EcoTeam

Sistema per la gestione di un'azienda che si occupa del 110%

Alessio Rocchi e Leonardo Tozzi Senesi

Gennaio 2022

Indice

1 Breve introduzione

Con questo progetto abbiamo voluto rappresentare la gestione di un'azienda che si occupa di efficientamento energetico di condomini. In particolare abbiamo voluto rappresentare come avviene la comunicazione tra gli amministratori richiedenti, la nostra azienda, con un proprio rappresentante e i vari tecnici che si occupano di effettuare i sopralluoghi per verificare le idoneità dei differenti condomini. In più è presente anche la comunicazione del responsabile della nostra azienda il quale può richiedere l'intervento di una delle differenti imprese edili che si occupano di effettuare l'efficientamento energetico grazie al loro intervento sui differenti condomini.

2 Statement iniziale

La nostra applicazione permette di mettere in comunicazione la nostra azienda, che si occupa del 110%, a cui è posto a capo il responsabile con i vari amministratori, i tecnici che lo aiutano nella valutazione dei requisiti per l'effettuazione dell'efficientamento energetico e le imprese edili le quali effettueranno i veri e propri interventi sui differenti condomini.

In particolare vogliamo che ciascun amministratore possa visualizzare tutti i condomini amministrati e aggiungere o rimuovere alcuni di essi,se necessario. L'amministratore deve essere in grado di richiedere il 110% per uno o più dei suoi condomini.

Il responsabile dell'azienda, d'altra parte, deve poter visualizzare tutte le richieste relative ai vari condomini e poterle accettare o rifiutare. Se una di esse viene accettata deve essere inviata una richiesta di sopralluogo, per il condominio, ad uno dei tecnici specializzati iscritti al sistema. Il responsabile si deve occupare inoltre, dopo aver visionato i responsi relativi a vari sopralluoghi, di fare le offerte agli amministratori, i quali possono accettare o rifiutare l'offerta ricevuta. Nel caso in cui l'offerta venga accettata il responsabile può inoltrare la richiesta di inizio lavoro a una delle imprese edili registrate nel sistema.

I vari tecnici d'altro canto devono poter visualizzare le loro richieste di sopralluogo, a quali condomini si riferiscono e inviare i responsi dei vari sopralluoghi.

Le imprese invece possono visualizzare gli operai che lavorano per loro, assumerne di nuovi o licenziarne alcuni. Possono anche visualizzare sia i cantieri attivi sia tutti quelli aperti anche se conclusi, e su quale condominio sono stati effettuati. A lavoro terminato l'impresa può chiudere il cantiere.

L'azienda poi deve tenere anche uno storico delle richieste, delle offerte e dei sopralluoghi effettuati con tutti i loro parametri, in modo che essi siano sempre disponibili per essere consultati.

3 Requisiti

3.1 Use Case Diagram

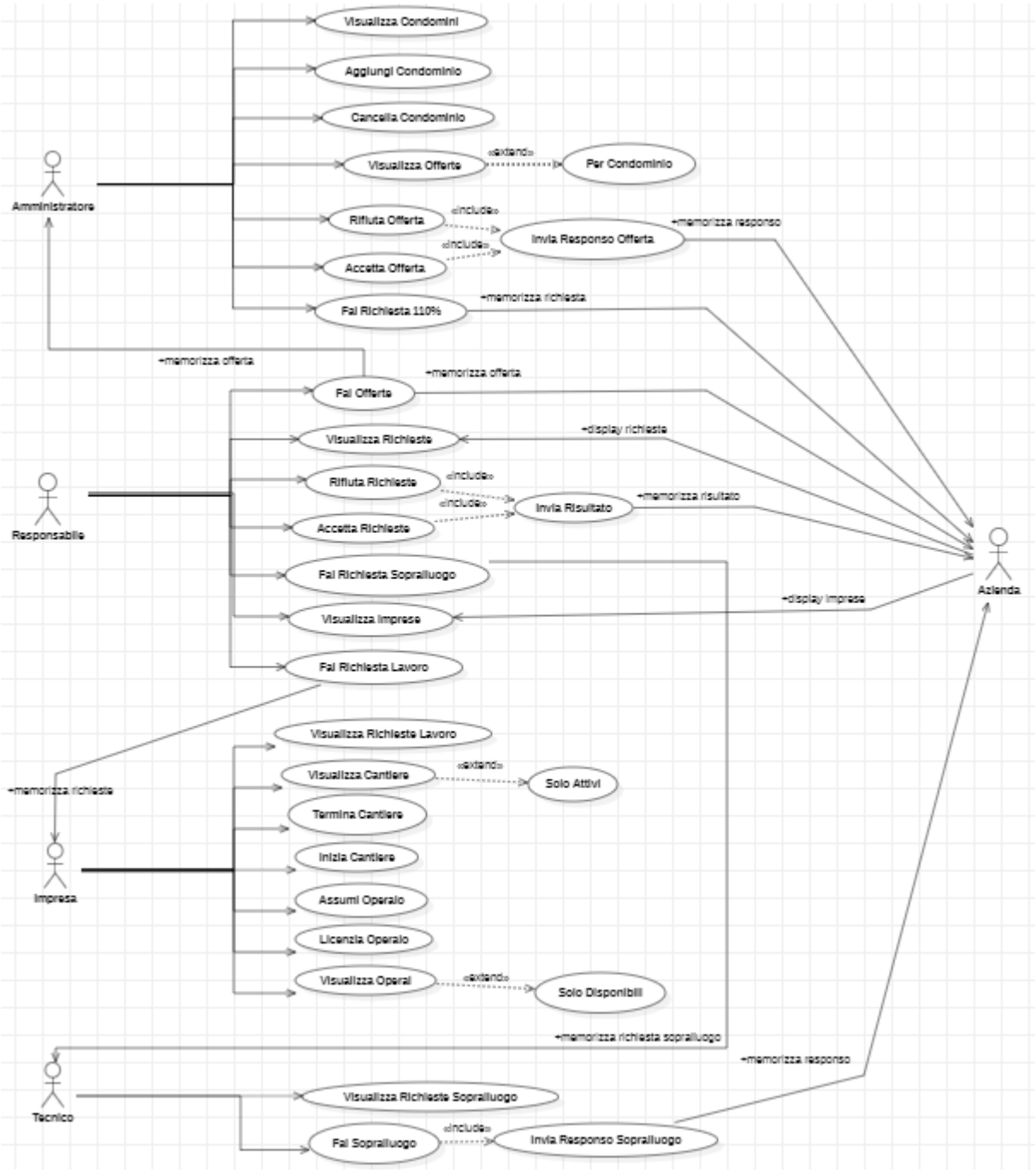


Figura 1: Use Case Diagram

4 Progettazione

4.1 Problema

Vediamo come il principale problema da risolvere è la comunicazione tra tutte le varie parti del sistema, infatti il responsabile deve poter relazionarsi sia con tutti gli amministratori presenti nel sistema, ma anche con tutti i tecnici e tutte le imprese, in più deve anche tenere traccia delle varie richieste degli amministratori e dei responsi dei sopralluoghi effettuati dai tecnici. Ciò significherebbe creare una classe estremamente complicata che dovrebbe tenere traccia di tutte le entità create nel nostro sistema, cosa che renderebbe il codice molto complicato sia per implementazione che per leggibilità.

4.2 Risoluzione

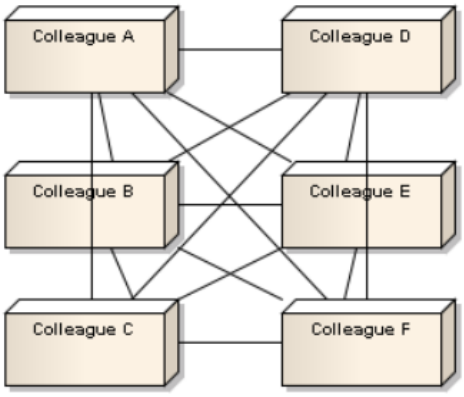
Per risolvere questo problema abbiamo utilizzato 2 design pattern nel quale il principale è il pattern Mediator che ci permette di disaccoppiare il compito di tener traccia di tutti i vari parametri quali le offerte, le richieste ecc... e il compito di comunicazione tra le varie parti del sistema le quali non devono referenziarsi a vicenda dato che la comunicazione passa attraverso un "nodo" centrale rappresentato dal Mediator che gestisce e implementa i metodi di comunicazione.

Questo pattern ci permette anche di affidare il compito di memorizzare tutto ciò che viene condiviso tra le varie parti del sistema, come le offerte e le richieste, alla classe centrale Azienda, togliendo la responsabilità di ciò alla classe Responsabile. Se non ci fosse questa accortezza quando si dovrebbe cambiare responsabile bisognerebbe rimemorizzare nella nuova classe tutte le informazioni della vecchia e in più notificare tutto il sistema del cambiamento, aggiornando tutte le singole classi che si referenziano direttamente con l'azienda.

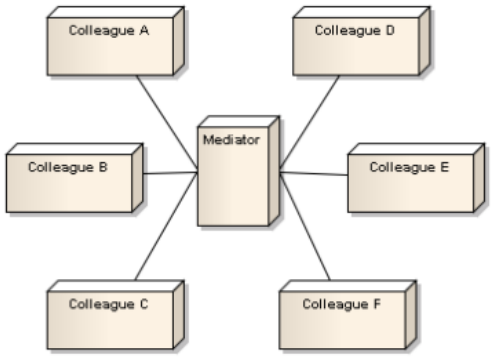
Questo rende anche il nostro sistema in grado di ricevere richieste dagli amministratori e i responsi dei sopralluoghi da parte dei tecnici anche se momentaneamente in assenza del responsabile.

Abbiamo anche utilizzato il pattern Singleton per fare in modo che non possano essere create due istanze del pattern Mediator. Questo potrebbe essere critico nel riutilizzo o nell'espansione del sistema, infatti se venissero creati due Mediator si creerebbero due parti del sistema isolate ognuna impossibilitata a comunicare con l'altra. Ciò rappresenterebbe una grande criticità risolta con il Singleton che fa in modo che non possa esistere più di una istanza del Mediator.

4.2.1 Mediator



(a) Complessa rete di comunicazione



(b) Rete semplificata tramite un Mediatore

Si tratta di un pattern comportamentale, ossia operante nel contesto delle interazioni tra oggetti, che ha l'intento di disaccoppiare entità del sistema che devono comunicare fra loro. Il pattern infatti fa in modo che queste entità non si referenzino reciprocamente, agendo da "mediatore" fra le parti. Il beneficio principale è il fatto di poter di modificare agilmente le politiche di interazione, poiché le entità coinvolte devono fare riferimento al loro interno solamente al mediatore. Nel gergo del Pattern queste entità vengono chiamate *Colleghi*. Questo Pattern è composto dai seguenti partecipanti:

- **Mediator**: definisce una interfaccia per comunicare con i Colleague. Questo compito è affidato alla classe astratta denominata Mediator.
- **ConcreteMediator**: mantiene la lista dei colleghi e implementa lo scambio di messaggi tra di loro. Questo compito è stato affidato alla classe *Azienda* la quale ha anche il compito di mantenere in memoria i risultati dei sopralluoghi, le richieste e le offerte.
- **Colleague**: definisce l'interfaccia dei Colleague. Questo compito è stato affidato alla classe astratta *Persona*.
- **ConcreteColleague**: implementa il singole collega e le modalità di comunicazione con il Mediator. Nel nostro progetto questi sono le classi *Responsabile, Amministratore e Tecnico* perchè esse sono le principali coinvolte nella comunicazione.

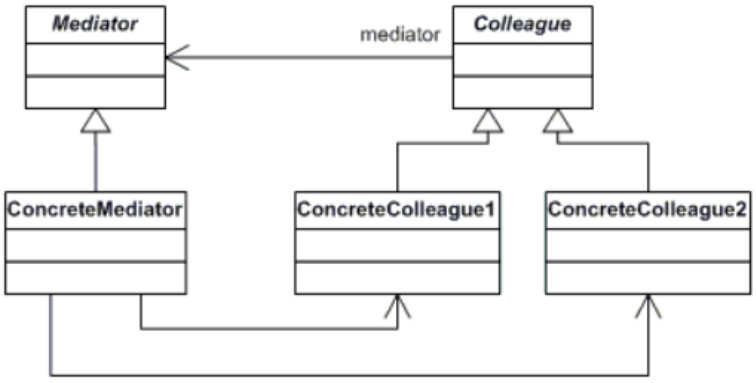
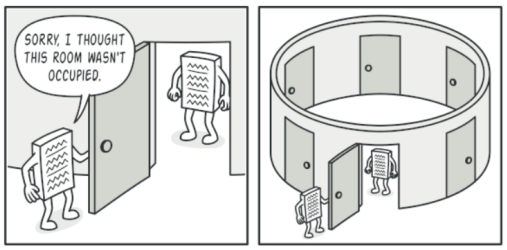
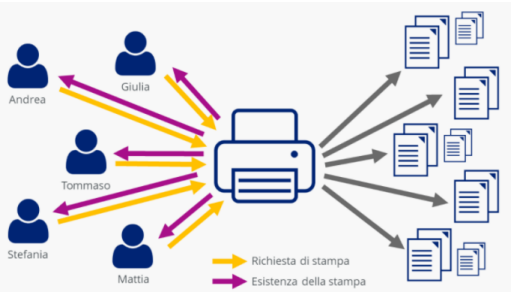


Figura 3: Mediator

4.2.2 Singleton



(a) I clienti potrebbero non rendersi nemmeno conto di lavorare sempre con lo stesso oggetto



(b) Esempio di Singleton

I pattern creazionali risolvono problematiche inerenti l'istanziazione degli oggetti. Il Singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

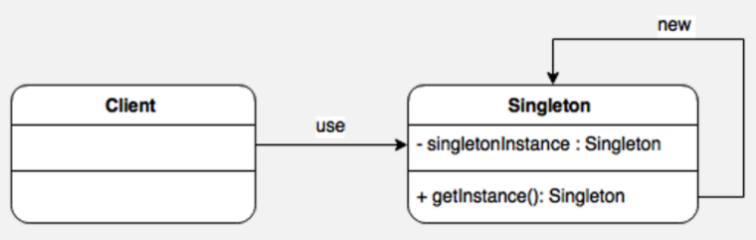


Figura 5: Singleton

Come detto in precedenza utilizzando Questo pattern sulla classe *Azienda* possiamo assicurarci che possa esistere solamente un'istanza alla volta. Questo risolve la criticità data dall'istanziamento

di due classi che fungono da mediator andando direttamente ad eliminare la possibilità che questo accada.

4.3 UML

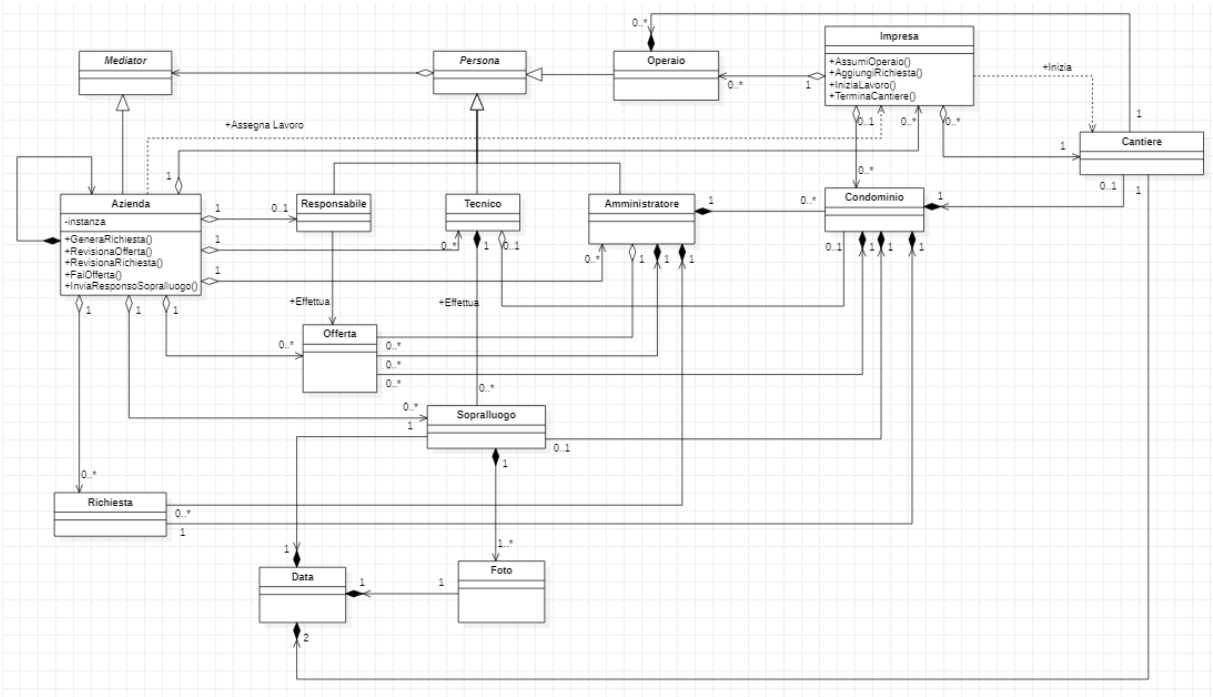


Figura 6: Diagramma UML

4.3.1 Discussione UML

Da questo diagramma possiamo vedere come agisce il Pattern Mediatorn: nessuna delle classi che derivano dalla classe Persona referenzia le altre direttamente, ma tutte hanno un riferimento alla classe Azienda che ha appunto la funzione di implementare i metodi di comunicazioni utilizzati da queste classi.

Ed essa, essendo il "nodo" centrale, è anche affidato il compito di memorizzare tutti i dati di comunicazione come le istanze della classe Offerta, quella della classe Sopralluogo e quelle della classe Richiesta, così che si possa anche avere uno "storico" degli interventi fatti.

In più la classe Azienda, come è consono al pattern Singleton, ha un riferimento a se stessa che utilizza per verificare che, durante tutta la vita del sistema, esista una singola istanza di se ed appunto evitare la creazione di altri oggetti di tipo Azienda.

5 Implementazione classi principali

5.1 Amministratore

La classe Amministratore rappresenta l'amministratore di condominio e si occupa di tenere conto dei condomini amministrati dall'utente in un ArrayList e da modo ad esso di fare richieste di efficientamento energetico alla nostra azienda. L'utente può anche accettare o rifiutare le offerte che gli arrivano dall'azienda le quali possono essere anche visualizzate attraverso specifici metodi.

Questa classe eredita i metodi e gli attributi dalla classe astratta Persona, che come detto in precedenza è l'analogo della classe Collegue del pattern Mediator, in modo che possa comunicare con le altre classi attraverso i metodi della classe Azienda.

Metodi:

- *aggiungiCondominio* e *aggiungiOfferta*: servono per memorizzare nelle rispettive ArrayList rispettivamento un condominio che si aggiunge alla lista di quelli che sono amministrati o un'offerta che viene inviata dal Responsabile e giunge all'Amministratore attraverso la classe Azienda.

```
1     public void aggiungiCondominio(Condominio c) {
2         condominio.add(c);
3     }
4
```

```
1     public void aggiungiOfferta(Offerta o) {
2         offerte.add(o);
3     }
4
```

- *stampaCondomini*: ha lo scopo di visualizzare i condomini amministrati dall'Amministratore sotto la forma di una lista numerata, i cui indici sono usati per i metodi successivi.

```
1     public void stampaCondomini() {
2         for (int i = 0; i < condominio.size(); i++) {
3             System.out.println("Numero " + i + " :");
4             condominio.get(i).stampaInfo();
5             System.out.println();
6         }
7     }
8
```

- *faiRichiesta*: genera una richiesta di preventivo per un determinato condominio appartenente a quelli amministrati. Per selezionare univocamente il condominio per il quale di vuole effettuare la richiesta si utilizza l'indice dello stesso all'interno del relativo ArrayList di condomini. Se l'indice è fuori dal range di quelli possibili si stampa un messaggio di errore.

  Questo tipo di selezione è fatta per simulare l'interfaccia grafica sulla quale sarebbe presente una lista di tutti i condomini. L'utente sceglierebbe tra essi semplicemente cliccando sulla linea corrispondente della lista, così selezionando il condominio corrispondente ad essa.

```
1     public void faiRichiesta(int index) {
2         if (index < 0 || index > condominio.size()) {
3             System.out.println("Condominio selezionato non    valido");
4         } else {
5             azienda.generaRichiesta(condominio.get(index), this);
6         }
7     }
8
```

- *consultaOfferte*: serve a visualizzare tutte le offerte non ancora accettate o rifiutate che l'utente relativo alla classe ha ricevuto dall'Azienda. Questa visualizzazione viene fatta sotto forma di lista numerata.

  Se non fossero presenti offerte non ancora accettate o rifiutate il metodo lo notifica scrivendo: "Non ci sono nuove offerte".

```
1     public void consultaOfferte() {
2
3         int num = 0;
4
5         for (int i = 0; i < offerte.size(); i++) {
6             if (!offerte.get(i).revisionata()) {
7                 num++;
8                 System.out.println("Offerta numero " + i + ":");
9                 offerte.get(i).stampaInfo();
10                System.out.println();
11            }
12        }
13
14        if (num == 0) {
15            System.out.println("Non ci sono nuove Offerte");
16            System.out.println();
17        }
18    }
19
```

- *consultaTutteOfferte*: serve per visualizzare tutte le offerte che l'utente ha ricevuto durante tutto l'utilizzo del sistema, ciò quindi include anche le offerte già accettate o rifiutate.

  Anche questo metodo visualizza le offerte sotto forma di lista numerata.

```
1     public void consultaTutteOfferte() {
2         for (int i = 0; i < offerte.size(); i++) {
3             System.out.println("Offerta numero " + i + ":");
4             offerte.get(i).stampaInfo();
5             System.out.println();
6
7         }
8     }
9
```

- *revisionaOfferta*: serve per valutare un'offerta e decidere, a seconda del booleano in ingresso, se accettarla o meno. Per selezionare univocamente un'Offerta si utilizza l'indice della stessa nell'ArrayList delle offerte.

  Questo riprende la precedente funzione *faiRichiesta* in modo che l'interfaccia grafica funzioni in modo analogo sia per la selezione dei condomini che per quella delle offerte.

```
1     public void revisionaOfferta(int index, boolean accetto) {
2         if (index < 0 || index >= offerte.size() || offerte.get(index).
    revisionata()) {
3             System.out.println("L'offerta selezionata non    valida");
4             System.out.println();
5         } else {
6             offerte.get(index).setAccettata(accetto);
7             offerte.get(index).setRevisionata();
8             azienda.revisionaOfferta(this, offerte.get(index).getCondominio(),
    accetto);
9         }
10    }
11
```

5.2 Azienda

Questa è la classe centrale del nostro progetto, infatti su di essa sono stati utilizzati i due Design Pattern discussi in precedenza.

La maggior parte dei metodi di questa classe sono atti a eseguire la comunicazione tra i vari Collegue. Per far ciò questa classe eredita i metodi e gli attributi dalla classe astratta Mediator.

Dato che la maggior parte della comunicazione del nostro sistema passa attraverso questa classe è tassativo che ne possa esistere solo un'istanza alla volta, se infatti ciò non dovesse accadere avremmo i problemi discussi in precedenza. Per questo sulla classe è stato utilizzato anche il pattern Singleton.

Metodi:

- *getIstanza*: serve a implementare il pattern Singleton, infatti viene utilizzato il costruttore della classe, il quale è stato posto private, solo nel caso in cui non esistesse già un istanza della classe.

```
1     public static Azienda getIstanza() {
2         if(istanza == null){
3             istanza = new Azienda("Azienda.srl");
4         }
5         return istanza;
6     }
7
```

- *cancellaIstanza*: serve a semplificare la fase di testing, infatti utilizzando il Singleton e non rimuovendo l'istanza si hanno risultati diversi a seconda se si eseguono i test uno alla volta o in blocco.

```
1      public static void cancellaIstanza(){
2         istanza = null;
3     }
4
```

- *assumiResponsabile*,*aggiungiTecnico* e *aggiungiAmministratore*: servono rispettivamente per assumere il responsabile il quale, se già presente, esso sarà rimpiazzato e ad aggiungere elementi all'ArrayList dei tecnici e a quello degli amministratori.

```
1     public void assumiResponsabile(Responsabile r) {
2         responsabile = r;
3     }
4
```

```
1     public void aggiungiTecnico(Tecnico t) {
2         tecnici.add(t);
3     }
4
```

```
    private void aggiungiAmministratore(Amministratore a) {
        amministratori.add(a);
    }

```

- *verificaSeEsiste*: è un metodo utilizzato dalla funzione generaRichiesta è serve a fare in modo che un Amministratore non possa richiedere l'efficientamento energetico su un condominio da lui amministrato più di una volta. Infatti questo metodo ritorna *true* se esiste già una richiesta non revisionata con i parametri uguali a quelli in ingresso, se invece non è presente ritorna *false*.

```
    private boolean verificaSeEsiste(Amministratore a, Condominio c) {
        for (int i = 0; i < richieste.size(); i++) {
            if (richieste.get(i).getAmministratore() == a && richieste.get(i).
    getCondominio() == c
                    && !richieste.get(i).revisionata()) {
                return true;
            }
        }
        return false;
    }

```

- *generaRichiesta*: crea una nuova richiesta di efficientamento energetico relativa a un'amministratore e ad un condominio. Questo metodo utilizza la funzione precedente per verificare se è già presente una richiesta, non revisionata, con i gli stessi parametri di quella che vogliamo creare. Questo è fatto per evitare che un'amministratore possa erroneamento, o anche volontariamente, creare più di una richiesta di efficientamento energetico per uno specifico condominio.

```
    @Override
    public void generaRichiesta(Condominio c, Amministratore a) {

        if (!amministratori.contains(a)) {
            aggiungiAmministratore(a);
            richieste.add(new Richiesta(a, c));
        } else {
            if (!verificaSeEsiste(a, c)) {
                richieste.add(new Richiesta(a, c));
            }
        }
    }

```

- *consultaRichieste*: questa prima implementazione del metodo serve all'amministratore a visualizzare tutte le richieste relative solamente ai condomini amministrati da esso, quindi a vedere se sono state accettate o rifiutate.

```
    @Override
    public void consultaRichieste(Amministratore a) {
        int num = 0;

        for (int i = 0; i < richieste.size(); i++) {
            if (richieste.get(i).getAmministratore() == a) {
                num++;
                System.out.println("Richiesta numero: " + i);
                richieste.get(i).stampaInfo();
                System.out.println();
            }
        }
        if (num == 0) {
            System.out.println("Non ci sono richieste effettuate");
            System.out.println();
        }
    }

```

- *consultaRichieste*: questa seconda implementazione del metodo ha un'implementazione molto simile alla precedente e serve all'amministratore a visualizzare le richieste relative ad uno specifico condominio.

```
    @Override
    public void consultaRichieste(Condominio c) {
        int num = 0;

        for (int i = 0; i < richieste.size(); i++) {
            if (richieste.get(i).getCondominio() == c) {
                num++;
                System.out.println("Richiesta numero: " + i);
                richieste.get(i).stampaInfo();
                System.out.println();
            }
        }
        if (num == 0) {
            System.out.println("Non ci sono richieste effettuate per questo
    condominio");
            System.out.println();
        }
    }

```

- *consultaRichieste*: questa terza implementazione del metodo è molto simile alle precedenti, ma ha una funzione completamente diversa, infatti essa è utilizzabile solo dal Responsabile, al quale verranno visualizzate tutte le richieste che non ha ancora accettato o rifiutato.

```
    @Override
    public void consultaRichieste() {

        boolean flag = true;

        for (int i = 0; i < richieste.size(); i++) {
            if (richieste.get(i).revisionata() == false) {
                flag = false;
                System.out.println("Richiesta numero: " + i);
                richieste.get(i).stampaInfo();
                System.out.println();
            }
        }
        if (flag) {
            System.out.println("Non ci sono richieste da revisionare");
            System.out.println();
        }
    }

```

- *consultaTutteRichieste*: anche questa implementazione è utilizzabile solo dal Responsabile al quale verrano visualizzate a schermo tutte le richieste fatte nell'arco di tutto l'utilizzo del sistema.

  Tutti questi metodi stampano le richieste come una lista numerata.

```
     @Override
    public void consultaTutteRichieste() {
        for (int i = 0; i < richieste.size(); i++) {
            System.out.println("Richiesta numero: " + i);
            richieste.get(i).stampaInfo();
            System.out.println();
        }
    }

```

- *consultaSopralluoghi*: è utilizzato dal Responsabile per visualizzare la lista dei sopralluoghi effettuati, con relativa data e Tecnico quale ha effettuato il controllo. Questo metodo filtra la lista completa a solo i sopralluoghi a cui non è associata un'offerta.

```
    @Override
    public void consultaSopralluoghi() {

        boolean flag = true;

        for (int i = 0; i < sopralluoghiEffettuati.size(); i++) {
            if (sopralluoghiEffettuati.get(i).offerta() == false) {
                flag = false;
                System.out.println("Sopralluogo numero: " + i);
                sopralluoghiEffettuati.get(i).stampaInfo();
                System.out.println();
            }
        }
        if (flag) {
            System.out.println("Non sono stati effettuati altri sopralluoghi");
            System.out.println();
        }
    }

```

- *consultaTuttiSopralluoghi*: è utilizzato dal Responsabile per visualizzare la lista di tutti i sopralluoghi effettuati, con relativa data e Tecnico quale ha effettuato il controllo.

```
    @Override
    public void consultaTuttiSopralluoghi() {
        for (int i = 0; i < sopralluoghiEffettuati.size(); i++) {
            System.out.println("Sopralluogo numero: " + i);
            sopralluoghiEffettuati.get(i).stampaInfo();
            System.out.println();
        }
    }

```

- *consultaOfferte*: serve a far visualizzare al Responsabile le offerte sotto forma di lista numerata.

```
    @Override
    public void consultaOfferte(){
        for (int i = 0; i < offerte.size(); i++) {
            System.out.println("Offerta numero: " + i);
            offerte.get(i).stampaInfo();
            System.out.println();
        }
    }

```

- *faiOfferta*: serve al Responsabile a creare un'offerta, obbligatoriamente corrispondente un sopralluogo precedentemente effettuato. Quest'offerta viene memorizzata dalla classe Azienda e viene inviata all'amministratore interessato che deciderà in seguito se accettarla o rifiutarla.

  Si utilizza sempre l'idea della selezione univoca con l'indice per rimanere coerenti con quanto detto prima relativamente all'intefaccia grafica.

```
1     @Override
2     public void faiOfferta(int index, int valoreOfferta) {
3         if (index < 0 || index >= sopralluoghiEffettuati.size()) {
4             System.out.println("Il sopralluogo selezionato non esiste");
5             System.out.println();
6         } else {
7             if (sopralluoghiEffettuati.get(index).offerta()) {
8                 System.out.println("E' gia' stata effettuata un offerta
    relativa a questo sopralluogo");
9             } else {
10                Offerta offerta = new Offerta(sopralluoghiEffettuati.get(index)
    .getCondominio(),
11                        sopralluoghiEffettuati.get(index).getCondominio().
    amministratore, valoreOfferta);
12                offerte.add(offerta);
13                sopralluoghiEffettuati.get(index).getCondominio().
    amministratore.aggiungiOfferta(offerta);
14                sopralluoghiEffettuati.get(index).setOfferta();
15            }
16        }
17    }
18
```

- *consultaImprese*: serve al Responsabile per visualizzare tutte le imprese edili presenti nel sistema tra cui può scegliere a quale affidare un determinato lavoro.

```
1     @Override
2     public void consultaImprese() {
3         for (int i = 0; i < imprese.size(); i++) {
4             System.out.println("Impresa numero: " + i);
5             imprese.get(i).stampaInfo();
6             System.out.println();
7         }
8     }
9
```

- *commissionaLavoro*: serve a commissionare un lavoro su un condominio, il quale deve corrispondere obbligatoriamente ad un offerta accettata da un'amministratore, a una delle imprese edili presenti nel sistema, le quali provvederanno a svolgere il vero e proprio efficientamento energetico.

```
    @Override
    public void commissionaLavoro(int offIndex, int impIndex) {
        if(offIndex >= 0 && offIndex < offerte.size()){
            if(offerte.get(offIndex).revisionata() && offerte.get(offIndex).
    accettata()){   //controllo se    revisionata e accettata
                if(impIndex >= 0 && impIndex < imprese.size()){

                    imprese.get(impIndex).aggiungiRichiesta(offerte.get(
    offIndex).getCondominio()); //qua fa effettivamente il lavoro

                } else {
                    System.out.println("L'impresa selezionata non esiste");
                    System.out.println();
                }
            } else {
                System.out.println("L'offerta selezionata non    valida");
                System.out.println();
            }
        } else {
            System.out.println("L'offerta selezionata non    valida");
            System.out.println();
        }
    }

```

- *inviaResponsoSopralluogo*: è utilizzato esclusivamente dai tecnici i quali inviano il responso di un sopralluogo da loro effettuato che viene memorizzato nell'azienda.

```
    @Override
    public void inviaResponsoSopralluogo(Sopralluogo s) {
        sopralluoghiEffettuati.add(s);
    }

```

5.3 Impresa

Questa classe rappresenta un'impresa edile che ha il compito di iniziare e dismettere cantieri per l'efficientamento energetico dei condomini richiesti dall'azienda. Questi cantieri sono creati dagli operai che lavorano per l'azienda, la quale può monitorare il loro stato, cioè se sono disponibili o se sono già impegnati in un lavoro, assumerne di nuovi o licenziare quegli che lavorano tutt'ora per lei.

Metodi:

- *stampaOperai*: serve all'Impresa per visualizzare tutti gli operai, che ha assunto, e visualizzarne lo stato che può essere "disponibile" o "non disponibile" a seconda se sono impegnati in un cantiere o no.

```
    public void stampaOperai() {

        for (int i = 0; i < operai.size(); i++) {
            System.out.println("Operaio numero " + i + ":");
            operai.get(i).stampaInfo();
            System.out.println();
        }
    }

```

- *stampaOperaiDisponibili*: rispetto al metodo precedente fornisce una lista di operai ridotta. Infatti il metodo filtra la lista e stampa a schermo solo quelli disponibili.

```
    public void stampaOperaiDisponibili() {
        boolean flag = true;
        for (int i = 0; i < operai.size(); i++) {
            if (operai.get(i).disponibile()) {
                System.out.println("Operaio numero " + i + ":");
                operai.get(i).stampaInfo();
                System.out.println();
                flag = false;
            }
        }

        if (flag) {
            System.out.println("Non ci sono Operai Disponibili");
            System.out.println();
        }
    }

```

- *assumiOperaio* e *licenziaOperaio*: sono i metodi atti al management degli operai, infatti questi ci danno la possibilità di assumere un nuovo operaio, o di licenziarlo scegliendolo sempre grazie all'indice che esso ha nell'ArrayList per rimanere coerenti con le scelte precedenti.

```
    public void assumiOperaio(Operaio o) {
        operai.add(o);
    }

```

```
   public void licenziaOperaio(int index) {
        if (index < 0 || index >= operai.size()) {
            System.out.println("L'operaio Selezionato non esiste");
        } else {
            operai.remove(index);
        }
    }

```

- *scegliOperai*: seleziona un numero di operai, da quelli disponibili, che saranno utilizzati poi per creare un cantiere su un determinato condominio. Questo metodo è posto privato perchè viene utilizzato dalla funzione iniziaLavoro che calcola il numero di operai necessari a svolgere un determinato lavoro e lo passa a questa come valore in ingresso.

```
    private ArrayList<Operaio> scegliOperai(int necessari) {
        ArrayList<Operaio> operaiScelti = new ArrayList<>();
        int scelti = 0;
        for (int i = 0; scelti < necessari; i++) {
            if (operai.get(i).disponibile()) {
                scelti++;
                operai.get(i).setDisponibile(false);
                operaiScelti.add(operai.get(i));
            }
        }
        return operaiScelti;
    }

```

- *aggiungiRichiesta*: è utilizzata dalla classe Azienda la quale, su richiesta del Responsabile, manda una richiesta di lavoro, specificata da un Condominio all'Impresa edile, la quale la aggiungerà all'ArrayList corrispondente.

```
    public void aggiungiRichiesta(Condominio c) {
        richieste.add(c);
    }

```

- *consultaRichieste*: serve a visualizzare tutte le richieste di lavoro fatte dall'Azienda.

  Questo come sempre viene fatto sotto forma di lista numerata per essere coerente con le scelte discusse in precedenza.

```
    public void consultaRichieste() {
        boolean flag = true;
        for (int i = 0; i < richieste.size(); i++) {
            System.out.println("Richiesta " + i + ":");
            richieste.get(i).stampaInfo();
            System.out.println();
            flag = false;
        }

        if (flag) {
            System.out.println("Non ci sono nuove richieste");
            System.out.println();
        }
    }

```

- *iniziaLavoro*: serve all'Impresa per aprire un cantiere su un determinato condominio, il quale è stato indicato dall'azienda con una richiesta di lavoro. Un cantiere deve corrispondere necessariamente ad una richiesta di lavoro.

  Il metodo in seguito calcola gli operai necessari per aprire un cantiere sul determinato condominio. Questo valore è poi passato alla funzione scegliOperai che ci restituirà gli Operai che andranno a formare il cantiere. Dopo di ciò gli Operai saranno posti come "non disponibili" e verrà creato il vero e proprio cantiere.

```
    public void iniziaLavoro(int index) {
        if (index < 0 || index >= richieste.size()) {
            System.out.println("La richiesta selezionata non esiste");
            System.out.println();
        } else {
            if (richieste.get(index).getTotaleUnita() / 3 >
    getNumOperaiDisponibili()) {
                System.out.println("Non ci sono abbastanza operai per iniziare
    un cantiere");
                System.out.println();
            } else {
                int necessari = (int) richieste.get(index).getTotaleUnita() /
    3;
                ArrayList<Operaio> operai = scegliOperai(necessari);
                Cantiere cantiere = new Cantiere(operai, richieste.get(index),
    new Data());
                cantieri.add(cantiere);
            }
```

```
        }
    }

```

- *stampaCantieri*: stampa tutti i cantieri, sia quelli attualmente attivi, che quelli già dismessi, con annesse date di inizio lavoro, e se presente quella di terminazione lavoro.

```
    public void stampaCantieri() {
        for (int i = 0; i < cantieri.size(); i++) {
            System.out.println("Cantiere " + i + ":");
            cantieri.get(i).stampaInfo();
            System.out.println();
        }
    }

```

- *stampaCantieriAttivi*: a differenza del metodo precedente questo crea una lista contenente solamente i cantieri attualmente attivi.

```
    public void stampaCantieriAttivi() {
        boolean flag = true;
        for (int i = 0; i < cantieri.size(); i++) {
            if (cantieri.get(i).attivo()) {
                System.out.println("Cantiere " + i + ":");
                cantieri.get(i).stampaInfo();
                System.out.println();
                flag = false;
            }
        }
        if (flag) {
            System.out.println("Non ci sono cantieri attivi");
            System.out.println();
        }
    }

```

- *terminaCantiere*: serve a terminare un cantiere attualmente attivo, i quale verrà aggiornato con la data di terminazione attuale. In seguito alla terminazione gli Operai coinvolti nel lavoro verranno riportati nello stato "disponibile" e potranno essere utilizzati per un lavoro successivo.

  Questo metodo utilizza la funzione *equals* degli ArrayList che ci permette di vedere se un'oggetto è uguale ad un'altro. Ciò ci permette di non far andare in crash il programma quando si termina un cantiere in cui è presente un Operaio che è stato licenziato durante lo stato "attivo" del cantiere.

```
1     public void terminaCantiere(int index) {
2         if (index < 0 || index >= cantieri.size()) {
3             System.out.println("Il cantieri selezionato non esiste");
4             System.out.println();
5         } else {
6             if (!cantieri.get(index).attivo()) {
7                 System.out.println("Il cantieri selezionato non    in corso");
8                 System.out.println();
9             } else {
10                cantieri.get(index).terminaCantiere();
11                ArrayList<Operaio> operaiCoinvolti = cantieri.get(index).
    getOperaiCoinvolti();
12
13                for (int i = 0; i < operaiCoinvolti.size(); i++) {
14                    for (int j = 0; j < operai.size(); j++) {
15                        if (operai.get(j).equals(operaiCoinvolti.get(i))) {
16                            operai.get(j).setDisponibile(true);
17                        }
18                    }
19                }
20
21            }
22        }
23
```

5.4 Tecnico

Questa classe rappresenta un Tecnico che deve poter: riceve richieste di Sopralluogo dal Responsabile tramite la classe Azienda, consultare queste richieste ed effettuare i Sopralluoghi. A seguito di questo poi deve inviare il responso all'Azienda in modo che possa essere consultabile dal Responsabile.

Metodi:

- *aggiungiRichiestaSopralluogo*: serve alla classe Azienda per aggiungere una richiesta di sopralluogo, la quale è specificata attraverso il condominio sul quale si deve effettuare l'operazione.

  Questo poi verrà aggiunto al determinato ArrayList che contiene tutte le richieste.

```
1     public void aggiungiRichiestaSopralluogo(Condominio c) {
2         richiesteSopralluogo.add(c);
3     }
4
```

- *consultaRichieste*: serve al Tecnico a visualizzare le richieste arrivate dall'Azienda.

  Questo viene sempre fatto attraverso una lista numerata in modo che si rimanga coerenti con le scelte fatte in precedenza.

```
1     public void consultaRichieste() {
2         if (richiesteSopralluogo.size() == 0) {
3             System.out.println("Non ci sono richieste di Sopralluogo");
4             System.out.println();
5         } else {
6             for (int i = 0; i < richiesteSopralluogo.size(); i++) {
7                 System.out.println("Richiesta " + i + ":");
8                 richiesteSopralluogo.get(i).stampaInfo();
9                 System.out.println();
10            }
11        }
12    }
13
```

- *faiSopralluogo*: crea un sopralluogo il quale viene inviato alla classe Azienda utilizzando il metodo della stessa *inviaResponsoSoprelluogo*. Inoltre viene simulata l'importazione delle foto eseguite sul campo con il metodo *importaFoto*.

```
    public void faiSopralluogo(int index){
        if (index >= richiesteSopralluogo.size() || index < 0){
            System.out.println("La richiesta selezionata non    valida");
            System.out.println();
        } else {
            int numFoto = (int) Math.random() * 10 + 10;
            ArrayList<Foto> foto = importaFoto(numFoto);
            Sopralluogo sopralluogo = new Sopralluogo(this,
   richiesteSopralluogo.get(index), new Data(), foto);
            azienda.inviaResponsoSopralluogo(sopralluogo);
            richiesteSopralluogo.remove(index);
        }
    }

```

- *importaFoto*: è utilizzato dal metodo precedente per creare un'ArrayList di foto, il cui numero è scelto attraverso un parametro in ingresso.

```
    private ArrayList<Foto> importaFoto(int numFoto){
        ArrayList<Foto> foto = new ArrayList<>();
        for(int i=0; i<numFoto; i++){
            foto.add(new Foto(new Data()));
        }
        return foto;
    }

```

6 Testing

Per il testing, effettuato con JUnit, abbiamo deciso di testare il funzionamento dei metodi principali delle classi *Azienda* e *Impresa* essendo le più complesse e più critiche dal punto di vista del funzionamento corretto del sistema.

6.1 Testing su Azienda

Per la classe *Azienda* abbiamo dovuto usare una speciale accortezza durante il testing. Infatti su essa è stato utilizzato il Design Pattern Singleton, che come discusso in precedenza genera dei problemi in fase di testing. Come possiamo vedere alla fine di ogni unità di test viene utilizzato il metodo *cancellaIstanza* della classe Azienda, questo è fatto per evitare che si abbiano risultati diversi quando si eseguono i test singolarmente o in cascata. Infatti quando andiamo a eseguire i test in cascata l'istanza di azienda viene mantenuta tra un'unità di test e l'altra facendo fallire alcuni test che risulterebbero passati se eseguiti singolarmente.

- Testing del metodo *assumiResponsabile*: si assume il Responsabile dalla classe Azienda e si confronta quello che risulta dalla classe con quello l'oggetto che è stato inserito.

```
1     @Test
2     public void testAssumiResponsabile() {
3         Azienda a = Azienda.getIstanza();
4         Responsabile r = new Responsabile("Test", "Test", a);
5
6         a.assumiResponsabile(r);
7         assertEquals(r, a.getResponsabile());
8
9         a.cancellaIstanza();
10    }
11
```

- Testing del metodo *aggiungiTecnico*: Si inseriscono due tecnici e come nel test precedente si verifica che i risultati siano corretti.

```
1     @Test
2     public void testAggiungiTecnico() {
3         Azienda a = Azienda.getIstanza();
4         Tecnico t1 = new Tecnico("Test", "Test", a);
5         Tecnico t2 = new Tecnico("Test", "Test", a);
6
7         a.aggiungiTecnico(t1);
8         a.aggiungiTecnico(t2);
9         ArrayList<Tecnico> expected = new ArrayList<>();
10        expected.add(t1);
11        expected.add(t2);
```

```
        ArrayList<Tecnico> actual = a.getTecnici();
        for (int i = 0; i < actual.size(); i++) {
            assertEquals(expected.get(i), actual.get(i));
        }

        a.cancellaIstanza();
    }

```

- Testing del metodo *generaRichiesta*: si genere una richiesta senza passare dalla classe Amministratore e poi si crea la richiesta manualemente, in seguito si confrontano i due risultati.

```
    @Test
    public void testGeneraRichiesta() {
        Azienda a = Azienda.getIstanza();
        Amministratore amm = new Amministratore("Test", "Test", a);
        Condominio c = new Condominio("Test", 10, 9, 2, 2, amm);

        a.generaRichiesta(c, amm);
        ArrayList<Richiesta> richieste = a.getRichieste();
        assertEquals(richieste.get(0).getAmministratore(), amm);
        assertEquals(c, richieste.get(0).getCondominio());

        a.cancellaIstanza();
    }

```

- Testing del metodo *revisionaOfferta*: si simula uno scambio si Richieste e Offerte tra il Responsabile e l'Amministratore il quale decidarà di accettare o rifiutare l'offerta. Questo risultato verrà comparato con quanto presente nella classe Azienda per vedere se si effettua il corretto aggiornamento su entrambe le classi dei valori.

```
    @Test
    public void testRevisionaOfferta() {
        Azienda a = Azienda.getIstanza();
        Responsabile r = new Responsabile("Test", "Test", a);
        a.assumiResponsabile(r);
        Tecnico t1 = new Tecnico("Test", "Test", a);
        a.aggiungiTecnico(t1);

        Amministratore amm = new Amministratore("Test", "Test", a);
        Condominio c = new Condominio("Test", 10, 9, 2, 2, amm);

        amm.aggiungiCondominio(c);
        amm.faiRichiesta(0);

        r.revisionaRichiesta(0, true);
```

```
16
17          t1.faiSopralluogo(0);
18
19          r.faiOfferta(0, 1000);
20
21          amm.revisionaOfferta(0, true);
22
23          ArrayList<Offerta> offerte = a.getOfferte();
24          assertEquals(true, offerte.get(0).revisionata());
25          assertEquals(true, offerte.get(0).accettata());
26          assertEquals(amm, offerte.get(0).getAmministratore());
27          assertEquals(c, offerte.get(0).getCondominio());
28
29          a.cancellaIstanza();
30      }
31
```

- Testing del metodo *revisionaRichiesta*: dopo che l'Amministratore ha generato una richiesta il Responsabile decide se accettarla o no, poi si confronta il valore di verità con quello che è presente nella classe richiesta.

```
1       @Test
2       public void testRevisionaRichiesta() {
3           Azienda a = Azienda.getIstanza();
4           Responsabile r = new Responsabile("Test", "Test", a);
5           a.assumiResponsabile(r);
6           Tecnico t1 = new Tecnico("Test", "Test", a);
7           a.aggiungiTecnico(t1);
8
9           Amministratore amm = new Amministratore("Test", "Test", a);
10          Condominio c = new Condominio("Test", 10, 9, 2, 2, amm);
11
12          amm.aggiungiCondominio(c);
13          amm.faiRichiesta(0);
14
15          r.revisionaRichiesta(0, true);
16
17          ArrayList<Richiesta> richieste = a.getRichieste();
18          assertEquals(true, richieste.get(0).revisionata());
19          assertEquals(true, richieste.get(0).accettata());
20          assertEquals(amm, richieste.get(0).getAmministratore());
21          assertEquals(c, richieste.get(0).getCondominio());
22
23          a.cancellaIstanza();
24      }
25
```

- Testing del metodo *faiOfferta*: dopo uno scambio di Richieste e Sopralluoghi, il Responsabile genera un Offerta con un valore che corrisponde a quello monetario del preventivo. In seguito si controlla se i parametri della classe offerta sono coerenti con quelli attesi.

```
1     @Test
2     public void testFaiOfferta() {
3         Azienda a = Azienda.getIstanza();
4         Responsabile r = new Responsabile("Test", "Test", a);
5         a.assumiResponsabile(r);
6         Tecnico t1 = new Tecnico("Test", "Test", a);
7         a.aggiungiTecnico(t1);
8
9         Amministratore amm = new Amministratore("Test", "Test", a);
10        Condominio c = new Condominio("Test", 10, 9, 2, 2, amm);
11
12        amm.aggiungiCondominio(c);
13        amm.faiRichiesta(0);
14
15        r.revisionaRichiesta(0, true);
16
17        t1.faiSopralluogo(0);
18
19        r.faiOfferta(0, 1000);
20        ArrayList<Offerta> offerte = a.getOfferte();
21        assertEquals(false, offerte.get(0).revisionata());
22        assertEquals(false, offerte.get(0).accettata());
23        assertEquals(amm, offerte.get(0).getAmministratore());
24        assertEquals(c, offerte.get(0).getCondominio());
25        assertEquals(1000, offerte.get(0).getOfferta());
26
27        a.cancellaIstanza();
28    }
29
```

- Testing del metodo *inviaResponsoSopralluogo*: a seguito di una Richiesta da parte dell'Amministratore di condominio, accettata dal Responsabile, il Tecnico riceverà una richiesta di sopralluogo. Il Tecnico poi farà l'operazione e invierà il responso all'Azienda, nella quale si controllerà che tutti i valori deglia attributi della classe Sopralluogo siano congrui con quelli attesi.

```
1     @Test
2     public void testInviaResponsoSopralluogo() {
3         Azienda a = Azienda.getIstanza();
4         Responsabile r = new Responsabile("Test", "Test", a);
5         a.assumiResponsabile(r);
6         Tecnico t1 = new Tecnico("Test", "Test", a);
7         a.aggiungiTecnico(t1);
8
9         Amministratore amm = new Amministratore("Test", "Test", a);
10        Condominio c = new Condominio("Test", 10, 9, 2, 2, amm);
11
12        amm.aggiungiCondominio(c);
13        amm.faiRichiesta(0);
14
15        r.revisionaRichiesta(0, true);
16
17        t1.faiSopralluogo(0);
18
19        ArrayList<Sopralluogo> sopralluoghi = a.getSopralluoghi();
20        assertEquals(t1, sopralluoghi.get(0).getTecnico());
21        assertEquals(c, sopralluoghi.get(0).getCondominio());
22
23        a.cancellaIstanza();
24    }
25
```

6.2 Test su Impresa

Nella classe *Impresa* effettuare i test risulta più facile e immediato perchè non dobbiamo risolvere i problemi causati dal Pattern Singleton.

- Testing del metodo *assumiOperaio*: l'impresa utilizza il metodo e poi si confronta il valore come fatto in precedenza per la classe Azienda.

```
    @Test
    public void testAssumiOperaio() {
        Impresa imp = new Impresa("Test.spa");
        Operaio o1 = new Operaio("Test", "Test");

        imp.assumiOperaio(o1);
        ArrayList<Operaio> operai = imp.getOperai();
        assertEquals(operai.get(0), o1);
    }

```

- Testing del metodo *getNumOperaiDisponibili*: dopo che l'Impresa ha assunto 3 diversi Operai si compara il risultato ottenuto dal metodo con quello atteso.

```
    @Test
    public void testGetNumOperaiDisponibili() {
        Impresa imp = new Impresa("Test.spa");
        Operaio o1 = new Operaio("Test1", "Test");
        Operaio o2 = new Operaio("Test2", "Test");
        Operaio o3 = new Operaio("Test3", "Test");

        imp.assumiOperaio(o1);
        imp.assumiOperaio(o2);
        imp.assumiOperaio(o3);

        int actual = imp.getNumOperaiDisponibili();
        assertEquals(actual, 3);
    }

```

- Testing del metodo *aggiungiRichiesta*: generiamo una richiesta direttamente utilizzando il metodo della classe, e vediamo se il risultato è consistente con quello atteso.

```
    @Test
    public void testAggiungiRichiesta() {
        Impresa imp = new Impresa("Test.spa");
        Operaio o1 = new Operaio("Test1", "Test");
        Operaio o2 = new Operaio("Test2", "Test");
        Operaio o3 = new Operaio("Test3", "Test");

```

```
        imp.assumiOperaio(o1);
        imp.assumiOperaio(o2);
        imp.assumiOperaio(o3);

        Condominio c = new Condominio("Test", 10, 9, 2, 2);
        imp.aggiungiRichiesta(c);
        ArrayList<Condominio> richieste = imp.getRichieste();
        assertEquals(c, richieste.get(0));
    }

```

- Testing del metodo *iniziaLavoro*: dopo aver iniziato un cantiere che impegna tutti e 3 gli Operai che lavorano per l'Impresa si testa se essi sono stati correttamente assegnati a quel cantiere e se è stato correttamente aggiornato il loro stato.

```
    @Test
    public void testIniziaLavoro() {
        Impresa imp = new Impresa("Test.spa");
        Operaio o1 = new Operaio("Test1", "Test");
        Operaio o2 = new Operaio("Test2", "Test");
        Operaio o3 = new Operaio("Test3", "Test");

        imp.assumiOperaio(o1);
        imp.assumiOperaio(o2);
        imp.assumiOperaio(o3);

        Condominio c = new Condominio("Test", 10, 9, 2, 2);
        imp.aggiungiRichiesta(c);
        imp.iniziaLavoro(0);

        ArrayList<Cantiere> cantieri = imp.getCantieri();
        assertEquals(o1, cantieri.get(0).getOperaiCoinvolti().get(0));
        assertEquals(o2, cantieri.get(0).getOperaiCoinvolti().get(1));
        assertEquals(o3, cantieri.get(0).getOperaiCoinvolti().get(2));
        assertEquals(true, cantieri.get(0).attivo());
        assertEquals(false, o1.disponibile());
        assertEquals(false, o2.disponibile());
        assertEquals(false, o3.disponibile());
    }

```

- Testing del metodo *terminaCantiere*: dopo la creazione e la terminazione di un determinato cantiere si guarda se è stato correttamente impostato il valore dell'attributo attivo della classe Cantiere e su gli operai coinvolti sono tutti nello stato "disponibile".

```
1     @Test
2     public void testTerminaCantiere() {
3         Impresa imp = new Impresa("Test.spa");
4         Operaio o1 = new Operaio("Test1", "Test");
5         Operaio o2 = new Operaio("Test2", "Test");
6         Operaio o3 = new Operaio("Test3", "Test");
7
8         imp.assumiOperaio(o1);
9         imp.assumiOperaio(o2);
10        imp.assumiOperaio(o3);
11
12        Condominio c = new Condominio("Test", 10, 9, 2, 2);
13        imp.aggiungiRichiesta(c);
14        imp.iniziaLavoro(0);
15        imp.terminaCantiere(0);
16
17        ArrayList<Cantiere> cantieri = imp.getCantieri();
18        assertEquals(false, cantieri.get(0).attivo());
19        assertEquals(true, o1.disponibile());
20        assertEquals(true, o2.disponibile());
21        assertEquals(true, o3.disponibile());
22    }
23
```

7 UseExample

La cartella UseExample contiene classi per simulare l'utilizzo dell'applicazione e verificare la correttezza dei risultati. Differentemente dal Testing non si testa un singolo metodo di una classe bensì il funzionamento del sistema nella sua interezza. Si compone di tre classi:

- *Example1*: In questa classe abbiamo simulato il caso in cui un Amministratore di condominio effettua due richieste su due condomini distinti, le richieste vengono poi revisionate e accettate dal Responsabile. Il Tecnico effettua quindi il sopralluogo su uno dei due condomini e invia quindi il responso del sopralluogo effettuato.

  Come possiamo vedere infatti nella classe Azienda è presente un'istanza della classe Sopralluogo a cui è associato il Tecnico che ha effettuato l'operazione.

- *Example2*: Abbiamo voluto testare il funzionamento dell'Impresa. In questa classe un'Impresa, dopo aver assunto tre operai, riceve due richieste per effettuare il lavoro. La prima richiesta viene accettata e il lavoro viene iniziato mentre la seconda richiesta, non avendo più operai disponibili, viene quindi rifiutata e viene ricevuto il messaggio di errore.

  Infatti nella classe Impresa è presente solo un'istanza della classe Cantiere corrispondente alla prima richiesta di lavoro. Mentre quando si prova a iniziare un secondo cantiere ci viene notificato dal sistema la mancanza del numero necessario di Operai disponibili per effettuare il lavoro.

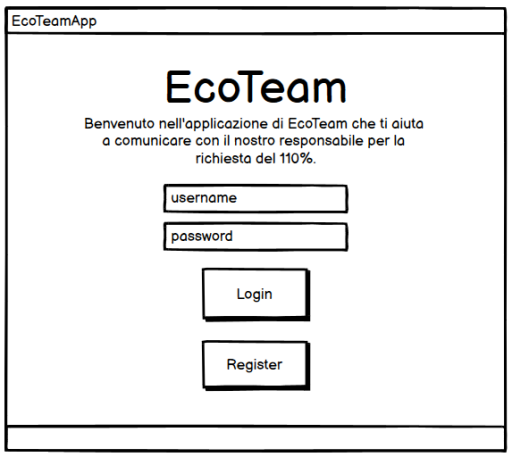
- *Example3*: In questo caso abbiamo testato il funzionamento nel caso in cui un Amministratore effettua richiesta per due condomini, le richieste vengono ricevute dal Responsabile che crea e invia le rispettive Offerte. A questo punto l'Amministratore valuta le due offerte e accetta solo una delle due. Il Responsabile può in seguito verificare che le offerte inviate sono state visionate e valutate dall'Amministratore e che solamente una è stata accettata.

  Come possiamo vedere infatti nella classe Azienda sono presenti entrambe le richieste, una accettata e l'altra no, le quali possono essere visualizzate dal Responsabile.

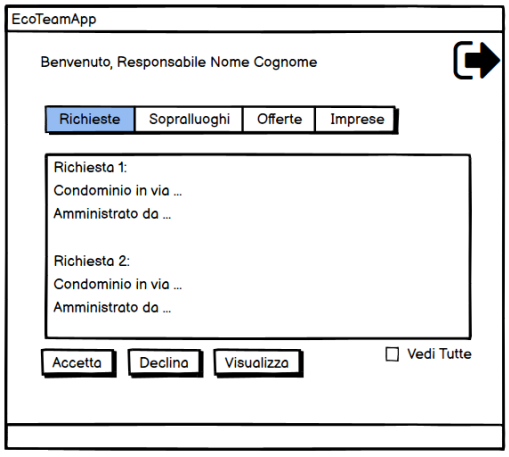
8 Possibile rappresentazione

8.1 Mock-Up

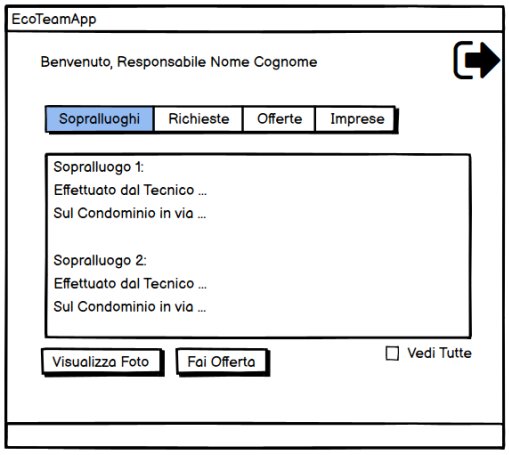
Come abbiamo discusso in precedenza alcune scelte sono state pensate per un'interfaccia grafica che presenta liste numerate con cui si può interagire selezionando univocamente un singolo elemento. Queste appunto sono le interfacce grafiche con cui abbiamo pensato che la nostra applicazione venisse sviluppata:



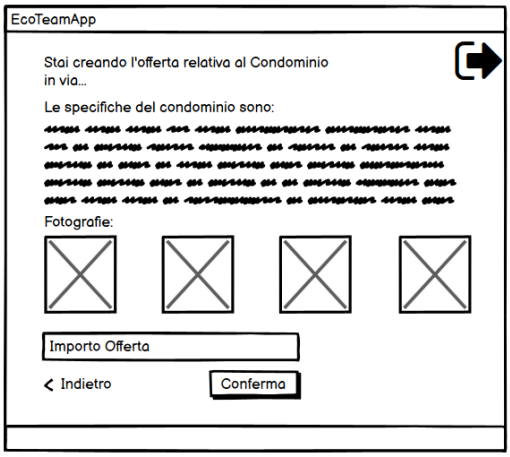
(a) Schermata iniziale applicazione



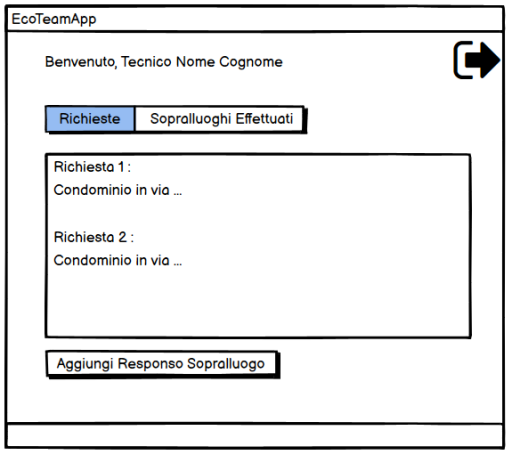
(b) Schermata Responsabile



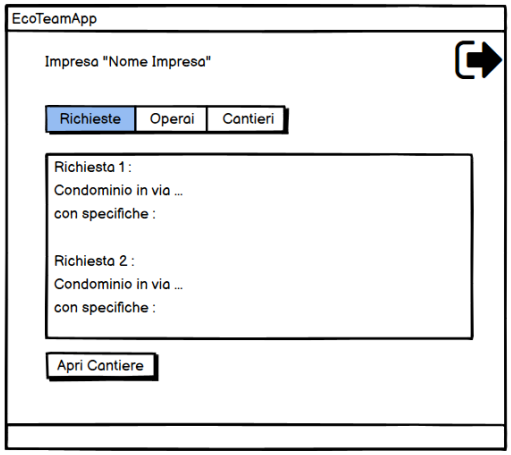
(c) Schermata Responsabile



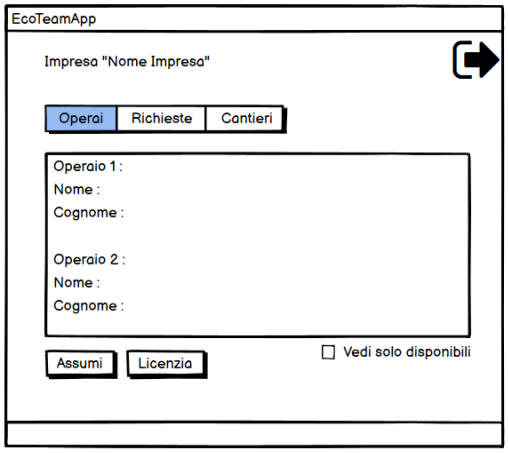
(d) Schermata Responsabile



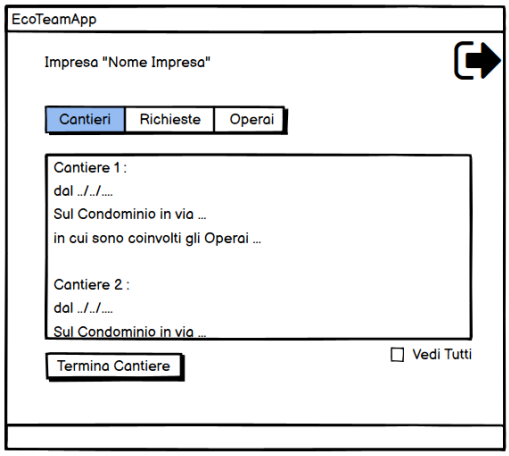
(a) Schermata Tecnico



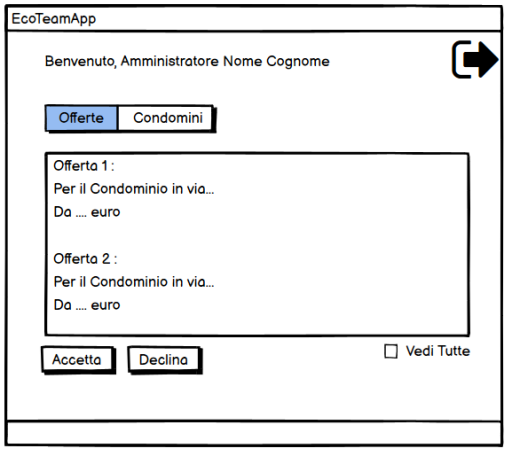
(b) Schermata Impresa



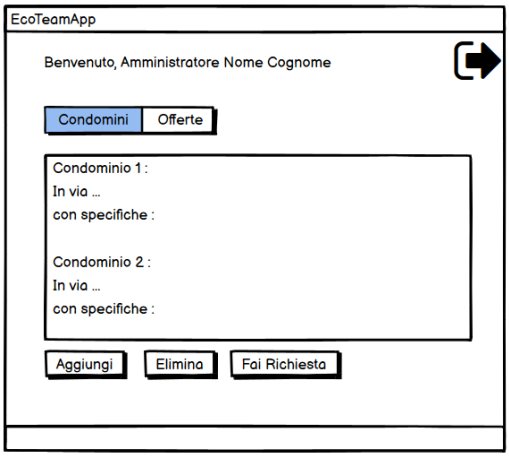
(c) Schermata Impresa



(d) Schermata Impresa



(a) Schermata Amministratore



(b) Schermata Amministratore