نام اعضای تیم و شماره دانشجویی‌ها

سید ابوالحسن رضوی (۴۰۲۲۱۲۶۵۵)
ایمان محمدی (۹۹۱۰۲۲۰۷)
علی اسلامی نژاد (۴۰۲۲۱۱۷۸۹)
شماره گروه: ۲۰

جواب سوال ۱

جدول درستی عبارت P

جدول درستی برای عبارت $P: (a \wedge b) \vee (\neg a \wedge b)$ به شرح زیر است:

a	b	P
True	True	True
True	False	False
False	True	True
False	False	False

نیازمندی های آزمون برای پوشش Predicate و Clause

- **Predicate Coverage**:
  - P صحیح: زمانی که a و b هر دو صحیح یا a غلط و b صحیح باشد.
  - P غلط: زمانی که a صحیح و b غلط باشد یا هر دو غلط باشند.
- **Clause Coverage**:
  - برای $a \wedge b$: برآورده می‌شود زمانی که a و b هر دو صحیح باشند (True) و زمانی که a یا b (یا هر دو) غلط باشند (False) .
  - برای $\neg a \wedge b$: برآورده می‌شود زمانی که a غلط و b صحیح باشد (True) و زمانی که b غلط باشد (False) .

جواب سوال ۲

عبارات منطقی معادل برنامه‌ها

۱. انتخاب عدد میانی از بین سه عدد

فرض کنید سه عدد a ، b ، و c داریم. عدد میانی m به صورت زیر تعریف می‌شود:

$$m = \begin{cases} a & \text{اگر} (a > b \land a < c) \lor (a < b \land a > c) \\ b & \text{اگر} (b > a \land b < c) \lor (b < a \land b > c) \\ c & \text{اگر} (c > a \land c < b) \lor (c < a \land c > b) \end{cases}$$

۲. تشخیص زوج یا فرد بودن یک عدد

یک عدد n زوج است اگر:

$$\text{زوج}(n) \Leftrightarrow n \mod ۲ = ۰$$

و فرد است اگر:

$$\text{فرد}(n) \Leftrightarrow n \mod ۲ \neq ۰$$

۳. قدر مطلق تفاضل میان دو عدد

قدر مطلق تفاضل دو عدد x و y به صورت زیر است:

$$|x - y| = \begin{cases} x - y & \text{اگر} x > y \\ y - x & \text{اگر} y \geq x \end{cases}$$

جواب سوال ۳

سوال ۱: مقایسه مفاهیم ،fault ،error و failure

Fault (عیب) ، Error (خطا) و Failure (شکست)، سه مفهوم اصلی در تست نرم‌افزار هستند که برای درک عمیق‌تر علل و پیامدهای بروز مشکل در نرم‌افزارها، مورد بررسی قرار می‌گیرند.

- Fault (عیب) به مشکل یا نقص موجود در کد نرم‌افزار یا در مستندات آن اشاره دارد که می‌تواند زمینه‌ساز بروز خطا شود. این نقص‌ها به دلیل تصمیمات طراحی یا پیاده‌سازی نادرست توسط توسعه‌دهندگان نرم‌افزار ایجاد می‌شوند.
- Error (خطا) وقتی اتفاق می‌افتد که نرم‌افزار در حین اجرا با عیب (Fault) مواجه شده و به دلیل آن، حالت داخلی نرم‌افزار از حالت مورد انتظار منحرف می‌شود. خطا لزوماً به معنای بروز شکست نیست اما نشان‌دهنده انحراف از رفتار مطلوب است.
- *Failure* (شکست) زمانی رخ می‌دهد که نرم‌افزار نتواند عملکرد مورد انتظار را ارائه دهد و خروجی نرم‌افزار با خروجی مورد انتظار مطابقت نداشته باشد. شکست، نتیجه مشاهده شده از بروز یک یا چند خطا است.

سوال ۲: مدل RIPR

مدل RIPR یک چارچوب برای فهم و تحلیل فرآیند بروز شکست در نرم‌افزار است و شامل چهار شرط اصلی است: Reachability (قابلیت دسترسی)، Infection (آلودگی)، Propagation (انتشار)، و Revelation (آشکارسازی).

الف) Reachability (قابلیت دسترسی): تست باید قادر باشد نقطه‌ای از کد که عیب در آن وجود دارد را فعال ک

ب) Infection (آلودگی): پس از رسیدن به نقطه‌ای که عیب در آن وجود دارد، حالت برنامه باید به گونه‌ای تغییر کند که نشان‌دهنده وجود خطا باشد.

ج) Propagation (انتشار): حالت آلوده باید از طریق اجرای برنامه منتشر شود تا به خروجی یا وضعیت نهایی برنامه برسد و آن را به گونه‌ای اشتباه تحت تأثیر قرار دهد.

د) Revelation (آشکارسازی): تست باید قادر به آشکارسازی شکست باشد، به این معنا که شکست باید به روشنی قابل مشاهده و قابل تشخیص باشدند.

ه) Infection (آلودگی): پس از رسیدن به نقطه‌ای که عیب در آن وجود دارد، حالت برنامه باید به گونه‌ای تغییر کند که نشان‌دهنده وجود خطا باشد.

و) Propagation (انتشار): حالت آلوده باید از طریق اجرای برنامه منتشر شود تا به خروجی یا وضعیت نهایی برنامه برسد و آن را به گونه‌ای اشتباه تحت تأثیر قرار دهد.

ز) Revelation (آشکارسازی): تست باید قادر به آشکارسازی شکست باشد، به این معنا که شکست باید به روشنی قابل مشاهده و قابل تشخیص باشد.

این مدل نشان می‌دهد که برای مشاهده یک شکست، همه این چهار شرط باید برآورده شوند. این یک فرآیند پیچیده است که نشان‌دهنده چالش‌های موجود در تست نرم‌افزار و اهمیت طراحی دقیق تست‌ها برای شناسایی عیب‌ها است.

سوال ۳: فرآیند Model-driven Test Design

Model-driven Test Design (طراحی تست مبتنی بر مدل) یک رویکرد سیستماتیک برای تولید موارد تست است که در آن مدل‌های فرمال از سیستم به عنوان اساس برای طراحی تست‌ها استفاده می‌شوند. این رویکرد به توسعه‌دهندگان اجازه می‌دهد تا پوشش دهی تست‌ها را به طور دقیق‌تری کنترل کنند و اطمینان حاصل کنند که تمام جنبه‌های مهم سیستم تست شده‌اند.

فرآیند طراحی تست مبتنی بر مدل عموماً شامل مراحل زیر است:

الف) *ایجاد مدل*: توسعه مدل‌هایی که جنبه‌های مختلف سیستم را نشان می‌دهند، مانند رفتار، ساختار، وابستگی‌ها، و سناریوهای استفاده.

ب) *تعریف معیارهای پوشش*: تعیین معیارهایی برای ارزیابی کامل بودن تست‌ها، مانند پوشش دستورات، شرایط، یا مسیرهای اجرایی.

ج) *تولید موارد تست*: استفاده از مدل‌ها برای تولید خودکار موارد تست که بر اساس معیارهای پوشش تعریف شده هستند.

د) *اجرای تست و تحلیل نتایج*: اجرای تست‌ها بر روی سیستم و تحلیل نتایج برای شناسایی و رفع عیب‌ها. این مرحله شامل بررسی نتایج تست‌ها برای تعیین اینکه آیا سیستم مطابق با مشخصات و الزامات عمل کرده است یا خیر.

این رویکرد به تیم‌های توسعه امکان می‌دهد تا تست‌های دقیق و کامل‌تری را با استفاده از منابع محدود انجام دهند. با تمرکز بر مدل‌ها، تسترها می‌توانند اطمینان حاصل کنند که تمام جنبه‌های مهم سیستم تحت پوشش قرار گرفته‌اند و احتمال عبور عیب‌ها از فیلتر تست‌ها را به حداقل می‌رسانند.

استفاده از Model-driven Test Design نیازمند درک عمیقی از سیستم و توانایی تبدیل دانش به مدل‌های دقیق و قابل استفاده برای تست است. این رویکرد همچنین به ابزارهای تخصصی برای تولید خودکار موارد تست از مدل‌ها و اجرای آن‌ها بر روی سیستم نیاز دارد.

جواب سوال ۴

مقایسه مفاهیم از لحاظ ریزدانگی و درشت‌دانگی

در حوزه آزمون نرم‌افزار، مفاهیم مختلفی وجود دارند که از لحاظ سطح بررسی و تحلیل (ریزدانگی و درشت‌دانگی) با یکدیگر متفاوت هستند. این تفاوت‌ها در معماری‌های Microservices و Monolithic به‌خوبی قابل مشاهده است.

- Unit Testing: در معماری Microservices ، آزمون واحد به دلیل تقسیم‌بندی و ماژولاریتی بالا، اهمیت بیشتری پیدا می‌کند. در معماری Monolithic ، آزمون واحد ممکن است به دلیل وابستگی‌های متقابل درونی پیچیده‌تر باشد.
- Integration Testing: در Microservices ، آزمون ادغام عمدتاً بر روی تعاملات بین سرویس‌ها تمرکز دارد، در حالی که در Monolithic ، بر روی تعاملات بین مؤلفه‌ها در یک سیستم واحد تمرکز می‌کند.
- E2E Testing و Acceptance Testing: هر دو در معماری‌های Microservices و Monolithic اهمیت دارند، اما در Microservices ، تست انتها به انتها ممکن است شامل تعاملات پیچیده‌تر و بیشتری بین سرویس‌ها باشد.
- Black Box و White Box Testing : این دو نوع آزمون از نظر دانش آزمون‌دهنده از کد مورد بررسی قرار می‌گیرند و در هر دو معماری کاربرد دارند، اما در Microservices ، آزمون جعبه سفید ممکن است بر روی ماژولاریتی و API ها تمرکز بیشتری داشته باشد.
- Happy Path Testing: این روش آزمون بر سناریوهایی تمرکز دارد که در آنها همه چیز طبق انتظار پیش می‌رود. این نوع آزمون در هر دو معماری مهم است، اما در معماری Microservices ، توجه خاصی به تضمین اینکه تعاملات بین سرویس‌ها به خوبی کار می‌کنند، نیاز دارد.
- Exceptional Testing: آزمون استثنایی به بررسی نحوه رفتار نرم‌افزار در شرایط خطا می‌پردازد. در Microservices ، اهمیت دارد که چگونگی مدیریت خطاها و استثناها در سرویس‌های مختلف و تعاملات بین آنها بررسی شود. در معماری Monolithic ، بررسی چگونگی مدیریت خطاهای داخلی و وابستگی‌ها مهم است.
- Mocking: این تکنیک به شبیه‌سازی بخش‌هایی از سیستم برای آزمون بدون نیاز به وابستگی‌های واقعی اشاره دارد. در معماری Microservices ، Mocking برای شبیه‌سازی سرویس‌هایی که یک سرویس به آنها وابسته است، کاربرد دارد. در معماری Monolithic ، Mocking می‌تواند برای شبیه‌سازی پایگاه داده‌ها یا سایر مؤلفه‌های داخلی استفاده شود.
- TDD (Test-Driven Development): این رویکرد که در آن تست‌ها قبل از نوشتن کد نوشته می‌شوند، در هر دو معماری کاربردی است. در Microservices ، TDD می‌تواند به تضمین اینکه هر سرویس به صورت مستقل قابل آزمون و توسعه باشد کمک کند. در معماری Monolithic ، TDD به اطمینان از اینکه تغییرات در یک بخش از سیستم باعث ایجاد مشکل در بخش‌های دیگر نمی‌شود، کمک می‌کند.

با توجه به این توضیحات، می‌توان دید که چگونه هر یک از این مفاهیم آزمون نرم‌افزار، با توجه به سطح ریزدانگی و درشت‌دانگی، در معماری‌های Microservices و Monolithic کاربرد دارند. استفاده از این تکنیک‌ها و رویکردها در معماری‌های مختلف نه تنها به افزایش کیفیت و استحکام نرم‌افزار کمک می‌کند، بلکه اطمینان حاصل می‌شود که نرم‌افزار در برابر شرایط و سناریوهای مختلف به درستی عمل می‌کند.

در معماری Microservices ، تمرکز بر تست‌های مستقل برای هر سرویس و نیز بررسی تعاملات و ادغام آن‌ها با یکدیگر، اهمیت زیادی دارد. این موضوع باعث می‌شود که Mocking و Integration Testing به ابزارهای حیاتی برای تضمین عملکرد صحیح سیستم در سطح کلی تبدیل شوند. از سوی دیگر، در معماری Monolithic ، که اجزای مختلف نرم‌افزار به شدت با یکدیگر ادغام شده‌اند، آزمون‌های واحد و تست‌های ادغام داخلی برای شناسایی و حل مشکلات قبل از اینکه آن‌ها تاثیر گسترده‌ای بر سیستم داشته باشند، بسیار حیاتی هستند.

Happy Path Testing و Exceptional Testing نیز در هر دو معماری نقش مهمی دارند. این تست‌ها به تیم‌های توسعه اجازه می‌دهند تا از پایداری و قابلیت اطمینان نرم‌افزار در شرایط مختلف اطمینان حاصل کنند. Happy Path Testing بررسی می‌کند که نرم‌افزار در شرایط ایده‌آل چگونه عمل می‌کند، در حالی که Excep-tional Testing به چالش کشیدن نرم‌افزار در شرایط استثنایی و خطاها را بررسی می‌کند.

TDD (Test-Driven Development) به عنوان یک فلسفه و رویکرد در توسعه نرم‌افزار، به توسعه‌دهندگان کمک می‌کند تا با نوشتن تست‌ها قبل از کد، از ابتدا تمرکز خود را بر کیفیت و عملکرد متمرکز کنند. این رویکرد در هر دو معماری می‌تواند به ایجاد کدی منجر شود که نه تنها مطابق با نیازمندی‌ها است، بلکه از ابتدا برای آزمون و نگهداری آسان‌تر طراحی شده است.

در نهایت، انتخاب و به‌کارگیری این مفاهیم آزمون نرم‌افزار، بسته به اهداف خاص پروژه، معماری انتخابی، و محدودیت‌های موجود متفاوت خواهد بود. تفاهم و اجرای دقیق این تکنیک‌ها در معماری‌های Microservices و Monolithic به توسعه‌دهندگان کمک می‌کند تا با اطمینان بیشتری به سمت تولید نرم‌افزارهایی با کیفیت بالا و قابل اطمینان حرکت کنند.

توضیح مفاهیم با مثال

Testing Unit

در Microservices تست یک سرویس احراز هویت می‌تواند نمونه‌ای از آزمون واحد باشد. در Monolithic تست یک تابع محاسبه مالیات مثالی از آزمون واحد است.

Integration Testing

در Microservices آزمون ادغام می‌تواند شامل تست ارتباط بین سرویس احراز هویت و سرویس سفارشات باشد. در Monolithic ممکن است تست نحوه تعامل بخش‌های مختلف سیستم مانند ماژول کاربران و ماژول محصولات باشد.

E2E Testing

در هر دو معماری، آزمون انتها به انتها می‌تواند شامل تست کامل فرآیند خرید، از جستجوی محصول تا تکمیل سفارش باشد.

Testing Acceptance

آزمون پذیرش در Microservices ممکن است بر روی قابلیت‌های کلی سیستم تمرکز کند، در حالی که در Mono-lithic ممکن است شامل بررسی انطباق کارکردهای سیستم با نیازهای کسب‌وکار باشد.

Black Box Testing و White Box Testing

در Microservices آزمون جعبه سیاه می‌تواند شامل تست API ها بدون دانستن جزئیات پیاده‌سازی باشد. آزمون جعبه سفید می‌تواند بر روی تست پیاده‌سازی داخلی یک سرویس خاص تمرکز کند.

Happy Path Testing

تست مسیر خوشبختی (Happy Path Testing) به آزمون موفقیت‌آمیز فرآیندهای اصلی سیستم بدون هیچ خطایی اشاره دارد. به عنوان مثال، در یک سیستم خرید آنلاین، تست فرآیند خرید از انتخاب محصول تا پرداخت بدون مواجهه باهیچ خطا یا مشکلی نمونه‌ای از این نوع تست است. این آزمون اطمینان می‌دهد که مسیرهای اصلی و مورد انتظار سیستم به طور صحیح کار می‌کنند.

Exceptional Testing

آزمون استثنایی (Exceptional Testing) به تست نرم‌افزار برای مدیریت خطاها و استثناها می‌پردازد. این آزمون بررسی می‌کند که آیا نرم‌افزار می‌تواند به طور مؤثر از پس موقعیت‌های غیرمنتظره برآید یا خیر. به عنوان مثال، وارد کردن داده‌های نامعتبر توسط کاربر و بررسی نحوه رفتار سیستم در این شرایط.

Mocking

شبیه‌سازی یا Mocking به تکنیکی در آزمون نرم‌افزار اشاره دارد که در آن اجزای سیستم که آزمون وابستگی به آن‌ها دارد (مانند پایگاه داده‌ها، سرویس‌های وب یا سایر ماژول‌ها) به وسیله‌ی اشیاء جعلی (mock objects) جایگزین می‌شوند. این کار امکان آزمون بخش‌های خاصی از کد را بدون نیاز به وابستگی‌های خارجی یا پیچیده فراهم می‌آورد. به عنوان مثال، شبیه‌سازی یک سرویس پرداخت برای تست بدون نیاز به اتصال واقعی به سرویس پرداخت.

TDD (Test-Driven Development)

توسعه محور تست (TDD) یک روش توسعه نرم‌افزار است که در آن تست‌ها قبل از نوشتن کد اصلی نوشته می‌شوند. این رویکرد توسعه را به سمتی هدایت می‌کند که از ابتدا مطابق با الزامات تست باشد. TDD به تضمین کیفیت کد کمک می‌کند و اطمینان می‌دهد که نیازمندی‌ها به درستی پیاده‌سازی شوند. به عنوان مثال، نوشتن یک تست برای تابع محاسبه مجموع قبل از اینکه خود تابع پیاده‌سازی شود.

نتیجه‌گیری

معماری‌های Microservices و Monolithic هر کدام چالش‌ها و مزایای خاص خود را در حوزه آزمون نرم‌افزار دارند. درک و اجرای صحیح مفاهیم آزمون نرم‌افزار، از جمله Unit Testing ، Integration Testing ، E2E Testing ، Acceptance Testing ، Black Box و White Box Testing ، Happy Path Testing ، Exceptional Testing ، Mocking و (TDD) می‌تواند به بهبود کیفیت نرم‌افزار و افزایش رضایتمندی کاربران کمک کند. استفاده از تکنیک‌های آزمون مناسب برای هر معماری خاص از اهمیت بالایی برخوردار است. در معماری ،Microservices توجه به تست‌های ادغام و E۲E برای اطمینان از ارتباط صحیح بین سرویس‌ها ضروری

است، در حالی که در معماری ،Monolithic تمرکز بر آزمون‌های واحد و ادغام درونی اجزا می‌تواند به حفظ کیفیت کمک کند.

(Mocking) و (TDD) نیز رویکردهای مهمی در هر دو معماری هستند که به تسریع توسعه و اطمینان از پایداری نرم‌افزار کمک می‌کنند. با به کارگیری این استراتژی‌ها، توسعه‌دهندگان می‌توانند تضمین کنند که نرم‌افزار در برابر تغییرات آینده مقاوم است و به راحتی قابل نگهداری و توسعه خواهد بود.

در نهایت، انتخاب استراتژی‌ها و تکنیک‌های آزمون مناسب، بستگی به اهداف کسب‌وکار، معماری نرم‌افزار، و منابع موجود دارد. اجرای صحیح این تکنیک‌ها نیازمند درک عمیقی از هر دو معماری و چگونگی به کارگیری این مفاهیم در هر یک است. با پیشرفت و تکامل مستمر در روش‌های توسعه نرم‌افزار و آزمون، توسعه‌دهندگان باید به دنبال یادگیری و بهبود مهارت‌های خود در این زمینه‌ها باشند تا بتوانند بهترین کیفیت را در تولیدات نرم‌افزاری خود ارائه دهند.

جواب سوال ۵

بخش ۱: طبقه بندی کیفیت

الف. تفاوت عوامل کیفی نرم و سخت

بر اساس کتاب پرسمن، **عوامل کیفی نرم** مربوط به ویژگی‌هایی هستند که به طور مستقیم قابل اندازه‌گیری نیستند و بیشتر جنبه‌های ذهنی و تجربی کاربران را در بر می‌گیرند. مثال‌هایی از ابعاد کیفی نرم شامل قابلیت استفاده و رضایت کاربر می‌شود. این عوامل بیشتر *Subjective* (ذهنی) هستند.

در مقابل، **عوامل کیفی سخت** شامل ویژگی‌هایی است که به طور مستقیم قابل اندازه‌گیری و مشاهده هستند. مثال‌هایی از ابعاد کیفی سخت شامل کارایی و امنیت نرم‌افزار می‌باشد. این عوامل *Objective* (موضوعی) محسوب می‌شوند.

ب. سنجه‌های مستقیم و غیرمستقیم

پرسمن عوامل سخت کیفی را به **سنجه‌های مستقیم** و **غیرمستقیم** تقسیم می‌کند. سنجه‌های مستقیم به ویژگی‌هایی اشاره دارند که به طور مستقیم و بدون نیاز به تفسیر اضافی قابل اندازه‌گیری هستند، مانند زمان پاسخ سیستم. در حالی که سنجه‌های غیرمستقیم، ویژگی‌هایی را نشان می‌دهند که اندازه‌گیری آن‌ها نیازمند تفسیر یا محاسبه است، مانند رضایت کاربر. پرسمن بیان می‌کند که در بسیاری از موارد، سنجه‌های مستقیم دشوار است و تمایل دارد تا بر سنجه‌های غیرمستقیم تکیه کند.

بخش ۲: معمای لاینحل کیفیت نرم‌افزار

الف. معمای لاینحل کیفیت نرم‌افزار

بر اساس بیان برتراند مِیِر، معمای لاینحل کیفیت نرم‌افزار اشاره به این دارد که کیفیت نرم‌افزار همواره در تعادل با زمان و منابع (بودجه) قرار دارد. مِیِر بیان می‌کند که افزایش کیفیت معمولاً به زمان بیشتر یا منابع بیشتر نیاز دارد، و در نتیجه، هر پروژه‌ای مجبور است بین این سه مولفه تعادل ایجاد کند. این تعادل نشان‌دهنده چالش‌های مدیریت پروژه‌های نرم‌افزاری است، جایی که تصمیمات مهمی برای حفظ کیفیت ضروری است در حالی که همزمان باید زمان‌بندی و بودجه را نیز در نظر گرفت.

ب. رویکردهای جدیدتر مدیریت پروژه

در رویکردهای جدیدتر مدیریت پروژه، به‌خصوص آن‌هایی که تحت تأثیر روحیه چابک قرار دارند، عامل **ارتباطات** به عنوان یک عامل مهم و تاثیرگذار معرفی می‌شود. ارتباطات موثر درون تیمی و با ذینفعان، به افزایش شفافیت و درک متقابل از انتظارات کمک می‌کند و به تیم اجازه می‌دهد که به طور فعال در جهت حفظ کیفیت و رسیدن به اهداف پروژه با محدودیت‌های زمانی و بودجه، پیش رود.

ج. پارادوکس سرعت ایجاد در جراحی قلب باز

در دقایق ۱۲ تا ۱۹ این کلاس از رابرت مارتین، پارادوکسی مطرح می‌شود که نشان می‌دهد چگونه تمرکز بر کیفیت می‌تواند در واقع به افزایش سرعت توسعه نرم‌افزار کمک کند. مثال جراحی قلب باز از عمو باب نشان می‌دهد که چگونه تمرکز بر دقت و احتیاط در ابتدا، به جلوگیری از خطاها و مشکلاتی که می‌تواند زمان و منابع بیشتری را در آینده نیاز داشته باشد، کمک می‌کند. این پارادوکس تأکید می‌کند که اختصاص دادن زمان کافی برای اطمینان از کیفیت در مراحل اولیه پروژه، می‌تواند به کاهش زمان کلی توسعه کمک کند. به‌خصوص در این ویدیو اشاره می‌شه که برای مثال کد کثیف، در ابتدا خیلی سریع پیاده‌سازی شده اما وقتی به توسعه می‌رسه، کلی وقت زمان نیاز هستش تا بررسی بشه و دوباره توسعه داده بشه. عملا کیفیت کار به خاطر کم بودن زمان ایجاد پایین بوده و در نهایت باعث می‌شه که زمان و منابع بیشتر در آینده نیز نیاز بشه برای توسعه.

د. توصیه نهایی پرسمن درباره معمای لاینحل کیفیت نرم‌افزار

در بخش ۱۹/۳/۶ از کتاب پرسمن، توصیه نهایی وی درباره معمای لاینحل کیفیت نرم‌افزار این است که توجه به کیفیت باید از ابتدای پروژه و در تمام مراحل توسعه نرم‌افزار ادامه یابد. پرسمن بر این باور است که اگرچه نمی‌توان تمامی خطاها را از بین برد، اما با اتخاذ رویکردهای مناسب و تمرکز بر اصول مهندسی نرم‌افزار، می‌توان تأثیر خطاها را به حداقل رساند و کیفیت نرم‌افزار را بهبود بخشید. این توصیه با توضیحات عمو باب در بخش ج در ارتباط است، زیرا هر دو بر اهمیت کیفیت و تأثیر آن بر سایر جنبه‌های پروژه، از جمله زمان، تأکید دارند.

ه. هزینه‌های مرتبط با حفظ کیفیت

پرسمن بیان می‌کند که هزینه حفظ کیفیت تنها به منابع تیم محدود نمی‌شود؛ بلکه هزینه‌های دیگری نیز مطرح می‌شود، از جمله هزینه‌های از دست دادن اعتبار به دلیل عرضه محصولی با کیفیت پایین، هزینه پشتیبانی و نگهداری بیشتر برای رفع اشکالات پس از عرضه، و هزینه‌های مربوط به از دست دادن مشتریان. این هزینه‌ها نشان‌دهنده اهمیت توجه به کیفیت در طول چرخه حیات توسعه نرم‌افزار است و اینکه چرا سرمایه‌گذاری در کیفیت از ابتدا می‌تواند به کاهش هزینه‌های کلی کمک کند.

و. راه حل میان بر برای معمای لاینحل کیفیت نرم‌افزار

در کتاب پرسمن، یک راه حل میان بر برای معمای لاینحل کیفیت نرم‌افزار مطرح شده است که به *اندازه کافی خوب* نامیده می‌شود. این مفهوم بیان می‌کند که در برخی شرایط، پذیرش نرم‌افزاری که تمامی ویژگی‌ها و کیفیت مطلوب را ندارد اما «به اندازه کافی خوب» برای عرضه و استفاده است، می‌تواند مفید باشد. این رویکرد به ویژه در محیط‌های چابک و با ریتم سریع توسعه، که ارزش سریع رساندن محصول به بازار و گرفتن بازخورد زودهنگام از کاربران را در اولویت قرار می‌دهند، معنادار است. با این حال، این رویکرد نیازمند توازن دقیقی است تا از قربانی شدن کیفیت به نحوی که بر تجربه کاربری اثر منفی بگذارد یا هزینه‌های نگهداری را در آینده افزایش دهد، جلوگیری کند. در نهایت، این میانبر ممکن است در شرایط خاصی معقول باشد، اما نباید به عنوان راه‌حلی همه‌کاره برای هر پروژه نرم‌افزاری در نظر گرفته شود.

ز. پیشنهاد راه حل دیگری برای مسئله کیفیت

به جای پذیرش نرم‌افزاری که تنها «به اندازه کافی خوب» است، یک راه حل دیگر برای مسئله کیفیت می‌تواند تمرکز بر اصول مهندسی نرم‌افزار و ادغام رویکردهای چابک با استانداردهای کیفیت سختگیرانه باشد. این شامل استفاده از

تست خودکار، ادغام مداوم، تحویل مداوم، و بازخورد مداوم از کاربران برای بهبود مستمر محصول است. همچنین، این رویکرد به تیم‌ها اجازه می‌دهد تا بر روی بهبود کیفیت در حین حفظ سرعت و انعطاف‌پذیری تمرکز کنند. با پیاده‌سازی فرهنگ بازخورد و یادگیری مداوم، تیم‌ها می‌توانند از خطاها یاد بگیرند و فرآیندهای خود را برای جلوگیری از تکرار خطاها بهبود ببخشند.

این راه حل نه تنها به حفظ کیفیت کمک می‌کند بلکه اطمینان می‌دهد که تیم‌های توسعه می‌توانند به طور موثر به نیازهای تغییرپذیر کاربران و بازار پاسخ دهند بدون آنکه مجبور به قربانی کردن کیفیت یا انعطاف‌پذیری شوند. این نگرش همچنین به تیم‌ها این قابلیت را می‌دهد که به جای اتکا به میانبرها، بر روی ایجاد ارزش واقعی برای کاربران تمرکز کنند.

جواب سوال ۶

جواب این سوال خیلی گسترده‌ست، سعی می‌کنم تا جای ممکن خلاصه جواب را بنویسم.

اهداف کیفیت نرم‌افزار

پرسمن در فصل ۲۱ چهار هدف اصلی برای کیفیت نرم‌افزار پیشنهاد می‌دهد:

الف) **کارایی**: عملکرد نرم‌افزار در زمان اجرا و سرعت پاسخگویی به کاربر.
ب) **قابلیت اطمینان**: توانایی نرم‌افزار برای انجام دقیق وظایف مورد نظر بدون خطا و نقص.
ج) **قابلیت استفاده**: سهولت استفاده از نرم‌افزار و رابط کاربری آن.
د) **قابلیت نگهداری**: توانایی نرم‌افزار برای به‌روزرسانی و نگهداری آسان.

صفات و متریک‌های کیفی

برای هر هدف، دو صفت و یک متریک کیفی تعیین می‌کنیم:

کارایی:

- صفت: زمان پاسخ - متریک: میانگین زمان پاسخ به درخواست‌ها.
- صفت: استفاده از منابع - متریک: درصد استفاده از CPU و حافظه در حالت بار کامل.

قابلیت اطمینان:

- صفت: نرخ خطا - متریک: تعداد خطاها به ازای هر هزار خط کد.
- صفت: زمان بازیابی پس از خطا - متریک: میانگین زمان لازم برای بازیابی سیستم پس از خطا.

قابلیت اطمینان

میانگین زمان تا شکست (MTTF) و میانگین زمان تا بازیابی (MTTR)

- **MTTF**: میانگین زمانی که سیستم بدون خطا کار می‌کند.
- **MTTR**: میانگین زمان لازم برای تعمیر یا بازیابی سیستم پس از یک خطا.

میانگین زمان بین شکست‌ها (MTBF)

MTBF: میانگین زمان بین شکست‌ها، که نشان‌دهنده دوره زمانی متوسط بین وقوع دو خطا در سیستم است. این شاخص برای ارزیابی قابلیت اطمینان و پایداری یک سیستم مورد استفاده قرار می‌گیرد. MTBF به عنوان یک معیار کلیدی در تضمین کیفیت نرم‌افزار، به سازمان‌ها کمک می‌کند تا درک بهتری از عملکرد محصولات خود در طول زمان داشته باشند و برنامه‌ریزی دقیق‌تری برای نگهداری و بهبود محصولات انجام دهند. محاسبه MTBF به صورت زیر انجام می‌شود:

$$MTBF = \frac{\text{کل زمان کارکرد سیستم}}{\text{تعداد کل شکست‌ها}} \quad (۱)$$

این معیار به ویژه برای سیستم‌هایی که نیازمند دسترسی مداوم و قابلیت اطمینان بالا هستند، مانند سیستم‌های مالی، پزشکی، و حیاتی دیگر، اهمیت فراوانی دارد. افزایش MTBF نشان‌دهنده کاهش تعداد خطاها و افزایش قابلیت اطمینان سیستم است.

محاسبات مربوط به قابلیت اطمینان

برای محاسبه میانگین زمان تا شکست (MTTF) ، میانگین زمان تا بازیابی ،(MTTR) و میانگین زمان بین شکست‌ها (MTBF) ، ابتدا باید داده‌های مربوط به زمان‌های خرابی و بازیابی را جمع‌آوری کنیم. سپس با استفاده از فرمول‌های زیر این مقادیر را محاسبه می‌کنیم:

$$MTTF = \frac{\text{کل زمان کارکرد بدون خطا}}{\text{تعداد کل شکست‌ها}} \quad (۲)$$

$$MTTR = \frac{\text{کل زمان صرف شده برای بازیابی پس از شکست‌ها}}{\text{تعداد کل شکست‌ها}} \quad (۳)$$

$$MTBF = MTTF + MTTR \quad (۴)$$

دسترسی‌پذیری سیستم

دسترسی‌پذیری سیستم نشان‌دهنده درصد زمانی است که سیستم در دسترس و قابل استفاده است. این معیار را می‌توان با استفاده از فرمول زیر محاسبه کرد:

$$\text{دسترسی پذیری} = \left(\frac{MTTF}{MTTF + MTTR}\right) \times ۱۰۰ \quad (۵)$$

معیار شکست بر زمان (FIT)

معیار FIT نشان‌دهنده تعداد خطاهایی است که در هر یک میلیارد ساعت کارکرد سیستم رخ می‌دهد. این معیار به صورت زیر محاسبه می‌شود:

$$FIT = \frac{۱۰^۹}{MTBF} \quad (۶)$$

داده‌ها:

- زمان کل خرابی‌ها = ۱/۵ + ۹/۵ + ۱۴ + ۱۱ = ۳۶ ساعت
- تعداد شکست‌ها = ۴
- زمان کل کارکرد سیستم از ۱ تا ۳۰ آذر = ۳۰ روز

محاسبات:

- **MTTF (میانگین زمان تا شکست):**
  - کل زمان کارکرد بدون خطا = (۳۰ روز × ۲۴ ساعت) - ۳۶ ساعت خرابی
  - $\text{MTTF} = \frac{\text{کل زمان کارکرد بدون خطا}}{\text{تعداد شکست‌ها}}$
- **MTTR (میانگین زمان تا بازیابی):**
  - $\text{MTTR} = \frac{\text{ساعت}۳۶}{۴} = \frac{\text{کل زمان خرابی}}{\text{تعداد شکست‌ها}}$
- **MTBF (میانگین زمان بین شکست‌ها):**
  - $\text{MTBF} = \text{MTTF} + \text{MTTR}$
- **دسترسی‌پذیری سیستم:**
  - دسترسی‌پذیری = $\left(\frac{MTTF}{MTTF+MTTR}\right) \times ۱۰۰$
- **FIT (شکست بر زمان):**
  - $\text{FIT} = \frac{۱۰^۹}{MTBF}$

با فرض که زمان کل کارکرد سیستم ۷۲۰ ساعت (۳۰ روز × ۲۴ ساعت) باشد و کل زمان خرابی ۳۶ ساعت است، می‌توانیم محاسبات را انجام دهیم:

- کل زمان کارکرد بدون خطا = $۷۲۰ - ۳۶ = ۶۸۴$ ساعت
- $\text{MTTF} = \frac{\text{ساعت}۶۸۴}{۴} = \text{ساعت}۱۷۱$
- $\text{MTTR} = \frac{\text{ساعت}۳۶}{۴} = \text{ساعت}۹$
- MTBF = ۱۷۱ + ۹ = ۱۸۰ ساعت
- دسترسی‌پذیری = $\left(\frac{۱۷۱}{۱۷۱+۹}\right) \times ۱۰۰ = ۹۵٪$
- $\text{FIT} = \frac{۱۰^۹}{۱۸۰} = ۵,۵۵۵,۵۵۶$

این محاسبات به ما نشان می‌دهند که سیستم به طور متوسط هر ۱۸۰ ساعت یک بار دچار شکست می‌شود، دسترسی‌پذیری آن تقریبا ٪۹۵ است، و معیار FIT نشان می‌دهد که در هر یک میلیارد ساعت کارکرد، تقریبا ۵.۵ میلیون خطا رخ می‌دهد.

استانداردهای تضمین کیفیت

استانداردهای تضمین کیفیت مانند ISO 9000 با هدف ایجاد چارچوبی برای سیستم‌های مدیریت کیفیت طراحی شده‌اند تا اطمینان حاصل شود که سازمان‌ها قادر به تولید محصولات و خدماتی هستند که به طور مداوم نیازهای مشتریان و الزامات قانونی را برآورده می‌سازند. دریافت استاندارد کیفیت ISO 9000 مستلزم این است که سازمان‌ها فرآیندهای خود را مطابق با الزامات استاندارد تنظیم و اجرا کنند و توانایی خود را در رعایت این استانداردها از طریق ارزیابی‌ها و بازرسی‌های دوره‌ای نشان دهند.

نقش تضمین کیفیت در تیم‌های توسعه

نقش تضمین کیفیت (QA) در تیم‌های توسعه فراتر از انجام آزمون‌های دستی پیش از استقرار محصول است. تضمین کیفیت به عنوان یک فرآیند جامع عمل می‌کند که از برنامه‌ریزی، طراحی، توسعه، و تست محصول تا نگهداری آن پس از عرضه را شامل می‌شود. این فرآیند مستلزم همکاری نزدیک بین تیم‌های توسعه، تست، و عملیات است تا اطمینان حاصل شود که محصول نهایی نه تنها بدون خطا است بلکه به نیازهای کاربران نیز پاسخ می‌دهد. تضمین کیفیت به معنای واقعی کلمه در تمام مراحل چرخه زندگی نرم‌افزار دخیل است و شامل فعالیت‌هایی مانند تحلیل نیازمندی‌ها، طراحی آزمون‌ها، اجرای آزمون‌های خودکار و دستی، و بازنگری کد است.

اردهای تضمین کیفیت و نقش تضمین کیفیت در تیم‌های توسعه نرم‌افزار را ارائه می‌دهد. متن تکمیلی در این بخش به تشریح اهمیت استانداردهای بین‌المللی مانند ISO 9000 در ارتقاء کیفیت محصولات و خدمات نرم‌افزاری و همچنین به تاکید بر این موضوع می‌پردازد که چگونه یک فرآیند تضمین کیفیت جامع و همه‌جانبه می‌تواند به توسعه محصولات نرم‌افزاری با کیفیت بالا و رضایتمندی کاربران کمک کند.

در این بخش همچنین بر اهمیت همکاری میان تیم‌های مختلف در فرآیند توسعه نرم‌افزار تاکید شده است. تضمین کیفیت نباید تنها به عنوان یک فعالیت پایانی در نظر گرفته شود، بلکه باید به عنوان بخشی از تمام مراحل چرخه توسعه نرم‌افزار، از جمله تحلیل نیازمندی‌ها، طراحی، پیاده‌سازی و تست، ادغام شود. این رویکرد همه‌جانبه به تضمین کیفیت کمک می‌کند تا از بروز مشکلات در مراحل پایانی پروژه و هزینه‌های اضافی ناشی از آن‌ها جلوگیری کند.

همچنین، نقش افراد متخصص در تضمین کیفیت باید فراتر از انجام آزمون‌های دستی ساده باشد و شامل فعالیت‌هایی مانند تحلیل ریسک، مدیریت تغییر، بهبود فرآیندها و اطمینان از رعایت استانداردها و بهترین شیوه‌ها در تمام مراحل توسعه نرم‌افزار شود. این امر مستلزم دانش فنی عمیق، مهارت‌های ارتباطی قوی و درک کاملی از اهداف کسب‌وکار و نیازهای کاربران است.

از طریق اجرای دقیق استانداردهای تضمین کیفیت و تعهد به یک فرآیند تضمین کیفیت جامع، سازمان‌ها و تیم‌های توسعه می‌توانند به بهبود مستمر کیفیت محصولات نرم‌افزاری و افزایش رضایتمندی کاربران دست یابند.