# CORSO DI PROGRAMMAZIONE E MODELLAZIONE AD OGGETTI

## PROGETTO PER LA SESSIONE AUTUNNALE 2023/2024

DOCENTE: PROF.SSA SARA MONTAGNA

SVOLTO DA GIULIO MARIA BIANCHI

MATRICOLA 301993

FUORICORSO

INDIRIZZO MAIL: g.bianchi12@campus.uniurb.it

# RELAZIONE

Il programma vuole simulare e gestire una clinica veterinaria “automatizzata”, ovvero ci è cercato di automatizzare le funzionalità che coprono la gestione degli appuntamenti, la registrazione e l'autenticazione degli utenti (clienti e veterinari), il calcolo del rischio per la salute degli animali, e la gestione dei pagamenti. La struttura del programma è realizzata con un’architettura modulare che separa le responsabilità principali in classi e servizi distinti, rendendo il codice più mantenibile, estendibile e testabile.
Le principali aree funzionali del programma sono:

## 1. GESTIONE DELLE PRENOTAZIONI E DEGLI APPUNTAMENTI

I clienti possono prenotare appuntamenti per i loro animali, specificando le informazioni rilevanti come tipo di animale, sintomi, e data dell’appuntamento.
Gli appuntamenti vengono gestiti centralmente tramite un calendario, che verifica la disponibilità degli slot (giorni lavorativi dalle 8:00 alle 20:00) e previene conflitti di orario.

## 2. CALCOLO DEL RISCHIO DI SALUTE SUGLI ANIMALI

Il sistema valuta il rischio di salute degli animali, basandosi su specie, razza, età, condizioni croniche e sintomi segnalati.
Questa funzionalità consente ai veterinari di avere una panoramica dello stato dell’animale e di assegnare priorità agli appuntamenti.

## 3. GESTIONE DEL PAGAMENTO E FATTURAZIONE

Include la generazione di fatture, l’applicazione di sconti (per esempio, sconti fedeltà per clienti abituali) e l’aggiunta di penalità per cancellazioni tardive.
I pagamenti vengono elaborati e confermati attraverso un sistema che supporta metodi come carta di credito, PayPal, contanti, ecc.

## 4. AUTENTICAZIONE E REGISTRAZIONE DEGLI UTENTI

Consente la registrazione di clienti e veterinari. L’autenticazione degli utenti, gestita da AuthenticationService, assicura che solo i clienti registrati possano accedere alla funzionalità di prenotazione.

## 5. GESTIONE DELLA CARTELLA CLINICA

Ogni animale ha una cartella clinica che contiene dettagli sullo storico delle visite, trattamenti precedenti, sintomi e condizioni croniche. Questo modulo consente di monitorare lo stato di salute dell'animale e di fornire cure su misura.

## CLASSI DI DOMINIO

User, Customer e Veterinarian: Queste classi rappresentano le entità utente del sistema.
Customer e Veterinarian estendono User, utilizzando l'ereditarietà per specificare i ruoli e i diritti degli utenti. I veterinari, ad esempio, possono completare gli appuntamenti e aggiornare le cartelle cliniche.
Animal: Rappresenta gli animali trattati nella clinica, con attributi come razza, specie, e genere, nonché informazioni sull'animale come il proprietario (cliente).
Appointment e AppointmentRegistry: Appointment rappresenta un singolo appuntamento, con dati associati come il veterinario, il cliente, e la data/ora. AppointmentRegistry gestisce la lista di tutti gli appuntamenti, con metodi per aggiungere, rimuovere e trovare appuntamenti specifici.

## SERVIZI

AuthenticationService:
gestisce la registrazione e l'autenticazione dei clienti e dei veterinari. Rende sicura la registrazione e impedisce la duplicazione degli utenti.
AppointmentCalendarService:
funzionalità di prenotazione e calendario per gestire gli orari e gli appuntamenti. Fornisce metodi per controllare la disponibilità degli slot e prevenire conflitti di appuntamento.
AnimalHealthService:
serve per calcolare il rischio di salute degli animali in base ai dati della cartella clinica e ai sintomi segnalati, generando descrizioni dettagliate per il rischio basso, medio o alto.
PaymentService:
gestisce la fatturazione e il pagamento, con sconti e penalità in base allo storico degli appuntamenti e al tipo di visita. Inoltre, simula l'elaborazione dei pagamenti con diversi metodi.

## INTERFACCE

Nella costruzione dell'architettura del programma si è partiti dalle interfacce perché esse raggruppano dei comportamenti comuni di varie classi di servizio e non avendo una implementazione, anzi unendo diverse implementazioni.
In particolare sono state pensate 2 interfacce:
Le interfacce per il canale di prenotazione:
interfacce definiscono i contratti per i metodi di pagamento e le strategie di prenotazione.

Le interfacce per il metodo di pagamento:
Usare interfacce rende il codice più modulare, facilitando l'aggiunta o la modifica di nuove classi. Si volesse aggiungere un nuovo canale di prenotazione basterebbe creare una classe che implementa l'interfaccia base.
Queste IPaymentMethod specifica i metodi processPayment e completePayment, mentre IBookingStrategy permette l'estensione per strategie di prenotazione diverse, per altre implementazioni future.

## FLUSSO DI ESECUZIONE

1. Registrazione degli Utenti:
L'utente esegue il login o la registrazione come Customer o Veterinarian tramite AuthenticationService.

2.Prenotazione di Appuntamenti:
Un Customer può creare una prenotazione chiamando AppointmentCalendarService, che prima verifica la disponibilità dello slot richiesto e poi salva l'appuntamento in AppointmentRegistry.
In caso di conflitto, viene generata un'eccezione (AppointmentConflictException) e il cliente può scegliere un altro slot disponibile.

3. Valutazione del Rischio di Salute:
Il veterinario può calcolare il rischio di salute di un animale usando AnimalHealthService, che elabora informazioni come specie, età, sintomi, ecc., per classificare il rischio.

4. Gestione della Fatturazione:
A seguito di una visita, il sistema crea un'Invoice attraverso PaymentService e processa il pagamento.
Se il pagamento va a buon fine, Invoice è contrassegnata come pagata e l'appuntamento viene rimosso dal AppointmentRegistry.

5. Conferma dell'Appuntamento e Pagamento:
Dopo la visita, l'Appointment può essere completato dal veterinario. In caso di conferma del pagamento, l'Invoice viene archiviata e l'Appointment rimosso dal AppointmentRegistry.

## 1 CLASSE ASTRATTA USER

La classe User è una classe astratta che funge da superclasse per tutti i tipi di utenti del sistema. Essa definisce attributi comuni a tutti gli utenti e fornisce un'interfaccia uniforme attraverso cui gli utenti interagiscono con il sistema.
Rappresenta un utente generico del sistema che ha attributi nome, password e ruolo (definito dall'enumerazione Role).
La classe User è una classe astratta che funge da superclasse per tutti i tipi di utenti del sistema. Essa definisce attributi comuni a tutti gli utenti e fornisce un'interfaccia uniforme attraverso cui gli utenti interagiscono con il sistema.
La scelta di rendere la classe User astratta è motivata da diverse ragioni legate alla progettazione del sistema:
- Generalizzazione: User rappresenta un concetto generico di utente che può essere specializzato in diverse tipologie (ad esempio Veterinarian, AdminStaff, ecc.). Rendere User una classe astratta evita che istanze di User possano essere create direttamente, il che non avrebbe senso poiché un utente deve avere un ruolo specifico.
- Riutilizzo del codice: Nella classe User possono essere definite proprietà e metodi comuni a tutte le sottoclassi (come id, name, surname, e metodi come getId() o getName()). Le sottoclassi erediteranno questo codice, riducendo la duplicazione e promuovendo il riuso.
- Polimorfismo: Definendo User come astratta, possiamo sfruttare il polimorfismo. Ad esempio, una lista di utenti può contenere oggetti di tipo Veterinarian o AdminStaff, e possiamo trattare tutti questi oggetti come istanze della classe base User.
- Forzare l'implementazione di metodi specifici: Se ci fossero metodi che devono essere implementati da ogni tipo specifico di utente (ad esempio un metodo performDuties()), possono essere dichiarati come metodi astratti nella classe User, forzando ogni sottoclasse a fornire la propria implementazione.

Vantaggi:
- Manutenzione: Migliora la manutenibilità del codice grazie al riutilizzo e alla riduzione della duplicazione.
- Espandibilità: Facilita l'aggiunta di nuovi tipi di utenti in futuro.

## 2 ENUM ROLE

Definisce i ruoli disponibili nel sistema: VETERINARIAN, ADMIN_STAFF,  CUSTOMER.
La scelta dell’uso di enum per il ruolo (Role) è motivato dai seguenti fattori:

- Valori fissi e sicuri: Gli enum garantiscono che un ruolo possa avere solo uno dei valori predefiniti, riducendo la possibilità di errori come l'uso di stringhe non valide o incoerenti per rappresentare ruoli diversi.
- Chiarezza e leggibilità: Gli enum rendono il codice più leggibile e autoesplicativo. Ad esempio, Role.ADMIN è più chiaro e meno soggetto a errori di battitura rispetto a una stringa come "Admin".
- Facilità di gestione: Aggiungere, rimuovere o modificare ruoli diventa più facile e centralizzato. Basta aggiornare l'enum, e il cambiamento si rifletterà in tutto il codice.
- Uso in switch-case: Gli enum sono molto utili nei blocchi switch-case, dove si può gestire ogni caso specifico in modo ordinato e sicuro.
- Consistenza: Utilizzare un enum per i ruoli assicura consistenza nel sistema, evitando variabili stringa incoerenti o valori non standard.

Si è scelto di estendere User con le classi "Veterinarian", Customer", "AdminStaff" poiché la classe User contiene proprietà e metodi comuni a tutti i tipi di utenti (come id, name, surname).
Estendendo User, le sottoclassi ereditano automaticamente queste funzionalità, evitando la duplicazione del codice.
Per quanto riguarda aspetti di programmazione ad oggetti come Generalizzazione e Polimorfismo, con questa struttura, possiamo trattare tutti gli oggetti di tipo Veterinarian, Customer, e AdminStaff come User. Questo permette, ad esempio, di gestire liste di utenti generiche o eseguire operazioni comuni senza preoccuparsi del tipo specifico di utente.
Estendere User impone che tutte le sottoclassi abbiano almeno gli attributi e i metodi definiti in User. Questo assicura che ogni tipo di utente abbia una struttura coerente e prevedibile.
Ciò comporta una maggior facilità di manutenzione: Se le proprietà o i metodi comuni a tutti gli utenti devono essere modificati o estesi, possiamo farlo nella classe User, e le modifiche si propagheranno automaticamente a tutte le sottoclassi.

## 4 CLASSE APPOINMTENT

Nella classe Appointment è stato usato il pattern Builder, che viene usato quando una classe richiederebbe molti costruttori con differenti combinazioni di parametri. In questo modo si evita di creare un numero eccessivo di costruttore e si hanno alcuni vantaggi come:
- maggior flessibilità permettendo di creare oggetti complessi passo per passo, scegliendo quali campi opzionali includere
- si evita l'overload di costruttori , rendendo il codice più leggibile
- sicurezza: non permette di creare oggetti con combinazioni errate

## 5 CLASSE APPOINTMENTREGISTRY

Tale classe funge da database per la gestione centralizzata degli Appointment. Questa classe fornisce dei metodi CRUD che garantiscono le operazioni fondamentali come: aggiungere un nuovo appuntamento, rimuovere un appuntamento, ricercare appuntamenti in base a criteri diversi (ad esempio, per data, proprietario o tipo di prenotazione), visualizzare tutti gli appuntamenti.

Per implementare le ricerche con metodi generici, possiamo usare un approccio che permette di specificare i criteri di ricerca in modo più flessibile. Ad esempio, possiamo creare un metodo generico che accetta un predicato (una funzione che restituisce true o false per un dato slot), per filtrare gli slot in base a criteri specifici.

Sono stati usati gli stream che presentano 2 vantaggi fondamentalmente: essi sono detti "lazy", cioè le operazioni non vengono eseguite fino a quando non è necessario. Questo può portare a miglioramenti delle prestazioni, poiché le operazioni possono essere ottimizzate e combinate ed inoltre gli stream possono essere facilmente concatenati, consentendo di costruire pipeline di elaborazione complesse in modo semplice e intuitivo.

Vantaggi dell'Approccio Generico:

- Flessibilità: Un unico metodo gestisce tutte le possibili ricerche di slot, riducendo la duplicazione del codice.
- Predicati Riutilizzabili: Puoi passare diversi predicati per filtrare gli slot in base alle tue esigenze.
- Limitazione del Risultato: Puoi usare il parametro limit per ottenere solo un numero specifico di slot (ad esempio, i primi 10 disponibili).

## 6 AUTHENTICATION SERVICE

Questa classe rappresenta una classe di servizio che permette di autenticare i vari utenti e inoltre tiene traccia di tutti gli utenti registrati in una lista statica. Essendo statica, è condivisa tra tutte le istanze della classe, il che significa che tutti gli utenti registrati sono accessibili da qualsiasi parte del programma.

La classe fornisce un metodo per autenticare gli utenti in base al loro ruolo. Questo è utile per garantire che solo gli utenti con i diritti appropriati possano accedere a determinate funzionalità del sistema. La classe gestisce la registrazione degli utenti, assicurandosi che non ci siano duplicati e che solo gli utenti con i ruoli appropriati possano registrarsi.

Attraverso l'incapsulamento, l'ereditarietà, il polimorfismo e l'astrazione, la classe fornisce un'interfaccia semplice per l'autenticazione degli utenti, mantenendo al contempo la complessità interna nascosta. Questo approccio non solo migliora la manutenibilità del codice, ma facilita anche l'estensibilità dell'applicazione, consentendo l'aggiunta di nuovi ruoli o funzionalità in futuro.
Il polimorfismo consente a oggetti di classi diverse di essere trattati come oggetti della stessa classe base. Nella classe AuthenticationService, il metodo registerUser accetta un oggetto di tipo User, ma può restituire un oggetto di tipo Customer o Veterinarian a seconda del ruolo dell'utente.
Il metodo registerUser() è flessibile in quanto utilizza il polimorfismo per restituire un oggetto specifico in base al ruolo dell'utente, mantenendo la coerenza dell'interfaccia comporta una semplificazione del Codice poichè questo approccio riduce la necessità di scrivere codice duplicato per gestire diversi tipi di utenti.
Questo comporta anche un miglioramento in termini di:
- Estensibilità e manutenibilità del programma: Se in futuro si desidera aggiungere nuovi ruoli (ad esempio Manager), è possibile farlo senza modificare significativamente la logica esistente. Occorre solamente creare una nuova classe che estende User e implementare la logica necessaria per gestire il nuovo ruolo all'interno della classe AuthenticationService. Questo approccio riduce il rischio di introdurre bug nel codice esistente.
- Separazione delle Responsabilità: La classe AuthenticationService si concentra esclusivamente sulla logica di autenticazione e registrazione. Questo rispetto del principio di Single Responsibility Principle (SRP) rende il codice più facile da comprendere e mantenere.

## 7 CLASSE APPOINTMENTCALENDARSERVICE

La classe AppointmentCalendarService è responsabile della gestione del calendario per gli appuntamenti della clinica veterinaria. Questa classe fornisce funzionalità per aggiungere Appuntamenti, se lo slot orario è libero, e rimuovere Appuntamenti dal calendario, liberando lo slot associato. La classe gestisce gli slot disponibili dalle 8:00 alle 20:00 (escludendo i fine settimana). Ogni giorno lavorativo sarà suddiviso in slot di 1 ora, e viene controllata la disponibilità degli slot e per inserire nuovi appuntamenti solo in quelli liberi.
I conflitti di Orario vengono gestiti attraverso il metodo isSlotAvailable. Questo metodo verifica se lo slot è libero, controllando se lo slot esiste in occupiedSlots, ovvero se è già stato prenotato.
Se lo slot è già occupato, la funzione isSlotAvailable() restituisce false, impedendo così di prenotare lo stesso orario due volte e generando un'eccezione in caso di conflitto (AppointmentConflictException).
Ogni volta che viene aggiunto o rimosso un appuntamento, la classe aggiorna una struttura dati interna HashSet di occupiedSlots (slot occupati) per tenere traccia delle prenotazioni.

Per gestire gli slot temporali occupati è stata scelta una struttura di tipo HashSet per i seguenti motivi:

- Velocità di Accesso: Le operazioni add e remove in HashSet sono veloci, con una complessità di tempo media O(1), rendendo il controllo della disponibilità degli slot molto efficiente.
- Nessuna Duplicazione: Un HashSet non permette duplicati, garantendo che ogni slot venga registrato una sola volta.
- Accesso Diretto: Permette di verificare direttamente se uno slot è occupato, senza dover scorrere una lista, ottimizzando così il controllo della disponibilità degli slot.

## 8 ANIMALHEALTHSERVICE

La classe AnimalHealthService è progettata per fornire servizi relativi alla salute degli animali, in particolare per calcolare il rischio di malattia e fornire informazioni sui sintomi comuni associati a diverse specie e razze di animali. In questo modo le classi possono collaborare per fornire funzionalità complesse, mantenendo al contempo una chiara separazione delle responsabilità.
La classe AnimalHealthService funge da intermediario tra l'utente e la logica di calcolo della salute degli animali. Quando un utente chiama il metodo calculateHealthRisk, inzialmente vengono validati gli input, poi viene chiamato il metodo calculateIllnessProbability della
classe AnimalHealthProbabilityCalculator per ottenere il rischio di malattia e infine viene restituito il risultato.
La classe utilizza un'istanza di AnimalHealthProbabilityCalculator, che viene creata nel costruttore. Questo approccio consente di separare le responsabilità e facilita i test unitari. Si è deciso di ideare una funzione getCommonSymptoms che restituisce un elenco di sintomi comuni per una data specie e razza di animale, utile per i veterinari e i proprietari di animali per identificare rapidamente i sintomi associati a specifiche razze. Il metodo utilizza una mappa annidata per associare le specie e le razze ai loro sintomi comuni. Se la specie o la razza non è trovata, restituisce una lista vuota.
All'interno del metodo è la struttura dati di tipo mappa annidata (Map<String, Map<String, List<String>>>) per organizzare i sintomi per specie e razza. Questo approccio consente di accedere rapidamente ai sintomi in base alla specie e alla razza specificate. La struttura è così composta: la chiave principale è la specie (ad esempio, "dog" o "cat"). Il valore associato a ciascuna chiave di specie è un'altra mappa, dove la chiave è la razza e il valore è una lista di sintomi comuni per quella razza.
Le mappe forniscono un accesso O(1) ai dati, il che significa che il recupero delle informazioni è molto veloce. Questo è importante in un contesto in cui si desidera fornire risposte rapide agli utenti ed inoltre fornisce flessibilità poiché le mappe consentono di aggiungere facilmente nuove specie e razze

senza dover modificare la struttura esistente. È sufficiente aggiungere nuove voci alla mappa, il che rende il sistema più scalabile.
Utilizzando mappe nidificate, è possibile organizzare i dati in modo gerarchico, il che rende più facile comprendere le relazioni tra specie, razze e sintomi. Questo approccio migliora la leggibilità del codice e facilita la manutenzione. Per esempio, se un utente richiede i sintomi per un "labrador", il metodo cerca nella mappa per la specie "dog" e poi nella mappa interna per la razza "labrador", restituendo la lista di sintomi associati, come "cough", "vomiting", e "none".

Logica di Accesso: Il metodo cerca nella mappa i sintomi associati alla specie e alla razza fornita. Alla fine, il metodo restituisce una lista di sintomi comuni per la razza specificata della specie fornita. Se non ci sono sintomi associati, restituisce una lista vuota. In particolare, utilizza getOrDefault per gestire i casi in cui la specie o la razza non sono presenti nella mappa.

Analizzando la classe in base ai principi SOLID che sono stati trattati all'interno del corso di PMO, essa rispetta li rispetta pienamente in quanto è possibile rilevare:
- Single Responsibility Principle (SRP): La classe ha una singola responsabilità, ovvero fornire servizi relativi alla salute degli animali. La logica di calcolo è delegata a un'altra classe, il che è in linea con questo principio.
- Open/Closed Principle: La classe è aperta per l'estensione (ad esempio, aggiungendo nuovi metodi o funzionalità) ma chiusa per la modifica (non è necessario modificare il codice esistente per aggiungere nuove funzionalità).
- Liskov Substitution Principle: Se AnimalHealthProbabilityCalculator fosse un'interfaccia, potresti sostituire le sue implementazioni senza modificare il comportamento di AnimalHealthService.
- Dependency Inversion Principle: La classe dipende da un'astrazione (un'interfaccia) piuttosto che da una classe concreta. Questo approccio migliora la flessibilità e la testabilità.

## 9 ANIMALHEALTHPROBABILITYCALCULATOR

E' una classe di utilità che viene utilizzata da AnimalHealthService. Questa classe contiene la logica per calcolare i rischio associato ad ogni animale ovvero il calcolo della Probabilità di Malattia. Alla suddetta classe viene delegata la logica di calcolo da AnimalHealthService. Questo approccio consente di mantenere la classe di servizio snella e focalizzata sulla logica di business, mentre la classe di calcolo si occupa della logica specifica per determinare i rischi.

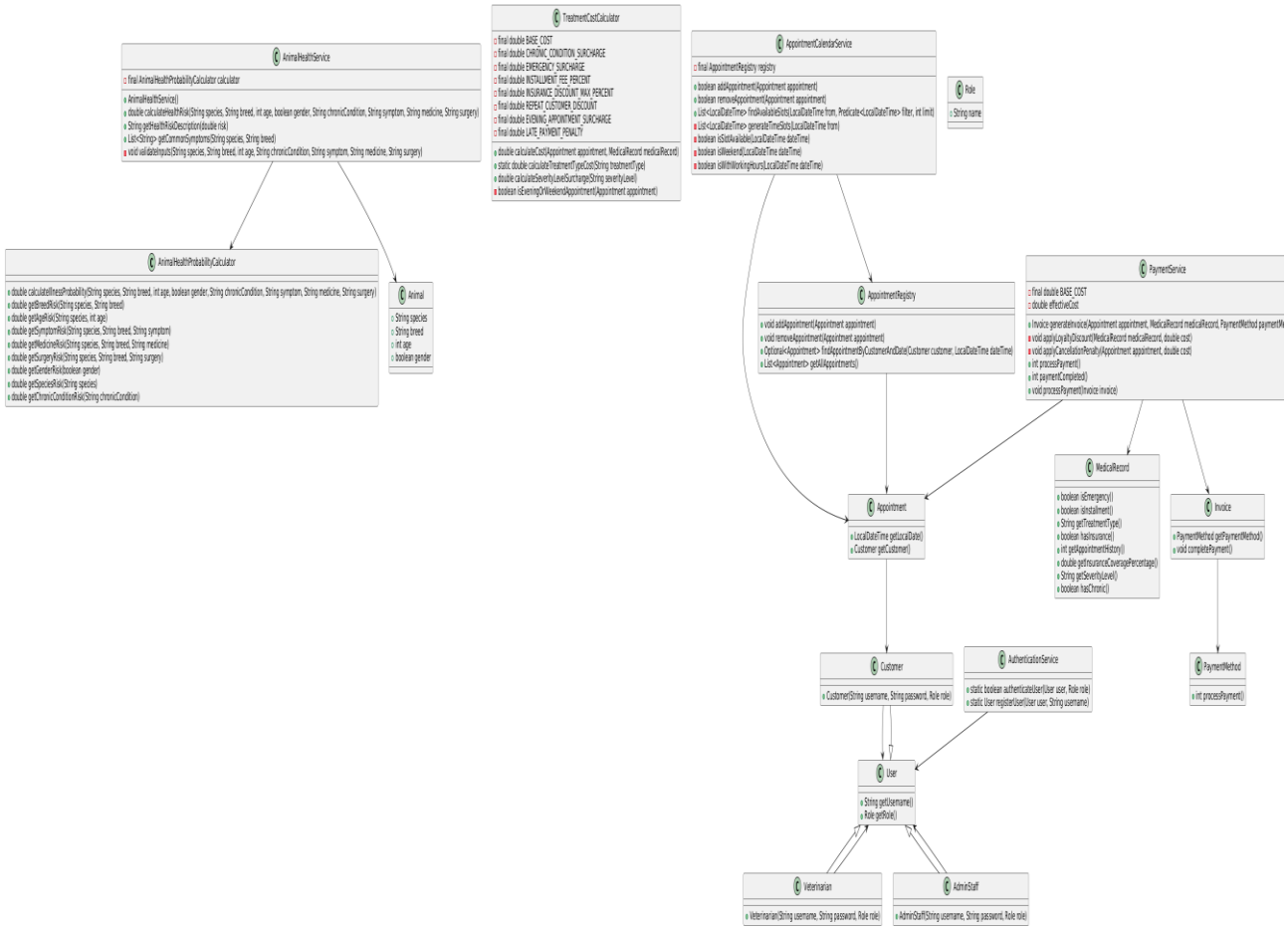
Il metodo principale calculateIllnessProbability accetta diversi parametri relativi all'animale, come specie, razza, età, genere, condizioni croniche, sintomi, farmaci e interventi chirurgici. Utilizza questi parametri per calcolare un punteggio di rischio totale, che rappresenta la probabilità che l'animale possa ammalarsi.
Per calcolare i rischi specifici, la classe utilizza vari metodi privati per calcolare il rischio associato a ciascun fattore:
- Rischio di specie: getSpeciesRisk restituisce un valore di rischio basato sulla specie dell'animale (cane, gatto, coniglio, ecc.).
- Rischio di razza: getBreedRisk restituisce un valore di rischio specifico per la razza dell'animale, considerando le razze comuni di cani e gatti.
- Rischio di età: getAgeRisk calcola il rischio in base all'età dell'animale, suddividendo le età in categorie (giovane, adulto, anziano).
- Rischio di genere: getGenderRisk restituisce un valore di rischio basato sul genere dell'animale (maschio o femmina).
- Rischio di condizioni croniche: getChronicConditionRisk aumenta il rischio se l'animale ha una condizione cronica.
- Rischio di sintomi: getSymptomRisk calcola il rischio in base ai sintomi presentati dall'animale.
- Rischio di farmaci: getMedicineRisk considera i farmaci che l'animale sta assumendo e il loro impatto sulla salute.
- Rischio di interventi chirurgici: getSurgeryRisk valuta il rischio associato a eventuali interventi chirurgici subiti dall'animale.

Infine viene fatta una normalizzazione del rischio, infatti dopo aver calcolato i vari rischi, la classe somma tutti i punteggi di rischio e normalizza il risultato su una scala da 0 a 1. Se la somma supera 1, viene limitata a 1.0, garantendo che la probabilità di malattia non possa superare il 100%.

Gestione dei Rischi:
Ogni metodo di rischio utilizza mappe per definire i valori di rischio per razze, sintomi, farmaci e interventi chirurgici. Questo approccio consente di estendere facilmente la classe per includere nuove razze o sintomi semplicemente aggiornando le mappe.

## 11 CLASSE PAYMENTSERVICE

Essa è progettata per gestire il processo di pagamento per appuntamenti medici, calcolando i costi associati e gestendo eventuali sconti o penalità.
Questa classe implementa l'interfacciaIPayment quindi può essere facilmente sostituita o estesa con altre implementazioni di pagamento. Ad esempio, se si desidera supportare un nuovo metodo di pagamento, è possibile creare una nuova classe che implementa IPaymentMethod senza modificare il resto del sistema.
La classe si preoccupa di generare un oggetto Invoice (fattura) per un appuntamento specifico, calcolando il costo del trattamento in base a vari fattori.
Il trattamento medico ha un costo di base che è fissato a 50.0.
La classe accetta 3 parametri: l'appuntamento per il quale si sta generando la fattura, la cartella clinica del paziente contenente informazioni rilevanti per il calcolo del costo, il metodo di pagamento scelto dal paziente.
La classe ha il compito di calcolare il costo del trattamento chiamando calculateTreatmentCost, di applicare eventuali sconti fedeltà e penalità per cancellazione, ed infine creare e restituire una fattura con i dettagli dell'appuntamento e il costo totale.
Esistono poi alcuni metodi che vanno a modulare l'ammontare del costo del trattamento come:
- applyLoyaltyDiscount: Applica uno sconto ai pazienti che hanno effettuato più di 10 appuntamenti. Se il paziente ha una storia di appuntamenti sufficienti, il costo viene ridotto di 20.0.
- applyCancellationPenalty: Aggiunge una penalità di 30.0 se l'appuntamento viene cancellato meno di 24 ore prima dell'orario previsto.
- calculateTreatmentCost: Calcola il costo totale del trattamento considerando vari fattori, e nello specifico applicando delle penalità se il trattamento è un'emergenza, se il paziente ha condizioni croniche, se il pagamento è rateizzato.


AnimalHealthService
- final AnimalHealthProbabilityCalculator calculator
+ AnimalHealthService()
+ double calculateHealthRisk(String species, String breed, int age, boolean gender, String chronicCondition, String symptom, String medicine, String surgery)
+ String getHealthRiskDescription(double risk)
+ List<String> getCommonSymptoms(String species, String breed)
- void validateInputs(String species, String breed, int age, String chronicCondition, String symptom, String medicine, String surgery)
TreatmentCostCalculator
- final double BASE_COST
- final double CHRONIC_CONDITION_SURCHARGE
- final double EMERGENCY_SURCHARGE
- final double INSTALLMENT_FEE_PERCENT
- final double INSURANCE_DISCOUNT_MAX_PERCENT
- final double REPEAT_CUSTOMER_DISCOUNT
- final double EVENING_APPOINTMENT_SURCHARGE
- final double LATE_PAYMENT_PENALTY
+ double calculateCost(Appointment appointment, MedicalRecord medicalRecord)
+ static double calculateTreatmentTypeCost(String treatmentType)
+ double calculateSeverityLevelSurcharge(String severityLevel)
- boolean isEveningOrWeekendAppointment(Appointment appointment)
AppointmentCalendarService
- final IAppointmentRegistry registry
+ boolean addAppointment(Appointment appointment)
+ boolean removeAppointment(Appointment appointment)
+ List<LocalDateTime> findAvailableSlots(LocalDateTime from, Predicate<LocalDateTime> filter, int limit)
+ List<LocalDateTime> generateTimeSlots(LocalDateTime from)
+ boolean isSlotAvailable(LocalDateTime dateTime)
+ boolean isWeekend(LocalDateTime dateTime)
+ boolean isWithinWorkingHours(LocalDateTime dateTime)
Role
+ String name
AnimalHealthProbabilityCalculator
+ double calculateIllnessProbability(String species, String breed, int age, boolean gender, String chronicCondition, String symptom, String medicine, String surgery)
+ double getBreedRisk(String species, String breed)
+ double getAgeRisk(String species, int age)
+ double getSymptomRisk(String species, String breed, String symptom)
+ double getMedicineRisk(String species, String breed, String medicine)
+ double getSurgeryRisk(String species, String breed, String surgery)
+ double getGenderRisk(boolean gender)
+ double getSpeciesRisk(String species)
+ double getChronicConditionRisk(String chronicCondition)
Animal
+ String species
+ String breed
+ int age
+ boolean gender
AppointmentRegistry
+ void addAppointment(Appointment appointment)
+ void removeAppointment(Appointment appointment)
+ Optional<Appointment> findAppointmentByCustomerAndDate(Customer customer, LocalDateTime dateTime)
+ List<Appointment> getAllAppointments()
PaymentService
- final double BASE_COST
- double effectiveCost
+ Invoice generateInvoice(Appointment appointment, MedicalRecord medicalRecord, PaymentMethod paymentMe
+ void applyLoyalty(Discount)(MedicalRecord medicalRecord, double cost)
+ void applyCancellationPenalty(Appointment appointment, double cost)
+ int processPayment()
+ int paymentCompleted()
+ void processPayment(Invoice invoice)
Appointment
+ LocalDateTime getLocalDate()
+ Customer getCustomer()
MedicalRecord
+ boolean isEmergency()
+ boolean isInstallment()
+ String getTreatmentType()
+ boolean hasInsurance()
+ int getAppointmentHistory()
+ double getInsuranceCoverage Percentage()
+ String getSeverityLevel()
+ boolean hasChronic()
Invoice
+ PaymentMethod getPaymentMethod()
+ void completePayment()
Customer
+ Customer(String username, String password, Role role)
AuthenticationService
+ static boolean authenticateUser(User user, Role role)
+ static User registerUser(User user, String username)
PaymentMethod
+ int processPayment()
User
+ String getUsername()
+ Role getRole()
Veterinarian
+ Veterinarian(String username, String password, Role role)
AdminStaff
+ AdminStaff(String username, String password, Role role)