Guida alla Rifattorizzazione di PubliScript

Ristrutturazione modulare per un codice più mantenibile

Obiettivi della Rifattorizzazione

- Rendere il codice più **modulare** e **manutenibile**
- Separare chiaramente le **responsabilità** (UI, AI, Framework, Analisi)
- Consentire in futuro di **sostituire o integrare** nuove AI senza modificare il core

Principi da Rispettare

- Non alterare le **logiche interne** delle funzioni e classi esistenti
- Spostare **fisicamente** il codice nei nuovi moduli
- Aggiornare correttamente gli **import**

Problematiche Attuali

Metodi fuori classe

Metodi come `_analyze_market_crisp` definiti erroneamente fuori dalla classe che li utilizza.

Duplicazioni di codice

Metodi duplicati come `replace_variables_advanced` e `check_unresolved_placeholders`.

Import circolari

Dipendenze circolari tra i moduli, che rendono difficile la gestione del codice.

Nuova Struttura del Progetto

Organizzazione modulare con responsabilità ben definite

Struttura Directory

- publiscript/
  - ai_interfaces/
    - genspark_driver.py
    - browser_manager.py
    - interaction_utils.py
    - file_text_utils.py
  - framework/
    - crisp_framework.py
    - crisp_extractors.py
    - crisp_utils.py
    - analysis/
      - market_analysis.py
      - buyer_persona.py
      - gap_analysis.py
  - ui/
    - app_launcher.py
    - ai_connection.py
    - book_builder.py
    - chat_manager.py
    - cooldown_manager.py
    - log_utils.py
    - components.py
  - main.py

Riorganizzazione dei Moduli

File Originale: book_builder.py

- ⚠ `_analyze_market_crisp` fuori dalla classe
- ⚠ `replace_variables` non definita
- Duplicazioni: `replace_variables_advanced`
- Duplicazioni: `check_unresolved_placeholders`
- Duplicazioni: `get_clean_input_box`

↓

ai_interfaces/

Interazione con servizi AI esterni

- `send_to_genspark`
- `handle_context_limit`
- `reset_context_manual`
- `get_clean_input_box` (unificato)

framework/

Logica di business principale

- `_analyze_market_crisp` (corretto)
- `replace_variables` (implementato)
- `replace_variables_advanced` (unificato)
- `check_unresolved_placeholders` (unificato)

ui/book_builder.py

Interfaccia utente e coordinamento

- `AIBookBuilder` (classe principale)
- `create_interface`
- `load_analysis_results`
- `format_analysis_results_html`

</> Pattern di Correzione Import

Prima

```
from crisp_utils import replace_variable
```

Dopo

```
from publiscript.framework.crisp_utils import replac
```

Processo di Rifattorizzazione: Step by Step

Focus sul file book_builder.py

Fase 1: Correzioni Base

- Correggere indentazioni
- Risolvere metodi fuori classe
- Unificare metodi duplicati
- Definire funzioni mancanti

Correzione

```
class AIBookBuilder:
    def __init__(self):
        # ... codice esistente

    # Metodo correttamente indentato all'int
    def _analyze_market_crisp(self, book_typ
        # ... logica esistente
```

Fase 2: Estrazione Moduli

- Estrarre metodi per ai_interfaces
- Estrarre metodi per framework
- Mantenere solo UI nel file originale
- Riutilizzare la stessa logica

Estrazione

```
# In framework/analysis/market_analysis.py
def analyze_market_crisp(book_type, keyword,
                         crisp_framework=None, dr
                         chat_manager=None, selec
    # logica originale dal metodo _analyze_m
    # con self.crisp → crisp_framework
    # con self.driver → driver
    # con self.add_log → chat_manager.add_lo
```

Fase 3: Aggiornamento Import

- Aggiornare import nei nuovi moduli
- Importare i nuovi moduli dal file originale
- Risolvere eventuali import circolari
- Verificare compatibilità

Import

```
# In ui/book_builder.py

from publiscript.ai_interfaces.genspark_driv
from publiscript.framework.analysis.market_a
from publiscript.framework.crisp_utils impor

class AIBookBuilder:
    # ... codice della classe
```

Esempio Completo: Trasformazione di _analyze_market_crisp

Prima della Rifattorizzazione

```
# Metodo erroneamente definito fuori della classe
def _analyze_market_crisp(self, book_type, keyword, language, market, selected_phases=None):
    try:
        self.add_log(f"Avvio analisi CRISP 5.0 per: {keyword}")
        # Codice di implementazione...

        def process_prompt(prompt_text):
            self.add_log(f"Elaborazione prompt: {len(prompt_text)} caratteri")
            response = self.send_to_genspark(prompt_text)
            return response

        # Resto dell'implementazione originale...

        return self.chat_manager.get_log_history_string()
    except Exception as e:
        self.add_log(f"❌ Errore nell'analisi CRISP: {str(e)}")
        return self.chat_manager.get_log_history_string()
```

↓

Step 1: Correzione all'interno della classe

```
class AIBookBuilder:
    # ... altri metodi della classe

    # Metodo correttamente indentato all'interno della classe
    def _analyze_market_crisp(self, book_type, keyword, language, market, selected_phases=None):
        try:
            self.add_log(f"Avvio analisi CRISP 5.0 per: {keyword}")
            # Implementazione originale...
```

↓

Step 2a: framework/analysis/market_analysis.py

```
def analyze_market_crisp(book_type, keyword, language,
                         market, selected_phases=None,
                         crisp_framework=None,
                         driver=None,
                         chat_manager=None):
    try:
        chat_manager.add_log(f"Avvio analisi CRISP 5.0 per: {keyword}")

        def process_prompt(prompt_text):
            chat_manager.add_log(f"Elaborazione prompt: {len(prompt_text)}
            from publiscript.ai_interfaces.genspark_driver import send_to_g
            response = send_to_genspark(driver, prompt_text)
            return response

        # Resto dell'implementazione originale...

        return chat_manager.get_log_history_string()
    except Exception as e:
        chat_manager.add_log(f"❌ Errore nell'analisi CRISP: {str(e)}")
        return chat_manager.get_log_history_string()
```

Step 2b: ui/book_builder.py

```
from publiscript.framework.analysis.market_analysis import analyze_market_c

class AIBookBuilder:
    # ... altri metodi della classe

    def _analyze_market_crisp(self, book_type, keyword, language, market, s
        # Delega al modulo dedicato mantenendo la stessa interfaccia
        return analyze_market_crisp(
            book_type=book_type,
            keyword=keyword,
            language=language,
            market=market,
            selected_phases=selected_phases,
            crisp_framework=self.crisp,
            driver=self.driver,
            chat_manager=self.chat_manager
        )
```

Vantaggi della Rifattorizzazione

- **Separazione delle responsabilità:** Logica di analisi separata da UI
- **Codice più testabile:** Funzioni pure con dipendenze esplicite
- **Riuso facilitato:** Analisi usabile da altre interfacce
- **Manutenibilità migliorata:** Moduli più piccoli e focalizzati

Correzione di Errori Comuni in book_builder.py

Esempi concreti per risolvere i problemi identificati

Risoluzione Metodi Duplicati

Problema: metodi duplicati

Funzioni come `replace_variables_advanced`, `check_unresolved_placeholders`, e `get_clean_input_box` appaiono due volte nel codice.

Soluzione: unificare in un unico modulo

Spostare le funzioni duplicate nel file appropriato e implementare una singola versione.

framework/crisp_utils.py

```
def replace_variables_advanced(text, project_data):
    """Versione avanzata di sostituzione variabili con iterazione completa
    if not project_data:
        return text

    result = text
    import re
    placeholders = re.findall(r'\{([A-Za-z_]+)\}', text)

    for placeholder in placeholders:
        if placeholder in project_data:
            value = project_data[placeholder]
            if value is not None:
                if not isinstance(value, str):
                    value = str(value)
                result = result.replace(f"{{{placeholder}}}", value)

    return result


def check_unresolved_placeholders(text):
    """Verifica placeholders non risolti nel testo"""
    import re
    placeholders = re.findall(r'\{([A-Za-z_]+)\}', text)
    return placeholders if placeholders else None
```

Correzione Metodi Fuori Classe

Problema: metodo fuori dalla classe

Il metodo `_analyze_market_crisp` è definito erroneamente fuori dalla classe ma usa `self` come parametro.

Soluzione: spostare dentro la classe

Spostare il metodo all'interno della classe `AIBookBuilder` e poi delegare alla nuova implementazione.

ui/book_builder.py

```
class AIBookBuilder:
    # ... altri metodi ...

    def _analyze_market_crisp(self, book_type, keyword, language, market,
        """
        Analizza il mercato usando il framework CRISP 5.0.
        Ora delega alla funzione implementata nel modulo dedicato.
        """
        from publiscript.framework.analysis.market_analysis import analyze

        return analyze_market_crisp(
            book_type=book_type,
            keyword=keyword,
            language=language,
            market=market,
            selected_phases=selected_phases,
            crisp_framework=self.crisp,
            driver=self.driver,
            chat_manager=self.chat_manager
        )
```

Gestione di Funzioni Mancanti e Import Circolari

Problema: `replace_variables` non definita

La funzione viene chiamata ma non è definita nel codice.

Soluzione: implementare o sostituire

Implementare la funzione mancante o sostituirla con `replace_variables_advanced` già disponibile.

framework/crisp_utils.py

```
def replace_variables(text, project_data):
    """Versione base di sostituzione variabili"""
    return replace_variables_advanced(text, project_data)
```

Problema: import circolari

Moduli che si importano reciprocamente, causando errori di esecuzione.

Soluzione: import dinamici

Spostare gli import all'interno delle funzioni che li utilizzano, quando necessario.

Risoluzione import circolari

```
# Invece di questo (in cima al file):
from publiscript.ai_interfaces.genspark_driver import send_to_genspark

# Usare questo (all'interno della funzione):
def process_prompt(prompt_text):
    # Import spostato all'interno della funzione
    from publiscript.ai_interfaces.genspark_driver import send_to_genspark

    response = send_to_genspark(driver, prompt_text)
    return response
```

Checklist di Validazione Post-Rifattorizzazione

- [x] **Verificare che tutti i metodi duplicati** siano unificati in un unico punto
- [x] **Assicurarsi che tutti i metodi siano definiti** nelle classi appropriate
- [x] **Correggere tutte le indentazioni** per una leggibilità ottimale
- [x] **Testare che tutte le funzionalità** funzionino come prima della rifattorizzazione
- [x] **Verificare la correttezza degli import** in tutti i nuovi moduli creati
- [x] **Aggiornare la documentazione** con la nuova struttura organizzativa

Piano di Implementazione e Conclusioni

Roadmap e vantaggi della nuova architettura modulare

Roadmap di Implementazione

Fase 1: Correzione preliminare
Correggere errori sintattici e metodi fuori classe prima della riorganizzazione dei file.

Fase 2: Creazione della nuova struttura
Creare le directory e i file vuoti secondo la nuova organizzazione.

Fase 3: Spostamento graduale del codice
Spostare modulo per modulo, assicurando sempre la compatibilità con import temporanei.

Fase 4: Rivedere gli import
Aggiornare tutti gli import secondo il nuovo schema, risolvendo import circolari.

Fase 5: Test e validazione
Verificare che tutte le funzionalità operino esattamente come prima della rifattorizzazione.

Riepilogo Obiettivi

- ✓ **Mantenere la logica invariata** - Focus solo sulla ristrutturazione
- ✓ **Separare le responsabilità** - UI, AI, Framework, Analisi
- ✓ **Eliminare duplicazioni** - Codice centralizzato e riutilizzabile
- ✓ **Facilitare l'estendibilità** - Nuove AI senza modificare il core

Vantaggi della Rifattorizzazione

Modularità
Componenti indipendenti e facilmente sostituibili, con interfacce chiare tra i moduli.

Manutenibilità
Struttura logica e organizzata che rende più facile trovare e modificare il codice.

Scalabilità
Nuove funzionalità possono essere aggiunte senza impattare il codice esistente.

Testabilità
Componenti isolati più facilmente testabili, con dipendenze esplicite.

Best Practices per la Rifattorizzazione

1. **Approccio incrementale**: rifattorizzare un modulo alla volta, evitando modifiche globali che potrebbero destabilizzare il sistema.
2. **Test di regressione**: dopo ogni modifica, verificare che le funzionalità esistenti continuino a operare come previsto.
3. **Documentare i cambiamenti**: tenere traccia di cosa viene spostato dove, facilitando la transizione per il team.
4. **Mantenere la compatibilità**: garantire che il comportamento esterno rimanga invariato, solo l'organizzazione interna cambia.

La rifattorizzazione è un investimento nella **qualità del codice** e nella **sostenibilità a lungo termine** del progetto, non solo un miglioramento estetico dell'organizzazione.

Ricorda: **la rifattorizzazione mantiene tutte le funzionalità esistenti**, migliorando solo la struttura interna del codice.

Con questa nuova organizzazione, PubliScript sarà più facile da estendere, mantenere e comprendere per tutti gli sviluppatori.