Università degli studi di Firenze

Corso di Laura in Ingegneria Informatica

Elaborato di ingegneria del software

Applicativo in java per la gestione degli ordini in un fast food

Anna Valanzano

28 settembre 2020

Indice

Capitolo 1

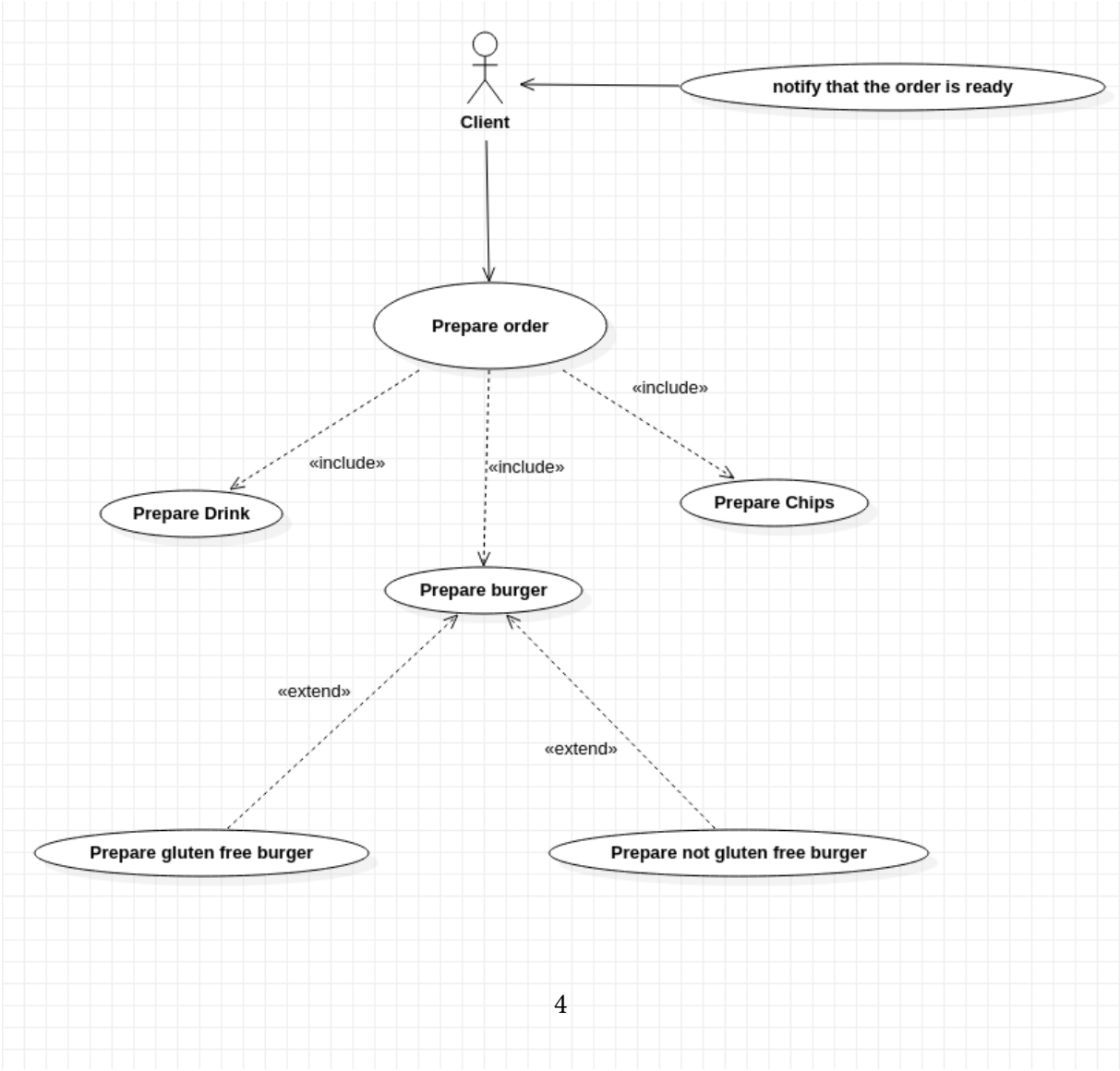
Introduzione

L'elaborato è un sistema semplicistico per la gestione degli ordini dei clienti in un fast food. In particolare, il sistema permette a un cliente di ordinare un menù completo costituito da panino, patatine e bibita oppure soltanto un panino. Inoltre, il cliente può ordinare un panino senza glutine. La preparazione dell'ordine a livello pratico consiste semplicemente nella costruzione di un oggetto Order tramite il design pattern Builder. Per gestire la costruzione di due tipi di menù (completo oppure no) viene utilizzato lo Strategy. Quando l'ordine è pronto il sistema manda una notifica al cliente tramite l' Observer.

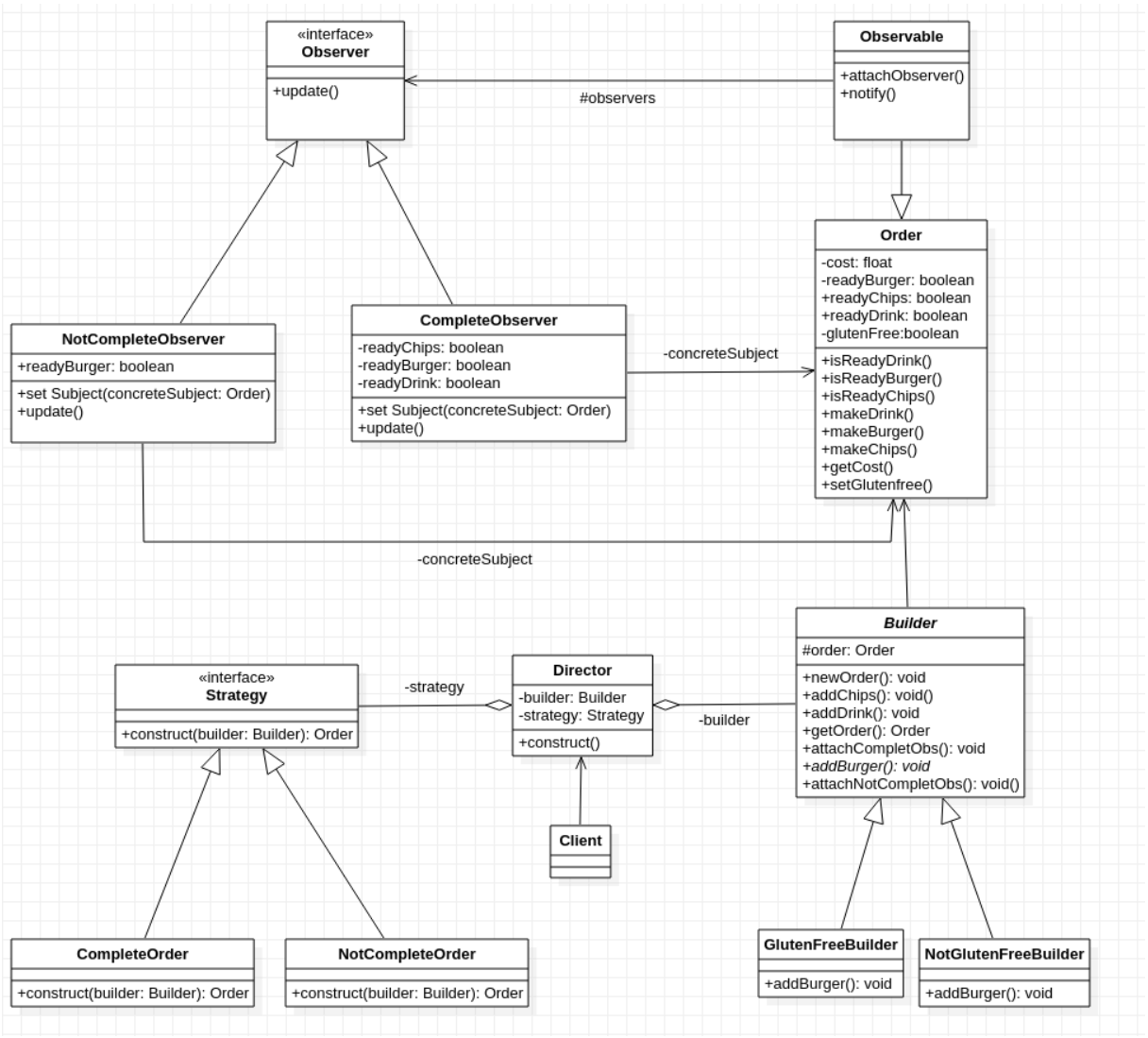
Capitolo 2

Progettazione

2.1 Use case

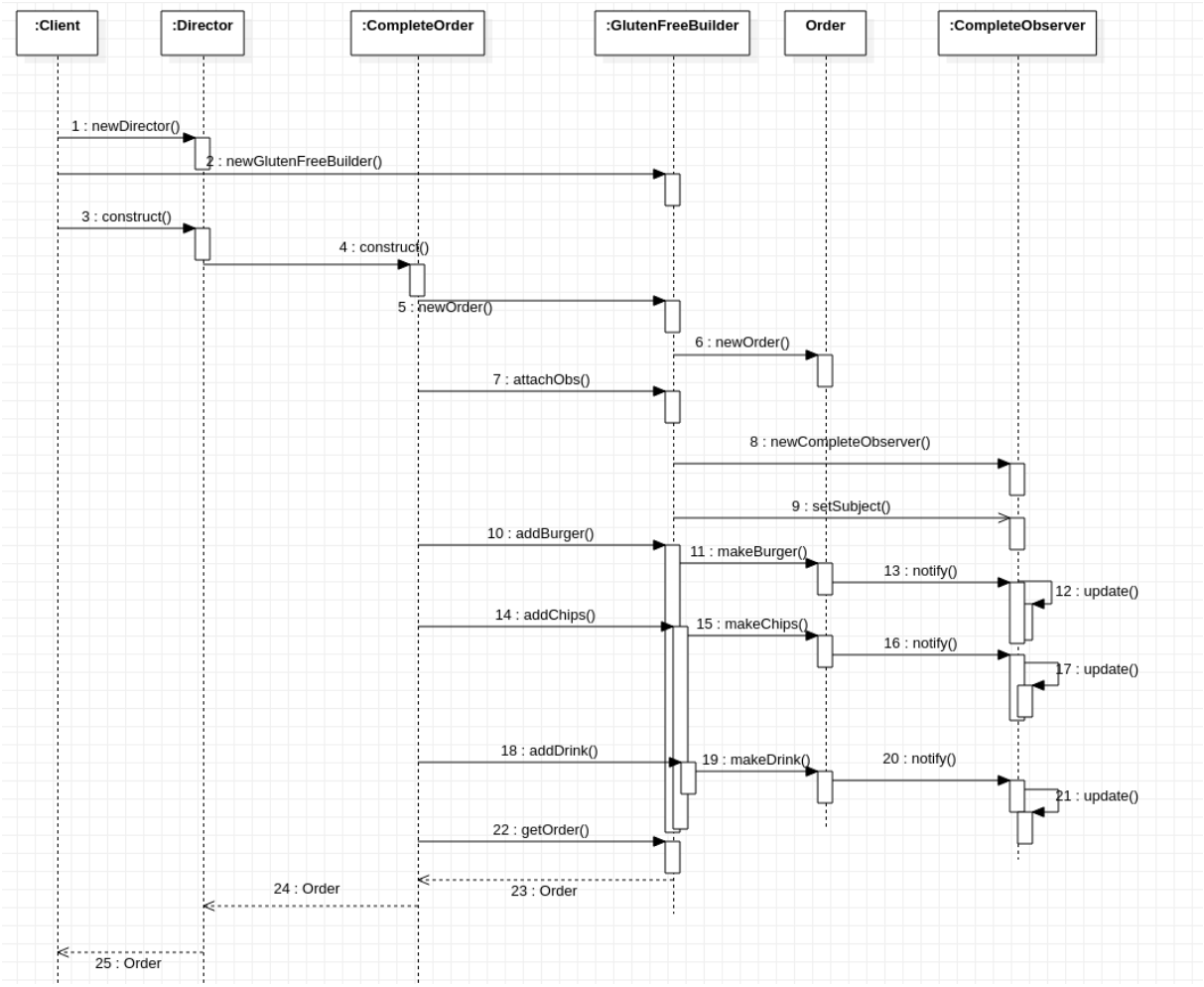


2.2 Class Diagram



2.3 Sequence Diagram

Si riporta il sequence diagram che riporta le azioni del sistema necessarie per costruire un menù completo e notificare l'avvenuta preparazione al cliente.



Capitolo 3

Implementazione

3.1 Classi e interfacce

3.1.1 Director

É la classe con cui interagisce il client e che avvia il processo di costruzione tramite il Builder e lo Strategy. Infatti, il costruttore prende in ingresso il riferimento al Concrete Builder (Gluten Free o Not Gluten Free) e al Concrete Strategy (Complete Order o Not Complete Order) in base al tipo al tipo di oggetto che si vuole costruire.

```
public class Director {
    private Builder builder;
    private Strategy strategy;

    public Director(Builder builder, Strategy strategy){
        this.builder= builder;
        this.strategy = strategy;

    }
    public  Order construct() { return strategy.construct(builder); }
}
```

3.1.2 Builder

É una classe astratta che ha i metodi necessari per costruire l'oggetto desiderato, e quindi in questo caso per costruire le singole parti dell'ordine. Inoltre, implementa anche i metodi per associare all'ordine un Observer, in modo che esso venga notificato appena l'ordine è pronto.

```
public abstract class Builder {
    protected Order order;
    public void newOrder() { order = new Order(); }

    abstract void addBurger();
    void addChips() { order.makeChips(); }
    void addDrink() { order.makeDrink(); }

    public Order getOrder() { return order; }

    public void attachCompleteObs() {
        CompleteObserver completeObserver = new CompleteObserver();
        order.addObserver(completeObserver);
        completeObserver.setSubject(order);
    }

    public void attachNotCompleteObs() {
        NotCompleteObserver notCompleteObserver = new NotCompleteObserver();
        order.addObserver(notCompleteObserver);
        notCompleteObserver.setSubject(order);
    }

}
```

3.1.3 Gluten free builder

Ha la funzione di Concrete Builder: fa override del metodo *addBurger()* per costruire il panino senza glutine.

```
class GlutenFreeBuilder extends Builder{

    @Override
    void addBurger() {
        order.setGlutenFree(true);
        order.makeBurger();

    }
}
```

3.1.4 Order

```
import java.util.Observable;

public class Order extends Observable {
    private float cost;
    private boolean readyChips;
    private boolean readyBurger;
    private boolean readyDrink;
    private boolean glutenFree;
    public float getCost() { return cost; }
    public void setGlutenFree(boolean glutenFree) { this.glutenFree = glutenFree; }
    public boolean isReadyDrink() { return readyDrink; }
    public boolean isReadyBurger() { return readyBurger; }
    public boolean isReadyChips() { return readyChips; }
    public boolean isGlutenFree() { return glutenFree; }

    public Order() {
        this.readyChips = false;
        this.readyBurger = false;
        this.readyDrink = false;
        this.cost = 0;
    }

    public void makeDrink() {
        this.readyDrink = true;
        this.cost += 2;
        setChanged();
        notifyObservers();
    }
    public void makeBurger() {
        this.readyBurger = true;
        this.cost += 2.5;
        setChanged();
        notifyObservers();
    }
    public void makeChips() {
        this.readyChips = true;
        this.cost += 1.5;
        setChanged();
        notifyObservers();
    }
}
```

3.1.5 Complete Order

É l'implementazione dell'interfaccia Strategy che nel metodo *construct()* contiene la sequenza effettiva della costruzione del tipo di prodotto voluto, in questo caso dell' ordine completo.

```
public class CompleteOrder implements Strategy{
    @Override
    public Order construct(Builder builder) {
        builder.newOrder();
        builder.attachCompleteObs();
        builder.addBurger();
        builder.addChips();
        builder.addDrink();
        return builder.getOrder();
    }
}
```

3.1.6 Complete Observer

Il pattern Observer è stato implementato con modalità Pull, ovvero l'Observer attraverso un riferimento al Concrete Subject aggiorna il suo stato.

```
import java.util.Observable;
import java.util.Observer;

public class CompleteObserver implements Observer {
    private Order concreteSubject;
    private boolean readyChips;
    private boolean readyBurger;

    public void setReadyChips(boolean readyChips) { this.readyChips = readyChips; }
    public boolean isReadyChips() { return readyChips; }
    public boolean isReadyBurger() { return readyBurger; }
    public boolean isReadyDrink() { return readyDrink; }
    private boolean readyDrink;

    public CompleteObserver(){
        this.readyChips = false;
        this.readyBurger = false;
        this.readyDrink = false;

    }
    public void setSubject(Order concreteSubject) { this.concreteSubject = concreteSubject; }


    @Override
    public void update(Observable o, Object arg) {
        if (concreteSubject.isReadyBurger()!= this.readyBurger) {
            this.readyBurger = concreteSubject.isReadyBurger();
            System.out.print("\nThe burger is ready");

        }
        if (concreteSubject.isReadyChips()!= this.readyChips){
            this.readyChips = concreteSubject.isReadyChips();
            System.out.println("\nThe chips are ready");
        }
        if (concreteSubject.isReadyDrink()!= this.readyDrink){
            this.readyDrink = concreteSubject.isReadyDrink();
            System.out.println("\nThe drink is ready");
        }
    }
}
```

Capitolo 4

Testing

4.0.1 Complete Observer Test

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CompleteObserverTest {
    @Test
    public void testObserver(){
        Order order = new Order();
        CompleteObserver observer = new CompleteObserver();
        order.addObserver(observer);
        observer.setSubject(order);
        order.makeDrink();
        assertEquals(observer.isReadyDrink(), order.isReadyDrink());

    }

}
```

4.0.2 Complete Builder Test

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CompleteBuilderTest {

    Builder builder = new GlutenFreeBuilder();
    Director director = new Director(builder, new CompleteOrder());
    Order order = director.construct();
    @Test
    void constructTest(){
        assertEquals(order.isReadyBurger(), actual: true);
        assertEquals(order.isReadyChips(), actual: true);
        assertEquals(order.isReadyDrink(), actual: true);
        assertEquals(order.getCost(),  actual: 6);
        assertEquals(order.isGlutenFree(),  actual: true);
    }

}
```

4.0.3 Not complete builder Test

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class NotCompleteBuilderTest {

    Builder builder = new NotGlutenFreeBuilder();
    Director director = new Director(builder, new NotCompleteOrder());
    Order order = director.construct();
    @Test
    void constructTest(){
        assertEquals(order.isReadyBurger(), actual: true);
        assertEquals(order.getCost(),  actual: 2.5);
        assertEquals(order.isGlutenFree(),  actual: false);
    }

}
```

4.0.4 Order Test

```
import org.junit.jupiter.api.BeforeEach;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class OrderTest {
    Order order;
    @BeforeEach
    void setUp() { order = new Order(); }

    @Test
    void makeDrinkTest(){

        assertEquals(order.isReadyDrink(), actual: false);
        assertEquals(order.getCost(), actual: 0);
        order.makeDrink();
        assertEquals(order.isReadyDrink(), actual: true);
        assertEquals(order.getCost(), actual: 2);
    }

    @Test
    void makeChipsTest(){
        assertEquals(order.isReadyChips(), actual: false);
        assertEquals(order.getCost(), actual: 0);
        order.makeChips();
        assertEquals(order.isReadyChips(), actual: true);
       assertEquals(order.getCost(), actual: 1.5);
    }

    @Test
    void makeBurgerTest(){
        assertEquals(order.isReadyBurger(), actual: false);
        assertEquals(order.getCost(), actual: 0);
        order.makeBurger();
        assertEquals(order.isReadyBurger(), actual: true);
        assertEquals(order.getCost(), actual: 2.5);
    }

}
```