Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Future MOD

Dokumentácia k riadeniu projektu

Verzia 2.0

Tím č.17 : Future MOD

Vedúci projektu: Ing. Peter Pištek PhD.

Predmet: Tímový projekt I

Študijný program: Softvérové inžinierstvo, ročník: 1

Akademický rok: 2016/2017, zimný semester

Tabuľka 1 História zmien

Verzia	Dátum	Autor	Popis
1. kontrolný bod	6.10.2016	Matúš Pohančeník	Vytvorenie dokumentu
	18.11.2016	Tomáš Baránek	Metodika evidencie úloh
	18.11.2016	Matúš Slovík	Metodika komunikačných kanálov
	18.11.2016	Lukáš Mastiľak	Sumarizácia šprintov
	19.11.2016	Lukáš Mastiľak	Export úloh
	19.11.2016	Jaroslav Tóth	Metodika k verziovaniu
	19.11.2016	Jaroslav Tóth	Definition of Done (DSoD)
	19.11.2016	Tomáš Sokolík	Role členov tímu
	19.11.2016	Matúš Pohančeník	Metodika Dokumentácie
	19.11.2016	Radoslav Zápach	Metodika správy zdrojov
	20.11.2016	Matúš Pohančeník	Manažment rozvrhu
	20.11.2016	Matúš Pohančeník	Manažment dokumentácie
	20.11.2016	Matúš Pohančeník	Používané metodiky
	20.11.2016	Matúš Slovík	Manažment komunikácie
	20.11.2016	Tomáš Baránek	Úvod
2. kontrolný bod	13.12.2016	Lukáš Mastiľak	Export úloh
	13.12.2016	Lukáš Mastiľak	Sumarizácia šprintu 3
	13.12.2016	Lukáš Mastiľak	Sumarizácia šprintu 4
	13.12.2016	Lukáš Mastiľak	Manažment testovania
	13.12.2016	Tomáš Sokolík	Konvencie písania zdrojového kódu
	13.12.2016	Tomáš Sokolík	Manažment code review
	13.12.2016	Jaroslav Tóth	Doplnenie metodiky k verziovaniu
	13.12.2016	Tomáš Sokolík	Metodika code review
	13.12.2016	Matúš Pohančeník	Globálna retrospektíva

Obsah

1 Úvod ... 1

2 Role členov tímu ... 2

3 Aplikácia manažmentov ... 5

3.1 Manažment komunikácie ... 5
3.2 Manažment dokumentácie ... 5
3.3 Manažment rozvrhu(časové plánovanie) ... 5
3.4 Manažment testovania ... 6
3.5 Manažment code review ... 6

4 Sumarizácia šprintov ... 7

4.1 Šprint 1 ... 7
4.2 Šprint 2 ... 8
4.3 Šprint 3 ... 8
4.4 Šprint 4 ... 9

5 Používané metodiky ... 11

5.1 Metodika evidencie úloh ... 11
5.2 Definition of Done (DoD) ... 11
5.3 Metodika správy zdrojov ... 11
5.4 Metodika komunikačných kanálov ... 11
5.5 Metodika k verziovaniu ... 11
5.6 Metodika dokumentácie ... 11
5.7 Metodika testovania ... 12
5.8 Metodika code review ... 12

6 Globálna retrospektíva ... 13

6.1 Zimný semester ... 13

7 Prílohy ... i

A. Exportovanie úloh ... i
B. Metodika evidencie úloh ... ii
C. Definition of Done (DoD) ... iv
D. Metodika správy zdrojov ... vii
E. Metodika komunikačných kanálov ... viii
F. Metodika k verziovaniu ... x
G. Metodika dokumentácie ... xiv
H. Konvencie písania zdrojového kódu ... xv
I. Metodika pre code review ... xxvi
J. Metodika testovania ... xxviii

1 Úvod

V každom tíme hrá dôležitú úlohu riadenie. V našom tíme bola zvolená agilná metóda riešenia úloh, čo sa v úvode nášho projektu veľmi ťažko realizovalo, nakoľko sme od zákazníka dostali príliš veľa analytických zadaní. Dĺžka šprintu bola stanovená na 2 týždne no spočiatku sa veľmi ťažko plnila aj vzhľadom na špecifickosť projektu.

V úvode semestra boli rozdelené základné, nevyhnutné manažérske úlohy a postupne sa tieto manažérske úlohy obmieňali prípadne dopĺňali. Tieto úlohy sú opísané v časti 2.

Ďalšia časť dokumentu sa venuje aplikácií manažmentov, ktoré sa riadia metodikami, ktoré sú tiež obsahom tohto dokumentu. Tieto metodiky vznikajú postupne počas práce na projekte.

V časti číslo 3 môžete nájsť sumarizáciu doposiaľ ukončených šprintov.

Dokument obsahuje v prílohách jednotlivé metodiky, ktoré tím aplikoval v rámci riadenia tímového projektu.

2 Role členov tímu

Členovia tímu

Tomáš Baránek

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Internetové technológie. Má skúsenosti s programovacími jazykmi Java, C a C#. V rámci bakalárskej práce sa venoval tvorbe mobilnej aplikácie pre Android, ktorá bola použitá v prvej slovenskej stratosférickej sonde. Momentálne pracuje ako junior Android developer.

Dlhodobé úlohy: vedúci tímu, manažér úloh

- Plánovanie formálnych a neformálnych stretnutí
- Dohľad nad evidenciou úloh
- Dohľad nad priraďovaním Definition of Done pri vytváraní úloh

Lukáš Mastiľak

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Internetové technológie. Má skúsenosti s programovacími jazykmi C, C# a Python. Absolvoval 4 semestre sieťového kurzu CCNA.

Dlhodobé úlohy: manažér testovania

- Dohľad nad tvorbou testov
- Spravovanie nástrojov pre automatizované a manuálne testovanie

Matúš Pohančeník

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Internetové technológie. Má skúsenosti s programovacími jazykmi Java a C. V rámci bakalárskej práce sa venoval tvorbe mobilnej aplikácie pre Android, ktorá bola použitá v prvej slovenskej stratosférickej sonde. Momentálne pracuje ako junior AngularJS developer.

Dlhodobé úlohy: manažér dokumentácie

- Dohľad nad tvorbou dokumentácie
- Spravovanie úložiska dokumentácie
- Spájanie častí dokumentácie

Matúš Slovík

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Internetové technológie. Má skúsenosti s administráciou serverov a programovacími jazykmi Java, C a Perl.

Absolvoval 2 semestre Cisco certifikátov. Momentálne pracuje ako člen technického tímu pre podporu a vývoj aplikácie.

Dlhodobé úlohy: manažér komunikácie

- Udržiavanie komunikácie medzi členmi tímu
- Spravovanie nástrojov pre komunikáciu
- Spravovanie nástroja na prideľovanie úloh v rámci agilného vývoja softvéru

Tomáš Sokolík

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Internetové technológie. Má skúsenosti s programovacími jazykmi Java, C a C# a frameworkom .NET.

Dlhodobé úlohy: manažér kvality zdrojového kódu

- Dohľad nad dodržovaním konvencií pre zdrojový kód
- Dohľad nad dodržovaním pravidiel pre Code review

Jaroslav Tóth

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Internetové technológie. Má skúsenosti s SQL, návrhom a konfiguráciou počítačových sietí a programovacími jazykmi Java, C a C#. Absolvoval CCNA Routing and Switching a CCNA Security.

Dlhodobé úlohy: manažér verzií

- Zadefinovanie a udržiavanie pravidiel Definition of Done (DoD).
- Dohľad nad verziovaním zdrojového kódu

Radoslav Zapach

Bakalárske štúdium absolvoval na FIIT STU v Bratislave v odbore Informatika. Má skúsenosti s SQL a špecifikáciou JPA s využitím Hibernate a EclipseLink a programovacími jazykmi Java, JavaScript a AngularJS. Momentálne pracuje ako Java developer.

Dlhodobé úlohy: manažér webovej stránky

- Správa a aktualizácia webovej stránky projektu
- Dohľad nad štruktúrou ukladacieho miesta a správa zdrojov

Tabuľka 2 Krátkodobé úlohy členov tímu

Krátkodobá úloha	**Členovia tímu**
Scrum master	Tomáš Baránek, Matúš Pohančeník
JIRA manažér	Tomáš Baránek, Matúš Slovík
Písanie zápisnice zo stretnutia	Všetci

3 Aplikácia manažmentov

Táto časť opisuje aplikáciu manažmentov v rámci riadenia tímu a vývoja tímového projektu. Opisuje tiež hodnotenia metodík, ktoré boli v rámci tímu vytvorené.

3.1 Manažment komunikácie

Komunikácia je hlavným pilierom tímu, bez ktorého by tím nevedel efektívne napredovať. To si uvedomujeme aj v našom tíme a preto sa ju neustále snažíme zlepšovať. Okrem formálneho tímového komunikačného kanála, ktorým je nástroj Slack, členovia medzi sebou komunikujú aj na svojich súkromných účtoch alebo osobne, a tak udržiavajú medzi sebou neustále kontakt.

Zároveň sa členovia tímu stretávajú aj vo voľnom čase mimo dohodnutých tímových stretnutí. Takéto neformálne stretnutia zlepšuje medzi členmi vzťahy, pretože často na nich padnú slová, ktorých sa na formálnych stretnutiach zdržali. Takáto forma otvorenosti je pre tím veľkým plusom.

Formálna komunikácia je opísaná v časti Metodika komunikačných kanálov.

3.2 Manažment dokumentácie

Dokumentovanie jednotlivých úloh prebiehalo priebežne v rámci riešenia pridelených úloh. Vytvorenie dokumentu bolo aj súčasťou Definition of Done prideľovaných úloh. Vďaka tomu tím pred finálnym odovzdaním iba vložil svoje vypracované časti do výsledného dokumentu k dokumentácii projektu.

Celkový výsledok dokumentácii je tvorený všetkými jej členmi. Tím si rozdelil jednotlivé časti dokumentácie, ktorých spojením vznikol tento dokument.

Problémom pri riešení finálneho dokumentu boli niektoré nesprávne definované štýly, ktoré sú v dokumente používané. Vďaka postrehom jednotlivých členov tímu a ich nasadeniu pri tvorbe dokumentácie to však kvalitu výsledného dokumentu neovplyvnilo.

3.3 Manažment rozvrhu(časové plánovanie)

Stretnutia sú z pravidla organizované jeden krát týždenne, pričom každý druhý týždeň prebieha plánovanie šprintu. Každý šprint má trvanie dvoch týždňov a preto stretnutia v rámci šprintu slúžia na diskutovanie aktuálneho stavu pridelených úloh. V prípade potreby sa členovia tímu stretávajú viac dní v týždni a to aj za neúčasti vedúceho, kedy sa preberajú technické záležitosti v rámci analýzy a návrhu projektu.

Na konci šprintu prebieha retrospektíva práce tímu a zároveň jej jednotlivých členov. Prezentujú sa vypracovania pridelených úloh a ich výsledky sa komunikujú s produktovým

vlastníkom. Ukončenie šprintu sa zväčša končí predbežným návrhom použitých technológií, na základe schválenia produktovým vlastníkom.

3.4 Manažment testovania

Testovanie je potrebné pre odhalenie chýb v skorej fáze projektu, aby sa minimalizoval dopad chýb na projekt a znížilo sa riziko neúspechu projektu. Preto sme zaviedli niekoľko druhov testov. Každý člen tímu je povinný písať a vykonávať unit testy pri implementácií novej funkcie. Pri integrovaní komponentu do systému je žiadané prv napísať a vykonať integračný test. Až po úspešnom vykonaní integračného testu je možné komponent integrovať do systému. Posledný druh testu je akceptačný test. Tento test sa vykonáva na základe vygenerovaných akceptačných testov z prípadov použitia a testovania so zákazníkom, či je to funkcionalita, ktorú on chce.

3.5 Manažment code review

Zdrojový kód by mal byť napísaný čo najzrozumiteľnejšie, čiže čo najjednoduchšie a najprehľadnejšie. To je možné docieliť definovaním a dodržiavaním jednotných konvencií a zásad pre všetkých vývojárov v rámci vývojového tímu pre písanie zdrojového kódu. Pri procese code review si zdrojový kód po autorovi prečíta iný člen tímu za účelom nájdenia rôznych nedostatkov a chýb v porovnaní s definovanými konvenciami. Vykonávanie code review je možné až po dostatočnom otestovaní daného zdrojového kódu autorom. Každý člen tímu má vopred priradeného posudzovateľa, ktorého môže poveriť na vykonanie code review. Ak posudzovateľ pri code review nájde v danom zdrojovom kóde určité nedostatky a chyby, je povinný o nich poskytnúť autorovi daného zdrojového kódu spätnú väzbu vo forme komentárov. Autor zdrojového kódu je následne povinný si nájdené chyby opraviť a znovu poveriť posudzovateľa na code review.

4 Sumarizácia šprintov

4.1 Šprint 1

Prvý šprint bol rozdelený na dve častí, ktoré sa riešili paralelne. Prvá časť bola zameraná na organizačnú stránku tímu a druhá časť bola zameraná na zoznámenie sa s projektovým manažérom.

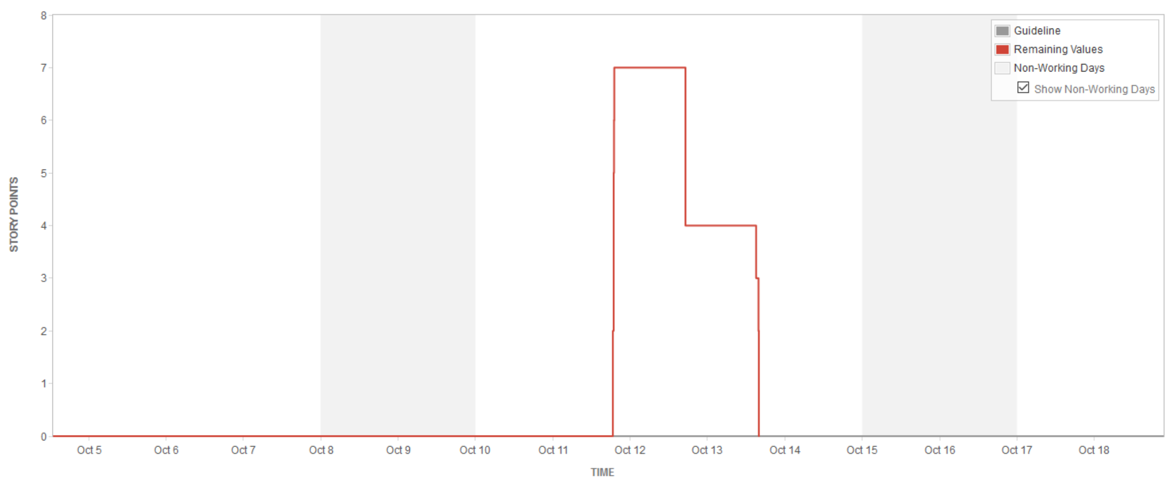
V rámci organizácie tímu sme si rozdelili roli v tíme. Určili sme si kto je zodpovedný za dokumentáciu, komunikáciu a prehliadky kódu.

Základný komunikačný nástroj sme si vybrali Slack. Ďalej sme si určili, že na evidenciu a manažovanie úloh použijeme nástroj Jira. Tiež sme sa dohodli, že na ukladanie a riadenie verzii kódu použijeme webové úložisko GitHub. Tieto tri nástroje sme navzájom prepojili a následne sme sa zoznámili so základným používaním týchto nástrojov. Pre ukladanie všetkých dokumentov, ktoré sa vytvoria počas tohto projektu, sme vybrali úložisko na OneDrive. V tejto časti sa riešilo aj vytvorenie a nasadenie webovej stránky tímu.

V organizačne časti sa tiež riešila metodológia pre riadenie tímu. Vybraný člen tímu naštudoval pojem „Definition of done“ a jeho typy. Výsledok potom prezentoval celému tímu. Tiež sa vytvorili metodiky pre dokumentáciu a komunikáciu tímu.

V druhej časti šprintu sa uskutočnilo prvé stretnutie s projektovým manažérom. Projektový manažér nám prezentoval základnú predstavu o systéme a vypísal na tabuľu biznis požiadavky, ktoré sme si mali rozbiť na menšie požiadavky a následne ich analyzovať, aby sme dostali požiadavky na systém.

V retrospektíve šprintu, väčšina členov hodnotila za najväčšie pozitívum tohto šprintu ochotu sa stretnúť aj mimo času vyhradeného pre tímový projekt. Niektorí členovia navrhli popracovať na zlepšení komunikácie a plánovania úloh.



Obrázok 1 Burndown chart pre šprint 1

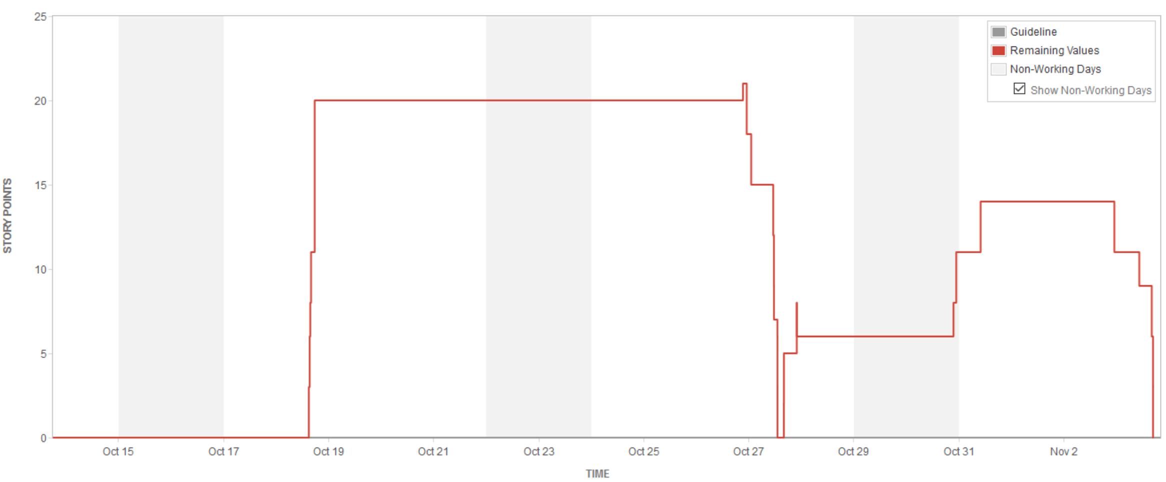
4.2 Šprint 2

Na začiatku tohto šprintu sme vybrali sedem biznis požiadaviek z tých, ktoré sme získali zo stretnutia s projektovým manažérom. Následne sa tieto požiadavky prediskutovali a určili sa body, ktoré by mali byť obsiahnuté v našich analýzach. Pri niektorých požiadavkách sa určilo vytvorenie rozhodovacej matice, aby sa pre ďalšie časti projektu vybralo najlepšie možné riešenie.

Na konci šprintu sa konalo stretnutie s projektovým manažérom, ktorému sme prezentovali naše analýzy a rozhodovacie matice. Projektový manažér mal výhrady takmer ku každej vybranej oblasti, že sme buď zabudli zohľadniť niektoré parametre alebo sme sa nezamysleli nad celým postupom riešenia daného problému. Počas tohto stretnutia sme si zaznamenali všetky jeho výhrady. Po skončení tohto stretnutia sme sa rozhodli predlžiť šprint a dopracovať jednotlivé úlohy, aby spĺňali jeho požiadavky.

V retrospektíve sa zhodol tím, že najväčším pozitívom tohto šprintu bolo zlepšenie komunikácie medzi členmi tímu. Niektorí členovia tímu ako negatívum hodnotili, že veľa času sa venuje analýze alebo že málo chápeme celému systému. Tiež padol návrh na lepšiu organizáciu vypracovania analýz, lebo jedno zle rozhodnutie môže ovplyvniť analýzy aj iných členov.

Väčšina členov tímu sa zhodla, že by sme sa mali zúčastniť TP Cupu. Následne sa vyplnila a odovzdala prihláška do súťaže.



Obrázok 2 Burndown chart pre šprint 2

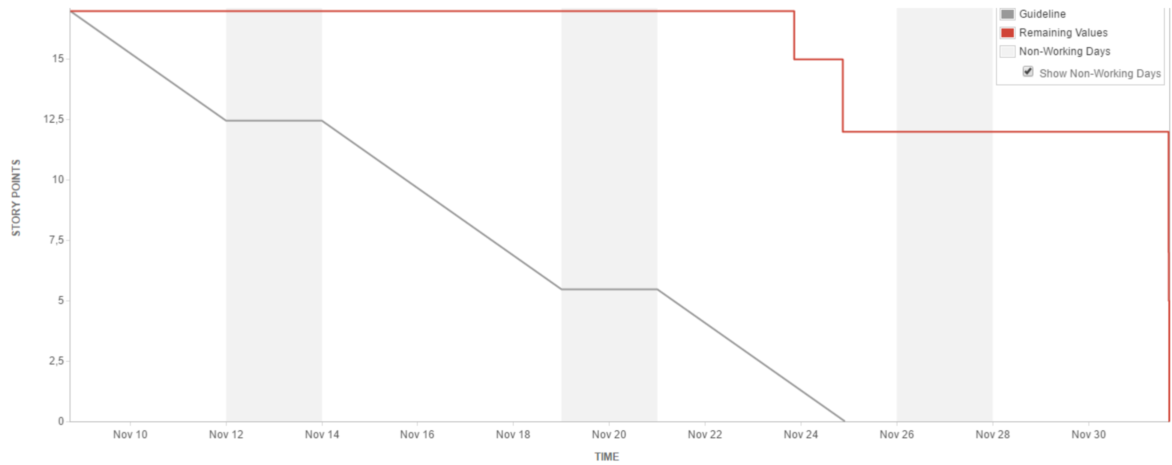
4.3 Šprint 3

V tomto šprinte sa pokračovalo v analýze a zároveň sa pridali prvé návrhy ako by mohli vyzerať jednotlivé veci. V rámci web portálu sa riešilo, ktoré konzoly sa nachádzajú v starom systéme a určilo sa, ktoré konzoly sa použijú v novom systéme. Pre jednotlivé konzoly sa vytvoril prvý návrh v rámci, ktorého sa zadefinovalo akú funkcionalitu daná konzola bude ponúkať a pre koho. Ďalej sa riešili možnosti zobrazovania reklám počas

prehrávania filmu. Tiež sa tu navrhol prototyp na zahodenie v ktorom sme si otestovali spôsob prehratia reklamy počas filmu.

Veľa času sa venovala návrhu ukladacieho priestoru. Tu sa navrhla vhodná, adresárová štruktúra, aj na základe analýzy starého systému. Dôležitá časť sa venovala návrhu zabezpečenia adresárovej štruktúry.

V retrospektíve sa zhodol tím, že najväčším pozitívom tohto šprintu bola práca na úlohách v dvojici, stretnutie u zákazníka, zlepšenie komunikácie a stretávanie sa aj mimo oficiálneho času. Za negatívum tím označil horšia efektivita práce a prevládajúca zlá nálada v tíme.



Obrázok 3 Burndown chart pre šprint 3

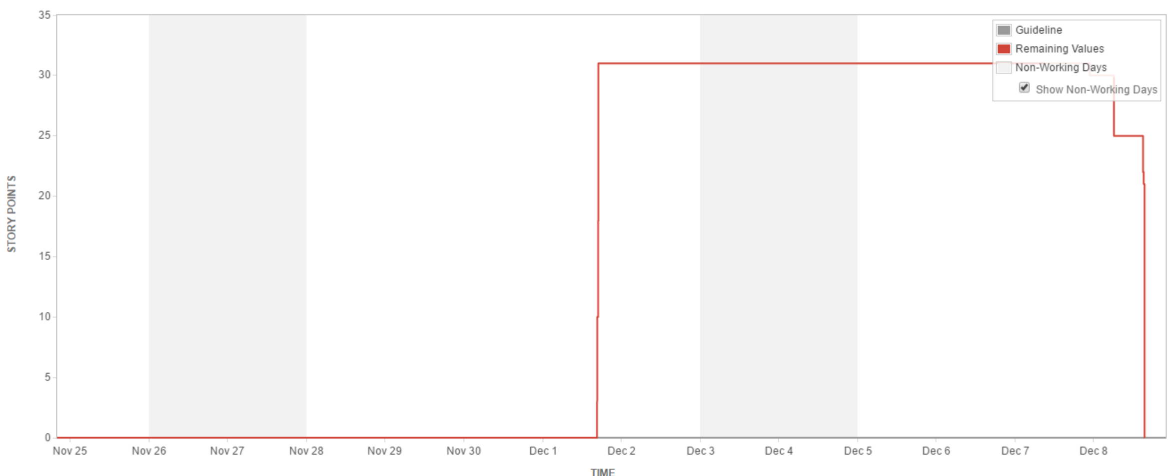
4.4 Šprint 4

V tomto šprinte sa vytvorili diagramy prípadov použitia a napísali sa scenáre pre jednotlivé prípady použitia. Následne sa spísali všetky doteraz zistené funkcionálne a nefunkcionálne požiadavky. Pokračovalo sa v návrhu architektúry systému, kde sme jednotlivé požiadavky rozbili do komponentov. Navrhol sa diagram komponentov. Ďalej sa riešili možnosti komunikácie medzi jednotlivými komponentami. Následne sa začalo premýšľať nad jednotlivými vrstvami softvérovej architektúry a ako budú navzájom vrstvy prepojené.

V rámci tohto šprintu sme absolvovali aj stretnutie s projektovým manažérom, ktorému sme prezentovali zatiaľ dosiahnuté výsledky. Projektový manažér vyslovil niekoľko pripomienok, ktoré sa zaznamenali a budú sa riešiť v nasledujúcom šprinte. Vyjadril sa, že je zatiaľ spokojný s dosiahnutými výsledkami.

Zaviedli sa a napísali sa dve nové metodiky, a to pre prehliadku kódu a testovanie v programovacích jazykoch Java a C.

V retrospektíve sa tím zhodol na tom, že sme mali nedostatočné riadenie úloh, to sa prejavilo v podobe toho, že sme si zobrali do šprintu priveľa úloh, aj napriek tomu, že sme mali v rámci štúdia priveľa úloh na riešenie. Ďalším negatívom bolo, že členovia tímu nedávali pozor pri riešení a diskutovaní iných častí projektu, než sa ich týkali. Vyhodnotili sme za pozitívum šprintu konzultovanie navrhovanej architektúry s osobou mimo náš projekt, ktorá nám dala odborné rady a nápady z danej oblasti. Ďalším pozitívom šprintu bolo viacnásobné riešenie úloh v podobe spoločného stretnutia tímu v labe, čím sa zlepšila komunikácia medzi členmi tímu.



Obrázok 4 Burndown chart pre šprint 4

5 Používané metodiky

Tím sa pri práci na projekte riadi viacerými metodikami, preto boli vypracované metodiky, ktoré sú opísané v tejto časti dokumentu.

5.1 Metodika evidencie úloh

Táto časť zabezpečuje opis jednotnej metodiky evidencie úloh v rámci práce na tímovom projekte. Definuje pozície členov tímu, rozsahy jednotlivých úloh, stavy úloh od ich vytvorenia, až po ich pridelenia a následné vykonanie.

5.2 Definition of Done (DoD)

K rôznym typom úloh, ktoré sú zašpecifikované v kapitole B (Metodika evidencie úloh) je potrebné formálne zašpecifikovať, kedy je špecifická úloha ukončená. DoD určuje, aké aspekty práce je potrebné vykonať na to, aby sme mohli označiť user story/šprint/release ako ukončený. DoD charakterizuje a smeruje prácu v tíme.

5.3 Metodika správy zdrojov

Táto časť popisuje možnosti a miesta uloženia rôznych prevzatých, alebo tímom vytvorených zdrojov. Zabezpečuje členom tímu jednoduchý prístup ku všetkým informáciám, ktoré v rámci vypracovanie projektu potrebujú.

5.4 Metodika komunikačných kanálov

Táto časť popisuje spôsoby komunikácie členov tímu v rámci riešenie tímového projektu. Opisuje jednotlivé kanály pre jednoduché pochopenie ich účelu a využitia.

5.5 Metodika k verziovaniu

Účelom tejto časti dokumentu (metodika na správu verzií) je prezentovanie pravidiel, ktoré musia byť dodržiavané pri verziovaní zdrojového kódu v projekte. Obsahuje tiež opis operácií, ktoré sú vykonávané pri verziovaní zdrojového kódu.

5.6 Metodika dokumentácie

Táto časť popisuje spôsoby tvorby dokumentácie a uvádza štruktúru a štýly, ktoré bude tím používať. Využije sa pri tvorbe všetkých dokumentov v rámci tímového projektu, ktoré sa budú nachádzať v dokumentácii inžinierskeho diela aj dokumentácii riadenia.

5.7 Metodika testovania

Účelom tejto metodiky je zadefinovať jednotný spôsob tvorby unit testov, integračných testov a akceptačných testov. Táto metodika sa dodržiava počas celého projektu.

5.8 Metodika code review

Účelom tejto metodiky je definovanie pravidiel a postupov, ktoré je nutné dodržiavať pri vykonávaní code review. Dodržiavanie jednotných pravidiel pre písanie kódu zvyšuje jeho celkovú kvalitu a kód je tak ľahší na pochopenie pre iných programátorov, preto je kontrola týchto pravidiel pri implementovaní dôležitá.

6 Globálna retrospektíva

6.1 Zimný semester

V práci na projekte náš tým na začiatku doplácal na nedostatočné skúsenosti s agilným vývojom a odhadovaním bodov pre jednotlivé user story. Rovnako boli problémy s plánovaním neformálnych stretnutí a komunikáciou členov tímu. Postupnými skúsenosťami a aplikovaním vypracovaných metodík sa práca nášho tímu výrazne zlepšila. Vďaka vytvoreniu spoločného kalendáru bolo uľahčené plánovanie neformálnych stretnutí, potrebných pre vypracovanie spoločných úloh. Časté neformálne stretnutia výrazne zlepšili medziľudské vzťahy v rámci tímu a uľahčili komunikáciu členov.

Niektoré metodiky sa formovali počas priebehu semestra, patria sem napríklad metodika dokumentácie a komunikácie, v ktorých sme v rámci fungovania tímu narazili na problémy. Vďaka týmto zmenám sme dokázali zlepšiť fungovanie v rámci tímu.

Prácu tímu výrazne ovplyvňovali pravidelné stretnutia s produktovým vlastníkom, ktorý pripomienkoval dosiahnutý progres a smeroval vývoj produktu svojimi poznatkami z danej oblasti. Produktový vlastník, ako certifikovaný scrum master, napomáhal svojimi radami v oblasti riadenia tímu a agilného vývoja a viedol tak tím k pravidelným zlepšeniam.

Ďalším pozitívnym impulzom pre náš tím bol mentoring v rámci Tímového projektu, kde sme mali možnosť stretnúť sa s viacerými odborníkmi v oblasti agilného vývoja, ktorý nám rovnako poskytli množstvo nových informácií. Cieľom nášho tímu bude všetky získané poznatky aplikovať v rámci letného semestra.

7 Prílohy

A. Exportovanie úloh

JIRA

JIRA

Displaying **18** issues at **13/dec/16 12:41 AM**.

Issue Type	Key	Summary	Assignee	Status	Priority	Sprint	Description
Story	FUT-102	Code Review (Inštraštruktúrna)	Tomáš Sokolík	Done	Medium	FUT Sprint 4	- Konvencie písania kódu - Ako sa bude realizovať code review - Ak je nástroj na code review, treba ho rozbehať
Story	FUT-101	Virtuálny server (Infraštruktúrna)	Jaroslav Tóth	Done	Medium	FUT Sprint 4	- Do 27.11. dodať aplikácie, ktoré treba nainštalovať na server - Nainštalovanie a nakonfigurovanie aplikácií - Dual boot mechanizmus - GitHub - Server veľkosť flash 8GB a na média cca 25GB, nastaviť tak aby sa nedal rozširovať, RAM
Story	FUT-104	Funkcionálne a nefunkcionálne požiadavky	Tomáš Baránek	Done	Medium	FUT Sprint 4	- spísať požiadavky (každý za seba (pre tím)) - prekonzultovať v tíme - (stanoviť priority)
Story	FUT-103	Testovanie	Lukáš Mastiľák	Done	Medium	FUT Sprint 4	Ako budú realizované: - unit, - akceptačné, - integračné testy. - Čo sa musí zrealizovať pred realeasom
Story	FUT-106	Architektúra	Matúš Slovík	Done	Medium	FUT Sprint 4	Vyplynie z funkcionálnych požiadaviek Graf toku údajov Rozhrania aké kde Diagram komponentov Aké štýly, vzory
Story	FUT-113	Softvérová architektúra	Radoslav Zápach	Done	Medium	FUT Sprint 4	Aké budú vrstvy a ako budú realizované. Ktoré služby má na starosti web-portál a ktoré má na starosti systém. Ako budú prepojené C služby a WEB portál. Rozbiť na malé veci - diagram komponentov. Aké návrhové štýly použiť.
Sub-task	FUT-114	FUT-102 Konvencie pre Javu	Tomáš Sokolík	Done	Medium	FUT Sprint 4	
Sub-task	FUT-115	FUT-102 Konvencie pre C	Tomáš Sokolík	Done	Medium	FUT Sprint 4	
Sub-task	FUT-107	FUT-101 Nainštalovanie a nakonfigurovanie aplikácií	Jaroslav Tóth	Done	Medium	FUT Sprint 4	- Vytvorenie dokumentu, do ktorého budú členovia tímu dopisovať zoznam potrebných aplikácií. - Základné aplikácie, ktoré sú potrebné pre zabezpečenie rozhrania medzi vytvorenými aplikáciami, systémom a sieťou - inštalácia a konfigurácia. - Auto-recovery konfigurácia pre nainštalované aplikácie.
Sub-task	FUT-109	FUT-101 Nastavenie Github na serveri	Jaroslav Tóth	Done	Medium	FUT Sprint 4	- Vývoj softvérových modulov na serveri. - Vyriešenie doručovania
Sub-task	FUT-116	FUT-102 Konvencie pre bash skripty	Tomáš Sokolík	Done	Medium	FUT Sprint 4	
Sub-task	FUT-117	FUT-102 Code review manažment	Tomáš Sokolík	Done	Medium	FUT Sprint 4	
Sub-task	FUT-110	FUT-101 Vytvorenie obrazu servera	Jaroslav Tóth	Done	Medium	FUT Sprint 4	- Konfigurácia virtuálneho stroja. - Distribúcia vytvoreného virtuálneho stroja členom tímu. - Server: veľkosť flash 8GB a na média cca

Sub-task	FUT-108	FUT-101 Konfigurácia dual-boot mechanizmu	Jaroslav Tóth	Done	Medium	FUT Sprint 4	- Automatické spustenie záloženého jadra systému v prípade zlyhania primárneho jadra systému. - Logging poruchy. - Obnova pôvodného jadra systému.
Sub-task	FUT-118	FUT-104 Preštudovať typy požiadavok na softvér	Tomáš Baránek	Done	Medium	FUT Sprint 4	
Sub-task	FUT-119	FUT-104 Spísanie požiadavok + určiť priority	Tomáš Baránek	Done	Medium	FUT Sprint 4	
Sub-task	FUT-112	FUT-103 Integračné testy Java	Lukáš Mastiľák	Done	Medium	FUT Sprint 4	
Sub-task	FUT-111	FUT-103 Unit testy Java	Lukáš Mastiľák	Done	Medium	FUT Sprint 4	

Generated at Tue Dec 13 00:41:01 CET 2016 by Lukáš Mastiľák using JIRA 7.2.2#72004-sha1:9d5132893cc8c728a3601a9034a1f8547ef5c7be.

B. Metodika evidencie úloh

Účel dokumentu

Úlohou tohto dokumentu je vypracovanie jednotnej metodiky evidencie úloh v rámci práce na tímovom projekte.

Dedikácia metodiky

Metodika je určená všetkým členom tímu.

Tabuľka 3 Opis členov v riešení projektu

Rola	Úloha
Správca JIRY	Zodpovedá za správnu technickú funkčnosť JIRY, taktiež má na starosti správu používateľských kont (strata mena, hesla, správna funkčnosť).
Člen tímu	Evidencia svojich úloh v nástroji JIRA (zmena statusov úloh, správny popis úlohy).
Product owner	Definovanie príbehov a k ním príslušným rizík, DoD.
Dočasný správca JIRY	Zodpovedá za správne vytvorenie úlohy v nástroji JIRA počas stretnutia na ktorom sa plánuje nový šprint.

Typy úloh

Úlohou je chápaná činnosť, ktorú treba vykonať v rámci tímového projektu. Nakoľko sa v tíme postupuje agilnou metódou SCRUM sú definované viaceré druhy úloh.

Úlohy z hľadiska určenia rozdeľujeme na:

1. **Hlavné úlohy**

Do tejto skupiny úloh spadajú úlohy, ktoré sú viditeľné pre zákazníka, teda majú priamy dosah na zákazníka. Do tejto skupiny úloh patrí napríklad úloha: Vytvorenie vstupného prihlasovacie formulára.

Tabuľka 4 Opis štuktúrovanie úloh 1

Jira	Názov	Popis úlohy
Story	Story (príbeh)	Príbeh je úloha, ktorá sa rieši v rámci šprintu a jej ukončenie vo veľkej miere závisí od spokojnosti a akceptovania zo strany zákazníka. Ide o väčšiu úlohu, ktorá obsahuje viacero tzv. Sub-taskov.
Sub-Tasks	Sub-task (pod úloha)	Jeden príbeh v rámci šprintu sa delí na viacero pod úloh.

2. **Vedľajšie úlohy**

Druhú skupinu úloh tvoria úlohy, ktoré priamo nesúvisia s tým, čo musí zákazník schváliť / akceptovať. Typicky do tejto skupiny môžeme zahrnúť úlohy, ako Tvorba dokumentácia, Nasadenie servera, Tvorba prihlášky na TPcup.

Tabuľka 5 Opis štruktúrovania úloh 2

Jira	Názov	Popis úlohy
Task	Task (úloha)	Nová úloha v rámci vedľajších úloh.
Bug	Bug (chyba)	Túto úlohu vytvára tester, ktorý bol priradený k úlohe v ktorej sa vyskytla chyba. Chyba musí byť prepojená s úlohou v ktorej vznikla táto chyba. V JIRE je to zabezpečené parametrom Issue, kde sa napíše názov úlohy.

Priebeh tvorby úloh

1. **Čas pre vytváranie úloh**

Úlohy sa tvoria na začiatku šprintu, kedy je jeden z členov tímu zvolený za dočasného správcu JIRY. Jeho úlohou je vyberanie úloh z backlogu a predkladanie ich tímu na ďalšiu diskusiu o ich rozdelení do jednotlivých druhov úloh (Story, Task, Sub-task...).

2. Definovanie úlohy

V nástroji JIRA dočasný správca JIRY vybranej úlohe z backlogu podľa pokynov product ownera a členov tímu doplní popis, Definition of done (DoD), riziká.

3. Rozdelenie na pod úlohy

V prípade potreby je úloha rozdelená do viacerých pod úloh u ktorých sa samostatne definujú riziká, DoD, popis.

4. Ohodnotenie úloh

Každej úlohe je pridelený počet bodov podľa obťažnosti. Toto hodnotenie prebieha Scrum Planning Poker metódou. Je to metóda pri ktorej sa kolektívnym hlasovaním tím snaží dostať k jednému bodovému ohodnoteniu.

5. Pridelenie úlohy

Nástroj JIRA umožňuje ku každej úlohe pridať jedného zodpovedného, preto sa väčšie úlohy rozdeľujú do pod úloh, aby každá z nich mohla byť pridelená jednému členovi tímu.

Status úloh

- To Do – úloha je definovaná a pripravená na začiatok realizácie.
- In Progress – na úlohe sa aktuálne pracuje.
- Done – úloha spĺňa DoD.

C. Definition of Done (DoD)

Účel dokumentu

K rôznym typom úloh, ktoré boli zašpecifikované v kapitole B (Metodika evidencie úloh) je potrebné formálne zašpecifikovať, kedy je špecifická úloha ukončená. DoD určuje, aké aspekty práce je potrebné vykonať na to, aby sme mohli označiť user story/šprint/release ako ukončený. DoD charakterizuje a smeruje prácu v tíme.

Dedikácia metodiky

Metodika je určená pre všetkých členov tímu, ktorý pracujú na pridelených úlohách. Tabuľka 6 zobrazuje podrobné opis dedikácie metodiky.

Tabuľka 6. Dedikácia metodiky Definition of Done

Rola	Úloha
Správca JIRY alebo Dočasný správca JIRY	Zodpovedá za vytvorenie DoD (priradenie ku konkrétnej úlohe) na základe dohodnutého obsahu úlohy, typu úlohy a všeobecných pravidiel DoD. DoD sa vytvára pri vytváraní samotnej úlohy.
Člen tímu	Zodpovedá za splnenie dohodnutého a prideleného DoD (k úlohe, za ktorú je zodpovedný). Člen tímu musí byť oboznámený s DoD.
Product owner	Kontroluje progress plnenia úloh porovnaním dohodnutého DoD a výstupu od člena tímu, ktorý má na starosti konkrétnu úlohu.

Navrhnuté rozdelenie

Rozdelenie podľa domény úlohy:

1. Analytické úlohy
2. Návrhové úlohy
3. Vývojové úlohy

Rozdelenie DoD podľa úrovní, na ktorých sa kontroluje/plánuje progress plnenia:

1. User story
2. Šprint
3. Release

Rozdelenie podľa kategórie (ktorých oblastí sa týka DoD):

1. Vývoj
2. Testovanie
3. Dokumentácia
4. Manažment
5. Ostatné

Analytické úlohy

- Naštudovaná problematika a spísané kľúčové body analýzy.
- Vypracovaná vyhodnocovacia matica a záver z matice v prípade porovnávania viacerých technológií alebo postupov.
- Uložené príslušné dokumenty na spoločnom OneDrive úložisku v adresári "Analýza" s dodržanými postupmi pre tvorbu dokumentov.

Návrhové úlohy

- Navrhnutý a odsúhlasený (minimálne tými členmi tímu, ktorých sa návrh týka) model riešeného systému alebo podsystému (diagramy a opisy).

- Zdokumentovaný model systému - uložené dokumenty na spoločnom OneDrive úložisku s dodržanými postupmi pre tvorbu dokumentov.

Vývojové úlohy

1. **User Story**
    a. **Vývoj**
        - Vývoj softvérovej súčiastky (súčiastok) bol z hľadiska funkcionality ukončený.
        - Doplnená časť systému je uložená na online repozitári GitHub-u.
        - Doplnený kód/skript bol skontrolovaný ešte aspoň jedným vývojárom (code review).
        - Napísaný kód/skript spĺňa dohodnuté formátovanie.

    b. **Testovanie**
        - Funkcionalita softvérovej súčiastky je pokrytá a otestovaná pomocou jednotkových testov alebo manuálnym testovaním (iba ak jednotkové testy nie je možné zrealizovať).

    c. **Dokumentácia**
        - Napísaný kód/skript je dostatočne zdokumentovaný na porozumenie funkčnosti vyvinutej softvérovej súčiastky (súčiastok).
        - Všetky vykonané GitHub operácie sú dostatočne zdokumentované.

    d. **Manažment**
        - Prípadné chybové situácie a neriešiteľné prekážky sú zdokumentované a prekonzultované s produktovým vlastníkom.
2. **Sprint**
    a. **Vývoj**
        - Vývojové vetvy (branches) sú zlúčené do hlavnej vetvy v GitHub a uložené v online repozitári (ak nejaké existujú).

    b. **Testovanie**
        - Pridaná funkcionalita spĺňa akceptačné kritériá (definované vlastníkom produktu).

    c. **Dokumentácia**
        - Projektová dokumentácia je doplnená o časti, ktoré pokrývajú vyvinuté softvérové súčiastky (vrátane modelov – diagramy a ich opis).
        - Dokumentácia k riadeniu projektu je doplnená (napr. doplnené metodiky).

    d. **Manažment**
        - Vyvinutá časť systému je úspešne skompilovaná a nasadená na serveri.
        - Bola vykonaná retrospektíva šprintu (v rámci neho je možné upraviť DoD).
        - Doplnený web o zápisky zo stretnutí.
3. **Release**
    a. **Vývoj**
        - Nasadenie systému.

    b. **Testovanie**
        - Verzia produktu spĺňa požiadavky zákazníka.

    c. **Dokumentácia**
        - Odovzdaná dokumentácia (riadenie aj projekt).

- Doplnený web o dokumentáciu.

D. Metodika správy zdrojov

Účel dokumentu

Tento dokument popisuje možnosti a miesta uloženia rôznych prevzatých, alebo tímom vytvorených zdrojov.

Dedikácia metodiky

Metodika je určená všetkým členom tímu, ktorý chcú archivovať, alebo zverejniť určitý zdroj ostatným členom tímu.

Tabuľka 7 Opis členov tímu v rámci evidencie úloh

Rola	Úloha
Člen tímu	Analyzuje úlohu, vytvára zdrojový kód a iné rôzne úlohy.
Product owner	Zadáva požiadavky na tím, kontroluje výstupy tímu.

Miesta na uloženie zdrojov

- **GitHub** – miesto určené na všetky zdrojové súbory projektu. Jedná sa hlavne o súbory webportálu a rôznych služieb operačného systému, logovania, centrálnej správy a iné. Vytváranie a pridávanie sa riadi pravidlami popísanými v metodike verziovania.
- **OneDrive** – hlavné úložisko všetkých ostatných súborov, ktoré sa nenachádzajú v Git-e. Jedná sa hlavne o dokumentácie, analýzy a návrhy na riešenie projektu. Taktiež sú tu uložené nahrávky alebo fotky zo stretnutí a podobne. Tento priestor je rozdelený na adresáre, ktoré sú zamerané na určitú špecifickú oblasť, napríklad zápisnice, analýzy (s prislúchajúcimi podadresármi), TP-cup a iné. Členovia tímu môžu vytvárať ďalšie adresáre podľa vlastného uváženia.
- **Slack** – nástroj určený na komunikáciu medzi členmi tímu. Umožňuje aj nahrávanie súborov, ale iba v limitovanej (časovo a priestorovo obmedzenej) podobe. Preto sa nesmie používať ako jediné úložisko dôležitých súborov. Nahratie súboru sa môže použiť, ak sa jedná o obrázok, graf a podobne, ktorý pridá hodnotu k prebiehajúcej diskusii.
- **Emailová schránka** – v prípade správy s dôležitou prílohou je potrebné ju skopírovať a archivovať do úložiska OneDrive.

Pre povolenie prístupu k danej službe je potrebné kontaktovať správcu služby podľa tabuľky nižšie.

Tabuľka 8 Opis komunikačných prostriedkov

Služba	Správca	URL
GitHub	Jaroslav Tóth	https://github.com/jaro0149/FutureMod
OneDrive	Matúš Pohančeník	https://stubask-my.sharepoint.com/personal/xpohancenik_stuba_sk/_layouts/15/onedrive.aspx
Slack	Tomáš Baránek	https://tm17.slack.com/messages/general/
Email	Radoslav Zápach	team17-2016@googlegroups.com

V prípade akýchkoľvek problémov alebo nejasností, je možné obrátiť sa na správcu danej služby, alebo na vedúceho tímu. Presný postup a informácie sú popísané v metodike komunikácie.

E. Metodika komunikačných kanálov

Účel dokumentu

Tento dokument popisuje spôsoby komunikácie členov tímu v súvislosti s riešeným projektom.

Dedikácia metodiky

Metodika je určená všetkým členom tímu.

Tabuľka 9 Opis členov tímu v rámci komunikácie

Rola	Úloha
Správca komunikačných kanálov	Dohliada aby komunikácia bola v príslušnom kanály pre danú tému.
Člen tímu	Komunikuje projekt s iným(i) členom / členmi tímu.

Hlavný komunikačný kanál

Hlavným nástrojom na komunikáciu ohľadom projektu je webová služba Slack. Táto služba podporuje rozdelenie komunikácie do tzv. kanálov.

- Medzi základné kanály patria:
- General – kanál na všeobecnú komunikáciu medzi členmi
- Stretnutia_a_terminy – kanál kde sú uvedené dôležité termíny
- Random – komunikácia na témy netýkajúce sa projektu
- Backlog – kanál kde sa diskutujú nové úlohy, ktoré vznikajú počas behu šprintu

Ďalej sú tu kanály, do ktorých zapisujú externé nástroje ako napr. JIRA alebo GitHub:

- Github – kanál do ktorého zapisuje služba GitHub vykonané commity
- JIRA – kanál do ktorého zapisuje služba JIRA zmeny jednotlivých úloh

Nakoniec sú tu dynamicky vytvárané kanály, ktoré vznikajú (zanikajú) v prípade väčších aktuálne riešených tém:

- TP_CUP – kanál, ktorý vznikol kvôli zapojeniu sa do TP CUPu
- Linux - kanál, v ktorom sa riešil výber operačného systému
- a iné...

Každý člen tímu má právo vytvoriť nový kanál v prípade, ak to uzná za vhodné. Služba Slack rovnako podporuje aj súkromnú komunikáciu medzi používateľmi.

Pravidlá komunikácie

- Každý člen je povinný aspoň 1 krát počas dňa skontrolovať komunikáciu v hlavnom kanály. Počas víkendu toto pravidlo neplatí, iba v prípade vopred dohodnutej výnimky.
- Každý člen je povinný dbať na to, aby komunikácia prebiehala v príslušnom kanály.
- V prípade, že člen adresuje komunikáciu konkrétnemu členovi, nepoužíva na to spoločné kanály ale súkromné (člen – člen) kanály.

Špeciálne prípady

Pre prípad urgentných správ (alebo ak člen neodpovedá na hlavnom komunikačnom kanály) existuje dokument, kde sú spísané súkromné e-mailové adresy a telefónne čísla členov tímu.

Komunikácia s externými subjektami

Na komunikáciu s externými subjektami sa používa mailový kontakt, ku ktorému majú všetci členovia prístup.

e-mail: team17-2016@googlegroups.com

F. Metodika k verziovaniu

Účel dokumentu

Účelom tejto časti dokumentu (metodika na správu verzií) je prezentovanie pravidiel, ktoré musia byť dodržiavané pri verziovaní zdrojového kódu v projekte.

Dedikácia metodiky

Táto metodika je určený pre tých členov tímu, ktorí majú na starosti programový vývoj novej funkcionality v rôznych doménach projektu.

Použitý systém a rozdeľovanie repozitárov

Na správu verzií je používaný Github s viacerými Github repozitármi. Viaceré Github repozitáre sú potrebné z dôvodu vysokej distribuovanosti projektových súborov – rôzne programovacie jazyky a rôzne adresárové štruktúry, v ktorých sa kód nasadzuje. Všetky použité repozitáre musia byť súkromné („private“).

Abstraktná schéma rozvrhnutia Github repozitárov:

1. **Koreňový repozitár** – Obsahuje odkazy na ďalšie repozitáre vo forme submodulov a skripty slúžiace na nasadzovanie projektových súborov na server.
2. **Repozitáre pre submoduly** – Projektový súbory pre jeden pod-projekt v špecifickej doméne.
3. **Repozitáre pre ďalšie vnorené submoduly** – Repozitáre pre submoduly môžu obsahovať aj ďalšie odkazy na vnorené submoduly. Platí pravidlo, že hlavný repozitár obsahuje odkazy na tie repozitáre, na ktorých je jeho funkčnosť závislá.

Pravidlá pre pomenovanie repozitárov a vetiev

Vytvorenie a nastavenie hlavnej verzie

- **Interakcia s procesom:** projektový manažér (inicializácia, opis), manažér verziovania (možnosti verziovania), vývojári (diskusia k navrhnutému modelu verziovania).
- **Označenie:** A.X.X.X, kde A začína od hodnoty 0 a postupne sa inkrementuje pri každej novej hlavnej verzií (unikátna hodnota). Hodnoty X sa pri novej hlavnej verzií vynulujú.
- **Realizácia v repozitároch:** Oddelená vetva s pomenovaním A.0.0.0, kde A je označenie hlavnej verzie. V jednom čase môže byť vyvíjaná iba jedna hlavná verzia. Hlavná verzia musí obsahovať ďalšie podvetvy reprezentujúce vedľajšie verzie.
- **Kedy vytvárať novú hlavnú verziu:**
  - Modifikácia na úrovni architektonického štýlu (úplná zmena architektonického štýlu alebo pridanie nových modulov reprezentujúcich nový alebo doplnený štýl).
  - Pridanie, odstránenie alebo modifikácia celej používateľskej domény

k softvérovému systému.

- Plánované zmeny v kóde, ktoré neumožňujú spätnú kompatibilitu s aktuálnou hlavnou verziou – zmena rozhrania sieťovej komunikácie, zmena rozhraní na komunikáciu s iným softvérom, zmena formátu softvérových nastavení alebo používateľských dát.
- Plánované zmeny logického dátového modelu (schémy v databázovom systéme), do ktorého nie je možné nasadiť dáta z aktuálnej hlavnej verzie (vymazanie alebo modifikácia relácií).

- Zaraďujú sa sem aj beta verzie a prototypy, ktoré môžu a nemusia sa spájať do koreňovej vetvy.
- **Frekvencia:** Ovplyvňovaná požiadavkami zákazníkov alebo potrebami používateľov, riadená projektovým manažérom.

Vytvorenie a nastavenie vedľajšej verzie

- **Interakcia s procesom:** projektový manažér (inicializovanie funkcionálnych zmien), manažér verziovania (vytváranie a spájanie vedľajších verzií).
- **Označenie:** A.B.X.X, kde A je číslo hlavnej verzie a B je číslo vedľajšej verzie začínajúcej od hodnoty 0 s postupnou inkrementáciou. Hodnoty X sa pri novej vedľajšej verzií nulujú.
- **Realizácia v repozitároch:** Oddelená vetva s pomenovaním A.Bi.0.0, kde A je číslo hlavnej verzie a B je číslo vedľajšej verzie. V jednom čase môže byť vyvíjaných aj viac vedľajších verzií („i" – označenie, ktoré začína od písmena „a"). Vedľajšia verzia musí obsahovať ďalšie podvetvy reprezentujúce revízne verzie.
- **Kedy vytvárať novú vedľajšiu verziu:**
    - Automaticky pri vytvorení hlavnej verzie.
    - Vytvorená jedna časť funkcionality (alebo viac súvisiacich funkcionalít), ktorá má hodnotu pre zákazníka alebo používateľa (softvérové súčiastky, ktoré sú funkčné a zaradené do projektu, ale nemajú pre zákazníka / používateľa hodnotu, nie sú zaradené do novej vedľajšej vetvy). Kompatibilita s nadradenou hlavnou verziou.
    - Doplnenie jednej alebo viacerých funkcionalít o vylepšenia.
    - Oprava chýb z predchádzajúcej vedľajšej verzie. Chyby môžu pokrývať aj viacero funkcií projektu.
    - Odstránenie jednej alebo viacerých funkcionalít so zachovaním kompatibility.
- **Frekvencia:** Ovplyvňovaná požiadavkami zákazníkov alebo potrebami používateľov, riadená projektovým manažérom.

Vytvorenie a nastavenie revíznej verzie

- **Interakcia s procesom:** manažér verziovania (vytváranie, spájanie a nastavovanie frekvencie revíznych verzií), vývojári (vytváranie build vetiev v revízií).
- **Označenie:** A.Bi.C.X, kde A je číslo hlavnej verzie, B je číslo vedľajšej verzie a C je číslo revízie inkrementované od začiatočnej hodnoty 0. Hodnoty build verzie (X) sa pri novej revízií nulujú.

- **Realizácia v repozitároch:** Oddelená vetva s pomenovaním A.Bi.C.0, kde A je číslo hlavnej verzie, B je číslo vedľajšej verzie a C je číslo revíznej verzie . V jednom čase môže byť vyvíjaná iba jedna revízia pod jednou vedľajšou verziou. Vedľajšia verzia musí obsahovať ďalšie podvetvy reprezentujúce build verzie.
- **Kedy vytvárať revízne verzie:**
    - Automaticky pri vytvorení vedľajšej verzie.
    - V dôsledku nastavenej frekvencie revízií.
- **Frekvencia:** Každé 2 týždne, ak manažér verziovania nenastaví iné pravidlo.

Vytvorenie a nastavenie build verzie

- **Interakcia s procesom:** vývojári (písanie zdrojového kódu, prenos kódu do „build“ vetiev v centrálnom úložisko, kontrola spájania vetiev do revízií, aktualizovanie build verzií, dokumentácia kódu a „commit“ správ), manažér testovania (dohľad nad tvorbou a spúšťaním jednotkových testov).
- **Označenie:** A.Bi.C.D, kde A, Bi a C označujú hlavnú, vedľajšiu a revíznu verziu. D označuje build verziu pomocou číselného označenia. build verzia už nemôže obsahovať ďalšie odvodené vetvy. Na každej build verzií pracuje práve jeden vývojár. Viacero build verzií sa môže vyvíjať súčasne ale s inými označeniami, ktoré nemusia mať stúpajúci ani inkrementálny charakter. Prvá „build verzia musí byť označená číslom 0.
- **Realizácia v repozitároch:** Oddelená vetva s pomenovaním A.Bi.C.D, kde D je číselné označenie build verzie.
- **Frekvencia:** Frekvencia závisí od času, ktorý je dostupný / priradený na vývoj časti kódu.

Pravidlá pre Github operácie nad build verziami

1. **Commit**
    a. Časť kódu, ktorá sa bude commit-ovať musí byť dostatočne pokrytá dokumentáciou. Dostatočne znamená, že programátor, ktorý nevyvíjal túto časť kódu, porozumie jej funkcionalite, vstupom, výstupom, prípadne výnimkám alebo chybovým výstupom, ktoré môžu nastať.
    b. Pred vykonaním commit-u sa musíme uistiť, že zmenená, pridaná alebo odstránená časť kódu vedie k funkčnej verzií modifikovanej časti kódu (testovaním).
    c. Commit sa vykonáva iba po pridanej celistvej funkcionalite kódu (naprogramovaná celá metóda, funkcia alebo skript - nie len malá časť, ktorá by spôsobila znehodnotenie aktuálnej verzie vývojovej vetvy).
    d. Každá commit operácia musí obsahovať správu s nasledovným formátom:
    [ISSUE_KEY] #time [work_time] #comment [comment_string]
    [ISSUE_KEY] – kódové označenie story alebo subtask z JIRA (napr. FUT-70)
    [comment_string] – text komentáru, stručný opis vykonaných zmien
    [work_time] – dĺžka práce na funkcionalite (formát Ww Dd Hh Mm)
    Prípadne, je možné hashtag „#comment“ nahradiť s „#done“, ak chceme vykonanou zmenou v kóde aj označiť story alebo subtask ako ukončený.

2. **Fetch**
    a. Odporúčané poožívať, ak chceme skontrolovať vytvorené zmeny v online repozitári pred tým, ako sa budú aplikovať nad lokálnou verziou repozitára (aby sa nenarušila funkčnosť vyvíjanej časti projektu).
    b. Po operácií fetch je potrebné manuálne spustiť operáciu merge (niekedy je to potrebné spraviť cez checkout operáciu nad remote vetvou).
3. **Pull**
    a. Používať, iba ak sme si istý, že stiahnuté zmeny z online repozitára nenarušia vývojovú vetvu, na ktorej momentálne pracujeme (inak použiť najprv fetch). Niektoré nástroje však umožňujú vynechať automatické commit-ovanie po vnorenej operácií merge.
    b. Pri pull-och vytvorených nad merge operáciou je požadované, aby vnorené commit správy boli integrované v merge commit-e.
4. **Push**
    a. Pred vykonaním operácie push je vždy potrebné vykonať operáciu pull, aby nevznikali problémy s nekonzistenciou.
    b. Odporúča sa vykonávať operáciu push čo najčastejšie, aby online repozitár bol vždy aktuálny (väčšina vývojových prostredí podporuje vykonanie push a commit súčasne).
    c. Je zakázané používať operáciu „Force push“.
5. **Rebase**
    a. Rebase sa musí vykonávať pravidelne (najmä pri časovo dlhších vývojových vetvách), aby sa prípadné nekonzistencie s nadradenou vetvou mohli riešiť priebežne.
    b. Vždy je potrebné vykonať rebase tesne pred operáciou merge.
6. **Merge**
    a. Operáciu merge je možné vykonať iba v prípade, že code review nie je potrebný, alebo bol vykonaný iným spôsobom ako cez Pull Request. Inak, je potrebné spájať vývojové vetvy pomocou Pull Request.
    b. Pred vykonaním merge operácie je vždy potrebné vykonať rebase vzhľadom na nadradenú vetvu vývoja, aby sme zabezpečili konzistenciu medzi nadradenou vetvou vývoja a odvodenou vetvou vývoja.
    c. Po vykonaní operácie merge sa pripojená vetva nevymazáva.

Pull Request

d. V online repozitári sa používa pull request namiesto priamej merge operácie, aby mohol prebehnúť code review nad navrhovanou prídavnou funkcionalitou.
e. Pred vykonaním pull request je nutné zosynchronizovať lokálny repozitár s online repozitárom. Pull request sa vykonáva v online rozhraní Github.
f. Po code review sa cez online rozhranie vykoná merge (ak sa code review ukončil úspešne). Následne je potrebné spätne cez fetch alebo pull stiahnuť aktuálne verzie vývojových vetiev.

Pravidlá pre Github operácie nad submodulmi

- Adresáre pre submoduly musia byť umiestnené v koreňovom adresári nadradeného repozitára.
- Názov adresára submodulu musí byť zhodný s názvom vzdialeného repozitára.
- Najprv sa vykonáva synchronizácia submodulov, až následne sa vykonáva synchronizácia nadradeného modulu.

G. Metodika dokumentácie

Účel dokumentu

Tento dokument popisuje spôsoby tvorby dokumentácie a uvádza štruktúru a štýly, ktoré bude tím používať. Využije sa pri tvorbe všetkých dokumentov v rámci tímového projektu, ktoré sa budú nachádzať v dokumentácii inžinierskeho diela aj dokumentácii riadenia.

Dedikácia metodiky

Metodika je určená všetkým členom tímu, ktorý sa aktívne zúčastňujú na tvorbe dokumentácie.

Tabuľka 10 Opis členov tímu v rámci tvorby dokumentácie

Rola	Úloha
Správca dokumentácie	Zapracováva dodané dokumenty a dohliada na jej obsah.
Člen tímu	Zabezpečuje tvorbu textov do dokumentácie.

Dôležité informácie pre tvorbu dokumentácie

Formát všetkých dokumentov počas ich editácie je DOCX, ktorý sa pri odovzdaní môže meniť na formát PDF.

V dokumente používame na písanie textu štýl s názvom "Normálny", ktorý definuje potrebné parametre. Rovnako aj pre nadpisy, pre ktoré sú vytvorené až tri úrovne nadpisov, takže pri tvorbe dokumentácie umožňujú jednoduchú synchronizáciu medzi viacerými tvorcami. Označené sú "Nadpis 1", "Nadpis 2", "Nadpis 3". Pre ďalšie menšie nadpisy v texte je možné použiť štýl "Nadpis v texte", ktorý sa bude líšiť od normálneho textu pridaním parametru "bold".

Pri tvorbe tabuliek budeme používať štýl písma "Tabuľka vysvetlivky", pre popisné bunky tabuľky a "Tabuľka obsah", pre bunky, ktoré tvoria jej hodnotný obsah. Popisné bunky budú zároveň vyplnené sivou farbou, ktorá zabezpečí ich jasné odlíšenie od ostatných buniek.

Tabuľka 11 Názorný príklad tabuľky v dokumente

Bunka a pre popis	**Bunka a pre popis**
Bunka pre obsah	Bunka pre obsah

Odrážky v texte majú nasledovný tvar:

- prvý podzáznam
    - druhý podzáznam

Číslovanie v texte má nasledovný tvar:

1. prvý podzáznam
    a. druhý podzáznam

Ďalej je pri tvorbe dokumentácie dôležité zaznamenávať použité skratky v časti "Použité skratky". V prípade citácií a pridávania zdrojov sa vo všetkých formálnych dokumentoch používa norma ISO 690.

H. Konvencie písania zdrojového kódu

Účel dokumentu

Účelom tejto časti dokumentu je definovať konvencie pre tvorbu kódu v použitom programovacom jazyku v našom projekte.

Dedikácia metodiky

Táto metodika je určená pre tých členov tímu, ktorí implementujú novú funkcionalitu do nami navrhovaného systému.

Konvencie tvorby kódu v programovacom jazyku Java

V tejto kapitole sú definované konvencie pre tvorbu kódu v programovacom jazyku Java. Tieto konvencie sú prebraté z dokumentácie o konvenciách kódu pre programovací jazyk Java, ktorú je možné nájsť na oficiálnej stránke spoločnosti Oracle.

Štruktúra zdrojových súborov

Prvotné komentáre

Každý zdrojový súbor na začiatku obsahuje základné informácie o danom súbore.

Príklad:

```
/*
 * Classname
 *
```

```
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Špecifikácia balíkov a importov

Ak zdrojový súbor obsahuje špecifikácie pre balíky a importy, sú uvedené ako prvé nezakomentované riadky zdrojového súboru. Ich poradie je špecifikácia balíkov a následne špecifikácia importov.

Príkad:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Deklarácia triedy a rozhrania

Deklarácia triedy a rozhrania obsahuje nasledujúce časti:

- Dokumentácia k triede/rozhraniu vo forme komentárov
- Definícia a názov triedy/rozhrania
- Komentáre k implementácií triedy/rozhrania(/*...*/), ak sú potrebné
- Deklarácia statických premenných (poradie – public, protected, bez definovania, private)
- Deklarácia ostatných premenných (poradie – public, protected, bez definovania, private)
- Konštruktory
- Metódy

Konvencia k názvom

Balíky

Názvy balíkov v rámci ich špecifikácie začínajú malým písmenom.

Príklad:

```
edu.cmu.cs.bovik.cheese
```

Triedy a rozhrania

Názvy tried a rozhraní začínajú veľkým písmenom. Pri tvorbe názvov je potrebné použiť hlavne podstatné mená a to tak aby bol názov čo najvýstižnejší a čo najstručnejší. Pri viacslovnom názve majú všetky slová v rámci názvu veľké prvé písmeno.

Príklad:

```
class Raster;
class ImageSprite;
```

```
interface RasterDelegate;
```

Metódy a premenné

Názvy metód a premenných začínajú malým písmenom. Pri viacslovnom názve všetky slová okrem prvého začínajú veľkým písmenom.

Príklad:

```
run();
runFast();

char  c;
float myWidth;
```

Konštanty

Názvy konštánt obsahujú iba veľké písmená a podtržník (_) ako oddeľovací znak slov.

Príklad:

```
static final int MIN_WIDTH = 4;
```

Deklarácie a formátovanie

Riadkovanie

Ak sa k deklarácií rovnakých typov premenných neviaže komentár, uvádzajú sa v jednom riadku. V opačnom prípade sa každá deklarácia uvádza na novom riadku.

Príklad:

```
int level; // indentation level
int size;  // size of table

int level, size;
```

2 medzery sú oddeľovačom pre:

- sekcie v rámci zdrojového súboru,
- triedy a rozhrania v rámci zdrojového súboru.

1 medzera je oddeľovačom pre:

- metódy,
- lokálne premenné v metóde a prvý príkaz,
- nový blok,
- jednoriadkový komentár,
- logické sekcie v rámci metódy.

Maximálna dĺžka riadka je 80 znakov. Pri prekročení tejto dĺžky je nutné riadok rozdeliť. Rozdelenie riadka sa prevádza vhodnými metódami podľa daného prípadu, napríklad po úvodzovke alebo pred ďalším argumentom. Vzniknutý ďalší riadok je vhodne odsadený tabulátorom.

Inicializácia a umiestnenie deklarácie

Inicializácia premenných sa vykonáva na mieste ich deklarácie, s výnimkou prípadu, keď je inicializačná hodnota premennej získaná určitým výpočtom.

Umiestnenie deklarácie je na začiatku bloku. Blok je každý kód oddelený zátvorkami {}. Príklad:

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

Okrem toho môže byť deklarácia vykonaná aj v rámci „for“ cyklu.

Príklad:

```
for (int i = 0; i < maxLoops; i++) {
      // code
}
```

Deklarácie v rámci triedy a rozhrania

Medzi názvom metódy a zátvorkou „(“ sa nenachádza medzera. Otváracia zátvorka bloku „{“ sa nachádza na konci prvého riadka deklarácie, pričom od zátvorky „(“ je oddelená medzerou. Uzatváracia zátvorka bloku „}“ sa nachádza na ďalšom riadku po poslednom riadku bloku. Výnimkou je prázdny blok kedy sa táto zátvorka nachádza hneď za otváracou zátvorkou bloku „{“.

Príklad:

```
class Sample extends Object {
    int ivar1, ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

Formátovanie blokov

Každý riadok môže obsahovať iba jeden príkaz. Kód v rámci bloku okrem blokových zátvoriek { a } je odsadený tabulátorom.

Príklad:

```
{
      int x = 1;
      x++;
}
```

Pri podmienkových blokoch, blokoch cyklu a „try-catch“ sa všetky príkazy uvádzajú do zátvoriek, aj v prípade, že je blok tvorený len jedným príkazom.

Príklad:

```
if (x == 0) {
      return false;
} else {
      x == 0;
      return true;
}
```

Medzery pri príkazoch, podmienkach a cykloch

Podmienka v zátvorkách ( ) je z každej strany oddelená medzerami.

Príklad:

```
while (true) {
      // code
}
```

Argumenty v rámci príkazov pre výpis, všetky operátory a ich operandy a príkazy pre inkrement (++) a dekrement (--) sú oddelené medzerami. Pri inkrementovaní a dekrementovaní nie sú operandy oddelené od znakov „++“ a „--“.

Príklad:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
      n++;
}
printSize("size is " + foo + "\n");
```

Argumenty pri cykle, funkcii a pretypovaní sú od seba oddelené medzerou.

Príklad:

```
for (expr1; expr2; expr3) {
      // code
}

myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

Komentáre

Na komentovanie je podľa daného prípadu možné použiť riadkovú metódu (//) alebo blokovú metódu (/* */).

Blokový komentár sa oddeľuje zhora a zdola prázdnymi riadkami od ostatných častí kódu.

Príklad:

```
/*
 * Here is a block comment.
 */
```

Jednoriadkový komentár sa píše nad kód, ku ktorému sa vzťahuje a zhora sa oddeľuje prázdnym riadkom od ostatných častí kódu. Ak je jednoriadkový komentár dlhší ako maximálna dĺžka riadka, uvedie sa vo forme blokového komentára.

Príklad:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

Krátke komentáre sú uvedené v riadku kódu, ku ktorému sa vzťahujú. Odsadenie krátkych komentárov je do jednej línie pomocou tabulátora.

Príklad:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

Dokumentačné komentáre sa uvádzajú pred deklaráciou ku každej triede, rozhraniu a metóde. Ich forma je /**...*/. Používajú sa pri tom preddefinované tagy:

```
@author
@version
@param
@return
@exception
@see
@since
@serial
@deprecated
```

Dokumentačné komentáre vytvára autor triedy/rozhrania/metódy, pričom v ňom uvedie svoje meno, názov tímu a rok formou tagu `@author meno_autora (názov_tímu – rok)`.

Príklad:

```
@author Jožko Mrkvička (Future MOD – 2016)
```

Pri „public“ metóde sa do dokumentačných komentárov uvádzajú aj informácie o jej parametroch formou tagu `@param`, návratových hodnotách formou tagu `@return` a poprípade vyskytujúcich sa výnimkách, ktoré treba odchytiť, formou tagu `@exception`.

Konvencie tvorby kódu v programovacom jazyku C

V tejto kapitole sú definované konvencie pre tvorbu kódu v programovacom jazyku C.

Konvencie k názvom

Vo všetkých názvoch je oddeľovacím znakom slov podtržník (_). Názvy nemôžu podtržníkom začínať ani končiť.

Názvy funkcií sa skladajú len s malých písmen a je potrebné ich vyberať tak, aby čo najviac vystihovali to čo daná funkcia vykonáva. Používajú sa na to hlavne slovesá, napríklad *get, set*. Je vhodné používať sufixy, napríklad *max* – maximálna hodnota, *cnt* – počítadlo, *key* – kľúčová hodnota.

Názvy štruktúr a premenných sa skladajú len s malých písmen.

Príklad:

```
struct  foo {
        struct foo *next;      /* List of active foo */
        int bar;
};

Time    time_of_error;
```

Pri smerníkoch sa znak „*“ dáva hneď na začiatok názvu smerníka.

Príklad:

```
char    *name = NULL;
```

Globálne premenné majú ako začiatok názvu znaky „g_“.

Príklad:

```
Logger  g_log;
```

Názvy pre globálne konštanty sa skladajú len s veľkých písmen.

Príklad:

```
const   int A_GLOBAL_CONSTANT= 5;
```

Názvy pre makrá a #define sa skladajú len s veľkých písmen.

Príklad:

```
#define MAX(a,b) blah
#define MACRO(v, w, x, y)
do {
        v = (x) + (y);
        w = (y) + 2;
} while (0)
```

Názvy „enum“ sa skladajú len s veľkých písmen.

Príklad:

```
enum pin_state_type {
      PIN_OFF,
      PIN_ON
   };
```

Formátovanie

Na jednom riadka môže byť uvedený len 1 príklaz. Ak sa k deklarácií rovnakých typov premenných neviaže komentár, uvádzajú sa v jednom riadku. V opačnom prípade sa každá deklarácia uvádza na novom riadku.

Príklad:

```
char **a, *x;

char **b = 0;  /* add doc */
char  *y = 0;  /* add doc */
```

Pri podmienkových blokoch a blokoch cyklu sa všetky príkazy uvádzajú do zátvoriek, aj v prípade, že je blok tvorený len jedným príkazom. Podmienka v zátvorkách ( ) je z každej strany oddelená medzerami. Otváracia zátvorka bloku „{“ sa nachádza na konci prvého riadka deklarácie. Uzatváracia zátvorka bloku „}“ sa nachádza na ďalšom riadku po poslednom riadku bloku.

Príklad:

```
if (1 == somevalue) {
   somevalue = 2;
}
```

Zátvorka s parametrami funkcie nie je oddelená od názvu funkcie.

Príklad:

```
strcpy(s, s1);
```

Maximálna dĺžka riadka je 78 znakov.

Pri podmienkových blokoch „If Else“ začína else blok až na ďalšom riadku po uzatváracej zátvorke if bloku.

Príklad:

```
if (condition) {
}
else {
}
```

Pri switch bloku sú jednotlivé prípady (case) odsadené od kraja tabulátorom. Príkazy vzťahujúce sa k jednotlivým prípadom sa odsadzujú 1 medzerou. V prípade nutnosti vytvorenia nových premenných sa celý kód uvedie ako blok.

Príklad:

```
switch (...) {
        case 1:
          ...
         break;

        case 2: {
         int v;
         ...
         }
         break;
```

```
        default:
         ...
}
```

Pri podmienkovom príkaze „:?“ sa podmienka uvádza v zátvorkách ( ). Akcie pri tomto príkaze sa uvádzajú ako funkcie.

Príklad:

```
(condition) ? funct1() : func2();
```

Komentáre

Každý zdrojový súbor na začiatku obsahuje základné informácie o danom súbore.

Príklad:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Na komentovanie je podľa daného prípadu možné použiť riadkovú metódu (//) alebo blokovú metódu (/* */).

Blokový komentár sa oddeľuje zhoda a zdola prázdnymi riadkami od ostatných častí kódu.

Príklad:

```
/*
 * Here is a block comment.
 */
```

Jednoriadkový komentár sa píše nad kód, ku ktorému sa vzťahuje a zhora sa oddeľuje prázdnym riadkom od ostatných častí kódu. Ak je jednoriadkový komentár dlhší ako maximálna dĺžka riadka, uvedie sa vo forme blokového komentára.

Príklad:

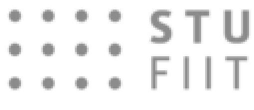
```
if (condition) {

    /* Handle the condition. */
    ...
}
```

Krátke komentáre sú uvedené v riadku kódu, ku ktorému sa vzťahujú. Odsadenie krátkych komentárov je do jednej línie pomocou tabulátora.

Príklad:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
```



```
}
```

Dokumentačné komentáre sa uvádzajú pred deklaráciou ku každej funkcií. Ich forma je /**...*/. Používajú sa pri tom preddefinované tagy:

```
@author
@version
@param
@return
@deprecated
@see
@todo
@bug
```

Dokumentačné komentáre vytvára autor funkcie, pričom v ňom uvedie aspoň svoje meno, názov tímu a rok formou tagu `@author meno_autora (názov_tímu – rok)`.

Príklad:

```
@author Jožko Mrkvička (Future MOD – 2016)
```

Konvencie pri tvorbe bash skriptov

V tejto kapitole sú definované konvencie pre tvorbu bash skriptov.

Štruktúra súborov, formátovanie a komentáre

Skripty začínajú s definovaním použitého shellu v samostatnom riadku.

```
#!/bin/bash
```

Nasleduje hlavička skriptu, ktorú vyplní autor vo forme:

```
#
#Autor: Meno Priezvisko (názov_tímu - rok)
#
```

Každá funkcia má pred deklaráciou v blokovom komentári vypísanú hlavičku funkcie. Otváracia zátvorka bloku { sa uvádza na rovnakom riadku ako názov funkcie a odsadzuje sa 1 medzerou od definovania parametrov v ( ).

```
#######################################
# Cleanup files from the backup dir
# Globals:
#   ...
# Arguments:
#   ...
# Returns:
#   None
#######################################
cleanup() {
  ...
}
```

Na odsadzovanie sa nepoužíva tabulátor ale 2 medzery. Maximálna dĺžka riadka je 80 znakov. Ak sa presiahne je potrebné riadok vhodným spôsobom rozdeliť. Pri dátovodoch sa znak „|" oddeľuje z každej strany medzerou. Ak je pri použití dátovodov presiahnutá maximálna dĺžka riadka, každý dátovod sa uvednie na nový riadok a odsadí od prvého 2 medzerami:

```
# All fits on one line
command1 | command2

# Long commands
Command3
  | command4
  | command5
  | command6
```

Pri blokoch „while", „for" a „if" sa nadväzujúce časti (; do) (; then) uvádzajú v tom istom riadku, pričom ďalšie príkazy sa uvádzajú na nový riadok. Nadväzujúca časť (else) sa uvádza samotná na novom riadku, pričom sa odsadí v jednej vertikálnej línií s otváracím príkazom. Príkazy patriace pod jednotlivé bloky sa odsadzujú 2 medzerami. Uzatvárací príkaz sa uvádza v jednej vertikálnej línií s otváracím príkazom, ku ktorému sa vzťahuje.

Príklad:

```
for dir in ${dirs_to_cleanup}; do
  if [[ -d "${dir}/${ORACLE_SID}" ]]; then
    log_date "Cleaning up old files in ${dir}/${ORACLE_SID}"
    rm "${dir}/${ORACLE_SID}/"*
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  else
    mkdir -p "${dir}/${ORACLE_SID}"
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  fi
done
```

Prípady v bloku case a príkazy vzťahujúce sa k jednotlivým prípadom sa oddeľujú 2 medzerami. Každý prípad sa uvádza samostatne na novom riadku.

Príklad:

```
case "${expression}" in
  a)
    variable="..."
    some_command "${variable}" "${other_expr}" ...
    ;;
  absolute)
    actions="relative"
```

```
    another_command "${actions}" "${other_expr}" ...
    ;;
  *)
    error "Unexpected expression '${expression}'"
    ;;
esac
```

I. Metodika pre code review

Účelom tejto časti dokumentu je definovanie pravidiel a postupov, ktoré je nutné dodržiavať pri vykonávaní code review. Dodržiavanie jednotných pravidiel pre písanie kódu zvyšuje jeho celkovú kvalitu a kód je tak ľahší na pochopenie pre iných programátorov, preto je kontrola týchto pravidiel pri implementovaní dôležitá.

Dedikácia metodiky

Táto metodika je určená vývojárom, ktorí sa v rámci tímového projektu aktívne podieľajú na tvorbe novej alebo úprave už existujúcej implementácie vo forme zdrojového kódu pre nami vyvíjaný softvér.

Roly

Rola	Úloha
Autor zdrojového kódu	Tvorca zdrojového kódu, ktorý vznikol pri tvorbe novej alebo úprave už existujúcej implementácie pre nami vyvíjaný softvér.
Posudzovateľ	osoba, ktorá vykonáva kontrolu zdrojového kódu, ktorý vytvoril „Autor zdrojového kódu“.

Pravidlá code review

Pravidlá code review definujú podmienky a vymedzujú povinnosti pre aktérov jednotlivých rol.

Základné pravidlá

- Posudzovateľ daného zdrojového kódu nemôže byť jeden z jeho autorov.
- Každý vývojár má pri každom projekte priradeného aspoň jedného posudzovateľa, ktorého môže poveriť na vykonanie code review ním vytvoreného zdrojového kódu.
- Autor zdrojového kódu môže poveriť posudzovateľa na code review až po dostatočnom otestovaní daného zdrojového kódu.

Pravidlá pre autora zdrojového kódu

- Je zodpovedný za otestovanie ním vytvoreného zdrojového kódu predtým ako ho predloží posudzovateľovi na code review.

- Ak po vykonaní code review označí posudzovateľ niektoré časti zdrojového kódu za chybné, povinnosťou autora je ich opraviť a zopakovať proces code review.
- Môže pri podávaní žiadosti na vykonanie code review poveriť touto úlohou iba posudzovateľa, ktorý mu bol priradený.

Pravidlá pre posudzovateľa

- Je zodpovedný za skontrolovanie zdrojového kódu, za účelom nájdenia možných chýb v porovnaní s definovanými konvenciami a pravidlami pre písanie zdrojového kódu.
- V prípade nájdenia chýb je povinný definovanou formou poskytnúť spätnú väzbu pre autora zdrojového kódu.
- Nie je zodpovedný za opravu ním nájdených chýb v zdrojovom kóde, ani sa na ich oprave aktívne nepodieľa ale môže v spätnej väzbe navrhnúť ich riešenie.

Procesy

Procesy code review opisujú jeho celkový priebeh a definujú postupy pri jeho vykonávaní.

Priebeh code review

Priebeh code review sa skladá z nasledujúcich procesov:

1. **Vykonanie pull requestu**

   Autor zdrojového kódu pomocou na to určeného nástroja vykoná operáciu push, čím zabezpečí aktuálnosť zdrojového kódu v repozitári. Následne vykoná operáciu pull request a spôsobom špecifickým pre daný nástroj označí za posudzovateľa niekoho z jemu priradených posudzovateľov.

2. **Stiahnutie aktuálnej verzie zdrojového kódu**

   Posudzovateľ poverený autorom na vykonanie code review si spôsobom špecifickým pre dané úložisko stiahne aktuálnu verziu zdrojového kódu, pre ktorý má byť vykonaný code review.

3. **Kontrola zdrojového kódu**

   Posudzovateľ skontroluje zdrojový kód z hľadiska splnenia definovaných konvencií a pravidiel pre písanie zdrojového kódu, kvality zdrojového kódu a bezpečnostných chýb.

4. **Vyhodnotenie**

   Posudzovateľ podľa chybovosti zdrojového kódu vyhodnotí výsledok code review. Ak niektorú časť zdrojového kódu označí za chybnú, pokračuje procesom č. 5. Ak je zdrojový kód bezchybný, autor zdrojového kódu prejde ku kroku č. 7.

5. **Spätná väzba**

   Posudzovateľ poskytne definovanou formou spätnú väzbu autorovi zdrojového kódu. V tejto spätnej väzbe spôsobom špecifickým pre daný nástroj označí chybné časti zdrojového kódu a môže k nim doplniť komentár s návrhom riešenia pre nájdené chyby.

6. **Opravenie nájdených chýb**

   Autor zdrojového kódu opraví posudzovateľom nájdené chyby v aktuálnej verzií zdrojového kódu a začne proces code review od začiatku, čiže od procesu č. 1.

7. **Vykonanie merge**

   Autor zdrojového kódu pomocou na to určeného nástroja vykoná operáciu merge pre aktuálnu verziu posudzovateľom skontrolovaného zdrojového kódu a tým priebeh code review končí.

J. Metodika testovania

Účel dokumentu

Účelom dokumentu je definovať konvencie pri testovaní zdrojového kódu.

Dedikácia metodiky

Táto metodika je určená pre všetkých členov tímu, ktorí sa podieľajú na písaní kódu. Pri písaní kódu novej funkcionality sa vytvorí unit test a otestuje sa nová funkcionalita. Pred integráciou novej funkcionality do aktuálnej verzie je nevyhnutné, aby sa všetky testy vyhodnotili úspešne. Inak kód nemôže byť integrovaný do aktuálnej verzie. Pred integrovaním nového modulu do systému je potrebné napísať a vykonať integračné testy. Takisto ani tu nemôže byť modul integrovaný do systému, ak nezbehli všetky testy úspešne.

Testovanie v Jave

Príprava testovacieho prostredia v IntelliJ

Pre správne fungovanie testov je potrebné vykonať nasledujúce úpravy:

- Stiahnuť: apache maven
- Rozbaliť napr. do C:\Program Files
- Pridať premenu prostredia: používateľská premena > PATH : cesta/apache-maven-3.3.9\bin
- Otvoriť cmd a zadať: maven -v , vypíše informácie o mavene a znamená, že všetko máte správne nastavené a môžete si otvoriť IntelliJ

Vytvorenie projektu

Pri vytváraní nového projektu je potrebné zvoliť maven projekt. Po vytvorení nového projektu máme balík main/java a test/java. Ak chceme pridať novú triedu do balíka main/java a v balíku java neexituje žiaden balík, vytvoríme nový balík a tam vložíme novú triedu. Tiež vytvoríme balík s rovnocenným názvom, kde nakoniec názvu pridáme príponu „Test" napr. „názovbalíkaTEST" v balíku test/java . V tomto balíku vytvoríme balík pre integračné testy „it" a balík pre unit testy „unit". V týchto balíkoch definujeme triedy jednotlivých testov. Pri vytváraní nového testu definujeme názov ako „nazovtriedyTest".

Definovanie vlastnosti v pom.xml

V súbore pom.xml je potrebné dodefinovať nasledujúce veci:

```
<!-- definícia verzie JUnit -->
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>

<!-- definícia pluginu pre integračne testy -->
<build>
    <plugins>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.19.1</version>
            <executions>
                <execution>
                    <id>integration-test</id>
                    <goals>
                        <goal>integration-test</goal>
                    </goals>
                </execution>
                <execution>
                    <id>verify</id>
                    <goals>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>

    </plugins>
</build>
```

Vytvorenie unit testov

Pomocou unit testov testujeme správnu funkčnosť implementovanej funkcionality alebo aj po vykonaní zmien v existujúcej. V prípade, ak nejaký unit test zlyhá a nemáme testovanú funkcionalitu implementovanú je potrebné ju do implementovať a vykonať znova unit testy.

Ak unit test zlyhá po vykonaní nejakých zmien v implementovanej funkcionalite, je potrebné vrátiť sa do bodu, keď unit test prešiel a hľadať príčinu prečo test zlyhal.

Na testovanie používame JUnit framwork verzie 4.12

Nový unit test pomenujeme podľa názvu triedy pre ktorú píšeme unit test ako „nazovTriedyTest“.

Príklad unit testu a reprezentácia rôznych assert metód:

```
import static org.hamcrest.CoreMatchers.allOf;
import static org.hamcrest.CoreMatchers.anyOf;
import static org.hamcrest.CoreMatchers.both;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.everyItem;
import static org.hamcrest.CoreMatchers.hasItems;
import static org.hamcrest.CoreMatchers.not;
import static org.hamcrest.CoreMatchers.sameInstance;
import static org.hamcrest.CoreMatchers.startsWith;
import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNotSame;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;

import java.util.Arrays;

import org.hamcrest.core.CombinableMatcher;
import org.junit.Test;

public class AssertTests {
  @Test
  public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays not same", expected, actual);
  }

  @Test
  public void testAssertEquals() {
    assertEquals("failure - strings are not equal", "text", "text");
  }

  @Test
  public void testAssertFalse() {
    assertFalse("failure - should be false", false);
  }

  @Test
  public void testAssertNotNull() {
    assertNotNull("should not be null", new Object());
  }

  @Test
```

```java
  public void testAssertNotSame() {
    assertNotSame("should not be same Object", new Object(), new Object());
  }

  @Test
  public void testAssertNull() {
    assertNull("should be null", null);
  }

  @Test
  public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    assertSame("should be same", aNumber, aNumber);
  }

  // JUnit Matchers assertThat
  @Test
  public void testAssertThatBothContainsString() {
    assertThat("albumen", both(containsString("a")).and(containsString("b")));
  }

  @Test
  public void testAssertThatHasItems() {
    assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
  }

  @Test
  public void testAssertThatEveryItemContainsString() {
    assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),
everyItem(containsString("n")));
  }

  // Core Hamcrest Matchers with assertThat
  @Test
  public void testAssertThatHamcrestCoreMatchers() {
    assertThat("good", allOf(equalTo("good"), startsWith("good")));
    assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
    assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
    assertThat(7, not(CombinableMatcher.<Integer>
either(equalTo(3)).or(equalTo(4))));
    assertThat(new Object(), not(sameInstance(new Object())));
  }

  @Test
  public void testAssertTrue() {
    assertTrue("failure - should be true", true);
  }
}
```

Integračné testy

Integračné testy slúžia na overenie správnej interakcie jednotlivých modulov systému. Pre kategorizáciu integračných testov je potrebné si zadefinovať rozhranie:

```java
public interface Integration {
}
```

Pri výtvarní integračných testov sa postupuje rovnako ako pri unit testov. Len sa ukladajú do balíka „it“. V danom teste je potrebné pred definovaním triedy uviesť kategóriu testu.

```
@Category(Integration.class)
public class SimpleTest {

        public void test() {
        }
}
```