گزارش پروژه

معرفی سیستم تحت آزمون

سیستمی که در این پروژه برای تست استفاده شده است، تمرین درس برنامه نویسی پیشرفته سال گذشته معرفی شده توسط استاد درس بوده است که با زبان جاوا بازنویسی شده است. این پروژه طراحی یک سایت رزرو بلیط هواپیمایی است که دارای امکاناتی مانند رزرو، ایجاد و مشاهده پرواز هاست. در این سیستم ابتدا لیست اطلاعات تمامی پرواز ها از یک فایل خوانده میشود. کاربران مختلف میتوانند پرواز ها را رزرو کنند، بلیت ها را مشاهده کنند و یا لیست پرواز ها را طبق خواسته خودشان فیلتر کرده و مشاهده کنند.

این سیستم به زبان جاوا پیاده سازی شده است و ابزاری که برای تست آن استفاده کرده ایم برای تمامی متد ها و کلاس های این سیستم، تست کیس هایی به صورت خودکار تولید میکند. در بخش بعدی راجع به ابزار تست استفاده شده توضیحات بیشتری خواهیم داد.

همچنین source code ها از طریق لینک زیر در دسترس هستند:

https://github.com/mhmdaminraeisi/test-project

روش و ابزار تست استفاده شده

برای تولید کردن تست کیس ها از ابزار evosuite استفاده شده است که با استفاده از روش search-based برای تمامی متد های سیستم به صورت خودکار مجموعه تست های jUnit تولید میکند. این ابزار تلاش میکند معیار های مختلف پوشش مانند branch coverage را بهبود ببخشد. این ابزار با استفاده از یک رویکرد تکاملی مبتنی بر الگوریتم ژنتیک برای تولید و استخراج تست کیس ها استفاده میکند. همچنین تعداد تست کیس های تولید شده را به حداقل میرساند و تنها تست کیس هایی که در بهبود معیار های coverage تاثیر دارند را نگه میدارد. و jUnit assert هایی که رفتار های فعلی کلاس های آزمایش شده را نشان میدهند را تولید میکند.

تمامی لایبرری های استفاده شده به کمک dependency های maven به پروژه اضافه شده اند.

گزارش پوشش کد و امتیاز موتاسیون به دست آمده

برای اندازه گیری معیار coverage از دو ابزار jacoco و PIT استفاده شده است که jacoco گزارش coverage و branch و instruction coverage را میدهد و PIT گزارش line coverage و mutation coverage را میدهد.

در مجموع ۲۵۲ تست کیس تولید شده است که طبق گزارش ابزار jacoco ، ۶۴ درصد branch coverage و ۶۷ درصد instruction coverage داشته است. در زیر گزارش درصد coverage هر پکیج و هر کلاس مشخص شده است:

test-evosuite

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.example		69%		70%	54	194	116	388	2	76	0	9
org.example.filter		62%		37%	36	62	17	67	12	38	0	6
Total	746 of 2,326	67%	98 of 277	64%	90	256	133	455	14	114	0	15

org.example

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Functions		38%		40%	32	55	63	110	1	16	0	1
Utravel		76%		77%	11	46	20	99	0	17	0	1
User		47%		50%	7	17	18	36	1	8	0	1
Flight		88%		96%	2	40	9	71	0	14	0	1
Ticket		93%		95%	1	18	2	26	0	8	0	1
Time		92%		100%	0	10	2	22	0	7	0	1
Main		83%		50%	1	3	2	13	0	2	0	1
FileScanner		100%		100%	0	3	0	9	0	2	0	1
Headers		100%		n/a	0	2	0	2	0	2	0	1
Total	604 of 1,949	69%	68 of 229	70%	54	194	116	388	2	76	0	9

org.example.filter

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Input		60%		n/a	12	25	1	14	12	25	0	1
TimeSpanFilter		57%		27%	9	12	5	19	0	3	0	1
CostRangeFilter		70%		45%	10	13	4	14	0	3	0	1
AirlineFilter		46%		25%	2	5	4	9	0	3	0	1
OriginDestinationFilter		69%		50%	3	6	3	10	0	3	0	1
Filter		100%		n/a	0	1	0	1	0	1	0	1
Total	142 of 377	62%	30 of 48	37%	36	62	17	67	12	38	0	6

جزئیات کامل گزارش های jacoco در فایل index.html داخل پوشه ی target/site/jacoco-ut قابل مشاهده است.
در پایین نیز گزارش هایی که توسط ابزار PIT آمده است که در مجموع ۵۷ درصد mutation coverage و ۹۳ درصد line coverage داشته است.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
13	93%	420/454	57%	171/298	65%	171/264

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.example	8	91%	354/388	51%	126/247	59%	126/213
org.example.filter	5	100%	66/66	88%	45/51	88%	45/51

Name	Line Coverage		Mutation Coverage		Test Strength	
FileScanner.java	100%	10/10	100%	2/2	100%	2/2
Flight.java	99%	70/71	42%	22/52	42%	22/52
Functions.java	81%	89/110	16%	13/81	24%	13/55
Main.java	100%	13/13	67%	2/3	67%	2/3
Ticket.java	100%	26/26	68%	17/25	68%	17/25
Time.java	100%	22/22	86%	12/14	86%	12/14
User.java	75%	27/36	50%	10/20	71%	10/14
Utravel.java	97%	97/100	96%	48/50	100%	48/48

Name	Line Coverage		Mutation Coverage		Test Strength	
AirlineFilter.java	100%	9/9	75%	3/4	75%	3/4
CostRangeFilter.java	100%	14/14	80%	12/15	80%	12/15
Input.java	100%	14/14	94%	15/16	94%	15/16
OriginDestinationFilter.java	100%	10/10	80%	4/5	80%	4/5
TimeSpanFilter.java	100%	19/19	100%	11/11	100%	11/11

جزئیات کامل گزارش های PIT نیز در فایل index.html داخل پوشه ی target/pit-reports قابل مشاهده است.

تحلیل نتایج

همانطور که از نتایج بالا مشخص است کلاس ها نتایج متفاوتی گرفته اند که به طور کلی میتوانیم آن ها را به چهار دسته تقسیم کنیم:

- دسته ی اول کلاس هایی هستند که هم branch coverage بالا و هم mutation coverage بالا گرفته اند. کلاس هایی مثل Time و FileScanner در این دسته قرار میگیرند. دلیل آن نیز به خاطر این است که این کلاس ها کوچک هستند و متد های اندک و ساده ای دارند و جست و جو و تولید تست هایی که پوشش نسبتا کاملی داشته باشند ساده است.
- دسته ی دوم کلاس هایی هستند که branch coverage بالا اما mutation coverage پایین داشته اند. مهم ترین کلاسی که در این دسته قرار گرفته است کلاس Flight میباشد که که علی رغم اینکه branch coverage آن ۹۶ درصد بوده است، تنها ۴۲ درصد mutation coverage داشته است. به عنوان مثال متد applyCostRangeFilter این کلاس که ۱۰ تا mutant داشته است که هر ۱۰ تای آن ها survived شده اند. در زیر انواع

mutant های تولید شده برای هر خط از این متد مشخص شده است:

```
109     public void applyCostRangeFilter(double min, double max, boolean minEntered, boolean maxEntered) {
110 1       if (!minEntered) {
111 2           costRangeFilterApplied = cost <= max;
112 1       } else if (!maxEntered) {
113 2           costRangeFilterApplied = min <= cost;
114         } else {
115 4           costRangeFilterApplied = min <= cost && cost <= max;
116         }
117     }
```

```
110  1. negated conditional → SURVIVED
111  1. changed conditional boundary → SURVIVED
     2. negated conditional → SURVIVED
112  1. negated conditional → SURVIVED
113  1. changed conditional boundary → SURVIVED
     2. negated conditional → SURVIVED
115  1. negated conditional → SURVIVED
     2. negated conditional → SURVIVED
     3. changed conditional boundary → SURVIVED
     4. changed conditional boundary → SURVIVED
```

دلیل اینکه تمامی mutant های تولید شده survived شده اند این است که اولا این متد از نوع void است و خروجی ندارد و همچنین بر اساس مقادیر ورودی اش، مقدار یک متغیر global را تغییر میدهد و side effect ایجاد میکند. برای همین این اثر مستقلا نمیتواند در داخل این متد تست شود. در نتیجه هر تغییری که روی مقادیر ورودی و mutant ها اتفاق بیفتد صرفا مقدار متغیر گلوبال را تغییر میدهد که این خللی در خروجی و نتیجه ی این متد ایجاد نمیکند و برای همین هست که هیچ یک از mutant ها kill نشده اند. اکثر متد های این کلاس همچین رفتاری دارند و این دلیل آن است که branch coverage بالا داریم چرا که متد ها منطق خیلی پیچیده ای ندارند و میتوانیم تست هایی تولید کنیم که همه ی شاخه ها را پوشش دهند اما به دلیلی که گفته شد mutation coverage کمی دارند.

- دسته ی سوم کلاس هایی هستند که branch coverage پایین اما mutation coverage بالا داشته اند. کلاس هایی که در داخل پکیج filter قرار دارند، جزو این دسته هستند. به عنوان مثال متد apply در کلاس CostRangeFilter را بررسی میکنیم که کد آن در تصویر زیر مشخص شده است:

```
21.  @Override
22.  public boolean apply(List<Flight> flights) {
23.      if (!minPriceEntered && !maxPriceEntered) {
24.          return false;
25.      }
26.      if (minPriceEntered && minPrice < 0 ||
27.          maxPriceEntered && maxPrice < 0 ||
28.          minPriceEntered && maxPriceEntered && minPrice > maxPrice) {
29.          throw new RuntimeException(Headers.BAD_REQUEST);
30.      }
31.      for (Flight flight : flights) {
32.          flight.applyCostRangeFilter(minPrice, maxPrice, minPriceEntered, maxPriceEntered);
33.      }
34.      return true;
35.  }
```

اگر به جزئیات تعداد برنچ های پوشش داده شده با دقت بیشتری نگاه کنیم، میبینیم که اولین if دارای ۴ شاخه میباشد که تنها یکی از آنها پوشش داده شده است. if بعدی دارای ۱۴ شاخه میباشد که ۷ تای آن ها پوشش داده شده است. دلیل این اتفاق این است که متغیر هایی که در شرط ها استفاده شده اند همگی global هستند و نه در ورودی داده میشوند و نه در داخل متد تعریف شده اند. در نتیجه یک مقدار ثابت دارند و حالت های مختلف آن ها توسط evosuite جست و جو نمیشوند. به همین دلیل است که تنها یک شاخه از ۴ شاخه ی if اول پوشش داده میشود. همچنین اگر یک نگاهی به تست های تولید شده بکنیم، میبینیم که تمامی تست کیس هایی که برای این متد تولید کرده است، مقدار لیست flights که در ورودی متد گرفته میشود یک لیست خالی است که این میتواند یکی از نقص های evosuite باشد که لیست با تعداد اعضای متفاوت برای تست تولید نکرده است. به همین دلیل است که for آخر تنها یکی از دو شاخه اش پوشش داده شده است و هیچ وقت داخل socpe را پوشش نداده است. اما این متد mutation coverage بالایی گرفته است. در تصویر زیر جزئیات mutant های تولید شده برای هر کدام از خطوط و شرط های متد را مشاهده میکنید:

```
21        @Override
22        public boolean apply(List<Flight> flights) {
23 2          if (!minPriceEntered && !maxPriceEntered) {
24 1              return false;
25           }
26 10         if (minPriceEntered && minPrice < 0 ||
27               maxPriceEntered && maxPrice < 0 ||
28               minPriceEntered && maxPriceEntered && minPrice > maxPrice) {
29               throw new RuntimeException(Headers.BAD_REQUEST);
30           }
31           for (Flight flight : flights) {
32 1             flight.applyCostRangeFilter(minPrice, maxPrice, minPriceEntered, maxPriceEntered);
33           }
34 1         return true;
35       }
```

23	1. negated conditional → KILLED 2. negated conditional → KILLED
24	1. replaced boolean return with true for org/example/filter/CostRangeFilter::apply → KILLED
26	1. negated conditional → KILLED 2. negated conditional → KILLED 3. changed conditional boundary → KILLED 4. negated conditional → KILLED 5. changed conditional boundary → KILLED 6. negated conditional → SURVIVED 7. negated conditional → KILLED 8. negated conditional → KILLED 9. negated conditional → SURVIVED 10. changed conditional boundary → KILLED
32	1. removed call to org/example/Flight::applyCostRangeFilter → SURVIVED
34	1. replaced boolean return with false for org/example/filter/CostRangeFilter::apply → KILLED

دلیل این که اکثر mutant ها kill شده اند این است که اکثر mutant هایی که ایجاد شده اند مثل انواع negated conditional و changed conditional boundary مقادیر predicate ها را و در نتیجه branch هایی که انتخاب میشوند را مستقل از اینکه مقادیر متغیر های global چه باشند تغییر میدهد و این باعث میشود که مسیر متفاوتی را برنامه طی کند و خروجی و نتیجه ی متفاوتی ایجاد شود که باعث kill شدن mutant ها بشود. اما مثلا دو تا از mutant هایی که بر روی predicate خط ۲۶ ایجاد شده اند survived شده اند. دلیل آن این است که if خط ۲۶ دارای ۱۴ شاخه میباشد که تعداد زیادی از آن ها true هستند و تعداد زیادی نیز false هستند. برای همین احتمال اینکه تغییر مقادیر predicate ها و تغییر branch ها نتایج یکسانی را داشته باشد (مثلا هر دو برنچ قبل و بعد از اعمال mutation داخل if بروند) زیاد است و این باعث میشود که تعدادی mutant های ایجاد شده روی آن survived شود.

- دسته ی آخر هم کلاس هایی هستند که هم branch coverage و هم mutation coverage پایین دارند. مهم ترین کلاسی که در این دسته قرار میگیرد کلاس Functions است که branch coverage آن ۴۰ درصد و mutation coverage آن تنها ۱۶ درصد است. مثلا متد applyFilter این کلاس که branch coverage آن تنها ۶ درصد و mutation coverage آن ۰ درصد میباشد را در نظر بگیرید. در زیر تصویری از خطوی که پوشش داده شده اند آمده است.

```java
public static String applyFilter(String inp, Utravel utravel) {
    inp = splitFirstSign(inp);
    String[] inputs = inp.split(Headers.SHOW_DELIMITER);
    if (inputs.length % 2 == 1) {
        throw new RuntimeException(Headers.BAD_REQUEST);
    }
    Input input = Headers.EMPTY_INPUT;
    for (int i = 0; i < inputs.length; i += 2) {
        if (inputs[i+1].length() == 0) {
            throw new RuntimeException(Headers.BAD_REQUEST);
        }
        String title = inputs[i];
        switch (title) {
            case Headers.FROM:
                input.setOrigin(inputs[i+1]);
                break;
            case Headers.TO:
                input.setDestination(inputs[i+1]);
                break;
            case Headers.MIN_PRICE:
                input.setMinPrice(Double.parseDouble(inputs[i+1]));
                break;
            case Headers.MAX_PRICE:
                input.setMaxPrice(Double.parseDouble(inputs[i+1]));
                break;
            case Headers.AIRLINE:
                input.setAirline(inputs[i+1]);
                break;
            case Headers.DEPARTURE_DATE:
                input.setDepartureDate(Integer.parseInt(inputs[i+1]));
                break;
            case Headers.MIN_DEPARTURE_TIME:
                input.getMinTime().set(inputs[i+1]);
                input.setMinTimeEntered(true);
                break;
            case Headers.MAX_DEPARTURE_TIME:
                input.getMaxTime().set(inputs[i+1]);
                input.setMaxTimeEntered(true);
                break;
            default:
                throw new RuntimeException(Headers.BAD_REQUEST);
        }
    }
    return utravel.applyFilter(input);
}
```

همانطور که در تصویر و آمار مشخص است تقریبا هیچ کدام از branch ها کاور نشده اند. دلیل آن این است که ورودی این متد یک متغیر از نوع string به نام input است که انتظار داریم فرمت خاصی را داشته باشد. مثلا به شکل tehran to ahvaz from باشد که دو کلید واژه ی from و to را داشته باشد و یا ... . اما evosuite تست کیس هایی که برای تست این متد ساخته است ورودی تابع را یک رشته ی رندم داده است که در تصویر زیر که یک تست کیس ایجاد شده است مشخص است.

```java
no usages  MohammadAmin Raeisi
@Test(timeout = 4000)
public void test39()  throws Throwable  {
    // Undeclared exception!
    try {
      Functions.applyFilter( inp: "b?&:Yi,,WjkkS;DEL{E", (Utravel) null);
      fail("Expecting exception: RuntimeException");

    } catch(RuntimeException e) {
       //
       // Bad Request
       //
       verifyException("org.example.Functions", e);
    }
}
```

رشته ای که به عنوان ورودی به این متد پاس داده میشود هیچ یک از فرمت های مورد انتظار را ندارد و در نتیجه وارد هیچ کدام از case هایی تعریف کرده ایم نمیشود و فقط به case default میرود. در نتیجه coverage branch بسیار کمی دارد. در تصویر زیر نیز همه ی mutant هایی که survived شده اند به همراه شماره ی خط کد مشخص شده اند:

88	1. negated conditional → SURVIVED 2. Replaced integer modulus with multiplication → SURVIVED
92	1. changed conditional boundary → SURVIVED 2. negated conditional → SURVIVED
93	1. Replaced integer addition with subtraction → SURVIVED 2. negated conditional → SURVIVED
99	1. Replaced integer addition with subtraction → SURVIVED 2. removed call to org/example/filter/Input::setOrigin → SURVIVED
102	1. removed call to org/example/filter/Input::setDestination → NO_COVERAGE 2. Replaced integer addition with subtraction → NO_COVERAGE
105	1. removed call to org/example/filter/Input::setMinPrice → NO_COVERAGE 2. Replaced integer addition with subtraction → NO_COVERAGE
108	1. Replaced integer addition with subtraction → NO_COVERAGE 2. removed call to org/example/filter/Input::setMaxPrice → NO_COVERAGE
111	1. Replaced integer addition with subtraction → NO_COVERAGE 2. removed call to org/example/filter/Input::setAirline → NO_COVERAGE
114	1. removed call to org/example/filter/Input::setDepartureDate → NO_COVERAGE 2. Replaced integer addition with subtraction → NO_COVERAGE
117	1. removed call to org/example/Time::set → NO_COVERAGE 2. Replaced integer addition with subtraction → NO_COVERAGE
118	1. removed call to org/example/filter/Input::setMinTimeEntered → NO_COVERAGE
121	1. Replaced integer addition with subtraction → NO_COVERAGE 2. removed call to org/example/Time::set → NO_COVERAGE
122	1. removed call to org/example/filter/Input::setMaxTimeEntered → NO_COVERAGE

همانطور که توضیح داده شد، چون ورودی متد به صورت رندم ساخته میشود، در نتیجه تغییر دادن predicate هایی که بر روی آن تعریف شده اند خیلی تغییر خاصی را بر روی نتیجه ایجاد نمیکند. چرا که انتظار داریم هیچ کدام از شرط ها روی رشته ی رندم ورودی بر قرار نباشند و در نتیجه رفتار تابع قبل و بعد از اعمال mutation ها یکسان باشد که باعث survived شدن تمامی mutant ها میشود.

نتیجه گیری

نتیجه ای که از توضیحات بالا و تحلیل های این ابزار گرفته میشود این است که بنظر میرسد evosuite برای سیستم هایی که ورودی آن رشته هایی هستند که از دامنه ی خاصی تبعیت میکنند، مثلا حتما باید به فرمت

POST signup ؟ username u password p

و یا سایر پترن هایی که مجاز هستند باشند، روش search-based خیلی مناسب نباشد. چرا که رشته هایی که برای تست تولید میکند تقریبا رندم هستند و هیچ یک از پترن های مجاز را رعایت نمیکند و در نتیجه خیلی نتایجی که حاصل میشود معتبر نیستند. برای این مدل پروژه ها که دامنه ی مشخصی دارند بهتر است روش هایی که دامنه ی مقادیر را در نظر میگیرند مثل روش های مبتنی بر افراز فضای ورودی استفاده شود.