علی نظری ۹۶۳۱۰۷۵

۱)

الف) مدل RIPR در مورد چهار مرحله‌ای که یک خطا منجر به شکست می‌شود صحبت می‌کند که این مراحل عبارتند از:

- مرحله اول Reachability: این مرحله به اجرای قسمتی از برنامه که منجر به خطا می‌شود اشاره دارد.
- مرحله دوم Infection: این مرحله به خراب‌شدن حالت (state) برنامه و PC در اثر اجرای خطا اشاره دارد.
- مرحله سوم Propagation: این مرحله به انتشار خرابی ناشی از مرحله‌ی قبل در برنامه اشاره دارد.
- مرحله چهارم Revealability: این مرحله به نمایان شدن خطای انتشار یافته در مرحله‌ی قبل در خروجی برنامه اشاره دارد که منجر به شکست شده است.

ب) آزمون‌پذیری معیاری است که نشان می‌دهد یک نرم‌افزار تا چه حدی قابل تست شدن است. آزمون‌پذیری دو مشخصه‌ی مهم Observability و Controllability را داراست.

- Observability (مشاهده‌پذیری): معیاری که نشان می‌دهد نتایج ورودی‌هایی که برنامه‌ی ما دریافت و اجرا می‌کند به چه میزان در خروجی قابل مشاهده است.
- Controllability (کنترل‌پذیری): معیاری که سهولت انتخاب ورودی برای اجرا و تست نرم‌افزار برای رسیدن به حالت‌های مختلف برنامه را نشان می‌دهد.

ج) مجموعه اقداماتی که برای اجرا و ارزیابی یک نرم‌افزار باید انجام دهیم را تشکیل می‌دهد که شامل چندین بخش است:

- Test case value: مقادیری ورودی که برای تست می‌دهیم.
- Prefix value: حالت ابتدایی‌ای که قبل از اجرای تست می‌خواهیم در آن قرار بگیریم.
- Postfix value: موارد انتهایی‌ای که بعد از اجرای تست می‌خواهیم اتفاق بیفتند.
- Expected result: مقادیری که ما با اجرای تست و ورودی‌هایی که داده‌ایم، انتظار داریم به عنوان خروجی ببینیم.

د) روشی است که برای حل چالش test oracle مورد استفاده قرار می‌گیرد و از طریق آن صحت خروجی برنامه به ازای یک ورودی تست را با کمک یک ورودی تست دیگر و بدست آوردن خروجی آن و سپس بررسی رابطه‌ی آن با ورودی دیگر مشخص می‌کنند. مانند به دست آوردن sin و cos با استفاده از روابطی که با هم دارند.

ه) مشکلی که وجود دارد ترتیبی است که ما برای integrate کردن تست‌ها به آن نیاز داریم تا هزینه‌ی کمتری برای ما داشته باشد که به این مسئله CITO می‌گوییم. برای حل این مشکل از dependency graph استفاده می‌کنیم.

و) در این روش از آزمون نرم‌افزار سعی می‌کنیم به صورت عمدی خطایی در نرم‌افزار ایجاد کنیم و بررسی کنیم که آیا تست‌های ما توانایی این خطا را پوشش می‌دهند یا خیر که در نتیجه‌ی آن کیفیت تست‌هایی که برای نرم‌افزارمان نوشته ایم بررسی می‌شود.

---

۲)

الف) یک ایراد در تابع equals وجود دارد و آن چیزی که در کلاس پدر است با کلاس فرزند هم‌خوانی ندارد زیرا یکی از ویژگی‌های رابطه‌ی برابری، ویژگی تقارنی (symmetric) است که در این تابع ایراد دارد. اگر به تست شماره‌ی ۱ و ۲ نگاه کنیم می‌بینیم که ۲ چیز با هم مقایسه شده اند و در یک کلاس مقدار برابری گرفته اند و در دیگری نابرابری گرفته اند. به نظرم این ایراد قابل اصلاح نیست چون ما از داخل کلاس فرزند نمی‌تونیم به مشخصات توابع فرزند دسترسی داشته باشیم که بخواهیم آن را اصلاح کنیم.

ب) اگر بخواهیم این عیب اجرا نشود باید تابع equals در کلاس فرزند را فراخوانی نکنیم؛ پس با یک آزمون که فقط تابع پدر را نیاز داشته باشد، درگیر اجرای این عیب نمی‌شویم. به عنوان مثال:

```
Point p1 = new Point (1,2);
Point p2 = new Point (1,2);
p1.equals(p2); // True
p2.equals(p1); // True
```

ج) برای اینکه عیب اجرا شود ولی منجر به خطا نشود، باید دو نمونه از کلاس فرزند بسازیم و تابع equals آن را فراخوانی کنیم. به عنوان مثال:

```
ColorPoint p1 = new ColorPoint (1,2, RED);
ColorPoint p2 = new ColorPoint (1,2, GREEN);
p1.equals(p2); // False
p2.equals(p1); // False
```

د) این امکان وجود ندارد زیرا اگر اجرای برنامه منجر به خطا شود، قطعا منجر به شکست هم خواهد شد.

ه) اولین حالت خطا بعد از اجرای خط زیر اتفاق می‌افتد:

p.equals (cp1);

زیرا برنامه برابری برمی‌گرداند در صورتی که به واسطه‌ی حالت تقارنی رابطه‌ی برابری ما باید منتظر یک نتیجه‌ی نابرابری بودیم.

---

۳)

آزمون مدل رانه (Model-Driven Test Design) و آزمون مبتنی بر مدل (Model-Based) هم‌پوشانی‌های و شباهت‌های زیادی با هم دارند اما اگر بخواهیم به تفاوت آن‌ها بپردازیم باید بگوییم که آزمون مدل رانه به این صورت است که یک مدل را از روی نرم‌افزاری که کد آن نوشته شده است به دست می‌آوریم ولی آزمون مبتنی بر مدل به این صورت است که طراحی تست‌های نرم‌افزار بر اساس یک مدلی مانند UML که نشان‌دهنده‌ی یک سری از ویژگی‌های نرم‌افزار به صورت abstract است، صورت می‌پذیرد.

---

۴)

معیارهایی که برای مقایسه‌ی معیارهای پوشش باید به آن‌ها توجه کنیم از این جنس اند که بتوانیم نیازمندی‌های تست را هر چه بیشتر خودکار کنیم، خطاها و عیب‌های بیشتری را بتوانیم به کمک آن‌ها پیدا کنیم، مقادیر ورودی بهینه‌تری را به کمک آن‌ها بسازیم یا حتی بتوانیم با تست‌های کمتری مجموعه‌ی مدنظرمان را پوشش دهیم که در راستای این موارد از روش‌هایی مثل subsumption و mutation testing می‌توانیم بهره بگیریم.

---

۵)

الف) متغیرهای ورودی testable function ما (search) برابر list و element هستند.

برای افراز و پیدا کردن ویژگی‌های آن و بلاک‌های هر کدام می‌توانیم دو رویکرد بر اساس اینترفیس و بر اساس عملکرد را داشته باشیم.

بر اساس اینترفیس:

characteristic	b1	b2	b3
List = null	True	False	
List = [ ] → empty	True	False	

بر اساس عملکرد:

characteristic	b1	b2	b3
element همان اولین عضو است	True	False	
element همان آخرین عضو است	True	False	
تعداد تکرار element در list	0	1	> 1

ب) برای بررسی ویژگی یک افراز خوب یعنی نداشتن اشتراک بین مجموعه‌ها و تشکیل مجموعه‌ی اصلی از اجتماع مجموعه‌ها برای این دو رویکرد داریم:

بر اساس اینترفیس:

ردیف اول: null بودن یا نبودن لیست با هم اشتراکی ندارند و اجتماع آن‌ها نیز کل مجموعه را تشکیل می‌دهد.

ردیف دوم: empty بودن یا نبودن لیست با هم اشتراکی ندارند و اجتماع آن‌ها نیز کل مجموعه را تشکیل می‌دهد.

بر اساس عملکرد:

ردیف اول: اولین عضو بودن یا نبودن element مورد نظر با هم اشتراکی ندارند و اجتماع آن‌ها نیز کل مجموعه را تشکیل می‌دهد.

ردیف دوم: آخرین عضو بودن یا نبودن element مورد نظر با هم اشتراکی ندارند و اجتماع آن‌ها نیز کل مجموعه را تشکیل می‌دهد.

ردیف سوم: تعداد تکرار element مورد نظر که صفر یا ۱ یا بیش از این، با هم اشتراکی ندارند و اجتماع آن‌ها نیز کل مجموعه را تشکیل می‌دهد.

ج) برای ترکیب بلاک‌ها از روش Each Choice Coverage یا همان ECC استفاده می‌کنم.

	List = null	List = [ ]	اولین element عضو	آخرین عضو element	تعداد تکرار element
1	False	True	False	False	0
2	False	False	True	False	1
3	False	False	False	True	> 1
4	True	False	False	False	0

ردیف اول:

list = [ ]

element = 1

output = -1

ردیف دوم:

list = [1, 2, 3]

element = 1

output = 0

ردیف سوم:

list = [1, 2, 3, 3]

element = 3

output = 2 or 3

ردیف چهارم:

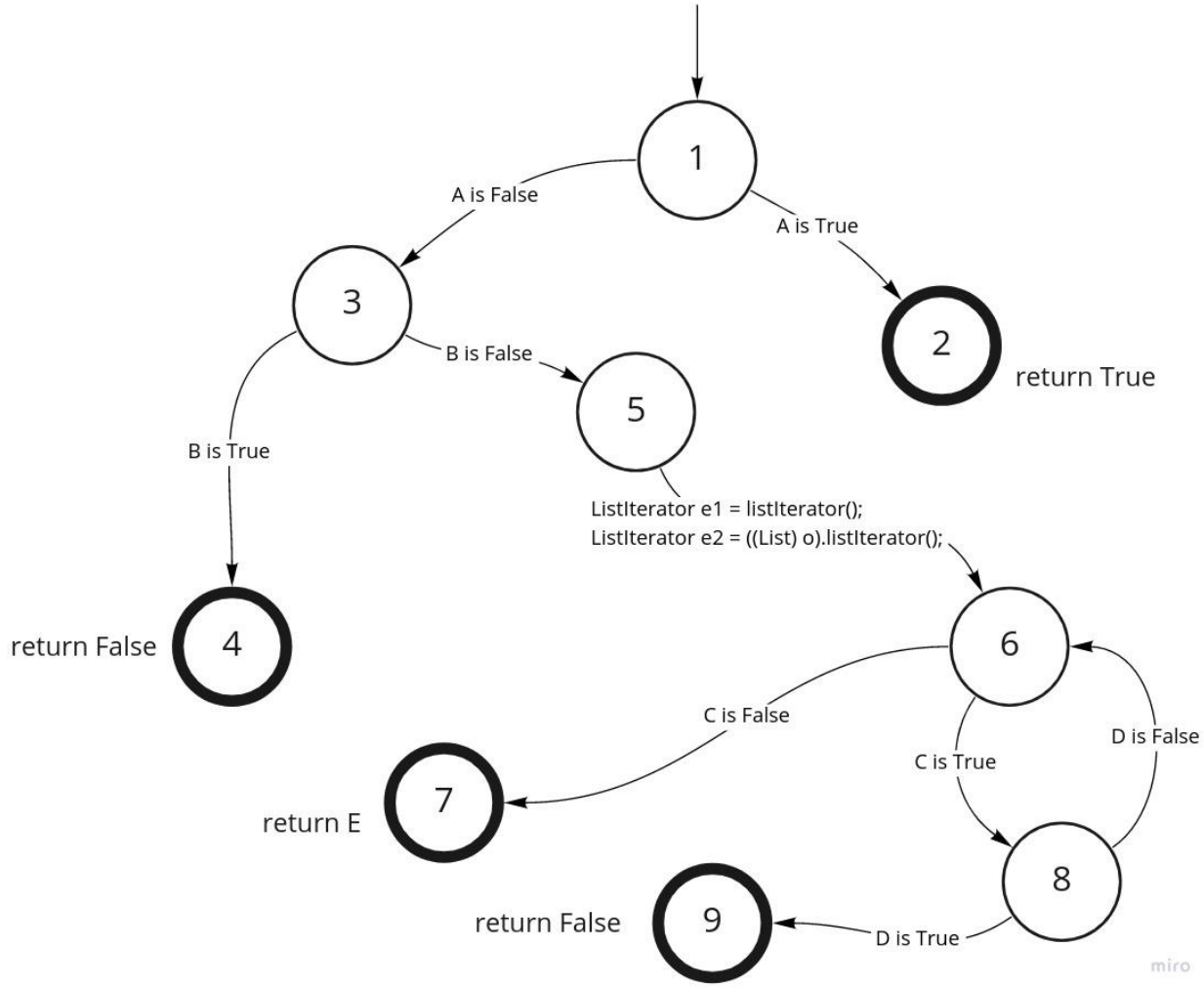
list = null

element = NullPointerException

---

۶)

الف) گراف جریان ورودی به شکل زیر است:



موارد اختصاری‌ای که روی یال‌ها قرار گرفته‌اند همان موارد ذکر شده در صورت سوال هستند یعنی:

A: o == this

B: !(o instanceof List)

C: e1.hasNext() && e2.hasNext()

D: !(o1 == null ? o2 == null : o1.equals (o2))

E: !(e1.hasNext() || e2.hasNext())

ب) حداقل نیاز به ۴ تست است که مسیرهای زیر هستند:

مسیر اول: 1 → 2

مسیر دوم: 1 → 3 → 4

مسیر سوم: 1 → 3 → 5 → 6 → 7

مسیر چهارم: 1 → 3 → 5 → 6 → 8 → 9

ج) نیازمندی‌های آزمون برای پوشش جفت-یال به صورت زیر اند:

[1, 2], [1, 3, 4], [1, 3, 5], [3, 5, 6], [5, 6, 7],
[5, 6, 8], [6, 8, 6], [8, 6, 8], [6, 8, 9], [8, 6, 7]

مقادیر آزمون برای این پوشش به صورت زیر است:

[1, 2], [1, 3, 4], [1, 3, 5, 6, 7], [1, 3, 5, 6, 8, 6, 8, 9], [1, 3, 5, 6, 8, 6, 7]

---

۷)

الف) عبارت منطقی‌ای که در این تابع تست می‌شود، عبارت زیر است:

$$P = (g \ \& \ f) \mid (f \ \& \ c) \mid (g \ \& \ c)$$

جدول درستی این عبارت، به صورت زیر است:

	g	f	c	P
1	1	1	1	1
2	1	1	0	1
3	1	0	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	0
8	0	0	0	0

با توجه به جدول درستی، آزمون‌های خواسته شده‌ی زیر را داریم:

GACC:

	test
g	(2,6), (2,7), (3,6), (3,7)
f	(2,4), (2,7), (5,4), (5,7)
c	(3,4), (3,6), (5,4), (5,6)

RACC:

	test
g	(2,6), (3,7)
f	(2,4), (5,7)
c	(3,4), (5,6)

ب) در قسمت قبل تمامی حالت‌ها پوشش داده شده‌اند و برای این قسمت آزمون جدیدی نیاز نیست.

---

۸) TDD یا همان Test Driven Development به این صورت است که ما سعی می‌کنیم توسعه را بر اساس تست‌ها انجام دهیم یعنی کد را برای پاس‌کردن تست‌ها می‌نویسیم و سعی می‌کنیم به صورت پیوسته کدها را بر اساس تست‌ها ریفکتور کنیم.

و اما BDD یا همان Behaviour Driven Development به این صورت است که بر اساس رفتار سیستم تعریف می‌شود و در آن یک نیازمندی قابل اجرا تعریف می‌کنیم که چون پیاده‌سازی نشده منجر به شکست در نرم‌افزار می‌شود و بعد از آن سعی می‌کنیم به شکلی ساده کدی بنویسیم که باعث پاس شدن آن نیازمندی شود و این‌کار تا زمانی که یک نسخه برای ارائه ایجاد شود، ادامه می‌دهیم.

در راستای مقایسه‌ی این دو روش باید بگوییم که BDD روشی است که از روی TDD به وجود آمده است و هدفشان این است که کدی زده شود که به خوبی تست شده است ولی در راستای اینکه به نیازهای کاربران هم‌خوانی بیشتری داشته باشد و در آن راستا باشد BDD برای تست User Storyها ایجاد شد یعنی تست‌ها همان نیازهای کاربران است ولی در TDD ما جز به جز تلاش می‌کنیم تست‌هایی برای داشتن یک کد مناسب داشته باشیم.