IDEA code inspections, практика

Юрий Литвинов
yurii.litvinov@gmail.com

30.01.2019

1. IDEA code inspections

По моему опыту, на продуктивность программиста в большей степени оказывает влияние не владение языком и не алгоритмическая подготовка, а скорее чисто технические навыки типа умения решат проблемы со сборкой и, главное, умение пользоваться инструментами, упрощающими разработку. Самый важный такой инструмент — IDE. Фич у современных IDE и инструментов вокруг них много, сегодня рассмотрим статические анализаторы, или точнее — IDEA code inspections.

Используется статический анализ для того, чтобы найти как можно больше багов ещё даже до запуска программы. Статические анализаторы — это не альтернатива юнит-тестам, а скорее дополнение к функциональности поиска ошибок, встроенной в компилятор и сам язык программирования. Статический анализ тоже не может найти всех ошибок в программе и уж тем более доказать, что их нет (например, из-за проблемы останова машины Тьюринга: невозможно написать программу, которая для любой программы скажет, закончит она работать или нет). Тем не менее, статический анализ полезен, потому что иногда может поймать такие ошибки, которые юнит-тесты бы никогда не поймали.

Настраивается встроенный в IDEA анализатор через меню Settings -> Editor -> Inspections, а запускается через меню Analyze -> Inspect Code... . Вы увидите, что встроенных анализаторов в IDEA очень много, причём они имеют разные параметры, позволяющие их настроить, и некоторые по умолчанию включены, некоторые нет. Имеет смысл почитать про эти анализаторы и включить как можно больше их. Почему не все — некоторые анализаторы проверяют взаимоисключающие правила или противоречат стайлгайду, так что надо смотреть, чтобы не включить лишнего.

2. Nullability Analysis

Самый полезный, пожалуй, вид статического анализа из поддержанных IDEA — это анализ на корректность использования null-ов, так называемый nullability-анализ. Работает он так: программист размечает свой код аннотациями `@NotNull` и `@Nullable` из пакета org.jetbrains.annotations, после чего анализато пытается доказать, что везде, где мы ожидаем NotNull, значение не может быть null-ом, и выдаёт предупреждение, если доказать не удалось. Аннотация `@Nullable` говорит, что значению разрешено быть null-ом, так что если

мы пытаемся присвоить переменной, помеченной `@NotNull` значение переменной, помеченной `@Nullable`, это точно приведёт к сообщению об ошибке. Естественно, статический анализ можно обмануть, поэтому генерируются ещё проверки времени выполнения, которые заставляют программу упасть, если предположения программиста не были выполнены.

Аннотациями `@NotNull` и `@Nullable` можно помечать поля класса, параметры методов, возвращаемые значения методов, переменные. Проверки выполняются в основном “на лету”, когда IDEA прямо в коде подсвечивает нарушения аннотаций. Ещё интересно, что она указывает на другие возможные ошибки, например, сравнение с null переменной, которая явно помечена как `@NotNull`.

Небольшой пример из документации:

```
import org.jetbrains.annotations.NotNull;
import java.util.ArrayList;

public class TestNullable {
    public void foo(@NotBull Object param) {
        int i = param.hashCode();
    }

    public void callingNotNullMethod(ArrayList list) {
        if (list == null) {
            otherMethod(list);
        }

        foo(list);
    }
}
```

Обратите внимание, что правильные аннотации находятся в пакете org.jetbrains.annotations, который IDEA умеет добавлять в проект сама. Но всё равно про него нужно помнить и не забывать включать в build.gradle, pom.xml и т.д. Такие же аннотации есть в некоторых других пакетах (например, были в com.sun.internal.что-то-там), с ними статический анализ работать, скорее всего, не будет.

По поводу рекомендаций к использованию этих аннотаций всё просто — чем больше, тем лучше. Локальные переменные с маленькой областью видимости помечать, наверное, не стоит, а вот поля и параметры методов (особенно public) бывает очень полезно (поэтому обязательно в домашке начиная с сегодняшнего дня).

Кстати, в Java есть тип `Optional<T>`, представляющий наличие или отсутствие значения (как Maybe в некоторых функциональных языках), он в каком-то смысле альтернатива nullability-анализу. В каком: мы договариваемся вообще не использовать null никогда, если значения всё-таки может не быть, представлять его Optional и отсутствие значения представлять Optional.empty(). Но проблема в том, что Optional сам ссылочный тип, следовательно, может быть null, и компилятор без сторонней помощи проверить корректность программы никак не может. Так что nullability-анализ круче. У него есть один важный минус — хорошо инструментально поддержан он только в некоторых средах разработки, конкретно то, про что тут рассказывается — вообще только в IDEA. В консольной сбор-

ке всё тоже работает, но только во время выполнения, предупреждения при компиляции выдаваться не будут.

3. Контракты

На самом деле, nullability-анализ — это частный случай более крутой функциональности, связанной с определением и проверкой *контрактов*. В IDEA есть специальная аннотация `@Contract`, позволяющая задать ожидаемое поведение метода (в духе “если выполнено такое-то предусловие, то должно быть выполнено такое-то постусловие”). Поведение проверяется как статически (ну, что возможно), так и генерацией проверок времени выполнения. Контракты описываются довольно хитро и вместе с тем довольно немногое можно описать. Вот некоторые примеры:

- `@Contract("null -> null")` — если методу передали в качестве параметра null, он должен вернуть null.
- `@Contract("_ -> this")` — что бы ни передали методу, он должен вернуть указатель на текущий объект.
- `@Contract("!null, _ -> param1; null, !null -> param2; null, null -> fail")` — метод возвращает первый из двух параметров, который не null (если первый параметр не null, возвращается он, если второй не null, а первый null, то возвращается второй, если оба null, бросается исключение).

Более подробно про это дело рекомендуется почитать в документации: https://www.jetbrains.com/help/idea/contract-annotations.html.

4. Задача на практику

Требуется реализовать generic-класс `Maybe<T>`, который очень похож по функциональности на `Optional<T>` из стандартной библиотеки. А именно, надо реализовать методы:

- `public static <T> Maybe<T> just(T t)` — создаёт новый объект типа `Maybe<T>`, хранящий в себе переданное значение.
- `public static <T> Maybe<T> nothing()` — создаёт новый объект типа `Maybe<T>`, указывающий на отсутствие значения.
- `public T get()` — возвращает значение, если оно есть, бросает исключение, если нету.
- `public boolean isPresent()` — говорит, есть ли значение или нет.
- `public <U> Maybe<U> map(Function<?, ?> mapper)` — возвращает новый объект типа `Maybe<U>`, полученный применением переданной функции из T в U к объекту `Maybe<T>`. Если было nothing(), должно и быть nothing(), если было значение, то применяется функция к значению и результат оборачивается в just(). Тут надо подумать, что написать вместо вопросиков, потому что просто `Function<T, U>` будет не очень.

Maybe должно быть можно создать только статическими методами just и nothing. Бонусное задание — метод nothing() не должен создавать новые объекты каждый раз.

Дальше надо попользовать Maybe для преобразования чисел из файла.

- Читаем файл построчно.
- Если в файле на строке было не число, должно создаться nothing().
- Если в файле было число, должно создаться Maybe с этим числом.
- Все прочитанные Maybe складываются в список.
- После того, как первый файл дочитан, создаётся второй файл и туда выводятся:
  - “nothing”, если в Maybe было не число;
  - квадрат числа, если в Maybe было число.