Univerzita Karlova v Praze

Pedagogická fakulta

Katedra informačních technologií a technické výchovy

BAKALÁŘSKÁ PRÁCE

Metodiky vývoje herní www aplikace

Methodologies of development gaming web application

Lukáš Hodáň

Vedoucí práce:	PhDr. Josef Procházka, Ph.D.
Studijní program:	B7507 Specializace v pedagogice
Studijní obor:	Informační technologie se zaměřením na vzdělání

2016

Prohlašuji, že jsem bakalářskou práci na téma *Metodiky vývoje herní www aplikace* vypracoval pod vedením vedoucího práce samostatně za použití v práci uvedených pramenů a literatury. Dále prohlašuji, že tato práce nebyla využita k získání jiného nebo stejného titulu.

V Praze dne 14. 4. 2016

....

podpis

Rád bych poděkoval vedoucímu bakalářské práce Dr. Josefu Procházkovi za odborné vedení mé práce, vstřícný přístup a za cenné rady, které mi při vypracování bakalářské práce poskytl.

ANOTACE

Práce pojednává o soudobých technikách a metodikách využívaných ve vývoji webových aplikací a přibližuje je čtenáři na příkladu vývoje textové hry. Popisuje agilní metodiky, testování, jazyk UML a návrhové vzory vhodné pro tvorbu webové hry. Součástí je rovněž popis samotného procesu vývoje hry od prvotního nápadu, návrhu uživatelského rozhraní a rozvržení úrovní hry, po implementaci hry s využitím výše zmíněných technik a následným zabezpečením aplikace.

KLÍČOVÁ SLOVA

agilní metodiky, UML, testování, návrhové vzory, vývoj webové hry, JavaScript

ANNOTATION

Thesis deals with contemporary techniques and methodologies used in the development of web applications and describes them to the reader on example of the development of the text game. It describes agile methodologies, testing, UML language and design patterns suitable for creating web games. It also deals with the very process of game development from the initial idea, through the design and layout of the user interface, level design, to programming the game using the aforementioned techniques and securing of application.

KEYWORDS

agile methodologies, UML, testing, design patterns, development of web game, JavaScript

Obsah

Úvod 8

1 Agilní metodiky 9

1.1 Vývoj agilních metodik 10

1.2 Crystal 10

1.3 Refaktoring 11

1.4 Scrum 11

1.5 Extreme programming (XP) 12

1.6 TDD – Test Driven Development 13

1.7 DDT – Design Driven Testing 14

2 Testování 16

2.1 Známé případy selhání 17

2.2 Druhy testů 18

2.3 Testování jednotek 18

2.4 Integrační testování 18

2.5 Systémové testování 19

2.6 Akceptační testování 19

2.7 Regresní testy a automatizované testování 20

3 UML diagramy 22

3.1 Skladba UML jazyka 23

3.2 UML diagramy 23

3.2.1 Use Case diagram 23

3.2.2 Domain model 24

3.2.3 Class diagram 25

3.2.4 Diagram robustnosti 25

4 Návrhové vzory ... 26
4.1 Factory Method ... 27
4.2 Prototype ... 27
4.3 Observer ... 27
5 Návrh a vývoj hry ... 29
5.1 V práci využité metody ... 29
5.2 Textová hra ... 29
5.2.1 Vývoj textových her ... 29
5.3 Zadání hry ... 30
5.4 Příkazy ve hře ... 32
5.5 Žánr hry ... 32
5.6 Návrh herního rozhraní ... 33
5.7 Level Design ... 33
5.8 Průběh hry ... 34
5.8.1 Cesty k vítězství ... 35
5.8.2 Cesty k prohře ... 35
5.9 Agilní metoda ... 35
5.10 Návrh programu podle DDT ... 36
5.10.1 Use Case diagram ... 36
5.10.2 Průběhy případů užití ... 37
5.10.3 Diagram robustnosti ... 38
5.10.4 Scénáře testů ... 39
5.10.5 Návrh aplikace ... 41
5.11 Využití TDD ... 43
5.11.1 Testovací framework - Jasmine ... 43

5.11.2 Testy jednotek ... 44
5.12 Integrační testy ... 45
5.13 Systémové testování ... 46
5.14 Akceptační testy ... 47
5.15 Refaktoring ... 49
5.16 Využití návrhového vzoru Factory Method ... 49
5.17 Zabezpečení ... 50
Závěr ... 52
6 Zdroje ... 54
7 Přílohy ... 57

Úvod

Cílem práce je popsat soudobé techniky vývoje webových aplikací a následně tyto techniky využít při vytvoření jednoduché webové textové hry. Teoretická část práce se zabývá agilními metodikami vývoje, testováním, UML diagramy a návrhovými vzory. V praktické části práce je popsán celý proces vývoje hry s využitím vybraných postupů a metodik.

Agilní metodiky a návrhové vzory bývají považovány za doménu velkých projektů. Cílem praktické části práce je ukázat, že i takovéto techniky se dají využít při vytváření malých aplikací a toto tvrzení předvést na konkrétním příkladu vývoje jednoduché počítačové textové hry.

Textová hra, jako výstupní program praktické části práce, byla vybrána z několika důvodů. Agilní metody vývoje vyžadují jiný styl myšlení, než který je zapotřebí k naprogramování jednoduchých programů. Je nutné přemýšlet více do budoucna, obecněji a k pochopení fungování některých návrhových vzorů je třeba začít přemýšlet mnohem abstraktněji. Tento krok je pro mnohé začínající programátory velmi náročný a ukázky, na kterých se daná problematika obvykle vysvětluje, jim příliš nepomáhají. Složitější techniky vývoje aplikací by měly být vysvětlovány na takových příkladech, u kterých není nutné popisovat jejich funkcionalitu. Za takový ideální příklad lze považovat jednoduchou počítačovou hru. Textová hra byla zvolena i proto, aby kód hry nebyl příliš obsáhlý, ale zároveň se na něm dalo demonstrovat využití všech v této práci popisovaných technik a metod. Dalším důvodem výběru textové hry bylo, že v rámci mé dřívější semestrální práce vznikla podobná hra bez jakéhokoliv předcházejícího návrhu a s použitím pouze manuálního testování. Výsledkem byl kód, který byl velice obtížně rozšířitelný o další funkcionalitu, příběh, místnosti apod.

Vyústěním práce je tak rovněž porovnání přístupu k vývoji aplikace s využitím agilních metod a bez nich. Závěr tohoto porovnání ukazuje míru přínosu i úskalí použití pokročilých technik vývoje softwaru na malých projektech.

1 Agilní metodiky

Agilní metodiky jsou takové metodiky vývoje softwaru, které upřednostňují flexibilní vývoj a očekávají časté změny ze strany zákazníka. Principy agilního vývoje jsou uvedeny v agilním manifestu. Agilní manifest byl sepsán roku 2001 sedmnácti IT odborníky v americkém Utahu. V tomto manifestu jsou uvedeny čtyři hlavní pilíře agilního vývoje a dalších dvanáct z nich odvozených principů.

„Jednotlivci a interakce před procesy a nástroji
Fungující software před vyčerpávající dokumentací
Spolupráce se zákazníkem před vyjednáváním o smlouvě
Reagování na změny před dodržováním plánu

Jakkoliv jsou body napravo hodnotné,
bodů nalevo si ceníme více.“(1)

V manifestu nejsou sepsána striktní pravidla vývoje. Jedná se pouze o doporučení, která vyplývají z praxe. Při vývoji se nelpí na dodržování přesného plánu, nejsou zde omezení na vývojové nástroje a využívají se schopnosti každého jednotlivého člena týmu.

Cílem agilního vývoje je vytvářet software co nejrychleji a bez chyb. Chyby jsou zde myšleny nejen jako nefunkční části programu, ale i špatné interpretace požadavků zákazníka. Aby se podobným chybám předešlo, je program neustále testován a funkční části programu jsou zákazníkovi dodávány průběžně v intervalech týdnů či měsíců. Zákazník pak poskytuje zpětnou vazbu, na základě které je program upraven. Agilně vyvíjený software by měl být flexibilní a snadno upravitelný ve všech fázích vývoje. To znamená, že při agilním vývoji se očekávají časté změny v zadání od zákazníka a program na ně bude tedy připraven a změny budou provedeny rychle a levně.

Základem agilního vývoje je velmi dobrá komunikace mezi členy vývojového týmu. Problémy jsou diskutovány a řešeny dle nejlepšího uvážení celého týmu. Dále je kladen velký důraz na komunikaci týmu se zákazníkem. Všechny nejasnosti v zadání jsou probírány a opravovány postupně. Z těchto dvou principů už vyplývá, že ne každý programátor může být členem agilního týmu, protože musí mít dobré komunikační schopnosti.(2)(3)

1.1 Vývoj agilních metodik

Přesně definovaný pojem agilní metodika se sice objevuje až roku 2001, ale mnoho z do té doby používaných postupů využívalo pravidel z manifestu. Ty postupy, které se v praxi ukázaly být nejpřínosnější, pak byly v manifestu použity. Postupně se objevilo množství různých metodik. Metodiky se od sebe liší podle technik, kterými reagují na změny, způsoby plánování a návrhu programu, způsoby, jakými testují kód. Dále v časových intervalech, ve kterých se dodává zákazníkovi produkt, v intervalech, ve kterých se schází celý tým, aby se zjistilo, co je hotové, a určil se cíl do další schůzky. Liší se i v počtu lidí v týmu, pro které je metoda vhodná.

Metodik je mnoho, z nichž některé jsou specializované na vývoj specifických programů a některé jsou vhodné pro vývoj téměř jakéhokoliv programu. Pro vývoj webových aplikací lze velmi dobře použít univerzálnější metodiky Crystal, Scrum, Extreme Programming, Test Driven Development nebo Design Driven Testing.

1.2 Crystal

Metodiku Crystal začal používat roku 1992 tým Alistera Cockburna, jednoho ze signatářů manifestu. Je to první metodika, která využívá agilní přístupy, jako časté dodávání programu zákazníkovi, rychlé reakce na změny, dobrá komunikace mezi členy vývojového týmu a zavedení času bez rozptylování vývojáře, kdy si každý člen vyhradí dvě hodiny denně ve své pracovní době, během kterých ho nesmí nikdo vyrušovat. Tato metodika je určena pro malé týmy mezi 6–8 vývojáři, ale některé projekty potřebovaly lehce odlišný přístup nebo bylo potřeba zahrnout více lidí. Postupně proto vznikly variace jako Crystal Yellow nebo Crystal Orange, které tyto potřeby řeší.(4)

1.3 Refaktoring

První studii o refaktorování napsal Bill Opdyke roku 1993, a ačkoliv se nejedná o agilní metodiku, je to technika, kterou metodiky nebo i samotní vývojáři často používají. Podstatou refaktorování je restrukturalizace funkčního kódu tak, aby byl čistší, jednodušší, čitelnější a obecnější bez změny jeho funkčnosti. Existuje mnoho technik refaktoringu spočívajících v rozdělení velkých funkcí do menších, zjednodušení podmínek či zlepšení organizace dat s využitím nových tříd.(5)(6)

1.4 Scrum

Scrum patří mezi nejznámější a nejpoužívanější metodiky na světě. Tato metodika byla popsána roku 1995 Jeffem Sutherlandem a Kenen Schwaberem. Původně byla určena pro malé 3–8 členné týmy pracující v jedné místnosti, ale prokázala, že ji je možné použít i ve velkých týmech majících stovky členů pracujících v různých státech. Scrum je metodika projektového řízení. Dá se tedy použít na tvorbu různých projektů, které se nemusí týkat vývoje software. Neříká vývojářům, co mají používat za techniky. Říká pouze, jak bude celý tým při vývoji postupovat.

Tým využívající metodiku Scrum je rozdělen na dvě skupiny. Pigs (prasata) a Chickens (kuřata). Toto rozdělení je podle povídky o praseti a kuřeti.[1] Pigs jsou osoby, které s vývojem projektu přímo souvisejí. Jsou to Product Owner (zákazník), Scrum Master a samotní programátoři. Product Owner určuje priority při vývoji. Scrum Master řídí vývojáře, stará se, aby všem programátorům fungovaly počítače a podobně. Je zakázáno, aby byl Scrum Master programátor kvůli tomu, aby mohl objektivně řešit spory a zajišťovat programátorům klidné prostředí bez rušivých vlivů. Mezi Chickens patří osoby, které se na vývoji programu přímo nepodílí. Jsou to Stakeholders (zainteresované strany) a Manažeři. Stakeholders jsou lidé od zákazníka mající nějaké připomínky. Často to jsou testeři nebo konzultanti zákazníka. Manažeři nastavují prostředí a řídí celý proces vývoje.

Na začátku vývoje se sepíše celkový popis funkčnosti výsledného programu. Následně se vytvoří tzv. User Stories (případy užití programu) a ty se následně zapíšou do Product Backlogu. Poté zákazník určí prioritu jednotlivých případů. Priorita určuje důležitost

[1] http://www.implementingscrum.com/2006/09/11/the-classic-story-of-the-pig-and-chicken/

případu, a tím i určí pořadí, ve kterém budou úkoly řešeny. Následně je každý z úkolů ohodnocen počtem bodů podle očekávané náročnosti. Vývoj programu probíhá ve třicetidenních iteracích, tzv. sprintech. Na začátku každého sprintu se určí, kolik bodů se má za jednu iteraci stihnout, a na konci se předvede funkční část programu zákazníkovi. Pokud se zákazník rozhodne změnit prioritu některého z úkolů nebo změnit zadání, může to udělat pouze mezi jednotlivými sprinty. Změny v průběhu sprintu jsou ve Scrumu zakázány, aby se programátoři mohli plně soustředit na připravené úkoly.

Zvláštností této metodiky je komunikace v týmu. Tak jako v každé agilní metodice je i zde na komunikaci v týmu kladen velký důraz, ale Scrum se zároveň snaží, aby programátoři věnovali maximum času přiděleným úkolům. Proto je veškerá komunikace ze strany Chickens pro programátory vedena přes Scrum Mastera, který pak rozhoduje, jestli nebo kdy novými požadavky programátory zatěžovat. Stejně tak při denních schůzích vývojářů, kde každý programátor řekne Scrum Masterovi, co dělal včera, co bude dělat dnes a jestli narazil na nějaký problém, mohou být pracovníci z řad Chickens přítomni, ale nesmí mluvit. Takto omezená komunikace mezi členy týmu bývá terčem kritiky.(7)

1.5 Extreme programming (XP)

Koncept extrémního programování vznikal od roku 1996, ale ucelenou formu získal až roku 1999, kdy jej přesně definoval Kent Beck v knize Extreme Programming Explained. Tato metodika vznikla výběrem nejpřínosnějších a vynecháním nejméně přínosných technik a postupů z jiných metodik.

Metodika XP zavádí několik pravidel, z nichž některá se zdají být v rozporu s rychlým vývojem. Například pravidlo, že vývojáři pracují ve dvojících. Jeden píše kód a druhý přemýšlí, jestli prováděné změny nebo vylepšení neovlivní negativně architekturu programu. Může se zdát, že když píše kód jen polovina vývojářů, musí být i celý vývoj dvakrát pomalejší. V praxi se ale ukazuje, že je sice psaní kódu pomalejší, ale razantně se snižuje chybovost kódu a snižuje se počet nutných oprav architektury programu. Díky tomu je ve výsledku vývoj programu rychlejší a levnější.

Extrémní programování přináší další změny. Komunikace v týmu není rozdělená do skupin. Každý může komukoliv říct svůj názor. Stejně jako v metodice Scrum je i zde vývoj

naplánován podle User Stories a následně rozdělen do jednotlivých iterací. Iterace jsou ale v délce jednoho až tří týdnů. Obvykle se volí kratší varianta. Dále je zde kladen velký důraz na komunikaci se zákazníkem. Tým musí mít možnost neustále se zákazníkem komunikovat a zjišťovat, co přesně chce. Na konci každé iterace je zákazníkovi předáván otestovaný funkční program a je očekáváno, že zákazník bude poskytovat zpětnou vazbu co možná nejrychleji. Krátkými dodávkami kódu, častou komunikací i zpětnou vazbou je docílen rychlý vývoj s minimem chyb.

Další pravidla určují, jaké techniky mají být využívány při psaní programu. Důležitý je refaktoring. Pravidlo XP říká, že je nutné refaktorovat tak často, jak je to jen možné, aby kód zůstal čistý, čitelný a jednoduchý. Další podstatnou částí této metodiky je psaní testů s využitím techniky testy řízeného vývoje.

I o testech má XP stanovená pravidla:

- testy se píšou před kódem
- všechen kód musí mít testy
- kód před vydáním musí projít všemi testy
- kdykoliv je nalezena chyba, píšou se nejprve testy

Tato metodika je už poměrné známá a často využívaná. Nicméně ne každý zákazník chce, aby byl jeho produkt vyvíjen právě touto metodikou z důvodu velkých až extrémních nároků na zákazníka.(8)

1.6 TDD – Test Driven Development

Testy řízený vývoj je někdy označován za techniku programování a někdy za samotnou metodiku. TDD říká, že návrh programu je určen testy. Aby se toho docílilo, psaní testů předchází psaní kódu. Tak jako ve všech agilních metodikách je i v této nejprve vytvořen určitý návrh architektury programu ještě před zahájením programování. Tohoto návrhu se pak programátoři snaží držet, ale průběžné změny návrhu jsou zde očekávány. Základem TDD jsou jednotkové testy, kterými se podrobněji zabývá kapitola 2.3.

Při použití testy řízeného vývoje si programátor vezme úkol, zkonzultuje jej se zákazníkem a napíše první test, který samozřejmě nevyhoví. Následně napíše kód tak, aby test prošel. Napíše další test a k němu vyhovující kód. Tento proces se opakuje tak dlouho, dokud není

úkol splněn. Posledním krokem je refaktoring. Je vidět, že návrh kódu je skutečně řízen testy, protože výsledný kód dělá přesně a pouze to, na co byl testován.

Výhodou tohoto přístupu je stoprocentní pokrytí kódu testy. Nevýhodou je časté přepisování kódu podle testů a až příliš častý refaktoring návrhu. Dalším negativem je přílišná krátkozrakost této metody. Návrh se často mění podle potřeby aktuálně psaných testů, které ve výsledku nemusejí být použity v důsledku refaktoringu nebo změn zákazníka. Testů v průběhu vývoje vzniká obrovské množství a nevyhnutelně vznikají duplicity. Obvykle také nevzniká žádný komplexní plán testování. Každý programátor nebo tester si testy píše sám podle sebe. Detailní dokumentace testování také není obvykle požadována. To znamená, že všechny jednotky kódu jsou otestovány, avšak nic to nevypovídá o chybovosti nebo připravenosti programu.

Tato technika je však velmi účinná při vytváření malých částí programu. i proto je součástí metodiky XP, protože tam se očekávají pouze malé přírůstky.(9)

1.7 DDT – Design Driven Testing

Metodika návrhem řízeného testování je obrácená metodika k TDD. V TDD se píšou testy bez nějakého většího uvážení, respektive záleží na jednotlivých programátorech, jak postupují. V DDT se naopak nejprve vytvoří plán testování. Analyzuje se požadavek zákazníka, napíšou se případy užití a následně se sepíšou scénáře užití. Vytvoří se tak seznam bodů, ve kterých je zaznamenáno, co by uživatel mohl s danou částí programu dělat a co by program neměl uživateli dovolit. Na základě scénářů jsou pak vytvořeny samotné testy.

Výhodou tohoto postupu je, že kód je nejen dobře otestovaný, ale zároveň je známé, na co všechno je testovaný. Tím, že se nejprve vytvoří scénáře užití programu, se předchází tvorbě duplicitních testů. Zároveň je zamezeno, aby programátor nebo tester při vývoji na některý z testů zapomněl. Což se může stát, pokud si vývojář nezapíše všechny testy, které ho napadly při psaní kódu.

Hlavním nástrojem metodiky návrhem řízeného testování je grafický jazyk UML. Pomocí UML se vytvářejí diagramy, které zobrazují, jak může uživatel program použít, jak funguje program, co obsahují testovací třídy i co má být v které části programu ošetřeno

a otestováno. Diagramy jsou při práci vhodnější než text, protože jsou přehledné a rychlé na pochopení. Díky diagramům je přesně vidět, jak by program měl vypadat ještě dřív, než se začne psát kód. Odpadá tak potřeba častého refaktorování kódu i testů. Diagramy se dále používají jako dokumentace.

Tuto metodiku dobře doplňuje metodika TDD. Programátor před psaním kódu zná nároky kladené na program. To mu usnadní psaní testů podle TDD a tím zajistit funkční kód. DDT může být součástí jiných metodik, jako Scrum nebo Extreme programming, nebo může být využitá samostatně na menších projektech. Její určitou nevýhodou pro někoho může být poměrně dlouhé plánování, které se nicméně ve výsledku vyplácí.(9, s. 57-78)

2 Testování

Definovat, co přesně je a co není testování softwaru, není jednoduché. Definice se průběžně mění podle toho, jaké jsou kladeny požadavky na testování. Záleží i na subjektivním názoru jednotlivých testerů nebo autorů publikací zabývajících se testováním. V počátcích testování (50. – 60. léta) byly pouze vyhledávány defekty v programu. Dnes se výsledky testování používají i k měření kvality programu a k jiným analýzám. Definice z knihy Řízení kvality software: Průvodce testováním uvádí, že testování je „proces řízeného spouštění softwarového produktu s cílem zjistit, zda splňuje specifikované či implicitní potřeby uživatelů."(11, s. 45) Z ní vyplývá, že testování je řízený proces, který je třeba nejprve naplánovat a potom vyhodnotit výsledky. Tímto způsobem se zkoumá produkt a ze získaných údajů se zjišťuje, jestli je již produkt připraven ke spuštění, nebo se z nich odhaduje čas potřebný k dokončení vývoje. Také se mohou získávat údaje o jednotlivých členech týmu. Zjistí se, který tester odhalil který problém a který programátor jej opravil. i údaje, které části programu byly nejproblematičtější, se dají dále využít, například k detailnějšímu plánování podobných funkčností u dalších projektů.(11, s. 43)

Cílem testování je zajištění funkčního a bezchybného programu při dosažení co nejlevnějšího vývoje. Objevení a opravení chyby u právě vyvíjeného kódu je podstatně rychlejší, a proto i levnější, než když se hledá chyba v již hotovém programu.

Testování provádí tester, člen testovacího týmu. Ten zjišťuje, jestli jednotlivé funkce programu nebo části kódu dělají doopravdy to, co dělat mají, a zároveň nedělají, co dělat nemají. Hledá tzv. „bugy", tedy chyby způsobující nežádoucí chování programu. Když takovou chybu najde, nejprve ji musí analyzovat, zjistit příčinu chyby a jasně vymezit hranice, kde se chyba nachází. Po analýze chybu nahlásí a vytvoří hlášení v nějakém společností používaném systému. Hlášení obsahuje popis chyby a postup, jak chybu vyvolat. Někdo z programátorů chybu převezme a nastaví hlášení do stavu řešení, aby ji neřešil ještě někdo jiný. Po opravení změní stav na „vyřešeno". Tato chyba je pak znovu otestována, a jestli test projde správně, je uzavřena.(11, s. 139-155)

K nalezení co možná největšího počtu chyb je potřeba, aby tester přemýšlel velmi kriticky. Musí neustále zpochybňovat každou část programu. Účelem testerů není jen zajistit, aby kód programu fungoval bezchybně, ale aby i program jako celek fungoval vždy tak, jak má.

U složitých programů je potřeba vymyslet všechny způsoby, které by mohly program ovlivnit, a zároveň je nutné vymyslet způsoby otestování zařízení využívající vyvinutý program. Vymýšlení případů, které mohou reálně nastat, a hlavně těch, které by nastat mohly, ale není to příliš pravděpodobné, je to nejtěžší a nejabstraktnější na práci testera. Testeři by měli mít velmi kritický způsob uvažování, které jim umožní vymýšlet i nejnepravděpodobnější scénáře. Jedna věc je ošetření vstupních hodnot u přihlášení do aplikace a druhá věc je zajistit, aby např. meteorologický satelit stále správně vyhodnocoval data, i když je ovlivněný nějakým vesmírným zářením.

2.1 Známé případy selhání

Testování je poměrně drahá a časově náročná součást vývoje software. Nicméně některé případy selhání systému dokazují, jak důležitý tento proces je. Chyby ve webových aplikacích nebo kancelářských programech mohou rozčílit uživatele, ale chyby v programech pro vesmírné projekty, vojenské systémy nebo pro lékařské přístroje mohou ohrozit lidské životy.

Jedno z prvních velkých selhání se stalo roku 1962, kdy softwarová chyba zapříčinila zničení vesmírné rakety Mariner 1 nesoucí první meziplanetární sondu. Raketa se po startu postupně odchylovala od svého kurzu, až ji po 293 sekundách museli vědci z NASA zničit. Chybou bylo neopsání znaku horního podtržítka z ručně psaného vzorce. To mělo za následek špatné vyhodnocování hodnot z letu a následně špatné korelace letu.

Velmi známé je selhání softwaru ozařovacího přístroje Therac 25 v letech 1985-1987. Vinou několika programových defektů zemřelo 6 pacientů a mnoho dalších bylo zraněno, když přístroj pacienty ozařoval desetkrát větší dávkou radiace, než měl. Odhalení tohoto selhání však trvalo velmi dlouho, neboť se nedařilo chyby odhalit z důvodu jejich složité reprodukce. Systém selhával, když byly příkazy zadávány příliš rychle za sebou.

Další případ mohl zapříčinit třetí světovou válku, kdy sovětský satelit roku 1983 detekoval pět raket mířících na Sovětský svaz. Chyba nastala, když satelit špatně interpretoval odrazy slunečních paprsků od mraků jako letící rakety. Sloužící podplukovník Petrov však označil toto hlášení jako falešný poplach s odůvodněním, že by nikdo nezačínal třetí světovou válku pouze s pěti raketami.(10)

2.2 Druhy testů

Aby testy odhalily maximální množství chyb, je potřeba program testovat několika způsoby. Od testování jednotlivých funkcí a tříd, po testování správné posloupnosti volání metod a otestování funkčnosti hotového programu. Tyto testy se dělí do čtyř úrovní.

2.3 Testování jednotek

Testování jednotek je nejzákladnější testování, tedy testování první úrovně. Je to testování na úrovni kódu, kde testy může psát sám programátor. Snaží se otestovat správnou funkčnost nejmenších částí programu, tedy funkcí, metod, procedur a tříd. Podstatou tohoto procesu je otestovat každou jednotku zvlášť, nezávisle na ostatních, protože samotné závislosti mohou vést k chybám. Každý test jednotky by měl testovat pouze jeden případ. Pro každou jednotku je nutné vymyslet vstupy s co největší variabilitou a porovnávat je s očekávanými výstupy. Například funkce, která má sečíst dvě čísla, přijatá jako parametry, a vrátit číselný výsledek, musí být testována několika testy. Test zavolá funkci s parametry, například add(1,2), a očekává vrácenou hodnotu rovnou 3. Dále je potřeba zjistit, jak se funkce zachová, když jeden parametr bude záporný, oba budou rovny nule, jeden nebo oba parametry budou větší než maximální hodnota datového typu Integer, když parametry nebyly vloženy vůbec nebo byly zadány nečíselné hodnoty apod. Napsání takovýchto testů ke každé z jednotek by mělo zajistit její správnou funkčnost v další úrovni testování.(11, s. 61-63)

Jednotkové testy jsou podstatou metodik testy řízeného vývoje a návrhem řízeného testování.

2.4 Integrační testování

Integrační testování je testování druhé úrovně. Tester postupně integruje úspěšně otestované jednotky, neboli postupně přidává volání dalších jednotek. Integrují se na sobě závislé jednotky. Tím vzniká systém. Postupným rozšiřováním systému o další jednotky se dají zjistit chyby, které vznikají za běhu programu. i když je funkce otestována a na přijaté vstupy vrací správné výsledky, není ještě zajištěno, že dostala správné vstupy. Je například možné, že u některého volání funkce jsou zaměněné parametry a systém jako takový pak bude vracet nesprávné výsledky.

Integrační testování dále rozlišuje intrasystémové a intersystémové testování. První způsob označuje testování uvnitř systému, tedy integrování jednotek do systému. Druhé označení testuje integrování více systémů do funkčního celku.(11, s. 63-65)

2.5 Systémové testování

Výsledkem integračního testování by měl být stabilní systém. Ve fázi systémového testování se testuje funkčnost programu, jak byla definovaná zákazníkem. Tyto testy se nazývají funkční testy. Dále se testuje, jak program reaguje na špatné vstupy, testuje se bezpečnost systému a použitelnost programu, tedy jak komfortní práce s programem je. Dále se zjišťuje, zdali program správně komunikuje s jinými systémy. Ověřuje se jeho spolehlivost při déletrvajícím používání a provádějí se výkonnostní testy. To jsou testy, které mají zjistit, jak se program chová při různé zátěži.

Cílem systémového testování je tedy zjistit, jestli je program funkční natolik, aby mohl být předán zákazníkovi. Vždy se očekává, že v programu jsou chyby a kvůli tomu je testování vlastně nekonečný proces. Proto je potřeba zvolit kompromis mezi potencionálními defekty a přesvědčením o funkčnosti programu, a rozhodnout, kdy je program na předání připraven. Pokud se však stane, že všechny testy vždy prochází bez chyb, je pravděpodobné, že je chyba v samotném testování.(11, s. 65-66)

2.6 Akceptační testování

Jestliže dodavatel nabyl dojmu, že je program funkční, a předá ho zákazníkovi, je na zákazníkovi, aby si ověřil, že program doopravdy funguje, před tím než jej uvede na trh. K tomu používá akceptační testy. Mnohé z testů mohou být podobné nebo úplně stejné jako u systémového testování na straně dodavatele. V akceptačním testování se ale navíc do procesu testování zařazují uživatelé ze strany zákazníka, kteří program testují používáním. Používají jej podle připravených testovací plánů, příp. i podle svých znalostí (např. kreslicí program může testovat grafik). Netestují jen funkčnost programu, tedy jestli funguje dobře a je stabilní, ale testují i manuály, pomocnou dokumentaci, zjišťují, jestli popisky dávají smysl a podobně. Následně poskytují zpětnou vazbu o nalezených problémech. Vedoucí tohoto testování pak vyhodnocují závažnost chyb a případně vracejí produkt dodavateli k opravě.

S problematikou akceptačního testování souvisí pojmy alfa a beta testování. Oba mají společné, že produkt testují uživatelé používáním. Alfa testování obvykle probíhá ještě u dodavatele. Podle výsledků alfa testování se dodavatel může rozhodnout o připravenosti programu. Beta testování probíhá na straně zákazníka. U některých aplikací, zejména her, probíhá beta testování ve dvou krocích. První krok je uzavřené beta testování. Hráči jsou pozváni přímo do místa společnosti, kde si zahrají nejnovější hru, a pak poskytnou zpětnou vazbu. Když toto testování proběhne úspěšně bez větších chyb, spouští se otevřené beta testování. Vlastník hru zpřístupní hráčům z celého světa na svém serveru na krátkou dobu, například na jeden víkend. Během tohoto víkendu zjistí, jak stabilní je hra i jak výkonné servery by měl v budoucnu používat. Samozřejmě od hráčů očekává opět zpětnou vazbu, pokud narazí na chybu.(11, s. 67)

2.7 Regresní testy a automatizované testování

Smyslem všech předchozích způsobů testování je nalézt defekty. Jestliže je defekt opraven, může tím vzniknout nový defekt, který nezjistí opakované otestování původní chyby. Může se stát, že k opravení chyby bylo potřeba udělat více úprav i v jiných funkcích, a tím mohlo dojít k vytvoření nové chyby na jiném místě. Aby se zajistila správná funkčnost programu i po opravení chyby, využívá se regresní testování.

Regresními testy mohou být i všechny doposud napsané testy. Smyslem regresního testování je opakované spuštění všech testů, nebo vybrané série testů, které mají ověřit, že i po modifikování programu nebo odstranění chyby funguje systém stále dobře. Tyto testy by měly být spouštěny buď po každé změně, což v případě velkého množství testů není úplně praktické, nebo v pravidelných časových intervalech. K tomu slouží automatizované testování.(11, s. 68-69)

Automatizované testování v jeho nejjednodušší formě slouží k tomu, že se vybraný balík testů pravidelně spouští a vyhodnocuje podle nastaveného časového intervalu. Takovéto automatizované testování už umějí i jednoduché, volně přístupné testovací frameworky. Nicméně využití automatizovaného testování je rozsáhlejší. Například lze využívat generování velkého množství vstupních i výstupních hodnot do externí tabulky a následně automaticky číst tyto hodnoty a testovat vybrané funkce podle těchto dat. Tato technika je označována jako Data Driven Testing, neboli daty řízené testování. Automatizaci lze využít

i při simulování zátěže tisíců uživatelů, nahrání a následném opakování aktivity uživatelů nebo při automatickém generování skriptů, testovacích tříd a testovacích metod. Jak je vidět, automatizace se dá využívat poměrně rozmanitě a ne vždy musí být použita v souvislosti s testováním. Avšak pro využívání takových nástrojů je třeba používat sofistikovanější programy, jako jsou Selenium, Watir, Silk Test, Visual Studio Test Professional nebo Apache JMeter.(11, s. 175-186)

3 UML diagramy

Prvním krokem vývoje software je vytvoření návrhu programu a až po jeho dokončení se začne psát kód. Návrh spočívá v promyšlení všech zákoutí programu a zapsání všech potřebných údajů. Je třeba zaznamenat, co vše má program dělat, a poté dle těchto požadavků navrhnout architekturu programu. Požadavky i navržená architektura mohou být zapsány běžným jazykem nebo lze použít grafický záznam ve formě diagramů. Grafické zpracování se používá z důvodu lepší přehlednosti, možnosti zaznamenat vazby mezi jednotlivými částmi diagramu a usnadňuje komunikaci mezi členy vývojového týmu. Od jednoduchých náčrtků bez nějakých pravidel se postupně začaly používat přesně stanovené postupy, které se dále zformovaly v grafické modelovací jazyky.

Od začátku osmdesátých let se objevovalo množství různých návrhových metodik a grafických jazyků, které měly usnadnit návrh programů. Každá nová metodika měla řešit nedostatky předchozí nebo přinést nové možnosti při navrhování programů či jiných projektů. Roku 1990 už existovalo více než 50 takových metodik a grafických jazyků. Používaly se metodiky jako Booch, Object Modeling Techniques (OMT), OOSE, Fusion a další. Jenže množství různých metodik mělo za následek, že různé programátorské týmy využívaly různé metodiky a ve chvíli, kdy jeden tým předával svůj návrh jinému týmu, museli se členové druhého týmu nejprve naučit nový grafický jazyk. Problém byl, že žádná z těchto metodik nebyla standardizována. Tento problém vyřešil roku 1997 právě jazyk UML. Cílem standardizování bylo sjednocení terminologie a grafického zápisu.

UML (Unified Modeling Language) znamená unifikovaný modelovací jazyk. Jeho vývoj začal už v roce 1990 a postupně přejímal syntaxi jiných jazyků, a to hlavně z metodiky OMT. UML verze 1 byla vytvořena roku 1996 a v roce 1997 byla standardizována. Od té doby prakticky ustal vývoj i používání jiných grafických jazyků a pro návrh programů začal být využíván výhradně jazyk UML. V současné době se využívá nejnovější UML verze 2.5. Také se objevují nové variace jazyka UML, například System Modeling Language (SysML) a Business Process Model and Notation (BPMN), které zahrnují nové techniky vhodné pro vývoj určitých systémů nebo jiných projektů.(12)(13)(17)

3.1 Skladba UML jazyka

UML je programovací jazyk, přestože nevyužívá stejnou syntaxi jako jiné programovací jazyky. Má však přesně danou syntaxi a sémantiku, která dává význam syntaktickým výrazům. Patří mezi programovací jazyky, protože na základě UML diagramu lze vygenerovat kód do některých klasických programovacích jazyků (Java, C#). Technika využití diagramu pro vygenerování kódu se nazývá Model Driven Development.

Jazyk UML využívá grafických prvků, tzv. předmětů (kružnice, elipsa, zaoblený obdélník a jiné) k zobrazení určitých programátorských prvků. Například malá kružnice značí rozhraní, obdélník třídu a podobně. Mezi jednotlivými předměty jsou relace, tedy vztahy. Vztahy jsou zakresleny jako čáry nebo šipky mezi dvěma předměty a určují, jestli je jeden předmět na druhém závislý, nebo jeden rozšiřuje druhý. Dále UML nabízí symboly pro balíčky, do kterých můžeme vložit třídy, aby se zpřehlednil diagram. Pro přehlednost je možné používat poznámky nebo popisky nad vztahy.(15)(16)

3.2 UML diagramy

Součástí standardu UML je 14 typů diagramů, ale existují i další, které tyto základní typy rozšiřují. Každý z diagramů odráží určitou část vývoje tak, aby pomocí všech byl pokryt celý vývojový proces. Každý z nich používá jiné prvky jazyka a každý se navrhuje jiným postupem. Diagramy se dělí do dvou skupin, diagramy struktury a diagramy chování. Strukturální diagramy (např. Class diagram, Object diagram, Domain diagram) popisují strukturu programu, tj. složení programu ze tříd, balíčků a jiných prvků. Diagramy chování (např. Use Case diagram, Activity diagram, Sequence diagram) popisují, co program dělá, nebo co má dělat. Nejčastěji používanými diagramy jsou Use Case diagram a Class diagram.(17)

3.2.1 Use Case diagram

Use Case diagram, neboli diagram případů užití, zobrazuje chování programu z podhledu uživatele. V prvním kroku při vytváření diagramu je zvolen aktér (uživatel, administrátor apod.) Dále je položena otázka, jaké operace může aktér s programem vykonat. Každá odpověď bude jedním případem užití. Například u webové aplikace se uživatel může přihlásit, vkládat komentáře, mazat vlastní komentáře.

Tento diagram je obvykle první, který se při návrhu programu vytváří. Pomáhá návrhářům jasně vymezit funkcionalitu programu a obsáhnout funkční požadavky zákazníka. Při vytváření tohoto diagramu platí určitá pravidla. Jedním z nich je, že případy užití mají být popisovány pouze bodově, nikoli dlouhými větami. O tom, jak správně vytvářet tyto diagramy byla roku 2003 vydaná kniha Use Case Modeling autorů Kurta Bittnera a Iana Spence.(17)

Součástí diagramu je i Use Case Specification. Jedná se o dokument, který popisuje samotný diagram. Obsahuje krátký popis diagramu, seznam aktérů, podmínky pro spuštění, základní a alternativní toky programu a podmínky pro dokončení. Popis diagramu uvádí, čeho se diagram týká. Podmínky pro spuštění říkají, co musí být splněno, aby mohl program pracovat - například uživatel musí být přihlášený. Základní tok je scénář, který popisuje interakci mezi aktérem a jeho případy užití při správném použití programu. Alternativní toky jsou scénáře, kde dochází k různým odchylkám od základního toku. Například v alternativním toku se mohou měnit podmínky pro uživatele, kteří jsou přihlášeni. Podmínky dokončení říkají, kdy končí funkcionalita programu, kterou popisuje diagram. Může končit například ve chvíli, když se správně uloží data do databáze.

Další částí tohoto diagramu je Use Case Description. Jedná se o krátký slovní popis všech případů užití.(17)

3.2.2 Domain model

Doménový model je konceptuální model Class diagramu. Jedná se o zjednodušený Class diagram a často se uvádí jako jeho součást. Vytvoření tohoto modelu následuje po dokončení Use Case diagramů. V tomto modelu se již navrhuje architektura programu. Zobrazuje třídy, respektive entity, a vztahy mezi nimi. Někdy jsou ve třídách zaznamenány i jejich atributy, ale to není podmínkou. Doménový model pomáhá návrhářům uvědomit si, co za třídy budou v programu potřebovat a jaké mezi nimi budou závislosti. Jedná se pouze o určitý náčrt, podle nějž implementace kódu nemusí fungovat.(17)

3.2.3 Class diagram

Class diagram je rozšířený doménový model. Může být samozřejmě vytvořen i bez tohoto modelu. Class diagram představuje pohled na funkční systém. Předmětem tohoto diagramu jsou třídy, které musejí obsahovat všechny atributy včetně datových typů i jednotlivé metody včetně návratových hodnot. Zároveň se určuje, jestli jsou atributy a metody privátní, nebo veřejné. Diagram je řešen jako platformově závislý a musí být funkční. Tento diagram je základem návrhu programu.(17)

3.2.4 Diagram robustnosti

Diagram robustnosti nepatří mezi základní diagramy UML a ani není používán při návrhu programu. Tento diagram slouží k vytvoření testovacích případů na základě případu užití z Use Case diagramu. Tohoto diagramu využívá hlavně metodika návrhem řízeného testování. Diagram robustnosti zobrazuje průběh jednoho případu užití. Například při případu užití „Přihlášení“ budou v diagramu zaznamenány průběhy programu. Bude vidět, že když uživatel klikne v přihlašovací obrazovce na tlačítko „Zrušit“, zavře se stránka. Když klikne na tlačítko „Přihlásit“, tak se nejprve ověří syntaxe hesla. Pokud je heslo ve správném tvaru, vyhledá se uživatel v databázi. Poté se ověří heslo uživatele. Dále tu bude zaznamenáno, co se stane, pokud některý z těchto bodů neprojde.

Diagram robustnosti řeší funkčnost jednotlivých částí programu z pohledu toho, jak by program mohl použít uživatel. Ve spojení s tímto diagramem se objevuje pojem kontrola rozumnosti. Pokud je na základě případu užití příliš složité vytvořit diagram robustnosti, znamená to, že bude i velmi obtížné takový případ uchopit a napsat pro něj kód. Případ, který nejde příliš dobře naprogramovat, je tedy nerozumný a měl by být přehodnocen, rozdělen do více případů a definován jinak v diagramu případů užití. Díky tomuto diagramu jsou vidět části programu, které je nutné testovat. Některé nástroje mohou z diagramu robustnosti přímo automaticky generovat testovací třídy a testovací metody.(9, s. 62-65)

4 Návrhové vzory

Při navrhování programu jazykem UML nebo při samotném programování programátoři často narážejí na stále stejné problémy, jako je časté implementování stejných funkcionalit napříč programy (např. funkcí Zpět a Dopředu, zobrazení historie apod.) nebo rozsáhlý kód, který je nepřehledný a může v něm docházet k přepisování funkcí a globálních proměnných. Pro takovéto problémy byla postupně vytvořena a zdokumentována praxí ověřená obecná řešení. Tato řešení se nazývají návrhové vzory. Za popsáním prvních vzorů stojí čtyři programátoři, kteří vytvořili skupinu Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides. Ti v roce 1991 uvedli prvních několik návrhových vzorů jako Composite, Decider a Observer a později sepsali knihu Design Patterns: Elements of Reusable Object-Oriented Software.(14)

Návrhové vzory jsou tedy obecná řešení často se vyskytujících netriviálních problémů. Obecná řešení se dělí na implicitní a explicitní. Implicitní jsou řešení častých, ale triviálních problémů. Nejsou nikde zdokumentována a jejich řešení vychází ze zkušenosti programátora. Explicitní jsou právě zdokumentované návrhové vzory. Za návrhový vzor může být považováno jakékoliv řešení, které bylo v praxi výhodně použito alespoň třikrát. Základních vzorů je 33 a řeší různé problémy jako chování programu, jeho strukturu nebo problémy spjaté s tvorbou objektů. Proto se vzory dělí do tří základních skupin: Behavioral patterns, Structural patterns a Creational patterns. Existují i další skupiny, ve kterých jsou vzory pro více vláknové programování nebo vzory pro různé platformy.

Znalost návrhových vzorů je zásadní při navrhování velkých programů. Využití vzorů je zaneseno i do vývojových diagramů. Předchází se díky nim problémům, které by mohly vzniknout při implementování kódu. (18)

Ve webové textové hře se bude určitě řešit častá tvorba objektů, případně tvorba objektů s podobnými vlastnostmi a vzájemná komunikace mezi objekty. Proto jsou pro vývoj takové hry vhodné návrhové vzory Factory Method, Prototype a Observer.

4.1 Factory Method

Factory Method (tovární metoda) je často využívaný vzor ze skupiny Creational patterns. Používá se ve chvíli, kdy je potřeba vytvořit více objektů, které jsou příliš komplexní, nebo více instancí jednoho objektu se stejnými nebo s různými vlastnostmi. Hodí se i ve chvíli, když je potřeba vytvořit velký počet malých objektů. Tento vzor redukuje množství kódu potřebného k vytvoření objektů. Činí kód jednodušší a čitelnější.(19)

4.2 Prototype

Prototype je další vzor ze skupiny Creational patterns. Používá se při vytváření nových objektů, které jsou stejné, nebo jsou podobné jinému objektu. Pomocí vzoru Prototype se vytvářejí kopie objektů bez nutnosti jejich inicializace, kterým pak lze dodatečně změnit některé vlastnosti. Tento vzor lze použít ve chvíli, kdy vytvoření nových složitých objektů trvá příliš dlouho. Díky vzoru Prototype je potřeba takový objekt vytvořit pouze jednou, a pak z něj lze dělat kopie i za běhu programu, což je ve výsledku podstatně rychlejší. Stejný přínos bude mít i při vytváření velkého množství stejných objektů. Tento návrhový vzor se často doplňuje se vzory Factory Method nebo Abstract Factory Method.(20, s. 99-110)

4.3 Observer

Návrhový vzor Observer je ze skupiny Behavioral patterns. Jeho účelem je oddělit od sebe funkce, které nemusejí být pohromadě. Je používán ve chvíli, kdy se na základě nějaké události jedné funkce má zavolat jiná funkce při zachování oddělených implementací obou funkcí. Toho je docíleno tzv. pozorovateli. Každá událost, na kterou je třeba reagovat specifickým způsobem, má zaregistrovaného jednoho pozorovatele. Všichni pozorovatelé jsou při zaregistrování přidáni do seznamu pozorovatelů. Když událost nastane, pozorovatel ji oznámí všem ostatním pozorovatelům v seznamu. Pozorovatel, pro kterého je oznámení určeno, pak spustí druhou funkci. Velkou výhodou je, že díky tomuto vzoru je možné měnit funkčnost programu za běhu, a to díky možnosti odstranění pozorovatele u pozorovaného předmětu a přiřazením nového pozorovatele. Funkci tohoto vzoru si lze přiblížit na příkladu počítačové hry, ve které je hráč za splnění nějakých podmínek odměňován nějakými nálepkami nebo bonusy. Podmínkami ve hře mohou být například uběhnutí tisíce kilometrů, sesbírání sta věcí apod. Když hráč sebere stou věc, přiřazený pozorovatel tuto událost oznámí

pozorovateli, který má na starost přiřazení nového úspěchu hráči. Pozorovatel dosaženého úspěchu je pak odstraněn ze seznamu pozorovatelů. Nevýhodou tohoto vzoru může být poměrně velké zpomalení programu, pokud je používáno příliš velké množství pozorovatelů, kteří reagují na velmi časté změny.(20, s. 197-210)

5 Návrh a vývoj hry

V následujících kapitolách se budu zabývat navržením a následným naprogramováním jednoduché webové textové hry. Hru, jako program, na kterém budou ukázány pokročilé techniky vývoje software, jsem vybral ze dvou důvodů. Prvním důvodem je, že na příkladu hry se dá lehce a jasně vysvětlit použití různých technik bez nutnosti složitého vysvětlování, co má program vykonávat. Druhým důvodem je, že textová hra představuje program, který se dá vytvořit téměř bez jakékoliv přípravy s minimálními znalostmi o programování, nebo jej lze vyvíjet s využitím pokročilých programovacích technik - agilních metodik, různých návrhových vzorů i různými technikami testování.

5.1 V práci využité metody

Textová hra vytvořená v rámci této práce bude napsána v jazyce JavaScript. Bude tedy spouštěna ve webovém prohlížeči na straně klienta. Hra bude vyvíjena s využitím agilních metodik DDT a TDD. V programu budou zakomponovány návrhové vzory Observer a Factory Method. Hra projde všemi úrovněmi testování.

5.2 Textová hra

Textová hra je počítačová hra, která k předávání informací o dění nevyužívá grafické prvky, ale pouze textové. Hra předloží popis lokace a dění, hráč si jej přečte, zareaguje písemným příkazem zadaným z klávesnice, hra na příkaz zareaguje a předloží další popis děje.(21)

5.2.1 Vývoj textových her

Vývoj počítačových her začal roku 1952 hrou OXO, což byla jednoduchá verze piškvorek odehrávající se na ploše 3x3. První hry využívající plně textového rozhraní se objevily až roku 1969. Jedná se o hry Hamurabi, Lunar Lander a Space Travel. Hamurabi je strategická hra, kde si hráč v roli panovníka Chammurapiho spravuje svou zemi, zakládá pole, pěstuje plodiny a určuje, kolik plodin je určeno pro další výsev a kolik jako jídlo pro lidi. Hráč píše pouze čísla určující například, kolik akrů pole si má koupit. Nepoužívá žádné složité příkazy. Podobně jednoduché ovládání mají i obě výše zmíněné hry. (22)

První textová hra s příběhem, tedy textová adventura, se objevila roku 1975. Jedná se o hru The Colossal Cave Adventure. Tato adventura zprostředkovává interaktivní příběh, který si hráč může prožít a ovlivnit svým jednáním. Hráč využívá slovní příkazy, jako get, go, open

a jiné, následované podstatným jménem, jako get egg, go north nebo open door. Na základě tohoto principu pak vznikaly další podobné hry - série Zork a Planetfall.

Roku 1983 byla vydána hra Adventureland, která textový režim rozšiřovala o grafické zobrazení lokalit, ve kterých se hráč nacházel. Jednalo se pouze o statické osmibitové obrázky lesa, jeskyně apod.

Hned následující rok však vyšla hra King's Quest, která už nevypisovala popis prostředí textovou formou, ale celé prostředí vykreslovala ve formě obrázku. Hráč svými textovými příkazy ovládá postavičku, která se již pohybuje po obrazovce a interaguje s prostředím.(23)

V následujících letech se rozvíjely především hry s grafickým rozhraním a využitím myši. i v současné době se občas objeví komerční hry ovládané textovými příkazy - hra Façade (2005), která využívá pokročilý jazykový parser a dokáže odlišně reagovat na různé příkazy, a tím dokonce měnit průběh hry, nebo počin skupiny Cyborg Reality Commandverse (2015), kde hráč pomocí textových příkazů vytváří celý vesmír.

5.3 Zadání hry

Prvním krokem vývoje jakékoliv aplikace je sepsání jejího zadání, ve kterém je uvedeno, o jaký typ aplikace se jedná a co má ve výsledku dělat. V případě počítačové hry je tímto zadáním popis hlavního příběhu doplněn o popisy ovládání hry a problémů, které je potřeba vyřešit před začátkem vývoje.

Příběh:

Hráč začíná hru v hangáru vesmírné stanice. Jakožto specialista na vesmírné technologie byl na ni vyslán ze Země po té, co stanice záhadně přerušila spojení. Hráč zjišťuje, že na stanici neběží podpora života a musí se po ní pohybovat ve skafandru, který má jen omezené množství kyslíku. Cílem hry je znovunavázání spojení stanice se Zemí.

Ovládání hry:

Jedná se o textovou hru, tudíž ji hráč ovládá pomocí slovních příkazů „jdi“, „seber“, „polož“ atd. Hra s hráčem komunikuje pouze pomocí textového rozhraní.

Popis základních příkazů:

- „jdi“ – umožní hráči pohybovat se mezi jednotlivými místnostmi na stanici
- „coje“ – slouží ke zjištění, jaké věci se nacházejí v místnosti nebo v batohu
- „seber“ – umožní hráči sbírat věci v místnosti a vkládat je do batohu
- „poloz“ – umožní hráči pokládat věci z batohu do místnosti
- „pouzij“ – umožní hráči používat některé věci
- „pomoc“ – vypíše informace o hře a nápovědu ke hraní
- „konec“ – ukončí hru

Problémy:

- možnost zjistit, ve které místnosti se hráč zrovna nachází
- seznam věcí místnosti a v batohu bude zobrazen na obrazovce permanentně, nebo si je hráč vypíše příkazem
- omezení velikosti batohu počtem nebo váhou věcí nebo bude neomezen
- budou všechny věci sebratelné a použitelné?
- odčítání vzduchu ve skafandru po použití některých příkazů a doplnění vzduchu
- doplnění vzduchu do skafandru
- navržení cesty k vítězství s omezením vzduchu
- zprovoznění podpory života

Stanovené podmínky:

- každá věc má svoji váhu, přičemž nejmenší možná váha je rovna 1
- batoh je omezen na váhu 5 - do batohu vejde maximálně 5 věcí
- jen některé věci jsou sebratelné a použitelné
- věci v místnosti a v batohu jsou vypsány na základě příkazu „coje“
- východy z místnosti se vypíší při vstupu do místnosti
- stav vzduchu skafandru je zobrazen na obrazovce neustále, protože je to důležitá informace, díky které mohou hráči plánovat další postup
- všechny informace (cíl hry, aktuální místnost atd.) hráč zjistí příkazem „pomoc“

5.4 Příkazy ve hře

Je vhodné předem určit, které příkazy má mít hráč k dispozici a sepsat je všechny do tabulky včetně počtu jejich parametrů a popisu.

příkaz	**počet parametrů**	**parametr**	**popis**
jdi	1	místnost	pohyb mezi místnostmi
seber	1	věc	přidáni věci do batohu odebrání věci z místnosti
poloz	1	věc	odebrání věci z batohu přidání věci do místnosti
pouzij	1	věc	použití věci
pouzij	2	věc věc	použití jedné věci na jinou věc
coje	1	tady batoh	co je v místnosti co je v batohu
mluv	1	postava	promluvit si s postavou
vloz_kod	1	kód	vložení kódu do terminálu
konec	0		ukončení hry na přání uživatele
pomoc	0		souhrnné informace o hře
pomoc	1	věc příkazy	informace o věci nápověda k příkazům

Tabulka 1 - Příkazy ve hře

5.5 Žánr hry

Do hry budou zařazeny jednoduché logické úkoly a hádanky, které budou hráče nutit trochu přemýšlet a plánovat svůj postup, což bude nezbytné pro úspěšné dokončení hry. Hra nebude pouhým souborem logických mini her, ale bude hráči umožňovat stanici prozkoumávat, interagovat s věcmi a odhalovat skrytý příběh. Hra tedy bude logická adventura, respektive logická dobrodružná textová hra. Aby hráč musel alespoň trochu plánovat svůj postup, je do hry zařazen prvek skafandru, který má omezenou zásobu vzduchu. Hráč bude muset zprovoznit podporu života. To udělá zapnutím generátoru, který se však nachází za zablokovanými dveřmi, které se nedají otevřít. Musí tedy vymyslet, jak se do místnosti dostat. A na závěr bude muset získat kód pro navázání spojení se Zemí, který je ale zašifrovaný, a tudíž jej musí nejprve dešifrovat.

5.6 Návrh herního rozhraní

Textová hra nevyžaduje žádné složitě propracované uživatelské rozhraní, které by se museli hráči dlouho učit používat. Avšak při navrhování jakéhokoliv uživatelského rozhraní platí určitá, léty ověřená, pravidla. Rozhraní by mělo být jednoduché na pochopení i na použití. To znamená nevytvářet v okně desítky tlačítek různých tvarů, u kterých není na první pohled jasné, co vlastně dělají. Měly by být zobrazeny jen nejnutnější informace, aby hráč nebyl rozptylován nadbytečnými údaji, které nutně nepotřebuje pro další postup ve hře. Je vhodnější umožnit hráči zobrazit si, co sám chce a kdy chce. Na druhou stranu je třeba mít zobrazené ty informace, které bude hráč vyžadovat velmi často, aby nemusel každou chvilku otevírat a zavírat nějaké okno jen proto, aby zjistil, kolik vzduchu mu zbývá nebo kolik má životů. (24)

Rozhraní pro textovou hru podle výše uvedeného zadání bude velmi jednoduché. Je potřeba jedno okno, ve kterém budou zapsány hráčovy příkazy a odpovědi hry formou logu. Vedle tohoto okna bude panel se zobrazením stavu vzduchu ve skafandru a případně jiné stavové informace, které by v budoucnu mohly být do hry přidány.

5.7 Level Design

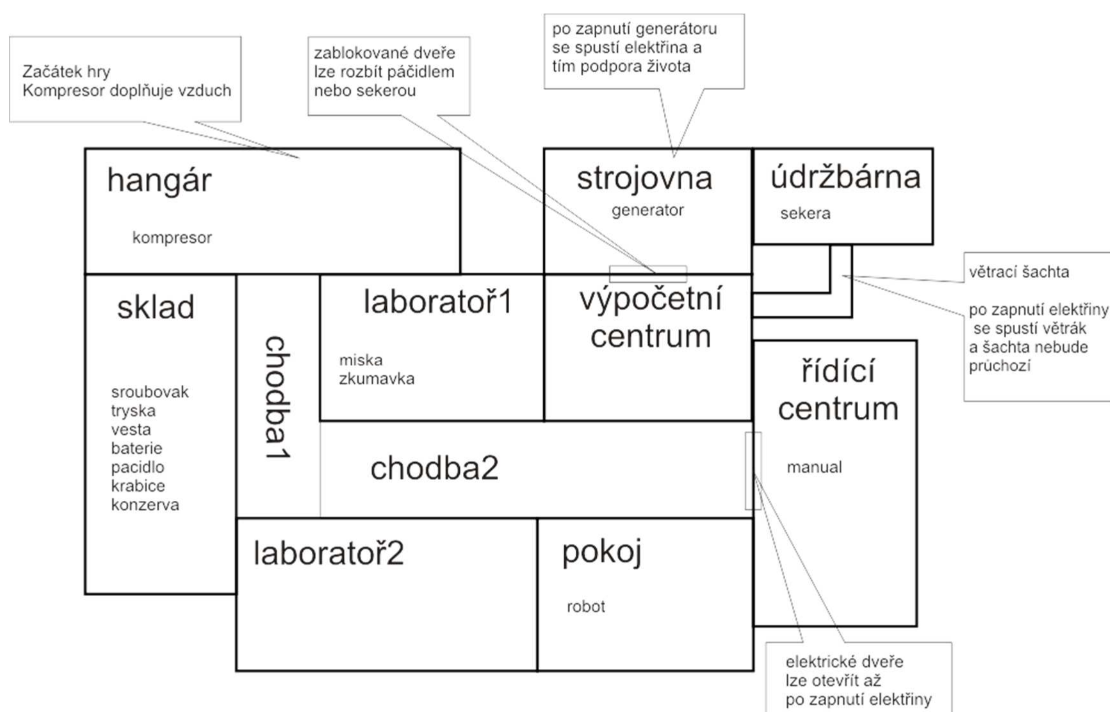
V tomto okamžiku vývoje je určeno, co by hra měla umět, jak by měla vypadat, o čem je i co by v ní mělo být. V následujícím kroku je potřeba promyslet, jak to všechno hráči zprostředkovat tak, aby ho hra bavila a nefrustrovala.

Aby bylo zajištěno dostatečné propracování všech úseků příběhu, je výhodné hru rozdělit na menší celky (levely) a ty řešit samostatně. Hráč postupně prochází jednotlivými levely a po dokončení všech úrovní vítězí. Pro úspěšné ukončení levelu by hráč měl splnit jednu nebo více podmínek, například dostat se z místa a na místo B, najít klíč od dveří, najít heslo od terminálu apod. Obtížnost hry lze měnit zakomponováním různých omezení a překážek, které mohou hráči v úspěšném ukončení zabránit, například omezené množství vzduchu ve skafandru, zablokované dveře atd.

Cesta k vítězství by neměla být úplně triviální. Hra by pro hráče měla představovat výzvu. Na druhou stranu, aby byla zábavná, neměla by být extrémně obtížná a měla by tolerovat občasnou chybu hráče.

Aby hráč nebyl při hraní zbytečně frustrován, je třeba se vyvarovat náhlých nesmyslných konců hry bez předchozího varování. Stejně tak je třeba se vyvarovat slepých konců, kdy by se hráč mohl dostat do situace, z níž nelze najít východisko (nemůže pokračovat dál ani se nemůže vrátit zpět). Ve hře by měla být zakomponovaná možnost alternativního průchodu.

Nejlepším způsobem, jak se podobným chybám vyvarovat, je nejprve si nakreslit mapu prostředí dané úrovně hry. Na mapě si pak vyznačit možné cesty k vítězství, domyslet všechny problémy, které by mohly nastat, a do hry přidat alternativní řešení, nebo úplně změnit mapu.(25)



Obrázek 1 Plán hry

5.8 Průběh hry

Celá hra bude velmi jednoduchá, a proto je navržena, jako kdyby se skládala pouze z jedné úrovně. Po nakreslení plánku následuje sepsání cest k vítězství a k prohře. Tím se získá přehled o problémech, které by mohly ve hře nastat.

U všech zapsaných cest se vychází z předpokladu, že hráč začíná v místnosti hangár a má na sobě skafandr. Není důležité sepsat všechny možné průběhy hry, protože těch může být

velmi mnoho. Jde o sepsání takových cest, které přímo vedou k vítězství či prohře, nebo takových, ve kterých hráči mohou narazit na nějaký problém.

5.8.1 Cesty k vítězství

Hráč vyjde z hangáru do místnosti chodba1 a následně do skladu. Sebere páčidlo a baterii. Vyjde ze skladu a pokračuje do místnosti chodba2. Odtud dál do výpočetního centra. Použije páčidlo na zablokované dveře. Jde do strojovny a spustí generátor. Vrací se zpět do místnosti chodba2 a jde do pokoje, kde si promluví s robotem, aby získal nápovědu k dešifrování kódu. Jde do řídícího centra. Vloží baterii do manuálu a získá kód. Hráč jej dešifruje a zadá do terminálu. Hráč vítězí a hra se ukončí.

Alternativní cesta k vítězství: Hráč ve skladu sebere pouze baterii. Z výpočetního centra proleze skrz větrací šachtu do údržbárny, kde sebere sekeru. Jde do strojovny a zapne generátor. Dále je průběh stejný jako v hlavním scénáři.

5.8.2 Cesty k prohře

Vykonání některých příkazů odečte hráči trochu vzduchu ze skafandru. Příkaz „jdi“ 10 % a příkazy „seber“, “polož“ a „použij“ 5 %. Když hráč bude stanici zkoumat příliš dlouho, už se nestihne vrátit do hangáru, aby si doplnil vzduch ve skafandru. Hráči se po vykonání příliš mnoha příkazů vyčerpá zásoba vzduchu a hra se ukončí.

Hráč může ukončit hru příkazem „konec“.

5.9 Agilní metoda

Po dokončení rozvržení hry následuje výběr metodiky, podle které se bude postupovat při jejím vývoji. Pro případ webové hry lze využít všechny výše popsané metodiky. Pro vývoj jedním programátorem jsou však nejvhodnější metodiky návrhem řízeného testování a testy řízený vývoj, které jsou nezávislé na velikosti týmu a určují, jak by programátor měl při vývoji aplikace postupovat. Ostatní metodiky především určují způsob, jakým je vedena komunikace mezi členy týmu a mezi týmem a zákazníkem.

5.10 Návrh programu podle DDT

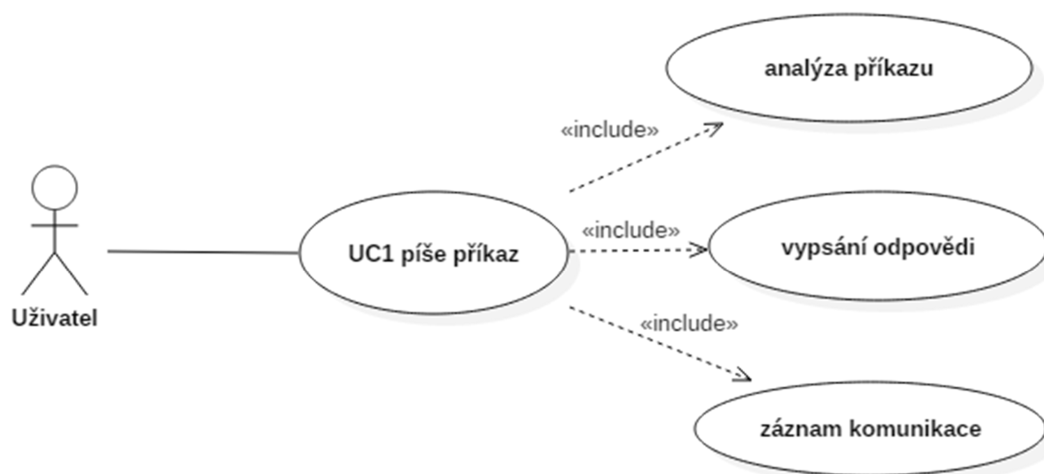
Po dokončení teoretické přípravy hry následuje vytvoření návrhu aplikace. Návrh aplikace podle metodiky DDT postupuje v následujících krocích:

1) vytvoření Use Case diagramu
2) sepsání průběhů případů užití
3) navržení diagramu robustnosti
4) připravení všech scénářů testů
5) vytvoření návrhu aplikace – Domain diagram
6) vytvoření testů dle scénářů testů - TDD
7) napsání kódu

5.10.1 Use Case diagram

Po rozvržení hry následuje návrh aplikace. Prvním krokem návrhu je vytvoření Use Case diagramu. Diagram textové hry je velmi jednoduchý, neboť jediné, co uživatel může dělat, je psát příkazy. Nebude zde žádná registrace ani tlačítka na obsluhu programu. Nicméně jeden případ užití nemusí být to jediné, co může být v tomto diagramu zapsáno. Pokud se za jedním případem užití skrývá složitější funkcionalita programu, může být tato funkcionalita zahrnuta do diagramu pomocí vazby *include*. Každý případ připojený touto vazbou se spustí pokaždé, když je spuštěn případ užití, na který jsou takové případy napojeny. Vazbu include lze použít i v případě, že má být spuštěna další funkce po spuštění jednoho případu užití.

V případě textové hry hráč napíše příkaz, potvrdí jej stisknutím klávesy ENTER a čeká na odpověď. Aplikace musí nejprve příkaz analyzovat, zjistit, o jaký příkaz se jedná, a vyhodnotit, jestli je příkaz ve správném formátu. Na základě příkazu je pak vypsána odpověď, která se odvíjí podle pozice hráče ve hře a podle stavu hry. Zároveň by veškerá komunikace uživatele s aplikací měla být zaznamenána a zobrazena na obrazovce. Po uvědomění si této funkcionality v rámci návrhu, je možné ji zaznamenat do diagramu. Proto k prvnímu případu užití „píše příkaz" budou připojeny vazbou *include* tři další případy: analýza příkazu, vypsání odpovědi, záznam komunikace.



Obrázek 2 Use Case diagram

5.10.2 Průběhy případů užití

Průběhy jednotlivých případů užití budou popsány podobně, jako byly v přípravě hry popsány průběhy hry (tj. cesty k vítězství a cesty k prohře) a k nim připojené základní a alternativní průběhy.

Základní průběh:

Uživatel napíše správný příkaz s požadovaným počtem korektních parametrů. Příkaz je následně analyzován – ověří se existence příkazu, zjistí se počet parametrů a ověří se správnost parametrů. Poté je příkaz zpracován a program vypíše odpověď. Příkaz i odpověď jsou zapsány do záznamu komunikace.

Alternativní průběhy:

Příkaz neexistuje (příp. je špatně napsaný) – systém zobrazí zprávu „Takový příkaz neexistuje."

Příkaz má více parametrů - systém zobrazí zprávu „Tento příkaz požaduje (počet) parametrů."

Příkaz má špatný parametr – zde se odpovědi budou lišit na základě příkazu.

Příklad:

příkaz polož:

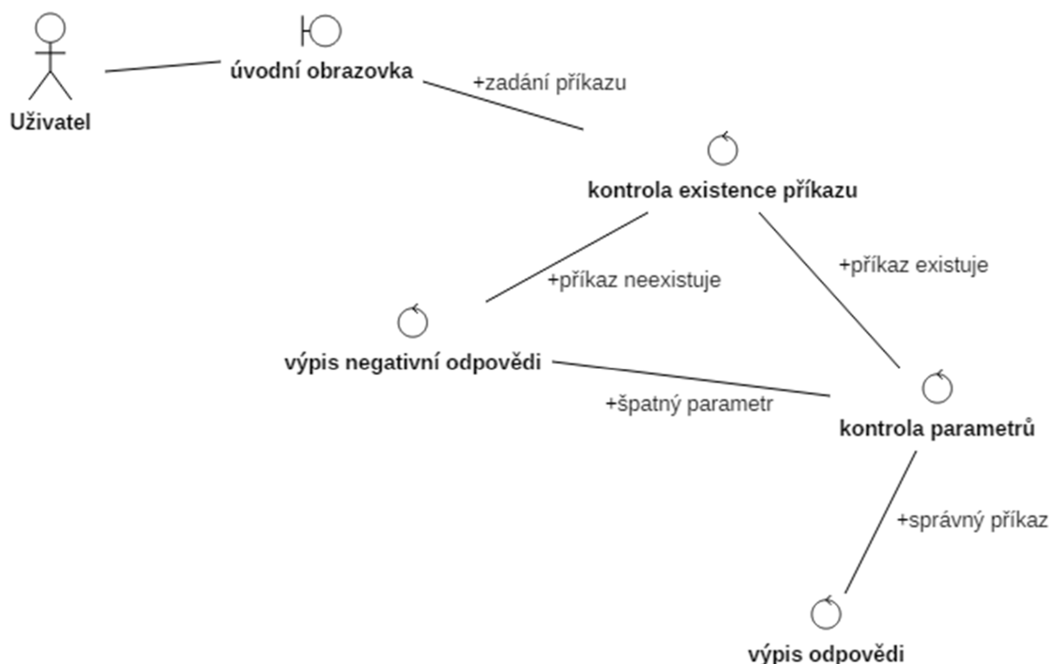
- věc není v batohu - systém zobrazí zprávu „Věc (věc) se nenachází v batohu."
- věc je v batohu - systém zobrazí zprávu „Věc (věc) byla vyndána z batohu."

příkaz seber:

- věc je v místnosti - systém zobrazí zprávu „Věc (věc) byla vložena do batohu."
- věc není v místnosti - systém zobrazí zprávu „Věc (věc) není v této místnosti."
- věc nejde sebrat - systém zobrazí zprávu „Věc (věc) nelze sebrat."
- batoh je plný (5 věcí) - systém zobrazí zprávu „Batoh je plný."
- věc je větší, než je volné místo v batohu - systém zobrazí zprávu „Věc (věc) je příliš těžká a nevejde se již do batohu."

5.10.3 Diagram robustnosti

Diagram robustnosti ukazuje, co má systém udělat na základě uživatelské činnosti. Uživatel zadá příkaz a následně je zkontrolována existence příkazu. Pokud příkaz neexistuje, vypíše se negativní odpověď. Pokud existuje, jsou zkontrolovány parametry příkazu. Vypíše se negativní odpověď v případě, že parametry nejsou validní. Jsou-li správné, vypíše se odpověď na základě příkazu. Vytvořením tohoto diagramu se zjistí, které části systému je potřeba otestovat. V diagramu robustnosti se nacházejí kolečka se šipkou. Tyto prvky se nazývají řadiče a každý z řadičů představuje jeden testovací případ. Všechny případy by měly být otestovány, aby se zaručila správná funkčnost těchto částí systému. Před psaním testů je vhodné si všechny nejprve promyslet a poté zapsat do tabulky scénářů testů.



Obrázek 3 Diagram robustnosti

5.10.4 Scénáře testů

Z diagramu robustnosti jsou vyvozeny testovací případy. Ke každému případu je připraveno co nejvíce scénářů, které by mohly nastat. Testovací scénáře se zapisují ve formě tabulky, která obsahuje čtyři sloupce: název testovacího scénáře, popis, testovací vstup a očekávaný výsledek testu. Tabulka by měla obsahovat alespoň jeden scénář, který projde správně.

První testovací případ je „kontrola existence příkazu“. Bude testováno, jestli se uživatelem zadaný příkaz shoduje s některým z příkazů ve hře. Jestliže se neshoduje, je očekávána negativní odpověď programu. Na každý vstup hráče by hra měla poskytnout nějaký výstup. Scénáře testů tohoto případu budou minimálně dva. V prvním je testována reakce programu na příkaz, který existuje, a ve druhém na příkaz, který neexistuje. Následně je testována správnost parametrů. Scénáře musejí pokrýt možnosti, kdy nebyl zadaný žádný parametr, kdy byl zadán správný parametr, špatný parametr, nebo více parametrů. Tím se zároveň testují i řadiče týkající se odpovědí programu. Takovéto scénáře by měly být navrženy pro všechny příkazy, protože odpovědi programu se budou lišit v závislosti na příkazu a na aktuálním stavu hry. Mnohé ze scénářů budou obsahovat již dříve popsané průběhy případů

užití. V tabulce scénářů testů jsou zapsány vstupy, které budou později testovány v různých stavech hry. Z průběhů případu užití jsou známy odpovědi programu, vůči kterým se budou vstupy testovat.

Testovací případ	**popis**	**vstup**	**kritéria přijatelnosti**
neexistující příkaz		exit	neprojde
příkaz jdi - správný východ	optimistický scénář	jdi chodba1	projde
příkaz jdi - nesprávný východ	pokoj není mezi východy	jdi pokoj	neprojde
příkaz jdi - špatný parametr	neexistující místnost	jdi východ	neprojde
příkaz jdi - více parametrů		jdi chodba1 chodba2	neprojde
příkaz jdi - žádný parametr		jdi	neprojde

Tabulka 2 - Ukázková tabulka se scénáři pro neexistující příkaz a pro příkaz "jdi"

V tabulce je vidět, že je testován příkaz, který neexistuje, i příkaz, který existuje. Příkaz, který existuje, je dále testován se správným a se špatným parametrem a s žádným, nebo s více parametry. Do scénářů je možné zařadit i takové testy, které testují neočekávané chování programu. Například případy „příkaz jdi - nesprávný východ“ a „příkaz jdi - špatný parametr“ se mohou zdát na první pohled stejné, protože oba testují nesprávný parametr, ale otestováním prvního případu je zajištěno, že se hráč nesmí posunout do místnosti, která není mezi východy aktuální místnosti. Je možné, že by si programátor píšící kód hry toto neuvědomil a hráč by se tak mohl teleportovat po celém plánu hry. Kdyby byl testován pouze nějaký špatně zadaný parametr, nezjistilo by se, že hra nefunguje správně. Proto je důležité vymyslet dostatečné množství scénářů a testovat vše, co by uživatel mohl zadat.

Tato tabulka samozřejmě není neměnná a je velmi pravděpodobné, že než se hra naprogramuje, přibude ještě množství dalších scénářů zajišťujících otestování krajních případů hry. Například další scénáře příkazu „jdi“ budou zjišťovat, jestli hráč může projít zablokovanými dveřmi, než je rozbije a jestli jimi může projít po jejich rozbití. Jestli může projít elektrickými dveřmi, když je elektřina vypnutá, nebo zapnutá. V závislosti na stavu elektřiny se bude ověřovat odčítání vzduchu ve skafandru apod. Na těchto příkladech je názorně ukázáno, že i v případě jednoduché textové hry musí programátor ošetřit poměrně velký počet očekávaných i neočekávaných situací.

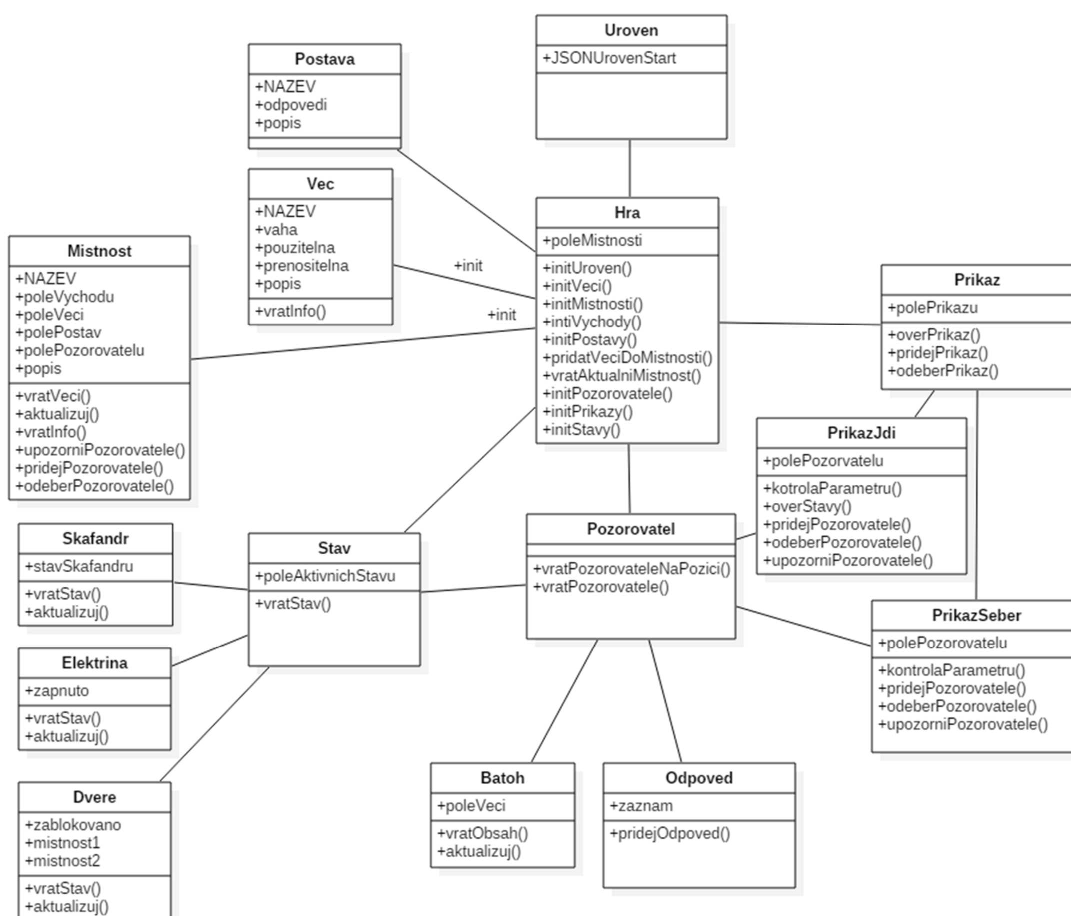
5.10.5 Návrh aplikace

Po dokončení přípravy je přesně definováno, co má aplikace dělat. Dalším krokem je navržení architektury samotného programu. Cílem agilního vývoje je vytvářet software, který bude vždy připraven na změny a rozšíření. Proto je třeba architekturu programu navrhnout tak, aby byl snadno rozšiřitelný o další funkcionality. V zadání je přesně vymezeno, co by program měl dělat, ale v budoucnu se do hry může například přidat hudba, která se bude měnit v závislosti na lokaci nebo situaci. Nebo se hra bude rozšiřovat o další úrovně.

Krokem, který zajistí snadnou rozšiřitelnost programu, je oddělení funkčnosti jednotlivých komponent, které nemusí být mezi sebou nutně provázány. U jednotlivých příkazů nemusí být odčítání vzduchu ve skafandru ani připravené odpovědi programu. Pro každý příkaz, skafandr, odpovědi, místnosti, věci aj. je lepší mít samostatné komponenty (třídy, objekty) a změny stavů provádět nepřímo skrze prostředníka, například s využitím návrhového vzoru Observer.

Návrh aplikace spočívá ve vytvoření alespoň Domain diagramu nebo Class diagramu. Při tvorbě diagramu se rozmýšlí, které objekty budou v programu potřeba a jak mezi sebou budou komunikovat. V textové hře hráč prochází skrze místnosti, které mohou obsahovat různé věci nebo osoby, se kterými by si hráč mohl promluvit. Věci může vkládat do batohu. K interakci s hrou používá příkazy, které fungují odlišně, a po jejich použití jsou vypisovány různé odpovědi. Diagram tedy obsahuje objekty Místnost, Věc, Postava, Batoh, Odpověď a jednotlivé příkazy Jdi, Seber atd. Aby byl program nezávislý na ději hry, je zde objekt Úroveň, který má v sobě JSON objekt s veškerými informacemi o jednotlivých úrovních. Obsahuje všechny místnosti s popisem, věci, postavy. Dále obsahuje informace o věcech a postavách v jednotlivých místnostech, o použitých příkazech a o využitých stavech (skafandr, zablokované dveře apod.) Podle těchto informací je inicializována celá hra v objektu Hra. Pro uchování informací o jednotlivých příkazech je použit objekt Příkaz, který v sobě má pole obsahující instance všech v úrovni použitých příkazů. Objekt Stav obsahuje pole aktivních stavů v úrovni. Komunikace mezi objekty je zprostředkována skrze objekt Pozorovatel podle návrhového vzoru.

Kód vytvořený podle doménového diagramu nemusí být funkční. Je velmi pravděpodobné, že se během vývoje budou měnit vazby mezi objekty nebo se budou do jednotlivých objektů přidávat další funkce. Diagram tedy popisuje pouze základní architekturu programu. Dalším krokem v návrhu aplikace může být vytvoření detailnějšího Class diagramu, nicméně je možné začít programovat i na základě doménového diagramu a případné chyby řešit během vývoje.



Obrázek 4 Domain diagram

5.11 Využití TDD

Technika testy řízeného vývoje spočívá v psaní testů jednotek před samotným kódem. Bez předchozího návrhu aplikace by její použití bylo značně obtížné, protože by vyžadovala psát testy a zároveň vymýšlet celou architekturu programu. Tento postup je možný, ale vedl by k velmi častému refaktorování kódu i testů.

Postup techniky TDD je následující:

- zvolit si novou jednotku k implementování
- napsat test, který očekává určitý výstup (např. návratovou hodnotu funkce)
- spustit test – bez implementované jednotky neprojde
- napsat jednotku tak, aby test prošel
- napsat další test (např. volat funkci s jiným parametrem)
- spustit test
- pokud test neprojde - upravit jednotku tak, aby prošly všechny testy

Poslední tři kroky se opakují tak dlouho, dokud funkce nepracuje přesně tak, jak programátor zamýšlí. Následně si programátor zvolí novou funkci a proces opakuje. Tento postup má několik výhod. Tím, že se spustí test před implementováním jednotky, je ověřeno, že se v kódu nenachází jiná jednotka se stejným názvem. Díky postupnému opravování jednotky je zajištěno, že jednotka funguje přesně podle požadavků programátora. Další výhodou je, že se tímto způsobem píše pouze testovatelný kód, tzn. že funkce vykonává pouze jednu činnost. Pokud se v takovém kódu vyskytne chyba, je snadné ji odhalit.

5.11.1 Testovací framework - Jasmine

Pro psaní testů se využívají testovací frameworky. Některá vývojová prostředí pro klasické programovací jazyky (Java, C++) mají v sobě přímo implementovaný některý framework - obvykle nejpoužívanější JUnit. Pro testování javascriptu se využívají externí knihovny, které testují kód ve webovém prohlížeči. Těchto knihoven je nepřeberné množství a každá z nich nabízí jiné možnosti testování specifických problémů. Mezi tyto knihovny patří Buster.js, Karma, QUnit, TestSwarm, Jasmine aj.

Jasmine je testovací framework běžící na straně klienta, který je nezávislý na jiných javascriptových frameworcích. Testy se v něm shlukují do *describe* bloků, ve kterých jsou

bloky *it*, které popisují, co má testovaná jednotka dělat. Pro porovnání očekávané hodnoty se získanou se používá příkaz *expect*.

5.11.2 Testy jednotek

Technika TDD se nejlépe vysvětlí na příkladu, ve kterém bude vytvořena metoda objektu Batoh, která má za úkol zkontrolovat přidávanou věc do batohu. Zkontroluje, zdali je přijatý parametr instance objektu Věc, jestli je věc přenositelná a jestli je v batohu dostatečné místo. Jestliže parametr jednomu z požadavků nevyhoví, vrátí hodnotu false, jinak vrátí hodnotu true.

Kód i testy jsou psány v angličtině. V tomto příkladu se počítá s již vytvořenými objekty Inventory a Thing.

Postup je následující:

1) vytvoření bloku *describe* s popisem „Inventory“ – testovaná metoda je součástí objektu Inventory
2) vytvoření bloku *it* pro testovanou metodu
3) vytvoření bloku *beforeEach* v bloku *describe* – kód v tomto bloku se spustí před vykonáním každého bloku *it*
4) připravení testovacích objektů v bloku *beforeEach* – objekt inventáře a objekty věcí
5) napsání testu, ve kterém se zavolá metoda checkThing(thing) a porovná se její návratová hodnota s očekávaným výsledkem, který je určen předaným parametrem „thing“
6) spuštění testu
7) přidání metody checkThing() do objektu Inventory
8) upravení metody tak, aby test prošel
9) dopsat další testy a upravovat metodu

```
describe('Inventory ', function() {
        var inv;
        beforeEach(function() {
              this.inv = new Inventory();
        }
}
```

Ukázka 1: inicializování *describe* a *beforeEach* bloku

```
it('should check thing being added to inventory', function() {
        var light = this.inv.checkThing(this.lightThing);
        expect(light).toEqual(true);
});
```

Ukázka 2: ověření hodnoty proměnné „light“

```
Inventory.prototype.checkThing = function(thing){
        if(!(thing instanceof Thing)) return false;
        if((this.weight + thing.weight) > this.maxSize) return false;
        return true;
}
```

Ukázka 3: výsledný kód

5.12 Integrační testy

Integrační testy zajišťují funkčnost systému složeného alespoň ze dvou jednotkově otestovaných funkcí. Testy porovnávají, zdali provedení funkcí ovlivní systém požadovaným způsobem. Nejjednodušším příkladem integračního testování může být otestování, jestli se po přidání věci do batohu změní i váha batohu podle váhy přidávané věci.

```
it('should change weight by weight of thing being added to inventory', function() {
        expect(this.inv.weight).toEqual(0);
        this.inv.addThing(this.lightThing);
        expect(this.inv.weight).toEqual(1);
        this.inv.addThing(this.hevThing);
        expect(this.inv.weight).toEqual(4);
}
```

Ukázka 4: integrační test

Dalším krokem je přidání volání funkce checkThing(). Test nejprve ověří, že předávaný parametr je věc, která může být vložena do batohu, a následně zkontroluje, jestli se po jejím přidání změní váha batohu. Potom je obdobně testován případ, kdy je věc odebírána z batohu. V dalším kroku se tyto dvě části integrují do sebe tak, že se testuje, jestli je věc přidávána do batohu na základě příkazu „seber“ nebo odebírána z batohu po zadání příkazu „poloz“. Postupně se tento test rozšiřuje a ve výsledku by měl integrovat všechny jednotky v programu.

5.13 Systémové testování

Po dokončení integračního testování následuje systémové testování. Tím se ověřuje, jestli je program funkční podle zadání zákazníka, zabezpečený, stabilní apod. Další klíčovou částí systémového testování je testování robustnosti, tedy ověřování, že program adekvátně reaguje na špatné vstupy zadané uživatelem.(11, s. 65-66) Některé části systémového, ale i akceptačního testování webových aplikací, lze velmi dobře otestovat pomocí nástroje Selenium IDE.

Práce s programem spočívá v zaznamenání akcí testera a k nim je doplněn test, který kontroluje, jestli program správně zareagoval, například ověřením, že se po kliknutí na tlačítko objeví očekávaný text. Všechny zaznamenané akce pak program může znovu provést sám. Tyto záznamy slouží k ověření, že ve chvíli, kdy je změněn kód, funkcionalita programu zůstane nezměněna. Další možností použití Selenia IDE je vytvoření záznamů sloužících k otestování průběhů případů užití (kapitola 5.10.2) a průběhů samotné aplikace (kapitola 5.8). Program lze použít i k připravení testů dopředu před napsáním samotné funkcionality. Tyto testy pak slouží jako vodítko programátorovi, co dále programovat a na jaké části programu se má zaměřit.

Příkladem testu hry s využitím programu Selenium IDE může být ověření, že nelze sebrat věc, která není v místnosti, v níž se nachází hráč. Příkaz je spuštěn stisknutím klávesy ENTER.

```
<tr>
        <td>type</td>
        <td>id=textInput</td>
        <td>seber hrnek</td>
</tr>
<tr>
        <td>keyPress</td>
        <td>id=textInput</td>
        <td>\13</td>
</tr>
<tr>
        <td>verifyText</td>
        <td>id=mainWindow</td>
        <td> Tato věc není v místnosti</td>
</tr>
```

Ukázka 5: test Selenium IDE

5.14 Akceptační testy

Akceptační testování je testování produktu na straně zákazníka a následuje po dokončení systémového testování a předání produktu zákazníkovi. Toto testování provádějí testeři podle připravených plánů.(11, s. 67) V případě hry budou testery hráči, kteří si hru zahrají a vyplní záznam o testování. Do něho zapíši veškeré chyby, na které při hraní narazili.

Záznam o testování:

Úkolem každého z testerů bylo zahrát si hru alespoň 2x. Poprvé se ji měli pokusit dohrát co nejrychleji a zaznamenat potřebný čas, který bude využit k vyhodnocení obtížnosti hry. Při druhém hraní se měli zaměřit na technické chyby, zejména ověření adekvátních odpovědí hry na různé vstupy, a na logické chyby, jako nesmyslné chování hry, špatné ovládání apod. Své jméno, potřebný čas, technické i logické chyby následně vyplnili do záznamu testování.[2]

[2] Hra je dostupná na www adrese http://kraken.pedf.cuni.cz/~hodanl/BP/ a na přiloženém CD.

Výsledky testování:

Na základě deseti vyplněných záznamů byly ve hře nalezeny dvě závažné technické chyby. První chyba se týkala nereagování hry na příkaz „vloz_kod“ zadaný v místnosti bez vysílačky. Druhou chybou bylo občasné zastavení odpočtu vzduchu ve skafandru. Obě chyby vznikly nepozorností po úpravě kódu po přidání nového příkazu ‚nova_hra‘. První chyba vznikla přepsáním pozorovatele pro vysílačku a druhá se objevila při změně podmínky k ukončení hry.

Další poznámky se týkaly nepřehledného výpisu při zadávání většího množství nesprávných vstupů. Na základě těchto připomínek byly hráčem zadané vstupy zvýrazněny. Po dohrání hry byl hráč dotázán formou dialogového okna, zda chce pokračovat ve hře. Čtyři testeři poznamenali, že po kliknutí na tlačítko „zrušit“ již není žádným způsobem umožněno hrát hru znovu. Proto byl do hry přidán nový příkaz „nova_hra“. Tento příkaz lze spustit kdykoliv během hraní a je to jediný příkaz, který lze do hry zadat po jejím ukončení. Poslední častěji se vyskytující poznámka se týkala krátkého a nedostatečně rozvinutého děje. Děj hry nebyl změněn, protože cílem práce bylo vytvořit funkční hru, na jejímž vývoji budou ukázány různé techniky a postupy vývoje. Hra však byla vytvořena tak, aby mohla být v budoucnu snadno rozšířena o další děj a další možnosti.

Ze záznamů testů dále vyplynulo, že náročnost hry je v souladu s očekáváním. Doba potřebná k dohrání hry byla odhadnuta na 15 – 25 minut. Nejdéle hrajícímu testerovi dohrání hry trvalo 36 minut a nejkratší doba hraní byla 6 minut. Průměrná doba hraní vypočtená z deseti záznamů byla 16 minut.

5.15 Refaktoring

Refaktoring je technika, za kterou se považuje zjednodušování, rozčleňování, zpřehledňování nebo zobecnění kódu. Je to nedílná součást programování. K častému využití refaktorování dochází ve chvíli, kdy programátor přidá jednu vlastnost k objektu a v průběhu programování stejnou vlastnost přidá dalším objektům. Je tedy vhodné, aby se programátor zamyslel, jestli není nějaký obecnější způsob přidávání vlastnosti k těmto objektům.(5)(6)

```
var cmdGo = this.command.getCommand("jdi");

cmdGo.addObserverAnswer(this.answer);

var cmdTake = this.command.getCommand("seber");

cmdTake.addObserverAnswer(this.answer);
```

Ukázka 6: kód vhodný pro refaktorování

```
for(i = 0; i < commands.length; i+=1){

        commands[i].addObserverAnswer(this.answer);

}
```

Ukázka 7: výsledný kód po refaktorování

5.16 Využití návrhového vzoru Factory Method

Návrhový vzor Factory Method se využívá hlavně ke zjednodušení vytváření objektů. V textové hře jej lze využít k vytvoření objektů věcí, míst a osob. Využití tohoto vzoru spočívá ve vytvoření funkce *faktory*, která přijme parametry, na základě kterých vytvoří objekt a ten vrátí jako návratovou hodnotu.(19)

```
var manual ={

        nazev : "manual",

        popis : "Elektronický tablet",

        pouzitelna : true,

        prenositelna : true,

        vaha : 1

};
```

Ukázka 8: vytvoření objektu „manual“ bez použití návrhového vzoru

```
function factory(nazev, popis, pouzitelna, prenositelna, vaha){

	var vec ={

		nazev : nazev,

		popis : popis,

		pouzitelna : pouzitelna,

		prenositelna : prenositelna,

		vaha : vaha

	};

	return vec;

}

var manual = factory("manual","Elektronický tablet", true, true, 1);
```

Ukázka 9: vytvoření objektu „manual“ s použitím návrhového vzoru Factory Method

5.17 Zabezpečení

Součástí systémového i akceptačního testování jsou testy bezpečnosti. Takové testy se nazývají penetrační testy. Jejich cílem je zjistit, jestli se v programu nevyskytují nezabezpečené nebo špatně navržené části, které by mohl někdo využít ke svému prospěchu.

Všechny aplikace psané v jazyce JavaScript jsou potencionálně nebezpečné, protože celý jejich kód je k dispozici v prohlížeči klienta. Pokud by aplikace posílala a přijímala nějaká data ze serveru, je možné, aby někdo záměrně používal skripty k získávání citlivých údajů ze serveru.

V případě textové hry, která kompletně běží na straně klienta a nijak nekomunikuje s žádným serverem, je její nejzranitelnější částí vstupní pole, do kterého hráči píší své příkazy. Webové prohlížeče se snaží spustit každý skript, a proto spouští i skripty zadané do vstupních polí. Pro textovou hru to znamená, že je možné, aby někdo místo zadání příkazu vložil do vstupního pole skript, a tím si přidal do batohu novou věc, změnil si aktuální místnost apod. Aby bylo této možnosti zamezeno, je nutné, aby byly veškeré speciální znaky (<, >, ‘, “, &) nahrazeny jejich zástupnými znaky („&“,“<“ atd.). Zástupné znaky prohlížeč vidí jako textové znaky a nesnaží se zadaný vstup spustit jako skript. Takovéto jednoduché zabezpečení je i základním ošetřením komunikace mezi serverem a klientem.

Dalším krokem je znesnadnění čtení kódu aplikace přímo v prohlížeči. Nejjednodušší možností je využití různých minifikátorů, které mají za úkol zmenšit velikost JavaScriptových souborů, aby jejich načítání a spouštění bylo rychlejší. Minifikace spočívá v nahrazení názvů proměnných samostatnými písmeny (nebo krátkými sekvencemi znaků) a v odstranění mezer a řádkování. V takovém kódu se ale zachovávají klíčová slova jazyka (function, return aj.). Proto lze minifikovaný kód znovu zpřehlednit s využitím jiných programů. Je však možné zmenšený kód ještě „poplést“ a změnit tak, aby jej takové programy nedokázaly jednoduše převádět zpět do čitelné formy. K tomu je možné využít i online aplikaci JavaScript Obfuscator, která nahradí znaky v textových řetězcích a názvy proměnných jejich zápisem v hexadecimální soustavě a zároveň se snaží kód zpřeházet a přerovnat tak, aby případnému útočníku trvalo co nejdéle kód převést zpět do čitelné formy. Tento krok sice nezajistí, že kód hry nikdo nedokáže znovu zpřehlednit, ale alespoň znemožní jeho čtení přímo v prohlížeči.(26)

```
Game.prototype.initPlaces = function(level){

    for(var i=0; i<level.places.length; i+=1){

        this.arrayOfPlaces.push(new Place(level.places[i][0],level.places[i][1]));

    }

}
```

Ukázka 10: metoda pro inicializování místností

```
var _0x8a4b=["\x69\x6E\x69\x74\x50\x6C\x61\x63\x65\x73",
"\x70\x72\x6F\x74\x6F\x74\x79\x70\x65","\x6C\x65\x6E\x67\x74\x68",
"\x70\x6C\x61\x63\x65\x73","\x70\x75\x73\x68","\x61\x72\x72\x61\x79\x4F\x66\x50\x6C\x61\x63
\x65\x73"]; Game[_0x8a4b[1]][_0x8a4b[0]]=function(_0x4850x1){for(var
_0x4850x2=0;_0x4850x2<_0x4850x1[_0x8a4b[3]][_0x8a4b[2]];_0x4850x2+=1){this[_0x8a4b[5]][_0x8
a4b[4]](new Place(_0x4850x1[_0x8a4b[3]][_0x4850x2][0],
_0x4850x1[_0x8a4b[3]][_0x4850x2][1]))}}
```

Ukázka 11: předchozí kód po minifikaci a po využití programu JavaScript Obfuscator

Je ale očividné, že pro textovou hru je takovéto zabezpečení naprosto dostačující.

Závěr

Cílem práce bylo seznámit čtenáře se soudobými technikami a postupy, které se využívají při tvorbě webových aplikací a ukázat jejich využitelnost na malých projektech. V teoretické části práce jsou popsány agilní metodiky, testování, UML diagramy a návrhové vzory. V praktické části práce je zdokumentován proces vývoje hry.

Výsledná textová hra byla navržena a vytvořena s využitím metodik *návrhem řízené testování* (DDT) a *testy řízený vývoj* (TDD), protože tyto dvě metodiky jsou nejvhodnější pro vývoj malých projektů jedním vývojářem. Kód hry byl navržen tak, aby byl co nejvíce nezávislý na ději. K tomu právě nejvíce přispělo využití obecných postupů, ke kterým metodiky vybízejí. Postup vývoje podle metodiky DDT spočíval ve vytvoření Use Case diagramu, sepsání průběhů případů užití, navržení diagramu robustnosti, připravení scénářů testů a navržení Domain diagramu. Provedení všech těchto kroků zabralo zhruba třetinu z celkového času potřebného k vytvoření celé hry, přičemž nejdéle trvalo dostatečně promyslet a navrhnout Domain diagram. Ačkoliv se vývoj aplikace s využitím metodiky DDT zdánlivě prodlužuje, opak se ukázal být pravdou. Každý z kroků v této metodice vede k hlubšímu promyšlení funkčnosti programu. Díky tomu jsem měl ještě před začátkem programování jasnou představu, jak jednotlivé objekty budou mezi sebou komunikovat, co bude potřeba za metody i co bude nutné ošetřit. Především z tohoto důvodu pak programování probíhalo velmi rychle. Připravené scénáře testů a scénářů průběhů užití urychlily přípravu odpovědí hry a posléze i systémové testování. Tato metodika se dá tedy výhodně použít při navrhování malých projektů nebo jen částí menších programů.

Hra zároveň prošla všemi úrovněmi testování. První úroveň se týkala jednotkového testování. Většina těchto testů vznikla v rámci metodiky TDD, ale některé metody byly dopsány navíc a jejich otestováním bylo nalezeno několik chyb. Jednotkové testování jak samotné, tak v rámci metodiky TDD, považuji za velmi přínosné, protože jeho využití vedlo k vytvoření poměrně čistého a čitelného kódu a navíc vedou programátora při programování a nedochází ke zbytečným chybám. Další úrovní bylo integrační testování, které se pro tento malý projekt neukázalo být příliš přínosné. Vytvoření testů, které integrují všechny metody jednotlivých objektů, je časově náročné a chyby, které by toto testování odhalilo, by byly odhaleny při systémovém nebo akceptačním testování. Dalším problémem byl poměrně

rozsáhlý refaktoring testů i při menších změnách ve funkčnosti programu. Přínos integračního testování bych viděl spíše při spojování částí programů od různých vývojářů.

Následovalo systémové testování s využitím programu Selenium IDE. Program byl využit i při samotném vývoji, kde sloužil k rychlému zadávání sekvencí příkazů s následnou manuální kontrolou odpovědí hry. Přínos Selenia IDE považuji za velmi výrazný, protože urychluje testování i vývoj. Systémové testování taktéž považuji za důležitou součást vývoje, protože díky tomuto testování bylo nalezeno množství překlepů, několik špatných odpovědí a odhaleno několik funkčních chyb, jako nereagující příkaz.

Poslední úrovní testování bylo akceptační testování, kdy hra byla testována desítkou testerů, kteří navrhli různá vylepšení uživatelského rozhraní, odhalili překlepy i závažnější chyby. Toto testování pomohlo k doladění hry po funkční stránce i po stránce estetické.

Na základě vytvoření této jednoduché hry je patrné, že lze vyvíjet jakkoliv velký projekt s využitím těchto metodik, ale v případě malých programů, u kterých se nepředpokládá budoucí rozšiřování, nemusí být jejich přínos tolik znát. V případě mé hry je ale přínos velmi výrazný, přestože výsledná hra obsahuje téměř 3x více kódu (cca 2000 řádků) a její vývoj trval skoro 5x déle (4 týdny) než dějově stejná hra, která nebyla vyvíjena podle žádné metodiky. Čas, který si vývoj nové hry vyžádal, se však rychle zúročí při dalším rozšiřování a doplňování funkcionality nebo při úplné změně děje.

6 Zdroje

1. Manifest Agilního vývoje software. In: *Agile manifesto* [online]. 2011 [cit. 2016-02-08]. Dostupné z:http://agilemanifesto.org/iso/cs/
2. CIMPL, Lukáš. Agile software development (Část první – Manifest Agilního vývoje software). In: *CZM Blog Centrum znalostního managementu* [online]. 12.2.2015 [cit. 2016-02-08]. Dostupné z: http://blog.czm-cvut.cz/agile-software-development-cast-prvni-manifest-agilniho-vyvoje-software
3. KNESL, Jiří. Agilní vývoj: Úvod. In: *Zdroják* [online]. 11.12.2009 [cit. 2016-02-12]. Dostupné z: https://www.zdrojak.cz/clanky/agilni-vyvoj-uvod/
4. Crystal Methodology COS 730. In: *Slide share* [online]. 2012 [cit. 2016-02-08]. Dostupné z: http://www.slideshare.net/bassuday/crystal-methodology
5. Refactoring. In: *Source Making* [online]. 2006 [cit. 2016-02-08]. Dostupné z: https://sourcemaking.com/refactoring
6. BANERJEE, Udayan. Brief History of Agile Movement. In: *Technology Trend Analysis* [online]. 23.3.2012 [cit. 2016-02-08]. Dostupné z: https://setandbma.wordpress.com/2012/03/23/agile-history/
7. KNESL, Jiří. Agilní vývoj: Scrum. In: *Zdroják* [online]. 18.12.2009 [cit. 2016-02-08]. Dostupné z: https://www.zdrojak.cz/clanky/agilni-vyvoj-scrum/
8. *Extreme Programming: a gentle introduction* [online]. Poslední změna 8.10.2013 [cit. 2016-02-11]. Dostupné z: http://www.extremeprogramming.org/
9. STEPHENS, Matt a Doug ROSENBERG. *Testování softwaru řízené návrhem*. Vyd. 1. Brno: Computer Press, 2011. ISBN 978-80-251-3607-2.
10. *Největší softwarové katastrofy* [online]. In: . 27.7.2009 [cit. 2016-02-18]. Dostupné z: http://www.chip.cz/casopis-chip/earchiv/vydani/r-2009/chip-07-2009/nejvetsi-sw-katastrofy/
11. ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. 1. vyd. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8.
12. History of UML: Methods and Notations. In: *Source Making* [online]. 2006 [cit. 2016-02-12]. Dostupné z: https://sourcemaking.com/uml/basic-principles-and-background/history-of-uml-methods-and-notations

13. Introduction. In: *Source Making* [online]. 2006 [cit. 2016-02-12]. Dostupné z: https://sourcemaking.com/uml/introduction
14. ROTEM-GAL-OZ, Arnon. Gang of Four Design Patterns - Does it stand the test of time? In: *InfoQ* [online]. 2007-7-30 [cit. 2016-03-20]. Dostupné z: http://www.infoq.com/news/2007/07/GoFCriticism
15. *UML – úvod* [online]. In: . 18.7.2006 [cit. 2016-02-12]. Dostupné z: http://mpavus.wz.cz/index.php
16. *UML – skladba* [online]. In: . 18.7.2006 [cit. 2016-02-12]. Dostupné z: http://mpavus.wz.cz/uml/uml-skladba-2.php
17. ČÁPKA, David. UML. In: *ITnetwork* [online]. [cit. 2016-03-21]. Dostupné z: http://www.itnetwork.cz/navrhove-vzory/uml
18. HANMER, Robert. *Pattern-oriented software architecture for dummies*. Chichester, England: Wiley, 2013, s. 54-58. ISBN 978-1-119-96399-8.
19. VALKOVIČ, Patrik. Factory (tovární metoda). In: *ITnetwork* [online]. 24.11.2015 [cit. 2016-02-21]. Dostupné z: http://www.itnetwork.cz/navrhove-vzory/factory
20. BÖHMER, Marian. *Návrhové vzory v PHP: [23 vzorových postupů pro rychlejší vývoj]*. 1. vyd. Brno: Computer Press, 2012. ISBN 978-80-251-3338-5.
21. MALÝ, Martin. PÍŠEME TEXTOVKU, DÍL PRVNÍ. In: *Misantropův zápisník* [online]. 21.2.2013 [cit. 2016-02-08]. Dostupné z: http://www.misantrop.info/piseme-textovku-dil-prvni/
22. TIŠNOVSKÝ, Pavel. Historie vývoje počítačových her (2.část – věk simulací). In: *Root* [online]. 22.11.2011 [cit. 2016-02-08]. Dostupné z: http://www.root.cz/clanky/historie-vyvoje-pocitacovych-her-2-cast-vek-simulaci/#ic=serial-box&icc=text-title
23. TIŠNOVSKÝ, Pavel. Historie vývoje počítačových her (4.část – zlatá éra textovek). In: *Root* [online]. 1.12.2011 [cit. 2016-02-08]. Dostupné z: http://www.root.cz/clanky/historie-vyvoje-pocitacovych-her-4-cast-zlata-era-textovek/#ic=serial-box&icc=text-title
24. SCHREIBER, Ian. Level 17: User Interfaces. In: *Game Design Concepts* [online]. 24.8.2009 [cit. 2016-02-08]. Dostupné z: https://gamedesignconcepts.wordpress.com/2009/08/24/level-17-user-interfaces/

25. RYAN, Tim. Beginning Level Design, Part 1. In: *Gamasutra* [online]. 16.04.1999 [cit. 2016-02-08]. Dostupné z: http://www.gamasutra.com/view/feature/131736/beginning_level_design_part_1.php?page=2
26. ALLEN, James. The Importance of Client-Side JavaScript Security. In: *Sitepoint* [online]. 10.8.2015 [cit. 2016-03-30]. Dostupné z: http://www.sitepoint.com/importance-client-side-javascript-security/

7 Přílohy

1. volně přiložené CD obsahující zdrojový kód hry