Факультет программной инженерии и компьютерной техники

Рефакторинг баз данных и приложений

Лабораторная работа №1

Вариант: https://github.com/it-pechenushka/ComMat-Lab1

Преподаватель: Логинов Иван Павлович

Выполнил: Кульбако Артемий Юрьевич P34112

Санкт-Петербург
2021

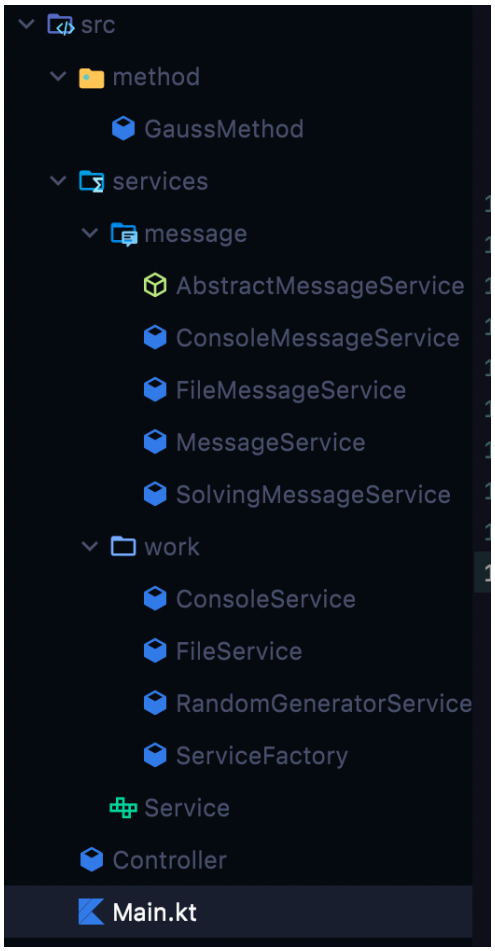
Задание

Провести минимум 3 рефакторинг для выбранного репозитория.

Описание проведённого рефакторинга

Так как исходный код был написан на Java, а выполнять задание интереснее да качественнее можно на Kotlin, первым делом проект был полностью переписан, но с сохранением файловой, пакетной и классовых иерархий. Далее был проведён рефакторинг согласно нижеизложенному плану.

Анализ оригинальной архитектуры



Теперь попытаемся понять цель и архитектуру приложения. Это программа, которая решает систему уравнений методом Гаусса, написана в рамка предмета Вычислительная математика в 4 семестре моим одногруппником и разбирается с его разрешения.

Main - только запускает приложение, делегируя всю работу...

Controller - выполняет инициализацию объектов взаимодействия с пользователем и передает управление...

Service - интерфейс с одной функцией doing(), которая запускает работу сервиса.

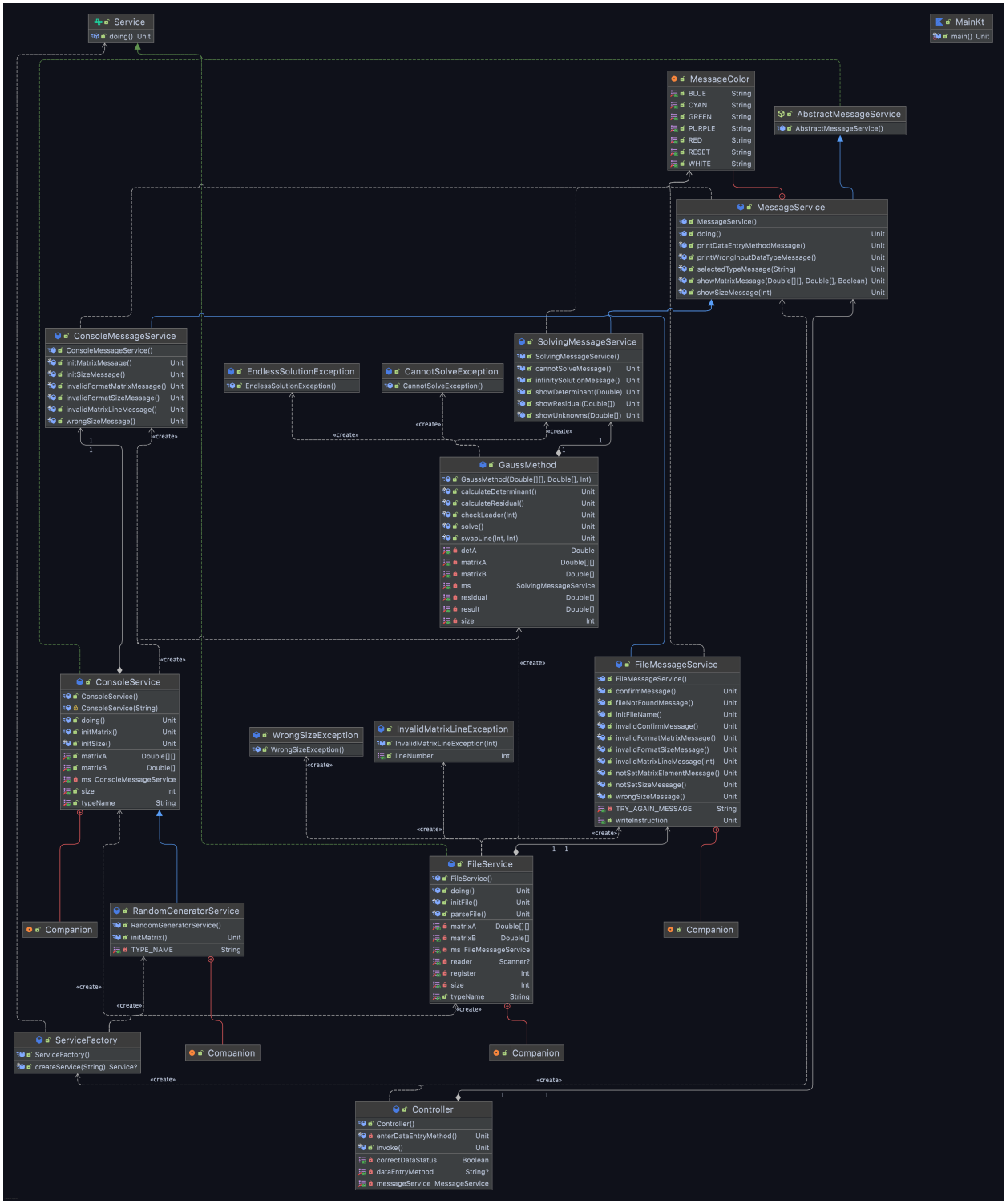
ServiceFactory - фабрика (на самом деле нет, но к этому вернёмся позже), которая на основе пользовательского ввода создаёт объект (один из классов пакета `services.work`), который будет дальше работать с матрицами **и** взаимодействовать с пользователя. Это не соответсвует принципу единственной обязанности класса, подобным грешат все классы из пакета `services.work`, что собственно и объяснение его слишком абстрактное название без привязки к конкретной области работы.

ConsoleService, FileService, RandomGeneratorService - отвечают за разные виды работы программы. Может показаться, что названия сервисов описывают их области обязанностей и возможностей, но на самом деле, все они строго завязаны на взаимодействии с консолью, о чем будет рассказано далее.

Классы из `services.message` - должны (исходя из названия) отвечать за формирование сообщений. По факту, они не только формируют, но и отправляют их. Выбрать поставщика и отправителя данных невозможно, все классы строго заточены под работу с консолью, хотя только один из них называется ConsoleMessageService. AbstractMessageService - абсолютно пустой класс, просто реализующий Service, от которого наследуются другие сервисы.

GaussMethod - класс, который занимается математикой - основной деятельностью программы. Принимает в конструкторе `matrixA` - матрицу элементов уравнений системы, `matrixB` - вектор решений системы (название выбрано неудачное, я долго

думал, что ему нужно скормить именно ещё одну матрицу) и `size` - размер матрицы (непонятно зачем этот параметр нужен, это ж не C/C++, размер спокойно можно получить внутри конструктора самостоятельно, а не обязывать пользователя класса делать это). С этим классом связано больше всего проблем.



Изменения

Первым делом избавимся от `AbstractMessageService` - никакой общей для всех потомков логики в нём нет. Интерфейсы в Java были придуманы как более понятный и структурированный ответ множественному наследованию. Другие

сервисы могут самостоятельно реализовать `Service`, к тому же, это сделает их более гибкими и расширяемыми, поэтому класс удаляется полностью (было, стало).

`ServiceFactory` также идёт утиль. Дело в том, что это и не паттерн фабрика вовсе (он должен генерировать семейства объектов по заданному признаку, а не просто быть обёрткой над оператором switch), а использование строк как api для создания объектов просто неуниверсальная идея, должно быть enum. Вся логика переезжает в единственное место его использования - в `Controller` (было, стало).

Займёмся классом `GaussMethod` (класс содержит в названии слово "метод", иронично). Я считаю, что использование класса для одноразового проведения математической операции идея крайне неудачная. Математика хорошо дружит с функциональным программированием, а поэтому всю математическую часть программы стоит переписать в виде набора функций, в идеале, без состояния, чтобы можно было безопасно их параллелить (что часто необходимо в научных расчётах). Я попытался сделать все функции статическими и независимыми друг от друга. Теперь можно переиспользовать некоторые "общие" математические функции, такие как нахождение детерминанта и взаимное перемещение векторов в матрице. К тому же, метод `solveByGauss()` теперь не модифицирует исходные данные, а копирует их и работает с копиями. Ещё одна проблема этого класса - получение результатов вычисления. Массив ответов закрыт модификатором private, геттера нет. Зато есть очень странная схема: класс почему-то хранит внутри себя экземпляр `SolvingMessageService`, который выводит результат, при этом заменить этот объект или как-нибудь получить к нему доступ нельзя. Я, как пользователь, хочу отдать систему нелинейных уравнений в метод, который умеет её решать, получить результат, а дальше сам решать, что с ним делать. В текущей реализации, результат решения улетает в трубу - в какой-то "левый" сервис. Здесь как бы "поставщик" данных (`GaussMethod`) главенствует над "потребителем". В итоге, мне пришлось фактически вывернуть этот класс наизнанку (было, стало). В будущем, этот метод можно ещё больше раздробить.

`SolvingMessageService` пытается быть универсальным классом, красиво преобразующим матрицы в `String`, вот только все его методы содержат вызовы `println`, а значит универсальностью тут и не пахнет - он по сути ничем не отличается от класса `ConsoleMessageService`. Во-первых, стоит сделать так, чтобы он просто возвращал строки (было, стало). А далее нужно определить, где всё же будут храниться эти функции. Здесь есть 2 пути:

A. Оставить всё как есть, т.е. отдельный класс с функциями преобразования матриц в строки.

B. Преобразованием будет заниматься `toString()` структуры, созданной для матрицы: это могло происходить в классе `GaussMethod` старого проекта.

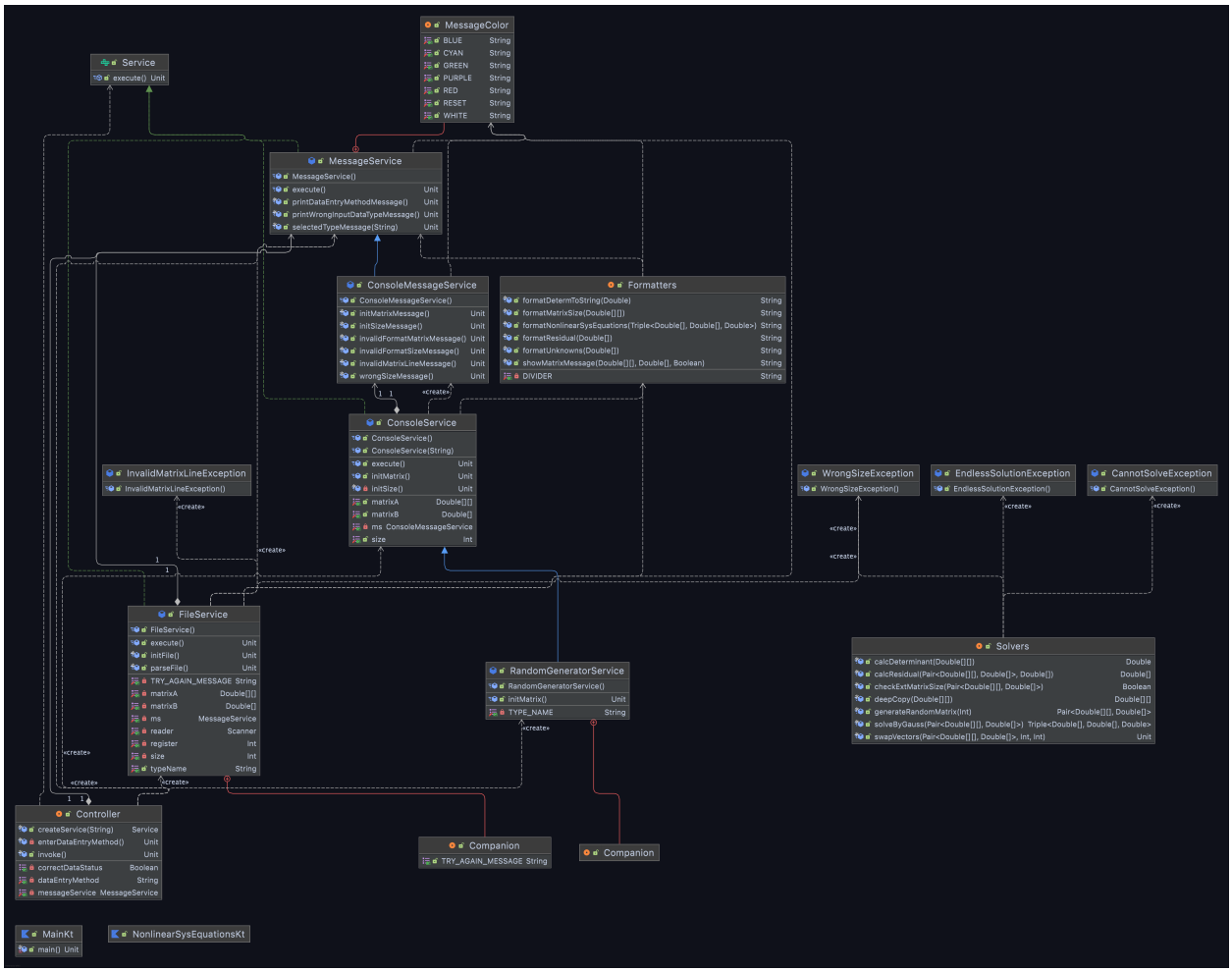
Так как подход B уже невозможен, ввиду превращения класса в набор функций, будет использоваться подход A, разве что класс будет переименован в `Formatters` и помещён в пакет `nonlinear_sys_equations`, как и функции из бывшего `GaussMethod`.

Не забудем переименовать `matrixA` и `matrixB` в более понятные `matrix` и solutionsVector.

В нескольких местах используется отлов самодельных исключений, а потом вывод сообщения соответствующего исключению. Это сообщение можно напрямую передать в конструктор класс `Exception(msg: String)` при наследовании, что я и сделал для всех исключений.

`getInstruction()` - геттер, который не возвращает что-то, а печатает в консоль. Сначала был переименован в `writeInstruction()`, а позже я от него отказался перенеся вывод в единственно возможное место вызова.

`RandomGeneratorService` - генерирует матрицу и далее передаёт управление контроллеру. Реализация очень странная: сгенерированная система уравнений передаётся вверх по дереву наследования, получить к ней доступ никак нельзя. Это значит, что я не могу просто получить систему для каких-то иных своих нужд. Метод генерации был вынесен в отдельную функцию (было, стало) в тот же пакет, где и другие математические функции. Хотелось бы также, чтобы этот класс не наследовался от `ConsoleService`, но это потребовало бы кардинального изменения архитектуры, а то и переписывания приложения к нуля.



Вывод

В результате лабораторной работы были проведены рефакторинги большой класс, группы данных, стрельба дробью, дублируемый код, теоретическая общность по Фаулере. Результатом я не доволен - код стал чище и понятнее, но далеко ему хорошего состояния. Я считаю, что это тот случай, когда проще и быстрее переписать всё с нуля, чем пытаться реанимировать то что есть. Свою задачу он выполняет отлично, но маловероятно, что его получится переиспользовать или расширить без полной архитектурной перепланировки.

Финальный код: