Progetto ingegneria del software

- Aboufaris Yassine 727829
- Arcaini Matteo 729794
- Piantoni Dario 730057

UNIVERSITÀ DEGLI STUDI DI BRESCIA

INDICE

Principio di separazione modello-vista

Abbiamo organizzato il modello-vista, in 3 classi:

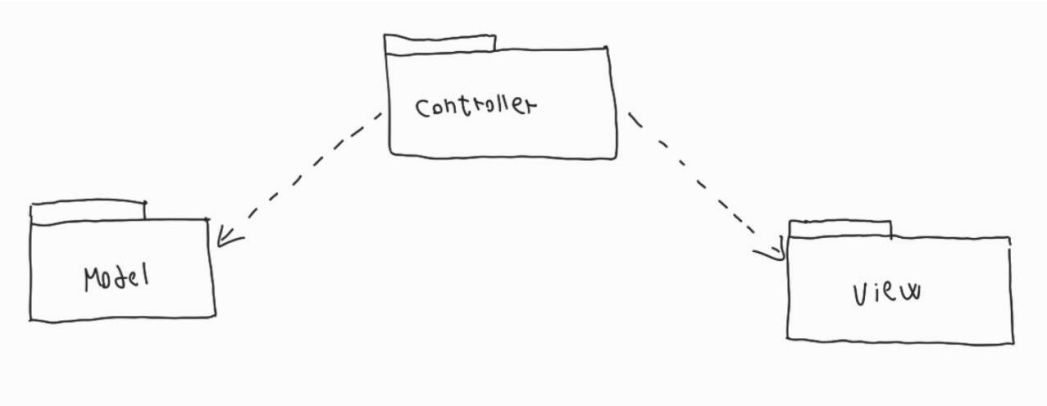
- **Model:** rappresenta i dati e il comportamento del sistema

-Controller: fa da tramite tra Model e View trasformando le richieste dell'utente sulla View in richieste al model

-View: rappresenta il modo in cui l'utente interagisce col sistema

Quando il Model ha bisogno di un' informazione, il Controller la chiede alla View (che la richiede all'utente) e la passa al Model.

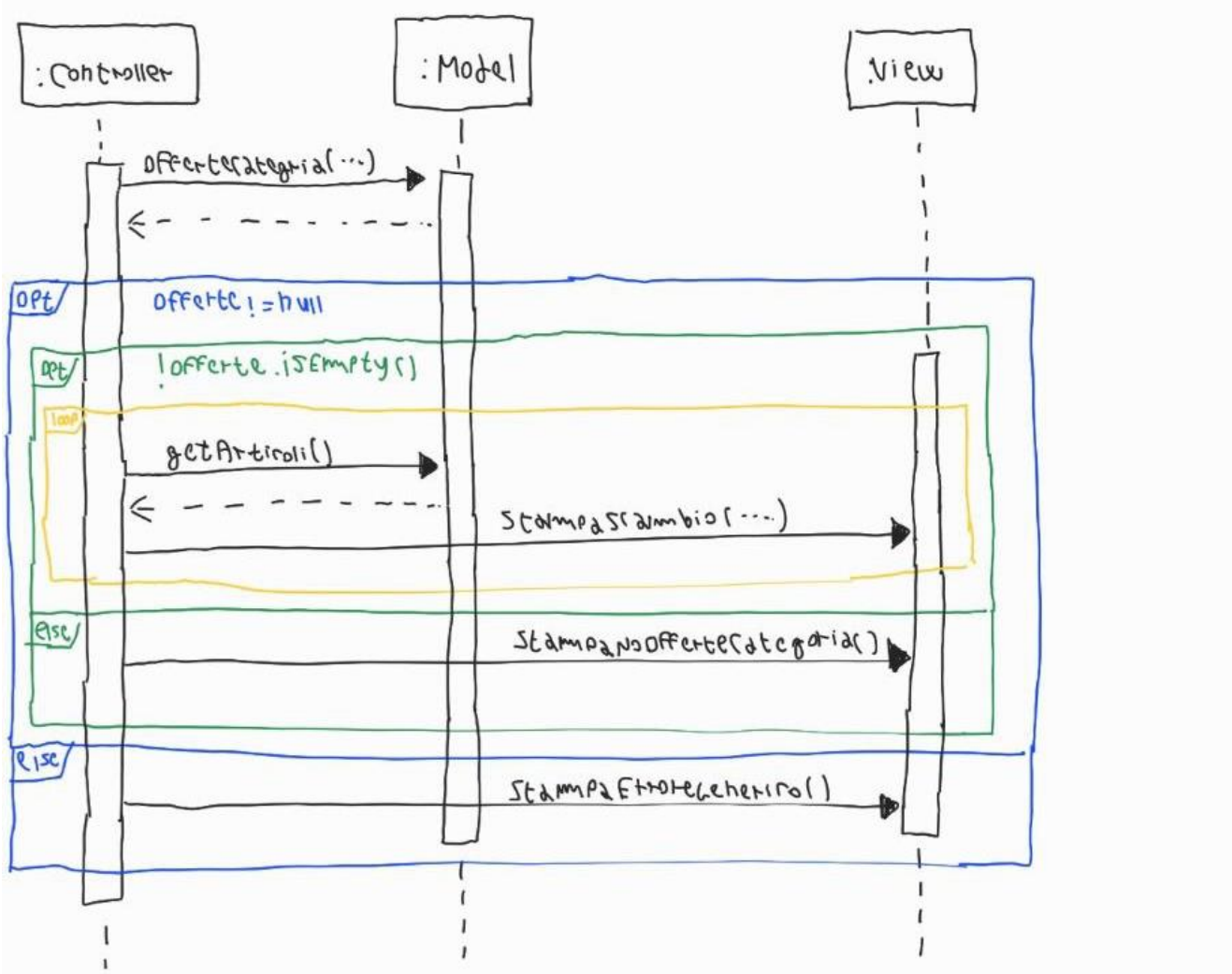
UML Package



La view e il model comunicano tra di loro solo tramite il controller.

Il controller richiama i metodi del model su richiesta della view

SSD



- Un esempio di interazione è quella mostrata nel diagramma a destra dove il controller richiede al model le offerte relative ad una categoria e gli articoli
- Una volta ottenuti i dati il controller li passa ai metodi della view che li visualizza a video
- Il model contiene la logica di business (logica di funzionamento del sistema)
- Attualmente la view contiene i metodi per visualizzare su console (in un futuro potrebbe essere sostituita da una GUI più sofisticata)

GRASP Information Expert

Information Expert afferma di assegnare le responsabilità alle classi che possiedono le relative informazioni necessarie per soddisfarle

Questo pattern è stato applicato fin dalla prima versione del progetto, e man mano che venivano aggiunte nuove funzionalità abbiamo ragionato secondo questa logica per assegnarla alla classe più appropriata.

Un esempio è la classe Articolo che siccome contiene tutte le informazioni necessarie relative all'articolo si occupa:

- di elaborare lo stato attuale dell' offerta;
- Cambiare lo stato dell'articolo;

```java
/**
 * Metodo che seleziona dallo storico l'offerta attualmente in corso
 * @return ritorna un oggetto Offerta
 *
 * @precondition storicoOfferte!=null
 * @postcondition storicoOfferte.size()==storicoOfferte'.size()
 */
public ArticoloState statoOffertaAttuale() {
    return this.storicoOfferte.get(this.storicoOfferte.size()-1).getOfferta();
}
```

Il metodo **StatoOffertaAttuale** usa le informazioni contenute nell'array StoricoOfferte per ricavare l'ultimo stato noto;

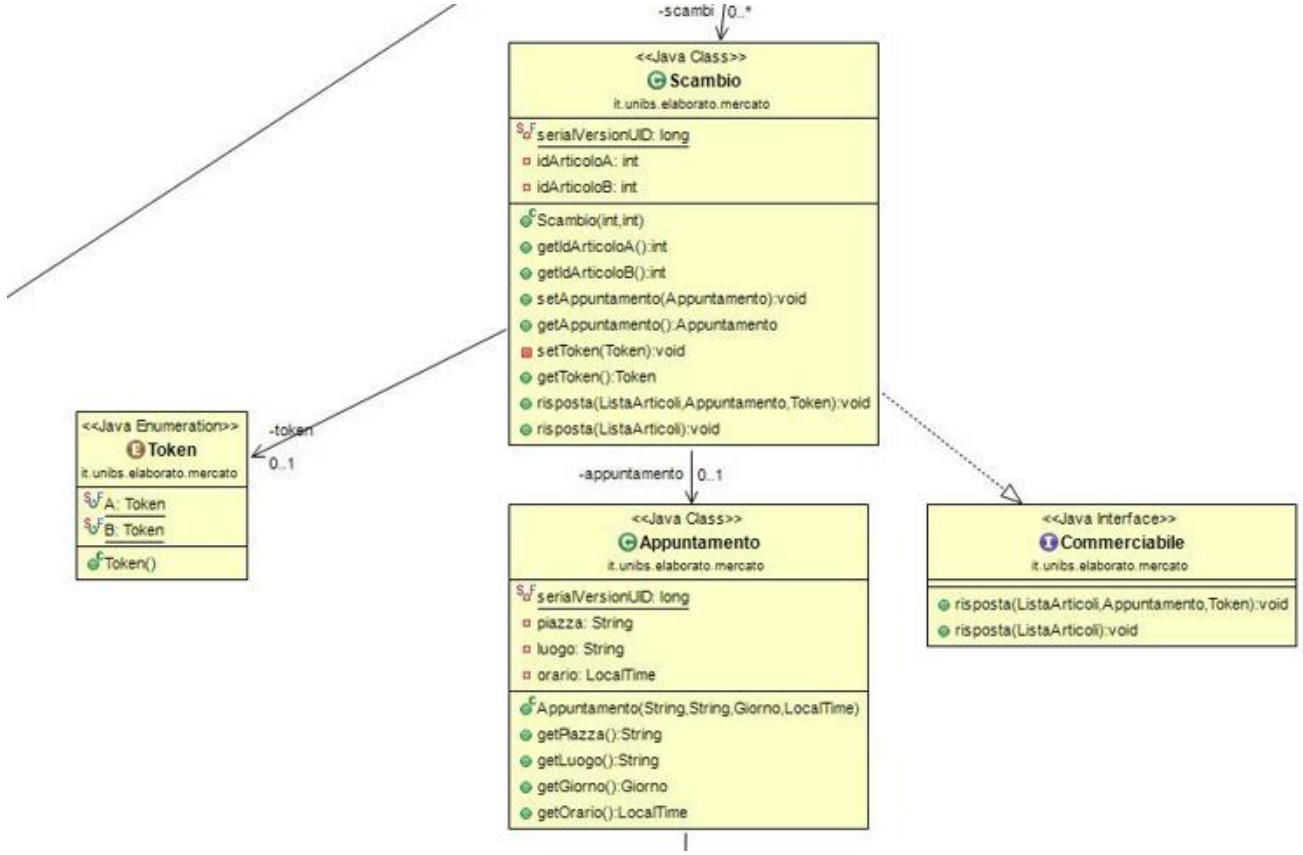
Il metodo **CambiaStato** si occupa di aggiornare lo stato dell'offerta relativa all'articolo.

```java
/**Metodo che cambia lo stato dell'articolo aggiungendolo in coda all'array
 * @param state stato da cambiare
 */
public void cambiaStato(ArticoloState state) {
    this.state = state;
    storicoOfferte.add(new StatoArticolo(LocalDateTime.now(), state));
}
```

GRASP Low coupling

Assegna le responsabilità affinchè si abbia accoppiamento basso e un basso impatto dei cambiamenti in modo da facilitare il riuso

- Un esempio di low coupling è quello della classe Scambio che ha un basso accoppiamento con le altre classi
- Infatti ListaScambi aggrega tanti oggetti scambio e Scambio utilizza Appuntamento e Token.



Inoltre a Scambio è stato introdotta l'interfaccia Commerciabile.

Interfaccia **Commerciabile** è stata usata nella classe Scambio, ed è utile per future espansioni dove si potrebbe pensare di aggiungere le vendite degli articoli in alternativa ai normali scambi, in questo caso si potrebbe pensare che le risposte conterranno anche la contrattazione del prezzo.

```
public interface Commerciabile {
    /** Metodo che in seguito a una proposta di scambio imposta
     * gli stati delle offerte relative agli articoli in scambio
    **/
    public void risposta(ListaArticoli articoli, Appuntamento a,Token t);

    /** Metodo che modifica lo stato delle offerte degli articoli in scambio in chiuse **/
    public void risposta(ListaArticoli articoli);
}
```

SOLID: Single responsability

Alta Coesione: ogni classe dovrebbe avere una sola responsabilità

- E’ stata aggiunta la classe **Scadenza** per raggruppare tutti i metodo relativi al controllo delle scadenze

In questo modo **Scambio** e **ListaScambio** ora hanno una sola responsabilità

Questa classe contiene due metodi:

- getDataScandenza(..) ottiene la data di scadenza
- controllaScadenza(..) elimina gli articoli che hanno superato la data di scadenza

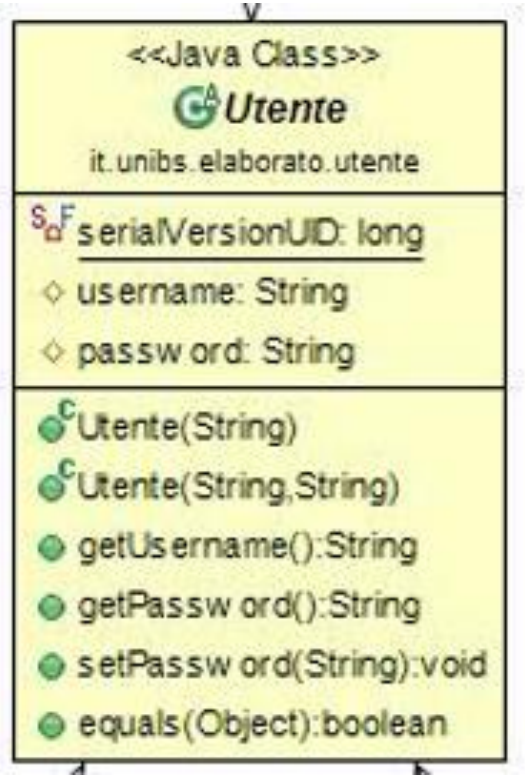
In precedenza questi metodi erano nelle classi **Scambio** e **ListaScambio** e queste due classi avevano più di una responsabilità violando cosi il pattern.

SOLID Liskov substitution

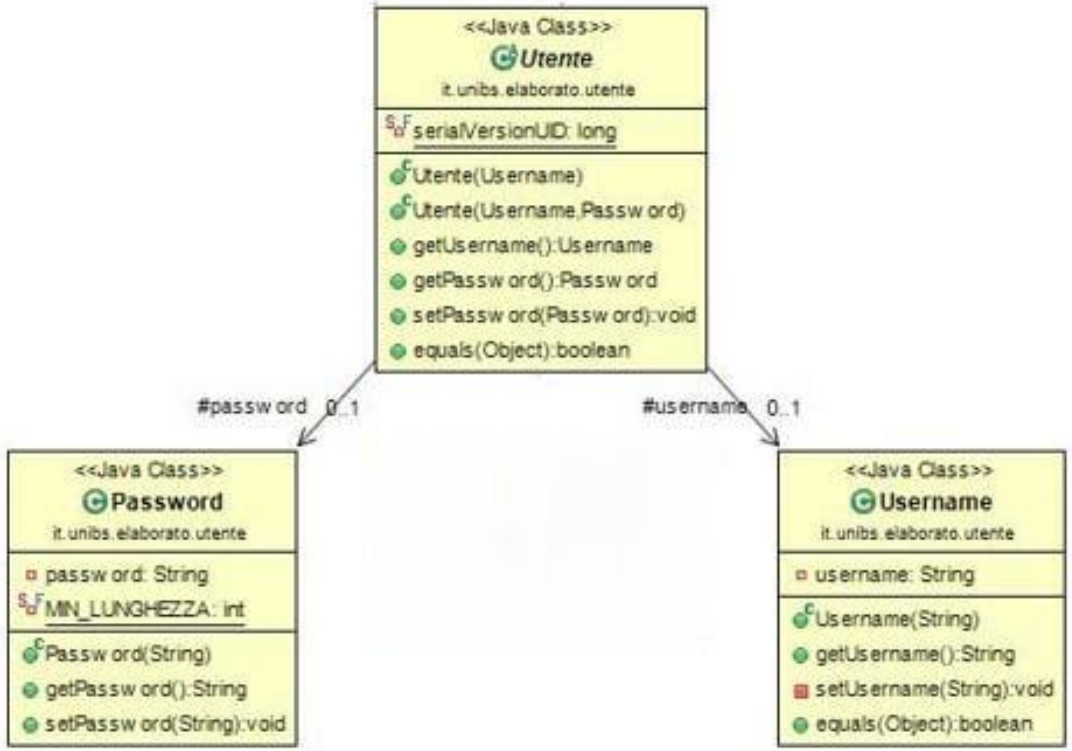
Nella classe **Utente** al posto di username e password precedentemente impostate come stringhe, sono state aggiunte le classi Username e Password.

Ciò ha permesso di inserire specifiche secondo le quali non tutti i valori del tipo String sono username o password validi.

Prima


<<Java Class>>
Utente
it.unibs.elaborato.utente
serialVersionUID: long
username: String
passw ord: String
Utente(String)
Utente(String,String)
getUsername():String
getPassw ord():String
setPassw ord(String):void
equals(Object):boolean

Dopo

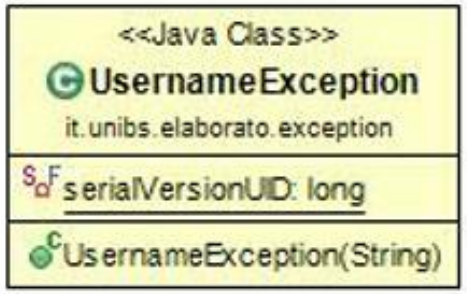

<<Java Class>>
Utente
it.unibs.elaborato.utente
serialVersionUID: long
Utente(Username)
Utente(Username,Passw ord)
getUsername():Username
getPassw ord():Passw ord
setPassw ord(Passw ord):void
equals(Object):boolean
#passw ord 0..1
#username 0..1
<<Java Class>>
Password
it.unibs.elaborato.utente
passw ord: String
MIN_LUNGHEZZA: int
Passw ord(String)
getPassw ord():String
setPassw ord(String):void
<<Java Class>>
Username
it.unibs.elaborato.utente
username: String
Username(String)
getUsername():String
setUsername(String):void
equals(Object):boolean

GLI USERNAME VALIDI
INIZIANO PER LETTERA

LE PASSWORD VALIDE HANNO
ALMENO 5 CARATTERI

Nel caso non siano rispettate vengono lanciate delle apposite eccezioni:

- Username exception: serve per gestire le eccezioni degli username
- Password exception: serve per gestire le eccezioni delle password



GoF State

Abbiamo applicato il pattern **State** per quanto riguarda lo stato dell'articolo in quanto il comportamento dell'oggetto dipende strettamente dallo stato attuale.

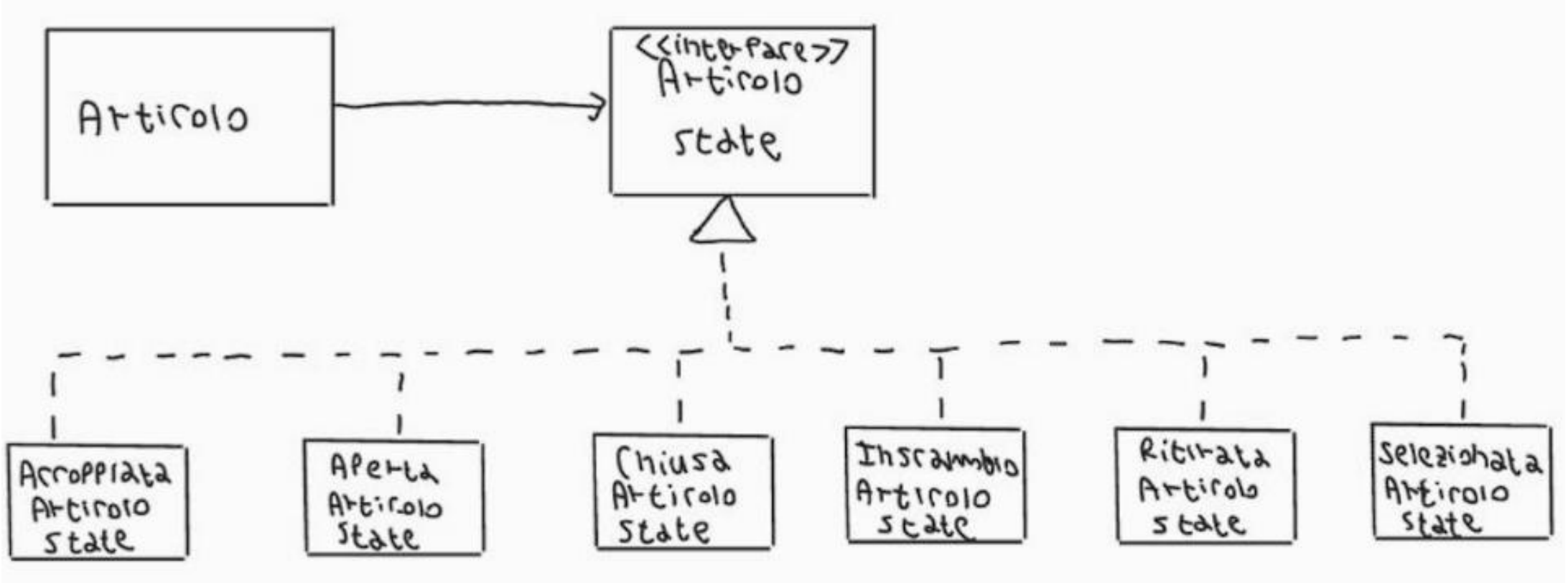
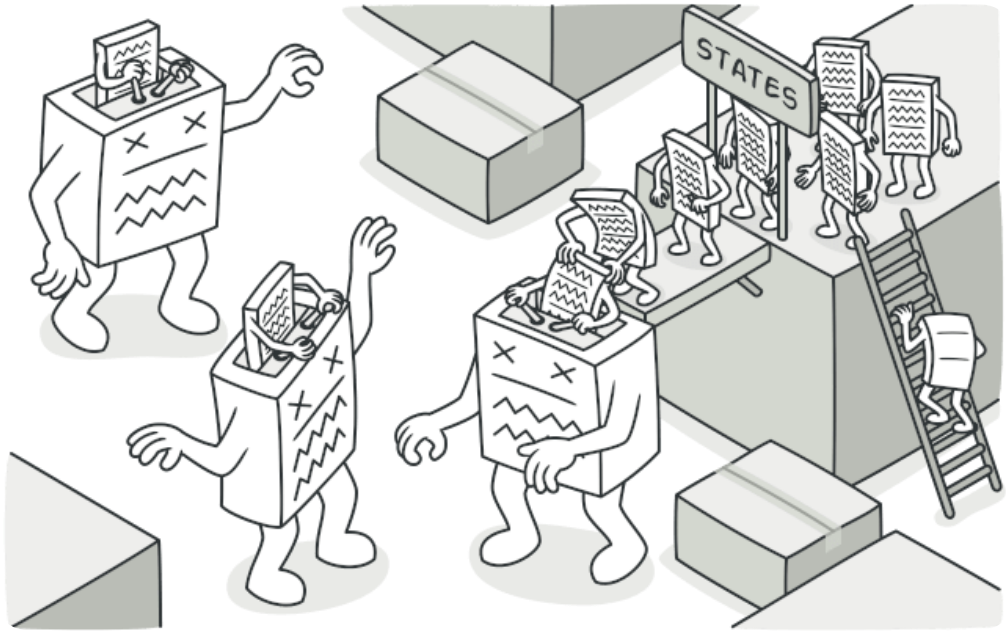


Diagramma UML delle classi pattern State applicato nel progetto



Sono state create tante classi, tante quanto sono gli stati dell’articolo.

Abbiamo creato un’interfaccia comune a tutte indicando i metodi per cambiare lo stato dell’articolo.

Ogni **Stato** implementa l'interfaccia comune **ArticoloState** e contiene i metodi per passare da uno stato all'altro

- In precedenza gli stati di un Articolo erano modellati con un enumerativo Offerta che conteneva tutti gli stati possibili.
- Ora grazie all'applicazione del pattern è più semplice aggiungere un nuovo stato
- Per esempio si potrebbe pensare di aggiungere un nuovo stato IN_VENDITA nell'ottica sempre di prevedere scambi con compensi economici

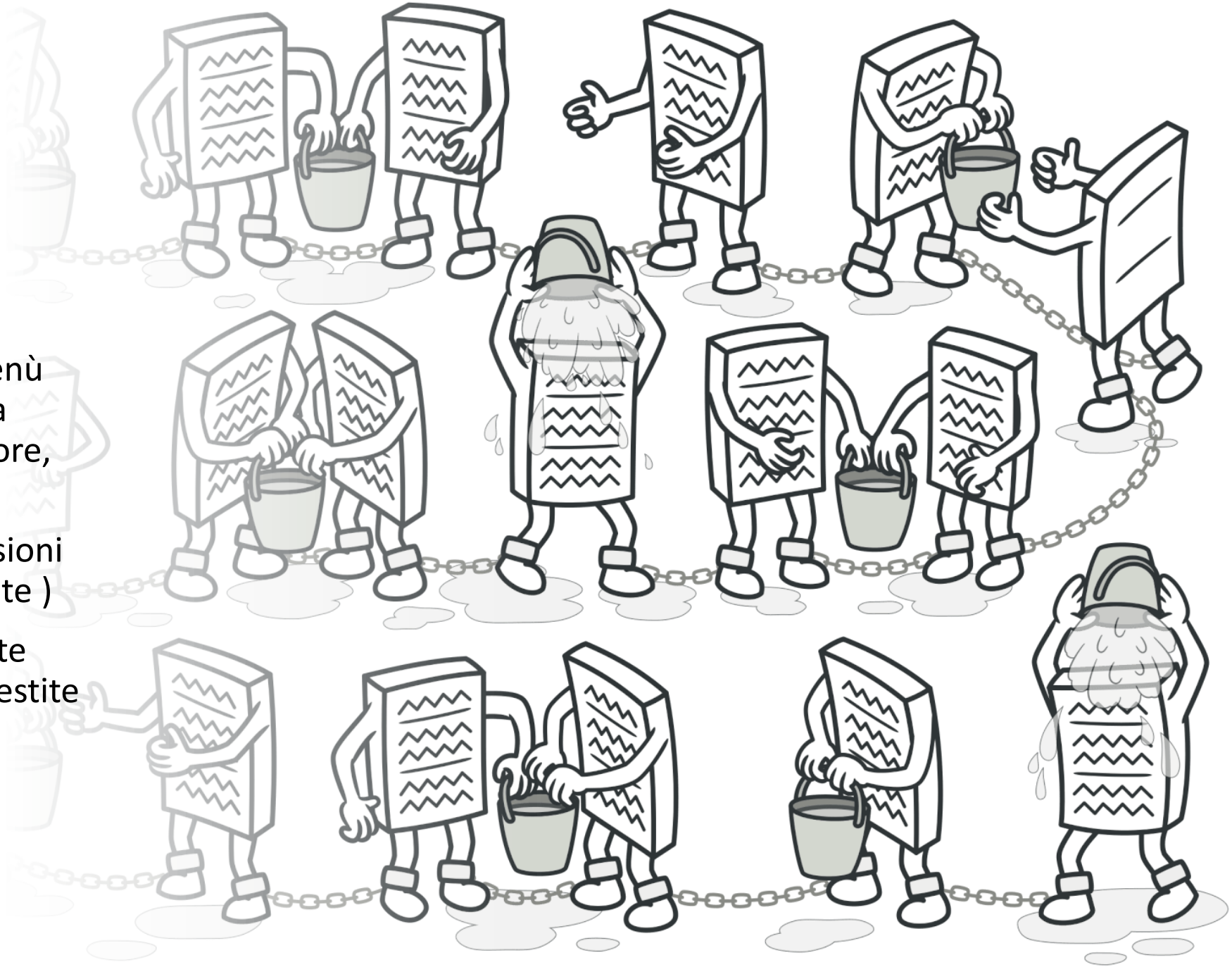
GoF Chain of Responsibility

Sostituisce instanceof nel codice per la selezione del menù fruitore o configuratore

Passo 1: creo un’interfaccia UtenteRenderer che definisce il metodo render per la gestione delle richieste

Passo 2: abbiamo creato delle sottoclassi per gestire le varie tipologie di utenti, che possono interagire con il nostro sistema creando una catena di gestori, l’ultimo gestore è il DefaultRenderer

- Motivo: selezionare il menù adeguato a seconda della tipologia di utente (Fruitore, Configuratore,…)
- Vantaggi: Possibili espansioni future (nuovi tipi di utente)
- Svantaggi: Alcune richieste potrebbero non essere gestite



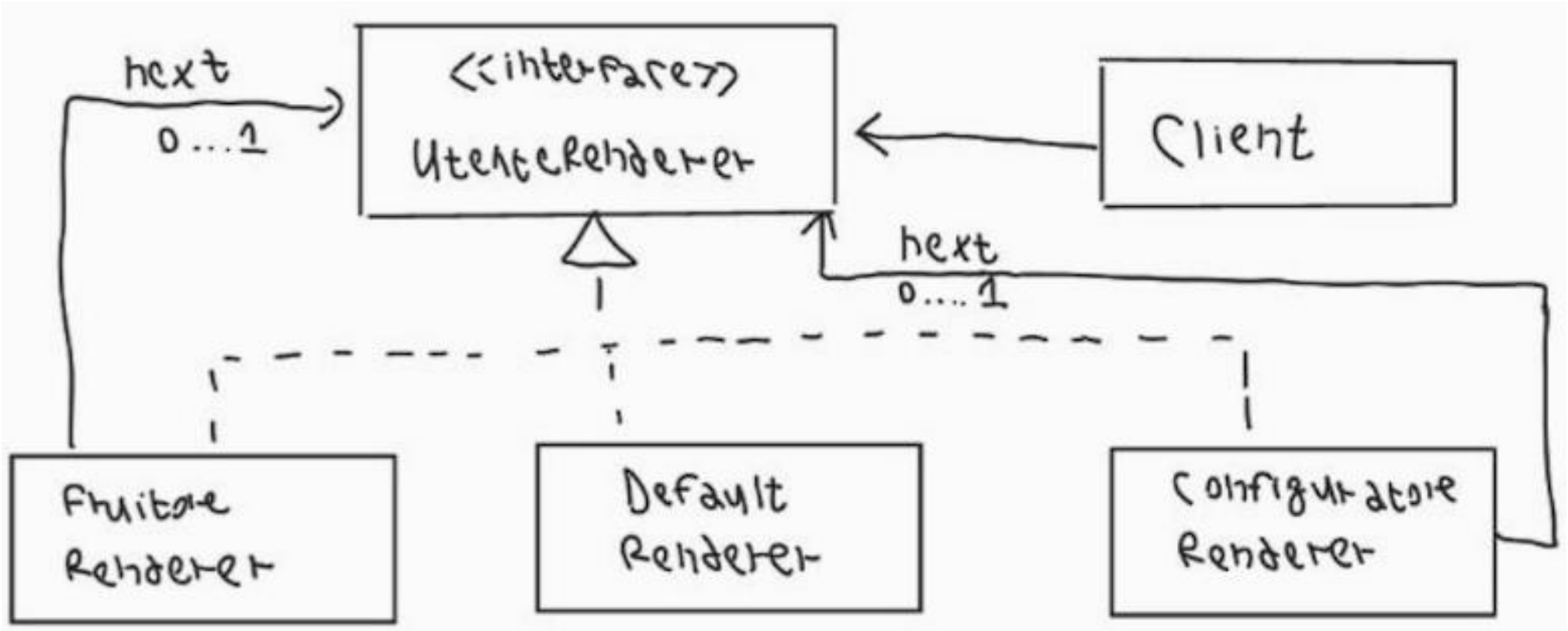


Diagramma UML delle classi pattern Chain of Responsability applicato nel progetto

Test black box

Effettuati verificando che a input significativi corrispondessero output attesi.

Applicati sulla validità relativa agli username e password.

Esempi di test blackbox relativi alla registrazione (valida,non valida)

```java
@Test
public void TestUsernameValido() {
    boolean aggiunto = false;
    Model model=new Model();
    Controller controller = new Controller(model, new View());
    controller.carica();

    try {
        aggiunto = model.registrazione("dario", "abcde", "abcde");
    } catch (UsernameException e) {
    } catch (PasswordException e) {
    }
    assertTrue(aggiunto);
}
```

valida

Verifica che l'username sintatticamente corretto sia aggiunto nel sistema

```java
@Test
public void TestUsernameNonValido() {
    boolean aggiunto = false;
    Model model=new Model();
    Controller controller = new Controller(model, new View());
    controller.carica();

    try {
        aggiunto = model.registrazione("1dario", "abcde", "abcde");
    } catch (UsernameException e) {
    } catch (PasswordException e) {
    }
    assertFalse(aggiunto);
}
```

non valida

Verifica che l' username sintatticamente scorretto «1dario» invalidi la registrazione

Esempi di test Black box l’inserimento dei campi in una categoria

Verifica che le categorie aggiunte ereditino i campi dal padre

```
@Test
public void testBlackBoxCampiCategoria() throws UsernameException, PasswordException {
    ListaCategorie categorie = new ListaCategorie();
    ArrayList<Campo> campiPadre=new ArrayList<Campo>();
    ArrayList<Campo> campiFoglia=new ArrayList<Campo>();

    campiPadre.add(new Campo("Campo di prova", TipoDato.DECIMALE, true));
    campiFoglia.add(new Campo("Campo di prova 2", TipoDato.DECIMALE, true));

    Categoria c = new Categoria("Categoria Test", "Categoria di prova", campiPadre, null);
    categorie.aggiungi(c);

    Categoria foglia = new Categoria("Foglia", null, campiFoglia, c);
    categorie.aggiungi(foglia);

    assertEquals(categorie.getCampiCategoria(foglia).size(), 4);
}
```

Extract Method

- Durante la programmazione, abbiamo avuto metodi troppo lunghi.
- Abbiamo deciso di rifattorizzare utilizzando Extract Method.

Passi effettuati

- Abbiamo creato un nuovo metodo
- Copiato il codice estratto dal metodo sorgente al nuovo metodo
- Passato i parametri necessari al metodo
- Sostituito il codice estratto con una chiamata al nuovo metodo

PRIMA

```
public void login() throws PasswordException {
    boolean login = false;
    Utente utenteLogin = null;

    do {
        utenteLogin = model.controllaLogin(view.getUsername(), view.getPassword());
        if (utenteLogin != null) {
            if (model.isPrimoAccesso(utenteLogin)) {
                boolean passwordCambiata;
                do {
                    view.stampaPrimoAccesso();
                    passwordCambiata = model.cambiaPassword(utenteLogin.getUsername(), view.getPassword(),
                            view.getConfermaPassword());
                    if (passwordCambiata) {
                        view.stampaBenvenuto(utenteLogin.getUsername().getUsername());
                        login = true;
                    } else
                        view.stampaPasswordNonCoincidenti();
                } while (!passwordCambiata);
            }else {
                view.stampaBenvenuto(utenteLogin.getUsername().getUsername());
                login = true;
            }
        } else
            view.stampaLoginNonRiuscito();
    } while (!login);

    int risposta;
    switch (model.scegliMenuUtente(utenteLogin)) {
    case 1:
        do {
            risposta = view.menuConfiguratore();
            menuConfiguratore(risposta);
        } while (risposta != ESCI);

        break;
    case 2:
        do {
            risposta = view.menuFruitore();
            menuFruitore(utenteLogin, risposta);
        } while (risposta != ESCI);
        break;
    }
}
```

DOPO

```
public void login() throws PasswordException {
    boolean login = false;
    Utente utenteLogin = null;

    do {
        utenteLogin = model.controllaLogin(view.getUsername(), view.getPassword());
        if (utenteLogin != null) {
            if (model.isPrimoAccesso(utenteLogin)) {
                boolean passwordCambiata;
                do {
                    view.stampaPrimoAccesso();
                    passwordCambiata = model.cambiaPassword(utenteLogin.getUsername(), view.getPassword(),
                            view.getConfermaPassword());
                    if (passwordCambiata) {
                        view.stampaBenvenuto(utenteLogin.getUsername().getUsername());
                        login = true;
                    } else
                        view.stampaPasswordNonCoincidenti();
                } while (!passwordCambiata);
            }else {
                view.stampaBenvenuto(utenteLogin.getUsername().getUsername());
                login = true;
            }
        } else
            view.stampaLoginNonRiuscito();
    } while (!login);

    sceltaMenu(utenteLogin);
}

/**metodo per scegliere se invocare un menù di un configuratore o di un fruitore
 * @param utenteLogin utente loggato nel sistema in quel momento
 */
public void sceltaMenu(Utente utenteLogin) {
    int risposta;
    switch (model.scegliMenuUtente(utenteLogin)) {
    case 1:
        do {
            risposta = view.menuConfiguratore();
            menuConfiguratore(risposta);
        } while (risposta != ESCI);

        break;
    case 2:
        do {
            risposta = view.menuFruitore();
            menuFruitore(utenteLogin, risposta);
        } while (risposta != ESCI);
        break;
    }
}
```

FINE

Grazie per l’attenzione