ROZWÓJ OPROGRAMOWANIA I ZAPEWNIENIE JAKOŚCI

WPROWADZENIE.

Rozwój oprogramowania może mieć wpływ na wszystkie sześć podstawowych zasad bezpieczeństwa informacji opisanych w rozdziale 3 niniejszego Podręcznika, ale najczęstsze problemy powodowane przez słabe oprogramowanie obejmują integralność, dostępność i użyteczność. Pomimo dostępności oprogramowania w pakietach na otwartym rynku, oprogramowanie takie często musi być dostosowywane do specyficznych potrzeb użytkowników. Tam, gdzie nie jest to możliwe, programy muszą być tworzone od podstaw. Niestety, podczas każdego projektu rozwoju oprogramowania, pomimo starannego planowania, nieuchronnie pojawiają się nieprzewidziane problemy. Niestandardowe oprogramowanie i niestandardowe pakiety często są dostarczane z opóźnieniem, są wadliwe i nie spełniają specyfikacji. Ogólnie rzecz biorąc, kierownicy projektów oprogramowania mają tendencję do niedoceniania wpływu trudności technicznych i nietechnicznych. Dzięki temu doświadczeniu rozwinęła się dziedzina inżynierii oprogramowania; jego celem jest znalezienie rozsądnych odpowiedzi na pytania, które pojawiają się podczas projektów rozwoju oprogramowania.

CELE ZAPEWNIENIA JAKOŚCI OPROGRAMOWANIA.

W słowniku terminologii inżynierii oprogramowania IEEE jakość definiuje się jako „stopień, w jakim system, komponent lub proces spełnia potrzeby lub oczekiwania klienta lub użytkownika”. Zgodnie z tą definicją, oprogramowanie powinno być mierzone przede wszystkim stopniem do których zaspokajane są potrzeby użytkownika. Ponieważ oprogramowanie często wymaga dostosowania do zmieniających się wymagań, powinno być możliwe do dostosowania przy rozsądnych kosztach. Dlatego oprócz troski o poprawność, niezawodność i użyteczność klient jest również zaniepokojony testowalnością, konserwowalnością, przenośnością i zgodnością z ustalonymi standardami i procedurami dotyczącymi oprogramowania i procesu rozwoju. Zapewnienie jakości oprogramowania (SQA) to element inżynierii oprogramowania, który stara się zapewnić, aby oprogramowanie spełniało akceptowalne standardy kompletności i jakości. SQA pełni funkcję strażnika nadzorującego wszystkie działania związane z jakością związane z tworzeniem oprogramowania. Główne cele SQA to:

* Odkryj wszystkie problemy programu.

* Zmniejsz prawdopodobieństwo, że wadliwe programy wejdą do produkcji.

* Chroń interesy użytkowników.

* Zabezpiecz interesy producenta oprogramowania.

Odkryj wszystkie problemy programu.

Jakość musi być wbudowana w produkt. Testy mogą jedynie ujawnić obecność wad produktu. Aby zapewnić jakość, SQA monitoruje zarówno proces rozwoju, jak i zachowanie oprogramowania. Celem nie jest przekazywanie ani certyfikacja oprogramowania; celem jest zidentyfikowanie wszystkich niedoskonałości i problemów w oprogramowaniu, aby można je było naprawić.

Zmniejsz prawdopodobieństwo, że wadliwe programy wejdą do produkcji.

Całe tworzenie oprogramowania musi być zgodne ze standardami i zasadami ustalonymi w organizacji, aby identyfikować i eliminować błędy, zanim wadliwe programy wejdą do produkcji.

Ochrona interesów użytkowników.

Ostatecznym celem jest uzyskanie oprogramowania spełniającego wymagania użytkowników i zapewniającego im potrzebną funkcjonalność. Aby osiągnąć te cele, SQA musi dokonać przeglądu i audytu procesu tworzenia oprogramowania oraz dostarczyć wyniki tych przeglądów i audytów kierownictwu. SQA, jako działający dział, może odnieść sukces tylko wtedy, gdy ma wsparcie kierownictwa i podlega kierownictwu na tym samym poziomie, co tworzenie oprogramowania. Podobnie każdy projekt SQA będzie posiadał określone atrybuty, a program SQA powinien być dostosowany do potrzeb projektu. Cechy, które należy wziąć pod uwagę, to: krytyczność misji, harmonogram i budżet, wielkość i złożoność projektu oraz wielkość i kompetencje organizacji personelu projektowego.

Ochrona interesów producentów oprogramowania.

Zapewniając, że oprogramowanie spełnia wymagania, SQA może pomóc w zapobieganiu konfliktom prawnym, które mogą powstać, jeśli zakupione oprogramowanie nie spełnia zobowiązań umownych. Gdy oprogramowanie jest opracowywane we własnym zakresie, SQA może zapobiec wskazywaniu palcem, które w przeciwnym razie zaszkodziłoby relacjom między programistami a użytkownikami korporacyjnymi.

CYKL ROZWOJU OPROGRAMOWANIA.

Dobre oprogramowanie, niezależnie od tego, czy zostało opracowane we własnym zakresie, czy kupione od zewnętrznego dostawcy, musi być budowane zgodnie z solidnymi zasadami. Ponieważ projekty programistyczne są często duże i wiele osób pracuje nad nimi przez długi czas, proces tworzenia oprogramowania musi być monitorowany i kontrolowany. Chociaż postęp takich projektów jest trudny do zmierzenia, zastosowanie podejścia etapowego może kontrolować projekty. W podejściu etapowym między początkiem a zakończeniem projektu ustala się pewną liczbę jasno określonych kamieni milowych. Powszechnie używa się analogii do budowy domu, w której fundamenty są kładzione na początku, a każda faza budowy jest osiągana w uporządkowany i kontrolowany sposób. Często zarówno w przypadku budowy domu, jak i rozwoju oprogramowania, płatności za etapy zależą od osiągnięcia wyznaczonych kamieni milowych. Opracowując systemy, określamy etapowy proces jako cykl życia rozwoju systemu (SDLC). SDLC to proces tworzenia systemów informatycznych poprzez badanie, analizę, projektowanie, kodowanie i debugowanie, testowanie, wdrażanie i konserwację. Te siedem faz jest wspólnych, chociaż różne modele i techniki mogą zawierać więcej lub mniej faz. Ogólnie rzecz biorąc, kamienie milowe zidentyfikowane w SDLC odpowiadają momentom, w których określone dokumenty stają się dostępne. Często dokumenty wyjaśniają i wzmacniają działania podjęte podczas właśnie zakończonej fazy SDLC. Dlatego mówimy, że tradycyjne modele stopniowego rozwoju są w dużej mierze oparte na dokumentach. Istnieją problemy i wady nieodłącznie związane z procesem opartym na dokumentach. Sposób przeglądania procesu rozwoju za pomocą SDLC nie jest całkowicie realistyczny w stosunku do rzeczywistych projektów. W rzeczywistości odnotowuje się błędy znalezione we wcześniejszych fazach, opracowywane są poprawki, wprowadzane jest prototypowanie i wdrażane są rozwiązania. W efekcie jest to więcej niż zakłada się, że będzie konieczne w fazie debugowania i konserwacji. Ponadto często problemy są rozwiązywane przed osiągnięciem tych późniejszych faz. Ze względu na uznanie, że wiele z tego, co w tradycyjnych modelach określa się mianem utrzymania, jest w rzeczywistości ewolucją, opracowano inne modele SDLC, znane jako modele ewolucyjne. W tradycyjnych modelach początkowy rozwój systemu jest ściśle oddzielony od fazy konserwacji. Głównym celem SDLC jest dostarczenie użytkownikowi pierwszej wersji produkcyjnej systemu oprogramowania. Nie jest niczym niezwykłym, że takie podejście skutkuje nadmiernymi kosztami utrzymania w celu dostosowania funkcjonującego lub końcowego systemu do potrzeb rzeczywistego użytkownika. Dostępne są inne modele i do każdego projektu należy wybrać konkretny model SDLC. Aby to zrobić, konieczne jest zidentyfikowanie potrzeby nowego systemu i

podążanie za tym poprzez określenie poszczególnych kroków i faz, możliwej interakcji faz, niezbędnych rezultatów i wszystkich powiązanych materiałów. Istnieje pięć głównych technik rozwoju systemu:

1. Cykl życia systemów tradycyjnych (SDLC)

2. Model wodospadu (wariant tradycyjnego SDLC)

3. Szybki rozwój aplikacji (RAD)

4. Wspólny rozwój aplikacji (JAD)

5. Ekstremalne programowanie

Chociaż te pięć technik rozwoju często postrzegane są jako wzajemnie wykluczające się, w rzeczywistości reprezentują one rozwiązania, które w różny sposób kładą nacisk na wspólne elementy projektowania systemów. Zdefiniowane w różnym czasie mocne strony każdej metodologii pokazują technologię, ekonomię i problemy organizacyjne, które były aktualne w czasie, gdy metodologia została zdefiniowana po raz pierwszy. Oprogramowanie na ogół jest konstruowane w fazach, a testy powinny być przeprowadzane na końcu każdej fazy, zanim rozwój będzie kontynuowany w następnej fazie. Cztery główne fazy, które zawsze są obecne w tworzeniu oprogramowania, to:

1. Faza analizy, w której określane są wymagania programowe

2. Faza projektowania, na podstawie wcześniej opisanych wymagań

3. Faza konstrukcji, programowania lub kodowania

4. Faza wdrożenia, w której oprogramowanie jest faktycznie instalowane na sprzęcie produkcyjnym i ostatecznie testowane przed wydaniem na produkcję

Faza programowania zawsze obejmuje testowanie jednostkowe poszczególnych modułów oraz testowanie systemowe wszystkich funkcji. Czasami, oprócz testowania podczas programowania i wdrażania, ustalana jest piąta faza, poświęcona wyłącznie testom funkcjonalnym. Przegląd i modyfikacja zwykle następują po wszystkich fazach, w których naprawiane są słabości i niedociągnięcia.

Fazy tradycyjnego cyklu życia oprogramowania.

Omawianych tutaj siedem faz obejmuje najczęściej spotykany tradycyjny SDLC:

1. Dochodzenie

2. Analiza

3. Projekt

4. Dekodowanie i debugowanie

5. Testowanie

6. Wdrożenie

7. Konserwacja

Z tego tradycyjnego modelu opracowano inne modele z mniejszą lub większą liczbą faz.

Dochodzenie.

Ta faza obejmuje określenie zapotrzebowania na system. Polega na ustaleniu, czy istnieje problem lub szansa biznesowa oraz przeprowadzeniu studium wykonalności w celu określenia opłacalności proponowanego rozwiązania.

Analiza.

Analiza wymagań to proces analizy potrzeb informacyjnych użytkowników końcowych, otoczenia organizacji i obecnego systemu oraz opracowania wymagań funkcjonalnych dla systemu w celu zaspokojenia potrzeb użytkowników. Ta faza obejmuje zarejestrowanie wszystkich wymagań. Dokumentacja musi być stale wykorzystywana podczas pozostałej części procesu rozwoju systemu.

Projekt.

Faza projektowania architektonicznego zawiera listę i opis wszystkich niezbędnych specyfikacji sprzętu, oprogramowania, zasobów ludzkich i danych oraz produktów informacyjnych, które spełnią wymagania funkcjonalne proponowanego systemu. Projekt najlepiej można opisać jako plan systemu. Jest to kluczowe narzędzie w wykrywaniu i eliminowaniu problemów i błędów, zanim zostaną one wbudowane w system.

Kodowanie i debugowanie.

Ta faza obejmuje faktyczne stworzenie systemu. Dokonują tego specjaliści IT, czasem pracownicy firmy, a czasem firmy zewnętrzne specjalizujące się w tego typu pracy. System jest kodowany i podejmowane są próby wyłapania i wyeliminowania wszystkich błędów kodowania przed wdrożeniem systemu.

Testowanie.

Po utworzeniu systemu niezbędne jest testowanie. Testowanie potwierdza funkcjonalność i niezawodność systemu oraz stanowi kolejny kamień milowy w znajdowaniu problemów i błędów przed wdrożeniem. Ta faza ma kluczowe znaczenie dla określenia, czy system spełni potrzeby użytkowników. Testy powinny być udokumentowane w formacie planu projektu, co do:

* Cel testów

* Zakres

* Oczekiwane rezultaty

* Plan testów

* Przypadki testowe; najlepiej, aby użytkownicy końcowi i/lub właściciele procesu biznesowego byli zaangażowani, aby przypadki testowe w jak największym stopniu odzwierciedlały rzeczywiste sytuacje

* Wyniki badań i analiza

* Zakończenie i określenie dalszych kroków

* Użytkownik końcowy/właściciel procesu biznesowego [podpisał] akceptację wyników testu

Te kroki pomagają zapewnić, że testy są zgodne z potrzebami użytkowników, a wyniki są reprezentatywne dla tego, czego można się spodziewać.

Wdrożenie.

Po zakończeniu i zaakceptowaniu poprzednich pięciu etapów, wraz z dokumentacją uzasadniającą, system jest wdrażany i użytkownicy mogą z niego korzystać.

Konserwacja.

Ta faza trwa i ma miejsce po wdrożeniu systemu. Ponieważ systemy podlegają rozbieżnościom, wadom i awariom, a także trudnościom podczas integracji z innymi systemami i sprzętem, konserwacja trwa tak długo, jak system jest używany.

Klasyczny model wodospadu.

Chociaż etapowe podejście do tworzenia oprogramowania pojawiło się w latach 60., model kaskadowy, przypisywany Royowi Royce'owi, pojawił się w latach 70. XX wieku. Model kaskadowy wymaga sekwencyjnego podejścia do tworzenia oprogramowania i zawiera tylko pięć faz:

1. Faza analizy wymagań

2. Projekt

3. Wdrożenie

4. Testowanie

5. Konserwacja

Analiza lub Analiza wymagań.

Model kaskadowy kładzie nacisk na analizę jako część fazy analizy wymagań. Ponieważ oprogramowanie jest zawsze częścią większego systemu, wymagania są najpierw ustalane dla wszystkich elementów systemu, a następnie pewien podzbiór tych wymagań jest przydzielany do oprogramowania. Zidentyfikowane wymagania, zarówno dla systemu, jak i oprogramowania, są następnie dokumentowane w specyfikacji wymagań. Konieczna jest seria testów walidacyjnych, aby upewnić się, że w miarę rozwoju systemu nadal spełnia on te specyfikacje. Model kaskadowy obejmuje walidację i weryfikację w każdej z pięciu faz.

Oznacza to, że na każdym etapie procesu wytwarzania oprogramowania konieczne jest porównanie uzyskanych wyników z wymaganymi. Testowanie odbywa się w każdej fazie, aby odpowiedzieć na to pytanie i nie występuje wyłącznie w fazie testowania, która następuje po fazie wdrożenia.

Projekt

Faza projektowania jest w rzeczywistości procesem wieloetapowym, skupiającym się na strukturze danych, architekturze oprogramowania, szczegółach proceduralnych i charakterystyce interfejsu. W fazie projektowania wymagania są przekładane na reprezentację oprogramowania, które można następnie ocenić pod kątem jakości przed rozpoczęciem faktycznego kodowania. Po raz kolejny dokumentacja odgrywa ważną rolę, ponieważ udokumentowany projekt staje się częścią konfiguracji oprogramowania. Kodowanie jest włączane w fazę projektowania zamiast stanowić oddzielną fazę. Podczas kodowania projekt jest tłumaczony na formę do odczytu maszynowego.

Wdrożenie

W fazie wdrożenia system zostaje przekazany użytkownikowi. Wielu programistów uważa, że dużym problemem z tym modelem jest to, że system jest wdrażany, zanim faktycznie będzie gotowy do przekazania użytkownikowi. Jednak zwolennicy modelu kaskadowego twierdzą, że konsekwentne

testowanie w całym procesie rozwoju pozwala systemowi być gotowym do czasu osiągnięcia tej fazy. Zwróć uwagę, że w modelu kaskadowym faza implementacji poprzedza fazę testowania.

Testowanie

Chociaż testowanie trwało przez cały proces tworzenia, faza testowania rozpoczyna się po wygenerowaniu kodu i wystąpieniu fazy implementacji. Ta faza koncentruje się zarówno na logicznych elementach wewnętrznych oprogramowania, jak i funkcjonalnych zewnętrznych. Najlepiej byłoby, gdyby do pomocy na tym etapie zaangażowani byli użytkownicy końcowi i właściciele procesów/firm, ponieważ zwykle:

* Znają wymagania biznesowe

* Dowiedz się, jak działa obecne oprogramowanie i sprzęt

* Dowiedz się, jakie rodzaje aktywności są przewidywane i planowane

* Zapewnij ostateczną akceptację oprogramowania i sprzętu

Chociaż nie jest to bezpośrednio wymagane przez wiele przeglądów regulacyjnych, zaangażowanie użytkowników końcowych i właścicieli procesów zapewnia obszarowi IT dodatkowe zasoby, nie tylko z perspektywy zasobów ludzkich, ale także wiedzy biznesowej i regulacyjnej perspektywy zgodności.

Konserwacja

Faza konserwacji ponownie stosuje każdy z poprzednich etapów cyklu życia do istniejących programów. Konserwacja jest konieczna z powodu wykrytych błędów, niezbędnej adaptacji do środowiska zewnętrznego oraz wymaganych i wymaganych przez klienta ulepszeń funkcjonalnych i wydajnościowych. Model kaskadowy jest uważany za niewiarygodny, częściowo z powodu nieprzestrzegania ścisłej kolejności faz zalecanej przez model tradycyjny. Ponadto model kaskadowy często zawodzi, ponieważ dostarcza produkty, których użyteczność jest ograniczona, gdy wymagania uległy zmianie w procesie rozwoju.

Szybkie tworzenie aplikacji i wspólne projektowanie aplikacji.

Rapid Application Development (RAD) wspiera iterację i elastyczność niezbędną do budowania solidnego wsparcia procesów biznesowych. RAD kładzie nacisk na zaangażowanie użytkowników i małe zespoły programistyczne, prototypowanie oprogramowania, ponowne wykorzystanie oprogramowania i zautomatyzowane narzędzia. Czynności muszą być wykonywane w określonych ramach czasowych, określanych mianem skrzynki czasowej. To podejście różni się od innych modeli rozwoju, w których wymagania są ustalane jako pierwsze, a ramy czasowe są ustalane później. RAD, starając się dotrzymać terminu i nienaruszalnego terminu, może poświęcić część funkcjonalności. RAD ma cztery fazy w swoim cyklu:

1. Planowanie wymagań

2. Projekt użytkownika

3. Budowa

4. Przecięcie

Główne techniki stosowane w RAD to wspólne planowanie wymagań (JRP) i wspólne projektowanie aplikacji (JAD). Słowo „wspólny” odnosi się do programistów i użytkowników pracujących razem

podczas intensywnego korzystania z warsztatów. JAD umożliwia identyfikację, definicję i wdrażanie infrastruktur informacyjnych. Technika JAD jest omawiana z RAD, ponieważ wzmacnia RAD.

Programowanie ekstremalne

Formą zwinnego tworzenia oprogramowania, która zyskała popularność w ciągu ostatnich dwóch dekad, jest programowanie ekstremalne (XP). Programiści ściśle współpracują ze sobą i użytkownikami, aby szybko tworzyć użyteczny kod. Testy jednostkowe i zintegrowane testy integracyjne są konstruowane przed kodowaniem i stale stosowane w całym procesie rozwoju. Wariant tej metody jest znany jako Scrum.

Znaczenie integracji bezpieczeństwa na każdym etapie

Bezpieczeństwo nigdy nie powinno być czymś dodanym do oprogramowania na końcu projektu. Bezpieczeństwo musi być brane pod uwagę w sposób ciągły podczas całego projektu, aby chronić zarówno oprogramowanie, jak i cały system, w którym działa. Niezależnie od tego, który model tworzenia oprogramowania jest używany, ważne jest, aby zintegrować zabezpieczenia w każdej fazie rozwoju i uwzględniać zabezpieczenia w testowaniu na każdym etapie. Zgrubne szacunki wskazują, że koszt naprawy błędu wzrasta dziesięciokrotnie z każdą dodatkową fazą. Na przykład wyłapanie błędu w fazie analizy może wymagać tylko kilku minut — czasu na poprawienie analityka przez użytkownika — i dlatego może kosztować tylko kilka dolarów. Wyłapanie tego samego błędu po włączeniu go do specyfikacji może kosztować 10 razy więcej; po wdrożeniu nawet 1000 razy więcej (wyklucza to wszelkie kary, straty lub odpowiedzialność za straty związane ze zgodnością).

RODZAJE BŁĘDÓW OPROGRAMOWANIA

Wewnętrzne błędy projektowe lub implementacyjne

Ogólna definicja błędu oprogramowania to niezgodność między programem a jego specyfikacją; bardziej szczegółową definicją jest „niepowodzenie programu w wykonaniu tego, czego rozsądnie oczekuje użytkownik końcowy”. Istnieje wiele rodzajów błędów oprogramowania. Niektóre z najważniejszych to:

* Inicjalizacja

* Przepływ logiczny

* Obliczenia

* Naruszenia warunków brzegowych

* Przekazywanie parametrów

* Warunki wyścigu

* Stan obciążenia

* Wyczerpanie zasobów

* Konflikt zasobów, adresu lub programu z systemem operacyjnym lub aplikacjami

* Względy dotyczące zgodności z przepisami

* Inne błędy

Inicjalizacja

Błędy inicjalizacji są podstępne i trudne do znalezienia. Najbardziej podstępne programy zapisują informacje inicjujące na dysku i kończą się niepowodzeniem tylko przy pierwszym użyciu — to znaczy przed utworzeniem pliku inicjującego. Za drugim razem dany użytkownik aktywuje program, nie ma dalszych błędów inicjalizacji. W związku z tym błędy pojawiają się tylko dla pracowników i klientów, gdy aktywują nową kopię lub instalację wadliwego programu. Inne programy z błędami inicjalizacji mogą wykazywać dziwne obliczenia lub inne wady przy pierwszym użyciu lub inicjalizacji; ponieważ nie przechowują swoich wartości inicjalizacji, te błędy inicjalizacji będą pojawiać się ponownie za każdym razem, gdy program jest używany.

Przepływ logiczny

Moduły przekazują kontrolę sobie nawzajem lub innym programom. Jeśli wykonanie przejdzie do niewłaściwego modułu, wystąpił błąd przepływu logicznego. Przykłady obejmują wywołanie niewłaściwej funkcji lub rozgałęzienie do podprogramu, który nie zawiera instrukcji RETURN, tak że wykonanie przechodzi przez logiczny koniec modułu i rozpoczyna wykonywanie innego modułu kodu.

Obliczanie

Gdy program błędnie interpretuje skomplikowane formuły i traci precyzję podczas obliczeń, prawdopodobnie wystąpił błąd obliczeniowy; na przykład wartość pośrednia może być przechowywana w tablicy z 16 bitami precyzji, gdy potrzebuje 32 bitów. Ta kategoria błędów obejmuje również błędy obliczeniowe spowodowane nieprawidłowymi algorytmami.

Naruszenia warunków granicznych

Termin „granice” odnosi się do największych i najmniejszych wartości, z jakimi program może sobie poradzić; na przykład tablica może być zwymiarowana z 365 wartościami, aby uwzględnić dni w roku, a następnie zawieść w roku przestępnym, gdy program zwiększy licznik dni do 366 i tym samym spróbuje zapisać wartość pod nieprawidłowym adresem. Programy, które ustawiają zmienne zakresy i alokację pamięci, mogą działać w granicach, ale jeśli są nieprawidłowo zaprojektowane, mogą ulec awarii w granicach lub poza nimi. Pierwsze użycie programu również można uznać za warunek brzegowy.

Przekazywanie parametrów

Czasami występują błędy w przekazywaniu danych tam iz powrotem między modułami. Na przykład wywołanie funkcji może zostać przypadkowo przekazana zła nazwa zmiennej, aby funkcja działała na niewłaściwych wartościach. W przypadku wystąpienia tych błędów przekazywania parametrów dane mogą zostać uszkodzone, a ścieżka wykonania może zostać zmieniona z powodu nieprawidłowych wyników obliczeń lub porównań. W rezultacie ostatnie zmiany danych mogą zostać utracone lub wykonanie może wpaść w procedury obsługi błędów, nawet jeśli zamierzone dane były poprawne.

Warunki wyścigu

Gdy między zdarzeniem A a zdarzeniem B dochodzi do wyścigu, do prawidłowego działania wymagana jest określona sekwencja zdarzeń, ale ta sekwencja nie jest zapewniana przez program. Na przykład, jeśli proces A blokuje zasób 1 i czeka na odblokowanie zasobu 2, podczas gdy proces B blokuje zasób 2 i czeka na odblokowanie zasobu 1, nastąpi śmiertelne uścisk, który zawiesza operacje. W systemach wieloprocesorowych i systemach interaktywnych można się spodziewać warunków wyścigowych, ale mogą one być trudne do odtworzenia; na przykład opisany właśnie śmiertelny uścisk może zdarzyć się tylko raz na 1000 transakcji, jeśli średni czas transakcji jest krótki. W związku z tym warunki wyścigu należą do najmniej przetestowanych.

Stan obciążenia

Wszystkie programy i systemy mają ograniczenia dotyczące pojemności pamięci, liczby użytkowników, transakcji i przepustowości. Błędy obciążenia mogą wystąpić z powodu dużej objętości, która obejmuje dużą ilość pracy przez długi czas, lub dużego obciążenia, które obejmuje maksymalne obciążenie jednocześnie.

Wyczerpanie zasobów

Wyczerpywanie się w programie szybkiej pamięci (pamięć o dostępie swobodnym lub RAM), pamięć masowa (dysk), cykle jednostki centralnej (CPU), wpisy w tabeli systemu operacyjnego, semafory lub inne zasoby mogą spowodować awarię programu. Na przykład niewystarczająca pamięć główna może powodować zamianę danych na dysk, zwykle powodując drastyczne zmniejszenie przepustowości.

Konflikty międzyaplikacyjne

W przypadku tak złożonych systemów operacyjnych (OS) producenci systemów operacyjnych rutynowo dystrybuują wymagania dotyczące kodu i określone parametry do producentów oprogramowania aplikacji, aby zminimalizować prawdopodobieństwo konfliktów programów lub nieoczekiwanych przestojów. Chociaż z pewnością pomaga to zmniejszyć liczbę problemów i poprawia zgodność w przód i wstecz z poprzednimi wersjami systemu operacyjnego, czasami nawet dostawcy systemów operacyjnych doświadczają lub powodują trudności, gdy nie są zgodne z parametrami ustalonymi dla ich własnych programów.

Inne źródła błędów

Często zdarza się, że programy wysyłają złe dane do urządzeń, ignorują powracające kody błędów, a nawet próbują używać urządzeń, które są zajęte lub których brakuje. Sprzęt może być zepsuty, ale oprogramowanie jest również uważane za błędne, gdy nie odzyskuje sprawności po takich warunkach sprzętowych. Dodatkowe błędy mogą wystąpić przez niewłaściwe kompilacje pliku wykonywalnego; na przykład, jeśli stara wersja modułu jest połączona z najnowszą wersją reszty programu, mogą pojawić się nieprawidłowe ekrany logowania, mogą zostać wyświetlone nieprawidłowe komunikaty o prawach autorskich, mogą pojawić się nieprawidłowe numery wersji i różne mogą wystąpić inne niedokładności.

Rozważania dotyczące zgodności z przepisami.

Jeśli organizacja podlega Sarbanes-Oxley (SOX), wówczas dokumentacja napotkanych błędów i wynikających z nich wewnętrznych działań sprawozdawczych i naprawczych ma kluczowe znaczenie. Błąd powinien wyraźnie wskazać, w jaki sposób została zidentyfikowana, kto ją zidentyfikował, w jaki sposób została zgłoszona kierownictwu, jakie będą środki zaradcze i kiedy przewiduje się ich zakończenie. Bez tych danych kierownictwo może napotkać znaczne trudności w potwierdzeniu, że istnieją odpowiednie mechanizmy kontroli wewnętrznej i że jest odpowiednio i odpowiednio poinformowany i zaangażowany. Ponadto błąd może skutkować zidentyfikowaniem przez audytorów zewnętrznych słabości kontroli lub, co gorsza, jako słabości istotnej, w zależności od jej charakteru i dotkliwości. Ponadto, jeśli i kiedy zostaną zauważone błędy, które mają wpływ na systemy lub aplikacje wielolokacyjne, znaczenie i istotność należy rozpatrywać z punktu widzenia zarówno lokalizacji oddziału, jak i siedziby głównej. Ponieważ przepisy różnią się w poszczególnych stanach i krajach, to, co może być uznane za prawnie dopuszczalne standardy prywatności lub rachunkowości w jednym miejscu, może nie być akceptowalne w innym miejscu.

Interfejs użytkownika

Mówiąc ogólnie, termin „interfejs użytkownika” oznacza wszystkie aspekty systemu, które są istotne dla użytkownika. Można go ogólnie określić jako wirtualną maszynę użytkownika (UVM). Obejmuje to wszystkie ekrany, mysz i klawiaturę, wydruki i wszystkie inne elementy, z którymi użytkownik wchodzi w interakcję. Poważny problem pojawia się wtedy, gdy projektanci systemów nie mogą postawić się na miejscu użytkownika i nie mogą przewidzieć problemów, jakie napotka użytkownik niesprawny technologicznie z interfejsem zaprojektowanym przez osobę znającą się na technologii. Dokumentacja jest kluczową częścią każdego systemu. Każda faza rozwoju – wymagania, analiza, rozwój, kodowanie, testowanie, błędy, rozwiązania błędów i modyfikacje, implementacja i utrzymanie – musi być udokumentowana. Wszystkie dokumenty i ich różne wersje muszą być zachowane zarówno do przyszłego wykorzystania, jak i do celów audytu. Dodatkowo ważne jest udokumentowanie prawidłowego użytkowania systemu oraz dostarczenie użytkownikowi odpowiednich materiałów instruktażowych i referencyjnych. Polityki bezpieczeństwa oraz związane z nimi egzekwowanie i kary również muszą być udokumentowane. Najlepiej byłoby, gdyby dokumentacja umożliwiała każdej osobie wykwalifikowanej technicznie naprawę lub modyfikację dowolnego elementu, o ile system działa.

Funkcjonalność

W programie występuje błąd funkcjonalności, jeśli wydajność, której można racjonalnie oczekiwać, jest zagmatwana, niezręczna, trudna lub niemożliwa. Błędy funkcjonalne często dotyczą kluczowych cech lub funkcji, które nigdy nie zostały zaimplementowane. Dodatkowe błędy funkcjonalności występują, gdy:

* Funkcje nie są udokumentowane.

* Brak wymaganych informacji.

* Program nie potwierdza prawidłowych danych wejściowych.

* Występują błędy rzeczowe lub sprzeczne nazwy funkcji.

* Występuje przeciążenie informacjami.

*Materiał jest napisany na nieodpowiednim poziomie czytania.

* Kursor znika lub znajduje się w niewłaściwym miejscu.

* Wyświetlacze na ekranie są nieprawidłowe.

* Instrukcje są niejasne.

* Identyczne funkcje wymagają różnych operacji na różnych ekranach.

* Istnieją niepoprawnie sformatowane ekrany wprowadzania.

* Hasła lub inne poufne informacje nie są odpowiednio zasłaniane ani chronione.

* Śledzenie wpisu lub zmian danych użytkownika jest niedostępne lub niekompletne.

* Segregacja obowiązków nie jest egzekwowana. (Może to być szczególnie istotne w przypadku organizacji podlegających ustawie o przenośności i odpowiedzialności w ubezpieczeniach zdrowotnych [HIPAA], SOX, Międzynarodowej Organizacji Normalizacyjnej [ISO] 17799 i/lub ustawie Gramm-Leach-Bliley [GLBA].)

Struktura sterowania (polecenia)

Błędy struktury kontrolnej mogą powodować poważne problemy, ponieważ mogą skutkować:

* Użytkownicy gubiący się w programie

* Użytkownicy marnują czas, ponieważ muszą radzić sobie z mylącymi poleceniami

* Utrata danych lub niepożądane ujawnienie danych

* Opóźnienie pracy

* Koszty finansowe

* Nieoczekiwane narażenie na wyciek danych lub narażenie na niebezpieczeństwo; może to skutkować znaczną odpowiedzialnością, jeśli dane osobowe konsumentów (PII) zostaną naruszone

* Dane nie są szyfrowane zgodnie z przeznaczeniem lub nie są widoczne dla nieautoryzowanych użytkowników

Niektóre typowe błędy to:

* Brak możliwości poruszania się między menu

* Mylące i powtarzające się menu

* Niedopuszczenie odpowiednich wpisów w wierszu poleceń

* Wymaganie wpisów wiersza poleceń, które nie są ani intuicyjne, ani wyraźnie zdefiniowane na ekranie

* Niezgodność aplikacji z konwencjami systemu operacyjnego

* Brak rozróżnienia między plikami źródłowymi a parametrami, skutkujący udostępnieniem użytkownikowi błędnych wartości za pośrednictwem interfejsu lub nieidentyfikacją źródła błędu

* Niewłaściwe korzystanie z klawiatury, gdy nowe programy nie spełniają standardów klawiatury, która ma oznaczone klawisze funkcyjne powiązane ze standardowym znaczeniem

* Brakujące polecenia w kodzie i ekranach, co powoduje, że użytkownik nie może uzyskać dostępu do informacji, korzystać z programów lub zapewnić tworzenia kopii zapasowej i możliwości odzyskania systemu, a także wielu innych poleceń, które pozostawiają system w stan nieoptymalnej operacyjności

* Nieodpowiednia prywatność lub bezpieczeństwo, które mogą skutkować ujawnieniem poufnych informacji, całkowitą zmianą lub utratą danych bez możliwości odzyskania, złym raportowaniem, a nawet niepożądanym dostępem osób z zewnątrz.

Wydajność

Szybkość jest ważna w interaktywnym oprogramowaniu. Jeśli użytkownik uważa, że program działa wolno, może to być natychmiastowy problem. Powolne działanie może zależeć (ale nie wyłącznie) od systemu operacyjnego, innych uruchomionych aplikacji, alokacji pamięci, wycieku pamięci i konfliktów programów. Na innym poziomie wydajność spada, gdy projekty programów utrudniają zmianę ich funkcjonalności w odpowiedzi na zmieniające się wymagania. Błędy wydajności obejmują powolną reakcję, niezapowiedzianą czułość na wielkość liter, niekontrolowane i nadmiernie częste automatyczne zapisywanie, niemożność zapisania i ograniczoną prędkość przewijania.

Format wyjściowy

Błędy formatu wyjściowego mogą być frustrujące i czasochłonne. Uważa się, że wystąpił błąd, gdy użytkownik nie może zmienić czcionek, podkreśleń, pogrubienia i odstępów, które wpływają na ostateczny wygląd danych wyjściowych; alternatywnie mogą wystąpić opóźnienia lub błędy podczas drukowania lub zapisywania dokumentu. Błędy pojawiają się, gdy użytkownik nie może kontrolować zawartości, skalowania i wyglądu tabel, rysunków i wykresów. Dodatkowo występują błędy wyjściowe, które wiążą się z wyrażeniem danych z nieodpowiednim poziomem precyzji.

PROJEKTOWANIE PRZYPADKÓW TESTOWYCH OPROGRAMOWANIA

Dobre testy

Żaden program nigdy nie może być całkowicie przetestowany, ponieważ nie byłoby opłacalne ani wydajne testowanie poprawności, parametrów, składni i granic każdego wiersza kodu. Nawet jeśli wszystkie prawidłowe dane wejściowe są zdefiniowane i przetestowane, nie ma możliwości przetestowania wszystkich nieprawidłowych danych wejściowych i wszystkich odmian synchronizacji danych wejściowych. Trudne, jeśli nie niemożliwe, jest również przetestowanie każdej ścieżki, jaką może obrać program, znalezienie każdego błędu projektowego i udowodnienie, że programy są logicznie poprawne. Niemniej jednak, dobra procedura testowa odnajdzie większość problemów, które mogą się pojawić, pozwalając projektantom i programistom naprawić te problemy i zapewnić prawidłowe działanie oprogramowania. Ogólnie rzecz biorąc, producenci systemów operacyjnych i aplikacji klasyfikują poziom ważności napotkanych błędów, a wielu (o czym świadczy częstotliwość i zawartość poprawek do programów) koryguje tylko te najpoważniejsze, które zauważają oni sami lub użytkownicy. W ciągu ostatnich kilku lat nasiliła się debata na temat tego, czy luki w oprogramowaniu powinny być natychmiast publikowane przez badaczy, czy też badacze powinni automatycznie powiadamiać i dać producentowi czas na naprawienie usterki. Ogólnie rzecz biorąc, z pobieżnego przeglądu luk w zabezpieczeniach Internet Explorera w wersji 6 firmy Microsoft w ciągu ostatnich kilku lat wynika, że trend wskazuje na to, że badacz powiadamia firmę Microsoft i pozwala firmie Microsoft na tworzenie i rozpowszechnianie łatki w różnym czasie (np. trzy do sześciu miesięcy) zanim badacz publicznie ogłosi słabość. Trend ten opiera się na publikowanych doniesieniach w mediach technicznych. Ilekroć oczekujesz takich samych wyników z dwóch testów, uważasz testy za równoważne. Grupa testów tworzy klasę równoważności, jeśli tester uważa, że wszystkie testy testują to samo, a jeśli jeden z testów wykryje lub nie wykryje błędu, inne prawdopodobnie zrobią to samo. Klasyczne testy graniczne sprawdzają reakcję programu na dane wejściowe i wyjściowe; Testy równoważności uczą jednak sposobu myślenia o analizowaniu programów, który usprawnia i wzmacnia planowanie testów. Znalezienie klas równoważności jest procesem subiektywnym Różne osoby analizujące ten sam program wymyślą różne listy klas równoważności ze względu na to, co programy wydają się osiągać. Przypadki testowe są często umieszczane w tej samej klasie równoważności, gdy dotyczą tych samych zmiennych wejściowych, skutkują podobnymi operacjami, wpływają na te same zmienne wyjściowe lub obsługują błędy w ten sam sposób. Klasy równoważności mogą być grupami testów zajmujących się zakresami lub wielokrotnymi zakresami liczb, członkami grupy, a nawet określonym czasem. Klasy równoważności to zazwyczaj najbardziej ekstremalne wartości, takie jak największa, najmniejsza, najszybsza i najwolniejsza. Podczas testowania ważne jest, aby przetestować każdą krawędź klasy równoważności ze wszystkich stron każdej krawędzi. Testerzy powinni używać tylko jednego lub dwóch przypadków testowych z każdej klasy równoważności, ponieważ program, który pomyślnie przejdzie testy, ogólnie zda każdy test wylosowany z tej klasy. Nieprawidłowe klasy równoważności danych wejściowych często umożliwiają przeoczenie błędów programu podczas debugowania.

Podkreśl warunki brzegowe

Granice są kluczowe dla sprawdzania odpowiedzi każdego programu na dane wejściowe i wyjściowe. Granice klas równoważności są najbardziej skrajnymi wartościami klasy; na przykład warunki brzegowe mogą składać się z największych i najmniejszych, najwcześniejszych i najpóźniejszych, najkrótszych i najdłuższych lub najwolniejszych i najszybszych członków klasy równoważności. Testy powinny zawierać wartości poniżej, przy i powyżej wartości granicznych. Dodatkowo powinny być testowane na różnych poziomach użytkownika (np. administrator, root lub superużytkownik, wprowadzanie danych i dostęp tylko do odczytu). Testy te muszą zapewnić, że na żadnym poziomie nie ma warunków, które umożliwiłyby niezamierzony dostęp lub nieuprawnioną eskalację uprawnień. Gdy programy zawodzą z wartościami niegranicznymi, zazwyczaj zawodzą również na granicach. Programy, które przejdą te testy, prawdopodobnie zdadzą również każdy inny test narysowany z tej klasy. Testy powinny również umożliwiać generowanie największych i najmniejszych uzasadnionych wartości wyjściowych, pamiętając, że wejściowe wartości graniczne mogą nie generować wartości wyjściowych granicznych. Obecnie najczęściej występującymi naruszeniami bezpieczeństwa są przepełnienia bufora w aktywnym kodzie (ActiveX i Java), w których nieodpowiednie sprawdzanie granic ciągów wejściowych pozwala na interpretację przepełnionego tekstu jako kodu, który może następnie wykonać niewłaściwe operacje. Ograniczenia dotyczące długości i typu danych wejściowych zapobiegłyby tym exploitom. Takie przekroczenia mogą skutkować narażeniem danych osobowych konsumentów i pociągnięciem do niechcianej odpowiedzialności za narażenie; obecnie odpowiedzialność istnieje głównie na poziomie państwa i może obejmować sankcje karne, w zależności od państwa zamieszkania konsumentów.

Zaznacz wszystkie przejścia stanów

Wszystkie programy interaktywne przechodzą z jednego stanu do drugiego. Stan programu jest zmieniany za każdym razem, gdy coś powoduje, że program zmienia wyjście (np. wyświetla coś innego na ekranie) lub zmienia zakres dostępnych wyborów (np. wyświetlanie nowego menu opcji użytkownika). Aby przetestować przejścia stanów, projektant testów powinien stworzyć macierz prawdopodobieństwa przejścia, aby pokazać wszystkie ścieżki, którymi ludzie mogą podążać. Przejścia między stanami mogą być złożone i często zależą nie od prostych wyborów, ale od wprowadzanych przez użytkownika liczb. Testerzy często uważają za przydatne tworzenie map menu, które pokazują dokładnie, gdzie należy się udać z każdego dostępnego wyboru. Mapy menu mogą również pokazywać, kiedy i gdzie użytkownicy przechodzą, gdy polecenia menu lub klawiatury przenoszą ich do różnych stanów lub okien dialogowych. Mapy są szczególnie przydatne podczas pracy z kodem spaghetti (kodem, który został źle zaprojektowany lub źle utrzymany, przez co trudno jest dostrzec logiczne relacje); mapy umożliwiają projektantowi lub użytkownikowi dotarcie do okna dialogowego na kilka sposobów, a następnie przejście z okna dialogowego do kilku miejsc. W przypadku kodu spaghetti mapy menu zapewniają prostszą metodę wykrywania relacji między stanami niż próby pracy wyłącznie z samego programu, ponieważ mapa pokazuje przejścia między stanami na papierze lub na ekranie. Po zmapowaniu relacji projektant lub użytkownik może sprawdzić poprawność programu z mapą. Pełne testowanie teoretycznie wymaga przetestowania wszystkich możliwych ścieżek. Jednak w praktyce pełne testowanie może być nieosiągalne dla złożonych programów. Dlatego warto najpierw przetestować najczęściej używane ścieżki.

Testuj każdy limit

Konieczne jest przetestowanie każdego limitu zachowania programu określonego przez dowolny dokument programu. Limity obejmują rozmiar plików, liczbę terminali, rozmiar pamięci, jaką program może zarządzać, maksymalny rozmiar, którego wymaga, oraz liczbę drukarek, które program może obsługiwać. Ważne jest, aby sprawdzić zdolność programu do obsługi dużej liczby otwartych plików zarówno w trybie natychmiastowym, jak i długoterminowym, w trybie ciągłym. Testowanie obciążenia jest w rzeczywistości testowaniem warunków brzegowych i powinno obejmować uruchamianie

testów, które program powinien przejść, oraz testów, których program nie powinien przejść. W przypadku aplikacji internetowych istnieją narzędzia upraszczające to zadanie, ponieważ niektóre sprawdzają granice i limity pól na różnych głębokościach wprowadzania danych, aby przetestować zabezpieczenia przed przepełnieniem bufora, nieprawidłowymi danymi i poprawną składnią danych; takie narzędzia obejmują między innymi WebInspect firmy Spi Dynamics.

Test warunków wyścigu

Po przetestowaniu systemu pod „normalnym" obciążeniem, należy go przetestować pod kątem warunków wyścigowych. „Stan wyścigu" jest zwykle definiowany jako zachowanie anomalne z powodu nieoczekiwanej krytycznej zależności od względnego czasu wydarzeń. Warunki wyścigu zazwyczaj obejmują jeden lub więcej procesów uzyskujących dostęp do współdzielonego zasobu, takiego jak plik lub zmienna, gdzie ten wielokrotny dostęp nie był odpowiednio kontrolowany. Systemy podatne na wyścigi, zwłaszcza systemy wielożytkownikowe z równoczesnym dostępem do zasobów, powinny przejść pełny cykl testów pod obciążeniem. Czasami warunki wyścigu można zidentyfikować poprzez testowanie przy dużym obciążeniu, małym obciążeniu, dużej i małej szybkości, wieloprocesorach działających współbieżnie, ulepszonych i liczniejszych urządzeniach wejścia/wyjścia, częstych przerwaniach, mniejszej ilości pamięci, wolniejszej pamięci i powiązanych zmiennych.

Użyj monitorów pokrycia testowego

W przypadku złożonych programów testowanie ścieżek zwykle nie może lub nie może przetestować każdej możliwej ścieżki w programie ze względów praktycznych oraz ze względów ekonomicznych. Bardziej praktycznym podejściem ze szklanego pudełka (tj. ze znajomością wewnętrznego projektu programu) jest użycie listy kodu źródłowego do wymuszenia przejścia programu do każdej gałęzi widocznej w kodzie. Gdy programiści dodają specjalny kod debugowania podczas tworzenia, unikalny komunikat zostanie wydrukowany lub dodany do pliku dziennika, gdy program osiągnie określony punkt. Zazwyczaj takie komunikaty mogą być generowane przez wywołanie procedury drukowania z unikalnym parametrem dla każdego bloku kodu. Gdy kod źródłowy jest kompilowany, wiele języków pozwala na ustawienie przełącznika umożliwiającego kompilację warunkową, tak aby testowa wersja kodu zawierała aktywne instrukcje debugowania, podczas gdy wersja produkcyjna nie. W przypadku kodu interpretowanego, takiego jak większość języków czwartej generacji, podobne przełączniki umożliwiają włączenie lub aktywację instrukcji debugowania. Ponieważ umieszczane są różne komunikaty, można dokładnie wiedzieć, jaki punkt programu osiągnął test. Programiści specjalnie wstawiają te komunikaty w znaczących częściach programu. Po dodaniu tych komunikatów każdy może uruchomić program i stwierdzić, czy poszczególne części rzeczywiście zostały uruchomione i przetestowane. Można również użyć specjalnych urządzeń lub narzędzi do automatycznego dodawania tych wiadomości do kodu. Po wprowadzeniu kodu źródłowego do takiego monitora pokrycia, analizuje on struktury kontrolne w kodzie i dodaje sondy dla każdej gałęzi programu. Dodanie tych sond lub wierszy kodu nazywa się instrumentacją programu. Można stwierdzić, że program został wciśnięty w każdą gałąź, gdy wszystkie sondy zostały wydrukowane. Oprócz kodu oprzyrządowania dobry monitor zasięgu może przechwytywać dane wyjściowe sondy, przeprowadzać analizy i podsumowywać je. Niektóre monitory zasięgu rejestrują również wykorzystany czas dla każdej procedury, wspierając w ten sposób analizę i optymalizację wydajności. Monitor pokrycia jest przeznaczony do testowania szklanych pudełek, ale znajomość elementów wewnętrznych testowanego programu nie jest potrzebna do korzystania z monitora. Monitor zasięgu zlicza komunikaty sondujące, raporty o liczbie wyzwolonych sond, a nawet raporty dotyczące dokładności testowania. Ponieważ monitor zasięgu może raportować o gałęziach, które nie zostały uruchomione, monitor może znaleźć i zidentyfikować kod, który nie należy, na przykład procedury włączone domyślnie, ale nigdy nie używane lub celowo wstawiane nieudokumentowany kod (koń trojański). Niektóre dostępne na rynku monitory zasięgu są

niestety same w sobie pełne błędów. Ważne jest, aby uzyskać od dewelopera pełną listę znanych błędów i poprawek. Ponadto takie narzędzia same powinny zostać przetestowane z programami zawierającymi znane błędy, w procesie zwanym seedingiem, w celu weryfikacji ich poprawności.

Seeding

Same procedury zapewniania jakości czerpią korzyści z testowania. Jedna z wydajnych metod, znana jako seeding, polega na dodawaniu znanych błędów do programu i mierzeniu, ile z nich zostało wykrytych w normalnych procedurach testowych. Wskaźnik sukcesu w identyfikacji takich znanych błędów może pomóc oszacować odsetek nieznanych błędów pozostawionych w programie. Takie rozsiewanie jest szczególnie ważne przy ustanawianiu automatycznych procedur testowych i monitorów pokrycia testami; jest również przydatny do testów SOX lub innych testów zgodności.

Budowanie zestawów danych testowych.

Jednym z najpoważniejszych błędów w zapewnianiu jakości jest umożliwienie programistom używania zestawów danych produkcyjnych do testowania. Dane produkcyjne mogą zawierać informacje poufne, które nie powinny być dostępne dla personelu programistycznego, dlatego należy zabronić dostępu nie tylko do danych produkcyjnych, ale nawet do kopii danych produkcyjnych, a nawet kopii podzbiorów lub próbek z danych produkcyjnych. Jednak procesy anonimizacji stosowane do kopii danych produkcyjnych (np. szyfrowanie nazwisk i adresów pacjentów w bazie danych dokumentacji medycznej) mogą generować zestawy danych testowych nadające się do wykorzystania w zapewnianiu jakości. Alternatywnie, jeśli organizacja posiada system testowy, który jest całkowicie (logicznie i fizycznie) oddzielony od systemu produkcyjnego, zastosowanie danych historycznych może być właściwe. Należy przy tym zachować szczególną ostrożność i odpowiednią dokumentację, aby zapewnić, że nie ma szans na połączenie systemu testowego z systemem produkcyjnym. Dodatkową korzyścią takiego testowania jest to, że może pomóc w przeprowadzaniu przez organizację okresowych testów i dokumentacji SOX lub HIPAA dotyczących dostępności, użyteczności i integralności nośników i systemów kopii zapasowych.

Fuzzing

Użytecznym dodatkiem do technik opisanych powyżej jest automatyczne wstrzykiwanie losowych danych do strumienia wejściowego. Fuzzing może wspierać planowane wyzwania dla systemu z nieoczekiwanymi danymi wejściowymi, o których programiści mogli nie pomyśleć.

PRZED ROZPOCZĘCIEM PRODUKCJI

Testy regresji

Testy regresyjne to podstawowa praca wykonywana przez testerów szklanych i czarnych skrzynek. Termin ten jest używany na dwa różne sposoby. Pierwsza definicja obejmuje te testy, w których błąd został znaleziony i naprawiony, a test, który ujawnił problem, jest wykonywany ponownie. Druga definicja polega na znalezieniu i naprawieniu błędu, a następnie wykonaniu standardowej serii testów, aby upewnić się, że wprowadzone zmiany lub poprawki nie zakłóciły niczego innego.

Pierwszy rodzaj testowania regresji służy do sprawdzenia, czy poprawka działa zgodnie z jej przeznaczeniem. Drugi typ testuje poprawkę, a także testuje ogólną integralność programu. Nie jest niczym niezwykłym, że ludzie, którzy wspominają o testowaniu regresji, odnoszą się do obu definicji, ponieważ obie dotyczą poprawiania, a następnie ponownego testowania. Zaleca się, aby obydwa rodzaje testów były wykonywane po naprawieniu błędów. Jeśli organizacja podlega SOX lub HIPAA,

należy zachować odpowiednią dokumentację, która szczegółowo opisuje metodologię badania, dobór próbki, częstotliwość, wyniki i środki zaradcze (jeśli takie istnieją).

Testowanie automatyczne

W niektórych przedsiębiorstwach za każdym razem, gdy usuwany jest błąd, za każdym razem, gdy program jest modyfikowany i za każdym razem, gdy powstaje nowa wersja, uruchamiany jest test regresji. Wszystkie te testy zajmują dużo czasu i pochłaniają zarówno zasoby ludzkie, jak i maszyny. Ponadto, powtarzalna praca może stać się paraliżująca, więc testerzy mogą przypadkowo pominąć przypadki testowe lub przeoczyć błędne wyniki. Aby zmniejszyć czasochłonność personelu i powtarzalność zadania, można zaprogramować komputer do przeprowadzania testów akceptacyjnych i regresyjnych. Ten rodzaj automatyzacji skutkuje wykonaniem testów, zebraniem wyników, porównaniem wyników ze znanymi dobrymi wynikami oraz przekazaniem wyników testerowi. Wczesna automatyzacja testów składała się zasadniczo z makr klawiaturowych lub skryptów o ograniczonej zdolności reagowania na błędy; zazwyczaj błąd powodowałby nieprawidłowe wyniki, które zostałyby wprowadzone do następnego etapu przetwarzania i prowadziłyby do długich łańcuchów bezsensownych wyników. Nieco bardziej zaawansowane wiązki testowe zatrzymywałyby proces testowy przy każdym błędzie i wymagałyby interwencji człowieka, aby wznowić. Obie metody były jedynie skromną pomocą dla zespołu testowego. Jednak dzisiejsze oprogramowanie do automatyzacji testów może zawierać testowe bazy danych, które umożliwiają uporządkowaną konfigurację testów, szczegółowe instrukcje ponownego uruchamiania sekwencji testowych po błędach oraz pełną dokumentację danych wejściowych i wyników. Aby uzyskać realistyczne testowanie obciążenia, niektóre systemy można skonfigurować tak, aby symulowały użytkowników na stacjach roboczych za pomocą skryptów, które definiują zakres losowo generowanych wartości, określone limity parametrów, a nawet zmienne czasy odpowiedzi. Testy obciążenia na dużą skalę mogą łączyć komputery za pośrednictwem sieci w celu symulacji tysięcy równoczesnych użytkowników, a tym samym przyspieszyć identyfikację wyczerpania zasobów lub warunków wyścigu. Automatyzacja testów może być warta wymaganych nakładów. Testowanie automatyczne jest zwykle bardziej precyzyjne, pełniejsze, szybsze i tańsze niż testy wykonywane przez personel ludzki. Zazwyczaj testy automatyczne mogą osiągnąć dziesięciokrotny lub stukrotny wzrost liczby testów osiągalnych metodami ręcznymi. Oprócz uwolnienia personelu do wykonywania bardziej satysfakcjonującej pracy, takie metody znacznie zmniejszają ogólne koszty konserwacji dzięki wykrywaniu błędów, zanim systemy wejdą do produkcji. Dodatkowo, ze względów prawnych, automatyczne testowanie wymaga znacznie mniej próbkowania, dokumentacji i testowania. Nadal wymagany jest odpowiedni przegląd zarządzania i raportowanie wyników, podobnie jak dokumentacja działań następczych po naprawie.

Śledzenie błędów od odkrycia do usunięcia.

Znalezienie błędu nie wystarczy. Nawet wyeliminowanie błędu jest niewystarczające. System musi również umożliwiać zidentyfikowanie, udokumentowanie i naprawienie przyczyn każdego błędu. Z tych powodów ważne jest, aby śledzić i dokumentować wszystkie problemy od momentu ich wykrycia do ich usunięcia, aby mieć pewność, że problemy zostały rozwiązane i nie wpłyną na system. Zapewnia to również ważne zapisy historyczne w diagnozowaniu i naprawianiu incydentów oraz dostarcza odpowiednią dokumentację do celów audytowych i regulacyjnych. Systemy śledzenia problemów muszą być używane do zgłaszania błędów, śledzenia rozwiązań i pisania raportów podsumowujących na ich temat. Zorganizowany system jest niezbędny, aby zapewnić odpowiedzialność i komunikację dotyczącą błędów. Typowe raporty obejmują: skąd pochodzą błędy (np. jacy programiści i które zespoły są odpowiedzialne za ich największą liczbę); rodzaje napotkanych problemów (np. błędy typograficzne, błędy logiczne, naruszenia granic); i czas na naprawę. Jednak systemy śledzenia problemów mogą podnosić kwestie polityczne, takie jak odpowiedzialność za projekt, monitorowanie

osobiste, kwestie kontroli i naprawa problemów związanych z danymi w bazie danych i ich właścicielem. Po ustanowieniu systemu śledzenia, raport o błędzie jest wprowadzany do systemu bazy danych, a kopia trafia do kierownika projektu, który albo nadaje mu priorytet i przekazuje go dalej, albo odpowiada na niego osobiście. W końcu programiści zostaną wciągnięci w pętlę, naprawią problem i oznaczą go jako naprawiony. Rozwiązany problem jest następnie testowany i wystawiany jest raport o stanie. Jeśli poprawka okaże się nieskuteczna, staje się to nowym problemem, który należy rozwiązać.

ZARZĄDZANIE ZMIANĄ

Wiele osób może być zaangażowanych w tworzenie i zarządzanie systemami; za każdym razem, gdy wprowadzane są zmiany w systemie, należy nimi zarządzać i monitorować je w zorganizowany sposób, aby uniknąć chaosu.

Wniosek o zmianę.

Żądanie zmiany jest ważnym dokumentem, który żąda albo poprawki, albo innej zmiany w programie lub systemie. Informacje zawarte w formularzu wniosku o zmianę zazwyczaj obejmują osobę, która wnioskuje o zmianę, datę złożenia wniosku, program i obszar, którego dotyczy, datę, do której zmiana jest potrzebna, oraz upoważnienie do kontynuowania i wprowadzenia zmiany. Większość firm posiada formularze papierowe, które są przechowywane do celów monitorowania i audytu. Ponieważ popularność podpisów cyfrowych rośnie, a społeczeństwo nadal zmierza w kierunku biura bez papieru, wydaje się logiczne, że niektóre z tych żądań zmian w końcu zostaną całkowicie zautomatyzowane.

System śledzenia

System śledzenia może być ręczny, zautomatyzowany lub może być kombinacją metod; do celów regulacyjnych i kontrolnych najlepszy jest system, który obejmuje automatyczne śledzenie, ponieważ zmniejsza ryzyko błędu i może usprawnić działania następcze i naprawcze. Niezależnie od tego, w jaki sposób jest przechowywany, oryginalny formularz zmiany jest zazwyczaj rejestrowany w systemie, albo składany ręcznie, albo wprowadzany do zautomatyzowanej bazy danych przez pewien aspekt identyfikujący, taki jak data, rodzaj zmiany lub dział wnioskujący. System służy do śledzenia, co dzieje się ze zmianą. W momencie wykonania akcji do systemu należy wprowadzić informację, kto pracował nad zleceniem, co zostało zrobione, kiedy akcja została podjęta, jaki był jej wynik, kto został powiadomiony o podjętych działaniach, czy zmiana została dokonana akceptowane i powiązane informacje. Informacje te mają kluczowe znaczenie dla zapewnienia odpowiedniego zarządzania problemami, eskalacji ich w razie potrzeby do wyższej kadry kierowniczej i śledzenia do momentu ich naprawienia.

Testowanie regresji

Po wprowadzeniu zmiany lub poprawki testy regresji sprawdzają, czy zmiana rzeczywiście została dokonana i czy program działa teraz w pożądany sposób, w tym wszystkie inne funkcje. Pomyślne zakończenie testów powinno być udokumentowane na piśmie; może to nastąpić za pośrednictwem systemu śledzenia lub w dokumentacji dołączonej do wpisu systemu śledzenia.

Dokumentacja

Dokumentacja jest kluczowa podczas rozważania, zatwierdzania i wdrażania zmian. Jeśli informacje są przechowywane wyłącznie w głowie jednostki, co się dzieje, gdy dana osoba wyjeżdża na wakacje, jest chora lub opuszcza firmę na stałe? Co jeśli osoba po prostu nie pamięta, jakie zmiany zostały wprowadzone lub kiedy; skąd organizacja wie, kto był zaangażowany, co właściwie zostało zrobione i

kto potwierdził, że zmiany zostały wprowadzone i zaakceptowane? Nieudokumentowane zmiany mogą skutkować:

*Awaria systemu

* Niespójne wprowadzanie danych

* Niewłaściwy podział obowiązków

*Kradzież lub korupcja danych

* Defraudacja

* Inne poważne przestępstwa

Brak dokumentacji może oznaczać, że dokonano nieautoryzowanych zmian i prawdopodobnie może skutkować naruszeniami przepisów i audytów. Nieudokumentowane zmiany mogą naruszać podział obowiązków (np. dana osoba może nieformalnie zatwierdzać zmiany na swoim koncie, zatwierdzać zmiany przeprowadzane przez przełożonego lub autoryzować zmiany przez osobę, która nie podlega zatwierdzającemu). Brak dokumentacji jest często naruszeniem polityki firmy i jest często przytaczany w audytach; często powoduje to znaczne trudności w zakresie działań i przeglądów zgodności SOX, GLBA i HIPAA, ponieważ twierdzenia kierownictwa mogą nie być odpowiednio poparte. Ponieważ zbudowanie odpowiedniej dokumentacji po fakcie jest prawie niemożliwe (i często można je uznać za fałszowanie zapisów w sektorach takich jak usługi finansowe), dokumentacja musi przebiegać krok po kroku z każdym etapem rozwoju i funkcjonowania systemu. Następnie dokumentacja musi być przechowywana zgodnie z polityką firmy i wymogami prawnymi.

ŹRÓDŁA BŁĘDÓW I PROBLEMÓW

Znajdowanie i naprawianie błędów to ważne zadanie, ale ważne jest również, aby spróbować określić, gdzie i dlaczego powstały błędy i związane z nimi problemy. Odnajdując i badając źródło, często można zapobiec powtórzeniu się tego samego lub podobnego rodzaju problemu.

Wady konstrukcyjne.

Wady projektowe często pojawiają się z powodu słabej komunikacji między użytkownikami a projektantami. Użytkownicy nie zawsze jasno wiedzą, czego chcą i potrzebują, podczas gdy projektanci źle rozumieją, źle interpretują lub po prostu ignorują to, co użytkownicy do nich odnoszą. W procesie projektowania, nawet jeśli uczucia i wymagania użytkowników są znane i rozumiane, mogą wystąpić wady projektowe. Jednak łatwiej je zidentyfikować i naprawić, jeśli naruszona zostanie odpowiednia dokumentacja. Bez odpowiedniej dokumentacji może nie być możliwe zidentyfikowanie źródła błędu; może to skutkować poprawkami lub środkami, które należy załatać. Często wady wynikają z prób przestrzegania nierealistycznych harmonogramów dostaw. Menedżerowie powinni wspierać swoich pracowników w opieraniu się presji pośpiechu na każdym z etapów projektowania, rozwoju i wdrażania.

Błędy implementacji

Zawsze, gdy program jest opracowywany i wdrażany, istnieją ograniczenia czasowe i terminy realizacji. Większość problemów podczas tworzenia oprogramowania powoduje opóźnienia, więc programiści i testerzy często są pospiesznie wykonywani. Czasami poświęcają dokumentację, przeglądową część projektu, a nawet skracają testy, pozostawiając nierozpoznane wady projektowe i implementacyjne. Menedżerowie powinni podkreślać wartość dokładnego przeglądu i testowania oraz poświęcić wystarczająco dużo czasu, aby uniknąć takich błędów.

Nieautoryzowane zmiany w kodzie produkcyjnym

Jeśli problemy wynikają z nieautoryzowanych zmian, kierownicy projektów powinni przeanalizować swoje zasady. Jeśli zasady są jasne, być może szkolenia pracowników i programy uświadamiające wymagają poprawy. Menedżerowie powinni również przeanalizować swoje zachowanie, aby upewnić się, że nie wywierają takiej presji na swoich pracowników, że pójście na skróty jest postrzegane jako dopuszczalne.

Niewystarczająca lub niespełniająca norm jakość programowania

Programista może stworzyć lub złamać program i projekt. Programiści odgrywają zasadniczą rolę w tworzeniu oprogramowania. Dlatego ważne jest, aby wszyscy programiści biorący udział w projekcie byli zdolni i niezawodni. Programiści projektowi powinni zostać dokładnie sprawdzeni przed umieszczeniem ich w projekcie rozwoju oprogramowania, aby upewnić się, że ich umiejętności spełniają wymagania projektu. Czasami luki w umiejętnościach programistów można wypełnić odpowiednim szkoleniem i coachingiem. Jeśli jednak problemy wydają się spójne, kierownictwo może chcieć sprawdzić doświadczenie programisty, aby sprawdzić, czy nie fałszował on informacji podczas ubiegania się o pracę. Naprawdę niekompetentni programiści muszą zostać usunięci z projektu. Dodatkowo, jeśli kierownictwo podejrzewa, że jakość programowania jest niewystarczająca, powinno rozważyć zlecenie niezależnego programisty przeglądu kodu, wykonującego szczególnie rygorystyczne testy zapewniania jakości (QA).

Uszkodzenie danych

Uszkodzenie danych może wystąpić z powodu złego programowania, nieprawidłowego wprowadzania danych, nieodpowiedniego blokowania podczas równoczesnego dostępu do danych i modyfikacji, nielegalnego dostępu jednego procesu do innego stosu danych procesu oraz awarii sprzętu. Uszkodzenie danych może wystąpić nawet wtedy, gdy program jest automatycznie testowany lub uruchamiany bez interwencji człowieka. W każdym razie, szukając źródeł błędów i innych problemów, ważne jest, aby po każdej rundzie testów dokładnie przejrzeć dane, aby zidentyfikować odchylenia od prawidłowego stanu końcowego. Przegląd powinien być odpowiednio udokumentowany i przekazany kierownictwu; następnie wesprze wysiłki w zakresie zgodności z przepisami.

Hakowanie

Kiedy pojawiają się błędy i problemy, hakerstwo - zarówno wewnętrzne, jak i zewnętrzne – powinno być brane pod uwagę i poszukiwane poprzez przeglądanie dzienników, kto pracował nad oprogramowaniem i kiedy ta praca została wykonana. Menedżerowie powinni być w stanie wykryć użycie legalnych identyfikatorów, gdy zaangażowane osoby były na wakacjach, były chore lub nie były dostępne do zalogowania się z innych powodów. Archiwizacja dzienników i ich odzyskiwanie jest coraz bardziej krytyczną funkcją, poza wymogami regulacyjnymi SOX, HIPAA, GLBA i tak dalej. Dzienniki są często używane do:

* Działania związane z reagowaniem na incydenty, w szczególności przy ustalaniu:

* Jeśli doszło do incydentu

* Kiedy doszło do incydentu

* Co wydarzyło się przed i po podejrzeniu wystąpienia incydentu

* Badania; może to być przegląd możliwości wydajności lub problemów ze sprzętem, oprogramowaniem lub siecią

Aby zapewnić dostępność dzienników w razie potrzeby, organizacja potrzebuje solidnej strategii, która okresowo przechowuje i archiwizuje dzienniki, tak aby osoba atakująca lub anomalia systemowa nie wymazała ich z dysku twardego systemu. Chociaż istnieje wiele sposobów, aby to osiągnąć, często stosowana niedroga metoda polega na ustanowieniu serwera dziennika. Na przykład organizacja może zidentyfikować zużyty komputer stacjonarny, wyposażyć go w nowy, szybki napęd optyczny o dużej pojemności tylko do odczytu (ROM) i zwiększyć pojemność pamięci (w razie potrzeby). Dzięki częstemu kierowaniu dzienników do napędu optycznego i nagrywaniu ich w ciągu dnia (może kilka razy w ciągu godziny) oraz wymianie dysku optycznego co najmniej raz dziennie, organizacja stworzy archiwum kryminalistyczne, które będzie można łatwo powielić w celach badawczych i organach ścigania. lub legalny użytek. Często ilość dzienników dziennych nie zapełni płyty DVD lub CD. Jednak regularna rutyna codziennego zmieniania mediów pomaga zapewnić, że media nie osiągną swoich możliwości w nieodpowiednim czasie – w rzeczywistości, jeśli robak, taki jak Blaster lub Nimda, zainfekuje organizację, logi prawdopodobnie szybko się pęcznieją. Ponadto, jeśli atakujący w systemie przerwie rejestrowanie, organizacja powinna mieć mniejsze trudności z określeniem daty, godziny (lub przedziału czasowego) i zakresu włamania, ponieważ pamięci ROM nie można nadpisać, w przeciwieństwie do nośników do odczytu i zapisu (RW). . Czasami można zidentyfikować nieautoryzowane zmiany w kodzie i danych, nawet jeśli sprawca zatrzyma rejestrowanie i usunie lub zmodyfikuje pliki dziennika. Jedną z metod jest tworzenie sum kontrolnych dla produkcji lub innych oficjalnych wersji oprogramowania oraz ochrona sum kontrolnych przed nieautoryzowanym dostępem i modyfikacją za pomocą szyfrowania i podpisów cyfrowych. Podobnie nieautoryzowane modyfikacje danych mogą być czasami utrudnione przez tworzenie sum kontrolnych dla rekordów i łączenie ich ze znacznikami czasu i danych oraz z sumami kontrolnymi dla autoryzowanych programów. W takich warunkach nieautoryzowany personel i intruzi mają trudności z utworzeniem prawidłowych sum kontrolnych dla zmodyfikowanych rekordów. Inne produkty, takie jak systemy wykrywania włamań (IDS) lub systemy zapobiegania włamaniom (IPS), mogą być przydatne w strategii ochrony i zarządzania ryzykiem organizacji.

WNIOSEK

Przedstawiono kompleksowy przegląd procesów tworzenia oprogramowania i zapewniania jakości, które należy stosować podczas opracowywania, wdrażania i modyfikowania oprogramowania. Tworzenie oprogramowania obejmuje coś więcej niż tylko wybór i wykorzystanie podejścia, takiego jak metodologia tradycyjna, kaskadowa lub RAD. Oznacza to pracę zespołową w celu opracowania, przeglądu, udoskonalenia i wdrożenia opłacalnego działającego produktu. Wiele dobrych technik i produktów można zastosować zarówno do części SDLC, jak i zapewniania jakości oprogramowania; niektóre z kluczowych elementów to dobra dokumentacja, zapewniająca wystarczającą ilość czasu na testowanie w procesach rozwoju i utrzymania, budowanie dobrych testów, ustalanie danych testowych, automatyzacja testowania i śledzenie żądań zmian.