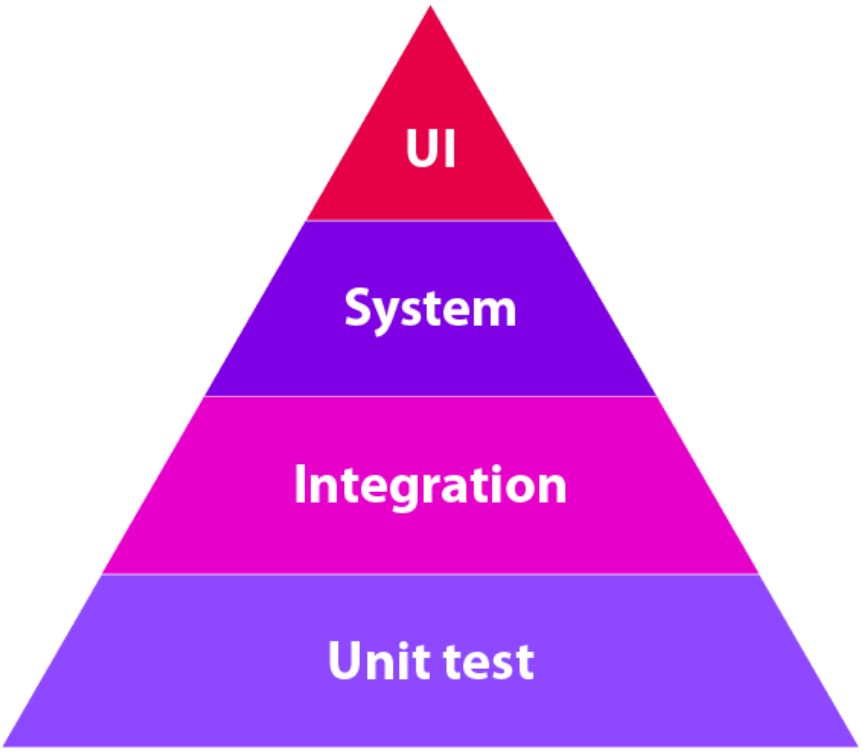
Typy testů a jejich použití

Rozdělení testů:

- **Podle fáze vývoje**
  - Jednotkové testy (Unit Testing)
  - Integrační testy (Integration Testing)
  - Systémové testy (System Testing)
  - Akceptační testy (Acceptance Testing)
- **Podle znalosti kódu**
  - Black box
  - White box
- **Podle způsobu realizace testů**
  - Manuální testování (Manual Testing)
  - Automatizované testování (Automated Testing)
- **Podle dimenzí kvality**
  - Funkční testy (Functional Testing)
  - Výkonové testy (Performance Testing)
  - Testování použitelnosti (Usability Testing)
  - Testování spolehlivosti (Reliability Testing)
  - Bezpečnostní testy (Security Testing)
  - Testování udržovatelnosti (Maintainability Testing)
  - Testování přenositelnosti (Portability Testing)
  - Localization Testing
- **Podle rozsahu**
  - Regresní testy (Regression Testing)
  - Sanity testy (Sanity Testing)
  - Smoke testy (Smoke Testing)
  - Explorativní testy (Exploratory Testing)
  - End-to-End (E2E) testování
- **Risk based testing**

Podle fáze vývoje



Jednotkové testy (Unit Testing)

Testování jednotlivých částí kódu (funkcí, metod) izolovaně. Provádí se vývojáři během kódování.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import org.junit.runners.JUnit4;

public class SolutionTest {
    @Test
    public void sampleTests() {
        assertEquals("dlrow", SomeClass.solution("world"));
    }
}
```

Klíčové aspekty unit testů

- Izolace: Jednotkové testy se zaměřují na jednu funkci nebo metodu, bez závislosti na jiných částech systému. Díky tomu se testy zaměřují na konkrétní chování testované jednotky.
- Automatizace: Jednotkové testy jsou obvykle automatizované a snadno spustitelné po každé změně kódu. To umožňuje rychle zjistit, zda nové změny neporušily existující funkcionalitu.
- Rychlost a efektivita: Jednotkové testy jsou rychlé, protože testují malou část kódu. Díky tomu jsou ideální pro průběžné testování během vývoje.
- Opakovatelnost: Testy jsou opakovatelné, takže je možné je spouštět kdykoliv, což zajišťuje konzistenci a spolehlivost testovaných funkcí.

Výhody jednotkových testů

- Rychlá detekce chyb: Chyby jsou odhaleny přímo na úrovni jednotlivých funkcí, což usnadňuje jejich nalezení a opravu.
- Podpora refaktoringu: Testy usnadňují refaktoring, protože vývojáři mají jistotu, že kód funguje i po provedení změn.
- Zvýšení kvality kódu: Jednotkové testy podporují psaní kvalitnějšího a strukturovanějšího kódu, protože je testována každá malá část aplikace.
- Snadná údržba: Díky izolaci a jasnému zaměření je snadné údržbu testů, což je obzvlášť užitečné pro velké projekty.

Nevýhody jednotkových testů

- Náklady na tvorbu a údržbu: I když jednotkové testy z dlouhodobého hlediska šetří čas, jejich počáteční tvorba a údržba mohou být časově a finančně náročné.
- Neodhalí problémy s integrací: Jednotkové testy testují jednotlivé části aplikace izolovaně, takže nezachytí chyby vznikající při propojení různých částí systému.
- Závislost na dobrém návrhu: Jednotkové testy jsou efektivní pouze tehdy, pokud je kód dobře navržený, modularizovaný a snadno testovatelný. Špatně napsaný kód může jednotkové testy zkomplikovat.

Integrační testy (Integration Testing)

Testování kombinace modulů nebo komponent společně. Cílem je ověřit správnou spolupráci mezi různými částmi systému.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
@Transactional
public class UserServiceIntegrationTest {

    @Autowired
    private UserService userService;

    @Test
    public void testCreateAndFindUser() {
        User user = userService.createUser("John Doe", "john@example.com");
        assertThat(user.getId()).isNotNull();
        User foundUser = userService.findUserByEmail("john@example.com");
        assertThat(foundUser).isNotNull();
        assertThat(foundUser.getName()).isEqualTo("John Doe");
        assertThat(foundUser.getEmail()).isEqualTo("john@example.com");
    }
}
```

Klíčové aspekty integračních testů

- Testování interakce mezi moduly: Integrační testy se zaměřují na rozhraní a komunikaci mezi moduly. Zajišťují, že data a procesy přecházejí správně mezi jednotlivými částmi systému.
- Zachycení chyb v propojení: Zatímco jednotkové testy testují každou část aplikace izolovaně, integrační testy odhalují chyby vznikající při spojení těchto částí, například nesprávnou integraci API, chybné volání funkcí apod.
- Testování ve fázích: Integrační testy se často provádějí postupně, kde se nejprve testují menší komponenty nebo moduly a následně se připojují k větším částem systému, až je celý systém plně integrovaný.
- Automatizace i manuální testování: Integrační testy mohou být jak automatizované, tak i manuální, v závislosti na složitosti systému a požadavcích na přesnost.

Výhody integračních testů

- Odhalení problémů mezi komponentami: Pomáhají identifikovat problémy, které by jednotkové testy nezachytily, jako jsou chyby v rozhraních nebo špatné formáty dat.
- Snazší detekce chyb dříve v procesu vývoje: Díky postupné integraci je možné odhalit a opravit problémy v raných fázích projektu.
- Validace softwarových požadavků: Zajišťuje, že všechny komponenty společně splňují specifikace a požadavky systému.

Nevýhody integračních testů

- Složitost a časová náročnost: Návrh, implementace a údržba integračních testů mohou být náročné, zejména ve velkých a složitých systémech.
- Závislost na stabilitě komponent: Když je některá komponenta nedokončená nebo nestabilní, může integrační testování selhat nebo být odloženo.
- Drahá údržba: V případě změn v rozhraních nebo integracích mohou být testy nákladné na aktualizaci.

Systémové testy (System Testing)

Komplexní testování celého systému jako celku. Testování funkčnosti, výkonu, bezpečnosti a dalších aspektů softwaru.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class SystemTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testUserRegistrationAndLogin() {
        User newUser = new User("testuser", "password123");
        ResponseEntity<User> registrationResponse = restTemplate.postForEntity("/auth/register", newUser, User.class);
        assertThat(registrationResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
        User registeredUser = registrationResponse.getBody();
        assertThat(registeredUser).isNotNull();
        assertThat(registeredUser.getUsername()).isEqualTo("testuser");

        ResponseEntity<User> loginResponse = restTemplate.postForEntity("/auth/login", newUser, User.class);

        assertThat(loginResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
        User loggedInUser = loginResponse.getBody();
        assertThat(loggedInUser).isNotNull();
        assertThat(loggedInUser.getUsername()).isEqualTo("testuser");
    }
}
```

Klíčové aspekty systémových testů

- Testování plně integrovaného systému: Systémové testy ověřují celý systém jako jednotku, včetně všech jeho modulů a rozhraní.
- Validace požadavků: Systémové testy kontrolují, zda aplikace splňuje funkční a nefunkční požadavky definované v požadavkové dokumentaci.
- Reálné uživatelské prostředí: Testování probíhá co nejvíce v prostředí, které odpovídá produkčnímu (např. se simulovanými uživateli, reálnými daty apod.).
- End-to-End testování: Systémové testy často zahrnují scénáře, které simulují reálné uživatelské úkoly a sledují celý proces od začátku do konce (např. od přihlášení až po dokončení transakce).

Typy systémového testování

- Funkční testování
- Nefunkční testování
- End-to-End testování
- Regresní testování

Výhody systémových testů

- Ověření systému jako celku: Zajišťují, že systém funguje jako kompletní aplikace, a ne pouze jako jednotlivé komponenty.
- Odhalení chyb v reálném prostředí: Testy probíhají v prostředí blízkém produkčnímu, což pomáhá odhalit problémy, které by mohly nastat v praxi.
- Validace požadavků uživatele: Systémové testy ověřují, že aplikace splňuje všechny definované funkční i nefunkční požadavky.

Nevýhody systémových testů

- Časová a nákladová náročnost: Systémové testy vyžadují rozsáhlé testovací prostředí a často simulaci reálných uživatelských scénářů, což může být časově a finančně náročné.
- Složitost při testování komplexních systémů: V rozsáhlých a komplexních systémech může být náročné pokrýt všechny možné uživatelské scénáře a závislosti mezi moduly.
- Náročná identifikace problémů: Pokud test selže, může být obtížné zjistit, která konkrétní část systému způsobila chybu, protože se testuje celý systém jako celek.

UI testy (UI Testing)

Cílem je ověřit, zda software splňuje požadavky a očekávání uživatelů. Může být manuální nebo automatizované

```
@Before
public void setUp() {
    System.setProperty("webdriver.chrome.driver", "C:\\Users\\hanss\\Desktop\\sda\\Kody\\SeleniumTr19\\src\\main\\resources\\chromedriver.exe");
    driver = new ChromeDriver();
}

HansSima
@After
public void tearDown() { driver.quit(); }

HansSima
@Test
public void mujTest() {
    driver.get("https://www.tutorialspoint.com/selenium/selenium_automation_practice.htm");
    driver.manage().window().setSize(new Dimension( width: 1936,  height: 1000));
    driver.findElement(By.xpath( xpathExpression: "/html/body/div[3]/div[2]/div[1]/div[2]/div[2]/button[1]")).click();
    driver.findElement(By.name("firstname")).sendKeys( ...keysToSend: "Jan");
    driver.findElement(By.name("lastname")).sendKeys( ...keysToSend: "Sima");
    driver.findElement(By.xpath( xpathExpression: "/html/body/main/div/div/div[2]/div[3]/div/form/div/table/tbody/tr[3]/td[2]/input[2]")).click();
    driver.findElement(By.xpath( xpathExpression: "/html/body/main/div/div/div[2]/div[3]/div/form/div/table/tbody/tr[4]/td[2]/span[3]/input")).click();
    driver.findElement(By.xpath( xpathExpression: "/html/body/main/div/div/div[2]/div[3]/div/form/div/table/tbody/tr[5]/td[2]/input")).sendKeys( ...keysToSend: "10.11.2022");
    driver.close();
}
```

Klíčové aspekty UI testů

- Testování funkčnosti prvků UI: Kontrola, zda jsou všechny interaktivní prvky použitelné (např. tlačítka reagují na kliknutí, formuláře lze odeslat).
- Vizuální konzistence: Ověření, zda je vzhled aplikace v souladu s designem (např. správné barvy, fonty, rozvržení) a že UI prvky správně reagují na změny velikosti obrazovky nebo přiblížení.
- Kompatibilita napříč zařízeními: UI testy zahrnují kontrolu kompatibility mezi různými zařízeními, prohlížeči, a rozlišeními obrazovky.
- Validace použitelnosti: Ověření, že rozhraní je intuitivní a snadno použitelné pro koncové uživatele.

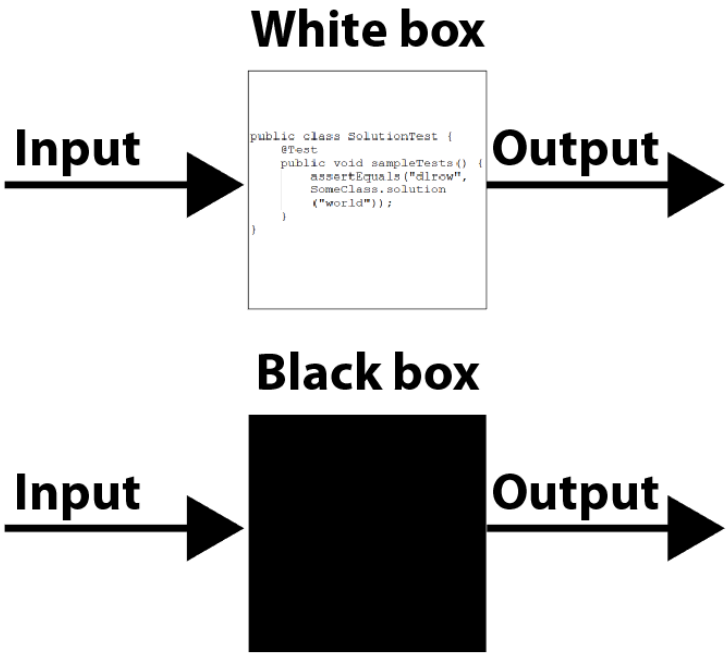
Výhody UI testů

- Zlepšení uživatelské zkušenosti: Dobře navržené UI zvyšuje uživatelskou spokojenost a usnadňuje práci s aplikací.
- Snadnější odhalení vizuálních problémů: UI testování pomáhá zachytit chyby v grafickém zobrazení, nesoulad s designem nebo chyby v rozložení prvků.

Nevýhody UI testů

- Časově náročné manuální testování: Manuální UI testování může být zdlouhavé, zejména u složitých aplikací s více obrazovkami.
- Nákladná automatizace: Automatizace UI testů vyžaduje údržbu a je náročná na správu při častých změnách UI.

Podle znalosti kódu



Black box

Black Box Testing (testování černou skříňkou) je metoda softwarového testování, při které testeři hodnotí funkčnost softwaru bez nahlížení na vnitřní strukturu kódu. Tento přístup lze uplatnit na všechny úrovně testování softwaru, jako je testování jednotek, integrační testování, systémové a akceptační testování.

Testeři vytvářejí testovací scénáře/případy na základě požadavků a specifikací softwaru. Tato metoda se proto také nazývá testování na základě specifikace, behaviorální testování a testování vstup-výstup.

Tester provádí testování pouze na funkční části aplikace, aby se ujistil, že chování softwaru odpovídá očekáváním.

Výhody

- Bez nutnosti znalosti kódu: Testování černou skříňkou nevyžaduje žádné znalosti vnitřní struktury nebo programovacího jazyka aplikace, takže je vhodné i pro netechnické testery.
- Zaměření na uživatelskou perspektivu: Tento typ testování se zaměřuje na požadavky a očekávání koncových uživatelů, což pomáhá zajistit, že aplikace bude odpovídat jejich potřebám a bude snadno použitelná.
- Efektivní pro odhalení chyb v požadavcích: Testování černou skříňkou umožňuje odhalit chyby v funkcionalitě, které mohou pramenit z nesprávně pochopených požadavků nebo specifikací.
- Vhodné pro všechny úrovně testování: Testování černou skříňkou lze provádět na různých úrovních (jednotkové, integrační, systémové, akceptační), což usnadňuje kontrolu aplikace v různých fázích vývoje.

Nevýhody

- Nemožnost odhalit skryté chyby v kódu: Jelikož se nezohledňuje interní struktura kódu, testování černou skříňkou nemůže odhalit chyby, které jsou skryté hluboko uvnitř kódu, například neefektivní algoritmy nebo špatné proměnné.
- Vyšší závislost na specifikacích: Úspěch testování závisí na kvalitě specifikací. Pokud nejsou požadavky nebo specifikace jasně definovány, může být obtížné vytvořit kvalitní testovací scénáře.
- Může být náhodné nebo neúplné: Jelikož tester nevidí kód, může se stát, že testování pokryje pouze omezenou část funkcionality nebo že některé cesty v aplikaci zůstanou neotestovány.
- Náročné odhalování chyb na nižší úrovni: Testování černou skříňkou je zaměřeno na celkové chování aplikace, což znamená, že může být obtížné identifikovat konkrétní příčiny chyb nebo problémů v konkrétních částech kódu.

White box

White Box Testing (testování bílou skříňkou) je technika softwarového testování založená na interní struktuře kódu aplikace. Při testování bílou skříňkou se využívají programátorské dovednosti k návrhu testovacích případů. Tento typ testování se obvykle provádí na úrovni jednotek (unit level). Název white box vyjadřuje schopnost „vidět“ přes vnější obal softwaru (tj. skříňku) do jeho vnitřních procesů.

Výhody

- Detailní pokrytí kódu: Testování bílou skříňkou umožňuje kontrolovat každý řádek kódu, takže může odhalit skryté chyby, logické nedostatky a překryté cesty, které by při jiných testovacích technikách zůstaly neodhaleny.
- Optimalizace kódu: Protože se testuje přímo kód, lze při testování najít a odstranit neefektivní nebo nepotřebné části, což může vést k optimalizaci a zlepšení výkonu aplikace.
- Včasná detekce chyb: White box testing je často prováděn na úrovni jednotkového testování, takže chyby jsou objeveny již v rané fázi vývoje, což snižuje náklady a čas potřebný na opravy.
- Zajištění správné logiky a bezpečnosti: Testování bílou skříňkou pomáhá otestovat, zda všechny logické podmínky, větve a bezpečnostní mechanismy fungují správně a neobsahují slabá místa.

Nevýhody

- Vyžaduje programátorské znalosti: K provedení testování bílou skříňkou je nutná znalost programování a detailní porozumění kódu, což může omezit účast testovacích týmů, které nejsou technicky zaměřené.
- Časová náročnost: Testování každé části kódu a návrh testovacích případů zaměřených na interní logiku a větvení kódu může být časově náročné a vyžaduje velké úsilí.
- Omezené zaměření na funkční požadavky: White box testing se soustředí na kód, což znamená, že se může stát, že nebude brát dostatečně v úvahu koncové uživatele a jejich očekávání ohledně funkčnosti.
- Nedokáže odhalit chybějící funkce: Testování bílou skříňkou se zaměřuje na kód a logiku, ale není schopno odhalit chybějící funkcionalitu, která v kódu není implementována.

Podle způsobu realizace testů

Manuální testování (Manual Testing)

- Testování prováděné ručně testery, kteří sledují a zaznamenávají chování softwaru.
- Vhodné pro explorativní testování, uživatelské testy a případy, kdy je automatizace obtížná.

Automatizované testování (Automated Testing)

- Použití nástrojů a skriptů k automatizaci testovacích scénářů.
- Vhodné pro opakované testování, regresní testy a testy, které vyžadují vysokou přesnost.

Podle dimenzí kvality

Funkční testování (Functional Testing)

Funkční testování zjišťuje, co systém skutečně dělá. Cílem je ověřit, zda každá funkce softwarové aplikace funguje podle specifikace uvedené v dokumentu s požadavky. Testují se všechny funkcionality tím, že se zadají odpovídající vstupy a zjišťuje se, zda skutečný výstup odpovídá očekávanému výstupu. S tímto typem testování se budeme setkávat po zbytek kurzu.

Klíčové aspekty funkčního testování

- Ověření požadavků: Testování podle specifikací a požadavků, aby se zajistilo, že aplikace poskytuje funkce popsané v dokumentaci a požadavcích.
- Testování vstupů a výstupů: Testeři poskytují různé typy vstupů (např. platné i neplatné) a ověřují, zda výstupy odpovídají očekávaným výsledkům. Pokud výstup neodpovídá, znamená to potenciální chybu.
- Zaměření na uživatelské scénáře: Funkční testování simuluje skutečné chování uživatele a testuje aplikační funkce způsobem, jakým by je mohl využívat koncový uživatel.
- Testovací případy: Případy jsou obvykle založeny na funkčních specifikacích a zahrnují různé scénáře, které by uživatel mohl v aplikaci použít. Testovací případy mohou zahrnovat různé kombinace vstupních hodnot, aby se pokryly všechny možné situace.

Typy funkčního testování

- Jednotkové testování (Unit Testing)
- Integrační testování (Integration Testing)
- Systémové testování (System Testing)
- Akceptační testování (Acceptance Testing)

Výhody funkčního testování

- Ověření správného fungování aplikace: Zajišťuje, že aplikace funguje podle očekávání a splňuje požadavky zákazníka.
- Identifikace chyb a nesrovnalostí: Pomáhá odhalit chyby a nesrovnalosti v raných fázích vývoje.
- Přístup z pohledu uživatele: Zaměřuje se na to, jak uživatel vnímá funkčnost aplikace, což přispívá k lepší uživatelské zkušenosti.

Nevýhody funkčního testování

- Neodhalí všechny chyby: Funkční testování se zaměřuje pouze na funkčnost aplikace, ale nemusí odhalit problémy s výkonem, zabezpečením nebo optimalizací kódu.
- Časová náročnost: Pro rozsáhlé aplikace může být vytváření a provádění funkčních testů časově náročné.

Funkční testování je klíčové pro zajištění, že aplikace splňuje své specifikace a očekávání uživatelů. Je jedním z nejdůležitějších typů testování, protože zajišťuje, že aplikace je spolehlivá, funkční a použitelná.

Výkonové testování (Performance Testing)

Cílem výkonového testování je určit, jak aplikace reaguje a funguje pod určitými podmínkami. Zahrnuje měření různých parametrů, jako jsou rychlost odezvy, propustnost, využití zdrojů (CPU, paměť) atd.

Typy výkonového testování

- Testování odezvy: Měří čas, který aplikace potřebuje k odpovědi na určitou akci.
- Testování propustnosti: Určuje, kolik transakcí nebo požadavků může aplikace zpracovat za určitou dobu.
- Testování škálovatelnosti: Zkoumá, jak dobře se aplikace přizpůsobuje zvýšení zatížení nebo rozšíření.

Příklad

- Měření doby načítání webové stránky při běžném provozu.
- Měření využití paměti a CPU při různých typech požadavků.

Výhody

- Identifikace a odstranění výkonových „úzkých míst“.
- Zajištění, že aplikace splňuje požadované výkonnostní specifikace.

Nevýhody

- Může být časově a finančně náročné.
- Vyžaduje speciální nástroje a odborné znalosti.

Výkonové testování: Zaměřuje se na obecný výkon aplikace a její odezvu pod různými podmínkami. Zjišťuje, jak rychle a efektivně aplikace funguje.

Příklady performence testů: https://www.perfmatrix.com/performance-test-result-analysis-basic/

Zátěžové testování (Load Testing)

Cílem zátěžového testování je zjistit, jak aplikace funguje při očekávaném zatížení. Simuluje reálné uživatelské zatížení, aby se zjistilo, zda aplikace zvládne předpokládaný počet uživatelů nebo transakcí.

Typy zátěžového testování

- Testování při špičkovém zatížení: Simuluje maximální počet uživatelů, které se očekává v nejvytíženější době.
- Testování dlouhodobé zátěže (Soak Testing): Simuluje běžné uživatelské zatížení po delší časové období, aby se odhalily problémy s výkonem nebo úniky paměti.

Příklad

- Simulace 50000 uživatelů přistupujících na webovou stránku současně.
- Testování e-commerce aplikace během Black Friday, aby se zajistilo, že zvládne vysoký počet nákupních transakcí.

Výhody

- Ověření stability a spolehlivosti aplikace pod očekávaným zatížením.
- Identifikace a odstranění výkonových problémů při vysokém zatížení.

Nevýhody

- Může vyžadovat rozsáhlou infrastrukturu pro simulaci zatížení.
- Náročné na přípravu a provedení.

Zátěžové testování: Simuluje specifické scénáře zatížení, aby zjistilo, zda aplikace zvládne očekávaný počet uživatelů nebo transakcí. Pomáhá zajistit stabilitu a spolehlivost při vysokém zatížení.

Stresové testování (Stress testing)

Stresové testování (stress testing) je typ testování softwaru, jehož cílem je zjistit, jak systém reaguje na extrémní zatížení a jaké jsou jeho limity. Tento typ testování se zaměřuje na zjištění bodu, kdy systém začne selhávat, a na identifikaci chování systému při přetížení.

Cíle stresového testování

- Identifikace hranic výkonu: Zjistit maximální kapacitu systému, jako je počet uživatelů nebo transakcí, které může systém zvládnout před selháním.
- Zjištění chování při selhání: Analyzovat, jak se systém chová při dosažení nebo překročení maximálního zatížení (např. zda dochází k pádům, ztrátě dat nebo degradaci výkonu).
- Ověření zotavení systému: Testovat, jak dobře se systém zotavuje po selhání nebo extrémním zatížení.
- Detekce úzkých míst: Identifikovat komponenty nebo oblasti systému, které se stávají úzkými místy při vysokém zatížení.

Příklady stresového testování

- Simulace extrémního uživatelského zatížení: Např. testování webové aplikace s 10 000 uživateli, kteří se přihlašují současně, aby se zjistilo, kdy server přestane odpovídat.
- Generování velkého objemu dat: Např. vkládání milionů záznamů do databáze v krátkém čase, aby se zjistilo, jak se databázový systém chová při extrémním zatížení.
- Dlouhodobé zatížení: Provozování systému na maximální kapacitě po delší dobu, aby se odhalily problémy jako úniky paměti nebo degradace výkonu.
- Náhlé zvýšení zátěže: Náhlé zvýšení počtu požadavků na server během krátkého období (např. 1000 požadavků za sekundu), aby se zjistilo, jak server zvládá náhlé špičky.

Výhody stresového testování

- Identifikace limitů systému: Pomáhá zjistit maximální kapacitu a hranice výkonu systému.
- Zvýšení spolehlivosti: Odhaluje potenciální problémy a slabá místa, která mohou být opravena před nasazením do produkce.
- Příprava na neočekávané situace: Pomáhá připravit systém na extrémní nebo neočekávané podmínky, čímž zvyšuje jeho odolnost.

Nevýhody stresového testování

- Náročnost na zdroje: Vyžaduje značné množství zdrojů (čas, hardware, software) pro simulaci extrémních podmínek.

- Komplexita: Návrh a realizace realistických scénářů stresového testování může být složitá.
- Potenciální riziko poškození: Testování při extrémním zatížení může potenciálně způsobit poškození systému nebo ztrátu dat, pokud není prováděno opatrně.

Stresové testování je klíčovou součástí zajištění kvality softwaru, která pomáhá zjistit, jak se systém chová při extrémním zatížení. Identifikuje hranice výkonu, odhaluje úzká místa a umožňuje týmům připravit se na neočekávané podmínky. I když je náročné na zdroje a složité na realizaci, poskytuje cenné informace, které mohou zvýšit spolehlivost a odolnost systému.

Usability testing (testování použitelnosti)

Usability testing (testování použitelnosti) je proces, který se zaměřuje na hodnocení produktu nebo služby z hlediska jeho snadnosti použití, intuitivnosti a celkového uživatelského zážitku. Cílem je zjistit, jak dobře uživatelé mohou produkt používat k dosažení svých cílů a jaké problémy při tom mohou nastat.

Klíčové aspekty usability testingu

- Zaměření na uživatele: Testování probíhá s reálnými nebo potenciálními uživateli produktu, kteří vykonávají běžné úkoly.
- Scénáře a úkoly: Uživatelům jsou zadány specifické úkoly, které by měli být schopni vyřešit pomocí testovaného produktu.
- Pozorování a záznam: Testování se provádí za přítomnosti pozorovatelů, kteří sledují uživatele, zaznamenávají jejich chování a shromažďují data o jejich zkušenostech.
- Kvalitativní a kvantitativní data: Výsledky testování mohou zahrnovat subjektivní zpětnou vazbu od uživatelů (kvalitativní data) i objektivní metriky jako je čas potřebný k dokončení úkolů, počet chyb nebo míra úspěšnosti (kvantitativní data).

Plánování

- Definování cílů: Určení, co se má testováním zjistit (např. snadnost navigace, srozumitelnost uživatelského rozhraní).
- Výběr účastníků: Výběr reprezentativní skupiny uživatelů, kteří odpovídají cílové skupině produktu.
- Příprava scénářů a úkolů: Vytvoření realistických úkolů, které uživatelé budou během testování provádět.

Provádění testování

- Úvodní briefing: Seznámení účastníků s cílem testování a vysvětlení průběhu.
- Pozorování a interakce: Uživatelé vykonávají zadané úkoly, zatímco pozorovatelé sledují jejich postup a zaznamenávají jejich chování.
- Sbírání zpětné vazby: Po dokončení úkolů uživatelé poskytují zpětnou vazbu a sdílejí své dojmy a zkušenosti.

Analýza a vyhodnocení

- Analýza dat: Vyhodnocení kvalitativních a kvantitativních dat získaných během testování.
- Identifikace problémů: Zjištění hlavních problémů a obtíží, které uživatelé zažili.
- Doporučení pro zlepšení: Návrhy na úpravy a vylepšení produktu na základě zjištěných problémů.

Implementace změn

- Úpravy produktu: Implementace doporučených změn a vylepšení.
- Opakování testování: Opakování testování po implementaci změn pro ověření, že problémy byly vyřešeny.

Výhody usability testingu

- Zlepšení uživatelského zážitku: Identifikace a odstranění problémů zvyšuje spokojenost uživatelů.
- Vyšší efektivita a produktivita: Uživatelé mohou dosáhnout svých cílů rychleji a s menšími obtížemi.
- Snížení nákladů: Včasné odhalení a oprava problémů může snížit náklady na podporu a opravy po nasazení.
- Zvýšení konkurenceschopnosti: Lepší použitelnost může zvýšit atraktivitu produktu na trhu.

Nevýhody usability testingu

- Náklady a časová náročnost: Organizace a provedení testování mohou být nákladné a časově náročné.
- Omezený vzorek uživatelů: Výsledky mohou být ovlivněny omezeným počtem testovaných uživatelů.
- Subjektivita: Některé problémy a zpětná vazba mohou být subjektivní a závislé na konkrétních uživatelích.

Usability testing je kritickou součástí vývoje uživatelsky přívětivých produktů. Poskytuje cenné informace o tom, jak skuteční uživatelé interagují s produktem, a pomáhá identifikovat a opravit problémy, které by mohly bránit optimálnímu uživatelskému zážitku. I když je tento proces náročný na zdroje, jeho přínosy v podobě zvýšené spokojenosti uživatelů a konkurenceschopnosti produktu často převyšují náklady.

Testování udržovatelnosti (Maintainability testing)

Testování udržovatelnosti (maintainability testing) je proces hodnotící, jak snadno může být software upravován, opravován, rozšiřován nebo přizpůsobován změnám po jeho nasazení. Zaměřuje se na aspekty, které ovlivňují schopnost vývojářů provádět změny a aktualizace softwaru efektivně a bez chyb.

Cíle testování udržovatelnosti

- Identifikace problémů s kódem: Zjištění částí kódu, které jsou obtížně pochopitelné, udržovatelné nebo náchylné k chybám.
- Hodnocení dokumentace: Posouzení kvality a úplnosti dokumentace, která usnadňuje budoucí údržbu.
- Analýza architektury: Posouzení softwarové architektury z hlediska modularity, škálovatelnosti a adaptability.
- Měření metrik: Použití metrik k hodnocení složitosti kódu, závislostí mezi moduly a dalších faktorů ovlivňujících udržovatelnost.

Plánování

- Definování cílů: Určení konkrétních cílů a oblastí, které budou testovány (např. čitelnost kódu, modularita, dokumentace).
- Výběr metrik: Určení metrik, které budou použity k hodnocení udržovatelnosti (např. počet řádků kódu, míra cyklomatické složitosti).

Provádění testování

- Statická analýza kódu: Použití nástrojů pro statickou analýzu kódu ke kontrole dodržování kodexových standardů a identifikaci problémových oblastí.
- Kontrola kódu: Manuální přezkum kódu provedený zkušenými vývojáři, zaměřený na identifikaci oblastí, které mohou být problematické z hlediska údržby.
- Hodnocení dokumentace: Kontrola kvality a úplnosti dokumentace, včetně komentářů v kódu, technické dokumentace a uživatelských manuálů.

Analýza výsledků

- Vyhodnocení metrik: Analýza naměřených hodnot a metrik, identifikace problémových oblastí a odhad náročnosti budoucí údržby.
- Identifikace rizik: Určení potenciálních rizik a oblastí, které mohou v budoucnu vyžadovat zvýšenou údržbu.

Zpráva a doporučení

- Vytvoření zprávy: Dokumentace výsledků testování, identifikovaných problémů a doporučení pro zlepšení udržovatelnosti.
- Doporučení pro zlepšení: Poskytnutí konkrétních návrhů na úpravy kódu, dokumentace nebo architektury za účelem zvýšení udržovatelnosti.

Výhody testování udržovatelnosti

- Snížení nákladů na údržbu: Identifikace a odstranění problémů s kódem předem může snížit budoucí náklady na údržbu.
- Zvýšení efektivity vývojářů: Snadno udržovatelný kód umožňuje vývojářům rychleji a efektivněji provádět změny a opravy.
- Lepší kvalita softwaru: Pravidelné hodnocení udržovatelnosti přispívá k vyšší celkové kvalitě a stabilitě softwaru.
- Snížení rizika chyb: Lepší dokumentace a čitelnější kód snižují riziko vzniku chyb při údržbě.

Nevýhody testování udržovatelnosti

- Časová náročnost: Testování a analýza udržovatelnosti mohou být časově náročné, zejména u velkých a komplexních systémů.
- Náklady na nástroje a školení: Vyžaduje investice do nástrojů pro analýzu kódu a školení vývojářů, aby byli schopni provádět efektivní testování.
- Subjektivní hodnocení: Některé aspekty udržovatelnosti, jako je čitelnost kódu, mohou být subjektivní a závislé na zkušenostech a znalostech jednotlivých vývojářů.

Testování udržovatelnosti je klíčovou součástí zajištění kvality softwaru, která pomáhá zajistit, že software bude snadno upravovatelný, rozšiřitelný a přizpůsobitelný budoucím požadavkům. Přestože je tento proces časově náročný a vyžaduje investice do nástrojů a školení, jeho přínosy v podobě snížení nákladů na údržbu, zvýšení efektivity vývojářů a zlepšení kvality softwaru jsou značné.

Testy spolehlivosti (Reliability testing)

Reliabilita je aspekt testování softwaru, který se zaměřuje na zajištění spolehlivosti softwaru.

Cíle testování spolehlivosti

- Cílem testování spolehlivosti je zajistit, že software bude fungovat bezchybně po určitou dobu a za různých podmínek. Spolehlivost se měří schopností softwaru poskytovat správné výsledky a udržovat svou funkčnost bez selhání.

Klíčové aspekty

- Stabilita: Posouzení, jak stabilní je software při běžném i extrémním použití.
- Dlouhodobé testování: Testování softwaru po delší časové období, aby se zjistilo, zda nedochází k degradaci výkonu.
- Zotavení po chybě: Zjištění, jak dobře se software zotavuje po výskytu chyb nebo selhání.

Typy testování spolehlivosti

- Testování životního cyklu (Soak Testing): Provozování softwaru po dlouhou dobu při normálním zatížení, aby se zjistilo, zda nedochází k degradaci výkonu nebo únikům paměti.
- Testování zotavení (Recovery Testing): Simulace chyb a selhání, aby se zjistilo, jak dobře se software zotavuje a obnovuje svou funkčnost.
- Testování odolnosti (Stress Testing): Testování softwaru za extrémních podmínek, aby se zjistilo, jak se chová při přetížení.

Výhody

- Zvyšuje důvěru uživatelů v software.
- Pomáhá identifikovat a odstranit potenciální problémy, které by mohly vést k selháním.
- Zajišťuje stabilní a spolehlivý provoz softwaru.

Nevýhody

- Časově náročné, zejména při dlouhodobém testování.
- Vyžaduje pečlivé plánování a rozsáhlé zdroje.

Testování spolehlivosti se zaměřuje na zajištění, že software bude fungovat stabilně a bezchybně po určitou dobu a za různých podmínek.

Testy přenositelnosti (Portability testing)

Portabilita je aspekt testování softwaru, který se zaměřuje na zajištění přenositelnosti softwaru napříč různými prostředími.

Cíle testování přenositelnosti

- Cílem testování přenositelnosti je zajistit, že software může být úspěšně nasazen a provozován v různých prostředích (operační systémy, hardwarové platformy, prohlížeče apod.) bez nutnosti úprav.

Klíčové aspekty

- Kompatibilita: Posouzení, jak dobře software funguje v různých prostředích.
- Přenositelnost dat: Zajištění, že data mohou být přenesena mezi různými systémy bez ztráty nebo poškození.
- Závislosti na prostředí: Identifikace a minimalizace závislostí softwaru na specifických prvcích prostředí.

Typy testování přenositelnosti

- Cross-platform Testing: Testování softwaru na různých operačních systémech a hardwarových platformách.
- Testování kompatibility prohlížečů: Testování webových aplikací v různých webových prohlížečích, aby se zajistilo, že fungují konzistentně.
- Testování migrace dat: Ověření, že data mohou být přesunuta mezi různými systémy bez problémů.

Výhody

- Umožňuje širší nasazení softwaru na různých platformách.
- Zvyšuje spokojenost uživatelů tím, že zajišťuje konzistentní uživatelský zážitek napříč různými prostředími.
- Snižuje náklady na údržbu a přizpůsobení softwaru pro různá prostředí.

Nevýhody

- Může být náročné a časově náročné testovat software ve všech možných prostředích.
- Vyžaduje přístup k různým platformám a konfiguracím, což může být nákladné.

Testování přenositelnosti se zaměřuje na to, že software může být úspěšně nasazen a provozován v různých prostředích bez nutnosti úprav.

Bezpečnostní testy (Security Testing)

Security Testing je typ softwarového testování zaměřený na identifikaci a odstranění zranitelností, které by mohly ohrozit bezpečnost aplikace nebo systému. Jeho cílem je zajistit, aby aplikace chránila data uživatelů a odolala potenciálním kybernetickým útokům. Security Testing zahrnuje různé metody a postupy, které pomáhají minimalizovat bezpečnostní rizika a zajistit soulad s právními požadavky na ochranu dat.

Klíčové aspekty

- Autentizace a autorizace: Ověření, že uživatelé mají přístup pouze k těm funkcím a datům, k nimž jsou oprávněni, a že se aplikace umí bránit proti pokusům o prolomení přístupových údajů.
- Ochrana citlivých dat: Zajištění, že aplikace správně zpracovává citlivá data (např. hesla, osobní údaje, finanční informace), šifruje je a chrání před neoprávněným přístupem.
- Zabezpečení připojení a komunikace: Testování, zda komunikace mezi klientem a serverem probíhá bezpečně (například pomocí HTTPS nebo jiných šifrovacích protokolů), aby se zabránilo odposlechu či změně přenášených dat.
- Ochrana před zranitelnostmi: Zahrnuje testování proti běžným typům útoků, jako jsou SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), buffer overflow a další.
- Správa relací (session management): Zajištění, že relace uživatelů jsou bezpečně řízeny, včetně bezpečného přihlašování, odhlašování a ochrany před únosem relací (session hijacking).
- Zálohy a zotavení po havárii: Kontrola, zda aplikace má mechanismy pro zálohování a obnovu dat, aby mohla fungovat i po narušení bezpečnosti nebo jiných neočekávaných událostech.
- Detekce a prevence bezpečnostních událostí: Nastavení monitoringu a protokolování, které dokáží identifikovat a zaznamenat podezřelé aktivity v reálném čase, aby bylo možné na útoky rychle reagovat.

Security Testing je zásadní, protože:

- Chrání důvěrná data uživatelů a chrání organizaci před ztrátou důvěry zákazníků.
- Pomáhá organizaci vyhnout se právním a finančním důsledkům, které mohou vzniknout při úniku dat.
- Minimalizuje riziko ztráty či poškození dat a škodlivých útoků.

Lokalizační testy (Localization Testing)

Localization Testing je typ softwarového testování zaměřený na přizpůsobení aplikace konkrétnímu regionu, jazyku nebo kultuře. Jeho cílem je zajistit, aby aplikace správně fungovala a odpovídala požadavkům místního trhu nejen technicky, ale i z hlediska obsahu, formátování a uživatelské zkušenosti.

Klíčové aspekty

- Překlady: Ověření, že všechny texty a zprávy byly správně přeloženy a odpovídají jazykovým a kulturním normám. Je třeba zkontrolovat, zda překlady jsou přesné, jasné a profesionální.
- Formát dat a času: Testování, zda aplikace používá správné formáty data, času, měny a telefonních čísel podle zvyklostí dané země nebo regionu.
- Měna a měrné jednotky: Ověření, že měny, váhy, délky a další jednotky odpovídají místním normám a jsou správně zobrazeny.
- Specifické kulturní a právní prvky: Ujištění, že aplikace neobsahuje obsah, který by mohl být nevhodný nebo v rozporu s místními zvyklostmi nebo právními předpisy.
- Textové rozvržení a vzhled uživatelského rozhraní (UI): Kontrola, zda texty, které mohou mít různou délku v závislosti na jazyku, neovlivňují rozvržení nebo čitelnost UI. Například dlouhý text může způsobit přetékání obsahu nebo neúplné zobrazení.
- Kódování znaků: Ověření správného zobrazení speciálních znaků, písmen s diakritikou nebo symbolů specifických pro daný jazyk.

Chyby v lokalizaci mohou negativně ovlivnit uživatelskou zkušenost, snížit důvěryhodnost produktu a vést ke špatným recenzím. Lokalizace je zásadní pro aplikace, které se snaží proniknout na zahraniční trhy, a dobré lokalizace pomáhá aplikaci působit přirozeně a profesionálně v cílové kultuře.

Podle rozsahu

Regresní testy (Regression Testing)

Testování, zda nové změny nebo opravy chyb nezpůsobily nové problémy v již funkčním kódu. Často se provádí automatizovaně.

Klíčové aspekty regresního testování

- Ověření neporušené funkčnosti: Cílem je zajistit, že nová verze softwaru neporušila žádnou existující funkci.
- Opakované testování: Při každé změně v kódu se spustí regresní testy, aby se zajistilo, že dříve otestované části aplikace stále fungují správně.
- Automatizace testů: Regresní testy jsou často automatizované, protože je třeba je opakovaně spouštět při každé změně. Automatizace šetří čas a zajišťuje konzistentní provádění testů.
- Rychlá identifikace chyb: Regresní testy odhalují, zda úpravy kódu nevyvolaly neočekávané chyby nebo vedlejší účinky na jiných částech aplikace.

Kdy provádět regresní testování

- Po opravě chyby (bug fix): Ověření, že oprava chyby neovlivnila ostatní funkce aplikace.
- Po přidání nové funkce nebo modulu: Zajištění, že nové funkce neporušily existující funkčnost.
- Po refaktoringu kódu: Kontrola, že změny ve struktuře nebo optimalizaci kódu nezpůsobily žádné problémy.
- Po aktualizaci prostředí nebo knihoven: Ověření, že aplikace funguje správně i po aktualizacích knihoven, frameworků nebo runtime prostředí.

Výhody regresního testování

- Zachování stability aplikace: Pomáhá udržet stabilitu aplikace i při častých změnách.
- Včasná detekce chyb: Rychle upozorní na chyby, které byly zavedeny při aktualizacích kódu.
- Zvýšení důvěry ve změny: Umožňuje vývojářům rychleji a s větší jistotou provádět změny, protože vědí, že regresní testy odhalí případné problémy.

Nevýhody regresního testování

- Časová a finanční náročnost: Ruční regresní testy mohou být nákladné a časově náročné, zejména při rozsáhlých aplikacích.
- Náročnost na údržbu: Automatizované regresní testy vyžadují průběžnou údržbu, protože změny v kódu mohou vyžadovat aktualizaci testovacích skriptů.
- Opakování podobných scénářů: Opakování stejných testovacích scénářů může být únavné a snížit pozornost testerů.

Další testy dle rozsahu:

Sanity testy (Sanity Testing)

- Ověření, že konkrétní funkce nebo opravy chyb fungují správně.
- Zjednodušená forma regresního testování zaměřená na konkrétní části systému.

Smoke testy (Smoke Testing)

- Rychlé testování hlavních funkcí softwaru po nové kompilaci nebo sestavení, aby se ověřilo, že základní funkce fungují.
- Předchází podrobnějšímu testování.

Explorativní testy (Exploratory Testing)

- Testování bez předem definovaných testovacích scénářů, kdy tester aktivně objevuje systém a hledá problémy.
- Vhodné pro identifikaci neočekávaných chyb a problémů.

Risk-based testing

Risk-based testing je přístup k testování softwaru, který se zaměřuje na identifikaci a prioritizaci rizik, aby se maximalizovala efektivita testování a minimalizovala pravděpodobnost selhání softwaru v produkčním prostředí. Tento přístup se snaží alokovat testovací zdroje tam, kde jsou nejvíce potřeba, na základě hodnocení rizik.

$$RL = L * I$$

- RI – Risk Level
- L – Likelihood
- I - Impact

Klíčové aspekty risk-based testingu

- Identifikace rizik: Určení možných rizik, která mohou ovlivnit kvalitu softwaru, jako jsou funkční a nefunkční požadavky, technické problémy, komplexnost kódu, nové technologie, změny v obchodních procesech, apod.
- Hodnocení rizik: Každé identifikované riziko je hodnoceno podle pravděpodobnosti jeho výskytu a potenciálního dopadu. Toto hodnocení často zahrnuje:
    - Pravděpodobnost (Likelihood): Jak pravděpodobné je, že riziko nastane.
    - Dopad (Impact): Jaký dopad bude mít riziko na projekt nebo produkt, pokud se stane skutečností.
    - Prioritizace testování (Risk level): Na základě hodnocení rizik jsou testovací aktivity prioritizovány tak, aby se nejvíce zaměřily na oblasti s vysokým rizikem (vysoká pravděpodobnost a vysoký dopad).

Plánování

- Testovací plán je vytvořen s ohledem na prioritizovaná rizika. Obsahuje specifikace testovacích případů, alokaci zdrojů a harmonogram testování.

Provádění testů

- Realizace testovacích aktivit podle plánu, s důrazem na oblasti s vysokým rizikem.

Monitorování a hodnocení rizik

- Pravidelné sledování a přehodnocování rizik během celého životního cyklu testování. Na základě nových informací nebo změn v projektu mohou být rizika znovu hodnocena a testovací plán upraven.

Výhody risk-based testingu

- Efektivní využití zdrojů: Pomáhá zaměřit testovací úsilí na nejkritičtější části systému, čímž optimalizuje využití času a zdrojů.

- Lepší pokrytí kritických oblastí: Zajišťuje, že nejdůležitější a nejrizikovější oblasti jsou důkladně testovány.
- Snížení rizika selhání: Snižuje pravděpodobnost výskytu kritických chyb v produkčním prostředí.
- Flexibilita a adaptabilita: Umožňuje dynamické přizpůsobení testovacího plánu na základě změn v projektu a nových informací o rizicích.

Nevýhody risk-based testingu

- Závislost na správném hodnocení rizik: Kvalita risk-based testingu závisí na správném identifikování a hodnocení rizik, což může být subjektivní a náročné.
- Počáteční náklady a čas: Identifikace a hodnocení rizik mohou být časově náročné a vyžadovat zkušené odborníky.
- Méně důkladné testování nízkorizikových oblastí: Může vést k méně důkladnému testování částí systému, které jsou hodnoceny jako nízkorizikové.

Proces risk-based testingu

- Shromažďování informací: Získávání informací o projektu, požadavcích, specifikacích a technických detailech.
- Identifikace rizik: Brainstorming, analýza požadavků, hodnocení technologií a konzultace s odborníky.
- Hodnocení a klasifikace rizik: Použití metodik jako je FMEA (Failure Modes and Effects Analysis) nebo jiných analytických nástrojů k ohodnocení rizik.
- Plánování testů: Vytvoření testovacího plánu s ohledem na rizika, prioritizace testovacích případů a alokace zdrojů.
- Provádění testů: Realizace testovacích případů podle plánu, s důrazem na kritické oblasti.
- Monitorování a přizpůsobení: Pravidelné sledování výsledků testování, přehodnocování rizik a úpravy testovacího plánu podle potřeby.

Risk-based testing je efektivní přístup k testování softwaru, který se zaměřuje na identifikaci a prioritizaci rizik. Pomáhá optimalizovat testovací úsilí tím, že se soustředí na nejkritičtější části systému, čímž se zvyšuje pravděpodobnost odhalení a opravy významných chyb. Tento přístup je zvláště užitečný v situacích s omezenými zdroji a časem, protože umožňuje efektivnější a cílenější testování.

Výběr techniky test designu

- Typ systému - finanční systémy jsou testovány jinak než systémy, na nichž závisí lidské životy.
- Požadavky ve smlouvě - vymahatelné zákazníky.
- Úroveň rizika určená během procesu analýzy rizik.
- Účel testování určuje přístup k testování, zda jde o hledání defektů nebo seznamování se s aplikací nebo příprava automatických testů.
- Testovací techniky záleží na úrovni testování. White-box techniky jsou na unit a integrační úrovni. Black-box techniky jsou na systémové a akceptační úrovni.
- Dle dostupné dokumentace můžete vytvořit test casy dle modelu nebo zkusit zjistit více o softwaru díky exploratory testingu.
- Znalosti testerů - každý tester má unikátní znalosti, které lze použít v různých typech testů.
- Časová náročnost testů a rozpočet na testování umožňuje určit testovací techniku použitím např. testovací pyramidy.
- Dle typu metodologie vývoje softwaru, testovací tým se zaměří jen na black-box testy nebo je u celého vývojového cyklu softwaru.
- Zkušenosti z předchozích projektů - některé projekty jsou si podobné, vyplatí se použít získané zkušenosti.
- Dle typů defektů, např. funkcionální a výkonostní defekty vyžadují jiné testovací techniky.

Výběr techniky test designu není jednoduché. Měl by být probrán s test manažerem a s týmem vývojářů.

Shrnutí

Typy testů v IT lze rozdělit podle různých kritérií, jako je fáze vývoje, zaměření testu a způsob provedení. Každý typ testu má svůj specifický účel a přispívá k celkovému zajištění kvality softwaru. Použití kombinace různých typů testování je klíčové pro vytvoření robustního, spolehlivého a uživatelsky přívětivého softwaru.