فصل ۲۰: آزمون نرم افزار و راهبردها

اهمیت آزمایش نرم‌افزار و اثرات آن بر کیفیت نرم‌افزار نیاز به تأکید بیشتر ندارد. Deutch در این باره این گونه بیان می‌نماید:

توسعهٔ سیستم‌های نرم‌افزاری شامل یک‌سری فعالیت‌های تولید می‌باشد که امکان اشتباهات انسانی در آن زیاد است. خطاها در ابتدای یک فرآیند و مراحل توسعه بعدی آن ظهور می‌نمایند. به دلیل عدم توانایی انجام کارها و برقراری ارتباط به صورت کامل، توسعهٔ نرم‌افزار همواره با فعالیت تضمین کیفیت همراه است.

آزمایش نرم‌افزار عنصری حیاتی از تضمین کیفیت نرم‌افزار می‌باشد و مرور تقریبی مشخصه، طراحی، و تولید کد را نشان می‌دهد.

یک شیوهٔ استراتژیک برای آزمایش نرم‌افزار

آزمایش، مجموعه فعالیت‌هایی است که می‌تواند از قبل به صورت سیستماتیک برنامه‌ریزی و هدایت شوند. به این دلیل، الگویی برای آزمایش نرم‌افزار باید برای فرآیند نرم‌افزار تعریف شود. این الگو شامل مجموعه مراحلی است که می‌توان تکنیک‌های خاص طراحی نمونه های آزمایش و روش های آزمایش را در آن قرار داد.

چند استراتژی آزمایش نرم‌افزار در این رابطه پیشنهاد شده است. همهٔ آنها برای توسعه دهندهٔ نرم‌افزار، الگویی را به منظور آزمایش فراهم می‌کنند و همگی دارای خصوصیات زیر هستند:

- آزمایش از سطح مؤلفه شروع می‌شود به سمت خارج در جهت مجتمع‌سازی کل سیستم کامپیوتری پیش می‌رود.
- تکنیک‌های متفاوت آزمایش، در نقاط زمانی مختلف مناسب می‌باشند.
- آزمایش توسط توسعه دهندهٔ نرم‌افزار و برای پروژه‌های بزرگ توسط گروه مستقل آزمایش، هدایت می‌شود.
- آزمایش و اشکالزدایی فعالیت‌های متفاوتی هستند، اما اشکالزدایی باید با هر استراتژی آزمایش همراه باشد.

یک استراتژی برای آزمایش نرم‌افزار باید آزمایش های سطح پایینی را هدایت کند که برای بازبینی صحت پیاده‌سازی یک قطعه کد کوچک لازم می‌باشند. همچنین این استراتژی باید آزمایش های سطح بالایی را سازماندهی کند که اکثر توابع سیستم را در رابطه با نیازهای مشتری اعتبارسنجی می‌نمایند. یک استراتژی باید راهنمایی‌هایی را برای مجری و مجموعه‌ای از علایم نشان دهنده را برای مدیر فراهم نماید. چون این مراحل استراتژی آزمایش زمانی انجام می‌شوند که فشار مربوط به پایان مهلت، شروع به افزایش می‌نماید، پیشرفت باید قابل اندازه‌گیری باشد و مشکلات باید تا حد امکان به سادگی برطرف شوند.

اصول آزمایش نرم‌افزار

آزمایش، موارد غیر معمول جالبی را برای مهندس نرم‌افزار آشکار می‌نماید. در ضمن فعالیت‌های اولیهٔ مهندسی نرم‌افزار، مهندس، سعی در ایجاد نرم‌افزار با استفاده از مفهومی مجرد و بدست آوردن محصولی واضح و کامل دارد. اینک آزمایش باید انجام شود. این مهندس یک‌سری نمونه های آزمایش ایجاد می‌کند که باید نرم‌افزار ایجاد شده را با شکست روبرو نماید. در واقع، آزمایش، یک مرحله در فرآیند نرم‌افزار است که می‌تواند به عنوان فرآیندی مخرب به جای سازنده در نظر گرفته شود (حداقل از نظر روانشناسی). به هر حال هدف از آزمایش چیزی متفاوت از آنچه انتظار می رود!

صحت و اعتبارسنجی

آزمایش نرم‌افزار یک عنصر از عنوان گسترده‌تری است که اغلب با **صحت و اعتبارسنجی** (V&V-Validation & verification) شناخته می‌شود. صحت اشاره به مجموعه فعالیت‌هایی دارد که مطمئن می‌سازند نرم‌افزار به

درستی یک تابع خاص را پیاده‌سازی می‌نماید. اعتبارسنجی اشاره به مجموعه‌ای متفاوت دارد که مطمئن می‌سازند نرم‌افزاری که ایجاد شده منطبق بر نیازهای مشتری است. Boehm این مطلب را این گونه بیان می‌کند:

صحت: " آیا محصول را درست ایجاد می‌کنیم؟

اعتبارسنجی: " آیا محصول درستی را ایجاد می‌کنیم؟

تعریف V&V شامل بسیاری از فعالیت‌هایی است که تضمین کیفیت نرم‌افزار (SQA) نامیده می‌شوند. صحت و اعتبارسنجی شامل گروه وسیعی از فعالیت‌های SQA می‌باشد شامل: مرورهای فنی رسمی، بررسی کیفیت و پیکربندی، نظارت بر کارایی، شبیه‌سازی، امکان‌سنجی، مرور مستندات، مرور بانک اطلاعاتی، تحلیل الگوریتم، آزمایش توسعه، آزمایش کیفی، و آزمایش نصب. اگرچه آزمایش نقش بسیار مهمی را در V&V دارد، بسیاری از فعالیت‌های دیگر نیز لازم می‌باشند.

اهداف آزمایش

در مورد آزمایش نرم‌افزار، Myers چند قانون زیر را بیان می‌کند که اهداف مناسبی برای آزمایش هستند:

۱- **آزمایش فرآیندی است شامل اجرای برنامه با هدف یافتن خطا.**

۲- **یک نمونه آزمایش خوب، نمونه ای است که با احتمال بالایی خطاها را بیابد.**

۳- **آزمایش موفق، آزمایشی است که خطاهای یافت نشده تاکنون را بیابد.**

این اهداف تغییری دراماتیک را در دیدگاه ایجاد می‌نمایند. این اهداف باعث تغییر در دیدگاه متداولی می‌شوند که آزمایش موفق را آن نوع آزمایش می‌داند که در آن خطایی یافت نشود. هدف، طراحی آزمایش هایی است که به طور سیستماتیک رده‌های متفاوتی از خطاها را آشکار نمایند، و این عمل را با حداقل مقدار زمان و فعالیت انجام دهند.

اصول آزمایش

قبل از بکارگیری روش های طراحی نمونه های مؤثر آزمایش، مهندس نرم‌افزار باید اصول اولیه‌ای را که آزمایش نرم‌افزار را هدایت می‌کنند بفهمد. Davis مجموعه‌ای از اصول آزمایش را پیشنهاد می‌کند:

- **تمام آزمایش های باید براساس نیازهای مشتری قابل پیگیری باشند.**
- **آزمایش ها باید مدتی طولانی قبل از شروع آزمایش برنامه‌ریزی شوند.**
- **اصل Pareto برای آزمایش نرم‌افزار بکار گرفته شود.**
- **آزمایش باید با " توجه به اجزاء " شروع شود و به سمت آزمایش " کلی " پیش رود.**
- **آزمایش کامل و جامع امکان‌پذیر نیست.**
- **به منظور داشتن بیشترین تأثیر، آزمایش باید توسط تیم مستقلی هدایت شود.**

قابلیت آزمایش

در موارد ایده‌آل، مهندس نرم‌افزار، برنامه‌ای کامپیوتری، سیستم، یا محصولی را با در نظر داشتن **قابلیت آزمایش** طراحی می‌کند. این مساله باعث می‌شود افرادی که مسوول آزمایش هستند، نمونه هاای آزمایشی مؤثر را ساده‌تر ایجاد نمایند. اما قابلیت آزمایش چیست؟ Bach James قابلیت آزمایش را این گونه توصیف می‌کند:

عملیاتی بودن(Operability). " نرم افزار هرچه بهتر کارکند، با کارایی بالاتری آزمایش می‌شود. "

- سیستم اشکالات اندکی دارد (اشکالات، تحلیل و گزارش اضافی را بر فرآیند آزمایش تحمیل می‌کنند).
- هیچ اشکالی، اجرای آزمایشات را متوقف نکند.
- محصول در مراحل عملیاتی تکامل می‌یابد (توسعه و آزمایش همزمان را امکان‌پذیر می‌نماید).

قابلیت مشاهده(Observability). " آنچه می‌بینید آزمایش می‌کنید ".

- خروجی‌های مجزا برای هر ورودی تولید می‌شوند.
- حالت‌های سیستم و متغیرها در ضمن اجرا قابل رؤیت و قابل پرس و جو باشند.
- حالت‌های قبلی سیستم و متغیرها قابل پرس و جو می‌باشند (برای مثال، ثبت تراکنشها).
- تمام فاکتورهای مؤثر بر خروجی قابل رؤیت باشند.
- خروجی غلط به راحتی مشخص شود.
- خطاهای داخلی به طور خودکار از طریق مکانیزم‌های خودآزمایی آشکار شوند.
- خطاهای داخلی به طور خودکار گزارش شوند.
- برنامه های مبدأ قابل دسترسی باشد.

قابلیت کنترل(Controllability). " هر چه نرم‌افزار بهتر کنترل شود، آزمایش بیشتر به طور خودکار و بهینه قابل انجام است. "

- تمام خروجی‌های ممکن نمی‌توانند از طریق برخی ترکیبات ورودی تولید شوند.
- تمام دستورات از طریق برخی ترکیبات ورودی قابل اجرا باشند.
- حالت‌ها و متغیرهای نرم‌افزار و سخت‌افزار مستقیماً توسط آزمایش کننده قابل کنترل باشند.
- قالب‌های ورودی و خروجی یکنواخت و ساختیافته باشند.
- آزمایش ها می توانند به طور مناسبی مشخص شوند، و به طور خودکار انجام گیرند و دوباره تولید گردند.

تجزیه‌پذیری(Decomposability). " با کنترل نمودن محدودهٔ آزمایش، با سرعت بیشتری مسایل تجزیه می‌شوند و آزمایش های هوشمندانه‌تری انجام می‌گیرد ".

- سیستم نرم‌افزار از پیمانه‌های مستقل ساخته می‌شود.
- پیمانه‌های نرم‌افزاری به طور مستقل قابل آزمایش هستند.

سادگی(Simplicity). هر چه مورد برای آزمایش کمتر باشد، آزمایش با سرعت بیشتری انجام می گیرد. "

- سادگی تابعی (برای مثال، مجموعه جنبه‌هایی که حداقل لازم برای دستیابی به نیازها هستند).
- سادگی ساختاری (برای مثال، معماری پیمانه‌بندی می‌شود تا انتشار اشکالات را محدود نماید).
- سادگی برنامه های مبداء (برای مثال، استاندارد کدنویسی برای سهولت بازبینی و نگهداری تعریف می‌شود).

پایداری(Stability). " هر چه تغییرات کمتر باشد، انحراف از آزمایش کمتر است. "

- تغییرات در نرم‌افزار غیرمتداول هستند.
- تغییرات در نرم‌افزار کنترل شده هستند.
- تغییرات در نرم‌افزار، آزمایش های موجود را نامعتبر نمی‌سازند.
- نرم‌افزار از شکست‌ها به خوبی خارج می‌شود.

قابلیت فهم(Understandability). " هر چه اطلاعات بیشتری در اختیار داشته باشیم، آزمایش هوشمندانه‌تری انجام می‌شود. "

- طراحی کاملاً قابل فهم است.
- وابستگی‌های بین مؤلفه‌های داخلی، خارجی، و اشتراکی کاملاً قابل فهم هستند.
- تغییرات طراحی منتقل می‌شوند.
- مستندات فنی قابل دسترسی است.
- مستندات فنی به طور مناسبی سازماندهی شده است.
- مستندسازی فنی دقیق انجام شده است.

طراحی نمونه های آزمایش

طراحی آزمایش هایی برای نرم‌افزار و محصولات مهندسی دیگر می‌تواند به اندازهٔ طراحی اولیهٔ خود محصول متغیر باشد. با این وجود، مهندسین نرم‌افزار اغلب با آزمایش به عنوان فعالیتی نهایی برخورد می‌نمایند، و نمونه های آزمایشی طراحی می‌کنند که ظاهراً درست هستند اما اطمینان کمی از کامل بودن آنها وجود دارد. اهداف آزمایش را به خاطر آورید، براساس آنها آزمایش هایی باید طراحی شوند که احتمال بالایی برای یافتن اکثر خطاها، با حداقل مقدار زمان و فعالیت داشته باشند.

مجموعه‌ای غنی از روش های طراحی نمونه های آزمایش برای نرم‌افزار تکامل یافته‌اند. این روش ها برای توسعه دهنده، روشی سیستماتیک را برای آزمایش فراهم می‌کنند. مهمتر این که، این روش ها مکانیزمی را فراهم می‌کنند که به اطمینان از کامل بودن آزمایش ها کمک می‌کند و احتمال بالایی برای کشف خطاهای نرم‌افزاری را نیز تضمین می‌نمایند.

هر محصول مهندسی (و اکثر چیزهای دیگر) می‌تواند به یکی از این دو روش زیر آزمایش شود:

۱. **با دانستن تابع خاصی که یک محصول برای انجام آن طراحی شده،** آزمایش هایی طراحی می‌شوند که مشخص کنند هر تابع کاملاً عملیاتی است در حالی که در عین حال در هر تابع برای یافتن خطاها جستجو نیز انجام می‌گیرد**(آزمایش جعبه سیاه)**.

۲. **با دانستن عملکرد داخلی محصول،** آزمایش ها به گونه‌ای طراحی می‌شوند که تعیین نماید اعمال داخلی مطابق با مشخصه‌ها انجام می‌شوند و تمام مؤلفه‌های داخلی به طور مناسبی آزمایش می گردند**(آزمایش جعبه سفید)**.

آزمایش جعبهٔ سفید

آزمایش جعبهٔ سفید، که گاهی آزمایش جعبهٔ شیشه‌ای نامیده می‌شود، یک روش طراحی نمونه های آزمایش است که از ساختار کنترل طراحی رویه‌ای برای هدایت نمونه های آزمایش استفاده می‌کند. با استفاده از روش های آزمایش جعبهٔ سفید، مهندس نرم‌افزار می‌تواند نمونه های آزمایشی را بدست آورد که

۱- تضمین نمایند که تمام مسیرهای مستقل داخل پیمانه حداقل یک بار آزمایش شوند،

۲- تمام تصمیمات شرطی را در دو بخش درست و غلط بررسی نمایند،

۳- تمام حلقه‌ها را در شرایط مرزی و در محدوده‌های عملیاتی اجرا کنند، و

۴- ساختمان داده‌های داخلی را بررسی نمایند تا از اعتبار آنها مطمئن شوند.

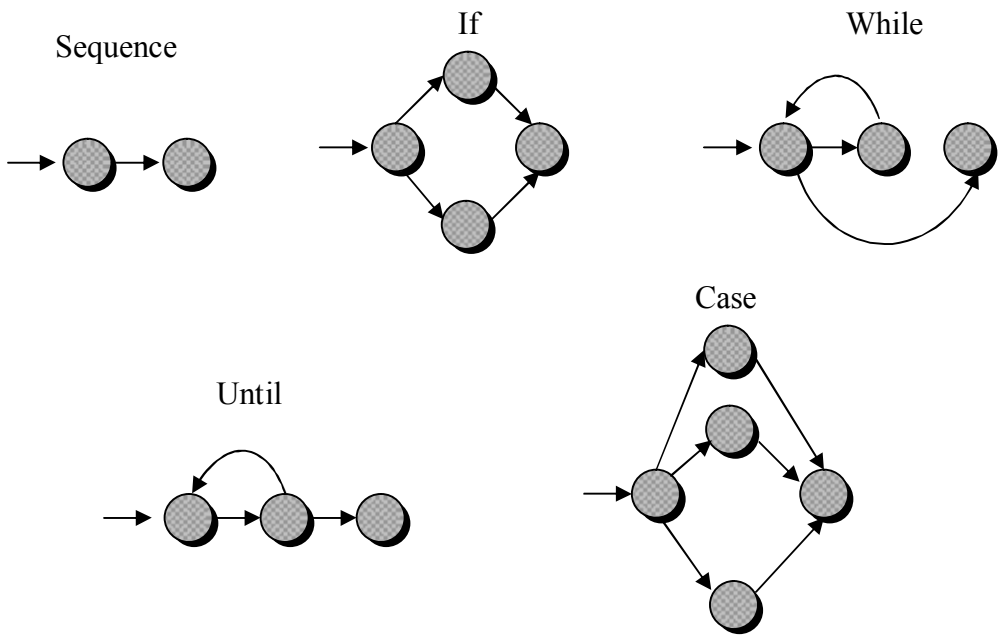
به منظور انجام آزمایش جعبه سفید با دو بحث مسیر پایه و گراف جریان مطرح گردد.

آزمایش مسیر پایه

آزمایش مسیر پایه یک تکنیک آزمایش جعبهٔ سفید است که ابتدا توسط Mc Cabe پیشنهاد شد. روش مسیر پایه، طراح نمونه های آزمایش را وادار می‌نماید که اندازهٔ پیچیدگی منطقی طراحی رویه‌ای را بدست آورد و این اندازه را به عنوان راهنمایی برای تعریف مجموعهٔ پایه مسیرهای اجرایی به کار ببرد. مسیر پایه تضمین می کند که با اجرای نمونه های آزمایش بدست آمده، هر دستور برنامه حداقل یک بار در ضمن آزمایش اجرا می گردد.

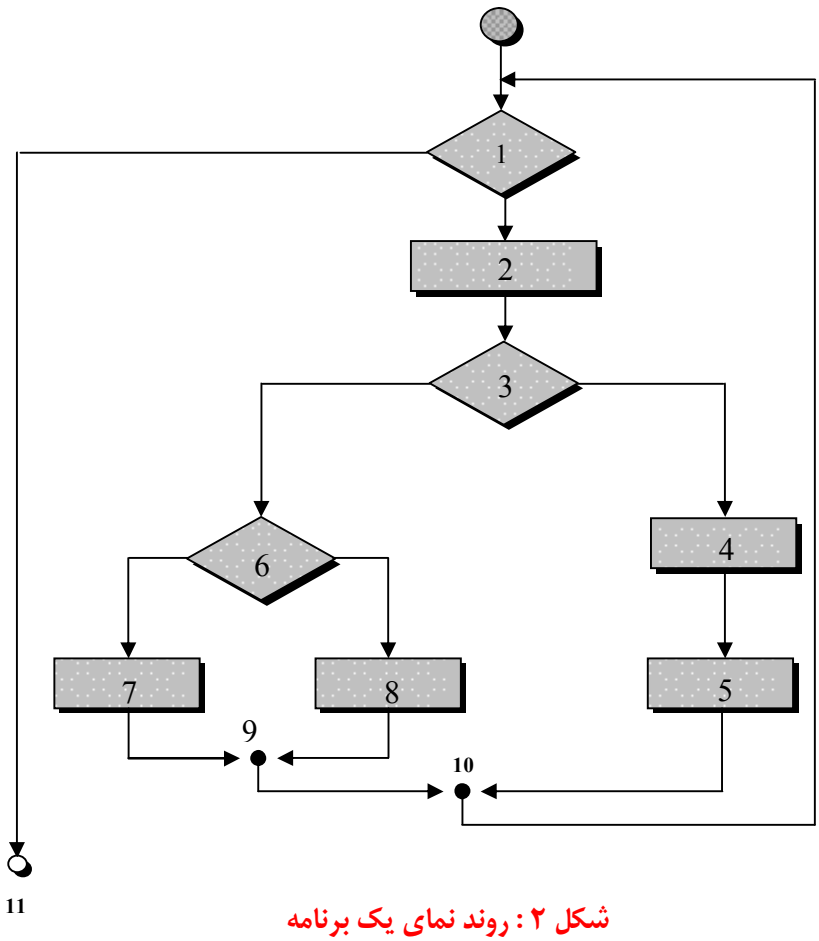
گراف جریان

قبل از معرفی روش مسیر پایه، یک نشان‌گذاری ساده برای نمایش جریان کنترل به نام **گراف جریان**(یا گراف برنامه) باید معرفی شود. گراف جریان، جریان کنترل منطقی را با استفاده از نشان‌گذاری نمایش داده شده در شکل ۱ نشان می‌دهد. هر واحد ساختاری (فصل ۱۶) یک نماد متناظر گراف جریان دارد.

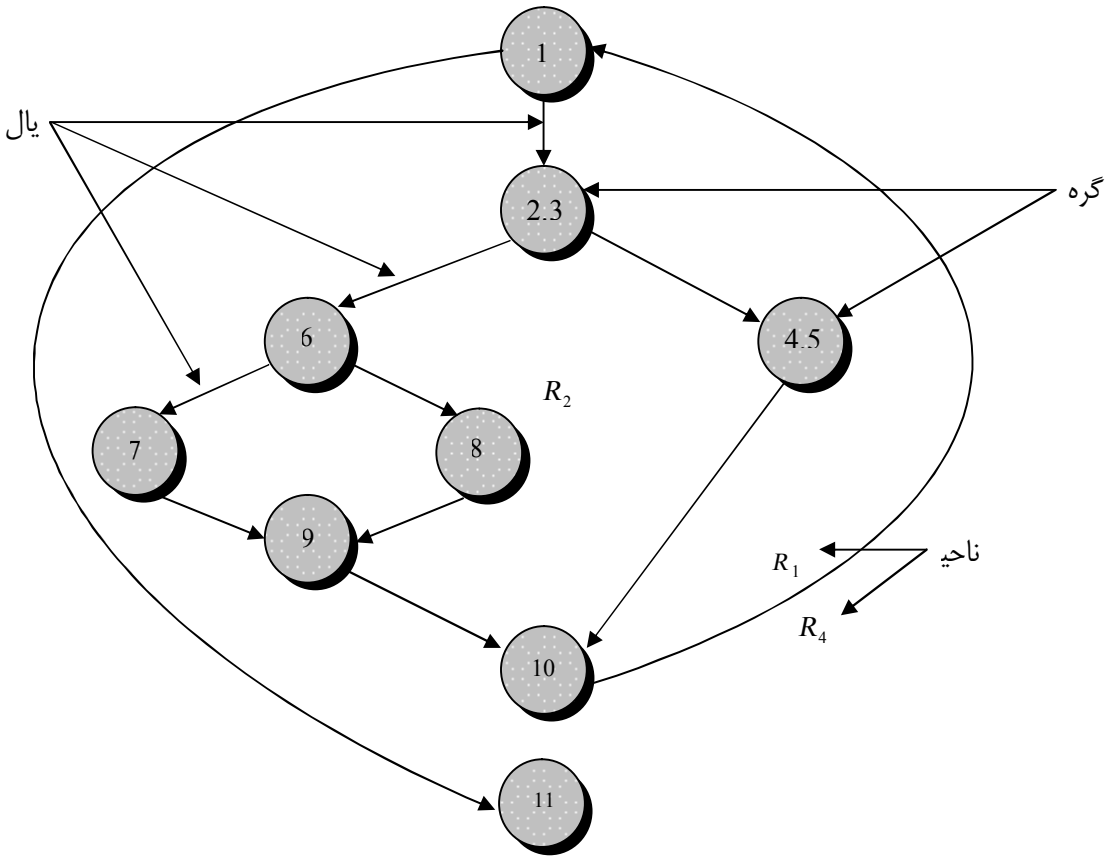


شکل ۱ : نمادگذاری گراف جریان

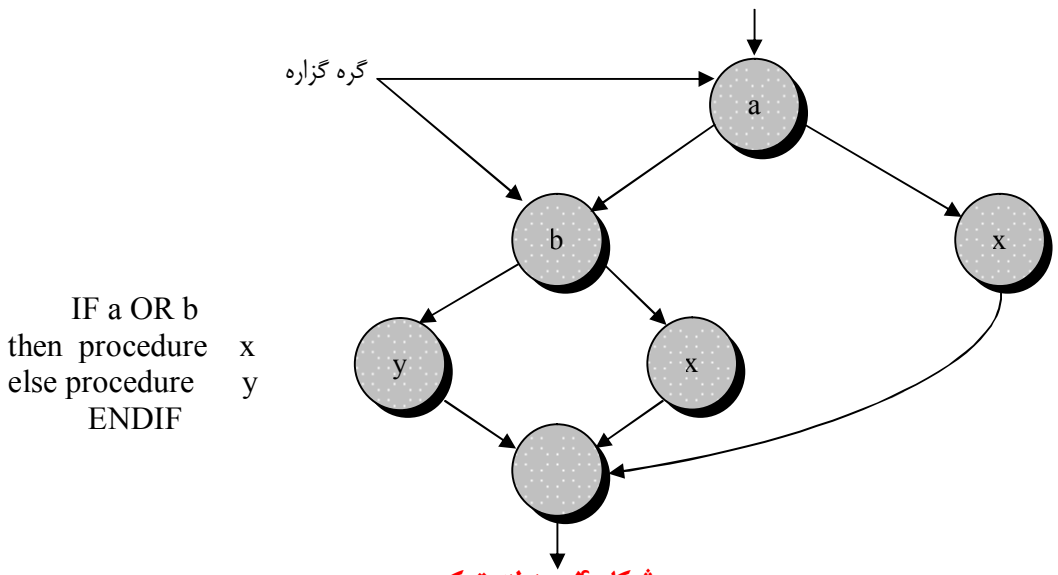
هردایره نشان دهندهٔ یک یا چند PDL بدون انشعاب یا دستورات برنامه مبداء می‌باشد. به مثال زیر توجه کنید.



شکل ۲ : روند نمای یک برنامه



شکل ۳ : گراف جریان مساله فوق



شکل ۴ : منطق ترکیبی

پیچیدگی دورانی(Cyclomatic Complexity)

پیچیدگی دورانی، معیاری از نرم‌افزار است که اندازه‌ای مقداری از پیچیدگی منطقی برنامه را مشخص می‌کند. وقتی در رابطه با روش آزمایش مسیر پایه استفاده می‌شود، مقدار محاسبه شده برای پیچیدگی دورانی، تعداد مسیرهای مستقل

را در مجموعهٔ پایه برنامه مشخص می‌کند و حد بالایی را برای تعداد آزمایش هایی مشخص می‌نماید که باید برای اطمینان از اجرای حداقل یک بار هر یک از دستورات انجام شوند.

یک مسیر مستقل، هر مسیری در برنامه است که حداقل یک مجموعهٔ جدید از احکام پردازش یا شرط جدیدی را مشخص می‌کند. هنگامی که مسیر مستقل، برحسب گراف جریان بیان می‌شود، باید حداقل در مسیر یالی حرکت کند که قبل از تعریف آن مسیر، از آن عبور نشده باشد. برای مثال، مجموعه‌ای از مسیرهای مستقل برای گراف جریان در شکل ۳ عبارتند از:

مسیر ۱: ۱ – 11

مسیر ۲ : ۱ – 2 – 3 – 4 – 5 – 10 – 1 – 11

مسیر ۳ : ۱ – 2 – 3 – 6 – 8 – 9 – 10 –1– 11

مسیر ۴ : ۱ – 2 – 3 – 6 – 7 – 9 – 10 –1– 11

توجه داشته باشید که هر مسیر جدید، یک یال جدید را شامل می‌شود. مسیر زیر:

1 – 2 – 3 – 4 – 5 – 10 – 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

به عنوان مسیر مستقل در نظر گرفته نمی‌شود زیرا فقط ترکیبی از مسیرهای مشخص شدهٔ قبلی است و یال جدیدی را پیمایش نمی‌کند.

وقتی مسیرهای پایه تعیین گردند، باید برای هر مسیر پایه نمونه های آزمایش طراحی کرد و سپس آنها را برای یافتن خطا های احتمالی اجرا نمود.

پیچیدگی دورانی به یکی از این سه شکل زیر محاسبه می‌شود:

۱- **تعداد نواحی گراف جریان متناظر با پیچیدگی دورانی می‌باش**د.

۲- **پیچیدگی دورانی، V(G)، برای گراف جریان، G، به این صورت تعریف می‌شود:**

$$V(G) = E - N + 2$$

که E تعداد یال های گراف جریان، و N تعداد گره‌های گراف جریان می‌باشد.

۳- **پیچیدگی دورانی، V(G)، برای گراف جریان، G، به این صورت نیز تعریف می‌شود:**

$$V(G) = P + 1$$

که P تعداد گزاره‌های موجود در گراف جریان G می‌باشد.

با مراجعه مجدد به گراف جریان شکل ۳، پیچیدگی دورانی با استفاده از هر یک از الگوریتم‌های ذکر شده این گونه محاسبه می‌شود:

۱- گراف جریان دارای چهار ناحیه است.

۲- V(G) = یال 11 - گره 9 + 2 = 4

۳- V(G) = گزارهٔ گزاره 3 + 1 = 4

بنابراین، پیچیدگی دوره‌ای گراف جریان شکل ۳ برابر 4 است.

ماتریس‌های گراف

رویه‌ای برای بدست آوردن گراف جریان و حتی مشخص نمودن مجموعه‌ای از مسیرهای پایه، برای مکانیزه نمودن مناسب می‌باشد. به منظور توسعهٔ ابزاری نرم‌افزاری که بر پایه آزمایش مسیر عمل می‌کند، ساختمان داده‌ای به نام **ماتریس گراف** بسیار مفید است.

ماتریس گراف، ماتریس مربعی است که اندازهٔ آن (یعنی تعداد سطرها و ستونها) برابر است با تعداد گره‌ها در گراف جریان که هر سطر و ستون معادل یک گرهٔ مشخص شده است، و وارده‌های ماتریس معادل ارتباطات (یال های) بین گره‌ها می‌باشند. یک مثال ساده از گراف جریان و ماتریس گراف معادل آن در شکل ۵ نشان داده شده است.

گره \ متصل به گره	۱	۲	۳	۴	۵
۱			a		
۲					
۳		d		b	
۴		c			f
۵		g	e		

شکل ۵ : ماتریس گراف متناظر با گراف جریان

با مراجعه به این شکل، هر گره در گراف جریان با عدد مشخص می‌شود، در حالی که هر یال با یک حرف مشخص می‌گردد. یک حرف در ماتریس در محلی متناظر با ارتباط بین دو گره وارد می‌شود. برای مثال، گرهٔ 3 به گرهٔ 4 با یال b متصل می‌گردد. تا این مرحله، ماتریس گراف چیزی بیش از نمایش جدولی گراف جریان نمی‌باشد. به هر حال، با افزودن ارزش اتصال به هر واردهٔ ماتریس، این ماتریس گراف می‌تواند به ابزاری قدرتمند برای ارزیابی ساختار کنترل برنامه در ضمن آزمایش تبدیل شود. این ماتریس ارزش اتصال اطلاعاتی اضافی در مورد جریان کنترل را فراهم می‌نماید. در ساده‌ترین شکل، ارزش اتصال، 1 (وجود ارتباط) یا صفر (عدم وجود ارتباط) می‌باشد. اما به ارزش های ارتباطی خواص جالب دیگری نیز نسبت داده می‌شود:

- احتمال این که یک اتصال (یال) اجرا می‌شود.
- زمان پردازش صرف شده در ضمن پیمایش اتصال.
- حافظهٔ لازم در ضمن پیمایش اتصال.
- منابع لازم در ضمن پیمایش آن اتصال.

به منظور نمایش این مطلب، ساده‌ترین ارزش را به کار می‌بریم تا ارتباط را نشان دهیم (0 یا 1). ماتریس گراف شکل ۵ دوباره در شکل۶ رسم شده است. هر حرف با 1 جایگزین شده است، و نشان می‌دهد که یک ارتباط وجود دارد (صفرها برای وضوح حذف شده‌اند). با نمایشی به این شکل، ماتریس گراف، ماتریس ارتباط نیز نامیده می‌شود.

گره \ متصل به گره	۱	۲	۳	۴	۵	
۱			1			$1 - 1 = 0$
۲						
۳		1		1		$2 - 1 = 1$
۴		1			1	$2 - 1 = 1$
۵		1	1			$2 - 1 = 1$

$3 + 1 = 4$ ■ ← **پیچیدگی دوره‌ای**

شکل ۶ : ماتریس گراف با در نظر گرفتن ارتباط

آزمایش ساختار کنترل

تکنیک آزمایش مسیر پایه توصیف شده، یکی از چند تکنیک آزمایش ساختار کنترلی است. اگرچه آزمایش مسیر پایه ساده و بسیار مفید است، ولی به تنهایی کافی نیست. در این بخش، حالت‌های دیگر آزمایش ساختار کنترلی بحث می‌شوند. این حالت‌ها پوشش آزمایش را گسترش داده و کیفیت آزمایش جعبه سفید را افزایش می‌دهند.

آزمایش شرط

آزمایش شرط روشی برای طراحی نمونه های آزمایش است که شرط‌های منطقی موجود در یک پیمانهٔ برنامه را بررسی می‌نماید. یک شرط ساده، متغیری بولی یا عبارتی رابطه‌ای است، احتمالاً با عملگر NOT که قبل از آن قرار گرفته. عبارت رابطه‌ای به این شکل است:

$$E_1 < \textbf{عملگر رابطه‌ای} > E_2$$

که E_1 و E_2 عبارت های محاسباتی هستند و < عملگرد رابطه‌ای > شامل یکی از عملگرهای > و ≤ و = ≠ (نامساوی) <، یا ≥ می‌باشد. یک شرط مرکب تشکیل شده است از دو یا چند شرط ساده، عملگرد بولی، و پرانتزها. فرض می‌کنیم که عملگرهای بولی امکان شرط‌های ترکیبی را با اضافه نمودن OR (|)، AND (&) و NOT (¬) می‌دهند. شرطی بدون عبارت های رابطه‌ای، عبارتی بولی است.

اگر یک شرط غلط باشد، حداقل یک مؤلفهٔ آن شرط غلط خواهد بود. بنابراین، انواع خطاها در شرط به شرح زیر هستند که بایـد مورد آزمایش قرار گیرند:

- خطاهای عملگر منطقی (غلط/ حذف شده/ عملگرهای بولی اضافی).
- خطای متغیر بولی.
- خطای پرانتزهای بولی.
- خطای عملگر رابطه‌ای.
- خطای عبارت محاسباتی.

آزمایش جریان داده

روش آزمایش جریان داده، مسیرهای آزمایش برنامه را طبق مکانهای تعاریف و استفاده از متغیرهای برنامه انتخاب می‌کند. چند استراتژی آزمایش جریان داده، مطالعه و مقایسه شده‌اند.

به منظور نمایش شیوهٔ آزمایش جریان داده، فرض کنید که به هر دستور در برنامه یک شمارهٔ دستور منحصر به فرد داده شده است و این که هر تابع، پارامترهای خود یا متغیرهای سراسری را تغییر نمی‌دهد.

برای دستوری با شمارهٔ دستورS:

DEF(S) = {X | دستور S حاوی تعریف X باشد}

USE(S) = {X | دستور S حاوی استفاده‌ای از X باشد}

اگر دستور S، یک دستور if یا حلقه باشد، مجموعهٔ DEF آن خالی است و مجموعهٔ USE آن وابسته به شرط دستور S می‌باشد. تعریف متغیر X در دستور S، در دستور S' زنده است اگر مسیری از دستور S به دستور S' وجود داشته باشد که حاوی تعریف دیگری از X نباشد.

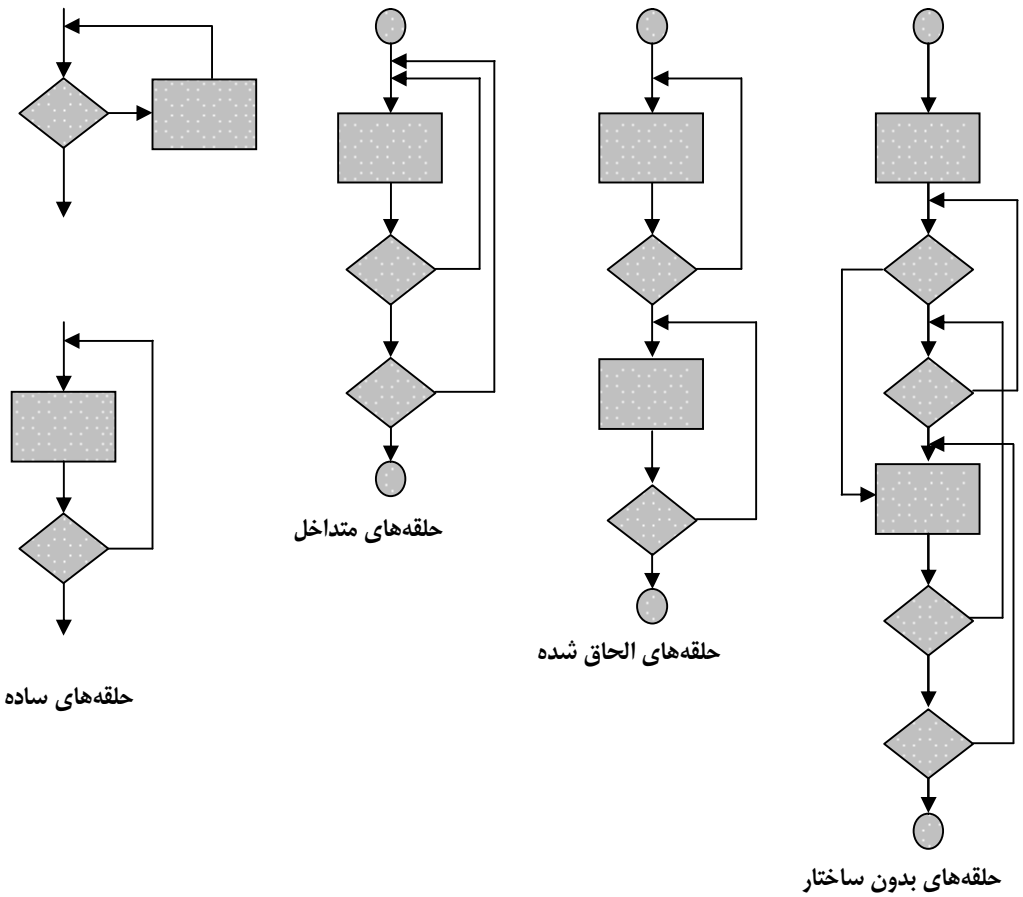
زنجیرهٔ تعریف- استفادهٔ (DU) متغیر X به شکل {x و S و S'} می‌باشد که S و S' شماره‌های دستورات هستند، X در DEF(S) و USE (S') قرار دارد و تعریف X در دستورS ، در دستور S' زنده است.

یک استراتژی سادهٔ آزمایش جریان داده، نیاز دارد که هر زنجیرهٔ DU حداقل یک دفعه پوشش داده شود. به این استراتژی، **استراتژی آزمایش DU** نیز گفته می‌شود.

نشان داده شده است که آزمایش DU پوشش تمام انشعاب‌های برنامه را تضمین نمی‌کند. به هر حال، تضمینی وجود ندارد که یک انشعاب توسط آزمایش DU پوشش داده شود فقط در موارد نادری مانند ساختارهای if-then-else که در آن، بخش then فاقد تعریف متغیر است و بخش else وجود ندارد. در چنین موقعیتی انشعاب else از دستور if لزوماً توسط آزمایش DU پوشش داده نمی‌شود.

آزمایش حلقه

حلقه‌ها برای اکثریت الگوریتم‌های پیاده‌سازی شده در نرم‌افزار نقش محوری دارند. با این وجود، اغلب در ضمن هدایت آزمایش های نرم‌افزار، توجه کمی به آنها می‌شود.



شکل ۷ : انواع حلقه ها

آزمایش حلقه تکنیکی بر پایه آزمایش جعبه سفید می‌باشد که منحصراً بر اعتبار ساختارهای حلقه تأکید دارد. چهار رده از حلقه‌ها قابل تعریف هستند: **حلقه‌های ساده، حلقه‌های الحاق شده، حلقه‌های متداخل و حلقه‌های بدون ساختار** که در شکل ۷ نشان داده شد است.

آزمایش جعبه سیاه

آزمایش جعبه سیاه که آزمایش رفتاری نیز نامیده می‌شود، بر نیازهای تابعی نرم‌افزاری تأکید دارد. یعنی، آزمایش جعبه سیاه باعث می‌شود مهندس نرم‌افزار مجموعه‌هایی از شرایط ورودی را بدست آورد که کاملاً تمام نیازهای تابعی برنامه را بررسی می‌کنند. آزمایش جعبهٔ سیاه راه جایگزینی برای تکنیک جعبهٔ سفید نیست. در عوض، روشی تکمیلی است که احتمالاً ردهٔ متفاوتی از خطاها را نسبت به روش های جعبهٔ سفید آشکار می‌کند.

آزمایش جعبهٔ سیاه سعی در یافتن خطاهایی در دسته‌بندی‌های زیر دارد:

۱- توابع غلط یا حذف شده،
۲- خطاهای واسط ها،
۳- خطا در ساختمان داده‌ها یا دسترسی به بانک اطلاعاتی خارجی،
۴- خطاهای رفتاری یا کارایی، و
۵- خطاهای آماده‌سازی و اختتامیه.

برخلاف آزمایش جعبهٔ سفید که در اوایل فرآیند آزمایش انجام می‌شود، آزمایش جعبهٔ سیاه در مراحل آخر آزمایش به کار گرفته می‌شود. چون آزمایش جعبهٔ سیاه عمداً به ساختارکنترلی توجهی ندارد، توجه بر دامنهٔ اطلاعات متمرکز می‌باشد. آزمایش های برای پاسخگویی به سؤالات زیر طراحی می شوند:

- **چگونه اعتبار عملکردی آزمایش می‌شود؟**
- **چگونه رفتار و کارایی سیستم آزمایش می‌شود؟**
- **چه رده‌هایی از ورودی، نمونه های آزمایش خوبی می‌سازند؟**
- **آیا سیستم مخصوصاً به مقادیر خاص ورودی حساس است؟**
- **چگونه مرزهای یک رده از داده‌ها مجزا می‌شود؟**
- **سیستم چه نواساناتی برای سرعت و حجم داده‌ها دارد؟**
- **ترکیبات خاص داده‌ها چه اثری بر عملکرد سیستم دارند؟**

با بکارگیری روش های آزمایش جعبه سیاه، مجموعه‌ای از نمونه های آزمایشی بدست می‌آیند که معیارهای زیر را برآورده می‌سازند:

۱- نمونه های آزمایشی که باعث کاهش بیش از حد یک واحد از تعداد نمونه های آزمایشی می‌شوند که برای رسیدن به آزمایش قابل قبول مورد نیاز می‌باشند، و

۲- نمونه های آزمایشی که چیزی در مورد حضور یا عدم حضور رده‌هایی از خطاها ارایه دهند. به جای اینکه یک خطا مربوط به یک آزمایش خاص در حال انجام را آشکار نمایند.

تحلیل مقدار مرزی

به دلایلی که کاملاً واضح نیستند، تعداد زیادی از خطاها در مرزهای دامنه ورودی اتفاق می‌افتند، به جای این که در مرکز آن اتفاق بیفتند. به این دلیل است که تحلیل مقدار مرزی (BVA-Boundary Value Analysis) به عنوان یک روش آزمایش توسعه داده شده است. تحلیل مقدار مرزی باعث انتخاب نمونه های آزمایشی می‌شود که مقادیر مرزی را مورد آزمایش قرار می‌دهند.

تحلیل مقدار مرزی یک تکنیک طراحی نمونه های آزمایش می‌باشد که مکمل تقسیم‌بندی مساوی است. به جای انتخاب هر عنصر از ردهٔ مساوی، BVA، انتخاب نمونه های آزمایش را به لبه‌های این رده هدایت می‌کند. به جای تمرکز بر شرایط ورودی، BVA، نمونه های آزمایش را از دامنهٔ خروجی بدست می‌آورد. رهنمودهای زیر در این آزمون راهگشا است:

۱- اگر شرط ورودی محدوده‌ای را با مقادیر a و b مشخص نماید، نمونه های آزمایش باید با مقادیر a و b و اندکی بالاتر و پایین تر از a و b طراحی شوند.

۲- اگر شرط ورودی چند مقدار را مشخص نماید، نمونه های آزمایش باید به گونه‌ای توسعه یابند که اعداد حداکثر و حداقل را بررسی نمایند. مقادیر اندکی بالاتر و پایین‌تر از حداقل و حداکثر نیز آزمایش می‌شوند.

۳- به کارگیری راهنمایی‌های 1 و 2 برای شراط خروجی. برای مثال، فرض کنید که جدول دما در مقابل فشار، به عنوان خروجی یک برنامهٔ تحلیل مهندسی لازم است. نمونه های آزمایش باید برای تولید یک گزارش خروجی طوری ایجاد شوند که حداقل و حداکثر عدد مجاز وارده‌های جدول را تولید نمایند.

۴- اگر ساختمان داده‌های داخل برنامه مرزهای مشخصی دارند (برای مثال، آرایه با محدودیت 100 وارده تعریف شده باشد)، از طراحی نمونه های آزمایشی که این ساختمان داده‌ها و مرزهای آنها را بررسی می‌کند مطمئن شوید.

اکثر مهندسین نرم‌افزار BVA را با درجه‌ای خاص اجرا می‌کنند. با به کارگیری این راهنمایی‌ها، آزمایش مرزی کامل‌تر خواهد شد، و احتمال بیشتری برای آشکارسازی خطا وجود دارد.

آزمایش برای محیط‌ها، معماری‌ها و کاربردهای خاص

نرم‌افزارهای کامپیوتر پیچیده‌تر شده، و نیاز برای شیوه‌های آزمایش خاص نیز رشد نموده است. روش های آزمایش جعبهٔ سیاه و جعبهٔ سفید بحث شده، برای تمام محیط‌ها، معماری‌ها، و کاربردها قابل به کارگیری هستند، اما راهنمایی‌های منحصر به فرد و شیوه‌هایی برای آزمایش گاهی توصیه می‌شوند. در این بخش، راهنمودهای آزمایش محیط‌ها، معماری‌ها و کاربردهای خاصی که به طور متداول مهندسین نرم‌افزار با آنها روبرو می‌شوند ارایه شده است.

آزمایش واسط های گرافیکی کاربران(GUI-Graphical User Interface)

واسط‌های گرافیکی کاربران(GUI ها) زمینهٔ جالبی را برای مهندسین نرم‌افزار ارایه می‌نمایند. به علت اجزاء قابل استفادهٔ مجددی که به عنوان بخشی از محیط‌های توسعهٔ GUI فراهم می‌شوند، ایجاد واسط کاربر زمان کمتری نیاز دارد و دقیق‌تر است. اما در عین حال، پیچیدگی GUIها نیز افزایش یافته است، و باعث مشکلات بیشتر در طراحی و اجرای نمونه های آزمایش می‌شود.

چون بسیاری از GUIهای مدرن، احساس و جلوهٔ یکسانی دارند، یک سری از آزمایش های استاندارد قابل انجام است. گراف های مدلسازی حالت محدود می‌توانند مورد استفاده واقع شوند تا یکسری آزمایش هایی را ایجاد کنند که اشیاء و داده‌های خاص مربوط به GUI برنامه را مورد توجه قرار دهند.

به دلیل ترکیبات زیاد اعمال GUI، آزمایش باید با نمونه های خودکار انجام شود. دستهٔ بزرگی از نمونه های آزمایش GUI در بازار در چند سال گذشته عرضه شده‌اند.

آزمایش معماری مشتری/ کارگزار

معماری‌های مشتری/ کارگزار (C-S) موارد مهمی را برای آزمایش کننده‌های نرم‌افزار نشان می‌دهند. ماهیت توزیع شدهٔ محیط‌های C-S، موارد کارایی مربوط به پردازش تراکنش، حضور بالقوهٔ سکوهای سخت‌افزاری متفاوت، پیچیدگی های ارتباط شبکه، نیاز به رایه سرویس به چندین مشتری از بانک اطلاعاتی متمرکز (یا توزیع شده)، و نیازهای هماهنگ‌سازی تحمیل شده بر کارگزار، همگی ترکیب می‌شوند و باعث می‌شوند آزمایش معماری‌های C-S، و نرم‌افزاری که بر روی آنها قرار می‌گیرد، تا حد قابل توجهی مشکل‌تر از کاربردهای مجزا باشد. در واقع، مطالعات اخیر صنایع نشان می‌دهد که افزایش عمده‌ای در زمان و هزینهٔ آزمایش محیط‌های C-S وجود دارد.

مستندسازی آزمایش و امکانات کمک

واژهٔ آزمایش نرم‌افزار، شامل تصاویری از تعداد زیادی از نمونه های آزمایش می‌باشد که برای بررسی برنامه‌های کامپیوتری و داده‌هایی که دستکاری می‌کنند آماده شده‌اند. تعریف نرم‌افزار را که در اولین فصل ارایه شد به خاطر آورید، توجه به این نکته مهم است که آزمایش باید به سومین عنصر پیکربندی نرم‌افزار، یعنی مستندات، توسعه یابد.

خطاها در مستندات می‌توانند به همان اندازهٔ خطاها که در برنامه های مبدأ یا داده‌ها، باعث اشکالاتی در قبول برنامه شوند. هیچ نگران کننده‌تر از این نمی‌باشد که راهنمای کاربر یا امکان کمک سیستم دقیقاً دنبال شود و به نتایج یا رفتاری منتهی شود که منطبق با آنچه توسط مستندات پیش‌بینی شده نباشد. به این دلیل است که آزمایش مستندات باید بخشی معنی‌داری برای هر طرح آزمایش نرم‌افزار باشد.

آزمایش مستندات در دو فاز قابل انجام است. اولین فاز که مرور و بازبینی نام دارد (فصل ۸)، مستندات را برای وضوح ویرایشی بررسی می‌کند. فاز دوم، که آزمایش زنده نام دارد، مستندات را همراه با استفاده از برنامهٔ واقعی، مورد استفاده قرار می‌دهد.

به شکل تعجب‌آوری، آزمایش زنده برای مستندات با استفاده از تکنیک‌هایی مشابه بسیاری از روش های جعبهٔ سیاه، قابل انجام است. آزمایش بر مبنای گراف می‌تواند استفاده از برنامه را توصیف کند. تقسیم‌بندی مساوی و تحلیل مقدار مرزی برای تعریف رده‌های گوناگون ورودی و ارتباطات مربوط به آنها استفاده می‌شوند. سپس استفاده از برنامه از طریق مستندات پیگیری می‌شود. سؤالات زیر باید در طول هر فاز پاسخ داده شوند:

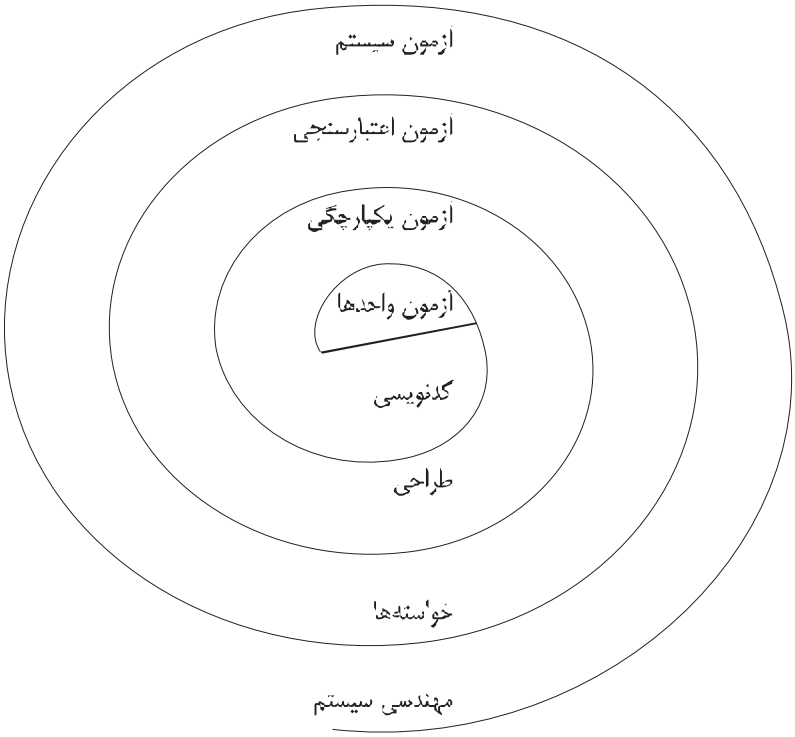
- **آیا مستندسازی به طور دقیق چگونگی استفاده از هر روش را توصیف می کند؟**
- **آیا توصیف هر دنبالهٔ ارتباط دقیق است؟**
- **آیا مثال ها دقیق هستند؟**
- **آیا بکارگیری لغات، توصیف‌های منوها، و پاسخ‌های سیستم منطبق بر برنامهٔ واقعی است؟**
- **آیا یافتن راهنمایی در مستندات نسبتاً ساده است؟**
- **آیا رفع اشکال با استفاده از مستندات به راحتی انجام می‌شود؟**
- **آیا فهرست مندرجات و اندیس مستندات کامل و دقیق است؟**
- **آیا طراحی مستندات (اجزاء، تایپ ظاهری، کنگره‌بندی، گرافیک‌ها) برای فهم و درک سریع اطلاعات مؤثر هستند؟**
- **آیا تمام پیغام های خطای نرم‌افزار که برای کاربر ظاهر می‌شوند، با جزییات بیشتر در مستندات توصیف شده‌اند؟ آیا اعمال قابل انجام در نتیجهٔ پیغام خطا، به وضوح بیان شده‌اند؟**
- **اگر ارتباطات فرامتنی (Hypertext) استفاده می‌شوند، آیا دقیق و کامل هستند؟**
- **اگر فرامتنی استفاده می‌شود، آیا طراحی نحوه حرکت در اتصالات، برای اطلاعات مورد نیاز مناسب است؟**

تنها راه عملی برای پاسخ به این سؤالات، وجود گروهی مستقل (برای مثال، کاربران انتخاب شده) است که مستندات را در رابطه با استفاده از برنامه آزمایش نمایند. تمام تفاوت ها تعیین و ذکر شوند و زمینه‌های ابهام یا ضعف، برای بازنویسی مجدد تعریف گردند.

استراتژی آزمایش نرم‌افزار

فرآیند مهندسی نرم‌افزار می‌تواند به صورت یک مارپیچی در نظر مانند شکل ۸ در نظر گرفته شود. در ابتدا، مهندس سیستم، نقش نرم‌افزار را تعریف می‌کند و به تحلیل نیازهای نرم‌افزار وارد می‌شود، که دامنه اطلاعات، عملکرد، رفتار، کارایی، محدودیت‌ها، و معیارهای اعتبارسنجی برای نرم‌افزار ایجاد می‌شود. با حرکت در مارپیچ به سمت طراحی و در نهایت برنامه نویسی می‌رسیم. به منظور توسعهٔ نرم‌افزار کامپیوتر، به سمت داخل مارپیچ حرکت می‌کنیم، در طول خطی که سطح انتزاع را در هر دور کاهش می‌دهد.

یک استراتژی برای آزمایش نرم‌افزار می‌تواند در زمینهٔ این مارپیچ مورد توجه قرار گیرد. **آزمایش واحد**، از مرکز مارپیچ شروع می‌شود و بر روی هر واحد(یعنی مؤلفه) نرم‌افزار متمرکز می‌باشد که با برنامه های مبدأ پیاده‌سازی شده است. با حرکت به سمت خارج در مارپیچ، آزمایش به سمت آزمایش **یکپارچه‌سازی** ادامه می‌یابد. این آزمایش بر طراحی و ساخت معماری نرم‌افزار تأکید دارد. با حرکت به اندازه یک دور دیگر به سمت خارج در طول مارپیچ، به آزمایش **اعتبارسنجی** می‌رسیم.

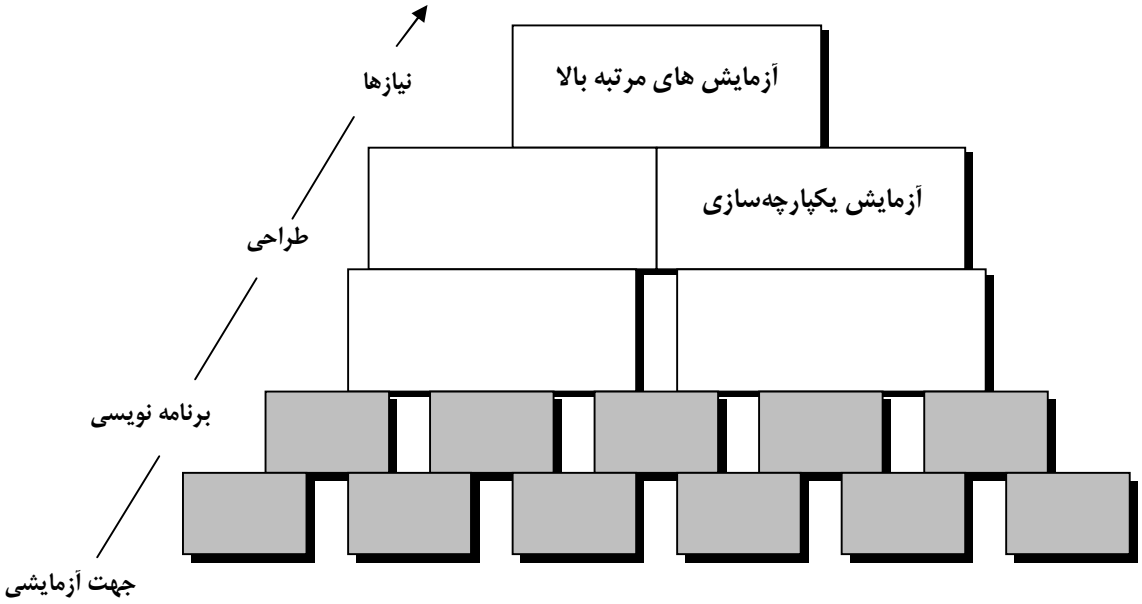


شکل ۸ : استراتژی آزمایش

نیازهایی که به عنوان بخشی از تحلیل نیازهای نرم‌افزار ایجاد شده‌اند، در مقابل نرم‌افزاری که ساخته شده اعتبارسنجی می‌شوند. در نهایت، به آزمایش **سیستم** می‌رسیم، که در آن نرم‌افزار و عناصر دیگر سیستم به صورت یک مجموعه آزمایش می‌شوند. به منظور آزمایش نرم‌افزار کامپیوتر، در طول مارپیچ حرکت نموده و در هر دور، محدودهٔ آزمایش گسترده‌تر می‌شود.

این فرآیند را از نطقه نظر رویه‌ای در نظر بگیرید، آزمایش در زمینهٔ مهندسی نرم‌افزار در واقع شامل چهار مرحله است که به صورت ترتیبی پیاده‌سازی شده‌اند. این مراحل در شکل ۹ نشان داده شده‌اند. در ابتدا، آزمایش بر هر مؤلفه به صورت منفرد تمرکز دارد، تا اطمینان حاصل شود به صورت یک واحد، درست کار می‌کند. در نتیجه، نام این مرحله، آزمایش واحد می‌باشد. **آزمایش واحد، استفاده زیادی از تکنیک‌های آزمایش جعبهٔ سفید می‌برد،** و مسیرهای خاصی را در ساختار کنترلی پیمانه بررسی می‌کند تا اطمینان حاصل شود پوشش کاملی داده شده و حداکثر خطاها آشکار می‌شوند. سپس، مؤلفه‌ها باید مونتاژ یا مجتمع شوند تا بستهٔ نرم‌افزاری کامل را تشکیل دهند. **آزمایش یکپارچه‌سازی، مسایل مربوط به مشکلات دوگانهٔ بازبینی و ساخت برنامه را مورد توجه قرار می‌دهد.** تکنیک‌های **طراحی نمونه های آزمایش جعبهٔ سیاه، در زمان یکپارچه‌سازی بیشترین استفاده را دارند،** اگرچه مقدار محدودی آزمایش جعبهٔ سفید نیز می‌تواند استفاده شود تا از پوشش اکثر مسیرهای کنترلی اطمینان حاصل شود. پس از یکپارچه‌سازی نرم‌افزار (ساخته شدن آن)، مجموعه‌ای از آزمایش های مرتبه بالا هدایت می‌شوند. معیارهای اعتبارسنجی (ایجاد شده در ضمن تحلیل نیازها) باید آزمایش شوند. **آزمایش اعتبارسنجی، اطمینان نهایی را ایجاد می‌کند که نرم‌افزار تمام نیازهای عملکردی، رفتاری، و کارایی را برآورده می‌نماید. تکنیک‌های آزمایش جعبهٔ سیاه به طور انحصاری در ضمن اعتبارسنجی استفاده می‌شوند.**

آخرین مرحلهٔ آزمایش مرتبه بالا، خارج از مرز مهندسی نرم‌افزار است و در مرز زمینهٔ مهندسی سیستم کامپیوتری است. نرم‌افزار، پس از اعتبارسنجی، باید با عناصر دیگر سیستم ترکیب شود(برای مثال، سخت‌افزار، افراد، بانک‌های اطلاعاتی). **آزمایش سیستم بازبینی می‌کند که تمام عناصر به طور منظم مرتبط شده باشند و این که عملکرد و کارایی کل سیستم نیز بدست آمده باشد.**



شکل ۹ : مراحل آزمایش نرم افزار

آزمایش واحد(Unit Testing)

آزمایش واحد، بر فعالیت بازبینی کوچکترین واحد طراحی نرم‌افزار، که مؤلفهٔ نرم‌افزار یا پیمانه نامیده می‌شود تمرکز دارد. با استفاده از توصیف طراحی در سطح مؤلفه به عنوان راهنما، مسیرهای کنترلی مهم آزمایش می‌شوند تا خطاهای موجود در مرز پیمانه پیدا شوند. پیچیدگی نسبی آزمایش ها و خطاهای آشکار شده، با محدودهٔ ایجاد شده برای آزمایش واحد، محدود می‌شود. آزمایش واحد، گرایش به جعبهٔ سفید دارد، و این مرحله می‌تواند به موازات برای چند مؤلفه هدایت شود.

در بین خطاهای متداول محاسبه، این موارد مشاهده می‌شوند:

۱- اولویت محاسباتی نادرست،

۲- اعمال ترکیبی،

۳- آماده‌سازی غلط،

۴- عدم دقت کافی در محاسبه،

۵- نمایش غلط یک نماد محاسباتی.

مقایسه و کنترل جریان تا حد زیادی به یکدیگر نزدیک هستند (یعنی تغییر جریان، معمولاً بعد از مقایسه انجام می‌شود). نمونه های آزمایش باید خطاهایی را از این قبیل آشکار نمایند:

(۱) مقایسهٔ انواع دادهٔ متفاوت،

(۲) عملگرهای منطقی یا اولویت نادرست،

(۳) داشتن انتظار تساوی در زمانی که خطا در دقت محاسبه باعث عدم تساوی می‌شود،

(۴) مقایسهٔ غلط متغیرها،

(۵) خاتمهٔ حلقهٔ نامناسب یا عدم وجود خاتمهٔ حلقه،

(۶) شکست در خروج، زمانی که حلقهٔ نامتناهی تشخیص داده می‌شود،

(۷) اصلاح نامناسب متغیرهای حلقه.

آزمایش یکپارچه سازی(Integration Testing)

یک فرد مبتدی در دنیای نرم‌افزار ممکن است سؤالی به ظاهر ساده را بعد از این که آزمایش واحد بر روی تمام پیمانه‌ها صورت گرفت بپرسد: " اگر همهٔ آنها به تنهایی کار می‌کنند، چرا مشکوک هستید که آنها وقتی کنار هم قرار گیرند کار می‌کنند یا خیر؟ " این مشکل، کنار هم قرار دادن مولفه ها، یعنی ارتباط بین آنها است. داده ممکن است در یک ارتباط ناپدید شود. یک پیمانه ممکن است اثر نامطلوب و ناخواسته‌ای را بر دیگری داشته باشد. زیر توابع، زمانی که با هم ترکیب می‌شوند، ممکن است تابع بزرگتر مورد نظر را تولید نکنند. مقادیر نادقیقی که در هر یک به تنهایی پذیرفته شده‌اند، ممکن است بزرگ شوند و به سطوح غیرقابل قبول برسند. ساختمان داده‌های سراسری ممکن است مشکل‌ساز شود. این لیست همچنان به طور نگران‌کننده‌ای ادامه دارد.

آزمایش یکپارچه‌سازی، روشی سیستماتیک برای ایجاد ساختار برنامه است در حالی که، آزمایش‌ها نیز انجام می‌شوند تا خطاهای مربوط به واسطها آشکار شوند. هدف، دریافت مؤلفه‌های آزمایش واحد و ایجاد ساختار برنامه‌ای است که توسط طراح دیکته شده است.

یکپارچه سازی بالا به پایین

آزمایش یکپارچه‌سازی بالا به پایین، روشی افزایشی برای ایجاد ساختار برنامه است. پیمانه‌ها با حرکت به سمت پایین در سلسله مراتب کنترل، مجتمع می‌شوند. کار، با پیمانهٔ کنترل اصلی (Main Program) شروع می‌شود. پیمانه‌های پایین‌تر(تقریباً پایین‌تر) نسبت به پیمانهٔ کنترل اصلی در این ساختار به صورت عمقی یا سطحی یکپارچه می‌شوند.

یکپارچه‌سازی شامل پنج مرحله است:

۱- پیمانهٔ کنترل اصلی (Main) به عنوان گردانندهٔ آزمایش استفاده می‌شود و جانگهدارها(Stubs) به جای تمام مؤلفه‌هایی که مستقیماً در سطح بعدی پیمانه کنترل اصلی قرار دارند، جایگزین می‌شوند.
۲- برحسب روش مجتمع‌سازی انتخاب شده، (یعنی، سطحی یا عمقی)، در هر مرحله، یک جانگهدار با پیمانهٔ اصلی در سطوح بعدی جایگزین می‌شود.
۳- آزمایش ها در ضمن مجتمع شدن هر مؤلفه هدایت می‌شوند.
۴- با تکمیل هر مجموعه از آزمایش ها، جانگهدار دیگری با مؤلفه اصلی جایگزین می‌شود.
۵- آزمایش رگرسیون انجام می‌شود تا اطمینان حاصل شود خطاهای جدیدی هنوز شناسایی نشده‌اند.

یکپارچه‌سازی پایین به بالا

آزمایش یکپارچه‌سازی پایین به بالا، همانطوری که از نامش مشخص می‌باشد، ساخت و آزمایش را با پیمانه‌های اتمی شروع می‌کند (یعنی، مؤلفه‌های پایین‌ترین سطوح ساختار برنامه). چون مؤلفه‌ها از پایین به بالا کنارهم قرار می‌گیرند، پردازش‌های لازم برای مؤلفه‌های سطح بعدی همیشه در دسترس می‌باشند و نیاز به جانگهدار مرتفع می‌گردد.

یک استراتژی یکپارچه‌سازی پایین به بالا با مراحل زیر پیاده‌سازی می‌شود:

۱- مؤلفه‌های سطح پایین در قالب خوشه‌هایی(Clusters) ترکیب می‌شوند که زیر تابع خاصی از نرم‌افزار را انجام دهند.
۲- گرداننده‌ای (برنامهٔ کنترل کنندهٔ آزمایش)(Drivers) نوشته می‌شود تا ورودی- خروجی نمونه های آزمایش را هماهنگ نماید.
۳- خوشه آزمایش می‌شود.
۴- گرداننده‌ها حذف می‌شوند، خوشه‌ها ترکیب می‌شوند، و حرکت به سمت بالا در ساختار کنترلی ادامه می‌یابد.

آزمایش رگرسیون(Regression Testing)

هر دفعه که پیمانهٔ جدیدی به عنوان بخشی از آزمایش یکپارچه‌سازی افزوده می‌شود، نرم‌افزار تغییر می‌کند. مسیرهای جریان دادهٔ جدیدی ایجاد می‌شوند، I/O جدیدی انجام می‌گیرد، و منطق کنترل جدیدی فراخوانی می‌شود. این تغییرات ممکن است باعث بروز مشکلاتی با توابعی شوند که قبلاً بدون خطا کار می‌کردند. در رابطه با استراتژی آزمایش یکپارچه سازی، آزمایش رگرسیون، اجرای مجدد زیر مجموعه‌ای از آزمایش هایی است که قبلاً انجام شده‌اند تا اطمینان حاصل شود که تغییرات، باعث انتشار اثرات جانبی ناخواسته نشده‌اند. در محدودهٔ وسیع‌تر، آزمایش های موفقیت‌آمیز (از هر نوع) باعث کشف خطاها می‌شوند، و خطاها باید اصلاح شوند. هر زمانی که نرم‌افزار اصلاح می‌شود، جنبه‌ای از پیکربندی نرم‌افزار (برنامه، مستندات، یا داده‌هایی که آنها را حمایت می‌کنند) تغییر می‌نماید. آزمایش رگرسیون می‌تواند به صورت دستی هدایت شود. این عمل با اجرای مجدد زیر مجموعه‌ای از تمام نمونه‌های آزمایش استفاده از ابزارهای خودکار Capture/Playback انجام می‌شود. ابزارهای capture/playback باعث می‌شوند مهندس نرم‌افزار نمونه‌های آزمایش را دریافت کند و با حرکت به عقب، مقایسه‌هایی را انجام دهد.

مجموعه آزمایش رگرسیون (زیرمجموعه‌ای از آزمایش هایی که باید انجام شوند) شامل سه ردهٔ متفاوت از نمونه‌های آزمایش می‌باشد:

- **نمونه‌هایی از آزمایش‌هایی که تمام عملکرد نرم‌افزار را بررسی می‌نمایند.**
- **آزمایش‌های اضافی که بر عملکردهایی از نرم‌افزار تأکید دارند که احتمالاً با این تغییرات تحت تأثیر قرار می‌گیرند.**
- **آزمایش‌هایی که بر مؤلفه‌های تغییر یافته در نرم‌افزار تأکید دارند.**

آزمایش دوُد(Smoke Test)

آزمایش دوُد یک روش یکپارچه‌سازی است که به طور متداول زمانی استفاده می‌وشد که محصولات نرم‌افزاری کم اهمیت توسعه داده می‌شوند. به عنوان مثال، مکانیزمی سریع و مرحله‌ای برای پروژه‌هایی که حساسیت زمانی دارند استفاده می‌شود و به تیم نرم‌افزار امکان می‌دهد پروژه را به تدریج انجام دهد. در نتیجه، روش آزمایش دود شامل فعالیت‌های زیر است:

۱- مؤلفه‌های نرم‌افزاری که به کد ترجمه شده‌اند، در قالب یک " بنا " یکپارچه می شوند. یک بنا شامل تمام فایل‌های داده، کتابخانه‌ها، پیمانه‌های قابل استفاده مجدد، و مؤلفه‌های ایجاد شده با فرآیند مهندسی است که برای پیاده‌سازی یک یا چند تابع محصول مورد نیاز می‌باشند.

۲- یک سری از آزمایش ها طراحی می‌شوند تا خطاهایی را آشکار نمایند که باعث می‌شوند یک بنا به طور منظم عمل خود را انجام ندهد. هدف، یافتن خطاهای بازدارنده‌ای است که بالاترین احتمال به تأخیر انداختن پروژه را دارند.

۳- این بنا، با بناهای دیگر یکپارچه می‌شود و محصول کامل (به شکل جاری) به صورت روزانه با این روش آزمایش می‌گردد. روش یکپارچه‌سازی می‌تواند بالا به پایین یا پایین به بالا باشد.

آزمایش اعتبارسنجی(Validation Testing)

در نتیجهٔ آزمایش یکپارچه‌سازی، نرم‌افزار به طور کامل به صورت یک بسته مونتاژ می‌شود، خطاهای واسطها آشکار و برطرف می‌شوند، و سری نهایی آزمایش‌های نرم‌افزار با عنوان آزمایش اعتبارسنجی شروع می‌شود. اعتبارسنجی می‌تواند به چندین روش تعریف شود، اما یک تعریف ساده این است که اعتبارسنجی موفق است اگر عملکرد نرم‌افزار به صورتی باشد که مورد انتظار کاربر می‌باشد. در این نقطه، یک توسعه‌دهنده نرم‌افزار پرخاشگر ممکن است اعتراض نماید، "چه کسی یا چه چیزی مشخص کنندهٔ انتظارات منطقی است؟ ".

انتظارات منطقی در مشخصهٔ نیازهای نرم‌افزار تعریف شده‌اندکه سندی است توصیف کنندهٔ تمام صفات قابل رؤیت نرم‌افزار. این مشخصه شامل بخشی است به نام معیارهای اعتبارسنجی، اطلاعات موجود در این بخش، مبنایی برای روش آزمایش اعتبارسنجی خواهد بود.

معیارهای آزمایش و اعتبارسنجی

اعتبارسنجی نرم‌افزار از طریق یک سری آزمایش های جعبه سیاه بدست می‌آید که تطابق با نیازها را مشخص می‌کند. یک طرح آزمایش، رده‌هایی از آزمایش را مشخص می‌کند که باید هدایت شوند، و یک رویهٔ آزمایش نمونه‌های آزمایش خاصی را تعریف می‌کند که برای نمایش تطابق با نیازها استفاده می‌شوند. این طرح و رویه، هر دو طراحی می‌شوند تا مطمئن حاصل گردد که تمام نیازهای تابعی برآورده شده‌اند، تمام خصوصیات رفتاری بدست آمده‌اند، تمام نیازهای کارایی حاصل شده‌اند، مستندسازی صحیح است، و نیازهای دیگر برآورده شده‌اند (برای مثال، قابلیت حمل، سازگاری، پوشش دادن به خطا و قابلیت نگهداری).

مرور پیکربندی

یک عنصر مهم فرآیند اعتبارسنجی، مرور پیکربندی است. ماهیت این مرور، حصول اطمینان از این است که تمام عناصر پیکربندی نرم‌افزار به طور مناسب توسعه داده شده باشند، ثبت شده باشند، و شامل جزییات لازم برای مرحله حمایت در دوره زندگی نرم‌افزار باشند. مرور پیکربندی گاهی تطبیق نامیده می‌شود و با جزییات در فصل ۹ بحث شده است.

آزمایش های آلفا و بتا(Alpha & Beta Testing)

برای توسعه دهندهٔ نرم‌افزار غیر ممکن است که پیش‌بینی نماید مشتری به طور صحیح و واقعی برنامه را مورد استفاده قرار می‌دهد. دستورات ممکن است به غلط تعبیر شوند، ترکیبات عجیبی از داده‌ها ممکن است به طور معمول استفاده شود، یک خروجی که برای آزمایش کننده واضح است، ممکن است برای کاربر غیرقابل درک باشد.

هنگامی که نرم‌افزاری متداول برای مشتری ایجاد می‌شود، یک سری آزمایش‌های پذیرش انجام می‌شوند تا باعث شوند مشتری تمام نیازها را اعتبارسنجی نماید. **آزمایش پذیرش** به جای مهندسین نرم‌افزار، توسط کاربر نهایی انجام می‌شود، و می‌تواند شامل هدایت غیر رسمی یا یک سری آزمایش‌های برنامه‌ریزی شده و سیستماتیک باشد. در واقع، آزمایش پذیرش می‌تواند در بازهٔ زمانی هفته‌ها یا ماه‌ها انجام گردد، و خطاهای موجود که ممکن است کارایی سیستم را در طول زمان کاهش دهند، کشف گردند.

اگر نرم‌افزار به صورت بسته نرم‌افزاری توسعه داده شود که توسط کاربران متعددی اجرا می‌گردد، اجرای آزمایش‌های پذیرش با هر یک، غیر عملی خواهد بود. اکثر سازندگان محصولات نرم‌افزاری، فرآیندی را به نام **آزمایش آلفا و بتا** استفاده می‌کنند تا خطاهایی را که به نظر می‌رسد فقط کاربر نهایی می‌تواند بیابد کشف نمایند.

آزمایش آلفا در سایت توسعه دهنده توسط مشتری انجام می‌شود. نرم‌افزار با تنظیمات معمول استفاده می‌شود و توسعه‌دهنده بر آن نظارت دارد و خطاها را ثبت می‌نماید. آزمایش های آلفا در **محیطی کنترل شده** انجام می‌شوند.

آزمایش بتا در یک یا چند سایت مشتری توسط کاربر نهایی نرم‌افزار انجام می‌شود. برخلاف آزمایش آلفا، توسعه دهنده عموماً حضور ندارد. بنابراین آزمایش بتا، بکارگیری زندهٔ نرم‌افزار در محیطی است که توسط توسعه دهنده قابل کنترل نیست. مشتری تمام مشکلات را (واقعی یا خیالی) که در طول آزمایش بتا شناسایی می‌شوند ثبت می‌کند و این گزارشات را به توسعه دهنده در بازه‌های زمانی منظم تحویل می‌دهد. در نتیجهٔ مشکلات گزارش شده در ضمن آزمایش های بتا، مهندسین نرم‌افزار اصلاحات را انجام می‌دهند و برای انتشار محصول نرم‌افزار به مشتری آماده می‌شوند.

آزمایش سیستم(System Testing)

در ابتدای بحث، بر این حقیقت تأکید داشتیم که نرم‌افزار فقط یک عنصر از یک سیستم بزرگ کامپیوتری است. به طور تقریبی، نرم‌افزار با عناصر دیگر سیستم یکپارچه می‌شود(برای مثال، سخت‌افزار، افراد، اطلاعات)، و یک سری آزمایش های یکپارچه‌سازی و اعتبارسنجی نیز انجام می‌گردد. این آزمایش‌ها خارج از محدودهٔ فرآیند نرم‌افزار هستند و کاملاً توسط مهندس نرم‌افزار صورت نمی‌گیرد. به هرحال، مراحل انجام شده در ضمن طراحی و آزمایش نرم‌افزار، تا حد زیادی احتمال یکپارچه‌شدن نرم‌افزار موفق در یک سیستم بزرگ ارتقاء می‌دهند.

یک مشکل کلاسیک آزمایش سیستم انداختن تقصیر به گردن دیگری است. این حالت زمانی اتفاق می‌افتد که خطایی یافت شود و هر عضو توسعهٔ سیستم، دیگری را مسئول آن مشکل می‌داند. به جای پرداختن به این مسائل بی‌ارزش، مهندس نرم‌افزار باید متوجه مشکلات ارتباطی باشد و

۱- مسیرهای ادارهٔ خطایی طراحی کند که تمام اطلاعات دریافت شده از عناصر دیگر سیستم را آزمایش می‌کنند،

۲- یک‌سری آزمایش‌ها را انجام ده که داده‌های نامناسب یا خطاهای بالقوه دیگر در رابطه نرم‌افزار را شبیه‌سازی کنند،

۳- نتایج آزمایش‌ها برای استفاده به عنوان شاهد ثبت شوند، تا تقصیر به گردن دیگری انداخته نشود،

۴- در برنامه‌ریزی و طراحی آزمایش‌های سیستم شرکت کند تا مطمئن شود که نرم‌افزار به طور مناسبی آزمایش می‌شود.

آزمایش احیاء(Recovery Testing)

بسیاری از سیستم‌های کامپیوتری باید بعد از بروز خطا قابل بازیافت باشند و پردازش را در زمان مشخص شده ادامه دهند. سیستم باید در مقابل اشکال مقاوم باشد، یعنی، خطاهای پردازش نباید باعث شوند کل عملکرد سیستم خاتمه یابد. در موارد دیگر، شکست سیستم باید در بازهٔ زمانی مشخص اصلاح شود در غیر این صورت ضربهٔ اقتصادی شدیدی ایجاد می‌شود. آزمایش بازیافت نوعی آزمایش سیستم است که باعث شکست نرم‌افزار به روش های گوناگون می‌شود و بازبینی می‌کند که آیا بازیافت به طور مناسبی انجام می‌شود.

آزمایش امنیت(Security Testing)

هر سیستم کامپیوتری که اطلاعات حساس را مدیریت می‌کند یا اعمالی انجام می‌دهد که می‌تواند باعث ضرر رساندن (یا فایده رساندن) به افراد شود، هدفی برای نفوذ غیرقانونی یا نامناسب می‌باشد. نفوذ شامل محدوده وسیعی از فعالیت‌ها است:

۱- افراد مهاجمی که سعی در نفوذ به سیستم‌ها را برای تفریح دارند،

۲- کارمندان ناراضی که سعی در نفوذ برای انتقام دارند،

۳- افراد متقلبی که سعی در نفوذ برای رسیدن به اهداف شخصی دارند.

آزمایش امنیت سعی در بازبینی مکانیزم‌های امنیتی ایجاد شده در سیستم دارد تا مطمئن شود سیستم را از نفوذ غیرقانونی محافظت می‌نمایند. Beizer چنین می‌گوید: *" امنیت سیستم باید برای آسیب‌پذیر نبودن از حملهٔ مستقیم آزمایش شود، اما باید برای آسیب‌پذیر نبودن از حملهٔ جانبی یا کناری نیز آزمایش شود".*

آزمایش فشار(Stress Testing)

در ضمن مراحل اولیهٔ آزمایش نرم‌افزار، تکنیک‌های جعبهٔ سفید و جعبهٔ سیاه، عملکردهای معمول و کارایی متداول سیستم را ارزیابی می‌نمایند. آزمایش‌های فشار طراحی می‌شوند تا برنامه‌ها را با موقعیت‌های غیر معمول مواجه نمایند. در نتیجه، آزمایش کننده‌ای که آزمایش فشار را انجام می‌دهد سؤال می‌نماید که: " **قبل از شکست تا چه مدت می‌توان آن را در حال کار نگهداشت؟**".

آزمایش فشار سیستم را به روشی اجرا می‌کند که منابع با کمیت، تکرار، یا حجم غیرمعمول درخواست شوند. برای مثال،

۱- آزمایش های خاصی می‌توانند طراحی شوند تا ده وقفه را در ثانیه تولید کنند، در زمانی که یک یا دو عدد وقفه سرعت متوسط باشد،

۲- سرعت داده‌های ورودی افزایش داده می‌شود تا حدی که مشخص شود چگونه توابع ورودی پاسخ می‌دهند،

۳- نمونه‌های آزمایش که حداکثر حافظه یا منابع دیگر را نیاز دارند اجرا می‌شوند،

۴- نمونه‌های آزمایشی طراحی می گردد که باعث صدمه به سیستم عامل شوند،

۵- نمونه‌های آزمایشی طراحی می‌شوند که باعث دستیابی مداوم به داده‌های ریسک می‌شوند. ضرورتاً، آزمایش کننده سعی در خراب کردن داده‌های سیستم دارد.

آزمایش کارایی(Performance Testing)

برای سیستم‌های بلادرنگ و توکار، نرم‌افزاری که تابع مورد نیاز را فراهم می‌کند اما منطبق بر لوازم کارایی نمی‌باشد، قابل قبول نیست. آزمایش کارایی برای آزمایش کارایی زمان اجرای نرم‌افزار در رابطه با سیستم یکپارچه شده است. آزمایش کارایی در طول تمام مراحل فرآیند آزمایش صورت می‌گیرد. حتی در سطح واحد، کارایی هر پیمانه ممکن است با آزمایش های جعبهٔ سفید بدست آید. به هر حال، تا زمانی که تمام عناصر سیستم به طور کامل یکپارچه نشده باشند، کارایی کامل سیستم قابل دستیابی نیست.

هنر اشکالزدایی(The Art of Debugging)

اشکالزدایی در نتیجهٔ آزمایش موفق انجام می‌شود. یعنی، هنگامی که نمونه های آزمایش خطایی را کشف می‌کند، اشکالزدایی فرآیندی است که باعث حذف خطا می‌شود. اگرچه اشکالزدایی فرآیندی پوششی است، ولی تا حد زیادی هنری است. مهندس نرم‌افزاری که نتایج آزمایش را ارزیابی می‌کند، اغلب با علایم یک مشکل نرم‌افزاری مواجه می‌شود. یعنی، وضوح خارجی خطا و علت داخلی خطا ممکن است رابطه واضحی با یکدیگر نداشته باشند. فرآیند رفتاری مرتبط کنندهٔ یک علامت ظاهر شده با علت آن که اندکی درک شده باشد، اشکالزدایی نامیده می‌شود.

فرآیند اشکال‌دایی

چرا اشکالزدایی این چنین مشکل است؟ شکل ۹ فرایند اشکالزدایی را نشان می دهد. در تمام احتمالات، پاسخ به این سؤال، بیشتر به روانشناسی انسان مربوط می‌شود تا تکنولوژی نرم‌افزار. به هر حال، چند خصوصیت اشکالات، کلیدهایی را فراهم می‌نمایند:

۱- علامت و علت ممکن است از نظر جغرافیایی از یکدیگر فاصله داشته باشند. یعنی، این علامت ممکن است در یک بخش برنامه ظاهر شود، در حالی که علت آن ممکن است در سایتی قرار داشته باشد که بسیار دور است. ساختارهای برنامه‌ای که با اتصال زیاد مرتبط هستند (فصل ۱۳) این وضعیت را بدتر می‌نمایند.

۲- علامت ممکن است وقتی که خطای دیگری اصلاح می‌گردد، به طور موقت ناپدید شود.

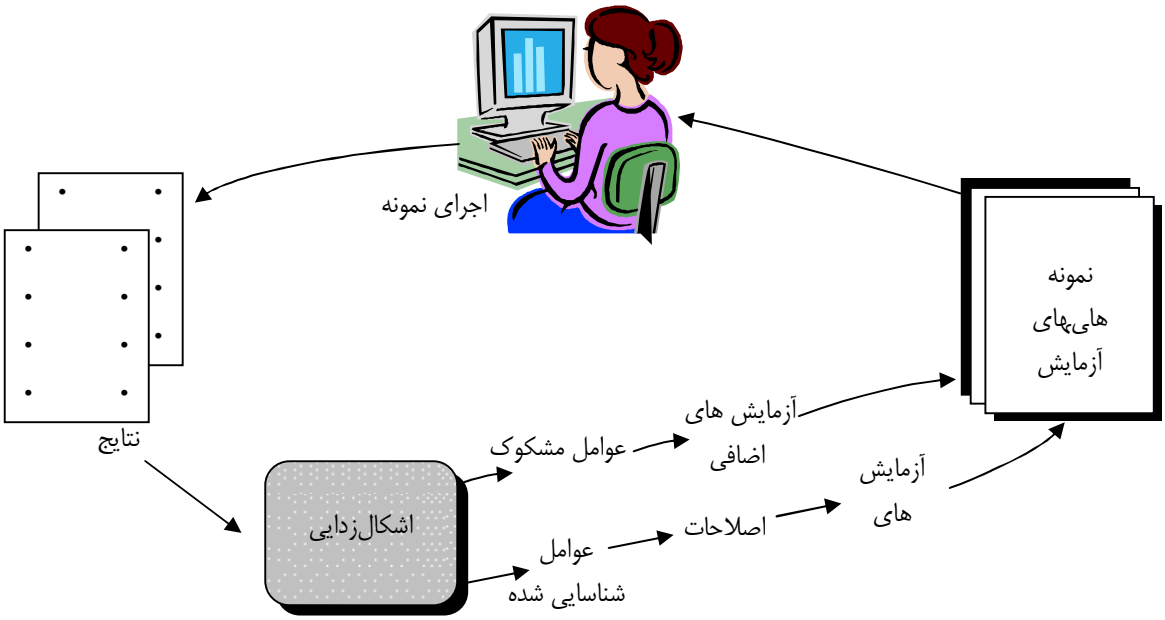
۳- علامت ممکن است توسط هیچ خطایی ایجاد نشده باشد (برای مثال، عدم دقت در نتیجهٔ گردکردن).

۴- علامت ممکن است ناشی از خطای انسانی باشد که به راحتی قابل پیگیری نیست.

۵- علامت ممکن است در نتیجهٔ مشکلات زمانبندی به وجود آید، به جای این که در اثر مشکلات پردازش پدید آمده باشد.

۶- ممکن است ایجاد مجدد شرایط ورودی با دقت، در سیستم‌ها جاسازی شده متداول است زیرا سخت‌افزار و نرم‌افزار کاملاً با یکدیگر متصل شده‌اند.

۷- علامت ممکن است در اثر عللی باشد که در چند task که بر روی پردازنده‌های متفاوت اجرا می‌شوند توزیع شده باشد.



شکل ۹ : ماتریس گراف با در نظر گرفتن ارتباط

شیوه‌های اشکالزدایی

علیرغم شیوه‌ای که به کار گرفته می‌شود، اشکالزدایی یک هدف پوشش دهنده دارد: **یافتن و تصحیح علت خطای نرم‌افزار**. این هدف با ترکیب ارزیابی سیستماتیک، ادارک، و شانس بدست می‌آید. Bradlley شیوهٔ اشکالزدایی را این گونه توصیف می‌کند:

در حالت کلی، سه دسته‌بندی برای روش های اشکالزدایی پیشنهاد می‌شود:

۱- **نیروی مطلق(Brute force)**،
۲- **عقبگرد(Backtracking)**، و
۳- **حذف علت(Cause elimination)**.

روش‌های نیروی مطلق اشکالزدایی احتمالاً متداول‌ترین و کم بازده‌ترین روش برای جدا نمودن علت خطای نرم‌افزار می‌باشد. روش های اشکالزدایی نیروی مطلق زمانی به کار گرفته می‌شوند که همهٔ روش های دیگر با شکست روبرو شده باشند. با فلسفهٔ " یافتن خطا توسط خود کامپیوتر "، محتویات حافظه بر روی صفحه نمایش داده می‌شود. پیگیری‌های زمان اجرا انجام می‌شوند، و احکام write در برنامه قرار داده می‌گیرند. انتظار می‌رود که جایی در اطلاعات تولید شده، کلیدی یافت شود که بتواند باعث هدایت به سمت خطا گردد. اگرچه حجم زیاد اطلاعات تولید شده ممکن است تا حدی به موفقیت منتهی شود، در اکثر موارد باعث اتلاف فعالیت و زمان می‌شود.

عقبگرد روشی نسبتاً متداول است که می‌تواند با موفقیت در برنامه‌های کوچک به کار گرفته شود. با شروع از محلی که علامت در آنجا ظاهر شده، برنامه مبدأ به سمت عقب (به صورت دستی) دنبال می‌شود تا زمانی که محل علت بروز خطا یافت شود. بدبختانه، با افزایش تعداد خطوط کد، تعداد مسیرهای عقبگرد بسیار زیاد خواهد بود.

روش سوم اشکالزدایی، **حذف علت**، با بسط یا حذف انجام می‌شود و مفهوم تقسیم‌بندی دودویی را به همراه دارد. داده‌های مرتب با خطا سازماندهی می‌شوند تا علل بالقوه را جدا نمایند. در یکی از حالات، لیستی از تمام علت‌های ممکن توسعه داده می‌شود و آزمایش‌های مربوطه انجام می‌شوند تا هر یک را حذف کنند. اگر آزمایش‌های اولیه نشان دهند که یک علت خاص، مربوط به علامت ظاهر شده است، داده‌ها پالایش می‌شوند تا خطا حذف شود.

تست‌های فصل ۲۰ : آزمون نرم‌افزار

۱. آخرین مرحله آزمون سطح بالا در چه حیطه‌ای قرار می‌گیرد؟

الف) کدنویسی ب) طراحی ج) خواسته‌ها د) مهندسی سیستم

۲. در حین آزمون واحدها کدام آزمون اهمیت ویژه‌ای دارد؟

الف) آزمون انتخابی(مسیرهای اجرایی کار) ب) آزمون جامعیت ج) آزمون رگرسیون د) آزمون دود

۳. کدام آزمون کوشش می‌کند تا وارسی کند که راهکارهای محافظ تعبیه شده در داخل سیستم واقعاً آن را از نفوذ نامناسب حفظ می‌کند؟

الف) آزمون امنیت ب) آزمون بازیابی ج) آزمون فشار د) آزمون کارایی

۴. کدام آزمون سیستم را به شیوه‌ای اجرا می‌کند که منابع را مقادیر غیر عادی، فراوانی غیر عادی یا حجم غیر عادی طلب کند؟

الف) آزمون کارایی ب) آزمون فشار ج) آزمون امنیت د) آزمون بازیابی

۵. در فرآیند اشکال‌زدایی کدام عمل مورد توجه قرار نمی‌گیرد؟

الف) آزمون‌های اضافی ب) تصحیحات ج) آزمون امنیت د) آزمون رگرسیون

۶. کم بازده‌ترین روش برای از بین بردن علت یک خطای نرم افزاری کدام است؟

الف) روش‌های نیروی مطلق ب) عقب‌گرد ج) حذف علت د) روش ردیاب

۷. بیشترین کار فنی در فرآیند نرم‌افزاری کدام است؟

الف) آزمون نرم‌افزار ب) طرح‌ریزی ج) اجرا د) کنترل

۸. کدام ویژگی باعث ایجاد طرح نرم‌افزار آزمون پذیر می‌شود؟

الف) قابلیت کار ب) قابلیت مشاهده ج) کنترل‌پذیری د) هر سه مورد

۹. آخرین وظیفه در مرحله آزمون واحدها چیست؟

الف) آزمون جامعیت ب) آزمون مرزی ج) آزمون رگرسیون د) آزمون سیستم

۱۰. اعتبارسنجی نرم‌افزار از طریق کدام آزمون انجام می‌پذیرد؟

الف) آزمون آلفا ب) آزمون بتا ج) آزمون جعبه سفید د) آزمون جعبه سیاه

۱۱. کدام آزمون نرم‌افزار را به طریق گوناگون وادار به شکست می‌کند و سپس در مورد اجرای مناسب بازیابی تحقیق می‌کند؟

الف) بازیابی ب) امنیت ج) کارایی د) فشار

۱۲. هدف اصلی اشکال‌زدایی چیست؟

الف) یافتن منبع مشکل از طریق استقراء

ب) یافتن منبع مشکل از طریق شانس قابل دستیابی

ج) یافتن و تصحیح علت یک خطای نرم‌افزاری توسط ترکیبی از سه روش نیروی مطلق، عقبگرد و حذف علت.

د) یافتن و تصحیح علت یک خطای نرم‌افزاری توسط ترکیبی از ارزیابی سیستماتیک و نبوغ و شانس قابل دستیابی است.

منابع

1- Pressman, R., Software Engineerig : A Practitioner's Approach, 5th editition, Mc Graw-Hill, 2005.
2- Bruegge Bernd, Dutoit, H.,Allen; Object Oriented Software Engineering, Prentice Hall, 2000.
3- Rumbaugh, J., Blaha Micheal, Premerlani William, Lorensen William; Object Oriented Moddeling and Design; Prentice Hall, 1991.
4- Kendall & Kendall; System Analysis and Design; 4th edition, Prentice Hall, 1999.
5- Sommerville, Ian; Software Engineering; 5th Edition, Addison-Wesley, 2000.
6- Parrington, N., Marc Roper; Understanding Software Testing; John Wiley & Sons, 1999.
7- Holmes, Jim; Object Oriented Computer Construction, Prentic Hall, 1995.

۸- محمد مهدی سالخورده حقیقی، مهندسی نرم‌افزار، ترجمه ویرایش ۵ مهندسی نرم‌افزار پرسمن، ۱۳۸۲

۹- تحلیل و طراحی سیستم‌ها در مهندسی نرم‌افزار، دکتر پارسا، ۱۳۷۷

۱۰- عین الله جعفر نژاد قمی، مهندسی نرم‌افزار، ترجمه ویرایش ۵ مهندسی نرم‌افزار پرسمن، ۱۳۸۱

۱۱- هاشمی طباء، مهندسی نرم‌افزار، ترجمه ویرایش ۵ مهندسی نرم‌افزار پرسمن، ۱۳۸۲

۱۲- اسلام ناظمی، مسعود زکی پور، امیرفرخ قنبرپور، ترجمه و تنظیم از ویراست های ۴ تا ۶ پرسمن، موسسه پارسه تابستان ۱۳۸۴

ضمیمه

آزمایش سیستم‌های بلادرنگ

ماهیت وابسته به زمان و غیرهمزمان بسیاری از کاربردهای بلادرنگ، عنصری جدید و احتمالاً مشکل را به نام زمان به آزمایش می‌افزاید. طراح نمونه های آزمایش باید نمونه های‌های آزمایش جعبهٔ سفید و جعبهٔ سیاه را همراه، با ادارهٔ واقعه (یعنی پردازش وقفه)، زمانبندی داده‌ها، و موازی بودن taskهای اداره کنندهٔ داده‌ها در نظر داشته باشد. در بسیاری از موارد، داده‌های آزمایشی زمانی فراهم می‌شوند که سیستم بلادرنگ در یک حالت خاص قرار دارد، ممکن است باعث بروز خطا شوند.

برای مثال، نرم‌افزار بلادرنگی که دستگاه کپی جدیدی را کنترل می‌کند، وقفه‌های اوپراتوری را بدون خطا می‌پذیرد (یعنی اوپراتور ماشین کلیدی کنترلی مانند RESET یا DARKEN را می‌زند)، وقتی ماشین در حال گرفتن کپی‌ها است (درحالت " کپی " قرار دارد). همین وقفه‌های اوپراتور، اگر در حالت " جمع شدن کاغذ " وارد شوند، باعث نمایش کد شناسایی می‌شوند تا محل جمع شدن کاغذ را که باید برطرف شود مشخص نمایند (خطا).

علاوه بر آن، رابط نزدیک بین نرم‌افزار بلادرنگ و محیط سخت‌افزاری نیز باعث می‌شود آزمایش با مشکل روبرو شود. آزمایش های نرم‌افزار باید تأثیر خطاهای سخت‌افزار را بر پردازش نرم‌افزار در نظر بگیرند. شبیه‌سازی چنین اشکالاتی ممکن است بسیار مشکل باشد.

روش های طراحی نمونه های‌های طراحی برای سیستم‌های بلادرنگ هنوز در حال تکامل است. به هر حال، یک استراتژی کلی چهار مرحله‌ای پیشنهاد می‌شود:

آزمایش task: اولین مرحله در آزمایش نرم‌افزار بلادرنگ، آزمایش هر task به طور مستقل می‌باشد. یعنی آزمایش های جعبه سفید و جعبه سیاه برای هر task طراحی و اجرا شوند. هر task به طور مستقل در ضمن این آزمایشات اجرا می‌شود. آزمایش task خطاهایی را در منطق و عملکرد آشکار می‌کند، اما خطاهای زمانبندی و رفتاری آشکار نمی‌شوند.

آزمایش رفتاری: با استفاده از مدلهای سیستم‌هایی که با نمونه های‌های CASE ایجاد شده‌اند، امکان شبیه‌سازی رفتار سیستم بلادرنگ و آزمایش رفتار آن در نتیجهٔ وقایع خارجی، وجود دارد. این فعالیت‌های تحلیل، مبنایی را فراهم می‌کنند برای طراحی نمونه های‌های آزمایشی که در زمان ایجاد نرم‌افزار بلادرنگ هدایت می‌شوند. با استفاده از تکنیکی مشابه تقسیم‌بندی مساوی، وقایع (برای مثال، وقفه‌ها، سیگنالهای کنترل) برای آزمایش دسته‌بندی می‌شوند. برای مثال، وقایعی برای دستگاه فتوکپی عبارتند از: وقفه‌های کاربر (برای مثال، کم شدن پودر قرمز)، و مودهای شکست (برای مثال، سربار رولر). هر یک از این وقایع به طور مجزا آزمایش می‌شوند و رفتار سیستم اجرایی آزمایش می‌گردد تا خطاهایی در نتیجهٔ پردازش مربوط به این وقایع، آشکار شوند. رفتار مدل سیستم (که در ضمن فعالیت تحلیل توسعه داده شده) و نرم‌افزار اجرایی برای مسائل کارایی مقایسه می‌شوند. رفتار سیستم آزمایش می‌شود تا خطاهای رفتاری آشکار گردند.

آزمایش بین taskها. پس از جدا شدن خطاهای هر یک از task ها و رفتار سیستم، آزمایش به سمت خطاهای زمانی هدایت می‌شود. task های غیرهمزمان که با یکدیگر ارتباط برقرار می‌کنند، با سرعت انتقال داده‌ها و بار پردازش متفاوت آزمایش می‌شوند تا مشخص کنند آیا خطاهای همزمانی ارتباط بین task ها اتفاق می‌افتند یا خیر. علاوه بر آن، task هایی که با استفاده از صف پیغام یا حافظهٔ داده‌ها ارتباط برقرار می‌نمایند آزمایش می‌گردند تا خطاهای مربوط به اندازهٔ این ناحیه‌های حافظه آشکار شوند.

آزمایش سیستم. نرم‌افزار و سخت‌افزار مجتمع می‌شوند و محدودهٔ کاملی از آزمایش های سیستم هدایت می‌شوند تا خطاهای ارتباط سخت‌افزار- نرم‌افزار آشکار شود. اکثر سیستم‌های بلادرنگ، وقفه‌ها را پردازش می‌کنند. بنابراین، آزمایش ادراهٔ این وقایع بولی ضروری است. با استفاده از نمودار تغییر حالت و مشخصهٔ کنترل (فصل ۱۲)، آزمایش

کننده، لیستی از تمام وقفه‌های ممکن و پردازش‌هایی را که در نتیجهٔ آن وقفه‌ها انجام می‌شوند و توسعه می‌دهد. سپس آزمایش هایی طراحی می‌شوند تا به خصوصیات سیستم که در زیر ارائه شده برسند:

- آیا اولویت‌های وقفه‌ها به طور منظم تخصیص داده شده و به طور منظم اداره می‌شوند؟
- آیا پردازش برای هر وقفه درست اداره می‌شود؟
- آیا کارایی (برای مثال، زمان پردازش) برای هر رویه اداره کنندهٔ وقفه با نیازها مطابقت دارد؟
- آیا حجم زیاد وقفه‌هایی که در زمانهای بحرانی دریافت می‌شوند مشکلی را در عملکرد و کارایی ایجاد می‌کنند؟

علاوه بر آن، ناحیه‌های داده‌های سراسری که برای انتقال اطلاعات به عنوان بخشی از پردازش وقفه استفاده می‌شوند، باید آزمایش شوند تا پتانسیلی را برای تولید اثرات جانبی مشخص کنند.

نکات استراتژیک

در ادامهٔ این فصل، یک استراتژی سیستماتیک برای آزمایش نرم‌افزار ارائه می‌گردد. اما حتی بهترین استراتژی با شکست روبرو می‌شود اگر یک سرسی موارد عمده مورد توجه قرار نگیرند. TomGillb بحث می‌کند که موارد زیر باید مورد توجه قرار گیرند اگر استراتژی آزمایش موفق نرم‌افزار قرار است پیاده‌سازی شود:

- مشخص نمودن نیازهای محصول به روش کمی، مدت طولانی قبل از شروع آزمایش.
- بیان صریح اهداف آزمایش.
- شناسایی خصوصیات کاربران نرم‌افزار و توسعهٔ پروفایلی برای هر دسته‌بندی از کاربران.
- توسعهٔ طرح آزمایشی که بر " دورهٔ سریع آزمایش" تأکید دارد.
- نرم‌افزاری تنومند ایجاد شود که برای آزمایش خودش طراحی شده باشد.
- از مرورهای تکنیکی رسمی مؤثر، به عنوان فیلتر، قبل از آزمایش استفاده شود.
- مرورهای تکنیکی رسمی به گونه‌ای هدایت شوند که به استراتژی آزمایش و خود نمونه های‌های آزمایش دست یابند.
- روشی پیوسته برای ارتقاء فرآیند آزمایش توسعه داده شود.

رویه‌های آزمایش واحد

آزمایش واحد به طور معمول با مرحلهٔ کدنویسی در نظر گرفته می‌شود. پس از توسعهٔ کد مبدأ، مرور آن، و بازبینی آن برای تطابق با طراحی در سطح مؤلفه، طراحی نمونه های آزمایش واحد شروع می‌شود. مرور اطلاعات طراحی، راهنمایی‌هایی را برای ایجاد نمونه های آزمایشی فراهم می‌کند که احتمالاً خطاها را در هر یک از دسته‌بندی‌های بحث شده آشکار می‌نمایند هر نمونه های آزمایش باید با مجموعه‌ای از نتایج مورد انتظار همراه شود.

چون یک مؤلفه، یک برنامهٔ مستقل نمی‌باشد، نرم‌افزار اداره کننده و stub باید برای هر آزمایش واحد توسعه داده شوند. این محیط آزمایش واحد در شکل نشان داده شده است. در اکثر کاربردها، اداره کننده چیزی بیش از یک "برنامه اصلی" نیست که داده‌های نمونه های طراحی را دریافت می‌کند، این داده‌ها را به مؤلفهٔ مورد نظر (که باید آزمایش شود) ارسال می‌کند، و نتایج بدست آمده را چاپ می‌کند stub ها برای جایگزین شدن با پیمانه‌هایی ایجاد می‌شوند که توسط مؤلفهٔ در حال آزمایش فراخوانی می‌شوند. یک stub یا " زیر برنامه ساختگی " با استفاده از رابط پیمانهٔ فراخوانی شده، حداقل دستکاری بر روی داده‌ها را انجام می‌دهد، نتیجهٔ بازبینی ورودی را چاپ می‌کند، و کنترل را به پیمانهٔ در حال آزمایش باز می‌گرداند.