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

Diplomová práce

Zajištění kvality webové hry Space Traffic

Plzeň, 2013 Bc. Pavel Bořík

Prohlášení

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

V Plzni dne 12. 8. 2013 ……..

Bc. Pavel Bořík

Abstract

This master's thesis called Quality Assurance for the Space Traffic web game deal with student's project developed at the Department of Informatics at the University of West Bohemia. The reasons for creation of project Space Traffic was effort get more high school students for studying at Faculty of Applied Sciences and present the work of current students.

This work starts with general description of quality assurance. In this part is described the term of quality and the possible ways and methods how to ensure a quality in software products. In next part is review of testing techniques, testing strategies and different testing types. Practice part of work contains short description of project Space Traffic, its history, purpose, educational elements, contribution of author this work and how is ensure quality in this project. 5th chapter provides information about programming language Starship Basic used for programmable control of space-ships. In the following two chapters are written types of tests used in Space Traffic, their assessment and plan for beta testing. The last part contains quality and functionality summarization and the recommendations for further development.

Obsah

Poděkování

Na tomto místě bych rád poděkoval všem studentům, kteří se v tomto akademickém roce, ale i v letech předchozích, podíleli na vývoji a návrhu hry Space Traffic. Dále bych chtěl poděkovat vedoucímu práce Ing. Petru Vaněčkovi, Ph.D. za podporu a rady během vedení práce, a dalším členům katedry, kteří různým způsobem přispěli k rozvoji hry Space Traffic. Obzvláště pak Doc. Ing. Přemyslu Bradovi, MSc. Ph.D. za dohled nad hrou, Ing. Romanu Moučkovi Ph.D. za podporu při semestrálních pracích, které studenti realizovali v rámci jím vedených předmětů, a Ing. Richardu Lipkovi Ph.D. za konzultace při návrhu gramatiky a interpretace jazyku Starship Basic.

Závěrem bych rád poděkoval své rodině a známým za podporu, jež mi během studia poskytli.

1 Úvod

Tato diplomová práce je zaměřena na zajištění kvality webové hry Space Traffic, jež je vyvíjena výhradně studenty Fakulty aplikovaných věd na Katedře informatiky a výpočetní techniky. Hra je ve vývoji již několikátým rokem a vzhledem k tomu, že byl vyvinut dostatečný základ hry, dospěl projekt do fáze, kdy je nevyhnutelné začít dbát na kvalitu výsledné hry.

Zajištění kvality softwarových produktů je obecně vcelku problematickou částí vývoje softwaru. Prvním problémem při zajištění kvality totiž často bývá definice kvality. Každý člověk má o kvalitě svoji představu, ale její definování činní mnoha lidem problémy. Především v dřívějších dobách se často stávalo, že velké množství softwarových produktů nebylo zdárně dokončeno, protože nebyly z různých důvodů dostatečně kvalitní. S postupem času se však začalo na kvalitu produktů více dbát. Tuto snahu potvrzuje i skutečnost, že bylo provedeno mnoho kroků, které vedli k vytvoření definic kvality, metrik, změně vývojových postupů a také k vyššímu procentu úspěšně dokončených projektů.

Specifickou oblastí softwarových produktů jsou počítačové hry, tedy i hry webové. Tak jako u ostatních produktů je třeba i v případě her hledět na kvalitu, navíc je však nutné se zabývat také tím, zda hra bude hráče bavit. Základem každé hry je zejména zábava, a pokud hra není zábavná, není důvod ji hrát.

Jedním ze základních postupů při zajišťování kvality je včasné a časté testování. Testování je opět stejně jako kvalita netriviálním pojmem a zahrnuje značné množství technik, strategií a typů testů, které různým způsobem a různou měrou k zajišťování kvality přispívají.

Cílem této práce je prostřednictvím různých testovacích strategií zjistit, jaký je stav aktuální verze hry Space Traffic z hlediska kvality, a zajistit, aby současná kvalita byla přinejmenším udržována nebo lépe, aby se kvalita hry zvyšovala.

Pro snadnější orientaci čtenáře je tato práce rozčleněna do 9 kapitol, z nichž jednu tvoří tento úvod. Druhá kapitola je věnována kvalitě softwarových produktů a metodám a přístupům vedoucím k zajištění kvality softwaru včetně her. Třetí kapitola seznamuje čtenáře s problematikou testování, obecných technik a strategií používaných při testování softwarových produktů i specifických strategií uplatňovaných pro testování her. V následující kapitole je čtenář seznámen s projektem Space Traffic, jeho historií, účelem či použitými výukovými prvky. Pátá kapitola popisuje jazyk Starship Basic a jeho využití ve hře pro programovatelné ovládání lodí. Šestá kapitola je pak zaměřena na testování prováděné v průběhu vývoje. Další kapitola obsahuje plán play, usability a beta testů. Osmá kapitola shrnuje kvalitu hry a uvádí doporučení pro další vývoj. Poslední kapitolou je závěr, který shrnuje výsledky a přínos autora této práce.

2 Zajištění kvality softwarových produktů

Softwarové produkty, nebo též jen software, jsou v posledních letech součástí našeho každodenního života. Jejich využití je různé a můžeme je nalézt v nejrůznějších odvětvích lidské činnosti. Nicméně pokud se zeptáme většího počtu lidí, co si pod pojmem software představují, pravděpodobně dostanete velmi rozmanité odpovědi. Velká většina z nich Vám patrně odpoví, že softwarový produkt je nějaký program, případně aplikace, která běží na jejich počítačích. Toto tvrzení je do jisté míry pravdivé, ovšem pouze částečně, jelikož program je jen jednou ze součástí celého produktu. Jak uvádí například [1] softwarový produkt zahrnuje programy jakékoliv velikosti a architektury, jež jsou vykonávány v počítači, dokumenty ať již v tištěné nebo ve virtuální formě, a data, která kombinují čísla a text, ale také obsahují obrazové, video a audio informace.

Jak již bylo uvedeno, softwarové produkty se používají v různých odvětvích a každý softwarový produkt je jiný, přesto mají jednu věc společnou. Každý vytvořený software by měl plnit účel, pro který byl vyvinut, jinými slovy, měl by být natolik kvalitní, aby přinášel svým uživatelům přidanou hodnotu, kterou od něj očekávají. Jestliže má ovšem výrobce softwaru dodat kvalitní software, je nutné definovat, co se pod pojmem kvalita rozumí. Každý člověk má na kvalitu nějaký názor, ale vytvořit obecnou definici kvality je těžké, protože každý vnímá kvalita jinak. Přesto snahy o definici kvality neutichají a způsob, jak lze alespoň částečně kvalitu definovat bude uveden v sekci 2.1.

Historie zajišťování kvality ve vývoji softwaru je propojena s historií kvality hardwarové výroby. Během počátků práce na počítači (1950 a 1960), byla kvalita výhradní odpovědností programátora. Standardy pro zajišťování kvality softwaru byly zavedeny nejprve ve vojenské smlouvě o vývoji softwaru v průběhu roku 1970 a rychle se rozšířili do vývoje softwaru v komerčním světě [1]. S tím jak velikost softwaru rostla, měnili se i způsoby, jakým byl software vyvíjen. Z počátku šlo převážně o menší projekty, které bylo možné realizovat takzvaným vodopádovým modelem, kdežto v současné době jsou realizovány iterativním (přírůstkovým) způsobem. Je tomu tak především z důvodu, že se zvětšující se velikostí softwarových projektů často docházelo k dodání nekvalitního softwarového produktu. Z tohoto důvodu budu vliv jednotlivých způsobů vývoje, respektive přínosy iterativního přístupu oproti klasickému (vodopádový přístup), popsán v kapitole 2.3.

Další dopad rostoucí velikosti softwarových produktů se projevuje rovněž v tom, kdo nese zodpovědnost za kvalitu. V době realizace prvních programů bylo zajištění kvality softwaru pouze na programátorovi, který zodpovídal za vše. V současnosti však platí, že za zajištění kvality softwaru zodpovídá mnoho odborníků z různých odvětví, například softwaroví inženýři, projektoví manažeři, ale také zákazníci či jednotlivci spadající do skupiny Software Quality Assurance (SQA). Skupina SQA funguje jako zástupce

zákazníka, což znamená, že lidé, kteří vykonávají SQA pohlížejí na software z pohledu zákazníka [1].

Zajištění softwarové kvality se skládá z množství úkolů, které jsou rozděleny mezi dvě různé skupiny. První skupinu tvoří softwaroví inženýři, kteří dělají technické práce a druhou SQA skupina, která má odpovědnost za plánování kvality, dohled nad dodržováním procesů, vedením záznamů o proběhu vývoje, analýzu a reporty. Softwaroví inženýři zajišťují kvalitu a kontrolní činnosti prostřednictvím definovaných technických metod a postupů, provádějí formální technické revize (budou popsány v sekci 2.4) a starají se o dobré naplánovaní testování softwaru. V dosažení kvalitního produktu jim pak pomáhá nezávislá skupina SQA, viz [1], která:

- Připravuje plán SQA pro projekt.
- Podílí se na popisu vývoje softwarového procesu.
- Hodnotí činnosti softwarového inženýrství s cílem ověřit dodržování definovaného softwarového procesu.
- Provádí audity, které ověřují, zda jsou definovaná ustanovení součástí softwarového procesu.
- Zajišťuje, že odchylky oproti normám jsou dokumentovány a je s nimi zacházeno dle dokumentovaného postupu.
- Zaznamenává neshody s normami a hlásí je vrcholovému managementu.

2.1 Definice kvality softwaru

Pokud chceme dodávat kvalitní software, musíme nejprve definovat, co kvalita vlastně je. Jak ji ovšem definovat? V různé literatuře bylo navrženo mnoho definic, které se zabývají kvalitou softwaru. Jak uvádí jeden neznámý autor: "*Každý program dělá něco správně, to prostě nemůže být to, čeho chceme docílit.*" [1]. Mezi některé z dalších definic patří například definice od Philipa B. Crosbyho: „*Kvalita je shoda s požadavky*“[2], či dvě definice provedené J. M. Juranem: *„Kvalita se skládá z těch vlastností produktu, které splňují potřeby zákazníků a tím zajistí spokojenost s produktem.“* a *„Kvalita se skládá z nepřítomnosti nedostatků.*[2].

Standard IEEE uvádí následující dvě definice: „*Míra s jakou systém, komponenty, nebo proces splňuje uvedené požadavky.“*[2] a „*Míra s jakou systém, komponenty, nebo proces splňuje potřeby nebo očekávání zákazníka či uživatele.“*[2]

Jak tvrdí Robert Glass [1], mezi spokojeností uživatele a kvalitou existuje následující vztah: *spokojenost uživatele = kompatibilní produkt + dobrá kvalita + dodání v rámci rozpočtu a časového harmonogramu.* Glass [1] rovněž tvrdí, že kvalita je důležitá, ale v případě, že uživatel není spokojen, pak na tom nezáleží. S tímto názorem se ztotožňuje i Tom DeMarco [3], který prohlásil, že: "*Kvalita produktu se odvíjí od toho, jak moc změní svět k lepšímu.*"

Není pochyb, že definitivní definice kvality softwaru by mohla být diskutována zřejmě donekonečna. Ať už si však kterýkoliv čtenář této práce vybere libovolnou definici, je z pohledu kvality potřeba zdůraznit čtyři důležité body:

1. Požadavky na software jsou základ, z něhož vychází kvalita softwaru. Jestliže není dosaženo dostatečné shody s požadavky, nemůže být docíleno dostatečné kvality výsledného produktu.
2. Standardy definují sadu vývojových kritérií, jimiž se řídí způsob, jakým je navržen software. Jakmile nejsou tyto kritéria dodržena, dojde velmi pravděpodobně k poklesu kvality.
3. Zákazníci často vysloví jen takzvané explicitní požadavky (např. přihlášení do portálu či zobrazení detailu položky), ale neuvedou požadavky implicitní (např. snadná ovladatelnost, dobrá udržovatelnost). Pokud software splňuje pouze explicitní požadavky, ale nesplňuje implicitní požadavky, pak je kvalita softwaru „podezřelá“ [1].
4. Pokud softwarový produkt poskytuje prospěch jeho koncovým uživatelům, mohou být ochotni tolerovat občasné výpadky spolehlivosti nebo výkonnostní problémy.

Kvalita softwaru je složitý mix faktorů, které se budou lišit v závislosti na aplikaci a konkrétních zákaznících, pro které bude software vyráběn. Různé faktory kvality softwaru a lidské činnosti potřebné k dosažení těchto faktorů budou uvedeny v kapitole 2.6, jež se zabývá metrikami.

2.2 Standardy kvality ISO

Základními normami pro zajištění kvality poskytované mezinárodní organizací pro standardizaci (International Organization for Standardization, dále jen ISO) jsou normy ISO 9000. Jak je uvedeno na stránkách ISO [4], normy ISO 9000 se zabývají různými aspekty řízení kvality a obsahují některé z neznámějších standardů této organizace. Normy poskytují pokyny a nástroje pro firmy a organizace, které chtějí, aby jejich produkty a služby důsledně plnili požadavky zákazníka, a pomáhají jim trvale zlepšovat kvalitu jejich výrobků. Popisují prvky zajišťování kvality, které lze použít pro jakýkoliv druh podnikání, bez ohledu na zaměření produktů či služeb. Nicméně, ISO 9000 nepopisuje, jak by organizace měly tyto pokyny a nástroje zavádět. Jak zavedení docílit záleží na samotné organizaci.

Tyto normy byly přijaty v mnoha zemích Evropy, v Kanadě, Mexiku, Spojených státech amerických, Austrálii, Novém Zélandu a zájem o tyto normy byl projeven také v Tichomoří a zemích Latinské a Jižní Ameriky.

Chce-li se společnost registrovat do některého z kvalitních modelů systému zabezpečování kvality uvedených v ISO 9000 (například do modelu ISO 9126, kterému bude věnována pozornost v sekci 2.6.2), musí být přezkoumána auditory třetích stran, kteří zjistí, zda společnost funguje v souladu s požadovaným standardem. Pokud je

registrace úspěšná, obdrží společnost certifikát a auditor v pravidelných pololetních intervalech kontroluje, zda společnost dodržuje požadavky daného standardu [1].

ISO 9001 je norma určená pro univerzální zabezpečování kvality, která se vztahuje k různým inženýrským oborům. Jelikož má vývoj softwarových produktů svá specifika, která nejsou v této normě zohledněna, byla vyvinuta snaha, která vedla k vytvoření speciální normy ISO 9000-3, jež je uzpůsobena pro vývoj softwarových produktů. Tato speciální norma je založena na 8 principech, mezi nimiž je mimo jiné uvedeno změření na zákazníka. Původní norma ISO-9003 (ISO 1997), která obsahovala celkem 20 požadavků, byla ve verzi z roku 2001 rozšířena na 22 požadavků, přičemž požadavky byly rozčleněny celkem do pěti skupin [2] :

- Management kvality
- Odpovědnosti managementu
- Řízení zdrojů
- Realizace produktu
- Management, analýza a zlepšování

2.3 Přístupy k vývoji software

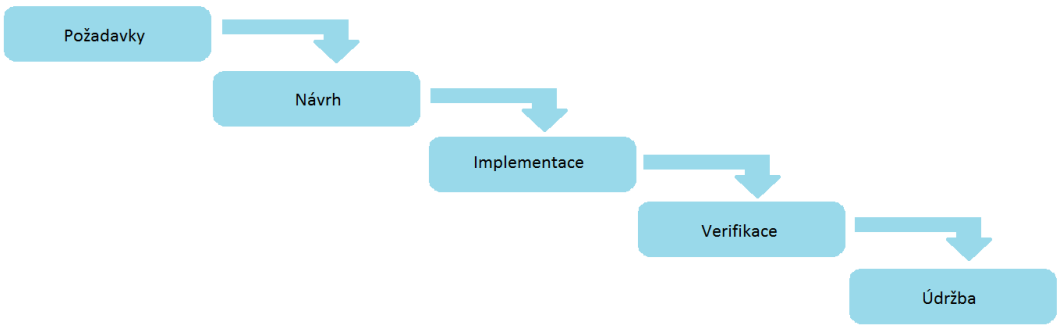
Přístupy k vývoji softwaru jsou rozmanité a přesahují rámec této diplomové práce, nicméně pro částečnou představu o jejich vlivu na kvalitu bude v krátkosti popsán vodopádový a agilní/iterativní přístup.

2.3.1 Vodopádový přístup

Takzvaný "vodopádový přístup" je v podstatě pouze jiným názvem pro tradiční přístup k vývoji softwaru. Tento přístup byl používán zejména v počátcích vývoje softwaru, ale v dnešní době se od něj ustupuje a je nahrazován iterativním vývojem. Vodopádový model je rozdělen do pěti fází, přičemž jedna navazuje na druhou, jak ukazuje obrázek 2.1. Každá fáze je vždy dokončena předtím, než se začne s další fází. Platí tedy, že u vodopádového přístupu, bude jen zřídka docházet ke snaze znovu otevřít fáze, které již byly dokončeny. Z tohoto důvodu se často projekty, jež se řídí vodopádovým přístupem, plánují pomocí Ganttova diagramu[1] [6].

Vodopádový přístup se řídí heslem, že vše je uděláno napoprvé! Tento přístup je ovšem velmi riskantní, často nákladný a obecně méně efektivní než agilnější přístupy. Hlavní zápory tohoto přístupu spočívají především v tom, že nelze vidět průběžný postup produktu, jelikož produkt je předán jako celek až na samém konci vývoje a že testování je ponecháno až do fáze, kdy je hotová veškerá implementace. Především z těchto důvodů byl tento přístup upraven tak, aby se přiblížil agilnímu přístupu a byl nazván iterativním vodopádovým přístupem. Více viz [5], [6].

[1] Ganttův diagram (Gantt chart) je mimo jiné využíván k zobrazení časové náročnosti a posloupnosti jednotlivých částí projektu (úkolů). Je používán pro lepší plánování a získávání lepších časových odhadů dokončení projektu.



Obrázek 2.1: Vodopádový model

2.3.2 Iterativní přístup

Iterativní přístup má, stejně jako vodopádový, základní fáze, kterými prochází. Rozdíl je ovšem v tom, že jednotlivé fáze nejsou zaměřeny výhradně na jednu činnost, jak tomu bylo v případě vodopádového přístupu, ale v každé etapě probíhá více činností najednou, což umožňuje začít s vývojem a testováním v dřívější fázi vývojového cyklu.

Iterativní přístup zavádí agilní metodiky, mezi které patří například Scrum, XP (extrémní programování) či adaptivní vývoj. Ty jsou populární zejména díky tomu, že omezují dokumentaci a další organizační záležitosti a více se soustředí na flexibilitu vývoje. Větší důraz je kladen na výrobu fungujícího softwaru, spíše než na vytváření rozsáhlé a komplexní dokumentace. Důležité je, že agilní vývoj umožňuje reagovat na změny, které nastanou v průběhu procesu. To ovšem neznamená, že plány a dokumentace nejsou důležité, pouze nejsou tak důležité jako samotné zajištění, že software bude fungovat nebo že se plán vývoje dokáže přizpůsobit měnícím se potřebám.

V předchozí části však nebyl zmíněn ještě jeden přístup, který se nazývá Rational Unified Process (RUP). Ten se od předchozích přístupů liší tím, že se o něco více zaměřuje na podporu dokumentace. Jak je uvedeno v [5], vždy záleží především na potřebách daného projektu.

2.3.3 Vliv na kvalitu

Iterativní postup se ukázal jako lepší přístup než vodopádový z důvodů uvedených v následujících odstavcích [5].

Iterativní přístup vychází vstříc měnícím se požadavkům. Změny požadavků či přidávání funkcí, které jsou technologicky nebo zákaznicky zaměřeny, patřily vždy mezi hlavní zdroj problémů. Často vedly k pozdnímu dodání produktu, nedodržení plánů, nespokojenosti zákazníků či velké zátěži vývojářů. Iterativní vývoj se zaměřuje na výrobu a předvedení spustitelného softwaru již po několika týdnech (v závislosti na velikosti projektu), což vede k nutnosti zaměřit se především na základní požadavky.

Integrace neobnáší "velký třesk" na konci projektu. Problémem integrace provedené na konci vývoje může být značná časová náročnost v případě, že je nutné něco přepracovat (dle [5] může toto přepracování stát až 40 procent z celkové úsilí vynaloženého na projekt). Aby se tomuto možnému problému zabránilo, přichází iterativní přístup s rozdělením vývoje na menší iterace[2], přičemž každá z nich končí integrací, při které jsou jednotlivé části softwaru integrovány postupně a plynule, čímž se minimalizuje pozdější přepracování.

Rizika jsou obvykle objevena nebo řešena už během prvních integrací. Výhodou navíc je, že všechny komponenty mohou být průběžně testovány, což vede k dalšímu zmírnění rizik. Vzhledem k tomu, lze rychle zjistit, zda se předpokládaná rizika ukážou jako oprávněná, a rovněž je možné odhalit nová, nepředvídaná rizika v době, kdy je možné uplatnit jednodušší a méně nákladné řešení.

Vedení má k dispozici prostředky, díky nimž může prosazovat změny ve vyvíjeném produktu. Iterativní vývoj přispívá k rychlému vytvoření spustitelné aplikace (i když s omezenými funkcemi), která může být použita pro rychlé uvolnění výrobku, což může poskytnout výhodu oproti konkurenci.

Usnadňuje opakované využití komponent (částí softwaru), jelikož je jednodušší identifikovat společné části, které jsou částečně určeny nebo realizovány v iteracích, než je rozpoznat již v době plánování. Díky tomu, mohou architekti odhalit potenciální možnosti pro opětovné použití částí kódu a ty poté mohou rozvíjet v následujících iteracích.

Umožňuje lepší využití lidí pracujících na projektu. Při vodopádovém vývoji docházelo k tomu, že analytici zaslali kompletní požadavky designérům, kteří předali kompletní design programátorům. Ti pak odeslali jednotlivé komponenty integrátorům, kteří je nakonec poslali systémovým testerům. Tento přístup však byl často zdrojem chyb a nedorozumění a také přispíval k tomu, že se lidé cítili méně zodpovědní za konečný produkt. Naopak při iterativním vývoji mají jednotliví členové týmu možnost rozšiřovat rozsah svých odborných znalostí, což jim umožňuje zastávat více rolí a umožňuje vést projekt s lepším využitím dostupného personálu.

Členové týmu se učí v průběhu vývoje. Vzhledem k tomu, že vývoj probíhá v iteracích, mají členové týmu možnost poučit se ze svých chyb a s novými iteracemi mohou zlepšovat své dovednosti. Naproti tomu ve vodopádovém přístupu, je vždy jen jeden pokus na návrh, kódování či testování.

Vývojový proces se průběžně vylepšuje. Na konci každé iterace je posouzeno nejenom, zda stav projektu odpovídá plánu, ale také se analyzuje, co a jak může být do další iterace zlepšeno.

[2] Iterace jsou součástí každého iterativního vývoje, přičemž v každé fázi se vždy musí uskutečnit alespoň jedna iterace. Délka jedné iterace se nejčastěji pohybuje v rozmezí 2-3 týdnů.

2.4 Formální technické revize

Formální technické revize, neboli též formální technický přezkum (z anglického Formal Technical Review), dále jen FTR, jsou jedním ze způsobů, jak zajistit kvalitu softwaru. FTR je prováděno převážně softwarovými inženýry, ale mohou se na něm podílet i další lidé. Dle [1] jsou cíle FTR:

1. Odhalit chyby ve funkcích, logice, nebo implementaci pro libovolný typ softwaru.
2. Ověřit, že software, který je předmětem přezkoumání, odpovídá požadavkům.
3. Zajistit, že vývoj softwaru se řídil podle předem stanovených norem.
4. Dosáhnout softwarového produktu, který je vyvinut jednotným způsobem.
5. Zajistit, aby projekty byly snáze zvládnutelné.

Kromě toho FTR přináší možnost seznámení se se softwarem lidem, kteří by jinak nemohli software vidět. Každé FTR probíhá jako schůzka (meeting) a může být úspěšné pouze tehdy, pokud je řádně naplánováno, kontrolováno, a navštěvováno.

2.4.1 FTR meeting

Bez ohledu na formát FTR, by se mělo každé jednání řídit následujícími omezeními:

- Počet lidí, kteří se jej účastní, by měl být typicky mezi třemi až pěti lidmi.
- Každý člen by se měl před jednáním připravit, přičemž příprava by neměla žádnému členovi trvat více než dvě hodiny.
- Doba trvání jednání by měla být kratší než dvě hodiny.

Vzhledem k uvedeným omezením, by se mělo FTR zaměřovat vždy na konkrétní část softwaru. Zaměření na konkrétní část navíc přispívá k vyšší pravděpodobnosti odhalování chyb. Postup FTR meetingu a příprav je opět nad rámec této diplomové práce, ale je k dispozici v [1]. Každý meeting je zakončen hodnocením, ze kterého musí vzejít rozhodnutí, zda je možné přijmout produkt bez dalších úprav, je nutné odmítnout produkt v důsledku závažné chyby, kterou je nutné opravit a poté provést další kontrolu, nebo je možné produkt přijmout dočasně s tím, že nalezené chyby budou opraveny [7].

2.4.2 Zaznamenávání průběhu

Nedílnou součástí každého meetingu je zaznamenávání uvedených problémů recenzentem[3]. Zaznamenané problémy jsou na konci celého meetingu shrnuty a je vypracován seznam s přehledem těchto problémů, který je doplněn o souhrnnou zprávu z tohoto meetingu. Seznam s přehledem problémů pak slouží jednak k identifikaci problémových oblastí v rámci softwarového produktu, ale také jako kontrolní seznam, který poskytuje přehled o tom, jaké opravy byly v projektu provedeny [1].

[3] Osoba, jež je pověřena zaznamenáváním důležitých údajů během FTR meetingu

Aby vše vedlo ke zvýšení kvality softwarového produktu, je důležité vytvořit také navazující postup, který zajistí, že problémy uvedené v seznamu budou řádně opraveny. Tento úkol může být přiřazen například vedoucímu FTR meetingu.

2.5 Statistické zabezpečení kvality

Statistické zabezpečování kvality softwaru, odráží rostoucí trend, jež má za cíl pohlížet na kvalitu kvantitativněji. V případě softwaru, zahrnuje statistické zabezpečování kvality následující kroky [7]:

1. Shromažďovat a třídit informace o defektech v softwaru.
2. Sledovat základní příčinu, která byla zdrojem závady (např. neshody se specifikací, konstrukční chyby, porušení norem či špatná komunikace se zákazníkem).
3. Na základě principu Pareto[4] izolovat 20 procent příčin všech chyb.
4. Jakmile byly zjištěny příčiny, zaměřit se na jejich odstranění.

Dodržování těchto kroků je cestou k celkovému zlepšení kvality výsledných produktů, přičemž je odstraňována již příčina problémů a nikoliv až následek. Přesto, že mohou být nalezeny stovky různých chyb, může ke všem chybám vést jedna či více z následujících příčin [7]:

- neúplné či chybné údaje (IES)
- dezinterpretace v komunikaci se zákazníkem (MCC)
- záměrné odchylky od specifikací (IDS)
- porušení programovacích standardů (VPS)
- chyba v reprezentaci dat (EDR)
- nekonzistentní rozhraní (ICI)
- chyba v návrhu logiky (EDL)
- neúplné nebo chybné testování (IET)
- nepřesné či neúplné dokumentace (IID)
- chyba při překladu z návrhu do programovacích jazyků (PLT)
- nejasné nebo v proměnlivé rozhraní člověk/počítač (HCI)
- různé (MIS)

Při odhalování příčin, jež vedou k objeveným chybám, je vhodné vytvářet tabulku, která zachycuje počet chyb způsobených danou příčinnou a jejich procentuální vyjádření. Pokud k chybám vede více příčin, je vhodné postupovat postupně a začít nejdříve s nápravou těch, které způsobují nejvíce chyb. Teprve po opravení jedné nebo více příčin je vhodné vrátit se ke zbylým a začít také s jejich nápravou.

Pro každý důležitý krok provedený v softwarovém procesu je možné vypočítat takzvaný chybový index (error index). Po dokončení jednotlivých aktivit (analýzy,

[4] Pareto princip znamená, že 80 procent všech chyb, které se vyskytnou v průběhu testování, bude mít pravděpodobně návaznost na 20 procent všech příčin.

návrhu, kódování, testování a uvolnění produktu k používání), jsou shromažďovány následující údaje [7]:

E_i = celkový počet chyb, zjištěných během i-tého kroku v softwarovém procesu

S_i = počet závažných chyb

M_i = počet středně vážných chyb

T_i = počet drobných chyb

P_S = velikost výrobku (například počet stran dokumentace) v i-tém kroku

w_s, w_m, w_t = váhové faktory pro závažné, středně vážné a triviální chyby, kde doporučené hodnoty jsou $w_s = 10$, $w_m = 3$, $w_t = 1$. Váhové faktory pro jednotlivé fáze by s probíhajícím vývojem měli růst, aby bylo odměněno brzké nalezení chyby.

Dále je v každém kroku softwarového procesu vypočítán fázový index PI_i, jako:

$$PI_i = w_s\,(S_i / E_i) + w_m\,(M_i / E_i) + w_t\,(T_i / E_i)$$

Chybový index je počítán na základě výpočtu kumulativního účinku každého PI_i, přičemž váha chyby zjištěné později v procesu softwarového inženýrství je větší než váha chyby zjištěné dříve:

$$EI = \sum (i \times PIi) / PS = (PI1 + 2PI_2 + 3PI_3 + \ldots iPI_i) / PS$$

Přesto, že může být užitečné znát výpočty a teorii o statistickém SQA a Pareto principu, nejdůležitější je pochopit základní myšlenku těchto technik, který lze vyjádřit následující větou. Při použití statického SQA je nutné zaměřit se na věci, které jsou skutečně důležité, ale nejdříve je třeba pochopit, co je opravdu důležité.

2.6 Metriky

Pokud má být zajištěna kvalita výsledného produktu, je třeba soustředit pozornost na celou řadu činností souvisejících s různými aktivitami prováděnými během vývojového cyklu. Kvality produktu může být dosaženo prostřednictvím dobré analýzy a návrhu, implementací vhodného zdrojového kódu a testování, ale stejně tak může být zajištěna zavedením FTR, vícevrstvými testovacími strategiemi, či uplatňováním přijatých norem softwarového inženýrství. Kromě toho však může být kvalita definována prostřednictvím široké škály kvalitativních faktorů a měřena s použitím různých indexů a metrik, které budou popsány (pouze částečně protože kompletní popis metrik přesahuje rámec této práce) v následujících odstavcích.

2.6.1 McCallovy faktory kvality

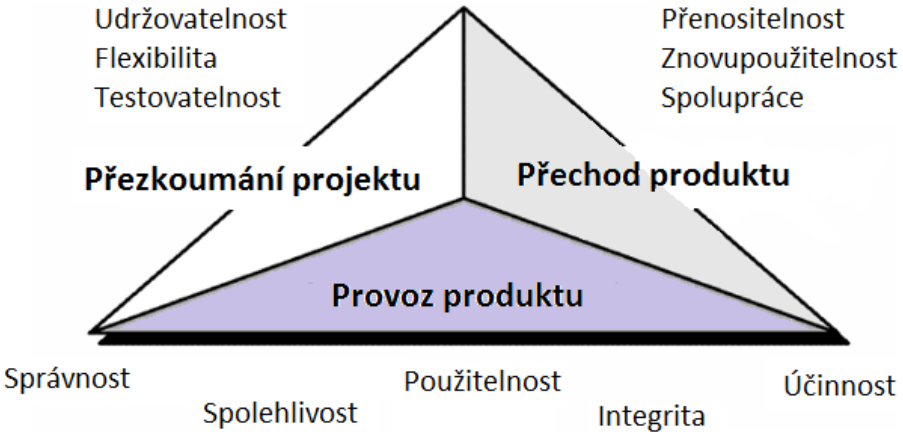
Faktory, které ovlivňují kvalitu softwaru lze dle [1] rozdělit do dvou velkých skupin:

1. faktory, které mohou být měřeny přímo (např. účinnost)

2. faktory, které mohou být měřeny pouze nepřímo (např. použitelnost a udržovatelnost).

Při zjišťování kvality prostřednictvím faktorů kvality je vždy nutné provést měření, aby bylo možné porovnat softwarový produkt (dokumenty, programy, data) oproti vztažným hodnotám a určit tak kvalitu produktu. Jak uvádí [1] McCall, Richards a Walters navrhují roztřídění faktorů, které ovlivňují kvalitu softwaru. Tyto faktory kvality software jsou znázorněny na obrázku 2.2 a zaměřují se na tři důležité aspekty softwarového produktu: jeho provozní vlastnosti, schopnost provést změnu a jeho přizpůsobivost novému prostředí. Navíc zavádějí další pojmy, které jsou taktéž znázorněny na obrázku 2.2. Jedná se o:

- Správnost - do jaké míry program vyhovuje specifikaci a splňuje cíle zákazníka.
- Spolehlivost - do jaké míry lze očekávat, že program plní svou zamýšlenou funkci s požadovanou přesností.
- Účinnost - množství výpočetních zdrojů a kódu požadovaného programem, aby plnit svou funkci.
- Integrita - míra, do jaké je umožněn neoprávněným osobám přístup k softwaru nebo datům.
- Použitelnost - úsilí, které je nutné vynaložit k tomu, aby se člověk s programem naučil, používal jej, připravoval vstup a interpretoval výstup programu.
- Udržovatelnost - úsilí nutné k nalezení a opravení chyby v programu.
- Flexibilita - úsilí nutné k úpravě programu.
- Testovatelnost - úsilí potřebné k testování programu, abychom se ujistili, zda program plní svou funkci.
- Přenositelnost - úsilí potřebné pro přenos programu z jednoho hardwarového nebo softwarového prostředí do druhého.
- Znovupoužitelnost - do jaké míry může být program (nebo části programu) znovu použit v jiných aplikacích.
- Spolupráce - úsilí potřebné pro propojení jednoho systému s druhým.



Obrázek 2.2: McCallovy faktory kvality softwaru

Uvedené kvalitativní faktory je však obtížné, a v některých případech dokonce nemožné, měřit přímo. Z tohoto důvodu, je definován a použit soubor ukazatelů k vytvoření výrazu pro každý z faktorů, podle následujícího vztahu [1]:

$$Fq = c_1 \times m_1 + c_2 \times m_2 + \ldots + c_n \times m_n$$

,kde F_q je softwarový činitel kvality, c_n jsou regresní koeficienty a m_n jsou metriky, které mají vliv na kvalitu faktoru. Nevýhodou těchto metrik je ovšem to, že je lze měřit pouze subjektivně. Nicméně tyto metriky lze zavádět alespoň formou kontrolního seznamu, který se používá k ohodnocení specifického atributu softwaru, přičemž se toto hodnocení pohybuje v rozmezí od 0 (nízký) do 10 (vysoký). Příkladem metrik používajících tento klasifikační systém jsou například *Úplnost* (do jaké míry bylo dosaženo úplné provedení požadované funkce), *Stručnost* (kompaktnost programu ve smyslu řádků kódu), *Konzistence* (použití jednotného designu a dokumentačních technik v průběhu vývoje softwaru) či *Hardwarová nezávislost* (míra, do jaké je software oddělen od hardware, na kterém je provozován). Definováno je ještě několik dalších metrik, které jsou k dispozici v [1]. Vztah mezi těmito metrikami a faktory kvality softwarových produktů je pak uveden v tabulce 2.1.

	Faktory kvality										
Metriky pro softwarovou kvalitu	Správnost	Spolehlivost	Účinnost	Integrita	Udržovatelnost	Flexibilita	Testovatelnost	Přenositelnost	Znovupoužitelnost	Spolupráce	Použitelnost
Možnost provést audit				x			x				
Přesnost		x									
Standardizace komunikace										x	
Úplnost	x										
Složitost		x				x	x				
Stručnost			x		x	x					
Konzistence	x	x			x	x					
Standardizace dat										x	
Tolerance chyb		x									
Účinnost provedení			x								
Rozšiřitelnost						x					
Obecnost						x		x	x	x	
Hardwarová nezávislost								x	x		
Přístrojové vybavení				x	x		x				
Modularita		x			x	x	x	x	x	x	
Provozuschopnost			x								x
Zabezpečení				x							
Samodokumentovatelnost					x	x	x	x	x		
Jednoduchost		x			x	x	x				
Softwarová nezávislost								x	x		
Sledovatelnost	x										
Školení											x

Tabulka 2.1: Vztah mezi kvalitativními faktory softwaru a metrikami

2.6.2 FURPS

Kvalitativní faktory popsané v předchozí sekci představují jen jeden z několika navrhovaných kontrolních seznamů pro zajištění kvality softwaru. Další kontrolní seznam je označován zkratkou FURPS[5] a byl vytvořen firmou Hewlett-Packard. Kvalitativní faktory uvedené ve FURPS vycházejí z dřívějších prací a definují následující atributy pro každý z pěti hlavních faktorů [1]:

- Funkčnost - posuzuje se na základě vyhodnocení sady funkcí a schopností programu, obecnosti funkcí, a bezpečnosti celého systému.
- Použitelnost - hodnotí se s ohledem na lidský faktor, celkovou estetiku, soudržnost a dokumentaci.

[5] FURPS je zkratka z anglických slov functionality, usability, reliability, performance, a supportability, které se do češtiny překládají jako funkčnost, použitelnost, spolehlivost, výkon a schopnost podpory

- Spolehlivost – je hodnocena na základě frekvence a závažnosti poruch, přesnosti výstupních výsledků, střední doby do poruchy (MTTF), schopnosti zotavit se z chyby a předvídatelnosti programu.
- Výkonnost – měří se na základě rychlosti zpracování, doby odezvy, množství spotřebovaných zdrojů, propustnosti a účinnosti.
- Schopnost podpory - kombinuje možnost rozšířit program (rozšiřitelnost), přizpůsobivost, snadnou údržbu (tyto tři atributy představují nejzákladnější množinu) a udržovatelnost. Kromě toho však ještě zahrnuje testovatelnost, kompatibilitu, konfigurovatelnost (schopnost organizovat a řídit prvky softwarové konfigurace), snadnost, s jakou může být systém instalován, a snadnost, s jakou mohou být lokalizovány problémy.

2.6.3 ISO standard pro kvalitativní faktory

ISO 9126 standard byl vyvinut ve snaze identifikovat klíčové atributy kvality pro počítačový software a určuje šest klíčových atributů kvality [8]:

- Funkčnost - míra, do jaké je software schopen plnit potřeby. Skládá se z vhodnosti, přesnosti, schopnosti systémů vzájemně spolupracovat, dodržování předpisů a zabezpečení.
- Spolehlivost - množství času, po který je software k dispozici pro použití. Spolehlivost obsahuje následující dílčí atributy: úroveň zralosti, odolnost proti chybám a schopnost zotavit se.
- Použitelnost - do jaké míry lze software použít. Použitelnost obsahuje následující dílčí atributy: srozumitelnost, provozuschopnost, jak snadné/obtížné je naučit se používat software a přitažlivost pro uživatele.
- Účinnost - míra, do jaké software umožňuje optimální využití systémových zdrojů. Účinnost je složena z následující dílčích atributů: chování v čase a využití zdrojů.
- Udržovatelnost - snadnost, s jakou mohou být provedeny opravy v softwaru. Zahrnuje následující dílčí atributy: analyzovatelnost, schopnost adaptovat se na změnu, stabilitu a testovatelnost.
- Přenositelnost - snadnost, s jakou software může být převeden z jednoho prostředí do druhého. Obsahuje následující dílčí atributy: přizpůsobivost, možnost instalace, schopnost fungovat společně s ostatními systémy a zaměnitelnost.

2.7 Spolehlivost

Dalším důležitým prvkem z hlediska kvality softwarového produktu je spolehlivost. Pokud program opakovaně a často neprovádí co má, nezáleží na tom, zda jsou jiné faktory kvality softwaru přijatelné. Spolehlivost softwaru má tu výhodu, že na rozdíl od mnoha jiných faktorů kvality je možné ji měřit a odhadnout na základě historických a vývojových údajů.

Z hlediska spolehlivosti je pro uživatele produktu důležitá zejména jeho dostatečná dostupnost, alespoň taková jaká byla garantována výrobcem, a nedocházelo k selhání produktu. Selhání může být jen drobnou obtíží, ale také může mít katastrofické následky. Některé selhání lze opravit během několika sekund, zatímco jiné může být opravováno týdny či měsíce. Navíc hrozí, že oprava jednoho selhání může vést k vzniku chyby, jež v konečném důsledku povede ke vzniku jiného selhání.

Matematické výpočty spolehlivosti softwaru zpočátku vycházeli z hardwarových výpočtů spolehlivosti. Většina z nich však selhala, neboť výpočty hardwarové spolehlivosti vycházejí ze skutečnosti, že selhání v důsledku opotřebení je pravděpodobnější než selhání v důsledku konstrukční vady. Problémem je, že v případě softwaru je tomu přesně naopak a opotřebení nehraje žádnou roli.

Pro jednoduché měření spolehlivosti lze použít střední dobu mezi výpadky (MTBF), definovanou vztahem [9]:

$$MTBF = MTTF + MTTR$$

,kde zkratky MTTF a MTTR jsou střední doba do poruchy respektive střední doba do opravy.

Pro koncového uživatele má však spíše než střední doba mezi výpadky větší význam dostupnost softwaru. Dostupnost softwaru můžeme definovat jako pravděpodobnost, že program je v daném okamžiku v provozu tak, jak je uvedeno ve specifikovaných požadavcích. Dostupnost pak lze dle [9] definovat následovně:

$$Dostupnost = MTTF \,/\, (MTTF + MTTR)$$

Jak je patrné ze vztahů pro spolehlivost (MTBF) a dostupnost, na spolehlivost mají oba faktory (MTTF a MTTR) stejný vliv, kdežto na dostupnost má mnohem větší vliv MTTR.

2.8 Zajištění kvality her

Hry[6] jsou specifickým softwarovým produktem, který má zvláštní vlastnosti, jež se nevyskytují u žádného jiného typu softwaru. To činní zajišťování kvality her komplikovanější, jelikož je třeba se kromě klasických problémů zaměřit na celou řadu dalších aspektů souvisejících s návrhem, jež kvalitu her výrazně ovlivňují a určují, zda ji zamýšlené cílové publikum bude mít v oblibě. Toto téma je však obšírné a proto bude jen částečně popsáno v kapitolách 2.8.1 až 2.8.3. Podrobnější informace o této problematice je však možné najít v [10] a [11]. Vzhledem ke složitosti většiny netriviálních her navíc platí, že jejich vývoj musí být iterativní. Jinými slovy není

[6] Termín hra není jednoduché definovat, ačkoliv každý člověk má představu o tom, co hra je. Nicméně vznikly různé pokusy o její definici a některé z těchto definic jsou uvedeny v [8].

vhodné používat vodopádový model, ale spíše spirálový[7] či iterativní. Navíc u her více než jinde platí, že čím více testujeme a zdokonalujeme návrh, tím lepší hra bude.

Podstatné je uvědomit si, že hra je určena pro hráče, proto musíme zjistit, co hráči chtějí a co se jim líbí. To však může být složité, jelikož ani hráči sami někdy nevědí, co vlastně chtějí, natož aby to uměli formulovat. Jednou z metod, kterou lze od hráče tuto informaci získat je tzv. playtesting (je popsán v kapitole 3.11). Aby mohl být platesting prováděn co nejdříve a přispět maximální možnou měrou k zvýšení kvality, je nezbytné vytvářet brzké prototypy hry. Tím lze nalézt včas hlavní problémy ve hře a napravit nedostatky spojené s návrhem již v raných fázích vývoje, díky čemuž lze ušetřit značné množství peněz a času. Nicméně v první řadě musíme vědět, kdo by hru měl hrát. Zde jsou rozhodujícími faktory především věk (dělení do kategorií věku je opět v [10]) a pohlaví. Není důležité mít přesný seznam preferencí jednotlivých pohlaví, ale uvědomit si, že mezi oběma pohlavími existují rozdíly, a pečlivě zvážit, zda hra je opravdu pro toho, pro koho ji navrhujeme. Například muži mají rádi zvládání věcí, soutěživost, ničení nebo prostorové hlavolamy a preferují „metodu“ pokusu a omylu (nepoužívají žádné návody). Naopak jak uvádí Dangelmaier [12]: „*Ženy chtějí zážitky, které přinášejí emoce a sociální objevy, které mohou zažít ve svém životě.*“ Proto ve hrách ženy vyhledávají spíše emoce, reálný svět, možnost pečovat o něco nebo někoho, dialog a slovní hádanky a tíhnou k učení příkladem (drží se návodů). Seznam preferencí obou pohlaví je detailněji popsán v [10] a prvky vyhledávané ženami v [12].

2.8.1 Funkčnost

Funkčnost je jedním ze základních aspektů, na který by během vývoje hry měl být kladen důraz. Mezi další dva pak patří vnitřní kompletnost (kapitola 2.8.2) a vyváženost (2.8.3).

Co je ale vlastně míněno pojmem funkčnost? V tomto případě jde o to, že hra je v takovém stádiu, že někdo, kdo neví hře o nic, může hru hrát. Funkčnost tedy neznamená, že tester nenašel žádnou chybu, nebo že zážitek ze hry bude dostatečně uspokojující, ale znamená to, že hráč může hrát hru bez cizí pomoci, jinými slovy, že hráči mohou pomocí ovládacích prvků ovládat hru a postupovat dále ve hře [11].

2.8.2 Kompletnost

Pokud je již hra funkční (viz 2.8.1) lze posoudit, zda je rovněž vnitřně kompletní (úplná), neboli určit, zda hra neumožňuje získat hráči neoprávněnou výhodu a že ve všech částech hry má hráč k dispozici vše, co potřebuje, a nemůže nastat případ, kdy nemůže (ne vlastní vinnou) postupovat dále ve hře. Pokud dojde k tomu, že hráči mohou využít některé části hry k získání nespravedlivé nebo nezamýšlené výhody, definujeme tuto vadu jako mezeru. Dokud existují nezamýšlené mezery, nelze hru považovat za úplnou. Někdy je však sporné určit, zda se skutečně jedná o mezeru, nebo zda jde o legitimní způsob hry. [11] uvádí příklad, kdy v některých MMORPG[8]

[7] Spirálový model je jedním z modelů simulujících životní cyklus vývoje softwaru pomocí spirály, přičemž vývoj probíhá od středu spirály směrem ven. Více například v [6]

[8] MMORPG - Massive Multiplayer Online Role Play Game neboli online hry na hrdiny pro více hráčů

figurovali hráči, kteří zabíjeli ostatní, a byli nazýváni jako zabijáci. Od ostatních hráčů proto vzešla iniciativa "zakázat" tyto postavy. V důsledku toho ovšem hra přišla o možnost svobodného rozhodnutí hráče zvolit libovolnou roli ve hře, a tím i o určité "bohatství". Z tohoto důvodu dnes mnoho MMORPGs nabízí dvě varianty. Jednu, kde může hráč ubližovat ostatním hráčům a jednu, kde nemůže, přičemž obě tyto varianty jsou vnitřně kompletní. Druhým problematickým bodem z hlediska kompletnosti jsou takzvané slepé uličky, které na rozdíl od mezer neumožňují hráčům využívat hru k neoprávněným výhodám, ale stejně jako mezery, musí být odhaleny před tím, než může být hra považována za vnitřně kompletní. Slepá ulička nastane, když se hráč dostane do situace, kdy nemůže pokračovat směrem ke splnění herního cíle bez ohledu na to, co dělá. Tímto typem vad se vyznačují především dobrodružné hry, kde hráči musejí sbírat předměty ve světě a později použít tyto objekty k vyřešení hádanek. Pokud hráč nemůže vyřešit hádanku, protože mu chybí nějaký díl (předmět), dospěje hra do slepé uličky. Slepé uličky můžeme nalézt ale také v jiných typech her [11].

Obecně lze myšlenku úplnosti shrnout do tohoto prohlášení: Vnitřně kompletní hra je ta, ve které hráči mohou ovládat hru, aniž by v jakémkoliv bodě byla ohrožena hratelnost nebo funkčnost. Ve skutečnosti žádná hra není nikdy úplná a vždy existuje prostor pro zlepšení. Podrobněji se problematice vnitřní kompletnosti věnuje [11].

Rozpoznat, zda je hra úplná, lze v některých případech jen obtížně. Proto je i v tomto případě k nalezení neúplnosti ve hře využíván playtesting, jelikož hráči jsou mnohem kreativnější a vynalézavější, a z tohoto důvodu mohou lépe odhalit neúplnost hry.

2.8.3 Vyváženost

Při hraní her se hráč může dostat do situace, kdy se těší na hru, od které očekává, že bude neuvěřitelně zábavná, jelikož hra má všechny předpoklady pro to, aby hráče zaujala, ale ve výsledku jej hra nebaví. Často je v těchto případech možnou příčinnou nesprávné vyvážení hry. Vyvážení hry spočívá v upravování prvků hry, dokud hra neposkytuje zážitek, který návrháři při tvorbě hry očekávali. Svým způsobem lze vyvažování hry přirovnat k vaření, při kterém kuchař upravuje poměr ingrediencí, dokud jídlo nechutná podle jeho představ. Při vyvažování hry jde v podstatě o totéž, jen jsou upravovány poměry mezi jednotlivými prvky hry. Cílem je docílit, že ve hrách pro více hráčů jsou výchozí pozice a hra spravedlivé (tzn., žádný z hráčů není zvýhodněn), a žádná jednotlivá strategie nedominuje nad ostatními. V hrách pro jednoho hráče je cílem zajistit, že úroveň dovedností je správně nastavená na cílovou skupinu hráčů.

Z hlediska zajištění kvality je podstatné, že vyvážení hry může začít, jakmile je hra hratelná. Z [10] vyplývá, že v minulosti skončilo mnoho her neúspěchem, protože byla testována pouze funkčnost hry, ale nezbyl čas pro její vyvážení před uvedením na trh. Na vyvážení hry je nutné ponechat dostatek času, jelikož otázka rovnováhy je jednou z nejobtížnějších částí návrhu hry. Navíc některá rozhodnutí učiněná během vyvažování mohou být založena na matematice či statistice, kdežto jiná na základě osobního vkusu, protože vyvažování je jak o číslech, tak o instinktu.

Jak je uvedeno v [10] existuje 12 nejčastějších druhů vyvažování, které jsou uvedené v následujících odstavcích.

Prvním druhem vyvažování je spravedlnost, jež je hráči ve hrách očekávána. Pokud by hráči měli pocit, že oponent, či počítač mají oproti nim výhodu, mohli by hru přestat hrát. Nejjednodušším způsobem jak dosáhnout spravedlnosti je vytvoření symetrické hry, která dává všem stejné zdroje a pravomoci. Tento způsob ale nelze aplikovat na všechny hry, jelikož některé hry ze své podstaty nemohou být symetrické. Například hry, v nichž jsou simulovány válečné boje, by v případě symetrického nastavení byly pro hráče nudné. Z tohoto důvodu tíhne značné množství her k asymetričnosti.

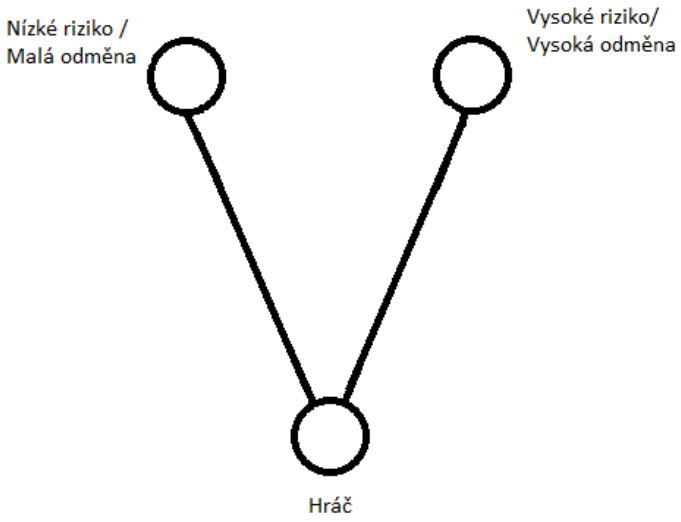
Výzva a úspěch stojí při vyvažování hry proti sobě. Pokud je hra příliš náročná, hráč ztrácí motivaci ve hře pokračovat, a pokud hráč uspěje příliš snadno, může se nudit. Cílem je udržet hráče uprostřed, což znamená udržet zážitky z výzvy a úspěchu v rovnováze. To může obtížné, protože hru mohou hrát hráči s různými úrovněmi dovedností. Mezi používané techniky pro nastolení rovnováhy patří [10]:

- Zvýšit obtížnost s každým úspěchem. Každá úroveň je těžší než ta předchozí.
- Nechat hráče projít rychle snadnými částmi. Pokud má hráč dostatečné dovednosti, je vhodné mu umožnit projít rychle začátek hry, aby se dříve dostal k částem, které jsou pro něj obtížnější.
- Vytvořit stupňovanou výzvu. Ohodnocením výkonu na konci každé mise, např. od A do F, kde A až C pustí hráče na další úroveň.
- Umožnit hráči zvolit úroveň obtížnosti.

Dobrá hra poskytuje hráči smysluplné volby/rozhodnutí, které mají dopad na další průběh hry a změnu herního světa. Jednou z nejzajímavějších voleb pro hráče je zda hrát "bezpečně" nebo na riziko. Dle [10] 8 z 10 hráčů přijde hra nudná z důvodu špatného vyvážení triangularity[9], která zahrnuje právě volbu bezpečí/rizika, viz obrázek 2.3. Jestliže jsou hráčům nabízeny takové volby, u nichž je jedna z variant jasně lepší než ostatní jedná o dominantní strategii, která by při vyvažování hry měla být odstraněna, stejně jako volby, které nemají žádný efekt. Otázkou zůstává, jak mnoho smysluplných možností hráči ve hře poskytnout. Michael Mateas [10] poukazuje na to, že počet možností, které hráči poskytnout, je závislý na počtu věcí, které si přejí.

- Pokud množství voleb je větší než si hráč přeje, pak je zahlcen.
- Pokud množství voleb je menší než jeho touhy, pak je hráč frustrovaný.
- Pokud je množství voleb rovné jeho touhám, pak má hráč pocit svobody a naplnění.

[9] Vztah mezi hráčem a způsobem hry, který volí. Hráč je jedním bodem trojúhelníku, bezpečná volba s nízkým rizikem je druhým bodem, a vysoké riziko je třetí bodem.



Obrázek 2.3:Princip triangularity zobrazující vztah hráče a míry rizika

Vyvážení náhodnosti a dovedností hráče je těžké, protože někteří hráči preferují hry, kde rozhoduje spíše náhoda, kdežto jiní hry, kde rozhodují spíše dovednosti. Zatímco dovednostní hry využívají systémy rozsudku, které určují, který hráč je nejlepší, hazardní hry mají volnější povahu a velká část výsledku závisí na osudu.

Dalším bodem je vyvážení myšlení a obratnosti. Některé hry jsou založeny čistě na přemýšlení, jiné na obratnosti (například vyhnout se překážkám) a zbylé kombinují přemýšlení i obratnost. Důležité je vědět pro koho hru vytváříme, a tomu ji přizpůsobit.

Tím, že hry obsahují prvky spolupráce či konkurence, poskytují lidem bezpečný způsob, jak prozkoumat chování lidí ve svém okolí ve stresových situacích. To je také jeden z důvodů, proč lidé rádi hrají hry společně. Častější jsou konkurenční hry, než hry, ve kterých se objevuje spolupráce, ale jsou i takové, které kombinují obojí. Například v režimu pro jednoho hráče jde o hru konkurenční a v režimu pro dva jde o spolupráci vedoucí k poražení společného nepřítele.

Délka hry je další z důležitých věcí, kterou je potřeba vyvážit v každé hře. Pokud je hra příliš krátká, nemohou hráči dostat šanci plně rozvíjet a realizovat smysluplné strategie, ale pokud trvá hra příliš dlouho, mohou se hráči začít nudit, nebo se mohou začít vyhýbat hře, protože hra vyžaduje příliš mnoho časového závazku. Změnit délku hry je možné úpravou podmínek definujících výhru nebo prohru.

Existuje několik běžných typů odměn pro hráče. Každý typ je jiný, ale všechny mají společný cíl, jímž je splnění hráčových tužeb. Může se jednat například o chválu, zisk bodů či nárůst síly. Problém ale je, jak odměny vyvážit. Obecně platí, že čím více druhů odměn může být do hry zapracováno, tím lépe.

Při vyvažování stojí za to, vzít v úvahu také prvek trestu ve hře. Ač se může zdát divné trestat hráče, protože hra by měla být zábavná, existují důvody, proč tak činit. Trest ovšem musí být použit jemně, jelikož hráči jsou ve hře ze své vlastní vůle. Tresty

jsou většinou opakem odměn, proto mohou být realizovány formou hanby, ztráta bodů či snížení síly. Je důležité, aby všechny tresty ve hře byly takové, aby jim byl hráč schopen porozumět a předcházet. Když cítí, že trest je náhodný a nelze jej zastavit, má hráč pocit nedostatku kontroly jen zřídkakdy je ochoten zapojit se do další hry.

Ve hrách je také otázkou kolik svobody hráči dopřát a jaké množství by mělo být ponecháno pro řízené zážitky. V případě, že hráči přinese hra prostřednictvím řízeného zážitku lepší zážitek, je vhodné upřednostnit tuto formu před svobodou.

Vyvážení jednoduchosti a složitosti je zaměřeno na odstranění problému, kdy je hra příliš jednoduchá a nudná, nebo kdy je příliš složitá a hráči jí nerozumějí.

Rozhodování o tom, co přesně by mělo být dáno, a co by mělo být ponecháno na hráčově představivosti je různé. Všechny hry však mají nějaký prvek fantazie a nějaký prvek reality. Cílem je najít rovnováhu mezi detaily a představivostí.

Pro správné vyvážení hry lze použít některé poučky, mezi něž patří například snaha o modulární myšlení, čistotu účelu, provedení jedné změny v čase či vytváření tabulek [11]. Modulární myšlení vychází z toho, že většina her není složena z jednoho systému, ale jde o soubor vzájemně propojených subsystémů. Dobrým způsobem, jak zjednodušit hru, je přemýšlet o ní z hlediska modularity. Pokud budou subsystémy modulární, bude snazší přesně určit, jaký měla úprava jednoho prvku hry dopad na jiné části. Čistota účelu znamená, že každá složka hry je jediná, jasně definovaná mise. K dosažení tohoto cíle, je nutné zobrazit herní mechaniku pomocí vývojového diagramu a definovat vztah a účel každé mechaniky. Poučka o jedné změně v čase vychází ze snahy snáze odhalit dopad změny na celý systém. Na druhou stranu sebou přináší nutnost testovat znovu celý systém. Poslední dobrou pomůckou jsou tabulky, které umožní mnohem snazší vyvažování a napomáhají udržovat přehled o již použitých hodnotách.

3 Testování

V předchozí kapitole byla pozornost zaměřena na zajištění kvality softwarových produktů, přičemž bylo zmíněno také testování. Testování je pouze jedním z aspektů zajištění kvality, nicméně je nezbytnou součástí každého vývoje softwarového produktu. Pokud produkt netestujeme, nemůžeme si být jistí, zda se bude chovat podle očekávání uživatelů či nikoliv. Na druhou stranu, testování není všemocné a i když budeme produkt testovat, nelze zaručit, že bude za každých okolností fungovat správně.

Aby testy přinesly co největší užitnou hodnotu, je nutné začít s jejich prováděním tak brzy, jak je to jen možné. Touto poučkou se řídí například iterativní vývoj, ve kterém část testování začíná již ve druhé fázi vývojového cyklu, naopak při vodopádovém přístupu se testování provádí až na samém konci vývoje, což už často znamená, že je příliš pozdě. Blíže se otázce kdy začít s testováním věnuje kapitola 3.2.

Vezmeme-li jednotlivé aktivity, prováděné v rámci vývojového cyklu (například analýzu, návrh, architekturu či implementaci), jsou všechny aktivity „konstruktivní“. Výjimku tvoří testování, jež je „destruktivní“. Pokud totiž vytvoříme nějaký test, který nepotvrdí, že vše funguje dle očekávání, může to u testera evokovat pocit viny, že ničí práci celého týmu. Takový pohled je ovšem špatný. Testy nesmí vytvářet negativní atmosféru mezi testery a zbytkem vývojového týmu. Naopak je nutné najít správný pohled na testy, tak aby vývojáři vnímali jejich přínos pro zlepšení kvality celého softwarového produktu.

Již byla nastíněna problematika, kdy začít s testováním, jak ale uvádí [5] je další otázkou testování rozhodnout, kdy s testováním skončit. Vezme-li v potaz situaci, že každá firma chce dodat produkt vynikající kvality, nikoliv jen uspokojivé, mohou u testerů a vedení probíhat snahy pokračovat s dalšími testy a provádět neustálé úpravy. To ovšem může znamenat prodlevy v dodání softwaru a způsobit, že pracovníci budou pracovat pod tlakem. Proto je nezbytné si uvědomit, že kvalita je důležitá, ale ne za každou cenu. Ať chceme nebo nechceme, každý software má nějaké chyby. Jak poznat kdy už je možné s testováním skončit bude uvedeno v kapitole 3.3.

Samotné testování může být velmi rozmanité. Existuje několik typů testů a metodik pro testování. Otázkou však zůstává, jaký typ testů je vhodný pro testování konkrétní části softwaru. Této problematice proto bude věnováno několik podkapitol této kapitoly.

3.1 Dobré testování a jeho cíle

Jak již bylo uvedeno, testování je široký pojem a zahrnuje celou řadu rozdílných typů testů, které však mají společný cíl, jímž je dodat kvalitní software. K zajištění tohoto cíle napomáhají tím, že se pokouší odhalit co největší množství nedostatků před dodáním softwarového produktu k zákazníkovi.

Například Glen Myers [13] přichází se třemi zásadami testování:

1. Testování je proces spuštění programu se záměrem najít chybu.
2. Dobrý test je ten, který má vysokou pravděpodobnost nalezení dosud neobjevené chyby.
3. Úspěšný test je ten, který odhaluje, dosud neobjevené chyby.

Z výše uvedených zásad vychází cíle testování. Tím hlavním je systematické odhalování různých druhů chyb s minimálním množstvím vynaloženého času a úsilí. Kromě toho by testy měli poskytovat ukazatel spolehlivosti softwaru a indikovat kvalitu softwaru jako celku. Zásadní ovšem je, že testování nemůže prokázat absenci chyb.

Jiný pohled na testování uvedený v [5] říká, že testování se zaměřuje především na posuzování kvality a je realizováno prostřednictvím několika základních postupů:

- Vyhledávání a dokumentování nedostatků v GEQ (Good Enough Quality) – neboli do češtiny přeloženo jako dostatečně dobrá kvalita (tento pojem je blíže popsán v kapitole 3.3).
- Týmová diskuze o vnímané kvalitě softwaru.
- Ověření předpokladů učiněných ve fázi navrhování a tvorby specifikace požadavků.
- Ověření, že funkce softwarového produktu fungují tak, jak bylo navrženo.
- Ověření, že požadavky byly vhodně implementovány.

A jaká kritéria by měl být splňovat samotný test? Jak uvádí Kaner, Falk, a Nguyen v [14] ideální test, že by měl mít následující atributy:

- Měl by mít vysokou pravděpodobnost nalezení chyby
- Neměl by být redundantní
- Měl by být nejlepší z testů, které lze v danou chvíli použít
- Neměl by být ani příliš jednoduchý, ani příliš složitý

3.1.1 Testovací principy

Předtím, než je možné navrhnout efektivní testovací případy, musí softwarový inženýr pochopit základní principy, jimiž se řídí testování softwaru. Mezi nejdůležitější testovací principy se řadí následující [1]:

- Všechny testy by měly mít návaznost na požadavky zákazníků, jelikož za nejzávažnější vady z pohledu zákazníka patří ty, které způsobují, že program nesplní jejich požadavky.
- Testy by měly být plánovány dlouho předtím, než začne samotné testování.
- Při testování softwaru je dodržován Pareto princip (viz kapitola 2.5).
- Testování by mělo začít v od testování individuálních komponent a postupně přecházet k nacházení chyb v integrovaných celcích a nakonec v celém systému.
- Úplné testování není možné, protože již pro středně velké programy existuje mnoho různých cest, kudy se běh programu může ubírat.

- Mají-li mít testy vyšší pravděpodobnost zjištění chyby, je lepší, aby byly provedeny nezávislou třetí stranou.

3.2 Kdy začít testovat a jak často testovat

Kdy začít testovat je dáno především zvoleným přístupem k vývoji softwaru. Například při iterativním vývoji se začíná s testováním již v raných fázích vývoje, což umožňuje najít větší množství chyb, ale především najít je zavčasu. Tím lze odhalit například nedostatky v návrhu architektury dříve, než je napsáno velké množství kódu či získat včasnou zpětnou vazbu, jež může mít za následek významnou úsporu času a nákladů. Naopak při použití vodopádového přístupu k vývoji se testování provádí až v poslední fázi, kdy je implementován veškerý kód. Pokud je při vodopádovém přístupu odhalena chyba, je vzhledem k velkému množství napsaného kódu oprava tohoto problému časově náročnější a je zapotřebí vynaložit více úsilí a peněz než v případě, že by byla odhalena dříve. Například [5] uvádí, že v případě vodopádového přístupu může být až 80 procent testovacího času a úsilí vynaloženo na plánování testů a definování testovacích případů, zatímco pouhých 20 procent je zpravidla vynakládáno na skutečné provedení a ladění testů. Navíc je velmi často nutné vyhradit dalších 20 procent času na nutné opravy. Nejen z tohoto důvodu je tedy ideální začít s testováním co nejdříve, jakmile je to jen možné.

Pokud jde o četnost testování i zde je rozdíl ve zvoleném přístupu. Iterativní vývoj umožňuje nejen dřívější testování, ale také častější testování. To je výhodné, protože provádění pravidelných testů dovoluje snáze nalézt moment, kdy chyba vznikla a tím pádem také snáze identifikovat změnu, která ji mohla způsobit. Příkladem pravidelně spouštěný testů jsou, mimo jiné, regresní testy (sekce 3.7.1), které lze pravidelně spouštět na konci každého dne/týdne či jiného časového období.

3.3 Kdy s testováním skončit

Ač se nemusí zdát, že by odpověď na otázku kdy s testováním skončit, respektive kdy je produkt dostatečně otestován, byla komplikovaná, opak je pravdou. Neexistuje žádná definitivní odpověď, nicméně se objevují různé pokusy o definování pouček, které by měli testerům či vývojářům umožnit snáze rozhodnout, kdy testování ukončit. Jednou z možností je ukončit testování v momentě, kdy dojdou čas a peníze vyhrazené pro projekt. Pokud se však podíváme na testování tak, že probíhá i v případě běžného provozu produktu uživatelem, pak testování nekončí nikdy. Mimo tyto fráze však byly vymyšleny i různé metriky. Například metrika chyb uvádí, kolik nových chyb přibylo za den a kolik jich za den bylo opraveno.

Poslední způsob, jak rozhodnout, kdy testování ukončit, přichází s pojem GEQ neboli dostatečně dobrá kvalita [5]. GEQ vychází z toho, že vydání softwarového produktu je možné i v případě, že není zcela dokonalý, ale je z pohledu vývojářů dostatečně kvalitní, aby jej zákazník mohl přijmout. K určení dostatečně dobré kvality jsou využívány různé modely, které jsou blíže popsány v [5], ale jeho využití

v softwarovém průmyslu, je bráno spíše jako reakce na překročení nákladů než cokoliv jiného.

3.4 Testovací techniky

Jak již bylo zmíněno v sekci 3.1 testování má za cíl odhalit chyby před dodáním softwaru k zákazníkovi. Aby bylo množství odhalených chyb co největší, bylo vynalezeno několik testovacích technik. Tyto techniky poskytují systematický návod pro navrhování testů, které vykonávají vnitřní logiku softwarových komponent a používají různé vstupy a výstupy s cílem odhalit chyby ve funkcích, chování a výkonu softwaru.

3.4.1 White-box testování

White-box testování [13], někdy také nazývané logic-driven testování, je testovací technika, která slouží k testování vnitřní struktury programu. Při odvození testovacích případů se vychází z řídicí struktury[10]. Za použití této techniky lze odvodit testovací případy, které zaručí, že všechny nezávislé cesty, kterými se může program uvnitř modulu ubírat, budou vykonány alespoň jednou, a že v případě, že program dorazí na logické „rozcestí“, budou testovány obě cesty (splnění a nesplnění podmínky) a budou provedeny všechny cykly s mezními[11] hodnotami.

White-box testování často se zaměřuje spíše na logické detaily a proto by se mohlo zdát, že tato technika nemá význam a je lepší čas a energii soustředit na komplexní testování softwaru, jímž se zabývá black-box technika (bude probrána v sekci 3.4.2). Jak je uvedeno v [1], odpověď spočívá v povaze chyby softwaru, přičemž každá z následujících typů vad poskytuje argument pro provedení white-box testů:

- Mohou se objevit logické chyby zapříčiněné nesprávnými předpoklady.
- Může dojít k situaci, kdy je určitá logická cesta prováděna pravidelně, ačkoliv bylo očekáváno, že k jejímu průchodu bude docházet zřídka.
- Do kódu mohou být zaneseny náhodné typografické chyby (překlepy).

Techniku White-box testování používá několik testovacích metod. Jednou z nich je Basis Path Testing, která patří mezi techniky testující řídicí struktury. Tato metoda umožňuje testerům z control flow diagramu (diagramu datového toku) odvodit míru logické složitosti programu a tu následně využít při stanovování základní sady cest, které by měly být testovány. Testovací případy odvozené tímto způsobem zajistí, že každý příkaz v programu bude vykonán během testování alespoň jednou. Více je k dispozici v [15].

[10] Řídicí struktury rozhodují o dalším průběhu programu a to tak, že samy o sobě nic nevykonávají, ale větví, cyklí nebo jinak mění běh programu. Patří sem složený příkaz, podmíněný příkaz, přepínač (switch) a také různé typy cyklů

[11] V tomto případě jsou mezními hodnotami myšleny hodnoty řídicích proměnných cyklů. Například pro cyklus s 50 opakováními může jít o hodnoty, 0,1, 49, 50 a 51.

3.4.2 Black-box testování

Zatímco white-box techniky se soustředí na logické problémy uvnitř komponent, black-box testy jsou určeny k ověření funkčních požadavků daných specifikacemi bez ohledu na vnitřní fungování programu [13]. Black-box techniky, někdy nazývané jako data-driven testování, jsou používány společně s white-box technikami protože se vzájemně doplňují a odhalují jinou třídu chyb. Vzhledem ke své povaze jsou black-box techniky využívány především v pozdějších fázích testování, kdežto white-box techniky jsou aplikovány spíše v raných fázích testování.

Při black-box testování jsou záměrně ignorovány řídící struktury a pozornost je zaměřena výhradně na funkčnost celku. Dle [1] je cílem najít chyby v následujících kategoriích:

1. Chyby způsobené nesprávnými nebo chybějícími funkcemi
2. Chyby v rozhraní
3. Chyby v datových strukturách nebo v přístupu k externím datům
4. Chyby v chování nebo výkonu
5. Chyby související s inicializací a ukončováním

V rámci black-box techniky bylo vytvořeno několik testovacích metod. Jednou z nich je metoda Equivalence class partitioning (ECP), česky lze tuto metodu nazvat jako rozdělení do ekvivalentních tříd, která sdružuje vstupní data produkující stejný typ výstupu do tříd. Díky rozdělení do tříd poskytuje tato metoda možnost vytvořit pouze jeden testovací případ pro každou ekvivalentní třídu. Konečným důsledkem pak je, že tato metoda skýtá možnost současného odhalení většího množství chyb, čímž zvyšuje efektivitu testování, jelikož v případě klasického testování by bylo nutné vytvořit pro odhalení těchto chyb několik testovacích případů. Více viz [2].

Další z black-box technik je Boundary Value Analysis (BVA), která byla vyvinuta z toho důvodu, že větší počet chyb je častěji způsoben mezními (hraničními) hodnotami vstupních dat než ostatními. BVA proto vede k výběru testovacích případů, které se zaměřují na testování hraničních hodnot. Mimoto je BVA často používána společně s ECP, přičemž při vytváření ekvivalentních tříd se pro testování nevolí náhodný prvek z této třídy, ale vybírají se hraniční prvky v rámci třídy. BVA není zaměřena pouze na testování vstupních dat, ale lze ji použít i pro testování výstupů [13].

Pro systémy, kde je spolehlivost softwaru naprosto zásadní (brzdný systém v automobilech, palubní počítač v letadlech), je určena metoda Comparison testing (CT). V těchto systémech je často použit redundantní hardware a software, aby se minimalizovala možnost vzniku chyby. Při využití metody CT se provádějí duplicitní testy, při kterých každá verze aplikace může testovat stejná testovací data, aby se zjistilo, zda všechny verze poskytují stejný výstup [1].

Existuje mnoho aplikací, ve kterých je vstup omezen relativně malým počtem platných hodnot. To znamená, že počet vstupních parametrů je malý, a hodnoty, které každý z parametrů může nabývat, jsou jasně ohraničené. Když jsou tato čísla velmi

malá (např. tři vstupní parametry o třech diskrétních hodnotách pro každý z parametrů), je možné vzít v úvahu všechny vstupní permutace a otestovat všechny kombinace vstupních dat. Nicméně, jak se počet vstupních hodnot zvyšuje a roste počet diskrétních hodnot, je tento způsob testování nepraktický nebo nemožný. Proto byla vytvořena metoda nazývaná Orthogonal array, která je blíže popsána v [1].

3.4.3 Techniky pro specifická prostředí, architektury a aplikace

Testovací techniky zmíněné v předchozích odstavcích byly použitelné pro libovolné prostředí, architektury a aplikace. Existují však specializovaná prostředí, architektury či aplikace, pro něž je lepší použít specializované metody. Mezi příklady, které vyžadují specializovaný přístup, patří např. testování grafických uživatelských rozhraní, klient/server architektury, dokumentace a nápovědy k produktu a systémů pracující v reálném čase.

Testování dokumentace a nápovědy je často opomíjeno. Nicméně je důležité si uvědomit, že softwarový produkt není jen program samotný, ale patří k němu mimo jiné i dokumentace a nápověda, a proto je vhodné zahrnout do testování také tyto prvky. Chyby v dokumentaci či nápovědě totiž mohou být pro fungování softwaru stejně závažné jako chyby ve vstupních datech nebo zdrojovém kódu. Například pokud se uživatelský manuál nebo on-line nápověda neshodují s programem, pak nepřinášejí uživateli žádný užitek, ale naopak jej matou. Při testování dokumentace a nápovědy je ideální zvolit dvoufázový přístup, kdy se v první fázi provádí přezkoumání (využívá se například FTR - viz kapitola 2.4), při němž se zkoumá dokument z hlediska přehlednosti. Ve druhé fázi následuje testování, při němž se používá dokumentace současně s aktuálním programem [1].

U systémů fungujících v reálném čase nestačí použít pouze white-box a black-box techniky, protože situaci komplikuje závislost na čase a asynchronní povaha mnoha aplikací. Problémy může způsobit například zpracování událostí, načasování dat či souběžnost úloh, jež zpracovávají data. Často může u těchto systémů dojít k tomu, že pokud se nachází systém v určitém stavu, budou data zpracována správně, zatímco když systém bude v jiném stavu, může nastat chyba. Softwarové testy navíc musí zvážit i dopad hardwarových závad, přičemž takové chyby může být velmi obtížné simulovat. Výše uvedené skutečnosti mohou být příčinnou toho, proč jsou metody pro systémy reálného času stálé ve vývoji [1].

3.5 Testovací strategie

Testovacích metod je značné množství, aby však došlo ke zvýšení jejich účinnosti, byly vytvořeny testovací strategie. Ty obsahují plán, jenž definuje série kroků, která má být při testování provedena. Kroky jsou poskládány tak, aby za využití white a black box technik postupně přecházeli od testování samostatných částí systému (odhalování chyb v logice programu a funkcích) k testování větších celků. Nejprve jsou testovány individuální komponenty, jež jsou následně integrovány. Testování pokračuje s těmito integrovanými částmi, které se mohou opětovně integrovat do větších celků. Jakmile je

kompletní program funkční, je provedena sada testů, jež jsou navrženy tak, aby odhalily chyby v požadavcích.

Součástí každé strategie je plánování, návrh testovacích případů, provedení testů, sběr dat a hodnocení výsledků, včetně informací o množství času a prostředků, které budou k provedení testů zapotřebí.

Testovacích strategií existuje několik a zahrnují různé typy testů, mezi něž patří například jednotkové, integrační, alfa či beta testy. Některým z nich budou věnovány následující kapitoly.

3.6 Jednotkové testy

Jednotkové (unit) testy patří mezi white-box metody a jsou určeny k testování nejmenších jednotek navrženého softwaru, jimiž jsou třídy, komponenty nebo moduly. Přičemž relativní složitost těchto testů je redukována omezeným rozsahem stanoveným pro tento typ testů.

Jedním z cílů, které mají jednotkové testy plnit, je testování rozhraní. Při tomto typu testů je záměrem ujištění, že do a z testované programové jednotky proudí správné informace. Dále je v rámci unit testů testována integrita dat místních datových struktur během všech fází vykonání algoritmu. Také jsou testovány okrajové podmínky, aby se zajistilo, že modul správně zpracovává data v mezích stanovenými limity a omezeními. Jsou vykonávány všechny nezávislé[12] cesty procházející přes řídicí struktury s cílem zajistit, že všechny příkazy v modulu byly provedeny alespoň jednou. Posledním úkolem unit testů je projít různé cesty související s chybovým chováním programu, například ošetření výjimek [1].

Testování zpracování okrajových hodnot je důležitým úkolem jednotkového testování, protože software často při zpracování těchto hodnot selhává. Například při zpracování *n-tého* prvku z *n-rozměrného* pole, když je vyvoláno *i-té* opakování smyčky z *i* průchodů, nebo když se narazí na maximální či minimální povolenou hodnotu. Testování, při němž jsou testovány hodnoty dat těsně pod a těsně nad maximem a minimem, a maximální a minimální hodnoty, má proto předpoklad, že s větší pravděpodobností povede k odhalení chyb.

V případě testování zpracování chyb patří mezi potenciální chyby, které by měly být testovány, následující typy chyb [1]:

- Popis chyby je nesrozumitelný.
- Uvedená chyba neodpovídá zjištěné chybě.
- Chybový stav způsobuje zásah ze strany systému ještě před zpracováním chyby.
- Zpracování výjimky je nesprávné.
- Popis chyby neposkytuje dostatek informací o příčině chyby.

[12] Nezávislými cestami jsou v tomto kontextu myšleny různé cesty, kterými program prochází a které ovlivňují jeho chování.

Kromě výše uvedených chyb by testy měly odhalit rovněž chyby typu [1]:

- Porovnávání rozdílných datových typů.
- Použití nesprávných logických operátorů nebo špatně ošetřené priority.
- Očekávání rovnosti při rozdílné (chybné) přesnosti dat.
- Nekorektní porovnání proměnných.
- Nesprávné nebo neexistující ukončení cyklu.
- Chybějící možnost ukončit provádění, pokud je zjištěna divergentní iterace.
- Nesprávně upravované proměnné cyklu.

Jak uvádí [5] jedním z osvědčených postupů agilního přístupu k testování (lze ho využít i na jiné typy testů) je myšlenka vytvoření testů před implementací kódu. Podstatou je napsat a realizovat testovací případy a až pak psát kód. Kód je napsán tak jednoduše, jak je to možné, a pak je postupně vylepšován. Při tomto postupu napíše programátor jeden nebo dva velmi jednoduché unit testy, a pak se upraví program tak, aby testy skončili s očekávaným výsledkem. Tato technika se nazývá Test-First design.

U objektově orientovaného softwaru je nejmenší testovatelnou jednotkou nejčastěji (zapouzdřená) třída případně objekt. Cílem testů je otestovat chování uvnitř třídy, přičemž se testuje jak očekávaný průběh, tak zpracování „chybných“ dat.

Jelikož jednotkové testování patři mezi základy každé testovací strategie, bylo vyvinuto značné množství automatizovaných testovací nástrojů. Rovněž byla uzpůsobena vývojová prostředí, která pomáhají testerům s vytvářením, krokováním či vyhodnocením testů.

3.7 Integrační testy

Integrační testy jsou dalším krokem, který by měl následovat po jednotkových testech. Jakmile jsou hotové a otestované jednotlivé části hry, přichází na řadu integrace těchto částí do výsledného celku. Integrace může probíhat nárazově, nebo postupně. Postupná integrace je vhodnější, protože umožňuje snáze izolovat a opravovat chyby, které byly nalezeny při integračních testech. Navíc při postupné integraci dochází k dřívějšímu (i když jen částečnému) propojení komponent, takže lze provádět testy po delší dobu.

Jak uvádí [1] dají se integrační testy realizovat dvěma základními způsoby. Prvním z nich je takzvané Top-down (shora dolů) testování, při němž jsou moduly integrovány směrem dolů. Začíná se od hlavního řídicího modulu a následně jsou připojovány podřízené moduly. Při začleňování podřízených modulů se dá opět postupovat dvěma způsoby, buď do hloubky, nebo do šířky. Top-down strategie má tu nevýhodu, že pokud je testována vyšší úroveň, jež potřebuje zpracovat data z nižší úrovně v hierarchii, nejsou tato data ještě dostupná a musí být nahrazena dočasným kódem. Druhým způsobem jak provádět integrační testování je metoda Bottom-up (zdola nahoru). Jak její název napovídá, začíná s jednoduchými moduly, tj. s komponentami na nejnižších

úrovních, ve struktuře programu a při integraci se postupuje směrem nahoru. Díky tomu odpadá problém s neexistujícími částmi kódu nižší úrovně.

3.7.1 Regresní testy

Kdykoliv je přidán nový modul, který se stává součástí integrovaného celku, je nutné zjistit, zda nevznikly nové datové cesty, nedošlo ke změně vstupních/výstupních dat nebo zda nepřibyla nová řídicí logika. Tyto změny mohou způsobit problémy s částmi kódu, které dříve fungovaly bezchybně. Proto existuje v rámci strategie integračního testování, regresní testování, které má za úkol opakované vykonávání jednotlivých podskupin testů, které již byly provedeny, aby se ověřilo, zda nová funkčnost nemá vedlejší účinky ovlivňující funkčnost dříve integrovaných částí kódu [1].

Regresní testování může být provedeno ručně, ale s narůstajícím množství kódu, začíná být toto vykonávání neefektivní. Proto je vhodné regresní testování automatizovat. Díky automatizování testů je možné provádět opakované vykonávání testovacích případů, zaznamenávat výsledky jednotlivých testů a porovnávat je s předchozími výsledky. Tímto způsobem mohou denně probíhat i stovky nebo tisíce regresních testů.

3.7.2 Smoke testy

Smoke (kouřové) testování je opět součástí strategie integračního testování a umožňuje vývojovému týmu pravidelně posuzovat stav produktu. Pro provádění kouřového testování je nutné, aby byl integrovaný celek pravidelně sestaven. Sestavení (build) zahrnuje všechny datové soubory, knihovny opakovaně použitelných modulů a komponent. Série testů je navržena tak, aby odhalila i chyby, které zabraňují, aby bylo dané sestavení úspěšně dokončeno. Ideální variantou je, pokud je na konci každého dne proveden build, který je přes noc otestován.

3.8 Funkční testy

Cílem funkčního testování je ověřit, zda aplikace vykonává činnost, která se od ní očekává a provádí ji správně [16]. Z tohoto důvodu by mělo být funkční testování prováděno pravidelně. Při funkčním testování je nejprve ze specifikací jednotlivých komponent zjištěno, jak mají správně fungovat. Následně se provádí samotné testování, které může probíhat podle předem připraveného scénáře. Tyto scénáře popisují kroky, které se při testování mají provést a často jsou odrazem akcí prováděných koncovými uživateli při používání softwaru. Díky scénářům lze navíc opakovat kroky se stejnou posloupností a zároveň lze vypozorovat, jaké kroky vedli k nalezení chyby, díky čemuž je možné lépe navodit chybový stav.

Krom správné funkčnosti dle specifikace by však při funkčním testování měl být kontrolován i vzhled komponenty, respektive zda komponenta odpovídá grafickým návrhům. Vzhledem k tomu že funkční testování je zaměřené z velké části na internetové aplikace, které využívají webový prohlížeč, je důležité ověřit funkčnost

komponenty ve všech podporovaných prohlížečích, jelikož v závislosti na prohlížeči se může měnit vzhled i funkčnost komponent.

V případě, že je softwarový produkt sestavován z již hotových modulů, u kterých je známo, co je nutné testovat, lze testerům poskytnout kontrolní seznamy k danému modulu či komponentě. Tento seznam obsahuje všechny funkce, které musí komponenta splňovat a tester tak může jednotlivé funkce snáze ověřovat. Navíc si může ke každému bodu poznamenat, zda byl úspěšně proveden či nikoli. Seznam s výsledky pak tester může předat dále, případně může chyby zavést do systému pro evidenci chyb.

Funkční testování není nutné dělat pouze manuálně procházením grafického uživatelského prostředí, ale je možné využít i testovacích nástrojů, které podstatnou část práce zautomatizují. Mezi zástupce těchto nástrojů se řadí například Selenium[13]. Tento nástroj umožňuje převést testovací scénáře do elektronické podoby a následně je spouštět opakovaně a testovat tak, zda se komponenta během vývoje nezměnila či nepřestala fungovat.

3.9 Validační testy

Provádění častých buildů vede ke kontinuální integraci, díky čemuž lze odhalit mnohé problémy s kvalitou a pravidelně ověřovat, zda kód funguje. Kromě toho je důležité, začít brzy s předvedením aplikace uživatelům, aby se včas získala zpětná vazba. I přesto, že jsou během vývoje provedeny různé testy, nemusí při nich být podchyceno, zda je aplikace opravdu užitečná a poskytuje požadované chování. Je totiž prakticky nemožné, aby vývojář předvídal, jak bude zákazník program opravdu používat. To lze zjistit až v momentě, kdy je software prověřen skutečnými uživatel.

Pro získání zpětné vazby od koncových uživatelů jsou používány validační[14] testy. Ty přicházejí na řadu, když je software kompletně sestaven a neobsahuje chyby v modulech ani chyby vzniklé propojením modulů. Validace může být definována mnoha způsoby, ale nejjednodušší definicí je, že ověření softwarové funkce je úspěšné tehdy, když splňuje očekávání, která lze rozumně předpokládat zákazníkem [1]. Otázkou ovšem je, co jsou rozumná očekávání. Rozumná očekávání jsou definována v požadavcích uvedených ve specifikaci požadavků, která popisuje všechny uživatelsky viditelné atributy softwaru. Specifikace požadavků obsahuje část nazvanou hodnotící kritéria, která obsahuje informace, jež tvoří základ pro validační testování.

Dalším přínosem při validačním testování je, že aplikace běží v cílovém prostředí. Naopak ostatní testování je prováděno ve vývojovém prostředí, které nemusí vytvářet stejné podmínky jako prostředí cílové a může vést k zavádějícím výsledkům.

Validace softwaru je dosaženo prostřednictvím řady black-box testů, které prokážou nebo vyvrátí shodu s požadavky. Plán testů vymezuje třídy testů, které mají být

[13] Nástroj Selenium je dostupný ke stažení na adrese http://docs.seleniumhq.org/, kde je rovněž popis nástroje

[14] Validace znamená ověření, validační testy lze tedy vnímat jako testy určené k ověření funkčnosti

provedeny, a definuje konkrétní testovací případy, které budou použity k prokázání shody s požadavky. Plán a postup jsou navrženy tak, aby zajistily, že bude dosaženo všech funkčních požadavků, behaviorálních charakteristik a výkonnostních požadavků, bude vytvořena správná dokumentace, a budou splněny další definované požadavky (např. přenositelnost, kompatibilita či zotavení po chybě) [1].

Každé validační testování by mělo skončit jedním ze dvou výsledků. Buď funkce a vlastnosti výrobku odpovídají specifikaci a software je přijat, nebo je nalezena odchylka od specifikace, která je zaznamenána do vytvořeného seznamu nedostatků.

Je-li software vyvíjen jako výrobek, který používá mnoho zákazníků, je nepraktické provádět validační testy u každého z nich. Z tohoto důvodu bylo v rámci validačního testování zavedeno alfa (kapitola 3.9.2) a beta testování (kapitola 3.9.3).

3.9.1 Akceptační testování

Akceptační testování je finálním testem před nasazením softwaru. Cílem akceptačního testování je ověřit, že software je připraven a může provádět funkce a úkoly pro které byl vytvořen. K dispozici jsou tři strategie pro akceptační testování [1]:

- Formální přijetí
- Neformální přijetí
- Beta testování

3.9.2 Alfa testování

Alfa testování je prvním krokem v případě validačního testování, v případě, že je softwarový produkt určen většímu množství zákazníků. Alfa testy mohou být využívány nejen pro integrované celky, nýbrž i k testování jednotlivých komponent. Navíc lze alfa testování využít jak pro testování klasického softwaru, tak pro testování her. Alfa testování je prováděno zákazníkem, který je pozván vývojáři, přičemž vše probíhá pod dohledem vývojáře, který se uživateli "dívá přes rameno" a zaznamenává chyby a problémy spojené s používáním softwaru [1].

3.9.3 Beta testování

Beta testování se na rozdíl od alfa testování provádí na jednom nebo více místech, ale vždy v cílovém prostředí u zákazníka a je prováděno koncovým uživatelem softwaru. Dalším rozdílem je, že se při něm nevyskytuje vývojář. Proto je beta test nazýván jako „živý“ test aplikace softwaru v prostředí, které nemůže být kontrolováno vývojáři. Zákazník zaznamenává všechny problémy (skutečné i domnělé), které vyvstávají v průběhu beta testování a hlásí je vývojářům v pravidelných intervalech. V důsledku problémů hlášených během beta testů, softwaroví inženýři provedou změny a připraví výsledný softwarový produkt, který poté dají k dispozici všem zákazníkům.

Pokud se dá očekávat, že se softwarem budou pracovat uživatelé s různou mírou zkušeností či software bude využíván jednotlivými uživateli k různým aktivitám, je vhodné při výběru koncových uživatelů pro beta tetovování vybrat uživatele, kteří

budou patřit do skupin od úplných začátečníků až po zkušené uživatelé, a rovněž uživatele z různých prostředí a různými potřebami [1]. Tato rozmanitost pomůže zajistit, aby všechny aspekty produktu byly řádně testovány. Rovněž je vhodné zahrnout do beta testování také pokyny k instalaci, uživatelské manuály, návody či školící materiál. V opačném případě hrozí, že zpětná vazba bude neúplná nebo nekvalitní.

3.10 Systémové testy

Systémové testy jsou ve skutečnosti řadou různých testů, jejichž hlavním účelem je plně uplatnit softwarový produkt. I když každý test má jiný účel, jejich společným cílem je ověřit, zda prvky systému byly správně začleněny a plní přidělené funkce. Krom mnoha dalších testů pod systémové testy spadají [STS]:

- Testování bezpečnosti se pokouší ověřit, zda ochranné mechanismy zabudované do systému budou skutečně systém chránit před neoprávněným přístupem.
- Zátěžové testy jsou navrženy tak, aby vystavili software do neobvyklé situace. Proto jsou spouštěny tak, aby vyžadovali prostředky v abnormální množství, frekvenci, nebo objemu.
- Testování výkonu je určeno k testování run-time výkonu softwaru v rámci integrovaného systému. Testování výkonu je často spojené se zátěžovými testy a obvykle vyžaduje jak hardware, tak software.
- Testování použitelnosti (tomuto typu testování je věnována kapitola 3.12).
- Recovery testování, též testování schopnosti zotavení, úmyslně vynucuje selhání systému mnoha různými způsoby, a ověřuje, že obnova po chybách je provedena správně.

3.11 Play testy

Specifickou součástí testování je playtesting (play testy či český ekvivalent herní testy) určený pro testování her. Jak již bylo popsáno v kapitole 2.8, u her hrozí velké množství nebezpečí, které mohou hru „odsoudit k zániku“. Play testy mají za cíl odhalit tyto hrozby ještě předtím, než je hra vydána a hrána koncovými hráči, případně ujistit vývojářský tým, že vytvářejí správnou hru pro správné publikum a přinášejí hráčům očekávané zážitky. Aby k tomu došlo, je nutné nechat testovací hráče, aby si hru zahráli. Ve skutečnosti jde však ještě o mnoho dalších věcí. Hraní hry je pouze jednou ze součástí procesu, který zahrnuje výběr, nábor, přípravu, řízení, a získání zpětné vazby a provedení analýz. Jinými slovy playtesting je komplexnější proces, jehož účelem je zlepšit zážitky hráče, protože návrháři her často přes usilovnou práci na hře zapomínají na hráče. Z tohoto pohledu je playtesting kritická část návrhu hry, kterou nelze uspěchat nebo ji odsunout do pozadí.

Některé z výše popsaných součástí playtestingu budou přiblíženy v následujících podkapitolách a detailnější informace o herních testech jsou k dispozici v [10] a [11].

3.11.1 Proč provádět play testy

Tato otázka již byla částečně zodpovězena výše, ale v této sekci je pozornost zaměřena na konkrétní důvod, který vede testery k realizaci play testů v daný okamžik. Pokud návrháři, vývojáři či testeři nemají v hlavě před provedením konkrétních herních testů specifické cíle, velmi reálně hrozí, že play testy budou jen ztrátou času. Čím konkrétnější otázky jsou připravené ve fázi přípravy play testů, tím větší přínos playtesting přináší. Jak uvádí [10], existují miliony otázek, na které je možné se při playtestingu zeptat, ale jednoduchá otázka typu *"Je moje hra zábavná?"* nestačí.

3.11.2 Výběr playtesterů

Po přípravě otázek nastává čas získání testerů, kteří budou testy provádět, přičemž výběr testerů závisí na cíly play testů. V první fázi vývoje by měl být testerem hry sám její tvůrce, proto se mluví o tzv. self-testingu (český ekvivalentem by mohlo být testování sebe sama) [11]. V této fázi se také často vytvářejí řešení k výrazným problémům s herními zážitky. Self-testing může probíhat po celou dobu vývoje hry, ale postupem času je nutné začít spoléhat na externí testery.

S externími testery je možné začít, jakmile je hotový prototyp. Předtím, než se přejde přímo ke koncovým hráčům, je však vhodné nechat hru otestovat příbuznými či přáteli, kteří se na vývoji hry nepodílely. Tím se získá nový pohled na hru a zároveň je snazší tyto lidi získat a o hře s nimi hovořit. Na druhou stranu je zde riziko, že rodina a přátelé nebudou zcela upřímní a budou se snažit zamlčet některé nedostatky, které hra bude mít [11]. V momentě kdy, je hotový prototyp, který hráči poskytuje dostatek informací k zahájení a hraní hry, je čas jej nechat otestovat playtestery z okruhu koncových hráčů. V které fázi vývoje je vhodné využít daný typ playtesterů je uvedeno v tabulce 3.1.

	Návrhář/vývojář	Rodina a přátelé	Koncový hráči
Základní návrh	●		
Tvorba struktury	●	●	
Formální detaily			●
Upřesnění/doladění			●

Tabulka 3.1: tabulka znázorňující využití různých playtesterů během vývoje

V případě hledání externích testerů je dobré najít takové, kteří pomohou posunout hru dále, o hře nikdy neslyšeli a patří mezi skupinu uživatelů, kteří by o ni mohli mít potenciálně zájem (například vhodná věková skupina či pohlaví) [11]. Problémem ovšem je, jak takové hráče najít. Mezi nejčastěji preferované možnosti při hledání patří například nábor na střední či vysoké škole, ve sportovním klubu či společenské organizaci, případně je možné hledat zájemce online na internetu. V případě, že je kandidátů velké množství, je dalším krokem jejich postupné vyřazení (např. pokud nemají rádi tento typ her). Ideální je najít takové testery, kteří reprezentují cílovou skupinu, jež hru bude hrát, protože takový testeři mohou poskytnout lepší zpětnou vazbu než ostatní a navíc mohou hru porovnat s typově podobnými hrami. Obecně platí, že čím rozdílnější skupinu hráčů z cílové skupiny se podaří pro play testy nalézt, tím lépe [11].

Jestliže se provádí playtesting opakovaně, je vhodné získat nové playtestery, aby opět přinesly nový pohled na hru. Někdy je však vhodné přibrat i testery, kteří se testů již účastnili, a poskytly přínosnou zpětnou vazbu, jelikož lze z jejich názoru odhadnout, jak dle jejich názoru hra pokročila.

3.11.3 Průběh

Standardní playtesting se skládá z nás následujících částí [11]:

- Představení
- Zahřívací diskuze
- Testování
- Získání zpětné vazby
- Závěr

Ve fázi představení probíhá základní konverzace, při které dochází k uvítání testerů, představení návrháře a poděkování testerům, že přišli. Rovněž je vhodné při představování ubezpečit testery, že jejich pomoc je pro další vývoj hry důležitá. Otázkou je, zda testerům popisovat hru, protože může dojít ke zničení jejich nezaujatého pohledu na hru. V případě, že se návrhář rozhodne testerům hru popsat, měl by jim sdělit jen nejdůležitější informace. Krom ztráty nezajatého pohledu totiž hrozí, že se tester zaměří na činnost, kterou návrhář zmínil, což odvede jeho pozornost a tester může přehlédnout věci, kterých by si jinak všiml. Navíc musí být návrhář opatrný, aby nevyvolal u testerů dojem, že si přeje, aby přehlíželi nedostatky, které hra má. K odstranění nebo alespoň minimalizování těchto problémů pomáhá vytvoření testovacího skriptu, který přispívá k tomu, aby návrhář řekl jen to, co je pro hraní hry nutné.

V případě potřeby je možné provést zahřívací diskuzi, aby se návrhář dozvěděl, které hry podobné této hře testeři hrají, co na nich mají rádi, jaké jsou jejich oblíbené hry nebo jak získávají informace o nových hrách.

Dle [11] by mělo probíhat samotné testování 15 až 20 minut, jelikož při delší hře, mají hráči tendenci se unavit. Na začátku této fáze je vhodné upozornit testery, že hra je stále ve vývoji a ujistit se, že pochopili, že jde o testování hry, nikoli jejich schopností. Taktéž je užitečné hráče ubezpečit, že veškeré problémy, které budou mít během hraní hry, mohou pomoci hru vylepšit. Návrhář by měl testery také vyzvat k tomu, aby přemýšleli nahlas a komentovali, co právě provádí. Během hry může být návrhář přítomen přímo mezi hráči a pozorovat je při hře, včetně poslouchání a případně i zaznamenávání toho co říkají, nebo se na ně může dívat přes sklo či sledovat kamery. Důležité je ovšem vědět, co sledovat. Většina lidí má tendenci sledovat, kam se dívá hráč, neboť tímto způsobem můžeme vidět, co vidí hráč. Někdy je ovšem lepší dívat se hráčům do tváře, protože je možné tímto způsobem zjistit, co hráč v danou chvíli cítí. Mimika může poskytnout informace, které nelze získat z následných rozhovorů ani z otázek v průzkumu. Díky použití moderní videotechniky však lze sledovat jak ruce, tak tvář testerů a získat komplexnější informace [11].

Jak již bylo uvedeno, jedním z faktorů, které mohou přispět k lepším výsledkům z playtestingu, je přemýšlení nahlas. U některých lidí se ovšem v takovém případě mění způsob, jakým se chovají (často se jejich chování stává více promyšlené a pečlivé). Další část lidí může být ochromena, když se snaží hrát a mluvit současně, a když se ve hře vyskytne stresující moment, často přestanou tito lidé mluvit úplně. Bohužel právě tyto momenty jsou často ty, kdy návrhář nejvíce potřebuje zjistit, o čem hráč přemýšlí. Nicméně, pro některé hráče je přemýšlení nahlas zcela přirozené, a tak mohou poskytnout velmi užitečné informace. Trik je v tom, nalézt právě tyto hráče.

Poslední otázkou zůstává, zda hráče rušit během hry. Pokud se návrhář rozhodne vyrušit hráče v průběhu hry, riskuje zbrzdění jejich přirozeného stylu hry. Na druhou stranu, kladením správných otázek v pravý okamžik má příležitost poznat věci, které by jiným způsobem nezjistil. Variantou je poznamenat si otázku a zeptat se na ni po skončení hry, bohužel v té době si hráč již nemusí vzpomenout, o čem je řeč. Většina návrhářů proto upřednostňuje variantu, při které hráče přerušuje pouze v případě, když dělá něco, co je skutečně překvapivé, a čemu návrhář nerozumí [11].

Po skončení samotného hraní přichází na řadu fáze získání zpětné vazby, často vedená formou diskuze s hráči, která by měla trvat opět 15 až 20 minut. Pro tuto diskuzi by měl mít návrhář připraven seznam otázek, který se postupným vývojem hry často více konkretizuje, přičemž diskuze může být vedena jak formou průzkumu/dotazníku, tak formou rozhovoru [11].

Průzkumy jsou ideálním prostředkem, jak získat od hráčů odpovědi na otázky týkající se hry. Odpovědi získané při průzkumech lze navíc snadno kvantifikovat, pokud se řídí následujícími doporučeními [10]:

- Používat obrázky kdykoli je to možné, protože pomáhají hráči pochopit, co má návrhář na mysli.
- Používat on-line průzkumy, které mohou ušetřit spoustu času, jsou snadno nastavitelné a levné (například Surveymonkey[15]).
- Nepoužívat v odpovědích stupnici bodování typu *1* až *5*, jelikož vede k získání konzistentních výsledků, ale spíše škálu možností typu *hrozný, spíše špatný, průměrný, dobrý* a *vynikající*
- Nedávat do průzkumu příliš mnoho otázek, jinak lidé přestanou ke konci vnímat a výsledky ztratí hodnotu.
- Podrobit hráče průzkumu hned poté, co dohráli.
- Mít po ruce někoho, kdo odpovídá na upřesňující otázky, které by mohli testeři k průzkumu mít.
- Poznamenat si věk a pohlaví každého sledovaného testera pro sledování závislosti mezi těmito údaji a stanoviskem hráče.

Další možností jsou rozhovory, které poskytují příležitost, jak se zeptat hráče na otázky, které jsou příliš složité pro jednoduchý průzkum. Je to také způsob, jak zjistit,

[15] Nástroj vhodný pro on-line průzkumy, který je dostupný na adrese http://www.surveymonkey.com/

jaké má tester ze hry pocity, protože můžeme pozorovat emoce v jeho tváři. Nejlepší je provádět tyto rozhovory s lidmi soukromě, pokud je to možné, protože pak budou mluvit upřímněji, než v případě, kdy je poslouchají jiní lidé. Také se mohou vyhýbat „bolestivým pocitům“, respektive kritice hry, proto je vhodné, aby je návrhář upozornil, že i tato kritika může přispět k vylepšení hry a že tuto kritiku chce slyšet [10].

Poslední částí playtestingu je závěr, při kterém zbývá čas na to, ujistit se, že má návrhář veškeré potřebné kontakty a že získal všechny potřebné informace k vyhodnocení zpětné vazby od testerů, poděkovat jim a rozloučit se s nimi.

3.11.4 Kde play testy probíhají

V předchozí části byl probrán průběh playtestů, přičemž pozornost byla věnována testování, které probíhá v laboratoři. To je způsob, který je preferován velkými firmami, ale nemohou si jej dovolit jednotlivci, či menší společnosti, protože je tento způsob finančně náročný. Existují však i jiné možnosti. Jednou z nich je využití svého vlastního ateliéru či budovy. Dále je možné vybrat si veřejné místo. Ani jedna z těchto možností se o mnoho neliší od možnosti využít sofistikovanou laboratoř, ale možnosti provést playtesting přímo u playtestera doma či prostřednictvím internetu jsou do jisté míry zcela specifické. V případě „domácího“ testování je výhodou možnost vidět, jak hráč hraje hru ve svém přirozeném prostředí. Naopak nevýhodou jsou omezené možnosti, díky nimž lze tímto způsobem realizovat jen malé množství testů. Playtesting realizovaný prostřednictvím internetu je pak naprosto unikátní. Jeho výhodou je, že si hru bude moci vyzkoušet velké množství lidí na strojích s mnoha různými konfiguracemi. Obzvláště v případě her pro více hráčů může být internet nejlepší volba. Na druhou stranu s velkým množstvím testerů často klesá kvalita herních testů [10].

3.11.5 Řízená hra

Nástrojem, který bývá někdy použit pro zlepšení efektivity playtestingu je využití tzv. řízené herní situace či řízené hry. Řízená hra nedovoluje testerovi provádět ve hře libovolné úkony, ale nastavením určitých parametrů jej nutí testovat specifickou část hry, která by jinak nemusela nastat. Jak uvádí [11], převážně se jedná o testování následujících částí:

- Konec hry
- Náhodný jev, který nastává jen ojediněle
- Zvláštní situace ve hře
- Zvláštní úroveň hry
- Nová funkce

3.11.6 Poznámky a příprava otázek

Během pozorování hráčů při hře je vhodné si rovněž dělat poznámky. Důležité je sepisovat je chronologicky a mít v nich přehled. Tím můžeme získat otázky, které lze spolu s připravenými otázkami využít při získávání zpětné vazby od testerů.

Seznam otázek by neměl být příliš dlouhý (například 20 a více otázek v řadě), aby byli testeři koncentrovaní na všechny otázky. Může se zdát, že čím více otázek na seznamu bude, tím více získáme od testerů informací. Důležitý však není počet otázek, ale kvalita odpovědí.

Při tvorbě seznamu otázek se lze inspirovat různými seznamy s předpřipravenými otázkami, např. seznamem uvedeným v [11]. Na druhou stranu každá hra je specifická a proto musí seznam obsahovat rovněž otázky, které by měli odpovídat na problémy konkrétní hry, protože odpovědi na tyto otázky budou mít největší přínos. Dobrým způsobem, jak vytvářet otázky je nalézt problematické oblasti hry a zaměřit se na získání zpětné vazby z těchto oblastí. Vzhledem k tomu, že otázek je často větší počet, je ideální všechny otázky na konci seřadit podle priority a vybrat jen ty nejdůležitější.

3.11.7 Shromaždování a interpretace dat

V předchozích částech byla pozornost zaměřena spíše na získání kvalitativní zpětné vazby, ale důležité je shromažďovat rovněž kvantitativní data, jako je například doba, kterou trvá někomu přečíst pravidla, počítání počtu kliknutí, které je potřeba pro vykonání určité funkce (například zabití nepřítele), nebo sledování jak rychle dělají hráči pokroky při dosažení další úrovně [11]. Opět platí, že data, která jsou sbírána, souvisí s tím, na jaké problémy se hledá odpověď. Nicméně není důležité mít jen statistiky o všech myslitelných aspektech hry, ale je důležité vědět, jak tato čísla interpretovat, jinak jsou k ničemu. Proto se na před samotnými play testy musí připravit seznam předpokladů, který má být potvrzen či vyvrácen, a účel měření. Pro shromažďování dat byly vyvinuty různé nástroje a techniky. Například mezi používané nástroje patří Microsoft Games User Research[16].

Jakmile jsou potřebná data shromážděna, je nutné je vyhodnotit. I pro tyto účely jsou vývojáři v hojné míře využívány specializované nástroje a vizualizační software, které jim pomáhají analyzovat shromážděná data či určit účinnost prvků a vlastností v porovnání s jinou hrou. Například mohou návrhářům pomoci určit, která část hry je dominantní a jak ji vylepšit.

3.12 Usability testy

Usability testy, česky testování použitelnosti, se zaměřují na zkoumání intuitivnosti a snadnosti použití uživatelských rozhraní [11]. Tyto testy jsou určeny pro „klasický“ i herní software a soustředí se na zkoumání komunikace mezi uživatelem/hráčem a rozhraním. Testy použitelnosti jsou často realizovány společně s play testy a na jejich průběh dohlížejí speciálně vyškolení psychologové nebo výzkumní pracovníci, kteří se zaměřují na zkoumání a hodnocení toho, jak uživatelé pracují s různými produkty.

Před zahájením usability testů vědci připravují testovací skripty, které požadují po účastnících testů projít několik částí uživatelského rozhraní nebo dokončit sadu úkolů. V případě testování her je pro větší úspěšnost testů důležité najít nejkritičtější oblasti,

[16] Tento nástroj je dostupný na http://mgsuserresearch.com/

které by měli být primárně testovány. Testování použitelnosti je často prováděné v laboratořích, které jsou vybavené sofistikovaným záznamovým zařízením. Díky tomuto zařízení je možné nezávisle pozorovat ruce hráče, reakce v obličeji hráče, pohyby očí, zaznamenávat co si hráč během hry nahlas povídá a rovněž poskládat různé další potřebné pohledy. Například propojit pohyb rukou a výraz v obličeji. Během testů vědci obvykle sedí za jednosměrným sklem s designéry a výrobcem produktu a komunikují s účastníkem přes interkom (telefon), aby se účastníci testů soustředili na hru nikoli na ně. Pokud hra není určena pro více hráčů, je nejlepší provést tento typ testování one-on-one (neboli realizátor testů pouze s hráčem). Důvodem je, že pokud lidé klopýtají nebo hádají co udělat, pak se často snaží skrýt, nebo kopírovat od souseda ve skupině. Jestliže však musí být účastníci během testu ve stejné místnosti, je dobré jim vysvětlit, že by si neměli s úkoly navzájem pomáhat.

Stejně jako v případě play testů je nutné najít správné testery, kteří budou použitelnost testovat. V tomto případě se však výběr vymezuje pouze na třetí skupinu lidí, tedy na externí uživatele, kteří nikdy hru nehráli a zastupují uživatele z cílového segmentu trhu. Jak uvádí [11] je ideální velikost skupiny pro testy osm lidí z každého segmentu trhu (v případě, že je více než jeden), ovšem dostatečný počet může být mezi třemi až pěti lidmi.

V případě testování použitelnosti u her patří mezi nejkritičtější oblasti pro testování nejčastěji začátek hry a některé z kritických rozhodnutí při výběru konkrétní funkce.

4 Projekt Space Traffic

Tato kapitole je věnována popisu projektu Space Traffic, jež je vyvíjen na Katedře informatiky a výpočetní techniky (dále jen KIV). Jedná se o studentský projekt jež je veden a vyvíjen studenty v rámci jejich závěrečných prací a rovněž vyvíjen studenty předmětů vyučovaných na KIV. Samotný postup při vývoji a vedení projektu je nad rámec této práce, ale je popsán v diplomových pracích [17] a [18] z předchozích let.

Jak uvádí [19] projekt Space Traffic podporovaný KIV má za cíl vytvoření webové hry pro více hráčů. Tématem této hry je vesmírné obchodování. Do hry je navíc přidáno programování jako prvek hratelnosti. Díky tomu mohou hráči ve hře vytvářet programy pro automatizaci některých herních činností. (Například ovládání kosmických lodí.)

V následujících sekcích bude krátce představen účel projektu, jeho historie, prvky pro zajištění kvality, využití výukových prvků a podíl autora této práce.

4.1 Účel projektu

Účelem projektu je zvýšit zájem o studium nabízených oborů u žáků středních škol prostřednictvím prezentace činnosti svých studentů. Prostředkem pro tento účel je stejnojmenná hra vytvářená v rámci tohoto projektu. Další přínos pak spočívá ve formě vývoje (studentský projekt vyvíjen výhradně studenty), který poskytuje studentům KIV příležitost prezentovat své dovednosti a získat cenné zkušenosti v oblasti týmové práce, vedení projektů a vývoje softwarových produktů.

4.2 Historie projektu

Historie projektu se datuje do akademické roku 2009/2010 kdy s realizací začal student Zbyněk Neudert, jenž chtěl původně pokračovat ve svojí dřívější práci věnované návrhu webové hry Zoo City. Vzhledem k tomu, že mu to nebylo umožněno, zahájil práce na vývoji webové hry Space Traffic. Jak popisuje diplomová práce [17] přes problémy způsobené časovou tísní a nedostatkem studentů realizujících vývoj byl vytvořen návrh hry, pevný základ pro vývoj a funkční prototyp založený na technologiích PHP, HTML a JavaScript.

V následujícím ak. roce (2010/2011) byl projekt převzat Richardem Kocmanem. Jak se však ukázalo, vytvořené řešení nemělo parametry, které by umožňovaly v pracích pokračovat. Předchozí tým totiž vzhledem k nedostatku zkušeností podcenil problematiku předání projektu do dalších let, jelikož se příliš soustředil na praktické výsledky projektu. Neuchopitelná architektura implementace spolu s nedostatečnou dokumentací nakonec vedly k rozhodnutí opustit původní řešení a vytvořit nové, založené na PHP frameworku Nette[17]. To zapříčinilo zdržení vývoje projektu, který byl

[17] Nette je český PHP framework dostupný na http://nette.org/cs/ .

následně v březnu roku 2011 nuceně přerušen, protože přišel o svého projektového manažera.

V ak. roce 2011/2012 byl projekt převzat Petrem Voglem a Martinem Štěpánkem. Jak zmiňuje autor [19] bylo v předchozím ak. roce provedeno jen velmi málo implementace a trval záměr nepokračovat v implementaci z roku 2009/2010. Proto bylo rozhodnuto o změně technologie na .NET. V tomto akademickém roce byl vystaven základ projektu, který byl realizován na bázi architektury MVC[18]. Mimo samotných diplomových prací [18] a [19] probíhal vývoj v rámci několika předmětů vyučovaných na KIV. Díky dostatku studentů a preciznímu vedení obou diplomantů bylo vytvořeno jádro hry, dokumentace, wiki a rovněž byl vytvořen materiál s důležitými informacemi o projektu, jenž byl předán následujícímu vedení. Vzhledem k dobrému základu hry navíc bylo rozhodnuto, že bude přistoupeno k testování hry.

V akademickém roce 2012/2013 bylo vedení projektu převzato Janem Dyrczykem a autorem této práce. Na základě předaných informací pokračoval vývoj aktuální verze hry a na základě doporučení předchozího vedení rovněž způsob začleňování nových studentů do projektu.

4.3 Zajištění kvality

Jelikož je projekt vyvíjen studenty v rámci semestrálních a diplomových či bakalářských prací, není možné zajišťovat kvalitu na takové úrovni, jako například v zaběhlých softwarových firmách. Důvodem jsou především časová omezení a také omezené množství studentů, kteří se na projektu podílejí. Navíc se studenti na projektu pravidelně prostřídávají. Přes tato omezení však byla a je při vývoji snaha zajistit maximální možnou kvalitu.

Prvním faktorem pro její zajištění je snaha o iterativní vývoj, který probíhá v rámci každého zadání semestrální práce studentům. Tím je možné na pravidelných schůzkách vedoucích projektu a studentů, kteří pracují na zadaných semestrálních pracích, probrat aktuální stav vývoje a získat další informace potřebné k vývoji. Tyto schůzky tak umožňují včas najít problematické části a zastavit případný špatný směr vývoje. Navíc poskytují možnost získat od vedoucích projektu dodatečné informace, které jsou nezbytné pro další vývoj. Poslední přínos je pak specifický pro akademické prostředí, respektive pro studentský vývoj. Studenti totiž často nechávají vývoj na poslední možnou chvíli. Pravidelnými schůzkami je tak zajištěno, že vývoj bude probíhat postupně, díky čemuž by vyvíjená část měla být dodána včas a měl by zbýt čas na opravení problémů či nedostatků. Více viz [18].

Dalším prostředkem pro zajištění kvality hry Space Traffic jsou komentáře v kódu a vytváření dokumentací. Vzhledem k tomu, že studenti se na vývoji často mění, je nutné zajistit, aby se nově příchozí studenti mohli seznámit s aktuálním produktem

[18] MVC je straka z Model View Controler. Jedná se o softwarovou architekturu, jež rozděluje aplikaci do tří nezávislých vrstev.

a pokračovat ve vývoji nastoleným směrem. Jak je napsáno výše, může být problém s nedostatkem informací větší, než je na první pohled patrné a způsobit tak zánik aktuálního produktu, tak jako tomu bylo u první verze hry. Na konci každé semestrální práce proto vzniká dokumentace realizované části. Navíc v projektu funguje wiki, která uchovává důležité informace o vývoji a na níž se objevují nově přidané části hry. Krom toho jsou studenti vedeni k tomu, aby kód hojně komentovali, především pak části kódu, u kterých nemusí být na první pohled patrné, jak fungují. Poslední částí je pak předání informací studentům, kteří přebírají vedení projektu pro další akademický rok. Tyto informace jsou předávány v dokumentu nazvaném *Handover-Notes* a obsahují základní informace nutné pro seznámení s projektem, informace o aktuálním stavu projektu či seznam lidí, na které je možné se obrátit.

Posledním a však neméně důležitým prostředkem pro zajištění kvality je testování. To je ve Space Trafficu realizováno prostřednictvím několika různých typů testů, které jsou popsány v kapitole 6.

4.4 Výukové prvky

Dalším významným přínosem webové hry Space Traffic by měla být možnost ovládat lodě pomocí programů, které si studenti sami vytvoří (blíže se programovatelnému ovládání lodí věnuje kapitola 5.). Vytváření vlastních programů spadá do skupiny takzvaných výukových prvků[19] a mělo by přinášet studentům KIV možnost využít studiem získávané znalosti v řešení netriviálních úloh a zároveň uplatnit i vlastní kreativitu.

4.5 Příspěvek autora

Jak je uvedeno výše, je projekt vyvíjen po více let a vystřídalo se na něm větší množství studentů, přičemž autor této práce se vývoje projektu účastnil v posledním roce vývoje (akademický rok 2012/2013) prostřednictvím oborového projektu a této diplomové práce.

Během této doby se autor této práce podílel na přípravě zadání pro studenty (pouze v zimním semestru), náboru studentů, pravidelných konzultacích se studenty, kontrole jejich práce a na konci zimního semestru také na připojení částí hry vyvíjených odděleně od hlavní vývojové větve.

Dále autor této práce realizoval přidání programového ovládání lodí a logování událostí spojených s provozováním lodí do lodního deníku. V rámci této nové funkcionality vytvořil GUI, implementoval zpracování dat na serveru, uložení persistencích[20] dat (události a programy) a propojení s modulem *StarshipBasicInterpreter*. Tento modul byl ovšem realizován Janem Šmajclem, který

[19] Výukové prvky se do her přidávají za účelem naučit hráče novým faktům, nebo pro zvýšení znalostí v daném oboru.

[20] Persistentní data jsou trvale uložená data, která mohou být uložena například v souboru, nebo v databázi

s autorem této práce spolupracoval. Šmajcl také na základě vytvořeného seznamu příkazů vytvořil ve spolupráci s autorem gramatiku pro jazyk Starship Basic popsaný v kapitole 5.

Další částí, na níž se autor zaměřoval, bylo zajištění kvality projektu Space Traffic prostřednictvím testů (této části se věnuje 6. kapitola). V rámci testování vytvářel, upravoval a kontroloval jednotkové testy realizované v rámci projektu, prováděl funkční testy, provedl úpravy nutné pro uskutečnění play a usability testů, které rovněž připravil a vyhodnotil, a připravil doporučení pro provedení beta testů a pro další vývoj hry.

V neposlední řadě se pak autor této práce účastnil akce Den otevřených dveří, dále jen DOD, na které byl potenciálním zájemcům o studium projekt představen, a na základě doporučení z předchozího akademického roku vytvořil dokument *Handover Notes* pro studenty, kteří budou pokračovat ve vedení projektu v příštím ak. roce.

5 Programovatelné ovládání lodí

Jak již bylo uvedeno, programovatelné ovládání lodí má ve hře výukovou povahu. Aby však měli hráči snahu psát vlastní programy a ovládat tak lodě, musí jim tato činnost přinášet určitý přínos. Ve hře totiž mohou lodě ovládat i běžným způsobem. První výhodou proto je, že díky programu mohou automatizovat „nudnou“ a opakovanou činnost spojenou s klikáním na objekty herního světa či odesíláním lodí na cílové planety. Druhou výhodou je pak možnost zlepšit své skóre, jelikož při vhodně napsaném programu lze například ušetřit palivo, či vybrat levnější nabízený produkt.

Při výběru programovacího jazyka pro programování bylo nutné zvolit vhodný programovací jazyk. Vzhledem k faktu, že programování je v současném výukovém systému probíráno z hlediska vyšších programovacích jazyků, např. Javy, studenti často nemají tušení, jak vše funguje na nižších vrstvách. Proto při výběru jazyka bylo rozhodnuto, že pozornost bude zaměřena právě na nízkoúrovňové programování, které nabízí studentům možnost setkat se s jiným typem programování, než jakému jsou běžně učeni. Tomuto výběru rovněž nahrává skutečnost, že není nutné vytvářet složité programy, které by vyžadovali použití vysokoúrovňových prostředků. Z výše uvedených důvodů byl vybrán dříve rozšířený programovací jazyk Basic. Z množství jeho různých verzí nakonec výběr padl na jazyk Sinclair Basic[21], který byl využit jako základ pro cílový jazyk pojmenovaný Starship Basic.

5.1 Postup návrhu a realizace programovacího jazyka

Základní idea o použití jazyka založeného na Sinclair Basicu vzešla od Martina Štěpánka. Tento student rovněž učinil základní návrh příkazů vycházejících z tohoto programovacího jazyka a uvedl je ve své diplomové práci [19]. Tento seznam byl rozdělen do tří základních částí. První obsahovala elementární příkazy jazyka převzaté z jazyka Sinclair Basic a další dvě části obsahovaly příkazy vytvořené přímo pro tuto hru. Konkrétně se jednalo o příkazy pro akce lodí a příkazy pro získávání informací o herním světě.

V zimním semestru tohoto akademického roku byl seznam revidován autorem této práce a některé příkazy byly vyřazeny[22]. Poté byla ve spolupráci s Janem Šmajclem navržena gramatika programovacího jazyka (celá gramatika je k dispozici v příloze A). Dle této gramatiky byl poté Janem Šmajclem vytvořen interpreter jazyka, který byl na pravidelných týdenních schůzkách projednáván s autorem této práce. Výsledkem bylo začlenění nového modulu, který je blíže popsán v kapitole 5.6. Vzhledem k množství dalších prací na projektu nebylo vhodné, aby začlenění programovatelného ovládání lodí probíhalo v hlavní vývojové větvi (trunk). Proto byla vytvořena pro Starship Basic

[21] Sinclair Basic je dostupný na http://www.worldofspectrum.org/ZXBasicManual/

[22] K vyřazení některých příkazů ze seznamu došlo především z časových omezení danými potřebou realizovat programovací jazyk co nejdříve, aby mohl být zapracován do play testů a případně předveden studentům na Dni otevřených dveří.

samostatná vývojová větev (branch) nazvaná *opswi_starship_basic*. Paralelně s interpreterem jazyka pak bylo realizováno i grafické uživatelské rozhraní (dále jen GUI, jež je popsáno v kapitole 5.3) pro tento jazyk, přičemž vývoj probíhal v téže samostatné větvi. Výsledný interpreter byl hotov do konce zimního semestru a v průběhu letního semestru byl propojen se zbylými částmi projektu včetně GUI.

5.2 Příkazy jazyka

Jak již bylo uvedeno, původní seznam příkazů byl rozdělen do tří částí a některé příkazy byly z tohoto seznamu vyřazeny. Jednalo se však výhradně o elementární příkazy, přičemž šlo převážně o různé matematické funkce, které z pohledu současné funkčnosti nejsou potřebné a jen by uživatele zatěžovali. Mimo matematické funkce byly vyřazeny i některé další elementární příkazy, které by však do budoucna mohlo být vhodné do hry integrovat. Na druhou stranu mezi elementární příkazy byly přidány příkazy pro celočíselné dělení (DIV a MOD) a také byl změněn příkaz pro odmocninu (místo původní funkce SQR byla použita funkce SQRT). Příkazy, které byly navrženy pro akce lodí a získávání informací, byly ponechány v původním rozsahu.

Některé ze standardně používaných jazykových konstrukcí jsou popsány v následujících podkapitolách. Kompletní seznam všech použitých i nepoužitých příkazů z původního seznamu je uveden v příloze B a rovněž byl společně s gramatikou umístěn na wiki projektu.

5.2.1 Skoky a návěští

Skoky v programovacích jazycích umožňují přechod na zvolené místo v programu a stejně tak tomu je i v případě Starship Basicu. Pokud se má skočit na zvolené místo v programu, použije se příkaz GO TO následovaný názvem návěští. Toto návěští pak musí být uvedeno na samostatné řádce a musí být následováno znakem dvojtečka. Při skoku na návěští se pokračuje s vykonáváním příkazu uvedeným pod řádkem s návěštím. Viz následující příklad.

```
GO TO tiskTextu
              .
              .
              .
tiskTextu:
PRINT "jsem na navesti"
              .
```

5.2.2 Podmíněné příkazy

Další základní konstrukcí je vykonávání podmíněných příkazů. Jazyk Starship Basic používá pro vytváření podmíněných příkazů klíčová slova IF, THEN a případně ENDIF. Jak je však patrné, není definované klíčové slovo ELSE. Toto klíčové slovo chybí z důvodu, že Starship Basic umožňuje pouze vytvoření takzvaného neúplného podmíněného příkazu, který má jen jednu větev, která je vykonána pouze v případě

splnění podmínky. V případě, že by hráč cítil potřebu vytvoření úplného podmíněného příkazu, pak si musí vypomoci skoky (viz předchozí sekce 5.2.1).

Pokud má být proveden pouze jeden podmíněný příkaz, je tento podmíněný příkaz napsán na téže řádce jako samotná podmínka a nepoužívá se klíčové slovo ENDIF. Například *IF cislo% = 5 THEN PRINT "cislo ma hodnotu 5"*. Pokud má však být vykonáno více podmíněných příkazů, pak je nutné psát každý podmíněný příkaz na samostatnou řádku. Ukončení vykonávání podmíněných příkazů potom musí být dáno klíčovým slovem ENDIF na samostatné řádce. Od tohoto klíčového slova poté program pokračuje s vykonáváním standardních příkazů. Viz následující příklad.

```
IF cislo% = 5 THEN
prvniPrikaz
druhyPrikaz
     .
     .
     .
posledniPrikaz
ENDIF
```

Krom výše uvedených odlišností od dnes nejčastěji používaných jazyků je vykonávání podmíněných příkazů víceméně standardní. Umožněno je vytvářet jak jednoduché podmínky, tak složené podmínky. Složené podmínky vznikají z podmínek jednoduchých, přičemž jednoduché podmínky jsou propojeny přes logické operátory AND a OR. Stejně jako u jiných jazyků platí, že v případě logického operátoru AND musí být splněny všechny podmínky, kdežto v případě použití logického operátoru OR stačí, když je splněna alespoň jedna z podmínek.

5.2.3 Cykly

Dalším standardem v programovacích jazycích jsou cykly. Starship Basic podporuje pouze jeden typ cyklu, kterým je cyklus s pevně daným počtem opakování (FOR). Tento typ cyklu umožňuje zadat spodní a horní mez vykonávání a také krok. K definování všech těchto částí používá klíčová slova FOR, TO, STEP a NEXT, přičemž klíčové slovo NEXT následované názvem proměnné je uvedeno na samostatné řádce za posledním příkazem, který se má v rámci cyklu vykonat. Všechny klíčová slova, krom klíčového slova STEP následovaného krokem, jsou povinná. Pokud není krok uveden, je proměnná zvětšována o jedničku. Naopak při použití klíčového slova STEP je možné zadat libovolný krok, tedy i záporný. Ukázka cyklu je uvedená níže.

```
FOR i% = minHodnota% TO maxHodnota% STEP 4
prikaz1
   .
   .
   .
prikazN
NEXT i%
```

5.3 GUI

GUI pro Starship Basic je stejně jako ostatní části grafického rozhraní realizováno v modulu *GameUi* prostřednictvím technologií HTML, kaskádových stylů (CSS, respektive optimalizované varianty less) a javascriptu, jež definují způsob zobrazení. Základem GUI je herní mapa s hvězdnými systémy, červími dírami a planetami doplněná o množství prvků používaných ke hře. Tyto prvky jsou tejně jako v jiných částech hry zobrazovány s anglickým popisem. Další část GUI, jež není zobrazována, ale slouží pro kontrolu a zpracování, tvoří kód umístěný v kontrolérech.

Příkazy jazyka Starship Basic jsou v současné verzi psány ručně do textového pole umístěného na záložce *navicomp* vybrané vesmírné lodi. K této záložce se musí uživatel nejdříve dostat tím, že buď vybere v menu položku *Ships* nebo koupí novou loď. Obě tyto možnosti jej přesměrují na přehled lodí, kde vybere konkrétní loď, pro niž chce napsat program. Tím se dostane na detail lodi a poté již může vybrat odkaz *navicomp,* čímž se dostane na cílovou záložku, ve které už může začít se zvolenou činností spojenou s programovacím jazykem Starship Basic. Aby došlo k výběru správné vesmírné lodi, je do cookies webového prohlížeče uložena hodnota identifikačního čísla lodi.

Záložka *navicomp* obsahuje celkem 2 textové oblasti a 5 tlačítek. Pro lepší představu je obsah této záložky, včetně zbylé části obrazovky, zachycen na obrázku 5.1. Jedna z textových oblastí je určena pro napsaný/načtený program a druhá je určena pro výstup. To znamená, že uživatel je nucen napsat nebo načíst kód, který vytvořil dříve. Tento způsob byl zvolen z hlediska snadnosti realizace, která byla vzhledem k časovému omezení hlavním rozhodujícím faktorem. Na druhou stranu však tento způsob klade větší nároky na hráče, který nejen, že musí vymyslet jak napsat kód, který mu přinese vyšší zisk či usnadní práci, ale musí jej napsat celý. Druhou zamýšlenou možností, která by mohla být v budoucnu realizována, bylo vytvářet program z předem připravených grafický komponent, které by bylo možné vzájemně napojovat. Pokud by byl tento způsob realizován, mohl by být ponechán i způsob stávající, přičemž vzhledem k větší náročnosti psaní programu by mohl být bodově či jiným způsobem zvýhodněn oproti tvorbě z předem připravených komponent. Druhá textová oblast je určena pro zobrazování výsledku kompilace nebo pro výpis hráčem požadovaných hodnot/zpráv, jinými slovy toho, co chce hráč na obrazovku vytisknout příkazem PRINT, případně vypsání výjimky pokud k nějaké při vykonávání programu došlo.

Obrázek 5.1: GUI pro práci s program Starship Basic

Jak již bylo zmíněno, je na záložce také 5 tlačítek. Z těchto tlačítek jsou však vidět při načtení stránky pouze 4. Páté tlačítko určené pro spuštění programu je z bezpečností důvodů vidět až po úspěšné kompilaci programu. Výběr správného tlačítka stisknutého uživatelem případně zavolání metody kontroléru se správnými parametry je realizován prostřednictvím funkcí v javascriptu.

Prvním tlačítkem je *Help* (nápověda). Po stisku tohoto tlačítka má hráč možnost vidět v nově otevřeném okně kompletní nápovědu k programu, která obsahuje důležité informace. Například jak psát potřebné jazykové konstrukce či přehled funkcí včetně příkladů jejich použití. Pro snadnější orientaci má celá nápověda přehled, z něhož je možné se dostat přímo na hledanou část. Na konci každé sekce je pak možnost návratu zpět na přehled.

Tlačítko s popisem *Save* (ulož) je určeno k uložení rozepsaného programu v libovolné fázi jeho psaní. Vzhledem k tomu, že pro každou loď může být uloženo více programů, má po stisku tohoto tlačítka hráč možnost vybrat v nově otevřeném okně název programu a jeho pozici. Výběr pozice je důležitý především v pozdější fázi hry, kdy hráč již vyčerpal volná místa pro uložení programu. V současné verzi hry je možné uložit maximálně 5 různých programů pro každou loď. Pokud však má již uživatel pro loď uložených 5 programů a chce pro ni uložit nový, musí vybrat program (kód), který bude tímto novým kódem nahrazen, přičemž jej buď může přejmenovat, nebo ponechat stejné jméno. Uložení pak hráč potvrdí stiskem tlačítka *Save* na příslušné pozici.

Díky tomu, že každá loď může mít uložených až 5 programů, je důležité mít možnost v případě potřeby načíst jiný program. K tomu je určeno tlačítko *Load* (načti). Rovněž po stisku tohoto tlačítka je uživateli zobrazeno nové okno, v němž hráč vybírá, který program chce načíst. Z tohoto pohledu je vhodné programy vhodně pojmenovávat, jelikož při načítání programu je hráči zobrazen pouze název programu a tlačítko *Load* pro každý program. Po stisku tlačítka *Load* u vybraného programu je identifikátor

načteného programu uloženo do cookies a na základě tohoto identifikátoru je potom do textového pole načten příslušný zdrojový kód, se kterým je možné dále pracovat.

Předposledním tlačítkem, a zároveň posledním viditelným při načtení stránky, je tlačítko *Compile* (kompilovat, sestavit). Po stisku tohoto tlačítka je zdrojový kód určený k sestavení odeslán ke zpracování do dalších vrstev, přičemž se čeká na výsledek kompilace. Tento výsledek je poté zobrazen uživateli ve výstupní textové oblasti. Pokud kompilace skončení úspěšně je vypsána zpráva o úspěšné kompilaci (*Compilation was succeeded, 0 errors.)* a hráči se zobrazí tlačítko *Run*. V opačném případě je hráči zobrazeno několik prvních chyb s krátkým popisem chyby a řádkem, na němž se chyba vyskytla (např. *Line 2 : ExpectingStringFactor - Expecting string or string variable*) a tlačítko Run zůstane skryté.

Tlačítko *Run* (spustit) je určeno ke spuštění programu. Z bezpečnostních důvodů je hráči zobrazeno až v případě úspěšného sestavení programu, aby nemohlo dojít ke spuštění programu, který ještě nebyl sestaven a nemá tak nastavené potřebné ukazatele či vyhrazenou paměť. Jelikož může běžet pro každou loď v danou chvíli pouze jeden program, není od spuštění programu až do jeho konce tlačítko *Run* vidět. Aby pak nedošlo k neočekávanému chování běžícího programu a problémům s pamětí, není viditelné ani tlačítko *Compile*.

Při načtení záložky *navicomp* mohou být obě textové oblasti prázdné nebo obsahovat text. Pokud se jedná o nově zakoupenou loď, pak jsou obě pole prázdná. Pokud však již byl pro loď uložen nějaký program, případně více programů, je otázkou, který kód by měl být načten. V případě, že existuje pouze jeden program, pak se načte on, v případě, že je jich více, načte se ten, který byl naposledy uložen (ať už hráčem, prostřednictvím tlačítka *Save* nebo automaticky při kompilaci).

5.4 Zpracování na serveru

Pokud hráč při práci s programy psanými pro vesmírné lodě stiskne libovolné tlačítko, krom tlačítka *Help,* je v kontroléru zavoláno vykonání odpovídající akce, která je herním serverem vytvořena, uložena do fronty akcí a poté v nejbližším možném okamžiku vykonána.

Server, jenž akce vykonává, je umístěn v modulu *GameServer*. Vytvoření správné akce serverem je realizováno službou *GameService*, která vytvoří na základě názvu akce a jejích parametrů novou akci. Pokud akce vrací nějaký výsledek, pak se na něj v třídě *GameService* čeká a tento výsledek je následně získán a předán do GUI, kde je náležitě zpracován. Vzhledem k tomu, že může být vykonáváno více akcí zároveň, je nutné zajistit, že bude vyzvednut správný výsledek. To je zajištěno přes proměnnou *ActionCode*, která má pro každou akci jedinečnou hodnotu. Jednotlivé akce jsou realizovány prostřednictvím jednotlivých tříd umístěných ve složce *Actions*.

Při ukládání programů (stisk tlačítka *Save* na příslušné pozici) je vytvořena akce s názvem *SaveStarshipBasicCode*. V této akci se na základě vstupních parametrů pouze

určí, zda byl program již uložen a má dojít jen k jeho aktualizaci, nebo zda se jedná o nový program. V obou případech je zavolán Data Access Object (DAO), který se stará o propojení s datovou vrstvou, která je zodpovědná za uložení dat do databáze. Pro veškerou práci s programy psanými pro vesmírné lodě je tímto objektem *SpaceShipProgramDAO*. Poté co je zavolána příslušná metoda této třídy se čeká na výsledek uložení, který je vrácen do booleovské proměnné. Tato hodnota je na konci celé akce uložena do proměnné *Result*, jež je určená k předání výsledku dále.

Při načítání programů (stisk tlačítka *Load* na příslušné pozici) dochází k opačné činnosti, jejímž cílem není uložení persistentních dat, ale naopak jejich získání. To je realizováno v akci *LoadStarshipBasicCode*. Na jejím konci je opět výsledek uložen do proměnné *Result*, ze které je později vyzvednut. Tentokráte je však uložena hodnota identifikátoru načteného programu.

Pokud chce hráč program kompilovat (stisk tlačítka *Compile*), je nutné provést větší množství činností. První z nich je samotné generování kódu, při kterém probíhají potřebná nastavení, například nastavení paměti či program counteru. Tato nastavení jsou prováděna v samostatném modulu vytvořeném pro Starship Basic, konkrétně v modulu *StarshipBasicInterpreter* ve třídě *CodeGenerator*. Poté je v akci zkontrolováno, zda se jedná o nový program, nebo zda byl kompilován program, který byl již dříve uložen. Tato činnost je prováděna téměř stejně jako v akci pro uložení programu, s tím rozdílem, že samotná aktualizace dat v databázi neprobíhá ihned, ale až na konci celé akce *CompileStarshipBasicCode*. Pokud kompilace proběhla bez chyb, je vrácena pouze informace o bezchybném průběhu a hráč může spustit program. V opačném případě je hráči vrácen seznam chyb, které byly během kompilace odhaleny.

Nejkomplikovanější je však samotné spuštění programu (stisk tlačítka *Run*), které je zpracováváno v akci *RunStarshipBasicCode*. Po počátečních nastaveních je z této akce zavolán *InterpreterManager* v němž probíhá interpretace samotného kódu. V tomto bodě se situace komplikuje, jelikož vykonání programu může trvat delší čas, než na který je nastaven limit pro dokončení každé akce. Tento limit je nastaven proto, aby dlouhé akce neblokovali jiné a nezpomalovali tak celou hru. Z tohoto důvodu je nutné kontrolovat, jak dlouho program běží a pokud se běh akce blíží limitu pro dokončení akce, musí být interpretace kódu ukončena a nastaveny všechny potřebné proměnné. V akci *RunStarshipBasicCode* je poté naplánovaná nová akce, která pokračuje s vykonáváním programu v místě, kde předchozí akce skončila. Obdobná situace nastává rovněž v případě, kdy je spuštěna nějaká déle trvající akce, například let lodi na planetu (FLY TO) či naložení zboží na loď (LDCARGO). I v tomto případě musí být program přerušen a musí dojít k naplánování nové akce. Tentokráte je však plánována přímo požadovaná akce, například akce *PrepareShipFlyTo* či *ShipLoadCargo*. Jakmile je tato akce dokončena, pak na základě toho, že byla tato akce spuštěna z programovacího jazyka Starship Basic (tato informace je předávána akci jako jeden z parametrů), dojde k naplánování nové akce pro spuštění programu, která opět pokračuje dále od místa, kde předchozí akce skončila. Poslední částí je zpracování výjimek, které mohou při běhu Starship Basicu nastat. Pokud je výjimka zachycena, je

informace o ní předána a v GUI potom ve výstupní textové oblasti zobrazena. Rovněž je ukončen běh akce a samotného programu.

5.5 Persistentní vrstva

Stejně jako všechny trvale uložené objekty ve hře, jsou ukládány i objekty vesmírné lodi a programu. Původně však tato data byla ukládána jako jeden objekt, přičemž objekt lodi obsahoval pouze proměnnou, ve které byl text programu umístěn. Jinými slovy byla použita pouze jedna tabulka pro vesmírné lodě (*SpaceShips*), která obsahovala sloupec s napsaným programem. Protože však tímto způsobem bylo možné uložit ke každé lodi pouze jeden program, byla vytvořena nová tabulka pro programy (*SpaceShipProgram*). Díky tomu je v současné době možné pro jednu loď uložit více programů, konkrétně až pět. Toto omezení je však zajištěno již v logické vrstvě a v databázi je propojení obou tabulek realizováno klasickou vazbou 1:N bez omezení.

Při ukládání dat do databáze je nutné provést objektově relační mapování (ORM), jelikož databáze je relační, kdežto kód v C# je psaný objektově. K tomuto účelu v projektu Space Traffic slouží Entity Framework[23]. Díky němu je možné uložit hodnoty uchovávané dočasně v jednotlivých entitách (třídách) do databázových tabulek. V případě Starship Basicu jsou využívány dvě entity, konkrétně *SpaceShip* a *SpaceShipProgram* umístěné v modulu *Core*. Ve třídě *SpaceTrafficContext* jež se nachází v modulu *GameServer*, pak dochází k propojení těchto entit a zmíněných tabulek. Proměnné těchto dvou tříd, jsou pak mapovány na sloupce tabulek, přičemž proměnné, které jsou veřejné, by měli být uloženy do tabulek. Pokud existuje proměnná, která z nějakého důvodu nemá být ukládána, pak musí být explicitně definováno, že se nemá ukládat. Například pro vesmírnou loď není v současnosti ukládána hodnota proměnné Start, proto musí být ignorováno pomocí příkazu *modelBuilder.Entity<SpaceShip>().Ignore(t => t.Start);*.

5.5.1 Tabulka SpaceShipProgram

Tabulka pro ukládání programů vesmírných lodí má pouze pět sloupců, které jsou popsány níže. Naproti tomu entita *SpaceShipProgram* má o několik proměnných více. To je dáno tím, že některé hodnoty jsou vztaženy pouze k právě zkompilovanému či běžícímu programu a nemá proto význam je trvale ukládat. Sloupce tabulky jsou:

- SpaceShipProgramId – identifikátor programu. Tento sloupec je rovněž primárním klíčem.
- UserProgram – textová hodnota programu napsaného pro loď.
- Name – jméno programu napsaného pro loď.
- LastModDate – datum a čas poslední modifikace.
- SpaceShipId – identifikátor vesmírné lodi. Tato hodnota je cizím klíčem z tabulky *SpaceShips* a zajišťuje propojení obou tabulek.

[23] Entity Framework je produkt společnosti Microsoft pro objektově relační mapování

5.6 Modul StarshipBasicInterpreter

V předchozích částech již bylo popsáno GUI pro Starship Basic, zpracování na straně serveru i uložení objektů do databáze. Poslední částí tak zůstává samotná kompilace a interpretace, jež jsou realizovány v nově vytvořeném modulu *StarshipBasicInterpreter*. Tento modul lze pustit i samostatně, ale z hlediska hry samotné je důležité, aby byl propojen s ostatními částmi, konkrétně s modulem *GameServer*. Pokud jde o kompilaci, pak již byla zmíněna třída *CodeGenerator*, přes kterou jsou realizovány další úkony spojené s kompilací a přes kterou je rovněž do serveru vracen výsledek kompilace. Při samotném spuštění programu je pak toto propojení realizováno přes třídu *InterpreterManager*.

Zmíněná třída *InterpreterManager* se nachází přímo v modulu *GameServer* a je obdobou třídy *Interpreter* v modulu *StarshipBasicInterpreter*. Mezi oběma třídami je však několik rozdílů, které jsou odrazem dvou možností spuštění. Pro samostatné spuštění modulu *StarshipBasicInterpreter* se používá třída *Interpreter*, kdežto pokud je realizováno spuštění z herního serveru, pak je použita třída *InterpreterManager*. Obě se starají o interpretaci kódu, ale ve třídě *InterpreterManager* je kromě toho realizováno i rozhodnutí o přerušení dlouho běžícího kódu či o přerušení z důvodu vykonání samostatné akce. Navíc tato třída umožňuje získat informace z herního světa, které jsou ve třídě *Interpreter* nahrazeny defaultními hodnotami.

Samotný modul *StarshipBasicInterpreter* je rozdělen do několika částí (složek). Těmi jsou *Application*, *Compilation*, *Constants*, *Interpreter*, *Memory* a *ProgramCode*. Složka *Application* obsahuje jedinou třídu *StarshipBasicInterpreter*, která je určena k samostatnému spuštění tohoto modulu. Ve složce *Compilation* je několik tříd, které jsou spojeny s kompilací programu. Ty zajišťují lexikální analýzu, obsahují definice jednotlivých chyb, které kód může obsahovat, seznam chyb, které jsou v sestavovaném kódu, informace o skocích, seznam skoků obsažených v kódu a definici všech povolených symbolů. Dále je zde ještě ve složce *Generators* několik tříd pro generování instrukcí neterminálních výrazů z gramatiky, například výrazů, funkcí, faktorů, příkazů či termů. Ve složce *Constants* jsou dvě třídy obsahující definice konstant pro instrukce a výjimky. Složka *Interpreter* je určená k interpretaci spuštěného kódu a obsahuje celkem tři třídy. První z nich je již dříve zmíněná třída *Interpreter* a zbylé dvě jsou určeny ke zpracování výjimek zachycených za běhu programu. Složka *Memory* obsahuje třídy pro definici konstant, proměnných, polí, výčet podporovaných typů proměnných a třídu pro práci s pamětí. Poslední složka *ProgramCode* pak obsahuje třídy pro definici programového kódu, definici instrukcí, seznam povolených instrukcí a seznam podporovaných operací.

6 Testování projektu Space Traffic

Testování je ve Space Trafficu základním prostředkem pro zajištění kvality a je realizováno 4 různými typy testů, které budou popsány v následujících kapitolách. Jejich důležitost je nutné zmiňovat především studentům, kteří se na vývoji podílejí prostřednictvím semestrálních prací. Pro tyto studenty je většinou důležité především realizovat samotnou implementaci a často se testováním nezabývají. To je způsobené především tím, že se jedná převážně o studenty bakalářského studia, kteří se setkávají s pracemi menšího rozsahu, u nichž testování není tak důležité. Naproti tomu Space Traffic je rozsáhlý projekt, u kterého je testování nevyhnutelné. Proto je nezbytné studenty na nutnost testování pravidelně upozorňovat.

6.1 Jednotkové testy

Celý produkt Space Traffic je složen z devíti modulů, z nichž tři jsou určeny výhradně pro testování. Tyto testovací moduly jsou vytvořeny pro testování tří základních modulů hry. Jsou jimi moduly *Core.Tests* určený pro testování modulu *Core*, *GameServer.Tests* pro testování modulu *GameServer* a modul *GameUi.Tests* pro testování modulu *GameUi*. V každém z těchto testovacích modulů je navíc dodržována stejná adresářová struktura jako v modulu, pro který je testovací modul vytvořen.

Samotné jednotkové testování probíhá ve vývojovém prostředí *MS Visual Studio*, které poskytuje dostatečné možnosti pro tento typ testů. Nový test se vytváří přímo ve vybraném modulu, přičemž existuje několik možností jak jej vytvořit. Vždy je však nutné kliknout pravým tlačítkem myši na vybraný modul, případně na složku, do které má být test přidán a v zobrazeném kontextovém menu zvolit možnost *Add* (přidat). První je možnost, kterou lze vybrat je *New Test…* (nový test). Při zvolení této možnosti se otevře nové okno, v němž jsou na výběr 4 typy tříd (*Basic Unit Test*, *Ordered Test*, *Unit Test* a *Unit Test Wizard*). Při výběru třídy *Basic Unit Test* či *Unit Test* je vytvořena nová testovací třída, která v obou případech obsahuje jednu prázdnou testovací třídu s defaultním názvem *TestMethod1*. Rozdíl je však v tom, že při zvolení možnosti *Basic Unit Test* neobsahuje nově vytvořená třída nic dalšího, kdežto při zvolení možnosti *Unit Test* je navíc generován prázdný bezparametrický konstruktor, proměnná *testContextInstance* pro kontext dané testovací třídy a vnořená třída *TestContext* s metodami pro nastavení a získání kontextu. Při výběru *Ordered Test* se nevytváří nový jednotkový test, ale posloupnost testů. Tímto způsobem je možné vybrat pouze zvolené testy, které se vykonají v přesně určeném pořadí. Seřazený test je navíc možné přidat přímo výběrem možnosti *Ordered Test* v kontextovém menu. Spuštění seřazených testů však dosud ve Space Trafficu nebylo použito. Poslední volba *Unit Test Wizard* umožňuje opět vytvořit jednotkový test. Tentokráte však není vytvořena nová třída, ale zobrazené okno s přehledem všech modulů, v nichž se dá zvolit libovolná třída, pro niž má být test vytvořen. Tato možnost je optimální, jelikož oproti dříve zmíněným možnostem *Basic Unit Test* a *Unit Test* je automaticky vytvořena testovací třída, která

má jméno složené z názvu vybrané třídy. Hlavní výhodou však je, že třída již obsahuje patřičné testovací metody. Stejného efektu lze rovněž docílit výběrem možnosti *Unit Test* v kontextovém menu.

Všechny testovací třídy ve Space Trafficu používají knihovnu *Microsoft.VisualStudio.TestTools.UnitTesting*. Tato knihovna poskytuje několik důležitých atributů, které jsou použity ve všech jednotkových testech. Každá testovací třída je označena atributem *[TestClass]* a jako každá třída obsahuje vlastnosti a metody. Metody v každé třídě jsou určeny buď pro nastavení či vytvoření všech hodnot a objektů, které jsou pro daný test potřebné, nebo pro testování. Metody, které jsou určené pro testování, jsou pak označené atributem *[TestMethod]*. Pokud má být některý test z testování dočasně vyřazen, lze jej označit atributem *[Ignore]*. Tento atribut lze vložit jak před třídu, pak budou ignorovány všechny testovací metody, tak před testovací metodu, pak bude z testování vyřazena jen tato metoda. Dále lze nastavit inicializaci testovací třídy atributem *[TestInitialize]*, činnosti, které se mají provést po skončení každé testovací metody (nejčastěji se jedná o úklid), a činnosti, které se mají provést po skončení poslední testovací metody v dané třídě. Činnosti prováděné po skončení testovací metody se označují atributem *[TestCleanup]* a činnosti prováděné pro skončení poslední testovací metody atributem *[ClassCleanup]*.

Výše uvedená knihovna také obsahuje třídu *Assert*. Z této třídy je v současnosti využíváno několik základních metod, které slouží k vyhodnocení výsledku testů. Pro zjištění zda instance neukazuje na žádný objekt (neukazuje na *null*) se využívá metoda *IsNotNull* a pro opačné ujištění, že instance ukazuje na *null* se používá metoda *IsNotNull*. Obě tyto metody jsou přetížené a mohou být volány s různým počtem parametrů. Ideální je využít možnost s dvěma parametry, kdy prvním parametrem je testovaný objekt a druhým zpráva zobrazená pokud daný předpoklad neplatí. Tuto zprávu je možné přidat do všech metod ze třídy *Assert* a ve Space Trafficu je této možnosti hojně využíváno, aby bylo možné lépe zjistit, k jakému problému došlo. Dalšími použitými metodami jsou *IsTrue* respektive *IsFalse*. Ty jsou používány k ověření, že dané tvrzení je/není pravdivé, nebo zda je počet objektů například v seznamu shodný s očekávaným počtem nebo rozdílný. Obě tyto metody jsou opět používány s dvěma parametry, přičemž prvním je testovaná podmínka či booleovská proměnná a druhým chybová zpráva. Metoda *AreEquals* slouží k porovnání dvou objektů a je v projektu používaná jak ve variantě se třemi parametry tak ve variantě se dvěma parametry. Vždy však obsahuje dva porovnávané objekty, předané jako první a druhý parametr. Třetí parametr, který je použit jen v některých voláních, je opět chybová zpráva.

V některých testech je místo třídy *Assert* použita třída *Debug*, která je poskytována knihovnou *System.Diagnostics*. Tato třída však neposkytuje tolik možností a jsou z ní využity jen dvě metody, kterými jsou *Assert* (slouží k ověření podmínky, která je prvním z parametrů, a v případě, že podmínka neplatí je vypsána chybová zpráva, která je uvedená jako druhý parametr) a *Equals* (slouží k porovnání dvou instancí na

shodnost). Právě z důvodů menší flexibilnosti byl tento způsob testování použit pouze v dříve psaných testech a v novějších se objevuje jen využití třídy *Assert*.

V každém testovacím modulu je vytvořeno několik testů, které je možné opakovaně spouštět. Při prvotním spuštění je však možné spustit pouze všechny testy z daného modulu (projektu). Spustit jeden test, respektive testovací třídu, samostatně v tomto bodě nejde. To je možné až po skončení testů. Testy se navíc spouštějí v takzvaném debugovacím režimu, takže je možné do testů vložit debugovací body, na kterých se vykonávaný test pozastaví. Pak je možné test procházet po krocích a lépe tak kontrolovat jeho průběh. Nicméně vzhledem k tomu, že jsou spouštěny všechny testy najednou, je lepší tyto debugovací body při tomto spouštění nepoužívat a vložit je až ve chvíli, kdy testujeme konkrétní test, který neskončil dle očekávání.

Jakmile všechny testy doběhnou, jsou zobrazeny výsledky, přičemž u každého testu se dají zobrazit podrobnosti o jeho průběhu. To je vhodné především u testů, které skončili s nějakou chybou, jelikož je možné zobrazit detail chyby a snáze zjistit k jakému problému došlo. V tomto bodě lze rovněž kromě spuštění všech testů, vybrat jen některé testy, které mají být znovu spuštěny. To se provede zaškrtnutím příslušných testů a výběrem možnosti spustit pouze zaškrtnuté testy. Testy, které skončili s chybou, navíc není nutné zaškrtávat, jelikož jsou zaškrtnuté automaticky. Navíc testy mohou být spuštěny buď v debugovacím režimu, nebo v klasickém režimu, při němž jsou debugovací body přeskočeny.

Výsledky testů je navíc možné exportovat do souboru. Při exportu výsledků tak vznikne speciální soubor *Visual Studio Test Results File* s příponu *trx*. Tento soubor je ve formátu XML a obsahuje informace nejen o výsledcích testů, ale také například o vývojovém prostředí. Ukázka tohoto souboru je v příloze C. Soubor s výsledky testů je vhodné pojmenovat výstižně, například tak aby název obsahoval informaci kým nebo na jakém počítači byl proveden a datum a čas provedení testů. Export výsledků testů v současné době nebyl využíván, ale pro další vývoj by mohl být užitečný. Vzhledem k tomu, že je celý projekt uchováván na úložišti, ke kterému je přistupováno přes nástroj SVN, mohli by být uchovávány výsledky v dlouhodobém horizontu a bylo by možné sledovat měnící se počet testů daného modulu a výsledky testů, což by mohlo vést k snazšímu odhalení doby, kdy problém vznikl. Ideální možností by mohlo být vytvoření samostatné složky v každém modulu, do níž by se ukládali výsledky testů.

6.1.1 Modul *Core.Tests*

Modul *Core.Tests* obsahuje testy pro testování funkčnosti modulu *Core*. Všechny tyto testy jsou zaměřeny výhradně na testování práce s daty ve formátu XML. Konkrétně s jejich generováním a kontrolou načtených dat. Celkově je v rámci tohoto modulu v současnosti vytvořeno 11 tříd, které obsahují celkem 25 testovacích metod. Všechny tyto třídy jsou umístěny ve složce *Data*. Právě v tomto modulu jsou použity obě výše zmíněné varianty pro vyhodnocení testů, tedy použití metod z třídy *Debug* i z třídy *Assert*.

6.1.2 Modul *GameServer.Tests*

Modul *GameServer.Tests* obsahuje testy určené pro testování funkčnosti modulu *GameServer*. V tomto modulu je největší množství testovacích metod, přičemž ne všechny jsou aktivně používány. Všechny aktivní testy (celkem 75) jsou napsány v 10 testovacích třídách umístěných ve složce *Dao*. Zbylé testy napsané pro testování akcí a služeb nejsou v současnosti používány.

Aktivní testy v tomto modulu se zaměřují výhradně na práci s persistentními daty, která jsou ve Space Trafficu ukládána do databáze. Testy jsou určeny pro operace získávání (get), aktualizace (update), vložení (insert) a mazání (remove) dat. Aby nedošlo k porušení dat určených pro hru, je pro testování vytvářená samostatná databáze, která je po skončení každé testovací třídy odstraněna. Konfigurace pro nastavení testovací databáze je v konfiguračním souboru *App.config*.

Většina z testů realizovaných v tomto modulu je zaměřena na testování správného průběhu, ale jsou zde zahrnuty i testy pro testování chybných dat. Testy s chybnými daty jsou používány ve velké většině pro operaci vkládání nových dat do databáze, ale jsou zde i testy s chybnými daty pro update. Chybnými daty jsou myšlena taková data, která mají například v řetězcové proměnné uložený řetězec, který je delší než je maximální délka řetězce, který je možné do sloupce uložit. Dalším příkladem chybných dat je nevyplnění hodnoty proměnné, která tak ukazuje na *null*, přičemž pro daný sloupec tabulky je nastaveno, že nesmí obsahovat prázdné hodnoty. V testech pro vložení chybných dat, je pak testováno, že není nový objekt do tabulky uložen.

Kromě testovacích metod však všechny testovací třídy pro testování databázových operací obsahují i další metody. První z nich je inicializace (uvedená atributem *[TestInitialize]*), ve které probíhá před každým testem uložení souvisejících objektů do databáze. Protože například program pro vesmírnou loď je vztažen k vesmírné lodi, kterou předtím musel hráč koupit na nějaké základně, jsou v této metodě všechny tyto objekty (loď, hráč a základna) uloženy do databáze. Dále po každém testu dochází v metodě *CleanUp* k vymazání souvisejících objektů z databáze. Nakonec je po skončení všech testů z dané testovací třídy zavolána metoda *DropDatabase*, která odstraní existující databázi.

6.1.3 Modul *GameUi.Tests*

Modul *GameUi.Tests* obsahuje testy pro testování funkčnosti modulu *GameUi*. Je zde jen jedna testovací třída (umístěná ve složce *Utils*), jež má celkem 5 testovacích metod. Ty jsou určeny pro testování načítání obsahu importovaných javascriptových souborů. Konkrétně k testování různého nastavení cesty k souborům a samotnému načtení dat z importovaných javascriptových souborů.

6.2 Funkční testy

Funkční testy jsou realizovány ve Space Trafficu formou manuálního klikání na jednotlivé prvky GUI. Vzhledem k tomu, že ve Space Trafficu nejsou definovány role

testerů a neexistuje tak žádný specializovaný tester, nebyl vytvořen žádný scénář pro tento typ testování. Každý vývojář je tak zodpovědný za ověření funkčnosti části, na jejímž vývoji se podílel. Vzhledem k tomu, že danou část vyvíjí, měl by vědět, jakým způsobem funguje. V případě, že se jedná o vývoj v rámci semestrální práce, je kontrola prováděna také vedoucími studenty, kteří v průběhu vývoje kontrolují, že studenti pochopili, co mají realizovat a jakým způsobem. Kontrola je tak v tomto případě prováděna minimálně vývojáři dané části a vedoucími studenty. Pokud je vývoj realizován v rámci kvalifikačních prací, pak toto testování provádějí jen samy vývojáři.

Tímto způsobem testování je však možné odhalit jen určitou část chyb ve funkčnosti produktu. Vzhledem k tomu, že se jedná o hru, která není vytvářena „na objednávku“, není možné provádět kontrolu funkčnosti se zákazníkem. Proto byly rovněž provedeny play testy a testy použitelnosti, které by měly přispět k odhalení dalšího typu chyb spojených s funkčností.

6.3 Play testy

Jak bylo uvedeno v třetí kapitole, je důležité získat zpětnou vazbu od hráčů, kteří mohou přispět významnou měrou k zvýšení kvality hry, protože pokud se hra nebude líbit samotným hráčům a nebudou ji hrát, přichází veškeré vynaložené úsilí vniveč. Proto bylo na konci letního semestru minulého akademického roku rozhodnuto, že budou připraveny play testy hry, přestože v té době byla připravena spíše architektura a z propracovaného game designu hry byla realizována pouze malá část. Většina vývoje spojeného s hratelností hry tak probíhala v zimním a letním semestru tohoto roku. Z tohoto důvodu také byla patrně autorem této práce podceněna příprava play testů, jelikož autor této práce chtěl připravit testy až v době, kdy bude implementována větší část z této hratelnosti. Jelikož však docházelo k různým průtahům s implementací i problémy s nasazením na server byly samotné testy realizovány až na začátku června.

6.3.1 Výběr playtesterů

Účelem hry je zvýšit zájem potenciálních zájemců o studium na Fakultě aplikovaných věd, potažmo přímo na katedře informatiky. Proto se jako logická volba nabízí vybírat mezi těmito uchazeči (převážně studenty středních škol). Jelikož fakulta každoročně před začátkem letního semestru pořádá akci DOD, jíž se účastní právě studenti středních škol, byla na tento den připravena speciální verze hry, která byla studentům prezentována autorem této práce a Janem Dyrczykem. Studenti se tak mohli seznámit s hrou a registrovat se. Tímto způsobem tak byla získána první skupina playtesterů, která činila 20 jedinců. Tato skupina testerů viděla hru, nicméně někteří si vyzkoušeli pouze demo, které mělo se samotnou hrou společný jen velmi malý základ. Zbylý zájemci, kteří viděli aktuální verzi hry, pak mohli posoudit rozdíl této verze a testované verze hry. Nutno podotknout že obě verze hry se v některých částech výrazně lišily.

Další skupinu pak vytvořili studenti, kteří na Fakultě aplikovaných věd již studují. Ti byli získání v průběhu letního semestru při přednáškách na předmětech KIV/ZSWI a KIV/ASWI. Během krátké prezentace byl těmto studentům představen celý projekt

včetně hry a nabídnuta spolupráce na playtestingu aktuální verze hry. Poté byl studentům rozeslán list papíru, na nějž zájemci mohli napsat svůj email. Dále jim byly ponechány lístky s kontaktem na autora této práce, pokud by se rozhodli pro účast později. Díky tomu bylo získáno dalších 16 zájemců. Mezi nimi byli studenti, kteří hru nikdy neviděli i studenti, kteří se podíleli na vývoji některých částí hry.

Díky výše popsaným aspektům tak byly do výsledné skupiny testerů zahrnuti jednak testeři s „čerstvým“ pohledem na hru, tak i testeři, kteří mohli porovnat, jak se hra posunula.

6.3.2 Vývoj verze pro testy

Verze hry určená pro testování vycházela z aktuální verze hry. Nicméně vzhledem k tomu, že stále probíhal vývoj, bylo nutné vytvořit novou branch pojmenovanou *play_and_usability_testing*. Jak název napovídá, v této vývojové větvi probíhaly úpravy stávající verze hry nejen pro herní testy, ale také pro testy použitelnosti (tyto úpravy jsou popsány v kapitole 6.4). V případě úprav pro herní testy šlo o vytvoření scénáře a dotazníku ve hře. S tím byla spojena také úprava některých dalších objektů, například objekt hráče, aby bylo možné měřit čas strávený ve hře, či jednotlivé úkoly, které museli být v rámci scénáře splněny a bodování za splnění těchto úkolů.

6.3.3 Seznam otázek

Seznam otázek pro playtesting byl skládán se záměrem získat co nejvíce důležitých informací. Jedna skupina otázek obsahovala základní obecné otázky, které je možné uplatnit na libovolnou hru. Tato část otázek byla vybrána, jelikož se jednalo o první play testy celé hry a měla přinést odpovědi týkající se například tématu hry či zábavnosti. Další skupina otázek se zaměřovala na jeden ze stěžejních prvků celé hry, kterým je programovatelné ovládání lodí. Navíc byla přidána otázka týkající se anglického popisu herních prvků. Herní prvky jsou totiž psané výhradně anglicky (jedinou výjimku tvoří nápověda k jazyku Starship Basic, která je psána česky). Jak se ovšem ukázalo na DOD, měli někteří studenti středních škol problémy s orientací ve hře způsobenou nedostatečnou znalostí anglického jazyka. Proto byla přidána otázka, která měla poskytnout odpověď na to, zda bude nutné hru rozšířit o české popisy prvků.

Výsledkem byl seznam otázek, který měl více než 30 otázek. Proto byl z důvodu přílišného zatížení testerů uvedeného v kapitole 3 redukován na konečných 13 otázek seznam otázek je v příloze D) a na konci byl ponechán prostor pro doplňující informace, kam mohli testeři napsat další poznatky získané při hře.

6.3.4 Kde testy probíhaly

Nejčastějším místem, kde play testy probíhají, je laboratoř, ve které může být průběh testů dobře sledován a nahráván. Tento způsob byl však kvůli různým faktorům zamítnut. Prvním problém bylo sehnání laboratoře či místnosti s dostatečným množstvím počítačů. Tu by bylo zřejmě možné v rámci univerzity zajistit, ale přesto by zde byly další překážky. Jednou z nich je nahrávání průběhu testů, které by bylo možné jen s omezenými možnostmi, ale které by se dalo v nejhorším případě zcela vynechat.

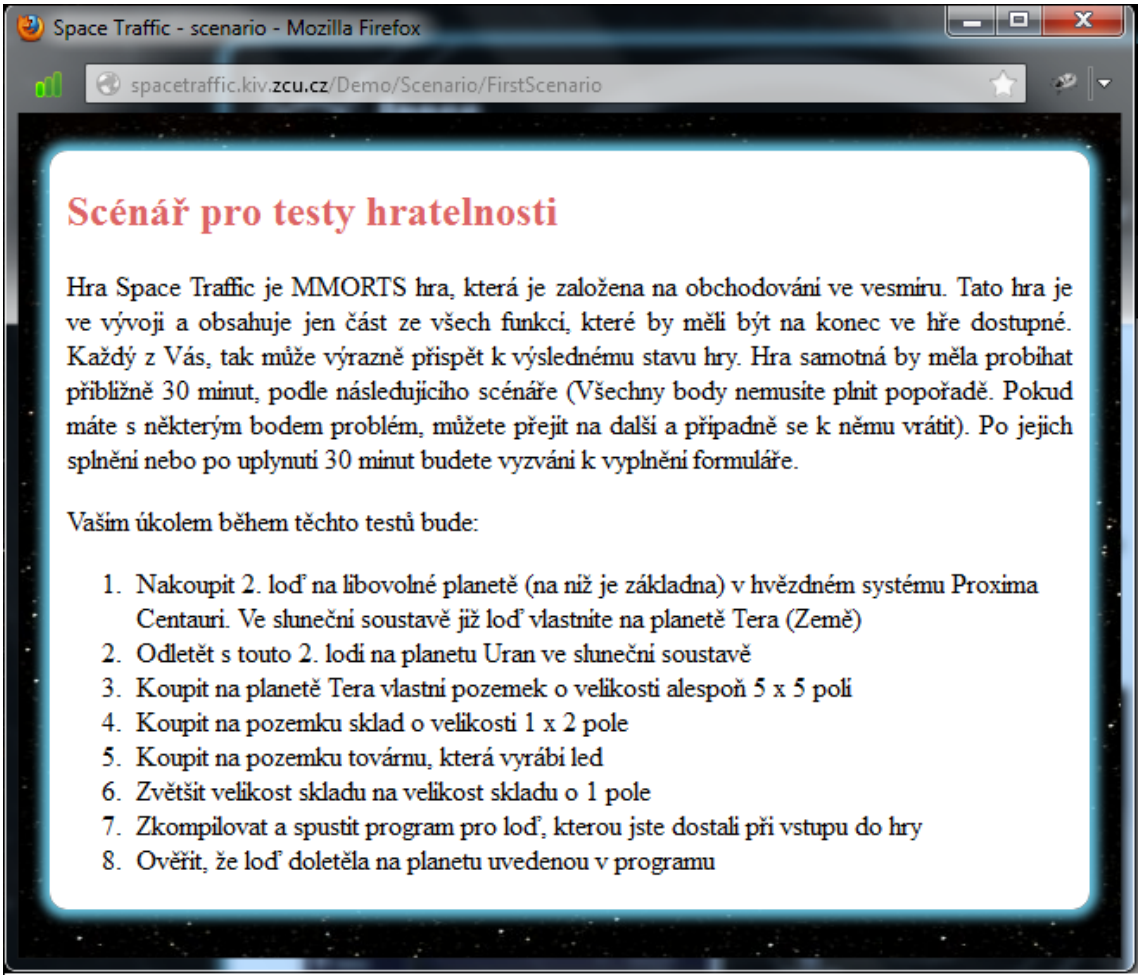
Dalším problémem by byla časová synchronizace dostupnosti místnosti, playtesterů a autora této práce. To by bylo problematické především u studentů středních škol, kteří nezřídka bydlí či studují větší počet kilometrů od univerzity a s velkou pravděpodobností by se na testy nedostavili. Ovšem i u studentů univerzity by bylo obtížné najít čas, který by vyhovoval alespoň větší části studentů.

Z množství dalších možností byla nakonec vybrána realizace přes internet. Jak je popsáno v kapitole 3 má tento způsob testů své výhody i nevýhody. Pro účely Space Trafficu však bylo jedním z rozhodujících faktorů to, že hra je určená pro více hráčů a hraná prostřednictvím internetu, což je jeden z případů vhodných pro použití této varianty testování. Díky testování přes internet je navíc možné zahrnout více testerů, každý tester může testy provést ze svého počítače a všechny počítače mají na rozdíl od laboratoří rozdílnou konfiguraci. Problémem však je, že není možné vnímat pocity, které testeři při hře mají, slyšet o čem právě přemýšlejí, pokud by přemýšleli na hlas, nebo se zeptat na doplňující otázky v průběhu testu, když dojde k neočekávané situaci. Není ani možné s nimi osobně probrat otázky a získat tak přesnější odpověď, z níž se dá lépe zjistit, co přesně měl tester na mysli. Přesto pozitiva zvítězila nad negativy a byl vybrán právě tento způsob.

6.3.5 Průběh testování

Po nasazení verze hry určené pro testy na server byl testerům rozeslán email s žádostí o otestování aktuální verze hry. Dle doporučení bylo zmíněno, že hra je stále ve vývoji a uvedeny informace potřebné pro hraní hry, například kde si hru mohou zahrát a které prohlížeče jsou doporučovány. Součástí emailu byl navíc rovněž scénář a dotazník pro případ, že by došlo k neplánovaným komplikacím. Protože studenti byli rekrutování dvěma různými způsoby, byl tento email zaslán ve dvou variantách.

Při testování hry se každý tester musel nejdříve registrovat. Po úspěšné registraci se přihlásil do hry a v novém okně prohlížeče se mu zobrazili základní informace o hře a scénář, podle kterého postupoval, viz obrázek 6.1. Body uvedené v tomto scénáři byly navrženy tak, že tester mohl přeskočit jen některé body, kdežto jiné museli být realizovány postupně. Zaměření bodů ve scénáři přitom bylo zacílené na základní herní prvky této hry, tedy práci s vesmírnými loděmi, s pozemky a budovami, a programem pro vesmírné lodě. Jestliže tester okno se scénářem v průběhu testů zavřel, bylo při jeho další aktivitě opět zobrazeno. Pokud bylo stále otevřené, ale minimalizováno, bylo pravidelně aktivováno, aby se hráč nemusel touto činností zdržovat a mohl se soustředit jen na hru. Toto testování probíhalo, dokud tester nestrávil hrou 30 minut, nebo dokud nedokončil všechny body ze scénáře. Poté byl hráči zobrazen dotazník (tato část je blíže popsána v následující části).



Obrázek 6.1: Scénář pro herní testy

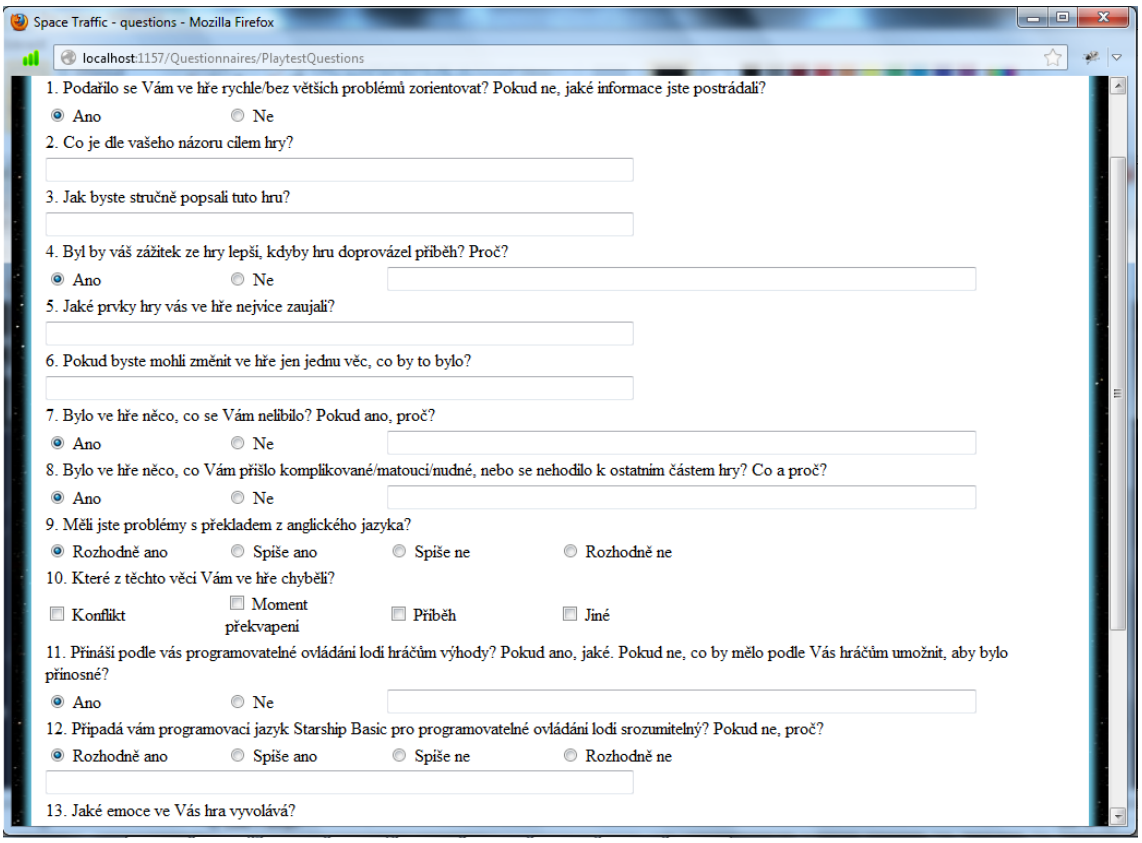
6.3.6 Shromažďování dat

Do hry byl začleněn dotazník, který byl rozdělen na tři části. První část byla věnována poděkování, informacím o dotazníku a otázkám týkajících se hráče. Tyto informace byly určeny pro získání pohlaví, studovaného stupně školy, jejich jazykovým schopnostem, především pro získání informace o úrovni znalosti anglického jazyka a preferencí uživatele (oblíbené žánry her a oblíbené hry). Informace o věku nebyla vyžadována, jelikož všichni studenti univerzity či středních škol většinou patří do stejné věkové kategorie. Druhá část byla věnována samotným otázkám o hře, které byly zmíněny v kapitole 6.3.3. Ukázka této části formuláře je na obrázku 6.2. Závěrečná část pak obsahuje pouze poděkování a informaci o úspěšném dokončení hry.

Aby byl výsledek získaný z testů co nejlepší, byly první dvě části formuláře strukturovaný tak, aby většina otázek a možností byla realizována nabídkou předem definovaných možností, ať už formou seznamů, zaškrtávacích polí či radio buttonů a pouze v případě otázek, u kterých nešlo nabídnout možnosti, a u doplňujících otázek byl testerům poskytnut prostor pro libovolnou odpověď. Navíc jestliže by tester některou z otázek nevědomky či úmyslně vynechal, byl by upozorněn na tuto skutečnost při odeslání dané části dotazníku.

Otázky a odpovědi hráčů uvedené v obou částech dotazníku byly uloženy do CSV souboru ve složce *Questionnaires*. Tento soubor byl pojmenován podle nicku hráče, který byl použit při registraci. Navíc byly výsledky uloženy ještě do společného CSV souboru pro snazší vyhodnocení. Tento soubor byl pojmenován *firstPlayTestsSummary*.

Pokud by navíc došlo k nějakým problémům, byl tento dotazník realizován také v aplikaci *Microsoft Excel* a zaslán jako příloha v emailech pro testery. Tento krok se ukázal jako prozíravý, jelikož k problémům, které budou popsány v další části, skutečně došlo.



Obrázek 6.2: Dotazník - část s otázkami ke hře

6.3.7 Problémy

Jak bylo popsáno výše, během provádění testů se vyskytli problémy s hrou běžící na serveru. První den testů se objevil první problém, když hra opakovaně spadla. Musela být proto provedena nápravná opatření, která však zdržela testy a část testerů zřejmě odradila. Po opravě této chyby testy pokračovali a poté znovu došlo k výpadku a tedy i opakované nápravě. Nicméně některé testy již nebyly úspěšně dokončeny a proto byly vyplněné dotazníky zaslány právě formou vyplněného formuláře v *Excelu*. Tento problém zjevně souvisel s podceněním rozdílného chování verze nasazené v cílovém prostředí a verze testované na lokálním počítači, a s podceněním skutečnosti, že hráč může provést aktivity, na něž není vývojář připraven.

Dalším problémem se stala skutečnost, že velká část testerů (jednalo se pak především o studenty středních škol) se testů vůbec nezúčastnila, což vedlo k malému množství získaných výsledků.

6.3.8 Interpretace dat

Informace o konání play testů byla formou emailu odeslána celkem 35 vybraným testerům (20 studentům Fakulty aplikovaných věd, dále jen FAV, a 15 zájemcům o studium na této fakultě). Z tohoto počtu se však do hry registrovalo pouze 7 testerů (4 studenti FAV a 3 zájemci o studium na FAV), přičemž výsledky byly získány pouze od 4 testerů (všichni studenti FAV). Část těchto výsledků byla získána přímo z formuláře umístěného ve hře (2 výsledky) a část z dotazníku vyplněného v *Excelu* (2 výsledky), který byl součástí emailů odeslaných testerům.

Jelikož byl získán velmi nízký počet výsledků, není možné provést relevantní závěry. Přesto lze vypozorovat 100% shodu u otázky *Byl by váš zážitek ze hry lepší, kdyby hru doprovázel příběh?*, na níž všichni odpověděli *ANO*. Stálo by proto za úvahu zvážit použití podobné otázky v dalším kole play testů pro potvrzení/vyvrácení této shody, případně popřemýšlet o vytvoření vhodného příběhu a jeho začlenění do hry. Další pozitivní informací je, že ¾ respondentů ocenili přínos programovatelného ovládání lodí ve hře. Naopak stejné procento odpovídajících uvedlo, že jim jazyk Starship Basic nepřipadá srozumitelný. Problémem taktéž je, že hypotéza o tom, zda anglický jazyk může ve hře způsobovat hráčům komplikace, nemohla být potvrzena/vyvrácena. Všichni respondenti sice uvedli, že jim anglický jazyk problémy nezpůsobuje, nicméně všichni tito studenti studují bakalářské či navazující studium na FAV a proto lze předpokládat, že anglicky umějí. Naopak od studentů, kteří by s anglickým jazykem mít problémy mohli, protože se s ním na střední škole nepotkali nebo potkali jen velmi zřídka (nejčastěji proto, že měli jako hlavní cizí jazyk zvolen jazyk německý), se výsledky získat nepodařilo.

Ač přínosy této první fáze playtestingu lze hledat jen velmi obtížně, mohl by mezi pozitiva patřit fakt, že je možné implementovat dotazník a scénář přímo do hry. Tím je možné získat odpovědi na všechny otázky, protože hráč je nucen vyplnit všechny údaje. Dotazník je navíc zobrazen ihned po uplynutí času vyhrazeného pro testy, nebo po splnění všech úkolů, takže hráč může vyplnit odpovědi na otázky, dokud má průběh testů v živé paměti. Navíc jsou data na serveru uložena do souboru, který je ukládán do jedné složky a také do souboru, kde jsou výsledky od všech testerů pro konkrétní kolo play testů. Tento společný soubor by měl usnadnit vyhodnocení, případně snadnější porovnání progresu oproti dříve provedeným testům.

6.4 Usability testy

Usability testy neboli testy použitelnosti, byly prováděny společně s play testy. Tyto testy se standardně provádějí v laboratořích, ale jak bylo popsáno v předchozí kapitole, shánění laboratoře a časová synchronizace studentů jsou značnou překážkou. V případě těchto testů však neexistují další doporučované možnosti, jak testy provést. Také je

u usability testů patrnější důsledek ztráty možnosti pozorovat testery v průběhu testů. Proto bylo nutné vymyslet způsob, kterým by bylo možné minimalizovat tento problém. Jako vhodný postup byl proto zvolen systém logování událostí, které zaznamenávají průběh hráčovi hry.

6.4.1 Příprava a průběh testů

Jak již bylo uvedeno, vývoj upravené verze hry probíhal v samostatné vývojové větvi. Do této verze bylo vloženo několik javascriptových funkcí, které zaznamenávali a zpracovávali události provedené uživatelem v GUI. Dále byla do modulu *GameServer* zabudována nová akce *WriteToLog*, která zpracovávala všechny tyto události.

Jelikož v GUI lze provést mnoho rozmanitých úkolů a také proto, že usability testy byly prováděny společně s play testy, při kterých nebylo nutné používat všechny herní prvky, nebyly logy vytvořeny pro celé GUI, ale pouze pro vybrané části. K tomuto kroku bylo přistoupeno nejen z důvodu menšího množství času, ale také kvůli tomu, že soubory s logy by byly zbytečně dlouhé a nepřehledné. Proto byla pozornost zaměřena pouze na části, které byly použity pro play testy. Pro tyto části následně byla vytipována užitečná data, ze kterých by mělo být možné vypozorovat, zda daná část hry způsobuje hráčům problémy či nikoliv.

V modulu *GameUi* byly vytvořeny javascriptové metody s názvem *WriteToLogFile,* které zpracovávali různé parametry předané do těchto funkcí. Vzhledem k tomu, že šlo o různé funkce pro různé události, byl počet těchto parametrů proměnlivý a také se jednalo o parametry různých datových typů. Maximální počet parametrů byl ovšem stanoven na 5. Aby bylo zpracování kontrolérem *LogWriterController*, který byl v modulu také vytvořen, snazší, byly všechny parametry převedené na řetězce a zbylé parametry byly nahrazeny prázdným řetězcem.

V modulu *GameServer* byla vytvořena akce *WriteToLog,* v jejímž těle jsou zpracovány všechny parametry, které jsou poté uložené do CSV souboru. Pokud se jedná o první logovanou událost pro hráče, je vytvořen nový soubor, v jehož názvu je obsaženo uživatelské jméno hráče (například *tester123.csv*). Tento soubor je ukládán do složky *Logs*. V opačném případě je na konec souboru přidána pouze nová řádka. Aby bylo možné lépe sledovat závislosti mezi jednotlivými logy, je log ukládán v následujícím formátu:

- Datum a čas – datum a čas zaznamenání nové události.
- Název akce – například při kupování nové lodi má akce název *CLICK_SHIP_BUY*.
- Parametry – proměnlivý počet parametrů, maximálně však 5. Jednotlivé parametry jsou oddělené středníkem.

6.4.2 Vyhodnocení

Stejně jako v případě play testů je počet výsledků získaných z usability testů velmi nízký. V tomto případě se jedná o data od 7 testerů, od nichž byla zaznamenána alespoň

jedna provedená akce. Jeden z výsledků však obsahuje pouze 4 zaznamenané logy a nemá žádnou vypovídající hodnotu. Zbylých 6 výsledků se stalo základem snahy najít alespoň určitá místa ve hře, která by mohla být z pohledu použitelnosti hráči kritická.

Protože souborů s logy bylo malé množství, byl pro získání výsledků zvolen postup manuálního průchodu každého souboru. Při větším množství výsledků by ovšem tento postup šlo realizovat jen velmi obtížně a bylo by vhodné vyhledávat v souborech konkrétní logy související s akcemi prováděnými v rámci připraveného scénáře, případně se soustředit na problémy, které by hráči popsali do dotazníku pro play testy.

Poznatky získané ze souborů s logy byly zaznamenány do dokumentu *results_june_2013.pdf,* který byl uložen do složky *UsabilityTestingResult* a nahrán na úložiště. Jak vyplívá z tohoto dokumentu, testeři, od nichž byly údaje získány, měli problém především s letem lodi z aktuální planety na cílovou planetu. Přestože nelze brát tento vzorek za dostačující, ukázalo se, že tato část hry může způsobovat problémy. Jak se ukázalo, měli 4 hráči z 6 s touto částí hry problém, přičemž zbylý dva testeři, od nichž byly výsledky získány, se podíleli na vývoji. Dva z těchto testerů zvládli alespoň odletět s lodí z jedné planety ve stejném hvězdném systému na jinou, ale nedokázali proletět do jiného hvězdného systému přes červí díru. Další 2 testeři pak nezvládli vůbec odletět z planety, na které se loď nacházela. U zbylých akcí prováděných v rámci testů použitelnosti se ovšem nedá vypozorovat žádný podobný problém, naopak se zdá, že především nákup lodi by nemusel hráčům způsobovat problémy. Zmíněný dokument obsahuje také tabulku 6.1, která shrnuje data o úspěšnosti dokončení jednotlivých akcí.

Prováděná akce	Úspěšně provedeno	Provedeno se špatným nastavením	Neprovedeno
Koupě lodi	5	1	0
Let lodi	2	2	2
Koupě pozemku	3	1	2
Koupě skladu	2	1	3
Koupě továrny	3	0	3
Změna velikosti skladu	3	0	3
Kompilace a spuštění programu	4	2	0

6.1: Vztah mezi provedenými akcemi a úspěšností provedení

7 Plány pro beta testy hry

Základem pro beta testování hry se mohou stát současné play a usability testy a to i přesto, že nelze ze získaných dat v současné době provést žádné závěry. Při samotných beta testech by však měla být zaměřena větší pozornost na přípravu stabilní verze na serveru a také na propagaci testů. Základem pro play testy bylo sice 35 testerů, ale výsledků bylo získáno málo.

První krokem před začátkem beta testů by mělo být zopakování play a usability testů provedených v červnu tohoto akademického roku. Ideální možností by bylo zopakování na začátku zimního semestru, přičemž povědomí o konání testů by mohlo být rozšířeno při zadávání semestrálních prací. Mezi testery by měli figurovat pouze studenti FAV. Část z nich by měla být rekrutována v prvním ročníku, čímž by se nahradila skupina zájemců o studium. Vzhledem k tomu, že se nepodařilo vyvrátit či potvrdit některé hypotézy a výsledků bylo málo, měli by být použity otázky z tohoto kola playtestingu. Jelikož není nutné připravovat nové testy, nemusel by být s jejich realizací a vyhodnocením problém. V případě, že přibude během zimního semestru do hry nová funkcionalita, která by mohla být otestována, mělo by proběhnout další kolo play a usability testů, přičemž není nutné nahradit všechny otázky v testech, ale měl by být zachován menší počet otázek (maximálně 15) a mělo by se jednat o otázky vztažené přímo ke hře, spíše než o otázky obecnější, které byly v první kole playtestingu také použity. Tyto testy by mohli proběhnout na začátku letního semestru (až po náběru testerů z řad zájemců o studium na DOD). Pokud vývojem dojde k dokončení všech základních prvků hry, mohou být na konci letního semestru provedeny beta testy. V opačném případě lze provést další kolo testování a beta testy realizovat později.

Při provádění dalších kol play a usability testů a při provádění závěrečných beta testů, lze zachovat současnou formu testování prováděnou přes internet a využívající logy. Mimo to by však mohlo být provedeno testování v místnostech na KIV. Toho by se účastnil jen malý počet testerů a bylo by tím možné odbourat nevýhody spojené s testováním přes internet.

Naopak změny by měly nastat v propagaci projektu a testů. Především pro závěrečné beta testy by měly být vyvěšeny v prostorách univerzity plakáty informující o testech, na nichž by byl kontakt na vedoucího projektu, nebo na studenty, kteří za testování ponesou zodpovědnost. Propagaci by zároveň měl napomoci také vznik komunity projektu. Testery lze nadále získávat i formou krátké prezentace na přednáškách.

Před každými testy by mělo dojít k včasnému nasazení nové verze na server a jejímu dostatečnému odzkoušení. Jak bylo ověřeno autorem této práce, nestačí při jejím odzkoušení provést pouze očekávané kroky, ale je nutné se připravit na nejrůznější možné scénáře.

Počet testerů je lepší zvolit vyšší, jelikož se nedá přepokládat, že výsledky budou poskytnuty všemi testery. Na druhou stranu by neměl být příliš vysoký, aby nedošlo ke komplikovanému vyhodnocování. Pokud budou testeři nabírání formou uvedenou výše, mělo by postačovat 20 až 30 testerů.

Jak bylo uvedeno výše, beta testování by mělo proběhnout nejdříve na konci letního semestru příštího akademického roku. Dá se ovšem očekávat, že pravděpodobnější je spíše uskutečnění v průběhu zimního semestru ak. roku 2014/2015.

8 Zhodnocení kvality

Současná verze hry je pokrytá množstvím jednotkových testů. Především v modulu *GameServer.Tests* jsou testy pro značnou část práce s perzistentní vrstvou. Co však chybí, jsou testy pro akce a služby, které by zvláště pro některé složitější akce byly vhodné. Modul *Core.Tests* obsahuje dostatečné množství testů, které pokrývají načítání dat ze souborů ve formátu XML. V posledním testovacím modulu *GameUi.Tests* je pouze jediný test, ale GUI je testováno prostřednictvím funkčních testů. Z pohledu jednotkových testů je tak kvalita více méně zajištěna. Pro vylepšení kvality hry by však měli být přidány testy pro akce a služby a současné testy by měli být pravidelně spouštěny a případně upravovány, pokud dojde ke změnám v související části hry.

Hodnocení herních vlastností a funkčnosti není možné na základě dat získaných z play a usability testů provést. Jak se však ukázalo, bylo by vhodné přemýšlet o změně třech věcí. První, která byla pozorována na DOD je způsobená možnými problémy s překladem z anglického jazyka. Proto by mělo být zváženo vytvoření české verze hry. Hráč by si následně mohl vybrat, zda chce prvky zobrazovat anglicky, nebo česky. Další vylepšení může spočívat v zahrnutí příběhu, který scházel všem testerům, od nichž byly výsledky získány. Poslední věcí je přepracování letu lodi z aktuální planety na cílovou, jelikož tato část způsobovala vzorku hráčů problémy. Všechny tyto úvahy však musí být brány s rezervou a bylo by vhodné je potvrdit/vyvrátit v dalším kole testů.

Problematickou části z hlediska kvality je podpora různých prohlížečů. Vývoj hry byl od počátku soustředěn převážně na prohlížeče *Mozilla Firefox* a *Google Chrome*. Díky tomu některé herní prvky nefungují v ostatních prohlížečích, tak, jak by měly. Problémy byly zjištěny například v prohlížeči *Opera* či *Internet Explorer*. Do budoucna by tak měli být při vývoji testovány alespoň výše zmíněné prohlížeče.

Vývoj by měl i nadále probíhat v rámci kvalifikačních prací (bakalářské a diplomové práce) a semestrálních prací. Jak se ukázalo, je vhodné dodržovat doporučení z dokumentu *Handover Notes*. Mezi základní doporučení patří včasné seznámení vedoucích studentů s projektem, pravidelné (nejlépe týdenní) schůzky se studenty, kteří pracují na semestrálních pracích, pravidelné ověřování, že studenti pochopili zadání a vyhnutí se zadávání duplicitních zadání. Poslední bod může působit přinejmenším kontroverzně, jelikož by se mohlo zdát, že čím více řešení bude, tím snáze se dá vybrat to nejlepší. Jak se však ukázalo v zimním semestru tohoto akademického roku, pokud na témže zadání pracuje více jedinců/týmů musí být vytvořena oddělená vývojová větev a na konci provedeno přidání změn do aktuální vývojové větve. Díky tomu, že během té doby probíhá vývoj i v hlavní vývojové větvi, může dojít během odděleného vývoje (může trvat i 13 týdnů) k velkému posunu mezi oběma verzemi a jejich propojení je poté značně složité.

9 Závěr

Tato práce byla rozdělena do několika bodů, které měly vést k zajištění kvality webové hry Space Traffic. Vzhledem k tomu, že některé z těchto bodů se podařilo splnit lépe a jiné hůře, je hodnocení rozděleno do více částí.

Pokud jde o realizaci jednotkových a funkčních testů, dá se říci, že proběhlo vše podle očekávání. Testy, které byly napsány dříve, byly z části ponechány v nezměněné podobě, a z části upraveny, byly připsány nové testy a celkové pokrytí projektu testy je uspokojující. Funkční testování probíhalo formou průchodu GUI, čímž se podařilo odhalit některé nedostatky, které byly následně opraveny.

Část zadání zaměřená na play a usability testy ovšem skončila pro autora této práce zklamáním. Nejprve bylo rozhodnuto, že testy proběhnou vzhledem k charakteru hry přes internet. Podařilo se připravit upravenou verzi hry, do níž byl zabudován scénář a dotazník pro hráče a rovněž se podařilo získat dostatečné množství testerů. Neočekávaný byl ovšem průběh testů. Nejprve se zdálo, že vše bude probíhat podle plánu, poté však spadla hra nasazená na serveru. Větším problémem ale bylo, že testů se zúčastnilo jen malé množství testerů. Ze získaných výsledků proto nebylo možné učinit žádné průkazné závěry a byly předneseny pouze teorie, které bude nutné v budoucnu potvrdit. Přesto, bylo tímto způsobem alespoň ověřeno, že je možné testy zasadit přímo do hry a testování hry přes internet provést, pokud bude verze hry na serveru stabilní a testování se zúčastní větší množství testerů.

Poslední částí práce bylo přidání programovatelného ovládání lodí, které se podařilo realizovat bez větších problémů. Hráč může v GUI provádět veškeré potřebné operace spojené s programem napsaným pro loď. Pokud je program spuštěn, je možné jej v případě dlouhého běhu přerušit, tak jak bylo plánováno, aby nedocházelo k zahlcení serveru jedním programem. Rovněž se podařilo realizovat funkčnost zajišťující hráči možnost uložit více programů pro jednu loď. Tato funkčnost nebyla dříve zamýšlena, ale pro hráče by měla být přínosem.

Seznam zkratek

BVA – Boundary Value Analysis. Analýza hraničních hodnot.

CT – Comparison testing. Testování srovnáním výsledků.

DOD – Den otevřených dveří. Akce pořádaná Fakultou aplikovaných věd určená pro zájemce o studium na této fakultě.

ECP – Equivalence Class Partitioning. Česky lze tuto metodu nazvat jako rozdělení do ekvivalentních tříd

FAV – Fakulta aplikovaných věd.

FTR – Formal Technical Review. Formální technické revize. Postup používaný k zajištění kvality softwaru.

FURPS – Functionality, Usability, Reliability, Performance and Supportability. Do češtiny lze tato slova přeložit jako funkčnost, použitelnost, spolehlivost, výkon a schopnost podpory.

GEQ – Good Enough Quality. Do češtiny lze přeložit jako dostatečně dobrá kvalita.

GUI – Graphic User Interface. Český výraz je grafické uživatelské rozhraní.

ISO – International Organization for Standardization. Mezinárodní organizace pro definici standardů.

KIV – Katedra informatiky a výpočetní techniky.

MMORPG - Massive Multiplayer Online Role Play Game. Online hra na hrdiny pro více hráčů.

MTBF – Mean Time Between Failures. Střední doba mezi poruchami.

MTTF – Mean Time To Failure. Střední doba do poruchy.

MTTR – Mean Time To Repair. Střední doba do opravy.

SQA – Software Quality Assurance. Skupina pro zajištění kvality softwaru.

Literatura

[1] PRESSMAN, Roger S. *Software engineering: a practitioner's approach*. 5th ed. Boston, Mass.: McGraw Hill, 2000, xxvii, 860 p. ISBN 00-736-5578-3.

[2] GALIN, Daniel. *Software quality assurance*. New York: Pearson Education Limited, 2004, xxvi, 590 p. ISBN 02-017-0945-7.

[3] Introduction to Software Engineering/Quality. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-06-30]. Dostupné z: http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Quality#cite_note-5

[4] ISO 9000 - Quality management. In: *International Organization for Standardization* [online]. [cit. 2013-06-30]. Dostupné z: http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm

[5] KROLL, Per a Philippe KRUCHTEN. *The rational unified process made easy: a practitioner's guide to the RUP*. 5th ed. Boston: Addison-Wesley, c2003, xxxv, 416 p. ISBN 03-211-6609-4.

[6] The Difference Between Waterfall, Iterative Waterfall, Scrum and Lean Software Development (In Pictures!). In: *Agile101* [online]. [cit. 2013-06-23]. Dostupné z: http://agile101.net/2009/09/08/the-difference-between-waterfall-iterative-waterfall-scrum-and-lean-in-pictures/

[7] Software Quality and Testing SOFTWARE QUALITY & TESTING. *Dr. B. N. Subraya* [online]. [cit. 2013-07-08]. Dostupné z: http://www.scribd.com/doc/6944749/Software-Testing

[8] Information technology - Software product quality. In: *The University of New South Wales* [online]. [cit. 2013-07-02]. Dostupné z: http://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf

[9] BIROLINI, Alessandro. *Reliability engineering: theory and practice*. 6th ed. New York: Springer, 2010, xvii, 610 p. ISBN 36-421-4951-0

[10] SCHELL, Jesse. *The art of game design: a book of lenses*. 5th ed. Boston: Elsevier/Morgan Kaufmann, c2008, 489 p. ISBN 01-236-9496-5.

[11] FULLERTON, Tracy, Christopher SWAIN a Steven HOFFMAN. *Game design workshop: a playcentric approach to creating innovative games*. 2nd ed. Elsevier Morgan Kaufmann, c2008, xx, 470 p. ISBN 02-408-0974-2.

[12] CASSELL, Justine a Henry JENKINS. *From Barbie to Mortal Kombat: gender and computer games*. Cambridge, Mass.: MIT Press, c1998, xviii, 360 p. ISBN 02-620-3258-9.

[13] MYERS, Glenford J. *The art of software testing*. 2nd ed. Hoboken: Wiley, c2004, xv, 234 s. ISBN 978-0-471-46912-4.

[14] KANER, Cem, Jack L FALK a Hung Quoc NGUYEN. *Testing computer software*. 2nd ed. New York: John Wiley, 1999, xv, 480 s. ISBN 04-713-5846-0.

[15] MCCABE, Thomas J. A Complexity Measure. *A Complexity Measure* [online]. 1972, č. 4 [cit. 2013-07-28]. Dostupné z: http://www.literateprogramming.com/mccabe.pdf

[16] WALTER, Jan. Návrh procesu Testování elektronických obchodů. Brno 2011. Diplomová práce. Masarykova Univerzita, Fakulta informatiky.

[17] KOCMAN, Richard. Řízení projektu webové hry. Plzeň, 2012. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[18] VOGL, Petr. Podpora vývoje webové hry pro více hráčů. Plzeň, 2012. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[19] ŠTĚPÁNEK, Martin. Architektura a implementace webové hry pro více hráčů. Plzeň 2012. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[20] VICKERS Steven. Sinclair ZX Spectrum: Basic Programming. In: Pete Robinson. World of Spectrum. [online]. 1995 [cit. 2013-07-05]. Dostupné z: http://www.worldofspectrum.org/ZXBasicManual/

Příloha A: Gramatika jazyka Starship Basic

Pro programovací jazyk Starship Basic byla vytvořena následující gramatika:

```
<program> :== <command_group> EOF

<command_group> :== <command> | <command> EOL <command_group>

<command> :== e | <let_command> | DIM <variable>'['<range>']' |
<if_command> | <for_command> | GO TO <label_name> | <label_name>: | REM <any
text until EOL> | PRINT <string_expresion> | <spacetraffic_command>

<let_command> :== LET <num_variable> = <expresion> | LET <string_variable>
= <string_expresion> | LET <num_variable>[] = <num_array_factor> | LET
<string_variable>[] = <string_array_factor>

<if_command> :== IF <expresion> THEN <command> | IF <expresion> THEN EOL
<command_group> EOL ENDIF

<for_command> :== FOR <num_variable> = <expresion> TO <expresion> EOL
<command_group> EOL NEXT <num_variable> | FOR <num_variable> = <expresion> TO
<expresion> STEP <expresion> EOL <command_group> EOL NEXT <num_variable>

<spacetraffic_command> :== FLY TO <string_expresion_list> | LDCARGO
<string_expresion> AMNT <expresion> FROM <string_expresion> | ULDCARGO
<string_expresion> AMNT <expresion> TO <string_expresion> | <buy_command> |
<sell_command> | REPAIR

<buy_command> :== BUY <string_expresion> AMNT <expresion> MAXP <expresion>
| BUY <string_expresion> AMNT <expresion> MAXP <expresion> FROM
<string_expresion>

<sell_command> :== SELL <string_expresion> AMNT <expresion> MINP
<expresion> | SELL <string_expresion> AMNT <expresion> MINP <expresion> TO
<string_expresion>

<string_expresion_list> :== <string_expresion> | <string_expresion>,
<string_expresion_list>

<string_expresion> :== <string> | <string> + <string_expresion>

<string> :== <string_variable> | '"'<ascii chars other than ">'"' |
(<string_expresion>)

<function> :== INT <faktor> | SQRT <faktor> | LENS <string_factor> | LEN
<variable>[] | RND | <spacetraffic_function>

<spacetraffic_function> :== <get_price_function> | <get_bases_function> |
<get_planets_function> | GET STARSYSTEMS | GET FUEL | GET SPACE <string> |
GET FLYTIME <string> | <get_supply_function> | GET EXITS | GET WEAR

<get_price_function> :== GET PRICE <string> | GET PRICE <string> FROM
<string>
```

```
   <get_bases_function> :== GET BASES | GET BASES IN <string>

   <get_planets_function> :== GET PLANETS | GET PLANETS IN <string>

   <get_supply_function> :== GET SUPPLY <string> | GET SUPPLY <string> FROM
<string> | GET SUPPLY <string> IN <string>

   <variable> :== <string_variable> | <num_variable>

   <num_variable> :== <int_variable> | <double_variable>

   <string_variable> :== <name>$ | <name>$'['<index>']'

   <int_variable> :== <name>% | <name>%'['<index>']'

   <double_variable> :== <name># | <name>#'[v<index>']'

   <expresion> :== <and_expresion> | <expresion> OR <and_expresion>

   <and_expresion> :== <logic_expresion> | <and_expresion> AND
<logic_expresion>

   <logic_expresion> :== <logic_term> | <logic_expresion> = <logic_term> |
<logic_expresion> < <logic_term> | <logic_expresion> > <logic_term> |
<logic_expresion> <= <logic_term> | <logic_expresion> >= <logic_term> |
<logic_expresion> <> <logic_term>

   <logic_term> :== <term> | <logic_term> + <term> | <logic_term> - <term>

   <term> :== <logic_factor> | <term> * <logic_factor> | <term> /
<logic_factor> | <term> MOD <logic_factor> | <term> DIV <logic_factor>

   <logic_factor> :== <factor> | NOT <factor>

   <factor> :== (<expresion>) | <function> | <num_variable> | <number> | -
<factor>

   <label_name> :== <name>

   <name> :== <letter> | <letter><letters_and_digits>

   <letters_and_digits> :== <letter_or_digit> |
<letter_or_digit><letters_and_digits>

   <letter_or_digit> :== <letter> | <digit>

   <letter> :== A..Za..z

   <digit> :== 0..9

   <int_number> :== <digit> | <digit><int_number>

   <double_number> :== <int_number>.<int_number> | .<int_number> |
<int_number>.

   <index> :== <expresion>

   <range> :== <expresion>
```

Příloha B: Příkazy jazyka Starship Basic

Seznam příkazů je rozdělen na příkazy, které byly zahrnuty do verze hry platné v době psaní této diplomové práce a příkazy, které je možné použít při rozšiřování hry.

B.1 Implementované příkazy

B.1.1 Elementární příkazy

Tato sekce obsahuje seznam elementárních příkazů vycházejících z jazyka Sinclair Basic.

AND - logická funkce konjunkce

DIM - deklarace pole

DIV – funkce pro celočíselné dělení

ENDIF – ukončení podmíněného příkazu

FOR - cyklus FOR s pevně daným počtem opakování

GO TO - příkaz skoku

IF - podmíněný příkaz

INT - převedení desetinného čísla na celé číslo s oříznutím desetinné části.

LEN - funkce vracející délku pole

LENS - funkce vracející délku řetězce

MOD – funkce pro získání zbytku po celočíselném dělení

NEXT - zvýší řídící proměnnou cyklu FOR na další hodnotu a skočí na jeho začátek

NOT - logická funkce negace

OR - logická funkce disjunkce

PRINT - vytiskne požadovanou hodnotu nebo zprávu

REM - komentář v kódu

RND - vrací náhodné číslo v intervalu <0,1)

SQRT - matematická funkce (druhá) odmocnina

STEP - velikost kroku inkrementace řídící proměnné cyklu FOR.

THEN – uvádí počátek výchozí větve podmíněného příkazu, viz příkaz IF

TO - horní mez pro řídící proměnnou cyklu FOR.

B.1.2 Příkazy pro akce lodí

FLY TO – příkaz pro let lodi na cílovou základnu. Syntaxe - FLY TO *jméno hvězdného systému*, *jméno planety*, *jméno cílové základny*.

LDCARGO – příkaz pro naložení požadovaného množství zadaného druhu zboží ze specifikovaného zdroje do nákladového prostoru vesmírné lodi. Syntaxe - LDCARGO *jméno zboží* AMNT *množství* FROM *jméno zdroje*.

ULDCARGO – příkaz pro vyložení požadovaného množství zadaného druhu zboží z nákladového prostoru vesmírné lodi do specifikovaného cíle. Syntaxe - ULDCARGO *jméno zboží* AMNT *množství* TO *jméno cíle*.

BUY – příkaz pro koupi specifikovaného množství určeného druhu zboží, jehož cena nepřekračuje maximální uvedenou cenu. Volitelně lze zadat i jméno hráče, od něhož se má zboží nakoupit. Syntaxe - BUY *jméno zboží* AMNT *množství* MAXP *maximální cena* [FROM *jméno hráče*].

SELL - příkaz pro prodej specifikovaného množství určeného druhu zboží, jehož cena je alespoň stejně vysoká jako minimální uvedená cena. Volitelně lze zadat i jméno hráče, kterému se má zboží prodat. Syntaxe - SELL *jméno zboží* AMNT *množství* MINP *minimální cena* [TO *jméno hráče*].

REPAIR – příkaz pro opravu vesmírné lodi před jejím odletem ze základny. Tato funkce nemá žádné parametry.

B.1.3 Příkazy pro získávání informací o herním světě

GET STARSYSTEMS - vrací pole s názvy všech hvězdných systémů, které v herním světě existují.

GET EXITS – vrací pole názvů hvězdných systémů, do kterých lze cestovat z aktuálního hvězdného systému, v němž se vesmírná loď nachází.

GET PLANETS - vrací pole názvů planet v hvězdném systému. Pokud není uvedeno nic dalšího, jsou vráceny planety, které se nacházejí v aktuálním hvězdném systému, ve kterém se nachází vesmírná loď. Pokud má funkce parametr IN, vrací planety v požadovaném hvězdném systému. Syntaxe – GET PLANETS [IN *jméno hvězdného systému*].

GET BASES – obdoba GET PLANETS, ale vracíí pole s názvy základen. Syntaxe - GET BASES [IN *jméno hvězdného systému*].

GET PRICE - vrací nejnižší cenu vybraného zboží na základně. Pokud je uveden parametr FROM, je možné specifikovat konkrétního hráče, od něhož má být cena zboží získána. Syntaxe - GET PRICE *jméno zboží* [FROM *jméno hráče*].

GET FUEL - vrací zbývající množství paliva ve vesmírné lodi.

GET SPACE - vrací volné místo v nákladových prostorech pro zadaný druh zboží. Syntaxe – GET SPACE *jméno zboží*.

GET FLYTIME - vrací odhad doby letu do zadaného hvězdného systému.

GET SUPPLY - vrací množství zásob uvedeného druhu zboží. Umožňuje zadat jméno hráče (pro získání množství zásob tohoto hráče), nebo místo ze kterého má být množství zjištěno. Syntaxe GET SUPPLY *jméno zboží* [FROM *jméno hráče*] | [IN *jméno skladu, lodi, továrny*].

B.2 Příkazy použitelné pro rozšíření

Následující sekce uvádí abecedně seřazený seznam elementárních příkazů, které nebyly zahrnuty do aktuální verze hry platné k červnu 2013.

ABS - matematická funkce absolutní hodnota

ACS - matematická funkce arkus kosinus

ASN - matematická funkce arkus sinus

AT - pozice pro tisk pomocí příkazu PRINT

ATN - matematická funkce arkus tangens

CHR$ - vrací řetězec obsahující znak s ASCII kódem číselného parametru

CLS – příkaz pro vymazání lodního deníku

CODE - vrací ASCII kód prvního znaku v řetězci

CONTINUE - pokračování ve vykonávání programu (opak pause)

COS - matematická funkce kosinus

DEF FN - definování uživatelské funkce

EXP - matematická funkce exponenciální funkce

FN - volání definované uživatelské funkce

GO SUB - příkaz skoku s návratem pomocí příkazu RETURN

LN - matematická funkce přirozený logaritmus

LOAD - nahraje a spustí program z hráčova úložiště - syntaxe LOAD *jméno programu*

ON ERR - nastavení příkazu, který se provede při chybě

PAUSE - zastavení provádění programu, lze jej opět ručně spustit

PI - matematická konstanta π

RETURN - návrat z podprogramu (volání GO SUB)

SGN - matematická funkce signum

SIN - matematická funkce sinus

STR$ - převede číslo na řetězec

TAB - tisk tabulátoru

TAN - matematická funkce tangens

VAL - převede řetězec na číslo

Příloha C: Soubor s výsledky unit testů

```
<?xml version="1.0" encoding="UTF-8"?>
<TestRun id="d3627c20-7c36-44c4-84a6-41af308dcf74" name="pavel@PAVEL-NOTAS 2013-08-03 13:21:29" runUser="Pavel-notas\pavel" xmlns="http://microsoft.com/schemas/VisualStudio/TeamTest/2010">
  <TestSettings name="Local" id="d61ea8d4-11b8-4122-ae2d-406b0f296996">
    <Description>These are default test settings for a local test run.</Description>
    <Deployment runDeploymentRoot="pavel_PAVEL-NOTAS 2013-08-03 13_21_29">
      <DeploymentItem filename="C:\Program Files\NLog\.NET Framework 4.0\NLog.dll" />
      <DeploymentItem filename="C:\DotNet\Diplomka\SpaceTraffic\trunk\StarshipBasicInterpreter\bin\Debug\StarshipBasicInterpreter.exe" />
      <DeploymentItem filename="C:\DotNet\Diplomka\SpaceTraffic\trunk\bin\Debug\SpaceTraffic.Core.dll" />
    </Deployment>
    <Execution>
      .
      .
      .
    </Execution>
  </TestSettings>
  .
  .
  .
  <TestDefinitions>
    <UnitTest name="FactoriesLoadTest" storage="c:\dotnet\diplomka\spacetraffic\trunk\bin\debug\core.tests.dll" id="2578c2b5-921f-b318-cad6-b09c32ce2ea5">
      <Execution id="ea9828ca-666e-4475-a83d-db2548a345a2" />
      <TestMethod codeBase="C:/DotNet/Diplomka/SpaceTraffic/trunk/bin/Debug/Core.Tests.DLL" adapterTypeName="Microsoft.VisualStudio.TestTools.TestTypes.Unit.UnitTestAdapter, Microsoft.VisualStudio.QualityTools.Tips.UnitTest.Adapter, Version=10.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" className="Core.Tests.Data.FactoriesLoaderTest, Core.Tests, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" name="FactoriesLoadTest" />
```

```
      </UnitTest>
      .
      .
      .
    </TestDefinitions>
    <TestLists>
      <TestList name="Results Not in a List" id="8c84fa94-04c1-424b-9868-
57a2d4851a1d" />
      <TestList name="All Loaded Results" id="19431567-8539-422a-85d7-
44ee4e166bda" />
    </TestLists>
    <TestEntries>
      .
      .
      .
    </TestEntries>
    <Results>
      <UnitTestResult executionId="f8a0e4e6-1faa-422d-a12d-a9b1f685d1d6"
testId="55497d82-4a19-17dc-550b-2cb2ad1027a0"
testName="PathPlannerTest_testPath1_speed10" computerName="PAVEL-NOTAS"
duration="00:00:00.3224603" startTime="2013-08-03T13:21:45.6370082+02:00"
endTime="2013-08-03T13:21:45.9960287+02:00" testType="13cdc9d9-ddb5-4fa4-
a97d-d965ccfc6d4b" outcome="Passed" testListId="8c84fa94-04c1-424b-9868-
57a2d4851a1d" relativeResultsDirectory="f8a0e4e6-1faa-422d-a12d-
a9b1f685d1d6">
        <Output>
          <DebugTrace>Test for ship speed: 10
                  [0] x: 0, y: -88, time: 0
                  .
                  .
                  .
                  [10] x: -20,687, y: -50,961, time: 98,723
          </DebugTrace>
        </Output>
      </UnitTestResult>
      .
      .
      .
    </Results>
  </TestRun>
```

Příloha D: Seznam otázek

1. Podařilo se Vám ve hře rychle/bez větších problémů zorientovat? Pokud ne, jaké informace jste postrádali?
2. Co je dle vašeho názoru cílem hry?
3. Jak byste stručně popsali tuto hru?
4. Byl by váš zážitek ze hry lepší, kdyby hru doprovázel příběh? Proč?
5. Jaké prvky hry vás ve hře nejvíce zaujali?
6. Pokud byste mohli změnit ve hře jen jednu věc, co by to bylo?
7. Bylo ve hře něco, co se Vám nelíbilo? Pokud ano, proč?
8. Bylo ve hře něco, co Vám přišlo komplikované/matoucí/nudné, nebo se nehodilo k ostatním částem hry? Co a proč?
9. Měli jste problémy s překladem z anglického jazyka?
10. Které z těchto věcí Vám ve hře chyběli?
11. Přináší podle vás programovatelné ovládání lodí hráčům výhody? Pokud ano, jaké. Pokud ne, co by mělo podle Vás hráčům umožnit, aby bylo přínosné?
12. Připadá vám programovací jazyk Starship Basic pro programovatelné ovládání lodí srozumitelný? Pokud ne, proč?
13. Jaké emoce ve Vás hra vyvolává?