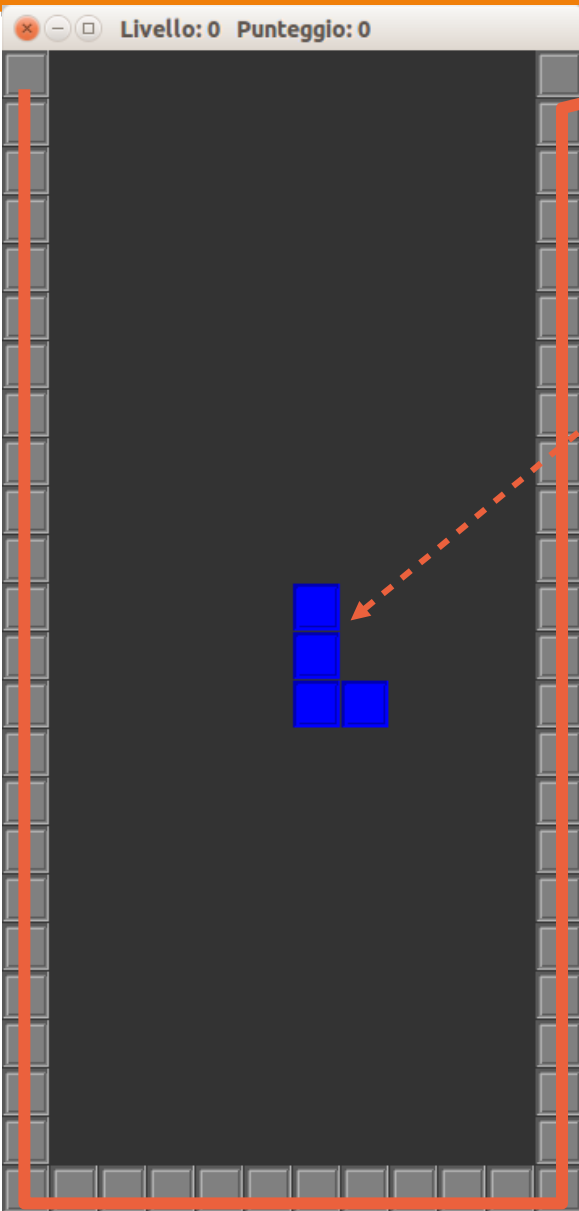


Programmazione Orientata agli Oggetti

Collezioni Insiemi:
Esercitazione Tetris

Sommario

- Tetris: Le regole del gioco
 - I tetramini
- Diagrammi delle Classi
 - Classi di “Alto Livello”
 - Classe **Pozzo** e conoscenti
- Esercizio 1
 - Codificare il criterio di eq. in **Cella**, **Posizione**
- Esercizio 2, 3, 4
 - Unit-testing
 - Completare i metodi di interrogazione del pozzo utilizzando
 - **Set/SortedSet/NavigableSet**
 - **TreeSet**
 - **Comparable/Comparator**
- Riflessione finale

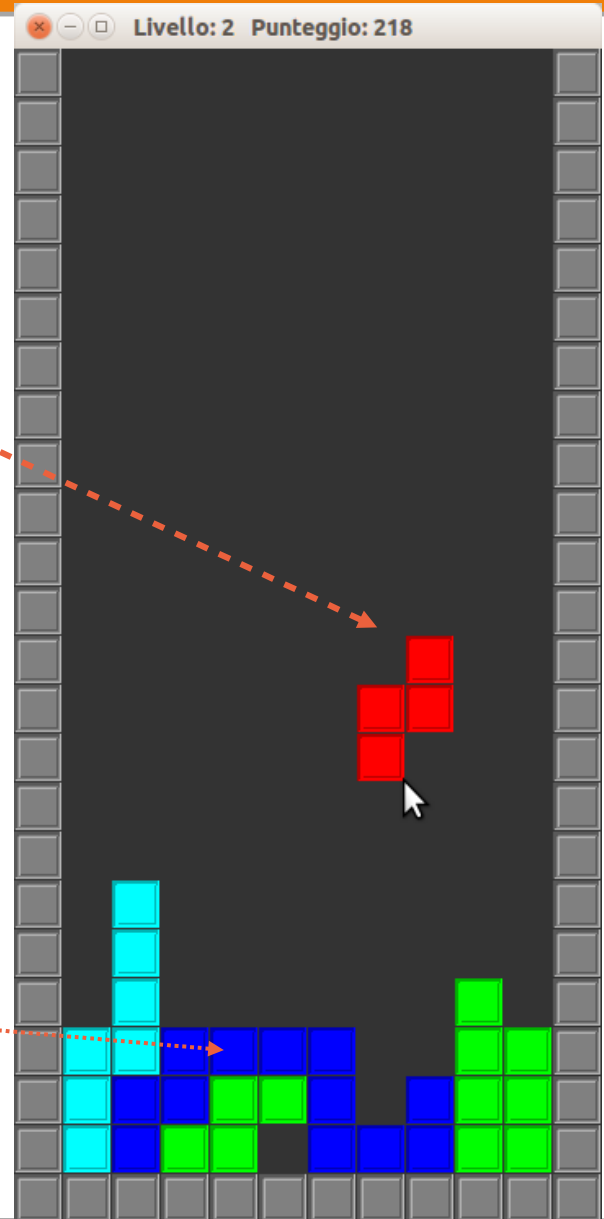


Pozzo

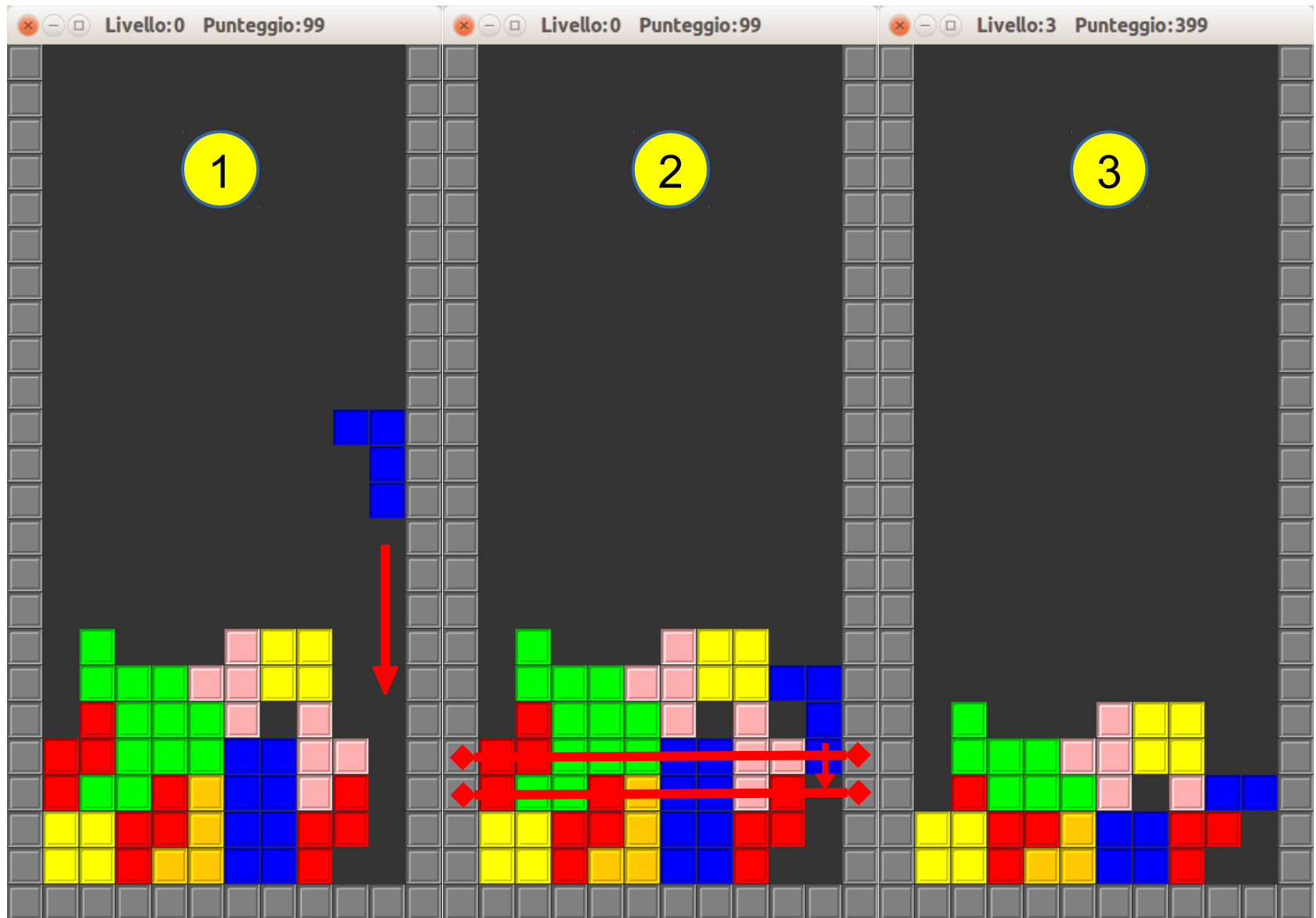
*Bordo del
Pozzo*

*Tetramino
corrente:
4 blocchi*

Blocchi
accumulati sul
fondo del pozzo



Completamento di linee



17 tetramini:

<https://it.wikipedia.org/wiki/Tetramino>

- I (anche detto "barra", "dritto" o "lungo"): quattro quadratini allineati

- J (anche detto "L rovesciata"): una riga di tre quadratini più un quadratino aggiunto sotto a destra

- L: una riga di tre quadratini più un quadratino aggiunto sotto a sinistra. Questo tetramino non è altro che il precedente riflesso, ma non si può passare dall'uno all'altro solo con rotazioni in due dimensioni, per cui esso è chirale

- O (anche detto "quadrato"): quattro quadratini in un quadrato

- S (anche detto "N" o "serpente"): due domino sovrapposti, con il superiore spostato a destra

- Z (anche detto "N rovesciata"): due domino sovrapposti, con il superiore spostato a sinistra. Come nel caso dei pezzi J e L, questi due tetramini sono chirali in due dimensioni e tra di loro speculari

- T: tre quadratini in fila più uno aggiunto sotto, al centro

Tetris: Diagramma delle Classi

- Classi di “alto livello” del package **tetris**

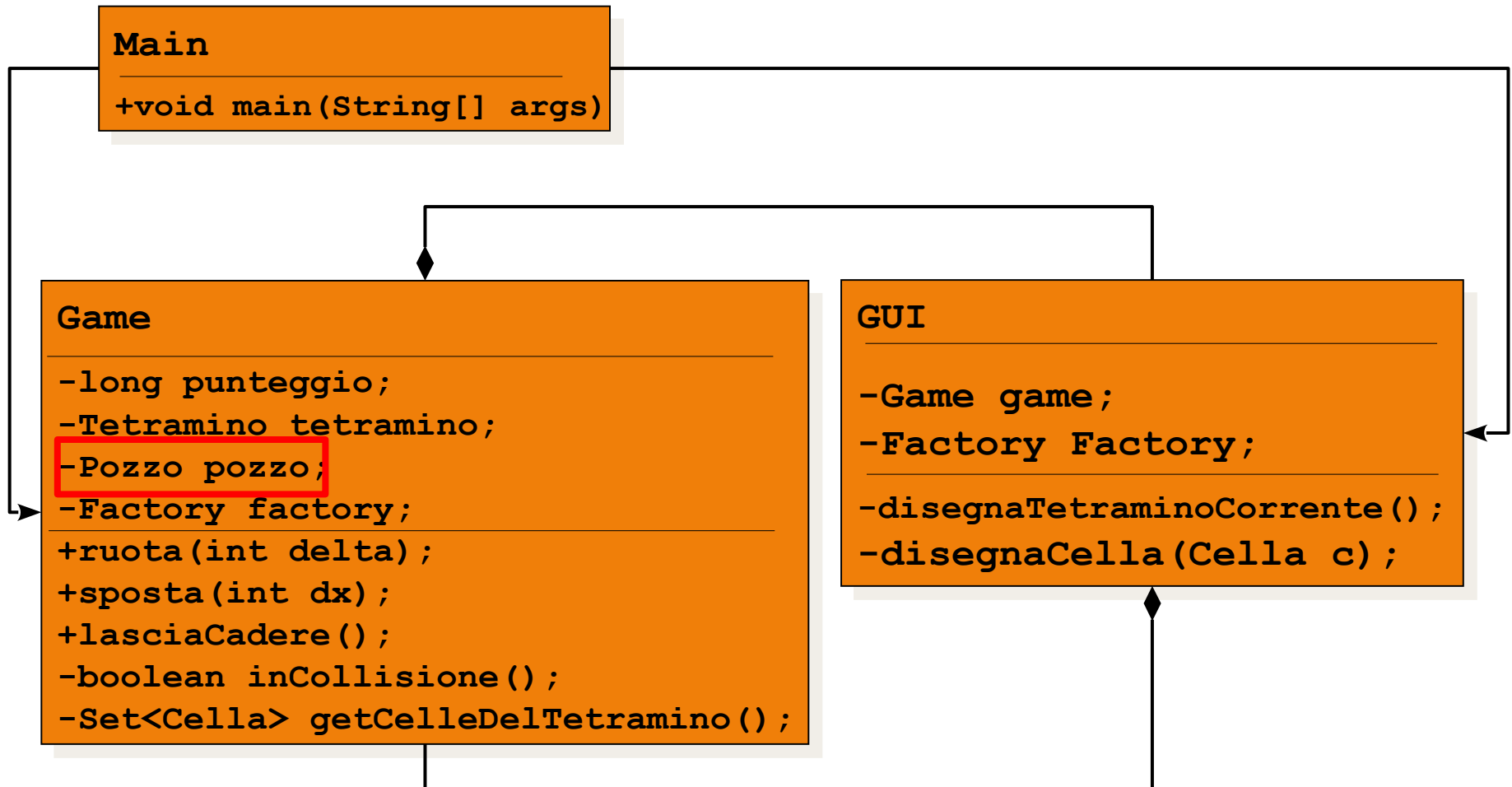
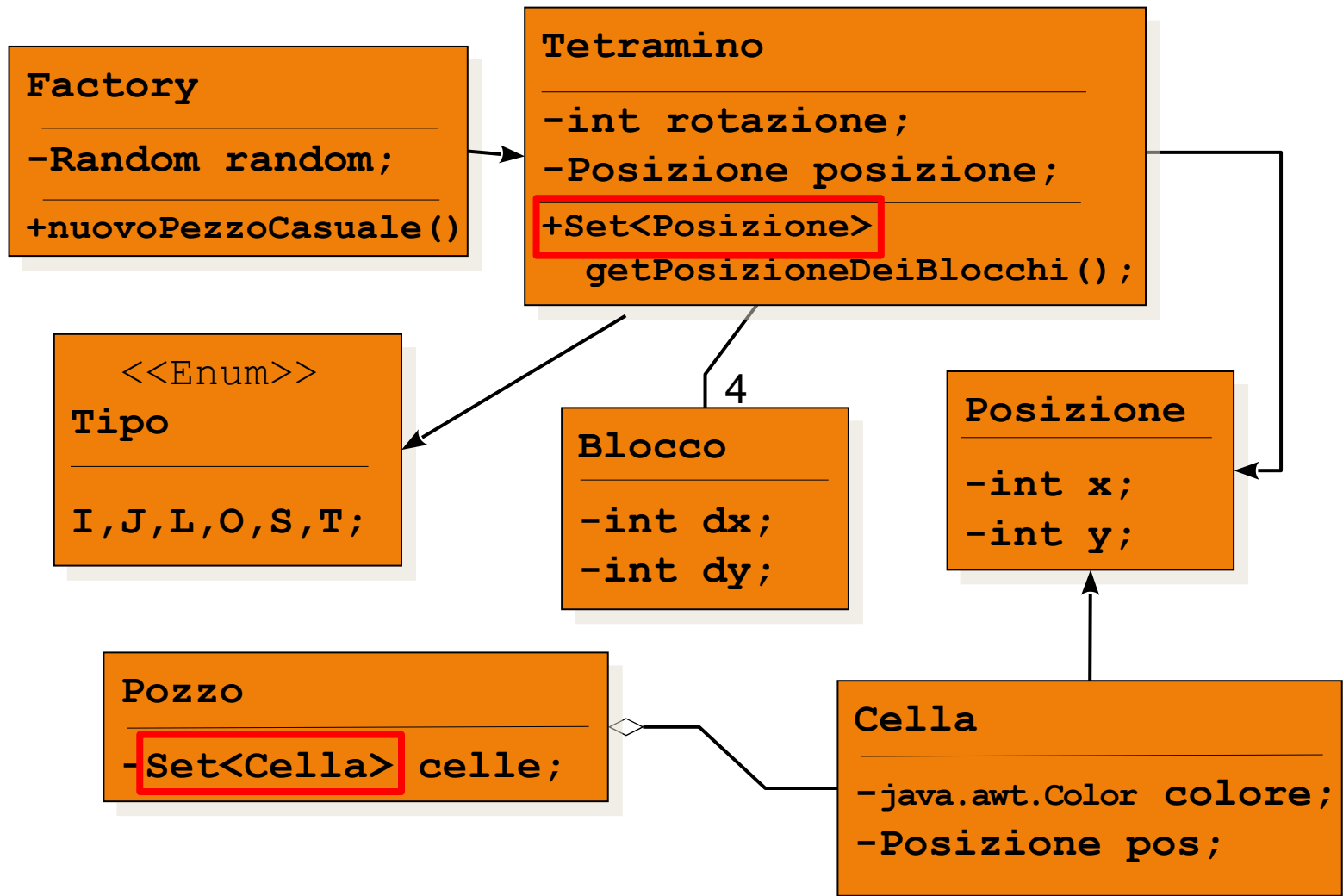
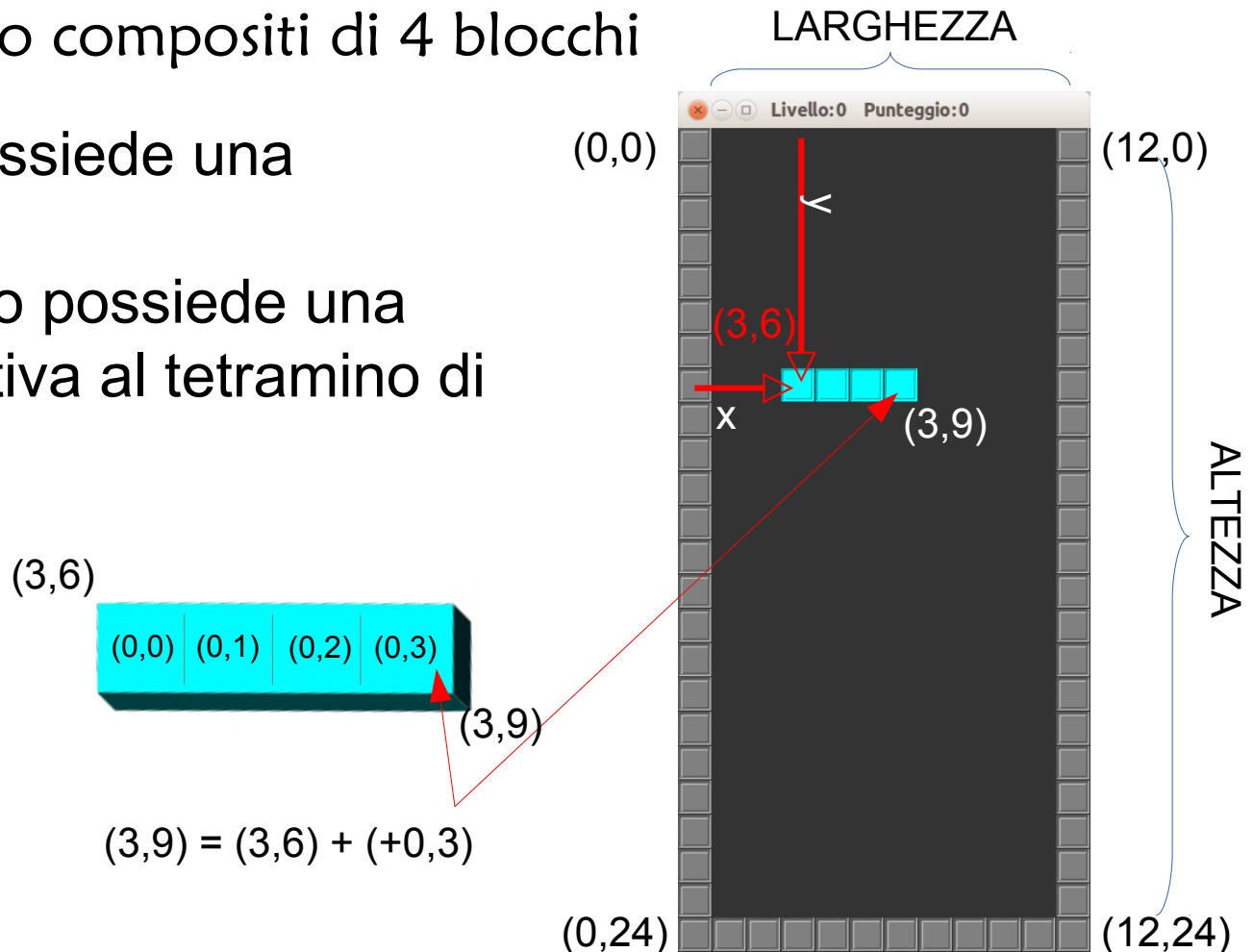


Diagramma delle Classi: Package tetris.tetramino/pozzo



Il Pozzo

- Una collezione di celle, delle quali una parte (quelle di colore grigio scuro) formano il bordo
- I tetramini sono composti di 4 blocchi
- Il tetramino possiede una posizione
- Ciascun blocco possiede una posizione relativa al tetramino di appartenenza
- Una volta depositati, i blocchi diventano celle del pozzo



Posizione.java

```
package tetris.pozzo;
/**
 * Rappresenta una posizione, modellata come coppia di interi
 * ascissa (x) ed ordinata (y), all'interno del {@link Pozzo}.
 */
public class Posizione {

    private int x,y;

    public Posizione(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return this.x; }
    public int getY() { return this.y; }

    public Posizione traslata(int dx, int dy) {
        return new Posizione(getX()+dx, getY()+dy);
    }

    /* DA COMPLETARE */

    @Override
    public String toString() {
        return "("+getX()+", "+getY()+")";
    }

}
```

Cella.java

```
package tetris.pozzo;
import java.awt.Color;
/**
 * Rappresenta una cella contenuta nel {@link Pozzo}.
 * Possiede una {@link Posizione} ed un {@link Color}.
 */
public class Cella {

    private Posizione posizione;

    private Color colore;

    public Cella(int x, int y, Color c) {
        this(new Posizione(x,y),c);
    }
    ...
    public Posizione getPosizione() { return this.posizione; }

    public Color getColore() { return this.colore; }

    public Cella scesaDiUnaRiga() {
        return new Cella(getPosizione().traslata(0, +1),this.getColore());
    }
    /* DA COMPLETARE */
    ...
}
```

Esercizio 1

- a) Le classi **Posizione** e **Cella** sono usate all'interno di insiemi (dentro **Pozzo** e vedi metodo **Tetramino.getPosizioneDeiBlocchi()**): stabilire un criterio di equivalenza per gli oggetti di queste classi
- b) Scrivere dei test di unità per codificare il comportamento atteso dal criterio di equivalenza delle due classi
- c) Implementare il criterio di equivalenza deciso aggiungendo alle due classi opportuni metodi **equals()** / **hashCode()**
- d) Verificare che i test di unità scritti al punto b. risultino soddisfatti

Pozzo.java

- Contiene la logica di gestione delle celle del pozzo
- Consiste, approssimativamente, di tre parti
 - 1) Logica di creazione del pozzo e dei suoi bordi
 - 2) Il metodo che realizza la logica per aggiungere celle quando un tetramino tocca il fondo del pozzo

```
int aggiungiCelleErimuoviRigheCompletate(Set<Cella> celle);
```

- ✓ A sua volta si basa su metodi di interrogazione (>>) e viene invocato da **Game.java** quando un tetramino raggiunge il fondo ed i suoi blocchi diventano oggetti **Cella** del pozzo

3) Metodi di interrogazione

Pozzo.java: Logica di Creazione (1)

```
package tetris.pozzo;

import static tetris.Costanti.COLORE_BORDO;

import java.util.*;
import tetris.tetramino.Tetramino;

public class Pozzo {

    static final public int LARGHEZZA = 12; // in celle, con bordo
    static final public int ALTEZZA    = 24; // in celle, con bordo

    /* serie di metodi factory per creare celle lungo il bordo */

    static final private Cella bordoDX(int riga) {
        return new Cella(LARGHEZZA-1,riga, COLORE_BORDO);
    }

    static final private Cella fondo(int colonna) {
        return new Cella(colonna, ALTEZZA-1, COLORE_BORDO);
    }

    static final private Cella bordoSX(int riga) {
        return new Cella(0,riga, COLORE_BORDO);
    }
}
```

Pozzo.java: Logica di Creazione (2)

```
final private NavigableSet<Cella> celle;

public Pozzo() {
    this(LARGHEZZA, ALTEZZA);
}

public Pozzo(int l, int h) {
    this.celle = /* DA COMPLETARE */
    this.addBordo(l, h);
}

private void addBordo(int l, int h) {
    this.addBordoSX(h);    // lato sx alto h
    this.addBordoDX(h);    // lato dx alto h
    this.addBordoFondo(l); // fondo largo l
}

private void addBordoSX(int h) {
    for (int riga=0; riga<h; riga++) {
        this.celle.add(bordoSX(riga));
    }
}

private void addBordoDX(int h) { ...omissis... }

private void addFondo(int l) {
    for (int colonna=0; colonna<l; colonna++) {
        this.celle.add(fondo(colonna));
    }
}
```

Game.java: invocazione del metodo aggiungiCelleErimuoviRigheCompletate()

```
/**
 * Ferma la caduta del pezzo nel pozzo decomponendo
 * i suoi {@link Blocco} in {@link Cella} del {@link Pozzo}
 *
 * @return il numero di righe complete eliminate
 */
private int fermaCaduta() {
    final Set<Cella> celle = getCelleDelTetramino();
    return
        this.pozzo.aggiungiCelleErimuoviRigheCompletate(celle);
}
```

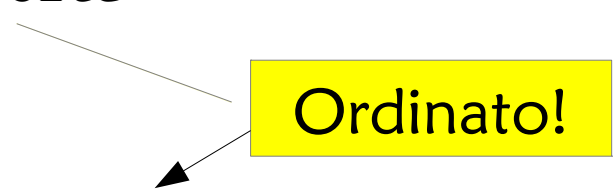
- La classe **Game**, invoca il metodo **fermaCaduta()** non appena il tetramino corrente tocca il fondo del pozzo per fermarne la caduta; quindi passa a generare randomicamente il prossimo tetramino
- Mediante il metodo **getCelleDelTetramino()**, si ottengono le celle che compongono il tetramino corrente da aggiungere al pozzo

Pozzo.java: il metodo aggiungiCelleErimuoviRigheCompletate()

```
/**
 * Aggiunge un insieme di celle e quindi rimuove tutte le righe
 * completate, dal basso verso l'alto, spostando le celle sopra
 * le righe completate di una posizione verso il basso
 * @param celle - un {@link Set} di celle da aggiungere
 * @return il numero di righe completate e rimosse
 */
public int aggiungiCelleErimuoviRigheCompletate(Set<Cella> celle) {
    this.celle.addAll(celle); // aggiungi nuove celle

    /* controlla le linee dal basso verso l'alto (y decrescenti) */
    final NavigableSet<Integer> righeCoinvolte =
        getInsiemeOrdinateY(celle);

    int contatoreRimosse = 0;
    while (!righeCoinvolte.isEmpty()) {
        int rigaPiùInBasso = righeCoinvolte.pollLast();
        if (èCompleta(rigaPiùInBasso)) {
            rimuoviRigaScendendoCelleSopra(rigaPiùInBasso);
            contatoreRimosse++;
        }
    }
    return contatoreRimosse;
}
```



Esercizio 2

Il metodo **getInsiemeOrdinateY()** è il primo dei metodi di interrogazione a supporto del metodo **aggiungiCelleErimuoviRigheCompletate()**

- a) Scrivere una batteria di test di unità *minimali* per codificare il comportamento atteso da questo metodo
- b) Implementare il corpo del metodo **getInsiemeOrdinateY()**
- c) Verificare che i test di unità scritti al punto a. risultino soddisfatti

Metodo di interrogazione: getInsiemeOrdinateY()

```
/**
 *
 * @param inputCelle - un insieme di celle
 *
 * @return l'insieme ({@link NavigableSet}) delle ordinate (y)
 *         delle posizioni delle celle passate. L'insieme è
 *         ordinato crescente.
 *
 * @see {@link Posizione},  {@link Cella}
 *
 */
```

```
NavigableSet<Integer> getInsiemeOrdinateY(
                                Set<Cella> inputCelle) {

    /* DA COMPLETARE */

}
```

Esercizio 3

Il metodo `èCompleta()` (la cui implementazione è fornita nella prossima slide) è un altro dei metodi di interrogazione a supporto del metodo `aggiungiCelleErimuoviRigheCompletate()`

- a) Scrivere una batteria di test di unità *minimali* per codificare il comportamento atteso dal metodo `èCompleta()`
- b) Scrivere una batteria di test di unità *minimali* per codificare il comportamento atteso dal metodo `getCelleDellaRigaSenzaBordo()` su cui il metodo `èCompleta()` si basa. Creare allo scopo dei pozzi *minimali* utilizzando il costruttore `Pozzo(int l, int h)` che permette di specificare le dimensioni del pozzo creato
- c) Implementare, senza utilizzare iterazioni di alcun genere, il corpo del metodo `getCelleDellaRigaSenzaBordo()`
- d) Verificare che i test di unità scritti ai punto a. e b. risultino soddisfatti

Metodo di interrogazione: getCelleDellaRigaSenzaBordo()

```
/**
 * @param y ordinata della riga controllata
 * @return true se e solo se la riga è attualmente completa
 */
public boolean èCompleta(int y) {
    return ( getCelleDellaRigaSenzaBordo(y).size() == LARGHEZZA-2 );
}
```

Bordo escluso!

```
/**
 * @param riga indice di una riga
 * @return l'insieme di celle (escluso i bordi)
 *         della riga di indice dato
 */
Set<Cella> getCelleDellaRigaSenzaBordo(int riga) {
    /* DA COMPLETARE (senza iterazioni) */
}
```

Esercizio 4

Il metodo **rimuoviRigaScendendoCelleSopra()** (la cui implementazione è fornita nella prossima slide) è un altro dei metodi di interrogazione a supporto del metodo

aggiungiCelleErimuoviRigheCompletate()

- a) Scrivere una batteria di test di unità *minimali* per codificare il comportamento atteso dal metodo **rimuoviRigaScendendoCelleSopra()**
- b) Scrivere una batteria di test di unità *minimali* per codificare il comportamento atteso dal metodo **getCelleSopraRigaYdecrecente()** su cui il metodo **rimuoviRigaScendendoCelleSopra()** si basa. Creare allo scopo dei pozzi *minimali* utilizzando il costruttore **Pozzo(int l, int h)** che permette di specificare le dimensioni del pozzo creato
 - ✓ N.B. il metodo **getCelleSopraRigaYdecrecente()** non include le celle del bordo nel risultato
- c) Implementare, senza utilizzare iterazioni di alcun genere, il corpo del metodo **getCelleSopraRigaYdecrecente()**
- d) Verificare che i test di unità scritti ai punto a. e b. risultino soddisfatti

Metodo di interrogazione: getCelleSopraRigaYdecresciente()

```
/** Sposta le celle sopra alla riga di riferimento
 * verso il basso (y risultata incrementato)
 * @param riga - indice della riga di riferimento
 */
void rimuoviRigaScendendoCelleSopra(int riga) {
    this.celle.removeAll(getCelleDellaRigaSenzaBordo(riga));
    for(Cella cella : getCelleSopraRigaYdecresciente(riga)) {
        this.celle.remove(cella);
        this.celle.add(cella.scesaDiUnaRiga());
    }
}

/**
 * @return il @link {@link NavigableSet} di tutte le celle
 * attualmente nel pozzo e sopra la riga specificata;
 * celle ordinate per Y decrescente, e quindi per X.
 * N.B. il risultato non include le celle del bordo! */
NavigableSet<Cella> getCelleSopraRigaYdecresciente(int riga) {
    /* DA COMPLETARE (senza iterazioni) */
}
```

Riflessione finale

- ✓ Confrontare il tempo che si spende facendo debugging nei due casi:
 - ◉ Mediante unit-testing con test minimali
 - ◉ Svolgendo intere partite a Tetris solo per attivare il metodo che si vuole debuggare