Exercise 3

Group members: Daniel Langbein, Chiheb Eddine Saidi, Artem Semenovykh, Sam Tadjiky

1 IntelliJ refactorings

Use Interface where possible:

This refactoring searches the code for occurrences of a class that can be replaced by the interface that it's implementing.
This refactoring is not represented on Refactoring Guru, but if it were it would probably be found under Dealing with Generalization.

Safe delete:

This refactoring deletes a file and also deletes all references to that file.
Refactoring Guru shows explicit refactoring for codes. Safe deleting a file is therefore not represented.

Migrate packages and classes:

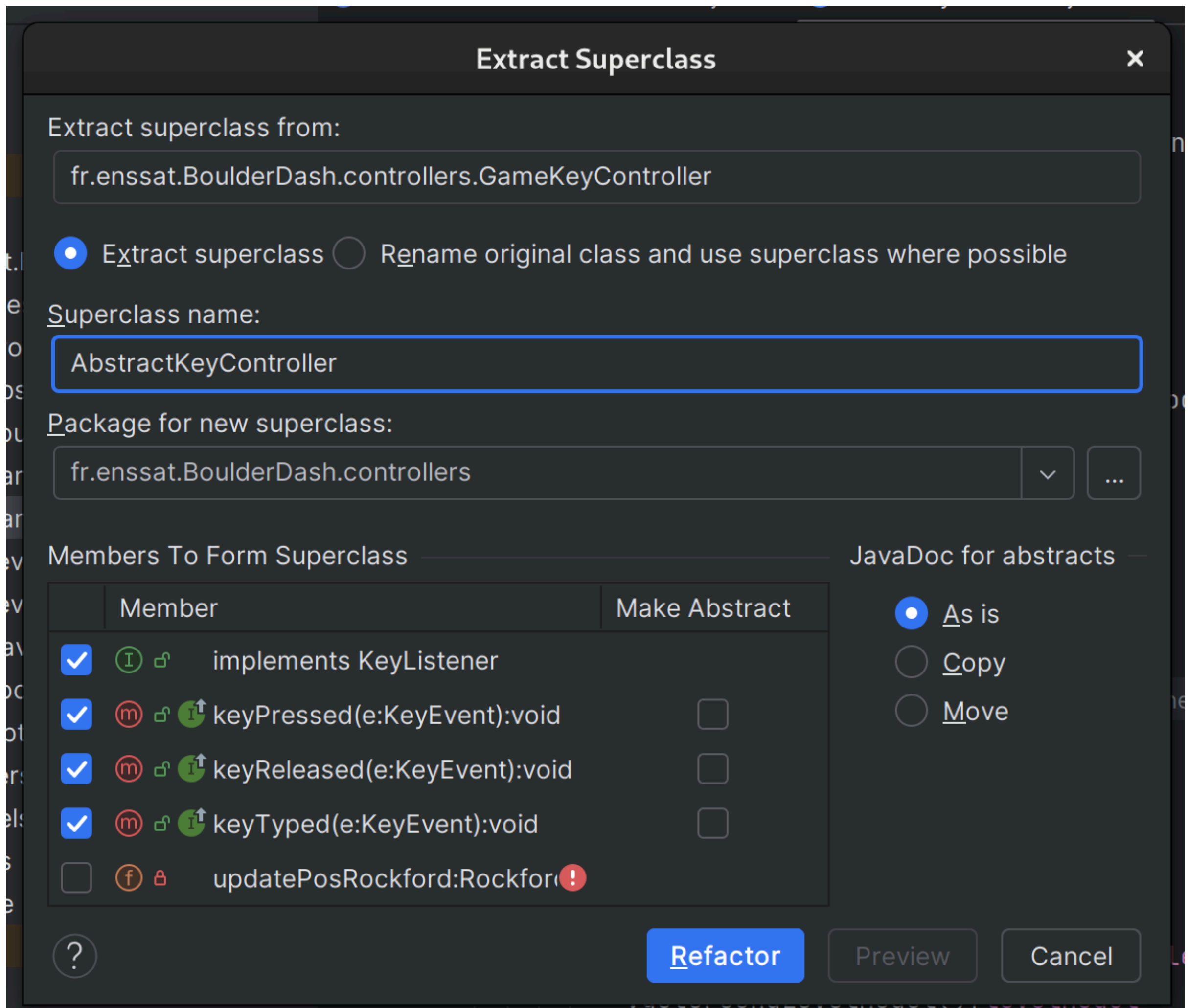
This refactoring updates references to packages and classes according to a migration map which you can define yourself.
Package or class migration is not shown on Refactoring Guru. It is necessary for maintenance, but not for getting rid of code smells.

2 Refactoring the class structure

(a)

- We moved the `levelModel` field up from all except the NavigationBetweenViewController class to a common superclass `AbstractLevelController`. This was done by using IntelliJs “Extract (field to) superclass” refactoring.
- Then we created a singleton of `AudioLoadHelper` as only one instance of this is needed. This reduced the duplicate audioLoadHelper objects within multiple of the controller classes.
- As GameController and LevelEditorController had another common field (named `navigationBetweenViewController` or `nav`), we moved this up to a superclass `AbstractNavController`. This superclass itself is a subclass of the previously introduced class AbstractLevelController. We were using the same refactoring tool of IntelliJ again.
- In the same way we created the super class `AbstractKeyController` for LevelEditorKeyController and GameKeyController. Here we did not move a field, but instead moved many of the functions up to the super class. In the refactoring tool we

did also select to move the “implements KeyListener” up:



- In the superclass, we moved the code of the different switch-cases from the `keyPressed` method into individual methods (e.g. left, right, up, down, etc.). This way we avoided to have two very similar switch-case constructs in both of the sub-classes. Instead, the subclasses are now overriding the behavior of the methods left, right, etc. where needed. In this step we have used the “extract (to method)” tool in IntelliJ.

Class diagram after refactoring:

NavigationBetweenViewController

Member	Type
levelEditorController	LevelEditorController
pickedLevelIdentifier	String
gameController	GameController
menuView	MenuView
getMenuView()	MenuView
getPickedLevelIdentifier()	String
actionPerformed(ActionEvent)	void
setMenuView()	MenuView

AbstractLevelController

Member	Type
levelModel	LevelModel

BoulderAndDiamondController

Member	Type
elementMovingThread	Thread
run()	void
expandWall(int, int)	String
manageFall(int, int)	void
manageFallingObject()	void

RockfordUpdateController

Member	Type
rockfordPositionX	int
rockfordHasMoved	boolean
elementMovingThread	Thread
rockfordPositionY	int
run()	void
moveRockford(int, int)	void

AbstractKeyController

Member	Type
keyPressed(KeyEvent)	void
handleKey(int)	void
right()	void
additionalSteps()	void
keyTyped(KeyEvent)	void
keyReleased(KeyEvent)	void
space()	void
shift()	void
down()	void
up()	void
left()	void

AbstractNavController

Member	Type
nav	NavigationBetweenViewController

LevelEditorKeyController

Member	Type
levelEditorView	LevelEditorView
capLocks	boolean
additionalSteps()	void
left()	void
space()	void
right()	void
up()	void
down()	void
shift()	void

GameController

Member	Type
menuView	MenuView
firstClickOnPause	boolean
gameView	GameView
actionPerformed(ActionEvent)	void
getGameView()	GameView
resetGame(String)	void

GameKeyController

Member	Type
updatePosRockford	RockfordUpdateController
down()	void
keyReleased(KeyEvent)	void
right()	void
up()	void
left()	void

LevelEditorController

Member	Type
levelEditorView	LevelEditorView
actionPerformed(ActionEvent)	void
getLevelEditorView()	LevelEditorView

(b)

The priority field determines what happens when Rockford (priority = 1) collides with another object (priority in [0,2,3]): if priority is lower than 1 the Rockfold can destroy the object and take its place. Otherwise the object will block his movement. Since every class has its own priority across all instances (static), it is unnecessary to declare the priority field in the super class as each subclass will redefine it either way into a static field. Thus we can use the Push Down Field refactoring

3 Refactoring a method

a) Method rename

Looking at the code, this method manages the falling behavior and interactions of boulders and diamonds with Character and environment. The method handles:

- Boulder/diamond falling straight down (`makeThisDisplayableElementFall(x,y)`)
- Boulder sliding off other boulders (`makeThisBoulderSlideLeft(x,y)`)
- Boulder/diamond crushing Rockford (`playSound("die")`)
- Boulder interactions with magic walls (`else if (spriteNameBelow == "magicwall")`)
- Boulder/diamond destroying brick walls (`exploseThisBrickWall(x, y)`)
- Boulder being pushed by Rockford (`moveThisBoulderToRight(x, y)`)

Based on this, a suitable name for this method would be `manageFallingObjectBehavior` - because method shows how objects (boulders and diamonds) which could fall, behave during interaction

We can fix the entire structure and references using the "Rename" refactoring function

b) Falling queries

To get rid of the constant call of the same element in the if clause, we can create a variable in which we will put the necessary method.
The refactoring function "Introduce Variable" helps us with this task - it creates a variable based on the repetition given to it

```
DisplayableElementModel displayableElementModel =
this.levelModel.getGroundLevelModel()[x][y];
```

c) Methods for getGrounfLevelModel

In fact, we reuse one method "get element" several times in different variations. First, we move the method out of the "`getGroundLevelModel`" line using the Extract Method refactoring.

```
public DisplayableElementModel getElement(int x, int y) {
    return this.levelModel.getGroundLevelModel()[x][y];
}
```

Then when extracting other methods with a change in position on the axes, we reuse the already created method, replacing the signature and output

```
public DisplayableElementModel getElementBelow(int x, int y) {
   return getElement(x,y + 1);
}
```

Finally we Move Methods to the LevelMethod class, since this data originally belongs to it.

This is result in LevelMethod:

```
public DisplayableElementModel[][] getGroundLevelModel() {
   return groundGrid;
}

public DisplayableElementModel getElement(int x, int y) {
   return getGroundLevelModel()[x][y];
}

public DisplayableElementModel getElementBelow(int x, int y) {
   return getElement(x,y + 1);
}

public DisplayableElementModel getElementBelowLeft(int x, int y) {
   return getElement(x - 1, y + 1);
}

public DisplayableElementModel getElementBelowRight(int x, int y) {
   return getElement(x + 1, y + 1);
}

public DisplayableElementModel getElementRight(int x, int y) {
   return getElement(x + 1, y);
}

public DisplayableElementModel getElementLeft(int x, int y) {
   return getElement(x - 1, y );
}

public DisplayableElementModel getElementTwoBelow(int x, int y) {
   return getElement(x , y + 2);
}
```

In the BolderAndDiamondCollector class we just call methods, specifying only x and y without additional calculations

d) Refactoring the manageFallingObjectBehavior

We can see that the aforementioned method is too long and handles many behaviors. We can refactor the method by breaking it down into smaller methods that each one handles one logic at time using the Extract Method Refactoring from IntelliJ

```java
private void manageFallingObjectBehavior(int x, int y) {  1 usage  OnlineSam *
    // Get surrounding elements
    DisplayableElementModel element = this.levelModel.getElement(x, y);
    DisplayableElementModel elementTwoBelow = this.levelModel.getElementTwoBelow(x, y);
    DisplayableElementModel elementBelow = this.levelModel.getElementBelow(x, y);
    DisplayableElementModel elementLeft  = this.levelModel.getElementLeft(x, y);
    DisplayableElementModel elementRight = this.levelModel.getElementRight(x, y);

    String spriteNameBelow = elementBelow.getSpriteName();
    String spriteNameLeft  = elementLeft.getSpriteName();
    String spriteNameRight = elementRight.getSpriteName();

    // Handle different cases based on the element's surroundings
    handleFalling(x, y, spriteNameBelow, element);
    handleBoulder(x, y, spriteNameBelow, element, elementTwoBelow);
    handleRockfordCollision(x, y, spriteNameBelow, element);
    handleMagicWall(x, y, spriteNameBelow, element, elementTwoBelow);
    handleDestructibleWalls(x, y, elementBelow, element);
    handleRockfordMovement(x, y,element, spriteNameLeft, spriteNameRight, elementLeft, elementRight);
}
```

```java
private void handleFalling(int x, int y, String spriteNameBelow, DisplayableElementModel element) {  1 usage  On
    if (spriteNameBelow.equals("black")) {...}
}

private void handleBoulder(int x, int y, String spriteNameBelow, DisplayableElementModel element, DisplayableEl

private void handleRockfordCollision(int x, int y, String spriteNameBelow, DisplayableElementModel element) {..

private void handleMagicWall(int x, int y, String spriteNameBelow, DisplayableElementModel element, Displayable

private void handleDestructibleWalls(int x, int y, DisplayableElementModel elementBelow, DisplayableElementMode

private void handleRockfordMovement(int x, int y,DisplayableElementModel element ,String spriteNameLeft, Strin
```