Relazione progetto Programmazione a Oggetti

Sofia De Blasi
Matricola: 2111014
Nails Products Library

Introduzione

QtNailsProject è un'applicazione per la gestione di un catalogo di prodotti per la Nail Art che permette di creare, modificare, cancellare e visualizzare i propri prodotti (CRUD). I prodotti che è possibile gestire sono di diversa tipologia: come smalti semipermanenti, basi/top coat e gel costruttori, ciascuno con le proprie caratteristiche e visualizzazione grafica che si adatta dinamicamente al tipo di prodotto selezionato.

Uno dei punti di forza del progetto è il sistema di filtraggio, che permette di cercare i prodotti sia tramite la barra di ricerca sia in base a criteri mirati, come tipologia, colore, finish, elasticità, viscosità e altri parametri tecnici. Poiché tali attributi sono distinti per tipologia, la selezione di un filtro specifico comporta automaticamente la scelta del corrispondente tipo di prodotto.

La persistenza dei dati è garantita tramite salvataggio e caricamento in formato JSON, implementati con il supporto del Visitor Pattern. Questa soluzione assicura portabilità e semplicità di gestione, permettendo di salvare e ricaricare interi cataloghi in pochi passaggi.

Ho scelto questo progetto perché il mondo della Nail Art è un mio hobby personale, e conosco abbastanza bene le caratteristiche comuni e le differenze tra i vari prodotti. Questo mi ha permesso di analizzarli con maggiore consapevolezza e di immedesimarmi nell'utente finale, progettando un'interfaccia che, pur essendo semplice, risulta intuitiva e soddisfacente nell'utilizzo.

Descrizione del modello

Il modello logico del progetto QtNailsProject è stato progettato per gestire un catalogo di prodotti per la Nail Art, con particolare attenzione alla differenziazione tra tipologie di prodotto e alla possibilità di effettuare ricerche e filtraggi avanzati.

Come mostrato nel diagramma UML in Figura 1,la struttura si articola in tre componenti principali: la gerarchia dei prodotti, la classe `Catalogo` che li gestisce, e le classi per la persistenza e la visualizzazione.

0.1 Gerarchia

Alla base del modello vi è la classe astratta `NailsProduct`, che rappresenta un prodotto generico e definisce gli attributi comuni come nome, immagine, formato e tempo di polimerizzazione; da essa derivano tre sottoclassi concrete: `SmaltoSemip`, `TopBase` e `GelCostruttore` (tutte e quattro presentano i metodi `getter` e `setter` con controllo di validità dove necessario).

Questa struttura consente di modellare in maniera fedele le differenze tra i vari prodotti reali, mantenendo allo stesso tempo una base comune riutilizzabile. La scelta di utilizzare una classe astratta come radice della gerarchia nasce dall'esigenza di garantire l'estendibilità del sistema

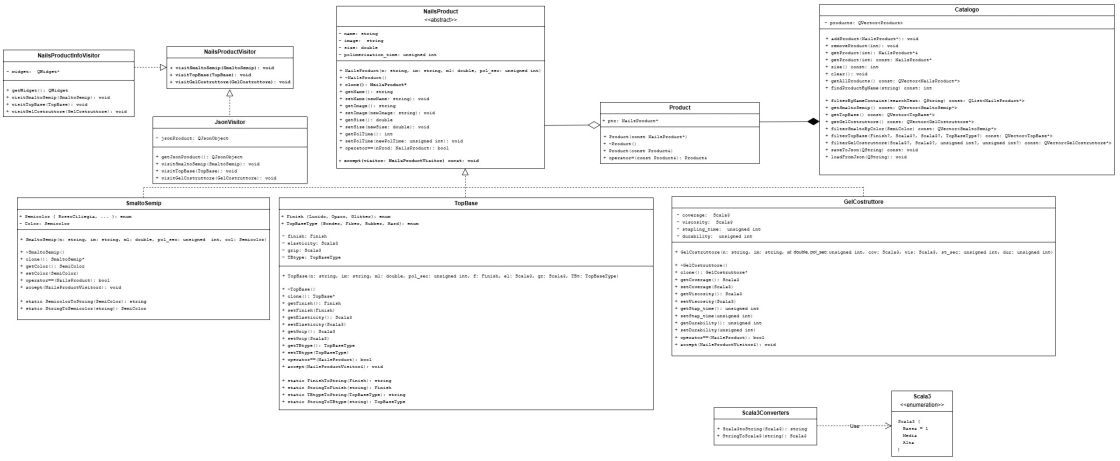

Figure 1: Diagramma UML del modello logico

infatti l'aggiunta di nuovi tipi di prodotto richiede la sola creazione di una nuova sottoclasse senza modificare il codice già esistente.

Le classi della gerarchia usufruiscono anche di classi di tipo `enumeration` esterne, come `Scala3`, in modo da essere riutilizzabile o interne, come `Semicolor`, `Finish` e `TopBaseType`, in quanto proprie della classe in cui vengono usate. Per ognuna di queste è incluso nel modello anche la gestione delle conversioni da enumeratori a stringhe (e viceversa) attraverso classi di supporto come `Scala3Converters` o metodi interni alle classi.

0.2 Contenitore

La classe `Catalogo` funge da contenitore polimorfo per i prodotti, utilizzando una classe annidata di incapsulamento `Product` che costituisce una classe di puntatori smart (super)polimorfi a `NailsProduct`.

Questo approccio consente di gestire copie e assegnazioni profonde tramite il metodo `clone()` ed evitare memory leak.

Inoltre, il catalogo offre metodi per aggiungere, rimuovere, visualizzare i prodotti attraverso metodi che estraggono il puntatore incapsulato in `Product`, metodi di filtraggio per tipo che usufruiscono di un template di metodo `getCategory()` che permette il riuso del codice e un supporto per la lettura e scrittura da file JSON.

0.3 Persistenza e visualizzazione

Per gestire la persistenza dei dati è stato utilizzato il Visitor Pattern: ogni sottoclasse di prodotto definisce come serializzarsi in JSON, mentre la classe `JsonHandler` si occupa di salvare e caricare l'intero catalogo, delegando la costruzione dei `QJsonObject` a `JsonVisitor`. In questo modo la logica di salvataggio non deve conoscere i dettagli di ciascun tipo di prodotto. Un ulteriore visitor, `NailsProductInfoVisitor`, permette di generare dinamicamente un widget Qt con le informazioni specifiche di ogni prodotto, garantendo modularità e separazione tra logica e interfaccia.

1 Polimorfismo

Il polimorfismo nel progetto è introdotto tramite la gerarchia NailsProduct e il Visitor Pattern.

1.1 Polimorfismo in Gerarchia

La classe astratta `NailsProduct` definisce gli attributi comuni a tutti i prodotti (nome, immagine, formato e tempo di polimerizzazione) e dichiara i metodi virtuali puri `clone()` e `accept()`. Le sottoclassi implementano i metodi specifici e sovrascrivono l'(`operator==`) per confrontare oggetti dello stesso tipo dinamico.

Clone polimorfo Il metodo `clone()` è virtuale in `NailsProduct` e implementato in ciascuna sottoclasse. Questo consente di ottenere copie profonde dei prodotti mantenendo il tipo dinamico corretto, evitando slicing, facilitando la gestione dei puntatori nel catalogo e garantendo l'integrità dei dati durante salvataggio, modifica o filtraggio.

Operatore di uguaglianza Il polimorfismo viene anche utilizzato nei test di ugualianza tra prodotti infatti vi è un overload dell'operatore di uguaglianza `operator==`, che viene ridefinito in ciascuna sottoclasse per garantire che due oggetti siano considerati uguali solo se:

1. Hanno lo stesso tipo dinamico.
2. Tutti gli attributi specifici della sottoclasse corrispondono.

In questo modo è possibile confrontare due puntatori a `NailsProduct` senza conoscere a priori il tipo concreto, evitando cast non sicuri o controlli esterni sul tipo dinamico, e mantenendo la coerenza del modello.

Il polimorfismo è quindi utilizzato non solo per l'ereditarietà di base, ma per realizzare confronti sicuri e coerenti tra oggetti di tipo dinamico diverso all'interno della stessa gerarchia.

1.2 Polimorfismo in Visitor

Per separare la logica di elaborazione dei dati dalla struttura dei prodotti, il progetto utilizza il `Visitor Pattern`. La gerarchia `NailsProduct` dichiara il metodo virtuale puro `accept()`, che ogni sottoclasse ridefinisce per accettare un `NailsProductVisitor`. Il visitor definisce le funzioni virtuali:

```
1 virtual void visitSmaltoSemip(const SmaltoSemip& prod) = 0;
2 virtual void visitTopBase(const TopBase& prod) = 0;
3 virtual void visitGelCostruttore(const GelCostruttore& prod) = 0;
```

Ogni prodotto implementa `accept()` richiamando il metodo appropriato del visitor, permettendo operazioni polimorfe senza controlli espliciti sul tipo concreto.

Applicazioni concrete

- **Serializzazione JSON**: `JsonVisitor` implementa `visitSmaltoSemip`, `visitTopBase` e `visitGelCostruttore`, costruendo un `QJsonObject` contenente i dati specifici di ciascun tipo di prodotto. Grazie al polimorfismo, la logica di salvataggio e caricamento nel `Catalogo` può iterare su puntatori a `NailsProduct` senza conoscere i dettagli delle sottoclassi.

- **Creazione dinamica di oggetti**: la funzione statica `createProductFromJson` di `JsonHandler` sfrutta il tipo salvato in JSON per costruire l'oggetto corretto, invocando il clone polimorfo, evitando cast manuali o controlli esterni sul tipo.
- **Generazione dinamica di widget Qt**: `NailsProductInfoVisitor` costruisce dinamicamente widget QWidget differenziati per ciascun tipo di prodotto, incapsulando le informazioni specifiche in layout e `QLabel`. Anche in questo caso, l'invocazione polimorfa di `accept()` seleziona automaticamente il metodo corretto del visitor.

2 Persistenza dei dati

La persistenza dei dati nel progetto è realizzata tramite il formato `JSON`, che consente di memorizzare e recuperare le informazioni relative ai prodotti del catalogo.

2.1 Struttura dei file

Ogni catalogo è rappresentato da un singolo file `.json`, che contiene un array di oggetti. Ciascun oggetto rappresenta un prodotto e raccoglie i suoi attributi (`nome`, `formato`, `immagine`, ecc.).

Per distinguere le diverse sottoclassi della gerarchia `NailsProduct`, la serializzazione include un campo aggiuntivo `‘‘type’’`, che specifica la tipologia del prodotto (ad esempio `"SmaltoSemip"`, `"TopBase"`, `"GelCostruttore"`). In fase di deserializzazione, tale informazione consente di ricostruire correttamente il tipo dinamico del prodotto senza la necessità di cast manuali.

2.2 Serializzazione e Deserializzazione

La gestione della persistenza è incapsulata in un componente dedicato:

- **Serializzazione**: viene realizzata tramite il `JsonVisitor`, che implementa i metodi `visit...()` per ciascun tipo di prodotto, costruendo un oggetto `QJsonObject` con i dati specifici.
- **Deserializzazione**: è gestita da `JsonHandler`, che interpreta il campo `‘‘type’’` e richiama i costruttori appropriati delle sottoclassi per ricostruire l'oggetto corretto.

In questo modo la logica di persistenza è indipendente dalla logica applicativa e sfrutta il polimorfismo per trattare uniformemente oggetti eterogenei.

2.3 File di esempio

Per illustrare la struttura dei file, insieme al codice è fornito il file `catalogo.json`, che contiene un po' di prodotti per tipologia ed essi si distinguono tra loro per attributi specifici, permettendo di testare efficacemente le funzionalità di filtraggio.

Un esempio di struttura per tipologia è

```
[
    {
        "color": "RossoCiliegia",
        "image": "./images/RossoCiliegia.jpg",
        "name": "SmaltoRossoCiliegia",
        "polimerization_time": 60,
        "size": 15,
        "type": "SmaltoSemip"
    },
```

```
    {
        "elasticity": "Media",
        "finish": "Lucido",
        "grip": "Media",
        "image": "./images/TopBase.jpg",
        "name": "TopBaseLucido",
        "polimerization_time": 60,
        "size": 15,
        "tbtype": "Bonder",
        "type": "TopBase"
    },
    {
        "coverage": "Bassa",
        "durability": 3,
        "image": "./images/Gel1.jpg",
        "name": "GelCostruttore1",
        "polimerization_time": 60,
        "size": 15,
        "stapling_time": 60,
        "type": "GelCostruttore",
        "viscosity": "Bassa"
    }
]
```

3 Funzionalità implementate

Le funzionalità del sistema si dividono in **funzionalità principali** e **funzionalità di supporto ed estetiche**.

3.1 Funzionalità principali

- **Gestione catalogo**: i prodotti sono caricati da file `.json` all'avvio dell'applicazione e vengono salvati automaticamente ad ogni operazione di modifica, inserimento o rimozione.
- **Inserimento e modifica prodotti**: tramite un'apposita finestra di dialogo è possibile aggiungere o modificare prodotti al catalogo. Durante l'inserimento vengono effettuati controlli di validità e, in caso di errori, vengono mostrati opportuni messaggi di avviso. Mentre la modifica avviene su prodotti già esistenti che possono essere aggiornati e volta confermate le modifiche, il catalogo viene salvato e la vista aggiornata in tempo reale.
- **Rimozione prodotti**: l'utente può eliminare un prodotto selezionato dal catalogo, previa conferma tramite finestra di dialogo e anche per la rimozione la vista aggiornata in tempo reale. Qualora il catalogo risulti vuoto, viene visualizzato un messaggio placeholder informativo.
- **Ricerca e filtri combinati**: il sistema consente di ricercare i prodotti per nome attraverso la barra di ricerca e di applicare filtri. Per quanto riguarda questi ultimi è possibile selezionale la tipologia, mentre se viene "attivato" un filtro di un attributo specifico di tipologia, il filtraggio per tipologia avviene automaticamente.

- **Visualizzazione dettagliata**: selezionando un prodotto dall'elenco viene mostrata, nella parte destra della finestra, una scheda completa con tutte le informazioni relative all'articolo scelto e tale pannello è specifico per ogni tipologia.

3.2 Funzionalità di supporto ed estetiche

- **Aggiornamento dinamico dell'interfaccia**: la sezione destra della finestra si aggiorna automaticamente in base alla selezione, sostituendo il placeholder con la scheda del prodotto.
- **Gestione placeholder**: quando nessun prodotto è selezionato oppure il catalogo è vuoto, viene mostrato un messaggio informativo al posto della scheda.
- **Messaggi di conferma e avviso**: in caso di rimozione o di errori durante l'inserimento o in caso di catalogo vuoto, il sistema notifica l'utente tramite finestre di dialogo dedicate.
- **Interfaccia a due pannelli**: la finestra principale è strutturata in due sezioni: a sinistra sono presenti ricerca, filtri ed elenco dei prodotti con lista a scorrimento, mentre a destra vengono visualizzati i dettagli del prodotto selezionato e le opzioni di modifica o rimozione.
- **Stile e tematizzazione**: l'interfaccia grafica utilizza un file `.qss` incluso nel progetto tramite un file di risorse `.qrc`, che applica uno stile uniforme ai widget (colori, font, bordi, effetti hover, ecc.) evitando così ridondanza di codice di styling.

  Alcuni esempi di funzionalità visive:

  – la selezione dei filtri nelle `QComboBox` si evidenzia scurendo lo sfondo dell'elemento selezionato,

  – i pulsanti cambiano colore al passaggio del mouse,

  – la lista dei prodotti mostra con evidenza l'elemento selezionato.

4 Rendicontazione ore

Attività	**Ore Previste**	**Ore Effettive**
Studio e progettazione	10	11
Sviluppo del codice del modello	10	12
Studio del framework Qt	10	15
Sviluppo del codice della GUI	10	16
Test e debug	5	7
Stesura della relazione	5	6
Totale	50	67

Table 1: Rendicontazione ore per le attività del progetto

Il monte ore è risultato leggermente superiore alle previsioni, principalmente a causa del tempo necessario per effettuare test approfonditi e debug, ottimizzare le funzionalità di inserimento, modifica e rimozione dei prodotti, e garantire che filtri e aggiornamento della vista funzionassero sempre in modo fluido e in tempo reale.