SVILUPPO DI PICCOLE APPLICAZIONI PER IL CORSO DI PROGRAMMAZIONE AVANZATA 2022/2023

*Colombo Alessandro 1066001*

*GESTIONALE DI UN BAR in JAVA*

Il software ha come obbiettivo quello di offrire un semplice gestionale di prodotti che possono essere trovati in un bar, utilizzando diverse caratteristiche del linguaggio Java viste a lezione. In questo documento verranno prima descritte le classi del package e successivamente come sono state utilizzate in una funzione “main” che stampa nella console di Eclipse i risultati delle varie operazioni.

La classe **Prodotto** è una classe *astratta* che fa da super classe di tutti i prodotti che il bar ha da offrire. Presenta gli attributi tipologia e costo, ottenibili tramite i metodi getTipo() e getCosto(), inoltre implementa l’*interfaccia* visitable, illustrata più avanti ed ereditata dalle sue sottoclassi.

La classe **Type** è un *enumerativo* che indica se è il prodotto è un cibo o una bevanda.

La prima sottoclasse che *estende* Prodotto è **Birra**, di tipologia “bevanda” con un costo fisso di 4.00 euro. Il metodo toString() di questa classe ritorna solo il nome del prodotto e il suo prezzo.

La seconda sottoclasse, più complessa, è **Panino.** Oltre agli attributi ereditati, presenta un array di ingredienti, un id *final* proprio del panino e un contatore globale dei panini già esistenti. Per poter analizzare questa classe, è necessario conoscere la classe **Ingrediente**: questa ha un nome ed un costo, ottenibili sotto forma di Stringa con il metodo toString e implementa *Comparable*, così che sia comparabile con un altro Ingrediente, più precisamente vengono confrontati i nomi dei due ingredienti per vedere se si tratta dello stesso ingrediente.

Durante la creazione di un nuovo panino, è necessario fornire un elenco di ingredienti che andranno a formare questo prodotto oltre al pane presente di default.

Al panino viene assegnato un id, successivo al valore attuale di idCounter, un intero *privato e statico* che serve solo per l’assegnazione automatica di tali identificativi ed evitare che si siano doppioni.

In questa classe abbiamo *override* sia del metodo getCosto() che ritorna la somma dei costi degli ingredienti e non un costo fisso, sia il metodo toString viene modificato, per mostrare il proprio identificativo progressivo.

Come anticipato sopra, la classe Prodotto e le sue sottoclassi implementano l’*interfaccia* **Visitable**, seguendo il *Design Pattern* di *Visitor*, ovvero hanno un metodo “accept(Visitor<T> v)” che accetta un visitatore di tipi generici. Questo metodo è overridato nella classe Panino, poiché la classe **Visitor** ha due metodi visit, con un *overload* per il passaggio sia di un oggetto Prodotto, sia Panino, poiché avranno scopi leggermente diversi a seconda di cosa viene visitato.

L’interfaccia Visitor è implementata nella classe Menu, che segue il Design Pattern *Singleton*: il menu presenta il nome del bar e un oggetto di classe Menu chiamato menucompleto: poiché il bar ha un solo menu (da intendere come visualizzatore dei prodotti a disposizione e non come menu cartaceo), non è necessario che ogni volta vengano creati nuovi menù, ma viene invece passato il Menu già esistente.

Tramite il menu, è possibile “visitare” un prodotto, ottenendo il nome e il prezzo se è una birra, o la lista degli ingredienti con i loro costi, l’id e il costo totale se l’oggetto visitato è un panino.

Nella classe **Bar** che presenta la funzione main del package viene prima creato un **Elenco** ordinato di ingredienti. Questo Elenco è composto da un array di *oggetti generici* non duplicati e ordinabili, ed partendo da un oggetto del medesimo tipo è possibile controllare se nell’elenco è presente un oggetto di egual valore tramite il metodo find().

Nel Bar, viene creato un elenco di Ingredienti, che sono quindi utilizzati per la creazione di due panini: durante la loro creazione ci si basa sul metodo find appena introdotto per assicurarsi che gli ingredienti inseriti siano solo quelli presenti nell’Elenco del Bar.

I vari prodotti sono quindi inseriti in una lista, che viene visitata dal Visitatore Menu nel metodo vediMenu per ottenere una lista di tutti i prodotti con i loro nomi, prezzo e dove possibile, ingredienti.

*GESTIONALE DI UN BAR in C++*

Il package presenta diverse headers che contengono le classi principali e una classe **Bar.cpp** che contiene il main del progetto in cui sono testate le varie funzionalità e i limiti del software.

La prima classe sviluppata è contenuta nel file **Ingrediente.h,** contenente un attributo “id” secondo un template definito nello stesso file, prima della classe Ingrediente. Questo attributo è privato e rappresenta l’identificativo dell’ingrediente, in questo caso verrà utilizzaun semplice intero. Il secondo campo, pubblico, indica il costo dell’ingrediente e sarà utilizzato in seguito per definire il prezzo dei prodotti. Tutti i costi sono espressi in double. Infine, contiene un campo descrizione per avere delle informazioni su tale prodotto, ad esempio un nome.

Il costruttore di Ingrediente riceve tutti i campi quando viene creato, inoltre la classe presenta due metodi per visualizzare solo il nome, siccome è privato, o sia il nome che il costo tramite l’operazione toString().

Nel file **Prodotto.h** è descritta la classe di Prodotto, che rappresenta un prodotto generico. Tutti i prodotti hanno il campo privato costante ID, che identifica il prodotto, e i campi pubblici nome e costo.

Essendo prodotto una classe generica non è realmente utilizzabile, infatti presenta come ID lo zero, interpretato dal software come un ID nullo e un costo pari a zero per simboleggiare che non è vendibile. Non è nemmeno possibile impostargli un nome attraverso un metodo, proprio perché questa classe serve solo per generalizzare le sottoclassi.

Sono presenti tre sottoclassi:

La classe **Birra** eredita prodotto in modo pubblico, ed ha come ID fisso il valore 1: tutte le birre create hanno lo stesso ID poiché si tratta dello stesso prodotto.

Il nome descrive il tipo di prodotto stesso, quindi è assegnato come “birra” e il costo è fisso a 4$. Tutti i campi sono formattati in una stringa con un override del metodo toString(), ereditato virtualmente dalla superclasse. Il metodo getCosto() invece esegue un redefine, poiché ritorna 4.00 invece di 0.00 e questo metodo non è virtual nella superclasse;

la classe **Acqua** eredita prodotto in modo privato ed è stata pensata per non essere inserita a menu ma essere disponibile gratuitamente. Come Prodotto ha quindi un costo fisso a zero ed un ID fisso a 999. L’unica aggiunta presente è un enumerativo “Tipo” che indica se l’oggetto creato è Frizzante o Naturale, specificato quando viene invocato il costruttore.

L’enumerativo **Tipo** presenta un override dell’operatore “<<” per far sì che quando viene stampato venga visualizzato il nome dell’enumerativo scelto e non il valore numerico.

La terza sottoclasse e più articolata è la classe **Panino,** anch’essa eredita Prodotto in modo pubblico. Oltre al campo privato ID e i campi ereditati da Prodotto, contiene una lista di Ingredienti chiamata “ingr” di un tipo generico T.

Per poter costruire un panino è necessario passare una lista di ingredienti, gestita da un initializer\_list che riempie in automatico la lista ingr. Inoltre, essendo che i panini sono diversi a seconda degli ingredienti scelti e anche dalla quantità, ad esempio più salse o carne, ogni panino ha un proprio ID, assegnato in modo crescente dal costruttore con un IDcount statico a partire dal numero 100.

Il campo “nome” è lasciato vuoto poiché un panino è descritto dagli ingredienti che contiene, ottenibili in una stringa dal metodo getIngredients(). Gli ingredienti vanno anche a definire il costo, con un sovrapprezzo di 1$ oltre al valore complessivo degli ingredienti.

È possibile inserire ingredienti ad un panino anche successivamente, con il metodo append().

Ogni volta che un prodotto viene cancellato, il suo prezzo viene impostato a zero.

Per gestire tutti i prodotti, esiste la classe **Menu** che contiene un vettore di Prodotti. Questi prodotti sono inseriti tramite il metodo nuovoProdotto() che riceve un puntatore ad un prodotto esistente. Questo per far si che nel menu non siano presenti più prodotti uguali.

Per fare ciò si utilizza il metodo trova di questa classe e il metodo check() di ogni prodotto. Se il nuovo prodotto restituisce lo stesso check() di un altro prodotto o “0”, il prodotto non viene inserito a menu, viene stampato un avviso e il nuovo prodotto è cancellato. Nel caso sia un panino, viene cancellato l’elenco di ingredienti, ed per qualsiasi prodotto il costo è azzerato.

Il menu contiene sempre i prodotti in ordine crescente di ID ed ha un metodo stampa() per stamparli con il loro prezzo e descrizione, che sia nome o elenco di ingredienti.

Nel main sono svolti i seguenti passaggi:

* Vengono creati e stampanti 6 diversi ingredienti, con un id di tipo int.
* Sono create tre birre.
* Una birra viene cancellata, e stampata: appare come un prodotto con id 0 e costo 0.
* Viene creato un menu vuoto a cui sono aggiunte le birre.
* Dal metodo stampa del menu si può vedere che solo una delle due birre è stata aggiunta, mentre l’altra, nulla, no.
* Vengono create e stampate due oggetti Acqua, una naturale e una frizzante, ed è testato l’operatore << dell’enumerativo Tipo.
* Sono creati 5 panini: tuttavia i panini p3 e p4 sono identici, seppur con ID diversi.
* I panini sono aggiunti al menu: il panino p4 essendo già presente non viene inserito ed è cancellato. Stampando questi due panini, vediamo che p3 è intatto mentre p4 è un prodotto non valido (con ID 0)
* Stampiamo il menu per verificarne lo stato corrente: 1 birra e 4 prodotti, ovviamente non troviamo p4.
* Cancelliamo p2 e modifichiamo p1 aggiungendogli un nuovo ingrediente.
* Stampando nuovamente il menu, troviamo che p2 è stato rimosso anche dal menu essendo un prodotto non valido, mentre p1 presenta la modifica appena apportata a suoi ingredienti.

*MACCHINETTA DEL CAFFè in HASKELL*

Nel file di testo è riportato il codice di un breve progetto che simula le funzionalità di un distributore di bevande che ricevuta la selezione di un prodotto e delle monete, calcola e distribuisce il resto a seconda delle monete disponibili. Lo scopo di questo progetto è mostrare alcune delle principali funzionalità di base del linguaggio Haskell.

Il **main** del programma inizia con le **impostazioni di avvio** della macchinetta, ovvero un array delle monete già presenti, altrimenti non sarebbe possibile dare il resto ai primi clienti.

Dopo di ciò viene inizializzata la funzione **loop**, che rappresenta il ciclo di operazioni svolte in un singolo accesso alla macchinetta.

Dopo una serie di stringhe solo estetiche e informative, in cui viene detto che sono accettate solo le monete con un valore superiore o uguale a 0.10 €, ovviamente solo monete “reali”. Per completezza e controllo dell’esecuzione viene stampato un array rappresentate le monete al momento presenti, inizialmente corrispondenti alle impostazioni di avvio.

Successivamente viene visualizzato a schermo l’elenco dei prodotti e i rispettivi costi e viene chiesto all’utente di selezionarne uno, e viene chiamata la funzione **prodotto:** questa funzione riceve un array di float, ovvero i prezzi, e un intero, ovvero il prodotto scelto. Se il prodotto è nei limiti dell’array (aumentati di 1 per avere degli id > 0) allora ritorna il prezzo del prodotto, altrimenti ritorna 0 e la macchinetta si resetta a prima dell’ultima selezione del prodotto.

Stampato il prezzo, viene chiesto all’utente di inserire le monete e per fare ciò viene utilizzata la funzione **credito:** questa riceve in input un numero decimale, ovvero il costo rimanente da pagare del prodotto, e chiede in input un altro decimale, che rappresenta la monetina inserita dall’utente.

Se la moneta è valida, verificando con il metodo **isValid** se il valore della moneta inserito è tra quello accettati come spiegato in precedenza.

In caso affermativo, la moneta è inserita nell’array della risposta e se il prezzo da pagare è ancora superiore a 0, viene chiamata ricorsivamente la funzione ma solo sul prezzo rimanente, per sapere se la moneta inserita è superiore al prezzo rimanente viene utilizzata la funzione **min’** così da ridurre il prezzo sempre a 0 e non a valori negativi. Altrimenti se la moneta non è valida viene chiamata con lo stesso prezzo attuale e la moneta non è inserita nella risposta.

La funzione **round’** è una funzione ausiliaria inserita per arrotondare il valore dei decimali alla prima cifra dopo la virgola. Questa è necessaria perché talvolta in calcoli con Float e/o Double venivano ritornati valori leggermente sfalsati che tuttavia rendevano sbagliate le successive operazioni della macchinetta, ad esempio effettuando 1.0 – 0.8 veniva restituito un valore di 0.199996 ed è quindi necessario arrotondarlo per poi calcolare correttamente il resto da consegnare.

Le monete inserite sono aggiunte all’array di monete già presenti nella macchina e tramite la funzione **resto** vengono calcolate le monete da restituire all’utente: il valore del resto è uguale al credito inserito meno il prezzo del prodotto selezionato, da qui vengono distribuite le monete in ordine decrescente con la seguente logica: se il valore della prima moneta della macchinetta è superiore all’importo da restituire si passa alla seconda, altrimenti viene aggiunta all’array di ritorno e il resto viene diminuito di tale quantità. Se le monete disponibili non sono sufficienti a pagare il resto viene restituita la massima somma minore del resto che si riesca a offrire.

Una volta restituito il resto, la macchinetta riparte ma come array iniziale ha le monetine presenti al giro precedente, più le monetine inserite dall’utente, meno le monetine restituite come resto. Questa sottrazione è effettuata con il metodo **remove.**

Un ultimo accorgimento è necessario sulla funzione sort, che riceve un array e lo restituisce in ordine discendete. Per fare ciò viene chiamato ricorsivamente su un array di tutti gli elementi maggiori del primo e su quelli minori, creando due array che saranno poi concatenati all’elemento iniziale.