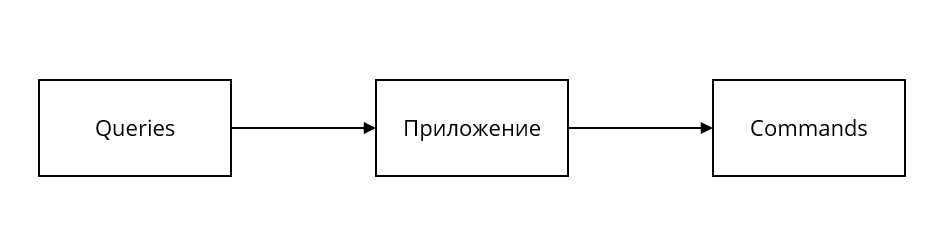
**Верификация источников данных**

Прежде чем пытаться верифицировать приложение через изображение на мониторе, попробуем начать с, казалось бы, более простой задачи, а именно проверки ПО, через взаимодействие с источниками данных.

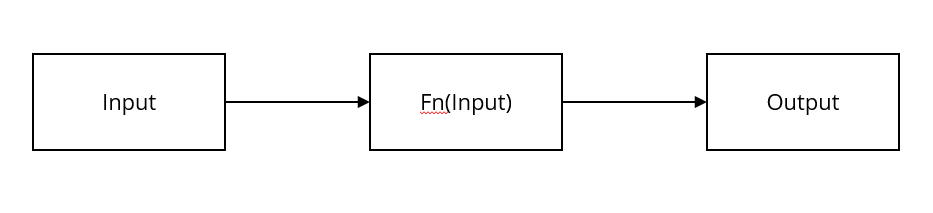
Уберем из схемы портов приложения все лишнее, оставив только те, которые находятся в зоне источников данных:



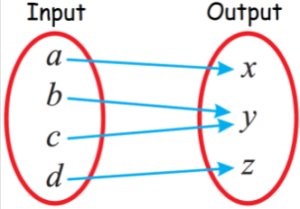
Queries являются источниками тех данных, которые входят в приложение. Команды же, представляют из себя место вывода. Ничего не напоминает?

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

Так выглядит базовое, схематичное определение любой чистой функции:

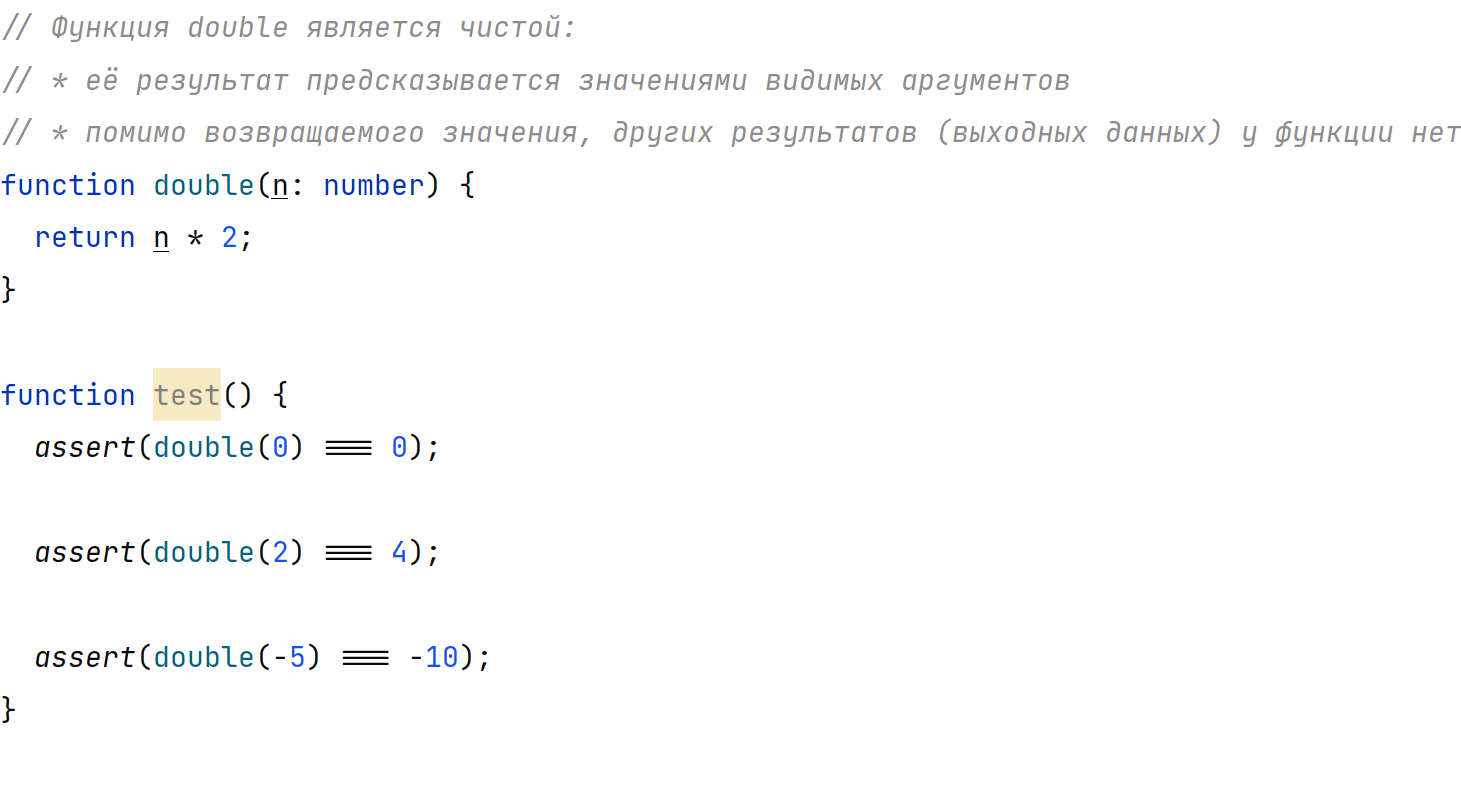


Чистая функция – это лишь *трансформация* входных данных, в результат. Другими словами, это простой *маппинг*, в котором выставляется ассоциация между входящим значением и исходящим:



*Любая чистая функция может быть представлена как ассоциация входящих значений на исходящие*

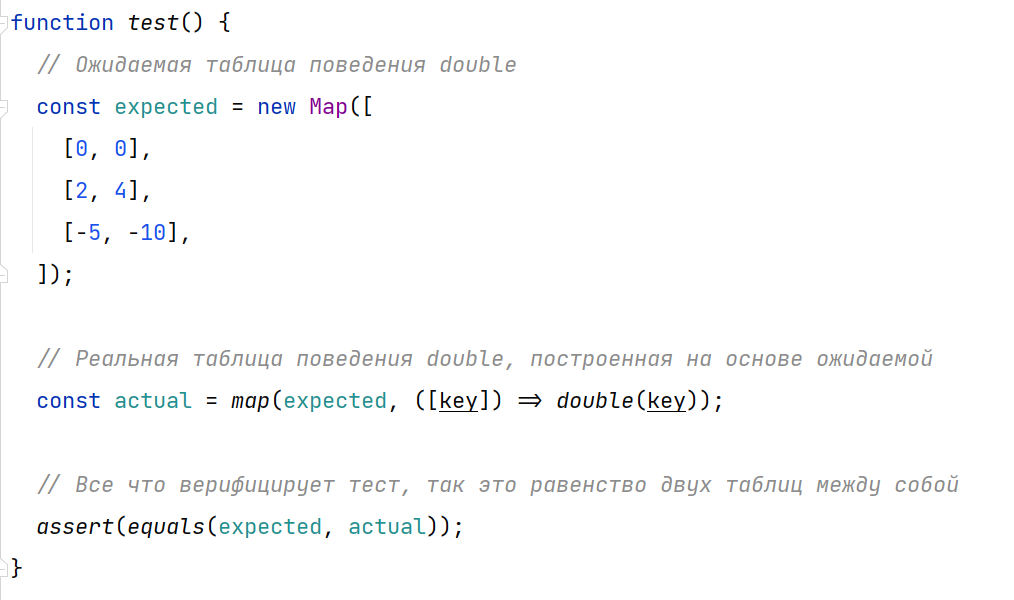
Давайте посмотрим, как выглядел бы тест, проверяющий корректность такой функции:



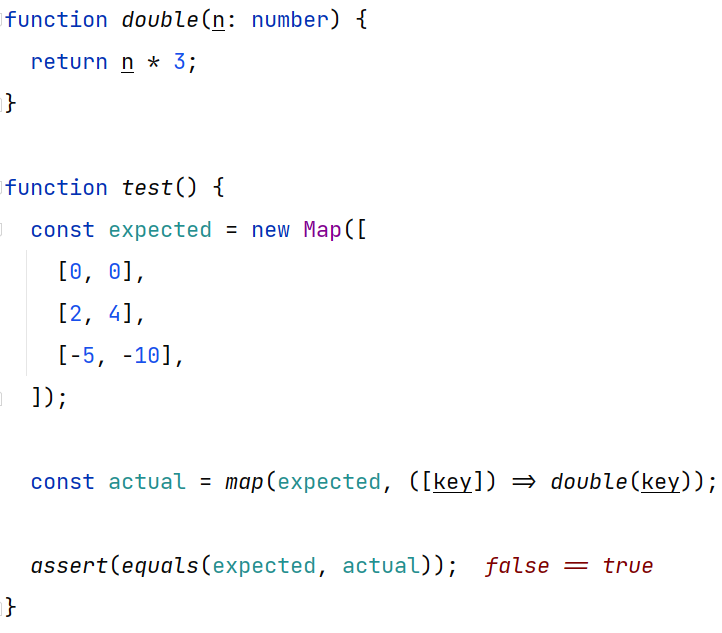
Теперь попробуем выдвинуть следующую гипотезу, тест — это ничто иное как процесс состоящий из следующих шагов:

1. Описания *ожидаемой* таблицы ассоциаций (arrange)
2. Генерация *актуальной* таблицы ассоциаций (act).
3. Сравнение обоих таблиц (assert)

Вот как бы выглядел тест из примера выше в указанной структуре:

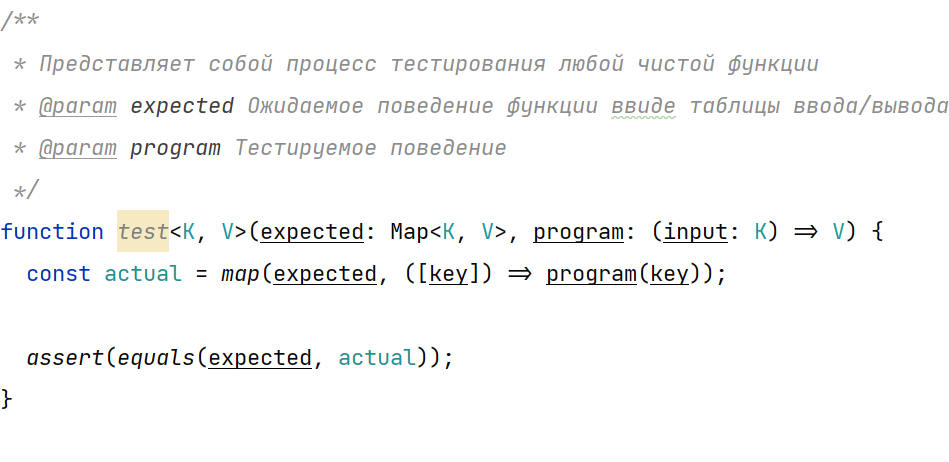


Теперь, чтобы доказать, что тест не потерял в эффективности, попробуем сломать поведение тестируемой функции double:



*Тест ожидаемо упал так как реальное поведение функции double не соответствует ожидаемому*

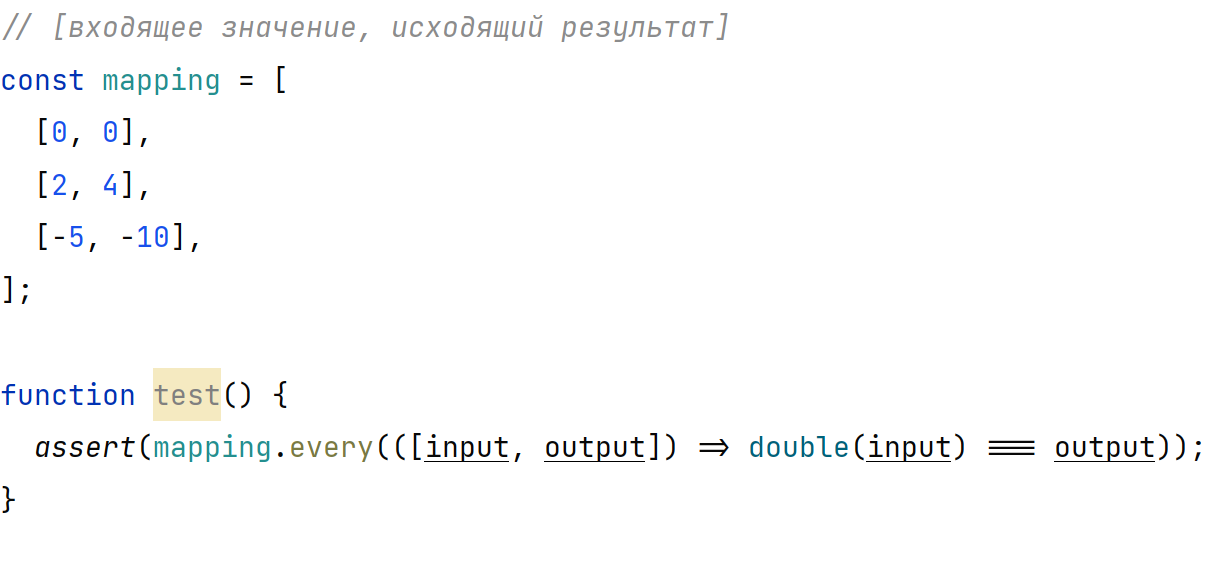
У любого инженера возникнет естественное желание абстрагировать типовую структуру теста, давайте так и поступим:



Теперь, все что нужно сделать для тестирования программы так это создать объект expected, который является таблицей, соотносящей ввод на вывод.

**Значимость значений**

Обратим своё внимание на очевидный и, казалось бы, маловажный факт, который заключается в том, что таблица I/O это прежде всего *значение*:

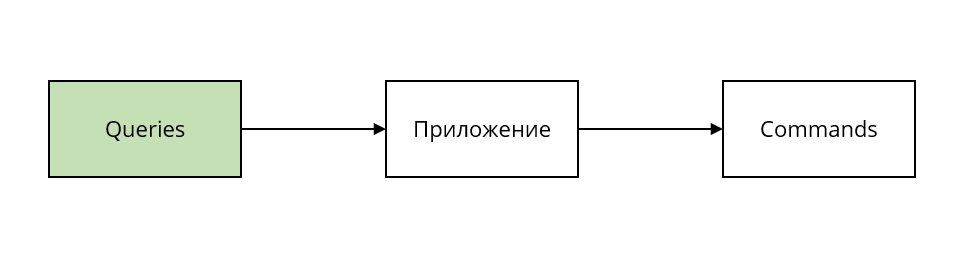


*Значение* можно записать в переменную, передать в функцию, вернуть из функции, иными словами – можно *сохранить в памяти*. Ключевым здесь является то, что таблица, как агрегирующее значение, требует, чтобы все его элементы (key, value) *также были значениями*.

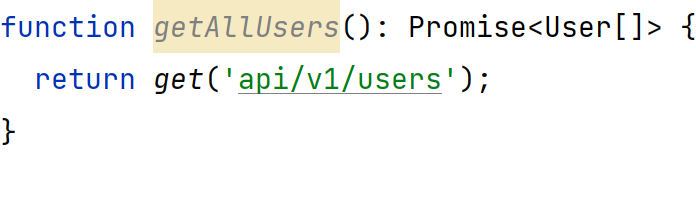
Другими словами, разработчику, для того чтобы протестировать свою программу, необходимо материализовать (превратить в значения) входящие и исходящие данные, что далеко не всегда является тривиальной задачей, как будет показано позднее.

**Материализация query**

Начнем с входных данных, вспомним что таковыми в случае источников данных являются запросы (queries):

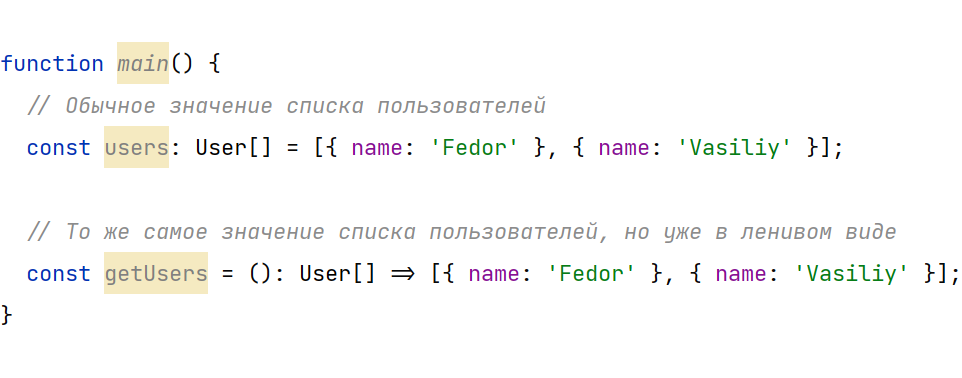


Рассмотрим типовой query:



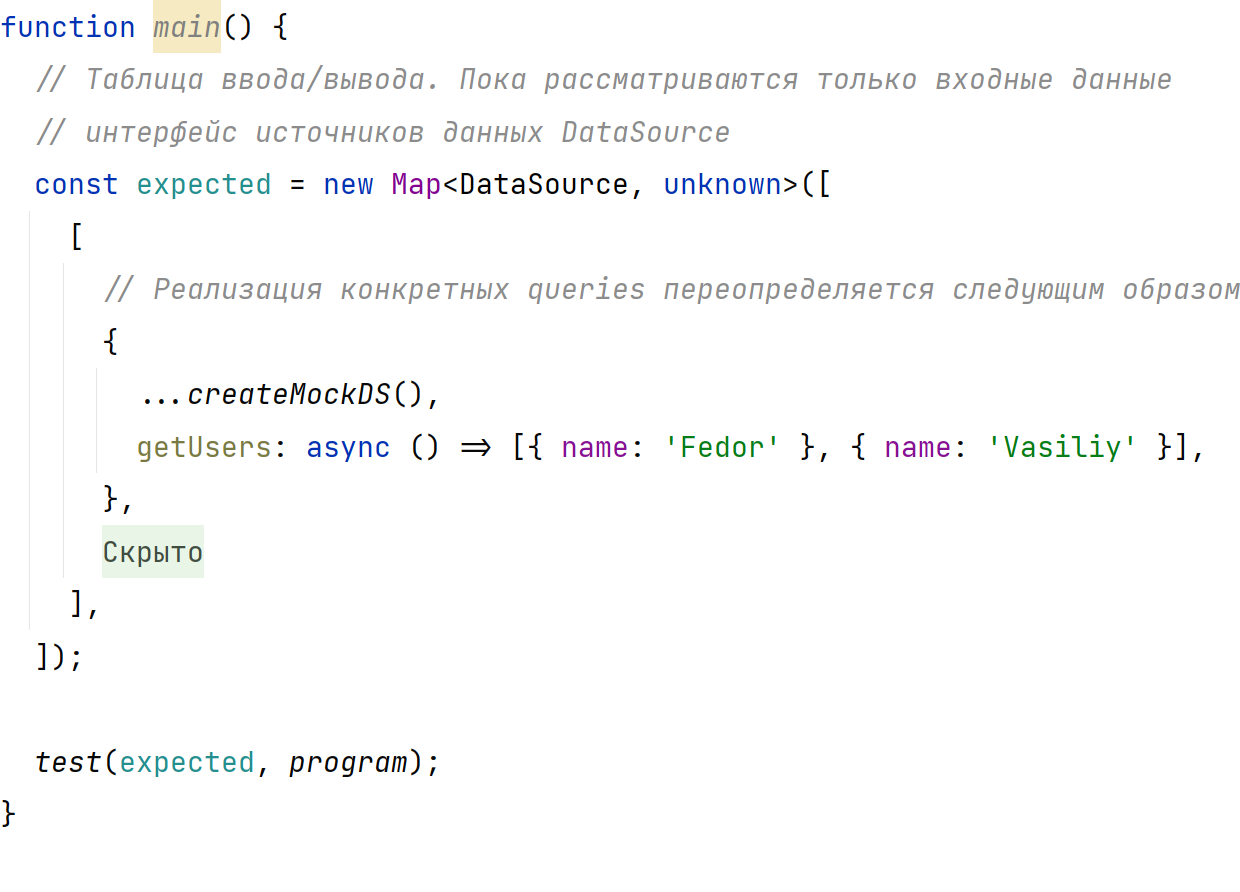
Как представить подобного рода структуру как ключ таблицы тестирования? Начнем с того что в JavaScript функции являются *first-class* элементами, то есть значениями, другими словами. Это уже решает половину проблемы.

Что же касается их “настройки”, сейчас достаточно воспринимать queries как те же самые данные, только в *ленивом* виде:



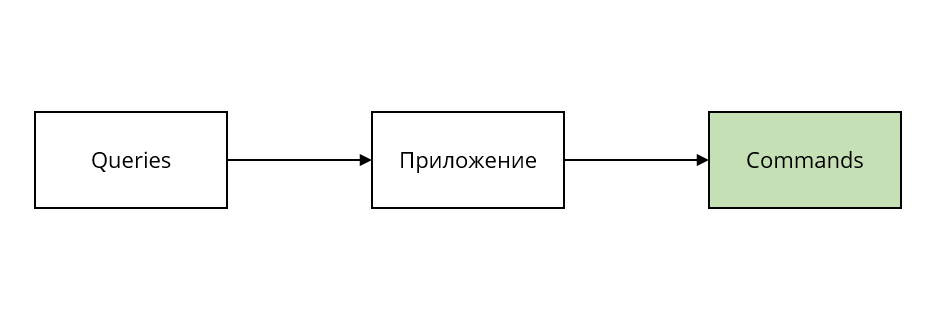
*Ленивым значение является только потому, что будет вычислено только тогда, когда клиентов явно об этом попросит (вызвав функцию).*

Исходя из этого, queries в тесте можно настраивать следующим образом:

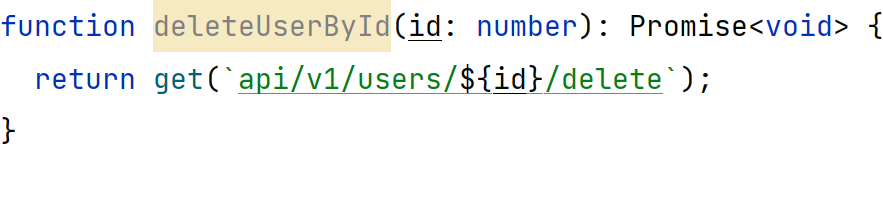


**Материализация commands**

После того как разобрались с вводом (queries), пришел черед вывода (commands):



Рассмотрим конкретный пример:

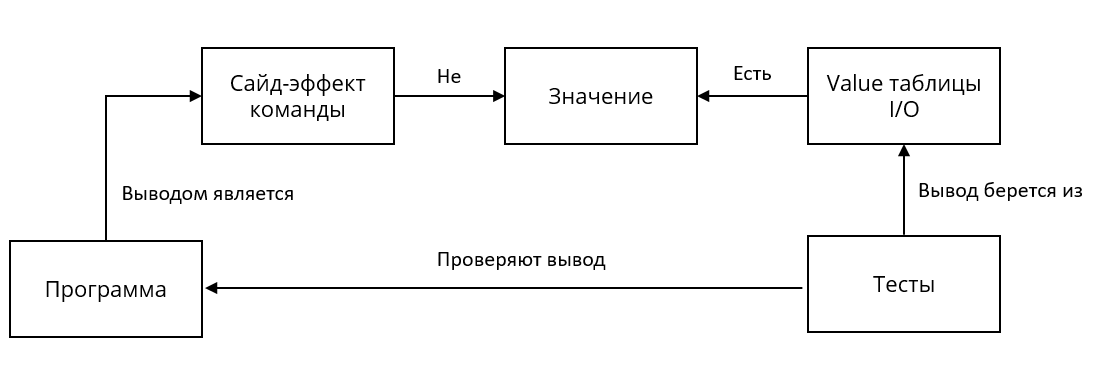


Как было показано ранее, единственным средством, через которое query могут управлять поведением приложения, является их возвращаемое значение. У них просто не может быть никаких других вариантов воздействия на ПО, так как они, по определению, не содержат в себе никаких сайд-эффектов. Поэтому при тестировании программы через queries нам достаточно получения контроля над тем, что конкретный запрос вернет во взятый момент времени, что мы и сделали.

В случае с командами ситуация немного иная:

* Во-первых, в нашей терминологии commands являются результатом работы программы (значениями таблицы I/O), а не её входящими аргументами как с queries.
* Во-вторых, результатом работы программы является не сама функция команда, а её сайд-эффект, которым, по определению, любая такая процедура должна обладать.

Противоречие можно представить следующим образом:

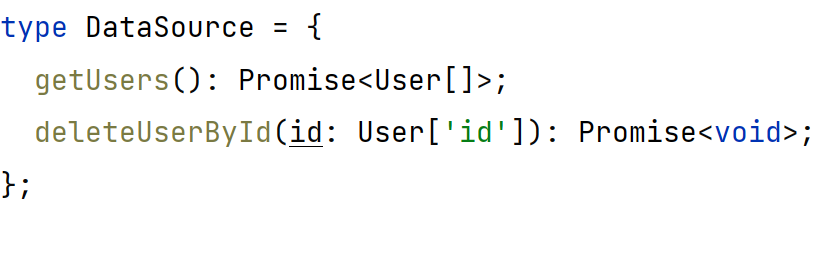


*Сайд-эффекты, чтобы быть верифицированными, должны быть значениями, коими они не являются по определению.*

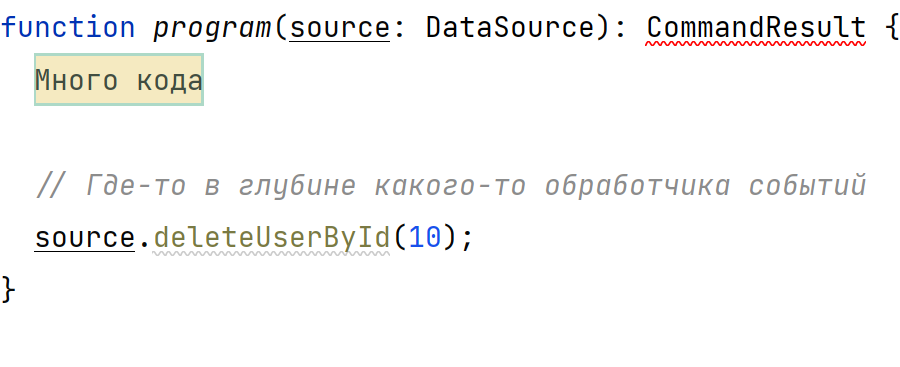
**С небес на землю**

Очевидно, что корнем проблемы, возникающей при формировании таблицы I/O, является тот факт, что программа выводит результат через сайд-эффекты, а не возвращаемые значения. В случае источников данных, виновниками являются команды.

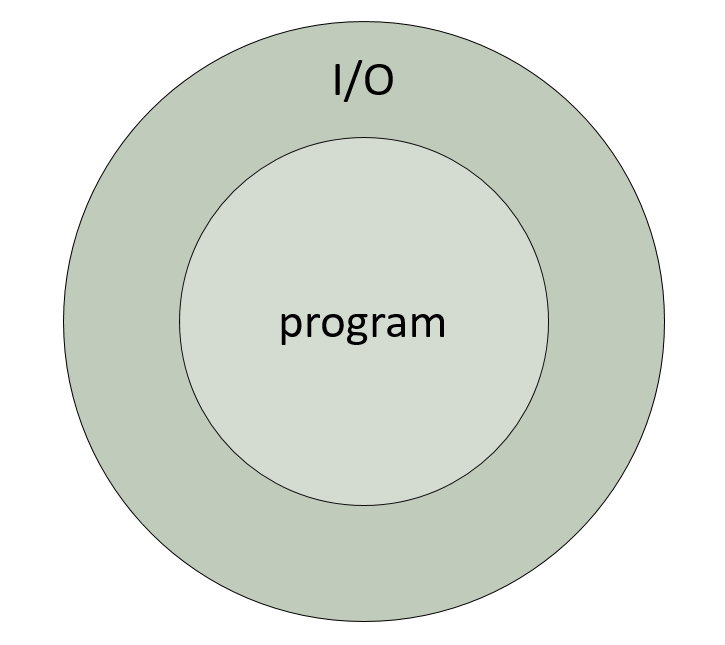
Если коротко, простого способа избавится от сайд-эффектов в командах нет. В частности, это вызвано тем, что на уровне сайд-эффект ожидается от команды на уровне *контракта источников данных*:

*То, что deleteUserById возвращает асинхронный void, но при этом оказывает влияние на наблюдаемое поведение программы и подразумевает наличие сайд-эффекта*

Более того, недостаточно просто сделать команду чистой, необходимо также, каким-то образом вернуть её результат из функции program:



Если коротко, то тестирование программы через описание таблицы I/O, требует от последней быть чистой:



*Чистая программа (ядро), сайд-эффекты вынесены наружу*

Каким образом можно организовать подобного рода структуру и процесс её тестирования будет показано в рамках отдельной статьи посвященной *функциональной архитектуре*.

Сейчас же вернемся к более практическому примеру, а именно программе, выполняющей сайд-эффекты.