UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Elaborato Ingegneria del Software

Lorenzo Fioravanti, Gabriele Quercioli

Applicativo Java di monitoraggio e gestione di stanze e vini in una cantina, finalizzato alla vendita

A.A 2021-2022

Indice

Unit Testing

Contenuti

L'applicazione, scritta in linguaggio Java, permette la gestione delle stanze e la gestione dell' invecchiamento dei vini in una cantina privata, permettendone poi l'acquisto.
I vini vengono inseriti nella cantina non appena arriva un rifornimento. Un vino, identificato da un nome e da una scheda tecnica che ne raccoglie le informazioni, può essere di tipo rosso, bianco o rosato, ed è disponibile alla vendita solo se è in buono stato di salute: ciò è indicato da una serie di parametri (Anidride Solforosa, PH, Zuccheri Riduttori, Grado Alcolico e Ossigeno) che devono stare all'interno di determinati range; tali range sono diversi a seconda del tipo, poiché i vini invecchiano in maniera differente.
Data la necessità di controllare i parametri, ogni vino della cantina è sottoposto a monitoraggio e, nel caso uno o più parametri violino i limiti di accettabilità, verrà scelta una politica di gestione del vino appropriata. Tale politica sarà differente a seconda di quale parametro ha superato il limite:

- Nel caso in cui uno tra Anidride Solforosa, PH, Zuccheri Riduttori e Grado Alcolico assuma un valore non accettabile, il vino osservato verrà rimosso dalla botte,quindi dalla cantina, e verrà buttato poiché non è possibile correggerlo.
- Nel caso in cui solo l' Ossigeno assuma un valore non accettabile, la politica adottata consiste nel correggere tale valore con un trattamento ad azoto e pertanto il vino non verrà buttato.

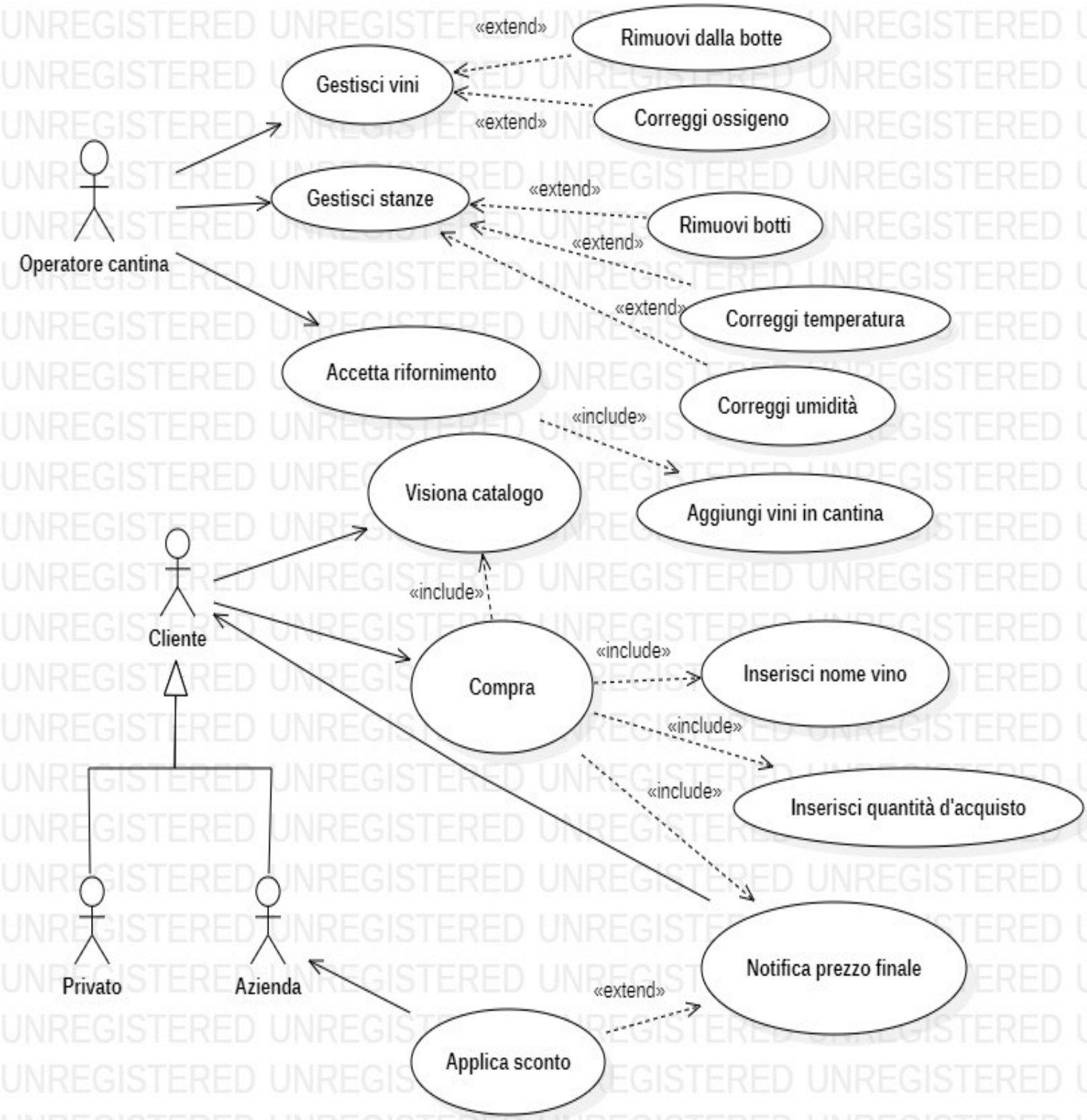
In ogni stanza (identificata da un numero) della cantina vengono monitorati due parametri di interesse: Temperatura e Umidità.
Col passare del tempo, tali parametri possono assumere valori inaccettabili facendo sì che la stanza non possa ospitare al suo interno dei vini.
Quindi l'obiettivo generale è quello di correggere tali valori inaccettabili, ma in caso di temperatura estrema verrà persa l' intera produzione della stanza.
Una volta terminata la gestione, i vini della cantina possono essere venduti. Un cliente può essere un privato o un'azienda e può acquistare un vino specificando nome e quantità richiesta dopo aver visionato il catalogo contenente i vini disponibili. La cantina notificherà

il prezzo finale di acquisto al cliente, che in caso sia un'azienda e richieda una quantità superiore a dieci litri, avrà diritto a uno sconto.

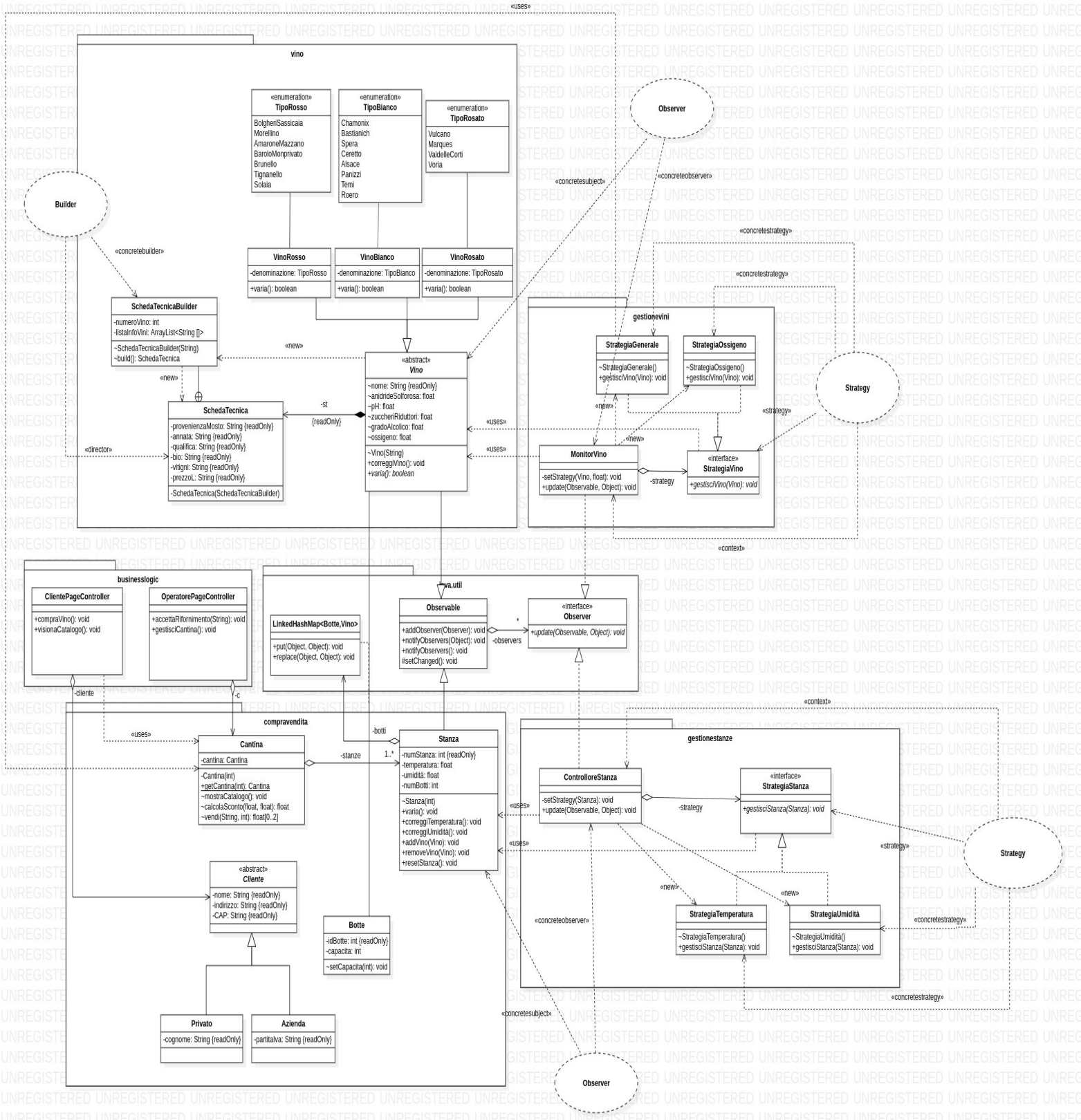
Strumenti

Per la realizzazione di tale elaborato è stato utilizzato il linguaggio di programmazione Java attraverso l'IDE IntelliJ IDEA v. 2021.2.
Sono stati disegnati dei diagrammi con StarUML sulla base del problema descritto precedentemente.
Per rendere la simulazione più credibile, i valori dei parametri dei vini e delle stanze sono generati in modo casuale tramite il metodo *Math.random()*, così come la loro variazione nel tempo.
Le forniture e il catalogo sono rappresentati con tabelle in formato *.ods*, le quali sono state gestite tramite la libreria *jopendocument*.
È presente anche la realizzazione di alcune classi di test (Unit Testing) attraverso il framework JUnit 4 integrato nell'ambiente di sviluppo.

Use Case Diagram



Class Diagram



Packages

La suddivisione logica del programma avviene tramite 5 packages più la *businesslogic*, dove ogni package racchiude le classi che insieme realizzano una determinata responsabilità:

1. **compravendita:** contiene la classe astratta Cliente e le sue derivate
   Privato e Azienda e le classi Cantina, Stanza e Botte. Il package è stato pensato per racchiudere le classi che, attraverso l'interazione tra loro, realizzano la responsabilità della vendita dei vini da parte della cantina, e del conseguente acquisto da parte dei clienti.
2. **vino:** contiene la classe astratta Vino e le sue derivate VinoRosso, VinoBianco e VinoRosato, oltre alla classe SchedaTecnica. Essendo il vino l'entità fondazionale del programma, gli è stato riservato un intero package in modo da realizzare una coesione logica e temporale tra le classi.
3. **java.util:** package già esistente, importato a scopo di utilizzare la classe Observable e l'interfaccia Observer.
4. **gestionevini:** contiene le classi che si occupano del monitoraggio e delle politiche di gestione dei vini. In particolare, la classe MonitorVino si occupa dell'osservazione e l'interfaccia StrategiaVino, insieme alle sue implementazioni StrategiaGenerale e StrategiaOssigeno, si occupa delle politiche di gestione.
5. **gestionestanze:** contiene le classi che si occupano del monitoraggio e delle politiche di gestione delle stanze della cantina. Nello specifico, la classe ControlloreStanza si occupa dell'osservazione e l'interfaccia StrategiaStanza, insieme alle sue implementazioni StrategiaTemperatura e StrategiaUmidità, si occupa delle politiche di gestione.
6. **businesslogic:** package al cui interno viene realizzata la responsabilità di fornire un' interfaccia all'operatore e al cliente.

3.1 Design Patterns

Nella realizzazione dell'applicazione si è rivelato vantaggioso l'utilizzo di alcuni design patterns.

3.1.1 Observer + Strategy

Nel progetto abbiamo due pattern sempre accoppiati, ovvero l'Observer e lo Strategy. Il motivo di tale scelta è stata la necessità di avere un sistema di monitoraggio automatico dei parametri, in grado di scegliere la corretta politica di gestione dinamicamente, avendo astrazione sulla concreta strategia istanziata. Questa struttura è stata sfruttata in due diverse situazioni: la gestione dei vini e la gestione delle stanze.

Nella prima è stato implementato un Observer in modo PUSH, poiché è fondamentale che il monitor conosca quale parametro ha innescato la notifica per scegliere la corretta strategia: correggere il vino se il parametro notificato è l' *ossigeno* oppure buttare il vino se il parametro notificato è uno tra *anidrideSolforosa, PH, zuccheriRiduttori* e *gradoAlcolico*.

```
11  @       private void setStrategy(Vino v, float param) {
12              if (param == v.getOssigeno())
13                  strategy = new StrategiaOssigeno();
14              else
15                  strategy = new StrategiaGenerale();
16              strategy.gestisciVino(v);
17          }
18
19          //PUSH
20          @Override
21  o↑      public void update(Observable observable, Object o) {
22              Vino v = (Vino) observable;
23              float param = ((Float) o).floatValue();
24              setStrategy(v,param);
25          }
```

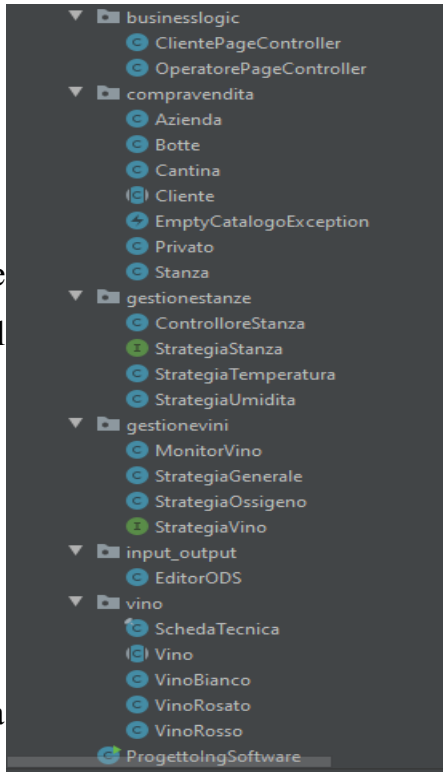
Nella seconda è stato implementato un Observer in modo PULL, poiché è responsabilità del monitor andare a recuperare entrambi i parametri della stanza(*temperatura, umidita),* dato che possono entrambi richiedere una strategia che porti alla loro correzione contemporaneamente.

```java
    private void setStrategy(Stanza s) {
        boolean stratApply = false;
        if (s.getTemperatura() < 6 || s.getTemperatura() >= 16) {
            strategy = new StrategiaTemperatura();
            strategy.gestisciStanza(s);
            stratApply = true;
        }
        if (s.getUmidita() < 60 || s.getUmidita() >= 85) {
            strategy = new StrategiaUmidita();
            strategy.gestisciStanza(s);
            stratApply = true;
        }
        if (!stratApply)
            System.out.println("\nLa stanza non ha richiesto alcun intervento");
    }

    //PULL
    @Override
    public void update(Observable observable, Object o) {
        Stanza s = (Stanza) observable;
        setStrategy(s);
    }
```

3.1.2 Builder

Il pattern Builder è stato utilizzato per dare la responsabilità della creazione della scheda tecnica a una classe dedicata (*SchedaTecnicaBuilder*), questo poiché la costruzione richiede la lettura di un file per il corretto settaggio dei molteplici attributi della scheda tecnica stessa. Non è stata implementata nessuna classe dedicata a svolgere la funzione di *director* perché l'ordine con cui vengono inizializzati gli attributi non è rilevante.

```java
public final class SchedaTecnica {  //immutabile
    private final String provenienzaMosto;
    private final String annata;
    ...

    static class SchedaTecnicaBuilder {
        ...

        SchedaTecnicaBuilder(String nomeVino) throws FileNotFoundException{...}

        String buildProvenienza() {return listaInfoVini.get(numeroVino)[2];}

        String buildAnnata() {return listaInfoVini.get(numeroVino)[3];}

        ...

        SchedaTecnica build(){return new SchedaTecnica(this);}
    }

    private SchedaTecnica (SchedaTecnicaBuilder stb) {
        this.provenienzaMosto = stb.buildProvenienza();
        this.annata = stb.buildAnnata();
        ...
    }
```



3.2 Classi ed Interfacce

Per l'implementazione dell'applicazione sono state definite classi e interfacce, e sono state utilizzate alcune tra quelle contenute nel package java.util.

3.2.1 Cantina

La cantina contiene un array delle stanze (almeno una) che la compongono.

```
1  void mostraCatalogo() throws EmptyCatalogoException {
2          ArrayList<String[]> infoVini = new ArrayList<String[]>();
3          ...
4          for (Stanza stanzaIes : stanze) {
5              String[] infoVino = new String[titoli.length];
6              for (int j = 0; j < stanzaIes.getHashMapSize(); j++) {
7                  Vino vinoJes = stanzaIes.getVino(j);
8                  if (vinoJes != null) {
9                      //TIPO VINO
10                     if (vinoJes instanceof VinoRosso)
11                         infoVino[0] = "Rosso";
12                     ...
13                     //NOME
14                     infoVino[1] = vinoJes.getNome();
15                     //PROVENIENZA
16                     infoVino[2] = vinoJes.getSt().getProvenienzaMosto();
17                     ...
18                     infoVini.add(infoVino);
19                 }
20                 infoVino = new String[titoli.length];
21             }
22         }
23         ...
24         EditorODS.scriviNuovoODS(infoVini,titoli,"files/Catalogo"+numCatalogo+".ods");
25         ...
26     }
```

Il metodo *mostraCatalogo()*, si occupa di fare una ricerca dei vini disponibili nella cantina e li aggiunge insieme alle loro schede tecniche a una tabella in formato *.ods* che verrà mostrata come catalogo.

```
float[] vendi(String nome, int quantita) throws IllegalArgumentException {
        ...
            Vino v = getVino(nome);
            Botte botteV = getBotte(v);
            Stanza stanzaV = _getStanza(v);
            float[] acquisto = new float[2];
            float prezzo = Float.parseFloat(v.getSt().getPrezzoL());
            float oldCapacita = botteV.getCapacita();
            if (quantita >= botteV.getCapacita()) {
                botteV.setCapacita(0);
                stanzaV.removeVino(v);
                prezzo *= oldCapacita;
                acquisto[1] = oldCapacita;
                ...
            }
            else {
                botteV.setCapacita(oldCapacita-quantita);
                prezzo *= quantita;
                acquisto[1] = quantita;
                System.out.println("\n Sono stati venduti "+ quantita + " L del vino " + nome);
            }
            acquisto[0] = prezzo;
            return acquisto;
        ...
    }
```

Il metodo *vendi()* controlla se il vino richiesto è disponibile e in tal caso calcola il prezzo dell'acquisto a seconda della quantità(in litri) indicata: se tale quantità è uguale o superiore a quella disponibile il vino verrà rimosso dalla botte, altrimenti la quantità di vino presente nella botte verrà aggiornata.

```
    float calcolaSconto(float prezzo, float quantita) {
        if (quantita > 10) {
            float scontoPerc = (quantita - 10)*0.5f;
             System.out.println("\nSconto del "+scontoPerc+" %");
             float sconto = (prezzo/100)*scontoPerc;
            prezzo -= sconto;
            return prezzo;
        }
        return prezzo;
    }
```

Il metodo *calcolaSconto()* si occupa di aggiornare il prezzo di acquisto in base alla quantità richiesta.

```
15      private static Cantina cantina = null;
16
17      private Cantina(int numStanze) throws Error {
18          if (numStanze == 0)
19              throw new Error("La cantina deve avere almeno una stanza!");
20          stanze = new Stanza[numStanze];
21          for (int i=0; i<stanze.length; i++) {
22              stanze[i] = new Stanza(i);
23          }
24      }
25
26      public static Cantina getCantina(int numStanze) {
27          if (cantina == null)
28              cantina = new Cantina(numStanze);
29          return cantina;
30      }
```

Data la necessità di avere un' unica istanza della cantina, il cui riferimento deve essere facilmente disponibile in più punti del programma, la classe è stata implementata come un Singleton tramite una attributo statico di tipo Cantina, un costruttore privato e uno static factory method getCantina().

3.2.2 Stanza

La classe Stanza estende la classe Observable e ha come attributi il suo numero identificativo, temperatura e umidità. Inoltre al suo interno è presente una mappa che associa i vini presenti nella stanza alle botti nelle quali sono contenuti. Per fare ciò, è stata utilizzata una struttura dati LinkedHashMap per garantire l'ordine di inserimento delle botti nella stanza. Inoltre è presente un contatore che indica il numero di botti contenute nella stanza in quel momento.

```
26      public void varia() {
27          temperatura = (float)(-3 + Math.random() * 28);
28          umidita = (float)(35 + Math.random() * 60);
29          printStanza();
30          setChanged();
31          notifyObservers();
32      }
```

Il metodo *varia()* setta umidità e temperatura a un valore randomico e invia a prescindere la notifica ai monitor della stanza.

```
37      public void correggiTemperatura() {//visibilità package
38          System.out.println("\nCorrezione temperatura stanza n° "+numStanza
39          temperatura = (float)(6 + Math.random() * 10);
40          printStanza();
41      }
42
43      public void correggiUmidita() {//visibilità package
44          System.out.println("\nCorrezione umidità stanza n° "+numStanza+".
45          umidita = (float)(60 + Math.random() * 25);
46          printStanza();
47      }
```

I metodi *correggiTemperatura()* e *correggiUmidita()* settano i rispettivi attributi nella stanza a valori corretti.

```
1  public void addVino(Vino v) throws IllegalArgumentException {
2          ...
3          Cantina c = Cantina.getCantina(0);
4          if (c.getStanza(v) != null)
5              throw new IllegalArgumentException("\nLo stesso v
6          boolean found = false;
7          for (Botte key: botti.keySet()) {
8              if (botti.get(key) == null && !found) {
9                  botti.replace(key, v);
10                 ...
11                 found = true;
12             }
13         }
14         if (!found) {
15             botti.put(new Botte(numBotti), v);
16             ...
17             numBotti++;
18         }
19     }
```

Il metodo *addVino()* si occupa di aggiungere un vino (se non è già presente nella cantina) nella prima botte vuota della mappa, se tale botte non esiste, viene creata una nuova botte.

```
1  public void removeVino(Vino v) throws IllegalArgumentException, NullPointerException {
2          ...
3          boolean found = false;
4          for (Botte key : botti.keySet()) {
5              if (v == botti.get(key)) {
6                  botti.replace(key,null);
7                  found = true;
8              }
9          }
10         ...
11 }
```

Il metodo *removeVino()* si occupa di cercare un vino nella mappa e se il vino è presente lo rimuove dalla sua botte.

```
135     public void resetStanza() {
136         numBotti = 0;
137         botti = new LinkedHashMap<Botte,Vino>();
138     }
```

Il metodo *resetStanza()* si occupa di assegnare una nuova mappa alla stanza e di azzerare il numero delle botti.

3.2.3 Botte

La classe botte ha come attributi il numero identificativo e la quantità di vino che contiene oltre al metodo *setCapacita()* che setta tale quantità.

3.2.4 Cliente, Privato e Azienda

La classe Cliente è astratta anche se non espone metodi astratti, questo perché non sono richieste istanze dato che è una generalizzazione dei tipi concreti di clientela (Privato o Azienda). Ha come attributi nome, indirizzo e codice CAP, i quali sono condivisi con le classi derivate. La classe Privato estende Cliente aggiungendo agli attributi ereditati il cognome del cliente, mentre la classe Azienda aggiunge la partita IVA.

3.2.5 Vino

La classe Vino estende la classe Observable ed è stata progettata come classe astratta perché le istanze utilizzate concretamente sono sono i suoi sottotipi VinoRosso, VinoBianco e VinoRosato. Gli attributi dichiarati nella classe indicano i parametri osservabili nel vino, il suo nome e la sua scheda tecnica.

```
Vino(String nome) throws NullPointerException, IllegalArgumentException, FileNotFoundException {
        ...
        this.nome = nome;
        this.st = new SchedaTecnica.SchedaTecnicaBuilder(nome).build();
        this.anidrideSolforosa = 110 + (float)(Math.random() * 50);
        this.pH = 3.3f;
        this.zuccheriRiduttori = 160 + (float)(Math.random() * 50);
        this.gradoAlcolico = 10 + (float)(Math.random() * 4);
        this.ossigeno = (float)(Math.random() * 0.5);
}
```

Il costruttore *Vino()* assegna valori randomici ai parametri del vino entro una soglia che li rende accettabili indifferentemente dal fatto che sia di tipo VinoRosso, VinoBianco o VinoRosato.

Il metodo *correggiVino()* si occupa di riportare l'ossigeno a un valore accettabile permettendo così di non buttare il vino.
Il metodo *varia()* è astratto, il quale avrà diversa implementazione in ogni classe derivata.

3.2.6 VinoRosso, VinoBianco e VinoRosato

Queste classi estendono Vino ereditandone gli attributi e aggiungendo un attributo di tipo enum che ne indica la denominazione. Questo attributo controlla se il nome del vino fa parte di una fornitura.

```
    @Override
    public boolean varia() {
        anidrideSolforosa = 120 + (float)(Math.random() * 42);
        pH = (float)(3.28 + Math.random() * 0.24);
        ...
        setChanged();
        if (anidrideSolforosa > 160) {
            notifyObservers(new Float(anidrideSolforosa));
            return true;
        }
        if (pH < 3.3 || pH > 3.5) {
            notifyObservers(new Float(pH));
            return true;
        }
        ...
        return false;
    }
```

Il metodo *varia()* ha una diversa implementazione per ogni tipo concreto di vino perché si occupa di modificare i parametri del vino in un range di valori diverso a seconda del tipo. Inoltre controlla se e quale parametro ha superato la soglia di accettabilità (diversa per ogni tipo e per ogni parametro) notificandolo ai monitor.

3.2.7 SchedaTecnica

La classe SchedaTecnica racchiude al suo interno le informazioni che rendono univoca un' istanza del vino: provenienza mosto, annata, qualifica, vitigni, prezzo al litro, ecc.

Ha un costruttore privato che setta i suoi attributi attraverso le funzioni di costruzione di SchedaTecnicaBuilder che è definita come *nested class* al suo interno. Le istanze della classe sono oggetti immutabili poiché le informazioni relative a un vino non devono poter essere modificate durante tutta la sua esistenza.

```
11  public final class SchedaTecnica {
12      private final String provenienzaMosto;
13      private final String annata;
14      private final String qualifica;
15      private final String bio;
16      private final String vitigni;
17      private final String prezzoL;
```

3.2.8 MonitorVino

La classe MonitorVino implementa l'interfaccia Observer e quindi il metodo *update()* dove, attraverso un downcast, recupera il riferimento al vino e il parametro che sono stati notificati passandoli al metodo *setStrategy(),* che si occupa di istanziare la strategia corretta(che mantiene come attributo) e di invocarci *gestisciVino()*.

3.2.9 StrategiaVino, StrategiaGenerale e StrategiaOssigeno

StrategiaGenerale e StrategiaOssigeno implementano l'unico metodo dichiarato dall' interfaccia StrategiaVino, ovvero *gestisciVino()*.

```
8       @Override
9   @   public void gestisciVino(Vino v) { v.correggiVino(); }
12  }
```

Nell' implementazione del metodo in StrategiaOssigeno viene invocata la correzione del vino(*correggiVino()* della classe Vino).

```
@Override
public void gestisciVino(Vino v) {
    ...
       Cantina cantina = Cantina.getCantina(0);
       cantina.getStanza(v).removeVino(v);
       ...
    ...
}
```

Mentre in quella di StrategiaGenerale viene recuperata la stanza in cui il vino è contenuto e viene chiamata la rimozione da essa (remove*Vino()* della classe Stanza).

3.2.10 ControlloreStanza

La classe ControlloreStanza implementa l'interfaccia Observer e quindi il metodo *update()* dove, attraverso un downcast, recupera il riferimento alla stanza di cui è stato notificato il cambiamento e lo passa al metodo *setStrategy(),* che ha il compito di recuperare lo stato della stanza al fine di istanziare la corretta strategia (che mantiene come attributo) di gestione, e invocarci *gestisciStanza()*.

3.2.11 StrategiaStanza, StrategiaTemperatura e StrategiaUmidita

StrategiaTemperatura e StrategiaUmidita implementano l'unico metodo dichiarato dall' interfaccia StrategiaStanza, ovvero *gestisciStanza()*.

```
    @Override
    public void gestisciStanza(Stanza s) { s.correggiUmidita(); }
}
```

Nell' implementazione del metodo di StrategiaUmidita viene invocata la correzione dell' umidità della stanza (*correggiUmidita()* della classe Stanza).

```
    @Override
    public void gestisciStanza(Stanza s) {
        if (s.getTemperatura() < -1 || s.getTemperatura() > 22 ) {
            s.resetStanza();
        }
        s.correggiTemperatura();
    }
}
```

Mentre in quella di StrategiaTemperatura viene prima fatto un controllo per verificare che il valore di temperatura non sia critico, poiché in tal caso viene chiamata la rimozione di tutte le botti e quindi i vini contenuti dalla stanza(*resetStanza()* della classe Stanza), infine a prescindere viene invocata la correzione del valore della temperatura(*correggiTemperatura()* della classe Stanza).

3.2.12 OperatorePageController

La classe OperatorePageController ha la responsabilità di fornire un'interfaccia all' operatore esponendo dei metodi di gestione della cantina.

```
1 public void accettaRifornimento(String path) {
2         ArrayList<Vino> fornitura = new ArrayList<Vino>();
3         File file = new File(path);
4         ArrayList<String[]> listaInfoViniP = EditorODS.leggiODS(file);
5         for (int i = 0; i < listaInfoViniP.size(); i++) {
6             String tipo = listaInfoViniP.get(i)[0];
7             String nome = listaInfoViniP.get(i)[1];
8             if (tipo.equals("Rosso")) {
9                 ...
10                    fornitura.add(new VinoRosso(nome));
11                ...
12            }
13            ...
14        }
15        for (int i = 0; i < fornitura.size(); i++) {
16            ...
17                int j = i % c.getNumeroStanze();
18                c.getStanza(j).addVino(fornitura.get(i));
19            ...
20        }
21 }
```

Il metodo accetta*Rifornimento()* si occupa di leggere una fornitura (tabella in formato *.ods*), creare le istanze dei vini indicati a seconda del tipo e aggiungerli suddividendoli equamente tra le stanze della cantina.

```java
public void gestisciCantina() {
        int k = 0;
        while (k < c.getNumeroStanze()) {
            ...
            c.getStanza(k).addObserver(new ControlloreStanza());
            c.getStanza(k).varia();
            ...
                ...
                for (int j = 0; j < c.getStanza(k).getHashMapSize(); j++) {
                    Vino v = c.getStanza(k).getVino(j);
                    v.addObserver(new MonitorVino());
                    v.varia();
                }
            }
            k++;
}
```

Il metodo *gestisciCantina()* assegna ad ogni stanza un observer e invoca il metodo *varia()* innescando così il meccanismo di gestione. Dopo aver gestito la stanza, prima di passare alla prossima, la stessa cosa viene fatta per ogni vino che contiene.

3.2.13 ClientePageController

La classe ClientePageController ha la responsabilità di fornire un'interfaccia al cliente esponendo dei metodi per visionare il catalogo dei vini disponibili ed effettuare un acquisto.

```java
public void compraVino() {
        Cantina cantina = Cantina.getCantina(0);
        ...
            float[] acquisto = cantina.vendi(nomeVino, quantita);
            ...
            if (cliente instanceof Azienda)
                acquisto[0] = cantina.calcolaSconto(acquisto[0], acquisto[1]);
            ...
        ...
}
```

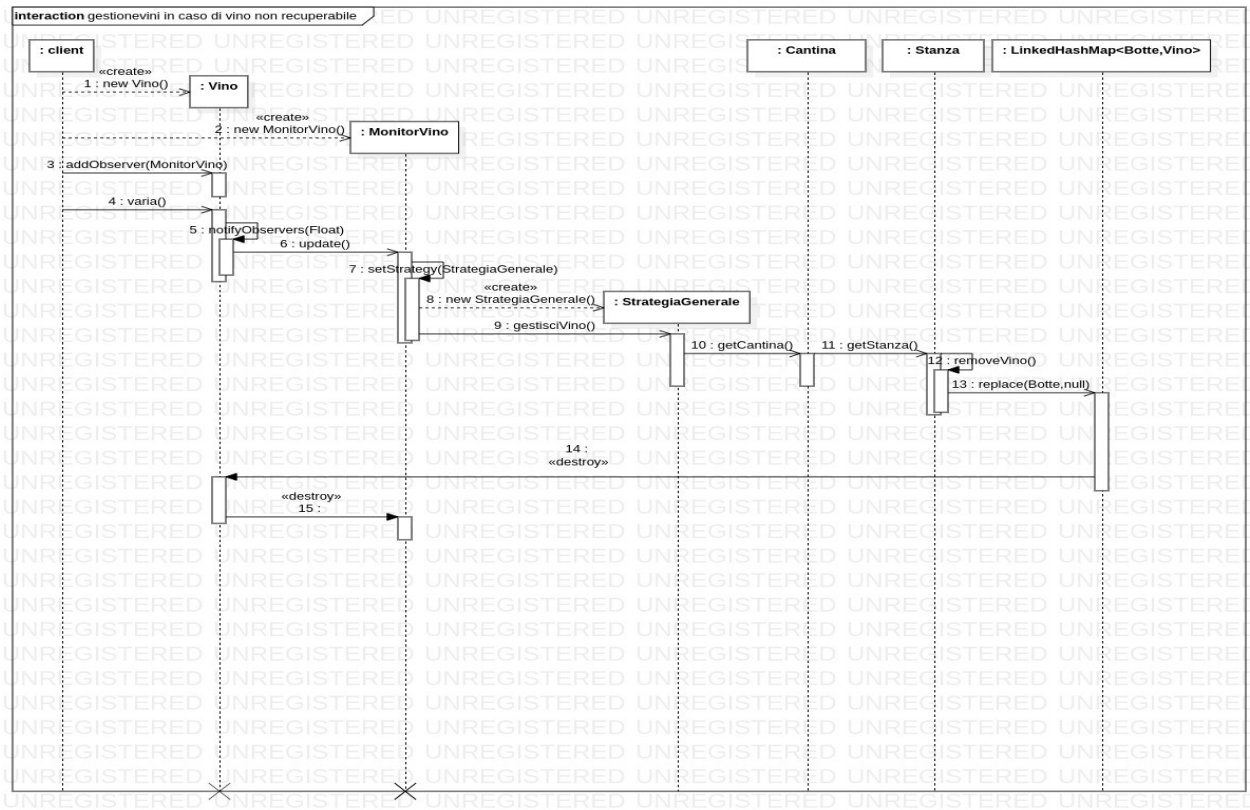
Il metodo *compraVino()* permette al cliente di specificare nome del vino e quantità che intende acquistare(input da tastiera), e ottenere, attraverso il metodo *vendi()* di Cantina il prezzo finale di acquisto, con uno sconto applicato se il cliente in questione è un'azienda.

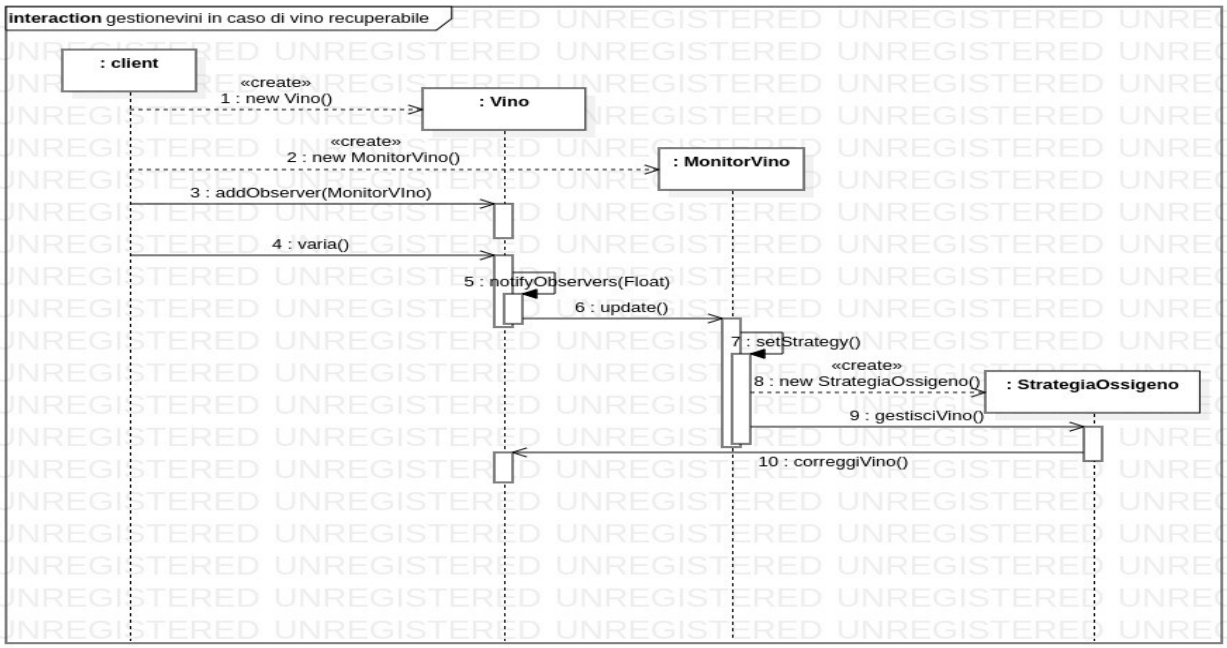
```java
public final void visionaCatalogo() throws EmptyCatalogoException {
    Cantina c = Cantina.getCantina(0);
    c.mostraCatalogo();
}
```

Il metodo *visionaCatalogo()* permette attraverso l'invocazione di *mostraCatalogo()* della Cantina di consultare la tabella dei vini disponibili all'acquisto.

3.3 SequenceDiagram

I sequence diagram mostrati rappresentano la differenza di esecuzione tra un caso di gestione di un vino che deve essere buttato e uno in cui deve essere corretto.





Il software prevede anche la presenza di Unit Testing del codice dove la strategia utilizzata per scrivere i casi di test è stata quella di voler testare il corretto funzionamento dei singoli metodi che con la loro interazione soddisfano i casi d'uso. Quindi per ogni classe principale è stata definita una classe di test i cui casi sono stati pensati per lo più in prospettiva funzionale con approccio *Black-box*, cioè andando a testare il corretto funzionamento dei metodi tralasciando la struttura del software e la sua possibile evoluzione. Ciò nonostante sono stati scritti anche pochi casi di test pensati in prospettiva comportamentale e strutturale con approccio *White-box,* cioè testando la corretta sequenza di eventi e il corretto funzionamento di una struttura interna del software.

4.1 CantinaTest

Questa classe racchiude i casi che mirano a testare il corretto funzionamento dei metodi della classe Cantina, in particolare:

- In *getCantina()* non deve essere possibile istanziare una cantina vuota
- In *vendi()* non deve essere possibile vendere un vino che non è presente nella cantina
- In *mostraCatalogo()* non deve essere possibile mostrare il catalogo se non ci sono vini disponibili
- In *getStanza(Vino)* dato un riferimento a un vino deve essere possibile recuperare il riferimento alla stanza che lo contiene, altrimenti il metodo deve restituire un riferimento nullo

```
55      @Test
56      public void vendiTest() throws InvalidAttributeValueException, FileNotF
57          c = Cantina.getCantina(4);
58          Vino v = new VinoRosso( nome: "Morellino");
59          c.getStanza( numStanza: 0).addVino(v);
60          float[] acquisto1 = c.vendi( nome: "Morellino",  quantita: 60);
61          float prezzoAspettato = Float.parseFloat(v.getSt().getPrezzoL());
62          assertEquals( expected: prezzoAspettato*60, acquisto1[0],  delta: 0);
63          assertEquals( expected: 60, acquisto1[1],  delta: 0);
64          Botte botte1 = c.getStanza(v).getBotte(v);
65          assertEquals( expected: 40, botte1.getCapacita(),  delta: 0);
66          float[] acquisto2 = c.vendi( nome: "Morellino",  quantita: 50);
67          assertEquals( expected: 40, acquisto2[1],  delta: 0);
68          assertNull(c.getStanza(v));
69      }
```

Testa la vendita di un vino in cantina: in particolare testa se il prezzo (senza sconto) ritornato è corretto, se la quantità comprata è quella richiesta, se la capacità della botte è aggiornata correttamente, se vende solo la quantità disponibile in caso di richiesta maggiore, e infine in tal caso, se il vino è stato rimosso dalla cantina.

```java
@Test
public void testMostraCatalogo() throws InvalidAttributeValueException, EmptyCatalogoException, Fil
    Cantina c = Cantina.getCantina(4);
    Vino v = new VinoRosso( nome: "Morellino");
    Vino w = new VinoBianco( nome: "Chamonix");
    c.getStanza( numStanza: 0).addVino(v);
    c.getStanza( numStanza: 0).addVino(w);

    c.mostraCatalogo();
    ArrayList<String[]> infoVini1 = EditorODS.leggiODS(new File( pathname: "files/Catalogo0.ods"));
    assertEquals(infoVini1.get(0)[1], v.getNome());

    c.getStanza( numStanza: 0).removeVino(v);

    c.mostraCatalogo();
    ArrayList<String[]> infoVini2 = EditorODS.leggiODS(new File( pathname: "files/Catalogo1.ods"));
    assertNull(c.getStanza( numStanza: 0).getVino( idBotte: 0));
    assertNotEquals(infoVini2.get(0)[1], v.getNome());

    c.getStanza( numStanza: 0).removeVino(w);
}
```

Testa la corretta corrispondenza tra i vini presenti nella cantina e quelli del catalogo, in particolare in questo test il fatto che un vino rimosso dalla cantina sia rimosso correttamente anche nel catalogo.

```java
@Test
public void testCalcolaSconto() {
    Cantina c = Cantina.getCantina(4);
    float prezzoFinale1 = c.calcolaSconto( prezzo: 100,  quantita: 20);
    float prezzoFinale2 = c.calcolaSconto( prezzo: 100,  quantita: 5);
    assertEquals( expected: 95f, prezzoFinale1,  delta: 0);
    assertEquals( expected: 100f, prezzoFinale2,  delta: 0);
}
```

Testa che uno sconto su un acquisto sia applicato correttamente solo in caso di quantità richiesta maggiore di una certa soglia (10 Litri).

4.2 StanzaTest

Questa classe racchiude i casi che mirano a testare il corretto funzionamento dei metodi della classe Stanza, in particolare:

- In *correggiTemperatura()* e *correggiUmidita()* i rispettivi parametri devono essere riportati all'interno di un range accettabile
- In *addVino()* non deve poter essere aggiunto un riferimento a vino nullo oppure un riferimento a un vino già presente nella cantina
- In *removeVino()* non deve essere possibile rimuovere un riferimento a vino nullo o che non è presente nella cantina

```
59      @Test
60      public void newBotteAddVino() throws InvalidAttributeValueEx
61          for (int i=0; i< 3; i++)
62              s.addVino(new VinoRosso( nome: "BolgheriSassicaia"));
63          Vino v = new VinoBianco( nome: "Chamonix");
64          s.addVino(v);
65          Botte botte = s.getBotte(v);
66          assertEquals( expected: 4,s.getHashMapSize());
67          assertEquals(v,s.getVino( idBotte: 3));
68      }
```

Testa che, al momento di aggiungere un nuovo vino alla cantina, venga creata una nuova botte in caso non ce ne siano vuote.

```
71      @Test
72      public void emptyBotteAddVino() throws InvalidAttributeV
73          s.addVino(new VinoRosso( nome: "BolgheriSassicaia"));
74          Vino oldVino = new VinoBianco( nome: "Chamonix");
75          s.addVino(oldVino);
76          s.addVino(new VinoRosso( nome: "BolgheriSassicaia"));
77          Botte botte = s.getBotte(oldVino);
78          s.removeVino(oldVino);
79          Vino newVino = new VinoRosso( nome: "Morellino");
80          s.addVino(newVino);
81          assertEquals(newVino,s.getVino( idBotte: 1));
82      }
```

Testa che un vino venga inserito nella prima botte vuota disponibile in una stanza.

```
102     @Test
103     public void testRemoveVino() throws InvalidAttributeValu
104         s.addVino(new VinoRosso( nome: "BolgheriSassicaia"));
105         Vino v = new VinoBianco( nome: "Bastianich");
106         s.addVino(v);
107         Botte botte = s.getBotte(v);
108         s.addVino(new VinoBianco( nome: "Chamonix"));
109         s.removeVino(v);
110         assertNull(s.getVino( idBotte: 1));
111     }
```

Testa che dopo la rimozione di un vino da una botte, il riferimento a quel vino in tale botte sia nullo.

4.3 VinoTest

Questa classe racchiude i casi che mirano a testare il corretto funzionamento dei metodi della classe Vino e delle sue derivate, in particolare:

- In *correggiVino()* il valore dell'ossigeno deve essere riportato all'interno di un range accettabile
- Non deve essere possibile creare un vino con nome nullo, non appartenente a nessuna fornitura o alla lista di nomi definita nell' enumerazione di quel determinato tipo
- In *varia()* di ogni tipo concreto di vino deve essere restituito un valore booleano *true* solo se almeno un parametro ha superato la sua soglia di accettabilità

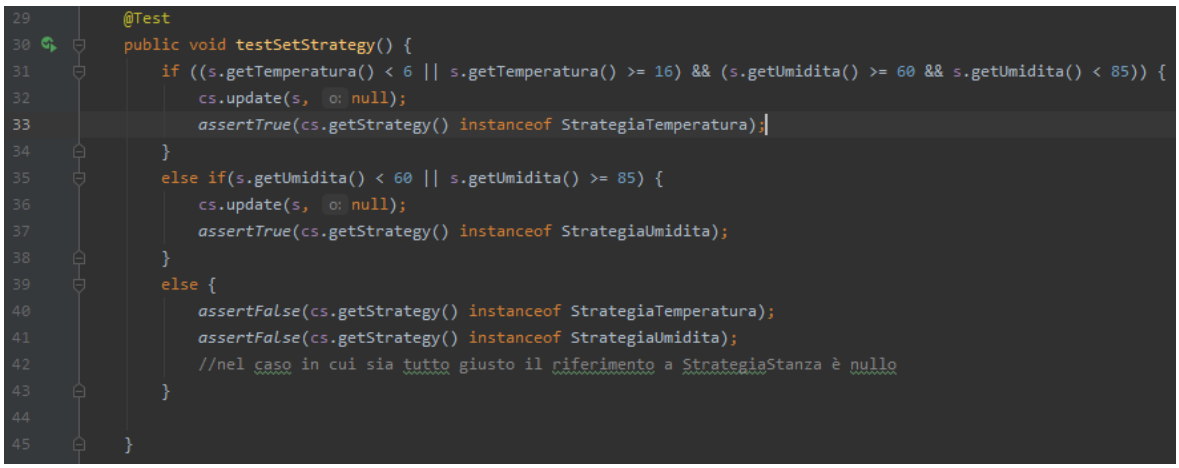
4.4 SchedaTecnicaTest

La classe testa che alla creazione di un vino sia creata una scheda tecnica i cui valori devono corrispondere a quelli specificati nella riga della tabella di fornitura dove è presente il nome del vino creato.

```
12      @Test
13      public void testBuilder() throws InvalidAttributeValueException, FileNotFoundException
14          Vino v = new VinoRosso( nome: "Morellino");
15          assertEquals( expected: "Scansano", v.getSt().getProvenienzaMosto());
16          assertEquals( expected: "2019", v.getSt().getAnnata());
17          assertEquals( expected: "DOCG", v.getSt().getQualifica());
18          assertEquals( expected: "No", v.getSt().getBio());
19          assertEquals( expected: "Sangiovese 96%, Ciliegiolo 4%", v.getSt().getVitigni());
20          assertEquals( expected: "28.20", v.getSt().getPrezzoL());
21      }
```

4.5 ControlloreStanzaTest

Questa classe racchiude i casi che mirano a testare il corretto funzionamento dei metodi della classe ControlloreStanza, in particolare nel metodo *update()* non deve essere possibile passare un riferimento nullo a Observable.



```
29      @Test
30      public void testSetStrategy() {
31          if ((s.getTemperatura() < 6 || s.getTemperatura() >= 16) && (s.getUmidita() >= 60 && s.getUmidita() < 85)) {
32              cs.update(s,  o: null);
33              assertTrue(cs.getStrategy() instanceof StrategiaTemperatura);
34          }
35          else if(s.getUmidita() < 60 || s.getUmidita() >= 85) {
36              cs.update(s,  o: null);
37              assertTrue(cs.getStrategy() instanceof StrategiaUmidita);
38          }
39          else {
40              assertFalse(cs.getStrategy() instanceof StrategiaTemperatura);
41              assertFalse(cs.getStrategy() instanceof StrategiaUmidita);
42              //nel caso in cui sia tutto giusto il riferimento a StrategiaStanza è nullo
43          }
44
45      }
```

Testa che vengano istanziati i corretti tipi concreti di strategia in base a quali parametri della stanza hanno superato la soglia.

4.6 StrategiaTemperaturaTest

La classe testa che se la temperatura di una stanza ha raggiunto valori critici vengano rimosse tutte le botti al suo interno, e infine che la temperatura sia riportata a un valore accettabile.

```
29      @Test
30      public void testGestisciStanza() {
31          assertEquals( expected: 2, s.getHashMapSize());
32          st.gestisciStanza(s);
33          assertEquals( expected: 0, s.getHashMapSize());
34          assertTrue( condition: s.getTemperatura() >= 6 && s.getTemperatura() < 16);
35      }
36  }
```

4.7 MonitorVinoTest

Questa classe racchiude i casi che mirano a testare il corretto funzionamento dei metodi della classe MonitorVino, in particolare nel metodo *update()* non deve essere possibile passare un riferimento nullo a Observable, passare un riferimento nullo a Object o un riferimento a Object di cui non si può fare il downcast a Float.

```
56      @Test
57      public void testSetStrategy() {
58          v.addObserver(mv);
59          do {
60              v.varia();
61          } while (v.getAnidrideSolforosa() <= 160 && v.getpH() >= 3.3 && v.getpH() <= 3.5 &&
62                  v.getZuccheriRiduttori() <= 210 && v.getGradoAlcolico() >= 10 && v.getGradoAlcolico() <= 16);
63          if (v.getAnidrideSolforosa() > 160 || v.getpH() < 3.3 || v.getpH() > 3.5 ||
64                  v.getZuccheriRiduttori() > 210 || v.getGradoAlcolico() < 10 || v.getGradoAlcolico() > 16)
65          {
66          assertTrue(mv.getStrategy() instanceof StrategiaGenerale);
67          } else {
68              assertTrue(mv.getStrategy() instanceof StrategiaOssigeno);
69          }
70      }
71  }
```

Testa che vengano istanziati i corretti tipi concreti di strategia in base a quali parametri della stanza hanno superato la soglia.