Архітектура програмного забезпечення

Лабораторна робота №2

Тема: Безперервна інтеграція та автоматизація тестування
Мета: Реалізація практики безперервної інтеграції, закріплення навичок використання патерна ін'єкції залежностей для спрощення тестування

Безперервна інтеграція досягається за рахунок наступних 3-х аспектів:

- підтримки історії версій вашого коду,
- автоматизації збірки,
- автоматизації тестування.

В контексті лабораторних робіт з курсу «Архітектури програмного забезпечення» ви продовжуєте працювати з системою контролю версій Git, чим забезпечуєте підтримку історії вашого коду – першого з аспектів. У даній роботі ми торкнемося реалізації двох інших аспектів, а також організації процесу безперервної інтеграції в команді розробників.

За даним посиланням ви знайдете шаблонний репозиторій для організації своєї роботи: https://github.com/roman-mazur/architecture-lab-2

Завдання 1. Автоматизація тестування

У даному завданні вам необхідно реалізувати та протестувати функцію роботи з префіксною або постфіксною формою математичних виразів (їх також називають польською та зворотною польською нотацією відповідно). Відповідно до свого варіанта, вам потрібно реалізувати функцію, яка або обчислює вхідний вираз, або перетворює його в інфіксну форму (звичний математичний запис) чи Lisp-подібну нотацію (Lisp - це функціональна мова програмування). Нижче наведено приклади одного й того ж виразу в різних формах:

Форма	Приклад
Інфіксна	`5 + (4 - 2) * (3 ^ 2)`
Префіксна	`+ 5 * - 4 2 ^ 3 2`
Постфіксна	`4 2 - 3 2 ^ * 5 +`
Lisp-подібна	`(+ 5 (* (- 4 2) (pow 3 2)))`

Математичні операції, які вам потрібно підтримувати включають **додавання** (+), **віднімання** (-), **множення** (*), **ділення** (/), **піднесення до степеня** (^). List-подібна форма є дуже подібною до префіксної, за винятком використання дужок та використання *pow* замість символу "^" у випадку піднесення до степеня, що робить цю форму синтаксично коректним висловлюванням мови програмування Lisp.

У файлі *implementation.go* вам потрібно реалізувати функцію обчислення чи конвертації відповідно до варіанта. У файлі *implementation_test.go* – реалізувати тести своєї функції на кількох прикладах: тести мають включати приклади обчислень для простих (з 2-3 операндами) та складніших (7-10 операндів) виразів та перевірку поведінки функції при опрацюванні неправильних даних (пустий рядок чи недопустимі символи).
Тести запускаються за допомогою команди *go test*.

Для написання перевірок результатів виклику функцій вам потрібно використовувати або пакет testify, або лише стандартну бібліотеку Go.

Варіанти

№	Функція	Бібліотека для перевірок
1	Обчислити постфіксний вираз	testify
2		стандартна бібліотека Go
3	Обчислити префіксний вираз	testify
4		стандартна бібліотека Go
5	Перетворити постфіксний вираз у Lisp-подібний	testify
6		стандартна бібліотека Go
7	Перетворити постфіксний вираз в інфіксний	testify
8		стандартна бібліотека Go
9	Перетворити префіксний вираз у Lisp-подібний	testify
10		стандартна бібліотека Go
11	Перетворити префіксний вираз в інфіксний	testify
12		стандартна бібліотека Go

Свій варіант ви можете визначити за допомогою команди в Slack:

```
/team-variant 2
```

У файлі *implementation_test.go* також має бути функція Example<Ім'яВашоїФункції>, яка ілюструє використання вашої реалізації. Даний приклад, який можна запустити і як тест, додається в документацію вашого пакета.

Спробуйте запустити команду *godoc* та відкрити браузер за адресою http://localhost:6060 – знайдіть свій пакет у списку та перевірте як ваші коментарі для функції та приклад-тест додано до документації.

Завдання 2. Заглушки в тестах через ін'єкцію залежностей

У файлі *cmd/example/main.go* вам потрібно реалізувати отримання виразу з параметрів командного рядка, виклик своєї функції з варіанта та відображення результатів (або виведення інформації про помилку у вхідних даних). Читання вхідного вираза має підтримувати передачу самого вираза в командному рядку,

```
-e "5 5 +"
```

та вказання файла з виразом.

```
-f my-input.txt
```

Вказання обох опцій має розглядатися як помилка користувача.

Також має підтримуватися опціональне (не обов'язкове) визначення файла, куди потрібно записати результат перетворення.

```
-o result.txt
```

Ящо файл для результа не вказано, використовуєтьсся стандартний потік виведення (*stdout*).

Запускати свою команду ви можете, наприклад, за допомогою

```
go run ./cmd/example –e “42 1 –” –o result.txt
go run ./cmd/example –f input.txt
```

з кореня проекта.

Для роботи з аргументами командного рядка використайте пакет flag.

Інформація про помилки (неправильна комбінація аргументів командного рядка або помилка в синтаксисі вхідного виразу) мають виводитися в стандартний потік помилок (*stderr*). У випадку помилку синтаксису вхідного виразу, ваша програма має завершитися з кодом 1.

У вашому файлі *main.go* не повинно бути прямого використання функції з *implementation.go*. Натомість, ви маєте створити екземпляр *ComputeHandler*, в його полях вказавши, звідки читати вираз та куди записати перетворення (створюючи відповідні екземпляри *io.Reader* та *io.Writer* - це, по суті, і є ін’єкцією залежностей компонента *ComputeHandler*).

Далі, ви маєте викликати метод *Compute()*, який прочитає дані, викличе функцію з *implementation.go* та запише результат. У випадку помилки синтаксису даний метод має її повернути, і код в *main.go* повинен її обробити (заврешуючи програму з кодом 1 та відправляючи деталі помилки в stderr).

У вас також має з’явитися файл *handler_test.go*, у якому ви маєте реалізувати тести для метода *Compute* структури ComputeHandler.
У цих тестах ви маєте перевірити, що

- в *Output* записується результат обчислення, який відповідає *Input*,
- повертається помилка при проблемі з синтаксисом даних в *Input*.

При тестуванні не повинно відбуватися взаємодії з файловою системою.
Вам не потрібно докладно перевіряти правильність обчислень, оскільки це вже зроблено на рівні окремої функції в 1-ому завданні.

Підказка
Зверніть увагу на те, що метод *strings.NewReader* створює екземпляр *io.Reader*, *os.File* імплементує і *io.Reader*, і *io.Writer*, а *os.StdOut* та *os.StdErr* є змінними типу **os.File*.

Завдання 3. Використання сервера безперервної інтеграції

Зверніть увагу, що в шаблоні вашого прожекта присутній файл *Makefile*, який має визначення для кількох задач збірок (запуску тестів, компіляції фінального бінарника) на Unix системах. У даній роботі вам потрібно налаштувати автоматичний запуск задач збірки для вашого прожекта на віддаленому сервері за допомогою Github Actions.

Для цього вам необхідно буде створити файл *.github/workflows/build.yml* у своєму проекті, у ньому описати запуск задач з *Makefle* та налаштувати їхній запуск на GitHub (докладніше читайте документацію за посиланням вище).

У результаті, для кожного нового коміта, який з’являється в github репозиторії, виділений сервер має компілювати вашу програму та запускати тести (через виклики *make*). У випадку помилки компіляції чи непроходження тестів, збірка має помічатися як “завалена” (FAILED).

В історії збірок проекту мають бути приклади успішної та неуспішної збірки.

У проекті також має бути принаймні один Pull Request, створений на GitHub, для якого було запущено збірку на Github Actions.

Посилання на приклади збірки розмістіть у readme вашого репозиторія.

Оцінювання

Для оцінки роботи, відправте її через команду у Slack звичним чином:
`/submit 2 <github url>`

Максимальный бал з виконання роботи – 8 (по 3 бала за перші два завдання та 2 бала за останнє).

Корисні ресурси

- Стаття про приклади використання як тести
- Використання модулів у Go
- Make для Windows
- Швидкий вступ у Github Actions
- Різноманітні приклади тестів на Go