Московский государственный технический университет
имени Н. Э. Баумана

Факультет «Информатика и системы управления»
Кафедра «Программное обеспечение ЭВМ
и информационные технологии»

Тестирование и отладка ПО

Лабораторные работы

Студент: Петухов И.С.
Группа: ИУ7-71
Преподаватель: Рогозин О.В.

Москва, 2016

Содержание

1 Аналитический раздел

Цель данной работы - протестировать web-приложение платформа для проведения футбольных турниров.

1.1 Описание тестируемой системы

Web-приложение платформа для проведения футбольных турниров состоит из следующих частей:

а) сервер
б) база данных
в) клиентская часть

Сервер написан на языке JavaScript с использованием программной платформы NodeJS. Основное предназначение - обработка http запросов.

База данных - нереляционная СУБД mongoDB. Обращение к базе данных осуществляется с помощью библиотеки Mongoose. Mongoose - обертка, позволяющая создавать удобные и функциональные схемы документов.

Клиентская часть написана на языке JavaScript с использованием библиотеки ReactJS.

Модели, описывающие объекты веб-приложения/базы данных:

а) federation
б) match
в) stage
г) team
д) tournament
е) user
ж) vuser

app.js - главный файл тестируемого веб приложения.

В файле **app.js** подключаются обработчки http запросов, описанных в файлах папки **routes**:

а) federation.js - обработка запросов, связанных с объектом федерация
б) match.js - обработка запросов, связанных с объектом матч
в) stage.js - обработка запросов, связанных с объектом этап
г) team.js - обработка запросов, связанных с объектом команда
д) tournament.js - обработка запросов, связанных с объектом турнир
е) vuser.js - обработка запросов, связанных с объектом виртуальный пользователь

ж) referee.js - обработка запросов, связанных с взаимодействием с мобильным приложением

з) main.js - обработка остальных запросов

В мобильном приложении авторизуется пользователь и получает список матчей, на которые он назначен судить. Судейство матча означает оправку на сервер следующих событий:

а) MATCH_STARTED - начало матча
б) MATCH_FINISHED - конец матча
в) TIME_STARTED - начало периода
г) TIME_FINISHED - конец периода
д) GOAL - гол
е) OWN_GOAL - автогол
ж) YELLOW_CARD - желтая карточка
з) RED_CARD - красная карточка
и) MIN - просто прислана текущая минута матча
к) ASSIST - прислано событие с игроком - который отдал голевой пас.

Каждое сообщение, посылаемое на сервер, должно содержать следующие поля:

а) idAction - порядковый номер события в матче
б) idMatch - id матча
в) minute - минута матча, на которой произошло событие
г) idEvent - тип события, описанные выше

1.2 Рассматриваемые виды тестирования

В данной работе будут рассмотрены следующие виды тестирования:

а) интеграционное
б) регрессионное
в) функциональное

Интеграционное тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).

Регрессионное тестирование - это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде.

Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

2 Технологический раздел

2.1 Интеграционное тестирование

В данной части работы будет осуществлено тестирование модулей сервера, обрабатывающие http запросы, взаимодействующие с базой данных, в которых импортированы модели.

2.1.1 Тестирование модуля vuser.js

vuser.js - набор методов для обработки запросов, связанных с объектом виртуальный пользователь.

В данном модуле описаны методы-обработчики:

а) **POST /get** - получение информации о виртуальном пользователе по заданном id

Код модуля **vuser.js** небольшой и представлен в листинге 2.1

Листинг 2.1 — Код метода обработчика урла /get

```
1  var express = require('express');
2  var router = express.Router();
3  var Vuser = require('../models/vuser');
4
5  router.post('/get', function(req, res, next) {
6      if (!req.body.id) {
7          return res.status(400).json(null);
8      }
9
10     Vuser.findById(req.body.id, (err, vuser) => {
11         if(err || !vuser) {
12             return res.status(404).json(null);
13         }
14
15         return res.status(200).json(vuser);
16     });
17 });
18
19 module.exports = router;
```

Тесты для модуля **vuser.js** описаны в таблице 2.1.

2.1.2 Тестирование модуля referee.js

referee.js - набор методов для обработки запросов, связанных с взаимодействием с мобильным приложением.

В данном модуле описаны методы-обработчики:

а) **POST /get-my-matches** возвращает список матчей, на которые назначен судить заданный пользователь

б) **POST /set-info** получает событие, произошедшее в заданном матче с описанием события.

Тесты для модуля **referee.js** описаны в таблице 2.2.

Таблица 2.1 — Тестирование модуля vuser.js

1. запрос юзера по существующему id	
Запрос	{id: 12345}
Ожидаемый ответ	200 OK {$\cdots$}
2. запрос юзера по несуществующему id	
Запрос	{id: 67890}
Ожидаемый ответ	404 NOT FOUND: {null}
3. запрос юзера с отсутствующим полем id в запросе	
Запрос	{}
Ожидаемый ответ	400 BAD REQUEST: {null}

Таблица 2.2 — Тестирование модуля referee.js

1. запрос списка матчей по существующему id пользователя	
Запрос	{id: 12345}
Ожидаемый ответ	200 OK {$\cdots$}
2. запрос списка матчей по несуществующему id	
Запрос	{id: 67890}
Ожидаемый ответ	404 NOT FOUND: {null}
3. запрос списка матчей с отсутствующим полем id в запросе	
Запрос	{}
Ожидаемый ответ	400 BAD REQUEST: {null}
4. отправка события MATCH_STARTED в существующем матче	
Запрос	{idMatch: match.id, idEvent: Match.EVENT.MATCH_STARTED.name, idAction: 0, minute: 0}
Ожидаемый ответ	200 OK: {null}
Ожидаемые действие	Заданный матч изменяет статус на RUNNING {null}
5. отправка события MATCH_FINISHED в существующем матче	
Запрос	{idMatch: match.id, idEvent: Match.EVENT.MATCH_FINISHED.name, idAction: 0, minute: 90}
Ожидаемый ответ	200 OK: {null}
Ожидаемые действие	Заданный матч изменяет статус на FINISHED {null}
6. отправка любого события по несуществующему id	
Запрос	{idMatch: — idEvent: — idAction: 0, minute: 0}
Ожидаемый ответ	404 NOT FOUND: {null}
7. отправка любого события с отсутствующим полем id в запросе	
Запрос	{}
Ожидаемый ответ	400 BAD REQUEST: {null}

2.2 Регрессионное тестирование

Для того чтобы знать, какие тесты перезапускать после того или иного изменения в программе, нужно определить, от каких конкретно частей программы (модулей, методов, и т.п.) зависит результат каждого теста. Для этого часто используется управляющий граф, отображающий поток управления программы, по которому легко отследить зависимости одних блоков/модулей/методов от других.

Учитывая лекцию от 25 октября 2016 и лабораторные, которые сдавали мои одногруппники, я могу выделить следующие используемые виды регрессионного тестирования:

а) на уровне методов
б) на уровне кода

2.2.1 Уровень методов

Строится матрица зависимостей методов от тестов. Однако Web-приложения строятся по принципу микросервисной архитектуры. Поэтому составные модули веб-приложений должны быть независимыми. Более того, методы-обработчики http запросов в этих модулях - также часто не зависят друг от друга. На каждый метод-обработчик пишется несколько тестов, которые проверяют качество данного метода. А учитывая специфику веб-приложений, эти тесты никак не будут связанны с другими методами. Из этого можно сделать вывод, что выполнять регрессионное тестирование на уровне методов и строить матрицу тестов-методов не имеет смысла (т.к. каждый тест проверяет определенный метод и ни как не связан с другими методами).

2.2.2 Уровень кода

При данном подходе нужно взять метод, взять тесты, которые написаны для этого метода, построить граф потока управления данного метода, построить зависимости узлов графа(строчек кода) от тестов и из этого сделать вывод, какие тесты необходимо запускать, при изменении определенного блока (строчкек кода), а какие нет.

Рассмотрим данный подход на примере метода /**set-info** модуля **referee.js**

Сервер по урлу /**set-info** получает событие, произошедшее в заданном матче с описанием события и сохраняет его в базе.

Для данного метода из раздела 2.1.2 описаны тесты:

а) отправка события **MATCH_STARTED** в существующем матче
б) отправка события **MATCH_FINISHED** в существующем матче
в) отправка любого события по несуществующему id

г) отправка любого события с отсутствующим полем id в запросе

Листинг 2.2 — Код метода обработчика урла /set-info

```
router.post('/set-info', function(req, res, next) {
    let idMatch = req.body.idMatch;
    let idEvent = req.body.idEvent;
    let idAction = req.body.idAction;
    let minute = req.body.minute;

    if (!idMatch || !idEvent || isNaN(idAction) || isNaN(minute)) {
        return res.status(400).json(null);
    }

    let now = new Date();

    let event = {
        idEvent: idEvent,
        idAction: idAction,
        minute: minute,
        realTime: now,
    };

    Match.findById(idMatch, function (err, match) {
        if (err || !match) {
            return res.status(404).json(null);
        }

        switch (idEvent) {
            case Match.EVENT.MATCH_STARTED.name:
                match.status = Match.STATUS.RUNNING.name;
                break;
            case Match.EVENT.MATCH_FINISHED.name:
                match.status = Match.STATUS.FINISHED.name;
                break;
            case Match.EVENT.GOAL.name:
            case Match.EVENT.OWN_GOAL.name:
            case Match.EVENT.YELLOW_CARD.name:
            case Match.EVENT.RED_CARD.name:
                event.idTeam = req.body.idTeam;
                event.idPlayer = req.body.idPlayer;
                event.teamName = req.body.teamName;
                event.playerName = req.body.playerName;
                break;

        }

```

```
44          match.events.push(event);
45
46          clients.forEach((ws) => {
47              ws.send(JSON.stringify(event));
48          });
49
50          match.save(function (err) {
51              if (err) {
52                  return res.status(500).json(null);
53              }
54
55              return res.status(200).json(null);
56          });
57      });
58  });
```

По листингу 2.2 построим граф потока управления. Каждый узел графа (блок) отмечен заглавной латинской буквой. Числа в скобках указывают на строчки кода в листинге 2.2.

Для проверки правильности составленной таблицы 2.3 изменим строчку №55, которая соответствует блоку K.

По рисункам 2.2 и 2.3 видно, что после изменения блока K стали падать тесты а, б, что соответствует построенной таблице 2.3

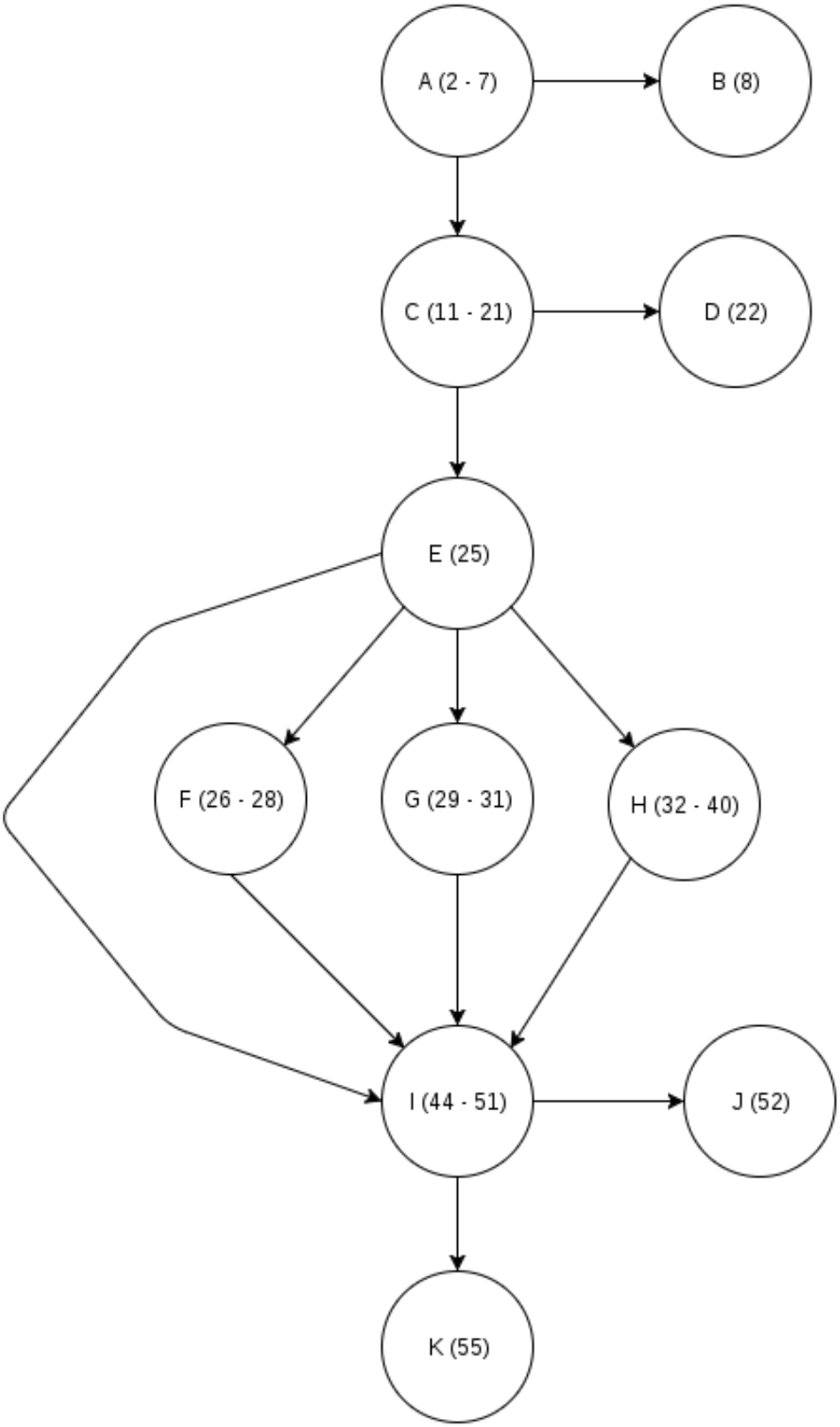

Рисунок 2.1 — Граф потока управления метода обработчика /set-info

Таблица 2.3 — Матрица зависимостей блоков графа от тестов

	тесты			
блок	а	б	в	г
A	-	-	-	-
B	-	-	-	+
C	-	-	-	-
D	-	-	+	-
E	-	-	-	-
F	+	-	-	-
G	-	+	-	-
H	-	-	-	-
I	-	-	-	-
J	-	-	-	-
K	+	+	-	-

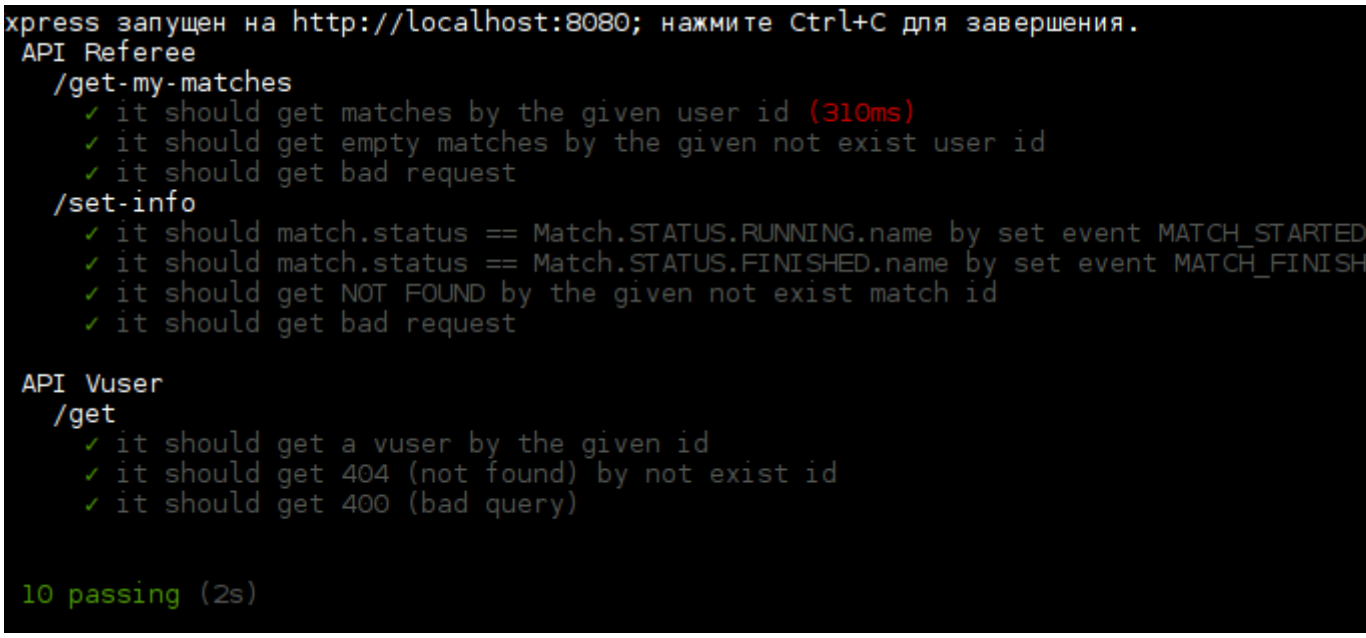

Рисунок 2.2 — Прохождение тестов до изменения блока K

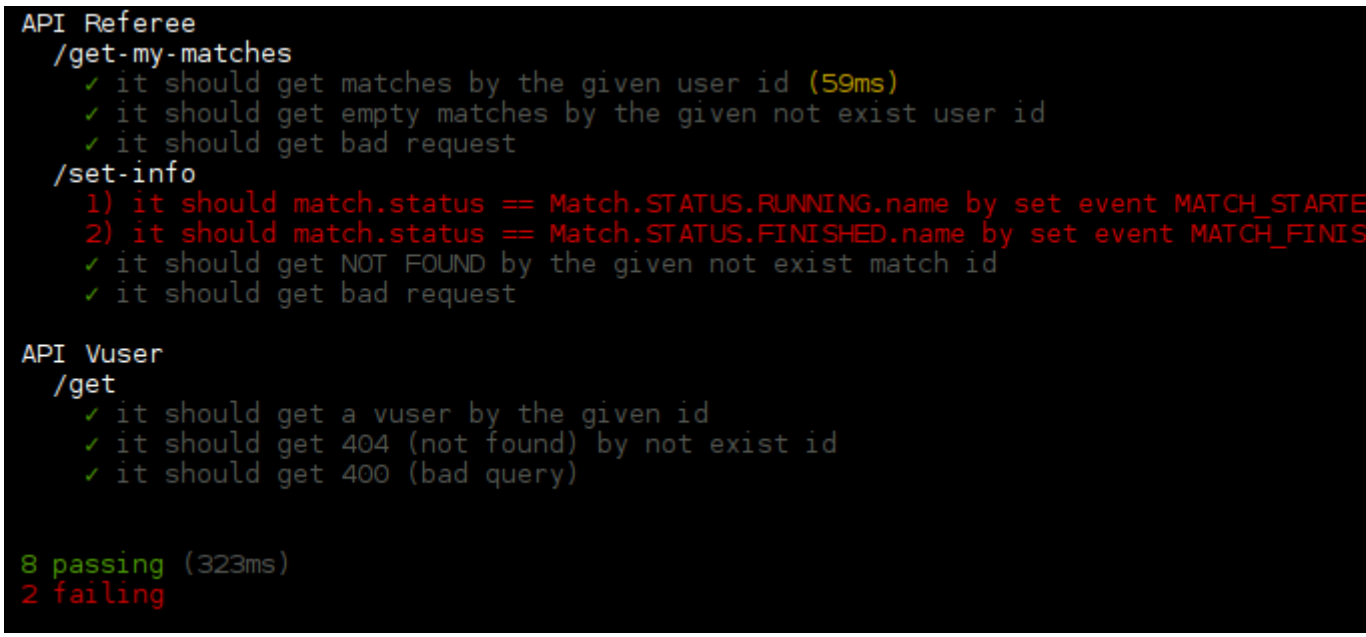

Рисунок 2.3 — Прохождение тестов после изменения блока K

2.3 Функциональное тестирование

Функциональные тесты основываются на функциях, выполняемых системой. Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).

2.3.1 Use Cases

Use Case (вариант использования, ВИ, Прецедент, юскейс) — это сценарная техника описания взаимодействия. С помощью Use Case может быть описано и пользовательское требование, и требование к взаимодействию систем, и описание взаимодействия людей и компаний в реальной жизни. В общем случае, с помощью Use Case может описываться взаимодействие двух или большего количества участников, имеющее конкретную цель. В разработке ПО эту технику часто применяют для проектирования и описания взаимодействия пользователя и системы, поэтому название Use Case часто воспринимает как синоним требования человека-пользователя к решению определенной задачи в системе.

Примеры Use Case для тестируемого приложения:

а) создание пользователем команды
б) создание пользователем федерации
в) создание пользователем турнира по футболу
г) создание пользователем матчей в турнире его федерации
д) подача пользователем заявки на участе его команды в турнире

2.3.2 Use Case создание пользователем команды

Рассмотрим подробнее use case создание пользователем футбольной команды.

Система - сайт web-приложения

Основное действующее лицо - пользователь-капитан команды

Цель - создать команду

Триггер - пользователь решает создать команду и заходит на главную страницу сайта

Результат - информация о команде сохранена в базе данных, пользователь получил ответ от сервера.

2.3.2.1 Основной поток событий

а) пользователь заходит на главную страницу сайта
б) пользователь авторизуется на сайте через сайт-вконтакте
в) пользователь нажимает на кнопку «создать команду»

г) пользователь заполняет поля «название» и «город»

д) пользователь нажимает отправить

е) система получает запрос

ж) система проверяет запрос

з) при верном запросе система сохраняет данные в базе

и) система посылает ответ пользователю

2.3.3 Автоматизация функционального тестирования

В связи с большим колличеством usecase и большим колличеством монотонных действий в каждом usecase достаточно распространенной является автоматизация функционального тестирования.

Автоматизация функционального тестирования выполнена с помощью Selenium WebDriver.

Selenium WebDriver - автоматизирует действия с браузером. Это именно то, что необходимо для тестирования сайта web-приложения. Частым является использование паттерна Page Object при написании тестов с использованием Selenium.

2.3.3.1 Тестирование создания команды

Код теста приведен в листинге 2.3 с учетом рассмотренного потока событий

Листинг 2.3 — Тестирование use case создание команды

```
1  it('create team', function(done) {
2          client.manage().timeouts().implicitlyWait(5000);
3
4          let teamName = 'NAME TEAM1';
5          let teamCity = 'CITY TEAM1';
6
7          vkPage.open()
8              .then(() => {
9                  vkPage.auth(secret.emailVk, secret.passVk);
10                 mainPage.open(URL);
11                 mainPage.auth();
12                 mainPage.createTeam(teamName, teamCity);
13                 client.quit();
14             })
15             .then(() => {
16                 return Team.find({name: teamName}, function (err, team) {
17                     assert.isNull(err);
18                     assert.isNotNull(team);
19                     done();
20                 })
21             })
```

Листинг 2.4 — Объект mainPage

```
function mainPage(driver) {
    let self = this;

    self.driver = driver;

    self.open = function(url) {
        return self.driver.get(url);
    };

    self.auth = function() {
        self.driver.findElement(wd.By.id('vk-auth-btn')).click();

        self.driver.findElements(wd.By.className("button_indent"))
            .then(arr => {
                if (arr.length > 0) {
                    arr[0].click();
                }
            });
    };

    self.createTeam = function(teamName, teamCity) {
        // var createTeamBtn =
    self.driver.wait(wd.until.elementLocated(wd.By.id('create-team-btn')),
    10000);
        // createTeamBtn.click();
        self.driver.findElement(wd.By.id('create-team-btn')).click();

        self.driver.findElement(wd.By.id('team-name')).sendKeys(teamName);
        self.driver.findElement(wd.By.id('team-city')).sendKeys(teamCity);

        self.driver.findElement(wd.By.id('send-team-btn')).click();
    }

}
```

Листинг 2.5 — Результаты тестирования

```
ilyaps@debian-ilyaps:~/projects/MyFootballFederation$ npm test

> mff@1.0.0 test /home/ilyaps/projects/MyFootballFederation
> mocha test/gui/ --no-timeoutsзапущенная


 база: mongodb://localhost:27017/football_test


```

```
10 Express запущен на http://localhost:8080; нажмите Ctrl+C для завершения.
11   GUI Create team
12     OK normal create team (21944ms)
13
14
15   1 passing (22s)
```