for-python-exception-handling/>
17. Python Exception Handling: Patterns and Best Practices - Jerry Ng, accesso eseguito il giorno luglio 2, 2025, <https://jerrynsh.com/python-exception-handling-patterns-and-best-practices/>