Gestione Preventivi EdilColor - Stima costi tinteggiatura, decorativi e cartongesso

Gruppo EdilColor Team

Documentazione di Progetto

Versione 1.0

Componenti del Gruppo:

- Nicola Avellino (243294)
- Riccardo Gonzato (246476)

3 gennaio 2026

Indice

Capitolo 1

Descrizione del Progetto

L'applicativo realizzato nasce per rispondere a una specifica esigenza operativa nel rapporto tra **committente** (cliente) e **professionista** (impresa).

Spesso, infatti, il cliente necessita unicamente di un ordine di grandezza immediato dei costi per valutare la fattibilità di un lavoro, mentre il professionista si trova a investire tempo prezioso in sopralluoghi e calcoli analitici anche per richieste che, frequentemente, non si traducono in commesse reali.

Il software risolve questa inefficienza strutturale offrendo un vantaggio reciproco immediato:

- **Per il Cliente:** Fornisce istantaneamente un'idea verosimile di spesa (*"stima preliminare"*), senza attese e senza l'impegno psicologico di richiedere un preventivo formale.
- **Per il Professionista:** Funge da *filtro preliminare*, automatizzando la fase di primo contatto. Questo permette all'impresa di dedicare le risorse umane e il tempo tecnico esclusivamente alle fasi esecutive o alle trattative già confermate.

1.1 Obiettivi del Progetto

Oltre alla risoluzione della problematica operativa, il progetto si pone precisi obiettivi tecnici e funzionali:

1. **Rapidità di Utilizzo:** Minimizzare il numero di input richiesti all'utente (soli mq e tipologia), delegando al sistema la complessità del calcolo dei materiali e della manodopera.
2. **Coerenza dei Prezzi:** Standardizzare le quotazioni applicando listini predefiniti e coefficienti di difficoltà (es. *locale abitato* vs *nuovo*) in modo automatico, eliminando la soggettività e l'errore umano tipico delle stime "a braccio".
3. **Architettura Estendibile (OOD):** Realizzare un sistema software basato su una solida progettazione a oggetti. L'uso di *Design Pattern* (come Builder e Strategy) garantisce che l'aggiunta futura di nuove categorie di lavori (es. pavimentazioni) non richieda la riscrittura del codice esistente, ma solo l'estensione delle classi base.

Capitolo 2

Requisiti del Sistema

In questo capitolo vengono dettagliati i requisiti funzionali (RF) e non funzionali (RNF) che hanno guidato lo sviluppo dell'applicazione.

2.1 Requisiti Funzionali

I requisiti funzionali descrivono i comportamenti specifici e le funzioni che il sistema mette a disposizione dell'utente.

- **RF1 - Gestione del Contesto Operativo (Difficoltà):** Il sistema deve permettere all'utente di definire lo stato dell'immobile (es. *Nuovo, Abitato, Disabitato*). Sulla base di questa scelta, l'applicazione deve applicare automaticamente dei **coefficienti correttivi** al listino base per adeguare il prezzo alla complessità logistica.
- **RF2 - Gestione Listini Centralizzata:** Il software deve mantenere un listino prezzi unico per tutte le lavorazioni. Deve essere possibile associare a ogni voce un prezzo unitario al metro quadro, garantendo che lo stesso tariffario venga applicato uniformemente a tutto il preventivo.
- **RF3 - Creazione e Composizione Preventivi:** L'utente deve poter creare un nuovo preventivo (associato a un ID univoco e a un cliente) e popolarlo aggiungendo diverse tipologie di voci di costo. Il sistema deve supportare l'inserimento di:
  - *Voci Semplici:* (es. Tinteggiature) che richiedono solo la superficie.
  - *Voci Complesse:* (es. Cartongesso) che richiedono una costruzione strutturata (struttura + lastre + finitura).
- **RF4 - Calcolo Automatico dei Totali:** Il sistema deve calcolare in tempo reale il costo di ogni singola voce (moltiplicando superficie $\times$ prezzo unitario $\times$ coefficiente difficoltà) e aggiornare dinamicamente il totale complessivo del preventivo.
- **RF5 - Persistenza e Esportazione Dati:** Al termine della sessione, il preventivo generato deve essere salvato su memoria di massa. Il sistema deve supportare l'esportazione in formati standard (CSV per l'analisi dati e TXT per la leggibilità) contenenti il dettaglio delle voci e il riepilogo finale.

- **RF6 - Interfaccia Utente (CLI):** L'interazione deve avvenire tramite interfaccia a riga di comando (Console), guidando l'utente attraverso menu numerici sequenziali per minimizzare gli errori di input.

2.2 Requisiti Non Funzionali

I requisiti non funzionali definiscono gli attributi di qualità del sistema e i vincoli progettuali.

- **RNF1 - Architettura Object-Oriented (OOD):** Il sistema deve essere progettato seguendo rigorosamente il paradigma a oggetti. Deve utilizzare il **Polimorfismo** per trattare in modo uniforme lavorazioni diverse (tramite classe astratta `VoceCosto`) e garantire l'incapsulamento dei dati.
- **RNF2 - Estendibilità (Open/Closed Principle):** L'architettura deve permettere l'aggiunta di nuove categorie di lavorazioni (es. Pavimentazioni) o nuove regole di calcolo senza dover modificare la logica del gestore dei preventivi, ma estendendo le classi esistenti.
- **RNF3 - Integrità dei Dati:** L'uso di puntatori intelligenti (`std::unique_ptr`, `std::shared_ptr`) è richiesto per garantire una corretta gestione della memoria (RAII), prevenendo memory leak durante la creazione e distruzione dei preventivi.
- **RNF4 - Performance:** Il calcolo del preventivo deve essere istantaneo al momento dell'inserimento dei dati, senza latenze percepibili dall'utente.

Capitolo 3

Attività Svolte

Lo sviluppo del progetto si è articolato in cinque fasi principali:

1. **Analisi del Dominio e dei Requisiti**
    - Studio delle dinamiche reali di preventivazione edile per individuare le entità fondamentali (Cliente, Preventivo, Voce di Costo, Listino).
    - Identificazione delle variabili critiche per il calcolo dei prezzi (superfici, coefficienti di difficoltà ambientale).
    - Formalizzazione dei requisiti funzionali (es. necessità di salvare su file) e non funzionali (es. estendibilità del codice).
2. **Progettazione Architetturale (OOD)**
    - Definizione della gerarchia delle classi tramite diagrammi UML.
    - Scelta delle strutture dati più idonee: utilizzo di `std::vector` per la gestione dinamica delle voci e `std::map` per l'accesso rapido ai listini prezzi.
    - Progettazione della gestione della memoria: decisione di utilizzare **Smart Pointers** (`std::unique_ptr`) per garantire la proprietà esclusiva delle voci all'interno del preventivo e prevenire memory leak.
3. **Sviluppo delle Classi Core**
    - Implementazione della classe astratta `VoceCosto` per definire l'interfaccia comune (metodo `subtotale()` virtuale puro).
    - Implementazione della classe container `Preventivo`, capace di gestire polimorficamente collezioni eterogenee di lavori.
    - Creazione del modulo `ListinoPrezzi` per centralizzare la gestione delle tariffe e dei coefficienti di maggiorazione.
4. **Implementazione dei Design Pattern**
    - **Pattern Builder (Predisposizione Architetturale):** Adozione di `VoceCartongessoBuilder` per la creazione della classe `VoceCartongesso`.

Sebbene nell'attuale versione del software la voce non presenti una struttura interna complessa, il Builder è stato implementato in ottica di **estendibilità futura**. L'obiettivo è predisporre il sistema ad accogliere configurazioni articolate (es. aggiunta di lastre ignifughe, isolanti acustici o orditure doppie) senza dover ristrutturare l'intero codice.

- **Pattern Strategy/Factory:** Implementazione delle classi `RegolaCosto` per disaccoppiare l'algoritmo di creazione e calcolo dal client, permettendo di selezionare la strategia corretta a runtime in base alla scelta dell'utente.

5. **Testing, Debugging e Refactoring**

   - Verifica del corretto funzionamento tramite casi d'uso reali (es. creazione di preventivi misti Cartongesso/Tinteggiatura).
   - Debugging della gestione dell'input utente (classe `GestioneInputUI`) per evitare crash su inserimenti non validi.
   - Test di persistenza: verifica della correttezza dei file CSV generati e della loro leggibilità.

Capitolo 4

Attività di Implementazione

In questo capitolo viene descritta l'architettura tecnica della soluzione, dettagliando le scelte di Object-Oriented Design (OOD), la mappatura tra casi d'uso e oggetti software, e le modalità di persistenza dei dati.

4.1 Object-Oriented Design (OOD)

L'architettura del software si basa su una rigorosa applicazione dei principi della programmazione a oggetti, sfruttando polimorfismo, incapsulamento e smart pointers per la gestione delle risorse.

Di seguito vengono descritte le classi principali, le loro responsabilità e le relazioni cardinali.

4.1.1 Descrizione delle Classi e delle Relazioni

Preventivo

È la classe "Container" principale del sistema. Rappresenta l'entità logica del preventivo associato a un cliente.

- **Proprietà:**
  - `id_ (string)`: Identificativo univoco del preventivo.
  - `cliente_ (string)`: Nome del committente.
  - `grado_ (GradoDifficolta)`: Enum che definisce lo stato dell'immobile (es. *Abitato*), influenzando i costi.
  - `voci_ (vector<unique_ptr<VoceCosto»)`: Collezione polimorfica delle lavorazioni.
- **Metodi Principali:**
  - `aggiungiVoce(unique_ptr<VoceCosto>)`: Inserisce una nuova lavorazione trasferendo la proprietà del puntatore (move semantics).
  - `totale()`: Itera sul vettore `voci_` invocando polimorficamente `subtotale()`.
  - `riepilogo()`: Genera una stringa formattata con il dettaglio di tutte le voci.
- **Relazioni:**

- **Composizione (1 a 0..*):** `Preventivo` possiede esclusivamente le istanze di `VoceCosto` tramite `std::unique_ptr`. La distruzione del preventivo comporta la distruzione delle voci.

VoceCosto (Classe Astratta)
Classe base che definisce l'interfaccia comune per qualsiasi tipo di lavorazione.

- **Proprietà (Protected):** `nome_`, `unitaMisura_`, `quantita_`, `prezzoUnitario_`, `coefficiente_`.
- **Metodi:**
  - `subtotale() (virtual pure)`: Metodo astratto che deve essere implementato dalle sottoclassi per calcolare il costo specifico.
  - `clone()`: Implementa il pattern Prototype per la duplicazione profonda degli oggetti.
- **Relazioni:**
  - **Generalizzazione:** Padre di `VoceTinteggiatura` e `VoceCartongesso`.

VoceTinteggiatura e VoceCartongesso
Classi concrete che implementano la logica specifica di calcolo.

- **Costruttore:** Riceve un riferimento a `ListinoPrezzi` per cercare il prezzo base in funzione del nome del ciclo selezionato e applica il coefficiente di difficoltà.
- **Relazioni:**
  - **Dipendenza:** Usano `ListinoPrezzi` e `GradoDifficolta` in fase di costruzione.

VoceCartongessoBuilder
Implementazione del **Pattern Builder** per la creazione controllata di voci complesse.

- **Proprietà:** Mantiene temporaneamente i parametri (`mq_`, `listino_`, ecc.) necessari alla costruzione.
- **Metodi:** `setMq()`, `setListino()`, `build()`. Il metodo `build()` restituisce un `unique_ptr<VoceCosto>` configurato.
- **Relazioni:**
  - **Associazione:** Mantiene uno `shared_ptr` al `ListinoPrezzi`.
  - **Creazione:** Istanzia oggetti `VoceCartongesso`.

CalcolatorePreventivo
Classe che orchestra la creazione delle voci utilizzando il **Pattern Strategy**.

- **Proprietà:** `regola_ (const RegolaCosto*)`: Puntatore alla strategia di creazione corrente.
- **Metodi:**
  - `setRegola()`: Cambia dinamicamente la strategia (es. da Tinteggiatura a Cartongesso).

- `aggiungiLavoro()`: Delega alla regola corrente la creazione della voce e la aggiunge al preventivo.

- **Relazioni:**
  - **Aggregazione (0..1):** Riferisce a una `RegolaCosto` senza possederla.

ListinoPrezzi
Database in-memory dei costi e dei coefficienti.

- **Struttura Dati:** Utilizza `std::map<string, double>` per associare i nomi dei cicli ai prezzi e `std::map<GradoDifficolta, double>` per i coefficienti.

4.2 Descrizione degli Use Case e Realizzazione

In questa sezione viene descritto come gli oggetti collaborano per realizzare le funzionalità principali.

4.2.1 UC1: Creazione di una Voce di Tinteggiatura

Descrizione: L'utente seleziona un ciclo di tinteggiatura e inserisce i mq.

- **Interazione Oggetti:**

1. `GestioneInputUI` acquisisce l'input utente (categoria e mq).
2. `CalcolatorePreventivo` viene configurato con la strategia `RegolaTinteggiatura`.
3. Il metodo `aggiungiLavoro()` invoca `RegolaTinteggiatura::creaVoce()`.
4. Viene istanziato un oggetto `VoceTinteggiatura` che interroga `ListinoPrezzi` per ottenere il costo unitario.
5. L'oggetto creato viene spostato (*move*) nel vettore di `Preventivo`.

4.2.2 UC2: Creazione di una Voce Cartongesso (Builder)

Descrizione: L'utente inserisce una lavorazione complessa in cartongesso.

- **Interazione Oggetti:**

1. `CalcolatorePreventivo` usa la strategia `RegolaCartongesso`.
2. La regola istanzia un `VoceCartongessoBuilder`.
3. Il Builder viene configurato passo-passo (`setMq`, `setNomeCiclo`, `setListino`).
4. Il metodo `build()` finalizza l'oggetto e lo restituisce al `Preventivo`.

4.2.3 UC3: Esportazione Preventivo

Descrizione: L'utente decide di salvare il preventivo su file.

- **Interazione Oggetti:**
  1. Il `main` invoca la classe statica `SalvataggioPreventivo`.
  2. Il metodo `salvaPreventivoSuTxt` richiede al `Preventivo` i dati tramite i getter e il metodo `riepilogo()`.
  3. Viene aperto uno stream `std::ofstream` e scritti i dati su disco.

4.3 Gestione Dati e Persistenza

L'applicazione non utilizza un DBMS relazionale, ma si appoggia al file system per la persistenza dei dati tramite file di testo strutturati.

4.3.1 Struttura dei File

La classe `SalvataggioPreventivo` gestisce due formati di output:

Formato TXT (Report Leggibile)
Pensato per la lettura umana. Struttura:

```
PREVENTIVO: [ID_Preventivo]
CLIENTE: [Nome_Cliente]
STATO IMMOBILE: [Nuovo/Abitato/...]
----------------------------------------
[Nome Voce] - [Mq] mq - [Prezzo] EUR
...
----------------------------------------
TOTALE: [Importo] EUR
```

Formato CSV (Interscambio Dati)
Pensato per l'importazione in fogli di calcolo (Excel). I campi sono separati da punto e virgola (;).

```
ID;Cliente;TipoLavoro;NomeCiclo;Mq;PrezzoUnitario;TotaleVoce
2023001;Mario Rossi;Tinteggiatura;Lavabile Interna;50;10.5;525.0
...
```

Il **Listino Prezzi** è invece hard-coded e inizializzato all'avvio tramite il modulo `ListinoDefault`, che popola le mappe della classe `ListinoPrezzi` con i valori base definiti nel codice sorgente.

4.4 Modalità di Interazione (I/O)

L'interazione con l'utente è di tipo testuale (CLI - Command Line Interface) ed è gestita dalla classe `GestioneInputUI`.

4.4.1 Flusso di Interazione

Il sistema presenta un menu interattivo ciclico:

1. **Input Iniziale:** Richiesta nome cliente e stato immobile (selezione numerica 1-3).
2. **Menu Principale (Loop):**

```
1. Aggiungi Tinteggiatura
2. Aggiungi Cartongesso
3. Visualizza Riepilogo
4. Salva ed Esci
```

3. **Sottomenu Categorie:** In base alla scelta (es. Tinteggiatura), viene mostrato l'elenco dei cicli disponibili prelevati da `CatalogoCicli` (es. "1. Traspirante", "2. Lavabile").
4. **Input Quantitativo:** Richiesta della superficie in mq (con validazione per impedire valori negativi).
5. **Output Finale:** Messaggio di conferma salvataggio e chiusura.

La classe `Utils` fornisce funzioni di supporto come `clearScreen()` e `pause()` per migliorare l'usabilità dell'interfaccia testuale.

Capitolo 5

Dettagli Tecnici Aggiuntivi

5.1 Librerie Integrate

Per garantire la massima portabilità e ridurre le dipendenze esterne, il progetto è stato sviluppato utilizzando esclusivamente la **Libreria Standard C++ (STL)**. Non sono state integrate librerie di terze parti (come Boost o Qt).

Di seguito il dettaglio delle librerie standard utilizzate e del loro scopo nel progetto:

`<iostream>` **e** `<iomanip>`
: Utilizzate per la gestione dell'Input/Output su console. In particolare, `<iomanip>` è fondamentale per la formattazione monetaria (es. `std::fixed`, `std::setprecision(2)`) per mostrare i prezzi con due cifre decimali.

`<fstream>`
: Utilizzata dalla classe `SalvataggioPreventivo` per le operazioni di I/O su file (scrittura dei report TXT e CSV).

`<memory>`
: Cruciale per la gestione moderna della memoria. Fornisce i template per gli smart pointers:

  - `std::unique_ptr`: Usato nel vettore di `Preventivo` per il possesso esclusivo delle voci.
  - `std::shared_ptr`: Usato per condividere l'istanza del `ListinoPrezzi` tra le varie componenti (Builder, Regole) senza duplicarla.

`<vector>` **e** `<map>`
: Implementano i container dati dinamici.

  - `std::map`: Scelta algoritmica precisa per il `ListinoPrezzi`. Garantisce una ricerca dei prezzi con complessità logaritmica $O(\log n)$ basata su chiavi stringa, molto più efficiente di una scansione lineare su array.

`<algorithm>`
: Inclusa per l'utilizzo di algoritmi generici, in particolare `std::sort`, utilizzato per ordinare le voci del preventivo.

`<limits>`
: Utilizzata nella classe `GestioneInputUI` per pulire il buffer di input (`std::numeric_limits`) e rendere robusta l'interfaccia contro inserimenti errati dell'utente.

5.2 Algoritmi Utilizzati

Sebbene il software non implementi algoritmi di calcolo scientifico complessi, fa uso di pattern algoritmici strutturali e di manipolazione dati:

1. **Ordinamento (Sorting Customizzato):** Nel metodo `Preventivo::ordinaPerNome()`, viene utilizzato l'algoritmo `std::sort`. È stata implementata una *Lambda Function* come comparatore personalizzato per ordinare i puntatori `unique_ptr<VoceCosto>` in base al nome della voce puntata (dereferenziazione e confronto di stringhe).
   Complessità media: $O(n \log n)$.

2. **Iterazione Polimorfica:** L'algoritmo di calcolo del totale (`Preventivo::totale()`) si basa sull'iterazione lineare su un container eterogeneo. Il sistema sfrutta il *Dynamic Dispatch* (binding dinamico) per invocare la versione corretta del metodo `subtotale()` a runtime, a seconda che l'oggetto sia una Tinteggiatura o un Cartongesso.
   Complessità: $O(n)$.

3. **Ricerca Associativa:** Per il recupero dei prezzi dal listino, non viene usata una ricerca sequenziale, ma una ricerca associativa su albero bilanciato (struttura interna di `std::map`). Questo permette di mantenere alte prestazioni anche qualora il listino dovesse contenere migliaia di voci.
   Complessità: $O(\log n)$.

4. **Validazione Input (Robustezza):** È stato implementato un algoritmo di validazione ciclica (`do-while`) nella classe `GestioneInputUI`. L'algoritmo controlla lo stato dello stream `cin`: in caso di fallimento (es. inserimento di una lettera dove atteso un numero), ripristina lo stato dello stream, svuota il buffer e richiede nuovamente l'input, prevenendo il loop infinito tipico dei programmi C++ base.

Bibliografia

[1] Prof. E. Blanzieri, Prof. P. Roberti. *Materiale didattico del corso di Programmazione Avanzata*. Università di Trento, Anno Accademico 2025/2026.

[2] CppReference. *C++ Standard Library documentation (std::unique_ptr, std::map)*. Disponibile su: `https://en.cppreference.com/` (Consultato per la gestione della memoria e dei container STL).

[3] Refactoring.Guru. *Design Patterns: Builder, Strategy and Factory Method*. Disponibile su: `https://refactoring.guru/design-patterns/cpp` (Riferimento per l'implementazione architetturale degli oggetti).

[4] OpenAI. *ChatGPT (Modello GPT-4)*. Utilizzato come supporto per la revisione del codice. 2025.