Exciting business opportunity!

After dominating the RTP monitoring industry, Voipfuture is looking for new ways to expand the business. A heated discussion ensued and finally another, mostly untapped market was identified: casual gaming! Why not combine our flagship RTP monitoring product “Qrystal” with some addictive gameplay and throw in some micro transactions for good measure ?
After hours of fruitless discussion, one of the team members finally manages to come up with a brand-new new game idea...VoIPTris !

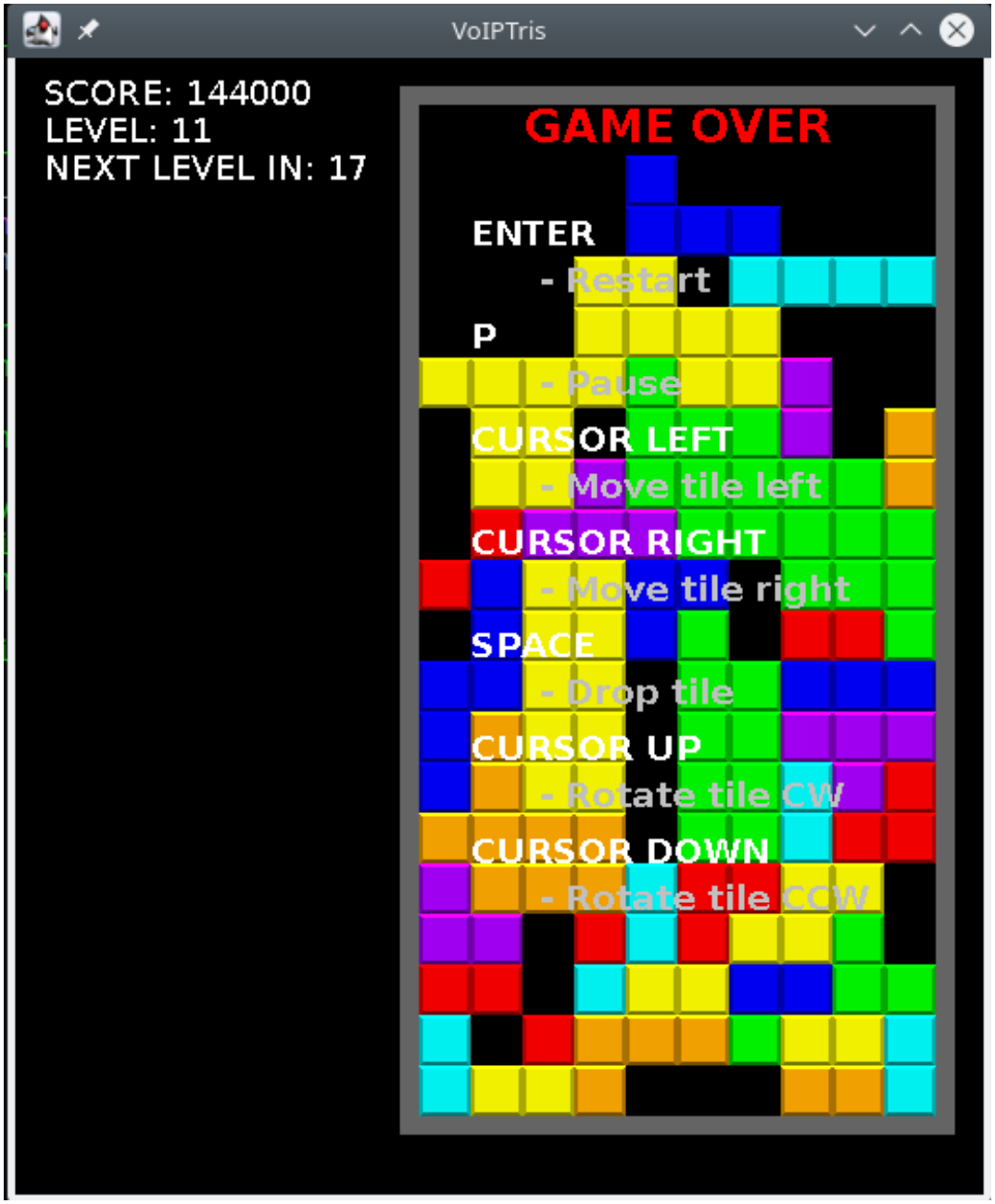

Illustration 1: Totally new game idea

The bad news

A few months later a first version of the game is almost ready. Unfortunately, after the one and only programmer working on the project went on extended parental leave, it was discovered that the source code to the most important part of the game accidently got deleted... the game logic itself. Since it was deemed too risky to release the game without the ability to make bugfixes to the gameplay and you accidently raised your hand (to scratch your head) during the team meeting, it was decided that you have to re-implement the game logic.

The good news

Your collegue conveniently setup a Github repository with the existing game code and you just need to clone this repository, implement the game logic and you're ready to ship !

Your helpful collegue also provided you with the original requirements for the game logic!

Game Logic

The game takes place on a rectangular playing field that is subdivided into grid cells. One of seven possible tiles appears randomly at the very top of the playing field and then slowly drop down until they either hit the bottom of the playing field or another tile. While the current tile is dropping, it may be moved left/right or rotated clockwise/counter-clockwise by 90 degrees. It's also possible to quickly drop the tile (called a "hard drop") but doing so ignores all further user input until the tile has either reached the bottom of the playing field or hit another tile. Once a tile has dropped in place, a new random tile appears at the very top of the playing field. If the newly generated tile intersects with an already existing one at this position, the game is lost. Whenever a row of the playing field is filled completely, it disappears and all tiles that were "suspended" by this row drop down by one row. Completing rows awards points to the player and counts towards the current difficulty level's goal. Once enough rows have been completed, the difficulty level increases, increasing tile drop speed.

Game tiles are placed on a rectangular grid with each tile taking up exactly four grid cells. Here are the available tiles that and the *com.voipfuture.voiptris.api.TileType* enumeration constants associated with them:

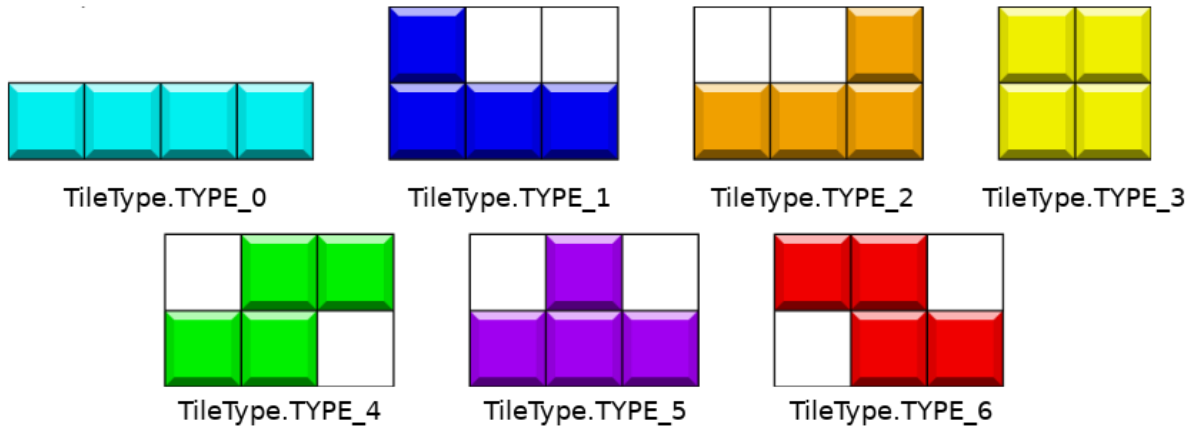

Illustration 2: Game tiles

When a new, randomly chosen tile appears at the very top of the playing field, it must be displayed in exactly the configuration shown here (so it must **not** be rotated in any way). The first populated

grid cell of a tile must be at the center of the very top row of the playing field (because of the fixed grid this will obviously not work for tiles with an odd width, just using *(playingField.width()/2 – tile.width()/2)* as grid cell X coordinate is perfectly fine).

Score Calculation

The player is only awarded points for completed rows. The actual number of points depends on how many rows were completed by dropping a single tile, with the largest number possible being 4 rows (as there is only one tile that spans 4 grid cells in one dimension).

The player is awarded an extra bonus for completing 4 rows multiple times back-to-back.

Base score:

1 completed row = 100
2 completed rows = 300
3 completed rows = 500
4 completed rows = 800 (first time), 1200 (completing 4 rows again back-to-back)

This base score then gets multiplied with the current difficulty level (game starts at difficulty level 1) to yield the final score.

A few examples:

Example #1: Current difficulty level is 3, player did not yet complete any rows at this level
Completing 3 rows would award: 500 x 3 = 1500 points.

Example #2: Current difficulty level is 1, player did not yet complete any rows at this level
Completing 4 rows and then completing 4 rows again back-to-back would award 800+1200 = 2000 points.

Difficulty levels

The game starts at difficulty level 1. Each difficulty level requires the player to complete a slightly higher number of rows than the previous level while also slightly increasing the tile drop speed.

Tile drop speed formula:

```
int timerTicks = Math.max((int) (60 - difficultyLevel*2.5f),1);
```

So the game controller has to move the current tile down by one row every X number of timer ticks (=*IGameController#tick(IUserInput)* invocations).

Number of rows to complete to advance to next level:

```
int rowCount = (int) (10+difficultyLevel*1.5f);
```

Whenever the player has completed this number of rows, the difficulty level increases by one.
Quirk: If the player completed more rows than necessary when crossing the current difficulty level's threshold, the excess rows count towards the next difficulty level (so if the player only needs

to complete two rows to advance but actually completes four, two rows will be carried over as 'completed' to the next difficulty level, slightly speeding up level advancement).

Some hints…
1. This is a simple game from a time when most CPUs couldn't even do integer multiplication… so whatever people did, it had to be simple.
2. For this task I don't care about JavaDoc, unit tests or other things you generally expect from production-quality code.
3. There is no such thing as a specification without gaps and bugs. The important part is being able to recognize those gaps and – in the absence of other information – come up with a sensible solution given the known constraints (for example point 1. above). You don't need to have a perfect solution, you just need to be able to justify why you did things in a certain way.

Your task

Your task is to implement that game logic that will later be exercised by the game framework.

Core of the game logic is an *com.voipfuture.voiptris.api.IGameController* implementation with a proper *tick(IUserInputProvider)* method.

This class gets instantiated dynamically using reflection, it must be inside the Java default package namespace and have a constructor that takes two integers as arguments (the width and height of the playing field).

An example implementation would look like this:

```
import com.voipfuture.voiptris.api.GameState;

import com.voipfuture.voiptris.api.IGameController;
import com.voipfuture.voiptris.api.IPlayingField;
import com.voipfuture.voiptris.api.IuserInputProvider;

public class GameController implements IGameController
{
    public GameController(int playingFieldWidth, int playingFieldHeight) {
         // TODO: Implement me
    }

    @Override
    public void tick(IUserInputProvider inputProvider)
    {
        // TODO: Implement me
    }
    @Override
    public GameState getState()
    {
        return null; // TODO: Implement me
    }
    @Override
    public int getScore()
    {
        return 0; // TODO: Implement me
    }
    @Override
    public int getLevel()
```

```
    {
        return 0; // TODO: Implement me
    }
    @Override
    public int getRowsUntilNextLevel()
    {
        return 0; // TODO: Implement me
    }
    @Override
    public IPlayingField getPlayingField()
    {
        return null; // TODO: Implement me
    }
}
```

The *tick(IUserInputProvider)* method will be invoked every 16 millisecond to process user input (if any) and then advance the game state. Right after the program is started, the initial game state must be *GameState#GAME_OVER* to give the player some breathing room and display the keyboard shortcuts.

Rendering requires access to the playing field's current state so you need to have a *com.voipfuture.voiptris.api.IPlayingField* implementation that is provided to the rendering code by the *IGameController#getPlayingField()* method.

Keyboard input gets translated into com.voipfuture.voiptetris.api.UserInput evens; note that *IGameController#tick()* is only called every 16 ms so a quick player is able to perform multiple key presses during this time. It's up to you how you want to deal with this (only honor the first, honor all of them..). Make sure to call *IUserInputProvider#clearInput()* if you want to "forget" any additional input events that might still be in the queue so things like the 'pause' function get switched on and off reliably.

Please check out the JavaDoc for more detail information (you can find the project at https://github.com/Voipfuture-GmbH/voiptris)

Other requirements

- Do **NOT** modify the Java source of the provided application framework in any way. Any (perceived) restrictions put forward by this framework are part of the task definition.
- No third-party libraries must be used, everything that ships with OpenJDK 11 is fair game
- Your solution must be uploaded as a a **private** repository to a GitHub (www.github.com) account.
- After creating your GitHub account, you can simply fork the repository at https://github.com/Voipfuture-GmbH/voiptris (**make sure to mark it as a private repository afterwards!)** and start coding.
- Make sure to do multiple commits while developing your solution so we can observe your thought process. Do **not** squash your commits.
- The project must build with Apache Maven 3.6.1 (should be the case out-of-the-box if you cloned our repository).