|  |  |
| --- | --- |
| **oggetto** | Relazione progetto Programmazione a Oggetti |
| **gruppo** | Cognome Nome, mat. 0000000  Cognome Nome, mat. 0000001 |
| **titolo** | Search Engine |

# Introduzione

*Questo modello offre un esempio di relazione per il progetto di Programmazione a Oggetti. I paragrafi scritti in corsivo, come questo, illustrano cosa ci si aspetta di trovare in ciascuna sezione o suggerimenti sui contenuti da scrivere, e vanno rimosse durante la stesura della relazione. I paragrafi con* testo normale *offrono invece un esempio dei reali contenuti di una relazione.*

*L’introduzione dovrebbe offrire una veloce panoramica del progetto, affinché il valutatore sia in grado di comprenderne la natura e lo scopo. Una buona introduzione comprende circa 200 parole e non necessita di immagini o illustrazioni.*

Search Engine è un motore di ricerca per siti web di e-commerce che permette di creare, modificare, cancellare e visualizzare i propri prodotti. I prodotti che è possibile gestire sono di diversa tipologia: prodotti semplici, virtuali, bundle e pagine web, ciascuno con le proprie caratteristiche e visualizzazione grafica.

Il principale punto di forza è il prototipo di motore di ricerca realizzato da zero prendendo a modello motori reali come Solr o ElasticSearch. Sebbene minimale, offre un esempio delle principali funzionalità di questi ultimi, come la possibilità di calcolare la pertinenza tra una chiave di ricerca e un prodotto, la possibilità di filtrare (ovvero escludere prodotti senza che questo influenzi la pertinenza) e gestire la paginazione. Il calcolo della pertinenza, in particolare, utilizza attributi diversi in base al tipo di prodotto considerato, per esempio in un prodotto semplice viene considerato il titolo, mentre in un bundle occorre considerare i titoli dei singoli prodotti che lo compongono.

Ho scelto questo progetto poiché ho una certa esperienza pregressa in ambito di e-commerce e motori di ricerca che ho pensato di sfruttare. Ho inoltre pensato che le diverse tipologie di prodotti potessero essere un buon pretesto per utilizzare il polimorfismo in maniera non banale, come richiesto nelle specifiche.

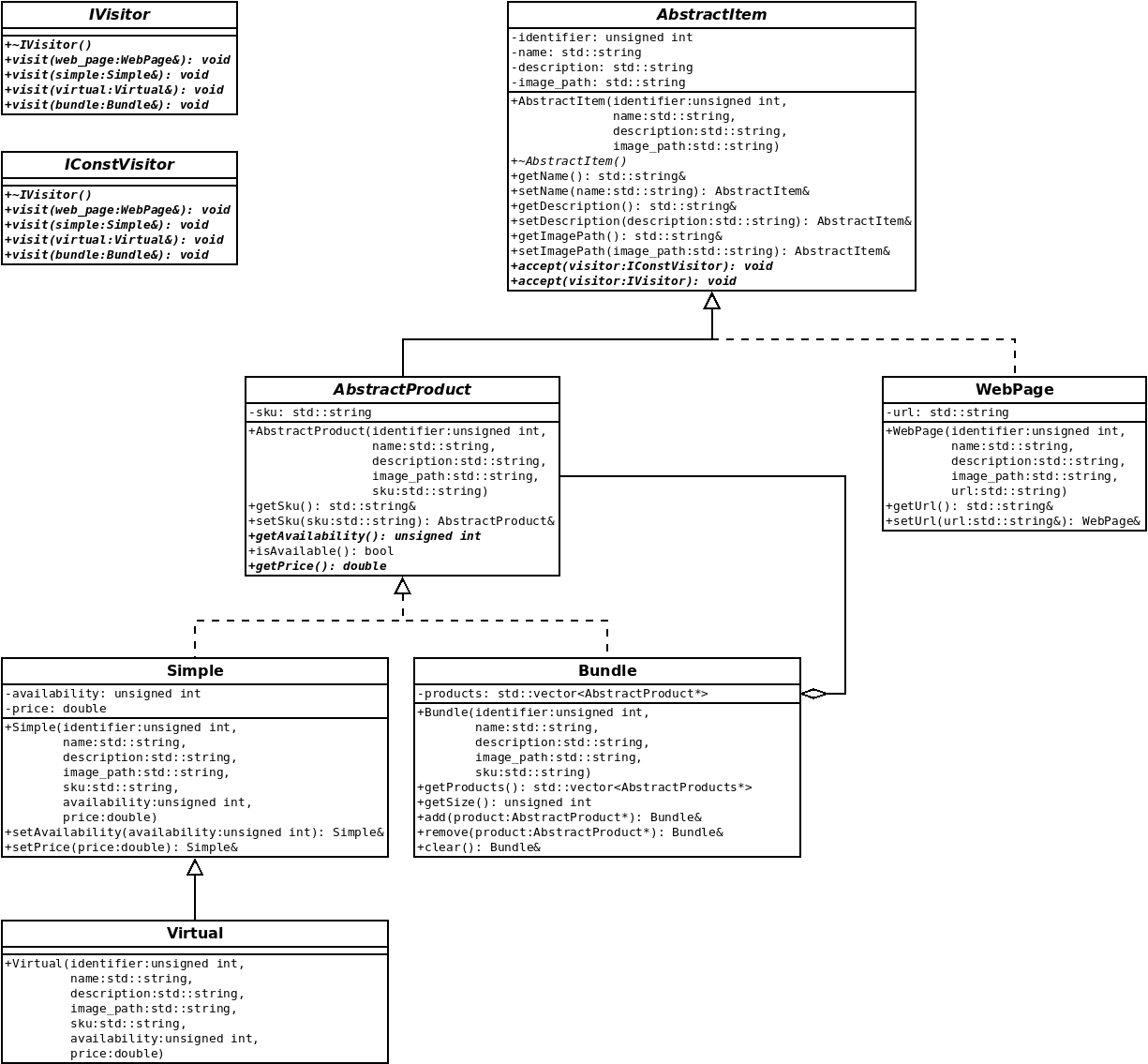
# Descrizione del modello

*Questa sezione descrive le principali classi che compongono il modello logico del progetto realizzato, soffermandosi in particolare sulle gerarchie e sull’uso dell’ereditarietà. Una buona descrizione parte da un diagramma UML delle classi e comprende un commento discorsivo sulle scelte architetturali e implementative.*

*Da evitare un mero elenco delle classi implementate e relativi metodi: quest’informazione è già riportata nel diagramma delle classi e, il più delle volte, una scelta ragionevole dei nomi degli identificatori è sufficiente, per esempio è ovvio che il metodo “getName” della classe “AbstractItem” restituisca il nome di un oggetto. È invece utile soffermarsi sugli aspetti specifici o settoriali legati al progetto, per esempio Search Engine è un motore di ricerca per l’e-commerce, e prevede una gerarchia di AbstractItem che comprende oggetti “semplici”, “virtuali” e “bundle”, oltre a pagine web. La scelta di questa particolare struttura è motivata da uno studio preliminare sulle strutture dati utilizzate dai siti di e-commerce, ed è bene che quest’informazione sia spiegata nella relazione.*

Il modello logico si articola in due parti: la gestione dei tipi di prodotto cercabili e il motore di ricerca vero e proprio. La prima comprende sia le classi che descrivono i prodotti, riportate nel diagramma in Figura 1, sia alcune classi di servizio per convertire i prodotti nel formato JSON (e viceversa) e salvarli su file. A tale scopo vengono utilizzati gli strumenti offerti da Qt, in particolare il *QjsonObject*.

Il modello parte da una classe astratta *AbstractItem* che rappresenta le informazioni comuni a tutti gli oggetti cercabili, ovvero identificatore univoco, nome, descrizione e percorso dell’immagine, per ciascuno dei quali sono implementati metodi *getter* e *setter*. La classe concreta *WebPage* rappresenta una pagina web e può essere utilizzata per realizzare un motore di ricerca per articoli all’interno di un blog o pagine di categorie di prodotti. La sotto-gerarchia *AbstractProduct* rappresenta prodotti veri e propri, i quali si articolano a loro volta nelle tre tipologie *Simple*, *Virtual* e *Bundle*. I prodotti semplici rappresentano i classici articoli acquistabili singolarmente, mentre quelli virtuali ne sono un caso particolare in quanto rappresentano prodotti scaricabili o ottenibili in altro modo (per esempio chiavi di attivazione, abbonamenti o servizi multimediali). I prodotti *Bundle* rappresentano invece collezioni di *AbstractProduct* venduti insieme. Tutti i prodotti hanno uno SKU, ovvero un ulteriore identificatore univoco testuale, una disponibilità e un prezzo, questi ultimi calcolati diversamente in base al tipo concreto: per i prodotti semplici (e virtuali) vengono utilizzati attributi appositi, mentre la disponibilità di un *bundle* è data dal minimo delle disponibilità dei prodotti che ne fanno parte (è sufficiente che un componente sia esaurito affinché il *bundle* non sia disponibile), così come il prezzo è la somma dei prezzi dei componenti. Per realizzare in maniera efficace questo comportamento dei *bundle* si è fatto riferimento al *design pattern Composite*.

Figura 1: Diagramma delle classi del modello

Poiché le classi del modello si comportano come dei *Data Transfer Object* (DTO) e non espongono alcuna funzionalità rilevante, si è scelto di utilizzare il *design pattern Visitor* per consentirne l’arricchimento in maniera dinamica. A tal fine sono state realizzate le classi astratte *IVisitor* e *IConstVisitor*, le quali differiscono unicamente per il fatto che la seconda lascia necessariamente inalterato l’oggetto visitato, ovvero consente di sfruttare la *const correctness* offerta dal compilatore. Di conseguenza, in *AbstractItem* sono stati inseriti i metodi virtuali puri *accept* per accettare i due tipi di *Visitor*.

Vi è inoltre la sezione relativa al funzionamento del motore di ricerca, dalla struttura più semplice. Un motore di ricerca *IEngine* si comporta come un contenitore nel quale è possibile aggiungere, rimuovere e cercare prodotti. La ricerca avviene attraverso un oggetto *Query*, che contiene informazioni relative alle parole cercate, eventuali filtri, e la paginazione (gestita come una coppia *offset* e *dimensione*). La ricerca avviene scorrendo tutti i prodotti presenti nel motore, filtrando quelli che non rispettano il filtro indicato da *Query* e assegnando ai rimanenti un punteggio di pertinenza sulla base delle parole cercate in *Query*. Gli oggetti così ottenuti vengono ordinati sulla base del punteggio di pertinenza e, utilizzando le informazioni relative alla paginazione, la porzione interessata viene restituita.

Poiché durante la ricerca e la visualizzazione dei risultati è utile conoscere il punteggio di pertinenza, così come avere la possibilità di ordinare gli oggetti rispetto a esso, gli *AbtractItem* sono decorati dalla classe *ItemDecorator*, che realizza il *design pattern Decorator* arricchendo il soggetto con un punteggio e l’*overloading* dell’operatore *<*.

Il test di soddisfacimento di un filtro e il calcolo del punteggio di pertinenza sono individuali per ogni oggetto, ovvero non si tiene conto della relazione con gli altri oggetti presenti nel motore. A titolo di esempio, nei motori di ricerca reali, la corrispondenza di una parola comune a molti i prodotti comporta un punteggio di pertinenza basso, mentre una parola molto specifica presente in pochi prodotti contribuisce generalmente in maniera maggiore. Per realizzare quest’effetto è necessario che il punteggio di pertinenza di un prodotto sia influenzato da tutti gli altri prodotti: per semplificare il progetto queste informazioni relazionali non vengono considerate, e il punteggio di un articolo dipende unicamente dal contenuto dei suoi attributi.

Resta tuttavia necessario applicare strategie diverse a seconda del tipo concreto: nelle pagine web si può cercare una corrispondenza dentro all’URL, nei prodotti semplici e virtuali all’interno di titoli, SKU e descrizioni, mentre dei *bundle* la ricerca deve considerare ricorsivamente i prodotti contenuti (effetto semplice da ottenere grazie al *design pattern Composite*). Per ottenere questo risultato si è fatto ricorso al *IConstVisitor* descritto nella sezione del modello, realizzando un *visitor* per il calcolo del punteggio di pertinenza e uno per il calcolo del soddisfacimento di un filtro.

## Polimorfismo

*Questa sezione descrive l’utilizzo non banale del polimorfismo. Per “banale” si intende un uso del polimorfismo scontato o imposto dal linguaggio C++, per esempio il rendere virtuali i distruttori in una superclasse, oppure definire metodi virtuali per gestire piccole variazioni del comportamento di una sottoclasse: non è necessario documentare questi utilizzi.*

*Un utilizzo più profondo e consapevole è generalmente offerto dai Design Patterns. Per esempio Search Engine utilizza dei Visitor per estendere dinamicamente le funzionalità delle proprie classi.*

*Da evitare assolutamente metodi virtuali come “getType” che restituiscono stringhe rappresentanti il tipo dell’oggetto d’invocazione: strumenti di questo tipo sono un modo per aggirare il polimorfismo e sono generalmente un segnale di progettazione errata.*

L’utilizzo principale del polimorfismo riguarda il *design pattern Visitor* nella gerarchia *AbstractItem*. Esso viene utilizzato per il calcolo del punteggio di pertinenza e soddisfacimento dei filtri durante le ricerche, per la conversione in JSON, e per la costruzione dei *widget* per mostrare le diverse tipologie di elementi. Rispetto a quest’ultimo punto, la visualizzazione prevede quattro possibili rese grafiche per ciascuno dei quattro prodotti, per un totale di sedici combinazioni. Le rese grafiche possibili sono:

* *ListItem*: un’anteprima compatta che ricorda un elemento di una lista, di altezza contenuta, pensato per essere utilizzato in un elenco di risultati
* *Card*: una visualizzazione rettangolare di un oggetto, più ricca ed estesa rispetto a *ListItem*, pensata per una visualizzazione a griglia
* *Teaser*: simile a una *Card,* ma con maggior enfasi sull’immagine e il titolo dell’articolo, pensato per articoli in evidenza
* *Full*: una vista a pagina intera, rappresenta la classica pagina con le informazioni del prodotto

Per ognuna di queste tipologie è stato implementato un *IConstVisitor* che costruisce un *widget* in base al tipo concreto dell’oggetto visitato.

Un ulteriore utilizzo del polimorfismo è dato dal *design pattern Strategy* impiegato nella scelta della visualizzazione: è possibile scegliere se mostrare i risultati di una ricerca come elenco, griglia oppure con un layout più elaborato che pone i primi prodotti in evidenza. Questa è una configurazione estetica che non ha alcun effetto sui risultati della ricerca, ed è stata implementata realizzando tre diverse *IresultRenderStrategy.*

L’utilizzo del *design pattern Composite* per la gestione dei *bundle* comporta a sua volta l’uso del polimorfismo per gestire il calcolo della disponibilità e del prezzo, come descritto in precedenza.

Infine vi è un utilizzo del polimorfismo di minor rilevanza citato per completezza: la classe astratta *IEngine* si basa sul *design pattern Strategy* in quanto la *member function* *search* è virtuale pura e si apre a diverse possibili strategie di ricerca. Tuttavia in questo progetto ne è stata implementata solamente una, quella descritta in precedenza, rendendo l’utilizzo del polimorfismo poco significativo in questo frangente, sebbene questa scelta renda il codice facile da estendere in futuro.

# Persistenza dei dati

*Questa sezione spiega se e come i dati vengono fatti persistere sul file system o altre tecnologie. Non è generalmente necessario entrare nel dettaglio del formato dei file o della struttura delle tabelle SQL, ma è gradita l’aggiunta di un file d’esempio o di uno schema tra i file del progetto, menzionandoli nella relazione.*

Per la persistenza dei dati viene utilizzato il formato JSON, un unico file per catalogo di prodotti, contenente un vettore di oggetti. Gli oggetti sono perlopiù semplici associazioni chiave-valore, e la serializzazione delle sottoclassi viene gestita aggiungendo un attributo “type”. Un esempio della struttura dei file è dato dai JSON forniti assieme al codice, in particolare “small.json” contiene un prodotto per ciascuna tipologia, in modo da illustrare brevemente le diverse strutture.

# **Funzionalità implementate**

*Una GUI ben progettata rende ovvio e intuitivo l’uso delle funzionalità del programma, tuttavia possono esserci alcune peculiarità non immediatamente visibili che è bene documentare. Search Engine, per esempio, implementa scorciatoie da tastiera, e chiede conferma all’uscita nel caso in cui vi siano modifiche non salvate: queste funzionalità non sono immediatamente visibili e vanno indicate. Search Engine offre inoltre la possibilità di modificare il layout della scheda dei risultati tramite una pulsantiera posta in alto a destra: benché visibili, i pulsanti sono collegati a una funzionalità secondaria e potrebbero pertanto passare inosservati, dunque è preferibile documentarne l’uso.*

*È possibile avvalersi di uno o più screenshot per illustrare alcune peculiarità, sebbene una lista di immagini di schermate sia da evitare.*

Le funzionalità implementate sono, per semplicità, suddivise in due categorie: funzionali ed estetiche. Le prime comprendono:

* gestione di quattro tipologie di articoli
* conversione e salvataggio in formato JSON
* funzionalità di ricerca con pertinenza, filtro e paginazione
* criteri di pertinenza e filtro differenti per ogni tipologia di oggetto

Le funzionalità grafiche:

* barra dei menù in alto
* utilizzo di una toolbar attivabile/disattivabile (tramite menù)
* utilizzo di icone nella toolbar e nelle voci del menù
* status bar in fondo alla finestra
* scorciatoie da tastiera (mostrate anche nelle voci del menù)
* controllo della presenza di modifiche non salvate prima di uscire
* gestione del ridimensionamento
* possibilità di scegliere la disposizione dei risultati di ricerca fra tre tipologie
* ogni tipologia di prodotto ha una propria visualizzazione
* quattro tipologie di visualizzazione (lista, carta, *teaser*, pagina intera)
* possibilità di navigare attraverso diverse schermate tramite interfaccia
* utilizzo di icone nei pulsanti
* utilizzo di immagini e pulsanti nella visualizzazione degli oggetti
* utilizzo di colori e stili grafici
* effetti grafici come cambio del colore al passaggio del mouse

Le funzionalità elencate sono intese in aggiunta a quanto richiesto dalle specifiche del progetto.

# Rendicontazione ore

*Questa sezione riporta la lista delle attività svolte e, per ciascuna di esse, le ore previste e le ore effettivamente richieste. Non sarà applicata alcuna penalità nel caso in cui le ore previste differiscano dalle ore effettive: essendo uno dei primi progetti impegnativi affrontati durante il corso di laurea è normale che ciò avvenga. Allo stesso modo non saranno applicate penalità se le ore effettive superano leggermente il monte ore indicato nelle specifiche, in quanto è chiaro che lo studio di una nuova tecnologia possa comportare dei tempi aggiuntivi. Se lo sforo è superiore di più del 10% delle ore previste è gradita una motivazione dietro al superamento del monte ore. Un superamento eccessivo potrebbe tuttavia essere penalizzato in quanto segnale di una cattiva progettazione, per esempio se un progetto per cui erano previste 50 ore ne ha richieste più di 80.*

|  |  |  |
| --- | --- | --- |
| **Attività** | **Ore Previste** | **Ore Effettive** |
| Studio e progettazione | 10 | 10 |
| Sviluppo del codice del modello | 10 | 12 |
| Studio del framework Qt | 10 | 9 |
| Sviluppo del codice della GUI | 10 | 13 |
| Test e debug | 5 | 7 |
| Stesura della relazione | 5 | 5 |
| **totale** | 50 | **56** |

Il monte ore è stato leggermente superato in quanto sviluppo del codice di modello e GUI ha richiesto più tempo di quanto previsto, in particolare sono stati svolti più test di quanto inizialmente preventivato per prendere confidenza col framework Qt.

# Differenze rispetto alla consegna precedente

*Questa sezione va inserita solo in caso di progetto riconsegnato, sia in caso di insufficienza nella precedente consegna, sia in caso di valutazione positiva che si cerca di migliorare. Per un progetto riconsegnato senza modifiche è preferibile inserire questa sezione indicando che non ci sono state modifiche rispetto alla consegna precedente: in questo modo il valutatore non perde tempo cercando di capire se ci sono state modifiche non riportate nella relazione.*

*Generalmente è una buona idea riferirsi ai punti negativi elencati nel feedback ricevuto alla consegna precedente, in modo da aiutare il valutatore a capire come questi siano stati affrontati semplificando (e velocizzando!) la valutazione del progetto.*

Rispetto alla consegna precedente ho implementato con maggiore attenzione gli aspetti estetici, come i testi dei pulsanti e l’utilizzo di icone e immagini, come indicato nel feedback.

Ho inoltre applicato un foglio di stile per applicare un effetto “over” al passaggio del mouse sulle card dei prodotti, per migliorare ulteriormente la resa grafica del progetto.