حوری دهش 9821413

سوال 1:

(الف)

سطح 0:

هیچ تفاوتی بین تست و دیباگ کردن وجود ندارد یعنی انجام تست ها به منظور یافتن و رفع خطاها است.

در این سطح یک نفر دیدش به این صورت است که تست دارم انجام میدهم و این همان معادل دیباگ کردن کد است یعنی کد را می نویسیم و بعد آن دیباگ میکنیم یعنی آن را با ورودی هایی که کارفرما به ما داده است تست میکنیم و ازش ران میگیریم

نمی توانیم از همچین سطح فکری که شخص راجع به تست داره انتظار داشته باشیم که نرم افزارش یک نرم افزار قابل اعتماد یا ایمن باشد.

سطح 1:

در این سطح می خواهیم نشان بدهیم که برنامه درست است یعنی نرم افزار باید مطابق با نیازمندی ها عمل کند. (هدف)

در این سطح از خودمون ورودی تولید میکنیم و به برنامه می دهیم تا ببینیم نتیجه درست است یا نه

فرقش با حالت قبلی این است که توی سطح 0 تست ها محدود است و به برنامه همان ورودی هایی داده میشود که از کارفرما دریافت کرده ایم ولی توی سطح 1 خودمون یکسری ورودی هایی علاوه بر انهایی که از کارفرما گرفتیم می سازیم

سطح 2:

در این سطح می خواهیم نشان بدهیم که برنامه درست نیست یعنی تست ها باید مواردی را که به درستی عمل نمی‌کنند، شناسایی کنند. (هدف)

تستر دنبال این است که روی نقاط چالش برانگیز دست بذاره تا failure های سیستم رو تشخیص بدهد.

اینجا ممکن است بین تستر و برنامه نویس دعوا پیش بیاد یعنی برنامه نویس ها ممکنه فکر کنند که تسترها رقیب هایشان هستند و بخواهند مچ برنامه نویس ها را بگیرند ولی سطح بالغ تر فکر اینه که برنامه نویس ها، تسترها را به عنوان همکار ببینند. (که این تیکه آخر که به صورت همکار همدیگر را می بینند در سطح بعدی گفته میشود)

سطح 3:

هدف از تست کردن ارائه چیز خاصی نیست بلکه کاهش خطر استفاده از نرم افزار است. (هدف)

تستر و برنامه نویس با هم دوست هستند و با هم همکاری می کنند.

ما به عنوان یک مهندس نرم افزار می دانیم که اگر Verification انجام ندهیم و در نهایت فقط متکی به تست کردن نرم افزار باشیم این کار یک ریسکی دارد و ما این ریسک را می پذیریم --> حالا توی بعضی از نرم افزارها این ریسک خیلی کوچیک است و عواقب کمی دارد ولی توی بعضی هاش بزرگ است

هدف این است که ریسک استفاده از نرم افزار کاهش پیدا کند.

سطح 4:

تست یک رویکرد فکری است که به همه متخصصان فناوری اطلاعات کمک می کند تا نرم افزارهای با کیفیت بالاتر را توسعه دهند. (هدف)

همه می دانند که تست فقط این نیست که ورودی بدهیم و خروجی بگیریم بلکه باید non functional requirementها را هم لحاظ بکنیم و این باید به عنوان یک جریان فکری باشد که از همان ابتدای کار که داریم requirementها رو درمیاریم، تسترها هم درگیر باشند چون می خواهیم یک نرم افزار باکیفیت داشته باشیم.

(ب)

فرق Validation و Verification:

Validation: یعنی کاربر ما که یکسری نیازهایی دارد و ما این ها را تبدیل میکنیم به یک اپلیکیشنی که دارد این نیاز ها را جواب میدهد --> این یعنی بریم ببینیم دسته اخر اون نیازهای کارفرما براورده شده است یا نه

محصول حتما باید نهایی بشه و بعد Validation انجام بشود

Validation رو وقتی که میخواهیم انجام بدهیم باید بر نیازهای واقعی محیطمون واقف باشیم یعنی اگر قراره ما نرم افزار هواپیما رو بنویسیم، Validation رو کسی باید انجام بدهد که با اصول خلبانی اشنا باشد پس کسی باید Validation رو انجام بدهد که مسلط باشد به domain knowledge

این فرایند بیشتر از طریق تست های کاربردی، آزمایش های سناریویی و ارزیابی عملکرد نرم افزار با استفاده از داده های واقعی صورت می گیرد.

Verification: به این معناست که آیا نرم‌افزار مورد نظر، نیازمندی ها و مشخصات فنی را به درستی پیاده سازی کرده است یا نه به عبارت دیگر Verification بررسی می کند که نرم افزار به طور صحیح از لحاظ فنی طراحی و پیاده سازی شده باشد.

این فرایند بیشتر از طریق مطابقت با مشخصات فنی، استانداردها و رویه های مشخص و تحلیل کد انجام می شود.

Verification در انتهای محصول نیست بلکه در مراحل داخلی آن است و میاد مراحل داخلی را یکی یکی بررسی و چک میکند.

Verification براساس مدل های ریاضی اثبات میکند که نرم افزار درست است و همانی است که میخواهیم.

در Verification همه مسیرها چک می شود ولی در تست نمی توانیم همه مسیرها را چک بکنیم.

پس Verification بررسی می کند که آیا نرم افزار به درستی ساخته شده است در حالی که Validation بررسی می کند که آیا نرم افزار ساخته شده به درستی کار می کند و نیازهای واقعی کاربران را برآورده می کند یا نه.

نکته: Verification رو فقط برای نرم افزارهایی که حساس هستن انجام میدهیم چون بسیار زمان بر است.

نکته: جفتش رو برای نرم افزار می خواهیم ولی حضور Verification الزامی نیست ولی Validation رو حتما میخواهیم --> Verification همیشه لازم نیست و وقتی که لازم است که میخواهیم ریسک استفاده کردن از نرم افزار رو به شدت کاهش بدهیم.

(ج)

Fault: خطایی است که در کد یا طراحی نرم‌افزار وجود دارد و منجر به بروز مشکلات در عملکرد نرم افزار می شود.

بعضی وقت ها درون نرم افزار خرابکاری هایی صورت می گیرد ولی ممکن است نتیجه نهایی کد با وجود همچین خرابکاری هایی درست دربیاید.

Fault سر منشا تمام اورها است.

Fault یعنی اشتباهات طراحی که بعدا باعث میشود ارور به وجود بیاید.

Faultها معمولا به عنوان نقاط ضعف یا اشکال در کد منبع یا طراحی نرم افزار تشخیص داده می‌شوند اما انها ممکن است در زمان اجرا مشخص نشوند.

Failure: نتیجه مستقیمی از Faultها هستند و ممکن است منجر به خطاهای قابل مشاهده توسط کاربران یا مشکلات اجرایی شود یعنی توی رفتار ظاهری نرم افزار خرابکاری را مشاهده کردیم و به خطا خوردیم.

Faultها در کد یا طراحی مشخص می شوند در حالی که Failureها در زمان اجرا یا استفاده از نرم افزار به وقوع می پیوندند و نشان دهنده عملکرد نادرست ان می باشد.

(د)

کد دارای fault:

```
bool isPrime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i < sqrt(n); i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

تست کیسی که منجر به failure میشود: عدد 4 است چون عدد 4 را عدد اول تشخیص میدهد.

رفع این fault به صورت زیر است:

```
bool isPrime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

(ه)

الف:

نمی توانیم تمام فضای ورودی را تست کنیم به علت این که فضای ورودی ممکن است بسیار بزرگ باشد مثلا اگر بخواهیم هر ترکیبی ممکن از تمام اعداد صحیح و اعشاری را برای ورودی ها تست کنیم، این کار بسیار پیچیده و بزرگ می شود و حتی می تواند محدودیت هایی را داشته باشد --> فضای ورودی برای یک عدد صحیح بسیار بزرگ است چون شامل اعداد منفی و مثبت می باشد به همین علت تست کردن هر ترکیبی از این مقادیر به طور کامل، غیر ممکن است و برای اعداد اعشاری هم به علت دقت محدود و اعداد ممیز شناور تست کردن تمام مقادیر مشکل و حتی ممکن است غیر ممکن باشد.

ب:

با استفاده از معیارهای پوشش می توانیم تعدادی از حالت های مهم و حیاتی فضای ورودی را پوشش دهیم و می توانیم تست هایی را طراحی کنیم که به طور موثر فضای ورودی را پوشش داده و مشکلات موجود در متد را شناسایی می کنند. بعضی از معیارها عبارتند از:

Path Coverage: با استفاده از این معیار می توانیم تمام مسیرهای مختلف اجرای متد را پوشش دهیم.

Condition Coverage: با تست کردن تمام شرایط مختلف در متد می توانیم مطمئن شویم که هر قسمت از متد به درستی کار می کند.

Boundary Coverage: با این معیار می توانیم مقادیر مرزی و حالت های ویژه ای که ممکن است به مشکل بربخورند را تست بکنیم و همچنین برای این متد می توانیم مقادیر حداقلی و حداکثری برای int , float را تست کنیم.

(ی)

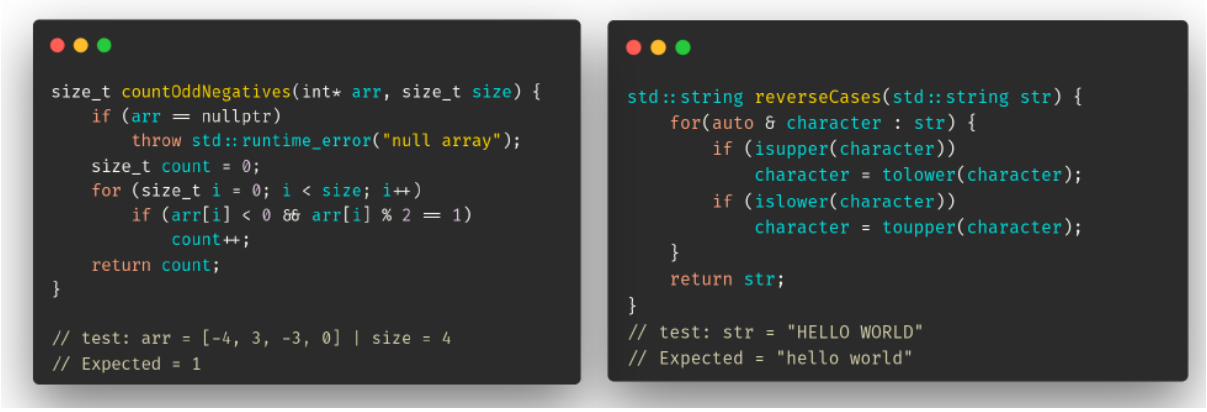
Reachability: اول از همه داخل اون تستی که داریم، به اون نقطه برسیم مثلا اگر حلقه for توی یک تابعی است که این تابع در تست هیچ وقت فراخوانی نشود یعنی تست را جوری نوشتیم که این تابع هیچ وقت صدا زده نمیشود پس در این حالت fault هیچ وقت تبدیل به Failure نمیشود پس اولین شرط این هست که به اون مکان برسیم --> پس برنامه باید به گونه ای اجرا شود که به اون قسمت های دارای خطا برسد.

Infection: یعنی الوده بشود --> ینی تاثیرشو روی استیت داخلی بگذارد به عبارت دیگر fault تاثیرشو روی استیت داخلی بگذارد پس باید خطا در حالت اجرایی برنامه قرار بگیرد.

Propagation: یعنی این استیت S2 که استیت مورد انتظار نبوده، بیاد و پخش بشود یعنی استیت بعدی هم که به عنوان سومین استیت داریم، مورد انتظارمون باز نبوده و استیت 4 هم به همین صورت و.. یعنی هیچ وقت به اون مسیر درست برنگردد --> پس وضعیت غلطی که در مرحله Infection ایجاد شده است باید به گونه ای باشد که خروجی یا وضعیت نهایی برنامه اشتباه باشد.

Reveal : اشکار شدن --> یعنی استیت نهایی خراب شده است ولی کاربر این را نمی بیند به عبارت دیگر از دید تستر پنهان مانده است پس وقتی از دید تستر پنهان بماند یعنی Reveal نشده است پس در این مرحله تستر باید بخشی از وضعیت نادرست برنامه را مشاهده کند یا بررسی کند. این بخش شامل توجیهات و تحلیل هایی است که تست گذار به دنبال یافتن علائم خطا و نشانه های ناهنجار در وضعیت برنامه است.

سوال 2:



```
size_t countOddNegatives(int* arr, size_t size) {
    if (arr == nullptr)
        throw std::runtime_error("null array");
    size_t count = 0;
    for (size_t i = 0; i < size; i++)
        if (arr[i] < 0 && arr[i] % 2 == 1)
            count++;
    return count;
}

// test: arr = [-4, 3, -3, 0] | size = 4
// Expected = 1
```

```
std::string reverseCases(std::string str) {
    for(auto & character : str) {
        if (isupper(character))
            character = tolower(character);
        if (islower(character))
            character = toupper(character);
    }
    return str;
}
// test: str = "HELLO WORLD"
// Expected = "hello world"
```

(الف)

کد سمت چپی:

:fault

در داخل for برای بررسی فرد و منفی بودن عدد از % استفاده شده است که همین باعث خطا در جواب نهایی کد میشود زیرا % برای اعداد منفی به درستی عمل نمی کند به علت اینکه باقی مانده عدد منفی فرد همیشه منفی خواهد شد.

خروجی:

با توجه به تست کیسی که درون سوال قرار دارد خروجی برای این تابع 0 خواهد شد در صورتی که این تابع بررسی می کند که ایا عدد مورد نظر فرد و منفی است یا نیست که ما اینجا عدد 3- را داشتیم ولی به علت اینکه باقی مانده 3- بر 2 عدد 1- میشود پس شرط if برقرار نمی باشد.

کد سمت راستی:

:fault

مشکلی که اینجا وجود دارد بخاطر دوتا if هست که پشت سر هم آمده چون در if اول اگر حرفی بزرگ باشد به حرف کوچک تبدیل شده و در if بعد چون حرف کوچکی می بیند ان را دوباره به حرف بزرگ تبدیل می کند که این خروجی مورد انتظار ما نیست پس مشکل این کد در استفاده از دو شرط if مستقل برای تغییر حالت حروف است که باعث عدم تغییری در رشته ممکن است بشود.

خروجی:

با توجه به تست کیسی که درون سوال قرار دارد خروجی برای این تابع همان HELLO WORLD میشود:

مثلا برای H ابتدا H وارد if اولی شده و را به h تبدیل میکند و همین h در if دومی وارد شده و به H تبدیل میشود و برای بقیه حروف هم به همین صورت

(ب)

fault نداشته باشد:

کد سمت چپی:

تست کیس: [] , size=0

خروجی: null array

کد سمت راستی:

تست کیس: "" --> یک رشته خالی باشد پس حلقه اصلا اجرا نمیشود چون هیچ کاراکتری نداریم

خروجی: یک رشته خالی چاپ میشود یعنی هیچی چاپ میشود

fault داشته باشد ولی ارور نداشته باشد:

کد سمت چپی:

تست کیس: size=4 , [4,7,1,-8]

خروجی: 0

کد سمت راستی:

تست کیس: "12%345"

خروجی: "12%345"

ارور داشته باشد ولی failure نداشته باشد:

کد سمت چپی:

وجود ندارد

کد سمت راستی:

وجود ندارد

(ج)

کد سمت چپی:

شرط داخل if را به صورت زیر تغییر می دهیم:

```
size_t countOddNegatives(int* arr, size_t size) {
    if (arr == nullptr) {
        throw std::runtime_error("null array");
    }
    size_t count = 0;
    for (size_t i = 0; i < size; i++) {
        if (arr[i] < 0 && arr[i] % 2 != 0) {
            count++;
        }
    }
    return count;
}
```

در این حالت خروجی مورد انتظار 1 می شود چون 3- در این شرط قابل قبول است.

کد سمت راستی:

if دومی را به else if تبدیل می کنیم با این تغییر هر کاراکتر تنها یک بار بررسی می شود و تنها یکی از شرایط اعمال میشود:

```
std::string reverseCases(std::string str) {
    for (auto &character : str) {
        if (isupper(character)) {
            character = tolower(character);
        } else if (islower(character)) {
            character = toupper(character);
        }
    }
    return str;
}
```

در این حالت خروجی مورد انتظار hello world می شود.

سوال 3:

(الف)

مشکلی که در کد وجود دارد این است که متد equals در کلاس ColorPoint فقط بررسی میکند که ایا شی ورودی از نوع ColorPoint است یا نه و این باعث میشود که p.equals(cp1) برابر با true شود ولی cp1.equals(p) برابر با false میشود که این قاعده تقارن در متد equals را نقض میکند پس باید این بررسی هم انجام شود که ایا شی ورودی از نوع Point هم هست یا نه.

اصلاح کد به صورت زیر است:

```
@Override public boolean equals(Object o){
        //location B
        if(!(o instanceof Point))
                return false;
        if(!(o instanceof ColorPoint))
                return o.equals(this);
        ColorPoint cp =(ColorPoint) o;
        return (super.equals(cp)&&(cp.color==this.color));
}
```

در این کد اگر o از نوع Point باشد مقدار o.equals(this) ریترن میشود که این باعث میشود که p.equals(cp1) و cp1.equals(p) هر دو true شوند و cp1.equals(cp2) برابر با false شود که این جواب با انتظارات ما یکی است.

(ب)

تست کیس:

```
Point p1 = new Point(1, 5);
Point p2 = new Point(1, 5);
p1.equals(p2);  // result=true , expected=true
```

این تست کیس هیچ وقت به خطی که دارای fault هست نمیرسد بخاطر اینکه در این تست کیس فقط دو نمونه از همان نوع با هم مقایسه میشوند یعنی یک Point با یک Point دیگر مقایسه میشود.

از طرفی ما زمانی fault داریم که یک نمونه از کلاس Point با یک نمونه از کلاس ColorPoint مقایسه شود.

(ج)

تست کیس:

```
ColorPoint cp1 = new ColorPoint(1, 5, RED);
ColorPoint cp2 = new ColorPoint(1, 5, RED);
cp1.equals(cp2);  // result=true , expected=true
```

(د)

تست کیس:

```
Point p = new Point(7, 8);
ColorPoint cp = new ColorPoint(1, 5, BLUE);
cp.equals(p);  // result=false , expected=false
```

(ه)

اولین حالت خطا یا error state در کد زمانی رخ می دهد که متد equals در کلاس ColorPoint فراخوانی میشود و شی ورودی از نوع Point است و نه ColorPoint پس در این متد ابتدا بررسی میشود که ایا شی ورودی از نوع ColorPoint است یا نه که این کار با استفاده از ((o instanceof ColorPoint)!)if انجام میشود و اگر شی ورودی از نوع ColorPoint نباشد متد equals مقدار false را ریترن میکند ولی اگر شی ورودی از نوع ColorPoint باشد سعی میکند ان را به یک ColorPoint با استفاده از دستور زیر تبدیل کند:

ColorPoint cp = (ColorPoint) o

حالا چون شی ورودی از نوع Point است و نه ColorPoint زمانی که میخواهد به یک ColorPoint تبدیل شود باعث ایجاد ارور می شود به علت این که یک نمونه از کلاس Point نمی تواند به یک ColorPoint تبدیل شود چون ColorPoint یک زیرکلاس از کلاس Point است و ویژگی های اضافی دارد مثل فیلد Color و از طرف دیگر هم استیت داخلی برنامه غلط میشود و ان استیتی نیست که مورد انتظار ما بوده چون pc برنامه هم این وسط غلط میشود و این یعنی استیت برنامه غلط شده است که در نهایت به ارور می رسد.

همچنین این ارور توسط تست کیس دوم یعنی cp1.equals(p) رخ میدهد چون در این تست کیس سعی میشود یک نمونه از کلاس ColorPoint cp1 را با یک نمونه از کلاس Point p مقایسه کند اما چون p از نوع Point است و نه ColorPoint، تلاش برای تبدیل p به یک ColorPoint باعث ایجاد ارور میشود.