UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

cdLT Ingegneria Informatica

Corso di Ingegneria del Software – A.A. 2018/19

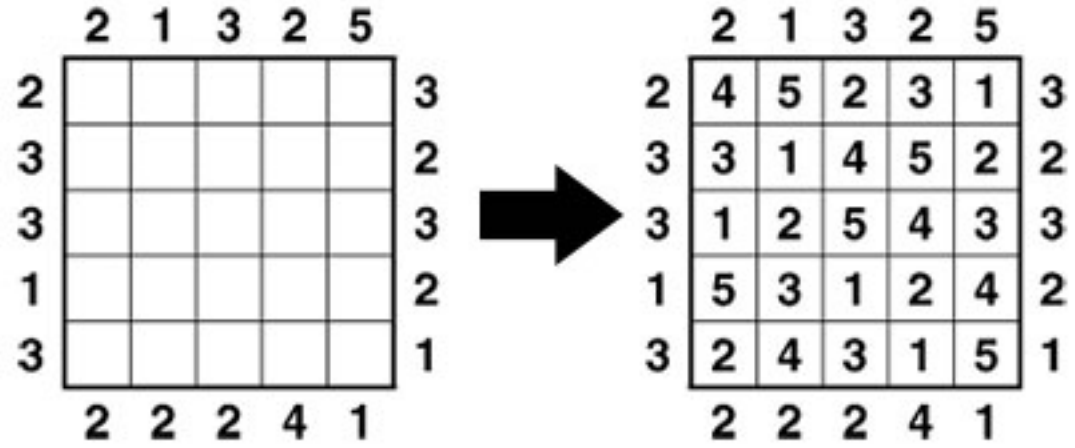
Prof. Angelo FURFARO

Progetto d'esame sui Design Patterns: GRATTACIELI

Giuseppe LONGO - matricola 175983

Grattacieli

La griglia rappresenta il quartiere di una città. Sapendo che in ciascuna riga o colonna non vi sono grattacieli della stessa altezza, e che i numeri all'esterno indicano quanti grattacieli sono visibili da quel punto di vista (quelli più alti nascondono quelli più bassi), occorre scrivere in ciascuna casella l'altezza del grattacielo corrispondente



Risolvere il gioco progettando e sviluppando un'applicazione Java basata su template method e la tecnica backtracking, realizzando una classe erede di Problema<P,S> specializzata al gioco dei GRATTACIELI. Prevedere una GUI con la quale si possa configurare il gioco (porre numeri nelle caselle, stabilire relazioni di precedenza tra caselle contigue etc.), definire il numero massimo desiderato di soluzioni, lanciare l'applicazione e, infine, navigare (con tasti tipo next/previous) sullo spazio delle soluzioni trovate, mostrando a video le varie soluzioni.
Nello sviluppo del progetto si devono utilizzare i Design Pattern ritenuti più adeguati motivandone opportunamente la scelta. Le fasi del processo di sviluppo devono essere documentate ricorrendo, ove necessario, all'uso di diagrammi UML.

Si richiede inoltre di effettuare il testing di uno o più moduli significativi impiegando un opportuno criterio e sfruttando le funzionalità offerte dal framework JUnit.

DESCRIZONE INFORMALE

È richiesta la progettazione procedurale, di un *solver* per un diffuso gioco di logica che prende il nome di *Grattacieli*.

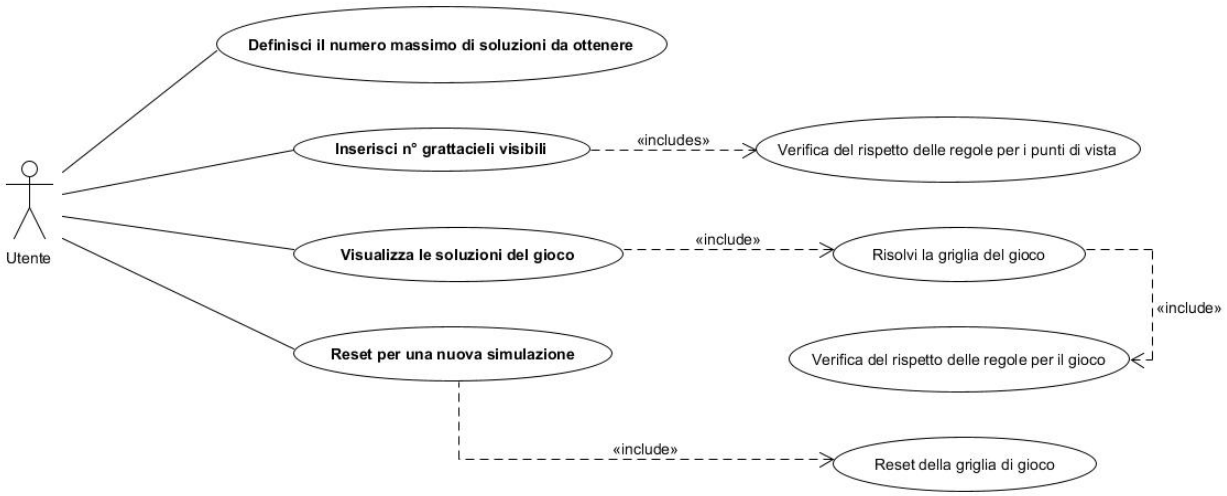
La griglia rappresenta una città in cui ciascuna casella è occupata da un grattacielo. L'esempio che verrà implementato propone una griglia quadrata avente 25 (*5x5*) grattacieli. Lo scopo del gioco è riempire tale griglia inserendo l'altezza del grattacielo corrispondente, sapendo che:

1. in ciascuna riga o colonna non posso esistere grattacieli della stessa altezza;
2. i numeri all'esterno della griglia indicano quanti grattacieli sono visibili da quel punto di vista (quelli più alti nascondono quelli più bassi);
3. le altezze ammissibili vanno da 1 a 5;
4. il solver si avvierà se è stato definito ALMENO UNO punto di vista;
5. se i "punti di vista" sono stati tutti definiti, la soluzione è unica.

La progettazione logica verrà attuata, come richiesto, tramite tecnica *backtracking*: è però evidente come essa non rapprensentI l'approccio risolutivo migliore. Difatti, è possbile estrarre dalle regole del gioco, delle regole "non scritte" che pemettono una risoluzione non più a "tentativi" ma più intelligente (e furba) e quindi efficiente (basta solo considerare che le combinazioni valide crescono in modo esponenziale all'aumentare della dimensione della griglia). Ad esempio, trovare *1* come numero di grattacieli visibili da un determinato *punto di vista*, renderà univoco il grattacielo da inserire nella prima cella disponibile, che avrà altezza pari alla massima ammissibile. Quindi iterando il tutto, l'applicazione della procedura di backtracking verrà si attuata, ma alla fine e come "ultima spiaggia" su un numero di celle irrisorio.

DECRIZIONE SCENARIO E *USE-CASE DIAGRAMS*

Gli *use-case diagrams* sono dedicati alla descrizione delle funzioni o servizi offerti dal sistema in esame, così come percepiti dagli attori o utenti che vi interagiscono. Questi diagrammi possono anche essere utilizzati durante la progettazione del software, considerandoli anche come strumento di rappresentazione dei *requisiti funzionali*.



Ad esempio, all'interno di questo software i casi d'uso con cui l'attore "utente" può relazionarsi sono:

CASO D'USO	**Definisci il numero massimo di soluzioni da ottenere**
Tipo	Primario
Pre-condizione	I *punti di vista* sono stati definiti correttamente.
Svolgimento normale	L'utente sceglie un numero intero magiore di 0.
Svolgimento alternativo	-
Post-condizione	-
Descrizione	Si definisce il numero massimo di soluzioni ammissibili. L'utente potrà navigare sullo spazio delle soluzioni trovate.

CASO D'USO	**Inserisci n° grattacieli visibili**
Tipo	Primario
Precondizione	La griglia deve rispettare le regole imposte per la definizione dei *punti di vista*.
Svolgimento normale	Si sceglie una cella vuota della griglia corrispondente a uno dei *punti di vista* disponibili e si inserisce un numero che va da 1 all' altezza massima pari a 5. Se l'operazione non soddisfa le regole, essa non viene accettata.
Svolgimento alternativo	In caso di cella non vuota, il numero corrente è sostituito con il nuovo numero di grattacieli inserito. Se l'operazione non soddisfa le regole del gioco, essa non viene accettata.
Post-condizione	La griglia aggiornata deve rispettare le regole imposte per i *punti di vista*. Il numero di grattacieli posti nei vari punti di vista deve essere coerente con le specifiche del gioco.
Descrizione	Il caso d'uso in esame permette all'utente di configurare il risolutore del gioco.

CASO D'USO	**Verifica del rispetto delle regole per i punti di vista**
Tipo	Secondario
Precondizione	-
Svolgimento normale	Si verifica che i grattacieli visibili al *punto di vista* siano stati definiti attraverso un numero compreso tra 1 e 5.
Svolgimento alternativo	-
Post-condizione	La griglia rispetta le regole imposte per i *punti di vista.*
Descrizione	Questo è un caso d'uso cruciale. Dalle verifiche attuate in questo contesto, deriverà tutta la corretta simulazione che si andrà ad analizzare.
CASO D'USO	**Visualizza le soluzioni del gioco**
Tipo	Primario
Precondizione	Le celle predisposte ai punti di vista devono avere valori da 1 a 5. Almeno una deve essere stata così definita.
Svolgimento normale	Si mostrano graficamente le soluzioni. La prima mostrata sarà la prima calcolata.
Svolgimento alternativo	Se la configurazione inserita non ha soluzione, si mostra graficamente un messaggio di errore/avviso.
Post-condizione	L'utente avrà a disposizione un'interfaccia interattiva con cui poter navigare e graficare le soluzioni.
Descrizione	Questo caso d'uso permette di mostrare graficamente tutte le soluzioni ottenute, in numero coerente col limite massimo scelto dall'utente.

CASO D'USO	**Risolvi la griglia di gioco**
Tipo	Secondario

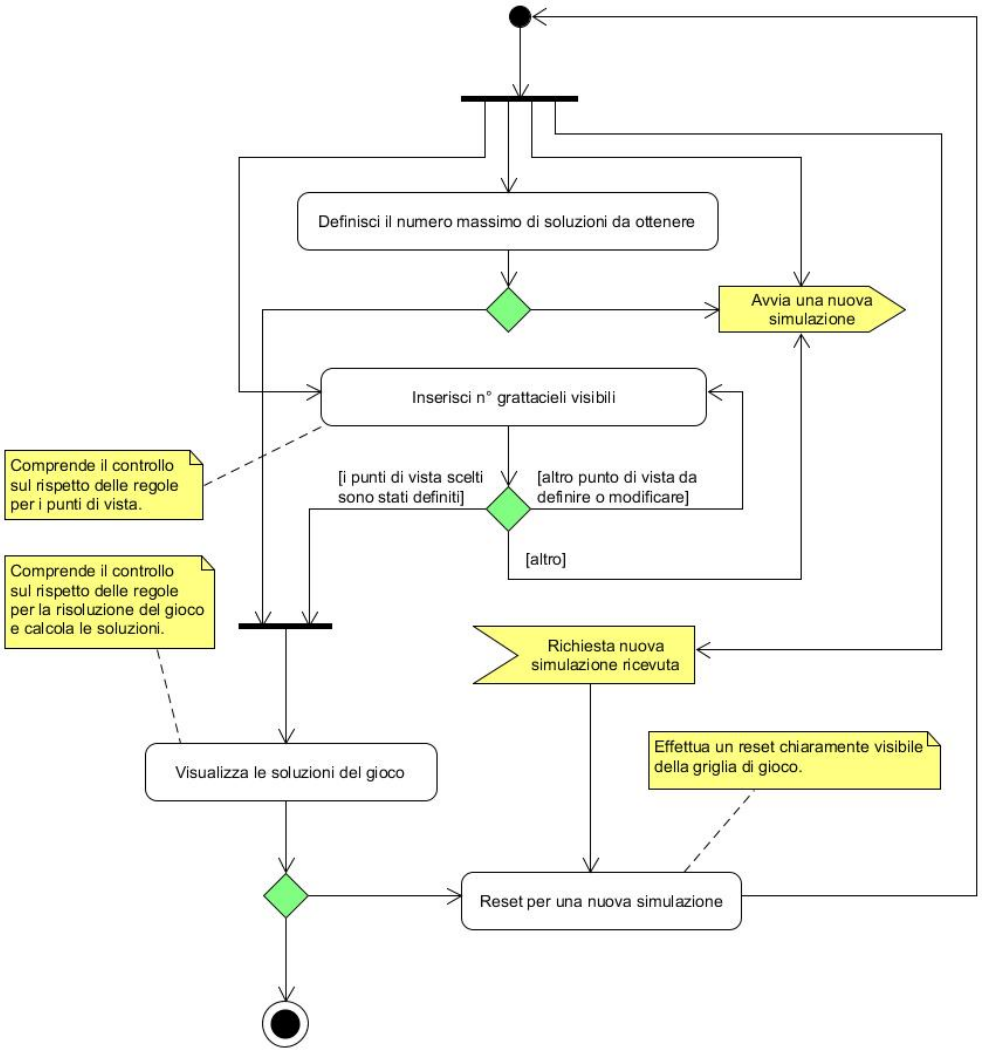
Precondizione	La griglia deve rispettare le regole imposte dal gioco. Almeno una cella deve avere valore da 1 a 5;
Svolgimento normale	Si ottengono le soluzioni per i specifici vincoli proposti (n° grattacieli nei punti di vista) nel rispetto delle regole del gioco.
Svolgimento alternativo	Se la configurazione inserita non ha soluzione, si solleva un messaggio di errore/avviso.
Post-condizione	-
Descrizione	L'operazione di risoluzione è implementata mediante un algoritmo in backtracking. Esso genera delle soluzioni candidate e poi ne seleziona quelle coerenti coi vincoli proposti.

CASO D'USO	**Verifica del rispetto delle regole del gioco**
Tipo	Terziario
Precondizione	-
Svolgimento normale	Si verifica che almeno un *punto di vista* porti il numero di grattacieli che possono essere visti "guardando" da esso stesso.
Svolgimento alternativo	-
Post-condizione	La griglia rispetta le regole imposte dal gioco.
Descrizione	Questo è un caso d'uso cruciale. Dalle verifiche attuate in questo contesto, deriverà tutta la corretta simulazione che si andrà ad analizzare.

CASO D'USO	**Reset per una nuova simulazione**
Tipo	Primario
Precondizione	-
Svolgimento normale	La griglia viene resettata. I risultati ottenuti vengono azzerati.
Svolgimento alternativo	-
Post-condizione	L'utente può inserire un nuovo limte per il numero massimo di soluzioni da visualizzare. L'utente può definire nuova configurazione di valori ai punti di vista.
Descrizione	Il caso d'uso corrente permette di avviare una nuova sessione di simulazione. Può essere invocato in qualsiasi momento e comporta l'annullamento di quanto inserito o già selezionato.

CASO D'USO	**Reset della griglia di gioco**
Tipo	Secondario
Precondizione	E' stata richiesta una nuova simulazione.
Svolgimento normale	Vengono cancellati tutti i valori della griglia, sia i vincoli che le *altezze* trovate, anche graficamente.
Svolgimento alternativo	-
Post-condizione	La griglia del gioco è completamente vuota e pronta per una nuona simulazione.
Descrizione	Il caso d'uso permette lo "svuotamento" immediato della griglia di gioco, visibile graficamente.

LOGICA PROCEDURALE: *ACTIVITY DIAGRAM*



L'*activity diagram* su raffigurato presenta l'articolazione del progetto a tempo di esecuzione. Nello specifico, si è scelto di analizzare la correlazione tra i vari casi d'suo primari nello scenario che si sta analizzando.

L'utente deve prima definire il numero massimo di soluzioni che potrà visualizzare a simulazione eseguita. La risoluzione effettiva del gioco potrà essere avviata solo dopo aver definito <u>almeno uno</u> tra i punti di vista disponibili. Inoltre, sempre dallo stesso diagramma, è evidente come l'utente possa decidere, in qualsiasi momento, di resettare la sessione corrente e quindi avviare una nuova partita.

I *casi secondari e terziari* visti negli use-case diagrams, sono qui specificati, per semplicità di diagramma, come commenti a quelli primari, che li includono.

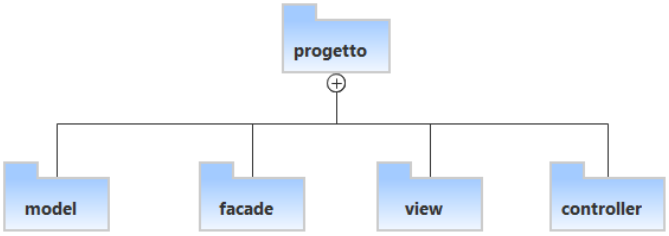
PROGETTAZIONE E *DESIGN PATTERNS*

Le varie fasi di sviluppo per la messa in opera del progetto proposto, seguono di pari passo l'implementazione dei *design patterns* che sono stati ritenuti più adeguati.

PATTERN ARCHITETTURALI

Model – View – Controller

E' la stessa traccia a "far pensare" a una progettazione del gioco sulla logica data da questo pattern: difatti, viene richiesta una GUI con cui l'utente possa interagire con il solver. Quindi il *model* contiene il cuore vero del gioco mentre la parte *view* visualizza i dati contenuti nel model e si occupa dell'interazione con il giocatore. Il *controller* infine, riceve i comandi dell'utente attraverso il view e li attua modificando lo stato delle classi e oggetti interpellati contenuti nel model, e viceversa.



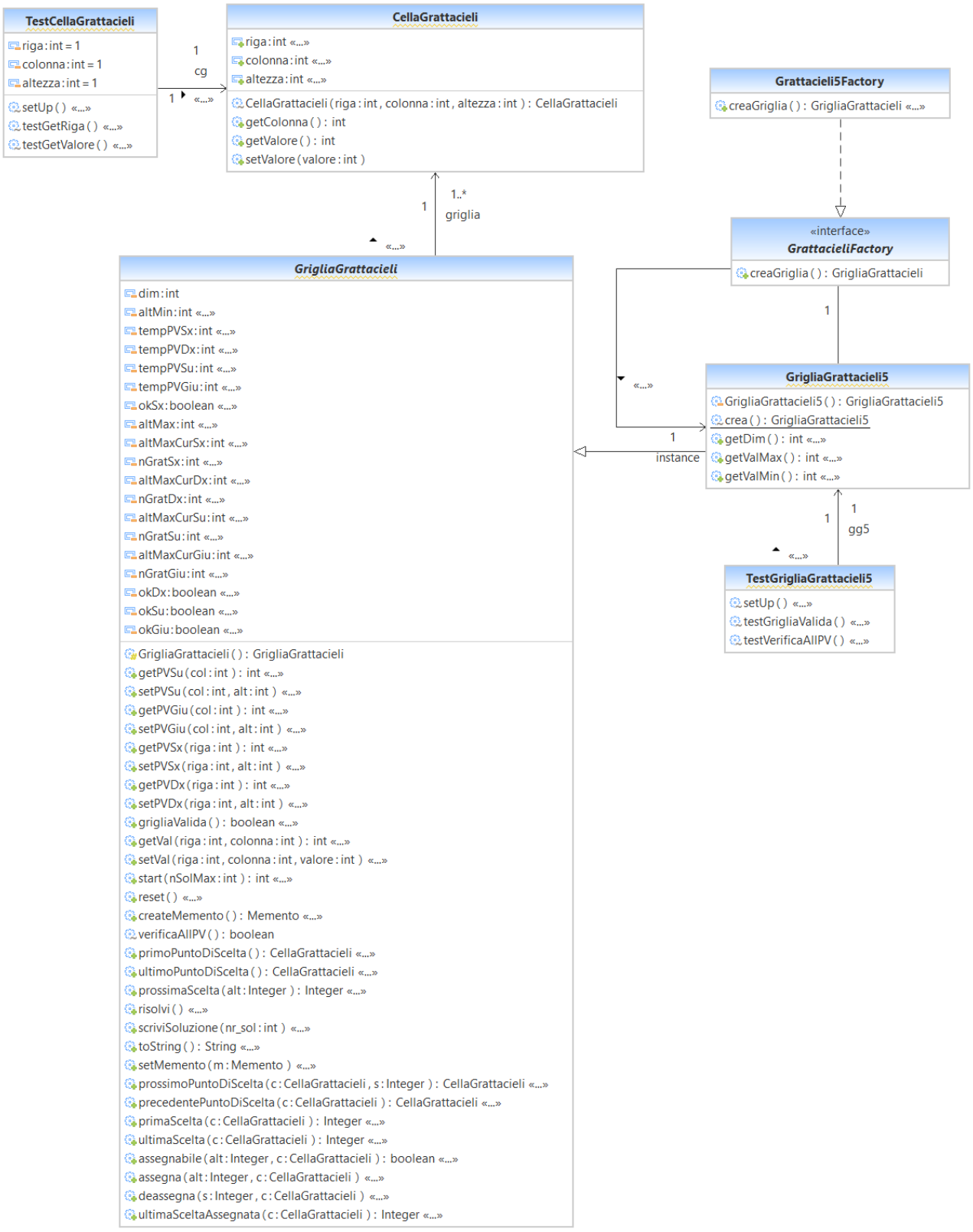
PATTERN CREAZIONALI

Factory Method

Può essere identificato come il *pattern per eccellenza.* Esso collabora alla programmazione ad oggetti permettendo alti livelli di *astrazione* e contribuendo al mantenimento di *modularità* nella stesura di codice. Nello specifico, fornisce un'interfaccia (o classe astratta) per creare un oggetto, ma lascia che le sottoclassi decidano che tipo di oggetto istanziare (anche applicando dell'*inheritance* all'istance di una classe). Quindi è evidente l'importanza di questo pattern, che sarà anche fondamentale (e quindi verrà utilizzato da questi) per dar senso ad altri pattern sia strutturali che comportamentali.

Abstract Factory

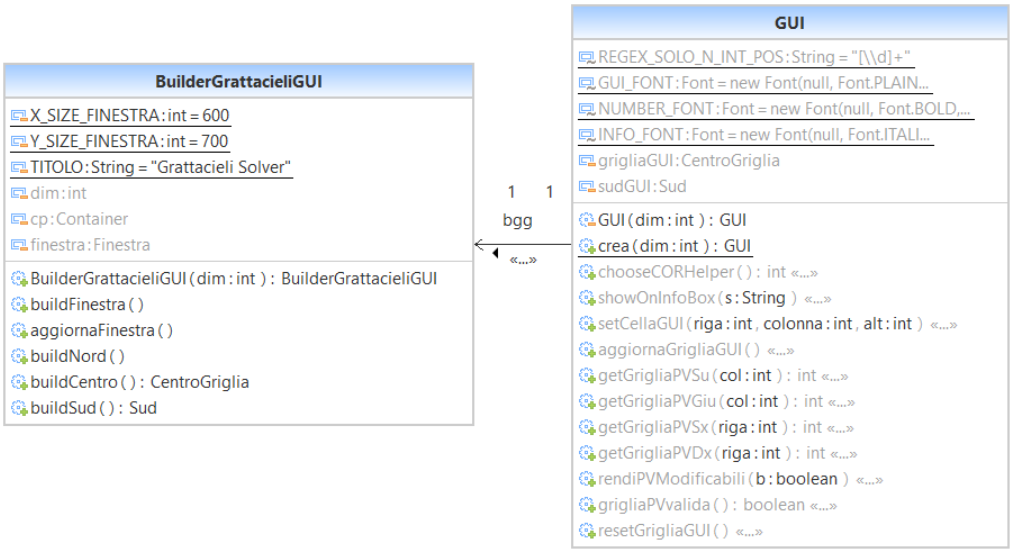
L'intento dell'Abstract Factory è quello di fornire una interfaccia che i *concrete factory* posso utilizzare per creare famiglie di oggetti in relazione o dipendenti tra loro. In questo modo, il programmatore che avrà necessità di istanziare un oggetto o classe, potrà "rivolgersi" allo specifico factory concreto, senza dover utilizzare direttamente i costruttori degli stessi. Nel caso particolare del progetto che stiamo definendo, l'Abstract Factory è stato utilizzato all'interno del model, facendo sì che l'oggetto *GrigliaGrattacieli5* possa essere istanziato (al di fuori del package *abstractFactory*) solo attraverso il metodo *creaGriglia()* definito dentro il factory concreto *Grattacieli5Factory().*



Builder

Il design pattern *Builder*, nella programmazione ad oggetti, separa la costruzione di un oggetto complesso dalla sua rappresentazione, cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. Ciò ha l'effetto immediato di rendere più semplice la “concretizzazione grafica”, permettendo a una classe

builder separata di focalizzarsi sulla corretta costruzione dell'istanza della classe lasciando che la classe originale si concentri sul funzionamento logico dell'oggetto che rappresenta. Qui in *Grattacieli,* l'oggetto builder *BuilderGrattacieliGUI* è stato definito con lo scopo di rendere subito disponibile la costruzione della GUI da parte della istanza di *GUI.* Difatti quest'ultima utilizzerà il builder per istanziare le varie parti della interfaccia grafica, mentre essa stessa si occuperà di adempiere le richieste che il *model* invierà all'interfaccia grafica e viceversa (tramite il *controller*).



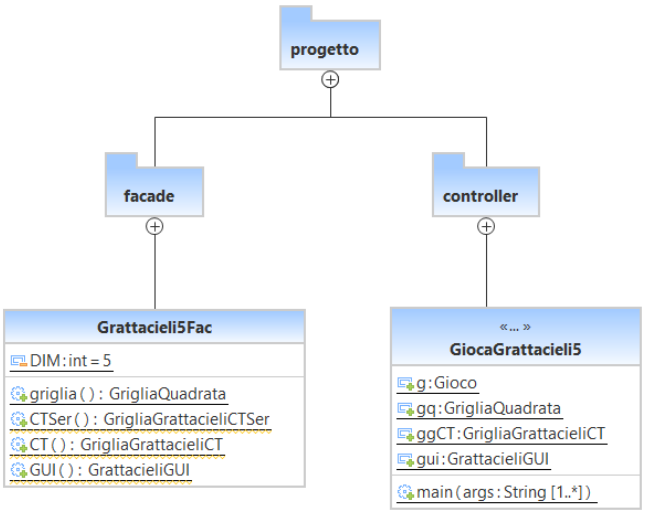
Singleton

Il singleton è un design pattern creazionale che ha lo scopo di garantire che una determinata classe possa essere istanziata una e una sola volta. Per fare ciò, viene fornito un punto di accesso globale a tale istanza tramite un metodo statico che restituisce l'istanza della classe, sempre la stessa. Essa infatti viene creata alla prima chiamata del metodo, memorizzandone il riferimento in un attributo privato anch'esso statico. Di conseguenza, la classe *singleton* deve avere un unico costruttore dichiarato privato, in modo da impedire l'istanziazione diretta della classe sia a livello *package* che pubblicamente. Nel caso del progetto in questione, le classi che implementano la politica singleton sono esattamente tre: *GrigliaGrattacieli5, GrigliaGrattacieliCT, GUI.* Essi difatti, rappresentano i tre oggetti fondamentali su cui tutto il gioco si basa, che devono appunto essere univoci per gli scopi a cui devono asserire. *GrigliaGrattacieli5* rappresenta il gioco vero e proprio e contiene tutte le logiche dello stesso; *GrigliaGrattacieliCT* è l'oggetto che permette la navigazione tra le soluzioni ottenute; *GUI* è proprio l'interfaccia grafica di Grattacieli.

PATTERN STRUTTURALI

Facade

Il *Facade Pattern* è un design pattern che ha la funzione di andare a nascondere la complessità di un sistema offrendo al programmatore che ci si vuole interfacciare, una classe di "facciata" contenente i metodi con cui egli può interagire. Nel caso corrente, il sotto-package *facade* contiene una classe che fa riferimento ai metodi di creazione dei tre oggetti cardine per la messa in opera del gioco. Difatti, verrà utilizzata dal *controller* dentro il *main.*



Decorator

Il design pattern *decorator* consente di aggiungere nuove funzionalità ad oggetti già esistenti. Questo viene realizzato costruendo una nuova classe decoratore che "avvolge" l'oggetto originale. Al costruttore del decoratore si passa come parametro l'oggetto originale o anche un differente decoratore. In questo modo, più decoratori possono essere concatenati l'uno all'altro, aggiungendo così in modo incrementale funzionalità alla classe concreta (che è rappresentata dall'ultimo anello della catena). I *framework* di *Java* quali *Swing* e *Awt* basano su questo metodo (e altri) la loro stessa logica di funzionamento, e sono gli stessi utilizzati per l'implementazione grafica del gioco.

Composite

Si tratta di un pattern che viene utilizzato quando si ha la necessità di realizzare una gerarchia di oggetti in cui l'oggetto contenitore può contenere oggetti elementari e/o oggetti contenitori. L'obiettivo è di permettere al client (utente o programmatore) che deve navigare la gerarchia, di comportarsi sempre nello stesso modo sia verso gli oggetti elementari e sia verso gli oggetti contenitori. Anche questo pattern fa parte intrinsecamente di *Swing* e *Awt,* quando ad esempio un oggetto *panel* più complesso tiene dentro di sé altri *panel* di secondaria funzionalità.

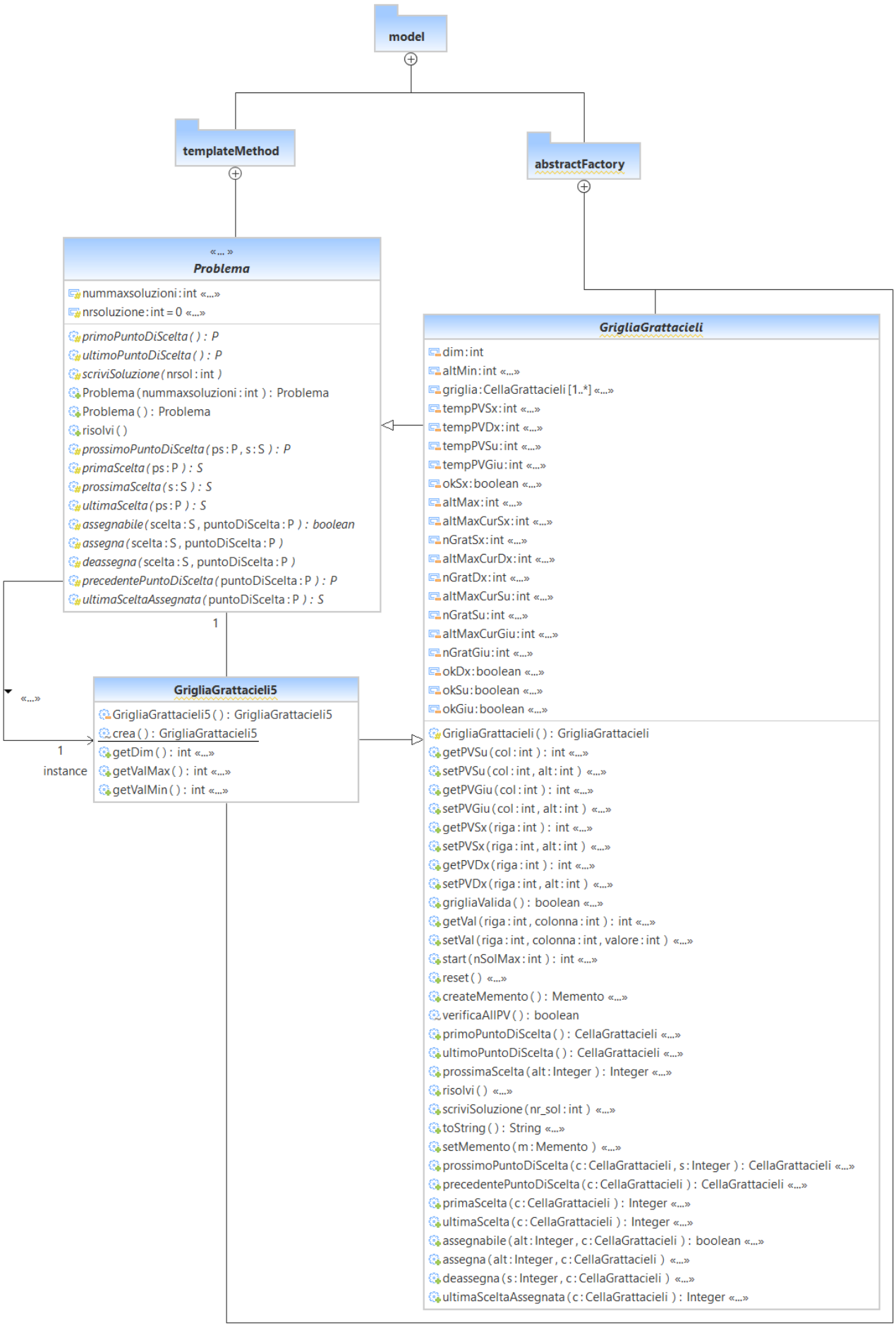
Adapter

Il pattern *Adapter* svolge un ruolo chiave quando quando si intende utilizzare un componente software ma occorre adattare la sua interfaccia per motivi di integrazione con l'applicazione esistente. Questo comporta la definizione di una nuova interfaccia che deve essere compatibile con quella esistente in modo tale da consentire la comunicazione con l'interfaccia da "adattare". E' necessario che venga citato tra i pattern utilizzati nel progetto in quanto ne risulta l'effettiva applicazione in *Swing* attracerso la definizione dei *Key Listener* tramite *Key Adapter* nella *GUI*. Difatti la stessa interfaccia *Key Listener* presenta una diversa tipologia di "ascoltatori" sui tasti fisici della tastiera, che possono invocare eventi a pressione terminata o a carattere inserito ad esempio. Nel caso di *Grattacieli*, la scelta è ricaduta su *Key Pressed*, rendendo inutile l'implementazione degli altri tipi di *key event.* Questi metodi vengono vengono quindi "tralasciati" e praticamente omessi e ciò è reso possibile proprio tramite l'uso dell'interfaccia *Key Adapter*.

PATTER COMPORTAMENTALI

Template Method

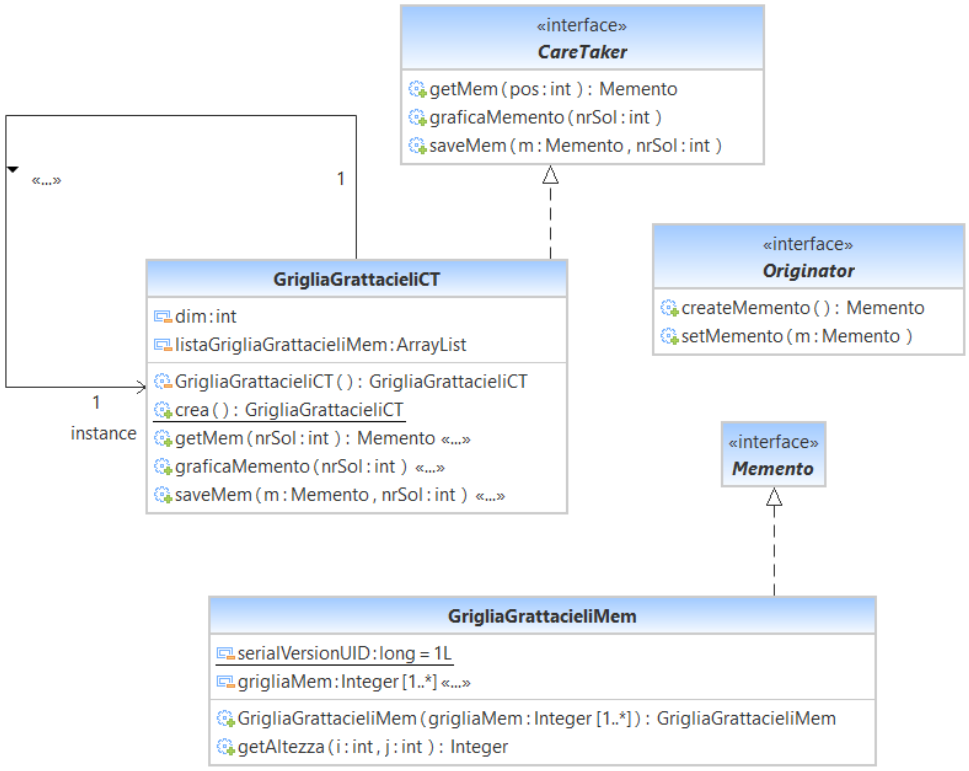
Questo pattern nasce dall'esigenza di delegare alle sottoclassi l'implementazione di alcune operazioni che una classe astratta (o interfaccia) non può ancora definire. Pertanto il metodo che definisce una specifica funzione viene implementato nella superclasse, mentre i metodi che determinano i comportamenti di dettaglio vengono dichiarati astratti ed implementati nelle sottoclassi concrete. E' chiaro quindi come vi sia un utilizzo intrinseco del *Factory Method.* In particolare poi nella progettazione qui analizzata, il *Template Method* è utilizzato nella definizione dell'algoritmo di *backtracking* su cui si basa interamente il *solver*. *Problema* è difatti la classe astratta che verrà estesa da *GrigliaGrattacieli,* anch'essa astratta ma che per sua natura può definire tutti i metodi necessari al gioco in *override*, a loro volta ereditati dalla classe concreta *GrigliaGrattacieli5* che andrà a chiudere quindi il quadro di progettazione.


model
templateMethod
abstractFactory
«...»
Problema
nummaxsoluzioni : int «...»
nrsoluzione : int = 0 «...»
primoPuntoDiScelta ( ) : P
ultimoPuntoDiScelta ( ) : P
scriviSoluzione (nrsol : int)
Problema (nummaxsoluzioni : int) : Problema
Problema ( ) : Problema
risolvi ( )
prossimoPuntoDiScelta (ps : P , s : S) : P
primaScelta (ps : P) : S
prossimaScelta (s : S) : S
ultimaScelta (ps : P) : S
assegnabile (scelta : S , puntoDiScelta : P) : boolean
assegna (scelta : S , puntoDiScelta : P)
deassegna (scelta : S , puntoDiScelta : P)
precedentePuntoDiScelta (puntoDiScelta : P) : P
ultimaSceltaAssegnata (puntoDiScelta : P) : S
1
«...»
1
instance
GrigliaGrattacieli5
GrigliaGrattacieli5 ( ) : GrigliaGrattacieli5
crea ( ) : GrigliaGrattacieli5
getDim ( ) : int «...»
getValMax ( ) : int «...»
getValMin ( ) : int «...»
GrigliaGrattacieli
dim : int
altMin : int «...»
griglia : CellaGrattacieli [1..*] «...»
tempPVSx : int «...»
tempPVDx : int «...»
tempPVSu : int «...»
tempPVGiu : int «...»
okSx : boolean «...»
altMax : int «...»
altMaxCurSx : int «...»
nGratSx : int «...»
altMaxCurDx : int «...»
nGratDx : int «...»
altMaxCurSu : int «...»
nGratSu : int «...»
altMaxCurGiu : int «...»
nGratGiu : int «...»
okDx : boolean «...»
okSu : boolean «...»
okGiu : boolean «...»
GrigliaGrattacieli ( ) : GrigliaGrattacieli
getPVSu (col : int) : int «...»
setPVSu (col : int , alt : int) «...»
getPVGiu (col : int) : int «...»
setPVGiu (col : int , alt : int) «...»
getPVSx (riga : int) : int «...»
setPVSx (riga : int , alt : int) «...»
getPVDx (riga : int) : int «...»
setPVDx (riga : int , alt : int) «...»
grigliaValida ( ) : boolean «...»
getVal (riga : int , colonna : int) : int «...»
setVal (riga : int , colonna : int , valore : int) «...»
start (nSolMax : int) : int «...»
reset ( ) «...»
createMemento ( ) : Memento «...»
verificaAllPV ( ) : boolean
primoPuntoDiScelta ( ) : CellaGrattacieli «...»
ultimoPuntoDiScelta ( ) : CellaGrattacieli «...»
prossimaScelta (alt : Integer) : Integer «...»
risolvi ( ) «...»
scriviSoluzione (nr_sol : int) «...»
toString ( ) : String «...»
setMemento (m : Memento) «...»
prossimoPuntoDiScelta (c : CellaGrattacieli , s : Integer) : CellaGrattacieli «...»
precedentePuntoDiScelta (c : CellaGrattacieli) : CellaGrattacieli «...»
primaScelta (c : CellaGrattacieli) : Integer «...»
ultimaScelta (c : CellaGrattacieli) : Integer «...»
assegnabile (alt : Integer , c : CellaGrattacieli) : boolean «...»
assegna (alt : Integer , c : CellaGrattacieli) «...»
deassegna (s : Integer , c : CellaGrattacieli) «...»
ultimaSceltaAssegnata (c : CellaGrattacieli) : Integer «...»

Memento

Memento può anch'esso essere considerato un pattern fondamentale per assolvere ai requisiti di progetto. Difatti viene utilizzato quando si ha necessità di ripristinare lo stato di un oggetto ad uno suo precedente. Ciò richiede di memorizzare gli stati pregressi dello stesso per poterli eventualmente ripristinare. Perciò nel caso specifico che stiamo trattando, si richiede propriamente che il giocatore possa navigare tra tutte le soluzioni ottenute. E proprio tramite *Memento* è avvenuta la definizione dell'oggetto *GrigliaGrattacieliCT,* che si occupa di conservare tutte le soluzioni che l'algoritmo di backtracking ha calcolato e ha ritenuto valide. Ovviamente quest'ultimo mette a disposizione dei metodi con cui *l'originator GrigliaGrattacieli* possa creare gli oggetti *GrigliaGrattacieliMem* gestibili dal *care-taker.*

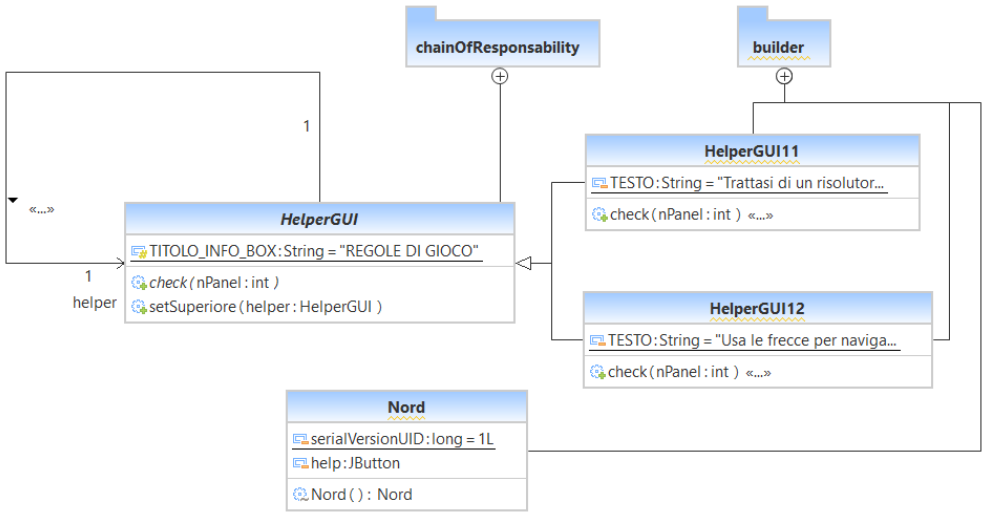
N.B. Il progetto mette anche a disposizione l'oggetto *GrigliaGrattacieliCTSer,* che corrisponde a una versione *Serializable* del *care-taker,* utilizzabile in alternativa a *GrigliaGrattacieliCT.* A diferrenza di quest'ulltimo infatti, la versione serializzabile salva in locale le varie soluzioni ottenute: una diversa e robusta implementazione del gioco potrebbe permettere un ripristino di soluzioni (e quindi griglie) ottenute da esecuzioni precendenti, anche a seguito di *crash* improvvisi dell'applicativo, ad esempio. Per contro però, a causa della natura stessa della logica di risoluzione richiesta (backtracking), latenze importanti possono sorgere nella scrittura di un considerevole numero di soluzioni (il cosiddetto "collo di bottiglia" causato dalla memoria secondaria). Quindi, non avendo particolari indicazioni in questi termini, si è optato, a titolo esemplificativo, per un care taker che lavori esclusivamente a *runtime*.



Chain on Responsability

Il pattern sopra citato permette di separare gli oggetti che invocano richieste, dagli oggetti che le gestiscono. Viene utilizzato il termine catena perché di fatto la richiesta inviata "segue una catena" di oggetti, passando da uno all'altro, finché non trova quello che la gestisce sulla base di una specifica condizione. In *Grattacieli,* questo pattern trova spazio nella corretta gestione delle richieste di *aiuto* generate dal click del tasto *Help* presente nella GUI. Difatti, in base al momento in cui viene premuto, verrà visualizzato un box diverso e relativo alle istruzioni con cui interagire col *panel* attualmente attivo. Tecnicamente succederà che la richiesta

sarà interrogata a catena dai due *helper HelperGUI11* e *HelperGUI12* finché non verrà evasa da quello corretto.



Command

Il design pattern *Command* viene utilizzato quando si ha la necessità di disaccoppiare l'invocazione di un comando dai suoi dettagli implementativi, separando colui che invoca il comando da colui che esegue l'operazione. Viene spesso utilizzato in accoppiata con *Chain of Responsability* e l'esempio più palese (e che vediamo nel progetto con l'implementazione della GUI) è nel framework *Swing.* Nello specifico difatti, *Command* diviene cruciale nella definizione degli *Action Listener* e dei *Key Listener* aggiunti ai vari *JButton* e alle varie *JTextField* della griglia grafica e della *GUI* in generale.

TESTING E *JUnit*

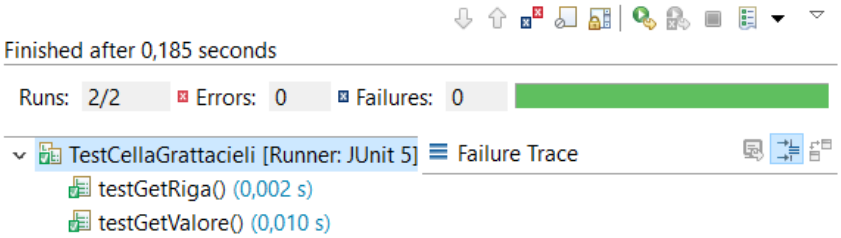
All'interno del progetto è stato utilizzato l'ambiente di testing *JUnit 5* per il linguaggio *Java*, per svolgere il controllo sul corretto funzionamento di alcuni moduli impiegati nel software. Questo strumento consente di scrivere *test* di oggetti e classi in *Java* in modo che siano eseguiti in maniera automatica, e l'esito consiste nella visualizzazione di una barra verde o rossa, a seconda che vada a buon fine o meno.

Un *test case* è una classe composta da uno o più *metodi di test*, che possono venire definiti sia in una logica di test *positiva*, ovvero controllano la corretta esecuzione dei metodi a cui si riferiscono, sia *negativa*, ovvero analizzano il comportamento in casi di funzionamento non ritenuti validi.

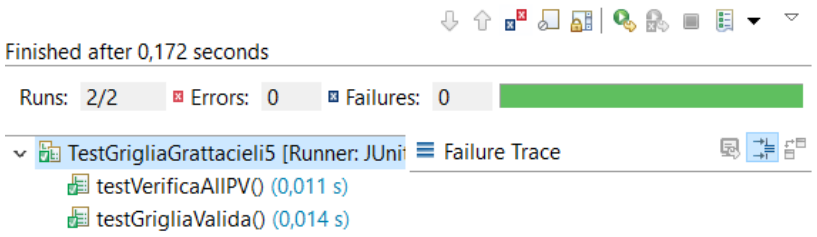
Nel spefico dei test cases definiti per *Grattacieli*, va caratterizzata la funzione *setUp()* accompagnata dal marcatore *@BeforeClass.* Essa difatti è un particolare metodo che grazie all'ambiente specifico di JUnit, verrà sempre eseguita prima di ogni altro test. Difatti, ciascuno di essi verrà avviato in maniera *atomica* senza poter in nessun modo influenzare ogni altro test del test case. I metodi da valutare, ovvero i test, sono anch'essi marcati ma dalla sigla *@Test*.

Si mostrano, a titolo di esempio, i due test cases che troviamo dentro il package *abastractFactory* del *model,* compresi di codice e valutazione dopo l'esecuzione in *JUnit*.

```java
package anno17_18.ingdelsoftware.progetto.model.abstractFactory;

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

class TestCellaGrattacieli {

    private CellaGrattacieli cg;
    private int riga = 1;
    private int colonna = 1;
    private int altezza = 1;

    @BeforeEach
    void setUp() throws Exception {
        cg = new CellaGrattacieli( riga, colonna, altezza );
    }// setUp

    @Test
    void testGetRiga() {
        assertEquals( cg.getRiga(), this.riga );
    }// testGetRiga

    @Test
    void testGetValore() {
        assertEquals( cg.getValore(), this.altezza );
    }// testGetValore

}// TestCellaGrattacieli
```



```java
package anno17_18.ingdelsoftware.progetto.model.abstractFactory;

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

class TestGrigliaGrattacieli5 {

    GrigliaGrattacieli5 gg5;

    @BeforeEach
    void setUp() throws Exception {
        gg5 = GrigliaGrattacieli5.crea();
    }// setUp

    @Test
    void testGrigliaValida() {
        assertTrue( !gg5.grigliaValida() );
    }// testGrigliaValida

    @Test
    void testVerificaAllPV() {
        assertTrue( gg5.verificaAllPV() );
    }// testVerificaAllPV

}// TestGrigliaGrattacieli5
```



ESEMPI DI GIOCO