Практика 2: Задача про grep

01.03.2017

1. Комментарии по домашней работе

Начнём мы с предыдущей домашней работы, про Command Line Interface. В общем-то, основная проблема с ней в том, что её почти никто не пытался пока сдавать. Это даже хорошо, потому что в этот раз можно будет продолжить обсуждать её архитектуру, уже более предметно, потому что кто-то уже воплотил её в коде, а кто-то хотя бы имел время подумать. По тем решениям, что были сданы, в общем-то, были лишь небольшие замечания касательно архитектурных решений, но было много замечаний касательно оформления кода.

Первое — это комментарии. Комментарии необходимы для сколько-нибудь больших проектов, иначе у них довольно быстро начинаются проблемы с сопровождаемостью. Кроме того, чуть ли не главная польза от комментариев в том, что они ещё раз заставляют задуматься, что и зачем вы пишете, и придумать краткую формулировку цели существования класса, интерфейса или метода. Обратите внимание, что классам и интерфейсам тоже нужны комментарии. Комментарии к методам — это хорошо, но единицей декомпозиции в объектно-ориентированных программах, в общем-то, является класс, и у классов тоже (и даже прежде всего) должно быть чёткое предназначение, описанное одним-двумя предложениями в комментарии над всем классом. Помните про принцип единственности ответственности — если вам приходится писать целый роман про то, что делает ваш класс и для чего он нужен, значит, этот класс слишком много на себя берёт и его можно разбить на несколько.

Комментарии к интерфейсам ещё важнее, чем комментарии к классам, потому что есть принцип инверсии зависимостей — всё должно зависеть от абстракций. А абстракции обычно представляются интерфейсами (ну или чисто абстрактными классами в C++, где нет интерфейсов). Если все вокруг пользуются интерфейсами, было бы неплохо, если бы про каждый интерфейс было написано, что он такое и для чего нужен. Ещё, кстати, есть принцип сегрегации интерфейсов, который тоже сводится к тому, что должно быть можно сказать в двух словах, что делает интерфейс.

Ещё надо комментарии ко всем public-методам, даже если это геттеры или что-то суперпростое. Почему — потому что есть тулы для проверки стайлгайда, их проще настроить, чтобы они ругались на отсутствие комментария, чем отключить эту проверку, не писать комментарии в «очевидных» местах и забыть что-то реально важное. На самом деле, хорошим тоном считается писать комментарии и к private-методам и полям, но это уже дело вкуса — такие комментарии полезны тем, кто сопровождает ваш код, а их по идее меньше, чем тех, кто им пользуется.

Есть некторые тактические соображения по поводу комментариев — если пользуетесь JavaDoc и подобными системами, не забывайте, что комментарии генерятся в два разных места — summary (который показывается в разных кратких списках, документации на пакет и т.д.) и details (который показывается только если открыта документация метода). Поэтому лучше писать «Returns something», чем «@returns something», второе сгенерится только в детальную информацию. Ещё бывает нелишне реально попробовать сгенерировать JavaDoc и посмотреть, что получится, многие узнают при этом много нового и интересного о том, как комментарии писать (например, что если комментарий к параметру начинать с заглавной, то получится некрасиво).

Ещё есть некоторые проблемы со стайлгайдом. Оформление программы — это то, в общем-то, что отличает нуба от матёрого программиста. Когда вы приходите на собеседование и вам дают написать на листочке задачу, задумайтесь — это же, скорее всего, задача уровня школьной районной олимпиады — действительно ли работодателю интересно, сможете ли вы её решить? Тогда они могли бы набрать школьников забесплатно, пообещав им опыт работы и трудоустройство по окончанию стажировки через 5 лет. Ну, людей, которые считают, что отступы не нужны, я в магистратуре не встречал, а вот людей, считающих, что один-два пропущенных или лишних пробела погоды не делают, я видел много. Причём, обычно такие проблемы даже не замечают, ну а что, программа-то компилится — но опытному программисту такой код режет глаз. Поскольку есть мнение, что то, что нельзя проверить автоматически, невозможно и соблюдать, есть много тулов, которые занимаются проверкой стайлгайда и иногда даже статическим анализом на предмет очевидных семантических ошибок. Примеры для Java — это `http://checkstyle.sourceforge.net/` или `https://www.codacy.com/` (вторая штука умеет не только Java и вообще весьма годна, рекомендую глянуть), для .NET — `https://github.com/StyleCop`, они наиболее полезны при наличии Continuous Integration, но и в локальном билде их неплохо бы запускать и делать так, чтобы они не ругались. Кажется, что это ерунда, но на самом деле, я думаю, процентов 70 ошибок в программах — глупые, их легко поймать простым статическим анализом и даже просто соблюдением правил стайлгайдов (которые не только для красоты, но ещё и предотвращают типовые проблемы).

2. Grep

Следующая задача — реализовать в рамках существующего CLI (ну, у большинства — пока потенциально существующего) команду grep, которая бы искала подстроки в файле или входном потоке. Весь синтаксис grep-а поддерживать не надо, надо уметь принимать регулярные выражения (так, как они реализованы в вашем любимом языке программирования, писать свой парсер регэкспов не нужно) и поддерживать ключи:

- *-i* — нечувствительность к регистру;
- *-w* — поиск только слов целиком;
- *-A n* — распечатать n строк после строки с совпадением.

Вот некоторое количество примеров вызова того, что должно получиться:

```
> grep plugin build.gradle
    apply plugin: 'java'
```

```
    apply plugin: 'idea'
> cat build.gradle | grep plugin
    apply plugin: 'java'
    apply plugin: 'idea'
> grep -A 2 plugin build.gradle
    apply plugin: 'java'
    apply plugin: 'idea'
    group = 'ru.example'
    version = '1.0'
```

С архитектурной точки зрения эта задача интересна тем, что писать разбор аргументов командной строки вручную — это дело муторное и, кажется, широко распространённое (существуют тысячи консольных утилит, они ведь должны как-то парсить свои аргументы). Поэтому, как обычно, перед тем, как кидаться что-то кодить, надо поискать уже готовые опенсорсные решения. Собственно, это одна из самых важных задач архитектора — во-первых, чувствовать, где можно применить третьесторонние компоненты, а где проще реализовать самим, во-вторых, уметь совершать обоснованный выбор третьесторонней компоненты среди аналогов, в-третьих, знать кучу всего, чтобы иметь возможность сказать «ага, а вот есть такая штука, она делает то, что нам надо! (и у неё наверняка есть аналоги, потому что я слышал о ней уже три года назад, она наверняка протухла)».

Каких-то неочевидных практик поиска третьесторонних компонент, наверное, не существует, вот немножко очевидных рекомендаций, на всякий случай.

- Миллион леммингов не может ошибаться, поэтому у хорошей либы наверняка много пользователей (правда, тут следует брать поправку на год появления), быстрое гугление покажет, насколько активно сообщество на GitHub, StackOverflow и подобных местах.

- Даже если компонента очень хороша, маленькое сообщество приведёт к проблемам — негде будет найти предыдущий опыт, труднее будет получить совет, если что-то пойдёт не так.

- Если есть несколько фаворитов, имеет смысл потратить время на то, чтобы поковыряться с ними со всеми и выбрать то, что больше подходит или даже просто больше понравилось. Пробовать прежде всего стоит то, что кажется сложным, потому что часто бывает, что что-то простое делается просто, а сложное — вообще никак, и придётся отказаться от выбранной компоненты посреди процесса интеграции.

- Обращайте внимание на лицензию. Даже лицензия типа GPLv3 может быть showstopper-ом для использования идеальной во всех остальных отношениях компоненты. Ни один нормальный архитектор не интегрирует в свою систему компоненту, которая хотя бы потенциально может иметь проблемы с авторскими правами.

  – Поэтому если вы всё ещё выкладываете код на GitHub без явного указания лицензии, подумайте, достаточно ли добродетельный вы человек. Код без лицензии по большинству законов использовать может только автор, даже если он лежит в открытом доступе.

Хочется, чтобы на этом простом примере вы попробовали сделать обоснованный выбор библиотеки разбора параметров командной строки. Выбор может быть не очевиден, но сами по себе такие библиотеки не очень сложные, так что тут реально хорошо подходит тактика «попробовать все и выбрать то, что пришлось по душе больше». Тем не менее, мы «играем во взрослые проекты», поэтому хочется в домашнем задании текстовый документ с описанием того, какие библиотеки были рассмотрены, их достоинства и недостатки и почему была выбрана именно та библиотека, которую вы в итоге выберите. Аргумент «Я всю жизнь ей пользовался» считается в данном случае невалидным.

Задачу надо сдавать в отдельной ветке, отведённой от ветки CLI, дедлайна у неё нет, но помните, что все задачи надо обязательно сдать до зачёта.

А сейчас мы продолжим то, на чём остановились на прошлой паре — проектировать CLI. Может, даже устроим небольшое кодревью, если кто-то хочет. Есть время подумать, порисовать диаграммы и предложить варианты архитектуры системы (чем «архитектурнее», тем лучше, так что сегодня хочется дать слово не олимпиадным программистам, а архитектурным астронавтам), уже с учётом новых требований, после чего — холивор, то есть плодотворное обсуждение.