Github Repository: https://github.com/PashaBarahimi/Software-Testing-Course-Projects

Last Commit: c52051176dc0c00aad57945f2f48bfe8cc2db58e

سوال اول

در پوشش جمله (statement coverage) همه جمله‌های کد حداقل یک بار اجرا می‌شوند و در پوشش شاخه (branch coverage)، همه شاخه‌های کد یک بار اجرا می‌شوند.

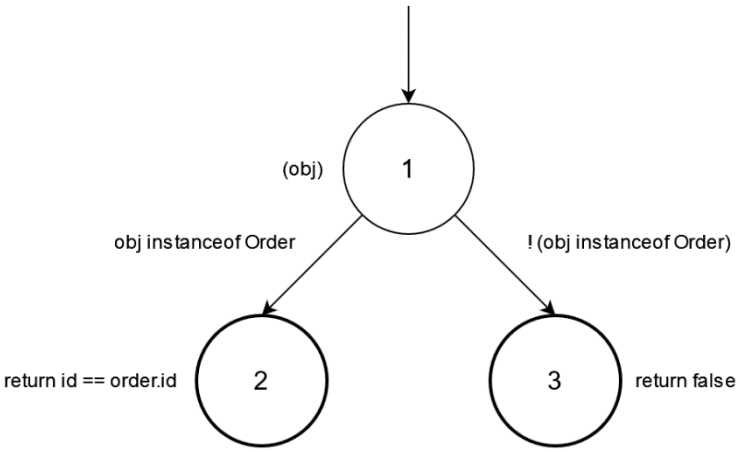
قطعه کد اول:

```
public boolean equals(Object obj) {
    if (obj instanceof Order order) {
        return id == order.id;
    }
    return false;
}
```

اگر فقط از یک عدد تست استفاده کنیم (به صورتی که در تست یک بار بیشتر این تابع assert نمی‌شود)، نمی‌توان در این تابع، به پوشش جمله و حتی پوشش شاخه 100 درصدی رسید.

این به این خاطر است که چه در if برویم و چه در آن نرویم، به یک return statement می‌رسیم که از آن نمی‌توان جلوتر رفت. در حالت کلی، اجرا شدن دو return در یک تست ممکن نیست. بنابراین پوشش جمله کامل نخواهیم داشت.

control flow graph کد:



همانطور که می‌بینیم، در یک path غیر ممکن است که هر دو نود پایین کاور شوند.

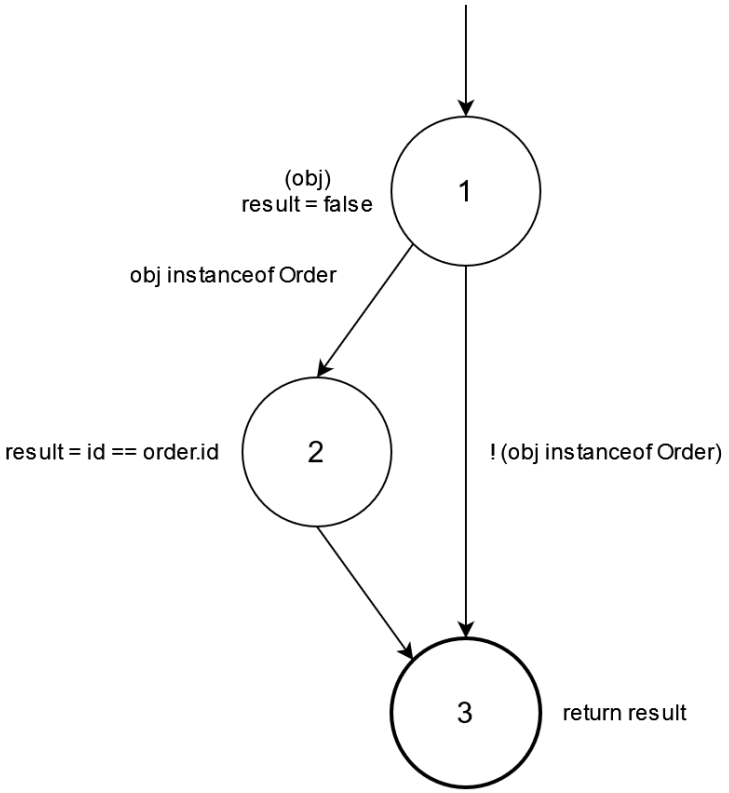
قطعه کد دوم:

```
public boolean equals(Object obj) {
    var result = false;
    if (obj instanceof Order order) {
        result = id == order.id;
    }
    return result;
}
```

در این حالت، می‌توان با نوشتن یک تست، همه statement-ها را پوشش داد و به پوشش جمله 100 درصد رسید.
برای این کار، کافیست که ورودی را یک آبجکت از تایپ Order قرار بدهیم که در این صورت، داخل branch ایف می‌رود و statement داخل آن اجرا می‌شود.
پس پوشش جمله داریم ولی پوشش شاخه نداریم.
تست نوشته شدن:

```
@Test
public void testEqualsComparesIds() {
    Order order = new Order() {{
        setId(0);
    }};
    Order anotherOrder = new Order() {{
        setId(0);
    }};
    Assertions.assertTrue(order.equals(anotherOrder));
}
```

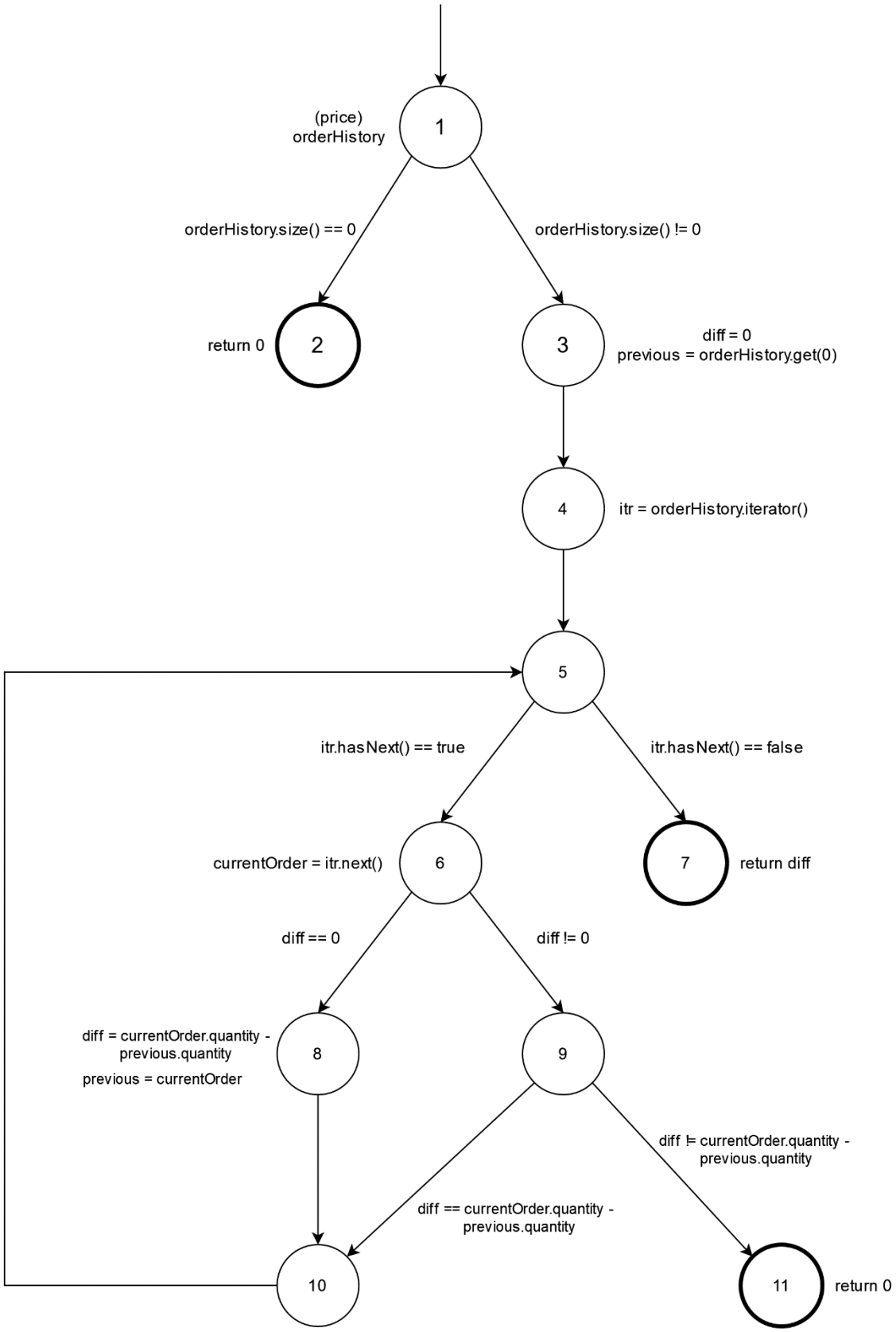
control flow graph کد:



همانطور که می‌بینیم، بازدید همه نودها از مسیر 1 2 3 ممکن است.

سوال دوم

1. Control Flow Graph



2. Prime Path

برای به دست آوردن Prime Path-ها، باید بلندترین Simple Path-هایی را در گراف پیدا کنیم به صورتی که یا لوپی در آن وجود نداشته باشد و یا اینکه تنها لوپ آن در رئوس اول و آخر path باشد. با بررسی گراف و مسیرهای موجود در آن، به جدول زیر برای Prime Path-ها می‌رسیم:

Length	Prime Path
0	-
1	[1,2]!
2	-
3	-
4	[1,3,4,5,7]! [5,6,9,10,5]* [5,6,8,10,5]* [6,8,10,5,6]* [6,9,10,5,6]* [8,10,5,6,8]* [9,10,5,6,9]* [9,10,5,6,8] [10,5,6,8,10]* [10,5,6,9,10]* [6,8,10,5,7]! [6,9,10,5,7]!
5	[8,10,5,6,9,11]!
6	[1,3,4,5,6,9,11]! [1,3,4,5,6,8,10] [1,3,4,5,6,9,10]

3. DU Path

Variable	DU Pair	DU Path
price	(1,-)	-
orderHistory	(1,(1,2)) (1,(1,3)) (1,3) (1,4)	[1,2]! [1,3] [1,3] [1,3,4]
diff	(3,7) (3,(6,8)) (3,(6,9)) (3,(9,10)) (3,(9,11)) (8,7) (8,(6,8)) (8,(6,9)) (8,(9,10)) (8,(9,11))	[3,4,5,7]! [3,4,5,6,8] [3,4,5,6,9] [3,4,5,6,9,10] [3,4,5,6,9,11]! [8,10,5,7]! [8,10,5,6,8]* [8,10,5,6,9] - [8,10,5,6,9,11]!
previous	(3,8) (3,(9,10)) (3,(9,11)) (8,(9,10)) (8,(9,11)) (8,8)	[3,4,5,6,8] [3,4,5,6,9,10] [3,4,5,6,9,11]! - [8,10,5,6,9,11]! [8,10,5,6,8]*
currentOrder	(6,8)	[6,8]

	(6,(9,10)) (6,(9,11))	[6,9,10] [6,9,11]!
itr	(4,(5,6)) (4,(5,7)) (4,6)	[4,5,6] [4,5,7]! [4,5,6]

سوال سوم

یک تابع با بدنه خالی و شامل یک return statement را در نظر بگیرید:

```
public int func() {
    return 0;
}
```

این تابع هیچ گونه du path-ای نخواهد داشت و برای ADUPC نیازی به هیچ نوع تستی ندارد. درصورتی که یک prime path به طول 0 دارد که بدون حداقل یک تست، کاور نمی‌شود.

مثالی دیگر:

```
public int func(int n) {
    if (true)
        System.out.println("Test");
    return n;
}
```

در این تابع دو شاخه و دو prime path داریم. این در صورتی است که با یکی از این شاخه‌ها، همه du path-ها کاور می‌شوند.

سوال چهارم

با وجود اینکه اگر تمام Prime Path-ها را پوشش دهیم، DU Path-ها نیز پوشش داده خواهند شد و اینکه پیدا کردن DU Path-ها معمولا دشوارتر از Prime Path-ها است، اما بعضی اوقات ترجیح می‌دهیم از DU Path به جای Prime Path استفاده کنیم.

از جمله این موارد، می‌توان به حالت‌های زیر اشاره کرد:

- در بعضی برنامه‌ها گراف بسیار پیچیده و بزرگی خواهیم داشت که نتیجه آن، Prime Path-های بسیار بزرگ است که باعث می‌شود پوشش تمام آن‌ها، کار بسیار دشوار و تست آن فرایندی زمان‌بر باشد. به همین دلیل، تست برنامه در حالتی که DU Path-ها را پوشش دهد، کار راحت‌تری خواهد بود.
- بعضی وقت‌ها می‌خواهیم برنامه را از لحاظ data flow تست کنیم که در این حالت، استفاده از ADUPC نتیجه بهتری نسبت به PPC خواهد داشت.
- یکی از مواردی که ADUPC می‌تواند به ما نشان دهد، عدم استفاده از یک متغیر تعریف شده است.
- تست‌ها در حالت ADUPC کوچک‌تر و بهینه‌تر خواهند بود.