Diary 2

Name:	Chingiz Kenshimov,	Marzhan Oshanova
Username:	ck2612,	mo1412
Class:	a5,	s5

Lecture title: Mock objects.

Lecture notes:

Key points from the lecture:

- *Test-driven Development gives us effective feedback about the quality of our code.* During the development process we constantly try to simplify the code as much as it possible, to make sure that it is easy to read and understand. The TDD helps us reach this simplicity by putting the development process into iterations and implementing a thin slice of functionality on each iteration. We can judge about quality of our design ideas by starting to write each feature with an acceptance test, and if it turns out to be difficult and we cannot clarify our ideas in the test method, that means we have to review our design.
- *Messages passed between objects are more important than objects themselves.* To use TDD approach effectively we have to see the system as dynamic structure of objects that communicate by passing messages to each other, rather than as static description of objects. The main idea of TDD is to push the domain model "into the gaps between objects, communication protocols" (Freeman and Price, 2010, p. 67). Therefore we should focus not on the internal state of the object, but on how it communicates with its peers by calling their methods.
- *Mock objects help us to simulate the behavior of the peers.* In unit tests, we usually face up with the fact that the test class is dependent on data from external sources, the state of which we can not control(for example: databases or unavailable service, etc.). In these cases, we can use mock objects which give us opportunity to test class in isolation from external dependencies. We can also use mock objects even to simulate peers that doesn't exist, by defining them as interfaces, to ensure that the roles they play are correctly fit to our design.

This lecture as an extension of previous lecture about testing, introduced us a concept of mocking objects. While doing tutorial exercise, we realized that mock objects are powerful and effective tool that helps to avoid creating fake objects for testing purposes, which in turn can save time of developers. For example, in the previous tutorial exercise instead of creating fake object for JPhotoShow interface, we could create a mock object, set expectations and verify it in the end.

Tutorial exercise:

In the tutorial exercise we needed to implement basic functionality of "Audio Tourist Guide", which consists of five significant behaviors, using Test-Driven Development approach. Luckily, we were given a "walking skeleton" (Freeman and Price, 2010, p. 32) and were able to start writing tests right away.

Implementing the first feature implies a request for the track from media library and playing this track. Because `MediaLibrary` interface was empty, we added a method `getTrackForLocation()` that returns a `Track` for provided `Location`. In the test method we wrote expectations for exactly one calling of `mediaLibrary.getTrackForLocation()` and one calling of `mediaPlayer.Play()`. We implemented the behavior just by calling these two methods one after another in the `locationChanged()` method of the `AudioGuide` class. We did not add any other conditions that check whether location is initial, because in this case it is implicit. In the stage of refactoring we just

extracted code, that searches for the track and plays it, into the new method, to keep code clear and assuming that in the future `locationChanged()` method will be stuffed with lots of code that responsible for other behaviors.

The second iteration was about adding behavior that does not play a track if media player plays another track, but plays it automatically after current track is finished. After writing tests and implementing the feature, we noticed that two methods `mediaLibrary.getTrackForLocation()` and `mediaPlayer.Play()` are always called together. Following the "composite simpler than the sum of its parts" principle (Freeman and Price, 2010, p. 53) we moved two related objects (`mediaPlayer` and `mediaLibrary`) into a containing object, called `AudioGuidePlayer`. Having new collaborating object `AudioGuidePlayer`, that has `playTrackForLocation()`, `isActive()` and `setNextTrackForPlaying()` methods, we were able to write tests using mock implementation of the new composite object and work on a higher level of abstraction. After these refactoring, test method of the second feature significantly decreased in the size and became easier to understand. *(To be specific about changes we made, `AudioGuide` class does not implement `MediaPlayerListener` interface anymore, because now it is not responsible for collaboration with `MediaPlayer`. Instead, we introduced new class that responsible for this and implement both `MediaPlayerListener` and `AudioGuidePlayer` interfaces. We, of course, also tested it, after had finished testing `AudioGuide` class.)*

Next thing, that worth mentioning, appeared during implementation of the fourth feature. According to the coding conventions in our project, we are not allowed to return `null` reference for `mediaLibrary.getTrackForLocation()` method in case of absence of a track for given location. To overcome this problem we considered two approaches. First, throw an exception when calling a `getTrackForLocation()` method, if appropriate track doesn't exist, and define new method `isTrackExists(Location)` in the `MediaLibrary` class, that we can call before trying to access the track. Second, we can use *Null Object Pattern* (Woolf, 1998), if we want be able to easily change the 'do nothing' behavior later. We chose the second approach, because it is predictable that later we might be asked to play some standard track that reports about absence of information for the location. We implemented this approach by creating an `EmptyTrack` class (that implements `Track` interface) and returning the instance of the class, if the appropriate track for location is missing.

To implement the last feature we had to store the list of visited locations somewhere. First we stored it directly in the `AudioGuide` class. But then noticed that the test method contains too many expectations and is not easy to understand. Moreover, we found ourselves not able to clearly describe responsibilities of the `AudioGuide` class ('to play the appropriate tracks for locations' and 'to store visited locations'). It suggested us that we are going to break 'single responsibility' principle (Freeman and Price, 2010, p. 51). Therefore, we implemented the role of storing visited locations in the new collaborating object.

Bibliography

Freeman, S. and Pryce, N. (2010). Growing Object-Oriented Software, Guided by Tests. Boston: Pearson Education, Inc.

Woolf, Bobby. "Null Object." In: Pattern Languages of Program Design 3. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998, http://www.cse.wustl.edu/~schmidt/PLoP-96/woolf1.ps.gz.