3 Устранение зависимостей при помощи стабов

В ЭТОЙ ГЛАВЕ

- ✓ Типы зависимостей — моки, стабы и т. д.
- ✓ Причины для использования стабов
- ✓ Функциональные методы внедрения
- ✓ Модульные методы внедрения
- ✓ Объектно-ориентированные методы внедрения

В предыдущей главе вы написали свой первый юнит-тест с использованием Jest, и мы внимательнее присмотрелись к тому, удобно ли сопровождать такой тест. Сценарий был довольно простым и, что еще важнее, полностью автономным. Password Verifier не зависел от внешних модулей, и мы могли сосредоточиться на его функциональности, не беспокоясь о других факторах, которые могут вмешаться в его работу.

В той же главе мы использовали первые два типа точек выхода для наших примеров: точки выхода с возвращаемыми значениями и точки выхода на основе состояния. В этой главе рассматривается последняя разновидность — *сторонний вызов*. Кроме того, в главе будет представлено новое требование — зависимость кода от времени. Мы рассмотрим два разных подхода к реализации этого

требования: рефакторинг кода и *исправление ошибок во время исполнения программы* (monkey-patching) без рефакторинга.

Зависимость от внешних модулей или функций может усложнить (и усложнит!) написание тестов и обеспечение их повторяемости, а также сделает их ненадежными.

Внешние сущности, на которые мы полагаемся в своем коде, называются *зависимостями* (dependencies). Они будут более подробно определены позднее в этой главе. К зависимостям могут относиться такие факторы, как время, асинхронное выполнение, использование файловой системы или сети или просто нечто, что очень сложно настраивается либо выполняется в течение очень долгого времени.

3.1. ТИПЫ ЗАВИСИМОСТЕЙ

По моему опыту существуют два основных типа зависимостей, которые могут использоваться нашей единицей работы.

- *Исходящие зависимости* — зависимости, представляющие собой точку выхода нашей единицы работы: вызов логгера, сохранение информации в базе данных, отправка электронной почты, уведомление API или веб-перехватчика о произошедшем событии и т. д. Все эти термины — «вызов», «отправка», «уведомление» — подразумевают действия, направленные *наружу* из нашей единицы работы по сценарию «выстрелил и забыл». Каждое из них соотносится с точкой выхода или конечной точкой конкретного логического потока в единице работы.
- *Входящие зависимости* — зависимости, не являющиеся точками выхода. Они не содержат требований к итоговому поведению единицы работы и существуют только для того, чтобы представить единице работы специализированные данные или поведение для конкретного теста: результат запроса к базе данных, содержимое файла в файловой системе, сетевой ответ и т. д. Все они представляют собой пассивные блоки данных, которые заходят *внутрь* единицы работы как результат предыдущей операции.

На рис. 3.1 представлены эти виды зависимостей.

Некоторые зависимости могут быть одновременно входящими и исходящими — в одних тестах они представляют точки выхода, а в других используются для моделирования данных, входящих в приложение. Такие ситуации встречаются довольно редко, но они существуют — например, во внешнем API, который возвращает признак успеха/неудачи для исходящего сообщения.

Помня об этих типах зависимостей, посмотрим, как в книге «xUnit Test Patterns» определяются разные паттерны для сущностей, которые в тестах должны выглядеть как другие сущности.

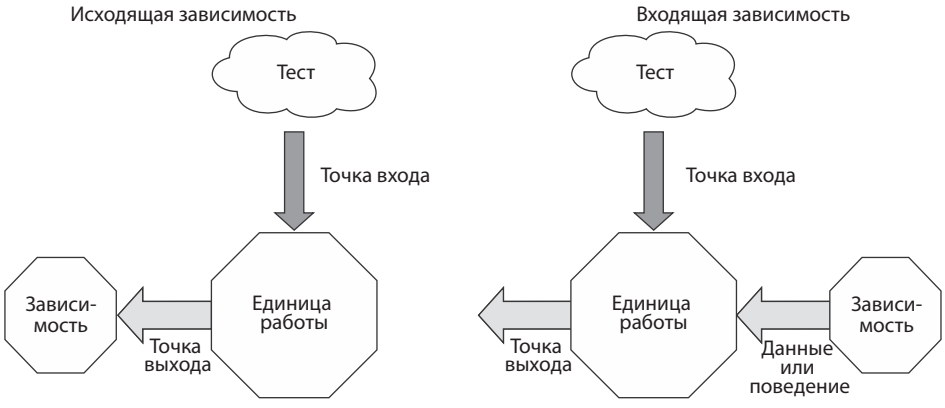

Рис. 3.1. Слева: точка выхода реализована как активация зависимости. Справа: зависимость предоставляет непрямой ввод или поведение и не является точкой выхода

В таблице 3.1 я привожу свою трактовку некоторых паттернов с веб-сайта книги по адресу http://mng.bz/n1WK.

Другая возможная точка зрения на эти термины для оставшейся части книги:

- *стабы* разбивают входящие зависимости (непрямые вводы). Стабы представляют собой фиктивные модули, объекты или функции, поставляющие фиктивное поведение или данные в тестируемый код. Они не используются для проверок. В одном тесте может быть много стабов;
- *моки* разбивают исходящие зависимости (неявные выводы или точки выхода). Моки представляют собой фиктивные модули, объекты или функции, вызов которых мы проверяем в своих тестах. Мок представляет собой *точку выхода* в юнит-тесте. По этой причине рекомендуется иметь не более одного мока на тест.

К сожалению, во многих обсуждениях приходится слышать, что термин «мок» используется как универсальное обозначение как для стабов, так и для моков. Фразы типа «создадим для этого мок» или «у нас есть мок-база данных» могут создать путаницу. Между стабами и моками существуют огромные различия (моки должны встречаться не более одного раза в тесте), и вам следует использовать правильные термины, чтобы ясно показать, о чем идет речь.

Если сомневаетесь, используйте термин «тестовый дублер» или «фейк». Часто одна фейковая зависимость может применяться как стаб в одном тесте и как мок в другом тесте. Пример такого рода будет представлен позднее.

Тестовые паттерны xunit и выбор имен

Книга «xUnit Test Patterns: Refactoring Test Code» Джерарда Месароша — классический справочник паттернов юнит-тестирования. В ней определяются паттерны для фейков, используемых в вашем коде, как минимум пятью способами. Когда вы освоитесь с тремя типами, которые рассматриваются здесь, я рекомендую ознакомиться с дополнительными подробностями, приведенными в книге Месароша.

Заметим, что в «xUnit Test Patterns» дается следующее определение термина «фейк»: «Замена компонента, от которого зависит тестируемая система (SUT), гораздо более легковесной реализацией». Например, можно использовать базу данных в памяти вместо полноценного рабочего экземпляра.

Я же отношу эту разновидность тестовых двойников к стабам и использую термин «фейк» для всего, что реально не существует, практически в том же смысле, что и термин «тестовый дублер», но слово «фейк» короче и проще произносится.

Таблица 3.1. Прояснение терминологии относительно стабов и моков

Категория	Паттерн	Предназначение	Применение
	Тестовый дублер (test double)	Общее название для стабов и моков	Я также использую термин *«фейк»*
Стаб	Объект-пустышка (dummy)	Используется для определения значений, которые должны применяться в тестах только как неактуальные аргументы или вызовы методов SUT	Передайте как параметр точке входа или используйте в части подготовки в паттерне AAA
	Тестовый стаб (test stub)	Используется для независимой проверки логики, зависящей от непрямого ввода со стороны других программных компонентов	Внедрите как зависимость и настройте для возвращения конкретных значений или поведения в SUT
Мок	Тестовый шпион (test spy)	Используется для независимой проверки логики, имеющей непрямые выводы в другие программные компоненты	Переопределите одну функцию для реального объекта и убедитесь в том, что фейковая функция была вызвана, как ожидалось

Таблица 3.1 (*продолжение*)

Категория	Паттерн	Предназначение	Применение
	Объект-мок (mock object)	Используется для независимой проверки логики, зависящей от непрямых выводов в другие программные компоненты	Внедрите фейк как зависимость в SUT и убедитесь в том, что фейк был вызван, как ожидалось

Может показаться, что я привел слишком много информации за раз. Эти определения будут глубже исследованы дальше в этой главе. А пока начнем с малого — со *стабов*.

3.2. ПРИЧИНЫ ДЛЯ ИСПОЛЬЗОВАНИЯ СТАБОВ

Что, если вы столкнулись с задачей тестирования фрагмента кода, приведенного в листинге 3.1?

Листинг 3.1. verifyPassword с использованием времени

```
const moment = require('moment');
const SUNDAY = 0, SATURDAY = 6;

const verifyPassword = (input, rules) => {
    const dayOfWeek = moment().day();
    if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
        throw Error("It's the weekend!");
    }
    // ...
    // возвращает список найденных ошибок...
    return [];
};
```

В нашей системе проверки пароля появляется новая зависимость: система не может работать по выходным. Почему? Кто его знает... Говоря конкретнее, в модуле появляется прямая зависимость от moment.js — очень распространенной обертки даты/времени для JavaScript. Прямая работа с датами в JavaScript — занятие на любителя, и мы можем предположить, что во многих организациях используется похожий код.

Как прямое использование библиотеки для работы со временем влияет на наши юнит-тесты? Проблема в том, что эта прямая зависимость заставляет тесты принимать во внимание правильную дату и время, причем у нас нет прямой возможности влиять на дату и время в тестируемом приложении. В листинге 3.2 показан неудачный тест, который может выполняться только в выходные.

Листинг 3.2. Исходная версия юнит-тестов для verifyPassword

```
const moment = require('moment');
const {verifyPassword} = require("./password-verifier-time00");
const SUNDAY = 0, SATURDAY = 6, MONDAY = 2;

describe('verifier', () => {
    const TODAY = moment().day();

    //тест всегда выполняется, но может ничего не делать
    test('on weekends, throws exceptions', () => {
        if ([SATURDAY, SUNDAY].includes(TODAY)) {
            expect(()=> verifyPassword('anything',[]))
                .toThrow("It's the weekend!");
        }
    });
    //в будни тест даже не запускается
    if ([SATURDAY, SUNDAY].includes(TODAY)) {
        test('on a weekend, throws an error', () => {
            expect(()=> verifyPassword('anything', []))
                .toThrow("It's the weekend!");
        });
    }
});
```

Проверка даты внутри теста

Проверка даты за пределами теста

Приведенный листинг включает две вариации одного теста. Одна проверяет текущую дату *внутри* теста, а в другой проверка выполняется *за пределами* теста, что означает, что тест вообще не выполняется в будние дни, а только по выходным. И это плохо.

Вернемся к одному из положительных качеств тестов, упоминавшихся в главе 1, — стабильности: каждый раз, когда я запускаю тест, это *тот же самый тест*, который выполнялся ранее. Используемые значения не изменяются. Проверки не изменяются. Если код не изменился (код тестов или рабочий код), то тест должен выдать точно такой же результат, как при предыдущих запусках.

Второй тест иногда даже не запускается. Уже этого достаточно, чтобы использовать фейк для устранения зависимости. Более того, мы не можем смоделировать выходной или будний день, что становится более чем убедительным стимулом для переработки тестируемого кода, чтобы он чуть лучше подходил для внедрения зависимостей.

Но постойте, это еще не все. Тесты, использующие время, часто бывают ненадежными (нестабильными, flaky). Они могут не проходить в отдельных случаях, когда не меняется ничего, кроме времени. Этот тест — типичный кандидат для такого поведения, потому что при его локальном запуске мы получаем обратную связь только по *одному* из двух состояний. Если вы хотите узнать, как он ведет себя по выходным, просто подождите пару дней. Н-да.

Тесты могут стать ненадежными из-за граничных случаев, которые влияют на переменные, неподконтрольные нам в тесте. Типичные примеры — сетевые проблемы, возникающие в ходе сквозного тестирования, проблемы с подключением к базам данных или различные проблемы с серверами. Когда тест не проходит, этот факт легко отбросить со словами «просто запустите его снова» или «все в порядке, это просто [вставить какую-нибудь проблему с изменяемостью]».

3.3. ОБЩЕПРИНЯТЫЕ ПОДХОДЫ К СОЗДАНИЮ СТАБОВ

В следующих разделах поговорим о некоторых распространенных формах внедрения стабов в единицы работы. Сначала мы обсудим базовую параметризацию как первый шаг, а затем перейдем к рассмотрению следующих методов.

- *Функциональные подходы:*
  - функция как параметр;
  - частичное применение (каррирование);
  - фабричные функции;
  - функции-конструкторы.
- *Модульный подход:*
  - внедрение модулей.
- *Объектно-ориентированные подходы:*
  - внедрение через конструктор класса;
  - объект как параметр («утиная типизация»);
  - общий интерфейс как параметр (для этого мы воспользуемся TypeScript).

Мы рассмотрим каждый из этих подходов, начиная с простого случая контроля времени в своих тестах.

3.3.1. Создание стаба для времени с внедрением параметра

Я могу представить как минимум две веские причины для контроля за временем в ситуации, описанной ранее:

- чтобы устранить изменчивости в наших тестах;
- чтобы упростить моделирование любого сценария, связанного со временем, для которого нам хотелось бы протестировать свой код.

Простейший рефакторинг, который я только могу придумать, несколько улучшает повторяемость теста. Добавим в функцию параметр `currentDay` для задания

текущей даты. Тем самым мы устраним необходимость использования модуля moment.js в нашей функции, и эта ответственность будет переложена на сторону вызова функции. Тогда в наших тестах время можно будет определять жестко фиксированным способом, а тест и функция станут повторяемыми и стабильными. В листинге 3.3 приведен пример такого рефакторинга.

Листинг 3.3. verifyPassword с параметром currentDay

```
const verifyPassword2 = (input, rules, currentDay) => {
    if ([SATURDAY, SUNDAY].includes(currentDay)) {
        throw Error("It's the weekend!");
    }
    // ...
    // возвращает список найденных ошибок...
    return [];
};

const SUNDAY = 0, SATURDAY = 6, MONDAY = 1;
describe('verifier2 - dummy object', () => {
    test('on weekends, throws exceptions', () => {
        expect(() => verifyPassword2('anything',[],SUNDAY ))
            .toThrow("It's the weekend!");
    });
});
```

Добавляя параметр `currentDay`, мы фактически передаем контроль за временем на сторону вызова функции (наш тест). То, что здесь внедряется, формально называется *пустышкой* (dummy) — это просто данные без поведения, но с этого момента мы можем называть ее «стабом».

Этот подход является разновидностью *инверсии зависимостей* (dependency inversion). Похоже, термин «инверсия зависимостей» впервые появился в статье Джонсона и Фута (Johnson and Foote) «Designing Reusable Classes», опубликованной в журнале «Journal of Object-Oriented Programming» в 1988 году. Термин «инверсия зависимостей» также является одним из паттернов концепции SOLID, описанной Робертом Мартином в статье «Design Principles and Design Patterns» в 2000 г. Высокоуровневые аспекты проектирования будут более подробно рассмотрены в главе 8.

Добавление этого параметра — простой, но весьма эффективный рефакторинг. Кроме стабильности, он предоставляет еще пару преимуществ:

- теперь мы можем легко смоделировать любой день на свое усмотрение;
- тестируемый код не отвечает за импортирование времени, поэтому у него будет одной причиной меньше для изменений в случае перехода на другую библиотеку времени.

Здесь происходит «внедрение зависимости» для времени в нашу единицу работы. Структура точки входа изменяется, чтобы значение дня передавалось

в параметре. Функция стала «чистой» по стандартам функционального программирования в том смысле, что она не имеет побочных эффектов. В чистых функциях встроены внедрения всех их зависимостей — это одна из причин, по которым структуры функционального программирования обычно намного проще тестируются.

Может показаться немного странным, что параметр `currentDay` называется стабом, хотя это обычное целочисленное значение. Но на основании определений из «xUnit Test Patterns» можно сказать, что это «пустышка», а с моей точки зрения, эта разновидность относится к категории стабов. Чтобы быть стабом, необязательно быть чем-то сложным. Просто значение должно находиться под вашим контролем. Это стаб, потому что мы используем его для моделирования некоторого ввода или поведения, передаваемого тестируемому юниту. На рис. 3.2 эта ситуация представлена наглядно.

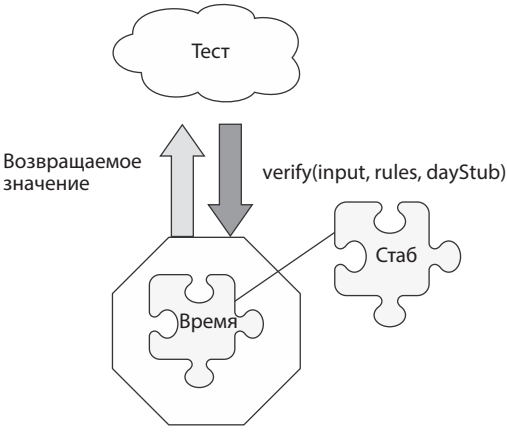

Рис. 3.2. Внедрение стаба для зависимости от времени

3.3.2. Зависимости, внедрения и контроль

В таблице 3.2 снова перечислены некоторые важные термины, которые обсуждались ранее и будут использоваться в оставшейся части этой главы.

Таблица 3.2. Терминология, используемая в этой главе

Зависимости	Сущности, затрудняющие тестирование и ухудшающие сопровождаемость кода, так как мы не можем *контролировать* их из своих тестов. Примеры — время, файловая система, сеть, случайные значения и т. д.
Контроль	Возможность предписать поведение зависимости. Говорят, что создатели зависимостей *контролируют* их, потому что у них есть возможность настроить эти зависимости до того, как они будут использоваться в тестируемом коде

	В листинге 3.1 наш тест *не* контролирует *время*, потому что его контролирует тестируемый модуль. Модуль решил всегда использовать *текущую* дату и время. В результате тест всегда делает одно и то же, что приводит к потере стабильности в тестах. В листинге 3.3 мы получили доступ к зависимости, *инвертируя контроль над ней* через параметр `currentDay`. Теперь тест контролирует время и может принять решение об использовании фиксированного времени. Тестируемый модуль должен использовать предоставленное время, что существенно упрощает наш тест
Инверсия контроля	Проектирование кода с устранением внутреннего создания зависимости и вывода его наружу. В листинге 3.3 представлен один из способов решения этой задачи, основанный на *внедрении параметра*
Внедрение зависимостей	Акт передачи зависимости через интерфейс для внутреннего использования в блоке кода. Место, в котором внедряется зависимость, называется *точкой внедрения* (injection point). В нашем случае используется точка внедрения параметра. Другой термин для обозначения места, в котором может происходить внедрение сущностей, — *шов*
Шов	Термин был предложен Майклом Физерсом в книге «Working Effectively with Legacy Code». Шов (seam) представляет собой место, в котором стыкуются два программных компонента и в котором возможно внедрение. Это место, в котором вы можете изменить поведение своей программы без редактирования ее «на месте». Примеры — параметры, функции, загрузчики модулей, замена функций, в объектно-ориентированном мире — интерфейсы классов, открытые виртуальные методы и т. д.

Швы в рабочем коде играют важную роль для облегчения сопровождения и улучшения читабельности юнит-тестов. Чем проще изменять и внедрять поведение или специальные данные в тестируемый код, тем проще будет писать, читать, а позднее и сопровождать тест при изменении рабочего кода. Некоторые паттерны и антипаттерны, относящиеся к проектированию кода, рассматриваются в главе 8.

3.4. ФУНКЦИОНАЛЬНЫЕ МЕТОДЫ ВНЕДРЕНИЯ

На данный момент решение оставляет желать лучшего. Добавление параметра решило проблему зависимости на уровне функции, но теперь каждая вызывающая сторона должна знать, как работать с датой. Такое решение получится длиннее, чем хотелось бы.

JavaScript поддерживает два основных стиля программирования — функциональный и объектно-ориентированный, поэтому я буду представлять решения в обоих стилях там, где это имеет смысл, а вы сможете выбрать тот вариант, который лучше подходит для вашей ситуации.