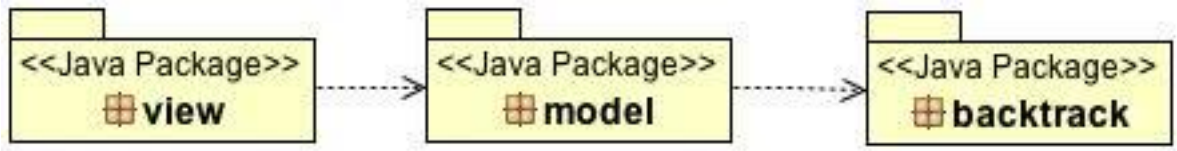
Ingegneria del Software

Progetto: Futoshiki

Il progetto Futoshiki consiste nella progettazione e implementazione di un risolutore di schemi di tale gioco. In una prima fase occorre creare lo schema con eventuali numeri o vincoli preinseriti e successivamente chiedere al risolutore di cercare e stampare tutte le possibili soluzioni per quello schema.

Il progetto è stato strutturato secondo il seguente package diagram:



Il package **view** fornisce le classi per l' interfaccia grafica. Il package **model** contiene le classi per la creazione dello schema di gioco ed infine il package **backtrack** contiene la classe responsabile della sua risoluzione.

Uno dei casi d'uso del sistema è il seguente:

1. L' utente seleziona la dimensione dello schema.

2. Inserisce dei numeri o dei vincoli nello schema appena creato e setta il numero massimo di soluzioni desiderate (di default calcolerà tutte le soluzioni possibili).

3. Chiede al sistema di risolvere lo schema.

4. Navigare sullo spazio delle soluzioni trovate.

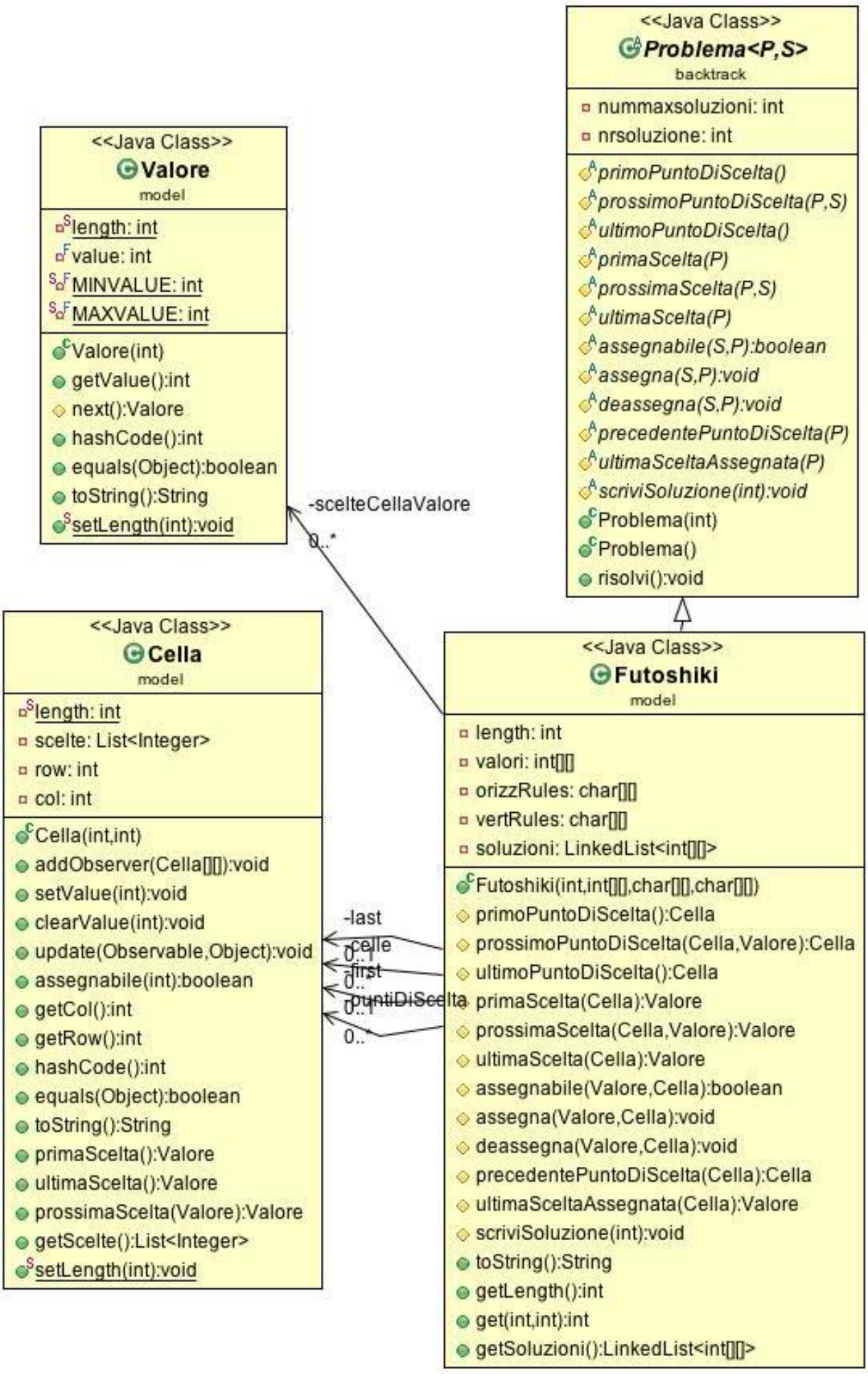
6. Azzera lo schema.

7. Cambia la dimensione dello schema.

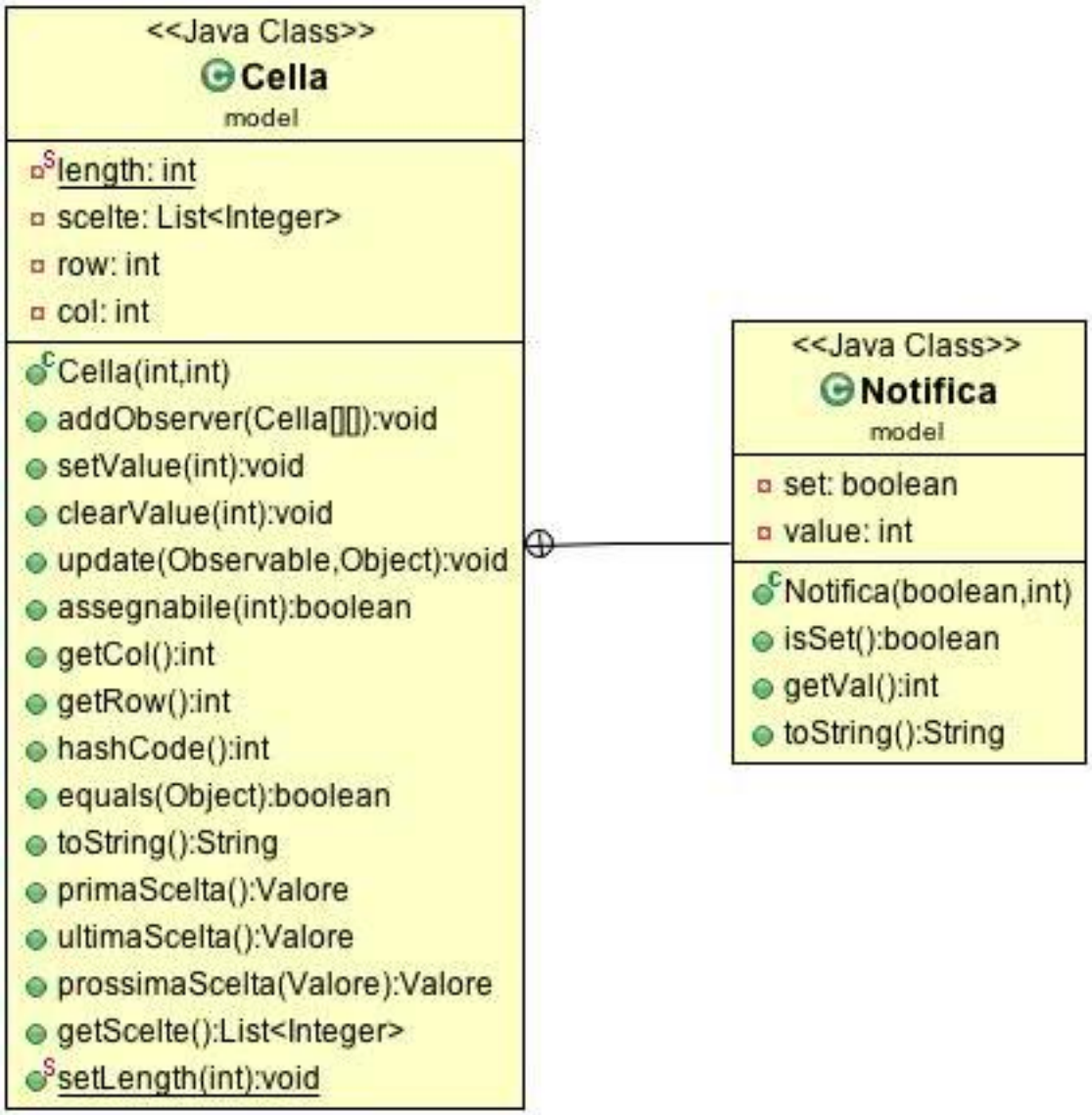
Date le specifiche, il problema è stato risolto usando il pattern **Template Method** e il **backtracking**.

In particolare, è stata estesa la classe Problema<P,S> vista a lezione con una classe specializzata al Futoshiki in cui P (punto di scelta) sono le celle dello schema e S (scelta) sono le disposizioni di numeri da 1 a n (con n dimensione dello schema).

Di seguito lo schema delle specializzazione della classe Problema in Futoshiki:


<<Java Class>>
Problema<P,S>
backtrack
nummaxsoluzioni: int
nrsoluzione: int
primoPuntoDiScelta()
prossimoPuntoDiScelta(P,S)
ultimoPuntoDiScelta()
primaScelta(P)
prossimaScelta(P,S)
ultimaScelta(P)
assegnabile(S,P):boolean
assegna(S,P):void
deassegna(S,P):void
precedentePuntoDiScelta(P)
ultimaSceltaAssegnata(P)
scriviSoluzione(int):void
Problema(int)
Problema()
risolvi():void
<<Java Class>>
Valore
model
length: int
value: int
MINVALUE: int
MAXVALUE: int
Valore(int)
getValue():int
next():Valore
hashCode():int
equals(Object):boolean
toString():String
setLength(int):void
-scelteCellaValore
0..*
<<Java Class>>
Cella
model
length: int
scelte: List<Integer>
row: int
col: int
Cella(int,int)
addObserver(Cella[][]):void
setValue(int):void
clearValue(int):void
update(Observable,Object):void
assegnabile(int):boolean
getCol():int
getRow():int
hashCode():int
equals(Object):boolean
toString():String
primaScelta():Valore
ultimaScelta():Valore
prossimaScelta(Valore):Valore
getScelte():List<Integer>
setLength(int):void
-last
celle
0..1
-first
0..
puntiDiScelta
0..1
0..*
<<Java Class>>
Futoshiki
model
length: int
valori: int[][]
orizzRules: char[][]
vertRules: char[][]
soluzioni: LinkedList<int[][]>
Futoshiki(int,int[][],char[][],char[][])
primoPuntoDiScelta():Cella
prossimoPuntoDiScelta(Cella,Valore):Cella
ultimoPuntoDiScelta():Cella
primaScelta(Cella):Valore
prossimaScelta(Cella,Valore):Valore
ultimaScelta(Cella):Valore
assegnabile(Valore,Cella):boolean
assegna(Valore,Cella):void
deassegna(Valore,Cella):void
precedentePuntoDiScelta(Cella):Cella
ultimaSceltaAssegnata(Cella):Valore
scriviSoluzione(int):void
toString():String
getLength():int
get(int,int):int
getSoluzioni():LinkedList<int[][]>

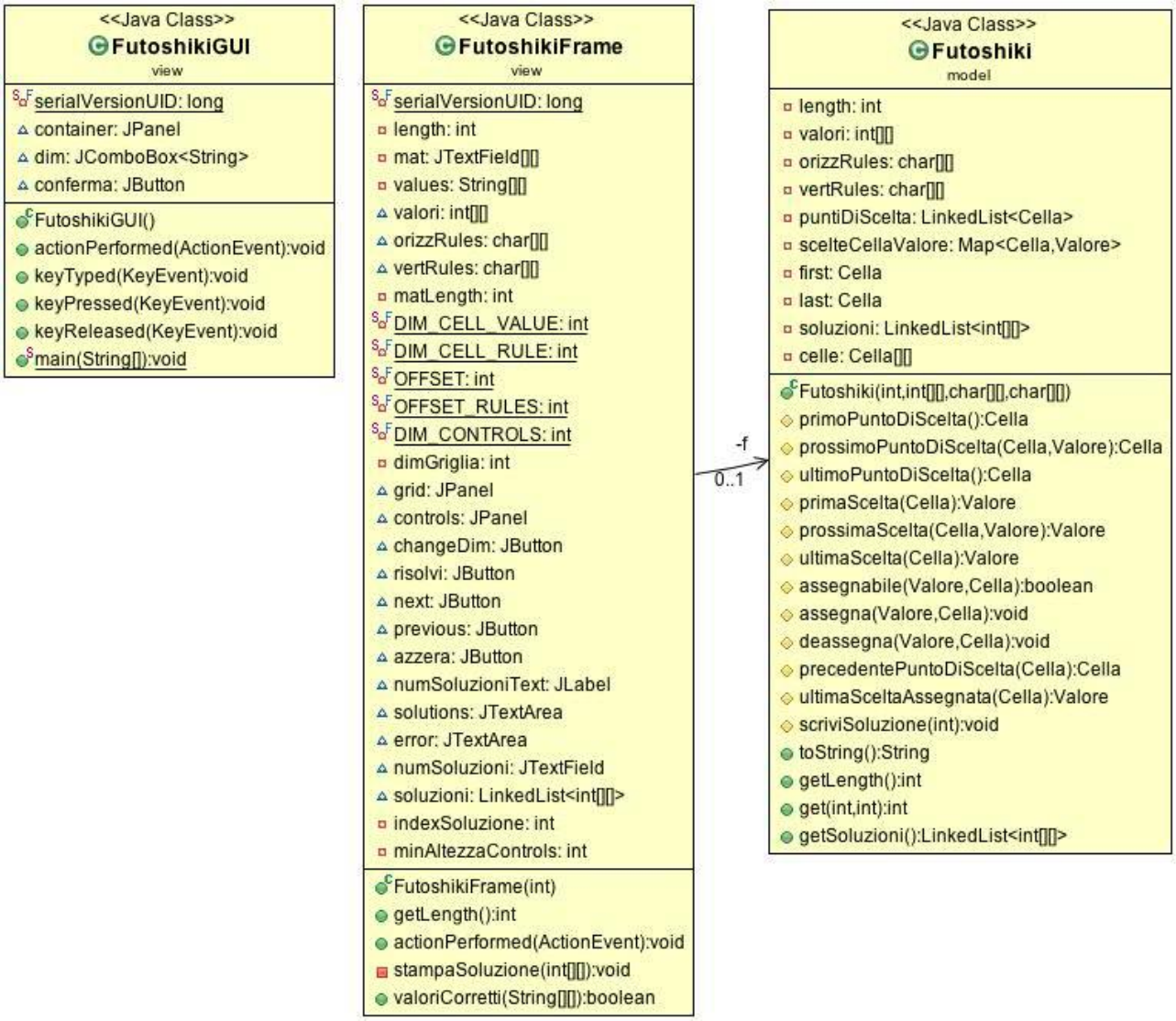
Visti i vincoli imposti dal gioco su righe e colonne, si è deciso di impiegare il pattern **Observer** per rispettarli. Ogni cella della matrice dello schema è un oggetto Observable ed è un Observer delle celle che giacciono sulla sua stessa riga e colonna. Dunque se a una cella viene assegnato un valore i suoi Observer sapranno che non potranno assegnarlo (a meno di de assegnazioni in seguito) perché violerebbero il vincolo che ogni valore deve essere presente una volta sola sulla riga e sulla colonna. Questa soluzione fa in modo che, ad ogni istante di tempo, ogni cella possa decidere indipendentemente se un certo valore le può essere assegnato o meno e ciò viene sfruttato per verificare se una scelta è assegnabile in un punto di scelta.



Per la comunicazione tra le celle si è deciso di realizzare una inner-class all'interno di Cella chiamata **Notifica** la quale ha il compito di riferire, a chi osserva la cella che cambia, se si sta settando o eliminando un valore. Essa inoltre riferisce il valore che assumerà/assumeva la cella affinché si possano ricalcolare le scelte possibili.

Il pattern Observer adottato svolge a dovere il suo compito a discapito però di legare la classe Cella a uno specifico contesto e quindi non favorisce la riusabilità del codice.

Infine è stato usato una variante del pattern **Model-View-Controller** per disaccoppiare la logica applicativa dalla sua rappresentazione grafica.



La classe FutoshikiFrame racchiude al suo interno un'istanza della classe Futoshiki che viene istanziata dopo aver ricevuto da input i valori richiesti per lo schema. Questa, dopo aver risolto lo schema, restituirà all'interfaccia grafica tutte le soluzioni che verranno dunque visualizzate all'utente. L'utente potrà inoltre scorrere le varie soluzioni con semplici tasti Next e Previous.

Infine si è fatto uso del framework di **JUnit** per il testing della classe Futoshiki. In particolare si è creata un'istanza del gioco e si è verificata la correttezza del risultato in uscita.