Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Klasifikace chyb SW na základě analýz z úložiště

Plzeň 2019 Bc. František Koleňák

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. dubna 2019

Bc. František Koleňák

Abstract

The text of the abstract (in English). It contains the English translation of the thesis title and a short description of the thesis.

Abstrakt

Text abstraktu (česky). Obsahuje krátkou anotaci (cca 10 řádek) v češtině. Budete ji potřebovat i při vyplňování údajů o bakalářské práci ve STAGu. Český i anglický abstrakt by měly být na stejné stránce a měly by si obsahem co možná nejvíce odpovídat (samozřejmě není možný doslovný překlad!).

Obsah

1 Úvod

2 Řízení projektů

Řízení projektu (někdy též projektové řízení) se zabývá řízením projektu, tedy časově ohraničené a ucelené sady činností a procesů, jejímž cílem je co nejefektivněji zavést, vytvořit nebo změnit něco konkrétního (definováno daným projektem).

Tradiční přístup řízení projektů je založen na důkladném naplánování na začátku projektu a řízení všech aktivit v průběhu projektu. Tento přístup (též známý jako vodopádový model) je nejvíce vhodný na projekty, které mají podobu cíle jasně danou a neměnnou (chodník, dům) a ve kterých je nutné dobře naplánovat všechny aktivity související s daným projektem. Tradiční přístup vyžaduje kvalitně popsaný cíl, výstupy a plán projektu.

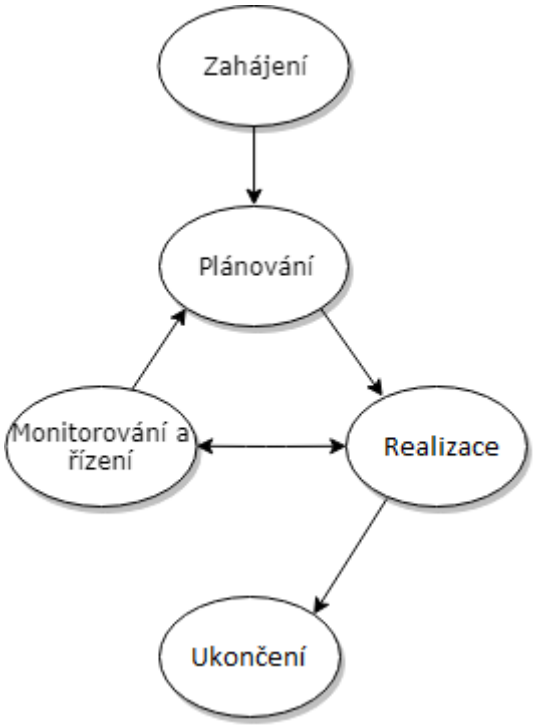
Opakem tradičního přístupu je agilní přístup. Ten je založen na průběžném upřesňování cíle, díky interakci se zákazníkem či s uživateli výsledků projektu. Tento proces se označuje jako získávání zpětné vazby. Reagováním na vstupy od zákazníka je projekt dynamicky upravován a proto je vhodný na takové projekty, kde dochází k vývoji produktu, tedy tehdy když nelze předem kvalitně popsat a naplánovat vše do podrobností a bez zpětné vazby. Agilní přístup se často používá ve vývoji software.

2.1 Fáze projektu

Projekt se může nacházet v různých fázích (procesních skupinách). Tyto fáze se mohou lišit použitou metodou na řízení projektu, ale víceméně je struktura a podoba fází velice podobná. Nejčastějšími fázemi jsou:

- Zahájení - Start projektu a vytyčení jeho mezí
- Plánování - Plánování částí projektu
- Realizace - Implementace rozplánované části
- Monitorování a řízení - Zpětná vazba od zákazníka, podpora
- Uzavření - Oficiální ukončení projektu

Projekt začíná zahájením a končí uzavřením. Fáze plánování, realizace a monitorování a řízení se mohou opakovat až do ukončení projektu. Diagram fází viz Obrázek 2.1. Vodopádový model má pak jednotlivé fáze za sebou tak, že přechod do další fáze je až tehdy, když je daná fáze kompletně ukončena.



Obrázek 2.1: Plánovací diagram

2.2 Softwarové chyby

Softwarové chyba (také nazývána jako „softwarový bug“) se dá definovat jako chyba, defekt, porucha či selhání v počítačovém programu nebo systému, která způsobuje vytváření nekorektních nebo neočekávaných výsledků a nebo nezamýšlené chování.

Softwarové chyby, jsou nedílnou součástí každého softwaru. Projekt větších rozměrů bez žádné softwarové chyby je velmi vzácný, tedy se dá předpokládat, že v nějaké fázi vývoje se objeví nějaký bug. Proces hledání a řešení bugů je nazýván jako „debugování“ a často používá techniky nebo nástroje pro určení bugu.

2.2.1 Testování aplikací

Testování aplikací slouží pro zamezení vydání produktu, který obsahuje chyby jakéhokoliv druhu. Testy mají vliv na kvalitu výsledného produktu, tím pádem se snaží pokrýt co nejvíce případů použití, které mohou nastat.

Samotné bugy vznikají již při realizaci daného projektu a jsou často nalezeny při testování daného řešení samotným vývojářem, nebo nástroji

používanými pro překlad a vývoj. Takovýmto testů se říká „Assembly tests“. Další proces, používaný pro nalezení bugů, je takzvaný „code review“, tedy revize kódu. Při tomto procesu se na cizí kód(vytvořený jiným vývojářem) podívá další vývojář a překontroluje kód, za účelem ověření funkcionality a nalezení potencionálních problémů, které původního autora kódu nemusely napadnout.

Dalším zdrojem odhalení bugů je automatizované testování kódu. Existuje spoustu druhů technik testování. Jedním z hojně používaných druhů testů jsou takzvané jednotkové testy. Ty z pravidla píší samotní vývojáři a slouží k ověření funkcionality části kódu, na co nejmenších úsecích kódu, zpravidla na jednotlivé třídy a metody. Hlavním účelem těchto testů je odhalení funkční chyby v případě, že je nutné změnit část kódu. Tyto testy pak slouží pro odhalení chyby co nejdříve, v rámci životního cyklu aplikace.

Bugy, které jsou nalezeny v předchozích testech, by se měli opravit okamžitě, proto nejsou z pravidla nikde uvedeny. Hlavním zdrojem objevení bugů jsou testy , které jsou popsány níže. Tyto testy se zaměřují na funkcionalitu aplikace z různých pohledů a slouží aby se k zákazníkům dostal co nejkvalitnější produkt.

Smoke testy

Když je aplikace ve fázi, kdy je testovatelná, tedy je vyvinuta, lze ji nainstalovat či spustit a je přístupná, pak se obvykle spouštějí takzvané smoke testy. Smoke testy slouží k ověření, zda je aplikace vhodná k testům a obvykle se zaměřují na hlavní funkce aplikace, které nebývají často upravovány, a to pouze v jejich pozitivním průběhu. Je velmi žádoucí tyto testy automatizovat, neboť se u nich předpokládá, že budou často spouštěny.

Funkční testy

Funkční testy mají za úkol ověřit jestli aplikace plní všechny úkoly, pro které byla určena. Svým obsahem jsou podobné jako smoke testy, avšak ověřují všechny funkce aplikace, které jsou implementovány a ověřují že fungují správně a odpovídají požadavkům zákazníka. [4]

Integrační testy

Přichází na řadu až v době, kdy jsou vývojáři hotovy s jejich testováním. Testy již nevyvíjí samotní vývojáři, ale tým testerů. Testy musí být ověřena bezchybná komunikace mezi jednotlivými komponentami uvnitř aplikace.

Integraci však lze ověřovat nejen mezi komponentami, ale také mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů. V této fázi se testuje integrace, již otestovaných částí. Postupně se testuje integrace mezi dvěma komponentami a přidávají se další. Testy mohou být jak automatizované tak manuální.

Systémové testy

Během těchto testů je aplikace ověřována jako funkční celek tedy se zaměřují na aplikaci tak jak by ji měl používat zákazník. Zde se již ověřuje soulad reálného chování s chováním očekávaným, provádí se testování možných negativních průběhů, validují se výstupy a podobně. Podle připravených scénářů se simulují různé kroky, které v praxi mohou nastat. Obvykle probíhají v několika kolech. Nalezené chyby jsou opraveny a v dalších kolech jsou tyto opravy opět otestovány. Součástí této úrovně jsou jak funkční tak nefunkční testy. Tato úroveň testů většinou slouží jako výstupní kontrola softwaru.

Akceptační testy

Akceptační testy, neboli *User Acceptance Tests* se provádí na straně zákazníka. Ten provádí, většinou se svým týmem testerů, předem připravené scénáře, které jsou na jeho testovacím prostředí. Nalezené nesrovnalosti, jsou ohlášeny zpět vývojovému týmu.

2.2.2 Významnost bugů

V agilním přístupu je spousta iterací, kdy je aplikace podrobena testy. V každé z této iterací může být objeven bug. Takový bug je nutno zařadit do kategorie významnosti. Když je nalezen bug tak první věcí, která se přiřadí bugu, je jak kritický daný bug je a jaký dopad má na funkčnost aplikace nebo celého systému. Díky kategorii, která je bugu přiřazena, je možné naplánovat jestli bude bug opraven a kdy. Zatímco různé organizace mají různé způsoby klasifikace chyb nebo trendů chyb, obecně uznávaná taxonomie má následující stupně závažnosti[1]:

- **Závažnost 1** Chyba, která zabraňuje provedení provozní funkce nebo funkce, která je nezbytná pro splnění úkolů, brání obsluze / uživateli vykonávat zásadní funkce nebo ohrožuje bezpečnost personálu.
- **Závažnost 2** Chyba, která má nepříznivý vliv na provedení provozní funkce nebo funkce nezbytné pro splnění úkolů a pro které nejsou k dispozici žádná přijatelná alternativní řešení.

- **Závažnost 3** Chyba, která má nepříznivý vliv na plnění provozní nebo základní funkce, pro kterou jsou k dispozici přijatelná alternativní řešení.
- **Závažnost 4** Chyba, která je pro uživatele / uživatele nepříjemná a ovlivňuje provozní funkce nebo funkce nezbytné pro splnění úkolů.
- **Závažnost 5** Jakékoliv jiné chyby

Alternativně je závažnost chyb klasifikována jako „kritická“, „vysoká“, „střední“, „nízká“ nebo „triviální / kosmetická“. Tyto úrovně závažnosti také nejsou standardizovány, protože dopady se liší podle odvětví: chyba, která způsobí havárii videohry, má mnohem menší dopad než chyba v softwaru pro řízení letu nebo kódu, který běží v lékařském zařízení. To je jeden z mnoha důvodů pro umístění závažnosti chyby do samostatné kategorie od její priority pro opravu, stejně jako kvantifikace a řízení těchto dvou odděleně.

3 Nástroje pro správu chyb

3.1 Jira

Jira je proprietární produkt pro sledování problémů vyvinutý společností Atlassian[3], který umožňuje sledování chyb a agilní řízení projektů. Jira je placený softwarový nástroj se zaměřením na týmy o velikosti 10 a více vývojářů. Nabízí jak cloudové řešení, tak možnost vlastního serveru pro běh Jiry. Největší popularitu získala v roce 2012, kdy byl do programu přidán takzvaný *Marketplace*. Marketplace umožňuje vývojářům třetích stran nabízet pluginy pro projektové řízení pro společnost Jira.

Jira nabízí možnost definování položek. Každá položka má řadu souvisejících informací včetně:

- Typ položky
- Souhrn
- Popis problému
- Projekt, ke kterému položka patří
- Komponenty v rámci projektu, které jsou s tímto problémem spojeny
- Verze projektu, které jsou tímto projektem dotčeny
- Verze projektu, které problém vyřeší
- Prostředí, ve kterém k problému dochází
- Priorita pro opravu
- Přiřazený vývojář
- Reportér - uživatel, který zadal problém do systému
- Současný stav
- Úplný záznam historie všech změn v poli, ke kterým došlo
- Komentáře od uživatelů
- Pokud je problém vyřešen - rozhodnutí

Dále nabízí pole pro určení časové náročnosti pro vyřešení chyby a aktuální stav věnovaného času. Je schopna spojit informace ze zvoleného verzovacího systému (podpora Bitbucket, GitHub, GitHub Enterprise a Fisheye) a přiřadit dané commity k dané chybě a umožňuje přidání vlastních polí podle potřeby.

Veškeré informace uvedené výše se dají z venčí přečíst pomocí *JIRA REST API* . Toto API umožňuje nejen čtení, ale i zápis dat. Například takto je možné získat informace o sdružených commitech pomocí dodávaného API: `https://<Path_To_JIRA>/rest/dev-status/1.0/issue/detail?issueId=<Issue_ID>&applicationType=githube&dataType=repository` Výsledek je pak vidět na obrázku Obrázek 3.1. Odpověď je ve formátu JSON a jsou na něm vidět informace jako ID commitu, kdo ho vytvořil a případně jaké soubory byly upraveny.


```
"repositories": [
  {
    "name": "dip_commit_classification",
    "url": "https:\/\/github.com\/fkolenak\/dip_commit_classification",
    "commits": [
      {
        "id": "02ca78c7130195aa386ddcad64cbb2ae8512c011",
        "displayId": "02ca78c",
        "authorTimestamp": "2019-04-12T13:20:22.000+0000",
        "url": "https:\/\/github.com\/fkolenak\/dip_commit_classification\/commit\/02ca78c7130195aa386ddcad64cbb2ae8512c011",
        "author": {
          "name": "Frantisek Kolenak",
          "avatar": "https:\/\/avatars3.githubusercontent.com\/u\/8156803?v=4",
          "url": "https:\/\/github.com\/fkolenak"
        },
        "fileCount": 3,
        "merge": false,
        "message": "BUG-1 update JIRA",
        "files": [
          {
            "path": "Tex\/literatura.bib",
            "url": "https:\/\/github.com\/fkolenak\/dip_commit_classification\/commit\/02ca78c7130195aa386ddcad64cbb2ae8512c011#diff-0",
            "changeType": "MODIFIED",
            "linesAdded": 20,
            "linesRemoved": 8
          },
          {
            "path": "Tex\/vzor_prace.pdf",
            "url": "https:\/\/github.com\/fkolenak\/dip_commit_classification\/commit\/02ca78c7130195aa386ddcad64cbb2ae8512c011#diff-1",
            "changeType": "MODIFIED",
            "linesAdded": 0,
            "linesRemoved": 0
          },
          {
            "path": "Tex\/vzor_prace.tex",
            "url": "https:\/\/github.com\/fkolenak\/dip_commit_classification\/commit\/02ca78c7130195aa386ddcad64cbb2ae8512c011#diff-2",
            "changeType": "MODIFIED",
            "linesAdded": 39,
            "linesRemoved": 12
          }
        ]
      }
    ]
  }
],
```


Obrázek 3.1: Zkrácená odpověď na detail o commitech

3.2 Github

3.3 Bugzilla

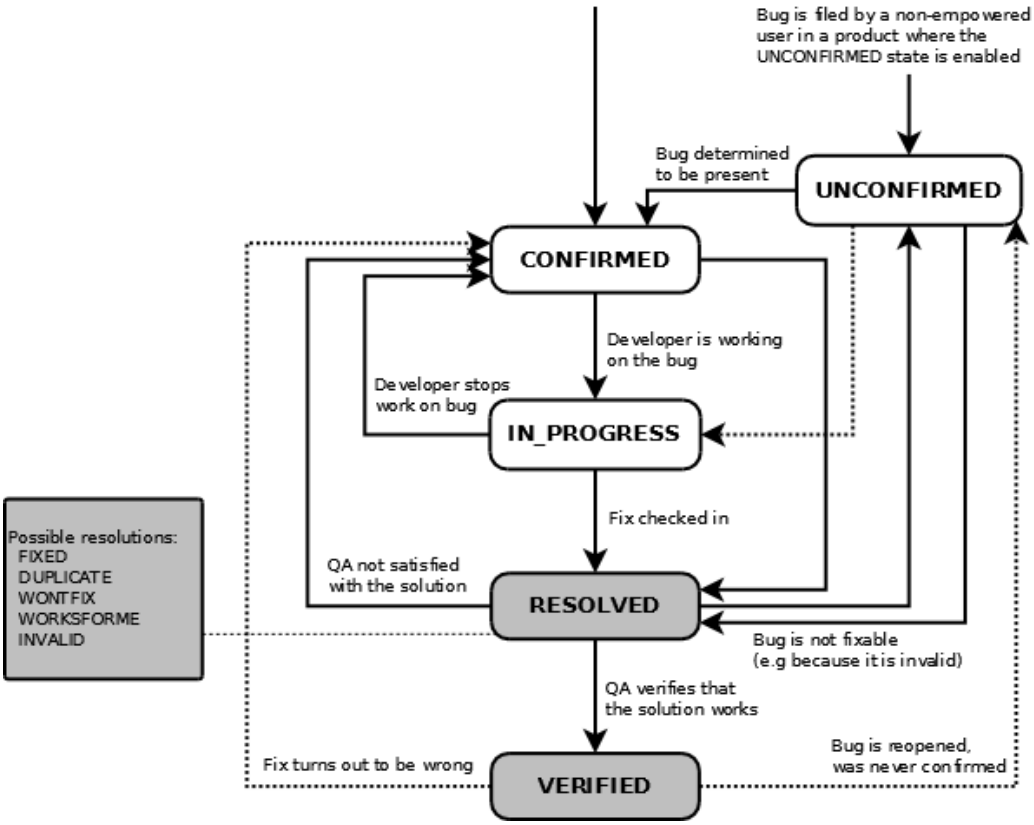
Bugzilla[2] je robustní a výkonný systém pro sledování defektů nebo systém pro sledování chyb. Má jednoduché funkce pro sledování defektů, které jsou zabudovány do integrovaných prostředí správy zdrojových kódů, jako jsou Github nebo jiné webové nebo lokálně nainstalované ekvivalenty. Bugzilla umožňuje správu pracovních postupů nebo kontrolu viditelnosti chyb (zabezpečení) nebo vlastní pole.

Bugzillu je možno upravovat podle libosti, neboť je open source. To znamená, že nejsou spojeny žádné náklady pro pořízení licence. Bugzilla má webové rozhraní a musí být nainstalována na vlastním serveru k tomu, aby ji bylo možno používat. To sebou nese výhody i nevýhody. Výhodou vlastního serveru je možnost uzavření serveru do vlastní sítě a tím pádem i zabezpečení. Nevýhodou jsou pak náklady na vlastní server a personál pro správu serveru.

Na obrázku Obrázek 3.3 je naznačen výchozí životní cyklus bugu. Tento cyklus se dá upravit podle potřeb organizace. Vlastnosti bugu se navíc dají upravit podle libosti. Vzhledem k tomu že vlastnosti jsou si dost podobné, například oproti Jiře, je zbytečné všechny vyjmenovávat. Hlavními pořád zůstává popis, priorita a stav. Odlišností může být pole osobních značek, které jsou viditelné pouze pro autora značky. Toho se dá využít například pro snadné nalezení takto označeného bugu. Další zvláštností jsou „flagy“. Ty mohou sloužit pro plánování, v tomto případě říkají, zda bude daný bug opraven nebo ne.

Integrace s verzovacím systémem je velice jednoduchá, stačí připojit požadovaný systém k Bugzille. Po připojení se propojují commity pomocí jejich popisu. K propojení dojde pokud v commitu Bugzilla nalezne klíčová slova „bug“ nebo „issue“ a číslo daného bugu nebo problému. Příkladem pak může být následující commit: „Tento commit opravuje *issue 12, bug 18* a *bug 92*.“. Od verze 5.0 nabízí Bugzilla i REST API, které je poté možno využít pro získání dat pro aplikace třetích stran. Pro starší verze Bugzilly je možné použít *BzAPI*, které podporuje verzi 3.4 a vyšší, nevýhodou však je, že je to samostatný serverový software.

3.4 Redmine



Obrázek 3.2: Životní cyklus bugu. [2]

4 Klasifikační metody

Literatura

[1] *Managing the Software Process*. SEI series in software engineering. Pearson Education, 1989. Dostupné z: https://books.google.cz/books?id=ZUXFqK1rwoEC. ISBN 9788177583304.

[2] *Bugzilla:: bugzilla.org* [online]. bugzilla.org, 2019. [cit. 2019/04/14]. Bugzilla. Dostupné z: https://www.bugzilla.org.

[3] *Jira Issue and Project Tracking Software Atlassian* [online]. Atlassian, 2019. [cit. 2019/04/12]. Jira bugtracking software. Dostupné z: https://www.atlassian.com/software/jira.

[4] Jorgensen, P. *Software Testing: A Craftsman's Approach, Fourth Edition*. CRC Press, 2016. ISBN 9781498785785.