**[Тестируемая](#содержание-компонентов) архитектура. Часть 1 Проблематика**

Второй закон термодинамики гласит: "Невозможен процесс, единственным результатом которого являлась бы передача тепла от более холодного тела к более горячему". Это связано с тем, что вселенная стремится к повышению энтропии или меры неопределенности (равновесия, другими словами). Если, например, не убирать в квартире хотя бы неделю, то поверхности покроются пылью, накопятся разбросанные вещи и прочие результаты жизнедеятельности.

Удивительно, но в исходном коде программ происходят аналогичные процессы. Любой разработчик знает, что со временем над проектом становится все сложнее работать, реализация новых требований начинает даваться с большим трудом. А с каким страхом мы вносим изменения в то, что уже сделано? С каким отвращением и горечью иногда приходится работать с функцией, написанной разработчиком-неумехой (а иногда оказывается, что сам когда-то все это и написал)! Как исправление одного дефекта порождает несколько новых… Какие решения иногда порождает воображение лишь бы случайно, чего лишнего не сломать, но реализовать то что требуется. Смысла продолжать дальше нет. Любой сколько-нибудь опытный разработчик и так поймет, о чем речь.

Является ли данная негативная тенденция неизбежной частью жизни любого проекта и программиста? Или, быть может, инженер способен взять ситуацию в свои руки, замедлить устаревание проекта или даже повернуть данный процесс вспять? Для того чтобы ответить на данный вопрос, необходимо начать с самых основ, а именно из того из чего состоит любая программа.

[**Две ценности**](#две-ценности)

Любую программу можно разбить на две составляющие:

* Поведение — это наблюдаемые пользователем эффекты (изображение на дисплее, звуки, осязательные ощущения и т. п.), то, как воспринимается приложение, то, за что прежде всего платит заказчик, то, на что заводят дефекты при тестировании и что требуется реализовать согласно аналитике.
* Структура — это скрытые от наблюдателя детали реализации, паттерны, библиотеки, структуры данных, фреймворки, другими словами это все то, что формирует поведение.

Отсюда можно вывести следующие категории сложности возникающие при разработке ПО:

* Доменная — *естественная* сложность, присущая моделируемой области.
* Структурная — приобретенная или *искусственная* сложность, исходящая от выбранной разработчиком структуры ПО.

Естественная сложность — неотъемлемый результат работы программиста. Если в требованиях написано, что должно быть условие с тремя исходами, значит так оно и будет, как бы инженер не пытался спрятать данный факт за различными конструкциями. Разработчик редко имеет возможность как-то повлиять на доменную сложность так как не определяет требования к поведению, которые естественную сложность и формируют. К счастью, домен редко становится основной причиной усложнения ПО.

Обычно главной причиной быстрого устаревания проектов, является приобретенная сложность. Иногда она вырастает настолько, что начинает контролировать домен, подминая требования под себя. Например, могут отклоняться куски функционала по причинам: "архитектура под такое не проектировалась", "это не планировалось" и "на перенос кнопки уйдет больше месяца работ". Насколько последнее является шуткой — решать вам.

Причин и точек влияния структуры программы на ее сложность может быть масса, при всем старании нельзя перечислить их все. Но для наглядности вот некоторые из них:

* избыточность паттернов, парадигм, состояний, сущностей и структур;
* хрупкость системы, где изменение, скорее всего, породит массу не очевидных дефектов, как в runtime, так и во время компиляции;
* ошибки в моделировании предметной области;
* несоответствие используемых подходов и компетенций разработки на проекте.

[**Инструмент изменения ПО**](#инструмент-изменения-по)

Единственный способ борьбы с ростом искусственной сложности — модификация её единственного источника, а именно структуры. Данная операция встречается настолько часто, что даже получила отдельное название - рефакторинг.

Отметим что, в случае рефакторинга наблюдаемое поведение не должно быть изменено, ведь это может привести к нежелательным отклонениям — дефектам. Ситуация, при которой ранее работоспособное поведение вдруг ломается, называется регрессом. Именно регресс делает рефакторинг непростым в применении инструментом.

Представим на секунду, что существует специальная программа с одной большой кнопкой “check”. При ее нажатии на экране одним из двух цветов загорается индикатор: зеленый, если система обладает корректным наблюдаемым поведением, и красный, если что-то сломалось. Такой инструмент под рукой существенно упростил бы процесс рефакторинга и как следствие процесс упрощения программы. Многие уже догадались что у такого помощника есть специальное имя – тесты. Давайте попробуем понять, какими свойствами они должны обладать, чтобы приносить разработчикам пользу.

* 1. **свойства**
* Защита от регресса. Одной из основных функций тестов является защита от регресса. Именно она избавляет нас от страха случайно изменить поведение приложения при модифицировании деталей реализации. Вся польза от упрощения структурной ценности сведется на нет, если система при этом потеряет в поведении.
* Сопротивляемость рефакторингу. Тесты не должны мешать (увеличивать время) основному методу упрощения ПО, а именно рефакторингу. Рефакторинг для тестов (как и для обычного пользователя) должен быть незаметным.
* Поддерживаемость. Тесты не должны требовать больших ресурсов на своё написание и изменение, ведь в противном случае вся работа в направлении оптимизации труда потеряет смысл, так как проще будет не писать такие тесты вовсе.
* Быстродействие. Если проверка требует много времени для выполнения, то она будет запускаться редко. Чем реже запуск, тем выше вероятность появления дефекта и, как следствие, тем сложнее его отыскать. В каких-то случаях запуски будут игнорироваться вовсе, сводя все усилия на нет. Это также оказывает негативное влияние на динамику и скорость рефакторинга.

Каждый из показателей можно измерить, это означает что они в сумме могут использоваться как средство оценки качества выбранного метода тестирования на проекте.

Давайте немного отвлечемся и отметим одну достаточно интересную взаимосвязь.

[**Связь ценности ПО и тестирования**](#связь-ценности-по-и-тестирования)

Написание теста, на первый взгляд, имеет очевидную цель — верификация наблюдаемого поведения. На самом деле цель зависит от того, что принимается как *наблюдаемое* в конкретном случае. Так, например, unit-тесты отличаются достаточно сильной привязанностью к структуре системы, потому что исполняют ее в сильно ограниченном окружении и, таким образом, верифицируют не только поведение, но и саму возможность (и простоту) использования компонента в отрыве от его зависимостей. Но не все так просто, существуют программы, часть наблюдаемого поведения которых выражена в их структуре! Они будут представлены немного позднее.

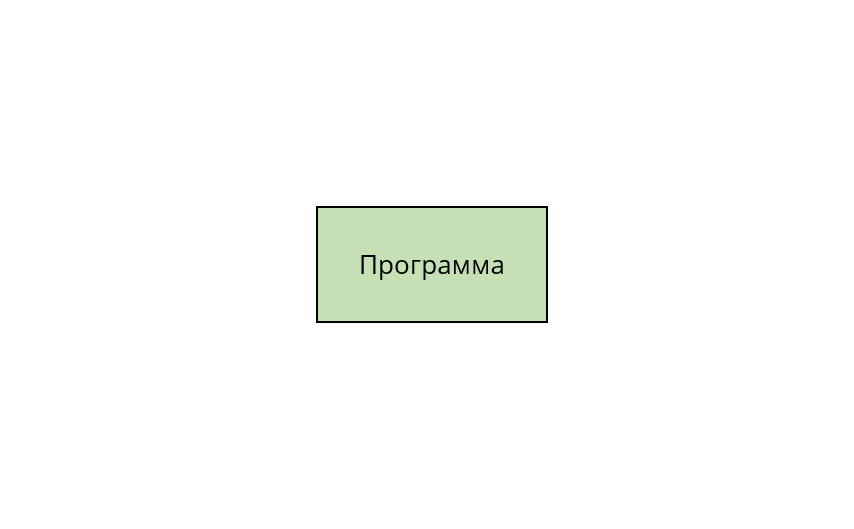
Если подняться выше и задействовать больше элементов системы в проверочных кейсах, абстрагироваться от большинства элементов структуры, то получится уже менее открытая программа, больше напоминающая полупрозрачный ящик, где соотношение связи тестов со структурой и поведением будет в пользу последнего. Это характерно для интеграционных тестов.

Крайней точкой являются тесты, полностью использующие целевую среду выполнения, средства взаимодействия с программой и её восприятия. Таким образом, тесты полностью или в большей части закрываются от структуры ПО и концентрируются исключительно на его поведении.

Из этого можно сделать вывод что классическое разделение тестов на категории есть ничто иное как попытка перечислить разные соотношения ценностей ПО в стратегиях верификации (что проверяется больше, структура или поведение).

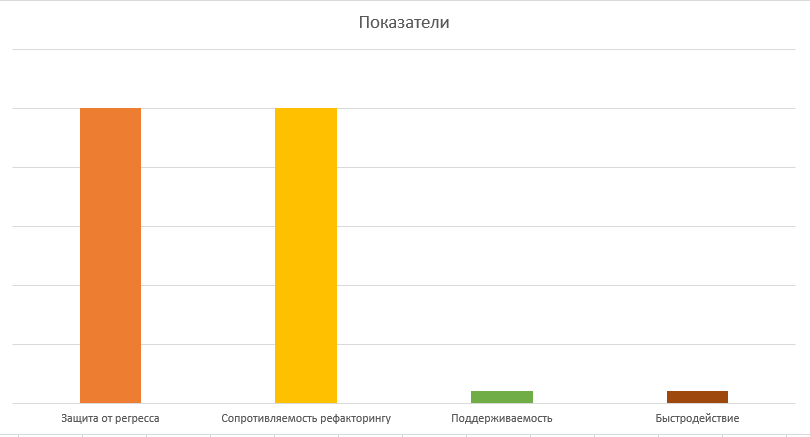
[**Привет, мир**](#привет-мир)

В качестве отправной точки возьмем экстремум, а именно тесты, не зависящие от структуры и поэтому целиком и полностью проверяющие исключительно наблюдаемое поведение (так называемые E2E тесты). Как следствие, они не требуют ничего конкретного от структуры программы, воспринимая ее как один атомарный компонент (монолит):



*Зеленым цветом будут выделяться тестируемые элементы системы (в данном случае тестируется вся система).*

При таких условиях сопротивляемость рефакторингу будет максимальной, равно как и протекция от регрессии. К сожалению, есть и минусы: скорость выполнения тестов и высокая сложность их написания.



*На диаграмме, чем выше столбец, тем больше “очков” оцениваемый метод тестирования набирает в данной категории. Так, E2E предоставляют отличную защиту от регресса (высокий столбец “ Защита от регресса”), но являются сравнительно медленными (низкий столбец “Быстродействие”).*

Вот некоторые из причин:

* Прямое использование всех или большинства компонентов системы естественным образом увеличивает время выполнения проверок, что может негативно сказаться на частоте запусков, а в некоторых случаях и вовсе тесты будут игнорироваться.
* В целом встаёт проблематика методов верификации наблюдаемого поведения. Каким образом проверить что приложение издает правильный звук? Что оно заставляет телефон вибрировать в точности как и когда нужно? Запускает специальные световые индикаторы у устройств? Правильно общается со специальной периферией? Да даже кажущееся простым визуальное тестирование и то имеет множество нюансов, которые разработчики должны учитывать. Всё это негативно влияет на показатель поддерживаемости тестов.
* Факт зависимости от внешних компонентов (например, сервера) также усложняет дело. Ввиду отсутствия прямого контроля, поведение таких систем не детерминировано: нет никакой гарантии, что она будет функционировать на момент запуска теста. Даже в случае работоспособности, отсутствует гарантия прогнозируемого времени и формы ответа.

В сумме данные факторы могут сделать тестирование губительным процессом с точки зрения затрат на разработку, поэтому выбор должен быть тщательно обдуман. В противном случае это может вылиться в большие затраты — и по времени, и по бюджету. Тесты станут скорее очередным источником проблемы, чем методом её решения.

[**Оптимизация**](#оптимизация)

Сам собой встает вопрос: а каким образом можно изменить подход к тестированию так, чтобы поправить отстающие показатели, но сохранить при этом основную пользу тестов? Низкая поддерживаемость и медленная скорость выполнения — основные недостатки E2E. Соответственно, усилия разработчика должны быть направлены в первую очередь в этом направлении.

Удивительно, но данная проблема может быть решена путем разработки подходящей архитектуры приложения, но обо всем по порядку.

[**Источники данных**](#источники-данных)

Первым и одним из самых очевидных элементов системы, тяжело поддающихся тестированию, являются запросы к данным. Это не только запросы к "удаленному серверу" в классическом понимании. Здесь подойдут любые функции, которые обладают двумя свойствами: наличие скрытых аргументов и/или внешних эффектов.

[**Скрытые аргументы**](#скрытые-аргументы)

Функция считается зависимой от скрытых аргументов (скрытых — значит не передаваемых в аргументы функции) в случаях, например, чтения данных из БД, состояние которой может отличаться в зависимости от времени запуска тестов. Другим примером будет процедура, возвращающая текущую дату. Она также зависит от состояния внешнего окружения. С точки зрения тестов, такая функция является сложной в верификации, так как отсутствуют очевидные средства контроля результата её выполнения.

Обратным здесь можно выделить процедуры, которые являются *идемпотентными*. Т. е. такими, результат которых полностью предсказывается их внешними видимыми аргументами.

[**Внешние эффекты**](#внешние-эффекты)

Внешний эффект в общем случае это любой наблюдаемый результат работы функции, существующий за пределами её возвращаемого значения. Это может быть выброшенное исключение, изменение данных в БД, изменение читаемой глобальной переменной и так далее. Возьмем, к примеру, local storage. Запись значения в это хранилище может повлиять на работу системы, тем самым усложняя процесс верификации поведения, но при этом функция, исполняющая саму запись может вернуть просто undefined (т. е. сигнатура функции не отражает её полного поведения).

Возьмем два сценария: добавления пользователя и отображения списка пользователей. Очевидно, что результат будет разным в зависимости от порядка выполнения этих тестов, так как первый выполняет сайд эффект, влияющий на результат второго.

Важно отметить что функции, содержащие сайд эффекты, являются конечным звеном во всех процессах, результатом которых является наблюдаемое поведение. Наблюдаемое поведение – это именно то, что должно верифицироваться при тестировании, это означает что мы не можем в полной мере избавиться от таких функций при тестировании, иначе сам процесс станет бессмысленным.

[**Влияние на тестирование**](#влияние-на-тестирование)

Оба признака в совокупности формируют общее (или глобальное) состояние в системе, что делает невозможным параллельное или селективное выполнение тестов. Порядок и содержание всегда задается строгим образом. Это негативно влияет на время выполнения сценариев тестирования и увеличивает стоимость их поддержки. Также усложняются и сами методы верификации поведения, так как результат тестового сценария может отличаться время от времени (не идемпотентность).

[**План А**](#план-а)

Проблемы, вызываемые *грязными* (не идемпотентными и/или содержащими сайд эффекты) функциями можно попытаться решить, реализуя механизм ручной установки и сброса целевого состояния (setup и teardown) функции. При этом глобальное состояние необходимо изолировать в рамках конкретного потока выполнения тестов. Это требует полного контроля над таким состоянием, что не всегда возможно. Примером могут служить rand-функции или неизменяемые данные (например, история изменений бизнес-сущности).

Реализация предикатов на корректность поведения может отличаться от случая к случаю. Зачастую, поведение верифицируется не полностью чтобы избежать проблем с не детерминированным поведением. Вместо честной визуальной верификации, например, в виде снимка, используются низкоуровневые проверки наличия элементов на странице, основанные на селекторах UI дерева, что выражается в сниженной защите от регресса и, зачастую, повышенной хрупкости таких тестов.

Такие методы решают только часть проблем при этом слабо маскируя остальные. Множество остальных недостатков, связанных с использованием реальных компонентов системы, остаются по-прежнему не решенными и выражаются в негативных свойствах, рассмотренных ранее.

[**План Б**](#план-б)

Можно ли каким-то образом ограничить негативное влияние грязных функций на процесс тестировании, но при этом сохранить тесты полезными? Для этого попробуем выделить не идемпотентные или обладающие внешними эффектами функции во внешний компонент системы и назовем его Источники данных (также известные как *репозитории*). Далее можно сказать, что, все то, что находится в этой области структуры не должно покрываться тестами.



*Обратите внимание что блок не окрашен в зеленый, так как намеренно не покрывается автоматизированными проверками.*

[**Humble object**](https://github.com/rkhaimov/brain-gym/blob/t_arch/plan.md#humble-object)

То, что источники данных не покрываются тестами, неизбежно приводит к пониженной защите от регресса (ведь кода теперь выполняется меньше). Это можно частично нивелировать, упрощая нетестируемые компоненты. Такая техника имеет особое название — Humble object (скромный объект).

Под скромностью здесь понимается отсутствие комплексной логики внутри компонента. Чаще всего такие компоненты характеризуются своей конкретностью (тяжело поддаются расширению, так как ссылаются на конкретные реализации) и простотой (содержат только тривиальные элементы, в основном обычное делегирование внешним функциям). Они тривиальны и у разработчика нет причин изменять их структуру, это и делает бессмысленным их тестирование.

Таким образом, программист должен обратить особое внимание на отсутствие управляющих конструкций в репозиториях и убедиться, что внутри нет ничего, что потребовало бы дополнительного тестирования.

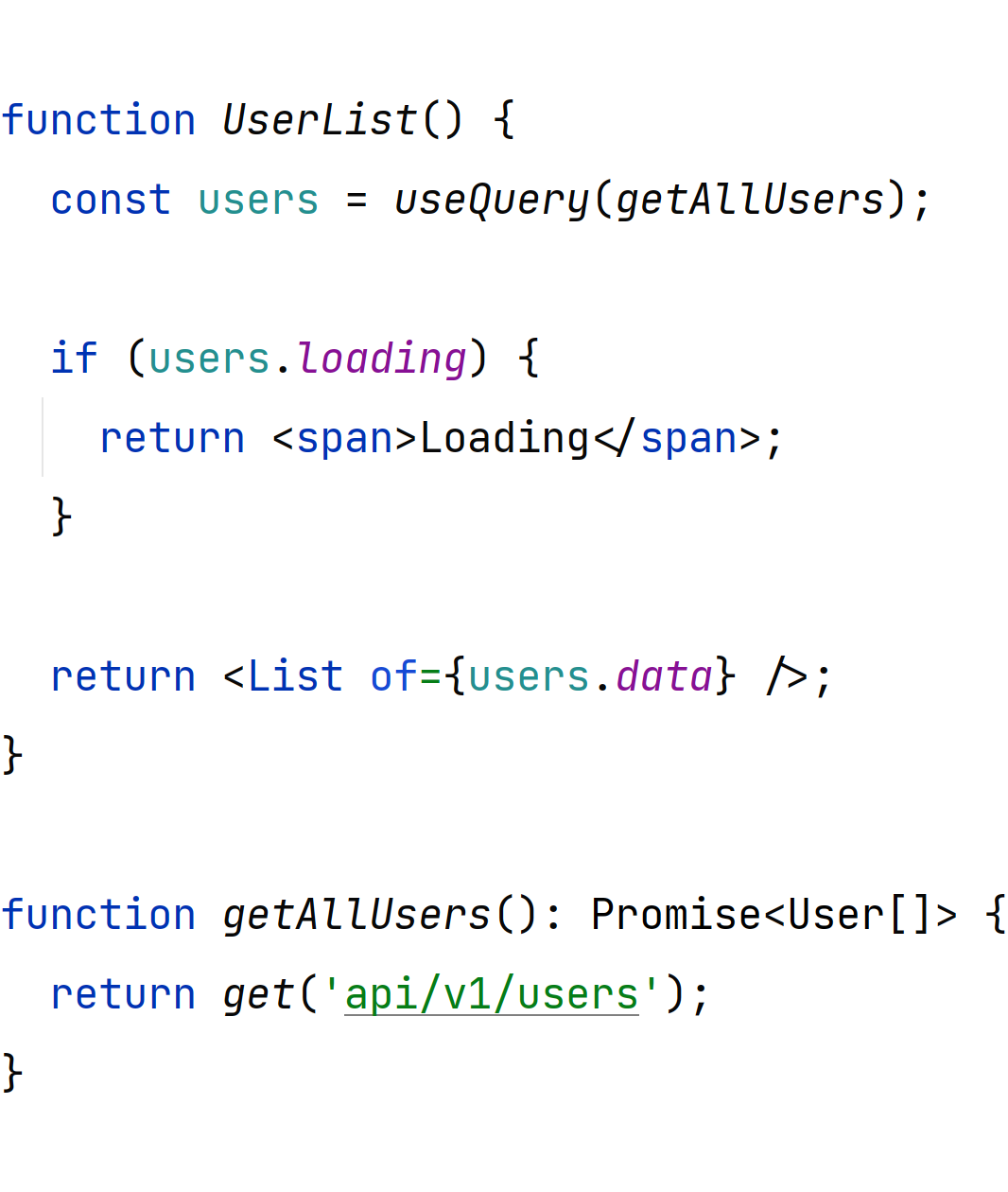
Прежде чем продолжить определимся с некоторыми понятиями.

[**Зависимость от конкретной реализации**](#зависимость-от-конкретной-реализации)

Зависимость в общем случае выражается в коде как явная ссылка на компонент. Компонент может быть обычной переменной, классом, функцией, типом, библиотекой и так далее.



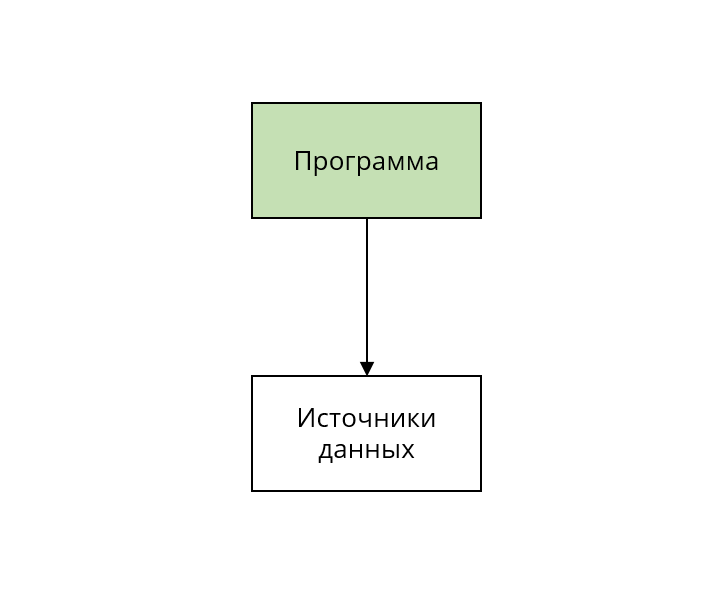
В примере ниже функция UserList напрямую ссылается на конкретную реализацию getAllUsers. getAllUsers скрыто ссылается на изменяемый источник данных, а значит не является идемпотентной.



Заставить компонент использовать иной источник данных (например, более удобный для тестирования), нельзя не меняя при этом исходного кода UserList

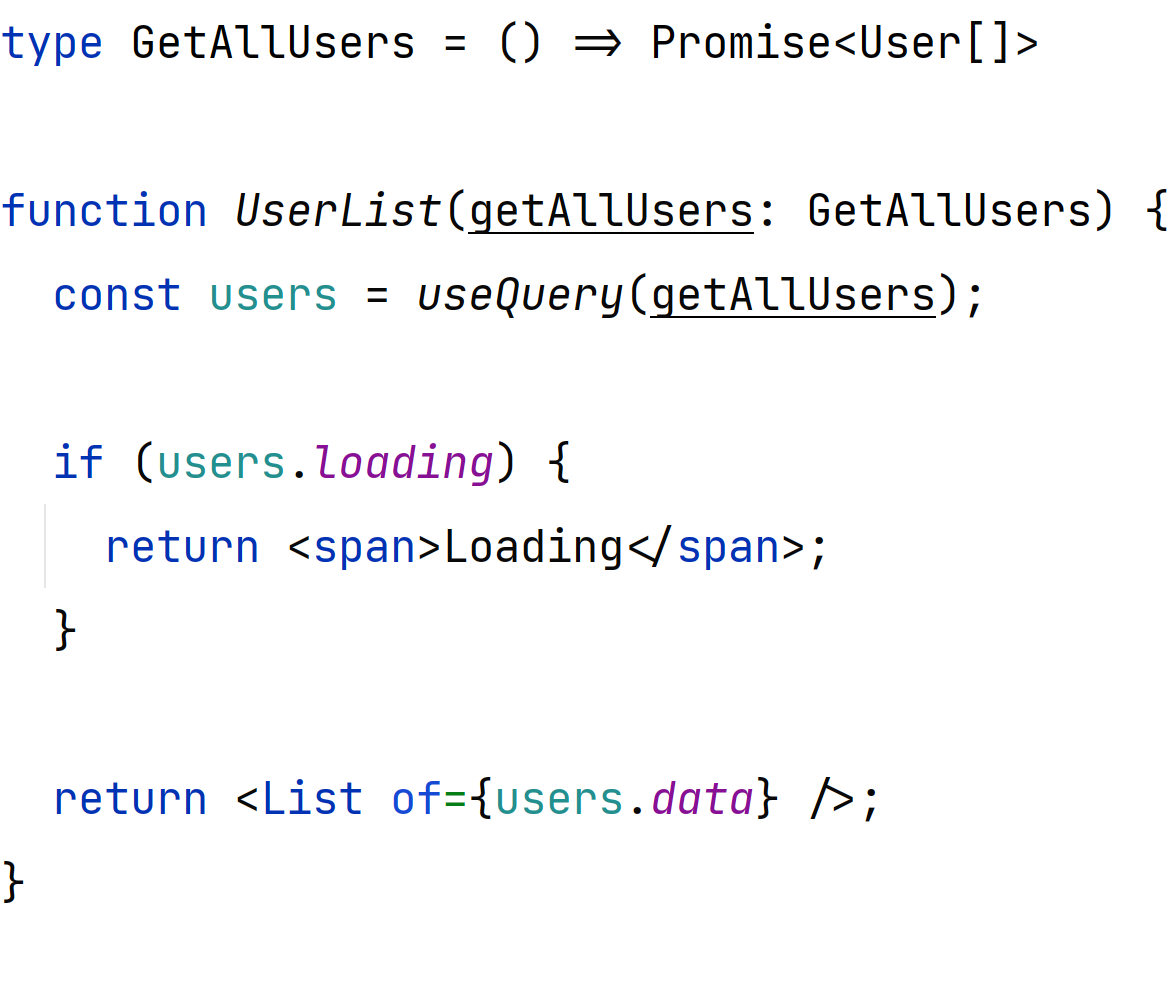
Таким образом, можно сказать, что компонент UserList не является расширяемым с точки зрения данного требования. Это и определяет зависимость от конкретной реализации. В данном случае конкретной функции getAllUsers.

На схеме такая связь отображается следующим образом:

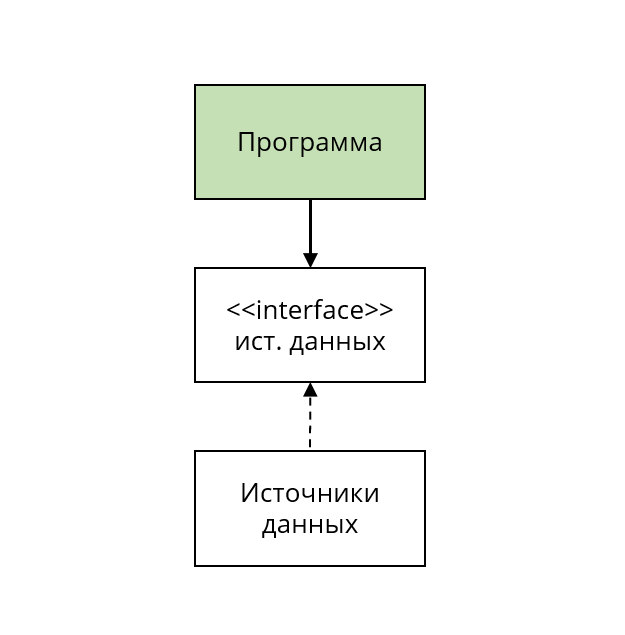


[**Зависимость от интерфейса**](#зависимость-от-интерфейса)

Теперь рассмотрим тот же компонент, но уже с несколько иной структурой:

Теперь функция UserList ссылается не на конкретную реализацию функции, а на ее интерфейс. Это позволяет подменять поведение getAllUsers без изменения исходного кода UserList. Компонент становится расширяемым в данной плоскости.

Изобразить это можно следующим образом:



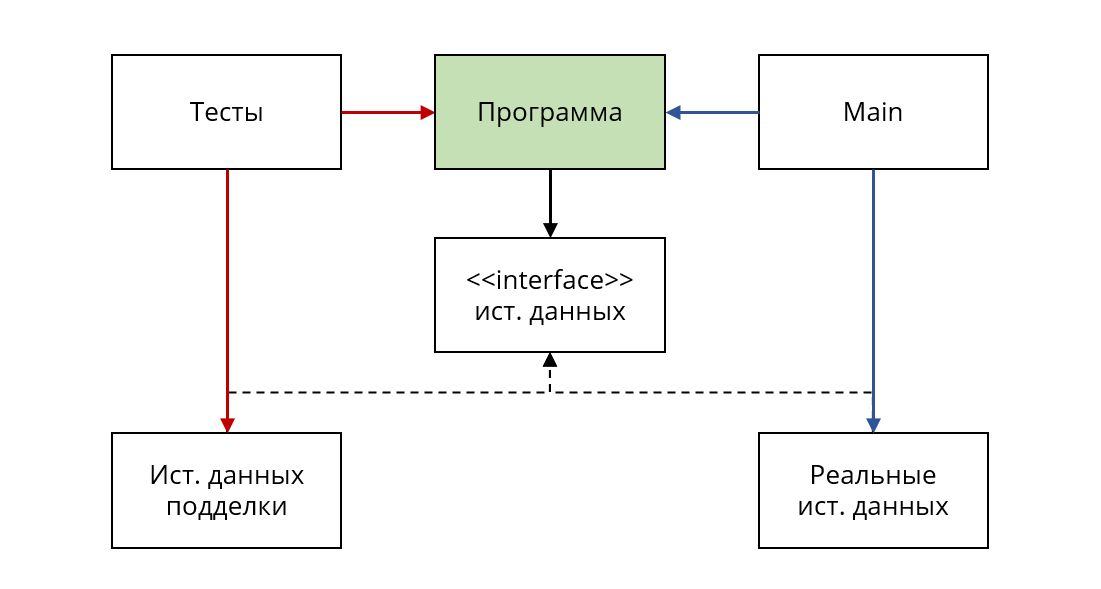
getAllUsers (часть блока источников данных) реализует (или удовлетворяет) интерфейс GetAllUsers (часть блока интерфейса источников данных), который в свою очередь используется в UserList (часть блока программы).

[**Заглушки**](#заглушки)

Расширяемость тестируемого компонента можно использовать следующим образом: можно создать отдельную функцию getAllUsersMock, которая вместо реального запроса к api будет брать данные из локального, неизменяемого источника. Функция легко поддается тестированию, так как лишена проблем, перечисленных ранее. Она является "чистой".



Таким образом у нас образуется два отдельных слоя или группы методов:



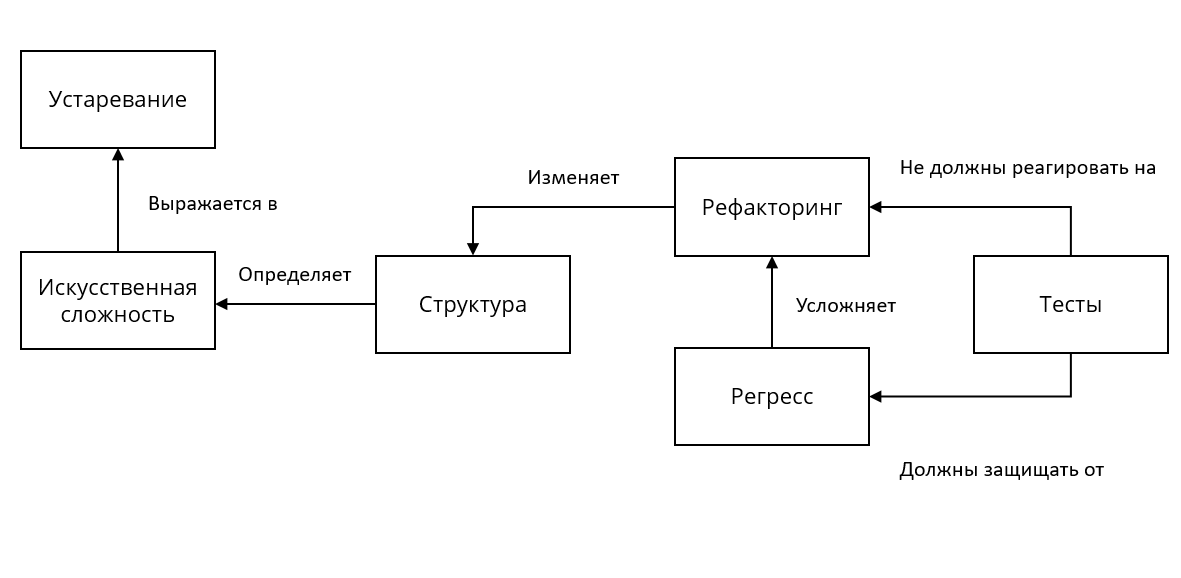
1. Реальные источники данных – методы, работающие с настоящими компонентами/хранилищами в системе, тем самым обладающие негативными свойствами, от которых мы стараемся избавиться. Они используются в *реальном* окружении (компонент Main на диаграмме).
2. Источники данных подделки – чистые функции (с некоторыми оговорками, которые мы рассмотрим позднее), которые реализованы таким образом, чтобы максимально упростить процесс тестирования, путем улучшения показателей поддерживаемости и скорости выполнения.

Какими свойствами обладает получившаяся структура? Являются ли источники данных конечным звеном в формировании наблюдаемого поведения, иными словами, каким образом они должны тестироваться и должны ли вообще? Что насчет верификации так называемой бизнес логики? На эти и многие другие вопросы мы ответим с последующих частях постепенно приближаясь к той самой *тестируемой архитектуре*.

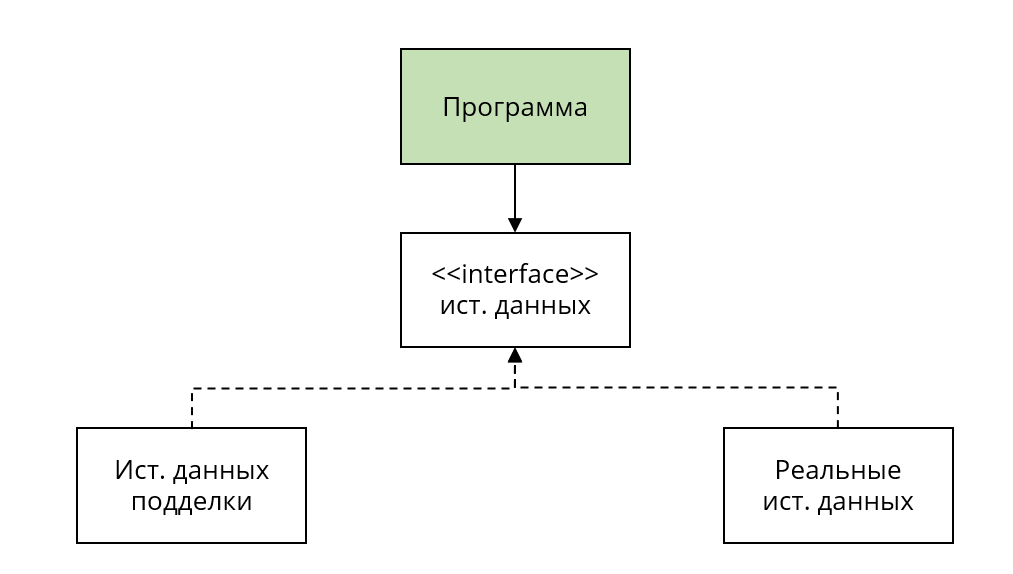
[**Тестируемая**](#содержание-компонентов) **архитектура. Часть 2: абстрактность и наблюдаемое поведение**

В прошлой части мы определили основную проблему: исходный код любой программы со временем устаревает. Это выражается в росте сложности разработки. Данное утверждение не требует доказательств, ведь почти каждый разработчик сталкивался с подобной ситуацией и понимает, о чем идет речь.

Основной способ борьбы с неизбежным усложнением ПО — модификация главной причины — структуры, тех самых деталей реализации, невидимых конечным пользователям. Но рефакторинг чрезвычайно непрост, особенно при работе с достаточно сложными или вовсе незнакомыми проектами. Причина тому — регресс, от которого призваны нас защитить тесты, но не любые, а те, которые обладают достаточными показателями в приведенных ранее свойствах.



Интересно, что разработка таких "правильных" тестов заставляет нас изменять архитектуру определенным образом.

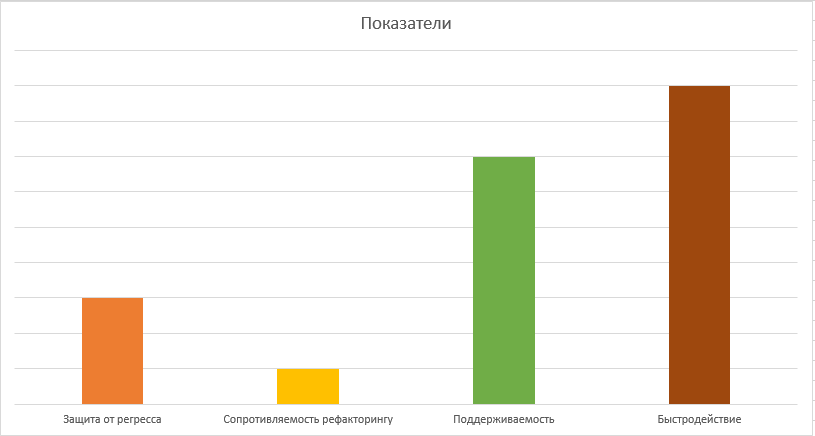


*Архитектура, полученная в конце прошлой части.*

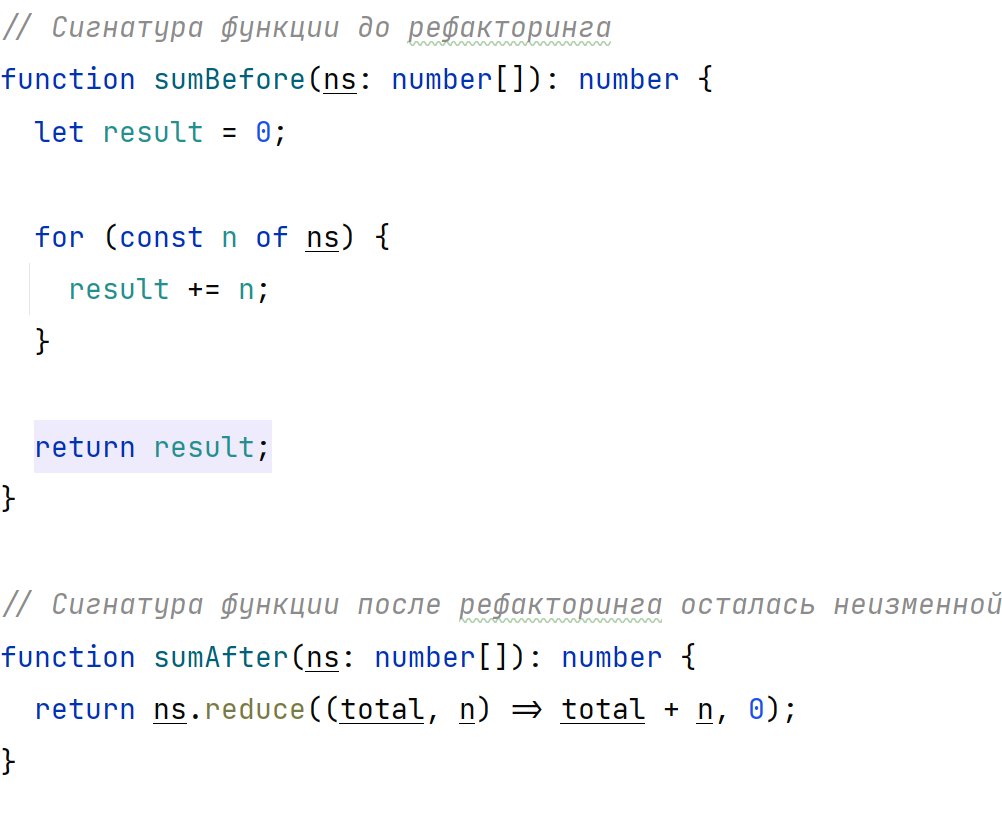
Программа изначально была атомарным монолитом, а теперь разделена на три элемента: источники данных, их контракт и блок "Программа".

[**Компиляция**](#содержание-компонентов) **как метод тестирования**

Давайте ненадолго отвлечемся и отметим один занимательный факт: привычная система типов и процесс статической валидации (компиляции), построенной на ней, — это очередной метод тестирования! Доказательство данного утверждения заключается в наличии самой возможности оценки этого явления с точки зрения четырех основных свойств, описанных ранее:



1. Защита от регресса. Типы — эффективный способ предотвращения и борьбы с дефектами. Степень защиты зависит от следующих параметров:
   1. Строгость системы типов. Чем она строже, тем лучше защита от выполнения *неподдерживаемых операций*. Почему именно операций? А об этом мы поговорим в отдельной серии статей о *теории типов*.
   2. Соответствие модели, представленной в виде типов, реальному положению вещей: общей логике и свойствам домена. При правильном подходе большинство невалидных состояний приложения и композиций элементов будут перехвачены еще на этапе компиляции.
2. Сопротивляемость рефакторингу. Кажется, что типы не могут сопротивляться рефакторингу совсем, ведь они — деталь структуры, то есть то, что при рефакторинге и изменяется. Дело в том, что тип есть некоторое множество значений, объединенных по определенному признаку. Другими словами, это достаточно абстрактная сущность, которая может пережить большое количество рефакторингов и остаться неизменной.



*Сигнатура функции (тип) осталась неизменной, а деталь реализации процедуры поменялась.*

1. Поддерживаемость. Это самый субъективный показатель, ведь целиком и полностью зависит от опыта разработчика, но в общем случае можно считать его достаточно высоким ввиду популярности ЯП со строгой и статической системой типов.
2. Быстродействие. Вероятно, это самое большое преимущество такого метода тестирования. В наши дни программы компилируются за доли секунды, а значит разработчик поймет, сломал он что-то рефакторингом или нет, сразу, как только закончит набирать код в IDE.

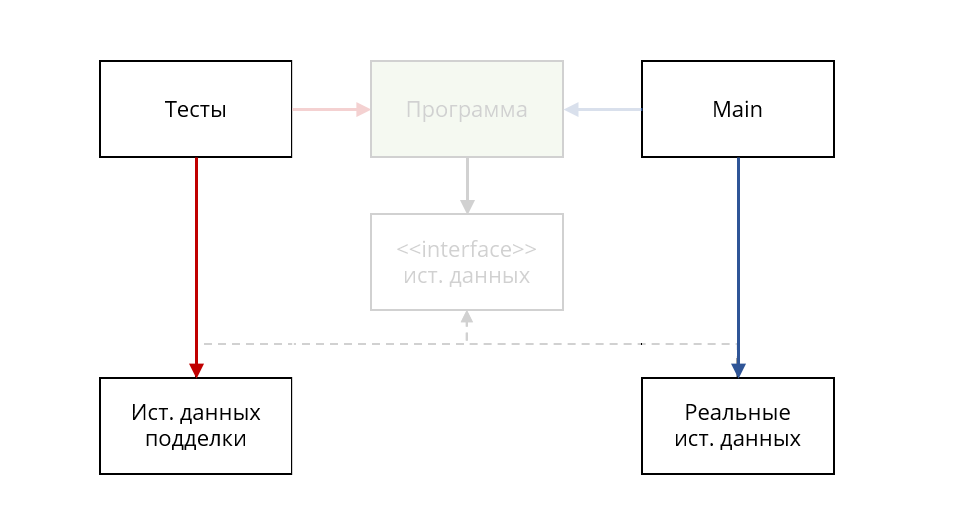
Такие показатели в совокупности делают этот метод тестирования *приоритетным*. Если какой-то дефект можно исключить с помощью типов (еще на этапе компиляции), лучше так и сделать, нежели прибегать к другим, более затратным методам защиты от регресса.

[**Содержание компонентов**](#содержание-компонентов)

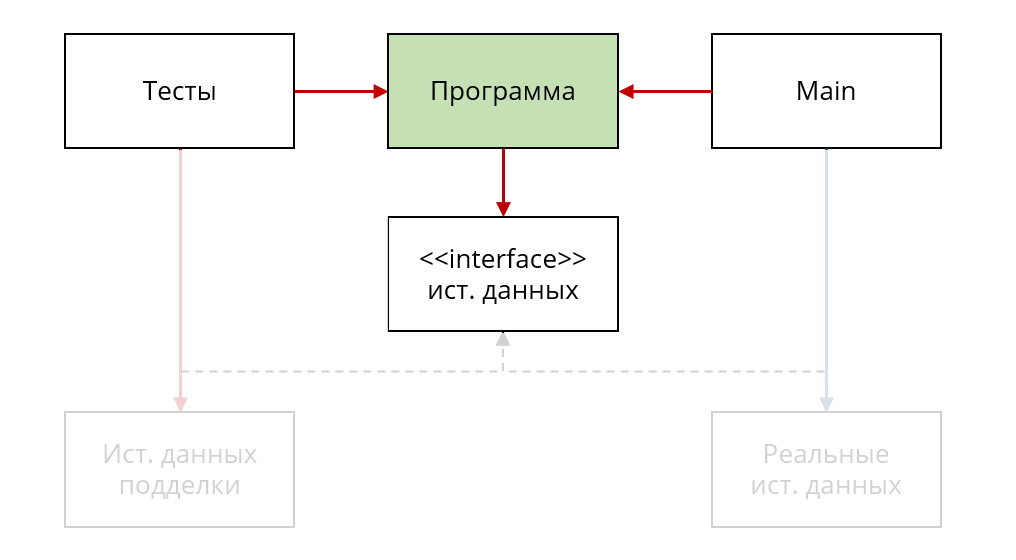
Вернемся к анализу получившейся архитектуры.

Клиентом будет называться импортирующий компонент, а сервером — импортируемый (то есть клиент использует сервер, а не наоборот).

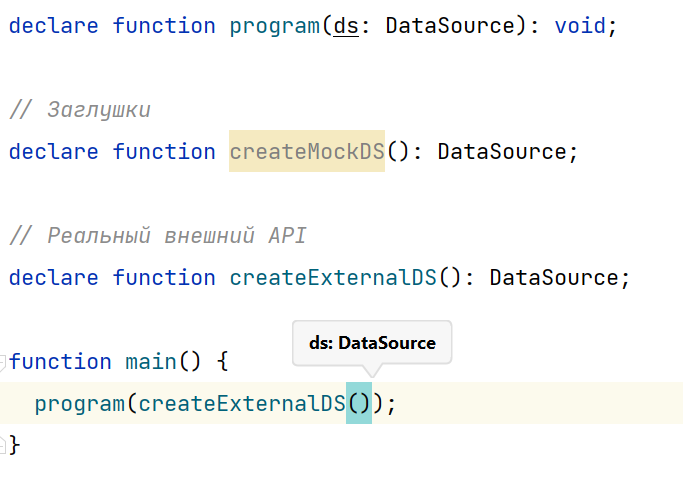
Обратим внимание на то, что у клиентов (Тесты и Main) есть выбор в том, какие конкретные реализации источников данных использовать:



Но вместе с этим у них отсутствует аналогичная опция по отношению к самому интерфейсу. Зависимость от контракта является транзитивной, так как в первую очередь клиенты пытаются использовать блок “Программа”:

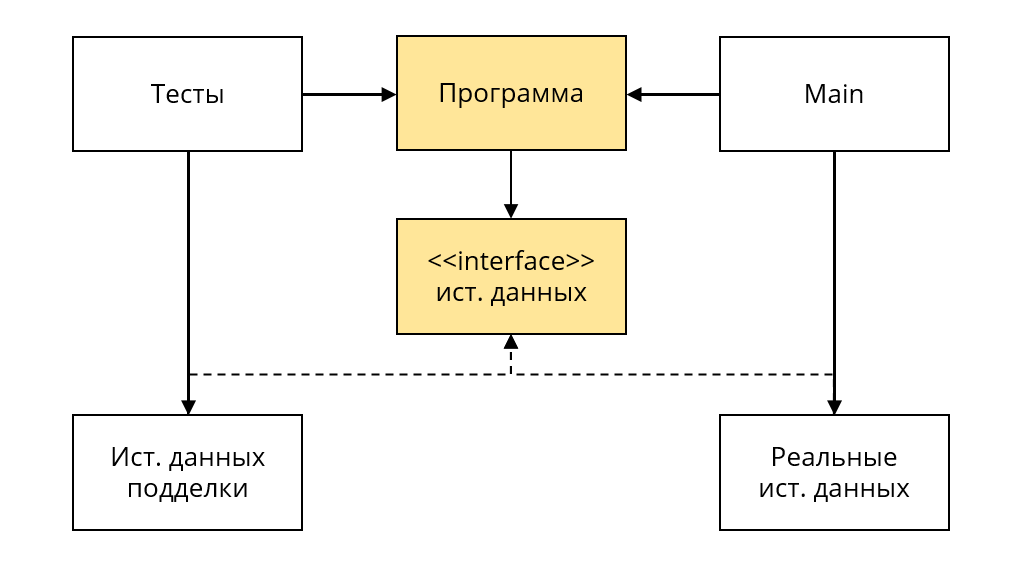


Это можно сравнить с тем, как клиенты становятся зависимыми от типов аргументов тех функций, которые пытаются вызвать:

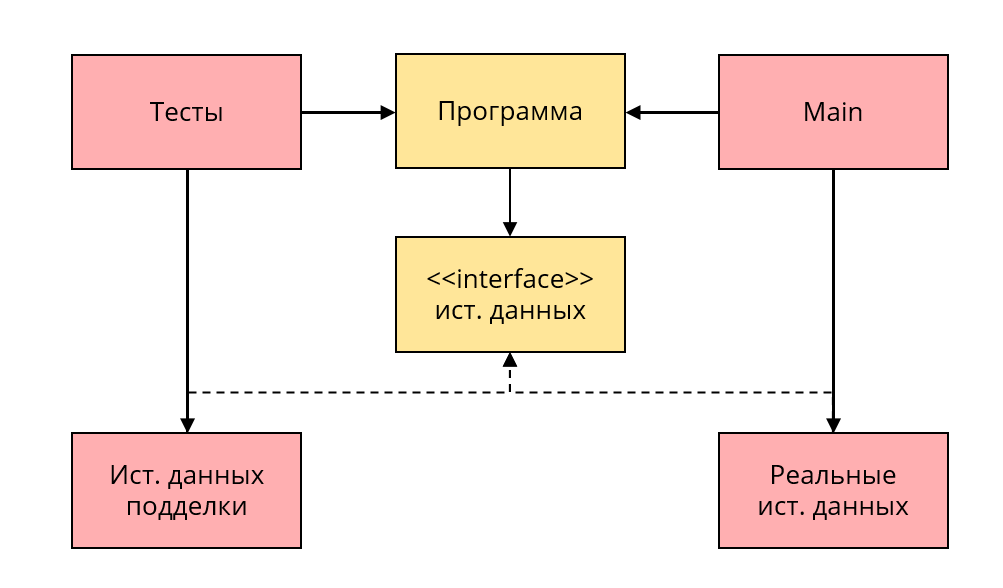


*Main зависит от конкретного интерфейса DataSource (нет возможности выбрать какой-то другой, т. к. именно его требует программа). Конкретная реализация же не фиксирована и зависит от вызывающей стороны (от функции main в данном случае).*

С другой стороны, заметим: как изменение внутренних деталей реализации функции может вызвать изменения в ее внешних аргументах, так и модификация внутреннего устройства блока программы имеет тенденцию (или возможность) вызвать перемены в ее внешнем интерфейсе (контракте для источников данных):



Но изменения в интерфейсе уже породят каскад неизбежных модификаций во всех его имплементаторах и провайдерах, коих обычно гораздо больше.



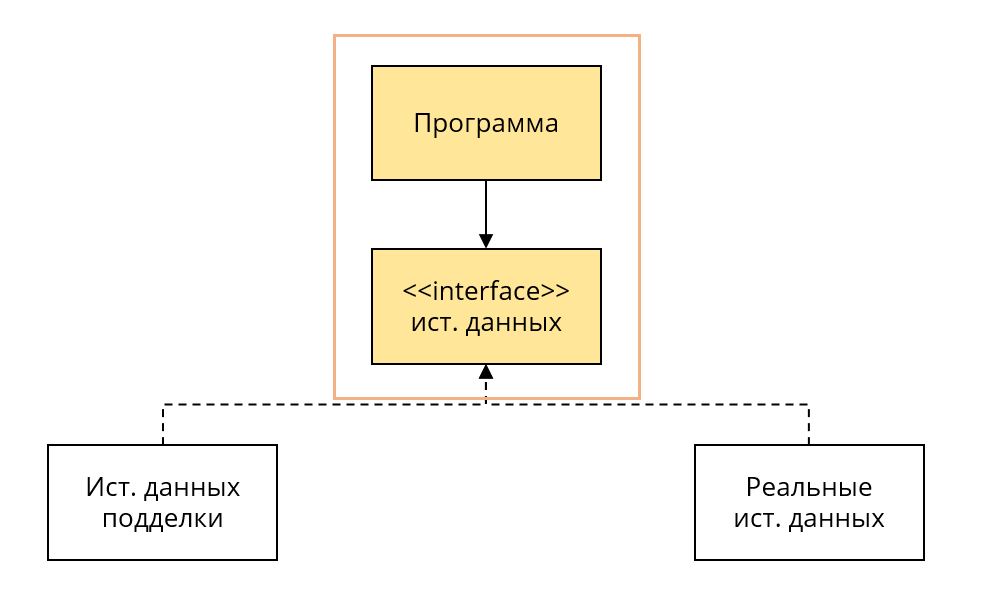
Здесь стоит отдельно упомянуть достаточно дискуссионный факт: единственная причина существования интерфейса — это его клиент. Именно клиент определяет требования, за счет чего он и подчеркивается как источник изменения интерфейса. Без клиента не существовал бы и контракт, и сервер, который его реализует. Можно сказать, что клиент и его внешний интерфейс связаны общей ответственностью.

Конечно, такая точка зрения слишком идеалистична и не учитывает реальное положение вещей в конкретном случае. Тем не менее, в рамках нашего рассказа именно клиент стоит во главе угла, этот факт и определит ход дальнейшего повествования.

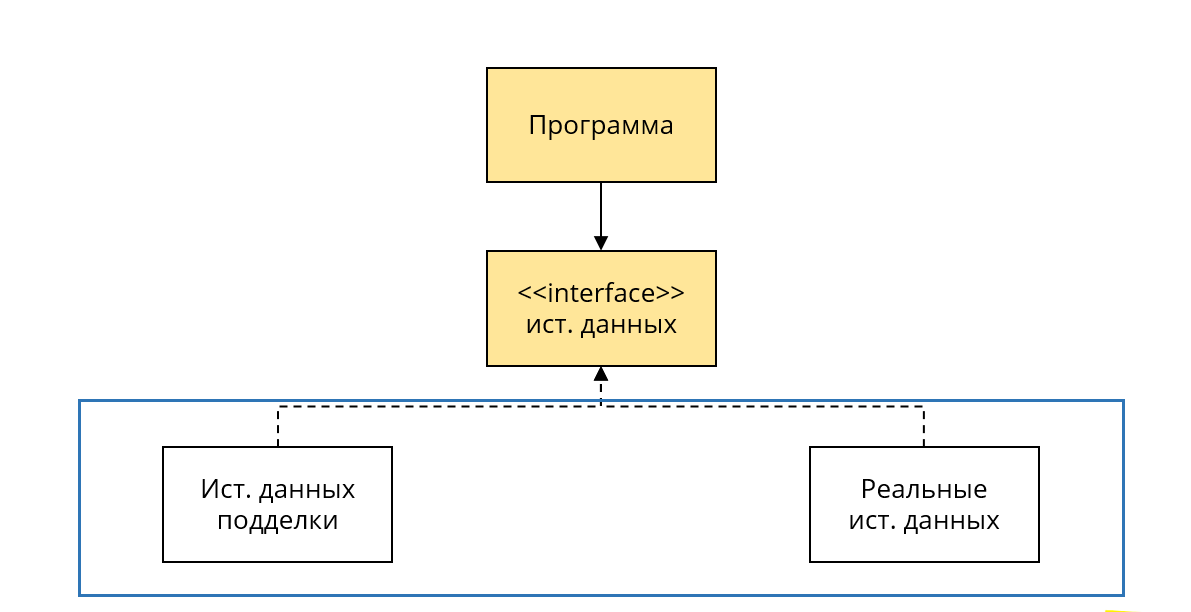
Итого:

* Изменения в блоке "Программа" нередко приводят к переменам и в ее внешних интерфейсах. Это вызывает каскад модификаций в реализациях измененного контракта и его провайдерах.
* Используя программу, клиент неизбежно использует и ее публичные интерфейсы.

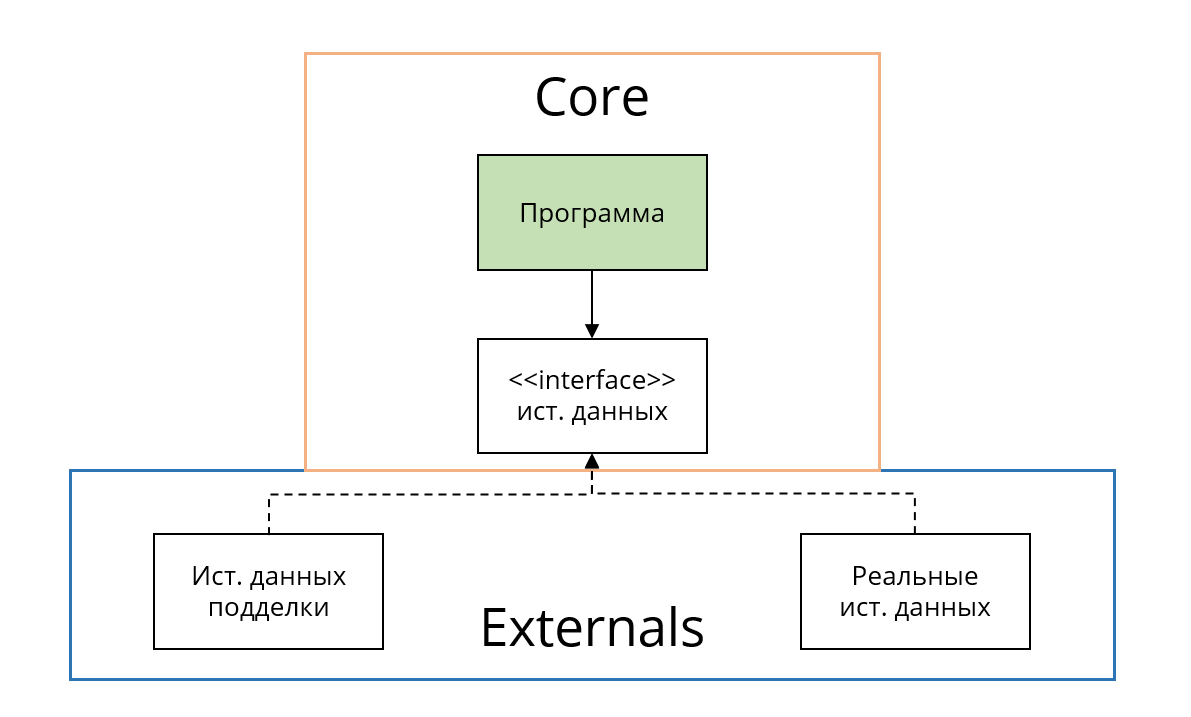
Таким образом можно сказать, что пара (клиент и интерфейс) образует компонент с высокой ответственностью, так называемое ядро системы. Ответственность является *высокой* по одной единственной причине: изменение этого компонента может вызвать многочисленные модификации в зависимых от него элементах, что было показано ранее на схеме.



Это значит, что разработчик должен обеспечить возможность изменять поведение блока программы, не изменяя или как можно реже модифицируя ее исходный код. Другими словами, она должна быть *расширяемой*. И часть этой сложной задачи уже решена в рамках организации тестирования! Программу можно использовать с разными источниками данных, не трогая ничего лишнего:



В итоге образуется два слоя:



1. Слой ядра, тестируемый компонент с высокой ответственностью.
2. Слой источников данных, нетестируемых компонентов. Единственное требование к элементам этой секции — простота. Они должны быть тривиальными, иначе итоговая защита от регресса будет небольшой.

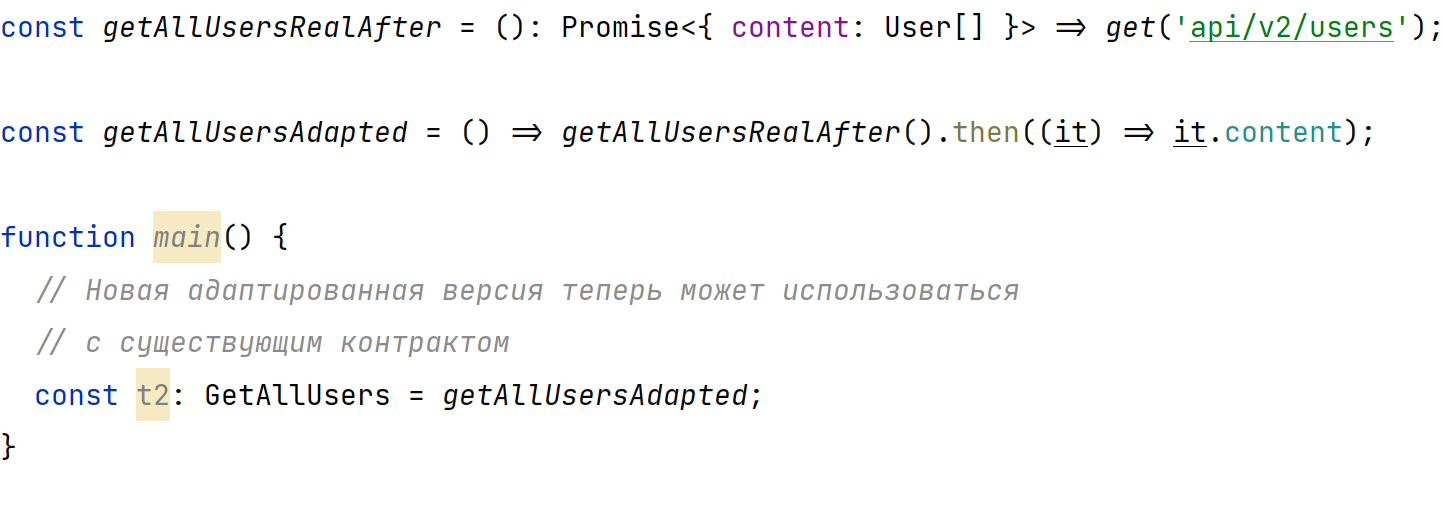
[**Инверсия**](#инверсия) **зависимостей**

Ясно, что теперь можно изменить реализацию сервера (в нашем случае — конкретных имплементаций источников данных), не касаясь при этом исходного кода его клиентов (это доказывается самой возможностью использовать тестовые заглушки). Но что если вместо реализации сервер изменит свой внешний интерфейс?

Представим, что после очередного обновления API функция getAllUsers вместо простого списка пользователей стала возвращать данные в несколько ином формате:



Казалось бы, это изменение не удовлетворяет контракту в программе, а значит придется изменить интерфейс источников данных. Однако проблему можно решить, воспользовавшись техникой под названием "Адаптер":

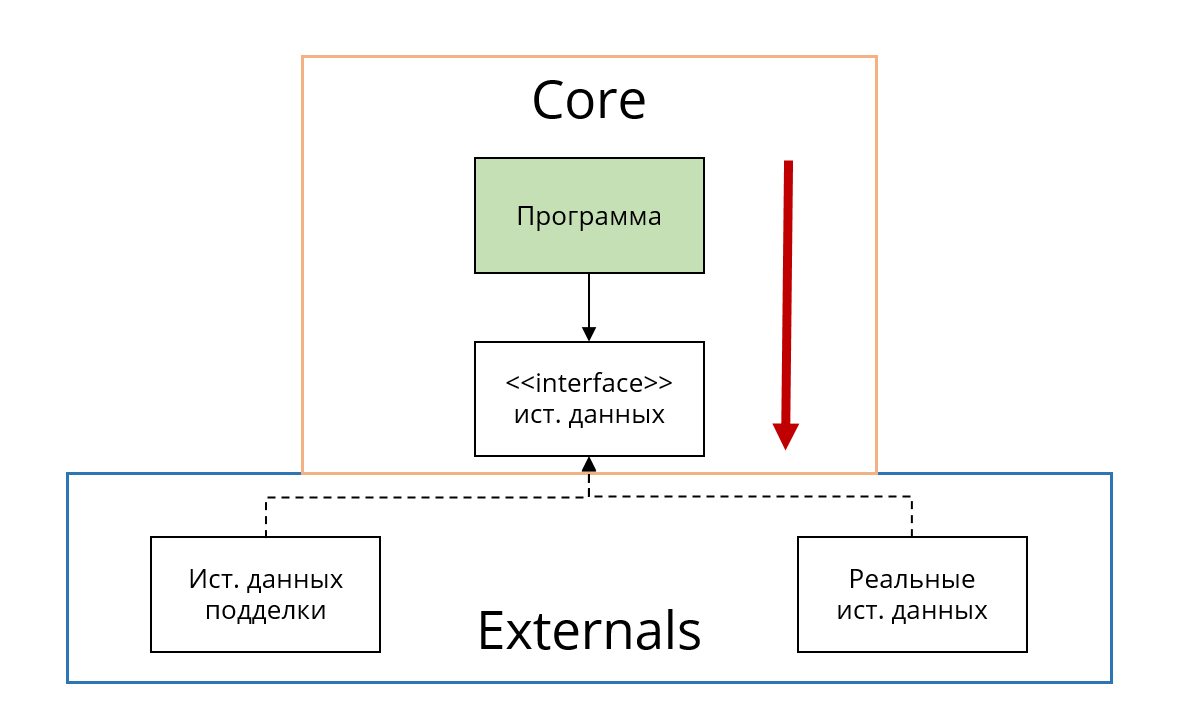


getAllUsersAdapted теперь можно передать ядру без модификаций исходного кода данного слоя.

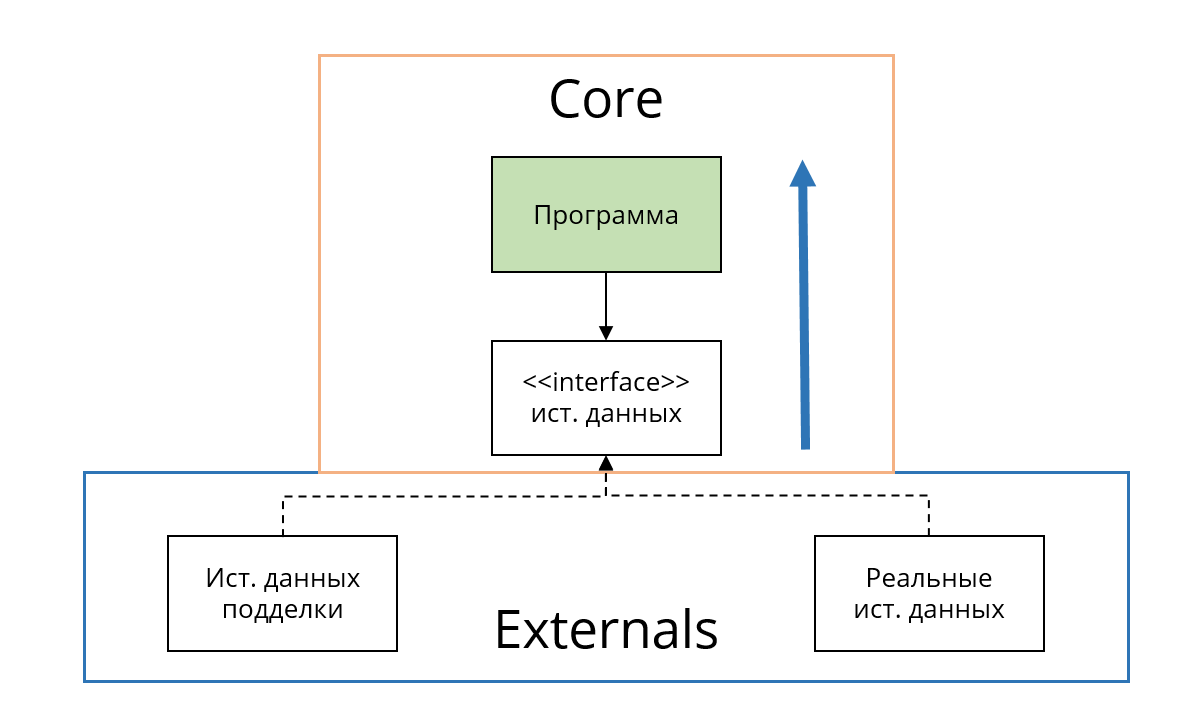
Даже такое серьезное изменение в публичном контракте конкретного источника данных не заставило ядро измениться. Это говорит о том, что оно полностью не зависит от своих серверов, но в то же самое время серверы остаются зависимыми от контрактов, которые этим самым ядром и определяются.

Можно сказать, что появились две картины происходящего:

* На уровне исполнения программы, на этапе runtime, управление передается от Core к Externals (именно ядро вызывает методы репозиториев).



* Но на уровне структуры эти зависимости обращены в противоположную сторону, то есть инвертированы:



Инверсия зависимостей произошла за счет внедрения дополнительного интерфейса и расположения его в границах компонента Core. Без этого направление зависимостей между runtime и compile time было бы совершено идентичным, что сделало бы ядро зависимым от конкретных источников данных, то есть *нерасширяемым относительно данного вектора изменений*.

Таким образом между слоями Core и Externals формируется так называемая *неполная архитектурная граница*. Неполная, потому что Externals ссылается напрямую на элемент слоя Core, на внешний интерфейс. Можно использовать Core с разными Externals, но Externals — только с одним конкретным слоем Core (его интерфейсом если быть точнее).

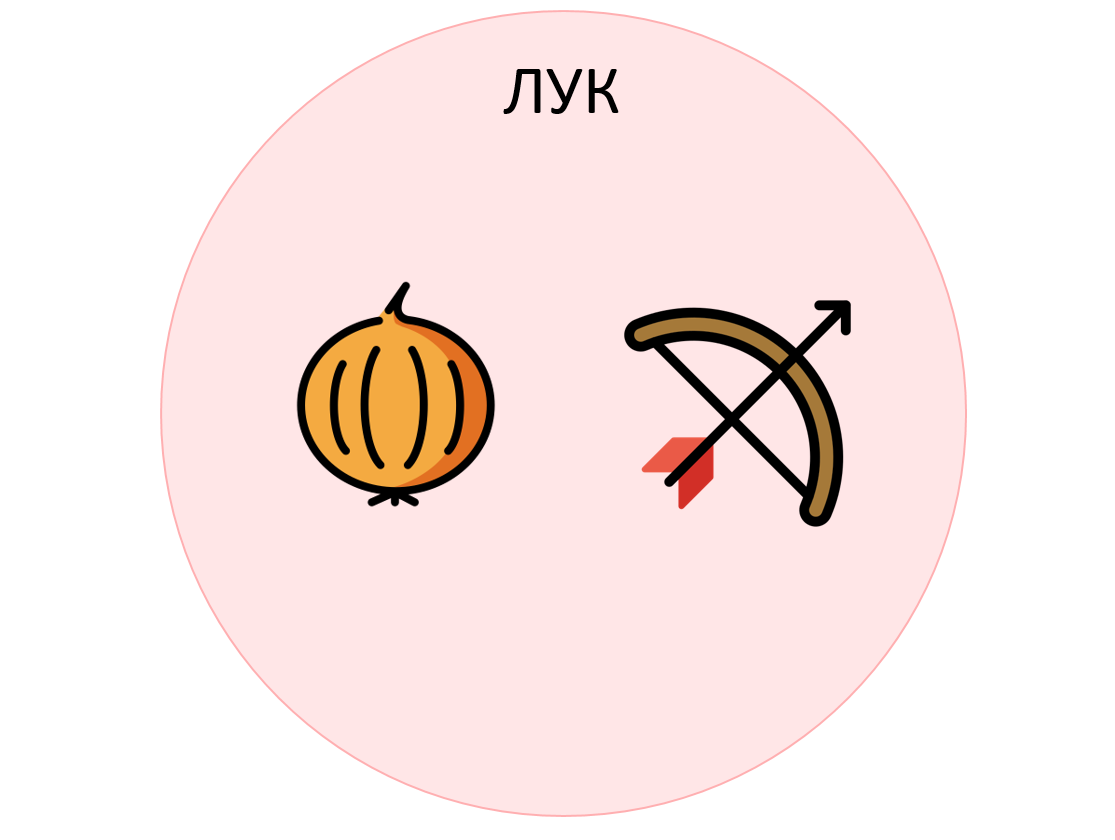
[**Открытость и закрытость**](#открытость-и-закрытость)

Давайте рассмотрим пример. Возьмем слово "лук". Оно состоит из двух компонентов:

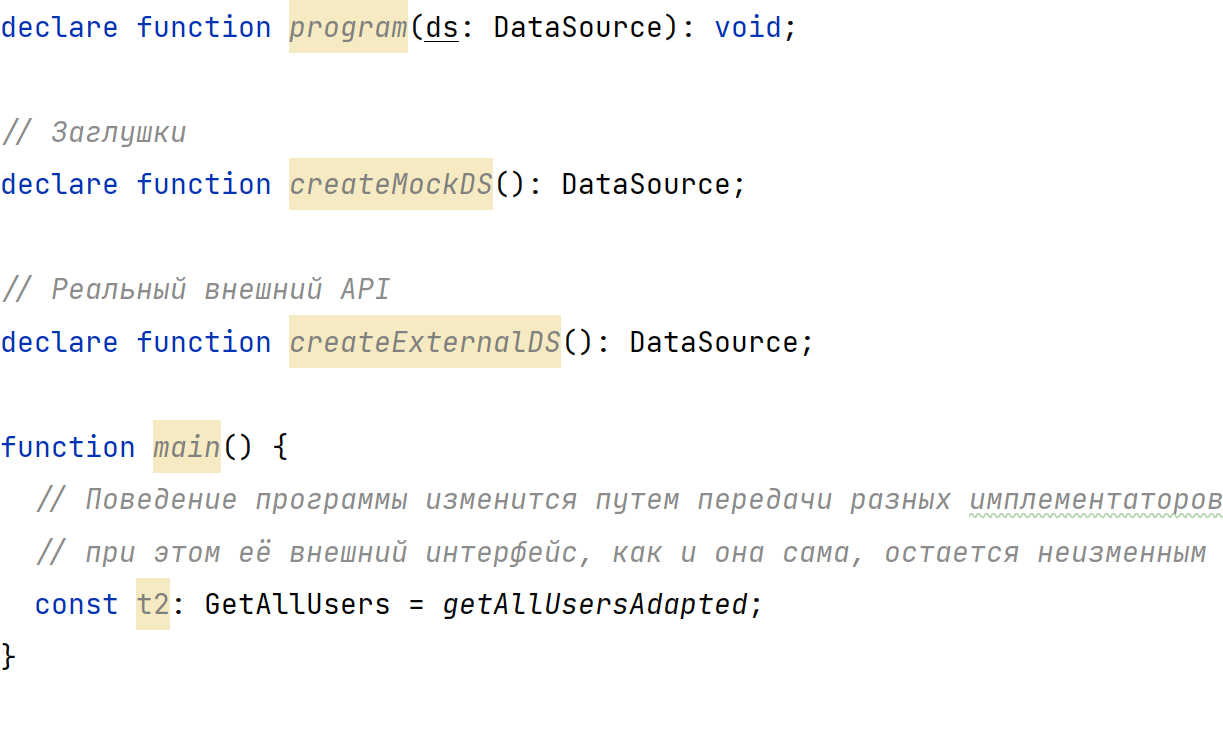
* Форма. Буквы "л", "у" и "к", идущие друг за другом, образуют "лицо" или "внешность" слова, если мы его видим, и его звучание, если слышим.
* Содержание. Смысл слова "лук". Он не фиксирован, а зависит от контекста. Например, в рамках кулинарного шоу речь, скорее всего, пойдет об овоще, а в случае обсуждения соревнований по стрельбе значение будет совершенно иным.

Видно, как при смене контекста форма слова остается прежней, в то время как его содержание изменяется достаточно сильно.

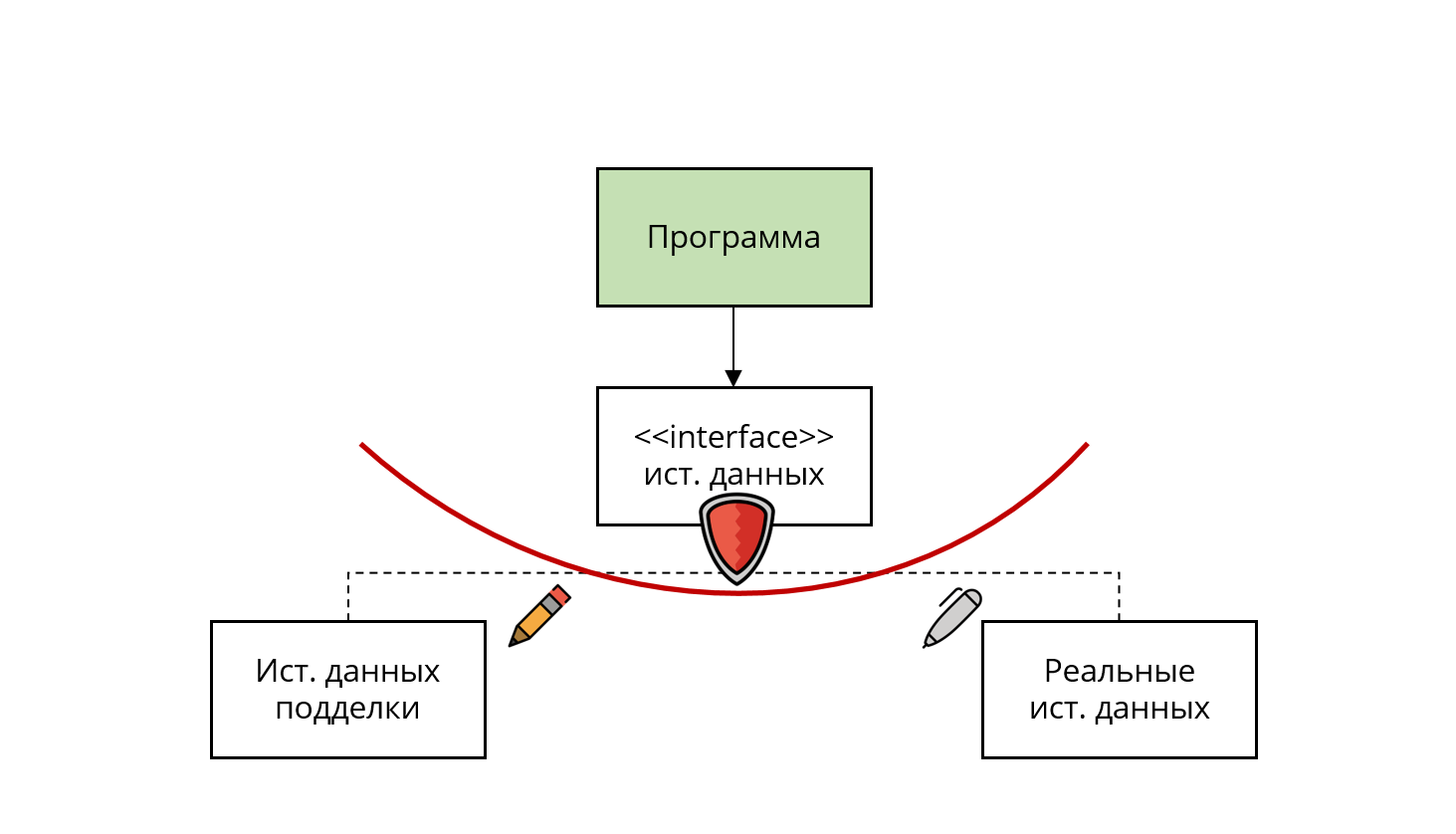
Можно сказать, что "лук" описывает множество, включающее в себя все возможные содержания данного слова, конкретные значения которых выбираются исходя из случая:



В мире ПО интерфейс — это форма, которая описывает множество (в данном случае — множество своих имплементаций). В то время как конкретный имплементатор может меняться от случая к случаю, описывающий его интерфейс будет оставаться нетронутым, как бы закрываясь от факта подмены реализации:

*Можно увидеть ту же картину, как и в примере со словом "лук": форма остается неизменной, в то время как содержание изменяется значительно.*

Факт того, что интерфейс не всегда реагирует на изменения в своих конкретных реализациях, и делает первый более *абстрактным* концептом, то есть более *устойчивым к изменениям* компонентом. Такая устойчивость к изменениям нужна не столько самому интерфейсу, сколько клиентам, которые его используют:



*Интерфейс выступает в роли щита, защищающего своих клиентов от изменений в его имплементаторах.*

Таким образом можно отметить, что инверсия зависимостей сделала Core закрытым от почти любых изменений в Externals, в то же самое время оставляя ядро открытым к добавлению новых наследников его внешнего интерфейса источников данных, то есть к расширению.

Это достаточно полезное свойство описывается принципом открытости-закрытости компонентов (англ. open-closed principle, OCP), который мы рассмотрим в отдельном цикле статей, посвященных принципам SOLID.

[**Наблюдаемое поведение**](#наблюдаемое-поведение)

Отвлечемся от анализа структуры и попробуем понять, каким образом можно построить процесс верификации поведения в получившейся архитектуре.

Для начала стоит отметить что приложения бывают разными, а их поведение и окружение совершенно непохожи. Чтобы добавить конкретики, далее примеры будут основываться на типовом веб-приложении, на его клиентской части. Тем не менее сами принципы достаточно абстрактны, поэтому будут полезны вне зависимости от контекста исполнения.

Одним из главных параметров для того, чтобы выбрать способ тестирования, является граница системы. То есть черта, разделяющая программу и её пользователя.

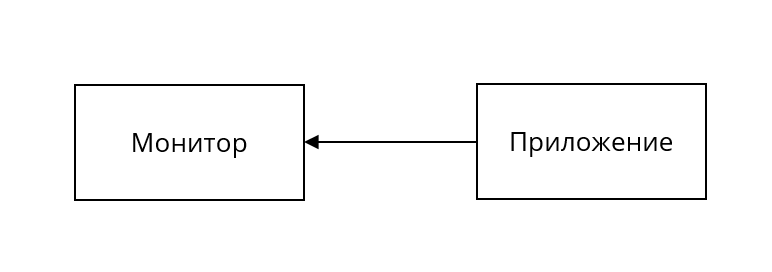
Для разработчика библиотеки утилит коллекций, например, пользователем станет другой разработчик. Таким образом, тесты будут верифицировать поведение, манипулируя исключительно публичным API пакета, как это делал бы реальный клиент библиотеки.

Другой пример — тесты на производительность. Здесь клиентом системы является профилирующий инструмент, он рассматривает поведение системы как, допустим, показатели величины занимаемого процессорного времени и объема памяти.

Если разрабатывается конечное приложение, клиент системы — реальный пользователь, воспринимающий поведение программы через изображение на мониторе. Это и будет наблюдаемым поведением в таком контексте.

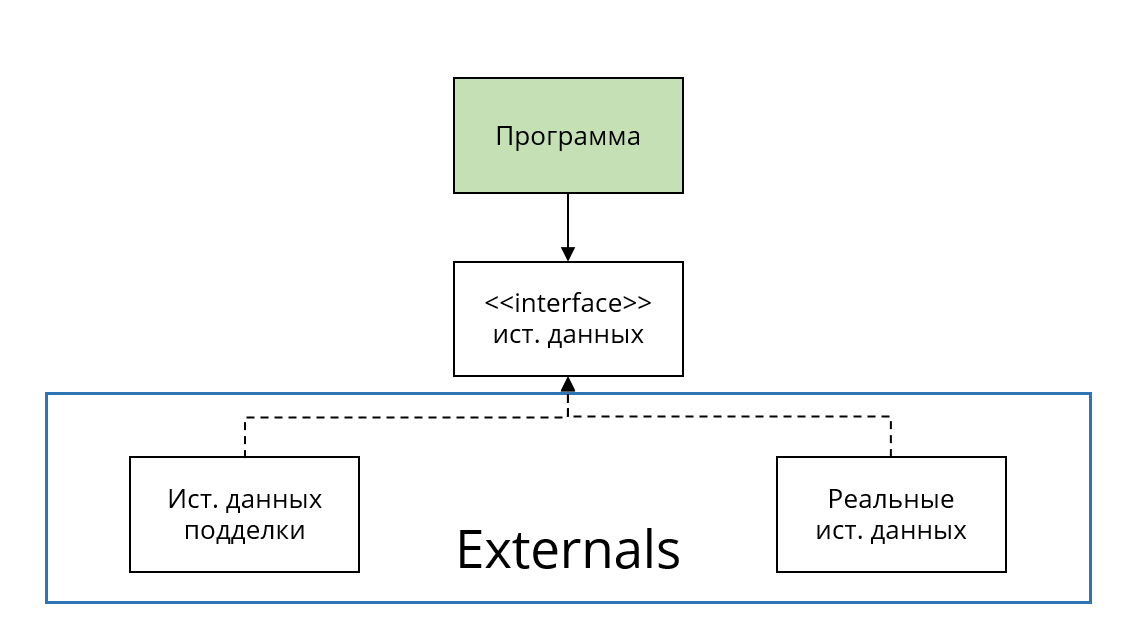
От точки, где была проведена граница, зависят не только конечные показатели эффективности тестирования, но и методы верификации.

Давайте остановимся на типовом клиентском приложении и выделим соответствующие методы верификации:

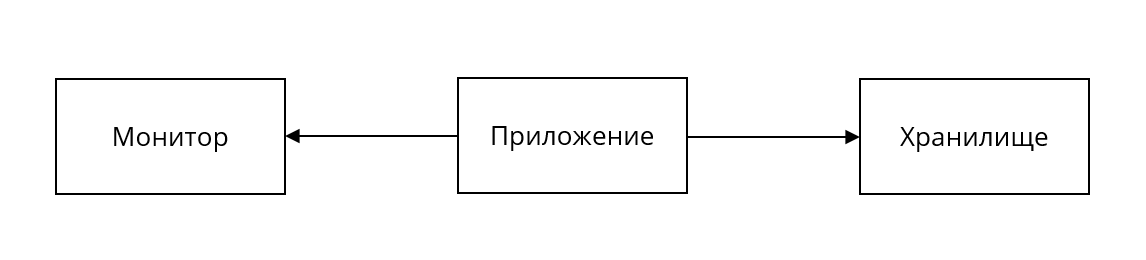


*Направление стрелки соответствует направлению течения информации. В нашем случае информация идет от приложения к монитору, происходит вывод, другими словами.*

Интересно, но монитор — это не единственное место, куда приложение проецирует информацию. Еще одной точкой вывода может стать локальное хранилище или сетевой интерфейс. В приведенной архитектуре методы, работающие с данными сущностями, уже были выделены в отдельный слой Externals.



Можно сказать, что локальное хранилище и сетевой интерфейс — это своеобразный "монитор". Стандартный снимок экрана в нем заменяется простым журналированием мутирующих методов. Важно контролировать то, в каком порядке и с какими аргументами последние были вызваны, ведь все это будет "замечено" сервером, то есть может повлиять на его наблюдаемое поведение (и, как следствие, — поведение его клиентов) в реальных условиях.



Журнал вызовов может быть представлен в следующем виде:

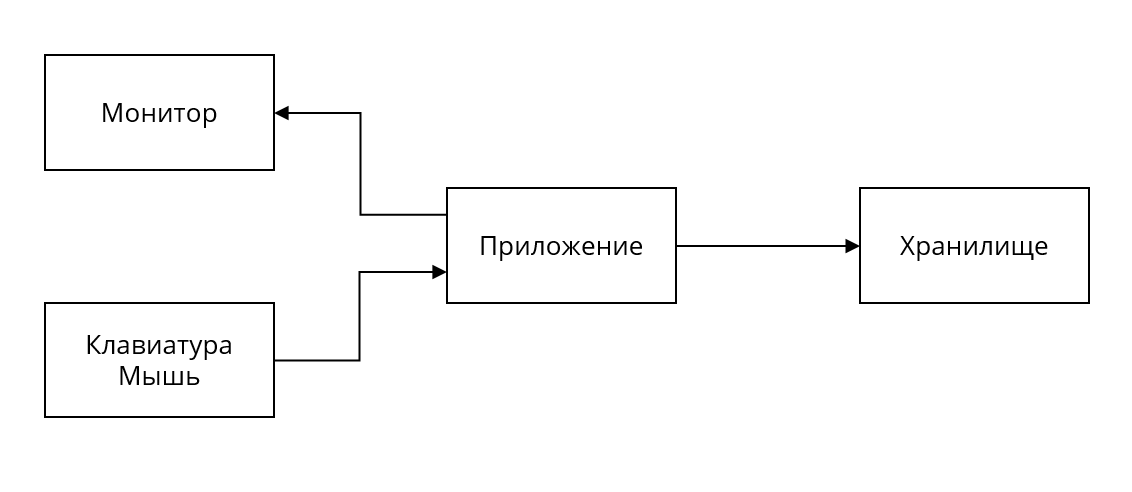


*Методы представлены в массиве, так как важен порядок. То же верно и для аргументов и их значений.*

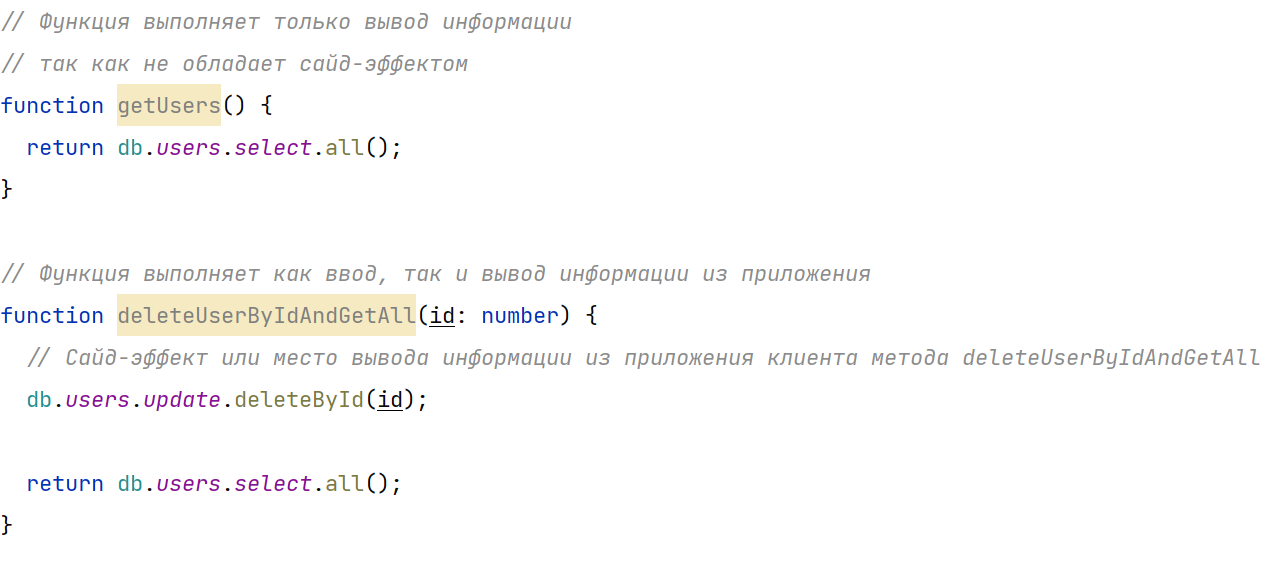
При первом прогоне тестового сценария журнал записывается в так называемый baseline и используется в качестве целевого экземпляра. При последующих запусках тестов актуальные журналы сравниваются с теми, что записаны в baseline. Если они совпадают, то тест считается пройденным, в противном случае возникает ошибка. Таким образом обычно и выполняется верификация поведения для данного источника вывода информации из приложения.

[**Команды и запросы**](#команды-и-запросы)

Помимо вывода информации (будь то монитор, сетевой интерфейс или что-либо еще), система также ожидает и ввода. Очевидными источниками (в случае человека и настольного компьютера) являются, например, клавиатура и мышь:



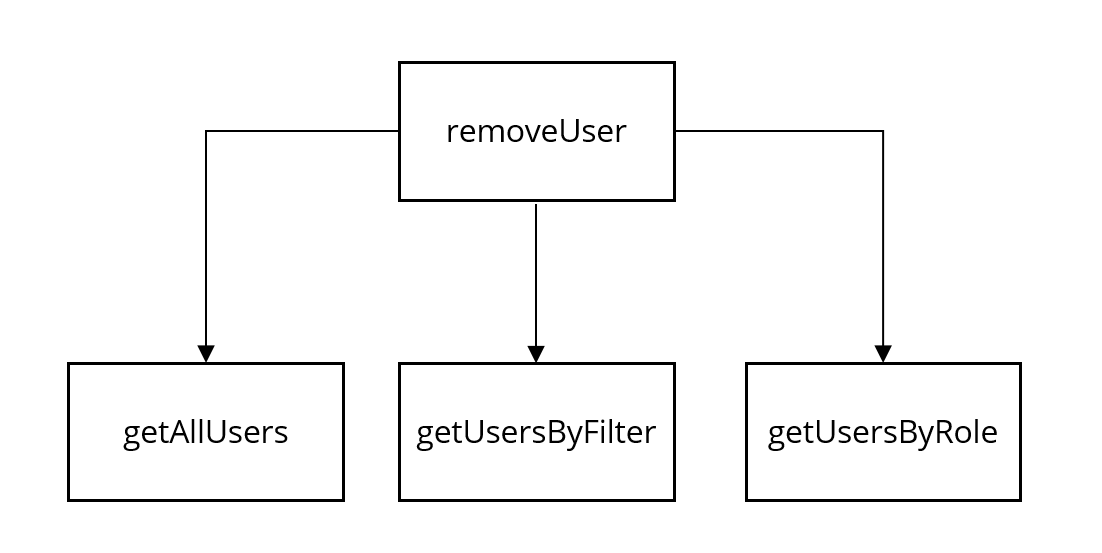
Сохраняя не совсем очевидную аналогию, то же самое можно сказать и про сервер. Вводом здесь будет считаться любой метод, возвращающий информацию. Если говорить про REST, то это любой GET-метод. Важно то, что такая функция не должна обладать сайд-эффектом, в противном случае это будет все тем же выводом информации из приложения.



Для простоты далее будем называть функции ввода информации в приложение запросами (queries), а те методы, которые выполняют вывод, — командами (commands).

* Запросы — это методы, не предполагающие никаких сайд-эффектов. Но они могут быть (и чаще всего являются) не идемпотентными, то есть результат их выполнения не зависит от их аргументов.
* Команды, наоборот, являются методами, обладающими внешним эффектом, при этом для нас совершено неважно, идемпотентные они или нет.

Стоит отметить, что сайд-эффект команды чаще всего проявляется в изменении информации, возвращаемой одним или несколькими запросами. Например, команда "удалить пользователя" изменит результат вызова метода для получения списка этих же пользователей (и других методов, связанных с изменяемым источником данных):



Такое упорядочивание (систематизация) содержимого Externals позволяет организовывать реактивность, кэширование и другие комплексные поведения внутри последнего в более простом и управляемом виде. В случае тестов факт наличия CQS (command and query separation) также может эксплуатироваться дополнительно. (Это будет показано в следующей части.)

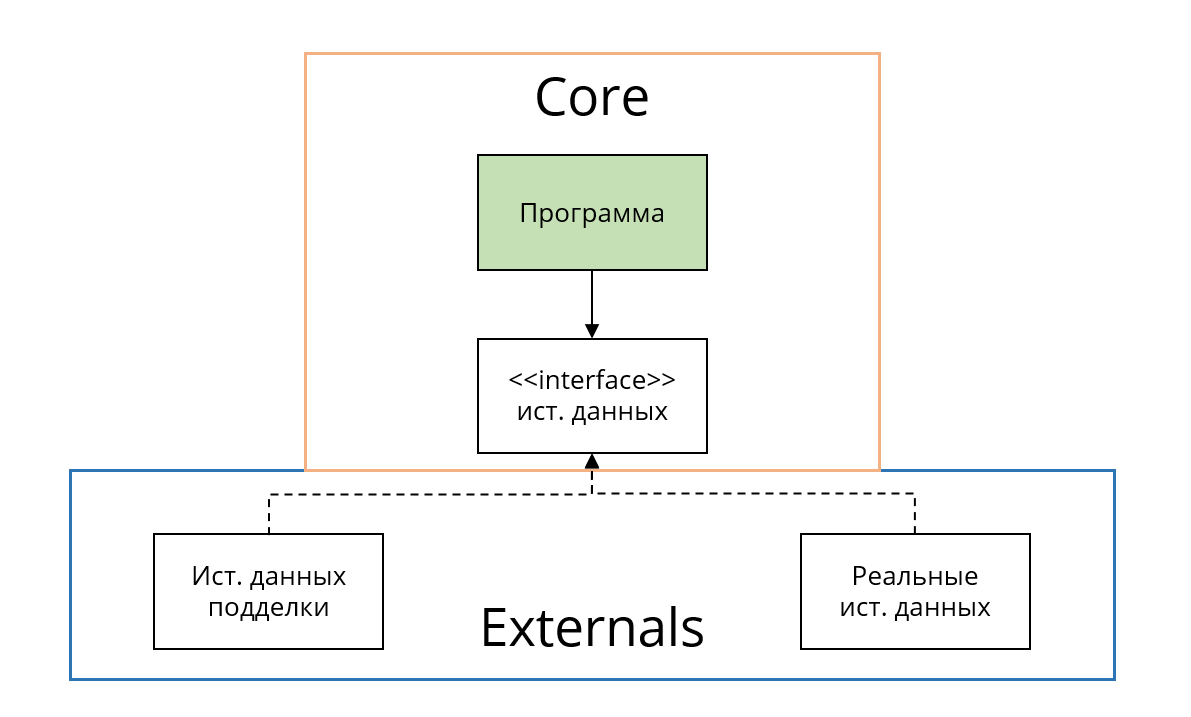


*Итоговая схема портов ввода/вывода приложения.*

Пожалуй, на этом стоит закончить вторую часть повествования. В следующей главе мы поговорим о том, как выстроить процесс верификации поведения на основе приведенной структуры портов I/O, а также обсудим чрезвычайно важную роль принципа подстановки и его влияние на процесс тестирования. Ну и, конечно же, продолжим свое уверенное движение к той самой *тестируемой архитектуре*.

[**Тестируемая**](#содержание-компонентов) **архитектура. Часть 3: граница тестирования**

В прошлой части был проведен анализ архитектуры, итогом которого стало внедрение дополнительного разделения на слои: Core (ядро) и Externals (источники данных):



После чего, в целях имплементации непосредственно самого процесса тестирования в системе были выделены порты ввода/вывода, те самые границы, которые и определят итоговый метод тестирования и его свойства:

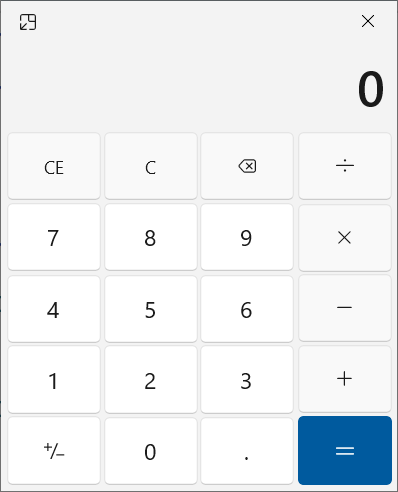


*Область слева обозначает порты, направленные в сторону пользователя, справа, порты с которыми работает сервер*

Но каким образом граница может повлиять на эффективность тестирования? Как выбор интерфейса взаимодействия с программой изменяет показатели свойств тестов: защиты от регресса, сопротивляемости рефакторингу, поддерживаемости и быстродействия?

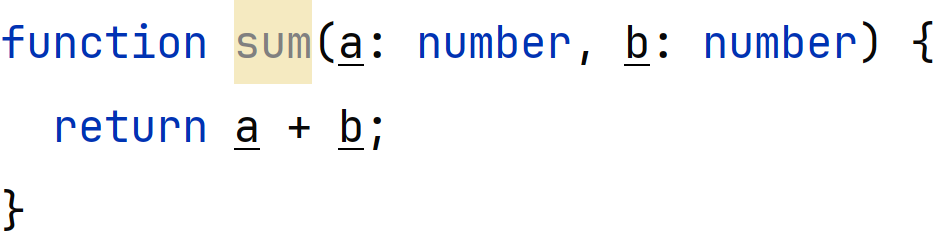
**Калькулятор**

Представим, что вам доверили разработать приложение калькулятор:



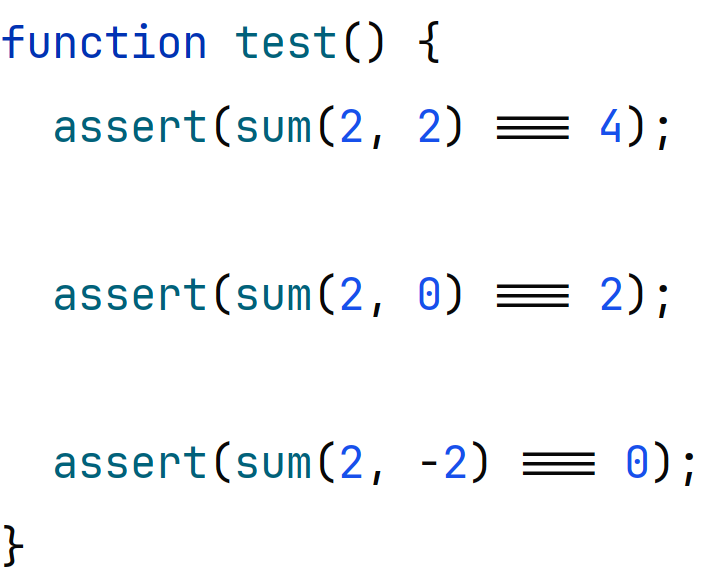
*Пример интерфейса приложения*

В ходе реализации его функций вам стало очевидно, что без тестов тут не обойтись. В поисках жертвы вы натыкаетесь на самую легкую добычу из всех возможных:

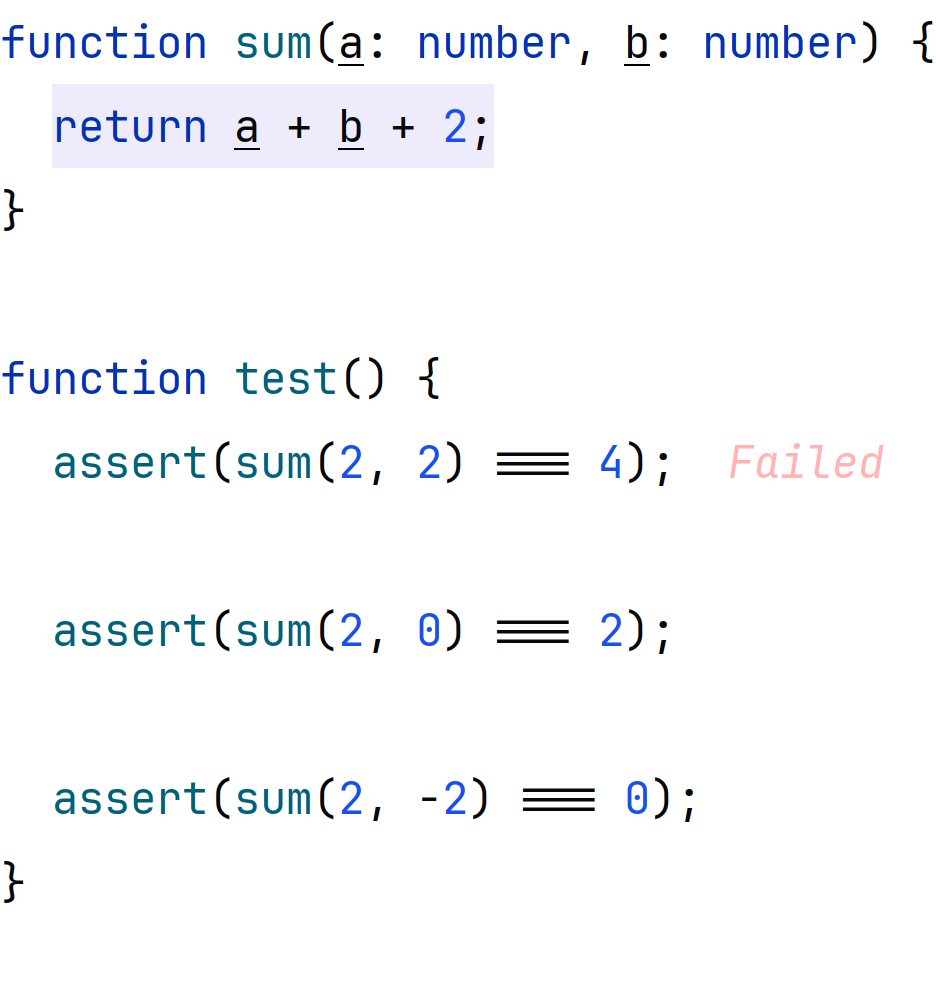


*Простая функция сложения двух чисел*

Настолько простая функция ожидаемо не вызывает больших трудностей при её тестировании:

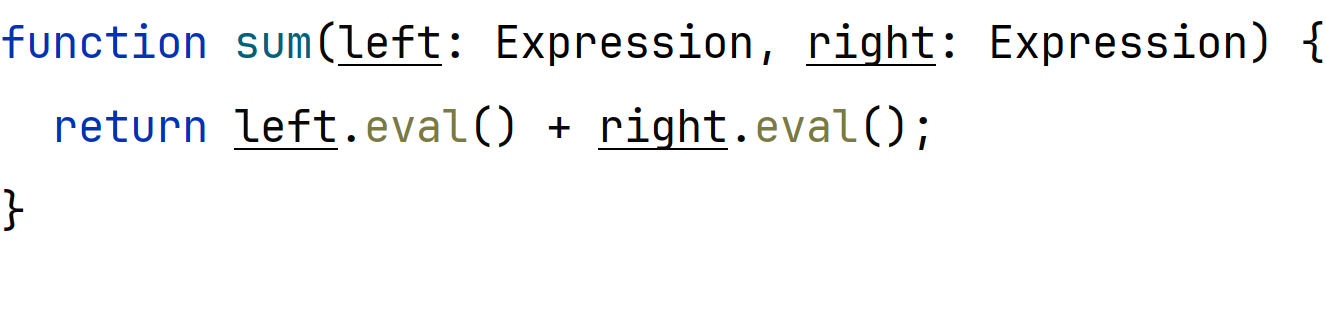


Для того чтобы убедиться что тест написан правильно необходимо проверить среагирует ли он на дефект в реализации:

 Как видно из результата, тест написан и даже падает в случае некорректной реализации. Но действительно ли это является достаточным показателем качества выбранного метода тестирования?

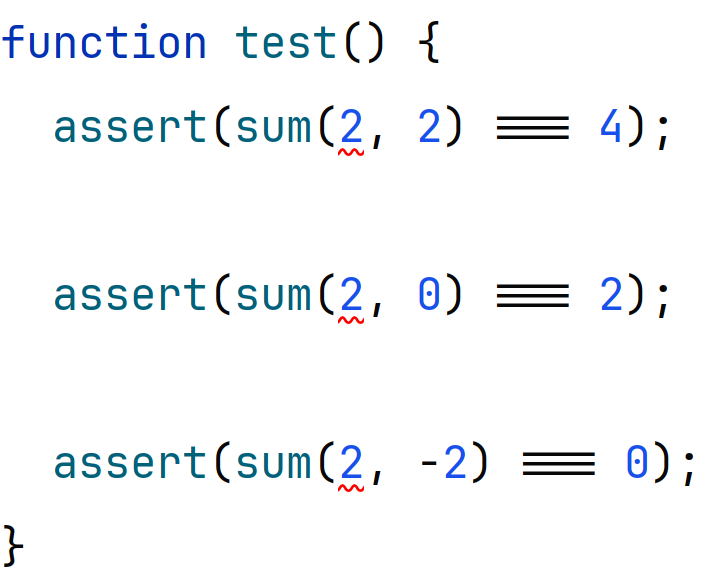
**Хрупкость**

Предположим, что в ходе рефакторинга исходного кода разработчиками было решено несколько изменить сигнатуру функции sum:



*Функция sum стала требовать больше условий от своих клиентов*

При компиляции и запуске основной программы с измененной функцией система ведет себя как прежде, числа все также складываются, то есть никакого регресса нет, но при запуске ранее написанных тестов мы получаем ошибку:

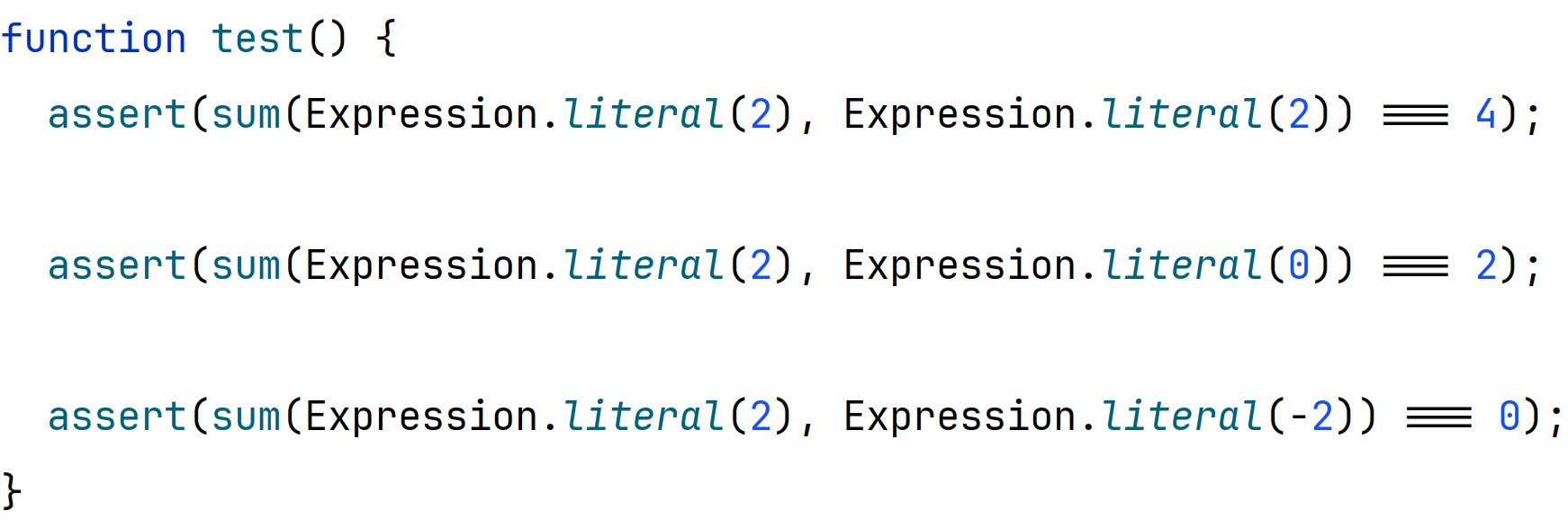


*Тип аргументов изменился, тесты упадут с ошибкой компиляции*

Тест упал, что обычно должно интерпретироваться как наличие регресса в системе, но по факту ничего не сломано, разработчики лишь немного изменили структуру кода определенной функции, то есть выполнили рефакторинг. Такой результат тестирования называется *ложноположительным*:

* Ложным, потому что не соответствует реальному положению вещей, а именно отсутствию каких-либо дефектов.
* Положительным, так как тест — это прежде всего поиск дефектов, как тест на грипп, например. Это можно считать простым соблюдением общей терминологии, хотя в падающем тесте для разработчика мало чего положительного.

Хрупкие тесты, а именно так их и называют, являются *единственным* признаком низкой сопротивляемости к рефакторингу. Теперь, для того чтобы тест снова заработал, требуется поправить его исходный код:



Работы над рефакторингом увеличились на ровном месте. Хрупкие тесты являются опасным гостем на проекте. В начале их воспринимают не всерьез, ну подумаешь, упал после рефакторинга. Но чуть позднее, когда проблемы становятся критическими, менять что-то становится уже поздно: ведь на написание тестов времени потрачено уже много, а удалять их жалко, ведь они действительно *иногда могут что-то поймать*.

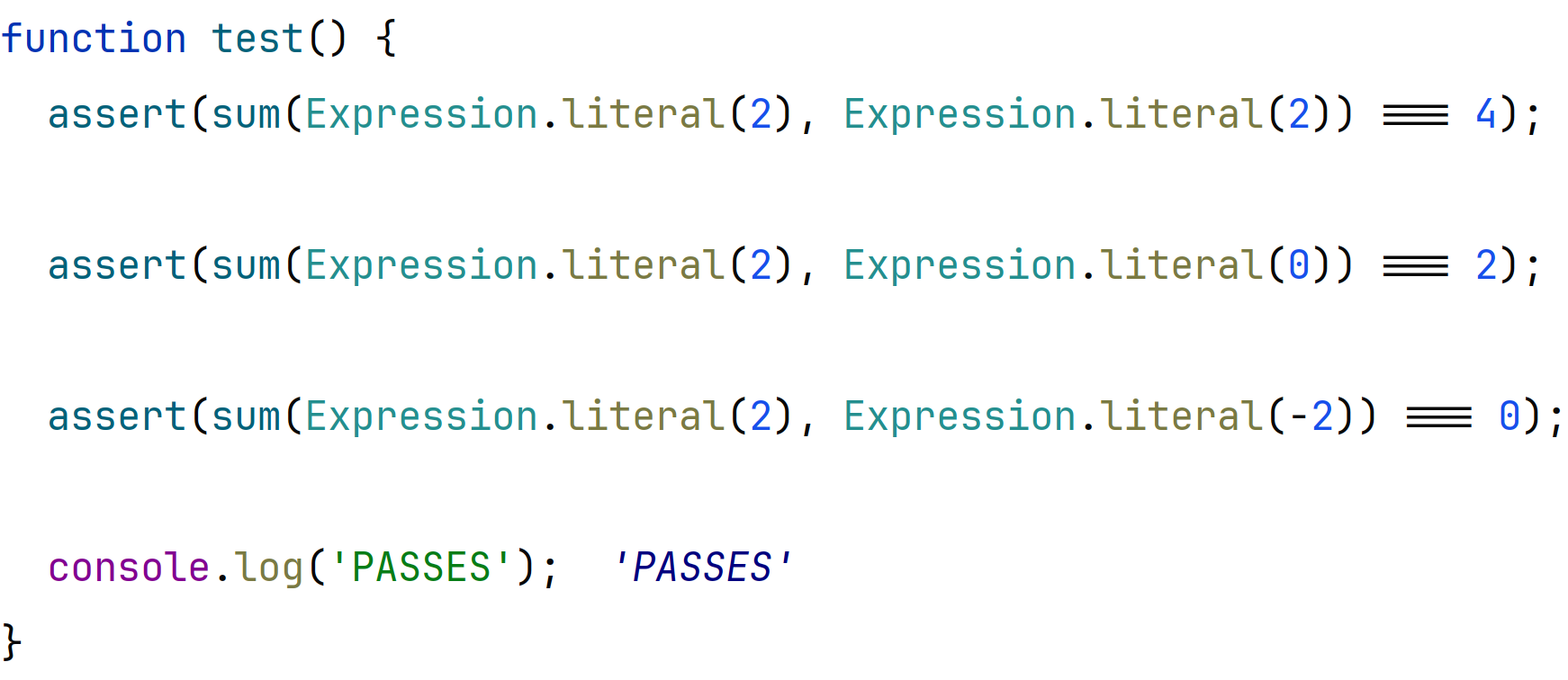
Данный балласт усложнит практически любой рефакторинг и несомненно возглавит список основных причин быстрого устаревания проекта. Хрупкие тесты и быстрый рост их влияния на проект можно сравнить с тем, как болото медленно, но уверенно тянет беспомощную жертву вниз, начиная с самого малого, но в большинстве случаев заканчивая начатое до конца.

**Ложь**

Отметим также один достаточно очевидный факт: в таком сложном и составном приложении как калькулятор, функция sum не будет использоваться сама по себе. Скорее всего она будет вызываться внутри других элементах системы, например в UI компонентах. Теперь представим, что после очередной правки дефектов, доработки или рефакторинга разработчик допустил ошибку в такой композиции:



Если запустить программу и сложить два разных числа, то можно увидеть неожиданный результат. На лицо явный дефект. Но в тоже самое время, если запустить тест, написанный ранее, то он пройдет успешно:



То, что тест прошел - не удивительно, ведь все что он проверяет, так это работу функции sum, а её реализация как раз осталась неизменной, изменилась лишь её композиция с другими элементами системы. Это одно из основных проявлений низкой защиты от регресса. Такой результат запуска тестов называется *ложноотрицательным*. По аналогии с предыдущим примером, ложным такой результат является потому, что проходящий тест понимается как признак отсутствия дефектов, хотя на самом деле система ведет себя некорректным образом.

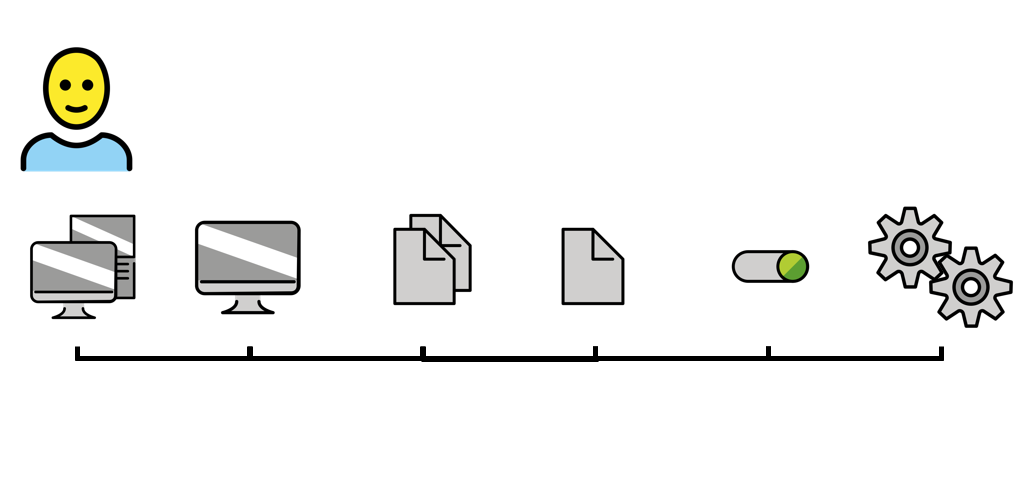
Конечно, здесь можно поспорить и сказать, что тест на самом деле выполнил свою задачу и выдал *истинно отрицательный* результат, ведь сама функция sum не сломалась. И это действительно так. Но проблема здесь заключается в том, что такая ситуация вообще есть, ведь она, целиком и полностью, является результатом зависимости теста от конкретных деталей реализации, тест *знает* о существовании функции sum и поэтому может протестировать её отдельно.

Как будет показано далее, в целях построения действительно полезных тестов, гораздо выгоднее рассматривать текущую ситуацию как негативную, то есть считать результат ложноотрицательным.

Тест соврал разработчику, один или пару раз на это могут закрыть глаза, но если такая ситуация станет постоянной, то результатам выполнения тестовых сценариев перестанут доверять, *их могут просто игнорировать*. Сам процесс запуска тестов станет бессмысленным и будет проходить скорее для галочки тем самым в пустую забирая и без того драгоценное время инженера.

**Основная причина**

Главная причина недостаточных показателей тестов в защите от регресса и сопротивляемости рефакторингу лежит в неверно расставленных границах тестирования:



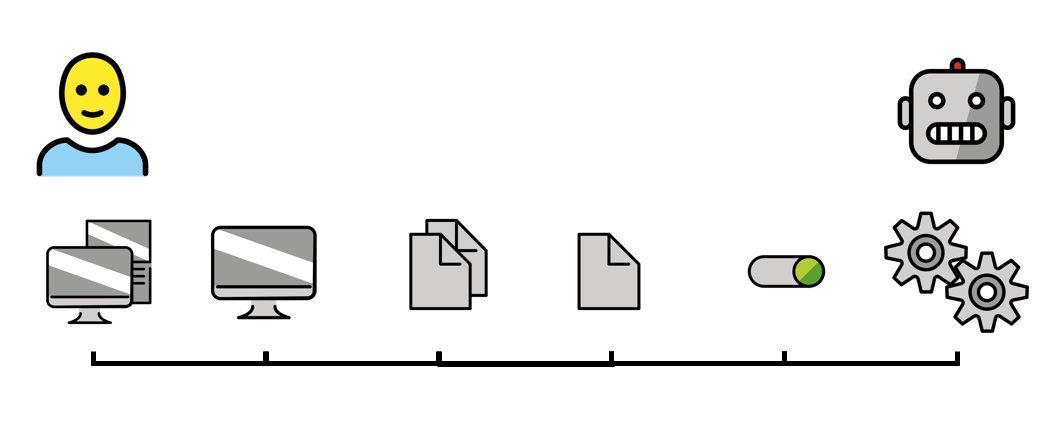
*Шкала обозначает расположение границы I/O*

На изображении выше обозначена условная шкала, на которой отмечены возможные границы взаимодействия с программой:

* Крайней правой точкой является такая граница, которой известны *полные* детали реализации программы (как настоящему разработчику).
* Далее, справа на лево, идут компоненты как композиция функций
* Страницы как композиция компонентов
* Процессы как композиция страниц
* Общая клиентская логика приложения как композиция процессов
* В крайней точке слева располагается граница, представляющая собой целевые средства ввода и вывода информации из программы, именно те порты, с помощью которых с ПО взаимодействует настоящий *пользователь системы*.

Важно отметит и то, что QA инженер, проверяющий требования *чаще всего* это делает именно от лица пользователя системы. Данный факт будет принят за данность в дальнейшем.

Почему именно расположение границы является виновником? Возьмем пример с калькулятором и отразим расположение границы реализованного метода тестирования на шкале:



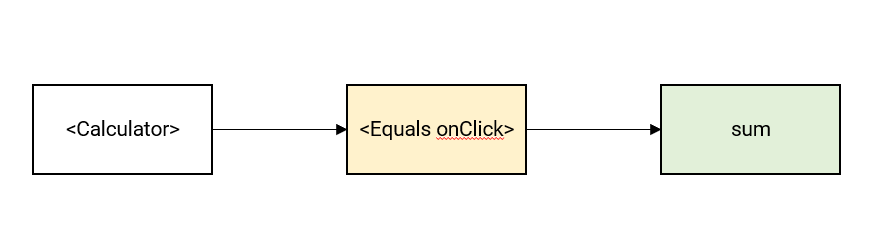
*Тесты изображены ввиде робота так как являются таким же пользователем, взаимодействующим с программой, только “автоматизированным”.*

Сам факт того, что тесты проверяют функцию sum в отрыве от тех компонентов, с которыми она используется в целевом виде и доказывает то, что граница тестирования располагается слишком близко к деталям реализации:



*Цепочка композиций элементов*

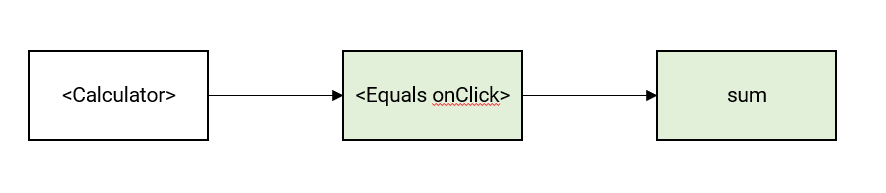
В нашем примере, разработчики в ходе правки дефекта или рефакторинга изменили исходный код одного из промежуточных компонентов данной цепочки:



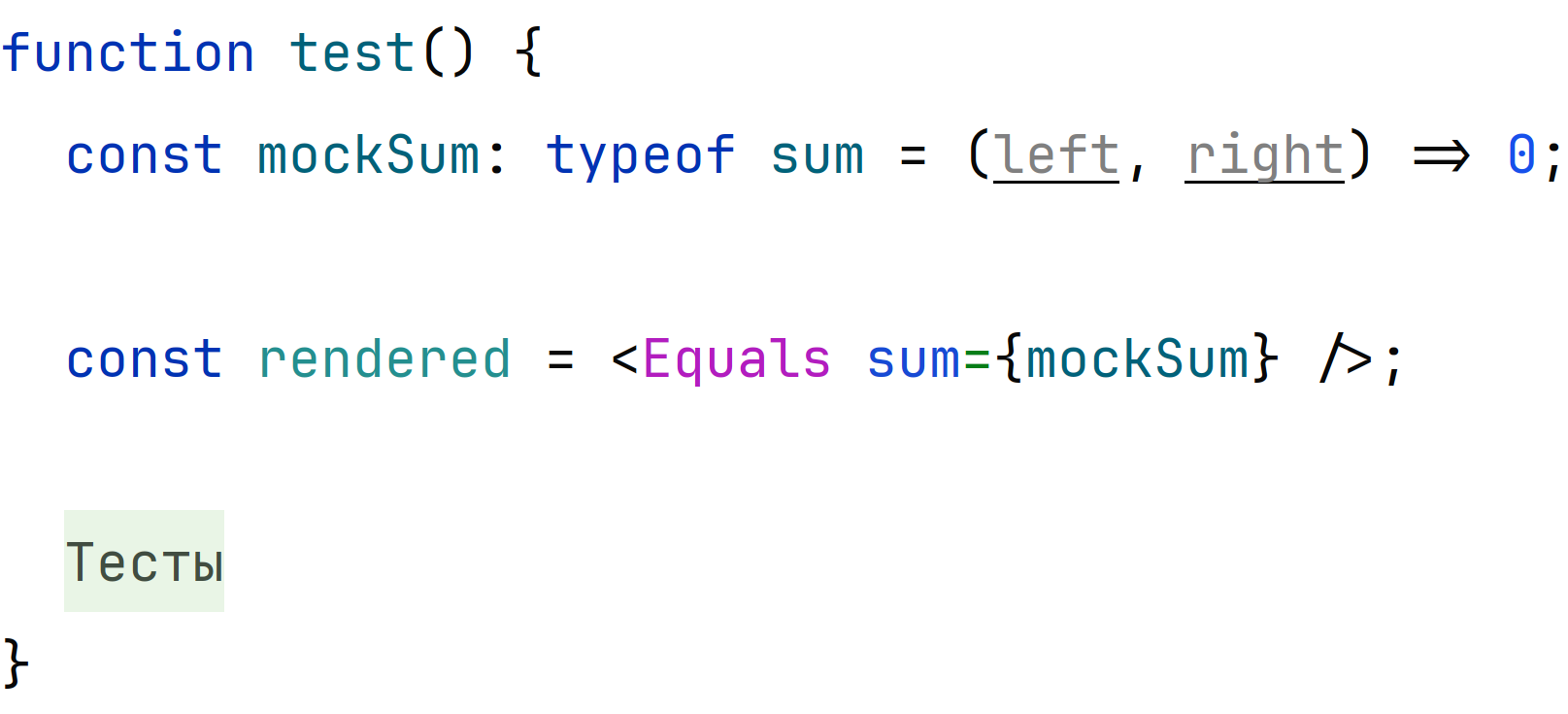
*Исходный код компонента Equals изменился*

В таком случае, выполняющийся тест функции sum будет доказывать работоспособность самой тестируемой функции, а не требования в целом, полностью игнорируя новое поведение измененного компонента Equals. Чего нельзя сказать про настоящих пользователей (QA), которые будут заводить дефекты основываясь на поведении всех компонентов в агрегации. Проблема заключается в том, что тесты в их текущем виде располагаются слишком далеко в своих границах от реальных клиентов системы.

Здесь можно возразить и сказать, что возможно написать отдельные тесты на компонент Equals:



Сначала предположим, что отдельный тест на компонент Equals изолирует его от функции sum таким образом, что проверяется корректность работы именно самого компонента:



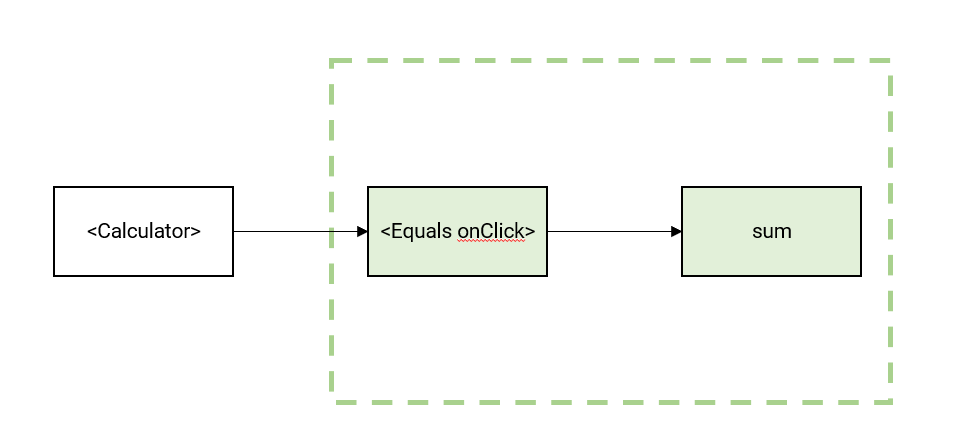
*sum при проверке Equals подменяется заглушкой так, что результат теста полностью не зависит от работоспособности настоящей функции суммирования*

Из этого можно вывести следующее:

* Часть регресса в компоненте Equals теперь предупреждается тестами. Ситуация уже лучше, чем была прежде.
* В случае падения теста источник дефекта будет легко определен, так как тесты проверяют сильно изолированный функционал.
* Сценарии тестов на sum и Equals будут пересекаться так как обязаны покрывать одни и те же общие требования и допущения. Это вызывает дублирование и негативно сказывается на поддерживаемости.
* Тесты верифицируют элементы в изоляции тем самым игнорируя проверки на *совместимости их контрактов.* Об этом будет рассказано в части про принцип подстановки.

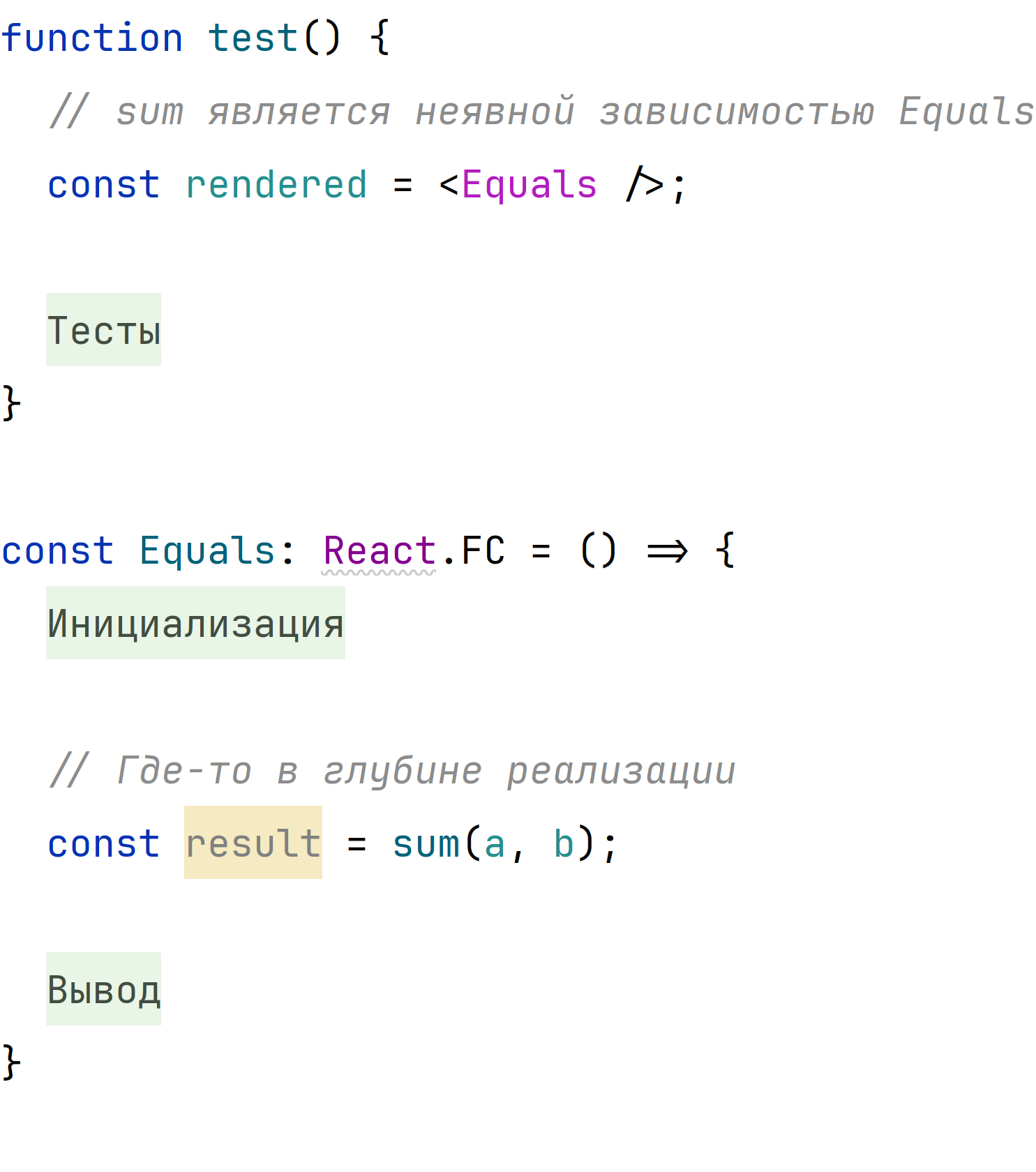
Ясно, что тестирование компонентов по отдельности, вне контекста их целевой композиции только частично улучшает показатели защиты от регресса, но за собой тянет еще более серьезные проблемы, приводящие к сниженной поддерживаемости таких решений. Сопротивляемость рефакторингу также не улучшилась так как тесты все также зависят от деталей реализации программы.

Теперь предположим, что тест на компонент Equals также включает функцию sum:



*Тесты на компонент Equals также включают в себя реальную функцию sum*

Исходный код самого теста при этом выглядит следующим образом:

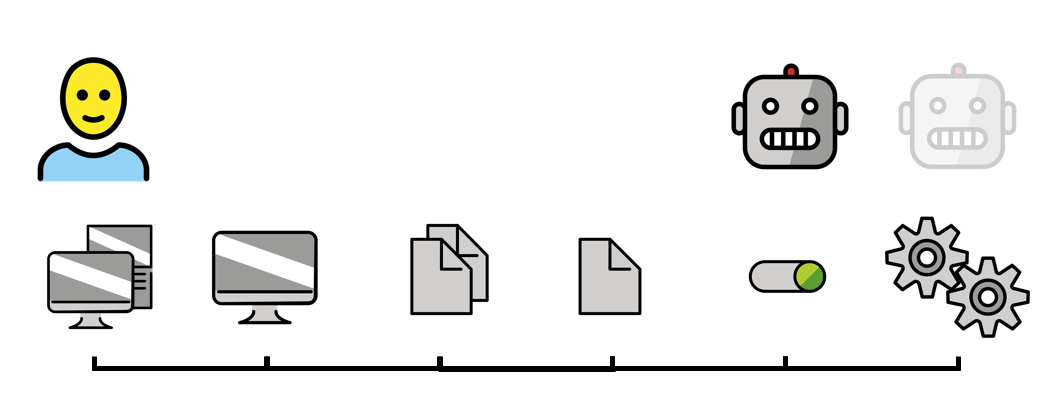


*В тестах на Equals используется настоящая функция суммирования*

Здесь выводы уже совершенно иные:

* Регресс теперь предупреждается как внутри компонента Equals, так и внутри функции sum.
* Элементы тестируется в композиции с друг другом, тем самым косвенно проверяется контракт их взаимодействия, что дает дополнительную защиту от возможных дефектов.
* В существовании отдельного теста на sum теперь нет смысла ведь данная функция транзитивно проверяется тестами на Equals.
* Тесты теперь не знают о существовании функции sum. Эту деталь реализации можно легко рефакторить и тесты при этом не будут сломаны.
* Кода в рамках тестирования выполняется больше, это негативно повлияет на время выполнения.
* В случае падения теста, ошибку будет найти сложнее так как покрывается бо́льшая площадь исходного кода.

Ситуация уже гораздо лучше. Тестирование более крупного блока выразилось в улучшенных показателях защиты от регресса, сопротивляемости рефакторингу и поддерживаемости. Основная причина заключается в том, что граница тестирования теперь располагается чуть ближе к реальному пользователю системы:



Давайте разберем причины такого положительного эффекта от движения границы в сторону пользователя более подробно, несмотря на то что он скорее всего интуитивно понятен.

**Защита от регресса**

Защита от регресса повысилась вследствие смещения границы тестирования на шкале влево, так как тестовые сценарии теперь верифицируют поведения более приближенные к тому, что видит перед собой пользователь.

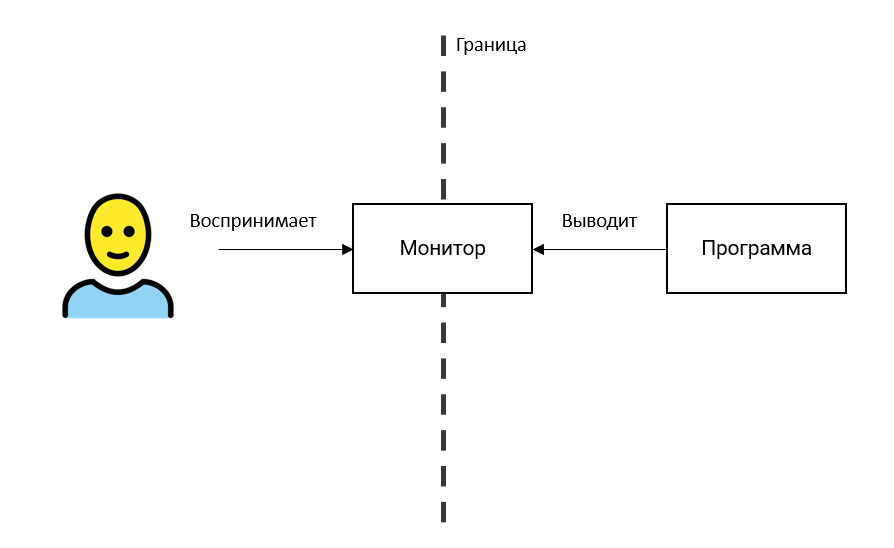
Клиенты системы (QA инженеры, в частности) и есть те самые источники дефектов и требований, те самые лица, которые решают, что является корректным поведением, а что нет. Обратите внимание, как это похоже на то, чем должен заниматься тест – определять какое поведение является дефектным, а какое нет.

Уменьшение расстояния между границами пользователей и тестов как бы делает последних более похожими на реальных клиентов системы, тем самым положительно влияя на вероятность предупреждения дефекта.

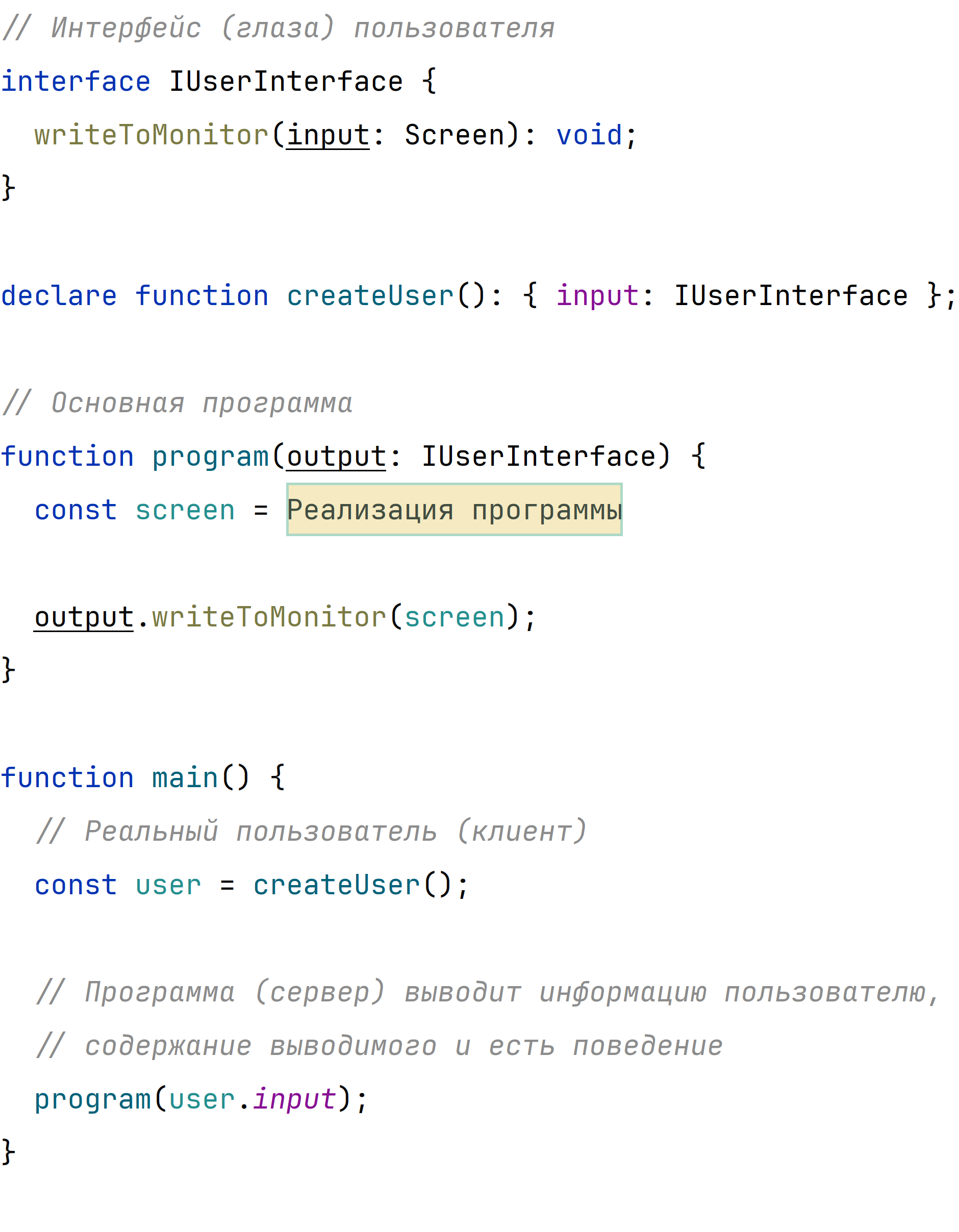
**Сопротивляемость рефакторингу**

Сопротивляемость к рефакторингу тоже увеличилась, но здесь ситуация немного сложнее. Рефакторинг, по определению, это процесс изменения деталей реализации без влияния на *наблюдаемое поведение.* Термин наблюдаемое поведение, можно сделать более конкретным с точки зрения рассмотренных ранее терминов:

* Наблюдаемое – это всего лишь место, где располагается конечный порт вывода информации. Это может быть монитор компьютера, API библиотеки, дисплей телефона, экран телевизора, аудио система, вариантов много и все они зависят от того, кем является клиент системы.
* Поведение – это содержание выводимой через порт информации



Попробуем представить то же самое, но на уровне псевдокода:



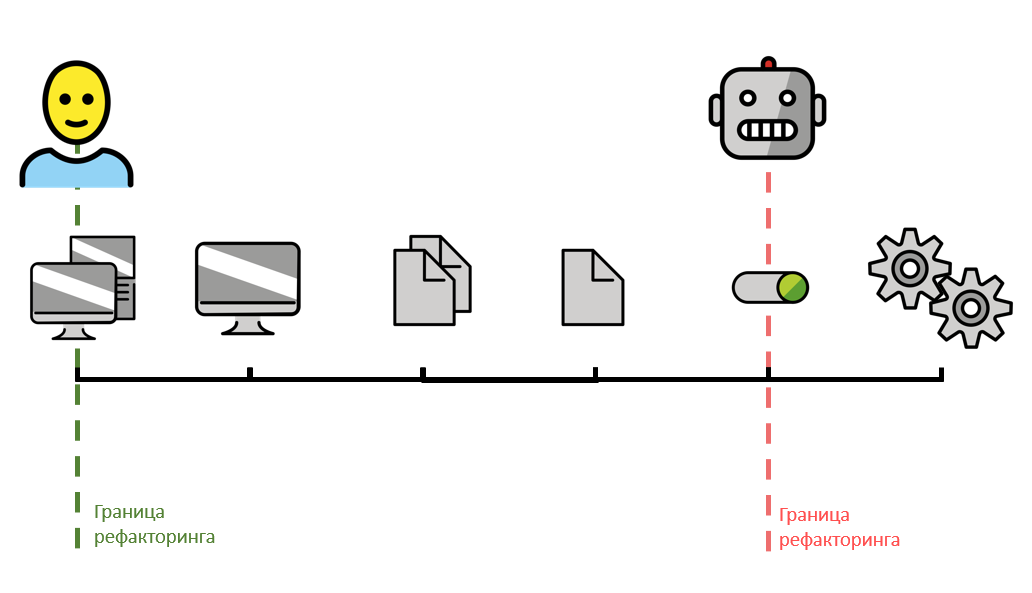
Можно сделать следующие замечания:

* *Клиент полностью определяет интерфейсы для своих серверов*. Это то самое допущение, сделанное в прошлой части. Обратите внимание как оно точно отражает реальное положение вещей. Ведь именно конечные пользователи диктуют разработчикам те средства вывода, которые должны реализовываться системой.
* Любое изменение, которое не влияет на контракт взаимодействия (интерфейс IUserInterface) и содержание выводимой через него информации будет скрытым от пользователя, то есть являться рефакторингом другими словами! Обратите внимание как здесь проявляются свойства, рассмотренные ранее, а именно как интерфейсы защищают своих клиентов от изменений в имплементаторах.

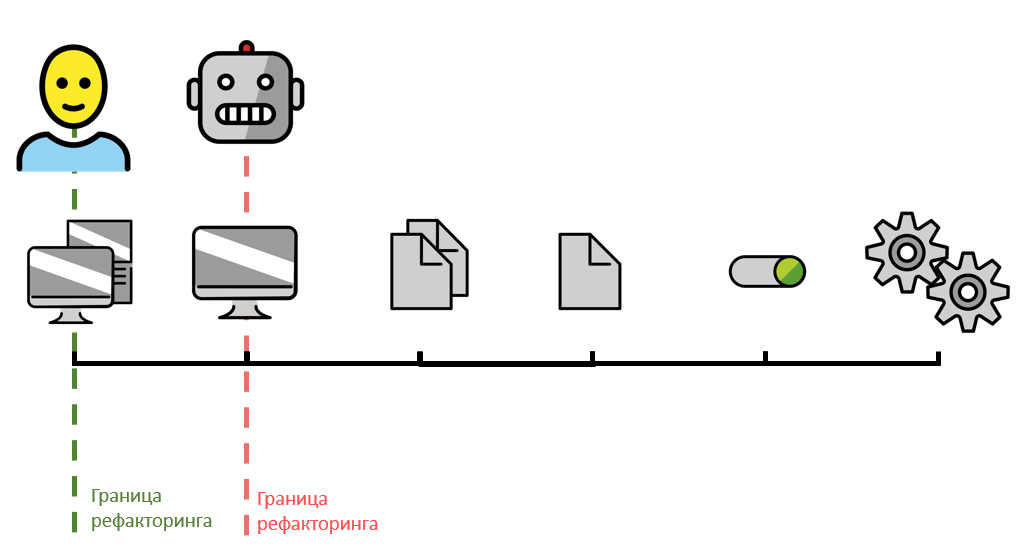
Из этой достаточно долгой, но необходимой подводки можно сделать уверенный вывод: *именно конечный пользователь системы определяет, что является рефакторингом, а что нет*.

Тесты, это такой же пользователь системы, который также рисует свои границы (или интерфейсы взаимодействия с системой). Из них следует что является рефакторингом, а что нет.

Для хрупких тестов, граница располагается очень близко к деталям реализации, сильно уменьшая пространство для рефакторинга как следствие:



Для тестов, предоставляющих достаточную сопротивляемость рефакторингу, граница максимально приближена к той, которая определяется пользователем, тем самым понятия рефакторинга обоих становятся *почти идентичными*:



*Границы наблюдаемого обоих клиентов почти совпадают, тем самым определяя почти равные возможности по рефакторингу.*

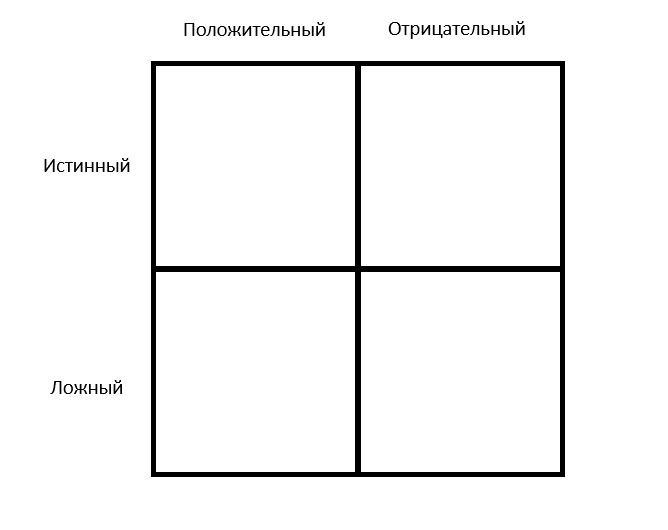
Подведем промежуточный итоги:

* Пользователь системы определяет то, что является наблюдаемым в поведении системы, он определяет её крайнюю границу, тот самый порт вывода информации.
* Тесты, это такой же пользователь, который в свою очередь устанавливает свои собственные границы.
* Защита от регресса зависит от позиции границы, если она располагается близко к деталям реализации, то итоговые тесты верифицируют крупный сценарий “по кусочкам” (по определению), игнорируя целевую композицию и пропуская целые цепочки функций.
* Граница также влияет и на сопротивляемость к рефакторингу, более того, именно позиция границы, определенная настоящим пользователем, и определяет то, что является рефакторингом, а что нет. Чем дальше граница тестирования располагается от границы основного пользователя, тем чаще тесты будут падать при изменении деталей реализации, в то время как настоящий клиент системы ничего не заметит (по определению рефакторинга).

**Матрица**

После того как был определен основной инструмент контроля защиты от регресса и сопротивляемости рефакторингу, необходимо понять, а каким образом можно измерить данные свойства для конкретного метода тестирования.

Для этого воспользуемся следующей таблицей:



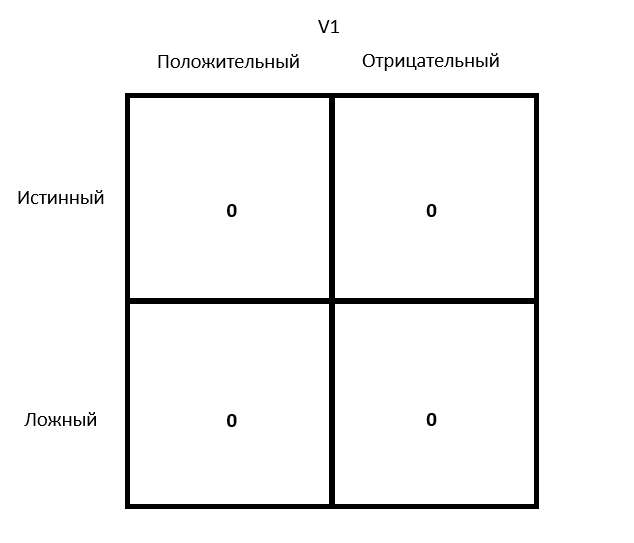
*Матрица эффективности тестирования*

Разберем матрицу расписывая ячейки слева на право:

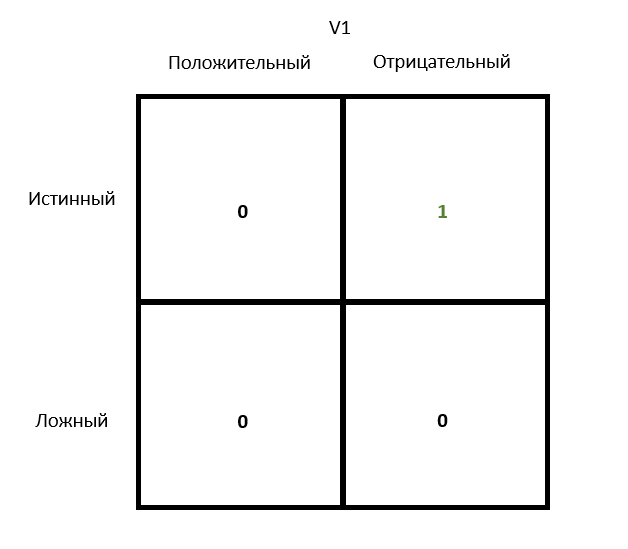
1. Истинно положительный – результат, при котором в реальности имеет место дефект и тесты на него среагировали, то есть упали. Иными словами, *предупредили регресс*.
2. Истинно отрицательный – результат, при котором в реальности дефектов нет (в рамках подразумеваемого разработчиком поведения) и тесты при этом проходят успешно, то есть не падают.
3. Ложноположительный – результат, при котором в реальности дефект отсутствует, но тесты при этом упали.
4. Ложноотрицательный – результат, при котором в реальности дефект присутствует, но тесты при этом проходят успешно, сигнализируя обратное.

В каждой из ячеек располагается число, обозначающее количество соответствующих событий, которые произошли во время эксплуатации выбранного тестирования.

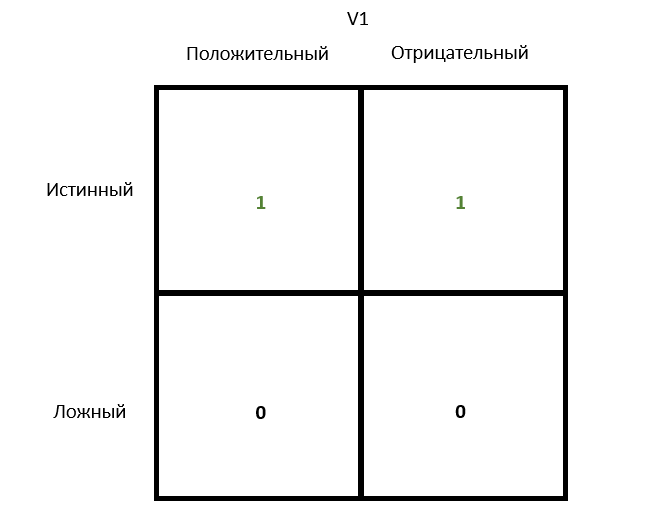
Например, представим, что разработчики реализовали на проекте некоторый метод тестирования, назовем его **v1**. В самом начале эксплуатации в каждой ячейке будет стоять 0.



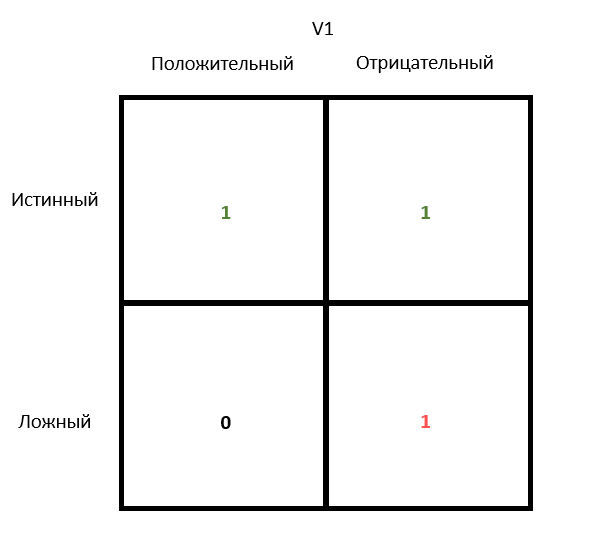
Далее, разработчики реализовали новое требование (изменили исходный код), убедились, что тесты проходят и отдали на проверку QA, которые подтвердили, что все работает исправно. Такое событие соответствует истинно отрицательной ячейке, поэтому её нужно дать одно очко:



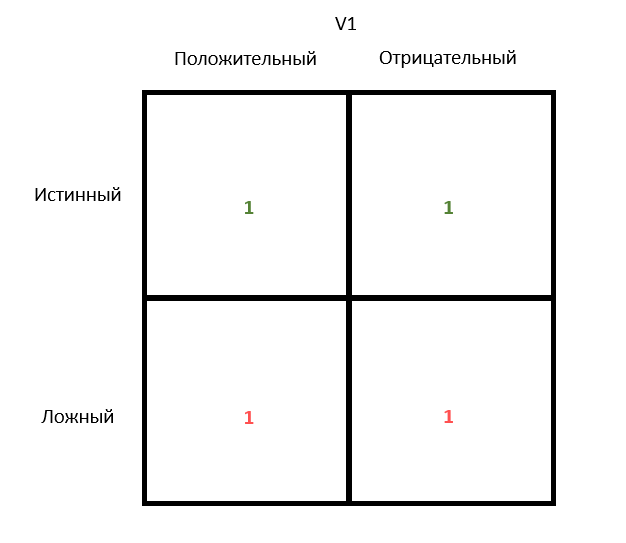
Через некоторое время, разработчики затеяли рефакторинг, изменили исходный код и тесты в результате упали. После фактической проверки оказалось, что дефект действительно есть. Данный результат является истинно положительным, в следствии чего и увеличивается значение соответствующей ячейки:



Далее, разработчики реализовали очередную требуемую заказчиком функцию (изменили исходный код), убедились, что тесты проходят и отдали QA, которые уже по итогу обнаружили дефекты в ранее работающем функционале, иными словами, поймали регресс. Тесты с новыми функциями прошли, хотя фактически имел место дефект, данный результат является ложноотрицательным:

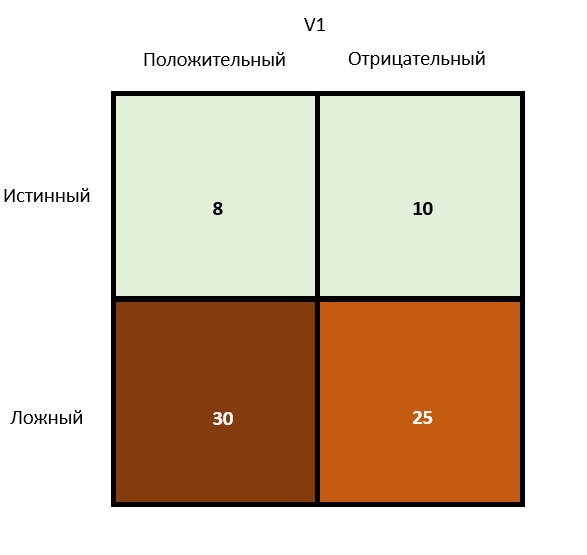


Затем, спустя некоторое время инженеры затеяли очередной рефакторинг. После изменения деталей реализации некоторой функции тесты сообщили о наличии проблемы, то есть упали. При фактической проверке разработчиком оказалось, что никакого дефекта в наблюдаемом поведении нет. Такой результат является ложноположительным:

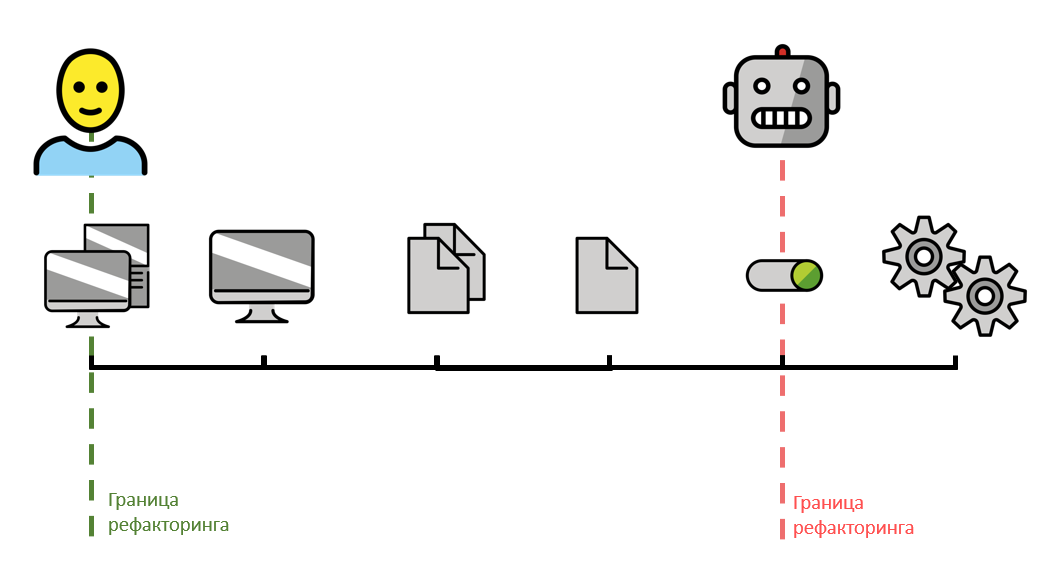


Обратите внимание на крайне важный факт, что данная матрица наполняется не в статике, а в динамике, то есть после каждого осмысленного изменения исходного кода. Это может быть реализация новой функции, правка дефекта или рефакторинг. Иными словами, каждое очко в таблице соответствует какой-то задаче.

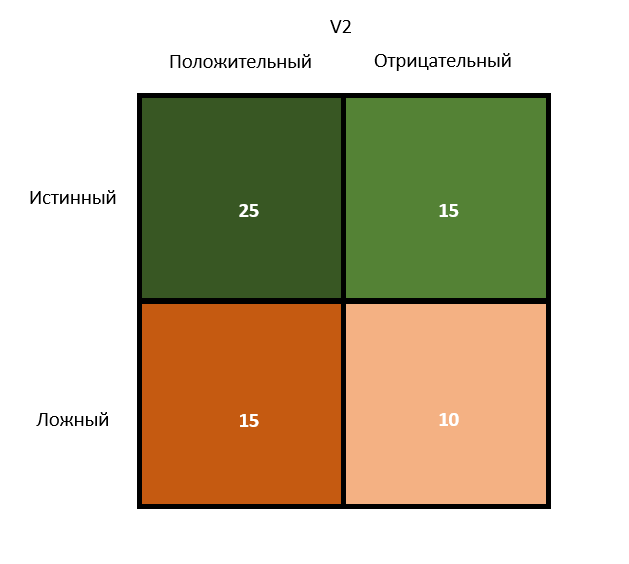
Теперь представим, что спустя некоторое время, у разработчиков получилась матрица следующего содержания (для наглядности ячейки обогащены цветовой индикацией, чем темнее цвет, тем относительно большее число стоит в ячейке):



Нижний ряд является более темным чем верхний. Это говорит о том, что выбранный метод тестирования является хрупким и не предоставляет достаточной защиты от регресса. Как говорилось ранее, это скорее всего связанно с неверно выставленной границей тестирования: она располагается слишком близко к деталям реализации:

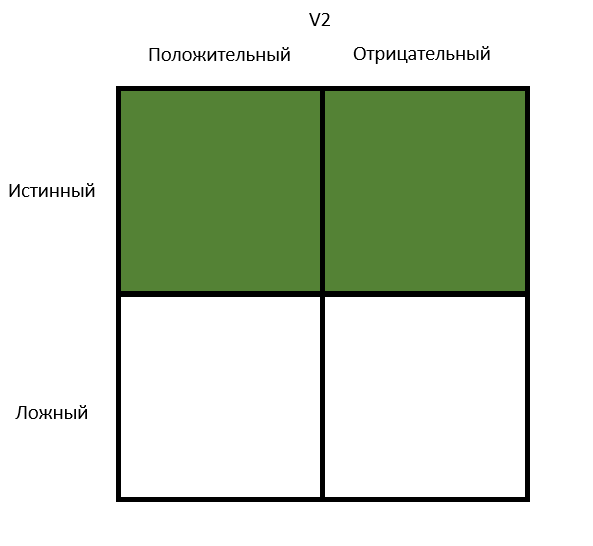


Теперь представим, что разработчики сдвинули границу тестирования, переместив её ближе к пользователю. Например, сменили тестирование простых функций тестированием UI компонентов как примере с калькулятором. Анализ показал следующее:



Ситуация уже лучше, тесты пропускают меньше дефектов (истинно положительный и ложно отрицательный), но по-прежнему реагируют на рефакторинг (ложно положительный).

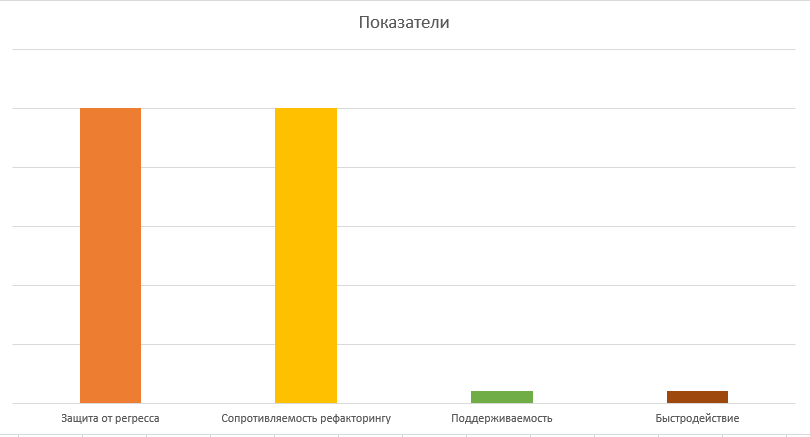
Логика процесса должна быть предельно ясной. Основной параметр оптимизации — это чаще всего граница тестирования. Целевая позиция матрицы, к которой должен стремится любой метод тестирования для того, чтобы быть эффективным выглядит следующим образом:



Подавляющее число событий, связанных с изменением исходного кода и последующей верификации его корректности, должны относится к первой строчке матрицы. Чем ближе актуальная матрица к идеальной, тем выше показатели защиты от регресса и сопротивляемости рефакторингу.

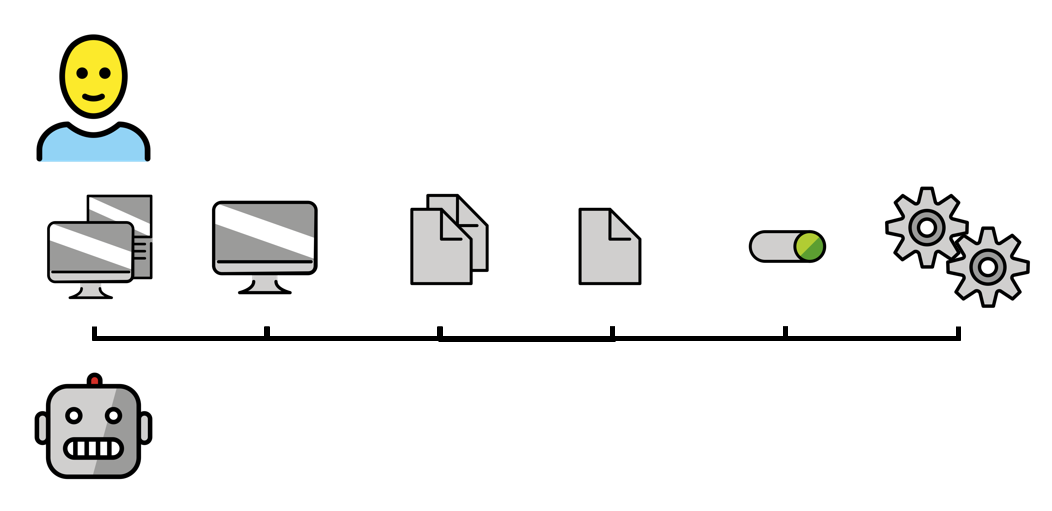
**Есть одно но**

Хорошая защита от регресса и сопротивляемость рефакторингу требуют приближения тестов к реальному пользователю. Крайней точкой данного процесса являются E2E тесты. Вспомним их показатели:



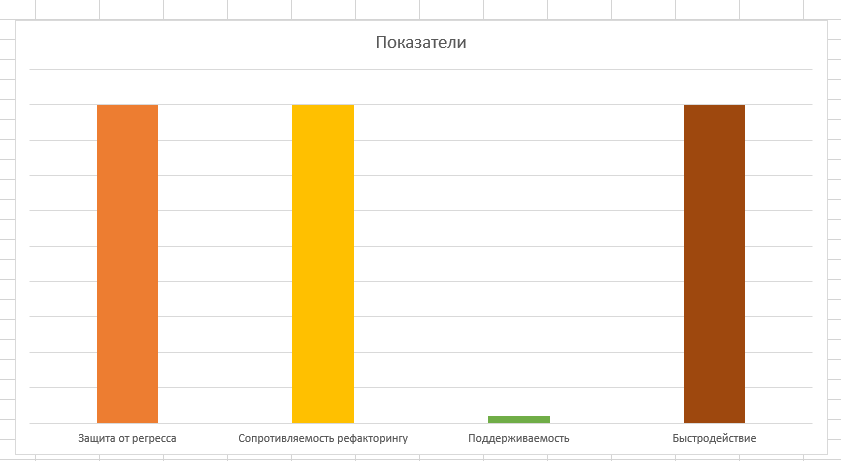
*Показатели свойств E2E тестов, взамен на достойные показатели в первых двух пунктах, они обладают сниженной поддерживаемостью и низкой скоростью выполнения*

У приведённой матрицы есть один большой минус: она не учитывает поддерживаемость и быстродействие итогового решения. Опираясь на приведенный анализ E2E тестов, можно сказать, что слишком близкое расположение тестов к пользователю или их фактическое “уравнивание”, может крайне негативно сказаться на поддерживаемости:



*E2E тесты используют все компоненты системы, как настоящий пользователь*

Быстродействие также не исключение, чем больше компонентов системы проверяется в агрегации (чем дальше граница тестирования), тем больше времени занимает выполнение тестов. На самом деле, защита от регресса вместе с сопротивляемостью рефакторингу являются *антагонистами* по отношению к быстродействию. Другими словами, нельзя написать такой метод тестирования, в котором все три показателя будут максимальными:

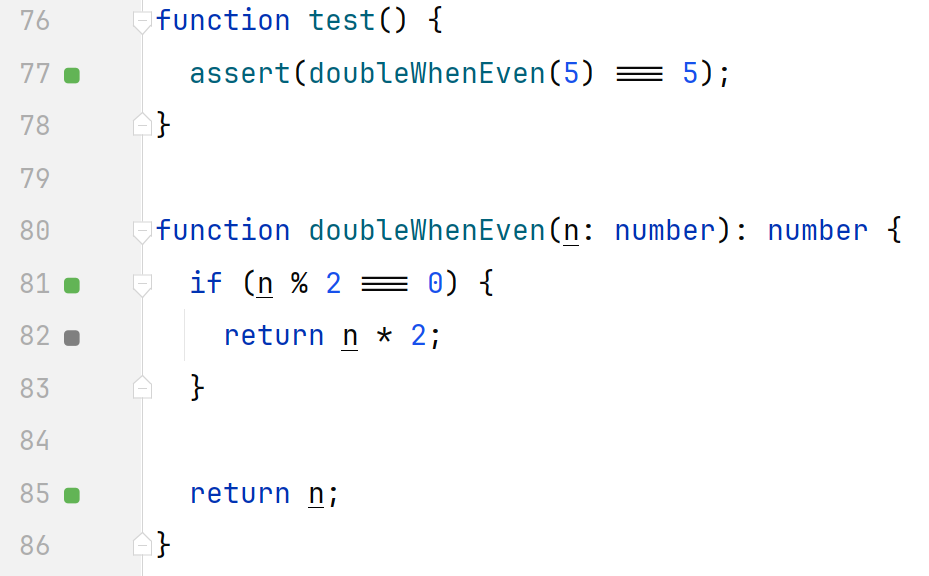


Напомним, что показатель быстродействия является *относительным.* Это значит, что если разработчик сумел написать такое решение для тестов, которое предоставляет отличную защиту от регресса и сопротивляемость рефакторингу, а также выполняется быстро, то в таком случае, у этого же самого разработчика получится ускорить эти же тесты *еще сильнее* просто исключив некоторые компоненты из процесса тестирования, тем самым делая изначальное решение *сравнительно слабым* в быстродействии.

Поддерживаемость также вступает в противоречие с первыми двумя свойствами, но только в самых крайних точках границы. Субъективность данного пункта делает его по большей части управляемым безотносительно других показателей.

**Покрытие кода**

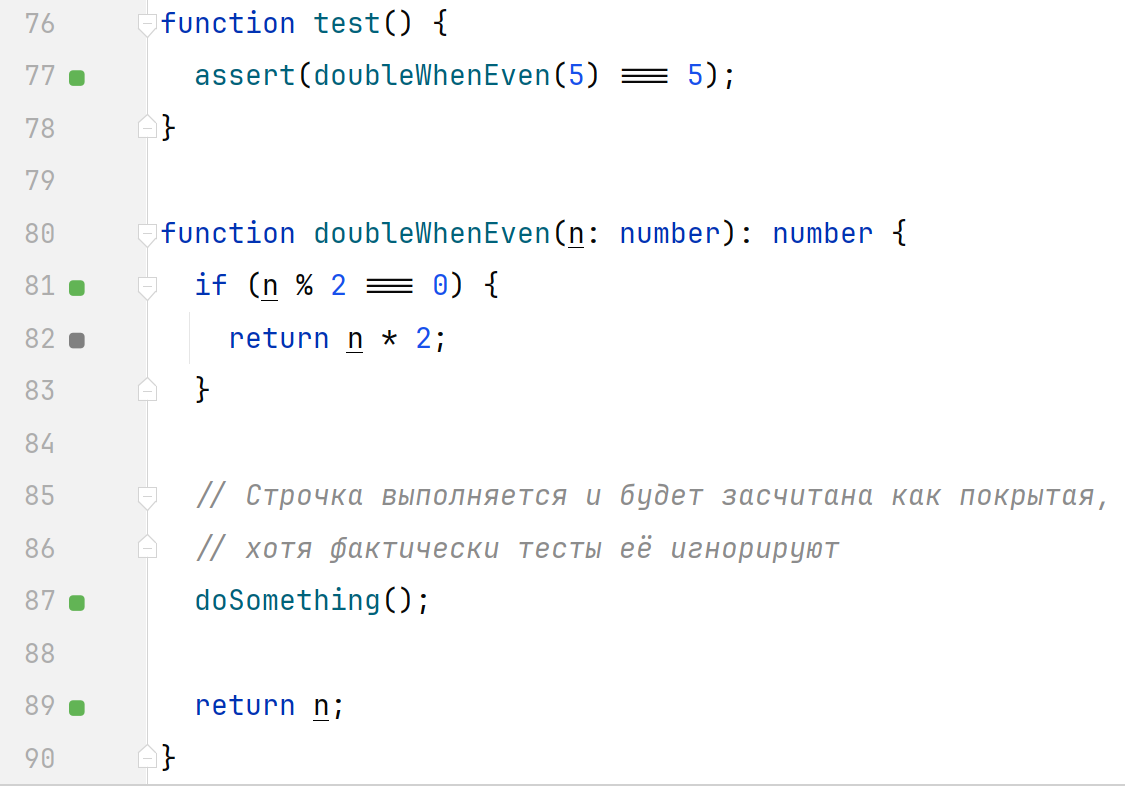
Отдельно стоит упомянуть такой метод измерения защиты от регресса как генерация отчетов о покрытии исходного кода тестами. Работает данный инструмент достаточно примитивным образом, в случае если какая-та строчка кода выполняется во время работы теста, то она помечается как покрытая. Все те участки кода, которые не задействуются во время выполнения, отмечаются как не покрытие:



*Зеленый индикатор (напротив нумерации строк) означает покрытый участок кода. Серый, напротив, не покрытый.*

Данный вид отчета является полезным *негативным* индикатором. То есть в случае, когда итоговый процент покрытия слишком низкий, это действительно может означать и скорее всего означает, что тесты предоставляют низкую защиту от регресса.

Показатель же покрытия в 100% или близкий к нему в свою очередь не гарантирует абсолютно ничего, ведь выполнение какой-то строчки кода не значит проверку корректности результата её работы:



Также инструменты сбора отчета могут, и скорее всего будут, игнорировать покрытие внутренних функций сторонних библиотек (в том числе и стандартной библиотеки языка), которые также стоит учитывать в процессе тестирования. Поэтому к подобного рода инструментам следует относительно с предельной осторожностью и внимательностью.

**Итоговая роль границы**

После рассмотрения того, каким образом граница тестирования влияет на их свойства, стоит передохнуть и подвести небольшой промежуточный итог:

* Основным инструментом измерения защиты от регресса и сопротивляемости рефакторингу является формирование матрицы результатов тестирования. Важно то, что конкретный метод тестирования сложно оценить в статике, напротив, более точные результаты оценки могут быть получены только в движении: в процессе выполнения задач, правки дефектов и рефакторинге.
* Основным методом управления данными показателями является расположение границы тестирования.
* Стремясь к идеальным показателям, не стоит забывать и о двух других: поддерживаемости и быстродействия.
* Поддерживаемость хоть и является по большей части субъективным, но может серьезно снижаться в крайних расположениях границ тестирования, например при использовании E2E решений.
* Быстродействие в свою очередь является прямым антагонистом защиты от регресса. Так, чем больше кода проверяется (то есть выполняется), тем выше защита от регресса, но и тем медленнее тесты как следствие.
* Процент покрытия тестов является хорошим негативным показателем. Однако, высокий процент в отчете в свою очередь не показывает абсолютно ничего по отношению к приведенным свойствам.

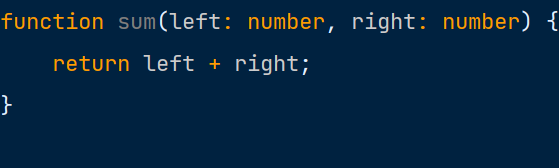
Разработчик, реализуя определенный метод тестирования, должен решить нетривиальную задачу, которая требует постоянно анализа (путем построения и заполнения матрицы) и доработки тестов таким образом, чтобы помимо достижения целевых показателей в защите от регресса и сопротивляемости рефакторингу, также оставить поддерживаемость и быстродействие на достаточно высоком уровне. Другими словами, идеального решения не бывает, а самое лучшее что можно получить, лежит где-то по середине, как оно чаще всего и бывает.

В следующей части мы затронем достаточно важную проблематику, возникающую при тестировании с заглушками, а именно проблему подстановки (описываемую принципом подстановки LSP). И конечно же продолжим своё движение к той самой *тестируемой архитектуре*.

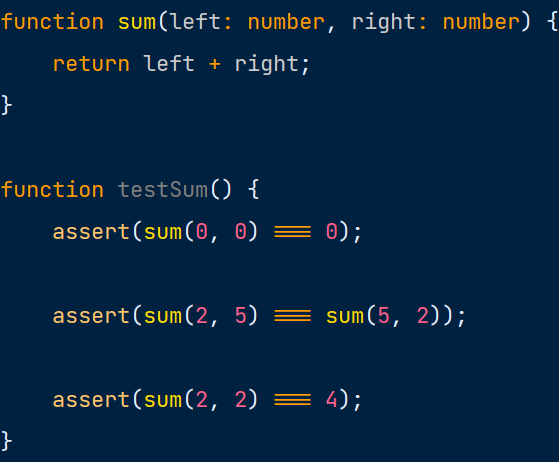
[**Тестируемая**](#содержание-компонентов) **архитектура. Часть 4: принцип подстановки**

Как изображение на мониторе зависит от совокупности логики приложения и действий пользователя (ввода данных), так и ответ от сервера будет зависеть от команд, исполняемых приложением (со стороны тестируемой программы).

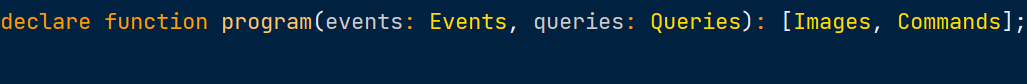
Предположим что стоит задача написать тест на функцию, складывающую два числа

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_18.png)

На таком простом примере очевидно, проверять имеет смысл только возвращаемое значение функции, так как именно оно и будет "наблюдаться" её клиентом. В то же самое время, верификация её аргументов, то есть входных данных, теряет всякий смысл и, напротив, приведет только к большей зависимости тестов от внутренней структуры функции (деталей её реализации).

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_19.png)

Программа в таком случае не является исключением, её можно представить в таком виде:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_20.png)

Сам процесс тестирования будет не сильно отличаться от аналогичного с функцией sum (сами аргументы естественно будут иметь более сложную структуру):

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_21.png)

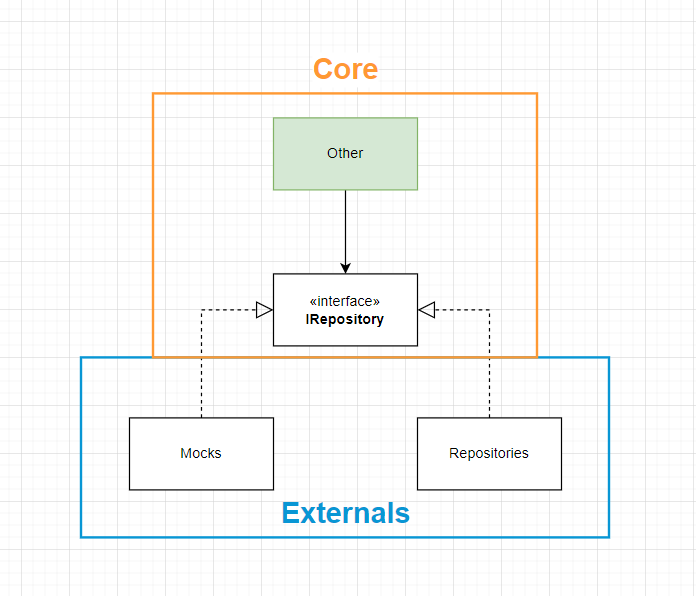
Пример выше является несколько идеализированным с одной единственной целью - показать связь между характеристикой функции (как запроса или команды) и её ролью в процессе тестирования.

[**Проверка запросов**](#проверка-запросов)

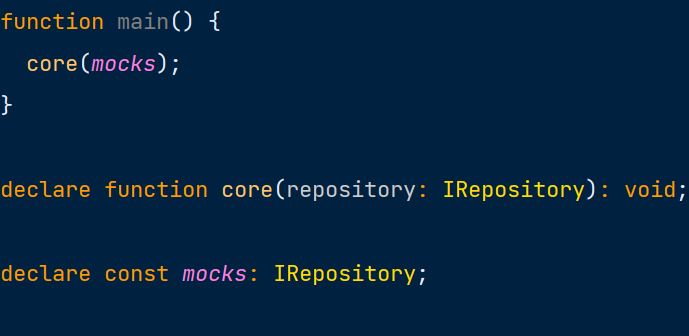
Если говорить откровенно, то всегда игнорировать проверку корректности запросов - не верно. Чтобы наглядно это продемонстрировать, рассмотрим принцип подстановки.

[**Принцип подстановки**](#принцип-подстановки)

Взглянем на схему еще раз:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_10.png)

Обратим внимание на то, что в процессе тестирования, в качестве реализации интерфейса IRepository будет *подставлен* блок Mocks:

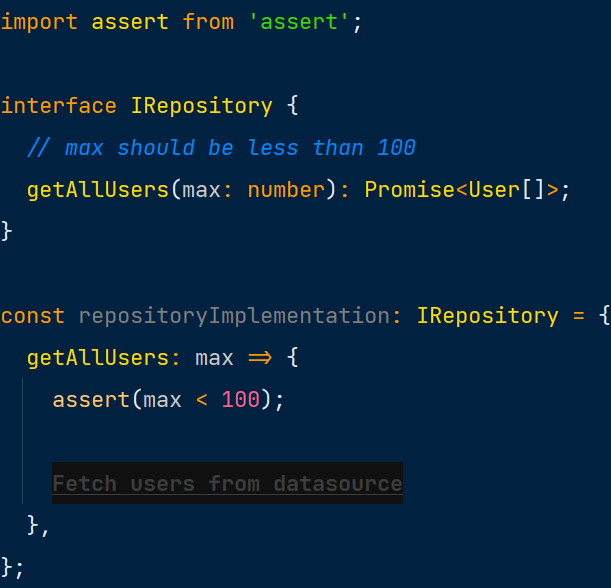
[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_23.png)

[**Compile time**](https://github.com/rkhaimov/brain-gym/blob/t_arch/plan.md#compile-time)

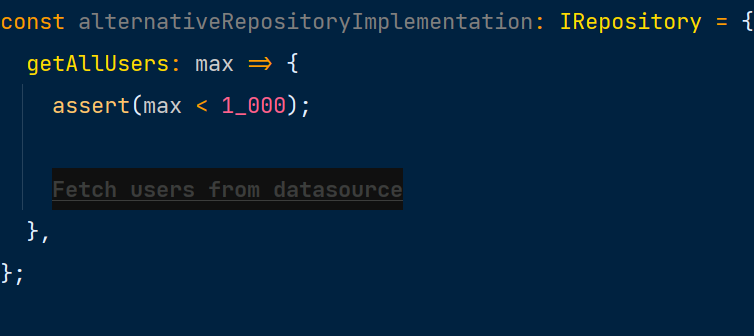
Для успешного прохождения статического анализа и последующей компиляции, подставляемый mocks (далее называемый *подтипом*) должен удовлетворять типу IRepository. Всю работу на верификацию данного условия берет на себя компилятор TypeScript, но, к сожалению, этого не всегда достаточно.

[**Preconditions**](https://github.com/rkhaimov/brain-gym/blob/t_arch/plan.md#preconditions)

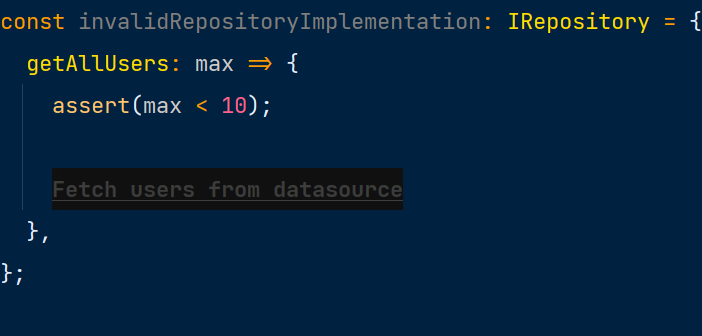
Помимо самого типа, IRepository также может подразумевать под собой некоторые дополнительные условия. Возьмем к примеру функцию getAllUsers.

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_26.png)

Как видно из примера выше, интерфейс устанавливает дополнительное условие на входящие аргументы (также известные как pre-conditions). Любой имплементатор IRepository, имеет право предъявлять условия *не строже*указанных в интерфейсе pre-conditions:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_24.png)

В случае более строгих требований, репозиторий не будет считаться подтипом интерфейса хоть и будет считаться валидным с точки зрения типов:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_25.png)

Почему это так? Дело в том, что если попытаться подставить репозиторий, ожидающий параметр max в пределах 10, клиентам, передающим в качестве аргумента 50 (что не противоречит контракту устанавливающему максимальную планку max в 100) - программа перейдет в некорректное состояние, либо вовсе завершится с ошибкой (что даже лучше).

Тестовые заглушки здесь не исключение, они также должны удовлетворять всем условиям на аргументы. Разница заключается в том, что в тестовом окружении preconditions и так проверяются, через снимки взаимодействий клиента (или тестируемой сущности) с подставленными mocks.

[**Post-conditions**](https://github.com/rkhaimov/brain-gym/blob/t_arch/plan.md#post-conditions)

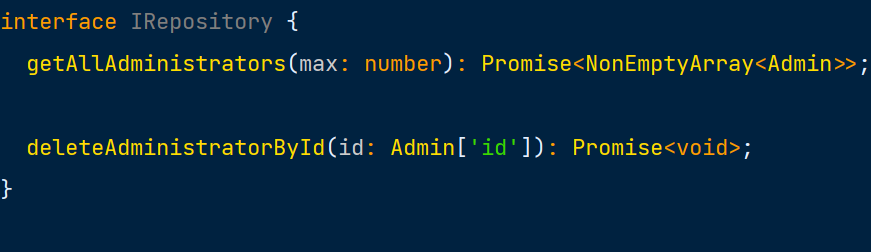
Рассмотрим функцию deleteUserById:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_27.png)

Функция deleteUserById содержит в себе как пост условие (post-condition) следующее утверждение - "После успешного удаления, функция getUserById должна вернуть null для данного пользователя". Таким образом, любой объект, считающий себя наследником интерфейса IRepository, должен удовлетворять условию *не слабее* данному в родительском типе.

[**Invariants**](https://github.com/rkhaimov/brain-gym/blob/t_arch/plan.md#invariants)

Помимо условий на аргументы и возвращаемое значение, система также может содержать закономерности (всегда истинные утверждения), сохраняющиеся при любых разрешенных операциях. Возьмем репозиторий контроля администраторов. Имеются методы получения списка всех уполномоченных лиц, добавления, удаления и редактирования. При этом, ставится условие, что в любой момент времени, в системе *всегда* должен присутствовать хотя бы один админ. То есть чтобы не случилось, метод getAllAdministrators должен вернуть не пустой массив.

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_28.png)

Инварианты также должны удовлетворяться всеми наследниками интерфейса, в том числе и mocks. Стоит обратить особое внимание на то, что в ряде случаев возможно подобрать такую структуру, из свойств которой и будут следовать искомые условия. Это исключит человеческий фактор и упростит работу при разработке заглушек для тестирования.

[**Контракт**](#контракт)

Приведенные выше условия в совокупности формируют контракт. Из данного перечня свойств следуют уже собственные свойства и контракты клиента (и его клиентов и так далее по цепочке). Стоит воспринимать это как дополнительный способ защиты ( помимо тестирования) от некорректного поведения системы, перехода её в противоречивое с требованиями состояние.

[**Queries**](https://github.com/rkhaimov/brain-gym/blob/t_arch/plan.md#queries)

Отсюда следует что запросы (queries), хоть и не имеют внешних эффектов, все равно зависят от изменяющегося источника данных, например БД. Это приводит к тому что, например, возвращаемое значение getAllUsers может отличаться в зависимости от того, был ли метод вызван до или после удаления пользователя (то самое следствие post-condition).

Существует три варианта:

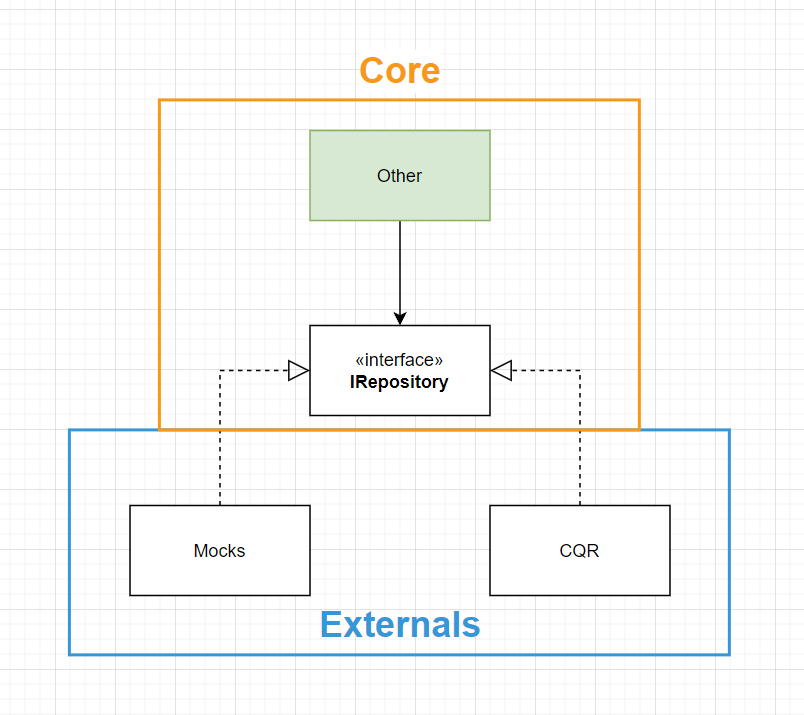
* Проверять порядок вызовов queries относительно других commands. К примеру, проверять что getAllUsers был вызван после успешного выполнения deleteUserById. Минусом является относительная хрупкость тестов, ввиду присутствия зависимости от деталей реализации работы с входящими данными (queries).
* Реализовывать post-conditions. В случае deleteUserById возможно эмулировать тривиальное хранилище и (понарошку) удалять пользователя. getAllUsers должен возвращать данные из хранилища. Очевидным недостатком данного способа является повышенная сложность тестов.
* Не реализовывать условия вовсе. Имеет смысл использовать в случаях когда верифицируемое поведение не зависит от требуемых условий.

В заключительной части мы построим механизм верификации визуальной составляющей наблюдаемого поведения и проведем анализ получившейся структуры, сравнив ее с другими популярными представителями.

**4 часть**

[**Поведение и представление**](#поведение-и-представление)

Рассмотрим итоговый результат:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_29.png)

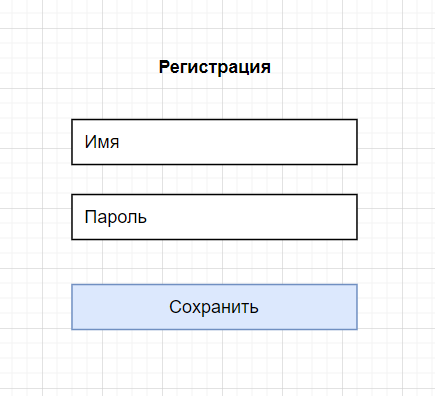
Итого:

* Автоматизированное тестирование покрывает большую часть кода в приложении. Данный показатель дополнительно поддерживается путем упрощения слоя доступа к данным (Externals).
* Верификация поведения (в данном случае) происходит с помощью снимков визуального представления и взаимодействия с сервером. Это требуется для достижения максимальных показателей защиты от регресса и сопротивляемости рефакторингу.
* Взамен на лучшую поддерживаемость, тесты зависимы от разделения ПО на два слоя. Это связь не представляет особых рисков ввиду высокой абстрактности самой архитектуры. В то же время, внутреннее устройство Core может изменяться в любом направлении, тесты при этом будут служить гарантом отсутствия регресса.

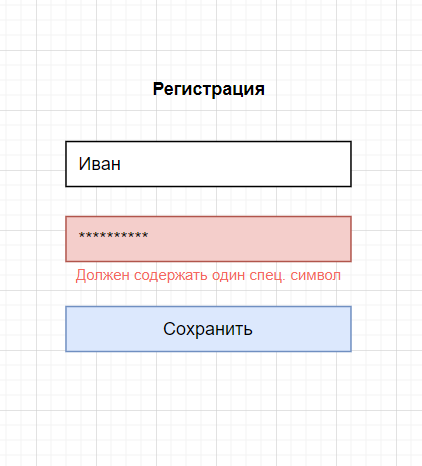
Но что насчет скорости выполнения? Безусловно, итоговый показатель будет сильно зависеть от среды выполнения и подхода к процессу тестирования в целом. Но в общем случае можно сказать наверняка - текущее решение имеет недостатки по быстродействию.

[**Единственная ответственность**](#единственная-ответственность)

Представим ситуацию, необходимо реализовать форму регистрации с двумя полями: имя пользователя и пароль.

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_30.png)

С точки зрения полноты тестирования, потребуется покрыть не только позитивный сценарий, но и также разные ветки валидации пароля. В определенных ситуациях их может быть подавляющее большинство, что приведет к ощутимому росту времени выполнения подпрограмм тестирования.

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_31.png)

Разработчик не всегда может избавиться от лишних требований, так как не всегда контролирует их. Вместо этого обратим внимание на текущий подход к тестированию. Большинству будет понятно даже на интуитивном уровне, что процесс визуальной верификации занимает собой наибольшую часть времени выполнения тестового сценария, так как включает в себя относительно тяжелые вычисления связанные с отображением интерфейса на экране. С другой стороны, требование строгости к паролю имеет только косвенное отношение к представлению.

Более наглядно это можно продемонстрировать с точки зрения причин для изменений:

* Валидация пароля изменится если изменятся требования к строгости пароля.
* Визуальное представление формы изменится если, например, заказчик захочет отображать ошибки сразу, а не только после того как пользователь попытался сохранить форму.

Ключевое утверждение здесь заключается в том, что эти причины *крайне редко*будут пересекаться по времени. Гораздо вероятней что за взятый момент времени будет меняться что-то одно, а не все сразу. Это понятно любому разработчику на основе эмпирического опыта.

Попробуем все же доказать данное утверждение. Кто является источником изменений для визуальной составляющей формы? Список скорее всего будут возглавлять UI/UX дизайнеры. Но что насчет правил валидаций? Здесь лица будут немного другими: аналитики и специалисты по безопасности. Конечно, между ними также могут быть пересечения, но основное и так бросается в глаза - источники изменений в основном *разные*.

Таким образом, функции валидации пароля и отрисовки формы имеют разные источники изменений. Это приводит к тому что данные функциональные блоки изменяются в рамках разных задач, поставленных в разное время и в целом могут выполняться разными людьми. Другими словами, данные блоки имеют *разные ответственности*.

[**Разделение ответственностей**](#разделение-ответственностей)

Наше намерение просто и понятно - разделить крупный блок таким образом, чтобы одна часть требовала визуальной верификации, а другая при этом - нет. Существует множество различных способов разделить компонент на составляющие, но чем руководствоваться? Если попробовать разбить крупный класс/модуль/функцию вслепую, можно только усугубить ситуацию тем, что при необходимости *изменений* в данном компоненте разработчику прийдется решать "пазл", собирая при этом разрозненные кусочки функционала в единую и понятную картину. Более того, необходимые изменения при этом будут разлетаться мелкой дробью по составляющим модуля, дополнительно усложняя при этом всю работу.

Ключевым здесь является *направленность* изменения, разность ответственностей (как между валидацией пароля и представлением) как раз и гарантирует то, что, вероятней всего, изменение затронет лишь один из компонентов, а не оба*одновременно*. Таким образом, хоть и несколько усложняя систему увеличивая кол-во сущностей в ней, взамен можно получить возможность существенным образом оптимизировать процесс тестирования.

Рассмотрим измененную схему:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_32.png)

Вместо одного блока Core на диаграмме указаны два компонента:

* Behaviour - слой, реализующий функции характеризующиеся следующими свойствами:
  + Отсутствие прямых зависимостей на средства отрисовки
  + Отсутствие прямых зависимостей на конкретные источники данных
  + Разность ответственности по отношению к представлению и доступу к данным
* View - слой содержащий в себе ответственность связанную с отображением. Имеет зависимость от средств отрисовки интерфейса (компонент фреймворка). То есть та совокупность функционала, которая может измениться по требованию, например, дизайнера.

Исходя из получившейся архитектуры можно заменить, что визуальные тесты напрямую зависят от конкретного представления и поведения. В то же самое время, в качестве Externals используют заглушки.

Для тестов компонента Behaviour ситуация иная - зависимость здесь исключительно от Behaviour. Конкретные источники данных также заменяются на заглушки, что касается примитивов рендеринга они не используются вовсе. Действительно, если бы Behaviour использовал компоненты представления, это бы заставило исполнять их в рамках snapshot тестов (например, на валидацию), что, опять же, привело бы к ситуации низкой скорости исполнения.

С другой стороны можно не заниматься в тестах честной отрисовкой и просто эмулировать некоторые из процессов (см. jsdom). Проблемы в таком решении ровно две:

* Эмуляция не предоставляет достаточной защиты от регресса так как не является реальной средой выполнения функций
* Эмуляция зачастую оказывает не сильно меньший эффект на производительность, чем реальная отрисовка компонентов

То же самое относится и к ссылкам на объекты событий в обработчика Behaviour (например React.MouseEvent). Для того чтобы вызвать такой обработчик в тестовой среде, потребуется инстанцировать комплексный объект события, где большинство свойств скорее всего не будут использоваться целевой функцией. В таком случае стоит либо выносить данные операции в слой представления, либо упрощать интерфейс (тип события), убирая из него неиспользуемые части.

[**Противоречие**](#противоречие)

По итогу:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_32.png)

В получившемся решении теперь возможно оптимизировать общее время выполнения тестов следующим образом: покрыть сквозные сценарии тестами с большей защитой от регресса (визуальные тесты), остальные не визуальные альтернативные сценарии закрыть с помощью тестов в изолированном окружении. В примере с паролем это выражается в небольшой группе визуальных тестов на саму форму регистрации и изолированных тестов на саму функцию валидации, покрывающую все требуемые сценарии.

Чем сильнее связь со структурной ценностью, тем меньше сопротивляемости к рефакторингу образуется в итоге.

Взамен на лучшую скорость, пришлось смириться с тестами, больше завязанными на структурную часть программы. В общем случае они будут более чувствительны к изменениям, не только в разрезе разделения слоев, но и в сигнатурах компонента Behaviour (аргументов и возвращаемого результата). Это накладывает дополнительную ответственность на разработчика при проектировании этих интерфейсов - они не должны раскрывать подробностей реализации и в то же самое время быть тривиальными для использования во View и в тестах.

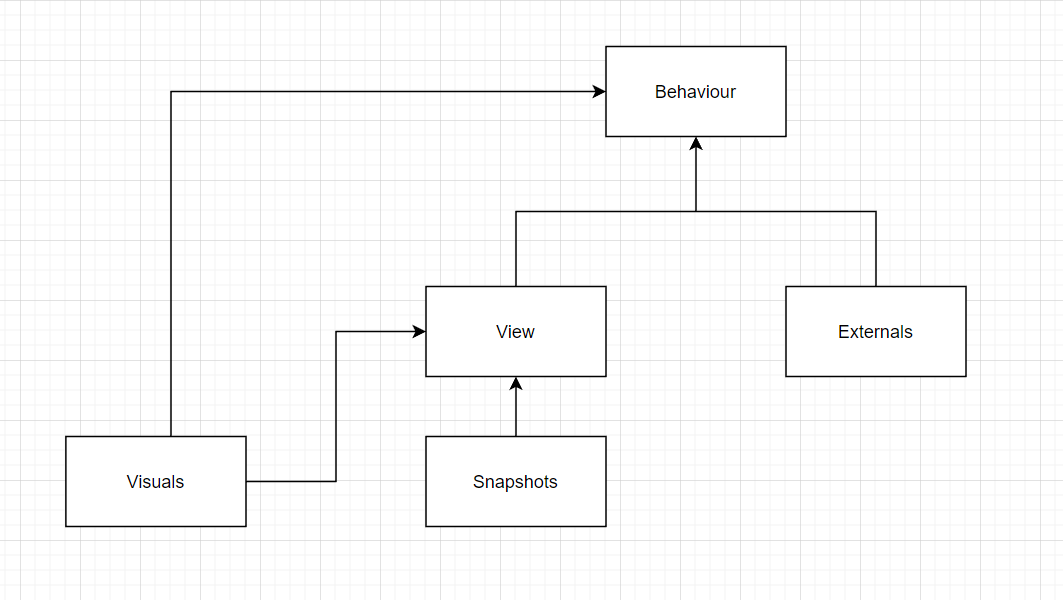
Опираясь на структуру, тесты вынуждают её быть простой в повторном использовании (то есть тестируемой). Что приводит к расширяемым функциям, классам, модулям (то есть объектам тестирования).

Данный факт должен привести к не простой мысли - идеальных тестов не существует. Причиной тому являются явные противоречия в свойствах последних. Например, чем выше защита от регресса, тем ниже их быстродействие. Ведь хорошая протекция требует выполнения большей части (если не всего) кода формирующего программу.

[**Оценка результатов**](#оценка-результатов)

[**Расширяемость**](#расширяемость)

Если посмотреть на архитектуру повнимательней, можно явно разглядеть иерархию, чем то похожую на пирамиду. Где компонент Behaviour находится на самом верху.

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_33.png)

Это говорит о том, что данный элемент является ответственным. То есть изменения внутри данного компонента, будут порождать изменения во всех зависимых от него частях (коих большинство). Разработчик может разными методами нивелировать данный эффект, но он редко доставляет большие проблемы, особенно при условии наличия тестов.

Положительным, и наиболее важным, здесь является свойство расширяемости:

* Возможно переиспользовать один и тот же Behaviour с разными View
* Возможно переиспользовать один и тот же Behaviour с разными Externals
* Возможно переиспользовать один и тот же Behaviour с разными фреймворками
* Возможно переиспользовать один и тот же View с разными Externals

Хорошей аналогией здесь будет видео редактор с системой плагинов. Знает ли что-то редактор (ядро) о существовании конкретных плагинов? Нет. Знают ли что-то конкретные плагины о существовании редактора? Да.

Структура не стала такой какая она есть исключительно ради возможности использовать один и тот же Behaviour с разными View. Вероятней всего это и не понадобится. Главная причина существования такой архитектуры - тестируемость, цели которой были раскрыты ранее. Такая взаимосвязь между расширяемостью и тестируемостью проявляется вследствие того, что тесты это ничто иное как процесс *повторного использования* компонента, но уже в совершенно другой среде. Именно это и форсирует структуру кода быть гибкой (то есть адаптироваться под окружение).

[**Адаптируемость**](#адаптируемость)

Даже в таком состоянии, все от чего зависят тесты - это разделение программы на три условных слоя. Это открывает возможность в изменении внутренней структуры данных слоев (например, в сторону упрощения системы) без необходимости при этом модификации самих тестов. При этом, во время всего процесса, последние будут выступать гарантом сохранения поведенческой ценности (отсутствия регресса).

[**Универсальная архитектура**](#универсальная-архитектура)

Может показаться, что разработчику предлагается универсальный подход к тестированию (а значит и архитектура), которая должна использоваться на каждом проекте.

Это, конечно же, не так.

Выбор сильно зависит от специфики, требований и общих условий проекта. Так, разработчики относительно тривиального интернет магазина или лэндинга могут выбрать для себя больший уклон в верификацию наблюдаемого поведения. Более гранулированные тесты могут оказаться бесполезными и только усложнят общее решение. Но даже в таком случае, изолирование доступа к данным может оказаться обязательным.

Разработчики библиотеки низкого уровня (например lodash), наоборот, неизбежно будут верифицировать и часть структуры. В их случае она будет составлять часть наблюдаемого поведения, так как клиентами данного пакета являются другие разработчики.

На проектах средней сложности и объема могут подойти "интеграционные" тесты, требующие, как было показано, разделения приложения на два слоя (Core/Externals).

В случае увеличения кол-ва функционала, можно разбить тестовые кейсы добавив новое разделение ( View/Behaviour/Externals). Сквозные сценарии и наиболее вероятные альтернативные могут проверяться тестами более высокого порядка. Все остальное возможно верифицировать уже на более низком уровне.

[**Изменение требований**](#изменение-требований)

"Проект, в котором требования изменяются очень часто, только пострадает от тестирования". Это один из частых аргументов противников тестирования. Корнем данного утверждения является простой факт - тесты увеличивают время на разработку. С этим сложно спорить, ведь сами по себе они увеличивают кол-ва кода, требуемого к написанию. Упрощая максимально - разработчику требуется напечатать больше символов при прочих равных, следовательно, времени на задачу уйдет больше.

Такое рассуждение является манипулятивным. Рассматривать тесты в рамках конкретной задачи неверно, ведь основным их эффектом является поддержка возможности контроля сложности программы. То есть такого параметра, который влияет на скорость и качество выполнения и поддержки всех задач в целом, вероятность появления дефектов также входит в их число. Сюда также примешиваются время работы QA инженеров, состояние команды, репутационные потери и так далее.

Частой причиной является низкая поддерживаемость тестов. Как показывалось раннее, это может быть вызвано проблемной структурой самих тестов или недостаточной расширяемости архитектуры тестируемого приложения (элементы системы трудно использовать повторного в рамках ограниченного окружения).

Также картину может портить факт "бесполезности" самих тестов. Например, в случае когда рефакторинг проводится редко, либо когда тесты не используются как "гарант" и приложение все равно проверяется разработчиками в ручную. Последнее больше говорит о низкой защите от регресса (частый симптом unit тестирования).

В самых неприятных случаях, тесты могут и вовсе мешать приложению развиваться. Если после модификации кода, наблюдаемое поведение не изменилось, но тесты падают - это называется хрупкостью. Такие тесты обладают низкой сопротивляемостью рефакторингу.

Как можно заметить, указанные проблемы сводятся к общим свойствам или характеристикам тестов. Методы управления последними были описаны ранее и при правильном их применении тесты не окажут ощутимого негативного эффекта, а то и увеличат производительность труда.

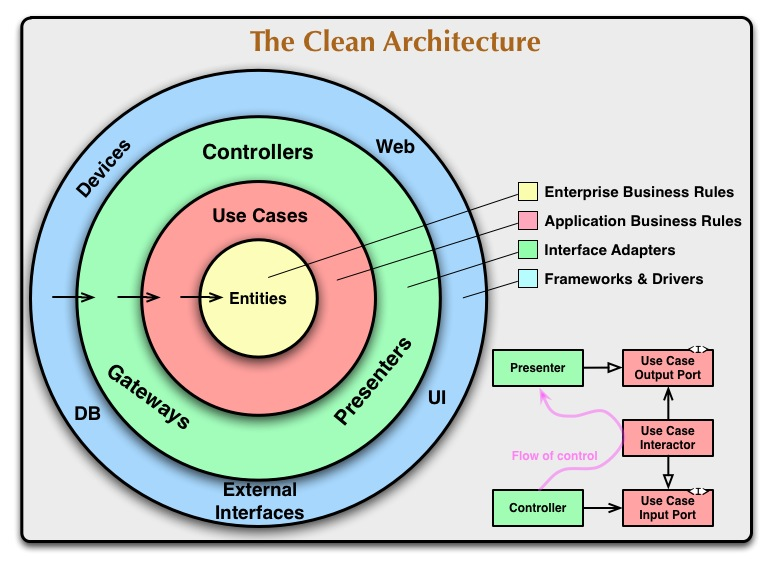
Вот некоторые из рекомендаций:

* Тесты должны помогать реализации, а не быть "долгом" разработчика. Они являются лучшим инструментом разработки/отладки функционала, так как предоставляют изолированную среду, где можно воспроизвести любое состояние приложения
* На проекте должны проводиться регулярные рефакторинги. Как следствие, тесты должны запускаться часто, это добавит им оправданности и пользы (в случае достаточной защиты от регресса и сопротивляемостью к изменениям структуры).
* На каждый найденный дефект должен быть написан тест, таким образом, проект будет расти инкрементально, следующий релиз никогда не будет хуже предыдущего.

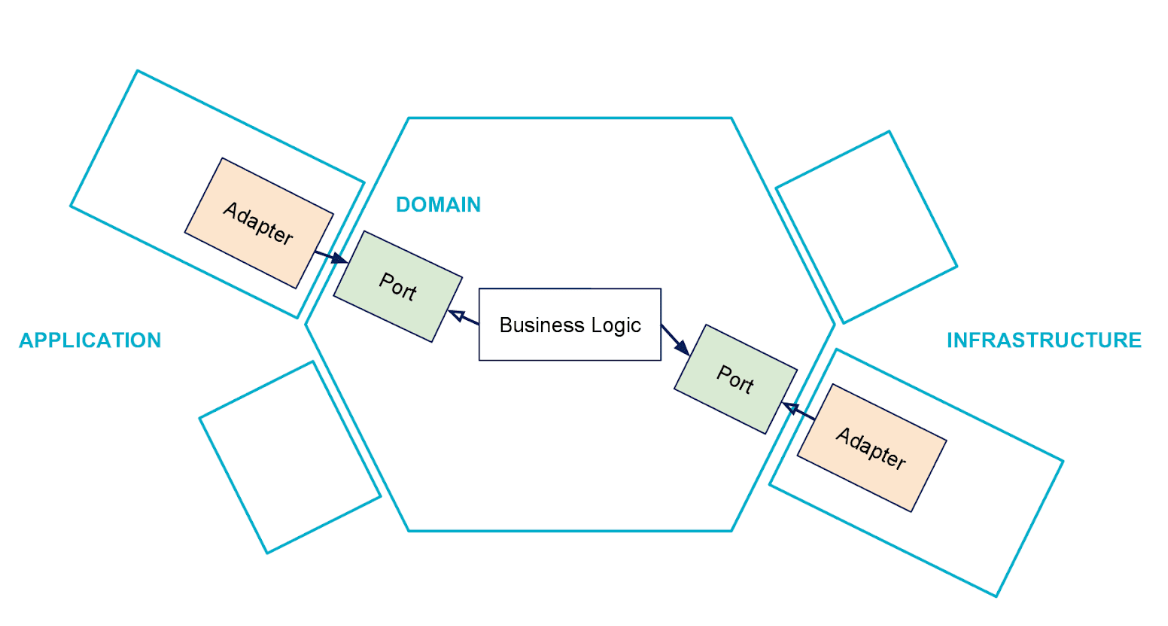
[**Сравнение с существующими архитектурами**](#сравнение-с-существующими-архитектурами)

Рассмотрим архитектуры:

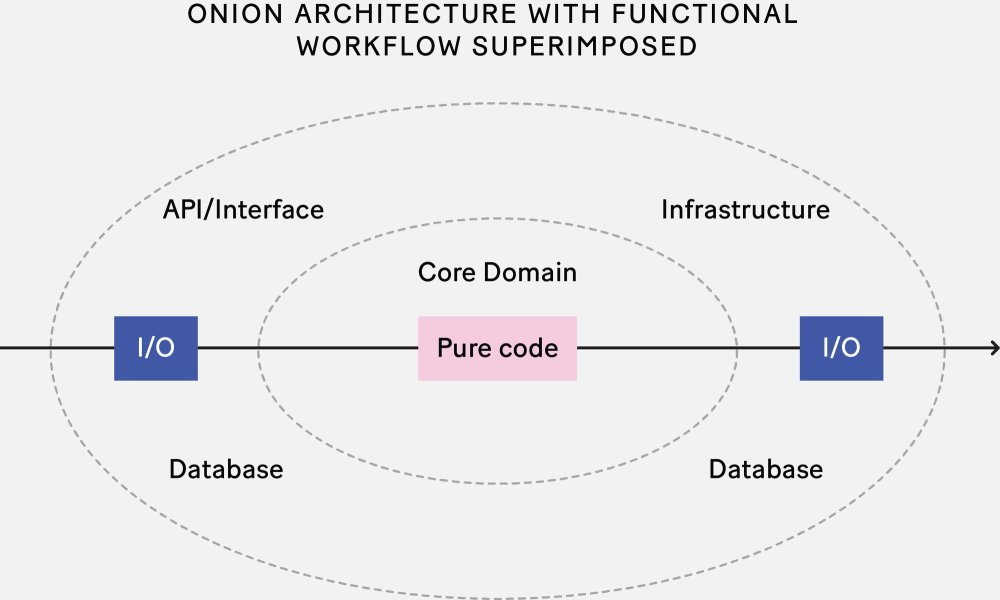
* Чистая (луковая) архитектура:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_34.png)

* Гексагональная архитектура (порты и адаптеры):

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_35.png)

* Функциональная архитектура:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_36.png)

* Теперь посмотрим на то что получилось в результате тестирования:

[](https://github.com/rkhaimov/brain-gym/blob/t_arch/img_32.png)

[**Пересечения**](#пересечения)

Если приглядеться, то можно заметить следующие объединяющие их свойства:

* В центре находится самый абстрактный компонент. То есть тот, что имеет меньше всего конкретных зависимостей от окружения и конкретных его реализаций. В нашей структуре также имеется данный элемент. Обратите внимание на то что обоснованием его появления являются быстродействие тестов и ответственности компонентов.
* Зависимости направлены в сторону повышения конкретности. То есть, компонент зависит от более абстрактного элемента, не наоборот. В случае с тестированием это также имеет смысл. Behaviour не должен зависесть конкретных примитивов рендеринга, иначе исполнение тестов не будет таким быстрым, а поддержка простой.
* Средства ввода вывода (инфраструктура) находятся на границах системы. Максимально конкретные и тривиальные компоненты. Данный факт является следствием предыдущего пункта.
* Отсутствие циклов между слоями. Цикл зависимостей сам по себе не имеет "правильного" порядка разрешения, несмотря на то что современные среды умеют успешно с ними работать (в большинстве случаев).

[**Различия**](#различия)

Помимо сходств, если и некоторые расхождения. Возьмем к примеру чистую архитектуру:

* Use Cases/Entities - составляют ядро системы. В нашем случае это один Behaviour. Entities - так называемый "чистый домен", то есть правила имманентные бизнесу (внутренне присущи, определяющие), характеризуется тем что никогда не изменяются (или крайне редко). Такое разделение не имеет обоснований с точки зрения тестирования, поэтому и является избыточным.
* Контроллеры/Презентеры - работают с входящими и выходящими данным соответственно, можно считать их адаптерами конвертирующими примитивы среды в примитивы ядра (и наоборот). С точки зрения тестирования важно чтобы в Behaviour не было зависимостей которые сложно использовать повторно (тестировать), также необходимо чтобы Behaviour не раскрывал деталей внутренней реализации. Обе задачи не требуют использования указанных сущностей и, как следствие, они также являются избыточными. Взамен, данные компоненты могут использоваться напрямую (явным или неявным образом) в самом слое View. Тестируемая инфраструктура не обязана взаимодействовать с ними напрямую, оперируя при этом исключительно пользовательскими событиями (кликами, жестами и тому подобное) и запросами, как было показано ранее.

Естественно, исходя из вышесказанного не значит что архитектуры содержат в себе лишние компоненты. Скорее это показывает, что даже в своём крайнем проявлении, тесты не требуют от архитектуры многого, позволяя последней подстраиваться под потребности бизнеса и производительности труда.

[**Заключение**](#заключение)

Приведенные выше техники не являются откровением. Они достаточно известны и часто встречаются в литературе, однако, их редко используют (или используют осознано). Это может быть связанно с непонимания решаемых целей и задач, либо при отсутствии успехов их согласования с заказчиком (согласование работ по техническому долгу и/или написание тестов). Оба фактора связаны. Относясь к паттернам как к "брэндам", используя их по форме, а не по содержанию, программист рискует только ускорить процесс гниения проекта.

Одной из основных ролей инженера заключается в понимании простого факта: с каждым реализованным требованием, с каждым исправленным дефектом, сложность системы неуклонно растет, выражаясь в возрастающем времени выполнения задач. Следовательно, стоимость проекта увеличивается, что не всегда можно сказать про его бюджет.

Поэтому чрезвычайно важно контролировать данный показатель, путем постоянной корректировки структуры в направлении упрощения поддержки и развития ПО. Это делается не один раз, не раз в неделю, а непрерывно на протяжении всего цикла разработки. Достичь этого нельзя не имея при этом качественных тестов. Они в свою очередь, требуют использования подходящих структурных парадигм и паттернов, таким образом, у последних появляется четкая, измеряемая причина для существования - деньги.

Что более важно, эта причина понятна не только разработчикам, но и менеджерам, заказчикам и руководителям, тем, перед кем и требуется чаще всего обосновывать данные работы. Этого не говорят напрямую, но от инженера ожидают умения сказать "нет", предупредить риски, обеспечить расширяемость и поддерживаемость системы на всем периоде её существования.

<https://habr.com/ru/companies/oleg-bunin/articles/735040/>