SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
MATERIÁLOVOTECHNOLOGICKÁ FAKULTA
SO SÍDLOM V TRNAVE

ČIASTKOVÝ NÁVRH SMERNICE PRE TESTOVANIE VEĽKÝCH SOFTVÉROVÝCH SYSTÉMOV

DIPLOMOVÁ PRÁCA

MTF-17392-57432

2011 **Bc. Martin Svoboda**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
MATERIÁLOVOTECHNOLOGICKÁ FAKULTA
SO SÍDLOM V TRNAVE

ČIASTKOVÝ NÁVRH SMERNICE PRE TESTOVANIE VEĽKÝCH SOFTVÉROVÝCH SYSTÉMOV

DIPLOMOVÁ PRÁCA

MTF-17392-57432

Študijný program: aplikovaná informatika a automatizácia v priemysle

Číslo a názov študijného odboru: 5.2.14 automatizácia, 9.2.9 aplikovaná informatika

Školiace pracovisko: Ústav aplikovanej informatiky, automatizácie a matematiky

Vedúci záverečnej práce/školiteľ: doc. Ing. Pavol Tanuška, PhD.

Trnava 2011 **Bc. Martin Svoboda**

Slovenská technická univerzita v Bratislave
Ústav aplikovanej informatiky, automatizácie a matematiky

Materiálovotechnologická fakulta so sídlom v Trnave
Akademický rok: 2010/2011
Evidenčné číslo: MTF-17392-57432

STU
MTF

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. Martin Svoboda**
ID študenta:	57432
Študijný program:	aplikovaná informatika a automatizácia v priemysle
Kombinácia študijných odborov:	5.2.14 automatizácia, 9.2.9 aplikovaná informatika
Vedúci práce:	doc. Ing. Pavol Tanuška, PhD.
Miesto vypracovania:	ÚIAM MTF STU v Trnave

Názov práce: **Čiastkový návrh smernice pre testovanie veľkých softvérových systémov**

Špecifikácia zadania:

1. Analyzujte problémovú oblasť.
2. Opíšte metódy a techniky používané v testovaní SW.
3. Navrhnite metodický postup testovania veľkých softvérových systémov s využitím UML.
4. Zhodnoťte navrhované riešenie.

Riešenie zadania práce od: 18. 02. 2011
Dátum odovzdania práce: 06. 05. 2011

Bc. Martin Svoboda
Študent

doc. Ing. Peter Schreiber, CSc.
Vedúci pracoviska

prof. Dr. Ing. Oliver Moravčík
Garant študijného programu

POĎAKOVANIE

Chcel by som sa poďakovať vedúcemu diplomovej práce doc. Ing. Pavlovi Tanuškovi, PhD., za jeho pomoc a usmernenie pri zostavovaní mojej práce.

SÚHRN

SVOBODA, Martin: *Čiastkový návrh smernice pre testovanie veľkých softvérových systémov.* [Diplomová práca] – Slovenská technická univerzita v Bratislave. Materiálovotechnologická fakulta so sídlom v Trnave; Ústav aplikovanej informatiky, automatizácie a matematiky. – Školiteľ: doc. Ing. Pavol Tanuška, PhD. – Trnava: MtF STU, 2011. 95 s.

Kľúčové slová: chyba, životný cyklus, testovanie softvéru, IEEE 829-2008, UML.

Diplomová práca sa venuje problematike testovania softvéru a jeho realizácii, ktorá si v praxi vyžaduje dôkladné plánovanie, prípravu a dokumentáciu. Cieľom práce je vytvoriť čiastkový návrh smernice pre testovanie veľkého softvérového systému. Prvá časť obsahuje teoretický prehľad a definície základných pojmov – chyba, životný cyklus, softvérový systém, testovanie softvéru, úloha testera a unifikovaný modelovací jazyk UML. Druhá časť práce je zameraná na zásady, techniky a metódy testovania softvéru. Tretia časť práce obsahuje návrh čiastkovej smernice testovania systému. Základom pre návrh testovacej dokumentácie (Test plán, Test report) bola norma IEEE 829-2008. Táto časť obsahuje aj postupy vykonania testov a reportovania chýb, ktoré sú znázornené pomocou diagramov v modelovacom jazyku UML. Záverečná časť práce hodnotí splnenie cieľov a význam návrhu smernice pre testovanie veľkých softvérových systémov.

ABSTRACT

SVOBODA, Martin: *Partial Suggestion Directive for Testing of Big Software Systems.* [Graduation Thesis] – Slovak University of Technology Bratislava. Faculty of Materials Science and Technology; Institute of Information Technology, Automation and Mathematics. – Supervisor: doc. Ing. Pavol Tanuška, PhD. – Trnava: MtF STU, 2011. 95p.

Key words: anomaly, life cycle, software testing, IEEE 829-2008, UML.

The graduation thesis deals with the field of software testing and its conducting which, in practice, requires detailed planning, preparation and documentation. The main aim of the thesis is to create partial suggestion directive for testing of big software systems. The first part contains theoretical overview and definitions of basic terms – anomaly, life cycle, software system, software testing, role of tester and unified modelling language UML. The second part of the thesis is focused at rules, techniques and methods of software testing. The third part of the thesis contains the partial suggestion directive for testing of big software system. The base for suggested test documentation (Test Plan, Test Report) is IEEE 829-2008 standard. This part also contains steps for conducting of the tests as well as steps for anomaly reporting which are both depicted in the form of diagram created in modelling language UML. The final part of the thesis evaluates the goal achievement and importance of partial suggestion directive for testing of big software systems.

OBSAH

ÚVOD ... 9
1 STAV RIEŠENEJ PROBLEMATIKY V LITERATÚRE ... 11
1.1 Chyba ... 11
1.1.1 Charakteristika pojmu chyba ... 11
1.1.2 Klasifikácia chýb ... 12
1.1.3 Najznámejšie softvérové chyby v histórii ... 13
1.1.4 Príčiny vzniku chýb ... 15
1.2 Životný cyklus a jeho modely ... 17
1.2.1 Charakteristika pojmu životný cyklus softvéru ... 18
1.2.2 Modely životného cyklu softvéru ... 19
1.2.2.1 Kaskádový model ... 20
1.2.2.2 Špirálový model ... 21
1.2.2.3 Concurrent engineering model ... 24
1.2.3 Výskyt chýb počas životného cyklu softvéru ... 25
1.3 Testovanie ... 26
1.3.1 Charakteristika pojmu testovanie ... 26
1.3.2 História testovania ... 27
1.3.3 Typy testovania ... 28
1.3.3.1 Funkčné testovanie ... 28
1.3.3.2 Užívateľské testovanie ... 29
1.3.3.3 Výkonnostné testovanie ... 29
1.4 Úloha a výber testera ... 30
1.5 Veľké softvérové systémy ... 31
1.6 UML – Unifikovaný modelovací jazyk ... 32
1.6.1 História vzniku UML ... 33
1.6.2 Štruktúra jazyka UML ... 34
1.6.2.1 Stavebné bloky ... 35
1.6.2.2 Mechanizmy ... 37
1.6.2.3 Architektúra ... 37
1.6.3 Produkty pre prácu s UML ... 39
1.7 Normy pre testovanie softvéru ... 39
1.7.1 Norma IEEE 829-2008 ... 40
1.7.1.1 Definície základných pojmov ... 41
1.7.1.2 Použitie normy IEEE 829-2008 ... 42
2 TECHNIKY A METÓDY TESTOVANIA ... 44
2.1 Rozdelenie testovania ... 44
2.2 Zásady testovania ... 46
2.2.1 Zásada 1 ... 47
2.2.2 Zásada 2 ... 47
2.2.3 Zásada 3 ... 48
2.2.4 Zásada 4 ... 48
2.2.5 Zásada 5 ... 48
2.2.6 Zásada 6 ... 49

2.3 Techniky testovania ... 50
2.3.1 Techniky funkčného testovania ... 50
2.3.1.1 Rozdeľovanie tried ekvivalencie (RTE) ... 51
2.3.1.2 Analýza okrajových hodnôt (AOH) ... 52
2.3.1.3 Kombinatorická analýza ... 53
2.3.2 Techniky štrukturálneho testovania ... 54
2.3.2.1 Testovanie blokov ... 54
2.3.2.2 Testovanie rozhodnutí ... 55
2.3.2.3 Testovanie podmienok ... 55
2.3.2.4 Testovanie základných ciest ... 55
2.4 Metódy testovania ... 57
2.4.1 Metóda FURPS ... 57
2.4.2 V - model testovanie ... 59
2.4.3 Vývoj riadený testami (TDD) ... 62
2.4.3.1 Testovanie softvéru pri TDD ... 64
2.4.4 Metóda SEARCH ... 65
3 NÁVRH METODICKÉHO POSTUPU TESTOVANIA SOFTVÉRU S VYUŽITÍM UML ... 67
3.1 Plánovanie testovania ... 67
3.1.1 Dokumentácia testovania ... 69
3.2 Čiastkový návrh základnej štruktúry smernice testovania ... 70
3.2.1 Zostavenie Test plánu ... 71
3.2.1.1 Testovací prípad ... 76
3.2.1.2 Testovací skript ... 77
3.2.1.3 Testovací scenár ... 77
3.2.2 Testovanie ... 78
3.2.2.1 Automatizované testovanie ... 80
3.2.2.2 Záťažové testovanie ... 82
3.2.3 Analýza a Oprava chýb ... 86
3.2.4 Zostavenie Test reportu ... 88
ZÁVER ... 91
ZOZNAM BIBLIOGRAFICKÝCH ODKAZOV ... 93

ÚVOD

Testovanie sprevádzalo a sprevádza vývoj počítačov ako aj softvéru, s ktorým počítače pracujú. Stále modernejšie a komplikovanejšie softvéry a softvérové systémy vyžadujú čoraz dokonalejšie a sofistikovanejšie metódy testovania, ktorých cieľom je odhaliť chyby softvéru a predísť tak ich zlyhaniu alebo nesprávnemu fungovaniu v situáciách, ktoré môžu ohroziť ľudský život, bezpečnosť prevádzky, či výsledky finančne náročných vedeckých pokusov, projektov, či expedícií. Aj zdanlivo malá chyba, ktorá nebola v dôsledku nedôkladného testovania detekovaná, môže mať dopad na spoločnosť, ktorá softvér používa, alebo ho poskytuje, v podobe poškodenia dobrého mena firmy, straty zisku, zvýšených nákladov na odstránenie chyby či odškodné pre poškodených klientov a samozrejme zhoršenia pozície na trhu. Zo štúdie The Economic Impacts of Inadequate Infrastructure for Software Testing za rok 2002 (Ekonomické dopady nevyhovujúcej infraštruktúry na testovanie softvéru), uskutočnenej americkým Národným inštitútom pre štandardy a technológie, vyplýva, že len v Spojených štátoch malo používanie chybných aplikácií s nedostatočným zabezpečením kvality za následok zvýšenie nákladov o 59,5 miliárd USD.

V minulosti sa firmy snažili chyby v softvéri zatajiť, dúfajúc, že pravdepodobnosť odhalenia chyby klientom je taká malá, že si nevyžaduje ďalšie opatrenia. Známy je prípad firmy Intel, ktorá ešte pred uvedením procesora Pentium na trh zistila, že chybný čip spôsobuje v niektorých prípadoch nesprávny výsledok pri delení desatinných čísel. Napriek tomu však procesor nestiahla z predaja a čip vymenila iba klientom, ktorý preukázali ujmu spôsobenú touto chybou. Napokon sa pod tlakom verejnej mienky Intel verejne ospravedlnil a vyčlenil 400 miliónov dolárov na výmenu chybných čipov. Podľa štúdie realizovanej agentúrou Forrester Group a Compuware takmer dve tretiny (64 %) opýtaných firiem utrpí podstatnú stratu tržieb v dôsledku výpadku aplikácií [10].

V súčasnosti sa postoj softvérových spoločností k chybám a testovaniu zmenil. Predovšetkým sa softvér testuje nielen počas vývoja, ale aj po jeho ukončení. Detekované chyby sú okamžite odstraňované bez ohľadu na ich závažnosť, prípadné chyby, ktoré sú detekované po implementácii softvéru sú publikované na webových stránkach a výrobcovia softvéru využívajú spätnú väzbu od klientov či spotrebiteľov, ktorí majú možnosť vyjadriť svoj názor a skúsenosti s používaním softvéru v diskusných fórach na Internete.

Aj keď v súčasnosti neexistuje vedná disciplína, ktorá by sa zaoberala výlučne testovaním softvéru a jeho metódami, rozsiahle využitie veľkých softvérových systémov v praxi (on-line rezervácie, kontrola leteckej dopravy, telekomunikačné siete, prehliadače webových stránok) vyžaduje prepracovaný spôsob testovania a prevencie. Spoločnosti, ktoré považujú testovanie kvality za pridanú hodnotu a ktoré zavádzajú rozsiahly proces jeho zabezpečenia, zlepšujú kvalitu svojich aplikácií a ušetria pri testovaní, implementácii a nasadzovaní aplikácií do praxe. Okrem toho budú uvádzať svoje aplikácie rýchlejšie na trh, čo im prinesie významné konkurenčné výhody.

Prepracovanosť postupu a metódy testovania majú v súčasnosti podobu štandardizovaných noriem s medzinárodnou platnosťou, ktoré sú bázou pre aplikáciu pri tvorbe smerníc testovania, ktoré sú nevyhnutné pre tímovú prácu testerov a zainteresovaných subjektov.

1 STAV RIEŠENEJ PROBLEMATIKY V LITERATÚRE

1.1 Chyba

1.1.1 Charakteristika pojmu chyba

Vo všeobecnosti sa testovanie softvéru považuje za vyhľadávanie chýb. Preto je potrebné chybu definovať z tohto aspektu. Okrem samotného pojmu chyba sa v súvislosti s testovaním hovorí aj o odchýlke, probléme, anomálii, nekonzistencii, zlyhaní či nedostatku.

Patton [1] definuje softvérovú chybu pomocou *špecifikácie produktu*, alebo skrátene pomocou *špecifikácie*. Špecifikácia definuje samotný produkt (softvér), ktorý bude vytvorený, podrobne popisuje ako bude vyzerať, ako sa bude správať, čo bude, ale aj čo nebude robiť. Potom podľa Pattona [1] o softvérovej chybe hovoríme, ak je splnená aspoň jedna z nasledovných podmienok:

- Softvér nevykonáva niečo, čo by podľa špecifikácie mal vykonávať.
- Softvér vykonáva niečo, čo by podľa špecifikácie produktu vykonávať nemal.
- Softvér vykonáva niečo, o čom špecifikácia produktu nehovorí.
- Softvér nevykonáva niečo, o čom špecifikácia nehovorí, ale mala by hovoriť.
- Softvér je ťažko zrozumiteľný, ťažko sa s ním pracuje, je pomalý, alebo podľa názoru testera ho konečný používateľ nebude považovať za správny.

Definícia softvérovej chyby bez špecifikácie produktu uvedená v inštrukčnej príručke IBM [3] znie: Chyba je čokoľvek, ohľadom programu, čo podľa niektorého zo zainteresovaných zbytočne znižuje kvalitu programu.

Paleta [12] definuje chybu z pohľadu užívateľa a z pohľadu programátora. Z pohľadu vývojového tímu je chybou to, keď sa program nespráva podľa zadania, resp. chybou je všetko čo sa odlišuje od analýzy podpísanej a schválenej zákazníkom. Z pohľadu užívateľa je chybou všetko, čo nejakým spôsobom vnímajú negatívne (chýba dôležitý údaj, program je na ich počítači príliš pomalý, nezrozumiteľné užívateľské rozhranie).

Robbins [18] definuje chybu v programe ako *„čokoľvek čo poškodzuje užívateľa“*.

1.1.2 Klasifikácia chýb

Pri testovaní softvéru je užitočné poznať s akými typmi chýb sa tester môže stretnúť, prípadne poznať ich možné zdroje či úroveň závažnosti.

Paleta [12] rozlišuje štyri základné typy chýb softvérového projektu:

- Prvý typ - programátorské chyby (preklep programátora, nesprávny algoritmus, nesprávne zapísaný alebo nezdokumentovaný kód, nesprávna metodika, časové obmedzenie).
- Druhý typ - nedostatky v zadaní (nepriama alebo nepresná komunikácia medzi zadávateľom a dodávateľom).
- Tretí typ - technologické nedostatky (výkonnostné problémy aplikácie).
- Štvrtý typ – ostatné (tieto chyby sa nachádzajú na rozhraní hore uvedených kategórií).

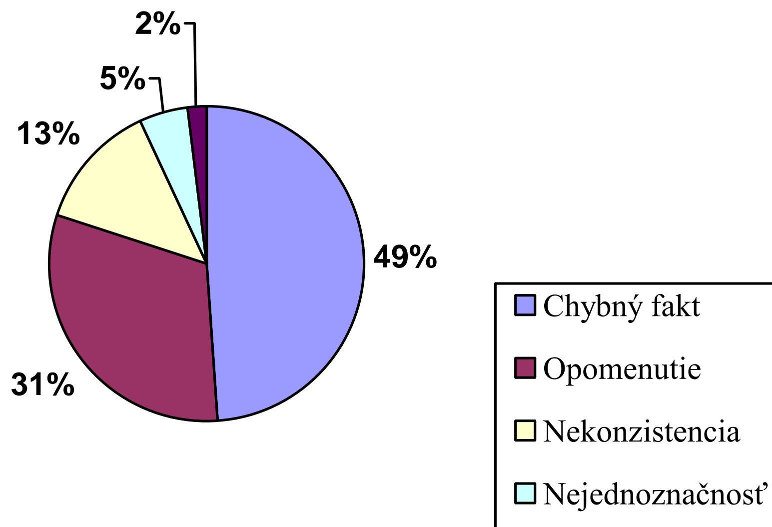
Robbins [18] klasifikuje chyby do nasledovných kategórií:

- nekonzistentné užívateľské rozhranie,
- nesplnené očakávania,
- zlá výkonnosť,
- kolízia alebo poškodenie dát.

Softvérové chyby sú podľa Tanušku a Schreibera [2] klasifikované „*typom chyby, miestom vzniku, časom vzniku, úrovňou vážnosti, frekvenciou vznikania a nákladmi. Individuálne chyby potom môžu byť agregované podľa nasledujúceho prístupu:*

- ***Nedostatočné podriadenie sa štandardom*** *– problémom je, že softvérové funkcie, dátové reprezentácie, preklady alebo interpretácie neboli prispôsobené procedurálnym procesom alebo formátu, ktorý špecifikuje štandard.*
- ***Nedostatočná prevádzkyschopnosť s inými produktmi*** *– problémom je neschopnosť softvérových produktov vymieňať a zdieľať dáta s inými systémami.*
- ***Slabá výkonnosť*** *– problémom je, že aplikácie síce pracujú, ale nie tak ako by sa očakávalo a ako je žiadané.*“

Faigl [4] uvádza ako základné druhy chýb: chybný fakt, pozabudnutie, nekonzistenciu, nejednoznačnosť a chybnú požiadavku. Všetky uvedené druhy chýb sú graficky znázornené na Obr.1.



Obr. 1: Percentuálny podiel zdrojov chýb pri vývoji softvéru podľa Faigla [4]

Zaujímavé je, s akými druhmi chýb sa môžeme stretnúť a z čoho tieto chyby pramenia. Najčastejším neduhom je zrejme jednoducho „chybný fakt“, ktorý vyplynul z analýzy problému, nech už k tomu došlo pri akejkoľvek činnosti. Sem patrí aj celý rad bezpečnostných chýb. Jednoducho povedané – zle odhadneme problém. Ďalšou veľmi častou chybou je „opomenutie“ a občas vo veľmi tesnom závese, obzvlášť keď sa projekt už chýli ku konci, sú objavované „nekonzistencie“. „Nejednoznačnosti“ a „chybné požiadavky“ naopak zaberajú z celkového počtu chýb len malú, ale aj tak neprehliadnuteľnú časť.

1.1.3 Najznámejšie softvérové chyby v histórii

Prípad vesmírnej sondy Mariner 1 – nesprávny prepis programu z papiera do počítača spôsobil v roku 1962 odchýlenie sondy od stanoveného kurzu, ktorá musela byť v dôsledku tejto chyby zničená za letu.

Prípad Therac-25 – rádioterapeutický prístroj Therac-25 zapríčinil v rokoch 1985 až 1987 smrť najmenej piatich pacientov v dôsledku neprimeranej dávky žiarenia. Softvérová chyba zapríčinila po viacnásobnom rýchlom stláčaní tlačidiel nastavenie hodnoty žiarenia na smrteľnú dávku.

Prípad raketového systému Patriot – jeho prvé nasadenie v roku 1991 vo vojne v Perzskom zálive bolo sprevádzané veľkým mediálnym ohlasom. Išlo o ochranu proti

raketám Scad, ktoré ohrozovali amerických vojakov. Systém ale nedokázal zastaviť niekoľko rakiet Scad a jedna z nich v Dhahrane usmrtila 28 amerických vojakov. Neskôr sa zistilo, že systém Patriot obsahoval chybu v softvéri. Po približne 14 hodinách prevádzky systému, časť softvéru na navádzanie rakety začala byť nepresná. Chyba bola lokalizovaná v časovaní systémových hodín, kde s narastajúcou dobou prevádzky zariadenia sa postupne zhoršovalo zameriavanie. Pri incidente v Dharhane bol systém Patriot v prevádzke nepretržite 100 hodín.

Prípad procesora Intel Pentium z roku 1993 – poškodil dobré meno spoločnosti Intel, hoci reálne zasiahla len malú časť jeho užívateľov. V chybnom procesore problém vznikal pri výpočtoch čísiel s plávajúcou desatinnou čiarkou v určitom rozsahu, ak mal napríklad procesor vydeliť číslo 4195835,0 číslom 3145727,0, ohlásil výsledok 1,33374 namiesto správneho výsledku 1,33382.

Prípad rakety Ariane 5 z roku 1996 – softvérový inžinieri použili softvér zo staršieho modelu Ariane 4. Rýchlejší počítač v Ariane 5 sa však pri práci so 64-bitovými číslami správal odlišne, výsledkom čoho bolo pretečenie buffera pri štarte v roku 1996. To zapríčinilo výpadok počítača, ktorý riadil motory, a 40 sekúnd po štarte došlo k explózií rakety.

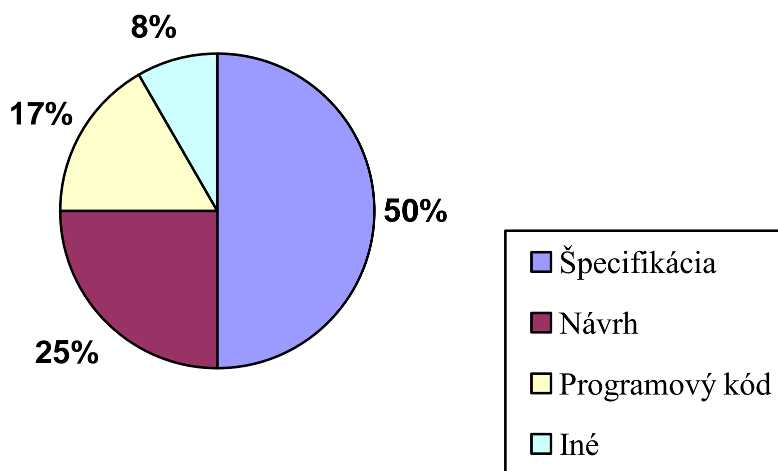
Prípad vesmírnej sondy Mars Polar Lander z roku 1999 – riadiaci počítač pri pristávacom manévri nad povrchom planéty nesprávne vyhodnotil údaje a došiel k záveru, že sonda už bezpečne pristála. Dôsledkom toho bolo vypnutie brzdiacich motorov a zrútenie sa voľným pádom z výšky 1800 metrov na povrch planéty. Chyba v riadiacom softvéri nebola odhalená pre zlé testovanie softvéru. Vývojové tímy síce otestovali im zverené úlohy, ale neotestovali zariadenie ako celok. Dva inak samostatne fungujúce časti softvéru po spojení do celku tak spôsobili jednu z najznámejších softvérových katastrof.

Prípad z Panamy z roku 2000 – mal spoločného menovateľa s prípadom Therac-25 z roku 1985. Miestni lekári objavili softvérovú chybu, ktorú chceli využiť vo svoj prospech. Rádioterapeutický prístroj, umožňoval pri ožarovaní použitie maximálne štyroch ochranných platní, namiesto požadovaných päť. Toto obmedzenie sa dalo obísť nakreslením jednej platne s dierou uprostred. Softvér v tomto prípade z neznámych dôvodov vypočítal dávku na päť ochranných platní. Bohužiaľ softvér reagoval aj na smer kreslenia diery uprostred platne. Jedným smerom síce bola dávka vyrátaná správne, ale pri nakreslení opačným smerom riadiaci softvér dávku zdvojnásobil. Na túto softvérovú chybu doplatilo životom 8 pacientov.

Na mnohých príkladoch sa dá ukázať aké následky môže mať zlyhanie softvéru [5]. V kritických prípadoch to môže byť aj katastrofa s následkami na životoch, ale v bežnom živote sú to udalosti, ktoré nám do určitej miery len znepríjemňujú život.

1.1.4 Príčiny vzniku chýb

Bolo vykonaných mnoho štúdií o vzniku softvérových chýb. Do úvahy sa brali programátorské projekty všetkých možných veľkostí. Všetky projekty od najmenších až po extrémne veľké skončili s rovnakým výsledkom. Príčinou vzniku najväčšieho počtu chýb nie sú omyly pri programovaní, ale zlá špecifikácia.



Obr. 2: Percentuálny podiel zdrojov chýb pri vývoji softvéru podľa Sochora [6]

Podľa Pattona [1] to, že hlavnou príčinou chýb je skutočne zlá špecifikácia, má niekoľko dôvodov. Pri vývoji softvéru sa v mnohých prípadoch totiž špecifikácia ani vôbec nenapíše. Ďalej sa často stáva, že špecifikácia nie je dostatočne podrobná, prípadne sa neustále mení, alebo sa s ňou dostatočne neoboznámili všetci členovia vývojového tímu. Dôkladné plánovanie vývoja softvéru je životne dôležité a pokiaľ sa nevykonáva správne, tak sa do softvéru zanášajú chyby.

V poradí druhým najväčším zdrojom chýb je návrh. V ňom programátori popisujú svoj plán vyvíjaného softvéru. Samotný návrh môžeme prirovnať k výkresom budovy, ktoré nakreslil architekt. Aj pri tvorbe návrhu dochádza k chybám z rovnakých dôvodov

ako u špecifikácií. V praxi to môže byť narýchlo vypracovaný návrh, často sa meniaci návrh, alebo málo prediskutovaný návrh.

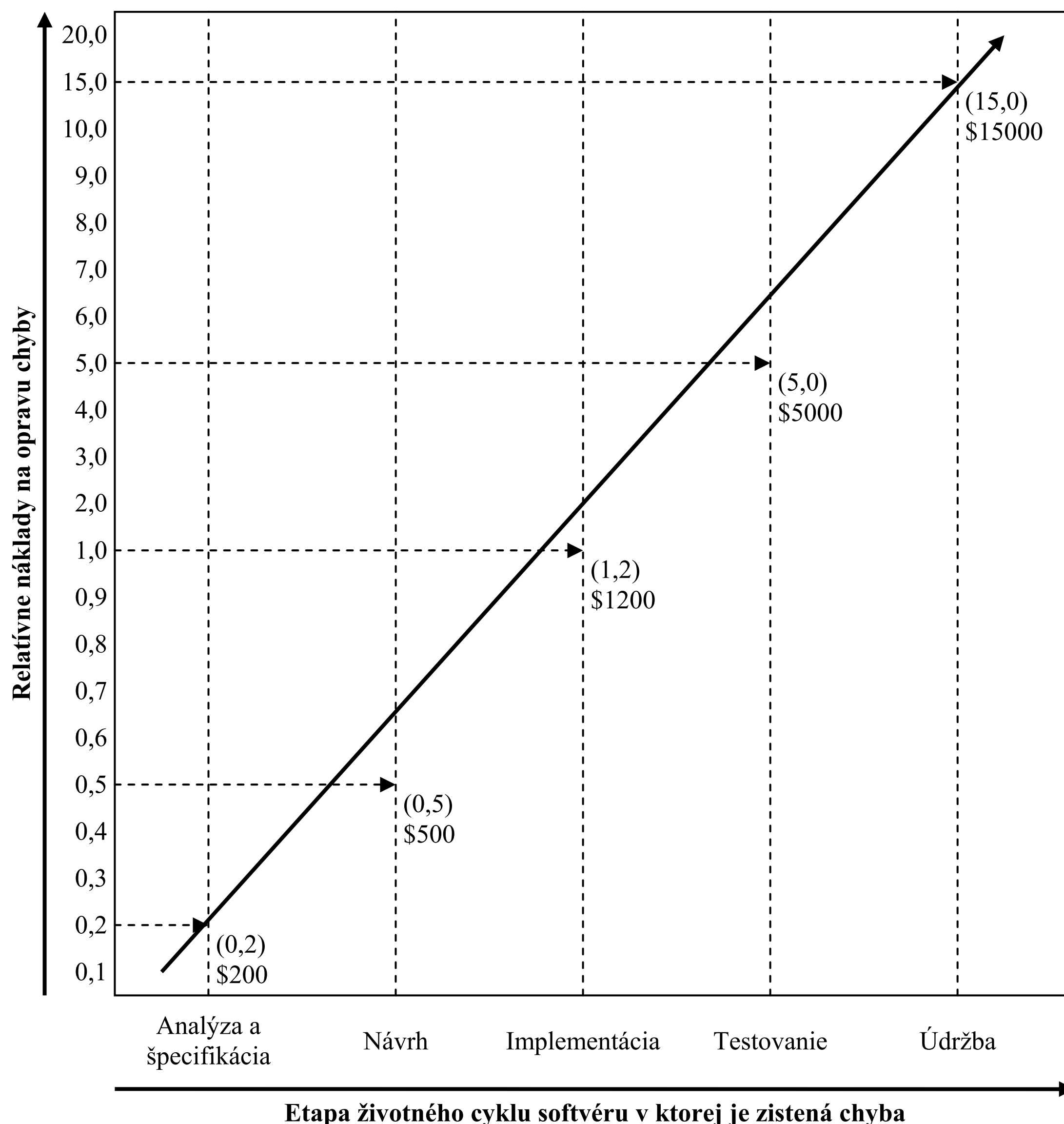
Ďalším zdrojom chýb v poradí sú chyby v programovom kóde. Príčinou týchto chýb môže byť zložitosť softvéru, nedostatočná dokumentácia (hlavne kódu), časová tieseň v termínoch a samozrejme aj chyby v kóde.

Do kategórie iné príčiny patrí všetko, čo nespadá do predošlých skupín. Za niektoré chyby môžu nesprávne predpoklady, niekde môžu vzniknúť opakované chyby alebo viac chýb, ktoré sú výsledkom jednej spoločnej príčiny.

Náklady na odstránenie chyby rastú s postupom času, pričom ich výška podľa Perryho [28] exponenciálne rastie. To znamená, že postupom času nám náklady stúpajú o desať násobok. Najdrahšie sú chyby, ktoré nájde až zákazník. Tu hrozí, že náklady na opravy môžu spotrebovať aj celkový zisk z predaja softvérového produktu. Uvedenú skutočnosť ilustruje Obr.3.

Paleta [12] uvádza šesť najčastejších príčin neodhalených chýb v softvéri:

1. Podcenenie zložitosti projektu – vo väčšine prípadov sú prvé odhady zložitosti projektu výrazne podcenené. Z toho dôvodu je potrebné vytvoriť rozumne veľkú rezervu na nepredvídateľné udalosti.
2. Snaha o dokončenie v termíne za každú cenu – testy sú zvyčajne prvou obeťou šetrenia času, ku ktorému dochádza zo snahy dodržať termín odovzdania projektu.
3. Použitie úplne nových technológií – prvé verzie noviniek v oblasti technológií môžu mať problémy s výkonnosťou, môžu obsahovať chyby a nemusia byť kompatibilné s používaným hardvérom a softvérom, čím prinášajú nové slabé miesta do vyvíjaného softvéru.
4. Nedostatočná kvalifikácia a neskúsenosť členov vývojového tímu.
5. Zlá komunikácia medzi zadávateľom a užívateľom – projekt vývoja softvéru by mal byť riadený tak, aby sa s klientom komunikovalo čo najviac a predišlo sa prípadným nedorozumeniam.
6. Podcenenie analýzy a požiadaviek – vznikne ak zákazník podcení spoluprácu na projekte a o svojich požiadavkách začne komunikovať neskoro.



Obr. 3: Približné relatívne náklady na zistenie a opravu chyby v danej etape životného cyklu softvéru ako ich uvádza Perry [28]

1.2 Životný cyklus a jeho modely

Koncom 60. rokov minulého storočia bol vývoj softvéru nesmierne náročný a aj keď sa mu venovali tí najlepší odborníci, neprinášal vždy želateľné výsledky. Dôvod bol jednoduchý: neexistoval žiadny formalizovaný postup, ktorý by hovoril o tom, ako sa má systém vytvárať. Preto začali vznikať prvé pravidlá a metodiky práce, ktorými sa má proces vývoja softvéru riadiť a objavili sa aj prvé modely, ktoré ich popisovali na teoretickej úrovni.

Medzi základné modely v tejto oblasti patrí model vývoja životného cyklu softvéru. V nasledujúcom období metodiky odrážali aktuálnu situáciu a snažili sa prispôsobiť vývoj softvéru konkrétnym požiadavkám danej doby, čím sa vývoj softvérových systémov stal prehľadnejším a kontrolovateľnejším.

1.2.1 Charakteristika pojmu životný cyklus softvéru

Nakoľko testovanie softvéru patrí medzi základné etapy životného cyklu softvéru, a pretože v súčasnosti sa kladie dôraz na testovanie softvéru počas celého životného cyklu, a nie len v jeho finálnej fáze vývoja, je nutné venovať pozornosť životnému cyklu ako celku.

Moravčík, Vaský, Mišút [11] definujú životný cyklus softvéru na základe jednotlivých fáz, ktoré ho tvoria. Proces vývoja softvéru, údržbu a starostlivosť o softvérový produkt rozčleňujú do časových a fenomenologických fáz.

Patton [1] definuje životný cyklus softvéru ako proces, podľa ktorého sa vytvára softvérový produkt od jeho prvotného zámeru až po jeho uvedenie na trh.

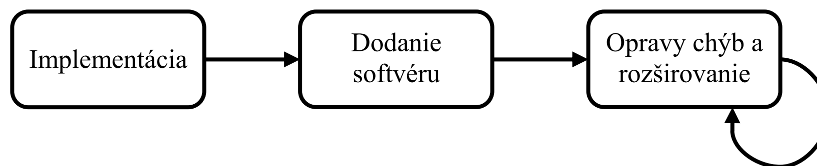
Kadlec [14] zaraďuje životný cyklus k nástrojom vývoja softvér. V tejto súvislosti používa synonymické pojmy ako „metodika“, a „proces vývoja softvéru“. Týmito pojmami označuje komplexné postupy a návody na vývoj softvérovej aplikácie.

Bieliková [15] definuje životný cyklus programu v rámci softvérového inžinierstva, kde je definovaný ako súbor definovaných etáp vývoja programového systému. *„Podstatnou charakteristikou každého modelu životného cyklu však je, že definuje jednotlivé etapy a pre každú z nich činnosti, ktoré sa majú vykonať, rovnako ako vstupy a výstupy etapy.“*

V súčasnosti sa používa niekoľko modelov životného cyklu vývoja softvéru. Niektoré sú formálne, so zložitou štruktúrou, iné sú naopak flexibilné. V praxi to znamená, že testeri sledujú a testujú softvér v priebehu celého životného cyklu podľa modelu, ktorý ich tím používa. Podľa neho vždy presne vedia, v akej etape vývoja sa softvér nachádza, čo napomáha procesu plánovania, samotnej realizácii ako aj uvedomenia si vlastnej úlohy testera, čo je kľúčovým predpokladom úspešného vývoja softvérového produktu.

1.2.2 Modely životného cyklu softvéru

Medzi najstaršie modely životného cyklu softvéru, ktoré sa objavili v 50. rokoch minulého storočia, patrí tzv. **model vytvor a oprav**. Tento model (pozri Obr.4) je príkladom nesystematického prístupu, nakoľko spočíval vo vytvorení aplikácie, jej uvedenia do prevádzky a oprave chýb. S týmto modelom sa môžeme stretnúť v praxi aj dnes, nakoľko je vhodný pre malé projekty, napr. prototypy a demonštračné programy. Vyžaduje minimálne prevádzkové náklady a malú dokumentáciu. Podľa Pattona [1] sa pri použití tohto modelu najskôr podľa hrubých predstáv vytvorí jednoduchý návrh, ktorý potom vstúpi do dlhého cyklu programovania, testovania a opravovania chýb, ktorý sa mnohonásobne opakuje.



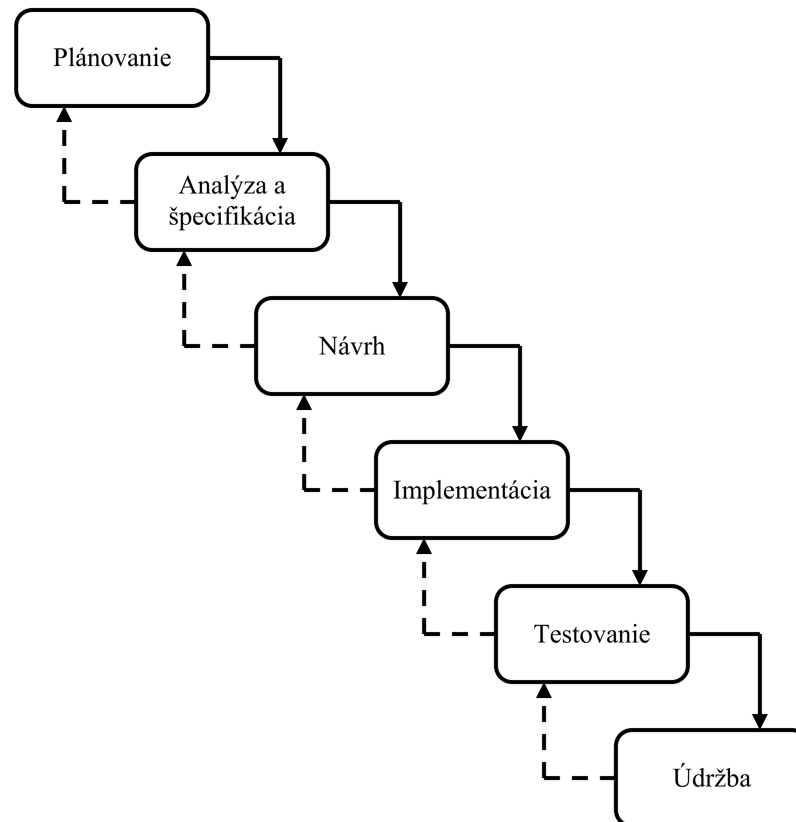
Obr. 4: Model životného cyklu softvéru „vytvor a oprav"

Nakoľko model vytvor a oprav nebol vhodný pre rozsiahlejšie programové systémy, bolo zrejmé, že vývoj softvéru sa bude uberať iným smerom. Aj preto sa už v roku 1957 objavil tzv. **stagewise model** životného cyklu, ktorý rozdelil vývoj softvéru na niekoľko etáp (definícia problému, špecifikácia požiadaviek, architektúra a návrh, implementácia, integrácia, prevádzka). Základným nedostatkom tohto modelu bola absencia spätnej väzby. Vývoj softvéru postupoval iba smerom vpred, návrat do niektorej z predchádzajúcich etáp neprichádzal do úvahy. Až po dodaní aplikácie bolo možné v rámci revalidácie urobiť krok späť a opätovne vstúpiť do procesu.

V nasledujúcom období boli vytvorené ďalšie modely, pričom ich vznik vždy súvisel s pokrokom v oblasti samotného softvéru a s narastajúcimi nárokmi a požiadavkami na proces jeho vývoja. Medzi najčastejšie uvádzané modely v literatúre patrí **kaskádový model** a **špirálový model**.

1.2.2.1 Kaskádový model

Dôsledkom vzniku softvérového inžinierstva a zvýšených nárokov na vývoj softvéru bol **kaskádový model**, ktorý v roku 1970 definoval Dr. Winston Royce. Od stagewise modelu sa odlišuje najmä zavedením spätnej väzby, čo znamená, že na konci každej fázy sa uskutoční jej hodnotenie a prípadné prepracovanie alebo oprava. To znamená, že bol možný návrat späť, v prípade, že stav vývoja softvéru nezodpovedal požiadavkám. Charakteristickým znakom tohto modelu je sekvenčnosť jednotlivých etáp (pozri Obr.5). To znamená, že každá nasledujúca etapa sa vykoná až po dokončení etapy predchádzajúcej. Projekt podľa kaskádového modelu postupuje smerom „dole“ v postupnosti krokov, ktoré vedú od prvotnej myšlienky až k výslednému produktu. Kaskádový model je príkladom striktného modelu, ktorý nepredpisuje žiadne vedľajšie kroky ani priebežnú komunikáciu so zákazníkom, či vytváranie prototypov.



Obr. 5: Kaskádový model životného cyklu softvéru

Kadlec [14] uvádza nasledovných šesť etáp kaskádového modelu:

1. Definícia problému, spoznanie zákazníka, preniknutie do cieľovej oblasti.
2. Analýza a špecifikácia požiadaviek.
3. Návrh systému.
4. Implementácia systému.
5. Integrácia a testovanie systému.
6. Prevádzka a údržba.

Podľa Pattona [1] má tento model výhodu oproti ostatným modelom v tom, že sa vyznačuje vysokou dôkladnosťou a prepracovanosťou. Nevýhoda spočíva v tom, že testovanie prebieha až na konci vývojového cyklu.

Podľa Bielikovej [15] výhoda kaskádového modelu spočíva v transparentnosti procesu vývoja a vo výstupoch, ktorými končí každá etapa. Nevýhodou je, že zákazník vidí produkt až na konci procesu, neskoré odhalenie nedostatkov, malá prispôsobivosť zmenám počas procesu vývoja, reálne projekty nedodržujú poradie jednotlivých krokov.

Podľa Kadleca [14] je výhodou tohto modelu jeho jednoduchosť, systematickosť, ktorá vnáša do celého procesu disciplínu, úplná dokumentácia, ktorá uľahčuje údržbu a prípadné opravy. Medzi nevýhody patrí jeho nepružnosť, dodanie zákazníkovi formou „veľkého tresku" (so zákazníkom komunikuje na začiatku a potom až na konci vývoja), čo prináša celý rad ďalších nevýhod.

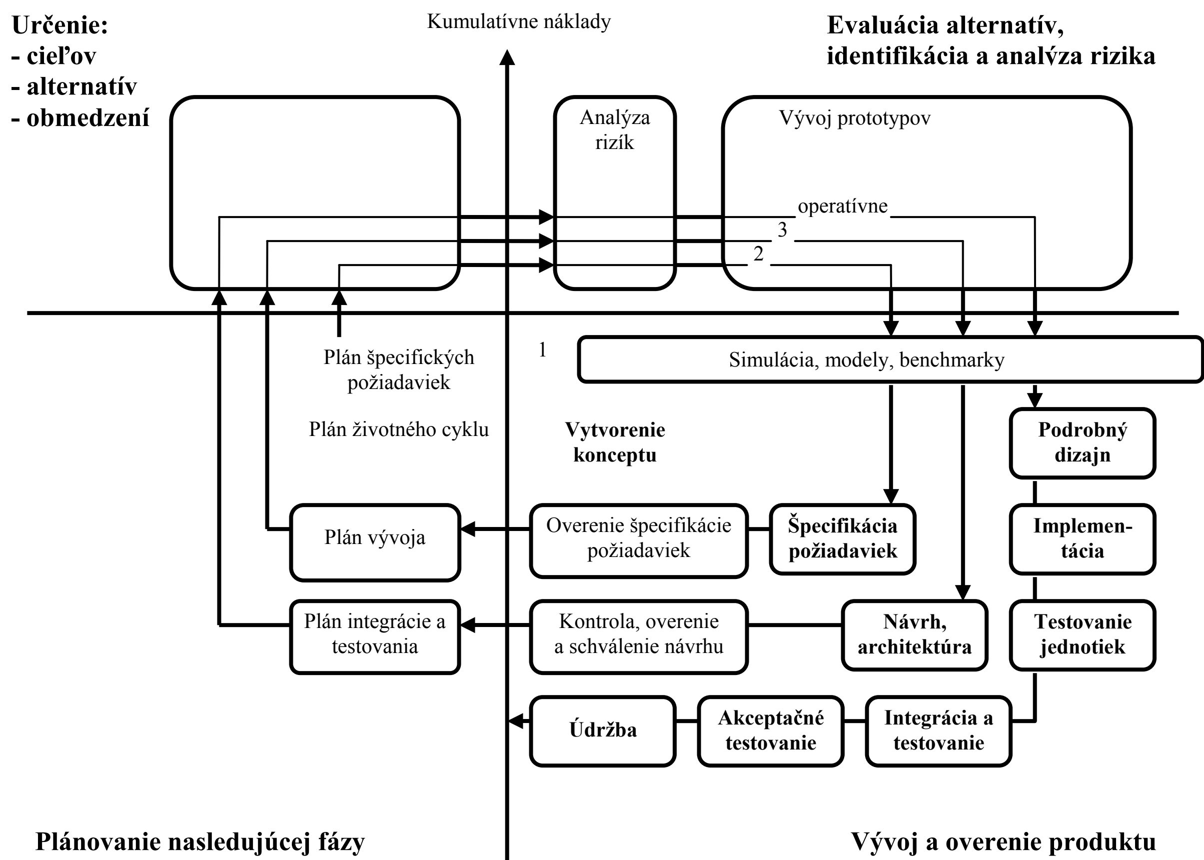
Pre vývoj softvéru, na ktorý je tento model vhodný, je ideálny. Nie je však vhodný na vývoj bežných softvérových systémov, vrátane webových aplikácií. K nevýhodám tohto modelu patrí aj istá nepružnosť (dlhé dodacie termíny), ale tento model sa stále vyvíja, vylepšuje a upravuje.

1.2.2.2 Špirálový model

V 80. rokoch minulého storočia bol zavedený špirálový model životného cyklu softvéru. Nielen že riešil nedostatky predchádzajúcich modelov, ale priniesol do procesu vývoja softvéru efektívnosť, čo viedlo k jeho častému používaniu v praxi.

Autorom tohto modelu je Barry Boehm, ktorý ho po prvý krát definoval v roku 1986. Vychádzal z niekoľkých modelov tej doby, ktoré postupne prestávali vyhovovať rastúcej zložitosti a zväčšujúcemu sa rozsahu softvérových projektov.

Základná štruktúra modelu (pozri Obr.6) pozostáva zo sekvencií cyklov rozmiestnených do štyroch kvadrantov.



Obr. 6: Špirálový model životného cyklu softvéru

Špirála znázorňuje iteratívny proces, ktorý postupuje zo stredu smerom von. Každý závit špirály reprezentuje jeden iteratívny cyklus s tým istým počtom krokov. Radiálny rozmer špirály reprezentuje časové a finančné náklady na vývoj softvéru. Každý zo štyroch kvadrantov predstavuje jednu zo štyroch hlavných etáp vývoja softvéru:

1. Určenie cieľov, alternatív a obmedzení – identifikácia a nastavenie konkrétnych cieľov pre súčasnú etapu projektu.
2. Evaluácia alternatív, identifikácia a analýza rizika – vyhodnotenie kľúčových rizík a vytvorenie krízových plánov.
3. Vývoj a overenie produktu – vlastná práca na projekte (špecifikácia požiadaviek, návrh, vývoj, testovanie).
4. Plánovanie nasledujúcej etapy – projekt je vyhodnotený a začína sa plánovanie ďalšieho cyklu špirály.

Každá špirála cyklu má svoj význam:

1. Prvý cyklus – má za úlohu nájsť globálne riziká, ktoré by mohli ohroziť vývoj softvéru.
2. Druhý cyklus – slúži na vytváranie a overovanie špecifikácií požiadaviek na softvér.
3. Tretí cyklus – je zameraný na vytvorenie a overenie detailného dizajnu softvéru.
4. Štvrtý cyklus – týka sa implementácie, testovania a integrácie.

Praktické využitie špirálového modelu uvádzajú Page a kol. [16] vo svojej knihe „Jak testuje software Microsoft". Softvérové tímy na začiatku naplánujú, navrhnú a vytvoria prototyp produktu. Potom tím získa údaje spätnou väzbou od zákazníka a analyzuje ich, vyhodnotí riziká a určí úlohy pre ďalšiu iteráciu špirály. Projekt týmto spôsobom pokračuje až kým nie je hotový, alebo pokým analýza rizík nepreukáže, že bude lepšie v projekte nepokračovať.

Bieliková [15] uvádza ako hlavné výhody tohto modelu možnosť znovupoužitia, skoré odstránenie chýb, kladenie dôrazu na akosť, inegráciu údržby a vývoja, pokrytie súčasného vývoja softvéru/hardvéru. Za nevýhody považuje jeho všeobecnosť, ktorú je potrebné rozpracovať pre daný projekt ako aj skutočnosť, že zmluva o vývoji softvéru vopred určuje procesný model a výstupy.

Kadlec [14] za hlavné výhody špirálového modelu považuje jeho nezávislosť na konkrétnej metodike či stratégii, vytváranie prostredia pre vývoj znovupoužiteľných komponent a ich opakované využívanie. Ďalšími výhodami modelu sú jeho komplexnosť, včasné vylúčenie nevhodných riešení a možnosť presného stanovenia časovej náročnosti. K nevýhodám autor zaraďuje komplikovanosť, nároky na expertízu pracovníkov, absencia prepracovania všetkých činností počas vývoja a taktiež skutočnosť, že zmeny požiadaviek je možné uskutočniť iba po dokončení cyklu.

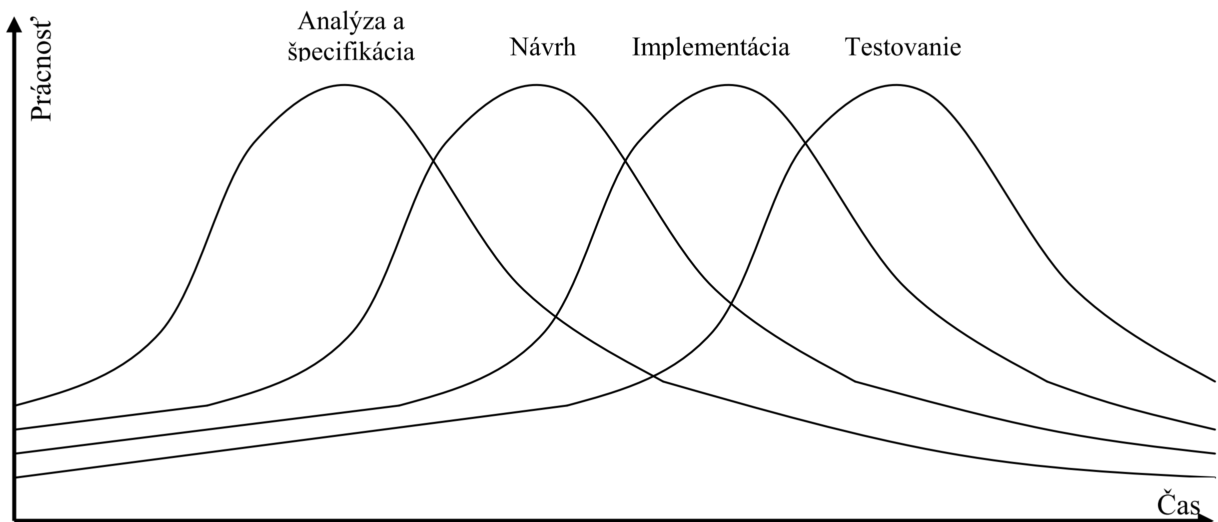
Patton [1] uvádza výhody použitia špirálového modelu z pohľadu testera a jeho práce. Tento model umožňuje aplikovať súčasný trend v testovaní, ktorým je testovanie softvéru vo všetkých fázach jeho životného cyklu. Model umožňuje softvérovému testerovi ovplyvňovať projekt už od jeho začiatku, pretože je zapojený do predbežných a počiatočných etáp vývoja. Produkt je testovaný priebežne a neustále, takže v závere stačí už iba overiť, že je všetko v poriadku a nie je nutné vykonávať všetko testovanie naraz, v závere projektu.

1.2.2.3 Concurrent engineering model

Concurrent engineering je módnym termínom v oblasti vývoja softvéru. S jeho použitím sa môžeme stretnúť pri veľkých softvérových systémoch, s ktorými pracuje NASA, ESA (European Space Agency) či Boeing.

ESA definuje **concurrent engineering** [17] ako systematický prístup k integrovanému vývoju produktu, ktorý zdôrazňuje zameranie sa na očakávania zákazníka. Predstavuje tímové hodnoty spolupráce, dôvery a zdieľania takým spôsobom, že proces rozhodovania sa stáva konsenzom, do ktorého sú paralelne zainteresované všetky perspektívy od začiatku životného cyklu produktu.

Moravčík, Vaský a Mišút [11] uvádzajú definíciu concurrent engineering modelu ako postupu, ktorý predpokladá časovú paralelu medzi jednotlivými etapami vývojového procesu softvéru. Všetky etapy začínajú a končia svoju činnosť v rovnakom okamihu, ale s rozdielnymi nárokmi na prácnosť. Tak je možné okamžite analyzovať, testovať a odstraňovať chyby a nedostatky v procese vývoja softvéru (pozri Obr.7).

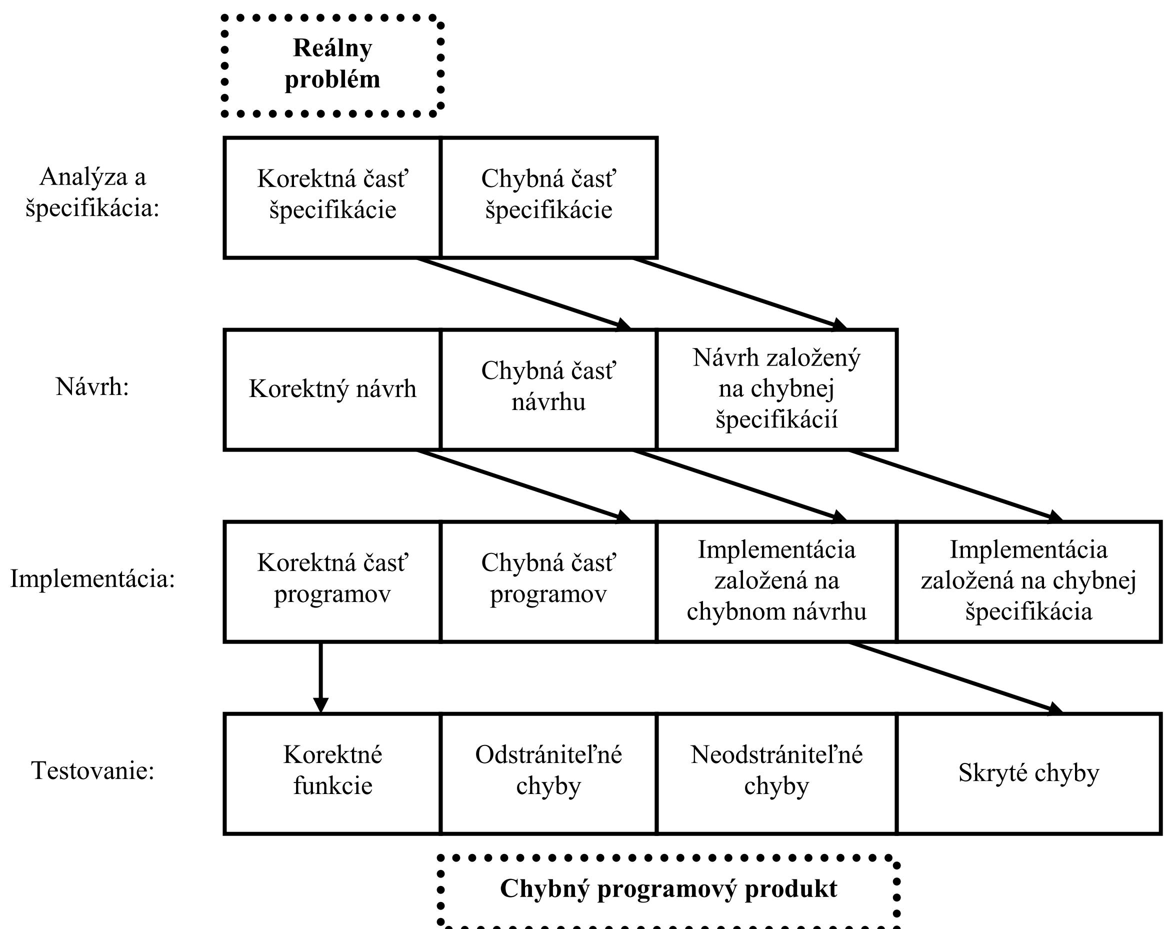


Obr. 7: „Concurrent Engineering“ model životného cyklu softvéru

1.2.3 Výskyt chýb počas životného cyklu softvéru

Chyby sa do softvéru môžu dostať v každej etape životného cyklu softvéru. Tie ak sa neodhalia, sú následne prenášane automaticky do ďalších etáp. Už chybná časť špecifikácie vyvolá návrh založený na chybnej špecifikácii. Aj keď vývojový tím návrh zostaví bezchybne, tak sa im chyba do návrhu prenesie z predošlej etapy. Výsledný softvérový produkt nakoniec obsahuje chyby, ktoré sa naukumulovali počas celého životného cyklu softvéru. Chyby môžu byť odstrániteľného, neodstrániteľného ale aj skrytého charakteru.

Čo sa týka výskytu chýb počas životného cyklu, platí jednoduchá úmernosť. Čím skôr sa počas vývojového cyklu podarí chybu odhaliť, tým menšie škody spôsobí a tým ľahšie je ju možné odstrániť. Výskyt chýb počas životného cyklu a ich prenos je znázornený na Obrázku 8.



Obr. 8: Kumulácia chýb počas životného cyklu softvéru podľa Faigla [4]

Chyby vznikajú vždy, ale ak je projekt správne riadený mala by byť prevažná väčšina chýb odhalená.

1.3 Testovanie

1.3.1 Charakteristika pojmu testovanie

Definície najčastejšie charakterizujú testovanie ako proces hodnotenia, skúmania, vyšetrovania či overovania softvéru, ktorého cieľom je poskytnúť alebo potvrdiť určité informácie o produkte (v našom prípade softvér).

Podľa Kanera [7] je testovanie softvéru vyšetrovaním, ktoré je vykonané s cieľom poskytnúť zainteresovaným informácie o kvalite produktu, alebo služby, ktorá je testovaná.

Podľa podrobnejšej definície dostupnej v encyklopédii na internete, je testovanie softvéru proces validácie a verifikácie, ktoré potvrdia, že softvérový program, aplikácia alebo produkt:

1. vyhovuje obchodným a technickým požiadavkám, ktoré riadia jeho návrh a vývoj,
2. pracuje tak, ako sa to vyžaduje,
3. a môže byť implementovaný s rovnakými vlastnosťami.

Práve pojem verifikácie a validácie sa v súvislosti s testovaním softvéru spomína najčastejšie. Tran [9] tieto dva pojmy definuje nasledovne:

- Verifikácia: Vytvorili sme softvér správne? (t.j., vyhovuje špecifikácii?)
- Validácia: Vytvorili sme ten správny softvér? (t.j., je to presne to, čo zákazník chce?)

Podľa Pattona [1] je verifikácia softvéru proces, ktorého cieľom je potvrdiť, že softvér vyhovuje zadanej špecifikácii. Pri validácii sa potom kontroluje, či softvér vyhovuje požiadavkám užívateľa.

Nakoľko sa v praxi tieto dva pojmy veľmi často zamieňajú, resp. používajú ako synonymá, uvádzam v závere ešte definíciu týchto pojmov podľa IEEE Standard Glossary of Software Engineering Terminology:

- Verifikácia je proces hodnotenia systému alebo jeho zložky, ktoré určí, či produkty v danom štádiu vývoja vyhovujú podmienkam stanoveným na začiatku tejto fázy.
- Validácia je proces hodnotenia systému alebo jeho zložky počas alebo na konci procesu vývoja, ktoré určí, či vyhovuje požadovaným špecifikáciám.

1.3.2 História testovania

Testovanie softvéru má iba niekoľko desaťročí starú históriu. Podobne ako softvérové inžinierstvo je mladšie ako niektorí jeho používatelia. Z tejto skutočnosti vyplýva, že softvérové testovanie prechádza rýchlym vývojom a výraznými premenami. Podľa Kadleca [14] by sa dalo povedať, že testovanie sa vyvíja „za pochodu“ tak ako sa vyvíja samotný softvér.

Kadlec [14] uvádza nasledovné etapy v histórií vývoja softvéru:

- Etapa do polovice 60. rokov minulého storočia je priekopníckym obdobím, v ktorom vznikali väčšinou neudržiavateľné a nemenné programy, ktoré boli vypálené do trvalej pamäte počítača (ROM, EPROM).
- Etapa prelomu 60. a 70. rokov súvisí so vznikom softvérového inžinierstva a v centre záujmu stojí význam správne zloženého vývojového tímu softvéru.
- Etapa 70. rokov prináša prvé aplikácie umožňujúce interakciu s ich užívateľmi, dostupné sú prvé verejne predávané softvéry. Zároveň sa objavujú prvé zásadné chyby a problémy s dodávkou, prípadne s finalizáciou softvéru. Až do konca 70. rokov neexistujú žiadne zavedené postupy pre vývoj softvéru.
- Etapa 80. rokov znamená masívny rozvoj softvérového inžinierstva, objavujú sa prvé metodiky, analýzy a štandardizácia softvérových produktov.
- Etapa 90. rokov prináša uznanie softvérového inžinierstva ako odboru s certifikátom v USA v roku 1997.

Dalo by sa povedať, že v súčasnosti sa vývoj softvéru zameriava na prevenciu chýb, ktoré spôsobili softvérovú krízu v 60. rokoch minulého storočia. Kadlec [14] uvádza ako charakteristické znaky softvérovej krízy neúnosné predlžovanie a predražovanie projektov, nízku kvalitu programov, náročnosť údržby a inovácií, zlú produktivitu práce programátorov, neefektívnosť vývoja a neistotu výsledku projektu.

Ako vyplýva z hore uvedeného, potreba testovania softvéru sa objavuje až koncom 70. rokov. Glendford J. Mayers v roku 1979 ako prvý navrhol oddelenie testovania od odstraňovania chýb (debugging).

Podľa Gelperina a Hetzela [8] prebiehal vývoj testovania počas histórie v nasledovných etapách:

- Etapa do roku 1956 znamenala iba odstraňovanie chýb (debugging), ktoré sa objavili počas používania daného hardvéru či softvéru.

- Etapa od roku 1957 do roku 1978 sa označuje ako „demonstration oriented“, čo znamená že bolo potrebné preukázať, že softvér vyhovuje stanoveným požiadavkám a že odstraňovanie chýb (debugging) treba odlíšiť od testovania.
- Etapa od roku 1979 do roku 1982 je etapou výlučne zameranou na odhaľovanie chýb softvéru a ich následne odstraňovanie (destruction oriented).
- Etapa od roku 1983 do roku 1987 je vývojovým obdobím zameraným na evaluáciu softvéru. Evaluácia softvérového produktu a meranie jeho kvality sa vykonáva počas životného cyklu softvéru.
- Etapa od roku 1988 do roku 2000 je obdobím zameraným na prevenciu (prevention oriented). Testovanie slúži na preukázanie skutočnosti, že softvér vyhovuje daným špecifikáciám, vrátane odhalenia chýb a ich prevencie.

1.3.3 Typy testovania

Predmetom testovania je minimalizácia rizika zlyhania softvéru, jeho nesprávnej funkcie, preťaženie alebo malého výkonu pri reálnej prevádzke. Testovanie softvéru umožňuje optimalizáciu návrhu a kvalifikované stanovenie rezerv z hľadiska záťaže a overenie z hľadiska funkčnosti. Testovanie môžeme deliť podľa Paletu [12] na tri základné kategórie:

- funkčné testovanie,
- užívateľské testovanie,
- výkonnostné testovanie.

Väčšina testov, ktoré počas vývoja softvéru vykonávame, patria do kategórie funkčné testovanie. Niektoré vývojové tímy sa už žiadnemu ďalšiemu testovaniu nevenujú.

Obr. 9: Rozdelenie testovania podľa Paletu [12]

1.3.3.1 Funkčné testovanie

Cieľom tohto testovania je predovšetkým overiť, či softvér vykonáva správne to, čo vykonávať má. Ide o počítanie správnych výsledkov, zapisovanie správnych údajov do databázy a podobne. Taktiež sa musí overiť, či softvér vyhovuje všetkým bodom

špecifikácie. To znamená, že vykonáva všetko to, čo má vedieť vykonávať. A ešte si treba overiť, že za žiadnych okolností, ani pri veľkom zaťažení alebo útoku, nezačne vykonávať niečo, čo nesmie. Ide napríklad o sprístupnenie chránených informácií, alebo aj zhavarovanie samotného testovaného softvéru.

Najvhodnejšie je si rozdeliť softvér na jednotlivé moduly, funkčné celky a testovať ich zvlášť. S testovaním sa nemôže čakať pokiaľ jednotlivé moduly alebo dokonca celý softvér bude dokončený. Začína sa od okamžiku, kedy sa podarí skompilovať úplne prvú verziu vyvíjaného softvéru.

1.3.3.2 Užívateľské testovanie

Užívateľské testovanie overuje softvér z pohľadu koncového užívateľa. Tieto typy testov vyžadujú aby celý softvér, prípadne aspoň jeho užívateľské rozhranie, bolo takmer v dokončenom stave. Testy by mali vykonávať budúci užívatelia, alebo osoby s podobnými znalosťami aké majú užívatelia. Vhodné je, aby užívateľov pozoroval tester, ktorý sleduje nielen ich schopnosť orientovať sa v danom softvéri, ale aj ich pripomienky a postrehy.

V praxi sa často uplatňuje variant rozposielania softvéru užívateľom a následne sa čaká na ich reakcie a pripomienky. Je to síce najjednoduchší spôsob, ale má niekoľko závažných nedostatkov. Užívatelia môžu mať protichodné požiadavky na softvér a často strávia zbytočne veľa času hľadaním potrebnej funkcie. Tieto problémy odstraňuje už spomínané spoločné sedenie užívateľa a testera nad vyvíjaným softvérom.

S testami je vhodné začať už od okamžiku, kedy bude softvér natoľko hotový, aby to malo zmysel aj pre budúceho užívateľa. Treba ho však vopred upozorniť čo je v softvéri už hotové a čo nie. Je veľmi dôležité predísť prípadnému sklamaniu užívateľa.

1.3.3.3 Výkonnostné testovanie

Testovanie výkonnosti má zabezpečiť aby softvér bol dostatočne rýchly na predpokladanom hardvéri pri maximálnej očakávanej záťaži. Od určitého okamžiku pri vývoji sa ďalšie zvyšovanie výkonnosti stáva zbytočným, alebo príliš nákladným. Často je lacnejším riešením kúpiť nový hardvér. Preto cieľom tohto testovania nie je dosiahnutie maximálneho možného výkonu, ale zistenie a odstránenie výkonnostne slabých miest.

Spočiatku výkonnostné testy môžeme spustiť pri nízkej záťaži a na malom množstve dát. Ale čo najskôr je potrebné sa priblížiť k reálnym podmienkam ostrej

prevádzky. Zvýšené množstvo dát a zvýšená záťaž užívateľov na testovaný softvér môže totiž spôsobiť, že sa objavia slabé miesta úplne inde, ako by sme ich čakali na základe skúseností s prevádzkou pri nízkej záťaži.

1.4 Úloha a výber testera

V súčasnosti sa kladie dôraz na vyššiu kvalitu softvéru a stým sú spojené zvýšené nároky na softvérového testera. Pre vytvorenie kvalitného softvéru sú potrební testeri s rôznou úrovňou profesionálnych skúseností. Navyše testeri, ktorí dokážu tiež programovať, môžu vykonávať testovanie bielej skrinky a taktiež vyvíjať automatizované testovacie mechanizmy. Tester, ktorý pracoval na niekoľkých projektoch, môže viesť malý tím ďalších testerov.

Podľa Pattona [1] je cieľom softvérového testera vyhľadávať chyby, vyhľadať ich čo najskôr a zabezpečiť ich nápravu.

Paleta [12] charakterizuje testera ako člena vývojového tímu softvéru, ktorého úlohou je zabezpečiť, aby mal výsledný produkt požadovanú kvalitu. Úlohou testera je pripravovať a vykonávať všetky typy testov.

V súčasnosti pracujú testeri v tímoch, pod vedením hlavného testera. Úlohami testera môžu byť poverení aj analytici alebo dizajnéri softvéru, ktorí už splnili svoje úlohy v počiatočnej fáze vývoja softvéru. Dôležité je nepozerať sa na prácu testera ako na vedľajšiu, alebo doplnkovú aktivitu.

Táto práca vyžaduje od testerov vysoký stupeň sebadisciplíny, systematickosti a zodpovednosti. V ich rukách je posledná možnosť chyby nielen nájsť, ale ich aj odstrániť a tým predísť škodám rôzneho rozsahu.

Správny výber testerov je preto dôležitým krokom v zostavovaní vývojového tímu softvéru a platia pre nich rovnaké kritériá ako pre výber programátorov, dizajnérov, či analytikov. Paleta [12] uvádza nasledovné kritériá:

- schopnosť splniť danú úlohu,
- schopnosť komunikovať,
- profesionalita a dôkladnosť,
- dostatočný technický prehľad.

1.5 Veľké softvérové systémy

Veľké softvérové systémy sú predmetom softvérového inžinierstva, ktoré študuje ich hlavné komponenty a ich vzájomnú interakciu. V tejto súvislosti sa často hovorí aj o softvérovej architektúre.

Wikipédia – encyklopédia dostupná na Internete definuje softvérový systém [19] ako systém založený na softvéri, ktorý tvorí časť počítačového systému (kombinácia hardvéru a softvéru). Pojem softvérový systém sa často používa ako synonymum pojmov počítačový program alebo softvér.

Hlavnými kategóriami softvérových systémov sú:

- aplikačný softvér (napr. kancelárske balíky, multimediálne prehrávače, účtovný softvér, grafický softvér, prehliadače webových stránok),
- programovací softvér - nástroje na vývoj softvéru a podpory iných programov a aplikácií (napr. Delphi od firmy CodeGea, NetBeans od Oracle, a od firmy Microsoft – Microsoft Visual Studio),
- systémový softvér - počítačový softvér navrhnutý na fungovanie počítačového hardvéru a na vytvorenie platformy pre fungovanie aplikačného softvéru (napr. operačné systémy Microsoft Windows 7 od firmy Microsoft, Mac OS X verzia 10.6 Snow Leopard od firmy Apple, Linux).

Každý softvérový systém má svoju softvérovú architektúru. Wikipédia definuje softvérovú architektúru ako systém štruktúr potrebných na to, aby sme mohli hovoriť o systéme, ktorý pozostáva zo softvérových elementov, vzťahov medzi nimi a ich vlastností.

V praxi sa používajú nasledovné typy softvérovej architektúry:

- **SOA** (service-oriented architecture) [21] – *„servisne orientovaná architektúra je na koncepte služieb založený architektúrny prístup k návrhu, implementácii a riadeniu distribuovaného spracovania, ktoré daná firma potrebuje k realizácii svojej stratégie a k dosiahnutiu obchodných cieľov. Tento prístup je postavený na princípe voľne viazaných, opakovane použiteľných, definovaných a na štandardoch založených služieb, ktoré sú dostupné a využiteľné nezávislými spotrebiteľmi služieb. SOA umožňuje organizáciám vhodne prepojiť obchodné a IT služby a zabezpečiť tak v prostredí neustálych zmien a turbulencií schopnosť riadenia, stability, predpovedateľnosti, bezpečnosti.“*

- **MDA** (model-driven architecture) – modelom riadená architektúra je podľa Šveřepu [22] koncepcia, ktorá slúži na štandardizáciu modelov, ktoré sú vytvárané počas vývoja aplikácie a definuje spôsoby ich mapovania. Taktiež štandardizuje a definuje spôsob transformácie jedného modelu na druhý. Primárnym cieľom tohto prístupu je zabezpečiť prenositeľnosť, interoperatibilitu a znovupoužiteľnosť. MDA popisuje modely aplikácií na štyroch úrovniach:
  1. CIM – model nezávislý na počítačovom spracovaní
  2. PIM – model nezávislý na platforme
  3. PSM – model špecifický na konkrétnej platforme
  4. Zdrojový kód aplikácie (výsledná implementácia)

  Prvá verzia bola publikovaná v roku 2001 konzorciom Object Management Group, ktoré vyvíja aj UML. Koncept MDA úzko súvisí s UML, ale nie je na tento modelovací štandard viazaná, a dá sa aplikovať aj iným spôsobom modelovania.

Počiatočné pokusy zachytiť softvérovú architektúru spočívali v diagramoch (box-and-line diagrams). Až v priebehu 90. rokov sa objavili snahy vytvoriť systém dizajnu, modelovacieho jazyka pre popis softvéru a formálnej logiky. Na popis architektúry softvéru sa používajú rôzne jazyky, ale ohľadom ich používania neexistuje žiadny konsenzus. UML (unified modeling language) je štandardizovaný modelovací jazyk pre popis softvéru, všeobecne používaný v softvérovom inžinierstve.

1.6 UML – Unifikovaný modelovací jazyk

UML (unifikovaný modelovací jazyk) je modelovací jazyk, ktorý umožňuje popis objektovo orientovanej analýzy a návrhu. Tento jazyk sa používa aj na vizualizáciu, špecifikovanie, konštruovanie a dokumentovanie softvérových systémov. Je to súhrn grafických notácií, ktoré slúžia na vyjadrenie analytických a návrhových modelov. Formálna syntax jazyka umožňuje modelovať jednoduché i zložité aplikácie. UML je výsledkom zjednotenia najlepších existujúcich postupov modelovacích techník a softvérového inžinierstva, ktorý vznikol pod záštitou firmy Rational.

Grafické modelovacie jazyky majú v súčasnosti svoje opodstatnenie v softvérovom priemysle, pretože programovacie jazyky nedosahujú takú vysokú úroveň abstrakcie, aby boli užitočné pri diskusiách o návrhu softvéru. Grafické jazyky umožňujú zdieľať výsledky práce s ostatnými návrhármi, sú pochopiteľné i pre zadávateľa aplikácie a uľahčujú komunikáciu medzi tvorcom softvéru a jeho užívateľom.

Paleta [12] uvádza okrem výhod tohto jazyka, aj niekoľko nevýhod. Hlavnou výhodou jazyka UML sú určite diagramy, ktoré používa. Tie sú intuitívne, zrozumiteľné a ľahko pochopiteľné aj pre laika. Používanie jazyka nevyžaduje dlhoročné štúdium, existuje veľa nástrojov, ktoré tento jazyk podporujú, čím umožňujú jeho široké používanie. Za nevýhody UML autor považuje jeho základnú jednotku – triedy. UML neobsahuje žiadne nástroje na modelovanie štruktúry databáz, a samozrejme graficky sa ním nedá znázorniť úplne všetko.

Podľa Tanušku a kol. [24] medzi výhody použitia UML patrí použiteľnosť tohto jazyka v rôznych oblastiach a schopnosť popísať čokoľvek: návrh softvéru, biznis procesov, požiadavky a analýzu systému. UML je možné úspešne použiť v doménach podnikových a finančných systémov, telekomunikácií, zdravotníckej starostlivosti, maloobchodu, zásobovania, atď. UML je možné použiť aj ako programovací jazyk.

1.6.1 História vzniku UML

Prvé objektovo orientované modelovacie jazyky existovali už koncom 70. rokov minulého storočia.

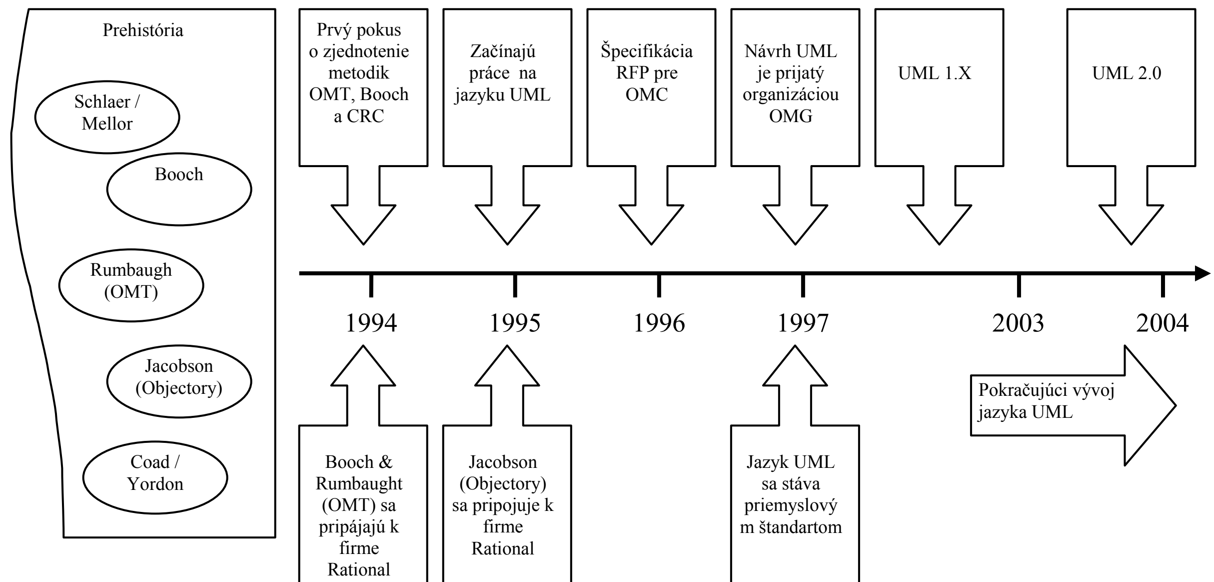
Do roku 1994 existovalo niekoľko modelovacích jazykov a metodík (BOOCH, OMT, OBJECTORY) s podobnými vlastnosťami, ktoré si navzájom konkurovali. Prvým pokusom o ich zjednotenie bola metodika Fusion, ale nakoľko k jej tvorbe neboli prizvaní autori už existujúcich metód (Booch, Jacobson a Rumbaugh), upadla do zabudnutia a na trhu ju konkurenčne predstihol jazyk UML (verzia 0.9) od firmy Rational.

Paralelne s vývojom jazyka UML prebiehala aj jeho štandardizácia, ktorá bola kontrolovaná otvoreným konzorciom spoločností Object Management Group (OMG) zostaveného za účelom vytvárania štandardov podporujúcich interoperabilitu objektovo orientovaných systémov. V roku 1997 prijalo združenie OMG jazyk UML (verzia 1.1) ako štandard objektovo orientovaného jazyka pre vizuálne modelovanie. Ďalej nasledovalo upresňovanie špecifikácie a vznikali ďalšie verzie 1.2 (1998) a 1.3 (1999).

V roku 2000 jazyk UML zaznamenal významný pokrok, keď bol rozšírený o sémantiku akcií, čo umožnilo podrobnejšiu špecifikáciu prvkov týkajúcich sa správania modelov UML (napríklad operácií). Nasledovali verzie 1.4 (2001) a 1.5 (2002).

V roku 2004 bola schválená verzia 2.0, ktorá priniesla podstatné rozšírenia. Je to zatiaľ najrozsiahlejšia špecifikácia (600 strán) a predstavuje najkompaktnejšiu verziu UML. Jednou z výrazných zmien tejto verzie je zavedenie nových typov diagramov

(diagram prehľadu interakcií, časový diagram a diagram zloženej štruktúry). Jazyk UML sa tak stal veľmi vyspelým modelovacím jazykom, ktorého kvalitu v priebehu nasledujúcich rokov potvrdili tisíce softvérových projektov. Najnovšia verzia 2.2 je z februára 2009.



Obr. 10: Zrodenie jazyka UML, Arlow a Neustad [23]

1.6.2 Štruktúra jazyka UML

UML je unifikovaný jazyk, ktorý bol navrhnutý a zostavený ako systém, to znamená, že ako taký má vlastnú štruktúru. Bol sám modelovaný a navrhnutý v jazyku UML. Tento návrh sa označuje ako **metamodel** jazyka UML.

Arlow a Neustadt [23] uvádzajú tieto súčasti štruktúry jazyka UML:

- Stavebné bloky – sú to základné prvky modelu, relácie (vzťahy) a diagramy.
- Spoločné mechanizmy – popisujú štyri stratégie modelovania objektov (špecifikácia, ornamenty, podskupiny, mechanizmy rozšíriteľnosti).
- Architektúra – organizačná štruktúra systému.

1.6.2.1 Stavebné bloky

Arlow a Neustadt [23] uvádzajú tri stavebné bloky UML:

- Predmety
- Relácie
- Diagramy

Autori delia blok **predmety** na :

- štruktúrne abstrakcie (podstatné mená v jazyku UML - napr. trieda, rozhranie, spolupráca, prípad užitia, aktívna trieda, komponent),
- správanie (slovesá v jazyku UML, ktoré opisujú napr. interakcie a stavy),
- zoskupenia (balíčky používané k zoskupovaniu významovo súvisiacich prvkov do súdržných jednotiek),
- poznámky (anotácie).

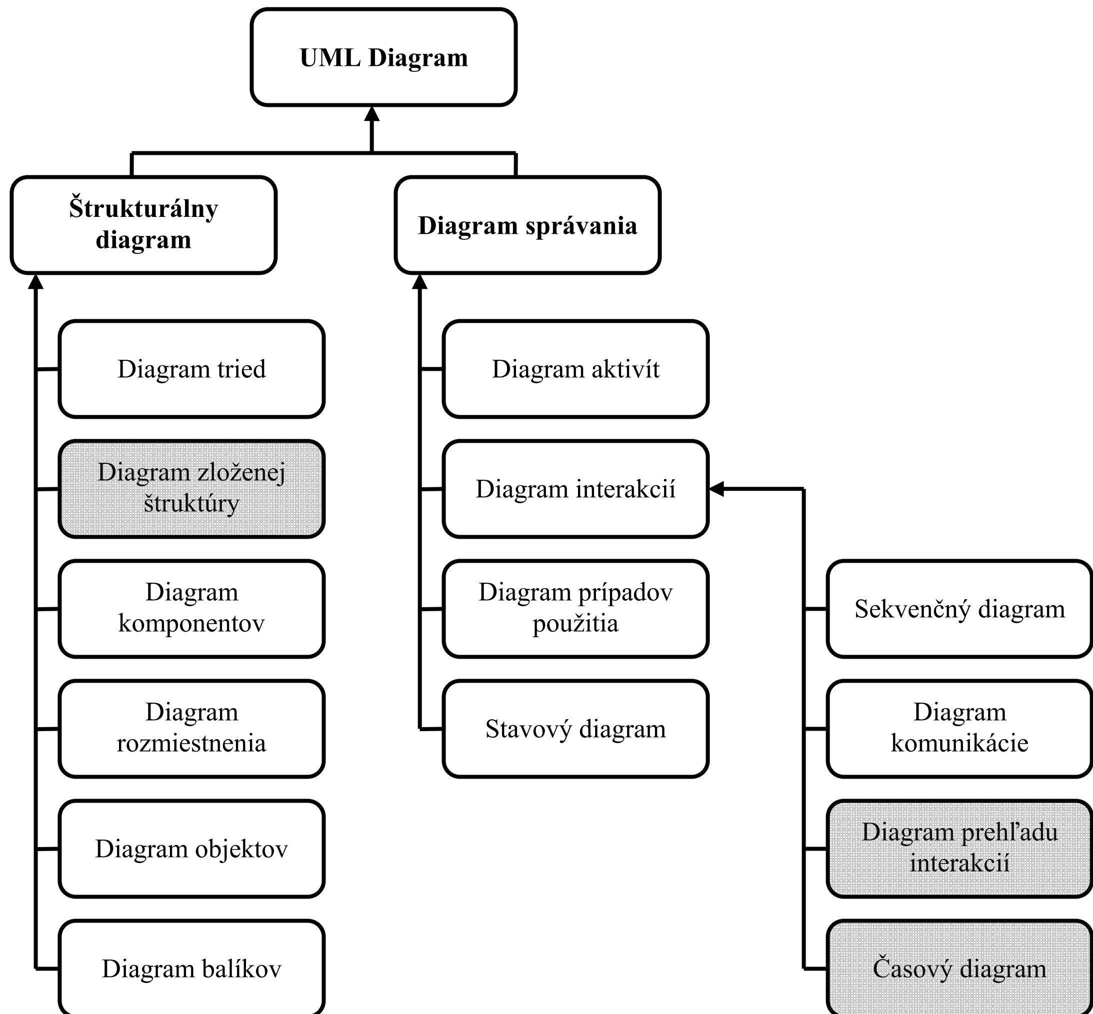
Blok **relácie** umožňuje na modeli znázorniť vzťahy medzi dvoma predmetmi a zachytiť sémantický vzťah medzi nimi. Dole (pozri Obr.11) je uvedený stručný prehľad relácií a ich grafického znázornenia.

Závislosť	Zmena v určitom predmete ovplyvňuje význam závislého predmetu.
Asociácia	Popis množiny spojenia medzi objektmi.
Agregácia	Cieľový prvok je súčasťou zdrojového prvku.
Kompozícia	Silnejšia forma agregácie (má viac obmedzení).
Ochranná nádoba	Zdrojový prvok obsahuje cieľový prvok.
Zovšeobecnenie	Jeden prvok je špecializáciou iného prvku a je ho možné nahradiť všeobecnejším (univerzálnejším) prvkom.
Realizácia	Asociácia medzi klasifikátormi, kde jeden klasifikátor určuje dohodu, ktorej uskutočnenie zaručuje druhý klasifikátor.

Obr. 11: Typy relácií UML podľa autorov Arlow a Neustadt [23]

Blok s názvom **diagramy** tvorí množina obsahujúca rôzne typy diagramov, pričom jeden diagram je „okno“ alebo „pohľad“ na model (Arlow a Neustadt, [23]). Diagramy sú základným mechanizmom na zadávanie nových informácií do existujúceho modelu.

UML model zvyčajne pozostáva z jedného alebo viacerých diagramov. Podľa Tanušku a kol. [24] diagram graficky znázorňuje entity a ich vzájomné vzťahy. Entitami môžu byť skutočné objekty reálneho sveta alebo softvérové entity. UML 2.0 opisuje 13 oficiálnych typov diagramov. Prehľad existujúcich typov diagramov je znázornený na Obrázku 12 dole.



Obr. 12: Typy diagramov UML vo verzii 2.0 podľa Tanušku [24]

Diagramy jazyka UML je možné rozdeliť na tie, ktoré modelujú statickú štruktúru systému (statický model) a na tie, ktoré modelujú dynamickú štruktúru systému (dynamický model). **Štrukturálne diagramy** znázorňujú statické vlastnosti modelu. Slúžia na opis fyzickej organizácie entít v systéme. **Diagramy správania** slúžia na opis správania elementov v systéme.

1.6.2.2 Mechanizmy

Modelovanie objektov v jazyku UML používa štyri základné mechanizmy, ktoré sú používané konzistentne. Arlow a Neustadt [23] uvádzajú nasledovné mechanizmy:

1. Špecifikácie – textové popisy sémantiky jednotlivých prvkov, ktoré dopĺňajú grafický rozmer, vizualizáciu pomocou diagramov. Sémantický podklad modelu je tvorený pomocou nástroja CASE.
2. Ornamenty (Adornments) – zvyšujú zrozumiteľnosť a čitateľnosť diagramov alebo zdôrazňujú určitú dôležitú funkciu modelu. Používajú sa vtedy, ak je nutné prostredníctvom diagramu zobraziť viac informácií.
3. Podskupiny – popisujú rôzne spôsoby videnia sveta.
   - Prvý spôsob predstavuje klasifikátor (abstraktné vyjadrenie predmetu) a inštancia (špecifický, konkrétny predmet). V jazyku UML sa používa taxonómia 33 klasifikátorov.
   - Druhý spôsob používa rozhranie (to, čo predmet vykonáva) a implementáciu (ako to predmet vykonáva).
4. Mechanizmy rozšíriteľnosti – sú to obmedzenia, ktoré sa týkajú jazyka, stereotypov, vlastností prvkov a profilu UML.

1.6.2.3 Architektúra

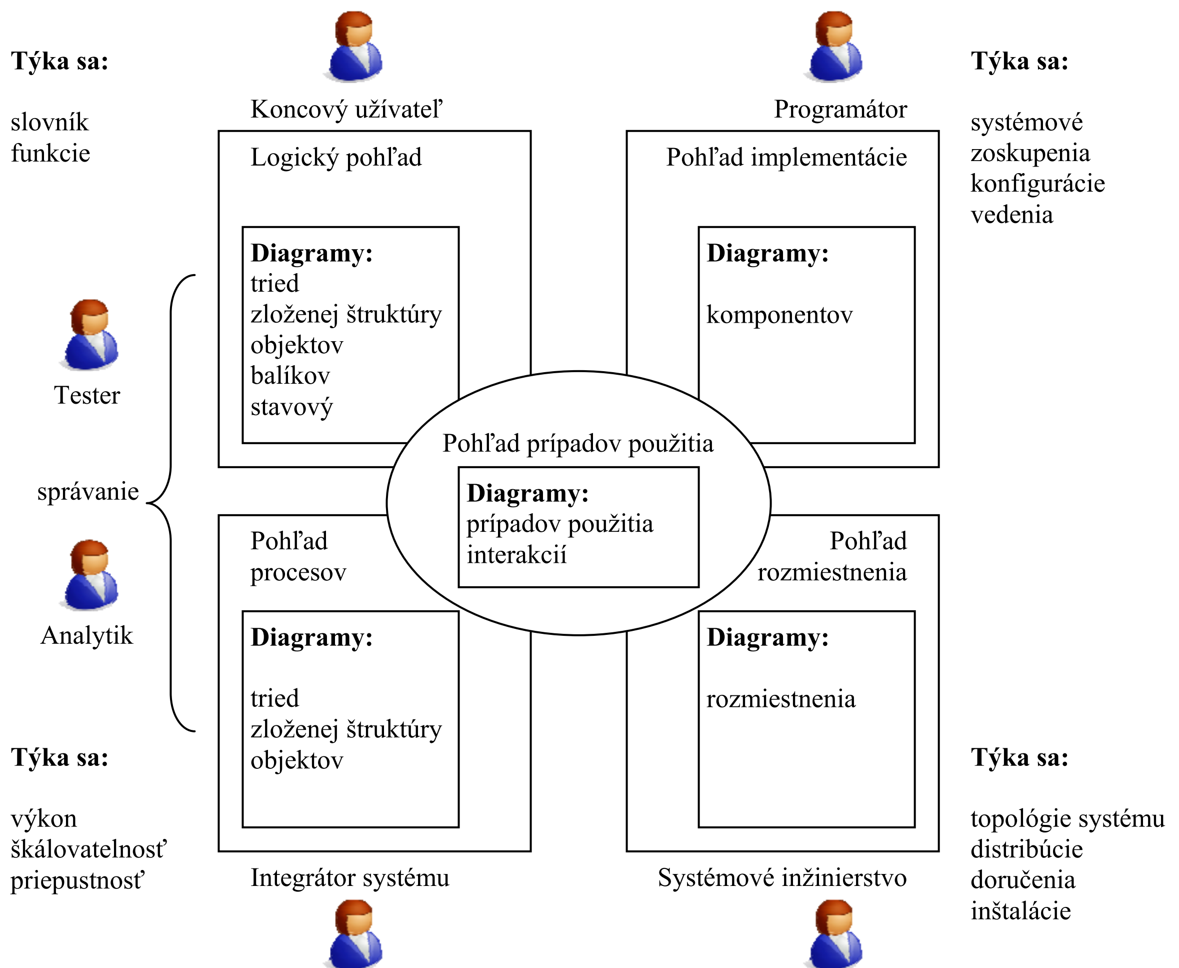
Arlow a Neustadt [23] definujú architektúru ako „*organizačnú štruktúru systému, vrátane jeho rozkladu na jednotlivé časti, ich prepojiteľnosti, interakcie, mechanizmov a zásad, ktorá preniká do návrhu systému*".

Podľa štandardu IEEE je architektúra najvyššia úroveň koncepcie systému v jeho vlastnom prostredí.

Tanuška a kol. [24] hovorí o architektúre systému v súvislosti s modelovaním statickej a dynamickej stránky systému. Nástroje a prostriedky jazyka UML nám dovolia modelovať vytváraný systém „*zo všetkých možných strán a na všetkých možných úrovniach*".

Jazyk UML definuje štyri základné pohľady na systém, ktoré nám umožňujú zachytiť všetky podstatné aspekty architektúry systému. Všetky tieto pohľady sú integrované do piateho, ktorý je znázornený na Obr.13.

1. Logický pohľad – zobrazuje systém ako množinu tried a objektov a spôsob, akým vytvárajú správanie systému.
2. Pohľad procesov – je procesne orientovaný variant logického pohľadu.
3. Pohľad implementácie – modeluje súbory a komponenty, ktoré tvoria hotový kód systému.
4. Pohľad rozmiestnenia – modeluje fyzické nasadenie komponentov na množinu fyzických výpočtových uzlov (počítačov a periférnych zariadení).
5. Pohľad prípadov použitia – zachytáva základné požiadavky, ktoré sa kladú na príslušný systém ako na množinu prípadov použitia.



Obr. 13: Základné pohľady na systém Arlow a Neustadt [23]

1.6.3 Produkty pre prácu s UML

Pre prácu s UML môžeme použiť rôzne produkty na komerčnej aj nekomerčnej báze. Medzi komerčné produkty patria:

- Rational Rose
- ARTiSAN Real-time Studio
- I-Logix Rhapsody
- MetaMatrix MetaBase Modeler
- Together Designer Community Edition

Nekomerčné produkty určené na prácu s UML sú väčšinou šírené na Internete zdarma pod licenciou GPL. V porovnaní s komerčnými produktmi sú však menej komplexné, čo znamená kratší čas potrebný na ich zvládnutie. Medzi nekomerčné produkty patria:

- Violet
- Umbrello UML Modeler
- ArgoUML
- StarUML

Z hore uvedených nekomerčných produktov bola za účelom práce s UML použitá aplikácia Violet.

Aplikácia **Violet**, ktorá je určená na tvorbu UML diagramov je šírená pod licenciou GPL a je dostupná na domovskej stránke http://violet.sourceforge.net**.** Violet slúži na tvorbu UML diagramov bez ďalšej náväznosti na zdrojový kód. Medzi výhody tejto aplikácie patrí, že je vytvorená v Jave a je tak bez problémov prenositeľná na rôzne platformy a operačné systémy. Práca s jednotlivými typmi diagramov je jednoduchá. Aplikácia je určená pre študentov a učiteľov, ktorí sa zaoberajú softvérovým inžinierstvom alebo objektovo orientovaným programovaním.

1.7 Normy pre testovanie softvéru

Počas všetkých etáp životného cyklu softvéru, celý realizačný tím podieľajúci sa na plánovaní, tvorbe či testovaní softvéru si môže vybrať a aplikovať už vypracované normy a štandardy. Krátky zoznam niektorých noriem súvisiacich s testovaním softvéru, ktoré sú v súčasnosti (v roku 2011) uvádzané na domovských stránkach svojich vydavateľov ako aktuálne, sú:

- BS 7925-1:1998, Software testing. Vocabulary.
- BS 7925-2:1998, Software testing. Software component testing.
- DO-178B:1992 Errata 1999, Software Considerations in Airborne Systems and Equipment Certification.
- IEEE Std 610.12:1990, IEEE Standard Glossary of Software Engineering Terminology.
- IEEE Std 829:2008, IEEE Standard for Software and System Test Documentation.
- ANSI/IEEE Std 1008:1987, IEEE Standard for Software Unit Testing.
- IEEE Std 1012:2004, IEEE Standard for Software Verification and Validation.
- IEEE Std 1028:2008, IEEE Standard for Software Reviews and Audits.
- IEEE Std 1044:2009, IEEE Standard Classification for Software Anomalies.
- ISO/IEC 14764 IEEE Std 14764:2006, International Standard - Software Engineering - Software Life Cycle Processes Maintenance.
- ISO/IEC 2382-1:1993, Information technology - Vocabulary - Part 1: Fundamental terms.
- ISO/IEC 25010:2011, Systems and software engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models (plne nahrádza normu ISO/IEC 9126-1:2001).
- ISO/IEC 12207:2008, Systems and software engineering - Software life cycle processes.
- ISO/IEC 14598-1:1999, Information technology - Software product evaluation - Part 1: General overview.
- ISO 9000:2005, Quality management systems - Fundamentals and vocabulary.

1.7.1 Norma IEEE 829-2008

Dokumenty noriem IEEE (Institute of Electrical and Electronics Engineers) sú vyvíjané ako konsenzus procesu vývoja schváleného Americkým národným inštitútom pre štandardy (the American National Standards Institute). IEEE je celosvetové združenie odborníkov, ktorí sa zameriavajú na rozvoj technických a technologických inovácií. Získané poznatky sú šírené vo forme publikácií a konferencií pre odbornú ale aj laickú verejnosť. IEEE je rozdelená do desiatich geografických regiónov a Slovensko patrí do regiónu 8 (Afrika, Európa, Stredný východ) a spadá do Československej sekcie.

Aktuálna verzia normy IEEE 829-2008 [31] vznikla revíziou predošlej verzie IEEE 829-1998. Táto norma podporuje všetky etapy životného cyklu softvéru, vrátane akvizície, dodania, vývoja, prevádzky a údržby. Procesy systémového a softvérového testovania určia, či výstup danej aktivity uspokojuje požiadavky tejto aktivity a či vývoj produktu vyhovuje zamýšľanému použitiu a potrebám užívateľa.

Táto norma je aplikovateľná na všetky softvérové systémy. Týka sa systémov a softvéru, ktoré sú vo vývoji, sú získané, funkčné, udržiavané alebo znovu uvedené do prevádzky. Norma IEEE 829-2008 slúži na tieto účely:

- Vymedzenie spoločného rámca pre procesy testovania, aktivity a úlohy na podporu všetkých etáp životného cyklu softvéru, vrátane akvizície dodania, vývoja, prevádzky a údržby.
- Definuje úlohy, požadované vstupy a výstupy.
- Identifikuje odporúčané minimum testovacích úloh korešpondujúcich s úrovňami integrity v 4-úrovňovej schéme integrity.
- Definuje použitie a obsah Master Test Plan a Testovací plán úrovne. (napríklad pre jednotky, moduly, systém, akceptačné testovanie).
- Definuje použitie a obsah súvisiacej testovacej dokumentácie (Návrh testovania, Testovací prípad, Testovacia procedúra, Report anomálií, Testovací log, Testovací report úrovne a Master Test Report).

1.7.1.1 Definície základných pojmov

Norma vymedzuje definície základných pojmov, ktoré sú použité pri charakteristike jednotlivých postupov a dokumentov. Patria medzi ne napríklad pojmy chyby, testovania, životného cyklu, dokumentácie, validácie, verifikácie, testovacieho prípadu, testovacieho skriptu, testovacieho scenára, plánu testovania, jednotlivých typov testov a úrovni testovania. Tieto pojmy sú definované v predchádzajúcich alebo nasledujúcich odsekoch textu, preto ich už neuvádzame. Z hľadiska prípravy a vykonania testovania je však dôležité definovať a následne stanoviť úroveň integrity.

Podľa normy IEEE829-2008 [31], úroveň integrity je miera, do akej softvér spĺňa alebo musí spĺňať súbor vlastností (napr. softvérová zložitosť, úroveň bezpečnosti, želateľná výkonnosť, spoľahlivosť, cena), ktoré určil zákazník v špecifikácii softvéru a ktoré sú definované tak, aby reflektovali význam softvéru pre jeho užívateľov. Úroveň integrity sa určí výberom zo schémy úrovní integrity.

Schéma úrovní integrity je súbor charakteristík systému (ako je komplexnosť, riziko, úroveň bezpečnosti, želateľná výkonnosť, spoľahlivosť, cena), ktoré boli vybrané ako dôležité pre užívateľa a rozvrhnuté do samostatných úrovní výkonu alebo plnenia (úrovne integrity) s cieľom napomáhať definovaniu úrovní kontroly kvality, ktorá má byť použitá pri vývoji a/alebo dodaní softvéru.

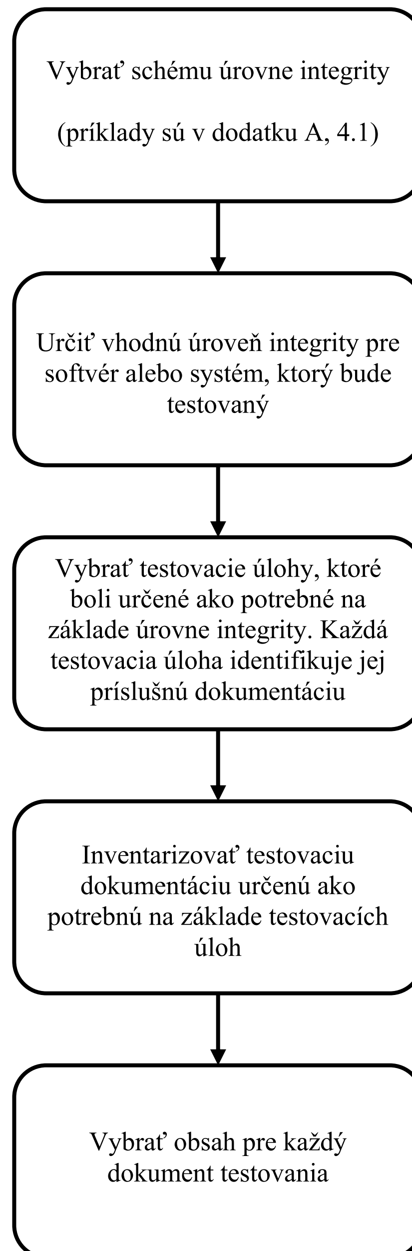
Schému integrity a definovanie jednotlivých úrovni integrity si môže stanoviť organizácia vykonávajúca testovanie, alebo môže prevziať úrovne integrity uvedené v norme. Úrovne integrity uvedené v norme sú definované na základe závažnosti dôsledkov nesprávneho fungovania softvéru a ich nápravy.

- Úroveň integrity 1 (zanedbateľné dôsledky) – softvér musí fungovať správne a zamýšľaná funkcia softvéru môže mať pri prevádzke zanedbateľné dôsledky. Náprava sa nevyžaduje.
- Úroveň integrity 2 (málo významné dôsledky) – softvér musí fungovať správne a zamýšľaná funkcia softvéru môže mať pri prevádzke málo významné dôsledky. Úplná náprava je možná.
- Úroveň integrity 3 (kritické dôsledky) – softvér musí fungovať správne a zamýšľané použitie softvéru/softvérového systému môže mať pri prevádzke kritické dôsledky (trvalé poškodenie, degradácia systému, poškodenie životného prostredia, ekonomické alebo sociálne dôsledky). Čiastočná náprava je možná.
- Úroveň integrity 4 (katastrofálne dôsledky) – softvér musí fungovať správne a zamýšľaná funkcia softvéru môže mať pri prevádzke katastrofálne dôsledky (straty na životoch, strata systému, poškodenie životného prostredia, ekonomické alebo sociálne dôsledky). Náprava nie je možná.

1.7.1.2 Použitie normy IEEE 829-2008

Obrázok 14. znázorňuje možné použitie tejto normy v plnom rozsahu. Začína vývojom alebo použitím schémy úrovní integrity, následne stanovuje úroveň integrity softvérového systému a v ďalšom kroku určuje vypracovanie zoznamu všetkých testovacích úloh. V poslednom kroku sa stanovia vstupy a výstupy pre každú testovaciu úlohu. Vstupy a výstupy stanovené pre jednotlivé úlohy môžu byť skompilované do zoznamu potrebných dokumentov. Následne sa vytvorí zoznam možných položiek obsahu z ktorých sa niektoré prijmú a niektoré zamietnu. Po dokončení zoznamu položiek obsahu

pre každý dokument je potrebné identifikovať všetky položky zoznamu, ktoré sú zdokumentované niekde inde.



Obr. 14: Použitie normy IEEE 829-2008 v plnom rozsahu [31]

2 TECHNIKY A METÓDY TESTOVANIA

2.1 Rozdelenie testovania

V súčasnosti sa v praxi pri testovaní softvéru používajú rozličné metódy a techniky testovania, ktoré sú vytvárané na základe rôznych pohľadov a prístupov k testovaniu softvéru ako takému.

Prvý spôsob klasifikácie testovania je založený na úrovni znalostí, ktoré má tester o testovanom softvéri. Na základe tohto prístupu poznáme:

- Testovanie bielej skrinky
- Testovanie čiernej skrinky

Testovanie bielej skrinky je testovanie softvéru so znalosťou kódu a jeho vnútorných princípov, ktoré užívateľ nepozná (Page a kol. [16]).

Podľa Pattona [1] pri testovaní bielej skrinky má tester prístup ku zdrojovému kódu programu a jeho skúmanie mu môže pomôcť pri testovaní. Z pohľadu do vnútra môže tester odhadnúť, či niektoré údaje povedú častejšie alebo menej často k chybe a podľa týchto informácií môže lepšie vykonávať ďalšie testovanie. Autor uvádza, že toto testovanie prináša jedno riziko, a síce, že tester „ušije testovanie na mieru" programovému kódu a neotestuje softvér objektívne.

Page a kol. [16] uvádzajú, že testovacie prípady založené výlučne na testovaní bielej skrinky sú síce veľmi dôsledné, ale netestujú reálne používanie softvéru užívateľom.

Testovanie čiernej skrinky je testovanie softvéru bez akejkoľvek znalosti jeho kódu alebo jeho vnútorného fungovania (Page a kol. [16]).

Podľa Pattona [1] pri testovaní čiernej skrinky tester vie len to, čo má predložený softvér vykonávať. Nemôže sa pozrieť do vnútra skrinky a teda nevie, ako softvér pracuje vo vnútri. Ak napíše údaj na vstup, tak dostane zodpovedajúci údaj na výstupe. Nevie však, prečo sa práve tento výsledok objavil. Môže ho iba pozorovať.

Page a kol. [16] považujú testovanie čiernej skrinky za užitočnú metódu simulujúcu reálny spôsob použitia softvéru. Naopak, striktné používanie tejto metódy vedie k nadmernému testovaniu niektorých častí softvéru a k nedostatočnému otestovaniu niektorých iných častí.

Druhý spôsob klasifikácie testovania je založený na tom, či je pri testovaní potrebné softvér spustiť. Poznáme nasledovné testovania:

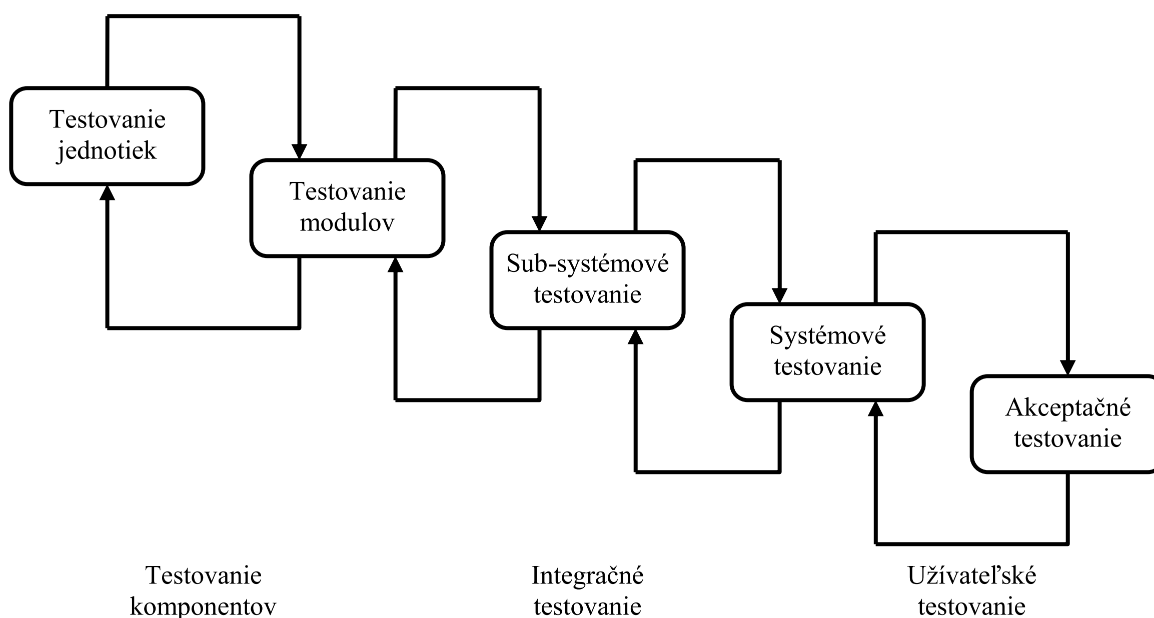
- Statické testovanie - testuje softvér, ktorý nie je spustený. To znamená, že testovaný softvér len prezeráme a kontrolujeme (Patton, [1]).
- Dynamické testovanie – testuje softvér, ktorý je spustený a pracuje sa s ním (Patton, [1]).

Tretí spôsob klasifikácie je založený na tom, kto testovanie softvéru vykonáva. Poznáme nasledovné testovania:

- Manuálne testovanie – vykonávané testerom.
- Automatické testovanie – vykonávané softvérom.

Štvrtý spôsob klasifikácie testovania je založený na tom, v akej etape vývoja softvéru sa testovanie vykonáva a v akom časovom horizonte prebieha. Testovací proces má podľa Somervilla [29] týchto päť stupňov:

- Testovanie jednotiek – každý komponent je testovaný nezávisle, bez ďalších komponentov v danom softvéri.
- Testovanie modulov – modul pozostávajúci so vzájomne súvisiacich komponentov môže byť testovaný bez ďalších modulov softvéru.
- Sub-systémové testovanie – zameriava sa na moduly, ktoré sú integrované do subsystémov, nakoľko väčšina problémov vo veľkých softvérových systémoch vzniká v dôsledku chýb vo vzájomnom prepojení modulov.
- Systémové testovanie – testovanie softvéru ako celku, s cieľom nájsť chyby, ktoré vyplývajú z nepredvídateľných interakcií subsystémov a ich vzájomného prepojenia.
- Akceptačné testovanie – finálna etapa testovacieho procesu pred tým ako je softvér zaradený do užívania. Softvér je testovaný pomocou dát, ktoré do softvéru dodá budúci užívateľ, t.j. nepoužívajú sa simulované testovacie dáta. Akceptačné testovanie tak dokáže odhaliť chyby a nedostatky v definovaných požiadavkách na softvér, pretože skutočné dáta preveria softvér iným spôsobom ako testovacie dáta.



Obr. 15: Testovací proces podľa Somervilla [29]

2.2 Zásady testovania

Testovací proces je nástroj, pomocou ktorého sa realizuje stratégia testovania. V súvislosti s tým testovací tím určí testy a metódy ich vykonania. Z uvedeného vyplýva, že testovací proces je vopred plánovaný proces. To má dve nasledovné výhody:

- testeri nemusia vytvárať samotný proces, pretože ten ako taký už existuje,
- všetci testeri postupujú podľa rovnakého plánu testovania a môžu priebežne pracovať na jeho zdokonalení.

Cieľom testovania ako projektu je jeho úspešné zavŕšenie, ktoré však nie je možné bez kvalitného návrhu testovania, respektíve bez jeho dôkladného naplánovania. Okrem plánovania je potrebné aj určiť aké typy testovania najefektívnejšie overia funkcionalitu softvéru a potvrdia správne ošetrenie chybových stavov. Page a kol. [16] za najdôležitejší aspekt návrhu testov považujú predvídanie požiadaviek a očakávaní zákazníka a následne vytváranie takých testov, ktoré týmto požiadavkám zodpovedajú. Dobrý návrh testu často začína oponentúrou, alebo kritikou návrhu softvéru.

Perry [28] uvádza šesť všeobecných zásad testovania softvéru, ktoré v praxi prispeli k výraznému zlepšeniu testovania softvéru:

1. Testovanie softvéru by malo znížiť riziká vzniku chýb pri vývoji softvéru.
2. Testovanie by malo byť vykonané efektívne.
3. Testovanie by malo odhaliť chyby.
4. Testovanie by malo byť vykonané ekonomicky.
5. Testovanie by malo prebiehať počas celého životného cyklu vývoja softvéru.
6. Testovanie by malo testovať štruktúru a funkčnosť súčasne.

2.2.1 Zásada 1.

Testovanie softvéru by malo znížiť riziká vzniku chýb, pri vývoji softvéru. Riziko je pravdepodobnosť, že sa vyskytnú neželateľné udalosti. Tieto zabránia úspešnej realizácii projektu. Kontrola je prostriedok, ktorý organizácia používa na minimalizáciu rizík. Testovanie softvéru je kontrolou, ktorá prispieva k eliminácii a minimalizácii rizík. Účelom kontroly, akou je aj testovanie softvéru, je poskytnúť manažmentu informácie aby mohol lepšie reagovať na rizikové situácie. Úlohou testerov je teda vyhodnocovať riziká a reportovať výsledky manažmentu. Podľa tejto zásady testeri vyvíjajú stratégie orientované na riziká, ktoré sú podľa manažmentu dôležité. Čím väčší je rozsah rizík, tým väčšie zdroje testovanie vyžaduje.

2.2.2 Zásada 2.

Testovanie by malo byť vykonané efektívne. Efektívnosť znamená dosiahnutie maximálneho účinku pomocou minimálnych zdrojov. Ak je testovací plán dobre definovaný, odchýlky v cene za vykonanie testovacích úloh od testera k testerovi by mali byť veľmi nízke. Cieľom testovacieho procesu z pohľadu efektívnosti je:

- testovací proces by mal redukovať odchýlky v jeho vykonávaní jednotlivými testermi,
- testovací proces by mal redukovať odchýlky prostredníctvom priebežného procesu zlepšovania.

2.2.3 Zásada 3.

Testovanie by malo odhaliť chyby. Ide o odhalenie a elimináciu chýb a odchýlok od očakávaného výsledku. Z tohto hľadiska sa uvádzajú dva typy chýb:

- odchýlka od špecifikácií – chyba z pohľadu tvorcu softvéru,
- odchýlka od toho čo sa očakáva – chyba z pohľadu používateľa.

Tieto chyby zvyčajne spadajú do troch kategórií:

- špecifikácie boli implementované nesprávne, táto chyba je odchýlkou od toho, čo chce zákazník,
- špecifikované, alebo očakávané požiadavky nie sú zapracované do softvéru – táto chyba je odchýlkou od špecifikácie a dokazuje, že špecifikácia nebola implementovaná, alebo že zákazník identifikoval požiadavku počas alebo až po vytvorení softvéru,
- požiadavka zapracovaná do softvéru nebola špecifikovaná – táto chyba je odchýlkou od špecifikácií a aj keď daná vlastnosť softvéru môže byť pre užívateľa želateľnou, je považovaná za chybu.

2.2.4 Zásada 4.

Testovanie by malo byť vykonané ekonomicky. Náklady spojené s identifikáciou a opravou chýb rastú exponenciálne s postupom celého projektu. Chyba, ktorá je identifikovaná aj opravená v tej istej etape projektu, má v porovnaní s rovnakou chybou opravenou po uvedení do prevádzky 100-krát nižšie náklady. Testovanie by preto malo začať v prvej etape životného cyklu a pokračovať celým životným cyklom softvéru. Takto je možné výrazne zredukovať náklady spojené s testovaním.

2.2.5 Zásada 5.

Testovanie by malo prebiehať počas celého životného cyklu vývoja softvéru. Testovanie počas životného cyklu znamená prebiehajúce testovanie riešení dokonca aj potom ako boli dokončené plány softvéru a testovaný softvér bol implementovaný. Tímy testerov vykonávajú testovania v niekoľkých bodoch vývojového procesu aby identifikovali chyby v najskoršom možnom okamihu. Testovanie počas životného cyklu sa musí riadiť presným rozvrhom, ktorý vymedzuje ukončenie testov pre rôzne etapy vývoja.

Rovnako musí stanoviť presné poradie v akom sú dodávané hotové komponenty softvéru a primerané testy, ktoré sa budú na nich vykonávať.

V okamihu keď začína celý projekt, začína sa proces vývoja softvéru a rovnako aj proces jeho testovania. To znamená, že testovací a vývojový tím začínajú pracovať v rovnakom čase a s rovnakými informáciami. Vývojový tím definuje a zdokumentuje požiadavky pre účely implementácie a testovací tím použije tieto požiadavky za účelom testovania softvéru.

2.2.6 Zásada 6.

Testovanie by malo testovať štruktúru a funkčnosť súčasne. Štrukturálne testovanie sa niekedy nazýva testovanie bielej skrinky a prednostne využíva verifikačné techniky. Funkčné testovanie sa niekedy nazýva testovanie čiernej skrinky a využíva takmer výlučne validačné techniky.

Ak vývojový tím vytvorí blok zdrojového kódu, ktorý umožní systému spracovávať informácie určitým spôsobom, testovací tím ho štrukturálne verifikuje prečítaním kódu a danej systémovej štruktúry, pričom overí, či zdrojový kód dokáže adekvátne pracovať. Ak dokáže adekvátne pracovať, je začlenený do softvérového systému. Následne je spustená jeho aplikácia, ktorá validuje zdrojový kód.

Použitie verifikácie na vykonanie štrukturálneho testovania zahŕňa:

- revíziu realizovateľnosti – testy pre tento štrukturálny prvok by verifikovali logiku fungovania komponentov v softvéry,
- revíziu požiadaviek – verifikujú softvérové vlastnosti, napríklad akú záťaž vyvíjaný softvér zvládne.

Funkčné testovanie využíva validačné testy. K nim patria:

- testovanie jednotiek,
- integračné testovanie,
- systémové testovanie,
- akceptačné testovanie.

2.3 Techniky testovania

Testeri pri testovaní softvéru vykonávajú funkčné alebo štrukturálne testy. Keď vykonávajú funkčné testy, používajú takmer výlučne validačné techniky. Podľa Perryho [28] sa validačné techniky používajú výlučne na fyzické testovanie, ktorého cieľom je zistiť, či nastane predpokladaný výsledok.

Naopak, štrukturálne testy prevažne využívajú verifikačné techniky. Podľa Perryho [28] sa verifikačné techniky používajú na potvrdenie správnosti systému pomocou revízie jeho vnútornej štruktúry a logiky.

V praxi sa na validáciu celého softvérového systému najčastejšie používajú obidve techniky testovania súčasne. Verifikačné a validačné techniky sa vzájomne nevylučujú. To vo výraznej miere prispieva k efektívnosti testovania.

2.3.1 Techniky funkčného testovania

Techniky funkčného testovania sú definované ako systematické postupy, ktoré pomáhajú pri podrobnom skúmaní funkčných vlastností a schopností softvéru. Pri týchto technikách sa zvyčajne využíva už samotné užívateľské rozhranie softvéru. Vo väčšine dostupnej literatúry sú tieto techniky použité na návrhy testov založených na modeloch čiernej skrinky. Page a kol. [16] uvádza, že tieto techniky používajú tiež aj na návrhy testov založených na modeloch bielej skrinky. Výhodou techník funkčného testovania je, že pri ich vhodnom použití sa znižuje pravdepodobnosť neodhalenia chyby. Ide však vždy hlavne o schopnosti samotného testera, ktorý musí zvoliť a použiť správnu techniku v správnej situácii. Musí mať dobré znalosti o vyvíjanom softvéri a o jeho budúcom spôsobe používania. Treba však zdôrazniť, že techniky funkčného testovania sú len nástroje, ktorými tester dokáže získať dôležité informácie o schopnostiach a vlastnostiach softvéru, a ktoré slúžia na odhalenie možných chýb.

Perry [28] uvádza nasledovné výhody funkčného testovania:

- Simuluje skutočné používanie softvéru.
- Neberie do úvahy vnútornú štruktúru softvéru.

Autor poukazuje aj na nevýhody funkčného testovania:

- Obsahuje potenciál opomenutia logických chýb v softvéri.
- Prináša riziko nadbytočného testovania.

Page a kol. [16] uvádzajú ako príklady často používaných techník funkčného testovania nasledovné:

- Rozdeľovanie tried ekvivalencie
- Analýza okrajových hodnôt
- Kombinatorická analýza

2.3.1.1 Rozdeľovanie tried ekvivalencie (RTE)

Technika rozdeľovania tried ekvivalencie je nástroj, ktorý umožňuje systematicky vyhodnocovať vstupné alebo výstupné premenné pre každý parameter softvéru.

Na dosiahnutie maximálnej efektivity tejto techniky je nutné vykonať podrobnú analýzu hodnôt premenných pre každý parameter. Pred samotným použitím RTE techniky musíme hodnoty týchto premenných precízne analyzovať, modelovať a následne určiť samotné podmnožiny – triedy validných hodnôt a triedy nevalidných hodnôt.

- Triedy validných hodnôt – obsahujú hodnoty, ktoré dávajú korektný výsledok, prípadne nespôsobujú chybu.
- Triedy nevalidných hodnôt – obsahujú hodnoty, ktoré nám v ideálnom prípade vyvolajú iba chybové hlásenie.

Príklad rozdelenia tried validných a nevalidných hodnôt je uvedený v Tabuľke 1.

RTE testy je možné odvodiť z jednotlivých zjednotení tried validných hodnôt, až kým nie sú všetky triedy použité. Následne je vykonané samostatné vyhodnotenie každej triedy nevalidných hodnôt.

Čím menej testeri vedia o vyvíjanom softvéri, tým viac narastá pravdepodobnosť nesprávneho použitia tejto techniky a narastá aj riziko opomenutia významnej chyby alebo narastie počet nadbytočne vykonávaných testov.

Technika rozdeľovania tried ekvivalencie je časovo náročná na samotnú prípravu testov, ale má niekoľko podstatných výhod:

- Vedie testera k vykonaniu detailnejšej analýzy testovanej funkcionality softvéru a používaných vstupných a výstupných parametrov.
- Pomáha testerovi identifikovať krajné prípady, ktoré by mohli inak zostať nepovšimnuté.
- Poskytuje zrozumiteľnú evidenciu toho, ktoré vstupné hodnoty boli testované a ako boli testované.

- Systematicky zvyšuje efektivitu testovania, čo napomáha znižovať riziko neobjavenia chyby.
- Zvyšuje efektivitu testovania odstraňovaním nadbytočných testov.

Vstup / Výstup	Validné triedy hodnôt	Nevalidné triedy hodnôt
Mesiac	V1: mesiace s 30 dňami V2: mesiace s 31 dňami V3: február	N1: >= 13 N2: <= 0 N3: neceločíselná hodnota N4: prázdny reťazec N5: viac ako 3 číslice N6: >= 32
Deň	V4: 1 - 30 V5: 1 - 31 V6: 1 - 28 V7: 1 - 29	N7: <= 0 N8: neceločíselná hodnota N9: prázdny reťazec N10: viac ako 3 číslice
Rok	V8: 1582 - 9999 V9: neprestupný rok V10: prestupný rok V11: storočie – neprestupný rok V12: storočie – prestupný rok	N11: <= 1581 N12: >= 10000 N13: neceločíselná hodnota N14: prázdny reťazec N15: viac ako 5 číslic
Výstup	V13: 1.1.1582 - 31.12.9999	N16: <= 31.12.1581 N17: >= 1.1.10000

Tab. 1: Rozdelenie tried ekvivalencie pre vstupné a výstupné parametre pre softvér, ktorý používa Gregoriánsky dátum

2.3.1.2 Analýza okrajových hodnôt (AOH)

Analýza okrajových hodnôt je pravdepodobne najznámejšou technikou funkčného testovania. Pri vývoji softvéru vzniká veľké množstvo chýb na krajných hodnotách intervalov hodnôt. Ak sa technika AOH použije s technikou RTE, môže byť veľmi efektívna pri systematickej analýze okrajových hodnôt nezávislých vstupných a výstupných parametrov. Je najmä užitočná pri odhaľovaní týchto typov chýb:

- zlé nastavenie umelých obmedzení pre dátový typ,
- použitie nesprávnych relačných operátorov,
- prekročenie hraníc dátových typov,
- problémy s cyklami,
- chyby v indexovaní (počítanie od nuly vs. od jednotky).

2.3.1.3 Kombinatorická analýza

Kombinatorická analýza je technika, ktorá sa používa na testovanie vzájomného pôsobenia vzťahov vzájomne závislých parametrov. Predstavuje najefektívnejšie riešenie pre testovanie zložitých funkčných vzťahov medzi vstupnými veličinami. Slúži na výber efektívnej podmnožiny zo sady všetkých mysliteľných testov. Pri správnom aplikovaní tejto metódy, táto metóda prináša nasledovné výhody:

- odhaľuje väčšinu chýb spôsobených vzájomným pôsobením vstupných parametrov,
- zabezpečuje lepšie pokrytie riadiacich štruktúr,
- môže výrazne znížiť celkové náklady na testovanie.

V kombinatorickej analýze sú definované dva základné prístupy k testovaniu vzájomného pôsobenia vzťahov:

- Náhodné (ad hoc) metódy
  - kvalifikovaný odhad,
  - náhodný výber.
- Systematické metódy
  - Testovanie každej premennej, alebo každej voľby (KV) – každá možnosť sa otestuje aspoň jeden krát.
  - Testovanie základnej voľby (ZV) – zvolí sa určitá kombinácia vstupov pre základný test, ktorá zvyčajne prezentuje najtypickejšiu kombináciu vstupov. Ďalšie testy sa vytvárajú vždy zmenou len jedného vstupu pričom ostatné zostávajú nezmenené.
  - Metóda ortogonálnej sústavy (OS) – každý nezávislý parameter pri aplikovaní OS sa musí dostať do rovnakého počtu rôznych stavov. Tieto stavy sú následne priradené prvkom sústavy, kde každú n-ticu hodnôt pokrýva rovnaký počet testov.
  - Testy kombinácií/n-tíc – s použitím matíc pokrytia tvorí jednu z najefektívnejších metód.
  - Úplne testovanie.

Kombinatorická analýza nie je ale vhodná pre každý typ softvéru. Je neefektívna pri nezávislých parametroch, ktoré sa navzájom neovplyvňujú. Taktiež je nevhodná na testovanie matematických výpočtov, nevhodná na testovanie kde sa postupne zadávajú vstupné parametre, alebo na testovanie vstupov kde sa vyžadujú sekvenčné operácie.

2.3.2 Techniky štrukturálneho testovania

Techniky štrukturálneho testovania na rozdiel od techník funkčného testovania spadajú výhradne pod model bielej skrinky. V tomto modely návrh testov vychádza z internej štruktúry testovaného softvéru. Tester zostavujúci testy musí dobre poznať použitý programovací jazyk a musí sa vedieť dobre orientovať aj v samotnom zdrojom kóde. Tieto techniky nie je vhodné používať ako východiskové alebo ako jediný postup testovania softvéru. Sú určené k podpore a dopĺňajú ostatné testovacie postupy a metódy. Poskytujú nám ďalší pohľad na riešený problém. Patria medzi časovo náročnejšie testy a kladú aj vysoké nároky na schopnosti testerov. Výrazne znižujú riziko vzniku chýb pri tvorbe zložitých softvérov a pomáhajú zabezpečiť vysokú úroveň spoľahlivosti softvéru.

Perry [28] uvádza nasledovné výhody štrukturálneho testovania:

- Umožňuje testovať logiku softvéru.
- Umožňuje testovať štrukturálne vlastnosti, ako napríklad efektivitu zdrojového kódu.

Autor poukazuje aj na nevýhody štrukturálneho testovania:

- Nezaručuje splnenie požiadaviek užívateľa.
- Nie vždy dokáže napodobniť situácie z reálnej prevádzky softvéru.

Page a kol. [16] uvádzajú ako príklady často používaných techník štrukturálneho testovania nasledovné:

- Testovanie blokov
- Testovanie rozhodnutí
- Testovanie podmienok
- Testovanie základných ciest

2.3.2.1 Testovanie blokov

Testovanie blokov je technika testovania, ktorá sa využíva pre jednoduché štrukturálne testovanie funkcií. Samotné testovanie je založené na vykonávaní sekvenčných skupín príkazov, ktoré tvoria ucelené bloky zdrojového kódu. Tieto bloky nesmú obsahovať vetvenie alebo volanie funkcií. Táto technika sa osvedčila hlavne pri testoch kontrolujúcich správnosť vykonávania príkazov pri použití *switch/case*. Naopak, pre robustné štrukturálne testy je nevhodná, lebo v určitých prípadoch nemusí zachytiť

dôležité prechody funkcií. Pri použití tejto techniky je veľmi ľahké prehliadnuť možné chyby.

2.3.2.2 Testovanie rozhodnutí

Hlavným cieľom techniky testovania rozhodnutí je návrh testov, ktoré overujú dopady kladného aj záporného výsledku danej podmienky. Zameriava sa na podmieňovacie príkazy, kde výsledok boolovského výrazu môže spôsobiť rozvetvenie toku riadenia. Výhodou tejto techniky je, že na rozdiel od techniky testovania blokov, výsledky lepšie sledujú skutočné priebehy tokov riadenia v rámci testovanej funkcie. Táto technika je efektívna pri testovaní jednoduchých podmieňovacích príkazov ako sú *if* alebo cykly. Na testovanie zložitejších podmieňovacích výrazov, ktoré obsahujú viac podmienok, je vhodnejšie použiť techniku s názvom testovanie podmienok.

2.3.2.3 Testovanie podmienok

Technika testovania podmienok lepšie sleduje tok riadenia v prípadoch, keď sú v zdrojovom kóde použité podmieňovacie príkazy zložené z viac podmienok a logických operátorov. Táto technika zahŕňa testovanie blokov, ako aj testovanie rozhodnutí a zároveň rieši aj použitie zložených podmienok. Netestujú sa ale všetky možné kombinácie. Je potrebné aby programátori namiesto vnorených podmieňovacích príkazov používali na zápis jeden prehľadnejší výraz, ktorý je poskladaný z logických operátorov AND a OR.

2.3.2.4 Testovanie základných ciest

Technika testovania základných ciest má preveriť každý jeden prechod v programe. Toto je náročné hlavne pri cykloch, pretože každá ďalšia iterácia je považovaná za cestu. Otestovanie všetkých možných ciest je v skutočnosti nepraktické a nesmierne časovo náročné. S riešením tohto problému prišiel Thomas McCabe, ktorý definoval vzťah medzi cyklomatickou zložitosťou funkcie a počtom testov potrebných na jej dostatočné otestovanie.

Vzorec (I) pre výpočet cyklomatickej zložitosti má tvar:

v(G) = (počet hrán) – (počet uzlov) + 2

Existuje aj jednoduchšia verzia vzorca (II) v tvare:

v(G) = (počet podmienok) + 1,

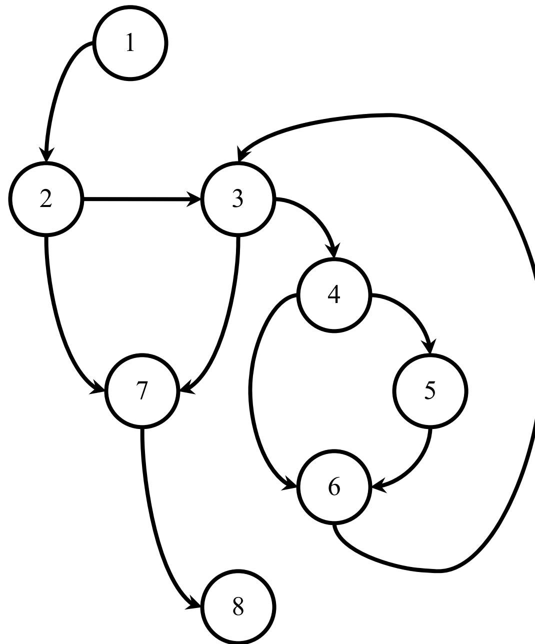
kde v(G) udáva počet ciest v softvéri, ktoré treba otestovať, aby bol vykonaný každý jeden príkaz. Je to vlastne počet testov potrebných na pokrytie všetkých možností.

Hore uvedený vzorec sme aplikovali na výpočet počtu testov potrebných na otestovanie jednoduchého softvéru s troma podmienkami (pozri Obr.16).

```
// Funkcia spočíta výskyt písmena C v reťazcoch začínajúcich
písmenom A
      private static int SpocitajC (string retazec)
      {
(1)   int index = 0, i = 0;
      char A = 'A', C = 'C';
      char[] pole = retazec.ToCharArray();
(2)   if (pole[index] == A)
      {
(3)         while (++index < pole.lenght)
            {
(4)               if  (pole[index] == C)
                  {
(5)               i = i + 1;
(6)               }
(7)         }
(8)   return i;
      }
```

Obr. 16: Zdrojový kód softvéru s tromi podmienkami

Na otestovanie všetkých ciest v softvéri je potrebné vykonať štyri testy. Na zdrojový kód softvéru (Obr.16) aplikujeme vzorec (II) a na tokový graf softvéru (Obr.17) aplikujeme vzorec (I).



Obr. 17: Tokový graf softvéru s tromi podmienkami

Výsledné cesty v softvéri:

(C1: 1-2-7-8, C2: 1-2-3-7-8, C3: 1-2-3-4-6-3-7-8, C4: 1-2-3-4-5-6-3-7-8)

2.4 Metódy testovania

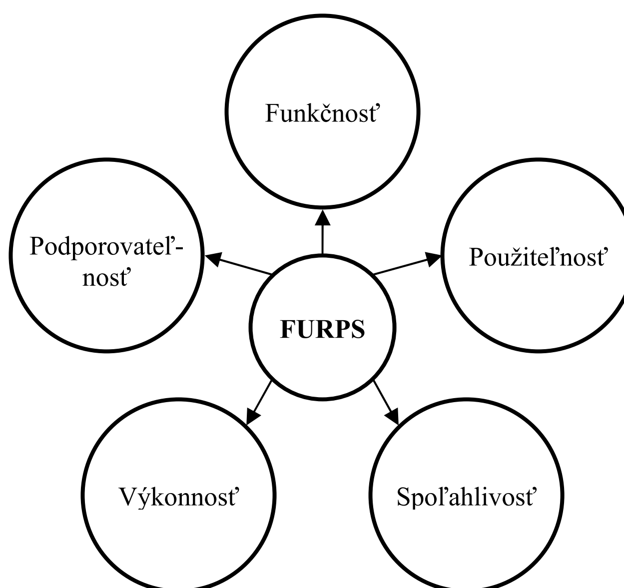
2.4.1 Metóda FURPS

Za vznikom tejto metódy stojí firma Hewlett-Packart, ktorá ju vyvinula na základe potreby definovať a overiť kvalitu vyvíjaného softvéru. Ako prví ju verejne publikovali Robert Grady a Deborah Caswell v knihe „Software Metrics: Establishing a Company-Wide Program“ v roku 1987. Názov metódy tvoria začiatočné písmena názvov jednotlivých požiadaviek.

Metóda na najvyššej úrovni definuje, čo by malo byť v procese vývoja softvéru hodnotené, ale vôbec nešpecifikuje, akým spôsobom majú byť tieto oblasti hodnotené [25]. Podľa tejto metódy každý kvalitný softvérový systém musí spĺňať päť základných požiadaviek :

- Funkčnosť (functionality) – zameriava sa na hlavné funkcionality softvéru, jeho schopnosti a bezpečnosť.

- Použiteľnosť (usability) – hodnotí softvér z pohľadu užívateľa, jeho celkový dojem zo softvéru, dodanej dokumentácie a príručiek.
- Spoľahlivosť (reliability) – hodnotí počet a závažnosť chýb, presnosť spracovávania údajov na vstupoch a výstupoch softvéru.
- Výkonnosť (performace) – hodnotí rýchlosť reakcií softvéru, taktiež sleduje i technické parametre ako sú: zaťaženie zdrojov OS, zaťaženie počítačovej siete, zaťaženie jednotlivých hardvérových komponentov.
- Podporovateľnosť (supportability) – hodnotí oblasť rozšíriteľnosti softvéru, jeho možnosti údržby a konfigurovania.



Obr. 18: Metóda FURPS

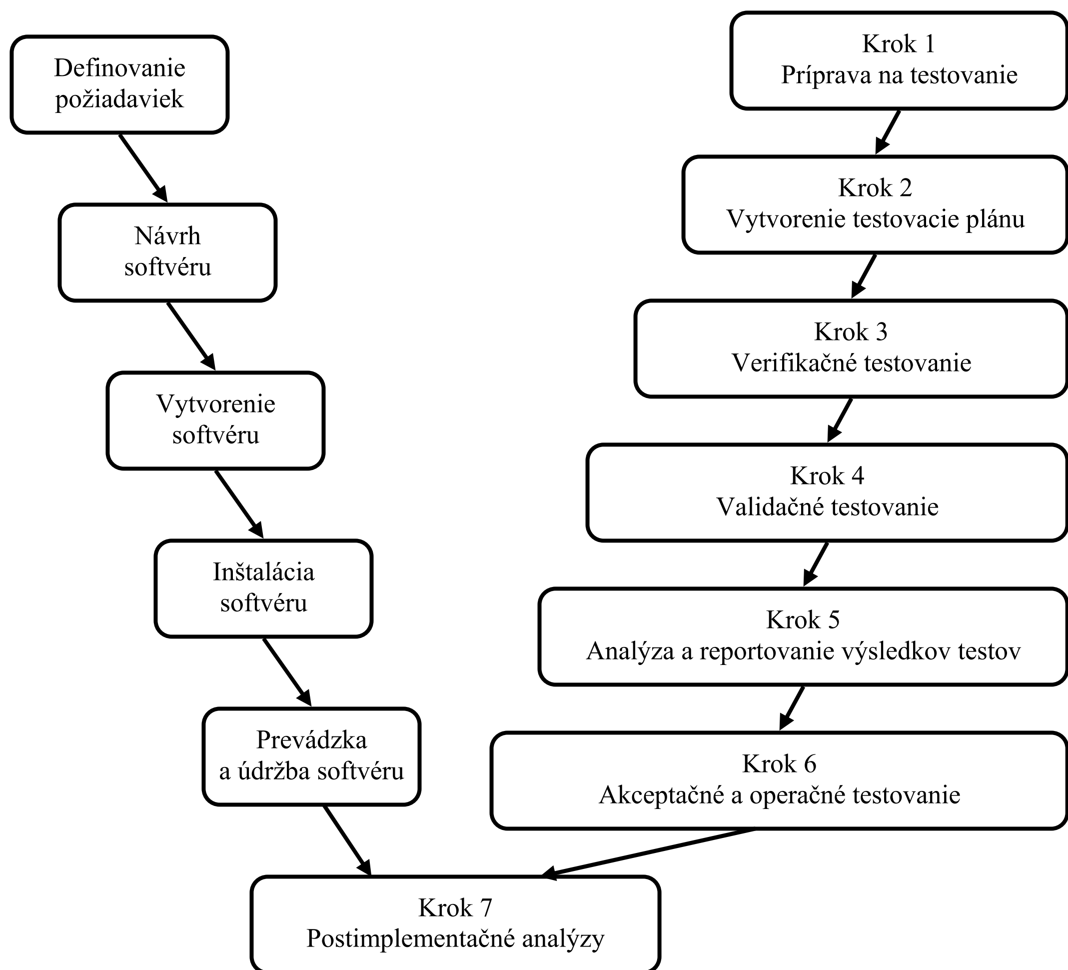
Na každú z týchto požiadaviek firmy, ktoré sa zaoberajú vývojom a testovaním softvéru, majú vypracované účinné testy. Tie zabezpečujú úspešné splnenie sledovanej požiadavky na testovanom softvéri. Pre jednotlivé požiadavky uvádza firma Unicorn [13] tieto testy:

- Funkčnosť
  - Funkčný test - overuje funkčnosť voči požiadavkám na vstupné a výstupné správanie systému.
  - Bezpečnostný test - preveruje, či sú dáta a systém prístupné iba určeným užívateľom.

    - Objemový test - preveruje, či je systém schopný v danom okamžiku spracovať veľký objem dát.
- Použiteľnosť
    - Test použiteľnosti - overuje systém z pohľadu koncového užívateľa.
- Spoľahlivosť
    - Integračný test - posudzuje systém z hľadiska odolnosti voči zlyhaniu a technickým požiadavkám (jazyk, pravidlá, využitie zdrojov).
    - Test štruktúry - u webových aplikácií overenie funkcie a vecnej správnosti všetkých odkazov.
    - Stress test - preveruje správanie systému v neštandardných podmienkach (nedostatočné zdroje a pod.).
- Výkonnosť
    - Záťažový test - overenie správania systému pri extrémnej záťaži.
    - Výkonnostný test - sledovanie výkonu systému v definovaných časových úsekoch z pohľadu prístupu k dátam, hláseniu funkcií atď.
    - Porovnávací test - porovnanie výkonu systému s referenčným systémom a jeho záťažou.
    - Konkurenčný test prístupu - overenie správnej funkčnosti systému pri viacnásobnom prístupe užívateľov v danom okamžiku k rovnakému zdroju.
- Podporovateľnosť
    - Konfiguračný test - overuje funkčnosť systému na rozdielne hardvérové alebo softvérové konfigurácie.
    - Inštalačný test - overuje možnosti inštalácie systému na rozdielnych hardvérových alebo softvérových konfiguráciách.

2.4.2 V - model testovanie

V- model reprezentuje proces vývoja softvéru, ktorý môžeme považovať za rozšírenie kaskádového modelu životného cyklu. Podľa Bielikovej [15] to znamená, že jednotlivé etapy architektonického návrhu a integrácie úzko súvisia s testovaním softvéru. Etapy na zostupnej hrane súvisia s návrhom a dekompozíciou softvéru na danej úrovni abstrakcie, zatiaľ čo etapy na vzostupnej hrane súvisia s integráciou softvéru (pozri Obr.19).



Obr. 19: V- diagram, sedem krokov procesu testovania softvéru podľa Perryho [28]

Perry [28] uvádza testovací proces softvéru založený na V- modely, ktorý pozostáva zo siedmich krokov. Podľa autora tento model predstavuje proces vývoja softvéru a súčasne testovací proces. Obidva procesy postupujú spoločne až ku koncu projektu, ktorý končí krokom sedem – postimplementačnou analýzou. Jej cieľom je určiť, či proces vývoja a/alebo testovanie môžu byť vykonané efektívnejšie.

Testovanie softvéru podľa V- modelu pozostáva z nasledovných sedem krokov:

1. Príprava na testovanie
   - Definovanie rozsahu testovania.
   - Zostavenie testovacieho tímu.
   - Zhodnotenie plánu vývoja softvéru – v tomto kroku testeri predkladajú námietky k úplnosti a správnosti plánu vývoja softvéru.
2. Vytvorenie testovacieho plánu
   - Vykonanie analýzy rizík.

- Napísanie testovacieho plánu.

3. Verifikačné testovanie
    - Testovanie požiadaviek na softvér – testeri musia určiť či požiadavky na softvér sú presné a úplné a či si vzájomne neprotirečia.
    - Testovanie návrhu softvéru – testeri hodnotia či návrh bude spĺňať ciele projektu a či bude efektívny a výkonný pre navrhovaný hardvér.
    - Testovanie štruktúry softvéru – skúsenosti ukazujú, že je výrazne lacnejšie identifikovať chyby počas etapy vytvárania štruktúry softvéru než v priebehu dynamického testovania počas validácie.
4. Validačné testovanie
    - Vykonanie validačného testovania – predstavuje testovanie zdrojového kódu v dynamickom stave.
    - Zaznamenanie výsledkov testovania.
5. Analýza a reportovanie výsledkov testov
    - Analyzovanie výsledkov testov.
    - Vytvorenie testovacích reportov.
6. Akceptačné a operačné testovanie
    - Vykonanie akceptačného testovania – umožňuje užívateľom softvéru hodnotiť použiteľnosť softvéru pri vykonávaní jeho každodenných funkcií.
    - Testovanie inštalácie softvéru.
    - Testovanie zmien vykonaných v softvéri – kedykoľvek sa zmenia požiadavky na softvér, musí sa zmeniť aj testovací plán, v dôsledku čoho sa musia otestovať a vyhodnotiť zmeny v softvéri.
7. Postimplementačná analýza – je vykonaná s cieľom dosiahnuť zlepšenie testovania, pričom sa hodnotí efektívnosť testovania na konci každej testovacej úlohy.

Perry [28] odporúča testerom používať V- model testovacieho procesu na testovanie veľkých, komplexných softvérových systémov.

2.4.3 Vývoj riadený testami (TDD)

Testovanie je neodmysliteľnou súčasťou životného cyklu softvéru. V posledných rokoch si obľubu získala programovacia technika, ktorá je známa pod menom „Vývoj riadený testami“ (Test Driven Development, TDD). TDD kladie dôraz hlavne na testovanie vyvíjaného softvéru. Testovanie je vykonávané priebežne ako neoddeliteľná súčasť všetkých aktivít vývoja. Tu testovanie netvorí len jednu vývojovú etapu v životnom cykle, ale tvorí základ celého procesu.

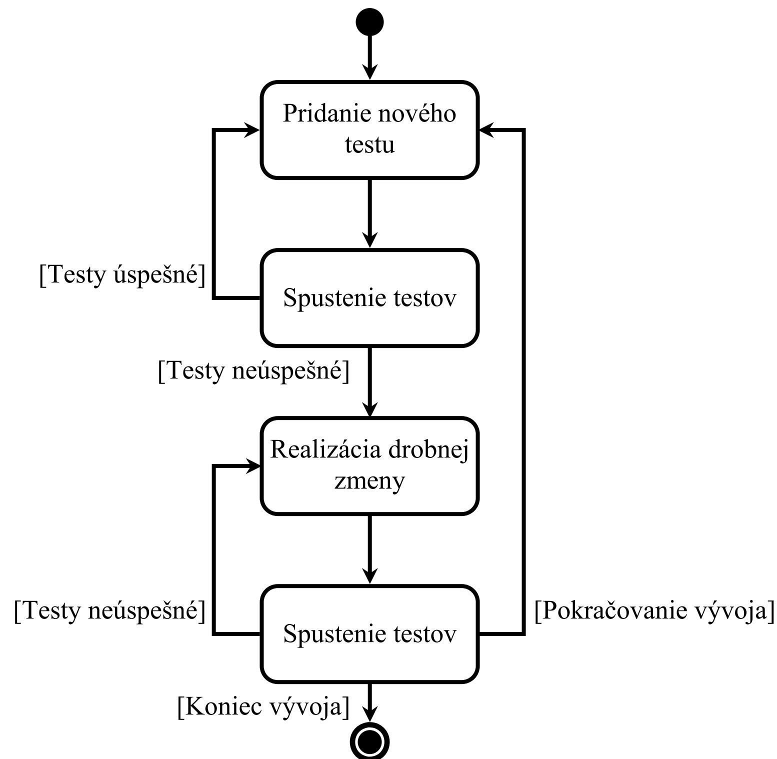
TDD vyžaduje, aby pre každú existujúcu funkcionalitu v zdrojom kóde softvéru bol najprv napísaný test. Test, ktorý má otestovať zdrojový kód, sa preto píše ešte pred napísaním samotného zdrojového kódu. Základným prvkom je nutnosť napísať test, ktorý zlyhá. Až po vytvorení tohto testu sa môže začať tvoriť zdrojový kód - funkcionalita vo vyvíjanom softvéri. Jej vývoj bude úspešný v okamihu, keď táto funkcionalita prvý krát úspešne prejde testom.

Gunderloy [26] udáva, že podľa jeho skúseností s používaním TDD, je zdrojový kód testov viac ako dvakrát väčší ako zdrojový kód samotného softvéru, ktorý sa testuje. Prínos TDD vidí hlavne v dôvere programátora vo svoj napísaný zdrojový kód. Táto dôvera sa v konečnom dôsledku transformuje do zvýšenej produktivity. Ďalším pozitívnym vplyvom je, že programátori pri použití TDD produkujú kvalitatívne lepší zdrojový kód. Písanie testov ich totiž núti premýšľať o tom čo píšu a za akých podmienok to môže zhavarovať.

Kadlec [14] uvádza päť základných etáp vývoja softvéru pri použití techniky TDD:

1. Vytvorenie testu, ktorý zlyhá.
2. Spustenie kompletnej testovacej sady - všetkých doposiaľ napísaných testov. Tu sa musíme presvedčiť, že nový test vytvorený v prvom kroku zlyhá.
3. Vytvorenie funkcionality (vytvorenie zdrojového kódu softvéru). Následne overíme či funkcionalita prejde testom vytvoreného v prvom kroku.
4. Spustenie všetkých doposiaľ napísaných testov na novo napísanej funkcionalite. Ak niektorý test zlyhá, je nutné opraviť zdrojový kód a znovu zopakovať testovanie.
5. Novo vytvorený test sa presunie do testovacej sady. Overí sa, či nedošlo k porušeniu integrity testov a overí sa aj ich logická previazanosť. Taktiež sa odstránia prípadné duplicity.

Týchto definovaných päť krokov sa neustále opakuje počas celého procesu vývoja softvéru. Cykly sú časovo krátke a trvanie jedného cyklu sa vždy predpokladá v minútach. Prebiehajú na úrovni jednotlivých funkcii, prípadne na úrovni modulov.



Obr. 20: UML Diagram aktivít, Test-Driven Development, TDD podľa Amblera [27]

TDD je metodika agilného programovania, ale predovšetkým programovacia technika. Kadlec [14] vo svojej knihe uvádza aj názor zástancov TDD, pre ktorých TDD je najefektívnejšia metóda vývoja softvéru, kde životný cyklus softvéru prebieha v tomto poradí:

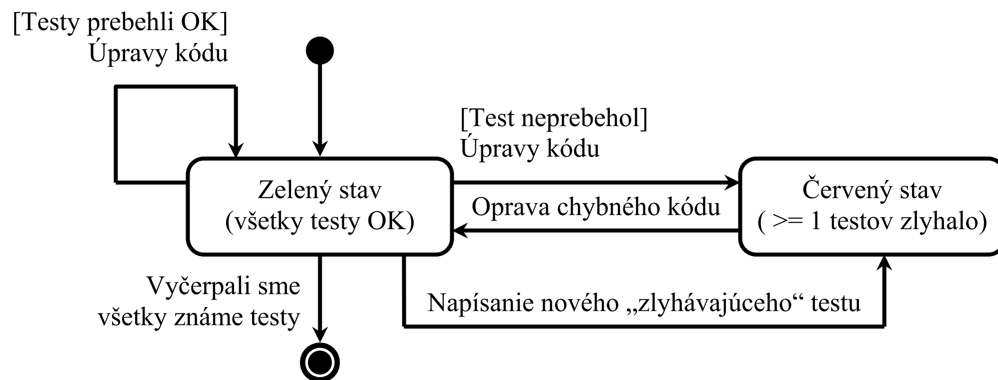
- testovanie,
- implementácia,
- návrh.

Týmto spôsobom vývoja softvéru je možné úspešne eliminovať známu a nepríjemnú etapu hľadania chýb. TDD neuznáva tradičný postup od implementácie k testom, ale správa sa presne naopak. Výsledkom je kvalitný softvér s predvídateľným a dobre prevereným správaním.

Pre programátorov je ale dosť nepohodlné, keď musia najprv napísať test pre ešte neexistujúci zdrojový kód softvéru.

2.4.3.1 Testovanie softvéru pri TDD

Na úspešné nasadenie programovacej techniky TDD je potrebné používať vývojové nástroje, ktoré pri vývoji softvéru podporujú testovanie. Medzi voľne dostupné nástroje môžeme zaradiť: xUnit, JUnit a VBUnit. Proces testovania vyvíjaného softvéru pri použití TDD je graficky zakreslený v jazyku UML nasledovne:



Obr. 21: UML Stavový diagram, Testovanie softvéru pri TDD podľa Amblera [27]

Kadlec [14] uvádza štyri základné body procesu testovania softvéru pri použití programovacej techniky TDD:

1. Vstup do testovacieho procesu – v tomto kroku sa predpokladá, že všetky testy úspešne prešli a vyvíjaný softvér je v poriadku. Zelený stav (všetky testy OK).
2. Ak je vyvíjaný softvér dokončený a plne funkčný, automaticky sa prechádza na posledný bod. V opačnom prípade vykonáme refaktorizáciu už existujúceho kódu, alebo doplníme novú funkcionalitu. Následne opäť vykonáme všetky existujúce testy.
    - Ak všetky testy úspešne prejdú, vraciame sa na začiatok, na prvý bod procesu.
    - Ak niektorý test zlyhá, znamená to, že máme v zmenenom zdrojovom kóde chybu. Červený stav (>= 1 testov zlyhalo). V takomto prípade pokračujeme ďalším bodom.
3. Vykoná sa oprava chybného zdrojového kódu, ktorý vyvolal zlyhanie testu a proces sa vracia o krok späť do druhého bodu.
4. Ak už nie je potrebné doplniť ďalšiu funkcionalitu a vyčerpali sme všetky známe testy, tak môžeme považovať vyvíjaný softvér za úspešne dokončený.

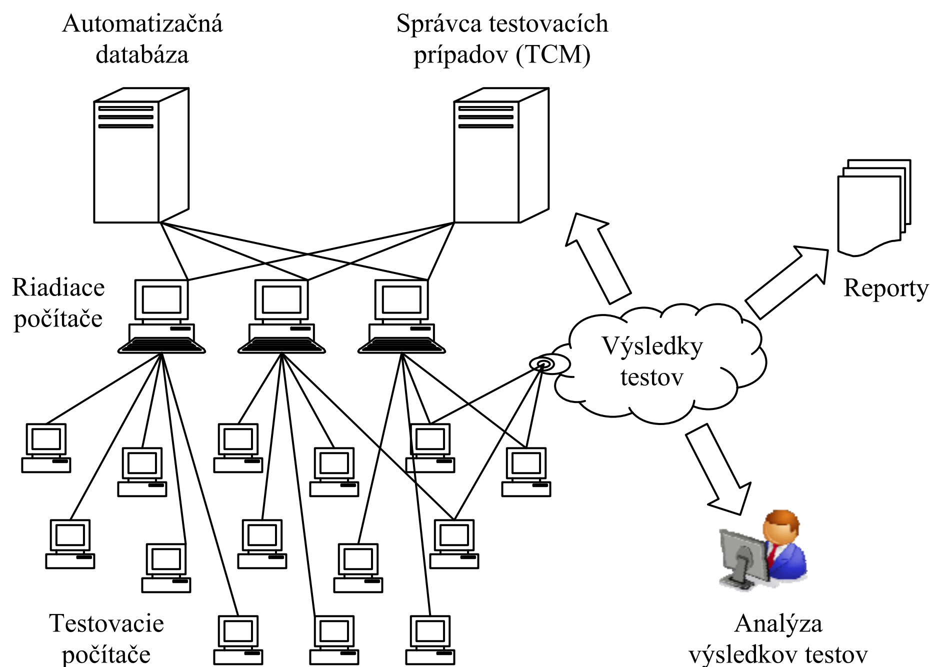
Pri TDD musíme však naďalej používať aj niektoré tradičné typy testovania. Je potrebné vykonávať napríklad testovanie užívateľského rozhrania, akceptačné testovanie, integračné testovanie. Tieto typy testov nie je možné zautomatizovať v dostatočne potrebnej miere, ale ich dôsledné vykonanie je potrebné na dosiahnutie požadovanej kvality vyvíjaného softvéru.

2.4.4 Metóda SEARCH

Testovacia metóda SEARCH patrí do skupiny automatického testovania. Publikovali ju Keith Stobie a Mark Bergman v článku „How to Automate Testing: The Big Picture" v roku 1992. Názov tvoria začiatočné písmena názvov jednotlivých etáp testovacieho procesu [16].

- Nastavenie (Setup) – pripravenie softvéru do stavu, kedy je možné vykonať jednotlivé kroky testu.
- Vykonanie (Execution) – samotné jadro testu, kde sa vykonávajú kroky k overeniu funkcionality a ošetreniu objavených chýb.
- Analýza (Analysis) – vyhodnotenie úspešnosti testu. Ide o najdôležitejšiu a často aj najzložitejšiu etapu testovacieho procesu.
- Reportovanie (Reporting) – spracovanie výsledkov analýzy. Tie sa môžu uložiť do databázy, do súborov a zobraziť na obrazovke.
- Čistenie (Clean up) – opätovné pripravenie softvéru do stavu, kedy je možné vykonať ďalšie testy.
- Dokumentácia (Help) – dôležitá etapa, ktorá podporuje udržiavateľnosť testovacieho prípadu počas jeho životnosti.

V jednoduchých prípadoch sa testovanie spúšťa pomocou testovacieho softvéru alebo skriptu. Ale pri vývoji veľkých softvérových systémov musia tieto procesy bežať v rámci zložitého systému. Ten musí zabezpečovať kompatibilitu a konzistentné výsledky pri každom vykonávaní testov. Pri metóde SEARCH sa buduje celá infraštruktúra, ktorá zabezpečuje automatické vykonávanie testov. Dobre navrhnutý systém umožňuje spustenie celej sady testov po celej vybudovanej infraštruktúre jedným vyslaným spúšťacím povelom.



Obr. 22: Infraštruktúra na automatizáciu testov (Page a kol., [16])

Metóda SEARCH má oddelenú automatizačnú databázu od správcu testovacích prípadov (Test Case Manager, TCM). Výhoda samostatnej automatizačnej databázy je, že informácie o automatizovanom teste, ktoré tvoria napríklad parametre príkazového riadku, uloženie testu a súvisiacich súborov sú úplne oddelené od informácií o testovacích prípadoch.

Pri použití tejto metódy sa správca testovacích prípadov (TCM) spolu s automatizačnou databázou spoja s riadiacimi počítačmi, ktoré riadia prípravu samotných testovacích počítačov. Po príprave každého testovacieho počítača je ihneď na ňom spustená pridelená časť testovania. Výsledky testovania sú následne vyzbierané z jednotlivých testovacích počítačov a sú uložené do TCM. Po spracovaní všetkých výsledkov sú tieto pripravené pre testovací tím na ďalšie posúdenie.

Automatické testovanie môže pri vývoji softvéru ušetriť veľké množstvo času a taktiež financií. Ale v žiadnom prípade nie je možné úplne nahradiť ručné testovanie vykonávané testermi. Ak je potrebné vykonať len určitú malú skupinu testov, je oveľa vhodnejšie využiť schopnosti mysliaceho človeka ako stroja.

3 NÁVRH METODICKÉHO POSTUPU TESTOVANIA SOFTVÉRU S VYUŽITÍM UML

3.1 Plánovanie testovania

Správne naplánované testovanie musí vytvárať priestor pre vzájomnú komunikáciu všetkých zainteresovaných, definovať potrebné prostriedky ako aj časový rozvrh testovania. Testovanie taktiež vyžaduje presnú dokumentáciu, ktorej formát, ak nie je definovaný samotnou spoločnosťou, ktorá testovanie vykonáva, tak sa môže napríklad riadiť normou IEEE 829-2008. Prípadne sa vyberú len určité časti tejto normy, ktoré sa ďalej zapracujú do interných smerníc na testovanie danej spoločnosti.

V literatúre sa stretávame s dvoma prístupmi k vytváraniu testovacieho plánu softvéru:

1. Prístup predstavuje používanie testovacích vzorov, respektíve šablón. Tento prístup kladie dôraz na dokumentáciu testovania. Úpravami testovacieho vzoru môžeme získať testovací plán konkrétneho projektu. Page a kol. [16] považujú testovacie vzory za dôležité, pretože ich testeri môžu využiť pri stanovení celkového zámeru testu a môžu zrozumiteľne a jednoznačne popísať techniky návrhu testu. Testeri v Microsofte používajú ako testovací vzor zjednodušenú šablónu, ktorá obsahuje:
    - Názov šablóny.
    - Problém – stručný popis problému, ktorý tento testovací vzor rieši.
    - Analýza – opis oblasti problému a zdôvodnenie použitia testovacej techniky.
    - Návrh – opisuje ako sa tento testovací vzor používa (ako sa z návrhu testov vyrábajú testovacie prípady).
    - Prognóza – vysvetľuje očakávané výsledky testovania.
    - Príklady – uvádza príklady hľadania chýb.
    - Riziká a obmedzenia – okolnosti a situácie v akých by sa tento testovací vzor nemal používať.
    - Súvisiace vzory – zoznam súvisiacich vzorov (ak také existujú).
2. Prístup k vytváraniu testovacieho plánu kladie dôraz na samotný proces plánovania testovania. Podľa Pattona [1] tento proces musí odpovedať na základné otázky o témach, ktoré by mali prediskutovať, pochopiť a odsúhlasiť všetci členovia projektového tímu testovania. Takéto plánovanie je dynamický proces a podľa

autora je zárukou, že každý člen tímu vie, čo robia ostatní a prečo. Medzi základné témy, ktorým je potrebné sa v priebehu plánovania venovať, patria:

- Stanovenie najvyššej úrovne očakávaní, ktoré má testovací tím od testovania. Tie sú sformulované do stručnej a všetkými odsúhlasenej definície cieľov kvality a spoľahlivosti daného produktu. Tieto ciele musia byť absolútne.
- Rozhodnutie o tom, čo sa bude a čo sa nebude testovať – v procese plánovania sa identifikuje každý komponent softvéru a presne sa stanoví, či sa bude, alebo nebude testovať.
- Stratégia testovania – popisuje postup, pomocou ktorého bude testovací tím testovať daný softvér. Definuje jednotlivé etapy testovania.
- Opis navrhnutých etáp testovania – musia byť stanovené kritéria na základe ktorých je možné deklarovať či jedna etapa testovania skončila a nasledujúca začala. Explicitne sa stanovia kritéria pre vstupy a výstupy jednotlivých etáp.
- Definície základných pojmov, ktoré budú členovia tímu používať (je dôležité aby programátori a testeri chápali pojmy rovnako).
- Požiadavky a prostriedky sumarizujú všetko, čo budeme v priebehu projektu využívať (osoby, vybavenie, priestory a laboratória, softvér, externé spoločnosti a pracovné prostriedky).
- Poverenie testerov konkrétnymi úlohami. Pri plánovaní úloh identifikujeme testerov, ktorí sú zodpovední za jednotlivé oblasti softvéru a za jednotlivé testované funkcie.
- Vzájomné povinnosti medzi skupinami pozostávajú z úloh a hotových častí projektu, ktoré môžu mať vplyv na testovanie. Zvyčajne sú to úlohy, ktoré spadajú do kompetencie niekoľkých osôb.
- Časový plán testov vychádza zo všetkých hore uvedených tém a premieta ich do celkového plánu prác na projekte, pričom presne definuje ich časový rozvrh.
- Testovacie prípady – plán musí uvádzať spôsob zápisu testovacích prípadov, spôsob ich uloženia, používania a údržby.
- Správy o chybách – plán musí presne opisovať proces riadenia chýb, aby každá chyba bola zaznamenaná a opravená.

- o Metriky a štatistiky sú prostriedkami merania a sledovania úspešného napredovania celého projektu a testovania.
- o Identifikácia potenciálnych problémov a rizikových oblastí projektu.

3.1.1 Dokumentácia testovania

Testovacia dokumentácia vymedzuje rámec testovania. Je základom pre prácu testera a jeho komunikácie s inými testermi, resp. zainteresovanými osobami. Testovacia dokumentácia vymedzuje ciele testovania, jeho priebeh, technické požiadavky, riziká, konkretizuje činnosti spojené s testovaním a vytvára priestor pre reportovanie výsledkov testovania. Testovacia dokumentácia taktiež umožňuje opakované spustenie testov, pričom garantuje, že budú vykonané za rovnakých podmienok. Dokumentácia testovania môže pozostávať z niekoľkých dokumentov, ktoré vymedzujú jednotlivé etapy testovania.

Norma IEEE 829-2008 odporúča v závislosti od zvoleného typu testovania použiť niektoré z nasledovných dokumentov:

- Master Test Plan (MTP)
- Level Test Plan (LTP)
- Level Test Design (LTD)
- Level Test Case (LTC)
- Level Test Procedure (LTPr)
- Level Test Log (LTL)
- Anomaly Report (AR)
- Level Interim Test Status Report (LITSR)
- Level Test Report (LTR)
- Master Test Report (MTR)

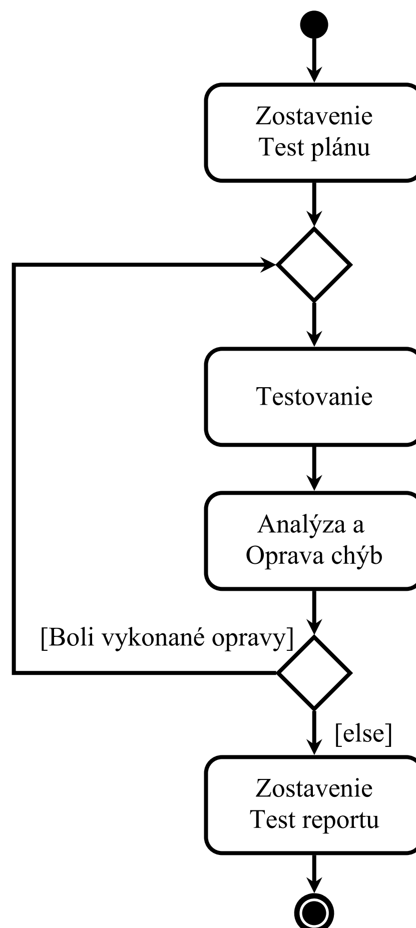
Podľa môjho názoru medzi najdôležitejšie dokumenty testovania softvérových systémov patria Master Test Plan (Test plán), Level Test Case (Testovací prípad), Level Test Procedure (Testovací skript) a Master Test Report (Test report). Týmto dokumentom a ich charakteristikám som sa venoval podrobnejšie. Smernica IEEE 829-2008 presne vymedzuje oblasti, obsah a štruktúry, ktoré majú dokumenty obsahovať. Vychádzajúc z odporúčaní uvedených v danej smernici som navrhol šablónu dokumentov Test plánu a Test reportu.

3.2 Čiastkový návrh základnej štruktúry smernice testovania

Čiastkový návrh smernice, ktorý som vytvoril, pozostáva zo štyroch základných ucelených krokov, ktoré sú prepojené do funkčného celku a zabezpečujú všetky potrebné úkony, ktoré sú nevyhnutné pri etape testovania softvéru. Návrh, ktorým som znázornil UML diagramom aktivít (pozri Obr.23), obsahuje nasledovné kroky:

- Zostavenie Test plánu
- Testovanie
- Analýza výsledkov testovania a oprava odhalených chýb
- Zostavenie Test reportu

V prípade, že boli po analýze výsledkov testovania vykonané opravy chýb a anomálii vo vyvíjanom softvéri, tak sa proces testovania zopakuje. Ak sa po testovaní nevykonajú žiadne úpravy zdrojového kódu aplikácie, tak sa zostaví výsledný Test report, ktorý informuje o celom priebehu testovania.



Obr. 23: UML Diagram aktivít, Návrh základnej štruktúry smernice testovania

3.2.1 Zostavenie Test plánu

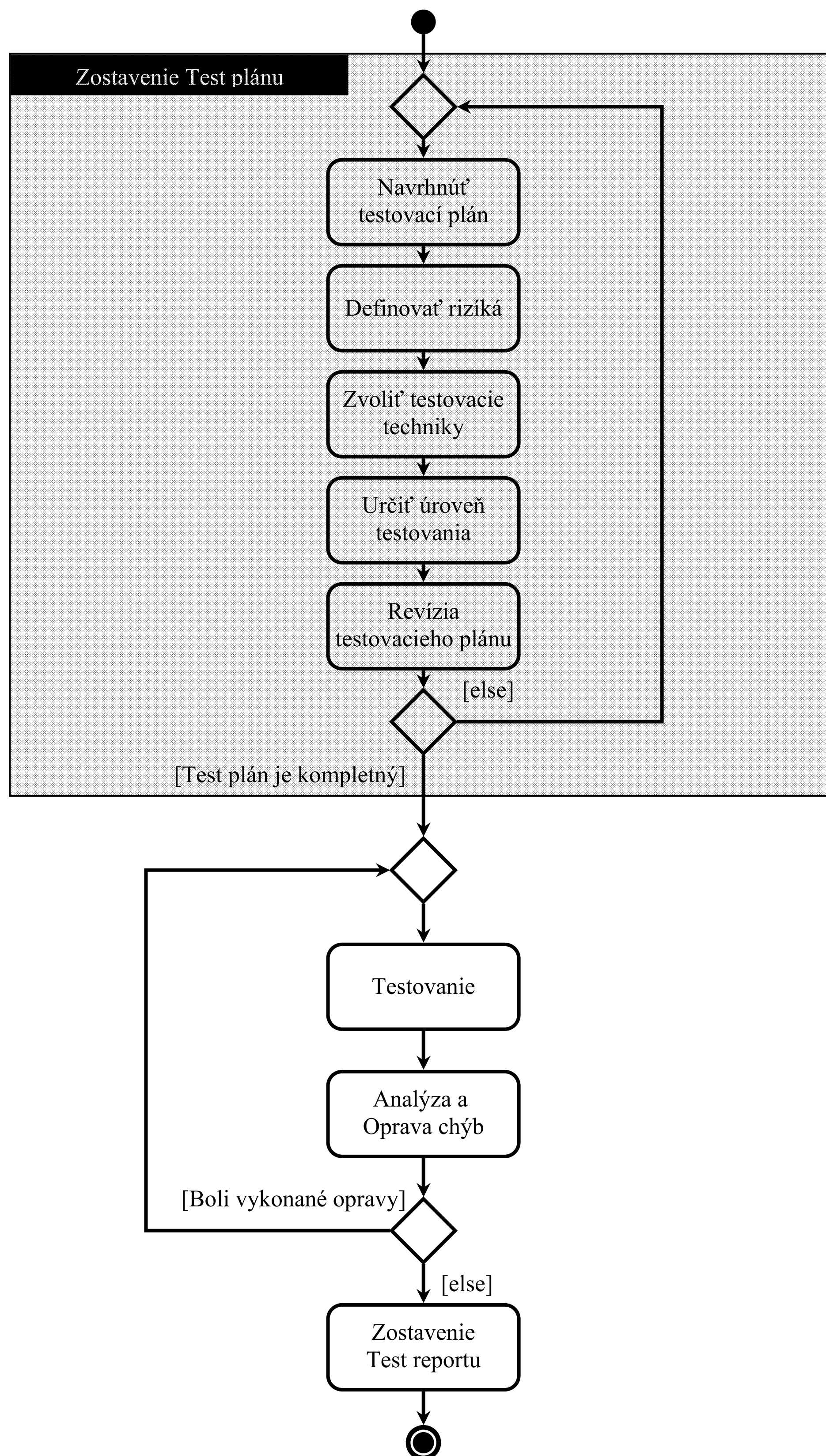
Testovací plán patrí medzi kľúčové dokumenty testovacej dokumentácie. Je určený pre manažment projektu testovania. Herbst [30] sumarizuje základné podmienky testovania, ktoré sú uvedené v testovacom pláne nasledovne:

- Cieľ testovania – definuje, čo sa má testovaním dosiahnuť.
- Prehľad plánovaných testov – zoznam testov, ktoré pokryjú všetky funkcionality vyvíjaného softvéru s dôrazom na ich použitie zo strany zákazníka.
- Stanovenie priorít – určuje prioritu testovaných prvkov v softvéri a teda aj prioritu príslušných testov. Najvyššiu prioritu majú tie prvky, pri ktorých v prípade ich nefunkčnosti hrozí najväčšie riziko nesplnenia cieľov testovania.
- Stratégia testovania – opisuje, aké typy testov budú pri testovaní použité a v ktorej etape testovania budú použité. Stanovuje techniky, ktorými budú testy vykonané ako aj spôsob, akým budú testy vyhodnocované, a kritériá, na základe ktorých bude testovanie možné označiť za úplné.
- Požiadavky na zdroje – definujú požiadavky, ktoré musia byť splnené, aby bolo možné vykonať testy (napr. testovací hardvér, softvér, sprístupnenie serverov a pod.). Taktiež je vymenovaný testovací tím vrátane ich ďalších spolupracovníkov.
- Definícia rizík – vymedzuje situácie, ktoré môžu ohroziť úspešnosť testovania. Stanoví sa miera týchto rizík a navrhnú sa protiopatrenia.

Podľa normy IEEE 829-2008 [31] je testovací plán dokument, ktorý popisuje rozsah testu, prístup k testovaniu, zdroje a rozvrh zamýšľaných testovacích aktivít. Identifikuje testované položky, vlastnosti, ktoré budú testované, úlohy testovania, kto bude vykonávať jednotlivé úlohy a identifikuje riziká, ktoré vyžadujú alternatívne plánovanie. Je to súčasne aj dokument, ktorý opisuje, ako má byť technicky a manažérsky zrealizované testovanie softvéru (systému alebo komponentu).

Pri návrhu testovacieho plánu som zohľadnil a použil niektoré odporúčania normy IEEE 829-2008. Táto norma komplexne rieši celý proces testovania a záleží len na používateľovi, ktoré časti tejto normy aplikuje pri riešení svojho projektu.

Obrázok 24. znázorňuje jednotlivé kroky procesu „Zostavenie Test plánu“, ktoré som implementoval do UML diagramu aktivít „Čiastkový návrh základnej štruktúry smernice testovania“.



Obr. 24: UML Diagram aktivít, Návrh - zostavenie Test plánu

<table>
<tr><td colspan="4">TEST PLÁN</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="2">Názov projektu</td><td colspan="2">Organizácia</td></tr>
<tr><td colspan="2"></td><td>Zadávateľ projektu:</td><td>Realizácia projektu:</td></tr>
<tr><td colspan="2">Verzia</td><td colspan="2" rowspan="2">Autor:</td></tr>
<tr><td></td><td>náčrt</td></tr>
<tr><td></td><td>preverená</td><td colspan="2">Dátum vyhotovenia:</td></tr>
<tr><td></td><td>opravená</td><td colspan="2">Schválil:</td></tr>
<tr><td>×</td><td>finálna</td><td colspan="2">Dátum schválenia:</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="2">Testovacia úroveň</td><td colspan="2">Použitá dokumentácia</td></tr>
<tr><td></td><td>unitov/jednotiek</td><td colspan="2" rowspan="4"></td></tr>
<tr><td></td><td>integračná</td></tr>
<tr><td></td><td>systémová</td></tr>
<tr><td>×</td><td>akceptačná</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">Zameranie testovania</td></tr>
<tr><td colspan="4">Časti/komponenty zahrnuté do testovania:</td></tr>
<tr><td colspan="4">Časti/komponenty vylúčené z testovania:</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">Charakteristika testovaného softvérového produktu</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">Charakteristika testovacieho procesu</td></tr>
<tr><td colspan="4">Ciele testovania:</td></tr>
</table>

Obr. 25a: Návrh - Testovací plán, prvá strana

Časový plán testovania			
Príprava prostredia	Analýza	Testovanie	Spolu

Schéma úrovní integrity podľa dôsledkov zlyhania			
Úroveň integrity		Definícia a identifikácia úrovne	
0	bez dôsledkov		
1	zanedbateľné dôsledky		×
2	málo významné dôsledky		
3	kritické dôsledky		
4	katastrofálne dôsledky		

Testovací tím		
Priezvisko, meno, titul	Funkcia	Kontakt

Zodpovedné osoby		
Priezvisko, meno, titul	Funkcia	Kontakt

Nástroje:
Techniky:
Metódy:
Metriky:

Obr. 25b: Návrh - Testovací plán, druhá strana

Testovacie prípady								
Testovací prípad č.	Prostredie	Metódy	Frekvencia	Vstupy	Výstupy	Predpokladaný výsledok	Termín od – do	Zodpovedná osoba

Obr. 25c: Návrh - Testovací plán, Testovacie prípady

3.2.1.1 Testovací prípad

Testovacie prípady sú základom dokumentácie, ktorá je určená pre testerov. Slúži ako podklad na testovanie jedného konkrétneho miesta vo vyvíjanom softvéri. Podľa Herbsta [30] tento dokument definuje podmienky, ktoré musia byť splnené, aby bolo možné daný prípad otestovať. Taktiež špecifikuje druh a formát vstupných a výstupných dát.

Page a kol. [16] definuje testovací prípad ako formálny dokument, ktorý opisuje, ako sa má vykonať určitá testovacia činnosť. Autori charakterizujú testovací prípad ako popis konkrétnej činnosti, ktorá je vykonaná na konkrétnom komponente softvéru a popis výsledkov tejto činnosti. Testovacie prípady môžu byť uvedené vo forme zoznamu krokov, ktoré je potrebné vykonať, a očakávaných výsledkov. V prípade automatizovaného testovania je testovací prípad uvedený ako súbor programových inštrukcií. Podľa autorov by kvalitne zostavený testovací prípad mal obsahovať:

- Účel – vysvetľuje, prečo je testovanie dôležité a na čo slúži.
- Podmienky – špecifikuje vplyvy prostredia dôležité pre vykonanie testu, ako aj potrebný softvér a hardvér.
- Konkrétne vstupy a kroky – zoznam krokov, ktoré vedú k presnému a opakovateľnému vykonaniu testovacieho prípadu.
- Očakávané výsledky – všetky informácie nevyhnutné k tomu, aby bolo možné overiť, či bol test úspešný, alebo nie.
- Frekvenciu (typ testu) – ako často má byť test vykonaný.
- Konfiguráciu – konfigurácia softvéru, na ktorom bude test vykonaný.
- Automatizáciu – stupeň automatizácie testu, t.j. či je plne automatizovaný, čiastočne automatizovaný alebo manuálny.

Podľa normy IEEE 829-2008 [31] je testovací prípad súbor vstupných hodnôt, podmienok pred vykonaním, očakávaných výsledkov a podmienok po vykonaní, ktorý bol vyvinutý pre konkrétny účel, akým je vykonanie určitej cesty v programe, alebo verifikovanie zhody s konkrétnou požiadavkou.

3.2.1.2 Testovací skript

Testovací skript je podľa Herbsta [30] ucelený a logicky vystavaný dokument. Testovací skript vznikne kombináciou niekoľkých testovacích prípadov, ktoré spolu tvoria logický celok. Sú v ňom obsiahnuté všetky vstupné dáta, jednotlivé kroky, ktoré sa majú vykonať a pre každý krok je definovaný aj očakávaný výsledok. Jednotlivé kroky musia na seba nadväzovať, t.j. výsledok jedného kroku sa stáva vstupom pre nasledujúci krok. Ak každý krok testovania prebehol v poriadku a jeho výsledok zodpovedá očakávaniam, testovací skript sa označí za úspešne otestovaný.

Medzi základné vlastnosti efektívneho testovacieho skriptu patrí jeho vykonateľnosť a s tým súvisiaca splniteľnosť vstupných podmienok jednotlivých krokov, ako aj preukázateľnosť očakávaných výstupov.

Podľa normy IEEE 829-2008 [31] je testovací skript definovaný ako test, ktorý pozostáva z jedného alebo viacerých testovacích prípadov.

3.2.1.3 Testovací scenár

Testovací scenár vznikne spojením niekoľkých testovacích skrípt. Testovacie skripty sú v scenári zoradené tak, aby na seba nadväzovali, t.j. aby výstup prvého skriptu tvoril vstup pre skript nasledujúci. Cieľom je simulovať konkrétny spôsob používania aplikácie u zákazníka.

Podľa normy IEEE 829-2008 [31] je testovací scenár definovaný ako opis série udalostí, ktoré sa môžu vyskytnúť súčasne alebo postupne. Je bežne používaný pre skupiny testovacích prípadov.

3.2.2 Testovanie

Proces testovania som navrhol podľa normy IEEE 829-2008. Použil som všetky potrebné kroky, ktoré sú nevyhnutné pri realizácii veľkých softvérových projektov.

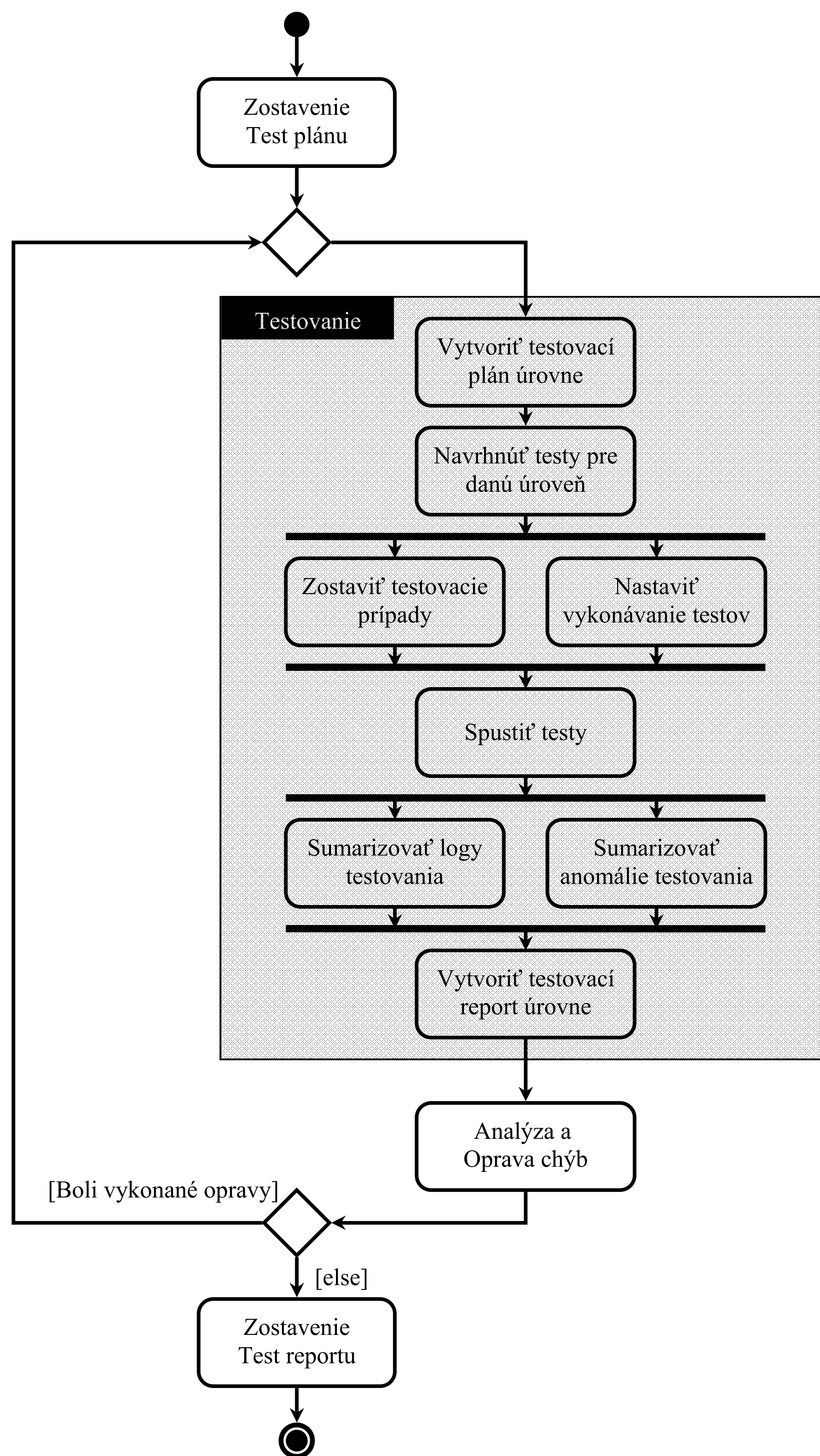
Proces testovania, ktorý som navrhol, by som zhrnul do nasledovných šesť krokov:

1. Vytvoriť testovací plán úrovne.
2. Navrhnúť testy pre danú úroveň.
3. Súbežne budú prebiehať následné kroky:
   - zostaviť testovacie prípady,
   - nastaviť vykonávanie testov.
4. Spustiť testovanie.
5. Súbežne budú prebiehať následné kroky:
   - sumarizovať logy testovania,
   - sumarizovať anomálie testovania.
6. Vytvoriť testovací report úrovne.

Obrázok 26. znázorňuje tieto jednotlivé kroky procesu „Testovanie“, ktoré som implementoval do UML diagramu aktivít „Čiastkový návrh základnej štruktúry smernice testovania“. Vytvoril som všeobecný návrh testovania, ktorý sa dá upraviť podľa aktuálnych potrieb daných testov.

Po vykonaní testovania nasleduje „Analýza a Oprava chýb“, kde sa vyhodnotia zistené anomálie a následne sa chyby zaevidujú a prípadne opravia. Ak po realizácii tohto kroku vznikne potreba nového testovania, tak sa celý proces testovania spúšťa odznovu.

Tento návrh testovania som zostavil ako univerzálny, takže sa dá takmer bez úprav aplikovať aj na automatizované testovanie. Automatizované testovanie som si zvolil pre návrh smernice testovania veľkých softvérových systémov, pretože to je v súčasnosti aktuálna téma, ktorá pri správnom aplikovaní môže priniesť úsporu zdrojov. Veľké softvérové systémy kladú vysoké nároky na testovanie. Firmy, ktoré vyvíjajú takéto systémy alebo sa zameriavajú na ich testovanie sa musia zaoberať aj automatizáciou testovania.



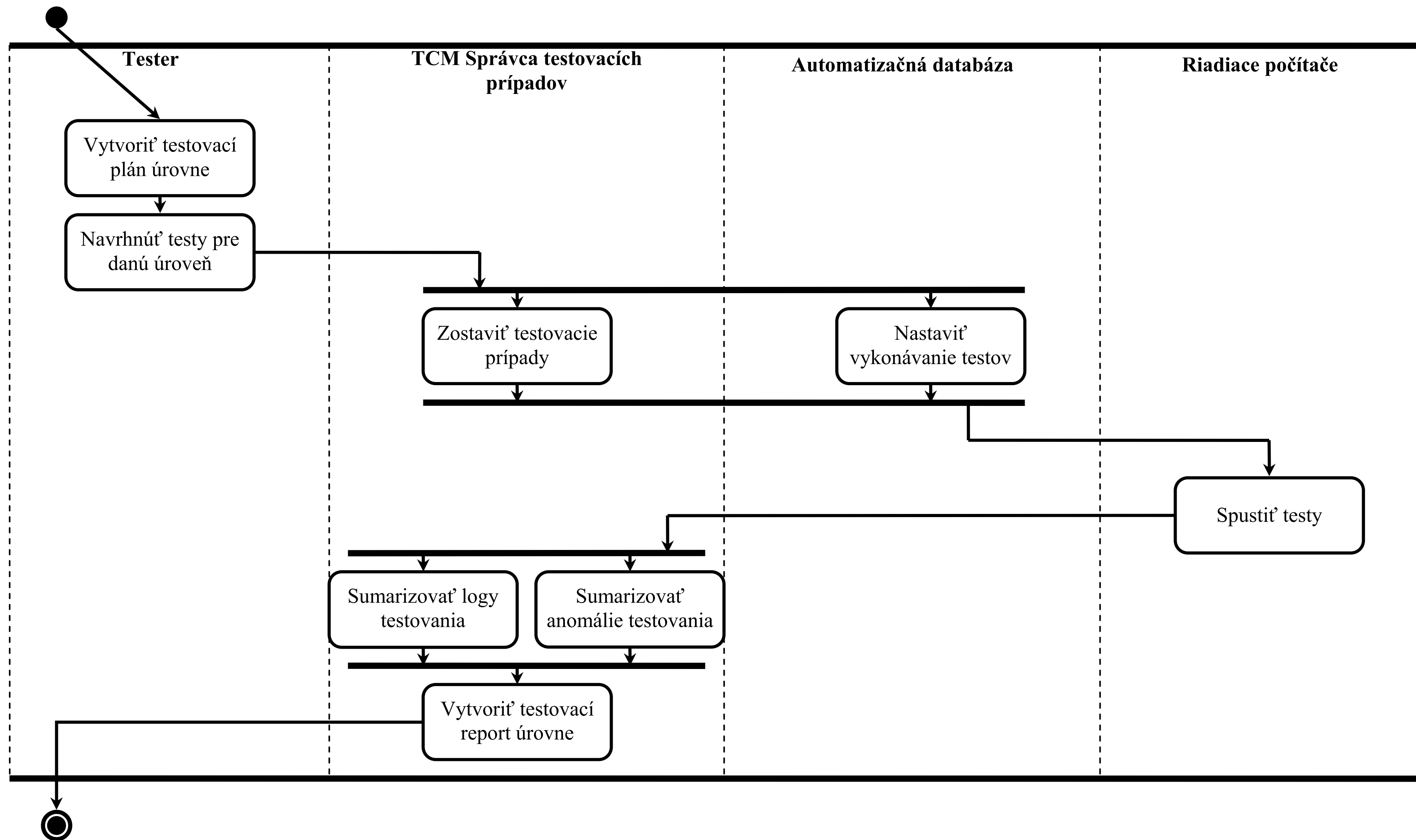
Obr. 26: UML Diagram aktivít, Návrh - Testovanie

3.2.2.1 Automatizované testovanie

Automatizácia je vhodná na testy, ktoré sa často opakujú. Má to ale aj úskalia, pretože každá zmena vo vyvíjanom softvéri môže následne vyvolať nutnosť opraviť testovacie skripty, prípadne nastavenie testovacieho nástroja. Preto je táto metóda vhodná u regresných testov, ktoré overujú, či sa po pridaní novej funkcionality do softvéru neočakávane objavia anomálie vo zvyšnej časti softvéru.

Automatizované testovanie som navrhol pre metódu SEARCH podľa normy IEEE 829-2008. Návrh som znázornil UML diagramom správania objektov pre aplikačný proces „Testovanie" na Obrázku 27. V uvedenom návrhu som presne vymedzil kroky a vzájomné prepojenia medzi testerom, správcom testovacích prípadov (TCM), automatizačnou databázou a testovacími počítačmi. Výsledkom testovania je testovací report danej testovanej úrovne, ktorý je ďalej odoslaný testerovi (testovaciemu tímu) na analýzu výsledkov testov. Obrázok 27 taktiež znázorňuje fungovanie infraštruktúry na automatizáciu testov pre metódu SEARCH, ktoré by som zhrnul do nasledovných krokov:

1. Správca testovacích prípadov (Test Case Manager, TCM) zostaví zoznam testov, ktoré sa majú vykonať.
2. Vytvoria sa krížové väzby medzi jednotlivými testovacími skriptami, ktoré sa budú vykonávať - tu sa aplikujú kódy alebo globálne identifikátory, ktoré sú zdieľané medzi TCM a automatizačnou databázou.
3. TCM a automatizačná databáza sa spoja s riadiacimi počítačmi, ktoré pripravia testovacie počítače na spustenie predpísaných testov.
4. Ihneď po príprave testovacieho počítača, riadiaci počítač spustí na tomto počítači vykonávanie testov.
5. Po zbehnutí testov na testovacom počítači si riadiaci počítač okamžite skopíruje protokol, ktorý postúpi TCM na spracovanie.
6. Spracovaním všetkých protokolov sa vygeneruje výsledok testov, ktorý sa ukladá do TCM. Testovací tím potom podľa týchto výsledkov začne skúmať jednotlivé objavené chyby.



Obr. 27: UML Diagram správania objektov pre aplikačný proces Testovanie, Návrh - Testovanie metódou SEARCH

3.2.2.2 Záťažové testovanie

Vo všeobecnosti platí, že na zautomatizovanie je vhodné použiť testy, ktoré sú založené na dátach. U týchto testov je priebeh testovania stále rovnaký, len sa menia vstupné a výstupné dáta. Táto skupina testov sa taktiež nazýva „Testy riadené dátami“.

Každý typ testu je dôležitý pri vývoji veľkého softvérového systému. Pri testovaní takýchto systémov sa najčastejšie môžeme stretnúť so záťažovými testami a testami funkčnosti. Preto som v tejto práci podrobnejšie vypracoval návrh pre záťažové testovanie.

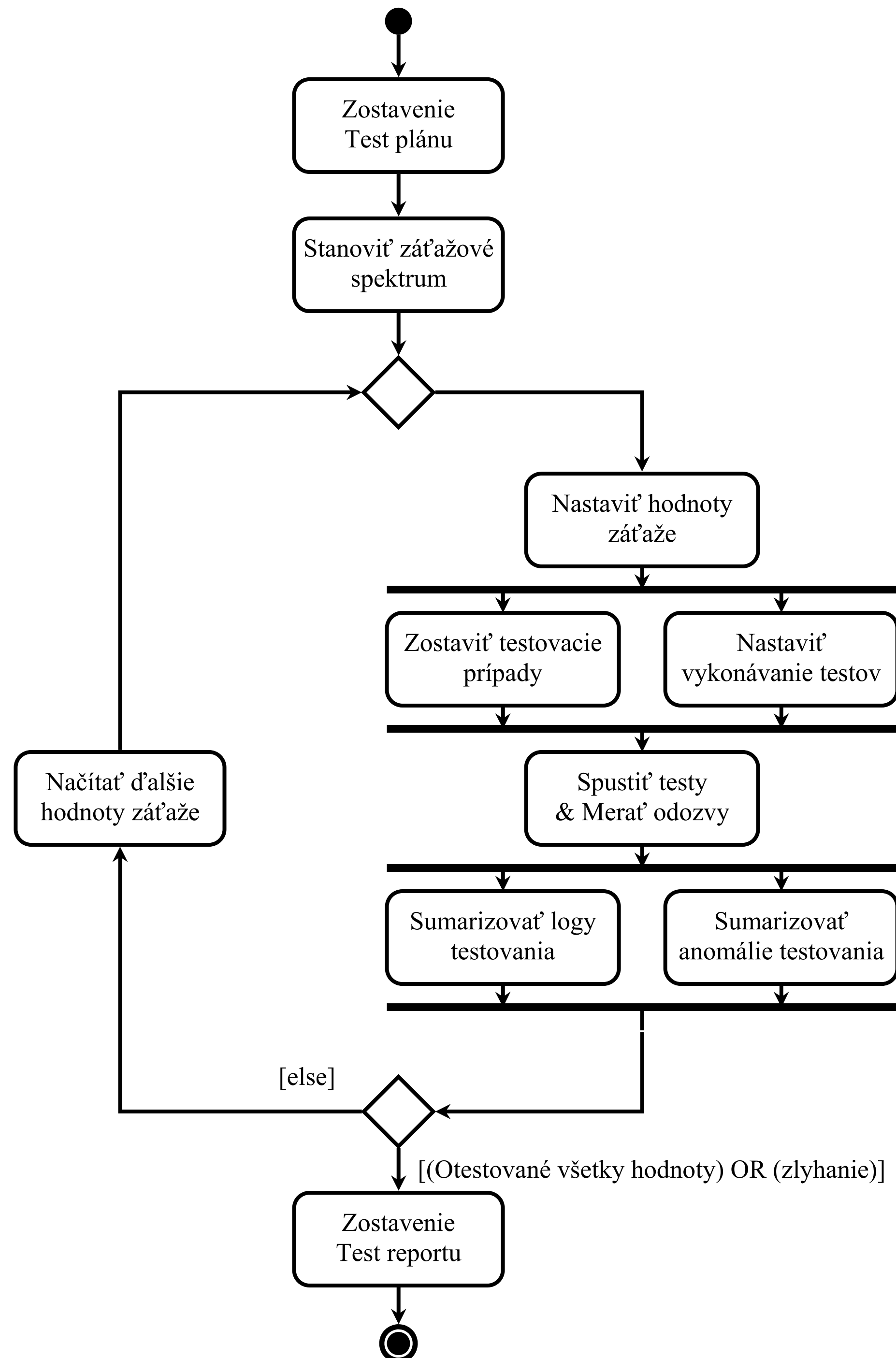
Cieľom záťažového testovania je overiť, či testovaný softvér vyhovuje definovaným požiadavkám. V reálnej podobe ide o zaťaženie vyvíjaného softvéru určitým počtom užívateľov. Počas testovania sú aplikované vopred stanovené hodnoty, ktoré zaťažujú softvér presne definovaným počtom súčasne pracujúcich užívateľov.
Podľa Klašku [32] správne zrealizované záťažové testovanie nám zabezpečí:

- Výkonnostnú akceptáciu vyvíjaného softvérového systému.
- Overenie správania softvérového systému pri predpokladanej bežnej záťaži, ale aj pri očakávaných špičkách (napríklad ročné zúčtovanie dane).
- Vyladenie softvérového systému ako celku a jeho komponentov pred uvoľnením do reálnej prevádzky.
- Zistenie výkonnostných limitov vyvíjaného softvérového systému.
- Overenie vhodnosti a výkonnosti zvolenej vývojovej platformy, použitej technológie a infraštruktúry softvérového systému.
- Overenie správania softvérového systému pri nežiaducom výpadku, či zlyhávaní jeho jednotlivých komponentov.

Samotný návrh záťažového testovania (pozri Obr.28) vychádza z môjho čiastkového návrhu základnej štruktúry smernice testovania. Vykonal som v ňom ale potrebné úpravy, ktoré vyžadovalo samotné použite a priebeh záťažového testovania. Úpravy by som zhrnul do týchto bodov:

- Test plán musí obsahovať záťažové spektrum. Ide o presne definované hodnoty udávajúce počet súčasne pracujúcich užívateľov na vyvíjanom softvérovom systéme.
- Testovanie spočíva v meraní rýchlosti odozvy na jednotlivé požiadavky, ktoré generuje testovací systém.
- Po úspešnom otestovaní sa načíta ďalšia hodnota, ktorá udáva novú záťaž – nový počet virtuálnych užívateľov a opätovne sa spustí testovanie.

- Testovanie končí otestovaním všetkých hodnôt záťažového spektra alebo zlyhaním vyvíjaného softvérového systému.



Obr. 28: UML Diagram aktivít, Návrh – Záťažové testovanie

Záťažové testy sa spúšťajú opakovane pre stanovený počet virtuálnych používateľov a doba testovania závisí na celkovom rozsahu a zložitosti testovania. Klaška [33] uvádza tieto výhody automatizácie záťažových testov:

- Možnosť testovania veľkých softvérových systémov pre niekoľko tisíc používateľov s použitím aj malých počtov počítačov.
- Riadená zaťaž z jedného miesta a jedným testerom.
- Možnosť presného viacnásobného zopakovania testu.
- Sledovanie správania softvérového systému pod záťažou a vyhodnocovanie výsledkov pokročilými štatistickými metódami.
- Jednoznačná preukázateľnosť nameraných výsledkov testovania.

Pre modelovanie spolupráce objektov sa v jazyku UML používajú také typy diagramov, ktoré majú vytvorené k tomuto účelu vyjadrovacie schopnosti znázorniť túto spoluprácu medzi objektmi. Skupinu týchto diagramov zastrešuje diagram interakcií. Na návrh záťažového testovania som použil z tejto skupiny diagramov sekvenčný diagram (pozri Obr.29).

Správanie testovacieho systému počas záťažového testovania, tak ako som ho navrhol, by som zhrnul do nasledovných krokov:

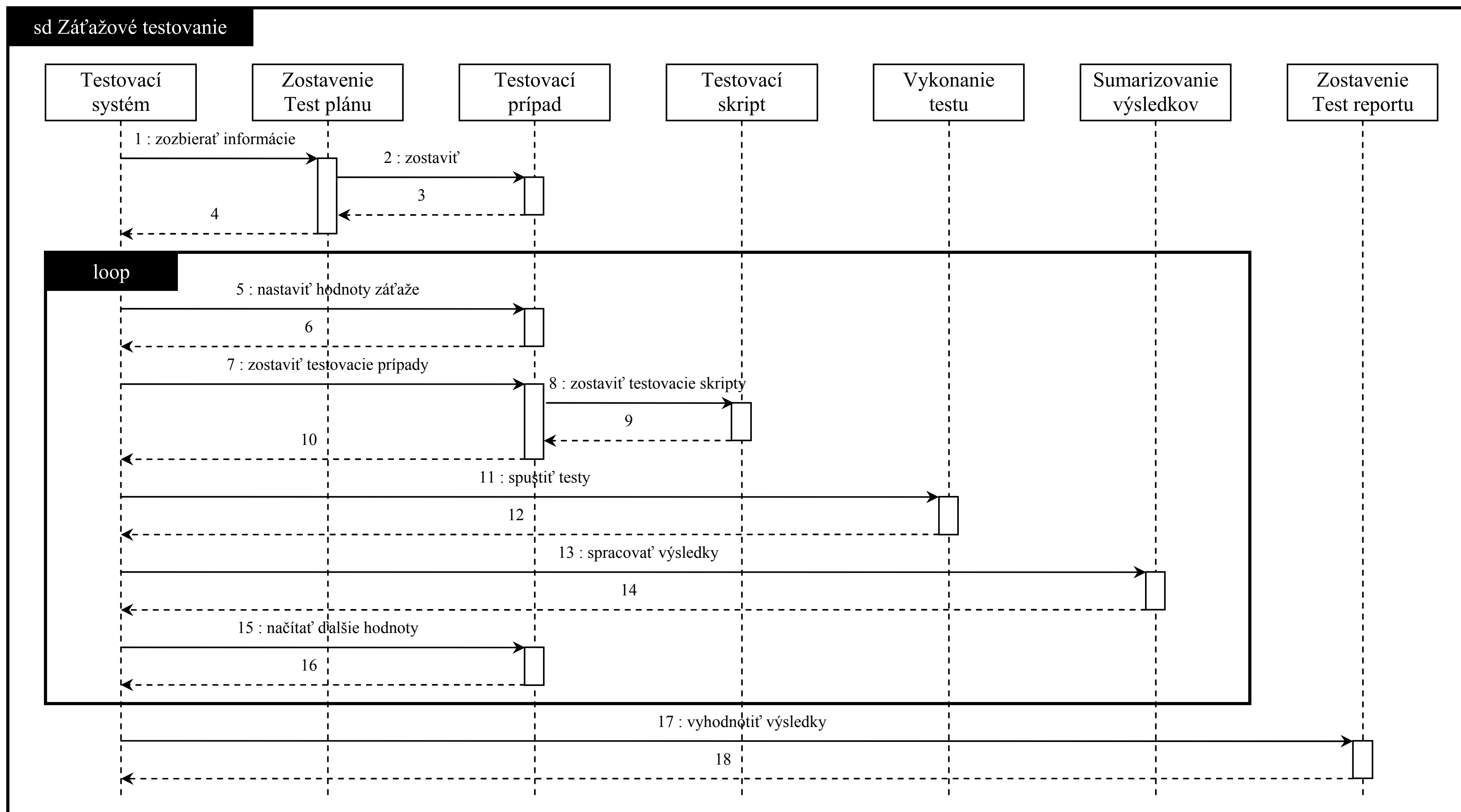
1. Testovací systém – dá pokyn na zozbieranie vstupných informácií.
2. Testovací systém – dá pokyn na zostavenie skupín testovacích prípadov.
5. Testovací systém – dá pokyn na nastavenie hodnoty záťaže.

   Nasleduje cyklus „loop“.

   7. Testovací systém – dá pokyn na zostavenie testovacích prípadov s testovacími hodnotami záťaže.
   8. Testovací systém – dá pokyn na kompletizáciu testovacích skriptov.
   11. Testovací systém – dá pokyn na spustenie testu.
   13. Testovací systém – dá pokyn na spracovanie výsledkov.
   15. Testovací systém – dá pokyn na načítanie ďalšej hodnoty záťaže.

   V prípade otestovania všetkých hodnôt stanovených v záťažovom spektre, alebo zlyhania testovaného softvéru sa testovanie končí. V opačnom prípade sa v testovaní pokračuje.

17. Testovací systém – dá pokyn na vyhodnotenie výsledkov testovania.



Obr. 29: UML Sekvenčný diagram, Návrh – Záťažové testovanie

3.2.3 Analýza a Oprava chýb

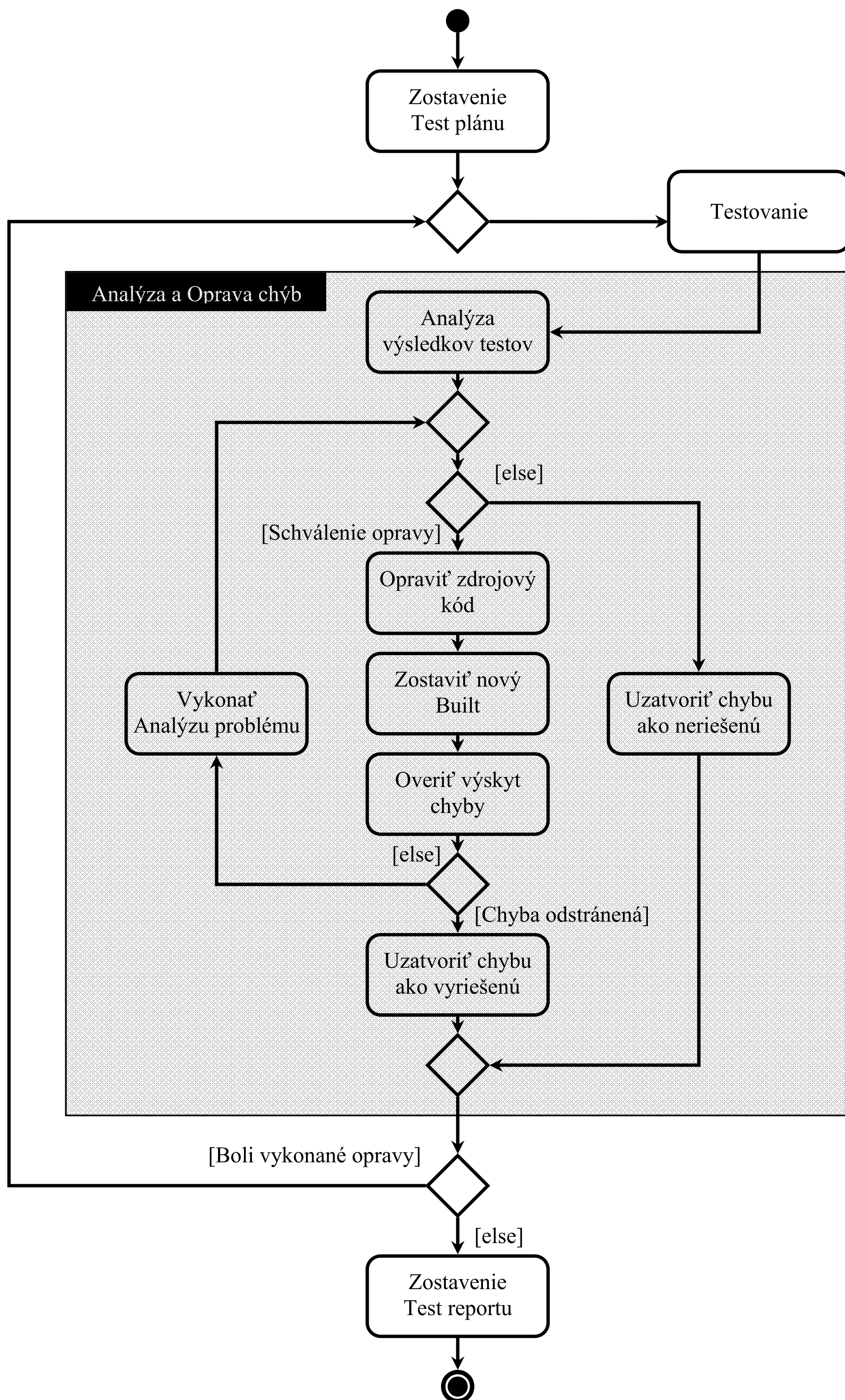
Môj návrh procesu „Analýza a Oprava chýb“ by som charakterizoval pomocou nasledovných krokov:

1. Analýza výsledkov testov získaných z procesu „Testovanie“.
2. Vyhodnotenie zistenej chyby:
    - neschváliť opravu chyby a uzatvoriť chybu ako neriešenú a následne ukončiť tento proces „Analýza a Oprava chýb“,
    - schváliť opravu a pokračovať ďalším bodom.
3. Opraviť zdrojový kód.
4. Zostaviť nový Built.
5. Overiť výskyt chyby.
6. Vyhodnotenie zistenej chyby:
    - ak bola chyba odstránená, tak uzatvoriť chybu ako vyriešenú a následne ukončiť tento proces „Analýza a Oprava chýb“,
    - ak chyba pretrváva, vykoná sa analýza problému a vrátime sa do druhého bodu.

Riešenie, ktoré som navrhol, ráta s možnosťou, že nie každá objavená chyba bude ihneď aj riešená. V praxi poznáme niekoľko stupňov závažnosti chýb, ktoré definujú akútnosť potreby riešenia. Medzi ne napríklad patrí:

- Kritická chyba - je nutná jej oprava, bez jej odstránenia sa nemôže ďalej napredovať vo vývoji softvéru.
- Dôležitá chyba – je nutná jej oprava v blízkej dobe, ale je možné ďalej napredovať vo vývoji softvéru.
- Stredná chyba – oprava je potrebná, chyba bude odstránená pri ďalšej revízii alebo v ďalšej verzii vyvíjaného softvéru. Pri tejto chybe je dôležité jej čo najpresnejšie zadefinovanie do „Test reportu“, aby bolo v budúcnosti možné jej úspešné lokalizovanie a odstránenie.

Obrázok 30. znázorňuje ako som jednotlivé kroky procesu „Analýza a Oprava chýb“ implementoval do UML diagramu aktivít „Čiastkový návrh základnej štruktúry smernice“.



Obr. 30: UML Diagram aktivít, Návrh - Analýza a Oprava chýb

3.2.4 Zostavenie Test reportu

Test report je dokument, v ktorom sa zhromažďujú a hodnotia finálne výsledky testovania uvedené v testovacích reportoch jednotlivých úrovní. Spolu s Test plánom patrí medzi najdôležitejšie dokumenty testovacej dokumentácie. Pri zostavovaní návrhu šablóny pre Test report, som použil nasledovné odporúčania uvedené v norme IEEE 829-2008:

- Test report sumarizuje testovacie procesy pre jednotlivé verzie, ktoré korešpondujú s testovacími procesmi uvedenými v Test pláne.
- Test report sumarizuje všetky testovacie prípady, ktoré boli vykonané pre jednotlivé vývojové verzie a taktiež by mali korešpondovať so zoznamom testovacích prípadov, ktoré boli uvedené v Test pláne.
- Ďalšou súčasťou Test reportu je sumarizovanie chýb odhalených počas testovania vykonaného pre jednotlivé vývojové verzie. Okrem toho by mal obsahovať oddelené zoznamy tých chýb, ktoré boli opravené a tých, ktoré opravené neboli.
- Test plán taktiež poskytuje celkové hodnotenie kvality softvéru pre jednotlivé vývojové verzie spolu s odôvodnením.
- Sumarizuje aj všetky použité metriky.

Ďalší celok tohto dokumentu slúži na špecifikáciu dôvodov, pre ktoré bol vyslovený záver o výsledku testovania softvérového systému (vyhovuje / nevyhovuje / čiastočne vyhovuje). Tento dokument taktiež prináša závery a odporúčania týkajúce sa akceptácie produktu dodaného v príslušnej vývojovej verzii. Hodnotí sa aj miera pripravenosti produktu na vydanie do užívania. V závere môže priniesť odporúčania a postupy, ktoré sa osvedčili, respektíve museli byť pozmenené počas vykonávania testov.

<table>
<tr><td colspan="4">TEST REPORT</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="2">Názov projektu</td><td colspan="2">Organizácia</td></tr>
<tr><td colspan="2"></td><td>Zadávateľ projektu:</td><td>Realizácia projektu:</td></tr>
<tr><td colspan="2">Verzia</td><td colspan="2" rowspan="2">Autor:</td></tr>
<tr><td></td><td>náčrt</td></tr>
<tr><td></td><td>preverená</td><td colspan="2">Dátum vyhotovenia:</td></tr>
<tr><td></td><td>opravená</td><td colspan="2">Schválil:</td></tr>
<tr><td>×</td><td>finálna</td><td colspan="2">Dátum schválenia:</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="2">Testovacia úroveň</td><td colspan="2">Použitá dokumentácia</td></tr>
<tr><td></td><td>unitov/jednotiek</td><td colspan="2" rowspan="4"></td></tr>
<tr><td></td><td>integračná</td></tr>
<tr><td></td><td>systémová</td></tr>
<tr><td>×</td><td>akceptačná</td></tr>
</table>

<table>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">Výsledný čas</td></tr>
<tr><td>Reálny štart testovania</td><td>Reálny koniec testovania</td><td>Plánovaný štart testovania</td><td>Plánovaný koniec testovania</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">Výsledné pokrytie testami</td></tr>
<tr><td>Prostredie</td><td>Plánované testy</td><td>Vykonané testy</td><td>Nevykonané testy</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><td colspan="5"></td></tr>
<tr><td colspan="5">Výsledné zhrnutie chýb</td></tr>
<tr><td rowspan="2">Celkový počet chýb</td><td rowspan="2">Prostredie</td><td colspan="3">Závažnosť chyby</td></tr>
<tr><td>Kritická</td><td>Dôležitá</td><td>Stredná</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Obr. 31a: Návrh - Test report, prvá strana

<table>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4">Výsledné zhrnutie testovacích prípadov</td></tr>
<tr><td rowspan="2">Testovací prípad</td><td colspan="2">Stav testovania</td><td rowspan="2">Poznámka</td></tr>
<tr><td>Vykonané</td><td>Nevykonané</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><td colspan="5"></td></tr>
<tr><td colspan="5">Výsledné zhrnutie chýb</td></tr>
<tr><td>ID</td><td>Popis</td><td>Lokalizácia</td><td>Závažnosť</td><td>Status chyby (opravené/ neopravené)</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Obr. 31b: Návrh - Test report, druhá strana

ZÁVER

V súčasnosti sú softvérové systémy natoľko rozšírené v rôznych oblastiach ľudskej činnosti a každodenného života človeka, že ich prípadné zlyhanie v dôsledku neobjavenej a následne neopravenej chyby môže mať zásadný dopad na životné prostredie, poškodenie majetku, ušlý zisk, alebo môže dokonca viesť k ohrozeniu, prípadne k strate ľudského života. Včasná detekcia chýb v procese testovania sa preto stala nevyhnutnou požiadavkou užívateľov softvérových produktov. Sme preto svedkami zmeny prístupu k testovaniu ako takému. Testovanie už nie je iba poslednou etapou životného cyklu vývoja softvéru, akýmsi „nutným zlom" vývojového procesu. Naopak, stáva sa jeho neoddeliteľnou súčasťou a postupuje všetkými etapami životného cyklu. Náklady na odstránenie chyby detekovanej v priebehu vývoja softvéru sú nižšie ako náklady na odstránenie chyby detekovanej na konci vývojového procesu. Okrem zmeneného prístupu k testovaniu sú v súčasnosti dostupné modernejšie a efektívnejšie metódy a techniky testovania. Za významný prínos v tejto oblasti považujem metódy automatického testovania, vďaka ktorým je možné vyvíjať veľké softvérové systémy. V mojej práci túto skupinu metód reprezentovala metóda SEARCH. Jej opisu som sa venoval v kapitole 2.4.4 a návrh testovania touto metódou v jazyku UML sa nachádza v kapitole 3.2.2.1. Pri vývoji veľkého softvérového systému sa najčastejšie môžeme stretnúť so záťažovými testami a testami funkčnosti. Preto som v kapitole 3.2.2.2 podrobnejšie vypracoval návrh na záťažové testovanie. Neodmysliteľnou súčasťou procesu testovania je aj tester a jeho úlohy, ktoré v súčasnosti získavajú čoraz väčší význam a prinášajú zvýšené nároky na kvalifikáciu testera, jeho flexibilitu a prístup k testovaniu. Testovací proces prebiehajúci počas celého životného cyklu vývoja softvéru sa tak stáva zložitým systémom čiastkových krokov, ktoré je potrebné naplánovať, zrealizovať, vyhodnotiť a zdokumentovať. Táto komplexná práca sa nezaobíde bez jednotného rámca, ktorý poskytujú rôzne normy a štandardy s medzinárodnou platnosťou. Na návrh čiastkovej smernice testovania veľkých softvérových systémov som vybral a použil normu IEEE 829-2008.

Cieľom práce bolo vytvoriť čiastkový návrh smernice pre testovanie veľkého softvérového systému. Prvá časť prináša teoretický prehľad základných pojmov – chyba, životný cyklus, softvérový systém, testovanie softvéru. Ďalej sa zameriava na charakteristiku úlohy testera, unifikovaného modelovacieho jazyka UML a prehľad noriem pre testovanie softvéru. Druhá časť práce je zameraná na zásady, techniky a metódy testovania softvéru. Tretia časť práce obsahuje návrh čiastkovej smernice testovania

systému. Táto časť obsahuje aj postupy vykonania testov a reportovania chýb, ktoré sú znázornené pomocou diagramov v modelovacom jazyku UML.

Prínosom mojej diplomovej práce je teoretický prehľad základných pojmov z hľadiska vývoja a ich používania v literatúre. Vychádzal som z moderného prístupu k testovaniu ako procesu, ktorý prebieha počas celého životného cyklu, pričom som pri tvorbe testovacej dokumentácie vychádzal z najnovšej verzie normy IEEE 829 z roku 2008. Najaktuálnejšie informácie obsiahnuté v tejto norme som zapracoval do návrhu Test plánu a Test reportu. Ich význam spočíva vtom, že sú to základné dokumenty testovacieho procesu na jeho začiatku a na jeho konci. Uvedené návrhy dokumentov môžu byť využité v praxi ako testovací vzor, respektíve šablóna. Nový prístup, ktorý smernica prináša, a ktorý som sa aj ja snažil aplikovať, spočíva v definícii a určení úrovne integrity, ktorá je podľa smernice IEEE 829-2008 kľúčom k určeniu a vytvoreniu potrebnej testovacej dokumentácie, prvým krokom, od ktorého sa odvíja testovanie počas jednotlivých etáp životného cyklu a ktorá slúži k určeniu minimálneho počtu testovacích prípadov pre jednotlivé testovacie procesy. Samotný proces testovania, detekcie chýb, opravy a reportovania chýb som navrhol a graficky znázornil s využitím jazyka UML. Vytvoril som čiastkové návrhy: návrh „Základná štruktúra smernice testovania“ (Obr.23); návrh „Zostavenie Test plánu“ (Obr.24); návrh pre „Testovanie“ (Obr.26); návrh pre „Testovanie metódou SEARCH“ (Obr.27); návrh pre „Záťažové testovanie“ (Obr.28 a 29); návrh „Analýza a Oprava chýb“ (Obr.30). Vychádzal som z V- modelu testovacieho procesu vývoja softvéru, ktorý má sedem etáp a podľa môjho názoru najlepšie vystihuje realitu testovacieho procesu v praxi. Myslím si, že ciele diplomovej práce sa mi podarilo splniť.

ZOZNAM BIBLIOGRAFICKÝCH ODKAZOV

1. PATTON, R. *Testování softwaru*. Brno: Computer Press, 2002. 313 s. ISBN 80-7226-636-5

2. TANUŠKA, P., SCHREIBER, P. *Proces testovania softvérových produktov*. [online]. [2011-01-16]. Dostupné na internete: <http://s.cnl.tuke.sk/~ywetka/T_tanuska.pdf>

3. IBM, Rational Software. *TST170 Principles of Software Testing for Testers*. v 2002.05.00, Instructor Guide

4. FAIGL, J. *Úvod do softwarového inženýrství.* [online]. [2011-01-16]. Dostupné na internete: <http://lynx1.felk.cvut.cz/mep/files/slides/lecture11.pdf>

5. Najznámejšie softvérové chyby v histórii. In *Denník SME* [online]. 1/2006 [2011-01-16]. Dostupné na internete: <http://pocitace.sme.sk/clanok.asp?cl=2532306>

6. SOCHOR, J. *Verifikace a validace SW (...jen pár pojmů).* [online]. [2011-01-16]. Dostupné na internete: <http://www.fi.muni.cz/~sochor/PA104/Slajdy/Testovani.pdf>

7. KANER, C. *Exploratory Testing.* [online]. Orlando, FL: Florida Institute of Technology, 2006 [2011-01-16]. Dostupné na internete: <http://www.kaner.com/pdfs/ETatQAI.pdf>

8. WIKIPEDIA. *Software testing.* [online]. [2011-01-16]. Dostupné na internete: <http://en.wikipedia.org/wiki/Software_testing>

9. TRAN, E. *Verification/Validation/Certification.* USA: Carnegie Mellon University. 1999, [online]. [2011-01-16]. Dostupné na internete: <http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html>

10. V oblasti informačných technológií si môžete objednať služby. In *Národná Obroda* [online]. 8/2004 [2011-01-16]. Dostupné na internete: <http://www.obroda.sk/clanok/13750/>

11. MORAVČÍK, O., VASKÝ, J., MIŠÚT, M. *Softvérové inžinierstvo*. Trnava: Gepard dizajn, 1999. 183 s. ISBN 80-967539-2-4

12. PALETA, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxy.* Brno: Computer Press, 2003. 337 s. ISBN 80-251-0073-1

13. UNICORN. *Testovanie IS.* [online]. [2005-05-08]. Dostupné na internete: <http://www.unicorn.sk/ugr/servis/testovanie_is.htm>

14. KADLEC, V. *Agilní programování – Metodiky efektivního vývoje softwaru.* Brno: Computer Press, 2004. 278 s. ISBN 80-251- 0342-0

15. BIELIKOVÁ, M. *Softvérové inžinierstvo - princípy a manažment.* Bratislava: STU, 2000. 220s. ISBN 80-227-1322-8

16. PAGE, A., JOHNSTON, K., ROLLISON, B. Jak *testuje software Microsoft.* Brno: Computer Press, 2009. 384 s. ISBN 978-80-251-2869-5.

17. WIKIPEDIA. *Concurrent engineering.* [online]. [2011-01-16]. Dostupné na internete: <http://en.wikipedia.org/wiki/Concurrent_engineering>

18. ROBBINS, J. *Ladení a testování aplikací pro .NET a WINDOWS.* Praha: Grada Publishing, 2004. 646 s. ISBN 80-247-0774-8

19. WIKIPEDIA. *Software system.* [online]. [2011-01-16]. Dostupné na internete: <http://en.wikipedia.org/wiki/Software_system>

20. WIKIPEDIA. *Software architecture.* [online]. [2011-01-16]. Dostupné na internete: <http://en.wikipedia.org/wiki/Software_architecture>

21. INTERWAY. *Service Oriented Architecture (SOA).* [online]. [2011-01-16]. Dostupné na internete: <http://www.interway.sk/technologie/service-oriented-architecture-soa/>

22. ŠVEŘEPA, J. MDA, aneb architektura řízená modelem. In *Softwérové noviny*, 2004, č. 10, s. 76-77 [online]. [2011-01-16]. Dostupné na internete: <http://www.lbms.cz/Reseni/_pdf/0410-SWN-JS-MDA-aneb-architektura-rizena-modelem.pdf>

23. ARLOW, J.,NEUSTADT, I. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky.* Brno: Computer Press, 2007. 567 s. ISBN: 978-80-251-1503-9

24. TANUŠKA, P., a kol. *Využitie RUP a UML pri tvorbe softvérových projektov* [CD-ROM]. Trnava: Tripsoft, 2007. ISBN : 978-80-89291-10-6

25. RŮŽIČKA, D. *FURPS – dimenze kvality software (1.díl)*. [online]. [2011-01-16]. Dostupné na internete: <http://www.testqa.cz/furps--dimenze-kvality-software-1-dil.html>

26. GUNERLOY, M. *Z kodéra vývojářem : nástroje a techniky pro opravdové programátory.* Brno: Computer Press, 2007. 287 s. ISBN: 978-80-251-1517-6

27. AMBLER, W., S. *Introduction to Test Driven Design (TDD)*. [online]. [2011-01-16]. Dostupné na internete: <http://www.agiledata.org/essays/tdd.html>

28. PERRY, E., W. *Effective Methods for Software Testing.* Indianapolis: Wiley Publishing, Inc., 2006. 973 s. ISBN: 978-0-7645-9837-1

29. SOMMERVILLE, I. *Software engineering.* Harlow: Pearson Education, 2001. 693 s. ISBN: 0-201-39815-X

30. HERBST, D. *Testovací dokumentace – plán, scenář, případ.* [online]. [2011-01-16]. Dostupné na internete: <http://www.swtestovani.cz/index.php?option=com_content&view=article&id=15:testovaci-dokumentace-plan-scena-pipad&catid=3:zaklady&Itemid=11>

31. IEEE Std 829™-2008, *IEEE Standard for Software and System Test Documentation.* New York: IEEE, 2008. 118 s. ISBN 978-0-7381-5746-7

32. KLAŠKA, J. *Výkonnostní testy informačních systémů (1.díl)*. [online]. [2011-01-16]. Dostupné na internete: <http://www.testqa.cz/vykonnostni-testy-informacnich-systemu-1dil.html>

33. KLAŠKA, J. *Výkonnostní testy informačních systémů (2.díl)*. [online]. [2011-01-16]. Dostupné na internete: <http://www.testqa.cz/vykonnostni-testy-informacnich-systemu-2dil.html>

Čestné vyhlásenie

Čestne vyhlasujem, že diplomovú prácu „Čiastkový návrh smernice pre testovanie veľkých softvérových systémov“ som vypracoval samostatne a pod odborným vedením vedúceho diplomovej práce. Použitú literatúru uvádzam v zozname bibliografických odkazov.

V Trnave dňa 1.5.2011

....

Podpis