



UNIVERSITÀ DEGLI STUDI DI FIRENZE
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Relazione Gestionale Affitti

Autori:

Marco Vignozzi, Marco De Stefano

Corso principale:

Ingegneria del Software

N° Matricola:

7025991

6014827

Docente corso:

Enrico Vicario

Indice

1 Analisi	2
1.1 <i>Statement</i>	2
1.2 <i>Use Case Diagram</i>	3
1.3 <i>Use Case Template</i>	4
2 Architettura	7
2.1 Interazione tra i moduli	7
2.2 Interfaccia utente	7
2.3 Memorizzazione dei dati	8
2.3.1 Schema <i>Entity Relationship</i>	8
2.3.2 Schema logico e tecnologie utilizzate	9
3 Progettazione	11
3.1 <i>Page Navigation Diagram</i>	11
3.2 <i>Mockup</i>	13
3.3 <i>Class Diagram</i>	15
4 Implementazione	16
4.1 <i>Packages</i>	16
4.1.1 <i>Packages</i> personalizzati	16
4.1.2 <i>Java Standard Libraries</i>	17
4.2 Classi	18
4.2.1 <i>model</i>	18
4.2.2 <i>table</i>	19
4.2.3 <i>menu</i>	20
4.2.4 <i>controller</i>	20
4.2.5 <i>dao</i>	21
4.2.6 <i>mail_service</i>	21
4.3 Dettagli di progettazione	23
4.3.1 <i>Model View Controller</i>	23
4.3.2 <i>Facade</i>	24
4.3.3 <i>Builder</i>	24
4.3.4 <i>Data Access Object</i>	24
5 Unit Testing	25

1 Analisi

In questa Sezione verrà analizzato il programma Gestionale Affitti da noi sviluppato come progetto per l'esame di Ingegneria del Software. Inizieremo con l'illustrare gli obiettivi dell'applicazione, dando una visione d'insieme sulle principali funzionalità offerte.

1.1 *Statement*

Questo programma è stato pensato per essere utilizzato da proprietari di immobili(**utenti**) che vogliono gestire gli affitti e i contratti delle loro proprietà attraverso una semplice interfaccia, oltre a mantenerne uno storico. Il sistema utilizza un *database* per memorizzare e gestire separatamente i dati di ciascun utente. Sono disponibili i seguenti servizi:

- Aggiunta/modifica/eliminazione contratti
- Aggiunta/modifica/eliminazione inquilini
- Aggiunta/modifica/eliminazione immobili
- Visualizzazione immobili/inquilini/contratti/resoconto
- Gestione della situazione finanziaria relativa a ciascun inquilino

Gli immobili, gli inquilini e i contratti sono organizzati in tabelle che possono essere visualizzate graficamente dall'utente. E' inoltre possibile visualizzare una tabella di resoconto che mostra i dati più importanti degli immobili e degli inquilini e contratti ad essi associati, se presenti.

L'utente può aggiungere dei versamenti o delle spese a carico degli inquilini per poi visualizzare, attraverso le tabelle degli inquilini e di resoconto, il debito di ciascuno di essi nei confronti dell'utente.

Il proprietario può inviare solleciti di pagamento automaticamente a tutti gli inquilini che hanno un debito, o a singoli inquilini scelti dall'utente.

L'utente può registrarsi esclusivamente come proprietario, dando le sue credenziali, i suoi nominativi e i recapiti.

1.2 Use Case Diagram



Figura 1: Use Case Diagram

In Figura 1 è riportato il diagramma dei casi d'uso individuati sulla base dello *statement*.

Lo stereotipo <<include>> viene utilizzato qui per descrivere la situazione in cui un caso d'uso ne comprende un altro. Ad esempio quando viene utilizzato sulla connessione tra "Aggiungere Contratto" e "Aggiungere Inquilino", si intende che l'aggiunta di un contratto comprende l'aggiunta dell'inquilino ad esso associato.

Con l'uso dello stereotipo <<extend>> si intende che il caso d'uso da cui parte l'associazione è un'estensione dell'altro e viene eseguito in circostanze speciali. Ad esempio il caso "Aggiungere Immobile" è una possibile diramazione dell'operazione di aggiunta di un contratto e viene eseguito nel caso non sia già presente l'immobile a cui si vuole associare il contratto.

Lo stereotipo <<precede>> indica invece che deve essere terminata l'operazione da cui parte l'associazione prima che possa essere eseguita l'altra. Nel caso in Figura 1 per eseguire "Aggiungere Contratto" deve prima essere stato aggiunto l'inquilino a cui si riferisce.

Nel diagramma viene specificato anche il livello di astrazione <<Summary>>, il quale indica che il caso d'uso cui si riferisce fornisce una visione d'insieme dei casi che include.

1.3 *Use Case Template*

In seguito sono mostrati degli Use Case Template che mostrano il flusso di esecuzione di alcuni casi d'uso rappresentati in Figura 1. Sono stati scelti casi considerati più importanti o rappresentativi.

Use Case	Aggiungi Immobile
Scope	Azione riservata all'utente
Level	User Goal
Actor	Utente
Normal Flow	<ol style="list-style-type: none"> 1. L'utente si trova nel Menu Utenti e entra nel Menu Immobili. 2. L'utente sceglie l'operazione "Aggiungi immobile". 3. L'utente inserisce le informazioni principali della proprietà. 4. Il software chiede all'utente se desidera inserire i dati catastali. 5. L'utente scegli di inserirli e li inserisce. 6. L'utente conferma i dati inseriti. 7. Il software aggiunge l'immobile al database e l'utente torna al Menu Immobili.
Alternative Flow	<ol style="list-style-type: none"> 4_A. L'utente inserisce valori non validi e sceglie se riprovare. 5_A. L'utente sceglie di non inserire i dati catastali. L'operazione prosegue riprendendo dal punto 6. 6_A. L'utente rifiuta i dati inseriti e sceglie se riprovare. <ol style="list-style-type: none"> 6_{A1}. L'utente rifiuta di riprovare. 6_{A2}. L'utente torna al Menu Immobili annullando l'operazione.

Tabella 1: Aggiungi Immobile

Use Case	Aggiungi Inquilino
Scope	Azione riservata all'utente
Level	User Goal
Actor	Utente
Normal Flow	<ol style="list-style-type: none"> 1. L'utente si trova nel Menu Utenti ed entra nel Menu Inquilini (vedi Figura 13). 2. L'utente sceglie l'operazione "Aggiungi inquilino". 3. L'utente inserisce i dati inerenti l'inquilino. 4. L'utente conferma i dati inseriti e il software mostra la tabella immobili. 5. L'utente seleziona l'ID dell'immobile a cui associare il contratto. 6. L'immobile esiste e non è affittato, l'utente inserisce i dati del contratto. 7. L'utente conferma i dati inseriti 8. Il software aggiunge l'inquilino e il contratto al database e l'utente torna al Menu Inquilini.
Alternative Flow	<p>4_A. L'utente rifiuta i dati inseriti.</p> <p>4_{A1}. L'operazione viene annullata e l'utente torna al Menu Inquilini.</p> <p>6_A. L'ID scelto non è presente nel database o l'immobile corrispondente è già affittato e il software chiede all'utente se vuole crearne uno nuovo.</p> <p>6_{A1}. L'utente decide se creare un nuovo immobile, in caso contrario viene annullata l'operazione e l'utente torna al Menu Inquilini.</p> <p>7_A. L'utente rifiuta i dati inseriti.</p> <p>7_{A1}. L'operazione viene annullata e l'utente torna al Menu Inquilini.</p>

Tabella 2: Aggiungi Inquilino

Use Case	Registrazione
Scope	Azione riservata all'utente (proprietario)
Level	User Goal
Actors	Utente
Normal flow	<ol style="list-style-type: none"> 1. L'utente si trova nel Menu Benvenuto (vedi Figura 10) e sceglie l'opzione "Registrati". 2. L'utente entra nel Menu Registrazione (vedi Figura 12) e inserisce la sua email. 3. L'email non è già associata a un account e l'utente inserisce gli altri dati. 4. L'utente conferma i dati inseriti 5. Il nuovo proprietario creato viene aggiunto al database e l'utente torna al Menu Benvenuto.
Alternative flows	<p>3_A. L'email inserita è già associata a un account, l'utente sceglie se riprovare.</p> <p>3_{A1}. L'utente rifiuta di riprovare.</p> <p>3_{A2}. L'utente torna al Menu Benvenuto annullando l'operazione.</p> <p>4_A. L'utente rifiuta i dati inseriti.</p> <p>4_{A1}. L'utente torna al Menu Benvenuto annullando l'operazione.</p>

Tabella 3: Registrazione

Use Case	Invia Sollecito a tutti i debitori
Scope	Azione riservata all'utente
Level	User Goal
Actor	Utente
Normal Flow	<ol style="list-style-type: none"> 1. L'utente si trova nel Menu Utente. 2. L'utente sceglie l'operazione "Invia sollecito". 3. L'utente sceglie l'opzione "Sollecito automatico". 4. L'utente conferma l'operazione. 5. Il software invia un'email predefinita di sollecito a tutti gli inquilini che hanno un debito con l'utente.
Alternate Flow	

Tabella 4: Invia Sollecito

2 Architettura

In questa Sezione viene discussa la struttura organizzativa del sistema *software*.

2.1 Interazione tra i moduli

Nel diagramma in Figura 2 sono mostrate graficamente le dipendenze tra i *packages* che compongono il progetto. Verranno forniti maggiori dettagli in Sezione 4.1.

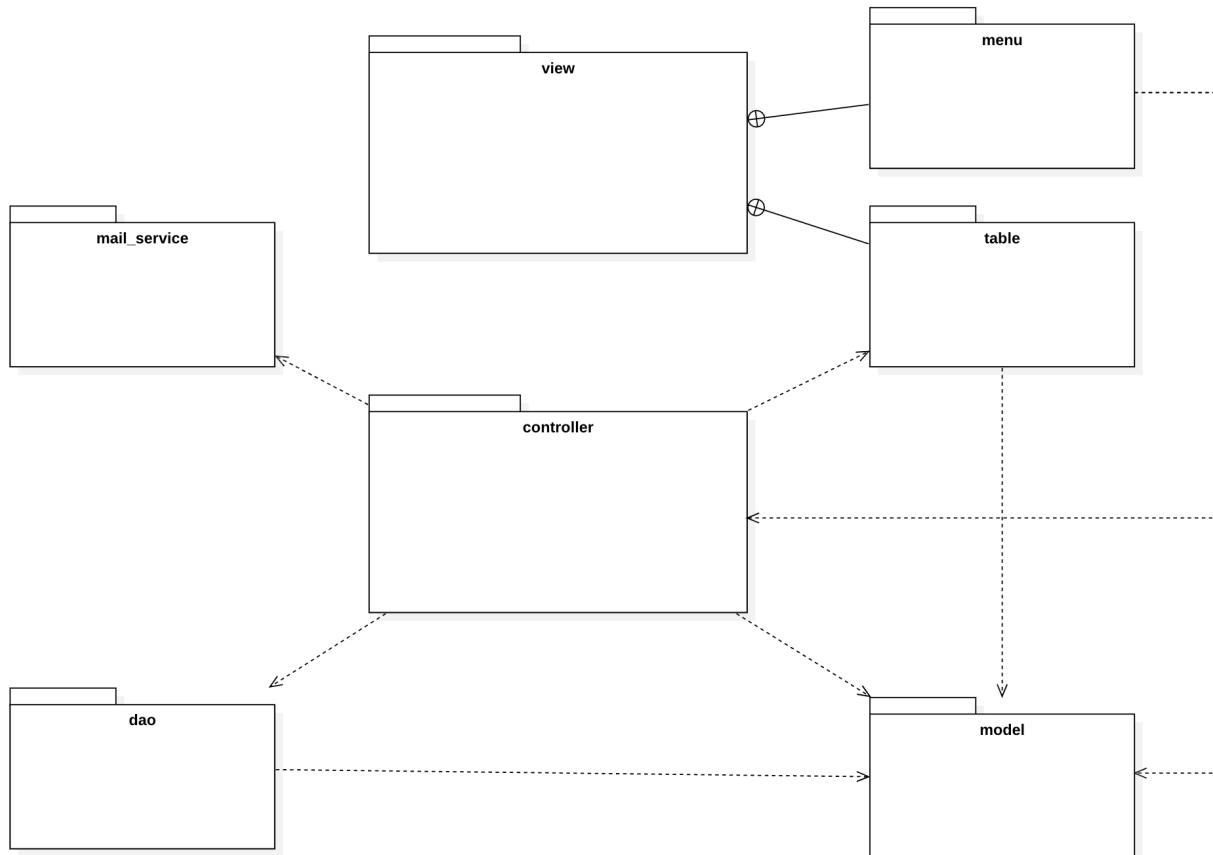


Figura 2: Diagramma rappresentante le dipendenze tra i *packages*

2.2 Interfaccia utente

L'utente interagisce con il programma prevalentemente attraverso la *Command-Line Interface* (CLI). In questo modo l'utente può navigare tra i vari menu presenti e scegliere le operazioni che desidera eseguire. Il programma offre anche la possibilità di visualizzare dati con delle tabelle attraverso una *Graphic User Interface* (GUI). Questo scopo viene raggiunto grazie alle librerie Java Swing. L'interazione con queste tabelle è però limitata alla visualizzazione: non è infatti possibile eseguire operazioni attraverso questa interfaccia.

2.3 Memorizzazione dei dati

In questa sezione viene descritta l'architettura del *database* di cui fa uso il programma e le tecnologie utilizzate per realizzarlo.

2.3.1 Schema *Entity Relationship*

Per memorizzare e gestire separatamente i dati di ciascun utente il programma utilizza un *database*, organizzato nelle seguenti tabelle:

- **utenti**
- **inquilini**
- **contratti**
- **immobili**

Le relazioni tra le entità e i relativi vincoli sono rappresentati in Figura 3 utilizzando il modello *Entity Relationship*. Per chiarezza non vengono mostrati gli attributi che non caratterizzano le relazioni tra le varie entità.

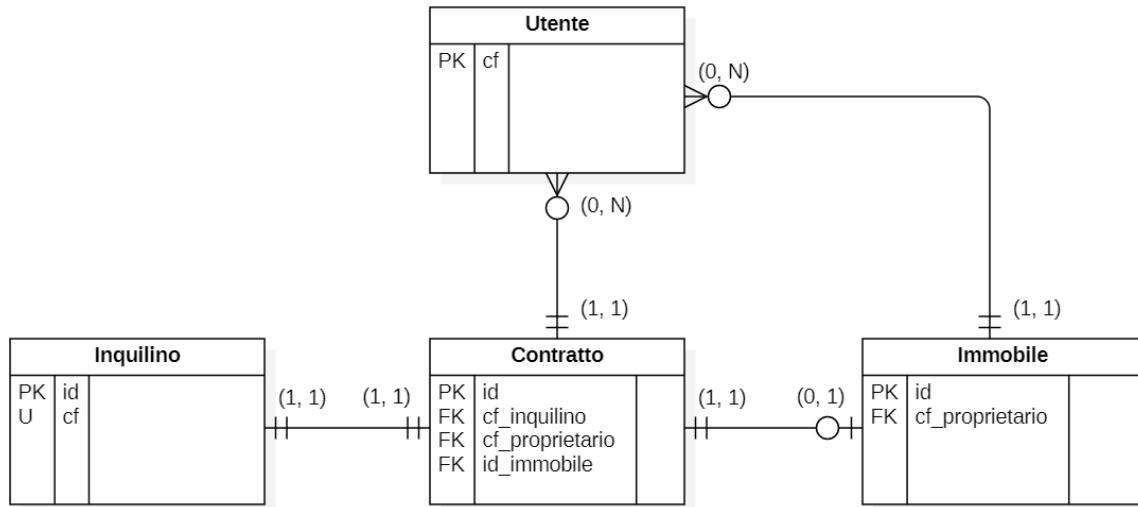


Figura 3: Modello Entity Relationship del database implementato

Si evidenzia che **Contratto** rappresenta una relazione tra **Inquilino**, **Utente** ed **Immobile**. In questa relazione la partecipazione è obbligatoria per tutte le entità che ne prendono parte.

Nella relazione tra **Immobile** e **Contratto** la partecipazione di quest'ultimo è invece opzionale: questa situazione si verifica quando viene creato un immobile che non è affittato. Per questo motivo si è reso necessario associare all'entità **Immobile** un riferimento al relativo proprietario(utente).

2.3.2 Schema logico e tecnologie utilizzate

Il *database* è stato creato con nome `gestionale_affitti` attraverso PHPMyAdmin e utilizzando un server Apache locale come *host*. La gestione è stata effettuata attraverso l'applicazione XAMPP. La comunicazione con il *database* all'interno del programma è effettuata utilizzando le librerie JDBC.

Nelle Figure 4, 6, 7 e 5 è riportato lo schema logico di ciascuna tabella implementata.

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Commenti	Extra	Azione
<input type="checkbox"/>	1 cf 	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	2 nome	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	3 cognome	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	4 email 	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	5 password	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	

Figura 4: Schema logico della tabella utenti

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Commenti	Extra	Azione
<input type="checkbox"/>	1 id 	int(11)			No	Nessuno		AUTO_INCREMENT  Modifica  Elimina  Più	
<input type="checkbox"/>	2 comune	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	3 foglio	int(11)			Si	NULL		 Modifica  Elimina  Più	
<input type="checkbox"/>	4 particella	int(11)			Si	NULL		 Modifica  Elimina  Più	
<input type="checkbox"/>	5 subalterno	int(11)			No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	6 categoria	varchar(255)	utf8mb4_general_ci		Si	NULL		 Modifica  Elimina  Più	
<input type="checkbox"/>	7 classe	varchar(255)	utf8mb4_general_ci		Si	NULL		 Modifica  Elimina  Più	
<input type="checkbox"/>	8 superficie_mq	float			Si	NULL		 Modifica  Elimina  Più	
<input type="checkbox"/>	9 rendita	float			Si	NULL		 Modifica  Elimina  Più	
<input type="checkbox"/>	10 cf_proprietario 	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	11 indirizzo	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	12 n_civico	varchar(255)	utf8mb4_general_ci		No	Nessuno		 Modifica  Elimina  Più	
<input type="checkbox"/>	13 affittato	tinyint(1)			No	Nessuno		 Modifica  Elimina  Più	

Figura 5: Schema logico della tabella immobili

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Commenti	Extra	Azione
<input type="checkbox"/>	1 id 	int(11)			No	Nessuno		AUTO_INCREMENT	 Modifica  Elimina  Più
<input type="checkbox"/>	2 cf_proprietario 	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	3 cf_inquilino 	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	4 id_immobile 	int(11)			No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	5 data_inizio	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	6 data_fine	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	7 prossimo_pagamento	varchar(255)	utf8mb4_general_ci		Si	NULL			 Modifica  Elimina  Più
<input type="checkbox"/>	8 canone	float			No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	9 sfratto	tinyint(1)			No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	10 proroga	tinyint(1)			No	Nessuno			 Modifica  Elimina  Più

Figura 6: Schema logico della tabella contratti

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Commenti	Extra	Azione
<input type="checkbox"/>	1 id 	int(11)			No	Nessuno		AUTO_INCREMENT	 Modifica  Elimina  Più
<input type="checkbox"/>	2 cf 	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	3 nome	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	4 cognome	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	5 data_di_nascita	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	6 città_di_nascita	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	7 residenza	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	8 telefono	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	9 email 	varchar(255)	utf8mb4_general_ci		No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	10 totale_dovuto	float			No	Nessuno			 Modifica  Elimina  Più
<input type="checkbox"/>	11 totale_pagato	float			No	Nessuno			 Modifica  Elimina  Più

Figura 7: Schema logico della tabella inquilini

3 Progettazione

3.1 Page Navigation Diagram

Immaginando di rendere disponibile un’interfaccia grafica completa all’utente abbiamo rappresentato la logica di navigazione tra alcune pagine del sistema nelle Figure 8 e 9. In particolare vengono mostrate graficamente le sequenze di navigazione dei casi rappresentati nei *template* delle Tabelle 2, 3 e 4.

I menu indicati nelle immagini come Menu Benvenuto, Menu Registrazione, Menu Login e Menu Inquilini sono mostrati nei *mockups* in Sezione 3.2.

Per maggiore chiarezza le pagine principali sono state colorate allo stesso modo nei due diagrammi.

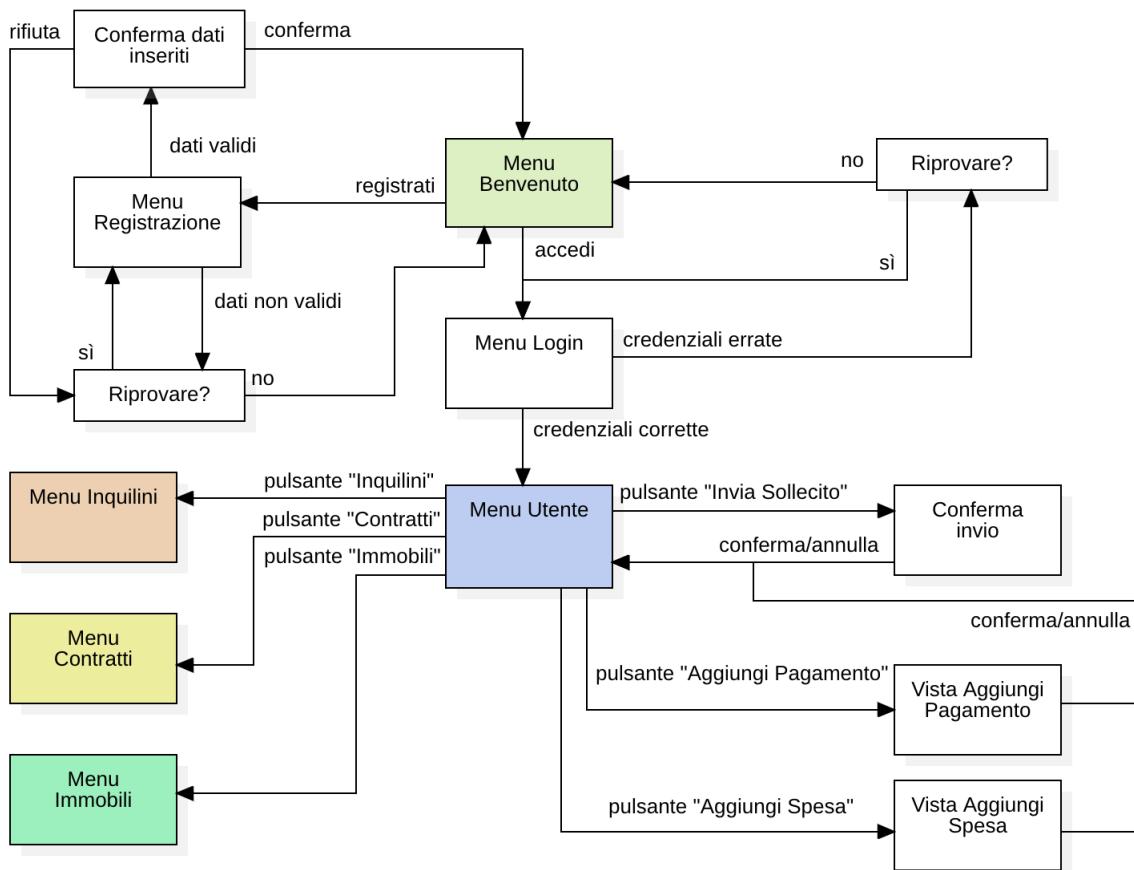


Figura 8: Page Navigation partendo dal Menu Benvenuto, in verde chiaro nell'immagine

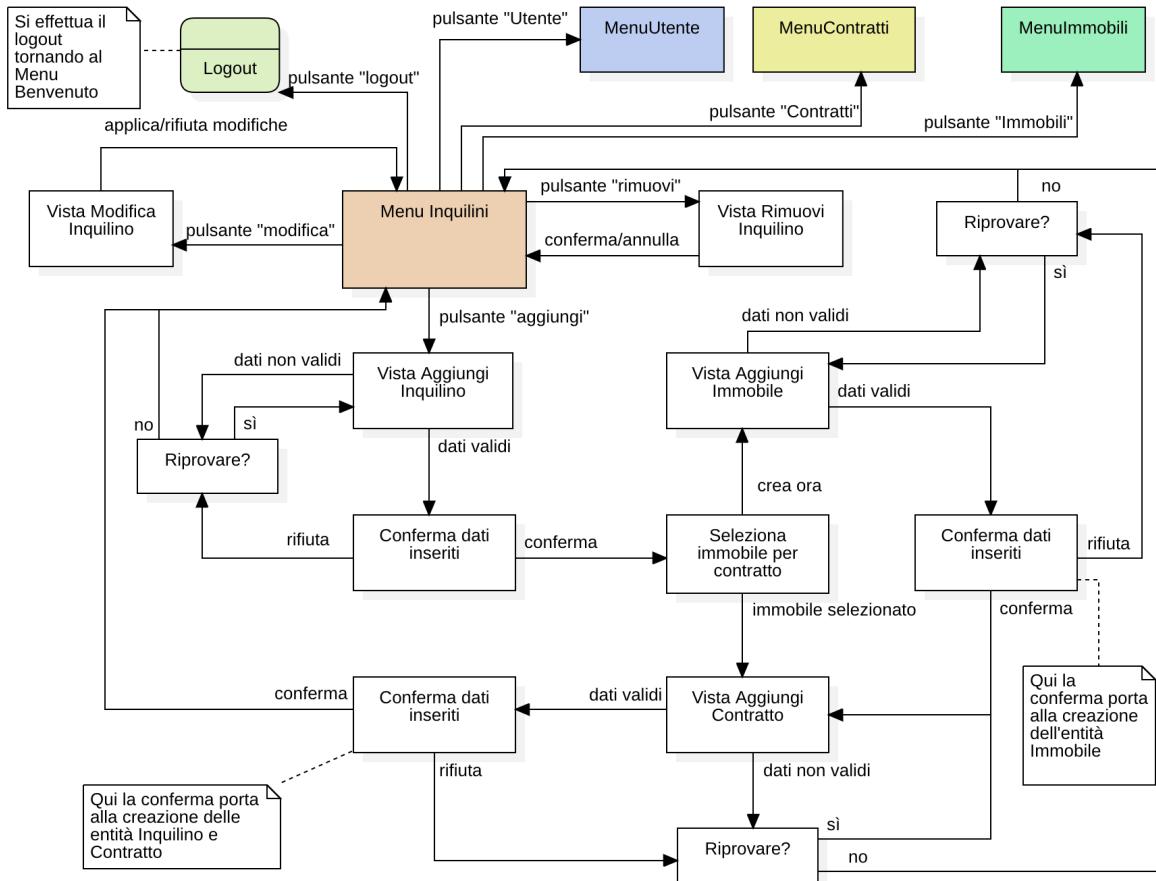


Figura 9: Page Navigation partendo dal Menu Inquilini, in arancione nell'immagine

3.2 Mockup

In questa Sezione vengono mostrati dei *mockups* rappresentanti un'ipotetica GUI per questo programma. Le immagini che riportiamo sono state create con l'aiuto dell'applicazione online Figma, e sono rappresentative dei menu citati nei diagrammi delle Sezioni 1.3 e 3.1.



Figura 10: Mockup del Menu Benvenuto

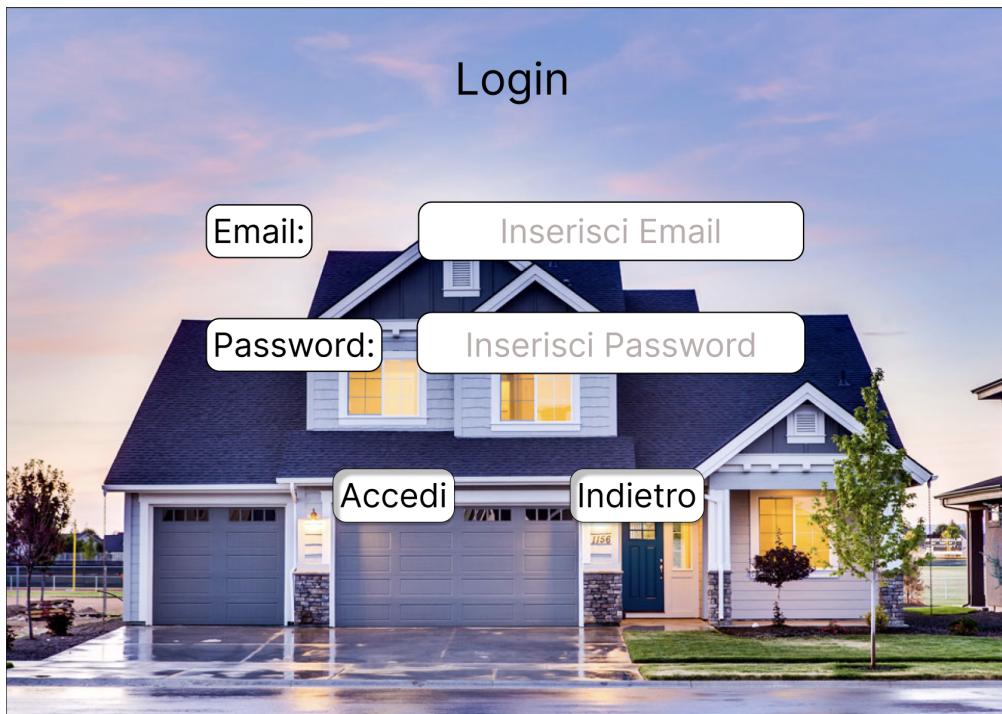


Figura 11: Mockup del Menu Login



Figura 12: Mockup del Menu Registrazione

Figura 13: Mockup del Menu Inquilini

3.3 Class Diagram

In Figura 14 è rappresentato il diagramma delle classi relativo all'intero progetto. Le classi sono suddivise in diversi *package* in base alla tipologia del servizio offerto.

Per maggiore chiarezza in questo diagramma vengono omessi i metodi e gli attributi che compongono le classi, eccezion fatta per quelli ritenuti più significativi ai fini della comprensione del funzionamento del programma, tra cui quelli che rappresentano i casi d'uso in Figura 1.

Le classi che compongono ciascun *package* verranno discusse nel dettaglio in Sezione 4.2, mentre i dettagli sulle dipendenze sono affrontati in Sezione 4.1.

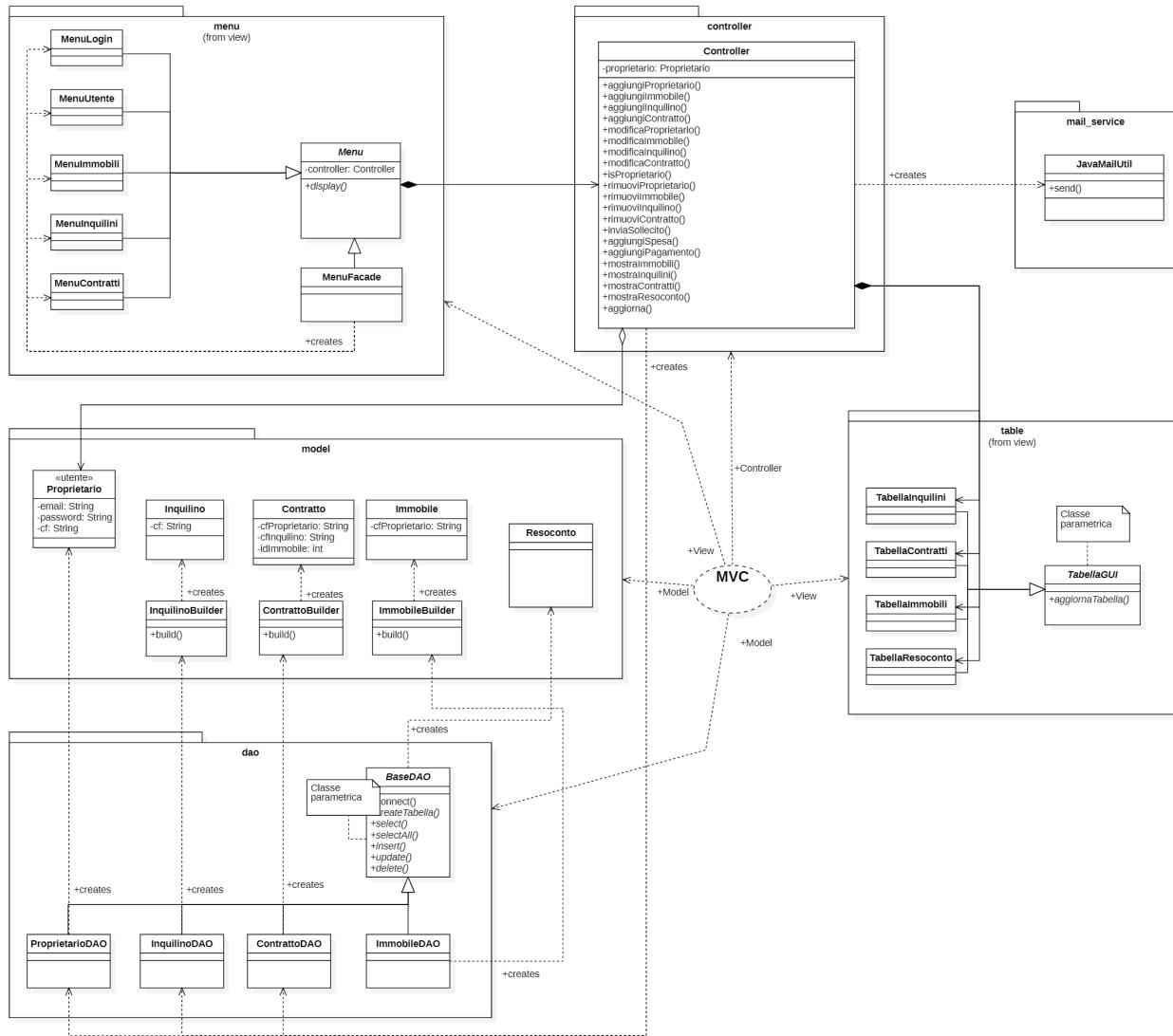


Figura 14: UML Class Diagram dell'intero progetto

4 Implementazione

In questa sezione discuteremo i dettagli implementativi del nostro progetto, concentrandoci sulle caratteristiche più interessanti e spiegando le motivazioni che ci hanno spinto ad adottare determinate soluzioni. L'intero programma è stato realizzato in Java.

4.1 Packages

In questa sezione vengono affrontate nel dettaglio le caratteristiche di implementazione principali dei *packages* utilizzati e le loro dipendenze. Vengono trattati separatamente quelli **personalizzati** - ovvero implementati da noi - e quelli facenti parte della **Java Standard Library** - che sono invece pacchetti predefiniti forniti da Java.

4.1.1 Packages personalizzati

I *packages* implementati da noi sono i seguenti:

- `model`
- `view`
- `controller`
- `dao`
- `mail_service`

Il *package view* contiene due sotto-package con tutte le classi che permettono l'interfaccia tra l'utente e il software. I due sotto-package sono:

- `menu`: contiene classi utilizzate per permettere all'utente di navigare e scegliere le operazioni da eseguire attraverso la CLI. Qui risiede tutta la logica di navigazione tra i vari menu del programma.
- `table`: contiene classi che sfruttano le librerie Swing di Java per rappresentare dati su tabelle attraverso una GUI. Questo *package* è interamente gestito dalla classe `Controller`.

Il *package model* rappresenta il *domain model* del programma, ovvero le classi che rappresentano le entità del dominio applicativo di questa applicazione. Queste vengono istanziate in tutti gli altri *packages* quando è necessario caricare, salvare o modificare dati sul *database*, o quando devono essere mostrati dei dati attraverso GUI.

In Figura 14 sono evidenziati gli attributi che mantengono le relazioni tra le varie classi del *package*.

Il *package controller* contiene tutta la *business logic* del programma. I suoi compiti principali comprendono il connettersi con il *database* per trattare le richieste dell'utente e l'aggiornare l'interfaccia, garantendo che vengano mostrati dati aggiornati nelle tabelle.

Il *package dao* contiene le classi che permettono al programma di interfacciarsi al *database*. Queste vengono istanziate all'interno dei metodi della classe `Controller` e vengono utilizzati solo all'interno del metodo per poi essere distrutti. Questa scelta mira a ridurre al minimo indispensabile il tempo di connessione al *database*.

Il *package mail_service* contiene una sola classe che sfrutta l'API JavaMail per permettere l'invio di email. Questa classe viene utilizzata solo all'interno dei metodi del controller.

Le classi che compongono ciascun *package* verranno trattate nel dettaglio in Sezione 4.2.

4.1.2 Java Standard Libraries

Nel programma vengono utilizzati diversi *packages* delle librerie standard di Java. Quelli più importanti sono:

- `java.time`: viene utilizzato dal *package controller* per gestire la *business logic* delle date.
- `java.swing`: viene utilizzato nel *package table* per creare tabelle visualizzabili attraverso una GUI.
- `JavaMail`: viene utilizzato nel *package controller* per mandare solleciti tramite email agli inquilini.
- `Java DataBase Connectivity (JDBC)`: viene utilizzato nel *package dao* per gestire la connessione al *database* e le operazioni che lo coinvolgono.
- `JUnit`: viene utilizzato per la parte di *Unit Testing* del *software* (vedi Sezione 5).

4.2 Classi

In questa sezione vengono descritte le classi implementate suddivise in base al *package* a cui appartengono.

4.2.1 model

Le classi contenute nel *package model* racchiudono le strutture dati che rappresentano le varie entità del *domain model* del programma, insieme alle classi *Builder* che si occupano della loro creazione.

Per completezza viene riportato in Figura 15 un *Class Diagram* che mostra nel dettaglio le strutture dati qui definite.

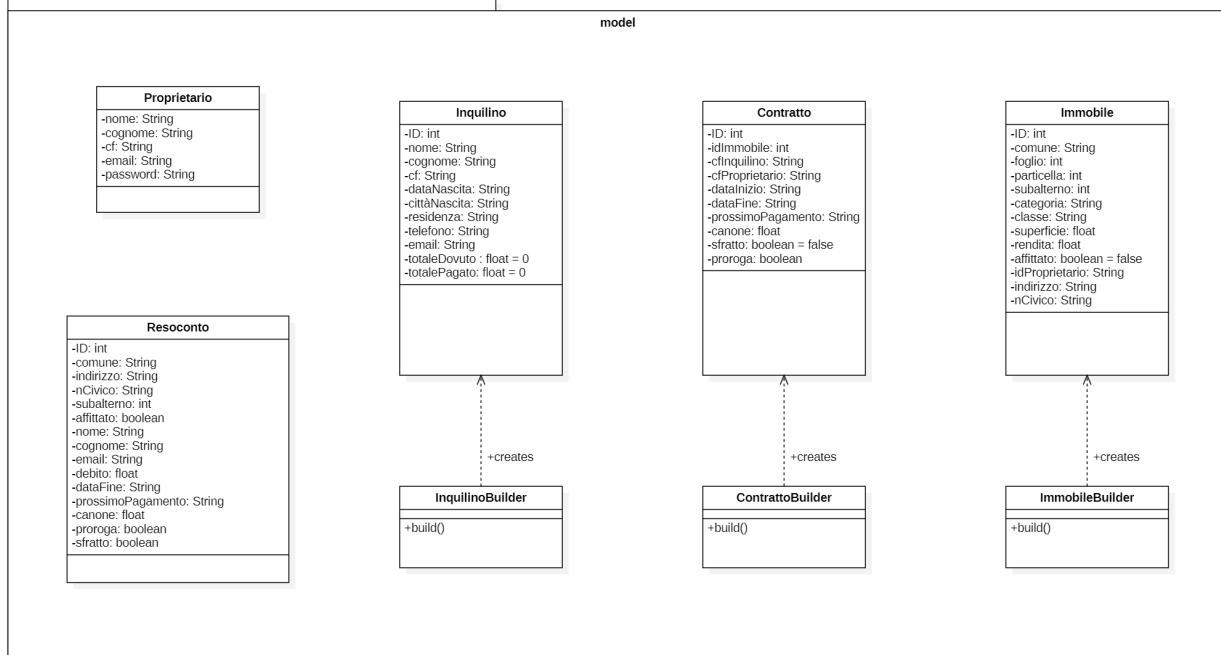


Figura 15: Class Diagram relativo al package model

Ecco un elenco delle classi e delle loro principali caratteristiche:

- **Proprietario**: classe che contiene i dati dell'utente che rappresenta.
- **Inquilino**: contiene informazioni anagrafiche e contatti di un inquilino.
- **Contratto**: contiene dati di un contratto e mantiene una relazione tra un immobile e l'inquilino che ci abita.
- **Immobile**: contiene dati relativi all'immobile.
- **Resoconto**: questa classe non rappresenta un'entità ma contiene dati di resoconto associati ad un singolo immobile.
- **InquilinoBuilder**: utilizzata per creare un inquilino.
- **ContrattoBuilder**: utilizzata per creare un contratto.
- **ImmobileBuilder**: utilizzata per creare un immobile.

Le classi *Builder* hanno tutti gli attributi degli oggetti che devono istanziare ed espongono metodi per assegnare singolarmente ciascun attributo. Inoltre espongono un metodo `build()` che "mette in vita" l'oggetto inizializzando solo gli attributi precedentemente assegnati con i metodi della classe.

Il funzionamento e le motivazioni sull'utilizzo delle classi *Builder* verranno analizzati in maniera più approfondita nella Sezione 4.3.3.

4.2.2 *table*

Contiene le classi che utilizzano le librerie `java.swing` e permettono di creare e mostrare dati organizzati in tabelle attraverso una GUI. Qui è presente una classe astratta `TabellaGUI` che espone due metodi generici - uno dei quali astratto (vedi Figura 16) - e che viene derivata da tutte le altre classi del *package*. Le classi implementate sono:

- `TabellaGUI`: classe parametrica astratta che viene specializzata nelle classi derivate in base all'entità di cui deve mostrare i dati.
- `TabellaInquilini`: permette di mostrare tutti i dati degli oggetti `Inquilino`.
- `TabellaContratti`: permette di mostrare tutti i dati degli oggetti `Contratto`.
- `TabellaImmobili`: permette di mostrare tutti i dati degli oggetti `Immobile`.
- `TabellaResoconto`: permette di mostrare tutti i dati degli oggetti `Resoconto`.

In Figura 17 viene mostrata l'implementazione del metodo generico `aggiornaTabella` della classe `TabellaInquilini` derivata da `TabellaGUI`.

```
5 usages 4 implementations  ↗ marco-vignozzi
public abstract void aggiornaTabella(List<T> lista);

4 usages  ↗ marco-vignozzi
public void mostraTabella(List<T> lista) {
    aggiornaTabella(lista);
    setVisible(true);
}
```

Figura 16: Metodi generici della classe `TabellaGUI`

```
5 usages  ↗ marco-vignozzi
@Override
public void aggiornaTabella(List<Inquilino> lista) {
    modelloTabella.setRowCount(0);

    for(Inquilino i: lista) {
        Object[] riga = {i.getID(), i.getCf(), i.getNome(), i.getCognome(), i.getDataNascita(), i.getCittàNascita(),
                        i.getResidenza(), i.getTelefono(), i.getEmail(), i.getTotaleDovuto(), i.getTotalePagato(),
                        i.getTotaleDovuto()-i.getTotalePagato()};
        modelloTabella.addRow(riga);
    }
}
```

Figura 17: Esempio di override del metodo astratto `aggiornaTabella` implementato nella classe `TabellaInquilini`

4.2.3 menu

Contiene classi che permettono all'utente di scegliere operazioni ed eventualmente immettere dati attraverso la CLI. Questo *package* è formato dalle seguenti classi:

- **Menu**: classe astratta che viene derivata da tutte le altre classi del *package*. Espone il metodo astratto `display()` che contiene la logica di navigazione.
- **MenuFacade**: classe che ha il compito di istanziare i vari menu e decidere quale deve essere mostrato. Gestisce il "Menu Home".
- **MenuLogin**: permette di registrare un nuovo utente o accedere con uno già esistente.
- **MenuContratti**: permette di aggiungere/rimuovere/modificare un contratto o mostrare i contratti tramite tabella.
- **MenuInquilini**: permette di aggiungere/rimuovere/modificare dati di inquilini o mostrare tutti gli inquilini tramite tabella.
- **MenuImmobili**: permette di aggiungere/rimuovere/modificare dati di un immobile o mostrare tutti gli immobili tramite tabella.
- **MenuUtente**: permette di inviare uno o più solleciti, aggiungere un pagamento fatto da un inquilino, aggiungere una spesa a carico di un inquilino e di mostrare la tabella di resoconto.

Ciascuna di queste classi mantiene un riferimento a un oggetto **Controller** utilizzato per eseguire le operazioni scelte dall'utente.

4.2.4 controller

Il *package* **controller** contiene solo l'omonima classe **Controller**.

La classe ha un ruolo cruciale per il corretto funzionamento del programma e svolge le seguenti funzioni:

- Gestione delle operazioni relative alle scelte effettuate dall'utente attraverso i menu.
- Interfaccia con il *database* per tutte le operazioni che necessitano l'aggiunta, la modifica e la rimozione di dati da esso.
- Operazioni di *business logic* sui dati ogni volta che viene effettuata una modifica al *database*.
- Aggiornamento delle tabelle GUI.

Tutta la *business logic* del programma risiede in questa classe: come si può vedere in Figura 14 tutti i metodi che rappresentano i casi d'uso della Figura 1 sono implementati qui.

Essa mantiene un riferimento a un oggetto tabella di ciascuna delle classi del *package* **table**, oltre che un riferimento a un oggetto **Proprietario** inizializzato con i dati del Proprietario al momento dell'accesso. Usa tutte le classi derivate del *package* **dao**, istanziandole all'interno dei metodi in modo da mantenere la connessione al *database* soltanto per il tempo necessario ad eseguire le operazioni richieste.

Nel metodo `inviaSollecito()` esposto dalla classe viene inoltre utilizzata la classe **JavaMailUtil** dal *package* **mail_service**, che viene istanziata per effettuare l'invio di email.

4.2.5 dao

Ciascuna classe DAO all'interno di questo *package* ha il compito di fare da intermediario tra il software e il database. Abbiamo strutturato il *package* con una classe DAO per ogni struttura dati, eccetto **Resoconto**. Ciascuna di queste classi implementa dei metodi che le permettono di aggiungere, modificare, eliminare e selezionare dati dal database. I metodi corrispondono alle operazioni SQL di INSERT, UPDATE, DELETE e SELECT.

Il *package* contiene le classi:

- **BaseDAO**: è una classe parametrica astratta da cui derivano tutte le altre classi del *package*. Espone il metodo **connect** che permette la connessione del nostro software al database. Essa definisce i metodi astratti mostrati in Figura 14 che corrispondono alle operazioni SQL elencate in precedenza. Questi metodi sono definiti come *generics* e nelle implementazioni delle classi derivate vengono specializzati sulla base dell'entità da loro gestita.
- **InquilinoDAO**: è la classe con il quale il software si interfaccia con la tabella "inquilini".
- **ContrattoDAO**: è la classe con il quale il software si interfaccia con la tabella "contratti".
- **ImmobileDAO**: è la classe con il quale il software si interfaccia con la tabella "immobili".
- **ProprietarioDAO**: è la classe con il quale il software si interfaccia con la tabella "utenti". E' più sicuro salvare le password in locale.

L'operazione di **SELECT** viene implementata nei metodi **select** e **selectAll**: la prima riceve come parametro l'ID di un elemento della tabella e ritorna il relativo oggetto (del tipo cui fa riferimento il DAO), la seconda non riceve parametri e ritorna una lista con tutti gli elementi della tabella che gestisce il DAO chiamante.

Il reperimento dei dati di resoconto viene affidato a un metodo esposto dalla stessa classe base **BaseDAO** che esegue una **SELECT** sulle tabelle "contratti", "inquilini" e "immobili" del *database*. Questa scelta è stata fatta per semplificare la creazione degli oggetti **Resoconto**. In alternativa, per consistenza con la nostra definizione di classe DAO, avremmo potuto affidare la loro creazione alla classe **Controller**, rendendo però il codice più complesso.

Il metodo che implementa l'operazione di **UPDATE** riceve invece un oggetto e un ID, aggiornando tutti i campi della riga con quell'ID con i valori inizializzati dell'oggetto ricevuto come parametro (l'oggetto ricevuto è creato con il *Builder pattern*, in questo modo saranno inizializzati solo i valori da modificare, come spiegato in Sezione 4.3.3).

Ciascun metodo di queste classi necessita di connettersi al *database* per eseguire le operazioni. La connessione viene effettuata all'inizio del metodo e chiusa appena prima di terminare la chiamata. In questo modo si evita il rischio di effettuare troppe connessioni contemporaneamente e si mantiene la connessione solo per il tempo strettamente necessario a completare l'operazione.

4.2.6 mail_service

Questo *package* contiene solo la classe **JavaMailUtil**. Per poter inviare email di sollecito agli inquilini tramite un servizio email automatico, questa classe utilizza l'API JavaMail, che fornisce un framework *platform-indipendent* - ovvero può essere eseguito su diverse piattaforme hardware o sistemi operativi senza richiedere modifiche significative - e *protocol-indipendent* - ovvero comunica senza essere vincolato da un protocollo specifico.

La classe è impostata per l'invio di email tramite il servizio SMTP(*Simple Mail Transfer Protocol*) di Gmail. L'utilizzo di questa classe è confinato al metodo **inviaSollecito** della classe **Controller**, dove viene istanziata e utilizzata solamente per la durata della chiamata.

Questa classe ha un metodo `send` che permette di mandare un dato messaggio ad un dato indirizzo email, entrambi specificati come parametri, mentre l'email del mittente viene inizializzata nel costruttore.

Affinchè l'invio di email funzioni correttamente è necessario aggiungere la password fornita da Google in un file di testo "password.txt" situato nella *root directory* del progetto. La password deve essere preceduta dall'indirizzo email a cui si riferisce, utilizzando il carattere ":" come separatore, nel seguente modo:

```
<indirizzo_email>:<password_google>
```

In questo modo è possibile inserire una password per ogni utente che utilizza l'applicazione semplicemente aggiungendo una riga nel file sopra indicato.

Per chiarezza in Figura 18 viene mostrato il codice che implementa questo meccanismo.

```
1 usage  ± marcodestefano
private String readPassword(){
    try(BufferedReader br = new BufferedReader(new java.io.FileReader( fileName: "password.txt"))){
        String line;
        while((line = br.readLine()) != null){
            String[] parts = line.split( regex: ":" );
            if(parts[0].equals(account)){
                return parts[1];
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Figura 18: Metodo della classe JavaMailUtil che ha il compito di estrarre la giusta password per l'account dell'utente che sta utilizzando il programma

4.3 Dettagli di progettazione

Dopo aver discusso le classi implementate spieghiamo adesso i dettagli di progettazione, con particolare attenzione ai *Design Pattern* utilizzati, che sono:

- Model View Controller
- Facade
- Builder
- Data Access Object

4.3.1 Model View Controller

Il *Model View Controller*, o MVC, è un *design pattern* architetturale che permette la separazione delle responsabilità e una struttura modulare. Esso suddivide l'applicazione in tre parti principali: il *Model*, la *View* e il *Controller*. Abbiamo affidato a ciascuna parte le seguenti responsabilità:

- *Model*: gestisce l'accesso, la manipolazione e la rappresentazione dei dati.
- *View*: gestisce la visualizzazione dei dati e riceve l'input dell'utente.
- *Controller*: gestisce la *business logic* ed il flusso dei dati tra *Model* e *View*.

Nel nostro progetto il modello e la vista sono naturalmente separate: l'utente comunica con il *software* attraverso CLI, mentre i dati su cui deve operare si trovano inizialmente su un *database*. Inoltre può visualizzare tabelle attraverso una GUI. Di conseguenza ci è sembrato utile ed efficace utilizzare un'entità frapposta tra queste componenti che si occupi di gestire la comunicazione tra utente e *database*, e che garantisca che i dati fruiti siano sempre aggiornati. Il *Controller* permette di organizzare e gestire questa interazione garantendo l'attualità dei valori utilizzati.

Il flusso di esecuzione avviene in questo modo:

1. L'utente naviga tra i menu da CLI e sceglie l'operazione da eseguire; viene quindi richiamato un metodo di una classe del package `menu` che gestisce l'operazione.
2. Il metodo richiama il *Controller* che crea una rappresentazione del *Model* interfacciandosi con il *database* attraverso le classi del package `dao` e mappando i dati in istanze delle classi del package `model`.
3. Il *Controller* esegue l'operazione richiesta e aggiorna il modello sul *database*.
4. Il *Controller* aggiorna i dati delle sue tabelle GUI - classi del package `table` - con i nuovi dati del modello risultanti dall'operazione.

Sulla base di queste considerazioni abbiamo scelto di associare alla *View* tutto ciò che mostra un'interfaccia all'utente, ovvero le classi del package `view`. Al suo interno abbiamo adottato una divisione di responsabilità: nel sotto-package `menu` sono racchiuse le classi che gestiscono l'interazione con l'utente, mentre in `table` sono contenute le classi che mostrano un'interfaccia grafica all'utente.

Abbiamo invece scelto di associare al *Model* il *database* e tutte le classi facenti parte dei packages `model` e `dao`.

4.3.2 *Facade*

Il *design pattern* strutturale *Facade* permette di semplificare l'utilizzo di un insieme di classi, fornendo un'interfaccia concisa e semplice da comprendere e utilizzare.

Data la complessità della logica di navigazione tra le classi all'interno del *package menu* (vedi Sezione 4.2.3), ci è sembrato necessario utilizzare uno stratagemma per rendere più semplice il loro utilizzo. E' qui che il *pattern Facade* entra in gioco: con l'utilizzo della classe `MenuFacade` è stato possibile racchiudere la logica di navigazione all'interno di un semplice metodo che istanzia i Menu richiesti dall'utente e gli passa il controllo dell'esecuzione fintanto che vengono utilizzati.

4.3.3 *Builder*

Il *Builder* è un *design pattern* creazionale che serve a separare il processo di creazione di un oggetto complesso dalla sua rappresentazione, consentendo di creare diverse rappresentazioni dell'oggetto utilizzando lo stesso processo di costruzione.

In questa implementazione viene utilizzato un tipo particolare di *Builder* senza *Director* chiamato *Telescoping Constructor* che permette di semplificare la creazione di oggetti con molti attributi. Il *pattern* permette infatti di assegnare singolarmente ciascuno di essi con un metodo. Per creare l'oggetto utilizza il metodo `build()` che ritorna un'istanza della classe con quei soli attributi inizializzati.

Nel nostro progetto sono presenti molte classi (vedi Sezione 4.2.1) con tanti attributi il cui valore, spesso, non viene utilizzato.

La scelta di implementare questo *pattern* mira a ridurre la ripetizione del codice e semplificare alcune operazioni sugli oggetti: quando, ad esempio, si effettua una modifica di un'entità - inquilino, immobile o contratto - e si deve eseguire un `UPDATE` sul *database*, questo *pattern* semplifica l'operazione permettendo di istanziare un oggetto inizializzando solo i valori che si desidera modificare. Questo rende possibile l'utilizzo di un'unico metodo di `UPDATE` - all'interno delle classi descritte in Sezione 4.2.5 - che andrà a eseguire l'aggiornamento solo sui campi inizializzati nell'oggetto passato come parametro.

4.3.4 *Data Access Object*

Il *Data Access Object* o *DAO* è un *pattern* architetturale utilizzato per separare la *business logic* dalla logica di accesso ai dati. Questo pattern viene utilizzato come intermediario tra il *software* e il *database*. Abbiamo utilizzato una classe astratta da cui derivano le altre classi del *package dao* (vedi Sezione 4.2.5).

Abbiamo scelto di creare un interfaccia *DAO* per ogni relazione per rendere il più modulare possibile il programma.

5 Unit Testing

Il *testing* del programma è stato effettuato con la libreria JUnit 5. Per questo progetto sono state testate solo le funzionalità ritenute essenziali, ovvero le operazioni indicate nei casi d'uso in Figura 1 e quelle considerate più critiche per il corretto funzionamento del programma. Ci siamo quindi focalizzati sui metodi delle classi del *package dao*, responsabile della memorizzazione e del caricamento dei dati, e della classe **Controller**, dove risiede tutta la *business logic* del programma.

E' stata creata un'unica classe **ControllerTest** che effettua 1 test in prospettiva *White Box* e 7 in prospettiva *Grey Box*. La scelta di utilizzare un'unica classe è stata dettata dal basso numero di test implementati in questo progetto.

L'unico test con approccio *White Box* riguarda la classe **ProprietarioDAO**, mentre i test in prospettiva *Grey Box* verificano l'integrazione dei *packages controller* e *dao*. La classe **ControllerTest** contiene i metodi di test elencati di seguito:

- **testAggiungiProprietario**: si verifica che l'aggiunta di un nuovo **Proprietario** al *database* venga eseguita correttamente - vedi Figura 21. Questo è l'unico test in prospettiva *White Box*.
- **testAggiungi**: si testa l'aggiunta al *database* di tutte le altre entità. Qui si verifica che il campo **affittato** dell'immobile venga stabilito correttamente, così come la data **prossimo_pagamento** del contratto - vedi Figura 28.
- **testModifica**: si verifica che la modifica sul *database* di un immobile avvenga correttamente - vedi Figura 22.
- **testAggiungiSpesa**: si verifica che aggiungendo una spesa il campo **totale_dovuto** dell'inquilino cui si riferisce sia aggiornato correttamente - vedi Figura 26.
- **testAggiungiPagamento**: si verifica che aggiungendo un pagamento da parte dell'inquilino, il campo **totale_pagato** venga aggiornato correttamente - vedi Figura 27.
- **testRimuoviImmobile**: si verifica che rimuovendo l'immobile dal *database* il contratto e l'inquilino ad esso associati vengano cancellati - vedi Figura 24.
- **testRimuoviInquilino**: si verifica che rimuovendo l'inquilino dal *database* il contratto ad esso associato venga cancellato - vedi Figura 25.
- **testRimuoviContratto**: si verifica che rimuovendo il contratto dal *database* l'inquilino ad esso associato venga cancellato - vedi Figura 23.

La classe contiene gli attributi **controller**, **proprietario**, **immobile** e **inquilino**.

Nel costruttore della classe viene inizializzato l'oggetto **controller** assegnando il suo attributo **proprietario**, in modo da poterlo utilizzare in tutti i test senza doverlo istanziare. Il riferimento al **proprietario** viene mantenuto anche come attributo della classe per semplificare l'accesso al valore dei suoi attributi.

Prima di eseguire ciascun metodo di test viene richiamato il metodo **setUp()** - vedi Figura 19 - marcato con l'annotazione **@BeforeEach**. Esso reinizializza gli attributi **immobile** e **inquilino** che possono essere modificati nei test successivi. Dopo l'esecuzione di ogni test viene richiamato il metodo **tearDown()** - vedi Figura 20 - marcato con l'annotazione **@AfterEach** per rimuovere dal *database* i dati inseriti durante i test precedentemente eseguiti.

```

@BeforeEach
void setUp() {
    ImmobileBuilder immobileBuilder = new ImmobileBuilder();
    immobileBuilder.comune("example")
        .foglio(1)
        .particella(1)
        .subalterno(1)
        .categoria("example")
        .classe("example")
        .superficie(1)
        .rendita(1)
        .indirizzo("example")
        .nCivico("example")
        .affittato(false);
    this.immobile = immobileBuilder.build();
    InquilinoBuilder ibuilder = new InquilinoBuilder();
    ibuilder.cf("EXAMPLE")
        .nome("example")
        .cognome("example")
        .cittàNascita("example")
        .dataNascita("example")
        .email("example@gmail.com")
        .residenza("example")
        .totaleDovuto(100)
        .totalePagato(0)
        .telefono("example");
    this.inquilino = ibuilder.build();
}

```

Figura 19: codice del metodo setUp

```

@AfterEach
void tearDown(){
    ProprietarioDAO proprietarioDAO = new ProprietarioDAO();
    List<Proprietario> utenti = proprietarioDAO.selectAll( cf: "" );
    for(Proprietario utente : utenti) {
        if(utente.getEmail().equals("example@gmail.com")) {
            proprietarioDAO.delete( id: 1, utente.get Cf() );
        }
    }
    InquilinoDAO inquilinoDAO = new InquilinoDAO();
    List<Inquilino> inquilini = inquilinoDAO.selectAll( cfProprietario: "A" );
    for(Inquilino i: inquilini) {
        inquilinoDAO.delete(i.getID(), cfProprietario: "A");
    }
    ContrattoDAO contrattoDAO = new ContrattoDAO();
    List<Contratto> contratti = contrattoDAO.selectAll( cfProprietario: "A" );
    for(Contratto c: contratti) {
        contrattoDAO.delete(c.getID(), cfProprietario: "A");
    }
    ImmobileDAO immobileDAO = new ImmobileDAO();
    List<Immobile> immobili = immobileDAO.selectAll( cfProprietario: "A" );
    for(Immobile immobile : immobili) {
        int id = immobile.getID();
        immobileDAO.delete(id, cfProprietario: "");
    }
}

```

Figura 20: codice del metodo tearDown

```

@Test
void testAggiungiProprietario() {           // va fatto per forza sul DAO visto che non c'è metodo getUtente() in Controller
    ProprietarioDAO proprietarioDAO = new ProprietarioDAO();
    Proprietario p = new Proprietario( email: "example@gmail.com", pwd: "example", nome: "example", cognome: "example", cf: "EXAMPLE");
    assertTrue(proprietarioDAO.insert(p));
    Proprietario nuovoProprietario = proprietarioDAO.select( email: "example@gmail.com", password: "example");
    assertEquals(nuovoProprietario.get Cf(), actual: "EXAMPLE");
    assertEquals(nuovoProprietario.getNome(), actual: "example");
    assertEquals(nuovoProprietario.getCognome(), actual: "example");
    assertEquals(nuovoProprietario.getPassword(), actual: "example");
    assertEquals(nuovoProprietario.getEmail(), actual: "example@gmail.com");
}

```

Figura 21: codice del metodo testAggiungiProprietario

```

@Test
void testModifica(){
    //test modifica immobile
    ImmobileDAO immobileDAO = new ImmobileDAO();
    ImmobileBuilder immBuilder = new ImmobileBuilder();
    immBuilder.indirizzo("modificato");

    Immobile immod=immBuilder.build();
    assertTrue(controller.aggiungiImmobile(this.immobile));
    assertTrue(controller.modificaImmobile(Integer.toString(this.immobile.getID()), immod));
    this.immobile = immobileDAO.select(this.immobile.getID(), this.proprietario.get Cf());
    assertEquals(this.immobile.getIndirizzo(), actual: "modificato");
}

```

Figura 22: codice del metodo testModifica

```

no usages ± marcodestefano
void testRimuoviContratto() {
    assertTrue(this.controller.aggiungiImmobile(this.immobile));

    assertTrue(this.controller.aggiungiInquilino(this.inquilino));

    ContrattoBuilder cbuilder = new ContrattoBuilder();
    cbuilder.proroga(true)
        .sfratto(true)
        .canone(1000)
        .prossimoPagamento( dataPagamento: "10")
        .dataFine("2025-01-01")
        .dataInizio("2023-01-01")
        .cfInquilino("EXAMPLE")
        .idImmobile(this.immobile.getID());
    Contratto contratto = cbuilder.build();

    assertTrue(this.controller.aggiungiContratto(contratto));

    // seleziona gli id di inquilino e contratto appena inseriti e verifico che rimuovendo inquilino venga
    // rimosso anche contratto
    List<Immobile> immobili = this.controller.getAllImmobili();
    for (Immobile i: immobili) {
        if(i.getIdProprietario().equals("A")) {
            this.immobile.setID(i.getID());
        }
    }
    List<Inquilino> inquilini = this.controller.getAllInquilini();
    for (Inquilino i: inquilini) {
        if(i.get Cf().equals("EXAMPLE")) {
            this.inquilino.setID(i.getID());
        }
    }
    List<Contratto> contratti = this.controller.getAllContratti();
    for (Contratto c: contratti) {
        if(c.getIdProprietario().equals("EXAMPLE")) {
            contratto.setID(c.getID());
        }
    }
    assertTrue(this.controller.rimuoviContratto(Integer.toString(contratto.getID())));
    assertFalse(this.controller.rimuoviInquilino(Integer.toString(this.inquilino.getID())));
    assertTrue(this.controller.rimuoviImmobile(Integer.toString(this.immobile.getID())));
}

```

Figura 23: codice del metodo testRimuoviContratto

```

// Qui si verifica se eliminando l'inquilino si elimina anche il contratto
@marcodestefano
@Test
void testRimuoviImmobile() {
    assertTrue(this.controller.aggiungiImmobile(this.immobile));

    assertTrue(this.controller.aggiungiInquilino(this.inquilino));

    ContrattoBuilder cbuilder = new ContrattoBuilder();
    cbuilder.proroga(true)
        .sfratto(true)
        .canone(1000)
        .prossimoPagamento(dataPagamento: "10")
        .dataFine("2025-01-01")
        .dataInizio("2023-01-01")
        .cfInquilino("EXAMPLE")
        .idImmobile(this.immobile.getID());
    Contratto contratto = cbuilder.build();

    assertTrue(this.controller.aggiungiContratto(contratto));

    // seleziono gli id di inquilino e contratto appena inseriti e verifico che rimuovendo inquilino venga
    // rimosso anche contratto
    List<Immobile> immobili = this.controller.getAllImmobili();
    for (Immobile i: immobili) {
        if(i.getIdProprietario().equals("A")) {
            this.immobile.setID(i.getID());
        }
    }
    List<Contratto> contratti = this.controller.getAllContratti();
    for (Contratto c: contratti) {
        if(c.getCfProprietario().equals("EXAMPLE")) {
            contratto.setID(c.getID());
        }
    }
    assertTrue(this.controller.rimuoviImmobile(Integer.toString(this.immobile.getID())));
    assertFalse(this.controller.rimuoviContratto(Integer.toString(contratto.getID())));
}

```

Figura 24: codice del metodo testRimuoviImmobile

```

// Qui si verifica se eliminando l'inquilino si elimina anche il contratto
// marcodelstefano
@Test
void testRimuoviInquilino() {
    assertTrue(this.controller.aggiungiImmobile(this.immobile));

    assertTrue(this.controller.aggiungiInquilino(this.inquilino));

    ContrattoBuilder cbuilder = new ContrattoBuilder();
    cbuilder.proroga(true)
        .sfratto(true)
        .canone(1000)
        .prossimoPagamento(dataPagamento: "10")
        .dataFine("2025-01-01")
        .dataInizio("2023-01-01")
        .cfInquilino("EXAMPLE")
        .idImmobile(this.immobile.getID());
    Contratto contratto = cbuilder.build();

    assertTrue(this.controller.aggiungiContratto(contratto));

    // seleziono gli id di inquilino e contratto appena inseriti e verifico che rimuovendo inquilino venga
    // rimosso anche contratto
    List<Inquilino> inquilini = this.controller.getAllInquilini();
    for (Inquilino i: inquilini) {
        if(i.getCf().equals("EXAMPLE")) {
            this.inquilino.setID(i.getID());
        }
    }
    List<Contratto> contratti = this.controller.getAllContratti();
    for (Contratto c: contratti) {
        if(c.getCFProprietario().equals("EXAMPLE")) {
            contratto.setID(c.getID());
        }
    }
    assertTrue(this.controller.rimuoviInquilino(Integer.toString(this.inquilino.getID())));
    // verifico che non ci sia più il contratto
    assertFalse(this.controller.rimuoviContratto(Integer.toString(contratto.getID())));
}

```

Figura 25: codice del metodo testRimuoviInquilino

```

//testo l'aggiunta di una spesa al totale dovuto
@marcodestefano
@Test
void testAggiungiSpesa(){
    assertTrue(this.controller.aggiungiInquilino(this.inquilino));
    // senza contratto l'inquilino non viene trovato con la select
    // e senza immobile non puoi creare un contratto
    assertTrue(this.controller.aggiungiImmobile(this.immobile));
    ContrattoBuilder cbuilder = new ContrattoBuilder();
    cbuilder.proroga(true)
        .sfratto(true)
        .canone(1000)
        .prossimoPagamento( dataPagamento: "10")
        .dataFine("2025-01-01")
        .dataInizio("2023-01-01")
        .cfInquilino("EXAMPLE")
        .idImmobile(this.immobile.getID());
    Contratto contratto = cbuilder.build();
    assertTrue(this.controller.aggiungiContratto(contratto));

    List<Inquilino> inquilini = this.controller.getAllInquilini();
    for (Inquilino i: inquilini) {
        if(i.get Cf().equals("EXAMPLE")) {
            assertEquals(this.inquilino.getTotaleDovuto(), i.getTotaleDovuto(), delta: 0.0);
            this.inquilino.setID(i.getID());
        }
    }
    // verifichiamo che la spesa sia stata aggiunta al totale dovuto dall'inquilino
    controller.aggiungiSpesa(Integer.toString(inquilino.getID()), spesa: "600");
    inquilini = this.controller.getAllInquilini();
    for (Inquilino i: inquilini) {
        if(i.getID() == inquilino.getID()) {
            assertEquals( expected: this.inquilino.getTotaleDovuto() + 600, i.getTotaleDovuto(), delta: 0.0);
        }
    }
}

```

Figura 26: codice del metodo testAggiungiSpesa

```


± marcodestefano
@Test
void testAggiungiPagamento() {
    assertTrue(this.controller.aggiungiInquilino(this.inquilino));
    // senza contratto l'inquilino non viene trovato con la select
    // e senza immobile non puoi creare un contratto
    assertTrue(this.controller.aggiungiImmobile(this.immobile));
    ContrattoBuilder cbuilder = new ContrattoBuilder();
    cbuilder.proroga(true)
        .sfratto(true)
        .canone(1000)
        .prossimoPagamento( dataPagamento: "10")
        .dataFine("2025-01-01")
        .dataInizio("2023-01-01")
        .cfInquilino("EXAMPLE")
        .idImmobile(this.immobile.getID());
    Contratto contratto = cbuilder.build();
    assertTrue(this.controller.aggiungiContratto(contratto));

    List<Inquilino> inquilini = this.controller.getAllInquilini();
    for (Inquilino i : inquilini) {
        if (i.get Cf().equals("EXAMPLE")) {
            assertEquals(this.inquilino.getTotalPagato(), i.getTotalPagato(), delta: 0.0);
            this.inquilino.setID(i.getID());
        }
    }
    // verifichiamo che la spesa sia stata aggiunta al totale dovuto dall'inquilino
    controller.aggiungiPagamento(Integer.toString(inquilino.getID()), pagamento: "600");
    inquilini = this.controller.getAllInquilini();
    for (Inquilino i : inquilini) {
        if (i.getID() == inquilino.getID()) {
            assertEquals( expected: this.inquilino.getTotalPagato() + 600, i.getTotalPagato(), delta: 0.0);
        }
    }
}


```

Figura 27: codice del metodo testAggiungiPagamento

```

    @Test
    void testAggiungi() {
        ImmobileDAO immobileDAO = new ImmobileDAO();
        // verifichiamo che l'immobile venga aggiunto
        assertTrue(this.controller.aggiungiImmobile(this.immobile));
        // verifichiamo che l'ID sia stato aggiunto correttamente all'oggetto immobile appena inserito
        List<Immobile> immobili = this.controller.getAllImmobili();
        for(Immobile i: immobili) {
            if(Objects.equals(i.getComune(), this.immobile.getComune()) && Objects.equals(i.getIndirizzo(), this.immobile.getIndirizzo()) &&
                Objects.equals(i.getnCivico(), this.immobile.getnCivico()) && i.getSubalterno() == this.immobile.getSubalterno()) {
                assertEquals(this.immobile.getID(), i.getID());
                assertEquals(this.immobile.getIdProprietario(), this.proprietario.getcf());
            }
        }
        // verifichiamo che affittato sia false automaticamente
        this.immobile = immobileDAO.select(immobile.getID(), this.proprietario.getcf());
        assertEquals(this.immobile.isAffittato(), actual: false);
        // verifichiamo che l'inquilino sia aggiunto correttamente
        assertTrue(this.controller.aggiungiInquilino(inquilino));
        ContrattoBuilder cbuilder = new ContrattoBuilder();
        cbuilder.priroga(true)
            .sfratto(true)
            .canone(1000)
            .prossimoPagamento( dataPagamento: "10")
            .dataFine("2025-01-01")
            .dataInizio("2023-01-01")
            .cfInquilino("EXAMPLE")
            .idImmobile(this.immobile.getID());
        Contratto contratto = cbuilder.build();
        // verifichiamo che il contratto sia aggiunto correttamente
        assertTrue(this.controller.aggiungiContratto(contratto));
        List<Contratto> contratti = this.controller.getAllContratti();
        // verifichiamo che la data del prossimo pagamento sia corretto in base alla data di oggi.
        LocalDate oggi = LocalDate.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        for(Contratto c: contratti) {
            LocalDate dataFine = LocalDate.parse(c.getDataFine(), formatter);
            LocalDate prossimoPagamento = LocalDate.parse(c.getProssimoPagamento());
            if (!oggi.isAfter(dataFine) && !(oggi.getDayOfMonth() > prossimoPagamento.getDayOfMonth())) {
                assertEquals(prossimoPagamento.toString(),
                    actual: oggi.getYear() + "-" + oggi.getMonthValue() + "-" +
                    LocalDate.parse(contratto.getProssimoPagamento(), formatter).getDayOfMonth());
            } else if(oggi.getDayOfMonth() > prossimoPagamento.getDayOfMonth() && !oggi.isAfter(dataFine)) {
                assertEquals(prossimoPagamento.toString(),
                    actual: oggi.getYear() + "-" + oggi.plusMonths( monthsToAdd: 1).getMonthValue() + "-" +
                    LocalDate.parse(contratto.getProssimoPagamento(), formatter).getDayOfMonth());
            }
            else {
                assertEquals(c.getProssimoPagamento(), actual: "");
            }
        }
        // verifichiamo che dopo l'aggiunta del contratto l'immobile sia affittato
        immobili = this.controller.getAllImmobili();
        for(Immobile i: immobili) {
            if(Objects.equals(i.getComune(), this.immobile.getComune()) && Objects.equals(i.getIndirizzo(), this.immobile.getIndirizzo()) &&
                Objects.equals(i.getnCivico(), this.immobile.getnCivico()) && i.getSubalterno() == this.immobile.getSubalterno()) {
                assertTrue(i.isAffittato());
            }
        }
    }
}

```

Figura 28: codice del metodo testAggiungi