Componenta echipei : Axinte Teodor-Ionut

Tema aleasa : Tema 2 (Search-based Software Testing)

***Chaining approach***

Metoda chaining approach este un tip de testare combinatoriala, care este o metoda de generare a cazurilor de testare care acopera toate posibilele combinatii ale parametrilor de intrare si a valorilor lor. In general, abordarea prin inlantuire este una dintre cele mai populare metode de testare combinatoriala, utilizata atat in industrie cat si in cercetare stiintifica. Este foarte eficienta pentru sistemele cu un numar mare de parametri de intrare, deoarece reduce numarul total de cazuri de testare necesare pentru a acoperi toate combinatiile posibile de valori, acesta fiind si scopul ei, totodata inca detectand erori in sistem.

Abordarea prin inlantuire functioneaza prin inceperea cu primul parametru de intrare si generare a tuturor valorilor posibile pentru acel parametru. Apoi, pentru fiecare combinatie de valori generate pentru primul parametru, algoritmul genereaza toate valorile posibile pentru al doilea parametru, si asa mai departe, pana cand toti parametrii au fost inlantuiti impreuna. Fiecare combinatie de valori pentru toti parametrii reprezinta un caz de testare unic.

Pentru a genera cazuri de testare folosing algortimul de abordare prin inlantuire, trebuie sa specificam parametrii de intrare si valorile posibile. Acest lurcu se face de obicei folosind o tabela sau o matrice, in care fiecare rand reprezinta o posibila combinatie de valori de intrare.

De exemplu, sa luam un sistem care are trei parametri de intrare: A, B si C. A poate lua valorile 1,2 sau 3, B poate lua valorile 4,5 sau 6, iar C poate lua valorile 7,8 sau 9. Pentru a utiliza abordarea prin inlantuire pentru a genera cazuri de testare pentru acest sistem, ar trebui create o tabela similara cu:

|  |  |  |  |
| --- | --- | --- | --- |
| Caz de testare | A | B | C |
| 1 | 1 | 4 | 7 |
| 2 | 1 | 4 | 8 |
| 3 | 1 | 4 | 9 |
| 4 | 1 | 5 | 7 |
| 5 | 1 | 5 | 8 |
| 6 | 1 | 5 | 9 |
| 7 | 1 | 6 | 7 |
| 8 | 1 | 6 | 8 |
| 9 | 1 | 6 | 9 |
| 10 | 2 | 4 | 7 |
| 11 | 2 | 4 | 8 |
| 12 | 2 | 4 | 9 |
| 13 | 2 | 5 | 7 |
| 14 | 2 | 5 | 8 |
| 15 | 2 | 5 | 9 |
| 16 | 2 | 6 | 7 |
| 17 | 2 | 6 | 8 |

Algoritmul de abordare prin inlantuire functioneaza prin generarea tuturor valorilor posibile pentru primul parametru de intrare, iar apoi pentru fiecare combinatie a valorilor primului parametru, se genereaza toate valorile posibile pentru al doilea parametru, si asa mai departe, pana cand se obtin toate combinatiile posibile ale valorilor parametrilor.

Este important de notat ca abordarea prin inlantuire poate fi utilizata nu numai pentru a genera combinatii de valori de intrare, ci si pentru a genera combinatii de evenimente sau de comportamente. Aceasta inseamna ca poate fi utilizata si in contextul testarii software-ului, pentru a testa combinatii complexe de functionalitati.

De exemplu, pentru a testa o aplicatie web care are mai multe functionalitati, cum ar fi autentificarea utilizatorului, cautarea produselor si plasarea comenzii, abordarea prin inlantuire poate fi utilizata pentru a identifica combinatiile de evenimente care trebuie testate. Acest lucru poate fi util in situatiile in care sunt necesare teste mai complexe si cuprinzatoare, care sa acopere mai multe functionalitati.

De asemenea, este important sa se mentioneze si ca abordarea prin inlantuire nu este intotdeauna cea mai eficienta metoda de testare si pot exista situatii in care alte metode ar fi mai potrivite. De aceea, este important sa se evalueze intotdeauna nevoile si caracteristicile specifice ale sistemului inainte de a alege o metoda de testare.

Sa luam exemplul unei aplicatii web de comert electronic care permite utilizatorilor sa caute produse pe site si sa plaseze comenzi. Aceasta aplicatie poate avea mai multi parametri de intrare precum :

* Campul de cautare al produsului
* Optiunea de sortare a rezultatelor cautarii
* Campul de filtrare dupa pret sau alte caracteristici ale produsului
* Optiunea de selectare a produselor si de adaugare in cosul de cumparaturi
* Campul de adaugare a informatiilor de livrare si plata

Pentru a testa aceasta aplicatie, putem aplica abordarea prin inlantuire pentru a identifica combinatiile de valori ale acestor parametric care trebuie testate. De exemplu, putem defini urmatoarele combinatii de valori:

* Cautare dupa un produs specific, sortat dupa pret crescator, filtrate dupa pret si caracteristici, apoi adaugat in cos si finalizat comanda cu informatiile de livrare si plata
* Cautare dupa un alt produs, sortat dupa popularitate, filtrat dupa caracteristici si pret, adaugat in cos, apoi revenit la cautare si adaugat un alt produs in cos inainte de a finaliza comanda
* Cautare dupa un produs, sortat dupa nume, adaugat in cos, apoi revenit la cautare si adaugat acelasi produs din nou in cos inainte de a finaliza comanda

Acest combinatii de valori pot fi utilizate pentru a testa diferite scenarii de utilizare ale aplicatiei, cu diverse combinatii de actiuni si valori de intrare.

Concret, putem aplica metoda de chaining approach pe urmatorul exemplu

|  |
| --- |
| class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  this.age = null;  this.email = null;  }  setAge(age) {  this.age = age;  return;  }  setEmail() {  this.email = email;  return this;  }  save() {  // salvare in baza de date a datelor personale  console.log(  `Saved ${this.firstName} ${this.lastName} (${this.age}, ${this.email})`  );  return this;  }  }  const person = new Person('John', 'Doe')  .setAge(30)  .setEmail('john.doe@example.com')  .save(); |

In acest exemplu, clasa Person defineste o metoda Save(), care salveaza datele personale ale unei persoane intr-o baza de date. Metodele setAge(age) si setEmail(email) sunt definite pentru a permite setarea varstei si a adresei de email ale unei persoane. Ambele metode intorc obiectul curent (this), astfel incat se poate aplica in continuare acelasi obiect.

In cele din urma, cu ajutorul metodei chaining approach, cream un obiect Person numit person, setam varsta si adresa de email, apoi salvam datele personale in baza de date. Rezultatul afisat in consola va fi ‘Saved John Doe (30, [john.doe@example.com](mailto:john.doe@example.com)).

Testarea automata a codului o putem realiza prin urmatoare secventa de cod:

|  |
| --- |
| describe('Person', () => {  it('should set age, email and save the data', () => {  const person = new Person('John', 'Doe')  .setAge(30)  .setEmail('john.doe@example.com')  .save();  expect(person).toBeInstanceOf(Person);  expect(person.firstName).toEqual('John');  expect(person.lastName).toEqual('Doe');  expect(person.age).toEqual(30);  expect(person.email).toEqual('john.doe@example.com');  });  }); |

In acest exemplu, am creat o metoda de testare care verifica daca instanta clasei Person este creata cu success si daca datele au fost salvate corect. Pentru a realiza acest lucru, am creat o instanta a clasei ‘Person’ si am apelat succesiv metodele setAge(), setEmail(), save(). In final, am verificat ca instanta are toate datele setate corect, utilizand metodele expect() si toEqual().

Un exemplu mai detaliat unde este implementata metoda de chaining approach este urmatorul:

|  |
| --- |
| const request = require('request-promise-native');  const { expect } = require('chai');  class ApiClient {  constructor(baseUrl) {  this.baseUrl = baseUrl;  this.headers = {};  this.queryParams = {};  this.body = {};  this.response = null;  }  setHeaders(headers) {  this.headers = headers;  return this;  }  setQueryParams(queryParams) {  this.queryParams = queryParams;  return this;  }  setBody(body) {  this.body = body;  return this;  }  async get(path) {  const options = {  uri: `${this.baseUrl}${path}`,  qs: this.queryParams,  headers: this.headers,  json: true,  };  this.response = await request.get(options);  return this;  }  async post(path) {  const options = {  uri: `${this.baseUrl}${path}`,  qs: this.queryParams,  headers: this.headers,  json: true,  body: this.body,  };  this.response = await request.post(options);  return this;  }  assertStatus(statusCode) {  expect(this.response.statusCode).to.equal(statusCode);  return this;  }  assertBody(body) {  expect(this.response.body).to.deep.equal(body);  return this;  }  assertHeader(headerName, headerValue) {  expect(this.response.headers[headerName.toLowerCase()]).to.equal(headerValue);  return this;  }  }  // Exemplu de utilizare  const client = new ApiClient('https://api.example.com')  .setHeaders({ Authorization: 'Bearer abc123' })  .setQueryParams({ limit: 10 })  .get('/users')  .assertStatus(200)  .assertHeader('Content-Type', 'application/json')  .assertBody([{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }]); |

Acest exemplu simuleaza un client HTTP care poate face requesturi de tip GET si POST la un server specificat prin baseUrl. Metoda chaining approach este utilizata pentru a seta optiuni pentru fiecare request in parte (de exemplu, headers, query params, body), pentru a face requestul si pentru a testa raspunsul primit de la server. Metode ‘assertStatus’, ‘assertBody’, ‘assertHeader’ sunt folosite pentru a testa starea de raspuns a serverului.

Metoda chaining approach poate fi folosita in acest exemplu pentru a efectua o serie de teste automate intr-un mod eficient si organizat. Sa presupunem ca dorim sa testam mai multe apeluri API si sa ne asiguram ca acestea returneaza coduri de stare corecte, ca raspunsurile sunt in formatul corect si ca contin datele dorite.

In acest caz, am putea utiliza metoda chaining approach pentru a defini toate testele noastre in lant, astfel incat sa putem executa toate testele pentru fiecare apel API intr-un mod simplu si eficient

Un exemplu de utilizare a metodei chaining approach este:

|  |
| --- |
| describe('API Tests', () => {  let client;  before(() => {  client = new ApiClient('https://api.example.com')  .setHeaders({ Authorization: 'Bearer abc123' });  });  it('should return a list of users', async () => {  const res = await client  .setQueryParams({ limit: 10 })  .get('/users')  .assertStatus(200)  .assertHeader('Content-Type', 'application/json');  // Verificăm că răspunsul conține datele dorite  expect(res.response.body).to.deep.equal([{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }]);  });  it('should return a single user', async () => {  const res = await client  .get('/users/1')  .assertStatus(200)  .assertHeader('Content-Type', 'application/json');  // Verificăm că răspunsul conține datele dorite  expect(res.response.body).to.deep.equal({ id: 1, name: 'John' });  });  it('should return a 404 error for non-existing users', async () => {  const res = await client  .get('/users/999')  .assertStatus(404)  .assertHeader('Content-Type', 'application/json');  // Verificăm că răspunsul conține un mesaj de eroare corespunzător  expect(res.response.body.message).to.equal('User not found');  });  }); |

In acest exemplu, am definit mai multe teste diferite in cadrul unui bloc de descriere, fiecare test fiind definit prin intermediul metodei chaining approach.

Inainte de a incepe fiecare test, am creat un nou client API si am setat un antet de autorizare, astfel incat sa nu fie nevoie sa repeteam aceasta actiuni pentru fiecare test.

Fiecare test face un apel API diferit si efectueaza mai multe teste deasupra raspunsului, utilizand metoda chaining approach. In cazul de fata, testele verifica codul de stare, antetul Content-Type si corpul raspunsului pentru a asigura ca sunt corecte.