Практика 1: Вступление, задача про CLI

17.01.2022

1. Формальности

Занятия у нас раз в неделю, курс длится два модуля. В конце надо будет получить оценку за этот курс (которая станет частью итоговой оценки вместе с оценкой по соответствующему теоретическому курсу, который читает Тимофей Брыксин). На практике мы будем пытаться применять знания, полученные на теоретическом курсе, поэтому будет некоторое (довольно большое) количество домашек, как на проектирование, так и на реализацию. Домашние задачи будут оцениваться по десятибалльной шкале, с текущим и максимальным возможным баллом. Итоговая оценка за задание получается, когда текущий балл становится равным максимальному, при этом текущий балл может только расти, а максимальный — только уменьшаться. Каждая задача имеет дедлайн, пропуск которого ополовинивает максимальный балл. Также максимальный балл будет снижаться пропорционально нереализованной на момент дедлайна функциональности из условия (то есть сдать пустой проект можно, но максимальный балл всё равно будет ополовинен). Ещё, возможно, по ходу курса появится список «грубых ошибок», за допущение которых максимальный балл будет снижаться (например, выкладывание в git бинарников или что-то такое). Разумеется, про такие вещи будет объявлено отдельно.

Итоговый балл будет вычисляться по смещённой линейной шкале с разными весами для разных задач, система устроена так, чтобы промотивировать сделать вообще все домашки, да ещё и работать на парах.

Все презентации и условия заданий, а также конспекты тех пар, по которым есть конспекты, будут выкладываться в вики курса. В качестве основного средства связи у нас будет чат курса в Telegram, также можно писать личные сообщения в Telegram (https://t.me/yurii_litvinov) и на почту: yurii.litvinov@gmail.com. Сдавать решения надо будет, выкладывая их на GitHub и делая пуллреквест в свой же собственный репозиторий, я и мой помощник будем стараться этот пуллреквест комментить и выставлять текущий балл в отдельную таблицу на Google Docs, которую также можно найти в материалах курса. Мы не будем смотреть решения, пока вы явно не сообщите, что решение готово к проверке (в чате курса, чтобы это видел и я, и помощник), так что можете коммитить какие угодно промежуточные результаты (как раз если у вас всё решение одним коммитом, это вызовет подозрения). Когда пуллреквест принят (accepted, мерджить я сам не буду), задача считается зачтённой и дальше её править нет смысла.

В качестве языка и среды программирования можно использовать то, что вам больше нравится, курс всё-таки больше про архитектуру, а не про реализацию, только уж совсем эзотерических языков не надо. Мы должны иметь возможность хотя бы собрать и

запустить то, что у вас получилось. Если процесс сборки нетривиален или используются какие-нибудь редкие технологии (включая Haskell), следует задокументировать (а ещё лучше — полностью автоматизировать) процесс сборки, описав в README.md последовательность шагов от git clone до запуска и проверки работоспособности вашей программы. Лучше писать на C++/Java/C#/F#, больше шансов получить содержательные комментарии, относящиеся к технике кодирования.

Ещё обратите внимание, что просто сделать домашку может быть недостаточно. У нас тут курс по проектированию, поэтому решение, удовлетворяющее всем функциональным требованиям, вообще, строго говоря, никому не интересно. Важна архитектурная красота, обоснованность принятых решений, чистая и аккуратная реализация, наличие необходимых сопровождающих артефактов. Поскольку архитектура — это в каком-то смысле больше искусство, чем наука, чётких критериев оценки не будет, и придираться мы будем в основном к форме — качеству кода, наличию документации, соответствию формальным правилам.

Студентами от практики по архитектуре обычно ожидается много критики архитектуры, но тут проблема в том, что если вы достаточно владеете программированием и имеете хоть какой-то опыт, вы окажетесь вполне способны породить адекватное решение к учебным задачкам, так что, скорее всего, критика архитектуры решения будет в духе «ок, мне нравится, но можно было ещё вот так» или её не будет вообще. Это плохо, но делать нечего, реальные проблемы с архитектурой могут возникнуть только на реальных проектах. Кстати, если интересно и у кого-нибудь будут интересные НИР, можно устроить их архитектурный разбор. В целом же, тренировать мы будем прежде всего владение инструментами, которые позволяют создать и описать архитектуру.

2. Что будет в курсе

Собственно, практические занятия по проектированию ПО — затея, изначально обречённая на провал, невозможно научиться проектированию ПО, посещая практические занятия в университете и даже делая все домашки. Лучший способ научиться проектировать ПО — проектировать реальные проекты, причём достаточно большие. С этим может не сложиться, по крайней мере, поначалу, и уж тем более это нельзя впихнуть в формат практических занятий. Поэтому мы будем практиковаться на игрушечных примерах, но требования к ним будут предъявляться такие, будто это огромные промышленные проекты в сотни человеколет трудозатрат. Так что аргументы в духе «да я это на питоне в двести строчек нафигачу без всякой архитектуры-шмархитектуры» заранее признаются невалидными.

На занятиях примерно в половине случаев я буду что-то рассказывать (больше про практическую сторону дела, какие-то примеры из реальной практики, про инструменты и методы, а иногда и просто продолжать лекции), в половине случаев прямо на паре надо будет сделать что-нибудь небольшое. Будет несколько задач на кодирование (всего штуки три крупные задачи) и много мелких задач в духе «нарисовать диаграмму», «написать документ», «проанализировать». Рассказывать я буду про архитектурную документацию, будет довольно много и теории и практики про UML и другие визуальные языки, которые используются при разработке архитектуры, про реализационные аспекты паттернов проектирования (что такое, как устроено и зачем надо, расскажут на лекциях, а как это

писать и где можно прострелить себе ногу — на практике), про антипаттерны (на лекциях будут рассказывать, как писать надо, на практиках — как не надо), примеры различных архитектур, разные подходы к созданию архитектуры и вещи, которые не так важны, чтобы включать их в курс лекций и программу экзамена, но тем не менее достойные упоминания.

Однако даже к задачам на кодирование надо относиться как к архитектурным, у нас тут всё-таки не практика по Java — надо сначала продумать и описать архитектуру разрабатываемой системы, уже затем кидаться её кодить, и главное, при кодировании не забывать про архитектуру и её связь с кодом. Овердизайн и массированное применение знаний, полученных на лекциях, приветствуется (правда, до разумных пределов). При проверке от решений задач будет ожидаться:

1. работоспособность и соответствие требованиям (явным или неявным);
    (a) кстати, negotiation является важной частью работы архитектора, так что требования можно (и иногда нужно) уточнять, договариваться, возможно, убеждать препода, что тот или иной пункт условия ему на самом деле не нужен и т.д., но, опять-таки, до разумных пределов;
2. наличие архитектурной документации;
    (a) её форма и количество будут меняться по ходу курса, от README и комментариев к коду в начале до UML-диаграмм и формального design document-а в конце;
    (b) комментарии к каждому классу, интерфейсу и public-методу, тем не менее, всегда будут обязательны;
    (c) краткое описание деталей реализации в README — тоже;
3. следование стайлгайдам и общепринятым правилам здравого смысла в программировании;
4. наличие юнит-тестов, покрывающих все требования условия (там, где это возможно);
5. максимально возможной кроссплатформенности и переносимости кода (писать ради этого на ANSI C не надо, но писать на Java программы, на ровном месте не работающие под Windows просто потому, что вы забыли об её существовании — тоже);
    (a) проверяться задача может как под Linux, так и под Windows, и если у неё нет веских причин не работать под одной из этих операционных систем, а она не работает, могут попросить исправить;
6. применение индустриальных практик, общепринятых при разработке production-кода: логирование, Continuous Integration, разумные стратегии обработки исключений.

Обратите внимание, что многое из того, что я буду рассказывать, и главное, требовать от решений, может показаться оверкиллом — например, комментарии к **каждому** public-методу, о боже, что? Или рисование пачки диаграмм для программы в 500 строк на

Питоне. Это как раз связано с тем, про что я уже говорил — мы считаем, что речь идёт о больших реальных проектах, где это всё не выпендрёж, а единственный способ не развалиться от собственной сложности. Часто возражают, что в реальных проектах так тоже не делают — я слышал от кого-то из магистров, что в Яндексе сколько-то гигабайт кода и ни одной UML-диаграммы — охотно верю, ~~поэтому у нас всё так плохо~~ но на этих парах мы учимся техникам, принципам и инструментам, так что некоторые требования действительно искусственны. Исключительно с учебными целями. Можно этого всего не делать и просто писать код, но тогда всю жизнь будете junior-ами, которым говорят, какой код писать.

Обратите внимание, часто курсами по архитектуре называют продолжение практик по программированию, иногда — курсы по разработке веб-приложений, иногда — курсы по конкретным технологиям. Мы затронем все эти вопросы, но это курс по архитектуре. Так что у нас не будет обучения деплою веб-сервисов на .NET на Amazon Web Services, хотя и про веб-сервисы мы поговорим, и про AWS, и даже будет немного примеров на .NET (обычно в Вышке учат C++ и Java/Kotlin, а расширять кругозор надо). Считается, что к началу этого курса вы уже достаточно хорошо владеете парой объектно-ориентированных языков, знакомы и с функциональной парадигмой, не испытываете проблем с системами контроля версий и настройкой CI (и даже понимаете, зачем CI для Питона), умеете писать юнит-тесты, понимаете хотя бы концептуально, как работают сети.

3. Задача про CLI

Начнём мы с задачи на попроектировать, пока как умеете. Задача такая: спроектировать простой интерпретатор командной строки, поддерживающий следующие команды:

- cat [FILE] — вывести на экран содержимое файла;
- echo — вывести на экран свой аргумент (или аргументы);
- wc [FILE] — вывести количество строк, слов и байт в файле;
- pwd — распечатать текущую директорию;
- exit — выйти из интерпретатора.

Кроме того, должны поддерживаться одинарные и двойные кавычки (full and weak quoting, то есть одинарные кавычки передают текст как есть, двойные выполняют подстановки переменных окружения с оператором $), собственно окружение (команды вида «имя=значение»), оператор $, вызов внешней программы через Process (или его аналоги) для любой команды, которую интерпретатор не знает. Должны ещё поддерживаться пайплайны (оператор «|»), то есть перенаправление ввода и вывода. Примеры:

```
> echo "Hello, world!"
Hello, world!

> FILE=example.txt
> cat $FILE
```

```
Some example text

> cat example.txt | wc
1 3 18

> echo 123 | wc
1 1 3

> x=ex
> y=it
> $x$y
```

Тимофей ещё не рассказывал, наверное, про Architectural drivers (честно как-то не встречал адекватного русского перевода, так что будем обходиться английским термином). Пока что — это основные факторы, определяющие архитектуру системы. Поскольку мы пишем объектно-ориентированный код (курс в основном про эту парадигму, хотя в принципе всё это верно для любых больших приложений), мы будем исходить из обычных для объектно-ориентированных программ качеств — сопровождаемость, расширяемость, переиспользуемость и т.д. В нашем случае это значит, в частности, что проектировать систему надо так, чтобы новые команды было добавлять легко (логично, что шелл будет постепенно расширяться новыми встроенными командами), но желательно поддерживать архитектуру достаточно простой и слабо связанной, чтобы можно было реализовать и другие требования, которые могут возникать по ходу. Может потребоваться внезапно реализовать ещё что-нибудь из того, что умеют обычные шеллы, как и в реальной жизни, желания заказчика непредсказуемы (поэтому, кстати, не надо пытаться их предугадать и заложить в архитектуру — то, о чём вы подумали, никогда не случится, случится то, о чём вы не подумали).

Собственно эта задача станет и домашней, и начать её делать надо в аудитории. Итоговым результатом должен стать документ, описывающий основные архитектурные решения, достаточно подробно, чтобы в процессе кодирования не надо было задумываться и о чём «архитектурном». Должна быть структурная диаграмма, как умеете: умеете UML — рисуйте диаграмму классов, не умеете — сойдёт структурная схема из соединённых стрелочками прямоугольников, лишь бы по ней было понятно, где что, и что примерно делает. Должно быть также и текстовое описание, поясняющее происходящее на диаграмме (примерно две-три страницы текста, описывающие каждую сущность на диаграмме и как они взаимодействуют). Обратите внимание, что код писать пока не надо — через неделю будет отдельное задание это реализовать (потом при проверке мы посмотрим, насколько реализация соответствует архитектуре).

На паре надо, во-первых, поделиться на команды примерно по три человека (только на эту задачу, у нас будут и другие командные задачи, их можно будет в другом составе делать). Во-вторых, проанализировать условие, выявить неоднозначности и вообще места, где условие недостаточно подробно. Может быть, посмотреть, как работают реальные шеллы. Дальше надо выполнить декомпозицию системы на компоненты, классы и основные методы, нарисовать первое приближение диаграммы (только не увлекайтесь, сделать её достаточно детальной даже в команде из трёх человек за пару не успеть), быть готовыми выйти в конце пары и рассказать решение. Текстовое описание пока не надо. В процессе

анализа задавайте вопросы по условию, оно намеренно неподробно. Надо постараться за пару сделать так, чтобы все примерно представляли, как стали бы это писать (потому что потом реально надо будет это писать). Кстати, в какой-то момент в следующих домашках надо будет что-то реализовать в чужой архитектуре (не вашей команды), поэтому постарайтесь не халявить при проектировании.

Дома надо будет по результатам обсуждения уточнить архитектуру, дорисовать диаграмму и написать текстовое описание. На что обратить внимание:

- выполняйте проектирование сверху вниз — сначала определитесь с общей структурой системы, определитесь с компонентами, их ответственностью и связями между ними, и только после этого переходите к проектированию компонентов;
- не закапывайтесь в деталях — задача намеренно такая, что можно всю пару обсуждать только вопросы парсинга и подстановки или особенности поведения wc; если закопаетесь — не достигнете цели (в реальной жизни это называется «архитектурный паралич» и случается довольно часто);
- кое-какие детали всё-таки надо продумать:
  - как представляются команды и пайпы из команд,
  - как и кем команды создаются,
  - как и кем они исполняются,
  - как происходит ввод-вывод в пайпе, что с потоком ошибок и кодом возврата,
  - кто и как выполняет разбор входной строки,
  - кто и как выполняет подстановки (тут особо аккуратно, недостаточно подробное описание стратегии подстановки может дать вам что-то вроде машины Маркова),
  - как представляются и кому передаются переменные окружения,
  - что с многопоточностью (тут тоже лучше осторожно, потому что потом это реализовывать придётся — однопоточное/однопроцессное решение вполне ок, хоть и отличается от реальных шеллов).