UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Relazione Elaborato SWE

Chisci Marco

April 2021

1 Introduzione e Requisiti

Questa relazione tratta della definizione di un problema di dominio, della sua progettazione, la successiva implementazione in java ed infine i suoi test.
Il problema risolto è la gestione di un ipotetico ristorante.

1.1 Problem Statement

Nel ristorante è presente un manager. Il manager gestisce l'inizio della giornata lavorativa e l'arrivo dei clienti. Ha la capacità di assegnare clienti ai suoi camerieri e tenere traccia del numero totale di clienti presenti nel ristorante. Infine può terminare la giornata lavorativa e decidere di non ammettere altri clienti notificando l'inizio della mattina successiva alla dispensa quando tutti i clienti sono stati serviti.
Ogni cliente ha il proprio budget e il ristorante ha una cassa che contiene i soldi disponibili.
Ogni cameriere ha il compito di occuparsi dei propri clienti (ovviamente ogni cameriere può gestire solo fino ad un certo numero di clienti contemporaneamente). Il cameriere consiglia al cliente cosa ordinare dal menù: se riesce a trovare una ricetta adatta il cliente paga, altrimenti il cliente se ne va senza pagare. Inoltre il compito del primo cameriere è quello di prelevare dalla cassa il budget giornaliero per le spese e consegnarlo alla dispensa.
Il menù del ristorante è composto da ricette a loro volta composte da ingredienti. Ovviamente sarà possibile cambiare costo alle ricette e agli ingredienti.
La dispensa gestisce gli ingredienti disponibili e la lista degli ingredienti necessari/mancanti.
Se è mattina (ovvero dopo il termine di una giornata lavorativa e prima dell'inizio di un'altra giornata lavorativa) attraverso la dispensa si potrà andare al mercato per comprare gli ingredienti mancanti utilizzando il proprio budget giornaliero (il budget non speso i giorni precedenti è cumulabile).
Il mercato ha una lista di ingredienti disponibili.

1.2 Use Case Diagram

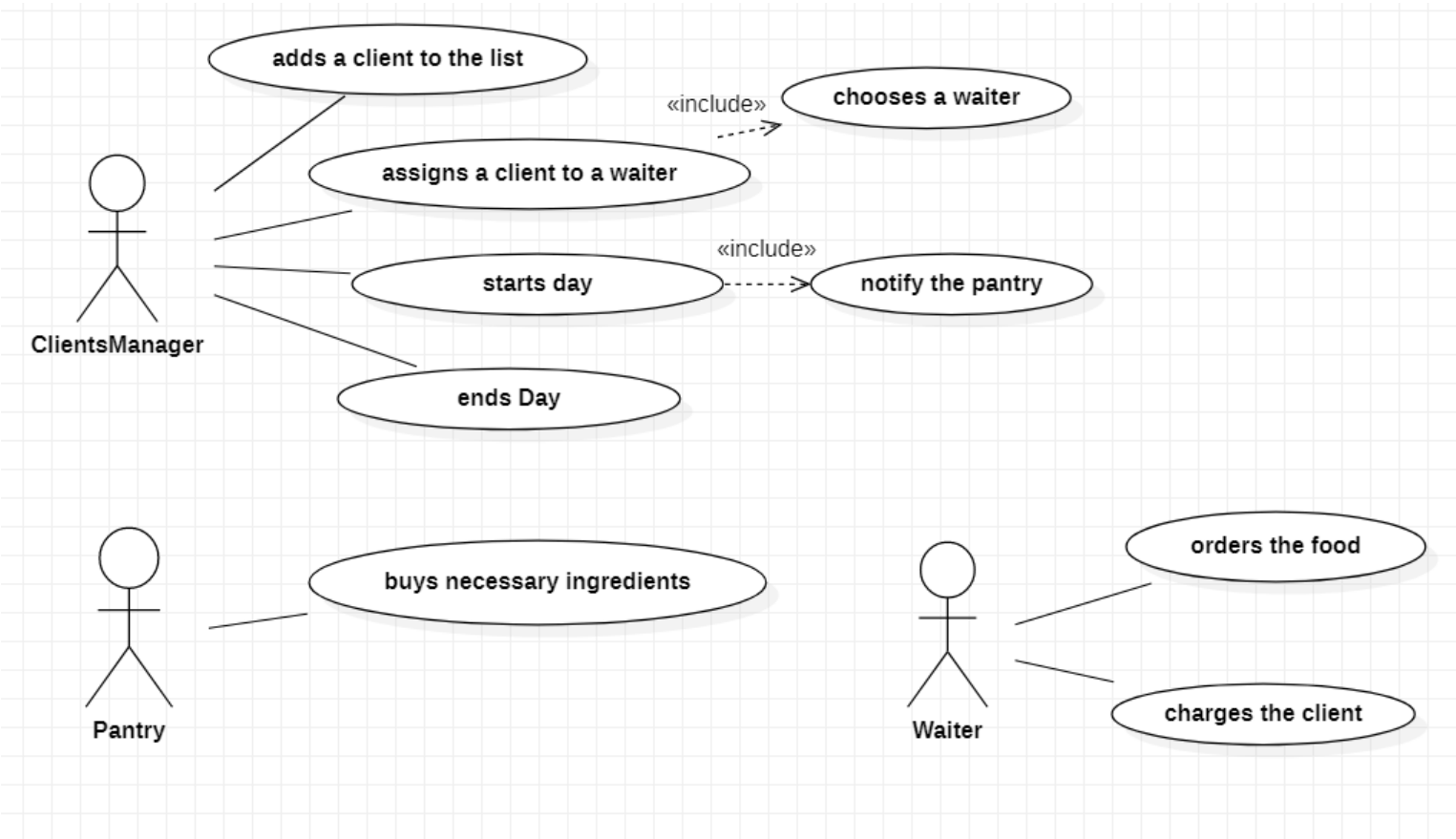

Figure 1: Use case Diagram

2 Progettazione

2.1 Class Diagram

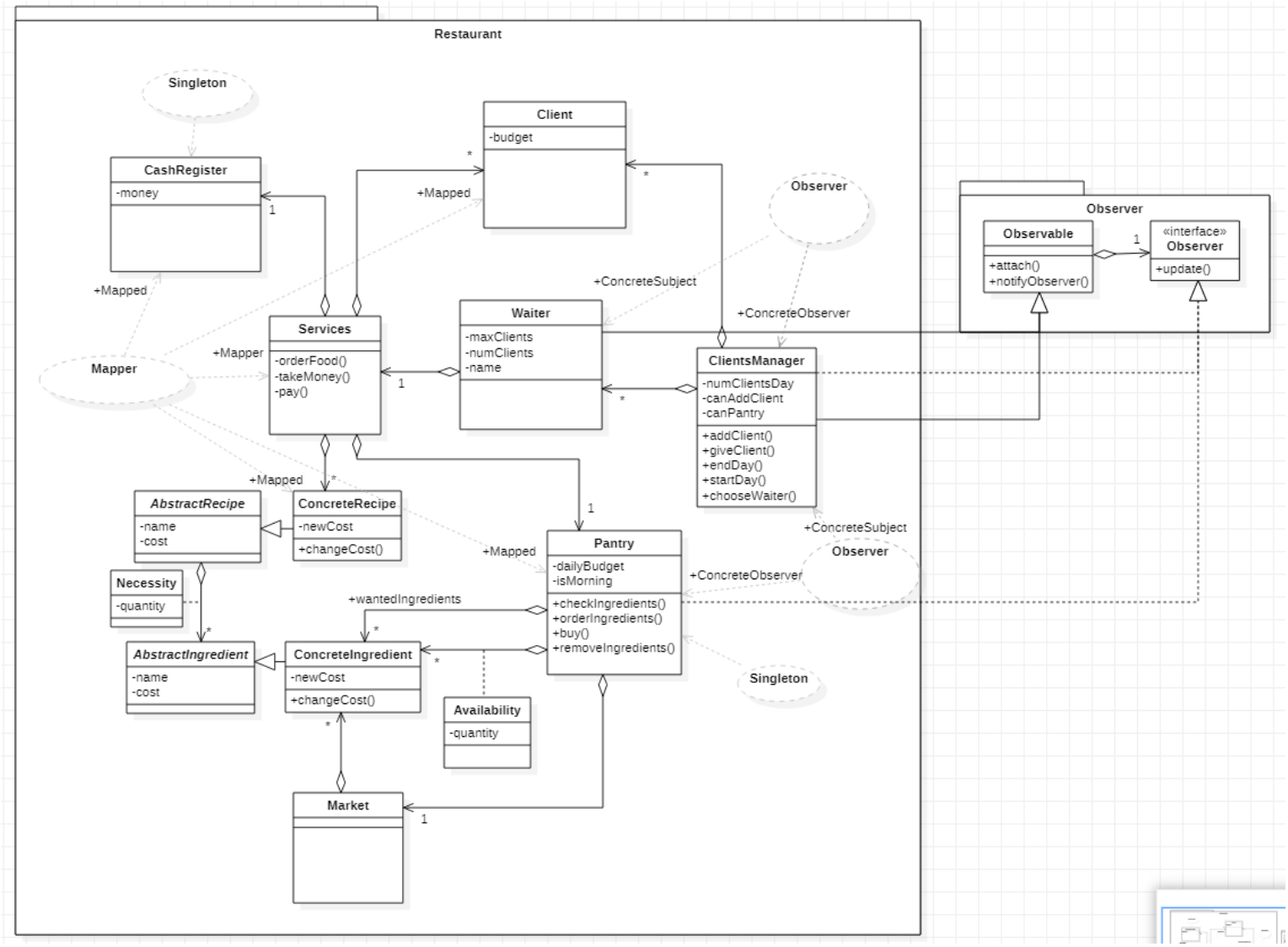

Figure 2: Class Diagram

2.2 Sequence Diagram

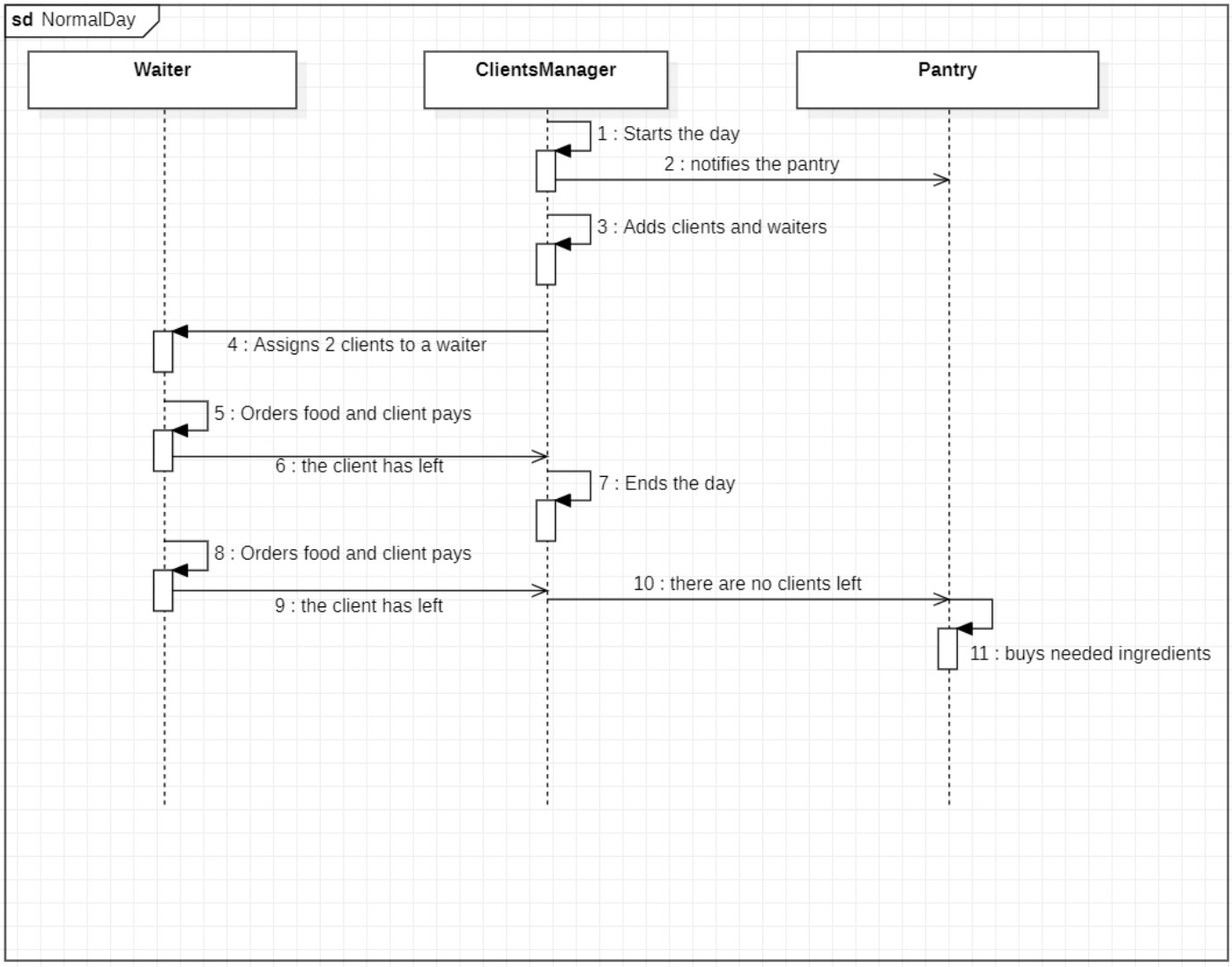

Figure 3: Sequence Diagram

In questo sequence diagram viene rappresentato il flusso di eventi di un giorno normale. Il manager inizia la giornata (notifica quindi alla dispensa che non è più mattina) e successivamente aggiunge dei clienti e dei camerieri. A questo punto assegna i clienti ai camerieri. I camerieri si occupano degli ordini dei clienti. Nel frattempo il ristorante decide di terminare la giornata e quando tutti i clienti rimasti finiscono di ordinare e pagare i camererieri notificano il manager che a sua volta notifica la dispensa dell'inizio della mattina successiva. A questo punto la dispensa può comprare gli ingredienti mancanti al mercato.

2.3 Mockup

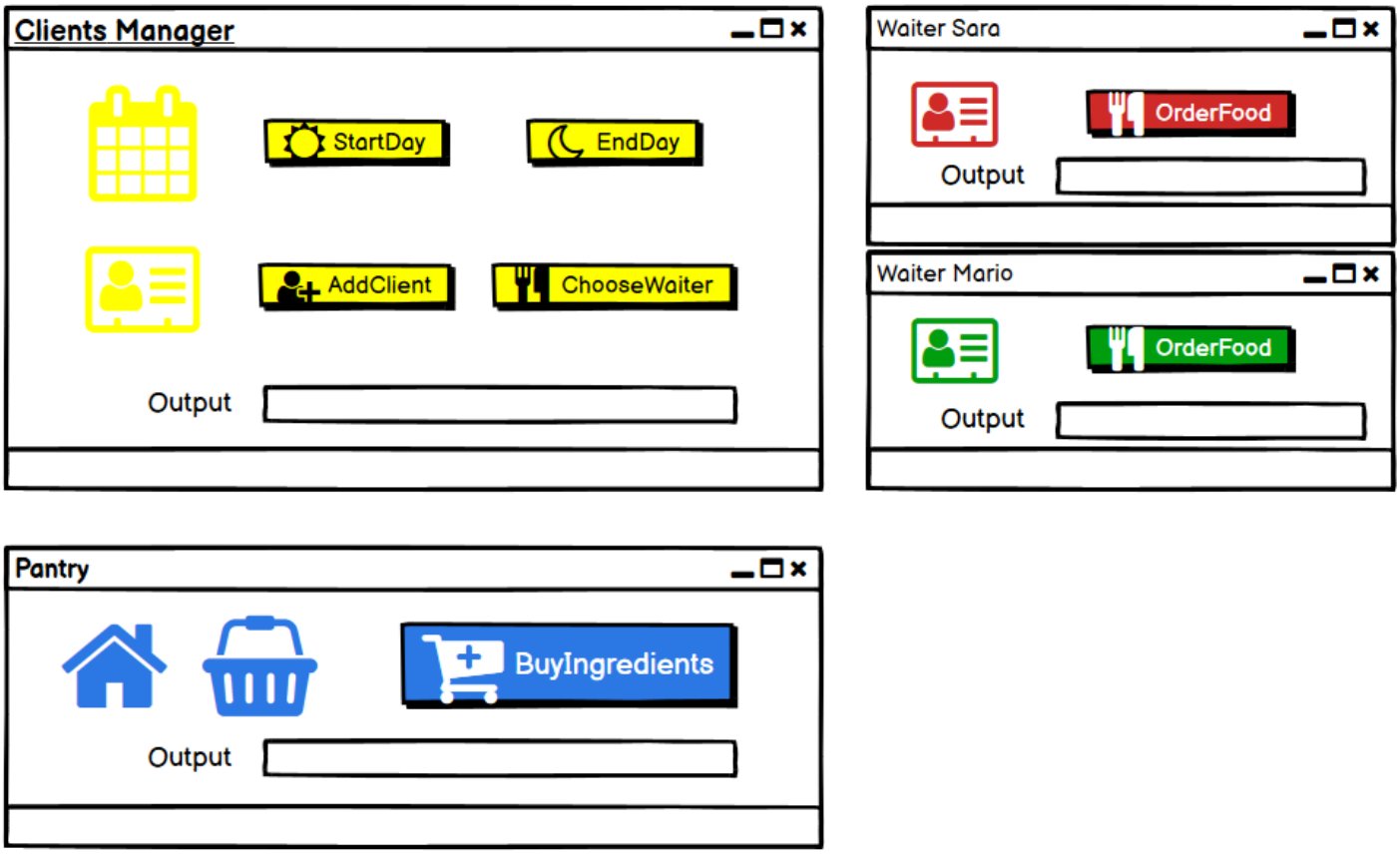

Figure 4: Mockup Restaurant

Questa possibile interfaccia include 3 tipi di finestra: quella del manager, quella del cameriere e quella della dispensa.
Ogni finestra contiene dei pulsanti attraverso i quali richiamare le funzioni omonime ed un display per il possibile output generato dalla funzione.

3 Implementazione

3.1 Classi

3.1.1 ClientsManager

La classe ClientsManager si occupa della gestione dei clienti. Per fare questo necessita di due liste: una contenente i camerieri a sua disposizione e una contenente i clienti ancora non assegnati a nessun cameriere. Inoltre implementa l'interfaccia Observer e estente la classe astratta Observable.
I compiti del ClientsManager sono:

1. iniziare la giornata lavorativa con startDay() se possiede almeno 1 cameriere: rende positivo il canAddClient (così è possibile aggiungere clienti) e, attraverso il primo cameriere, consegna alla dispensa il suo budget giornaliero notificandole la fine della mattina(isMorning=false).

2. aggiungere clienti alla propria lista di attesa con addClient() e tenere conto di quanti clienti siano ancora presenti nel ristorante.

3. assegnare un cliente ad un cameriere attraverso chooseWaiter() indicando il nome del cameriere desiderato. Con un ciclo while viene verificata la presenza del cameriere. Se il cameriere esiste e non è troppo occupato attraverso giveClient() gli viene assegnato il cliente, altrimenti comunica l'assenza del cameriere o la sua eventuale incapacità di gestire altri clienti.

```
public void chooseWaiter(String name) {
    boolean isHere=false;
    int i=0;
    while(i<waiters.size()&&isHere==false) {
        if((waiters.get(i).getName()).equals(name)) {
            isHere=true;
            giveClient(waiters.get(i));
        }
        i+=1;
    }
    if(isHere==false) {
        System.out.println("The waiter is not here");
    }
}
public void giveClient(Waiter waiter) {
    if(waiter.getNumClients()<waiter.getMaxClients()) {
        waiter.getServices().addClient(clients.get(0));
        waiter.setNumClients(waiter.getNumClients()+1);
        clients.remove(0);
    }
    else {
        System.out.println("This waiter already has too many clients");
    }
}
```

Figure 5: Metodo chooseWaiter() della classe ClientsManager

4. terminare la giornata lavorativa attraverso endDay(): fa diventare negativo il canAddClient e rende così impossibile aggiungere altri clienti. A questo punto diventerà mattina solo quando non ci saranno più clienti nel ristorante.

3.1.2 Waiter and Services

Il cameriere possiede un nome e tiene conto di quanti clienti sta gestendo (ha un numero massimo di clienti che può gestire contemporaneamente). Estende la classe astratta Observable. Inoltre può accedere a dei servizi offerti dalla classe Services (in questo modo divido le informazioni a diversi gradi di volatilità: i dati del cameriere probabilmente rimarranno statici mentre i servizi offerti possono cambiare nel tempo). Attraverso Services il cameriere ha a sua disposizione una lista di ricette (il menù), l'accesso alla cassa e alla dispensa ed una lista di clienti che sta gestendo.
I servizi a disposizione del cameriere sono:

1. consegnare il budget giornaliero dalla cassa alla dispensa con takeMoney() (in questo caso metà dei soldi in dispensa).

2. far pagare il conto della ricetta ordinata dal cliente con pay() aggiungendo soldi alla cassa e diminuendo il budget del cliente.

3. ordinare il cibo per il cliente con orderFood(): il cameriere sceglie in modo randomico (per simulare un input esterno) una ricetta tra quelle contenute nel suo menù, successivamente controlla che la ricetta sia adatta (ovvero il suo costo rientri nel budget del cliente con checkCost() e che ci siano abbastanza ingredienti in dispensa). Se la ricetta è adatta viene preparata (rimuovo gli ingredienti dalla dispensa) e il cliente, soddisfatto, paga il conto attraverso il cameriere con pay(). Se la ricetta adatta non viene trovata entro 3 tentativi il cliente si spazientisce e se ne va senza pagare.

```java
public boolean checkCost(Client client, ConcreteRecipe recipe) {
    boolean isEnough=false;
    if(client.getBugdet()>recipe.getCost()) {
        isEnough=true;
    }
    return isEnough;
}

public void orderFood() {
    if(clients.size()!=0) {
        Client currentClient=clients.remove(0);             //prende il primo cliente
        boolean chosen=false;
        int dissatisfaction=0;
        while(dissatisfaction<3&&chosen==false) {
            int rand= (int) (Math.random()*(menu.size()));
            if(pantry.checkIngredients(menu.get(rand).getIngredients())&&
                    this.checkCost(currentClient, menu.get(rand))) {
                chosen=true;
                pantry.removeIngredients(menu.get(rand).getIngredients());
                this.pay(menu.get(rand),currentClient);
                System.out.println("the client got his food");
            }
            else {
                dissatisfaction+=1;
            }
        }
        if(dissatisfaction==3) {
            System.out.println("the client went away");
        }
    }
    else {
        System.out.println("There are no clients left");
    }
```

Figure 6: Metodo orderFood() della classe Services

3.1.3 Client, CashRegister and Market

La classe Client rappresenta il cliente. In questa rappresentazione il cliente necessita solamente di un suo budget da spendere e dei metodi per utilizzarlo (getter and setter). Anche la classe CashRegister (che rappresenta la cassa) necessita solo di una variabile per tenere conto dei soldi che contiene. Deve però anche implementare il design patter Singleton per evitare l'esistenza di più casse diverse nello stesso ristorante data la presenza di più camerieri.
Infine anche la classe Market (il mercato) è molto semplice, possiede solamente una lista di ingredienti disponibili da acquistare.

3.1.4 Pantry

La classe Pantry rappresenta la dispensa e la sua gestione degli ingredienti per il ristorante. Poichè la dispensa deve rimanere unica implementa un Singleton. La dispensa contiene una mappa di ingredienti disponibili (le chiavi della mappa sono gli ingredienti ed i valori sono le quantità presenti in dispensa di tale ingrediente),una lista di ingredienti mancanti da comprare e un riferimento al mercato dove andare a fare compere. I compiti della dispensa sono:

1. rimuovere gli ingredienti utilizzati per preparare una ricetta dalla lista di ingredienti disponibili con removeIngredients().

2. ordinare gli ingredienti mancanti aggiungendoli alla lista apposita con orderIngredients().

3. controllare che gli ingredienti necessari a cucinare una ricetta siano disponibili nella dispensa con checkIngredients(): per ogni ingrediente della ricetta controlla che ci siano abbastanza ingredienti di quel tipo tra gli ingredienti disponibili. Se tutti gli ingredienti della ricetta sono presenti restituisce true, altrimenti restituisce false e aggiunge gli ingredienti mancanti alla lista apposita.

```
public boolean checkIngredients(Map<AbstractIngredient, Integer> neededIngredients) {
    boolean allPresent=true;
    for(AbstractIngredient key: neededIngredients.keySet()) {
        if((availableIngredients.containsKey(key)==false)
                ||(neededIngredients.get(key)>availableIngredients.get(key))) {
            allPresent=false;
            if(wantedIngredients.contains(key)==false) {
                this.orderIngredients(key);
                }
            }
        }
    return allPresent;
}
```

Figure 7: Metodo checkIngredients() della classe Pantry

4. comprare gli ingredienti mancanti al mercato con buy(): se è mattina (il ristorante è chiuso e non sta servendo clienti) per ogni elemento della lista di ingredienti mancanti verifica la sua presenza al mercato e che sia possibile acquistarne abbastanza (in questo caso 10) con il budget assegnato alla dispensa. Se entrambe queste condizioni sono soddisfatte acquista l'ingrediente (aumentandone la quantità se già presente in dispensa o aggiungendolo agli ingredienti disponibili se ancora non è presente) ovviamente diminuendo il proprio budget.

```java
public void buy() {
    if(isMorning==true) {
        for(AbstractIngredient wanted: wantedIngredients) {
            if(market.getAvailableIngredients().contains(wanted)){
                if(wanted.getCost()*10<dailyBudget) {
                    if(availableIngredients.containsKey(wanted)) {
                        availableIngredients.replace(wanted, availableIngredients.get(wanted)+10);
                    }
                    else {
                        availableIngredients.put(wanted, 10);
                    }
                    dailyBudget-=wanted.getCost()*10;
                }
            }
        }
    }
    else {
        System.out.println("It's not morning yet");
    }
}
```

Figure 8: Metodo buy() della classe Pantry

3.1.5 Recipe and Ingredient

Recipe and Ingredient rappresentano rispettavimante una ricetta e un ingrediente. Entrambi hanno una classe astratta con il loro nome e costo originale e una classe concreta che eridita nome e costo e che possiede un nuovo costo che può cambiare in caso di sconti o variazioni di prezzo future. In questo modo posso distinguere la ricetta/ingrediente originale (astratta) dalla ricetta/ingrediente utilizzata davvero (concreta) a cui posso modificare il costo.
La ricetta possiede una mappa di ingredienti necessari alla sua preparazione le cui chiavi sono gli ingredienti stessi e i cui valori sono le loro rispettive quantità.

3.2 Design Pattern

3.2.1 Observer

Il manager deve essere sempre aggiornato sul numero di clienti presenti nel ristorante, per questo deve essere avvisato dai camerieri ogni volta che un cliente lascia il ristorante. Inoltre il manager deve a sua volta avvisare la dispensa quando non ci sono più clienti nel ristorante ed è finito il giorno lavorativo poichè la dispensa deve essere a conoscenza del fatto che, essendo mattina, può comprare gli ingredienti mancanti.
Per questi motivi è necessario implementare il patter Observer:

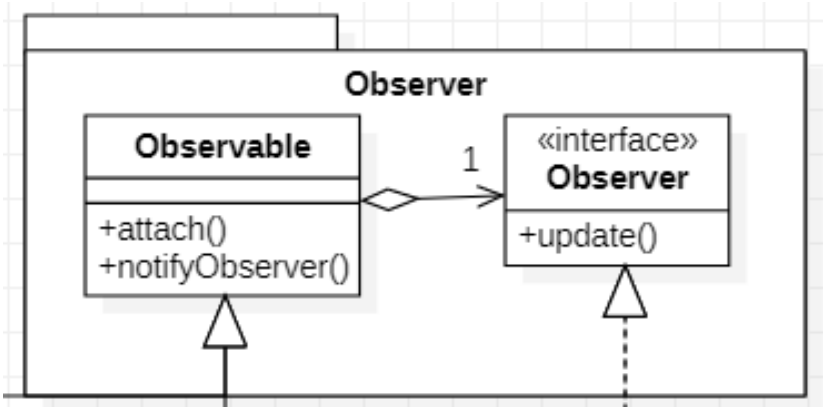

Figure 9: Class Diagram Observer

Il pattern observer è composto da una classe astratta e un'interfaccia. La classe astratta Observable verrà estesa da chi necessita di essere osservata e avrà un qualche dato da comunicare a chi osserva. L'interfaccia Observer verrà estesa da chi deve osservare e deve rimanere aggiornato sullo stato dall'altra classe. In questa implementazione attach() serve a "iscrivere" l'observer alla lista di chi vuole essere notificato e notifyObserver() serve a mandare un messaggio con le informazioni necessarie all'observer tramite l'invocazione di update().
Nel caso del ristorante Waiter estende Observable perchè deve mandare una notifica quando un cliente se ne va. ClientsManager implementa Observer perchè vuole essere aggiornato sullo stato dei clienti dei camerieri e estente Observable per notificare quando non ci sono più clienti nel ristorante. Infine Pantry implementa Observer perchè vuole essere aggiornato quando diventa mattina.
L'implemetazione di attach() è semplice: aggiunge un Observer alla lista di Observable. Il metodo viene utilizzato quando assegno un cameriere al manager (iscrivo il manager come observer del cameriere) e quando il primo giorno inizia e il manager comunica per la prima volta con la dispensa (iscrivo la dispensa come observer del manager).
NotifyObserver() è altrettanto semplice: chiama il metodo update del proprio Observer. Viene utilizzato quando un cameriere finisce di servire un cliente e quando il manager comunica di non avere più clienti dentro il ristorante.

```java
@Override
public void update() {
    this.numClientsDay-=1;
    if(this.numClientsDay==0) {
        this.notifyObserver();
        if(canAddClient) {
            this.endDay();
        }
    }
}
```

Figure 10: Metodo update() della classe ClientsManager

```java
@Override
public void update() {
    isMorning=true;
}
```

Figure 11: Metodo update() della classe Pantry

Infine update() ha due implementazioni: dentro il manager decrementa di uno il numero dei clienti presenti nel ristorante e controlla l'eventuale assenza di clienti rimanenti (in questo caso lo notifica alla dispensa e conclude la giornata lavorativa se non è già stata conclusa); dentro la dispensa semplicemente rende positivo isMorning.
In questo modo si crea una sincronizzazione: il numero di clienti nel ristorante è sempre pari alla somma del numero di clienti gestiti dai camerieri e del numero di clienti ancora non assegnati; è mattina sia per il manager che per la dispensa negli stessi momenti.

3.2.2 Singleton

All'interno del ristorante abbiamo degli elementi che devono necessariamente rimanere unici nonostante vengano usati da più utenti. In questo caso sia la cassa che la dispensa vengono utilizzati da più camerieri (chiaramente ogni cameriere utilizza la stessa dispensa e la stessa cassa).
Il design patter Singleton è utile in questi casi poichè impedisce a una classe di essere istanziata più volte attraverso un costruttore privato (quindi chiamabile solo all'interno della classe stessa) e un metodo getInstance() che restituisce una nuova istanza se e solo se non esistente già un'istanza di quella classe.
L'implementazione è semplice: attraverso una variabile privata "instance" (inizializzata a null) ed un costruttore privato si crea una situazione in cui l'unico modo per istanziare la classe (Pantry o CashRegister) è attraverso l'uso di getInstance(). getInstance() ha quindi il compito di instanziare la classe (utilizzando il costruttore privato) se instance è null (ovvero se ancora non è mai stata instanziata) o restituire l'instanza già creata altrimenti.
In questo modo l'instanza utilizzata da tutti i camerieri sarà necessariamente sempre la stessa.

```java
private Pantry(Market market, Map<AbstractIngredient, Integer> availableIngredients) {
    this.market=market;
    this.availableIngredients=availableIngredients;
}

public static Pantry getInstance(Market market, Map<AbstractIngredient, Integer> availableIngredients) {
    if (instance==null) {
        instance=new Pantry(market,availableIngredients);
    }
    return instance;
}
```

Figure 12: Singleton della classe Pantry

```java
private CashRegister(double money) {
    this.money=money;
}

public static CashRegister getInstance(double money){
    if(instance==null)
        instance= new CashRegister(money);
    return instance;
}
```

Figure 13: Singleton della classe CashRegister

3.2.3 Mapper

Molti elementi del ristorante devono comunicare tra loro: il cliente deve pagare alla cassa, la cassa deve dare il budget giornaliero alla dispensa e infine il cliente deve ordinare e quindi colloquiare con la dispensa ed il menù.
In questo caso è utile creare una classe che mappa le altri classi insieme gestendone le relazioni. Facendo così suddivido i dati statici delle classi mappate dalle loro relazioni esterne, probabilmente dinamiche e modificabili in futuro.
Per il ristorante l'elemento comune di tutte le classi citate sopra, e quindi il mappatore, è il cameriere attraverso i servizi che offre. Il cameriere si occupa quindi di: far pagare il cliente alla cassa, consegnare i soldi dalla cassa alla dispensa e ordinare il cibo del cliente relazionandosi con menù e dispensa.
Così facendo il cliente, la cassa, la dispensa e il menù contengono semplicemente i loro dati e le loro informazioni mentre i servizi del cameriere possono mutare ed evolversi senza dover modificare le altre classi.

4 Unit Testing

4.1 Test Recipe

La classe ConcreteRecipe viene instanziata a partire dall'inserimento di una mappa contenente gli ingredienti necessari e le loro quantità. Deve poi poter cambiare costo e restituire il costo aggiornato quando viene chiamato il metodo getCost().
Per questo il test effettuato controlla che, a partire da una mappa di ingredienti e quantità, la ricetta venga creata correttamente (la somma dei costi degli ingredienti sia corretta) e che sia possibile cambiare il costo della ricetta.

```
@Test
void testCreateRecipe() {
    assertEquals(14.9,torta.getCost());
}

@Test
void testChangeCost() {
    assertEquals(14.9, torta.getCost());
    torta.changeCost(13);
    assertEquals(13, torta.getCost());
}
```

Figure 14: Test della classe Recipe

4.2 Test Pantry

La classe Pantry deve riuscire a eseguire correttamente le funzioni descritte precedentemente nella sezione dedicata.
Per questo i test eseguiti sono i seguenti:

1. poichè la dispensa deve riuscire a rimuovere gli ingredienti il test semplicemente controlla che gli ingredienti rimossi non si trovino più tra quelli disponibili (originariamente sono presenti 3 "latte" in dispensa)

```
@Test
void testRemoveIngredients() {
    ingredients1.put(latte, 2);
    pantry.removeIngredients(ingredients1);
    assertEquals(1,pantry.getAvailableIngredients().get(latte));
}
```

Figure 15: Test del metodo removeIngredients() della classe Pantry

2. la dispensa deve ordinare gli ingredienti mancanti quindi questo semplice test asserisce che gli ingredienti ordinati si trovano nella lista di ingredienti mancanti.

```
@Test
void testOrderIngredients() {
    pantry.orderIngredients(uova);
    assertTrue(pantry.getWantedIngredients().contains(uova));
}
```

Figure 16: Test del metodo orderIngredients() della classe Pantry

3. la dispensa ha il compito di verificare la presenza di certi ingredienti(in questo momento in dispensa ci sono 5 "farina", 3 "latte" e 4 "uova"). Con questo test verifichiamo la reazione del metodo in 3 casi: quando gli ingredienti ci sono e deve restituire true, quando gli ingredienti non sono in dispensa e deve restituire false e quando gli ingredienti non sono sufficienti e deve restituire false. Inoltre controlla che nei due casi in cui non trova gli ingredienti questi siano stati inseriti nella lista di ingredienti mancanti.

```
@Test
void testCheckIngredients() {
    ingredients1.put(farina, 2);
    ingredients1.put(latte, 1);
    ingredients1.put(uova, 3);
    assertTrue(pantry.checkIngredients(ingredients1));
    ingredients2.put(zucchero, 2);
    assertFalse(pantry.checkIngredients(ingredients2));
    assertTrue(pantry.getWantedIngredients().contains(zucchero));
    ingredients3.put(farina, 30);
    assertFalse(pantry.checkIngredients(ingredients3));
    assertTrue(pantry.getWantedIngredients().contains(farina));
}
```

Figure 17: Test del metodo checkIngredients() della classe Pantry

4. andare a comprare gli ingredienti al mercato è compito della dispensa. In questo test verifichiamo la correttezza del metodo buy() osservandone il comportamento in due diversi casi.
   Nel primo caso non è ancora mattina quindi non compra niente (stampa un messaggio di errore)
   Nel secondo caso, dato un budget e una lista di ingredienti da comprare, chiama il metodo buy() e successivamente verifica la presenza degli ingredienti comprati in dispensa. Infine controlla che il budget rimanente sia corretto (budget totale - spesa per gli ingredienti).

```java
@Test
void testBuy() {
    pantry.setMorning(false);
    pantry.buy();
    pantry.setMorning(true);
    pantry.setDailyBudget(100);
    pantry.orderIngredients(farina);
    pantry.orderIngredients(vaniglia);
    pantry.buy();
    ingredients1.put(farina, 7);
    assertTrue(pantry.checkIngredients(ingredients1));
    ingredients2.put(vaniglia, 8);
    assertTrue(pantry.checkIngredients(ingredients2));
    assertEquals(48,pantry.getDailyBudget());
}
```

Figure 18: Test del metodo buy() della classe Pantry

4.3 Test Services

(Data la presenza del Singleton per la cassa i test devono essere ordinati correttamente, infatti poichè avviene dello scambio di soldi da e verso la cassa alcuni test risultano inesatti se compiuti in ordini diversi).
Per verificare la correttezza della classe Services eseguo i seguenti test sulle sue funzionalità:

1. un servizio che il cameriere offre è quello di consegnare il budget giornaliero alla dispensa. In questo semplice test si verifica che i soldi siano stati prelevati dalla cassa e consegnati alla dispensa (in cassa abbiamo 1000 euro iniziali).

```java
@Test
@Order(1)
void testTakeMoney() {
    services.takeMoney();
    assertEquals(500,pantry.getDailyBudget());
    assertEquals(500,register.getMoney());
}
```

Figure 19: Test del metodo takeMoney() della classe Services

2. il cameriere è colui che fa pagare il cliente quando se ne sta andando. Il test seguente verifica che il budget del cliente e i soldi della cassa siano rispettivamente diminuiti ed aumentati della quantità indicata dal costo della ricetta pagata (in cassa abbiamo 500 euro, il cliente ha un budget di 50 e la torta costa 12.5 euro).

```java
@Test
@Order(2)
void testPay() {
    services.pay(torta, client);
    assertEquals(512.5,register.getMoney());
    assertEquals(37.5,client.getBugdet());
}
```

Figure 20: Test del metodo pay() della classe Services

3. il cameriere ordina il cibo per il cliente. Con questo test verifico i 3 possibili casi:
   nel primo caso il cliente ordina correttamente una ricetta e paga (in cassa erano presenti 512.5 euro, il cliente ha un budget di 37.5 euro e la ricetta costa 12.5 euro).
   nel secondo caso il cliente se ne va perchè non ha abbastanza soldi per pagare (un messaggio lo comunica). Verifico che non ha pagato.
   nel terzo caso non ci sono abbastanza ingredienti per preparare la ricetta scelta ed il cliente se ne va (un messaggio lo comunica).Verifico che non ha pagato.
   Infine il cameriere prova ad ordinare ma non ha più clienti, per questo compare un messaggio che lo avvisa.

```java
@Test
@Order(3)
void testOrderFood() {
    services.orderFood();
    assertEquals(525,register.getMoney());
    assertEquals(25,client.getBugdet());
    services.orderFood();
    assertEquals(525,register.getMoney());
    services.orderFood();
    assertEquals(525,register.getMoney());
    services.orderFood();
}
```

Figure 21: Test del metodo orderFood() della classe Services

Inoltre verifico che la cassa è unica: provo a istanziarla due volte ma entrambe le volte ottengo la stessa istanza.

```
@Test
void testSingleton() {
    assertSame(register,register2);
}
```

Figure 22: Test pattern Singleton

4.4 Test ClientsManager

Per verificare la correttezza della classe ClientsManager eseguo i seguenti test sulle sue funzionalità:

1. iniziare e terminare la giornata lavorativa. Con questo test verico il corretto passaggio da inizio a fine giornata e viceversa.
   Partendo dalla mattina comincio la giornata terminando la mattina e verifico di poter aggiungere clienti. Controllo anche che il budget giornaliero sia arrivato.
   Successivamente ritento di cominciare la giornata ricevendo un messaggio di errore.
   Termino la giornata e verifico di non poter aggiungere altri clienti. Riprovando a concludere nuovamente la giornata ricevo un'altro messaggio di errore.
   Infine resetto lo stato della dispensa alla mattina (per gli altri test).

```
@Test
void testDayNight() {
    pantry.setMorning(true);
    manager.startDay();
    assertTrue(manager.isCanAddClient());
    assertFalse(pantry.isMorning());
    assertEquals(500,pantry.getDailyBudget());
    manager.startDay();
    manager.endDay();
    assertFalse(manager.isCanAddClient());
    manager.endDay();
    pantry.setMorning(true);
}
```

Figure 23: Test dei metodi startDay() e endDay() della classe ClientsManager

2. aggiungere clienti alla propria lista. Il test verifica l'aggiunta di un cliente alla lista, successivamente ritenta di aggiungere lo stesso cliente e invia un messaggio di errore. Infine prova ad aggiungere un cliente quando canAddClient è falso (il ristorante non accetta altri clienti) e mostra un messaggio di errore.

```java
@Test
void testAddClient() {
    manager.addClient(client);
    assertTrue(manager.getClients().contains(client));
    manager.addClient(client);
    manager.setCanAddClient(false);
    manager.addClient(client2);
}
```

Figure 24: Test del metodo addClient() della classe ClientsManager

3. affidare un cliente ad un cameriere.
   Questo test aggiunge un cliente alla propria lista (è l'unico cliente presente) e lo affida al cameriere "mario" (verifica che il cliente fa parte della lista clienti di "mario"), successivamente prova ad affidare un cliente a "giovanni" ma non essendoci clienti disponibili genera un messaggio di errore.
   Dopo questo aggiunge altri tre clienti alla sua lista e prova a assegnarli a "mario". I primi due vengono assegnati correttamente ma il terzo rimane senza cameriere poichè mario ha già raggiunto il suo limite di clienti contemporanei (massimo 3 clienti contemporanei). Lo verifico leggendo il messaggio di errore generato

```java
@Test
void testChooseWaiter() {
    manager.addClient(client);
    manager.chooseWaiter("mario");
    manager.chooseWaiter("giovanni");
    assertTrue(mario.getClients().contains(client));
    manager.addClient(client2);
    manager.addClient(client3);
    manager.addClient(client4);
    manager.giveClient(mario);
    manager.giveClient(mario);
    manager.giveClient(mario);
}
```

Figure 25: Test del metodo chooseWaiter() della classe ClientsManager

Infine in questo gruppo di test verifico anche il corretto funzionamento del pattern Observer utilizzato.
Il test cominicia la giornata lavorativa, aggiunge tre clienti e ne consegna due al cameriere "mario" e uno alla cameriera "sara". Entrambi i camerieri si occupano di un loro cliente lasciando "mario" con un cliente e "sara" con nessun cliente. Il test verifica la presenza di un solo cliente nel ristorante interpellando il manager. Questa è la prova del corretto dialogo tra camerieri e manager per quanto riguarda il numero di clienti totali nel ristorante.
Successivamente viene terminata la giornata lavorativa e "mario" si occupa del

suo ultimo cliente. Il test verifica che non ci sono più clienti rimasti nel ristorante e che (essendo finita la giornata lavorativa) per la dispensa è correttamente mattina. Questo ci dimostra che la sincronizzazione della mattina è avvenuta correttamente tra manager e dispensa.

```
@Test
void testObserver() {
    manager.startDay();
    manager.addClient(client);
    manager.addClient(client2);
    manager.addClient(client3);
    manager.chooseWaiter("mario");
    manager.chooseWaiter("mario");
    manager.chooseWaiter("sara");
    mario.orderFood();
    sara.orderFood();
    assertEquals(1,manager.getNumClientsDay());
    manager.endDay();
    mario.orderFood();
    assertEquals(0,manager.getNumClientsDay());
    assertTrue(pantry.isMorning());
}
```

Figure 26: Test del pattern Observer