**Практическая работа 3**

**Задание:** Для приложения разработать Unit-тест, который тестирует работу класса BoxService. Для теста добавить Stub или Mock объекты в зависимости от того, что тестируем, состояние или взаимодействие.

Класс BoxService реализует с помощью метода analize() запись ошибок в локальное хранилище в формате строки. Ошибки могут возникать при несоответствии имен файлов определенным требованиям.

Метод тестирования expect()… должен проверить. что метод analize() класса BoxService правильно вызывает метод logError() Mock объекта.

**Ход работы:**

1. **Создать приложение MocksAndStubsUnitTest в редакторе VSCode.**

Установить с помощью **npm** в devDependencies пакет jest, прописать скрипт запуска тестового пакета JEST в файле **package.json**:



Рис. 1. Файл **package.json**

**2. Начальные установки**

**2.1 Реализация шаблона проекта.**

Тесты пишем в файле ServiceBox.test.js. Для тестов в Jest ВАЖНО, чтобы файл оканчивался на test.js или spec.js ИЛИ был помещен в папку “\_\_tests\_\_”

Остальную бизнес логику пишем в других файлах-модулях.

**2.2. Задаем имя приложению: MocksAndStubsUnitTest**

В приложении также будут классы-модули: базовый LogService и производный MockLogService. См. рис.2

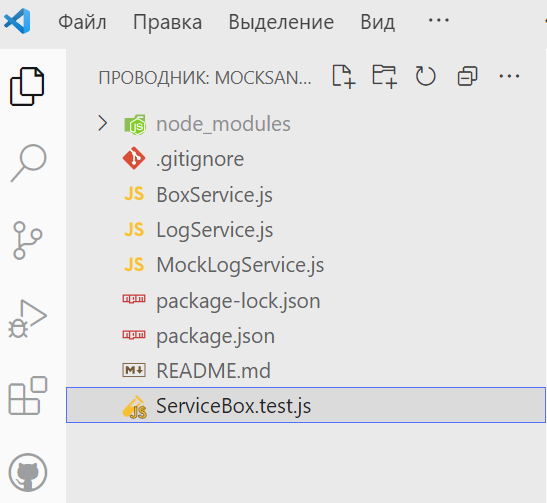


Рис. 2. Файловая структура проекта в Visual Studio Code

**3. Работа модулей в тесте.**

**3.1. Текст программы, которую нужно протестировать**

На данный момент текст программы, которую нужно протестировать, имеет вид:



Рис. 3. Код модуля BoxService.js

В этом примере мы наполним класс BoxService следующим функционалом:

Добавим метод analize(), который:

* принимает имя какого-то файла, проверяет, есть ли у этого файла расширение «txt» path.extname(fileName) != ".txt" ;
* используя некий сервис записывает в некий журнал ошибок сообщения ошибки "FileExtension error: " + fileName (ошибка разрешения файла);
* аналогично проверяет длину файла, если меньше 8 символов, пишет в логи "Filename too short:" + fileName

При этом метод analize использует внешнюю зависимость – экземпляр **service,** который реализует класс **LogService** через его производный класс **MockLogService.** В данном случае мы заменяем сервис Логгирования неким подставным объектом. Это может быть **MOCK** объект или другой.

Внешняя зависимость от **LogService в виде экземпляра service** внедряется в качестве параметра конструктора **BoxService**. В таких случаях говорят, что внедрение происходит через конструктор.

**3.2. Работа класса теста ServiceBox.test.js при тестировании модуля BoxService**

Если разобраться, то мы понимаем, что для тестирования метода analize нужно использовать тестирование взаимодействия, которое проверит, что метод analize правильно вызывает метод **LogError** объекта **service**, поэтому для замены объекта **service** мы будем использовать **MOCK** объект.

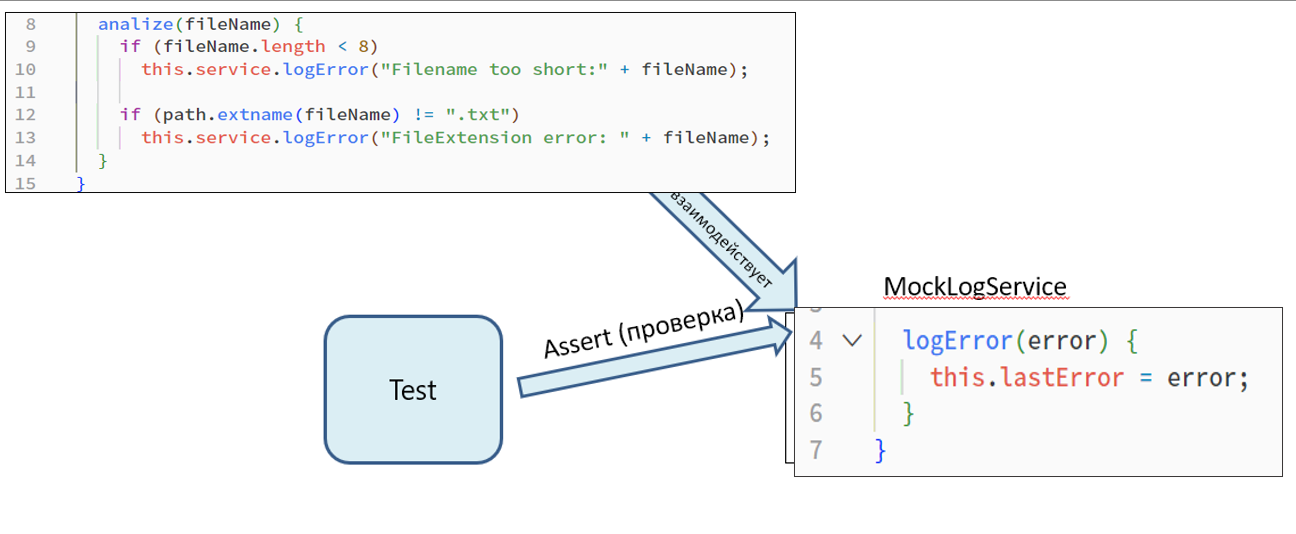


Рис. 4. Схема работы теста ServiceBox.test

В качестве Mock объекта выступает класс MockLogService, который содержит метод logError().

Суть метода logError состоит в сохранении параметра, который передается в метод logError при его использовании.

Тесты, которые используют объекты Mock, должны быть направлены на проверку состояния именно Mock объекта.

В нашем случае Mock объект будет выглядеть так:



Рис. 5. Код модуля MockLogService.js

Класс MockLogService является производным от класса LogService (см рис.5) За счет такой зависимости мы можем без риска подменить настоящий логгер на Mock объект в конструкторе класса BoxService.

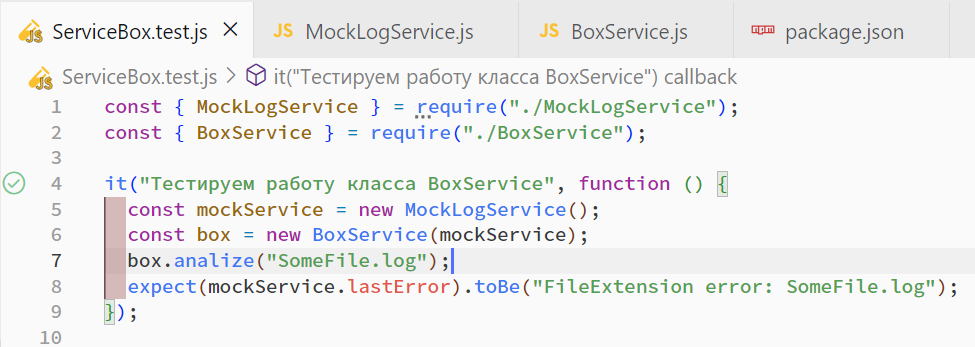


Рис. 6. Код модуля ServiceBox.test.js

Проверка взаимодействия класса BoxService с Mock объектом MockLogService происходит в тестовом модуле ServiceBox.test.js.

Здесь создается mockService-экземпляр объекта MockLogService, а затем box-экземпляр объекта BoxService, который принимает зависимость от Mock объекта mockService.

Далее вызываем метод analize объекта BoxService и передаем ему имя файла.

Имя файла "SomeFile.log" не соответствует нашим условиям в методе analize, генерируются сообщения об ошибке, которые мы проверяем методом expect().toBe() из фреймворка Jest.

На данный момент наша программа готова к тестированию.

**4. Запуск теста на выполнение и проверка результата тестирования**

В  Visual Studio Code для работы с Unit-тестами в разных фреймворках реализовано специальное меню команд, которое называется TestExploer Также в помощь тестировщикам созданы расширения. Будем использовать одно из них под названием Jest.

Данное расширение запускает службу, которая постоянно мониторит код и непрерывно выполняет тестирование,как изображено на рисунке 7.

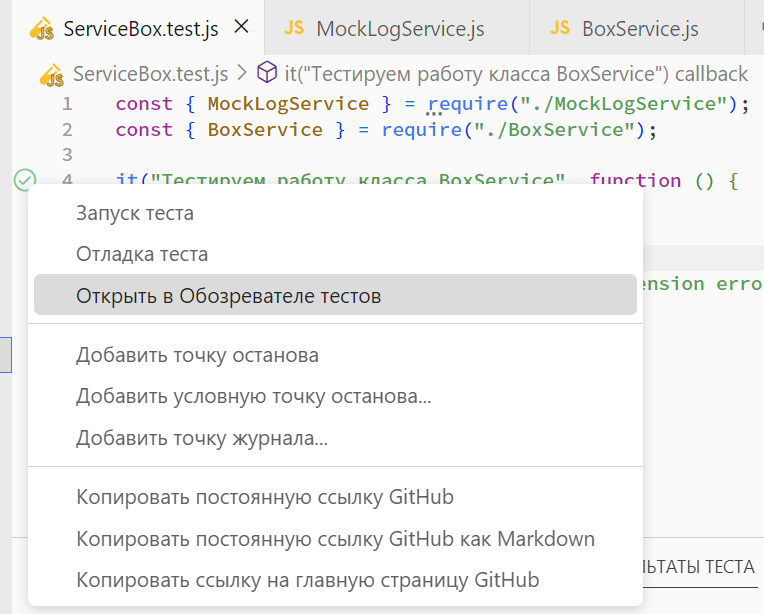


Рис. 7. Вызов команды запуска тестирования

После запуска теста, результат можно просмотреть в левой части в окне Test Explorer.

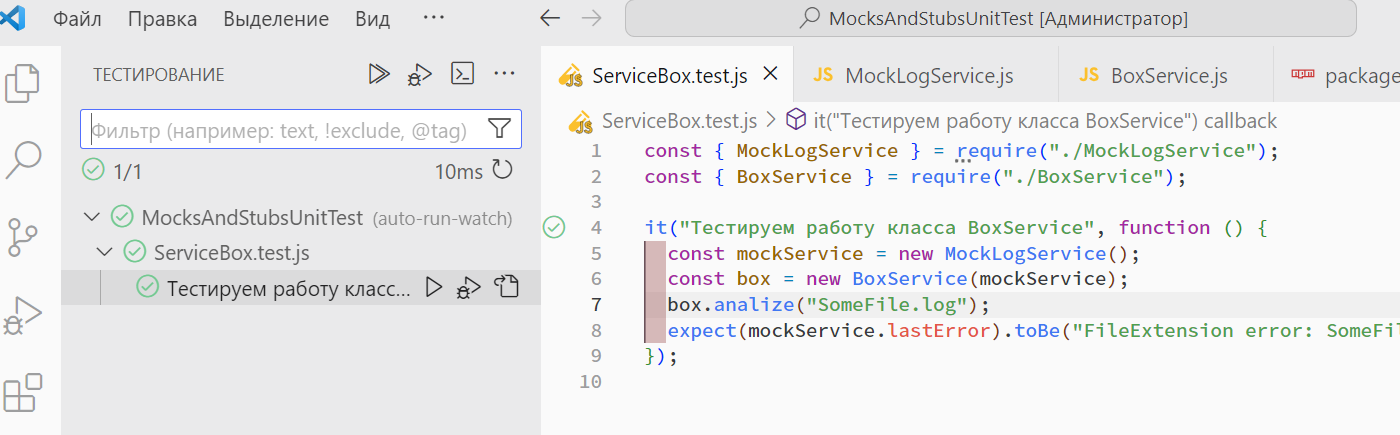


Рис. 9. Просмотр результата тестирования в Test Explorer

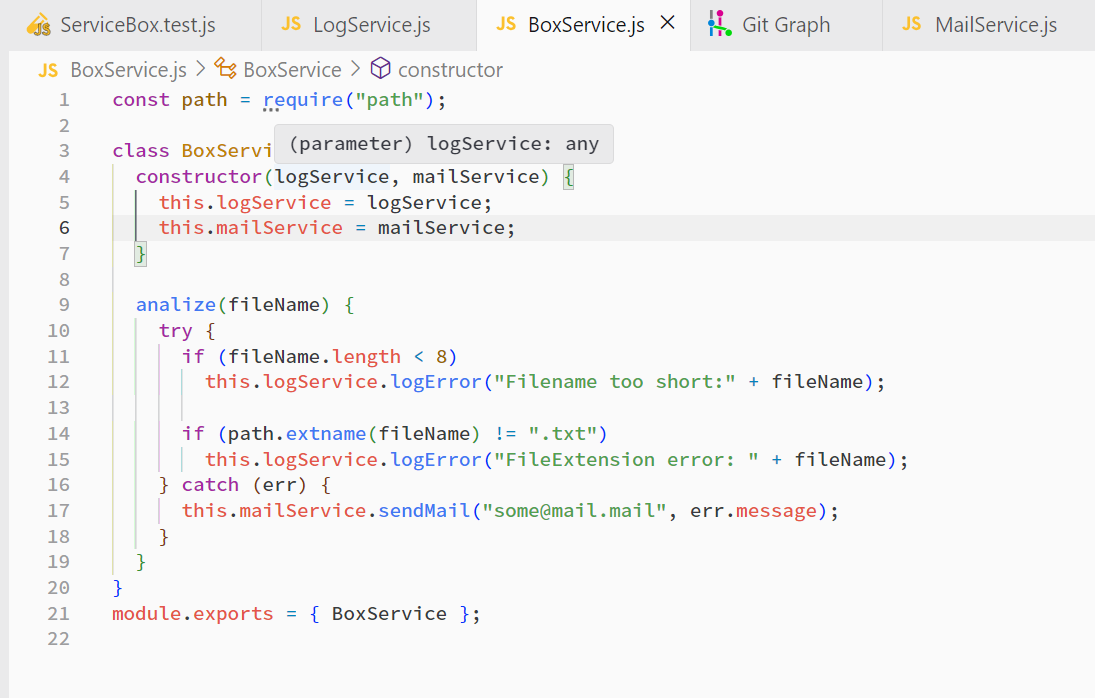
Соответственно в окне Test Explorer будет отображен положительный результат теста. После этого можно сделать вывод о том, что взаимодействие класса BoxService с Mock объектом MockLogService происходит в тестовом модуле ServiceBox.test.js правильно.

**Практическая работа № 3.1**

**Задание 1.** В этом задании будут параллельно использоваться Mock объект и Stub объект. Дополним метод analize() объекта BoxService новым функционалом:

если метод **analize()**  попробует послать объекту **logService** какое-то сообщение и при этом **logService** не будет доступен, то сгенерируется исключительная ситуация и мы пошлём сообщение в техподдержку с сообщением об этой исключительной ситуации с помощью объекта **mailService** (см. рис 10):

* Объект **logService** у нас будет Stub объектом.
* Объект **mailService** у нас будет Mock объектом.



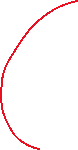


Рис. 10. Код модуля BoxService.js

Мы будем проверять взаимодействие объекта **BoxService** с объектом **mailService.**

Зависимость в класс **BoxService** мы будем внедрять с помощью конструктора класса **BoxService**.

Посмотрим на наши подставные объекты:

|  |  |
| --- | --- |
| **Stub** объект | **Mock** объект |
| **Класс LogService** | **Класс MailService** |
| оба создают экземпляры объектов, которые в тесте «моккаются» | |
| **stubLogService** | **mockMailService** |
| В результате оба экземпляра содержат простые методы имитирующие работу: | |
| генерации исключения (**LogError**)  с текстом той ошибки, которую мы передаём этому методу | отсылку сообщения (**SendMail**)  в техподдержку |

Теперь давайте перейдём к тестовому методу и посмотрим как он реализован:

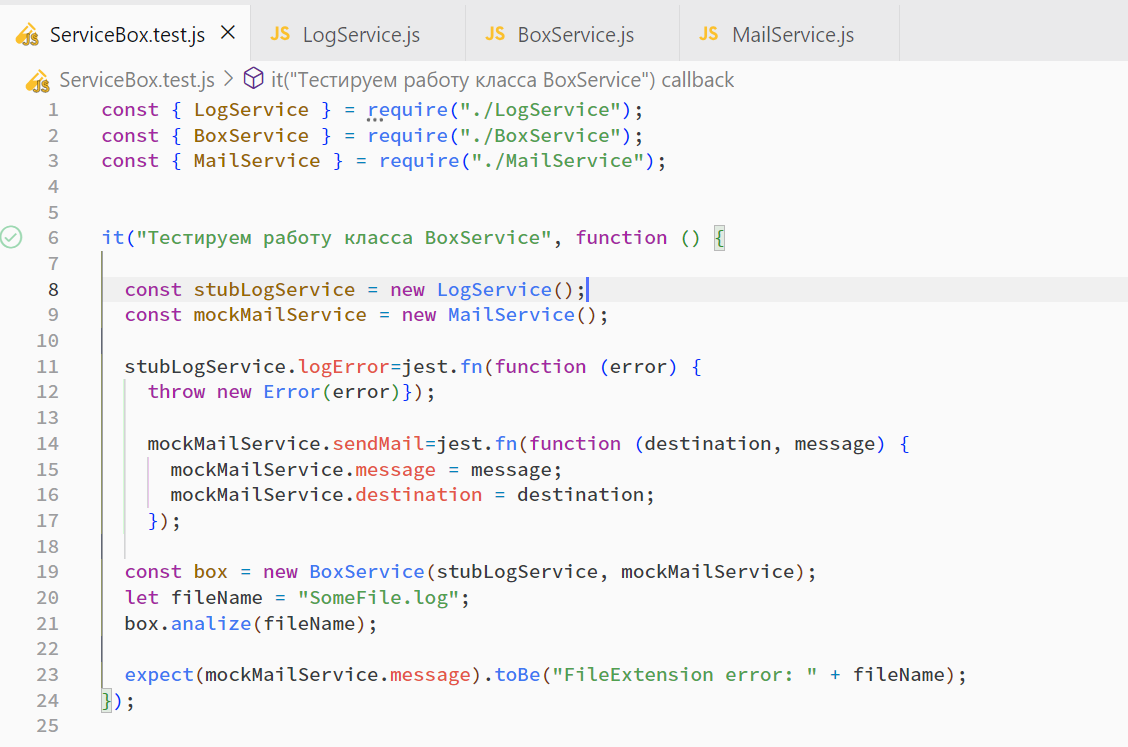




Рис. 13. Код модуля ServiceBox.test.js.

1. создали тестовый метод it(),
2. создали экземпляры объектов **LogService** и **MailService,**
3. Для созданных экземпляров объектов **LogService** и **MailService** создаемmock функции, которые имитируют реальные методы объектов **LogService** и **MailService ,** предоставляющие внешнюю функциональность для тестируемого модуля.
4. создали экземпляр объекта BoxService и внедрили в него зависимости от объектов stubLogService и mockMailService ,
5. подготовили имя файла, который хотим проанализировать,
6. вызвали метод analize(), который через метод logError будет генерировать Error с сообщением "FileExtension error: " и конкатенацией с именем файла (дополнительно см. рис.10), кроме того через конструкцию try catch мы отловим этот Error и вызовем метод sendMail с текстом исключения;
7. вызываем тестовый метод expect().toBe() и проверяем что BoxService успешно провзаимодействовал с MailService.

Результат тестирования: Тест успешно пройден:

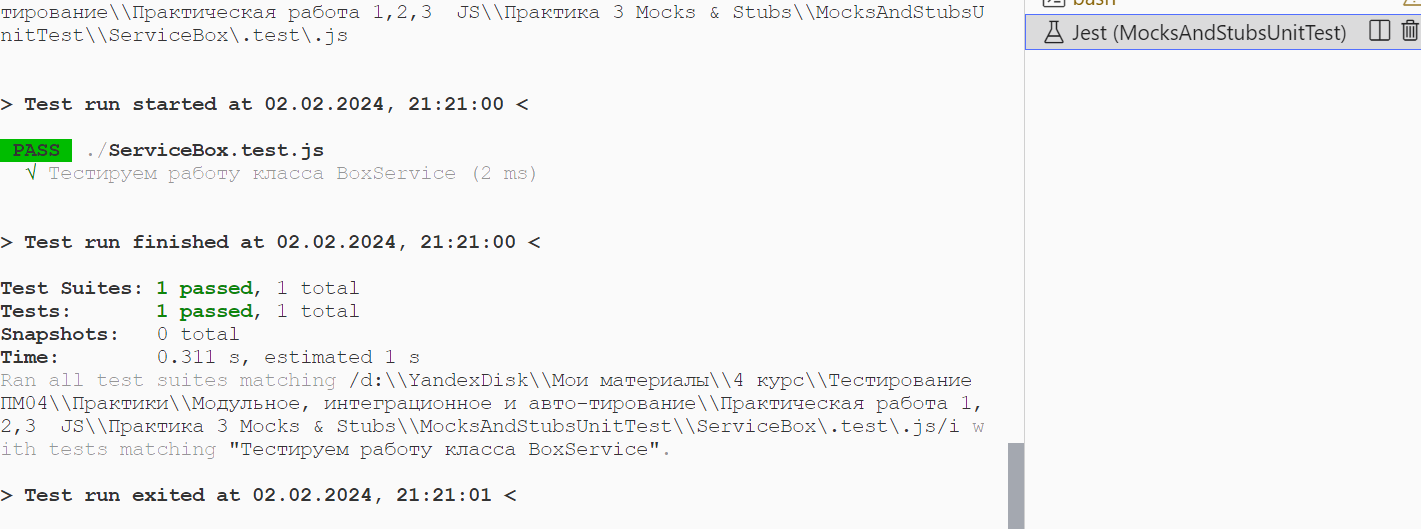


Рис. 14. просмотр результата тестирования

**Практическая работа № 3.2**

**Задание** .

1. В продолжение расширения функциональности предыдущего примера разработать программу, которая :

* реализует взаимодействие объекта BoxService с **LogService** и **MailService** (уже реализовано в предыдущем примере);
* реализует работу объекта BoxService при участии объекта **DataAccess**, реализующего получение массива имен файлов, извлекаемых из некой Базы Данных;
* реализует взаимодействие объекта BoxService с объектом **ReportManager**, реализующим сохранение на некоем файловом ресурсе отчета о полученной в предыдущем пункте данного Задания коллекции имен файлов.

Разработку производить с применением Stub и Mock объектов имитирующих работу неких реальных объектов. Самостоятельно определить в каком случае необходима заглушка, а в каком имитатор. Реализацию имитирующих методов производить с помощью функции jest.fn(), как это было сделано в предыдущем примере в Практической работе 3.1

1. Произвести тестирование взаимодействия объекта BoxService :

* С объектом **ReportManager** с помощью метода expect(Array.IsArray()).toBe(true) библиотеки Jest.
* С объектом **DataAccess** с помощью метода expect(массив возможных сообщений об ошибке).toContain(«строка содержащая сообщение об ошибке») библиотеки Jest.

Таким образом, для тестирования разных взаимодействий надо создать несколько методов expect в тестовом модуле. И, возможно, придется внести изменения в предыдущий код, поскольку новые объекты повлияют на работу метода объекта BoxService().

Успешное прохождение тестов будет указывать на правильно написанный код взаимодействия объектов.