Exercise 4

1 Encapsulate Classes with Factory

c)

Is the builder pattern suitable here instead of a factory pattern?

No, it is not suitable. Objects of the DisplayableElementModel subclasses are always created in the same way except for BoulderModel, but there we have just one parameter. So the process to create objects is way too simple. It is not worth it to create a separate builder class for each of the subclasses. The code is more readable and simple as is with one short constructor for each subclass.

What are the differences between the two design patterns?

Both, the factory and the builder pattern move the knowledge to create objects to a different class (e.g. from subclasses to abstract superclass or into a separate builder class).

A factory class often creates objects of multiple types whereas, with the builder pattern there has to be a separate builder class for each object type.

If object creation is really complex, then the builder pattern might be more suitable. It has a different approach to object creation where the process is split up into different build steps. This way a client can adjust some of the steps. He can e.g. create a subclass of just one of the build steps and then use that during the build process of the whole object.

With the factory pattern on the other hand, the whole logic of object creation is part of one factory class. Here it is more difficult to adjust just some parts of the build process. (It is still possible by extendingthe factory in a subclass, but individual build steps can’t be adjusted that easily.)

When is it suitable to use the builder pattern?

For example during the creation of a database connection: Many configurations and parameters are at play thus making the creation of a database connector complex and encumbrance. In this instance a builder pattern could be very beneficial: In multiple build steps small bundles of related parameters can be specified. This way the client has still full control but does not need to specify a large number of parameters at once.

When is it suitable to use the factory pattern?

A factory comes in handy if we have different specific products that we want to create where the creation process of each product itself is not too complex. Here the factory can provide multiple create methods with readable names for the different products (e.g. same type but

with different state or different subtypes). All of these products implement a specific interface.

Example: We have an interface called Car and a couple of classes which implement this interface. A CarFactory can have the methods createElectircCar, createSportsCar, createSUV, createFamilyCar.

With a factory the client just needs to know one class - the factory - (and the interface - Car) to create different objects. This makes the factory easier to use. This is great as long as the client does not need to configure parts of the build process!

2 Singleton pattern

Why is this class a good candidate for Singleton?

The `ModelConvertHelper` class has only two methods:

1. **toModel()**: This method takes a string representation of a game element ("rockford", "boulder", etc) and returns the corresponding `DisplayableElementModel` object.
2. **toString()**: This method takes a `DisplayableElementModel` object and returns its string representation.

It makes this class perfect for the Singleton pattern, because it offers a function to convert between string names and their model objects without storing any data. Having multiple instances would not be necessary. Using the Singleton pattern we save resources and ensure consistent behavior - there's only one `ModelConvertHelper` object throughout the entire application

Implementation

The first thing we have to do is create an instance and add `getInstance()` method

```
/**
* Holds the single instance of the class.
*/
private static ModelConvertHelper instance = null;
```

```
/**
* Gets the single instance of the ModelConvertHelper.
* Creates the instance if it doesn't exist.
*
* @return  The single instance of the ModelConvertHelper.
*/
public static synchronized ModelConvertHelper getInstance() {
   if (instance == null) {
       instance = new ModelConvertHelper();
   }
   return instance;
}
```

Now we can implement a new Singleton class and use it instead of the regular one. `ModelConvertHelper` is used in `LevelModel` and `LevelLoadHelper` classes. Let's begin from LevelModel

Firstly, we initialize `modelConverter` field with `getInstance()`

```
/**
* Gets the single instance of the ModelConvertHelper.
*/
private ModelConvertHelper modelConverter =
ModelConvertHelper.getInstance();
```

Then we go into `triggerBlockChange()` - it triggers a block change and uses ModelConverterHelper to create different types of elements and place them into the game grid at the cursor's position (with offset of +1 for both x and y coordinates).

We can remove instantiation of the new ModelConvertHelper and replace it with the call of `toModel()` method from existing instance. Everything else remain the same

```
try {
   DisplayableElementModel targetModel = modelConverter.toModel(blockValue,
false);

   // Apply new model in place of cursor
         this.groundGrid[xPos + 1][yPos + 1] = targetModel;

         // Disable cursor (important)
         //this.setShowCursor(false);
} catch (UnknownModelException e) {
   e.printStackTrace();
}
```

In the LevelLoadHelper changed almost the same - `new ModelConvertHelper()` was replaced with `ModelConvertHelper.getInstance()` to use the singleton instance

3 Multiple constructors

a) The class LevelModel has 3 constructors. The first constructor has 3 parameters and is the most general one. The second constructor has 2 parameters and is using the first constructor → Chain Constructors. The last constructor only has 1 parameter and is not using the first (general) constructor → Not chained

   In order to have the third constructor be chained as well we would need an even more general constructor than the first one. Creating a large general constructor just for one use seems unnecessary. It's better to just use the existing constructor and overwrite some of the default values.

b) As mentioned in a) the first constructor is the most general one. So this constructor becomes private. We will refactor the other 2 constructors to createMethods and we create a third createMethod to replace the occurrences of the general constructor outside of the class.