SOFE 3980U
Software Quality
Winter 2020

Assignment 2 | Software Process and Test Automation

Professor: Akramul Azin, PhD

Student: Emil Ilnicki (100659072)

Introduction

In this assignment we are tasked with developing good quality software by choosing a particular software engineering process such as agile, waterfall, incremental etc. Once we have chosen a process we have to develop a game and extend or improve its functionalities. After developing said game we then must write automated tests for the software.

Methodology & Model

For this given project I have decided to use the Waterfall (V) model (Appendix A) approach to solving this problem. The Waterfall V model follows much of what the original Waterfall model does such as requirements, design, implementation, verification and maintenance. However, the V model adds forms of verification and validation against the previous activities. These include unit testing, integration testing, system verification, and operation/maintenance. My choice for the Waterfall model was due to the low complexity and uncertainty regarding the development of my game as I knew there would be no issues in development and thus no time would be wasted in backtracking to previous steps. Furthermore, because the Waterfall model is plan-driven and that we were given 3 weeks worth of time to complete this project I knew I could segregate the project into distinct phases of specification and development.

Description of Code

The game I chose to improve upon was a simple terminal outputted tic-tac-toe game that I originally coded in java. Using JavaScript and the react native libraries my improvements to the game are adding a GUI, the option to go back to previous states of the game, and removing terminal input to make the game interactive. The java code consisted of three methods called *printGameBoard(), populateBoard(), and finally checkWinner()*. The JavaScript code also has similar functions, however, they are separated across different files called *Square.js,* which as the name implies creates buttons or "squares" for which the user can click on during the game to mark their spots. *Board.js* uses the render function to create three rows of class name "border-row". Once created we fill each row with three references to the object square giving us a 3x3 square of buttons that simulates the tic-tac-toe board. Finally, we have *Game.js* which holds the main functionality of the game. It holds the functions *jumpTo()* which allows players to return back to a previous state in the game and continue playing from there*, handleClick(), render(), and checkWinner()* each do what their name implies.

Testing

For the testing of my developed game I decided to use the Jest and Enzyme framework which makes it easier for me to test my react components output. Most of the testing will be GUI based as in I will be simulating clicks on the gameboard and observing changes in the state (Eg. simulating 3 horizontal clicks for X results in a win). In addition, I plan to check whether going back to a certain state of the board in a game will return back the correct state of that board. Other tests will be simple renders to check whether the UI is rendering my JSX properly. All test results and explanations can be found in Appendix B

Conclusion

During the development of the tic-tac-toe game there were very few difficulties associated with bugs. However, because it was my first time working with a react I needed some help from a reactjs tutorial as listed in the references section of the report. The main difficulties that came with this assignment was understanding how enzyme and jest worked and using those to develop test cases for my program. The hardest test to develop was in Figure 7 of Appendix B where I was testing state reversion of the gameboard. The issue was associated with my lack of knowledge in reactjs and not understanding DOM enough. Another difficulty was understanding functions in enzyme and script dependency as shown in Figure 2 of Appendix B.

References

https://reactjs.org/tutorial/tutorial.html

Appendix

Appendix A - Models

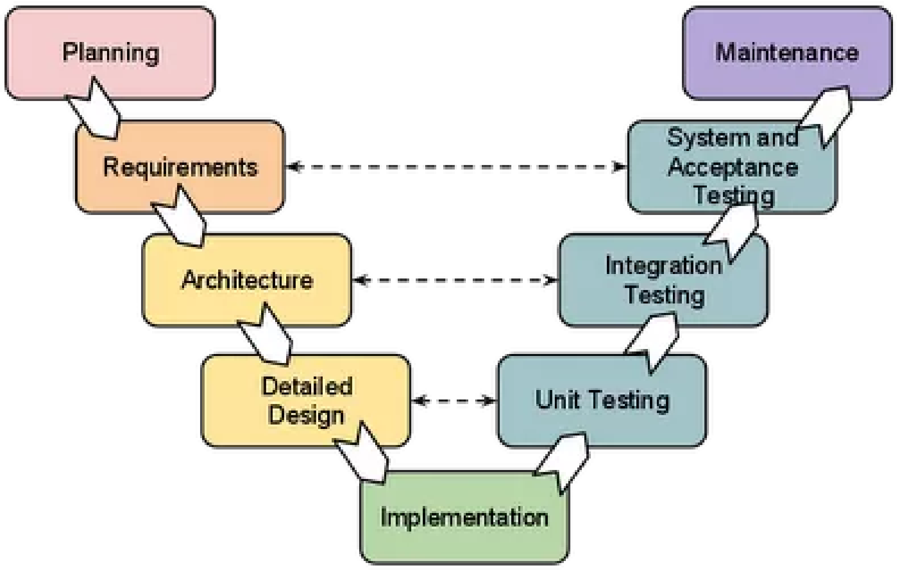
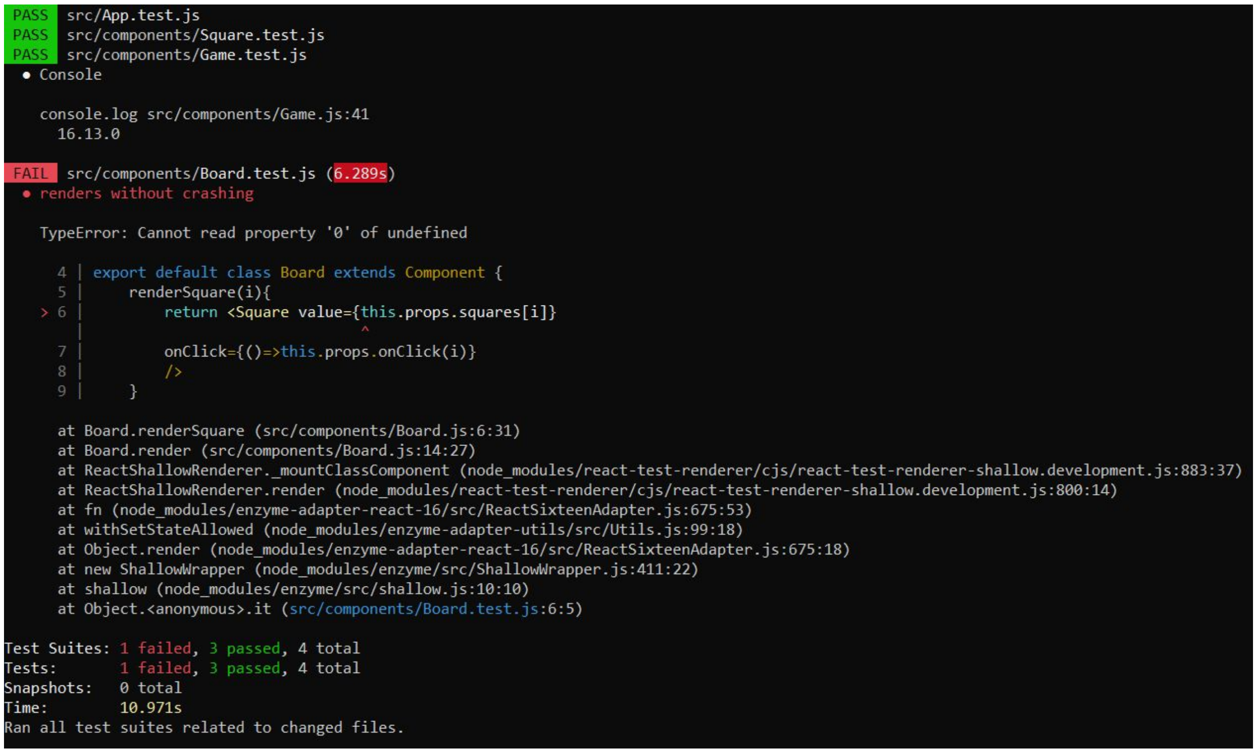


Figure 1: Waterfall V Model

Appendix B - Testing

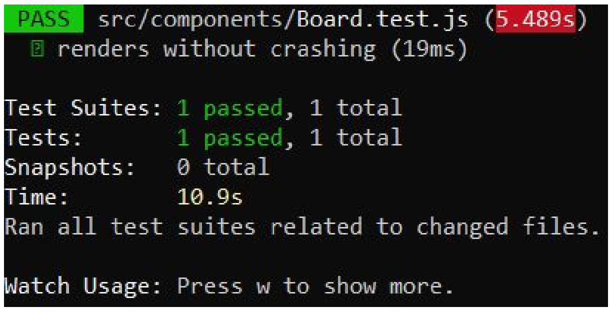
This appendix shows images and explanations of runned tests using Jest and Enzyme on the tic-tac-toe code.



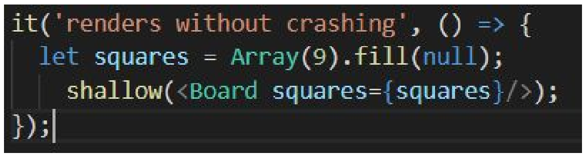
```
PASS  src/App.test.js
PASS  src/components/Square.test.js
PASS  src/components/Game.test.js
  ● Console

    console.log src/components/Game.js:41
      16.13.0

FAIL  src/components/Board.test.js (6.289s)
  ● renders without crashing

    TypeError: Cannot read property '0' of undefined

        | export default class Board extends Component {
        |     renderSquare(i){
      > |         return <Square value={this.props.squares[i]}
        |                                                   ^
        |         onClick={()=>this.props.onClick(i)}
        |         />
        |     }

      at Board.renderSquare (src/components/Board.js:6:31)
      at Board.render (src/components/Board.js:14:27)
      at ReactShallowRenderer._mountClassComponent (node_modules/react-test-renderer/cjs/react-test-renderer-shallow.development.js:883:37)
      at ReactShallowRenderer.render (node_modules/react-test-renderer/cjs/react-test-renderer-shallow.development.js:800:14)
      at fn (node_modules/enzyme-adapter-react-16/src/ReactSixteenAdapter.js:675:53)
      at withSetStateAllowed (node_modules/enzyme-adapter-utils/src/Utils.js:99:18)
      at Object.render (node_modules/enzyme-adapter-react-16/src/ReactSixteenAdapter.js:675:18)
      at new ShallowWrapper (node_modules/enzyme/src/ShallowWrapper.js:411:22)
      at shallow (node_modules/enzyme/src/shallow.js:10:10)
      at Object.<anonymous>.it (src/components/Board.test.js:6:5)

Test Suites: 1 failed, 3 passed, 4 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   0 total
Time:        10.971s
Ran all test suites related to changed files.
```

Figure 2: Basic Render Test

In the code about we see that out of the 4 tests 3 passed and 1 failed. The reason the board script failed is because of the use of ***shallow()*** which isolates a component from all other files, but, board needs the squares prop to be passed in to render correctly.



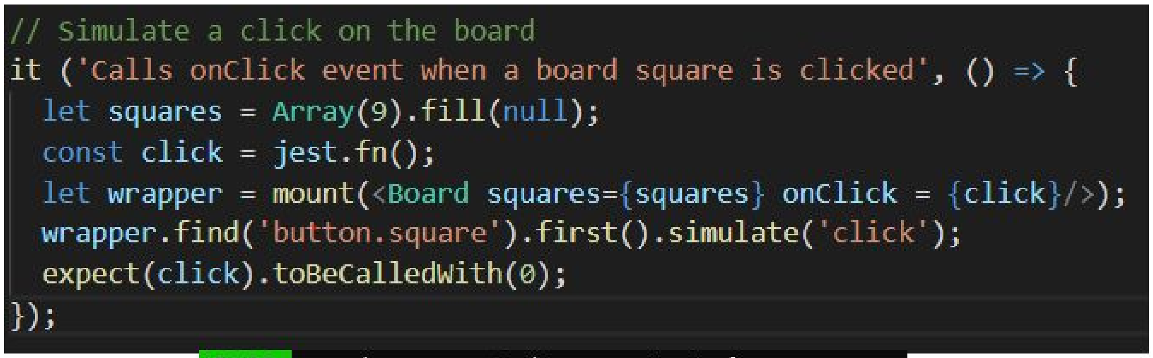
```
PASS  src/components/Board.test.js (5.489s)
  ✓ renders without crashing (19ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        10.9s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```



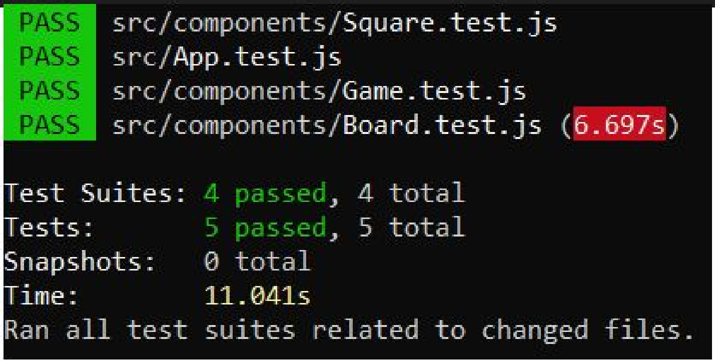
```
it('renders without crashing', () => {
  let squares = Array(9).fill(null);
    shallow(<Board squares={squares}/>);
});
```

Figure 3: Updated board Test

By defining a squares variable and passing it to the shallow function we see that the test that failed now passes.



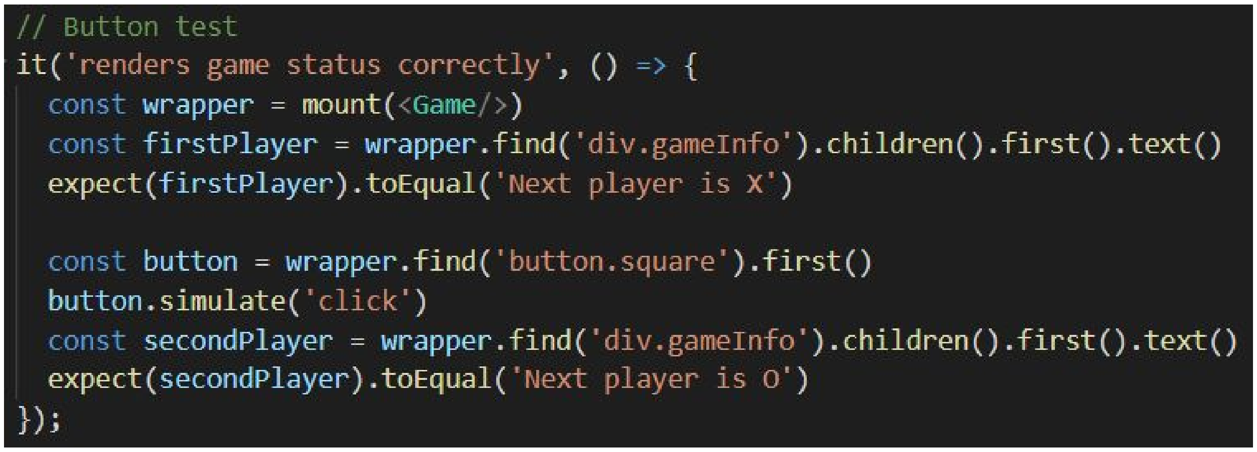
```
// Simulate a click on the board
it ('Calls onClick event when a board square is clicked', () => {
  let squares = Array(9).fill(null);
  const click = jest.fn();
  let wrapper = mount(<Board squares={squares} onClick = {click}/>);
  wrapper.find('button.square').first().simulate('click');
  expect(click).toBeCalledWith(0);
});
```



```
PASS  src/components/Square.test.js
PASS  src/App.test.js
PASS  src/components/Game.test.js
PASS  src/components/Board.test.js (6.697s)

Test Suites: 4 passed, 4 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        11.041s
Ran all test suites related to changed files.
```

Figure 4: Simple click test in Board.js

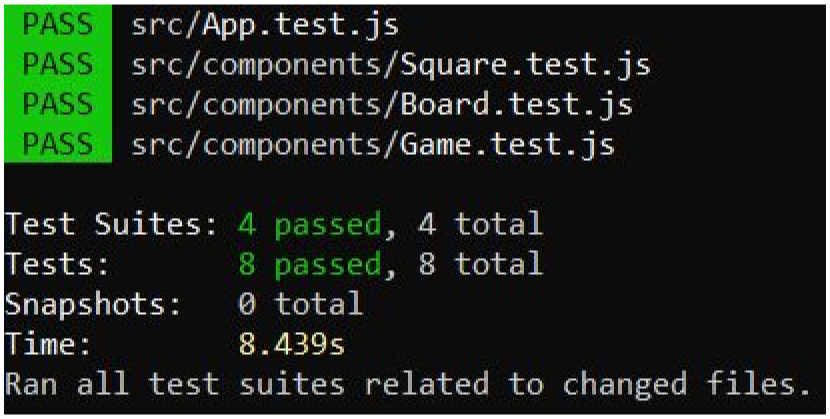
The above integration test uses both the board script and the square script and checks to see if a simulated click on a button works. This shows us that the buttons rendered on the board are functioning properly and that we can use the buttons for future use such as in the Game script. The reason there are 5 tests are because we have added an additional it function in the board script.



```
// Button test
it('renders game status correctly', () => {
  const wrapper = mount(<Game/>)
  const firstPlayer = wrapper.find('div.gameInfo').children().first().text()
  expect(firstPlayer).toEqual('Next player is X')

  const button = wrapper.find('button.square').first()
  button.simulate('click')
  const secondPlayer = wrapper.find('div.gameInfo').children().first().text()
  expect(secondPlayer).toEqual('Next player is O')
});
```

Figure 5: Button simulation

The above code is used to test state changes. When the first player goes (player x) and presses a square to mark their choice the website should change the text from “Next player is X” to “Next player is O”.

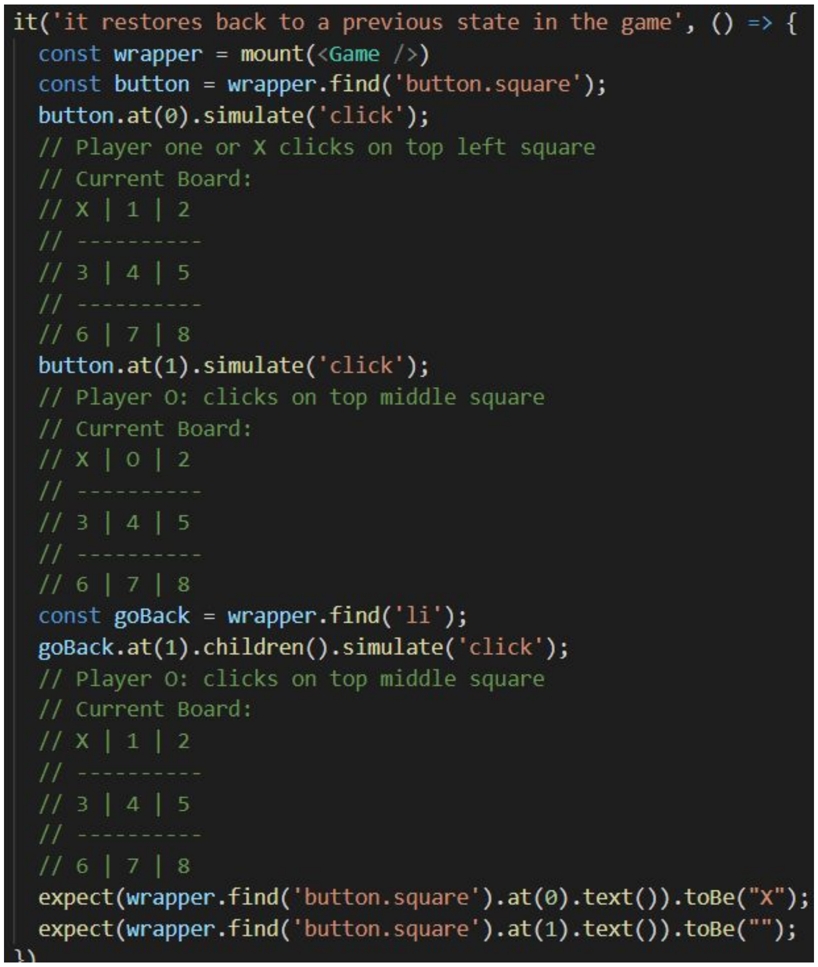


```
 PASS  src/App.test.js
 PASS  src/components/Square.test.js
 PASS  src/components/Board.test.js
 PASS  src/components/Game.test.js

Test Suites: 4 passed, 4 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        8.439s
Ran all test suites related to changed files.
```

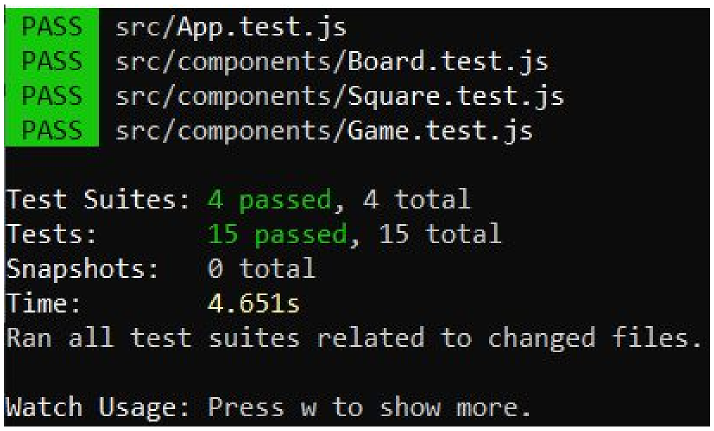
Figure 6: Win/Tie test in Game.js

The above passing tests simulated a game of tic-tac-toe and its state changes by using the code in Figure 5 I repeated the button presses to simulate a game being played. In these tests I made sure to check that the winning player displayed at the end of the game is actually the player who won. In addition I made sure to check that the game displays “Tie” when all squares are filled and nobody has won. These two tests ensure that a win condition is met and a tie condition is guaranteed to happen when nobody wins. Additional checks were made to make sure that horizontal, vertical and diagonal win conditions are triggered and that players cannot overwrite other plays choices.

gone.



```
it('it restores back to a previous state in the game', () => {
  const wrapper = mount(<Game />)
  const button = wrapper.find('button.square');
  button.at(0).simulate('click');
  // Player one or X clicks on top left square
  // Current Board:
  // X | 1 | 2
  // ----------
  // 3 | 4 | 5
  // ----------
  // 6 | 7 | 8
  button.at(1).simulate('click');
  // Player O: clicks on top middle square
  // Current Board:
  // X | O | 2
  // ----------
  // 3 | 4 | 5
  // ----------
  // 6 | 7 | 8
  const goBack = wrapper.find('li');
  goBack.at(1).children().simulate('click');
  // Player O: clicks on top middle square
  // Current Board:
  // X | 1 | 2
  // ----------
  // 3 | 4 | 5
  // ----------
  // 6 | 7 | 8
  expect(wrapper.find('button.square').at(0).text()).toBe("X");
  expect(wrapper.find('button.square').at(1).text()).toBe("");
})
```



```
 PASS  src/App.test.js
 PASS  src/components/Board.test.js
 PASS  src/components/Square.test.js
 PASS  src/components/Game.test.js

Test Suites: 4 passed, 4 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        4.651s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Figure 7: Testing previous state

Finally the last test of *Game.js* is to test whether using the “Go back to Move #1” works. As shown above, we simulate two clicks one for each player and a click to the “Go to Move #1” button. After clicking this button the move player O made on slot 1 should be gone.