Test2

Enigma / Team 29

Michael Beal
Lucas Solomon
Thomas Auburn
Ryan Early
Inna Strogonova

Daniel Pearce

Methods and approaches

For software testing we chose a combination of black-box and white-box testing using Junit framework, alongside with Mockito and PowerMock testing frameworks. Black-box testing method allowed us to test and examine the game's functionality without taking into account its internal structure or code. Additionally, this technique involves building test cases based on the product's specifications, which helped to analyse how well our software meets the given requirements. White-box testing, on the other hand, helped to test and analyse the game's internal structure, i.e., find bugs and inconsistencies that cannot be detected from the user's point of view. This method is appropriate, as it allows to test and verify code components written by the other team's members.

Using JUnit framework, we were able to create a number of automated tests for individual units of the game's code – classes and their methods. It allowed us to make sure that any new modifications in the software have not affected the rest of the code.

As a part of software testing, we also used Mockito and PowerMock testing frameworks which provide functionality to build and perform tests in isolation from the rest of the system, as well as to reduce interdependence between system's units. More specifically, Mockito allowed us to create mocks of untested dependencies, as well as stub their methods to simulate a specific behaviour, e. g, to return a specific value on any input. This helped to test game's components on different inputs and in different scenarios avoiding instantiation of their dependencies whenever possible.

Additionally, we used PowerMock - another testing tool with more powerful capabilities that, unlike Mockito, allows mocking of static and private methods, setting private fields and more. In our project, we used PowerMock to inject specific values into private fields of mocked objects which helped us to test the previous team's code where it would not be possible with only standard JUnit and Mockito libraries.

Alongside with automated tests, we also performed a series of manual tests – mainly as a part of black-box testing. Manual testing helped to analyse, and test game's functionalities related to specific requirements, including presence of various rooms on the map and aesthetic appeal of the game's graphics. These game's features as well as UI related aspects, e. g. presence of the demo mode, cannot be tested or analysed by simple unit testing, since there is no specific input or expected output that can confirm the correctness or realization of given requirements.

Testing results

Automated tests

Test case ID	Related requirements	Description	Input	Expected output	Actual output	Pass/ Fail
1	FR_MAP	Checks that the method *checkForCollision()* of the class *Collision* correctly detects collision if there is an obstacle to the left of the sprite, given the sprite's velocity < 0 on x axis.	velocity vector (-1, 0)	sprite cannot move to the left, returns velocity (0, 0)	velocity (0, 0)	Pass
2	FR_MAP	Checks that the method *checkForCollision()* of the class *Collision* correctly detects collision if there is an obstacle below the sprite, given the sprite's velocity < 0 on y axis.	velocity vector (0, -1)	sprite cannot move downwards, returns velocity (0, 0)	velocity (0, 0)	Pass
3	FR_TELEPORT	Tests *teleport()* method of the class *Player.* When the method is called, the player is teleported to the farthest of the teleporters.	position of the player is (1712, 3632)	player's position is changed to (1712, 2448)	player's position is (1712, 2448)	Pass
4	FR_HEAL	Tests *heal()* method of the class *Player* when healing by a specific amount.	player's health is *20* (injured) while healing by *30*	player's health is *50*	player's health is *50*	Pass
5	FR_HEAL	Tests *heal()* method of the class *Player*. Tests that the player's health cannot be >100 after healing.	player's health is *80* while healing by *30*	player's health is *100*	player's health is *100*	Pass
6	FR_HEAL	Tests *heal()* method of the class *Player*. Tests that the player's health is fully restored when no arguments are given.	player's health is *20*	player's health is *100*	player's health is *100*	Pass
7	UR_ABILITI ES	Tests *takeDamage()* method of the class *Player* which decreases player's health by a given amount.	player's health is *100*; *20* as an	player's health is *80*	player's health is *80*	Pass

			argument for *takeDamage()*			
8	FR_ENDGAME	Tests *gameOver()* method of the class *PlayScreen* when the player's health is <= 0.	player's health is *0*	*gameOver()* returns *true*	*gameOver()* returns *true*	Pass
9	FR_ENDGAME	Tests *gameOver()* method of the class *PlayScreen* when *crewmateCount > maxIncorrectArrests*, meaning player has made more incorrect arrests than allowed	*maxIncorrectArrests* = 4, *crewmateCount* = 5	*gameOver()* returns *true*	*gameOver()* returns *true*	Pass
10	FR_ENDGAME	Tests *gameOver()* method of the class *PlayerScreen* when all the above conditions return false and *destroyedKeySystemsCount()* <15.	*destroyedKeySystemsCount()* = *0*	*gameOver()* returns *false*	*gameOver()* returns *false*	Pass
11	FR_ENDGAME	Tests *gameWin()* method of the class *PlayerScreen* which returns true when the list of infiltrators is empty.	*NPCCretor.infiltrators* is empty	*gameWin()* returns *true*	*gameWin()* returns *true*	Pass
12		Tests that the method *beingDestroyedKeysystemsCount()* of the class *KeySystemManager* returns the right number of key systems that are being destroyed.	two out of three key systems are being destroyed	returns *2*	returns *2*	Pass
13		Tests that the method *destroyedKeysystemsCount()* of the class *KeySystemManager* returns the right number of destroyed key systems	two out of three key systems have been destroyed	returns *2*	returns *2*	Pass
14		Tests *createInfiltratorSprites()* and *selectSprite()* methods of the class *Infiltrator*.	calling *selectSprite()* with inputs *1.5, 10 and 15 consecutively*	correct sprites have been selected based on chance values	correct sprites have been selected based on chance values	Pass

15		Tests *createCrewSprites()* and *selectSprite()* methods of the class *CrewMemebers*.	calling *selectSprite()* with inputs *0.1, 1.5, 10 and 15 consecutively*	correct sprites have been selected based on chance values	correct sprites have been selected based on chance values	Pass

Manual tests

Test case ID	**Related requirements**	**Scenario**	**Steps**	**Expected results**	**Actual results**	**Pass/ Fail**
16	FR_ROOMS	Try to access different rooms	1. Launch the game; 2. Press 'Play' button; 3. Choose difficulty mode; 4. Move throughout the map using specified keys; 5. Try to enter/exit different rooms on the map in order;	Player can access all rooms	Player can access all rooms	Pass
17	FR_PAUSE_SCREEN	Try to pause the game	1. Launch the game Auber; 2. Press 'Play' button; 3. Choose difficulty mode; 4. Make sure the game has started; 5. Press 'ESC' button;	Pause screen is shown	Pause screen is shown	Pass
18	FR_DEMO	Try to access the demo after starting the game	1. Launch the game Auber; 2. Press 'Demo' button; 3. Watch the demo;	The demo has shown up	The demo has shown up	Pass
19	NFR_AESTHETICS	Make sure that game's graphics don't affect the gameplay	1. Launch the game Auber 2. Press 'Play' button 3. Choose difficulty mode; 4. Move throughout the map paying attention to graphics and visual style of the game	Game is aesthetically appealing; npc's, key systems and teleporters are clearly seen and recognizable	Game is aesthetically appealing; npc's, key systems and teleporters are clearly seen and recognizable	Pass

- No tests failed - tests are complete and correct.

Evidence of testing

https://enigmagroup29.github.io/auber-website/evidence/

JUnit test results: https://enigmagroup29.github.io/auber-website/tests.html