Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Космических и информационных технологий
институт
Кафедра «Информатика»
кафедра

ОТЧЕТ О ПРАКТИЧЕСКОЙ РАБОТЕ №4

Контролируемая среда тестирования
тема

Вариант 1

Преподаватель _____ А.С. Кузнецов
подпись, дата

Студент _____ В.А. Прекель
подпись, дата

Красноярск 2021

1 Цель работы

Изучить возможности создания контролируемой среды тестирования программного обеспечения, проведения регрессионного тестирования.

2 Общая постановка задачи

Научиться:

1. Пользоваться подставными объектами;

2. Провести регрессионное тестирование.

3 Ход работы

Для работы был выбран язык OCaml, библиотеки для написания тестов ppx_inline_test, ppx_expect и ручное создание подставных объектов используя техники парадигмы модульного программирования языка OCaml. Будет использоваться код из предыдущих работ. Реализует конечный автомат калькулятора, обобщённого для работы с числами с плавающей запятой, целыми числами, векторами трёх длин. Для этого в языке OCaml для этого можно написать функтор, принимающий модуль такого типа, который содержит в себе типы для значений и операций и функцию применения операции, и возвращает этот функтор модуль конечного автомата у которого есть типы для состояния, действия и функции применения действия.

Тип модуля, который содержит в себе типы для значений и операций и функцию применения операции:

Листинг 1 – фрагмент файла lib/calcs.ml

```
module type Calcs = sig
  type num [@@deriving sexp, equal]
  type op [@@deriving sexp, equal]

  val calculate : op -> num -> num -> num option
end
```

Функтор:

Листинг 2 – фрагмент файла lib/calc.ml

```
module MakeStateMachine (Calcs : Calcs) = struct
  type num = Calcs.num [@@deriving sexp, equal]
  type op = Calcs.op [@@deriving sexp, equal]
```

```
type state =
  | WaitInitial
  | WaitOperation of { acc : num }
  | WaitArgument of
      { acc : num
      ; op : op
      }
  | Calculation of
      { acc : num
      ; op : op
      ; arg : num
      }
  | ErrorState of state
  | ErrorInput of state * Error.t
  | ErrorOperation of state
  | Finish of num
[@@deriving sexp, equal]

type action =
  | Num of num
  | Op of op
  | Empty
  | Invalid of Error.t
  | Calculate
  | Back
  | Reset
[@@deriving sexp, equal]

let initial = WaitInitial

let result = function
  | Finish result -> Some result
  | _ -> None
;;

let update ~action state =
  match state, action with
  | _, Reset -> WaitInitial
  | WaitInitial, Num a -> WaitOperation { acc = a }
  | WaitInitial, (Op _ | Empty | Calculate) -> ErrorState state
  | WaitInitial, Back -> WaitInitial
  | WaitOperation _, (Num _ | Calculate) -> ErrorState state
  | WaitOperation { acc }, Op op -> WaitArgument { acc; op }
  | WaitOperation { acc; _ }, Empty -> Finish acc
  | WaitOperation _, Back -> WaitInitial
  | WaitArgument { acc; op }, Num arg -> Calculation { acc; op; arg }
  | WaitArgument _, (Op _ | Empty | Calculate) -> ErrorState state
  | WaitArgument { acc; _ }, Back -> WaitOperation { acc }
  | Calculation _, (Empty | Num _ | Op _) -> ErrorState state
  | Calculation { acc; op; _ }, Back -> WaitArgument { acc; op }
  | Calculation { acc; op; arg }, Calculate ->
    begin
```

```
        match Calcs.calculate op acc arg with
        | Some acc -> WaitOperation { acc }
        | None -> ErrorOperation state
      end
    | Finish _, _ -> state
    | ErrorInput (old_state, _), Invalid err -> ErrorInput (old_state, err)
    | (ErrorState old_state | ErrorInput (old_state, _) | ErrorOperation old_state), Back
      -> old_state
    | (ErrorState _ | ErrorInput _ | ErrorOperation _), _ -> state
    | _, Invalid err -> ErrorInput (state, err)
  ;;
end
```

Благодаря этому можно внедрять как передаваемый модуль, модули с дополнительной функциональностью, например запоминание передаваемых значений в функцию calculate.

Листинг 3 – фрагмент файла test/test_calc.ml

```
module MakeHistoryCalcsMock (Calcs : Calcs) = struct
  include Calcs

  let history = Stack.create ()

  let calculate op acc arg =
    Stack.push history (op, acc, arg);
    Calcs.calculate op acc arg
  ;;
end
```

Или так, чтобы для определённых входных значений возвращались определённые выходные значения:

Листинг 4 – фрагмент файла test/test_calc.ml

```
module FloatCalcsStub = struct
  include CalcsFloat

  let calculate op acc arg =
    match op, acc, arg with
    | `Add, 1., 2. -> Some 0.
    | _ -> assert false
  ;;
end
```

С помощью этого можно тестировать историю передаваемых значений или тестировать на нужных значениях (т.е. в контролируемой среде).

Листинг 5 – файл test/test_calc.ml

```
open Core
```

```
open Lab_calculator
open Calcs

module MakePrintAndUpdate (CalcStateMachine : Calc.S) = struct
  let print_and_update action state =
    print_s [%sexp (state : CalcStateMachine.state)];
    CalcStateMachine.update ~action state
  ;;
end

let%test_module "test with mock" =
  (module struct
    module MakeHistoryCalcsMock (Calcs : Calcs) = struct
      include Calcs

      let history = Stack.create ()

      let calculate op acc arg =
        Stack.push history (op, acc, arg);
        Calcs.calculate op acc arg
      ;;
    end

    module CalcsMock = MakeHistoryCalcsMock (CalcsInt)
    module MockStateMachine = Calc.MakeStateMachine (CalcsMock)
    open MakePrintAndUpdate (MockStateMachine)

    let%expect_test "test1" =
      Stack.clear CalcsMock.history;
      let final =
        MockStateMachine.initial
        |> print_and_update (MockStateMachine.Num 1)
        |> print_and_update (MockStateMachine.Op `Add)
        |> print_and_update (MockStateMachine.Num 40)
        |> print_and_update MockStateMachine.Calculate
        |> print_and_update MockStateMachine.Empty
      in
      [%expect
        {|
        WaitInitial
        (WaitOperation (acc 1))
        (WaitArgument (acc 1) (op Add))
        (Calculation (acc 1) (op Add) (arg 40))
        (WaitOperation (acc 41)) |}];
      print_s [%sexp (final : MockStateMachine.state)];
      [%expect {| (Finish 41) |}];
      print_s [%sexp (MockStateMachine.result final : int option)];
      [%expect {| (41) |}];
      print_s [%sexp (CalcsMock.history : (CalcsInt.op * int * int) Stack.t)];
      [%expect {| ((Add 1 40)) |}]
    ;;

    let%expect_test "test1" =
```

```
    Stack.clear CalcsMock.history;
    let final =
      MockStateMachine.initial
      |> print_and_update (MockStateMachine.Num 1)
      |> print_and_update (MockStateMachine.Op `Add)
      |> print_and_update (MockStateMachine.Op `Add)
      |> print_and_update (MockStateMachine.Num 40)
      |> print_and_update MockStateMachine.Calculate
      |> print_and_update MockStateMachine.Empty
    in
    [%expect
      {|
      WaitInitial
      (WaitOperation (acc 1))
      (WaitArgument (acc 1) (op Add))
      (ErrorState (WaitArgument (acc 1) (op Add)))
      (ErrorState (WaitArgument (acc 1) (op Add)))
      (ErrorState (WaitArgument (acc 1) (op Add))) |}];
    print_s [%sexp (final : MockStateMachine.state)];
    [%expect {| (ErrorState (WaitArgument (acc 1) (op Add))) |}];
    print_s [%sexp (MockStateMachine.result final : int option)];
    [%expect {| () |}];
    print_s [%sexp (CalcsMock.history : (CalcsInt.op * int * int) Stack.t)];
    [%expect {| () |}]
  ;;

  let%expect_test "test1" =
    Stack.clear CalcsMock.history;
    let final =
      MockStateMachine.initial
      |> print_and_update (MockStateMachine.Num 2)
      |> print_and_update (MockStateMachine.Op `Mult)
      |> print_and_update (MockStateMachine.Num (-40))
      |> print_and_update MockStateMachine.Calculate
      |> print_and_update MockStateMachine.Empty
    in
    [%expect
      {|
      WaitInitial
      (WaitOperation (acc 2))
      (WaitArgument (acc 2) (op Mult))
      (Calculation (acc 2) (op Mult) (arg -40))
      (WaitOperation (acc -80)) |}];
    print_s [%sexp (final : MockStateMachine.state)];
    [%expect {| (Finish -80) |}];
    print_s [%sexp (MockStateMachine.result final : int option)];
    [%expect {| (-80) |}];
    print_s [%sexp (CalcsMock.history : (CalcsInt.op * int * int) Stack.t)];
    [%expect {| ((Mult 2 -40)) |}]
  ;;
end)
;;
```

```
let%test_module "stub test" =
  (module struct
    module FloatCalcsStub = struct
      include CalcsFloat

      let calculate op acc arg =
        match op, acc, arg with
        | `Add, 1., 2. -> Some 0.
        | _ -> assert false
      ;;
    end

    module StubStateMachine = Calc.MakeStateMachine (FloatCalcsStub)
    open MakePrintAndUpdate (StubStateMachine)

    let%expect_test "test1" =
      let final =
        StubStateMachine.initial
        |> print_and_update (StubStateMachine.Num 1.)
        |> print_and_update (StubStateMachine.Op `Add)
        |> print_and_update (StubStateMachine.Num 2.)
        |> print_and_update StubStateMachine.Calculate
        |> print_and_update StubStateMachine.Empty
      in
      [%expect
        {|
        WaitInitial
        (WaitOperation (acc 1))
        (WaitArgument (acc 1) (op Add))
        (Calculation (acc 1) (op Add) (arg 2))
        (WaitOperation (acc 0)) |}];
      print_s [%sexp (final : StubStateMachine.state)];
      [%expect {| (Finish 0) |}];
      print_s [%sexp (StubStateMachine.result final : float option)];
      [%expect {| (0) |}]
    ;;
  end)
;;
```

Регрессионное тестирование: реализация этой работы началась после первой попытки сдать 3 работу, поэтому была расширена функциональность, относящаяся к 3 работе и проведено регрессионное тестирование и исправление ошибок, после чего загружена вторая попытка 3 работы.

4 Вывод

В данной работе мы ознакомились с основами тестирования в контролируемой среде тестирования программного обеспечения и проведения регрессионного тестирования. Исходный код доступен на GitHub.