Testování, Unit testování a Dokumentace zdrojového kódu

Testování je nepostradatelnou částí vývoje projektu.

Mělo by se sním začít až při dokončování projektu. Protože pokud by začala v začáteční fázi projektu, tak by se testy nebo unit testy musely přepisovat, pokud by byly změny v již vytvořené části projektu.

**Co se vše testuje**

Funkčnost

správné chování programů jejich metod, atributů, kolekcí, všeho…

Použitelnost

Schopnost programu vykonávat příkazy od uživatele nebo jiné typy příkazů.

Bezpečnost

Testování, které má za úkol odhalit bezpečností rizika, krádež dat,…

Spolehlivost

Testování chyb v programu, které by mohly vést k jeho pádu

Výkon

Jak se program chová při náročných akcí, které by neměly omezit jeho funkce.

Kompatibilita

Testování na rozdílných hardwarech, zjištění minimálních hardwarových specifikacích

**Druhy testů**

Čím pozdější fáze, tím by měl program obsahovat méně chyb.

Testování vývojářem

Ihned po vytvoření programového kódu, je tento kód prověřen programátorem. v praxi jsou tyto testy označovány jako „**Assembly tests**„. Většinou si však programátor netestuje svoji část kódu, ale realizuje se tzv. „test čtyř očí“. To znamená, že kód testuje jiný programátor, než ten který jej napsal. Program je v tomto stupni kontrolován na úrovni zdrojového kódu. V praxi je bohužel tento stupeň testování často podceňován. Přitom opravy chyby v této části testování software je nejméně nákladná.

Unit testy

dole

Integrační testy

V době, kdy je vývojář hotov se svými testy, přichází na řadu testovací tým. Integrační testy tedy nepřipravuje programátor, ale především testovací tým. Někdy bývají označovány jako „**testy vnitřní integrace**“. Musí být ověřena bezchybná komunikace mezi jednotlivými komponentami uvnitř aplikace. Integraci však lze ověřovat nejen mezi komponentami, ale také mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů. V této fázi se tak testuje integrace dosud jednotlivě ověřených částí. Integrační testy mohou být jak manuální, tak i automatizované.

Systémové testy

Během těchto testů je aplikace ověřována jako funkční celek. Tyto testy jsou používány v pozdějších fázích vývoje. Ověřují aplikaci z pohledu zákazníka. Podle připravených scénářů se simulují různé kroky, které v praxi mohou nastat. Obvykle probíhají v několika kolech. Nalezené chyby jsou opraveny a v dalších kolech jsou tyto opravy opět otestovány. Součástí této úrovně jsou jak funkční tak nefunkční testy. Poslední úroveň testů, které se provádějí před předáním produktu zákazníkovi, jsou tedy systémové testy. Tato úroveň testů tak většinou slouží jako výstupní kontrola softwaru. Systémové testování je obsaženo prakticky v každém procesu testování. Bez této úrovně by celé testování softwaru nemělo žádný význam. Bezporuchovost výsledného produktu by byla významně ohrožena. Proto tuto úroveň testů považuji za stěžejní v celém postupu testování software. Na realizaci těchto testů by se mělo myslet již v raném stádiu návrhu postupu testování, tak aby bylo možné obsahu testů co možná nejvíce přizpůsobit očekávanému softwaru.

Akceptační testy

**User acceptance test** (UAT) jsou akceptační testy na straně zákazníka. Pokud všechny předchozí etapy testů proběhly bez větších nedostatků, je možné předat aplikaci zákazníkovi. Ten si většinou se svým týmem testerů provede akceptační testy. Ty jsou často prováděny podle připravených scénářů, které společně připravil zákazník s dodavatelem. Testy probíhají na testovacím prostředí u zákazníka. Nalezené nesrovnalosti mezi aplikací a specifikací, jsou reportovány zpět vývojovému týmu. Opravené chyby jsou nasazeny na prostředí u zákazníka. V této úrovni je zřejmě nejdůležitější, definovat si předem jakou formou bude probíhat oznamování chyb od zákazníka a jak zabezpečit opravení těchto chyb v co možná nejkratší době.

**Dokumentace**

**Vnější** – nachází se mimo kód programu

**Dokumentace požadavků**

popis toho, co konkrétní software dělá nebo musí dělat.

Především se využívají při vývoji a slouží jako základní informace, pro to, co software má dělat. Dokumentace požadavků je výsledkem práce lidí, kteří se podílejí na výrobě softwaru: koncoví uživatelé, zákazníci, produktoví manažeři, projektoví manažeři, prodej, marketing, softwaroví architekti, inženýři, návrháři, vývojáři a lidé, co tyto softwary testují.

**Dokumentace architektury/designu**

Přehled softwaru. Zahrnuje vztahy k prostředí a stavebním základům, které budou použity v návrhu softwarových komponent. Zahrnuje model projektu.

**Technická dokumentace (vnitřní dokumentace)**

Technická dokumentace vzniká současně při tvorbě softwaru.

Samotný zdrojový kód je totiž nedostatečný.

Spolu s kódem musí být psán doprovodný text, který popisuje jednotlivé činnosti programu. Dokument musí být důkladný.

V C#

Umožňuje dokumentaci programu ve formátu XML.

To se nazývá dokumentační komentář.

Je umístěn před metodou, delegátem, atributem, třídou.

Začíná znaky “///“ do toho se píše popis dané metody, vlastnosti, třídy.

Programátor si může vytvořit vlastní značky jako <popis></popis >, ale je doporučeno pracovat s těmi, které jsou již připraveny a mají svůj význam.

Nástroj pro generování dokumentace se nazývá generátor dokumentace. Po vygenerování dokumentace vznikne soubor dokumentace, který může být čten nástrojem pro jeho zobrazení.

Značky

<param>

Používá se k popisu parametrů. Generátor dokumentace ověří, zda daný parametr existuje a zda jsou popsány všechny parametry v dokumentu. Pokud ne, bude vydáno upozornení.

<c>

Nastaví text dokumentace na font podobný kódu.

<code>

Slouží k napsání kódu programu jako ukázku. Např v metodě je list a místo toho aby bylo použito jen slovo list, které označuje jeho název instance. Bude <code>List<int> list = new List<int>();</code> a poté popis.

<example>

Ukázka zdrojového kódu.

<exception>

Slouží k dokumentaci vyjímek.

<include>

Umožňuje, zahrnou informace z externího dokumentu. Musí být ve formátu XML.

<list>

slouží k vytvoření seznamu nebo tabulky položek. Může obsahovat <listheader> pro definování hlavičky listu. Položky v něm jsou označovány <item> a ten se skládá <term> název položky a <description> popis položky.

<returns>

Používá se k popisu návratového typu metody

<summary>

Je to souhrn dokumentace.

**Uživatelská dokumentace**

Je psaná jednoduše, má za úkol obyčejnému uživateli jasně říct, co program dělá.

Dokumentace obsahuje popis všech funkcí programu a pomáhá uživateli při jejich používání, je stále aktualizována a psána tak, aby dokázala pomoci uživateli při řešení problémů.

**Marketingová dokumentace**

Jak prodávat produkt a analýza tržní poptávky. Dokumentace z pohledu marketingového oddělení

**Unit testování**

Testy se vždy píšou na základě **návrhu. Nikdy na základě implementace (to by test snadno prošel, i kdyby implementace nesplňovala zadání.)**

Testy reprezentují očekávanou funkčnost, která může být určena přímo zákazníkem nebo vývojářem pokud by zákazník neměl vědomosti k tomu, jak by metoda měla fungovat.

Unit testy testují metody tříd.

Ale ne všechny, třeba testy databáze jsou zbytečné vzhledem k použití databáze, která není součást programu, databázi testují akceptační testy.

Pravidla unit testů pro jejich nejlepší využití

* Testuje se podle specifikace, ne podle napsaného kódu.
* Testy by měly být na sobě nezávislé
* Testy by měly dopadnout vždy stejně, pokud nejsou nedeterministické jako náhodné číslo a výpočet pomocí datum.

Test má nějakou formu, je proto označen hranatými závorkami.

[TestClass]

označuje testovací scénář jedné třídy. Pokud jakýkoliv test ve třídě neprojde, tak celá TestClass je značena za neprošlý test.

[TestMethod]

označuje test metod, které jsou ve třídě. Tohle reprezentuje jednotlivé testy.

[TestInitialize]

Slouží jako pomocný konstruktor, který vytváří objekty k testům. Zavolá se před každým testem pro vytvoření nových prvků. Má na starosti aby testy byly na sobě nezávislé.

V unit testech musí být vždy výsledek true. K tomu se používá statická třída Assert a její metody.

Metody

AreEqual(prvek1, prvek2) může být i více prvků

Zjistí, jestli jsou si vložené prvky rovni.

AreNotEqual

Opak AreEqual

AreSame(prvek1, prvek2) může být i více prvků

Zjistí, jestli jsou prvky stejného datového typu

AreNotSame

Opak AreSame

IsTrue(podmínka)

Splněno pokud je podmínka splněna

IsFalse

Splněno pokud je podmínka nesplněna

Equals(prvek1, prvek2)

Pokud prvky implementují motody Equals, tak metoda zjistí jestli jsou prvky stejné.

IsNotNull(prvke)

Zjístí jestli referenční prvek není null

IsNull(prvek)

Opak IsNotNull