VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

UŽIVATELSKÉ NÁSTROJE PRO PODPORU AUTOMATIZOVANÉHO FUNKČNÍHO TESTOVÁNÍ PROTOTYPŮ VE VÝVOJI

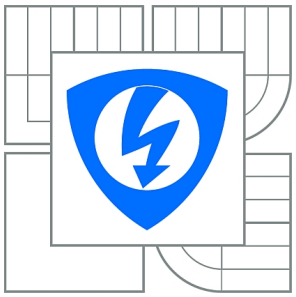
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ŠTĚPÁN GRABOVSKÝ
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

UŽIVATELSKÉ NÁSTROJE PRO PODPORU AUTOMATIZOVANÉHO FUNKČNÍHO TESTOVÁNÍ PROTOTYPŮ VE VÝVOJI

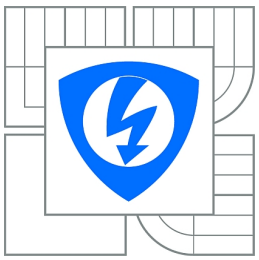
USER TOOLS TO SUPPORT AUTOMATED FUNCTIONAL TESTING OF PROTOTYPES IN PRODUCT DEVELOPMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ŠTĚPÁN GRABOVSKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADOMÍR SVOBODA, Ph.D.
SUPERVISOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Bakalářská práce

bakalářský studijní obor
Teleinformatika

Student:	Štěpán Grabovský	***ID:***	145998
Ročník:	3	***Akademický rok:***	2013/2014

NÁZEV TÉMATU:

Uživatelské nástroje pro podporu automatizovaného funkčního testování prototypů ve vývoji

POKYNY PRO VYPRACOVÁNÍ:

Aplikaci vytvořenou v rámci semestrálního projektu rozšiřte takovým způsobem, aby podporovala všechny možnosti skriptovacího jazyka dle dodané specifikace. Dále v aplikaci eliminujte výstup ze skriptu do konzole a doplňte plně grafické rozhraní pro záznam běhu skriptu. Chování grafického prostředí doladťe na základě požadavků pracovníků vývojového centra. Aplikace bude vytvořena v prostředí .NET a v jazyce C#.

DOPORUČENÁ LITERATURA:

[1] WATSON, Karli. Beginning Visual C# 2010. Indianapolis, IN: Wiley Pub., Inc., c2010, xxxix, 1037 p. ISBN 04-705-0226-6.

Termín zadání:	10.2.2014	***Termín odevzdání:***	4.6.2014

Vedoucí práce: Ing. Radomír Svoboda, Ph.D.
Konzultanti bakalářské práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Předložená bakalářská práce se zabývá vývojem aplikace pro podporu automatizovaného testovaní prototypů ve společnosti Honeywell, divizi ACS v Brně. V rámci práce byl vytvořen počítačový software, který načítá data z předloženého skriptovacího souboru, a na jejich základě provádí testování. Výběr skriptu i zadávání ostatních parametrů je realizováno pomocí grafického uživatelského rozhraní. Průběh a výsledky testování jsou v reálném čase vypisovány v informační kontrole programu. Aplikace dále umožňuje uložit soubor s logovacími informacemi. Program je realizován v programovacím jazyce C#.

KLÍČOVÁ SLOVA

testování, testovací skript, parsování, Honeywell, .NET, C#, interpret, logování

ABSTRACT

This work deals with develop of PC application to support automated testing of prototypes, which are developing in Honeywell International, Inc., ACS division in Brno. There was created a PC software that executes testing process based on input script files. For select script files and specify other necessary information was designed a graphic user interface. Testing information are being presented in a program information control in real-time as the same as results of tests. The application also allow to users save the log file. The program has been realized in C# programing language.

KEYWORDS

testing, script, Honeywell, parsing, .NET, C#, interpret, logging

GRABOVSKÝ, Štěpán *Uživatelské nástroje pro podporu automatizovaného funkčního testování prototypů ve vývoji*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2014. 64 s. Vedoucí práce byl Ing. Radomír Svoboda, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Uživatelské nástroje pro podporu automatizovaného funkčního testování prototypů ve vývoji“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

Štěpán Grabovský

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Radomíru Svobodovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

Štěpán Grabovský

PODĚKOVÁNÍ

Rád bych poděkoval konzultantovi bakalářské práce panu Ing. Jiřímu Macháčkovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

Štěpán Grabovský

odštěpený závod
Honeywell, spol. s r.o. - HTS CZ o.z.
Tuřanka 100/1387, 627 00 Brno
Česká Republika
http://www.honeywell.com

PODĚKOVÁNÍ

Tato bakalářská práce byla realizována u společnosti Honeywell, divize ACS v Brně. Za tuto příležitost patří této společnosti poděkování.

Brno ....

Štěpán Grabovský

OBSAH

SEZNAM OBRÁZKŮ

ÚVOD

Předložená bakalářská práce se zabývá vývojem testovacího prostředí pro společnost Honeywell v Brně, divizi ACS, a to konkrétně vytvořením interpretu, který zpracovává testovací skripty vytvořené v již zastaralém formátu, který chce společnost nahradit novým, a provádí na nich založené testování v kontextu nového testovacího systému. Výsledná aplikace má grafické rozhraní sloužící pro výběr skriptovacího souboru a zadání dalších parametrů, jejich ukládání a načítání a zobrazení průběhu testů v reálném čase. Program je realizován v programovacím jazyce C# platformy .NET.

Softwarové testování aplikovaných vylepšení i nových prototypů je nezbytnou součástí vývojového procesu. Tento typ testování umožňuje simulovat nejrůznější podmínky a situace při relativně nízkých nárocích na finanční i lidské zdroje. Funkční testovací systém je pro vykonávání takových úkonů nezbytný. Základními podmínkami pro jeho vysokou úroveň použitelnosti jsou modulárnost při současném zachování jednoduché a intuitivní ovladatelnosti, nekomplikované zadávání nastavovacích parametrů, samostatnost a zpětná kompatibilita.

Tato práce je součástí rozsáhlého projektu, jehož cílem bylo nahradit soudobé a zastaralé testovací nástroje novým komplexním systémem. Konkrétní aplikace bývalého systému, jež bude výsledkem této práce nahrazena, je především kvůli svému stáří technicky obtížně upravovatelná, nemodulární a uživatelsky nepohodlná.

Konstrukce hlavních kapitol tohoto textu odpovídá chronologickému postupu při zpracovávání projektu. Z tohoto důvodu, a také z důvodu přehlednosti a logické posloupnosti, je tento text rozdělen na tři hlavní části: Úvod do projektu, Pozadí projektu a Struktura aplikace. Kapitola Úvod do projektu je věnována především údajům o použitých technologiích, stručnému popisu společnosti Honeywell a obecnému náhledu na problematiku projektu. V části Kontext projektu je podrobněji popsána podstata celého problému a v sekci Struktura aplikace se nachází celková deskripce programu a programovacích postupů použitých při tvorbě aplikace. Závěrem je pak ještě vložena krátká úvaha o tom, jak by se daly testovací nástroje společnosti rozšířit a vylepšit.

1 ÚVOD DO PROJEKTU

Tato kapitola je věnována obecným tématům souvisejících s celou následnou praktickou prací. Ve většině případů se jedná o fakta snadno dohledatelná v tištěných i elektronických zdrojích, a tak jsou informace o nich zestručněné na nezbytné minimum při zachování smysluplnosti a relevantní informační hodnoty. V následujícím textu jsou zmíněny základní údaje o pojmech ze zadání, konkrétně o prostředí .NET, programovacím jazyce C# a také o společnosti Honeywell, pro kterou byla aplikace vytvářena.

1.1 .NET, C# a Visual Studio

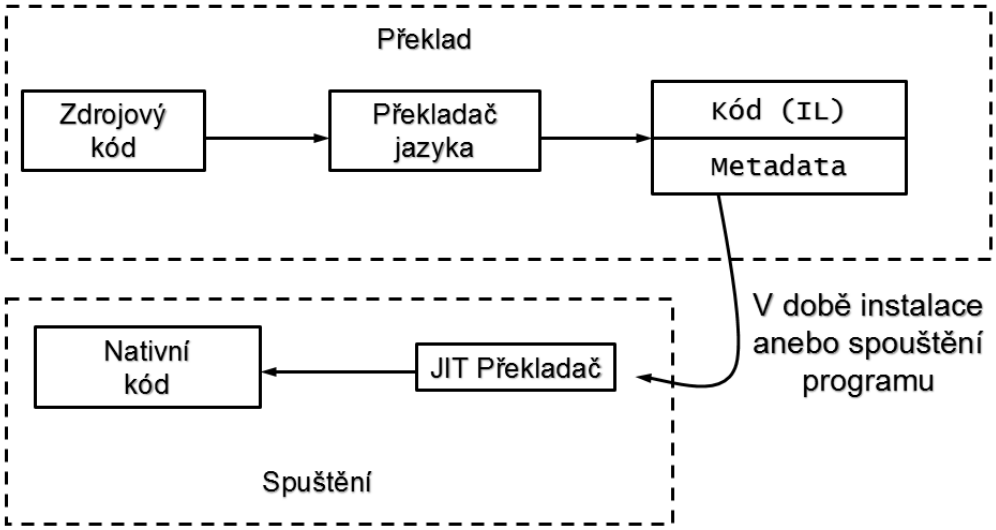
Tato podkapitola je věnována použitým vývojářským technologiím pro psaní počítačových programů. Těmito technologiemi jsou prostředí .NET, programovací jazyk C# a vývojové prostředí Visual Studio 2012 od společnosti Microsoft. Z následujícího textu by měly být zřejmé důvody volby zmíněných technologií, i když o nich bylo rozhodnuto již při zadávání práce. Hlavními důvody výběru jsou především aktuálnost, široká podpora a relativně jednoduché používání při zachování vysokého výkonu volených technologií. Jejich nevýhodou by mohla být skutečnost, že se jedná o proprietární systémy společnosti Microsoft a nejsou tzv. „freeware“. Není-li u věty uvedeno jinak, jsou informace zmíněné v této podkapitole převzaty z [10].

1.1.1 .NET

.NET Framework je prostředí vytvořené společností Microsoft pro vývoj aplikací. Primárně je soustředěn na moderní operační systémy Microsoft, tj. Windows 7 a vyšší a Windows Phone 7 a vyšší, dále Webové technologie a Databáze. Ovšem pomocí technologií třetích stran, známé jsou Mono a Kerberos, je možné .NET aplikace spouštět i vyvíjet na jiných OS. Zajímavý je i oficiální .NET Micro Framework, jenž umožňuje provozovat v .NET vytvořené programy na zařízeních s velmi omezenými zdroji [6].

Platforma .NET v sobě obsahuje dva systémy, jež společně umožnují vytvářet aplikace v různých programovacích jazycích a spouštět je na různých operačních systémech. Prvním ze systémů je velmi rozsáhlá knihovna kódů. Ta v sobě obsahuje nástroje, jež umožnují a usnadňují psaní programů pro různé moduly, Desktop Apps, Web Apps, atd., a definuje základní datové typy, jež jsou používány napříč modulovým spektrem. Tento systém se nazývá Společný typový systém (*Common Type System*).

Tak jak jsou poskytovány výše zmíněné knihovny kódu, .NET Framework obsahuje také jazykově nezávislé běhové prostředí (*Common Language Runtime*), které je zodpovědné za spouštění a provádění aplikací. Tato skutečnost zaručuje přenositelnost vytvořeného programu mezi různými operačními systémy a různými hardwarovými prostředky. Kód napsaný v jazyce C#, anebo v jakémkoliv jiném jazyce podporovaném .NET, je kompilátorem zkompilován do zprostředkovacího jazyka (*Common Intermediate Language*), který je nezávislý jak na OS, tak na použitém jazyce. Tento fakt dokonce umožňuje užití různých programovacích jazyků v rámci jednoho kódu. Kód zkompilovaný do CIL vytvoří v adresářové struktuře tzv. sestavení (*Assembly*). Sestavení obsahuje spustitelný soubor se známou příponou exe, případně přidané dynamické knihovny s příponou dll a soubor s metainformacemi. Při spuštění aplikace je sestavení postoupeno níže v pomyslné struktuře .NET nezávislému běhovému prostředí – CLR. Toto prostředí musí být pro spouštění aplikací na zařízení naistalováno a obstarává komunikaci programu s operačním systémem a hardware. Tento princip umožnuje nejenom vykonávání IL, ale také ochranu před neoprávněným přístupem do paměti, správu paměti (*Garbage Collector*), poskytuje čtení a zápis do souborů, řídí vlákna a mnoho dalšího [9]. Diagram překladu a spouštění je na obrázku1.1.



Obr. 1.1: Překlad a spuštění

1.1.2 C#

C# je jeden z programovacích jazyků nabízených prostředím .NET. Jedná se o objektově orientovaný jazyk vyvinutý a nadále vyvíjený společností Microsoft. Oficiálně

vychází z programovacích jazyků C a C++, ale u spousty metod a postupů se inspiroval jazykem Java [7]. Jeho vývoj započal v Redmondu společně s prostředím .NET, tj. v roce 2000. Jazyk C# je tak vyvíjen přímo na míru .NET a programátorům tak od ostatních jazyků nabízí úplné využití jeho možností. Relativní mládí jazyka C# mu umožňuje poučit se z chyb jeho předchůdců a bez artefaktů tak využívat moderních metod [7].

Od předchůdce C#, jazyku C++, ho odlišují dvě hlavní vlastnosti. První fakt je, že C# implicitně není schopný pracovat přímo s bloky paměti zařízení, o to se stará výše zmíněné CLR. Tento problém lze explicitně překonat pomocí užití klíčového slova unsave. Tato pokročilá programovací technika je ovšem potenciálně nebezpečná, jelikož je s jejím použitím možné neřízené přepisování paměti, včetně kritických systémových bloků. Druhá odlišnost je, že C# je typově zabezpečený (*type-save*) jazyk. Tato skutečnost znamená, že při přetypovávání proměnných musíme v C# užít delších kódových syntaxí. Výhody této vlastnosti jsou však větší robustnost kódu a snazší a účinější debuggování.

Všechny zmíněné výhody programovacího jazyku C# jsou umožněny jeho vnitřní složitostí a vysokou režií. Z tohoto důvodu je pravdivý fakt jeho kritiků, že se nehodí pro časově kritický či extrémně zatěžovaný kód [7].

1.1.3 Visual Studio

Vývojové prostřední Visual Studio je s .NET i C# úzce spojeno a je relativně obtížné najít vývojáře v .NET, který si bez Visual Studia dokáže vývoj představit. V tomto směru se však situace může začít měnit, a to díky projektu Xamarin, který se v tomto v tomto poli působnosti začíná prosazovat. Jeho potenciál je ukryt v možnosti rozšířit .NET prostředí na iOS, operační systémy firmy Apple.

Verze Visual Studia, jakožto programovacího prostředí, byly v minulosti vydávány vždy s novou verzí .NET. Nyní si Visual Studio žije nezávislejším životem a je tak vývojovým prostředím pro velmi široké spektrum programovacích jazyků. Od XML, přes funkcionální jazyky (F#) až po C# či J#. Dále má také implementován vývoj WPF aplikací, moderních, tzv. Store Apps, známějších spíše jako Metro aplikace, databázové systémy, umožňuje dokonce základní simulaci serveru pro vytváření Webových služeb a aplikací.

Jedinou výjimkou je absence nástroje pro tvorbu smyslného instalátoru. Tento problém je řešen limitovanou edicí InstallShield, což bych subjektivně neoznačil za šťastné řešení. Důvody pro toto tvrzení začínají u nepohodlného registrování se na různých serverech a pokračují nepřátelským uživatelským rozhraním.

1.2 O společnosti Honeywell

Společnost Honeywell International, Inc. je dle [8] mezinárodní korporace, která se zabývá výrobou široké škály komerčních i spotřebních výrobků, inženýrskými službami a problémy v leteckém průmyslu od individuálních soukromých spotřebitelů až po vládní organizace. Při troše zjednodušení informací ze [5] lze říci, že její základy jsou založeny na firmách pánů Alberta Butze a Marka Honeywella, kteří své společnosti Electric Heat Regulator Co. a Honeywell Heating Specialty Co. spojily v roce 1927.

V kontextu této práce je důležitá skutečnost, že takto vzniknuvší společnost Minneapolis-Honeywell Regulator Co. měla významné akvizice v oblasti průmyslové kontrolní a indikační techniky. Než se tato společnost v roce 1963 přejmenovala na dnešní Honeywell Inc., stihla se prosadit na nejvyšších příčkách leteckého průmyslu vynálezem prvního systému autopilota (rok 1942) a první komerční meteorologické stanice, v domácích elektrických systémech, termostatem T-86 „Round“ (rok 1953), v elektronických zabezpečovacích systémech, v chemickém průmyslu a neunikly jí ani počátky IT technologií.

1.2.1 Současnost

Dle náhledu společnosti dostupného zde [3] má nyní společnost Honeywell 132 000 zaměstnanců, včetně více jak 22 000 vědců a inženýrů, a působí v 1 300 městech 70 zemí světa. Jejími hlavními sektory působení jsou letectví, výzkum speciálních materiálů a technologií, automatizační a kontrolní technika a dopravní systémy. Nejvýznamnějším odvětvím je vývoj a výroba automatizačních a řídicích systémů. Působení v této oblasti průmyslu přineslo společnosti v roce 2012, i přes celosvětovou krizi, prodej za 15,9 miliard amerických dolarů.

Na oficiálních stránkách společnosti [1] se píše, že Česká republika je klíčovou základnou technologického rozvoje a vývoje společnosti Honeywell v Evropě. Činnost pražské pobočky byla zahájena v roce 1993 a brněnské vývojové centrum následovalo o deset let později. V roce 2006 se brněnské vývojové centrum stalo součástí společenství vývojových center – Honeywell Technology Solutions (HTS), která se nacházejí v USA, Číně, Indii a České republice. Společnost Honeywell má v České republice také dva výrobní závody, a to Honeywell Aerospace Olomouc a Environmental and Combustion Controls v Brně. V současné době zaměstnává společnost Honeywell v České republice více než 4 000 zaměstnanců.

1.2.2 ACS

Divize automatizace a řízení – ACS, vytváří komplexní řešení jak pro běžné domácnosti, tak pro průmyslové budovy. V jejím portfoliu nalezneme elektrické i neelektrické zabezpečovací systémy, systémy pro regulaci tepla v budovách, medicínské senzory a přístroje a další.

1.2.3 Honeywell v Brně

Honeywell v Brně se zaměřuje na produkty v oblastech spalování (řídící ventily pro plynové kotle na ohřev teplé užitkové vody a topení), elektroniky (řídící jednotky plynových a olejových kotlů, zobrazovací zařízení, termostaty a řídící jednotky nových technologií pro alternativní zdroje energie – palivové články a tepelná čerpadla), termoregulace (termostatické hlavice a ventily na topení) a systémů s pitnou vodou (armatury a ventily). Převzato z [4].

1.3 Téma úkolu

Při vytváření nových věcí, při inovování a vylepšování se dá jen stěží vyhnout chybám a je skoro nemožné stvořit produkt, který by byl bezchybný hned od počátku. Ovšem v současném světě je spotřebitel jen málokdy ochotný tolerovat chyby a poskytovat šance na jejich nápravu. V mnoha praktických aplikacích tato možnost ani není možná. Jako příklad by mohl být uveden požární senzor anebo motor dopravního letadla. Je-li kladen důraz na dobrou pověst firmy, či závažné chyby nejsou dovoleny z podstaty výrobku, je nutnost výsledky práce testovat. Při takovémto testování jsou pak simulovány jak běžné situace, tak extrémní, či dokonce pouze teoretické scénáře.

Problematika by mohla být znázorněna na primitivním případě. Nechť je testována řídící jednotka kotle, zdali zapne či vypne ve správný okamžik plamen. Bylo by možné sestavit plně funkční kotel s řídící jednotkou a vyhradit technika, který by celé sestrojení vizuálně kontroloval a zapisoval výsledky. Takovýto styl testování ovšem jen těžko umožňuje zkoumat chování výrobku v neběžných situacích a také pro ověření základní funkčnosti bychom museli vyhradit mnoho času, který je v konečném důsledku roven velkým finančním výdajům.

Zde se otvírá pole působnosti pro automatizované testovací systémy. Nejedná-li se pouze o testování softwaru, ale o sledování chování nějaké hardwarové součástky, například výše zmíněná řídící jednotka kotle, skládá se takovýto systém z více prvků. Řetězec spolupracujících elementů je ve valné většině případů zakončen uživatelským

rozhraním, které testovací technik využívá k zadávání pokynů a testovacímu systému umožňuje poskytovat uživateli zpětnou vazbu.

A právě aplikace umožňující ovládání testovacího procesu je výsledným produktem této práce. Ta je ve společnosti Honeywell, divizi ACS vyvíjena společně s dalšími články testovacího řetězce, který tak prochází kompletní obměnou a modernizací. Celá situace je ovšem zkomplikována potřebou zpětné kompatibility.

2 KONTEXT PROJEKTU

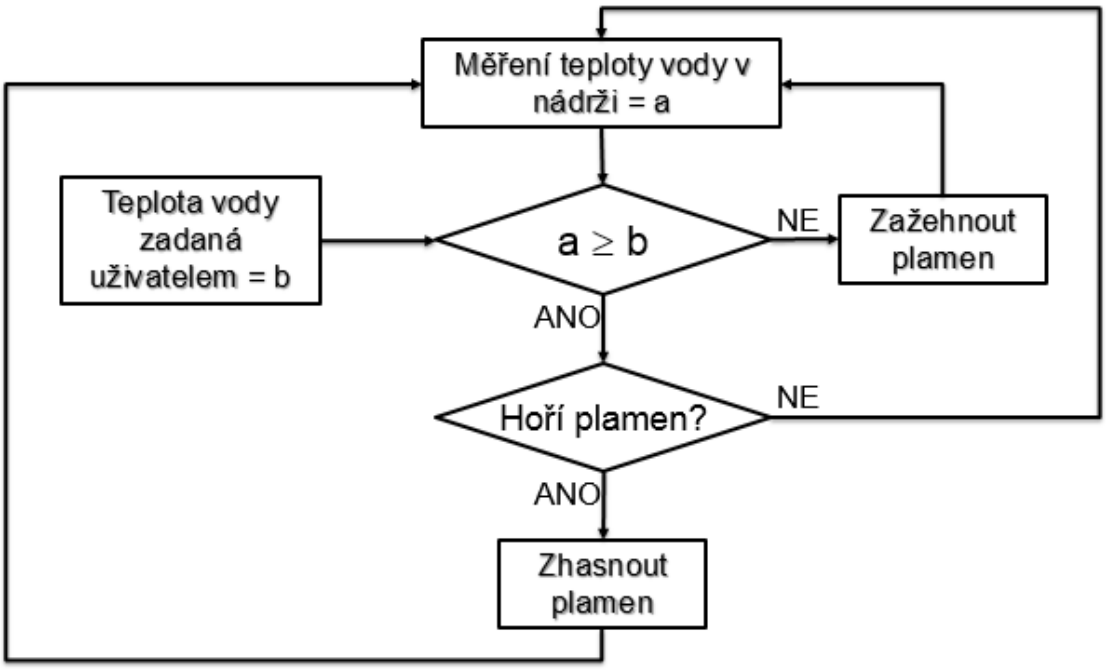
V této kapitole bude popsán testovací systém ve společnosti Honeywell, divizi ACS v Brně. Konkrétně budou nastíněny základní principy testování, struktura systému a jeho úskalí. Dále bude popsána role vytvořené aplikace a požadavky na ni kladené.

2.1 Principy testování

Jak bylo zmíněno na konci kapitoly 1, při testování výrobků je obvykle využíváno služeb automatizovaných nástrojů. Tyto nástroje umožňují provádět testování za simulování libovolných podmínek při zachování přijatelné úrovně nároků na časové i finanční zdroje. Dále bylo naznačeno, že pro testování prototypů je zapotřebí celé soustavy zařízení, která bude nyní popsána. Nejdříve však jednoduchý příklad.

Nechť je testovaným objektem prototyp řídící jednotky kotle. Úkolem dotyčného zařízení je vyhodnocovat informace přicházejících od senzorů umístěných na inkriminovaných místech kotle a jeho okolí a adekvátně na ně reagovat tím, že je vyslán signál k ovladatelným součástkám celého kotle, či informovat uživatele pomocí displeje, notifikačních diod, akustického systému anebo jakýmkoliv jiným způsobem, například přes vzdálený terminál.

Elementárním příkladem může být zjednodušené topení kotle. Cyklus začne v okamžiku, kdy je voda v nádrži plamenem ohřáta na teplotu vyšší, než je teplota nastavená uživatelem. Poté je plamen zhasnut. Z důvodu nižší teploty okolí nebo odčerpáním určitého objemu vody a jeho doplněním vodou studenou se celková teplota kapaliny v nádrži snižuje. Teploměr umístěný v nádrži měří v pravidelných intervalech teplotu vody a naměřenou hodnotu posílá řídící jednotce. Řídící jednotka teplotu vyhodnocuje. Je-li úroveň teploty vody vyšší anebo rovna hodnotě teploty nastavenou správcem kotle, není potřeba na situaci jakkoliv reagovat. Je-li však teplota v kotli nižší než udaná hodnota, je třeba zadat pokyn k ohřívací jednotce kotle, aby zahájila svou činnost. V praxi je vyslána signálová nástupná hrana daným pinem řídící jednotky směrem k ohřívací jednotce. Ta po příjmu tohoto signálu zažehne plamen pod ohřívanou vodou a nechá jej hořet, dokud nedostane signálový příkaz od řídící jednotky. Ten přijde v okamžiku, kdy řídící jednotka na základě údajů získaných z teploměrů rozhodne, že je teplota vody v kotli dostatečně vysoká, tedy rovna anebo spíše vyšší než teplota zadaná uživatelem. Grafické vyjádření situace ve strukturovaném diagramu lze zhlédnout na obrázku 2.1.



Obr. 2.1: Strukturovaný diagram ohřevu vody v kotli

2.2 Struktura testovacího systému

2.2.1 Testovací fixtura

Předchozí ukázka dále demonstruje, že většina součástek interaguje se svým okolím. Pro individuální testování jakékoliv takovéto součástky je zapotřebí její okolí simulovat. Za tímto účelem jsou v divizi ACS vyvíjeny testovací fixtury, do kterých je testovaný prototyp zasazen a které svým chováním nahrazují reálné prostředí. Při paralelním přirovnáním situace ke známému modelu ISO/OSI, jsou testovací fixtury první, tj. nejnižší, vrstvou systému. Obrázek 2.2.

Testovací fixtura je tedy hardware, jenž obstarává komunikaci vyšších vrstev testovacího modelu s testovanou součástkou. Na vstupní piny testovaného prototypu se připojí výstupní piny testovací fixtury a naopak. Těmi pak jsou neseny informace a pokyny od, respektive k, uživateli. Je zřejmé, že pro každou součástku musí být zapojení jiné a také samotné fixtury nemohou být univerzální.

2.2.2 Ovladače

Ovladače, neboli drivery, plní podobnou funkci jako jejich analogie například v operačních systémech desktopových počítačů. Jedná se o software, který přes svůj interface ukazuje vyšším vrstvám modelu služby poskytované testovací fixturou a zpro-

Obr. 2.2: Stuktura testovacího modelu

středkovává jejich užití. V opačném směru předává výše informace zachycené fixturou vyslané testovaným modulem.

2.2.3 Testovací nástroj

Testovací nástroje jsou v nejvyšší vrstvě modelu. Přijímají pokyny od testovacího technika a pomocí ovladačů a testovací fixtury provádějí testování. Zpátky k uživateli zobrazují zprávy zaslané testovanou součástkou či testovací fixturou. Dále by měly uživatele v reálném čase informovat o aktuálním průběhu testu, jeho úspěchu či neúspěchu a aktuální činnosti celého testovacího systému.

2.3 Testovací soubory

Z důvodů relativní komplikovanosti, která je nežádoucím efektem univerzálnosti testovacího systému, se ve společnosti Honeywell, divizi ACS pro testování používají dva hlavní vstupní soubory. První je konfigurační soubor. Tento soubor obsahuje doplňkové informace potřebné pro funkčnost testovaní. Těmito informacemi jsou údaje o testované součástce, použité ovladače, názvy vstupních a výstupních pinů a další údaje, které konkretizují konfiguraci nižších vrstev modelu. Syntaxe konfiguračního souboru je podobná jakékoliv obecné kolekci, využívající služeb značkovacího jazyka.

Druhým objektem, jenž je z hlediska této práce důležitější, je skriptovací soubor. Zde je určeno, dle jakého scénáře bude testování probíhat. Z informací získaných z tohoto skriptu se testovací nástroj dozví, jakou instrukci má testovanému zařízení zadat, tj. co má zařízení vykonat a pak zkoumat, zdali se tomu tak opravdu stalo. Vzhledem k možnosti testovat jednotlivé elementární komponenty individuálně, je

nutné provádět testování po primitivních krocích. Ukázka testovacího skriptu je na obrázku 2.3.

```
;   $History: DSTATE0.SCR $
;
;*****************  Version 3  *****************
;User: Dlindstr     Date: 5/06/97    Time: 8:33a
;Updated in $/SV2-DI/test/scripts
;add header
;
;*****************  Version 2  *****************
;User: Dlindstr     Date: 5/02/97    Time: 12:48p
;Updated in $/SV2-DI/test/scripts
;added history keyword
;
;   Tests:
;
   G Y F P W L P   WAIT    H L C H M P H I  TIMEOUT COMMENTS  ACTUAL
       L R   I O           U O O E A L S N
       A S   M W           M W O A E T I D
       M W   I E               L T N     C
       E     T R                         R
;
;Idle (STATE0)   Direct Ignition
;                                                          Check
 0 0 0 0 1 0 1 0 :      0: x x x x x x x 0 :    200;    Lim closed
 0 0 0 0 1 0 1 1 :      0: 0 / 0 0 0 x 0 0 :    300;    Power unit
 0 0 0 0 1 0 1 1 :    200: 0 1 0 0 0 x 0 0 :    300;    Delay
 0 0 0 0 1 1 1 1 :      0: 0 1 0 0 0 x 0 0 :    200;    W request
 0 0 0 0 1 1 1 1 :    500: 0 1 0 0 0 x 0 0 :    600;    Wait PS Open
;
; Wait PS Open
;
 0 0 0 0 1 0 1 1 :      0: 0 1 0 0 0 x 0 0 :    200;   W off
 0 0 0 0 1 0 1 1 :    800: 0 1 0 0 0 x 0 0 :    900;   Idle
;
; Flame Sensed
;
 0 0 0 1 1 0 1 1 :      0: 0 1 0 0 0 x 0 / :    200;    Apply Flame
 0 0 0 1 1 0 1 1 :    400: 0 1 0 0 0 x 0 1 :    500;    Wait Flame Lost
```

Obr. 2.3: Ukázka testovacího skriptu.

Soubor se skriptem tedy obsahuje textové informace o tom, jak bude testování probíhat a jaké vlastnosti testovaného předmětu budou zkoumány. Je psán a vykonáván po řádcích od shora dolů, na konec obsahu. Na začátku skriptu je informační část o testování a konkrétním testovacím skriptu. Toto záhlaví obsahuje název skriptu, údaje o verzi a čas a datum vytvoření. Celé záhlaví nemá na konkrétní průběh

testování žádný vliv, a tak je každý jeho řádek započat středníkem. Znak středníku označuje vše od něj do konce řádku za komentář a toto pravidlo platí pro celý skript.

Za záhlavím se nachází část pojmenování pinů testovací fixtury. Tyto názvy musí naprosto přesně korespondovat s údaji obsaženými ve výše zmíněném konfiguračním souboru. Při programovém volání metod z implementované knihovny TestConfigurator.dll se piny označují jejich názvy, nikoliv čísly. Počet pinů není konstantní, avšak jejich celkový maximální počet je 24 a dělí se na piny výstupní a vstupní z pohledu testovacího systému. Pro testovaný objekt je pak tato logika obrácená. Výstupní piny jsou vstupní a vstupní piny výstupní. Na tuto skutečnost je třeba pamatovat a nebude-li řečeno jinak, bude použito názvosloví platné pro testovací systém, testovací fixturu.

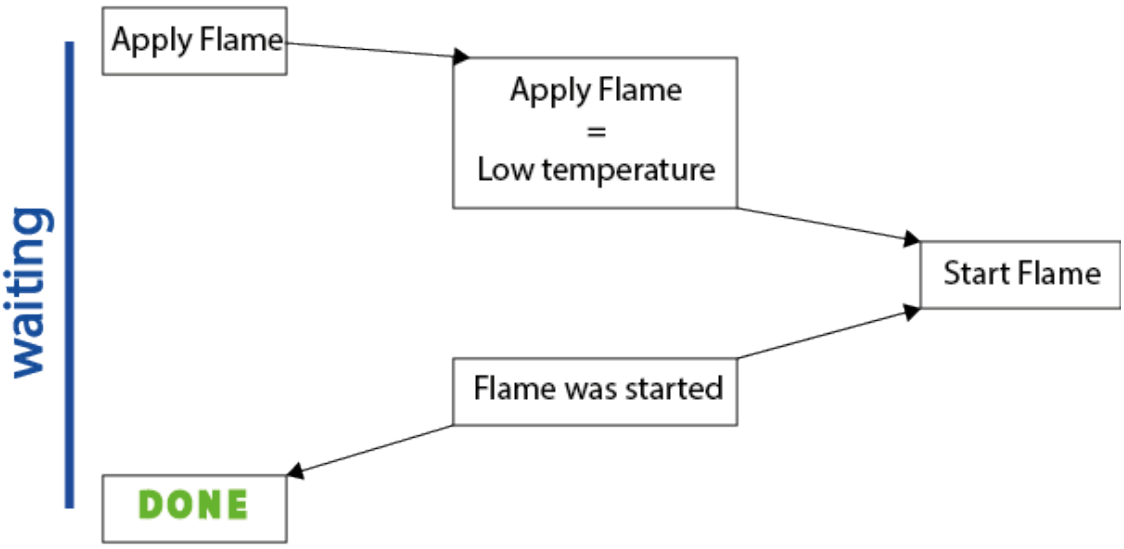
Níže ve skriptu, za výše popsanou částí, se nachází sektory s pokyny testování. Každý sloupec je vyhrazen jednomu pinu a znak na daném řádku určuje, jak se má konkrétní pin chovat. Toto platí pro výstupní, levou část skriptu. Pak následuje údaj o délce čekajícího intervalu, než se změny požadované scénářem testu projeví. Toto prodlení vzniká z konečně rychlé reakce reálných součástek. Údaj je zapsán v desetinách sekund. Ve třetí vertikální oblasti se nachází informace o tom, co má testovací nástroj očekávat na vstupním pinu, tj. jak má testovaná součástka reagovat na pokyn z levé části. Dále je na řádku délka trvání celého testovacího kroku. Rozdíl prvního časového údaje a délky trvání kroku, řádku, je čas, který je testované jednotce vyhrazen na akceptování pokynu a vykonání správné odpovědi. Dále je na řádku již jen komentář, který ve většině případů popisuje funkci daného kroku.

2.4 Testovací proces

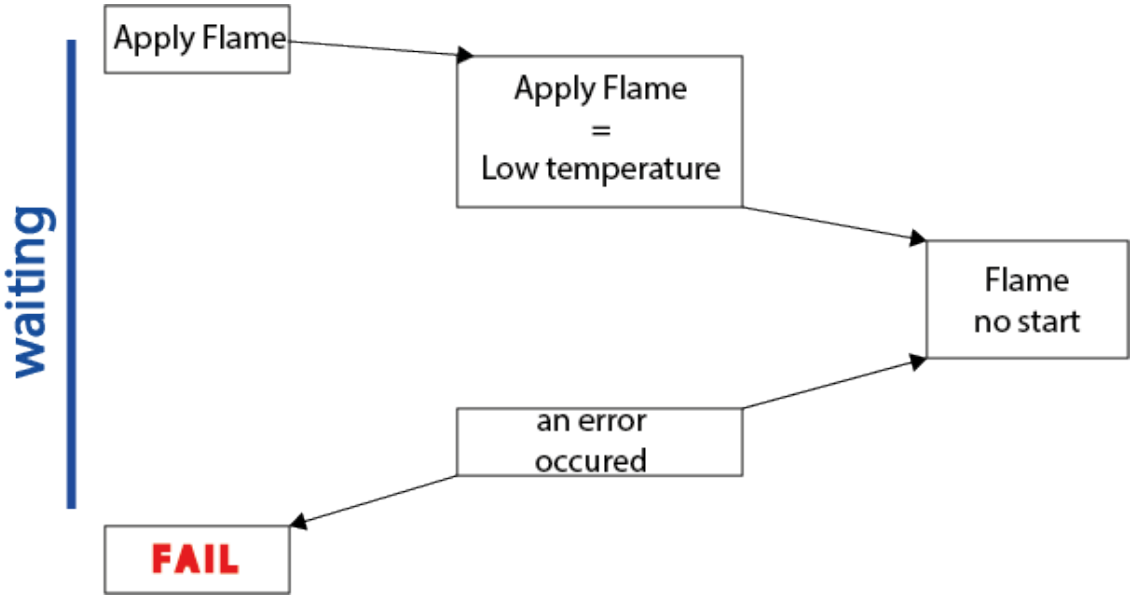
Samotný princip testování spočívá ve správné kombinaci nastavení logických jedniček a nul na vstupní piny do testovaného prototypu. Každý řádek skriptu se rovná jednomu kroku testu. V každém kroku testu jsou nastavením pinů testovanému objektu předány instrukce a testovací prostředí čeká, zdali na ně testovaná součástka správně zareaguje, tj. nastaví své výstupní piny na očekávané hodnoty.

Pro lepší znázornění problematiky bude opět použita situace se zapínáním plamene kotle. Celý proces probíhá následovně. Testovací nástroj se ze skriptu dovídá, že testovací technik chce simulovat pokles teploty vody v nádrži kotle, tj. chce testovat zažehnutí plamene. Vydá tedy pokyn „Zažehni plamen“, přes ovladače testovací fixtuře, která centrále simuluje informaci od teploměru, že teplota klesla pod udanou hodnotu. V reálném prostředí vše chvíli trvá, a tak tester čeká na odpověď. Centrála vydala pokyn k zažehnutí plamene a fixtura zachytává pokyn k zapalovači „Zapni se“. Odpovídá tedy zpátky testovacímu nástroji, že centrála plamen zažehla. Celý

akt byl vykován v časovém limitu, a tak test proběhl v pořádku. Neproběhl-li by celý proces tak, jak bylo naznačeno výše, krok, a tím i celý test, by skončil chybou. K vysvětlení situace mohou být využity ilustrace na obrázcích 2.4 a 2.5.



Obr. 2.4: Grafická ukázka principů testování – úspěch.



Obr. 2.5: Grafická ukázka principů testování – chyba.

2.5 Obměna testovacího systému v ACS

Celé testovací prostředí v divizi ACS společnosti Honeywell se v nedávné minulosti začalo obměňovat a modernizovat. Vývojáři začali psát obslužné metody testovacích desek v jazyce C#, společně s tím byly vytvářeny nové ovladače a také konečné grafické rozhraní.

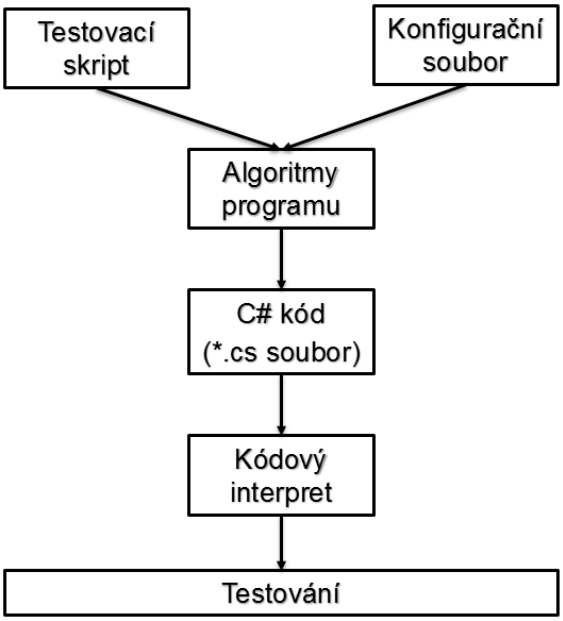
Zde však nastává střet dvou směrů. Jsou-li nové obslužné metody psány v jazyce C#, ovšem testovací scénáře, skripty, ve svém vlastním kódu, budou muset být napsány algoritmy, jež budou staré skripty číst a dle nich volat nově vytvořené metody v jazyce C#. Bylo přáním testovacích techniků divize ACS, aby skripty, které byly za mnoho let působení jejich systému napsány, byly nadále plně použitelné a funkční. Důvodů k této žádosti je více. Je důležité zachovat naprostou shodu testů prováděných v minulosti s aktuálními testy, většina testovacích techniků je na psaní skriptů ve starém formátu zvyklá a neposlední výhodou je jejich dobrá čitelnost, modulárnost, lehká rozšiřitelnost a programová nezávislost. Domnívám se, že jejich vymření nastane až v okamžiku, kdy bude existovat plně funkční a výkonné prostředí, kde se testovací scénáře budou zobrazovat a psát pomocí Drag&Drop systémů v desktopové aplikaci.

Primárním úkolem této práce bylo vytvoření takovéhoto konvertoru starých skriptů na nový typ testování. Program v této práci formovaný přebírá na svém vstupu testovací skript a konfigurační soubor a vytvoří z nich jeden *.cs soubor (soubor s kódem v jazyku C#), který je dále interpretován a tím je prováděno samotné testování. Struktura části nového systému je naznačena na obrázku 2.6.

2.6 Uživatelské nástroje pro testování

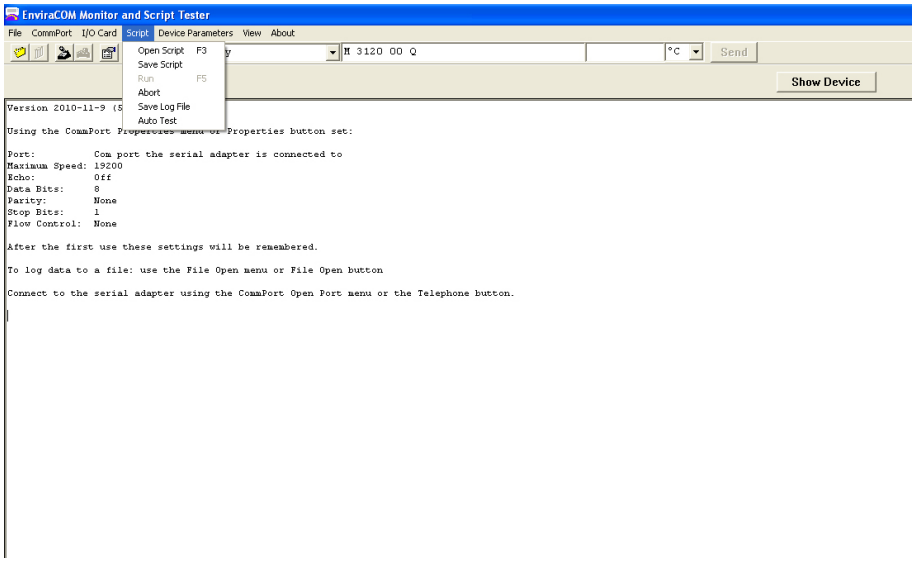
Jak již bylo napsáno, testovací nástroj, uživatelská aplikace, je na nejvyšší příčce testovacího modelu. Má za úkol komunikovat s uživatelem, konkrétně přijímat od něj instrukce k testování a poskytovat mu zpětnou vazbu testu. V tomto konkrétním případě musí obsahovat mechanizmy pro zadávání testovacího skriptu, testovacího souboru a umístění ovladačů k testovací fixtuře. Ve směru k uživateli pak vypisovat do logovacího okna úkony vykonané testovacím systémem a log probíhajícího testu.

V současné chvíli dosluhující nástroj pro testování v divizi ACS je program „EcoMon“ – „EnviraCOM Monitor and Script Tester“. Ukázka z programu na obrázku 2.7. Jedná se o program, který je dle testovacích a vývojových techniků ACS obtížně udržovatelný a rozšiřitelný, málo modulární a uživatelsky nepohodlný. Tento stav je zapříčiněn souběhem několika faktů. První je, že aplikace EcoMon je program, který byl vytvářen pro úplně jiné účely a situace a testování do něj bylo v průběhu jeho



Obr. 2.6: Struktura části nového systému

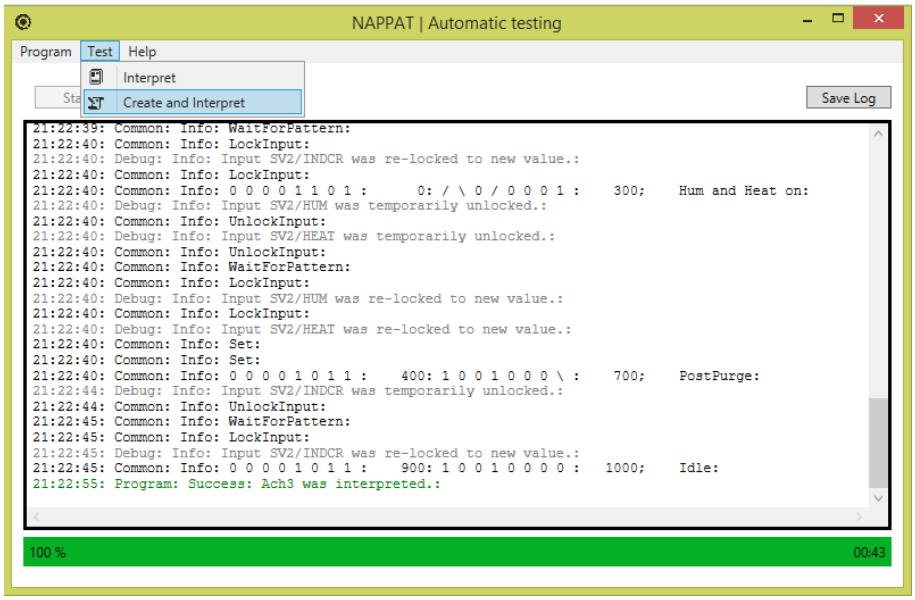
životnosti implementováno až dodatečně. Samo včlenění testovacích metod do programu EcoMon bylo velmi obtížné, poněvadž je aplikace psána v jazyce Visual Basic a problémem je nekompatibilita funkcí tohoto jazyka s novými Windows.



Obr. 2.7: Ukázka testovacího programu EcoMon.

S modernizací testovacího systému se vyskytl vhodný prostor pro vytvoření nového testovacího nástroje. Cílem by bylo vytvoření grafické desktopové aplikace,

jejíž primárním úkolem bude zprostředkovávat komunikaci mezi testovacím technikem a testovacím systémem. Její specializace bude umožňovat mnohem snadnější vytvoření uživatelky přátelského prostředí, jehož doktrínou bude jednoduchost, přehlednost a ergonomické ovládání. A tak byl vytvořen program Nappat, jenž bude detailněji popsán v následujících kapitolách. Ukázka jeho prostředí je na obrázku 2.8



Obr. 2.8: Ukázka testovacího programu Nappat

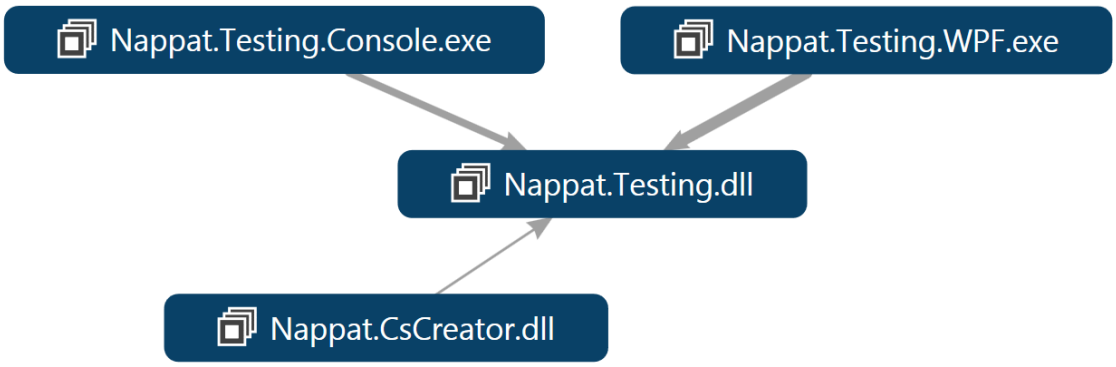
2.7 Shrnutí kapitoly

Na základě faktů sepsaných v přechozích částech textu je nyní jasnější pohled na zadání práce a v jeho definici nyní lze být konkrétnější. Cílem je napsání počítačového programu, který bude mít dva úkoly na sobě relativně nezávislé. Prvním je vytvoření funkčních algoritmů, které z předložených skriptů a konfiguračních souborů vytvoří jeden komplexní soubor, jež bude mít syntaxi programovacího jazyka C#. Druhým cílem je vytvoření koncové uživatelské aplikace na osobní počítače pro testovací techniky, která jim bude umožnovat nastavení testovacího procesu, v reálném čase je o průběhu testování informovat a testování spouštět.

3 STRUKTURA APLIKACE

Aby nebyly opakovány omyly předchozích testovacích systémů, byl kladen velký důraz na strukturu celého programu, která by se měla skládat z jednotlivých modulů, které mají za úkol elementární, ale esenciální úkoly. Programování v prostředí .NET nám situaci značně usnadňuje skutečností, že umí vytvářet tzv. dynamické knihovny, soubory s příponou dll. Výsledná aplikace pak může jen zastřešovat celý projekt, respektive obsahovat pouze odkazy na jednotlivé logicky dělené moduly celého systému.

Zde se však setkáváme ještě s jedním problémem. Je faktem, že moderní aplikace psané v moderním prostředí mají spoustu výhod a mnohonásobně usnadňují práci. Ovšem jen v případě, že jsou dodržena určitá pravidla a zákonitosti. V tomto kontextu je myšleno užití vhodného návrhového vzoru. Cílem moderních návrhových vzorů, známé jsou MVC 5 anebo nyní stále populárnější MVVM, je směřovat vývojáře aplikace tak, aby výsledný program byl snadno transformovatelný na různé platformy, například z desktopové WPF aplikace na Store App, Windows Mobile a podobně. Jedná se tedy o modulárnost směrem nahoru, od výkonného kódu. V situaci této práce však bylo potřeba postupovat podobně, s podobnými cíli, i dolů ve struktuře výkonné části aplikace. Zároveň však musela být zachována adekvátní projektová čitelnost.



Obr. 3.1: Závislostní diagram celého programu

Z obrázku 3.1 můžeme vyčíst, že středobodem celé aplikace je Testovací sestavení (Testing.dll). Na Testovací sestavení jsou směrem nahoru navázány s uživatelem komunikující rozhraní. Směrem dolů pak Interpret kódu a Tvůrce kódu (CsCreator.dll).

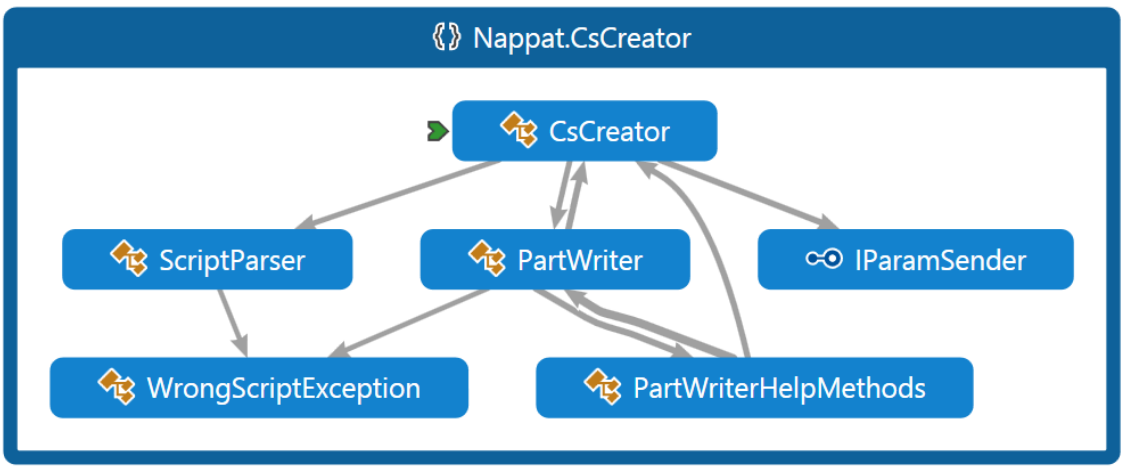
Jednotlivé části celého programu budou nyní popsány detailněji. Vzhledem ke skutečnosti, že program napsaný v této práci má být rozšiřitelný a je skoro jisté, že

do kódu bude zasahováno, jsou všechny proměnné a názvy metod psány v Anglickém jazyce, a také byla snaha, aby názvy nesly co nejvyšší vypovídající hodnotu. Z těchto důvodů je někdy obtížné najít k anglickému pojmenování český ekvivalent. V následujícím textu budou tedy ponechány originální názvy s tím, že bude nejlepší snaha o jejich vysvětlení následovat bezprostředně po jejich vyřčení.

3.1 Tvůrce kódu (CsCreator.dll)

Předchozí kapitola nám naznačila, že se z pohledu testovacích principů nyní nacházíme v přechodném období. Na jedné straně zde existují, a jsou reálně použitelné, i desítky let staré skripty a na druhé straně zde máme moderně vyvíjené testovací prostředí v programovacím jazyce, jenž je jednou tak mladší. Výhody psaní systému v jazyce C# již byly několikrát zmíněny a na základě těchto faktů je rozumné se těchto benefitů nevzdávat.

Z tohoto důvodu bylo třeba vytvořit program, jenž dokáže číst staré testovací skripty, a na základě informací z nich získaných generovat kód s identickými vlastnostmi, ovšem v jazyce C# a tento kód pak dávkově spouštět. Zmíněným problémem se bude tato sekce zabývat. Program je pro svou přehlednost rozdělen na několik tříd a systém podsekcí této sekce bude toto rozdělení reflektovat. Závislostní diagram sestavení je vykreslen na obrázku 3.2.

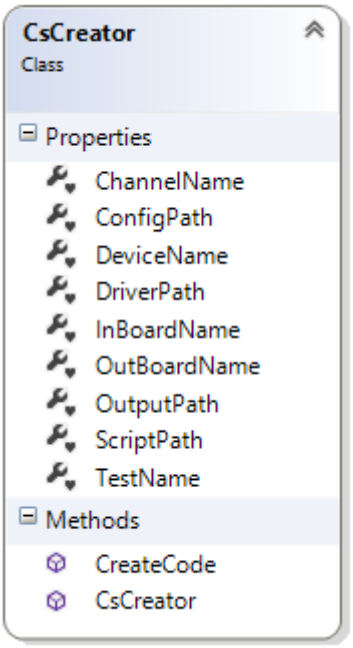


Obr. 3.2: Závislostní diagram sestavení Tvůrce kódu (CsCreator.dll)

3.1.1 CsCreator.cs

Jak ukazuje závislostní diagram na obrázku 3.2, je třída CsCreator hlavní třídou sestavení. Obsahuje dvě metody, a to svůj konstruktor a metodu *CreateCode*. Kon-

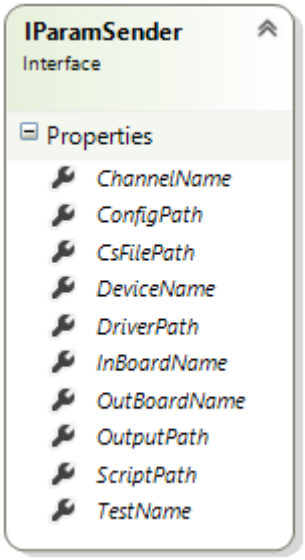
struktor přijímá ve svém parametru rozhraní IParamSender, které slouží jako zprostředkovatel dat zadaných uživatelem metodám, jež obstarávají tvorbu C# kódu, a přiřazuje je do statických proměnných – vlastností (Properties). Volba statických metod byla z důvodu snížení režie a komunikace mezi jednotlivými metodami navazujících tříd. Druhá metoda třídy, *CreateCode*, obstarává funkčnost celé transformace skriptu na C# kód. Jinými slovy vytváří instance pomocných tříd, jež budou popsány, a volá jejich metody. Tímto se vytvoří C# kód, který je na konci uložen do souboru s příponou cs a názvem, který zadal uživatel jako název testu, testování. Takto vytvořený soubor je regulérní soubor jazyku C#, tj. obsahuje již vše potřebné k jeho interpretaci. Pro lepší znázornění je připojen třídový diagram na obrázku 3.3.



Obr. 3.3: Diagram třídy CsCreator

3.1.2 IParamSender.cs

V textu byla zmínka o rozhraní IParamSender. Toto rozhraní obsahuje vlastnosti, jež jsou nezbytné pro vytvoření kódu a jsou získány od uživatele přes uživatelské rozhraní. Obrázek 3.4. Obsahují informace o umístění testovacího skriptu, konfiguračního souboru, ovladačů k testovací desce a názvy komponent testovací fixtury a název testu. Dále potom místo pro uložení souboru.



Obr. 3.4: Diagram rozhraní IParamSender

3.1.3 ScriptParser

Jak bylo zmíněno v předchozí části textu, aplikace má provádět testování dle skript souboru. To znamená, že v první fázi musí program skript přečíst a rozdělit na části dle toho, jak moc jsou pro testování významné a co obsahují. O to se stará kód v souboru a třídě ScriptParser. Její třídový diagram je na obrázku 3.5.

Kód v tomto souboru vyčlení z testovacího skriptu část komentářů, tj. řádků, které začínají znakem středník, část názvů pinů a sektor, ve kterém se piny nastavují – testovací instrukce. Dále pak zjistí, kolik pinů se používá pro výstup a kolik pro vstup.

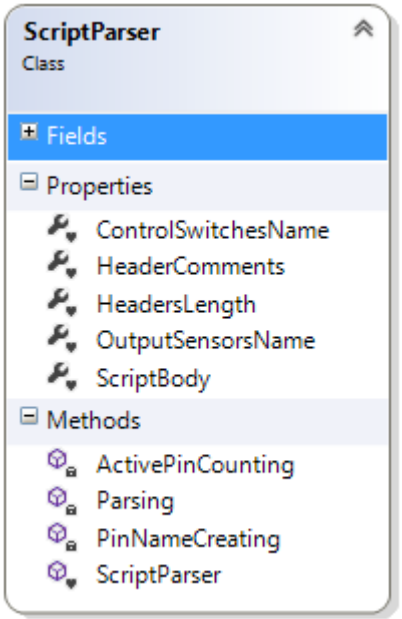
Soubor ScriptParser obsahuje jednu třídu ScriptParser v jejímž konstruktoru se volají výkonné metody. Základem kódu jsou metody třídy StreamReader, implementované v prostředí .NET, a to především metoda *ReadLine*. Ta je v cyklu volána pro každý řádek skriptu. Výhodou tohoto řešení je fakt, že nenačítáme do paměti celý skript, který může být obsáhlý, ale vždy jen jeden řádek, nad kterým se program dle podmínek rozhoduje.

```
StreamReader reader = File.OpenText(scriptPath);
string line = null;
int sectorIdentifier = 0;
int lineCounter = 0;
while ((line = reader.ReadLine()) != null)
{
        ( ... )
}
```

Při parsování jakéhokoliv textu je v onom textu hledána logika, dle které by mohl být soubor rozdělen, a zároveň je snaha, aby rozhodování bylo co nejefektivnější, nejjednodušší a nejmodulárnější. Jinak řečeno, je usilováno o stav, kde bude maximálně minimalizován lidský faktor působící při psaní scénáře testování. Je zřejmé, že mezera navíc se v testovacím skriptu snadno ztratí, zvlášť když není psán v žádném debuggovacím prostředí a specifikace je nezakazuje. Na druhou stranu, je nutné se dle něčeho řídit, a tak je hledán kompromis mezi těmito zmíněnými faktory.

V kódu je základním parametrem pro rozhodování první nebílý znak řádku. Jakmile je v cyklu nalezený takovýto znak, rozhoduje se dle jeho obsahu, jak bude s řádkem zacházeno dále, a do kterého zásobníku se zařadí. Může to být buďto komentářová hlavička, hlavičky (názvy pinů) anebo tělo skriptu. Ze základní stavby skriptovacího kódu vyplývá jeden problém, který musel být vyřešen. Část komentářová hlavička, jejíž řádky začínají jako každý komentář středníkem, je pro testování irelevantní, a tudíž je tento řádek předán správci paměti, který jej vymaže. Respektive takto to bylo zamýšleno ze začátku projektu. Testovací technici společnosti Honeywell, divize ACS, však změnili své požadavky a projevili přání tuto hlavičku vypisovat do logu testování. Díky modularitě napsaného kódu však toto rozšíření bylo snadné doimplementovat. Výhody tohoto řešení jsou však pořád stejně velké, jak bude ukázáno v následující sekci textu, která se stará o tvorbu C# kódu.

Některé výhody rozdělení typů komentářů však jsou znatelné ihned. Předchozí větou je myšlen především fakt, že datové proměnné jsou obsahově menší a OS



Obr. 3.5: Diagram třídy CsCreator ScriptParser

s nimi tak může lépe pracovat v paměti počítače. Nezanedbatelné jsou i vlastnosti jako lepší čitelnost a upravovatelnost kódu a snadnější debuggování.

Situace byla řešena přidáním další proměnné, která zaznamenává, zdali se řádek začínající středníkem nachází již za sektorem hlaviček. Je-li tomu tak, pak se jedná o komentář z těla skriptu a tento řádek, na rozdíl od komentáře z hlavičky, není zahozen.

V tento moment je vyřešena budoucnost řádků začínajících znakem „;“. Ovšem i nekomentářové řádky mají více typů. Konkrétně dva, a to hlavičky a testovací příkazy. Ve specifikaci, která upřesňuje psaní skriptů, je naštěstí striktně uvedeno, jakým znakem smí začínat testovací řádek. Jsou to znaky „0“, „1“, „!“ a „.“ a těmto řádkům bude ještě věnována pozornost. Tudíž, začíná-li řádek písmenem, jedná se o hlavičky anebo chybu.

V hlavičkách jsou obsaženy názvy pinů a jsou psány vertikálně, od shora dolů, tak jak je naznačeno na obrázku 3.6. List těchto řádků dále zpracovávají metody *ActivePinCoutnting* a *PinNameCreating*. Tyto metody jsou pro tvorbu C# kódu velmi důležité, jelikož získávají ze skriptu počet výstupních a vstupních pinů a jejich jednotlivé názvy.

```
;
  G Y F P W L P   WAIT    H L C H M P H I  TIMEOUT COMMENTS  ACTUAL
      L R   I O           U O O E A L S N
      A S   M W           M W O A E T I D
      M W   I E               L T N     C
      E     T R                         R
;
```

Obr. 3.6: Ukázka hlaviček testovacího skriptu

Původní algoritmus metody *ActivePinCoutnting* byl, že metoda procházela první řádek hlaviček (obrázek 3.6) a analyzovala jednotlivé znaky. Jakmile narazila na posloupnost znaků „T“ „I“ „M“ „E“, ukončila přidávání znaků do sektoru výstupních pinů a následující znaky patřily už vstupnímu nastavení. Tahle logika však nebyla příliš spolehlivá, vždy existovala pravděpodobnost, že první písmena názvů pinů mohou poskládat kombinaci TIME a také nebyla z pohledu kódového designerství správná. Byla tedy nahrazena algoritmem, který je používán v další části programu pro parsování příkazového testovacího řádku a je prováděn nad prvním validním řádkem těla skriptu – obrázek 3.7. Řádek je jednoduše získán při dělení skriptovacího souboru na části, který byl popsán výše tak, že pomocí jednoduchého boolin přepínače se rozhoduje o tom, zdali je řádek první, a pokud ano, je zapsán do řetězcové proměnné *referenceLine*. Tento systém je mnohem robustnější a jediné, co

vyžaduje, aby nebyla stylistická chyba v prvním příkazovém řádku skriptu.

```
   G Y F P W L P   WAIT    H L C H M P H I  TIMEOUT COMMENTS  ACTUAL
       L R   I O         U O O E A L S N
       A S   M W         M W O A E T I D
       M W   I E             L T N     C
       E     T R                       R
;
;Idle (STATE0)   Direct Ignition
;                                                          Check
 0 0 0 0 1 0 1 0 :      0: x x x x x x x 0 :    200;    Lim closed
 0 0 0 0 1 0 1 1 :      0: 0 / 0 0 0 x 0 0 :    300;    Power unit
 0 0 0 0 1 0 1 1 :    200: 0 1 0 0 0 x 0 0 :    300;    Delay
```

Obr. 3.7: První validní řádek skriptu.

Metoda PinNameCreating je postavena na funkčnosti .NET třídy StringBuilder z prostoru System.Text. Tato třída má oproti klasické a známé třídě String tu výhodu, že dokáže dynamicky měnit její řetězec. Zde už je pak jen procházeno pole hlaviček cyklicky po sloupcích a znaky na pozicích korespondujících s pozicemi získanými v metodě *ActivePinCoutnting* jsou přidávány do aktuální proměnné typu StringBuilder, čímž získáme pole řetězců obsahující názvy výstupních a vstupních pinů.

```
List<StringBuilder> tempNamesSB = new List<StringBuilder>();

for (int i = 0; i < headers[1].Length; i++)
{
    StringBuilder sb = new StringBuilder();
    tempNamesSB.Add(sb);

    for (int j = 0; j < headers.Count; j++)
    {
        tempNamesSB[tempNamesSB.Count() - 1].Append(headers[j][i]);
    }
}

for (int i = 0; i < controlSwitchesList.Count; i++)
{
    if (controlSwitchesList[i] != ' ' && controlSwitchesList[i] != ':')
    {
        ControlSwitchesName.Add(tempNamesSB[i].ToString().Trim());
    }
}

for (int i = 0; i < outputSensorsList.Count; i++)
```

```
{
    if (outputSensorsList[i] != ' ' && outputSensorsList[i] != ':')
    {
        OutputSensorsName.Add(tempNamesSB[secondStart +
            i].ToString().Trim());
    }
}
```

3.1.4 PartWriter.cs a PartWriterHelpMethods.cs

Nyní je situace ve stavu, kdy je skript rozdělený na části, ze kterých bude psán konkrétní C# kód, který bude následně interpretován. O vytváření C# kódu se stará právě třída FileCreator. Tato třída v konstruktoru přebírá datové typy List<String> obsahující výsledky práce třídy ScriptParser. Další informace potřebné k napsání funkčního kódu získané od uživatele jsou metodám třídy dostupné z hierarchicky nadřazené a již popsané třídy CsCreator, díky jejich statickému statusu. Mezi tato data patří umístění ovladačů, název testování, názvy testovacích desek atp.

Hlavním a jediným účelem třídy je vytvořit pole řádků C# kódu, který bude interpretovatelný a na jeho základě se bude provádět testování prototypů. Ke správné funkčnosti tohoto kódu je zapotřebí mnoho režijních informací, a proto je tvorba textových řetězců, budoucích řádků, C# kódu rozdělena na čtyři metody, jež se každá starají o jeden logický úsek skriptu. Tyto metody budou nyní popsány tak, že jejich posloupnost bude reflektovat jejich řazení v konečném testovacím C# kódu.

HeaderWriter

Tato metoda je volána jako první a stará se o první řádky C# souboru. Vypisuje odkazy na reference, jmenné prostory, a vytváří metodu *Run*, která bude obsahovat algoritmy pro testování. Dále připravuje testovací prostředí, tj. nastavuje výstupy a zamyká vstupy testovací fixtury.

```
FileCreatorList.Add("using System.Threading;");
FileCreatorList.Add("");
FileCreatorList.Add("namespace AutomatedTesting");
FileCreatorList.Add("{");
Indent++;
FileCreatorList.Add(IndentMethod() + "public class " +
    CsCreator.TestName);
FileCreatorList.Add(IndentMethod() + "{");
Indent++;
FileCreatorList.Add(IndentMethod() + "#region Testing Code");
FileCreatorList.Add(IndentMethod() + "public void Run()");
FileCreatorList.Add(IndentMethod() + "{");
```

Na této ukázce kódu lze zhlédnout postupy psaní výsledného kódu, konkrétně vytvoření třídy pojmenované dle názvu testu a metody *Run*. Každý budoucí řádek je nyní zapisován do dynamického pole textových řetězců List<string> pojmenovaného *FileCreatorList*. Stejný princip využívají i následující psací metody a stejný je i systém tvorby odsazení. Aby vytvořený C# kód byl dobře čitelný, používá programovací prostředí, v tomto případě Visual Studio, odsazování řádků dle logiky psaného kódu, jinými slovy jeho zanořování. Zde, protože je kód tvořen takto manuálně, je třeba se o odsazování postarat také samostatně. Konkrétně to má za úkol metoda *IndentMethod*. Ta ve své podsatě nedelá nic jiného, než že vrací řetězec mezer, bílých znaků, které pak ve výsledku vytvoří odsazení. Pro nastavení odsazení je použita inkrementace či dekrementace vlastnosti *Indent*, tak jak je použita v ukázce. Kód metody *Indent* následuje bezprostředně za touto větou. Násobení čtyřmi je zdůvodu běžně používané délky jednoho dosazní v jednotkách mezer.

```
internal static string IndentMethod()
{
    List<String> tempList = new List<String>();
    for (Int32 i = 0; i < PartWriter.Indent * 4; i++)
    {
        tempList.Add(" ");
    }
    return String.Join(String.Empty, tempList.ToArray());
}
```

HeaderCommentsWriter

V této metodě se zpracovává komentářová hlavička, o které bylo hodně psáno v předchozí podkapitole ScriptParser. Fakt, že tato stejnojmenná třída již oddělila komentáře z těla skriptu a tuto hlavičku, nám umožňuje v jednoduchém cyklu vypsat celé záhlaví. Kód celé metody tak zabral pouhé dva řádky výkonného kódu.

```
foreach (string line in HeaderComments)
{
    FileCreatorList.Add(IndentMethod() + "tc.LogInfo(" + "@" + "\"" +
        line + "\"" + ");");
}
```

BodyWriter

Tato metoda je nejobsáhlejší metodou třídy a stará se přímo o přepisování jednotlivých testovacích příkazů ze starého formátu testovacího skriptu do syntaxe jazyka C#.

Celý princip algoritmů je relativně jednoduchý a podobný systému třídy ScriptParser. Třída PartWriter, stejně jako třída ScriptParser, pracuje s polem textových řetězců. V tomto kontextu je jedno, jsou-li uloženy v externím textovém souboru, skriptu, anebo v programové proměnné typu List<String>. V cyklu je po jednotlivých řádcích procházen List řádků těla skriptu vyselektovaných v metodách třídy ScriptParser a na základně obsahu prvního nebílého znaku je s daným řádkem dále zacházeno. Z listu textových řetězců tak dostáváme matici, kterou procházíme dvěma cykly, jejíž proměnné nám určují aktuální pozici znaku z hlediska řádků a pozici v řádku. Nyní budou rozebrány možnosti, které mohou nastat.

a) Prvním znakem řádku je mezera nebo jiný bílý znak. V tomto případě je řádek procházen tak dlouho, dokud se nenajde validní znak.

```
if (ScriptBody[i][j] == ' ')
    continue;
```

b) Prvním znakem řádku je středník. Je-li prvním určujícím znakem středník, jedná se o komentář, který má být vypsán do logu. Testovací prostředí nabízí několik druhů textů s různou vážností. Text s normální, neboli informativní, důležitostí obsahu je předán metodě *LogInfo*. Na tomto řádku již nic jiného než komentář být nemůže, a tak je interní řádkový cyklus přerušen.

```
else if (ScriptBody[i][j] == ';')
{
    FileCreatorList.Add(IndentMethod() + "tc.LogInfo(" + "@" + "\"" +
        ScriptBody[i].Trim() + "\"" + ");");
    lineCounter++;
    break;
}
```

c) Prvním znakem řádku je 1 nebo 0. V případě, že je prvním znakem hodnota pinu, tj. 1 nebo 0, jedná se o příkazový řádek, který nastavuje výstupní a vstupní piny a provádí tak fyzické testování. Jeden řádek skriptu, tak je zvýrazněn na obrázku 3.8 však obsahuje více než hodnoty pro nastavování pinů. Je zde uveden také čas pro čekání, časový interval jednoho řádku a komentář. Proto je řádek nejdříve rozdělen dle dvojteček a středníků funkcí *Split* volanou nad textovým řetězcem. V argumentu pak tato metoda přebírá pole znaků, dle kterých bude řetězec rozdělen a ještě zařídí, že budou odebrány mezery okolo získaných řetězců. Tak je ukázáno zde.

```
char[] separators = new char[] { ':', ';' };
string[] tempStrings = ScriptBody[i].Split(separators,
    StringSplitOptions.RemoveEmptyEntries);
```

Nyní již je řádek rozdělen na jednotlivé logické části a s každou z nich bude zacházeno zvlášť. Nejdříve se provedou metody pro získání časů. Pro přehlednost

kódu je umístěna ve zvláštní třídě PartWriterHelpMethods souboru PartWriterHelpMethods.cs a jsou volány následovně:

```
waitTime = PartWriterHelpMethods.TimeSetter(tempStrings[1].Trim());
timeOutTime = PartWriterHelpMethods.TimeSetter(tempStrings[3].Trim())
    - waitTime;
```

Její relativní složitost je způsobena skutečností, že časový údaj smí být uveden jako proměnná, jejíž hodnota je uvedena v externím souboru přiloženém ke skriptu. Metoda tak musí rozhodnou, zdali se jedná přímo o číslo či proměnnou, a ve druhém zmiňovaném případě prohledat data získaná rozparsováním přiloženého souboru a proměnnou nahradit číslem. K tomuto účelu jsou případné časové údaje získané z přiloženého souboru ukládány do datového typu slovník, který je metodou prohledáván a je hledána shoda v klíči slovníku s textovým řetězcem získaným z řádku skriptu.

```
internal static int TimeSetter(string timeName)
{
    int retTime = 0;

    if (PartWriter.TimeVariables.ContainsKey(timeName))
        retTime = PartWriter.TimeVariables[timeName];
    else
        try
        {
            retTime = Convert.ToInt32(timeName);
        }
        catch
        {
            throw new Exception("Can not found time variables: " + "'"
                + timeName + "'.");
        }
    return retTime;
}
```

Nyní již mohou být zpracovávány úseky nesoucí příkazy k nastavení pinů. Na následujících řádcích je uveden konkrétní příklad použití.

Je dáno za úkol vytvořit kód na základě vyznačeného řádku skriptu z obrázku skriptu 3.8.
Je předpokládáno, že jsou již určeny názvy výstupních a vstupních pinů. Na aktuálním řádku se nachází tento příkaz k nastavení.

```
0 0 0 1 1 0 1 1 :  0: 0 1 0 0 0 x 0 /:  200;  Apply Flame
```

Již dopředu bude prozrazeno, že tento řádek skriptu říká našemu nástroji: „Řekni testovanému zařízení, ať zažehne plamen a testuj, zdali se plamen opravdu zažehl, a to do 200 časových jednotek.“ Nejdříve se píší metody pro výstupní piny. Oproti

```
 0 0 0 0 1 0 1 1 :   800: 0 1 0 0 0 x 0 0 :     900;   Idle
;
; Flame Sensed
;
 0 0 0 1 1 0 1 1 :     0: 0 1 0 0 0 x 0 / :     200;    Apply Flame
 0 0 0 1 1 0 1 1 :   400: 0 1 0 0 0 x 0 1 :     500;    Wait Flame Lost
```

Obr. 3.8: Výřez skriptu k vykonaní

předchozímu stavu došlo k jedné změně. Čtvrtý výstupní pin „FLAME“ ze změnil z hodnoty 0 na hodnotu 1, tj. ono „Řekni testovanému zařízení, ať zažehne plamen“. Pro provedení této změny budeme potřebovat jeden řádek takovéhoto kódu.

```
tc.DigitalOutputs.Set("SV2","FLAME",DigitalValue.High);
```

Dále k tomu je přidána informace pro logování a dobu čekání, která je v tomto případě rovna 0.

```
tc.LogInfo("Apply Flame");
Thread.Sleep(0);
```

U vstupních pinů je také registrována jedna změna, a to na posledním pinu, kde znak lomítka symbolizuje nástupnou hranu. To je ekvivalentní k „a testuj, zdali se plamen opravdu zažehl“.

```
if (tc.WaitForChannelPattern("SV2", "INDCR", 2000, DigitalValue.High)
    < 0)
{
    tc.LogFatalError("Apply Flame");
}
```

Algoritmus znamená, že je testováno, jestli se plamen nezažehl. Pokud je tato podmínka splněna, je zavolána metoda *LogFatalError*, které je jako argument předán komentář umístěný na konci řádku.

Před tímto řádkem, i po tomto řádku, se kód pro všechny řádky skriptu vytvářel analogicky. Je vidno, že je vždy nutno pamatovat si stavy předchozího řádku. Tato nevýhoda je ale vyvážena skutečností, že je-li nastaven pin do nějaké hodnoty, již v této hodnotě zůstane až do doby, dokud nebude explicitně změněn. Tento fakt přináší značnou úsporu kódu.

d) Prvním znakem řádku je vykřičník. Za tohoto předpokladu se jedná o funkce upravující tok kódu a o zprávy od testovacího technika k nižším vrstvám testovacího prostředí. Názvy funkcí jsou od sebe odlišeny návěštím následujícím za vykřičníkem. Z nabídky funkcí měnících tok skriptu autor může použít návěští „GOTO“ a „CYCLE“.

Značka „GOTO“ se používá společně s „LABEL“ a značí posun v plynulosti kódu od „GOTO“ přímo na „LABEL“, přičemž jsou přeskočeny řádky mezi těmito dvěma návěštími. Transkripce těchto příkazů do jazyku C# je jednoduchá, poněvadž tento jazyk disponuje stejnou funkcionalitou.

Značka „CYCLE“ označuje začátek kódu, který se bude cyklicky opakovat dle udaného počtu cyklů. Tvoří pár s označením „LOOP“, které se nachází za posledním řádkem cyklovaného úseku. Do kódu jazyku C# je tato funkce přepsána do cyklu „for“.

Pro posílání zpráv testovacímu zařízení slouží návěští „TX“ a „RX“. Z knihoven ovladačů testovací fixtury lze zavolat funkci *Send*, v jejímž argumentu je pole textových řetězců obsahujících zprávu.

```
else if (tempStrings[1].ToUpper() == "TX")
{
    FileCreatorList.Add(IndentMethod() + "tc.Send( (...) );");
}
```

e) Prvním znakem řádku je tečka. Znak tečky je poslední možností, jak může validní řádek začít. Jedná se o příkaz k načtení externího souboru, který obsahuje doplňující informace ke skriptu a hodnoty časových proměnných, jež lze psát místo číselného konstantního vyjádření času trvání a čekacího intervalu v řádku skriptu.

Zde je nutné věnovat zvýšenou pozornost názvosloví. Zmiňovaný externí soubor v předchozím odstavci je označován jako konfigurační soubor, ovšem s konfiguračním souborem, který obsahuje názvy fixtury a jejích pinů, nemá nic společného. V této sekci textu bude konfiguračním souborem textový soubor, jehož možná podoba je na obrázku 3.9.

```
;configuration file for times and ports setting

.T ppg : 500;   prepurge
.T ign : 900;   ignition time
.T soft_lock : 36000; soft-lockout time
;
;

.P PortA    :out ;
.P PortB    :out ;Portb as input
.P PortC    :out
.P PortD    :in
.P portb    :in
;
;
```

Obr. 3.9: Konfigurační soubor doplňující testovací skript

Tento konfigurační soubor obsahuje číselné časové hodnoty k jejich textovým proměnným a upravuje přiřazování portů, respektive určuje, zdali jsou výstupní anebo vstupní. Druhý význam konfiguračního souboru je v kontextu programu v této práci psaném irelevantní, poněvadž, jak bylo zmíněno výše, počet výstupních a vstupních pinů je určován vždy a pro každý pin zvlášť. Program tak na konstantnosti počtu pinů není vůbec závislý, může jich být libovolný počet v jakémkoli směru.

Časové proměnné jsou však dobrým pomocníkem spisovatele testovacího skriptu a tento program je musí podporovat. Jedná se o skutečnost, kdy každá součástka je testována podle velkého množství skriptů a každá modifikace časování by vyžadovala velký zásah do všech skriptů. Proto je lepší používat proměnné místo čísel. Pro čtení konfiguračního souboru byl použit algoritmus podobný čtení skriptovacího souboru. Základem je opět třída StreamReader, jež cyklicky načítá vždy další řádek souboru, nad kterým je pak aplikována rozhodovací logika.

Je-li v paměti načten řádek, který začíná symbolem tečky a bezprostředně za ním se nachází znak velkého písmena „T“, jsou informace z řádku uloženy do datového typu slovník, kde textový řetězec je slovníkový klíč a číselná hodnota je slovníková hodnota.

```
if (line[firstChar] == '.')
{
    if (line[firstChar + 1].ToString().ToUpper() == "T")
    {
        char[] separators = new char[] { ' ', ':', ';' };
        string[] tempStrings = line.Split(separators,
            StringSplitOptions.RemoveEmptyEntries);

        PartWriter.TimeVariables.Add(tempStrings[1].Trim(),
            Convert.ToInt32(tempStrings[2].Trim()));
    }
}
```

FooterWriter

V této metodě je jen zakončena metoda *Run* a je tak zakončena výkonná část testovacího C# kódu.

HelpMethodsWriter

Jedná se zde o dva hlavní úkoly. Prvním je inicializace testovacího prostředí a druhým vypsání funkcí pro odchytávání logovacích zpráv a jejich přeposílání.

Metody umožňující přímou komunikaci s testovacím hardwarem jsou obsaženy v tzv. ovladačích, o kterých již v této práci bylo napsáno. Tyto metody se nacházejí ve

třídě TestConfigurator, jež je třeba před začátkem samotného testování instanciovat. Děje se tak ve funkci Init, jež je na stejné úrovni s výše zmíněnou funcké *Run*. Pro lepší názornost je ukázka kódu z hotového C# souboru a ne z metod tyto řádky tvořících.

```
private TestConfigurator tc;
public string[] messenger = new string[10];
public event EventHandler NewMessage;
public long testTime = 4300;
public bool Init()
{
    tc = new TestConfigurator(@"konfigurační soubor",
       @"ovladače",false);
    if (!tc.IsHealthy)
        return false;
    else
    {
        tc.LoggerOutput += tc_LoggerOutput;
        return true;
    }
}
```

Nejdříve je vytvořená privátní instance třídy TestConfigurator tc. Dále je alokováno pole deseti textových řetězců messenger, které budou sloužit k předávání zpráv od testovací fixtury k vyšším vrstvám testovacího modelu. Systém zpráv bude nyní popsán.

Jak již bylo naznačeno, vytvořený C# kód bude dále interpretován kódovým interpretem, jemuž bude věnována zvláštní sekce textu později. Při aplikaci takového systému se vyskytl problém s delegováním a odchytáváním událostí. Aby bylo možné napojit na sebe datové typy, ať jsou jakékoliv, z interpretovaného kódu do uživatelské aplikace, bylo využito rozhraní, které obsahovalo stejně pojmenované proměnné. Tento princip však lze užít jen u jednoduchých typů. Projdou tak datévé typy: textový řetězec, číslo či EventHandler, který neobsahuje žádný argument.

Z tohoto důvodu byl vymyšlen systém, kdy nativní událost ovladače *tc*, *LoggerOutput*, při svém výskytu zapíše svůj obsah do pole řetězců messenger. Obsah této události je naplněn informacemi pro logování testování. Jedná se o čas, typ zprávy, její vážnost, textový obsah a další informace.

```
void tc_LoggerOutput(MessageContainer Message)
{
    OnNewMessage();
    messenger[0] = Message.ToString();
    messenger[1] = Message.Time.ToString();
    messenger[2] = Message.Category.ToString();
    messenger[3] = Message.Severity.ToString();
```

```
        messenger[4] = Message.Message.ToString();
        messenger[5] = Message.Data.ToString();
        messenger[6] = Message.CallingMethod.ToString();
        messenger[7] = Message.CallingType.ToString();
}
```

Zároveň při výskytu této zmíněné, řekněme interní, události, je vyvolána další událost pojmenovaná *NewMessage*, která je odchytitelná i v prostředí uživatelské aplikace, a její další zpracování tak bude popsáno níže.

Nakonec zbývá ještě vysvetlit proměnnou

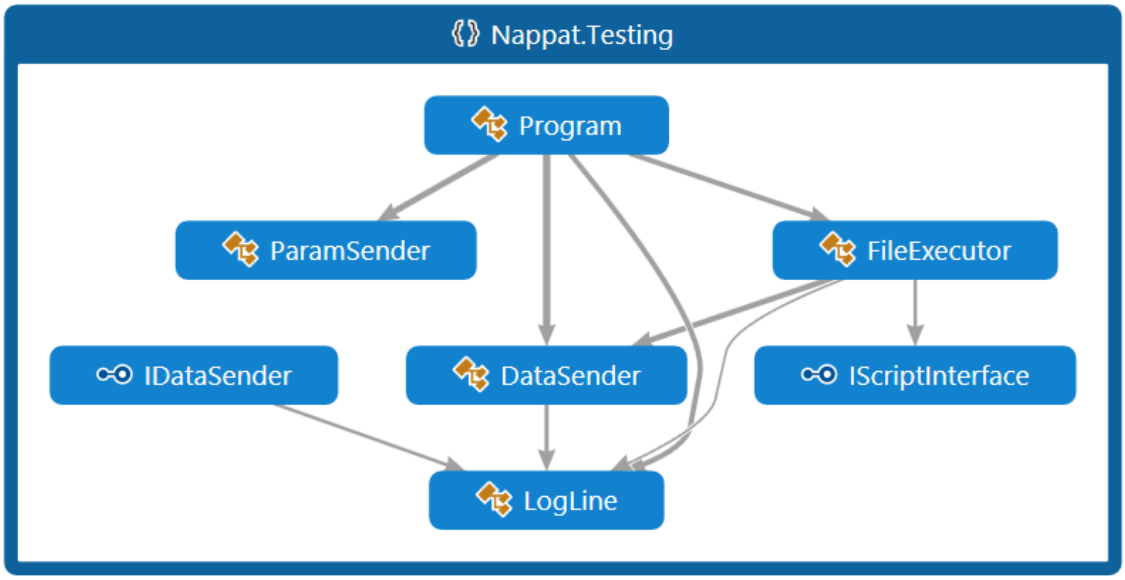
```
public long testTime = 4300;
```

.

Zde je prostor pro umístění informace o celkovém trvání skriptu. Ten je počítán během vytváření kódu v metodě *BodyWriter* a slouží ke správné funkčnosti ProgressBar (lišty pokroku) v GUI.

3.2 Testování (Testing.dll)

Nyní se situace nachází ve fázi, kdy byly testovací pokyny přeloženy ze starého formátu testovacích skriptů do kódu jazyka C#, který lze již užít v nově vyvinutém prostředí. V kontextu samotného testovaní se však jedná o počáteční stav. O provádění testování na základě pokynů obsažených v C# kódu se starají metody ve třídách sestavení Testing. Závislostní diagram sestavení je vykreslen na obrázku 3.10.



Obr. 3.10: Závislostní diagram sestavení Testování (Testing.dll)

3.2.1 DataSender.cs

V tomto souboru se nachází třída DataSender, jež obsahuje datové typy, které ukládají veškeré nastavitelné parametry testování i samotného programu . Je tak činěno z důvodu dobrého návrhového mravu a výhod z této skutečnosti plynoucích. V návrhovém vzoru MVVM bychom mohli vidět analogii ve třídě ViewModel. Jakékoliv vyšší a nižší struktury se pak napojují pouze na toto praktické rozhraní, které v sobě obsahuje veškeré potřebné informace k funkčnosti aplikace. Bylo-li by maximálně dodrženo návrhové schéma MVVM, stačilo by pak pomocí funkce Binding (volně přeloženo jako svázání či navázání) poskytované .NET Framework, provázat kontroly uživatelského rozhraní s vlastnostmi a událostmi této třídy. Kvůli složitosti celého programu tomu tak v tomto projektu není a problému bude ještě věnována pozornost.

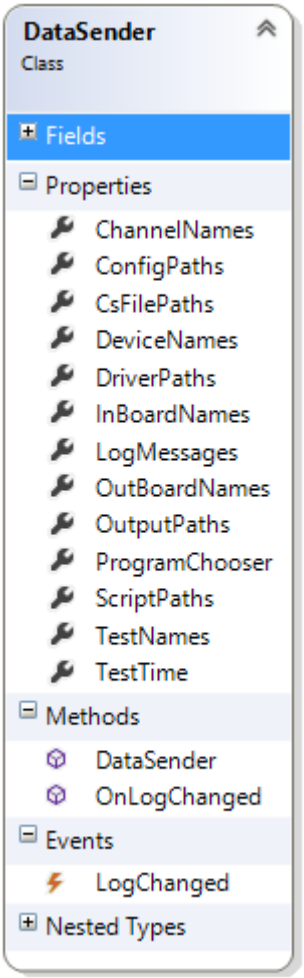
Jak bylo zmíněno výše, datové typy ve třídě DataSender v sobě nesou informace o umístění souborů potřebných k testování, název testování a názvy testovacích komponent. Dále informace o délce trvání testu, použitou pro funkcionalitu *ProgressBar*, zvoleném programu, tj. zdali se bude testovací předpis pouze interpretovat anebo je nutné jej vytvořit (volat metody sestavení CsCreator) a poslední logovací zprávu. S logováním je dále spojena událost *LogChanged*, která je vyvolána při příchodu další logovací věty.

Tato třída je serializovatelná a využívá se při uživatelem vyvolaném pokynu uložení nastavení testování nebo při inicializačních procedurách. Některé vlastnosti však serializované nejsou, poněvadž v situacích, kdy jsou výše zmíněné funkcionality využívány, nejsou známy anebo jsou irelevantní. Tyto vlastnoti jsou označeny atributem *XmlIgnore*. Třídový diagram je připojen na obrázku 3.11.

3.2.2 Program.cs

Třída Program umístěná ve stejnojmenném souboru s příponou cs je hierarchicky nejvýše v celém sestavení a její třídový diagram je na obrázku 3.12. Ve svém konstruktoru přijímá data typu DataSender, který byla popsána výše, a dále vytvoří instanci třídy ParamSender. Tyto třídy se liší svým obsahem, třída ParamSender je fakticky podmnožinou třídy DataSender a slouží pro komunikaci se sestavením CsCreator. Konverze třídy DataSender na typ ParamSender je prováděna v metodě *ParamSenderMethod* a je volána pouze v případech, kdy je třeba vytvořit nový C# kód z testovacího skriptu, tj. je požadováno služeb sestavení CsCreator.

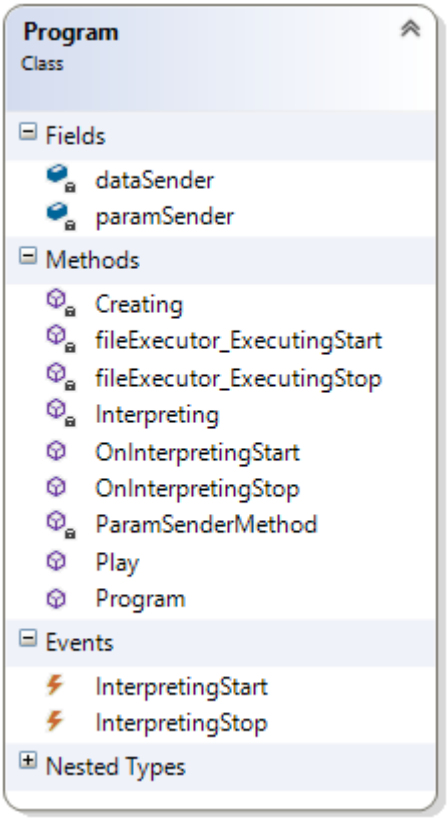
Hlavní metodou třídy je metoda *Play*, která volá metody *Creating* a *Interpreting*, respektive pouze *Interpreting*, dle toho, jaký typ programu je uživatelem zvolen, tj. zdali předkládá v minulosti již vytvořený C# předpis testování či starý testovací skript. Tato skutečnost je uložena v proměnné *ProgramChooser* třídy DataSender



Obr. 3.11: Diagram třídy DataSender

a nabývá hodnot „1“ nebo „2“. Hodnota této proměnné není měněná přímo uživatelem, je volena automaticky tím, zdali uživatel zadává parametry pro testování či pro vytvoření kódu a testování. Metoda *Creating* musí být volána před metodou *Interpreting* a nedělá nic víc, než že vytváří instanci třídy CsCreator ze sestavení CsCreator, v parametru ji předává data v datovém typu ParamSender, která byla předtím zkopírována z proměnné DataSender a volá její metodu *CreateCode*, jenž celý proces startuje. Kromě toho ještě obsahuje dva řádky s informací pro logovací.

Metoda *Interpreting* obstarává start samotného testování. Děje se tomu tak, že vytvoří instanci třídy FileExecutor (bude popsáno níže) a volá její metodu *Execution*, jež celý proces řídí. Mimo dvou řádků s informacemi pro logování obsahuje tato metoda i přidání metod *fileExecutor_ExecutingStart* a *fileExecutor_ExecutingStop* do jejich adekvátních delegátů – správců událostí typu ExecutingStatusEventHandler ze třídy FileExecutor. Tyto události jsou ve vyšší, uživatelské, vrstvě aplikace



Obr. 3.12: Diagram třídy Program

využívány pro správné fungování kontroly *ProgressBar*. Díky jejich univerzálnosti je však lze využít i v jiných případech. Pokud by bylo striktně dodrženo návrhového vzoru MVVM, byly by tyto události obsaženy ve třídě DataSender. Celou situaci aplikace by to ovšem značně zkomplikovalo a celkový benefit z tohoto řešení by nebyl kladný.

3.2.3 FileExecutor.cs

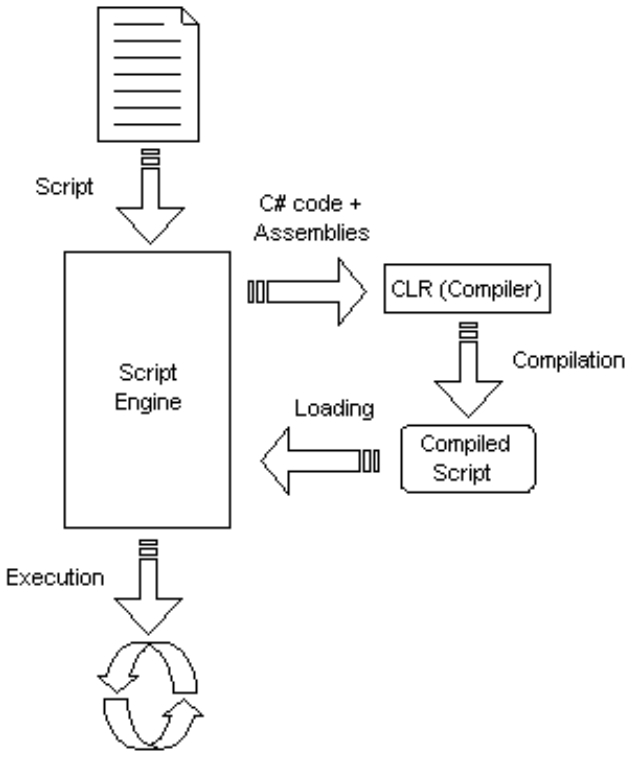
V třídě FileExecutor, se nachází vykonávací část programu a zde se v předchozích metodách vytvořený C# kód provádí. Vyplývá z předchozího, že celý princip systému je takový, že je vytvořený C# kód, který je zapsaný v souboru s příponou cs a následně je tento soubor bez dalších aditiv předán procesům, jenž jej provádějí. Toto řešení se může zdát složité, a kdyby z kódu generovaného na základě skriptů nebyly vytvářené řetězce a zapisovány do zmiňovaného cs souboru, ale rovnou byl prováděn, ušetřil by se zdrojový kód, výpočetní zdroje i čas, jelikož takovéto řešení by bylo mnohem jednodušší.

Bylo však rozhodnuto, že i přes komplikace spojené s interpretováním externího

kódu, bude systém aplikován. Výhoda systému nastává v případech, kdy jedno testování bude opakováno, v této situaci jsou volány pouze metody interpretace na v minulosti již vygenerovaný C# soubor a dále je celý program velice modulární. Pokud by během evoluce systému došlo ke změnění syntaxe C# testovacího kódu, vykonávající část aplikace bude stále plně funkční.

CS-Script

Pro interpretaci je v aplikaci využíváno kódu třetí strany. Jedná se o metody projektu CS-Script. CS-Script je „open-source“ skriptovací systém založený na CLR využívající programovacího jazyku C# a jeho implementace do projektu je díky podpoře tzv. NuGet Package (balíčků) velmi snadná. Princip fungování znázorňuje obrázek 3.13. Vstupním skriptem je v našem případě vytvořený C# soubor s metodami pro provádění testování. Výkonná část systému provede pomocí CLR kompilaci kódu a spustí jeho vykonávání. [2]



Obr. 3.13: Vizualizace prinipů fungování CS-Scriptu.

Manipulace s tímto systémem je jednoduchá. K jeho implementaci do projektu jsou potřeba dvě dynamické knihovny csscript a CSScriptLibrary. V konkrétním řešení je pak použit pro spouštění interpretace tento postup.

```
Assembly assembly = CSScript.LoadCode(interpretedCode);
Object obj = assembly.CreateObject("AutomatedTesting." +
    TestNameMethod());
```

Nejdříve se vytvoří sestavení *assembly* pomocí metody *LoadCode* z knihovny CSScript z vygenerovaného C# souboru. Poté je třeba vytvořit objekt, ze kterého budeme volat metody obsažené v C# souboru. V tomto daném případě se jedná o název třídy obsahující testovací kód. Metody i hodnoty proměnných lze z objěktované třídy volat, respektive získat jejích obsah, pomocí reflexe, jak je naznačeno na následujícím řádku.

```
obj.GetType().GetMethod("Run").Invoke(obj, null);
```

Tento řádek kódu provede faktické zavolání metody *Run* z externího C# souboru.

Je-li ve vytvořeném kódu takovýto obsah:

```
namespace AutomatedTesting
{
    public class Test00
    {
        public void Run()
        {
            (..)
        }
    }
}
```

řádek

```
obj.GetType().GetMethod("Run").Invoke(obj, null);
```

je pak ekvivalent k

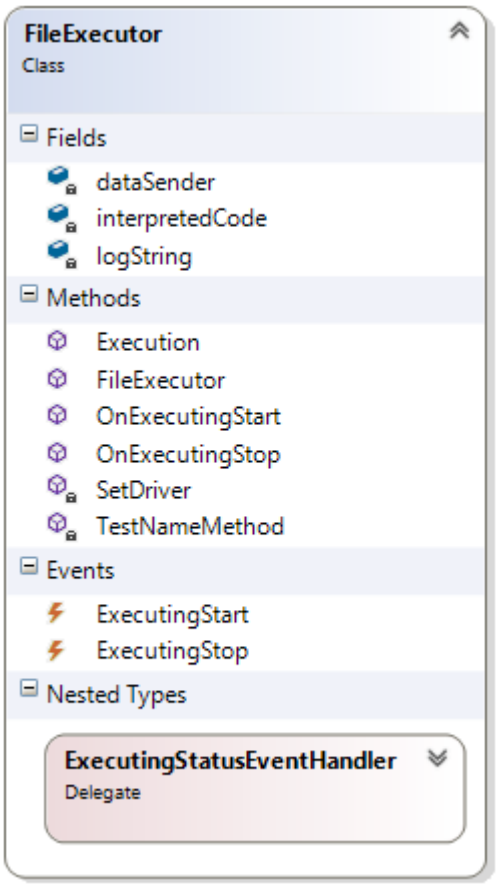
```
AutomatedTesting.Test00.Run();
```

.

Třída FileExecutor

V třídě FileExecutor, se nachází vykonávací část programu. Zde se v předchozích metodách vytvořený C# kód provádí. Diagram třídy je na obrázku 3.14.

Hlavní výkon provádí metoda Execution a lze tvrdit, že tato metoda fakticky spouští testování. Nejdříve je do paměti načten vytvořený C# testovací soubor pomocí třídy StreamReader, který je dále předáván interpretu. Následně jsou zavolány metody C# kódu *Init* a *Run*, které jsou již metodami samotného testování. Kód okolo těchto metod, které jsou volány pomocí reflexe, slouží ke zpracovávání informací posílaných výše v testovacím modelu od testovacích fixtur, k testovacímu technikovi.



Obr. 3.14: Diagram třídy FileExecutor

Jedná se o následující situace. Hlavní inicializační metoda kódu, popsaná v části CsCreator, vrací hodnotu pravda/nepravda dle toho, zdali proběhla inicializace v pořádku. Opakem je například případ, kdy by nebyly přiloženy funkční ovladače. Vrátí-li metoda *Init* pozitivní hodnotu, může výt zavolána metoda *Run* a započato tak testování. Před tím se však ještě nachází kód reagující na logovací události testování. Z důvodu jednoduchosti a přehlednosti kódu je tato událost připojena pomocí navázaného rozhraní IScriptInterface a zpracovávána pomocí lambda výrazu.

```
boundObject.NewMessage += (s, e) =>
{
    this.logString = obj.GetType().GetField("messenger").GetValue(obj)
        as string[];
    this.dataSender.LogMessages = new LogLine(logString[1],
        logString[2], logString[3], logString[4], logString[5]);
    if (logString[3] == "FatalError")
    {
        th.Abort();
    }
```

```
};
```

Jelikož systém navazování rozhraní v interpretaci neumožňuje, aby v parametru události byla posílána jakákoliv data, musí být při výskytu události získána pomocným algoritmem, jenž je obsahem lambda výrazu na předchozí ukázce. Při výskytu události, tj. nějaké logovací informace od testovací fixtury, je do lokální proměnné *logString* uložen obsah proměnné z interpretovaného kódu *messenger* stejného typu.

Byl hledán systém, který by při výskytu chyby při testování, testování ukončil. Testovací fixtura o tomto faktu podává informaci pomocí logu, jež obsahuje v poli vážnosti zprávy řetězec „FatalError“. Problém byl vyřešen spouštěním testování v samostatném novém vlákně *th* a kontrolováním každé příchozí zprávy tak, jak je naznačeno na předcházející ukázce.

Třída FileExecutor dále obsahuje metody pro zvýšení uživatelské pohodlnosti. Aby testovací technik nemusel při jednoduché interpretaci již vytvořeného kódu zadávat opět název testu, který je potřebný pro vytvoření objektu, jak bylo popsáno výše, existuje privátní metoda *TestNameMethod*, která z načteného cs souboru název testu získá.

Pro zachování základní přenositelnosti generovaného C# kódu mezi počítači, je umístění ovladačů přidáváno vždy až bezprostředně před interpretací kódu v metodě *SetDriver*. Při použití CS-Scriptu se přidávají reference jako komentář a uvozením „css_reference“. Takový odkaz může vypadat následovně:

```
//css_reference B:\\Libraries\\TestConfigurator.dll
```

.

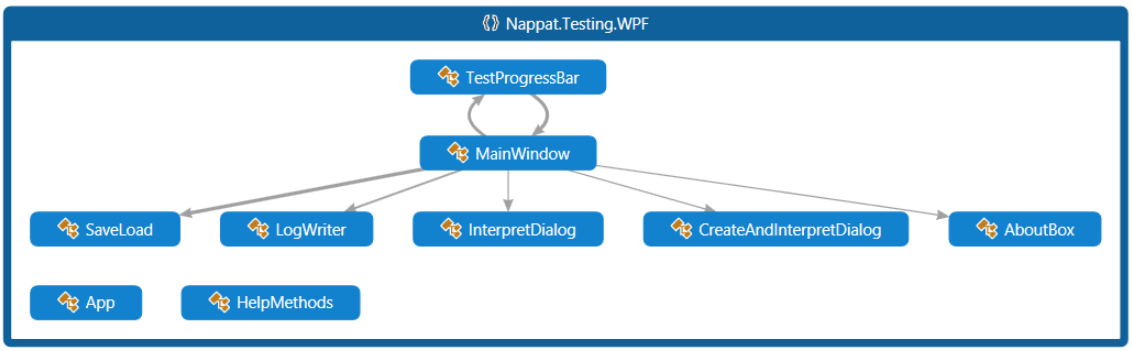
3.2.4 LogLine.cs

Ve třídě LogLine, umístěné v souboru LogLine.cs je obsažena definice logovací věty. Tato třída obsahuje vlastnosti dle jednotlivých logovacích informací a přetížený konstruktor, jeden pro nastavení nulových hodnot vlastností a jeden pro jejich hodnotné nastavení. Zajímavostí této třídy je přepsána metoda *ToString*, jež skládá řetězec znaků testovací věty dle přání techniků společnosti.

```
public override string ToString()
{
    string retString = this.Time.ToString("HH:mm:ss") + ": " +
        this.Category + ": " +
            this.Severity + ": " +
                this.Message + ": " +
                    this.Data;
    return retString;
}
```

3.3 Grafické uživatelské rozhraní – WPF.exe

Nejvýše položenou položkou testovacího nástroje je uživatelské rozhraní, přes něž testovací technik řídí testování a získává zpětnou vazbu. Z hlediska projektu jsou sestavením se souborovou příponou exe. Jsou spustitelným souborem, jež ovšem pro plnou funkčnost vyžaduje popsané dynamické knihovny CsCreator.dll a Testing.dll. V práci existují dvě uživatelská rozhraní a to grafické – GUI a zdrojově úspornější konzolová verze. Diagram grafického rozhraní je na obrázku 3.15.



Obr. 3.15: Závislostní diagram sestavení Testing.WPF

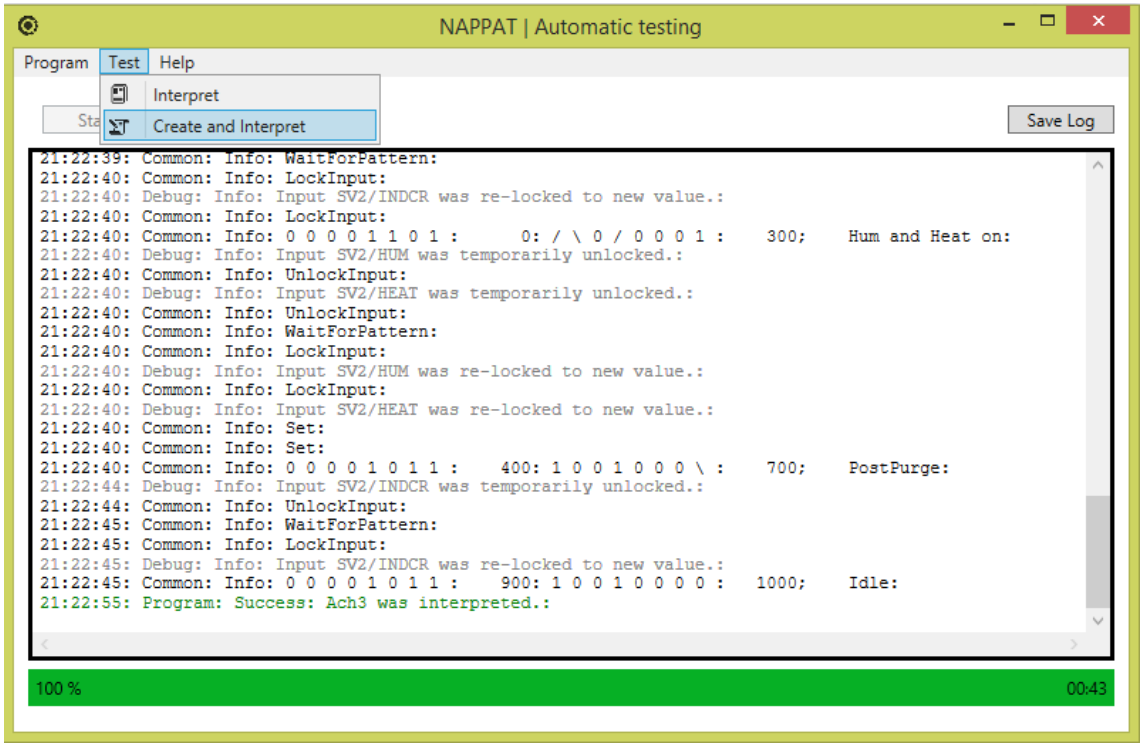
Celé spustitelné sestavení WPF.exe je relativně rozsáhlé, přičemž většinu jeho obsahu zaplňují obslužné metody událostí vyvolaných uživatelem interakcí s GUI kontrolami. Z tohoto důvodu budou v následujícím textu popsány pouze některé metody a sekvence kódu, a to ty, jež autor nepovažoval za primitivní či jsou důležité z hlediska kontextu celého projektu a jeho funkčnosti.

Objekt popisující grafické uživatelské rozhraní je u technologie WPF rozdělen do dvou souborů s příponou xaml a cs. Soubor xaml obsahuje definici grafického vzhledu okna, tj. pozici jednotlivých kontrol a jejich vlastností. Je psán v jazyce Xaml, jenž vychází ze značkovacích jazyků. V této práci nakonec nebyly použity funkcionality přesahující možnosti automatického generovaní kódu vývojovým prostředím, a tak souborům xaml nebude věnována pozornost.

3.3.1 MainWindow.xaml.cs

Tento soubor obsahující parciální třídu stejného jména je hlavním prvkem celého sestavení, je vykonáván bezprostředně po spuštění programu a v jeho xaml souboru je definováno grafické rozhraní hlavního okna.

Důraz byl kladen především na jednoduchost a uživatelskou přátelskost. Hlavnímu oknu, obrázek 3.16, dominuje prvek *TextBox*, s vlastností „Pouze pro čtení“, pro výpis informací o testovaní a logovacích informací. Nad touto komponentou jsou



Obr. 3.16: Testovací prostředí Nappat

umístěna tlačítka pro spuštění a zastavení testování a uložení logu. V horní části hlavního okna je umístěna lišta s nabídkou položek menu. Pod záložkou Program má uživatel na výběr z možnosti uložení a načtení nastavení a ukončení programu. Vlevo od záložky Program je umístěna záložka Test. V této záložce se vybírá typ programu. Kliknutím na daný prvek vyskočí určené dialogové okno pro zadání informací o testování.

Výkonný kód, jehož metody jsou v třídovém diagramu na 3.17 je složen z obslužných metod na události vyvolané stiskem tlačítek GUI a otevřením a zavřením okna.

Inicializační metody

Pro zvýšení efektivity práce s aplikací, jsou do programu implementovány inicializační funkce, které zajistí načtení předchozího testování při spuštění aplikace. Hlavní kód tohoto systému je uložen v souboru SaveLoad.cs, jemuž bude věnována zvláštní podsekce. V hlavním okně je spuštění metody pro načtení inicializačního souboru zabezpečeno následujícím kódem.

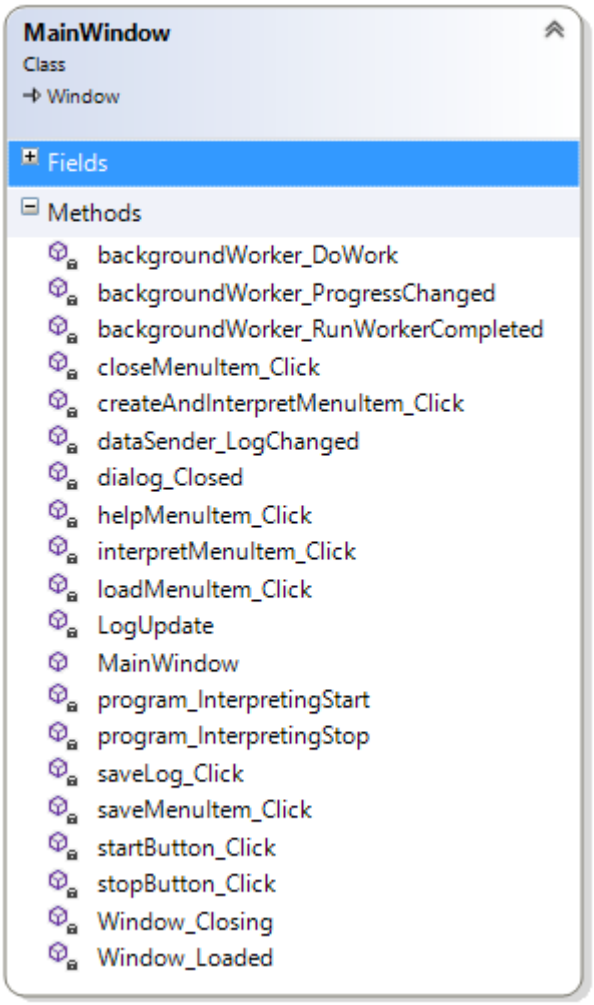
```
private void Window_Loaded(object sender, EventArgs e)
{
    (...)
```

```
    SaveLoad sl = new SaveLoad(dataSender);
    sl.DataTransporter(dataSender, sl.LoadInit());
                (...)
}
```

Ukládání je vyvoláno analogicky v metodě *Window_Closing* zavolané při události zavření hlavního okna, například kliknutím myši na červené ukončující tlačítko v pravém horním rohu okna.

Metoda startButton_Click

Tato metoda je obsluhou události kliknutí na tlačítko Start a způsobuje spuštění testu. Po spuštění aplikace je tlačítko neaktivní a aktivuje se vždy až po zadání testovacích údajů testovacím technikem. Po spuštění testu se toto tlačítko opět deaktivuje. Tímto mechanismem je uživatel nucen vždy zkontrolovat zadané parametry



Obr. 3.17: Diagram parciální třídy MainWindow

testování.

Po stisknutí tlačítka „Start“ se nejdříve ještě provede kontrola, zdali uživatel opravdu zadal všechna potřebná data pro spuštění testování. Bude-li tomu tak, je tlačítko „Start“ deaktivováno, aktivováno tlačítko „Stop“ a zavolána metoda instance *backgroundWorker RunWorkerAsync*.

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    if ((this.dataSender.DriverPaths != null &&
        this.dataSender.ScriptPaths != null) ||
        (this.dataSender.DriverPaths =! null &&
        this.dataSender.CsFilePaths != null))
    {
        if (backgroundWorker.IsBusy != true)
        {
            backgroundWorker.RunWorkerAsync();
            this.startButton.IsEnabled = false;
            this.stopButton.IsEnabled = true;
        }
    }
}
```

V předkládané práci je funkcionalita tlačítka „Stop“ omezená.

Ukládání logu

Logovací informace získané od nižších vrstev testovacího systému, ale i hlášky testovacího nástroje, jsou vypisovány do hlavní textové kontroly okna. Uživatel má mít možnost si záznam vypsaný v *TextBoxu* uložit. Tuto funkci poskytuje metoda *saveLog_Click* vyvolaná událostí kliknutí na tlačítko „Save Log“.

Nejdříve je vyvolán ukládací dialog instanciováním třídy SaveFileDialog poskytované prostředím .NET. Prostřednictvým tohoto dialogu uživatel vybere místo, kam si přeje log uložit. Ze seznamu textových řetězců obsahující stejné věty, jaké byly vypisovány do *TextBoxu*, se vytvoří textový soubor, jenž je automaticky pojmenován stejně jako název testování a uložen na vybrané místo pomocí třídy File z .NET knihovny System.IO.

```
private void saveLog_Click(object sender, RoutedEventArgs e)
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    saveFileDialog.Filter = "Plain Text|*.txt";
    saveFileDialog.Title = "Save Testing";
    if (loggerList != null)
    saveFileDialog.ShowDialog();
    if (saveFileDialog.FileName != "")
```

```
    {
        System.IO.File.WriteAllLines(saveFileDialog.FileName,
            loggerStrings);
    }
    (...)
}
```

LogUpdate

Kontrola *TextBox* hlavního okna slouží pro vypisování logovacích sekvencí a informuje tak uživatele aplikace. Princip vypisování zpráv je následující. Při vyvolání události instance objektu DataSender LogChanged, která nastane vždy, je-li změněna proměnná *logMessages* typu LogLine, tj. je přidána nová logovací věta, se zavolá metoda hlavního okna *LogUpdate*. O zapisování textových řetězců to *TextBoxu* logu se stará zvláštní třída LogWriter, jejíž metoda *UpdateBox* je volána. Tato třída bude posána níže v textu. Jelikož zprávy přicházejí většinou z testovacího systému, který je spuštěn v jiném vlákně než hlavní okno, je nutné popsaný algoritmus provádět pod dohledem dispečera tak, jak je ukázáno na následujícím kódu.

```
private void LogUpdate()
{
    this.Dispatcher.Invoke((Action)(() =>
    {
        LogWriter lw = new LogWriter(this.logTextBox);
        lw.UpdateBox(dataSender);
        if (testProgressBar.CanProgress)
            testProgressBar.Progress(dataSender);
    }));
}
```

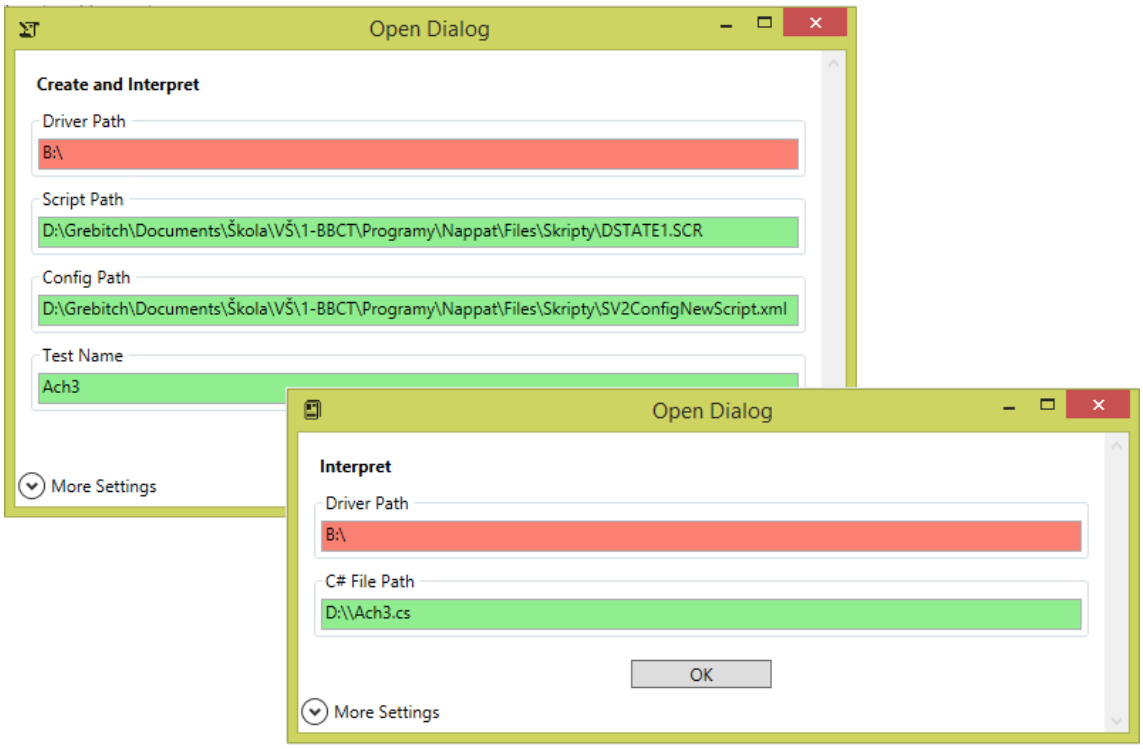
Metoda *LogUpdate* je také s výhodou využita pro obsluhu kontroly *ProgressBar*. Z tohoto důvodu je v kódové ukázce volána metoda Progress instance třídy testProgressBar.

3.3.2 Dialog.xaml.cs

Kromě hlavního okna jsou v aplikaci ještě dva objekty typu Window. Jedná se o dialogová okna, ve kterých uživatel zadává testovací parametry. Dle typu testování uživatel v menu liště hlavního okna zvolí, zdali chce testovaní provádět na základě starého testovacího skriptu anebo použije již vytvořený C# kód.

Obě okna jsou velmi podobná a plní stejnou funkci. Jediný rozdíl je v povinně zadatelných údajích, jež je u volby pouhé interpretace méně. Ukázáno na 3.18.

Hlavním prvkem obou oken je kontrola *TextBox*, do níž testovací technik zadává požadovaná data. Nad každým prvkem *TextBox* je volána validační metoda dle typu



Obr. 3.18: Dialogová okna programu Nappat

údaje. Jedná-li se o pole pro zadání adresy k souboru, je ověřováno, zdali k ní má program přístup a zdali existuje. V případě zadávání jména je kontrolován fakt, neobsahuje-li zadaný řetězec nebezpečné znaky, například znak zpětného lomítka. Je-li vyplněný údaj správný, pole zezelená, v opačném případě je pozadí kontroly zbarveno červeně. Ověřovací metody jsou spouštěny událostmi reagujících na změnu obsahu a při načítání kontroly. Pro ukázku byl vybrán kód použitý pro ověření existence ovladačů.

```
private void DirectoryValidator(object sender, EventArgs e)
{
    if (Directory.Exists(((TextBox)sender).Text))
        ((TextBox)sender).Background = Brushes.LightGreen;
    else
        ((TextBox)sender).Background = Brushes.Salmon;
}
```

4 MOŽNÁ VYLEPŠENÍ TESTOVACÍHO SYSTÉMU

Největší slabinou současného systému testování ve společnosti Honeywell, divizi ACS je nutnost používání testovacích skriptů ve starém formátu a konfiguračních souborů, bez kterých nelze testování provádět. Manipulace s těmito soubory je komplikovaná a zdrojem častých chyb.

Jednodušším, avšak stále provizorním řešením by bylo aplikovat do programu metody, které by automaticky přečetly konfigurační XML soubor a uživateli alespoň předvyplnily textová pole pro zadávání názvů testovacích komponent.

Rozsáhlejším projektem by bylo vytvořit aplikaci pro psaní skriptů. Tato aplikace by měla možnost načítat i staré typy skriptů a uživatelé by je mohli následně editovat v tzv. Drag&Drop prostředí a stejně tak i vytvářet nové testovací scénáře. Výstupem tohoto editoru by byl C# kód, který by v sobě obsahoval jednak testovací metody, ale i kompletní nastavení testování, čímž by přebíral i roli konfiguračního souboru. Fakt, že by se celý systém zjednodušil, by neměl vliv na přenositelnost, respektive byla by ještě navýšena, zmíněných C# testovacích souborů. Názvy testovacích objektů mají jen minimální vliv na tuto vlastnost kódů a jediná cesta k souborům, která by musela být uživatelem zadávána, by bylo umístění ovladačů, které by se zadávalo v programu globálně a ne pro každé testování.

Program v této práci psaný byl vytvářen tak, aby takovéto změny mohly být snadno implementovány přímo do něj anebo by bylo alespoň možné použít jeho části či sestavení, například sestavení pro překlad skriptů na kód.

5 ZÁVĚR

Bakalářská práce se zabývala tvorbou uživatelského nástroje pro podporu automatizovaného funkčního testování prototypů ve vývoji ve společnosti Honeywell divizi ACS v Brně. Jedná se o počítačový software, jenž zpracovává testovací skripty vytvořené ve starém formátu a provádí je v prostředí nového testovacího systému. Výsledná aplikace má grafické uživatelské rozhraní sloužící pro nastavení skriptovacích parametrů a zobrazení průběhu testu v reálním čase. Průběh a výsledky testu lze ukládat do souboru.

Architektura a implementace testovacího prostředí ve vývojovém centru není jednoduchá a její pochopení patřilo k obtížnějším prvkům práce. Seznamování se s oběma systémy probíhalo postupně, zejména na základě dodané specifikace, a nakonec bylo dosaženo stavu, kdy jsem byl srozuměn se vším potřebným pro sestavení testovacího nástroje.

Funkčnost aplikace byla úspěšně otestována na sadě skriptů a dále se také podařilo na program aplikovat uživatelsky příjemné, intuitivní a praktické grafické rozhraní. Návrh GUI byl průběžně konzultován s vývojovými a testovacími techniky společnosti.

Vzhledem k faktu, že primárním účelem programu je tvořit most mezi starým a novým testovacím systémem, je celý projekt koncipován tak, aby byl co nejmodulárnější a přinesl společnosti vysokou využitelnost i v budoucnosti. Proto je řešení rozděleno do tří nezávislých sestavení propojených definovanými rozhraními. Nejobtížnějším prvkem práce bylo vytvoření logovací infrastruktury, kdy se logovací informace musí dostat z interpretovaného kódu až do kontroly uživatelského rozhraní.

Kromě uživatelského rozhraní je k testovacímu nástroji přidán i konzolový ovladač, jenž plní funkci uživatelského rozhraní na méně výkonných výpočetních stojích typu Raspberry Pi. Díky modulárnímu konceptu řešení projektu byla aplikace konzolového rozhraní jednoduchou záležitostí.

LITERATURA

[1] Česká republika. *HONEYWELL.* [online]. 2014, [cit. 2014-5-19]. Dostupné z: http://honeywell.com/worldwide/Pages/Czech_Republic-en.aspx.

[2] Documentation Online Help. SHILO, Oleg. *CS-Script* [online]. 2004-2013, [cit. 2014-5-20]. Dostupné z: http://www.csscript.net/help/Online/indexTutorial.html.

[3] HONEYWELL. *Today's Honeywell: Company Overview* [online]. 2013, 22 s. [cit. 2014-5-20]. Dostupné z: http://honeywell.com/About/Documents/TodaysHoneywell_13_0711.pdf.

[4] Honeywell - Honeywell v ČR. HONEYWELL. *Honeywell: Domů* [online]. 2013, [cit. 2014-5-20]. Dostupné z: http://honeywell.jobs.cz/prace-v-honeywellu/honeywell-v-cr/cz/.

[5] Honeywell History. HONEYWELL. *Home* [online]. 2013, [cit. 2014-5-20]. Dostupné z: http://honeywell.com/About/Pages/our-history.aspx.

[6] Multiple Platform Support: Microsoft .NET Framework. *Microsoft Česká republika: Zařízení a služby* [online]. 2014, [cit. 2014-5-18]. Dostupné z: http://www.microsoft.com/net/multiple-platform-support.

[7] NAGEL, Christian, Bill EVJEN, Jay GLYNN, Morgan SKINNER a Karli WATSON. *C#2008:Programujeme profesionálně.* Brno: Computer Press, a.s., 2009. ISBN 9978-80-251-2401-7.

[8] Our Company. HONEYWELL. *Home* [online]. 2013, [cit. 2014-5-20]. Dostupné z: http://honeywell.com/About/Pages/our-company.aspx.

[9] ŠKOLICÍM CENTRU IT BRNO. *.NET Platforma.* Brno, 2010. [online]. 2013, [cit. 2014-5-19]. Dostupné z: http://www.fit.vutbr.cz/study/courses/IW1/public/info/lib/exe/fetch.php?media=iw5:01_net.ppt

[10] WATSON, Karli, Christian NAGEL, Jacob Hammer PEDERSEN, Jon REID a Morgan SKINNER. *Beginning Visual C# 2010* Indianapolis, Indiana: Wiley Publishing, Inc., 2010, xxxix, 1037 p. ISBN 04-705-0226-6.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ACS Automation and Control Solutions

CLI Common Intermediate Language

CLR Common Language Runtime

CS C# – Programovací jazyk C#

GUI Graphical User Interface

LI Intermediate Language

MVC Model-View-Controlle

MVVM Model-View-ViewModel

OS Operační systém

WPF Windows Presentation Foundation

XML Extensible Markup Language

SEZNAM PŘÍLOH

A POČÍTAČOVÝ PROGRAM NAPPAT

V této příloze se nachází praktický výsledek bakalářské práce. Jedná se o program Nappat, který slouží jako nástroj pro funkční testování prototypů ve vývoji ve společnosti Honeywell, divize ACS v Brně.

V adresáři Nappat se kromě zdrojových kódů programu nachází, řešení Nappat.sln, jehož spuštění bylo testováno v programech VS2012 a VS2013 a složky jednotlivých projektů.

Pro správnou funkčnost je nutno přidat knihovny

- CommonFunctions.dll
- TestConfigurator.dll
- TestInterfaces.dll
- TestTypes.dll

do projektu Nappat.Testing.WPF[1]. Tohoto stavu lze nejjednodušeji docílit zkopírování těchto sestavení do složky Debug/Release projektu Nappat.Testing.WPF. Pak již stačí pouze zkompilovat řešení.

A.1 Testování

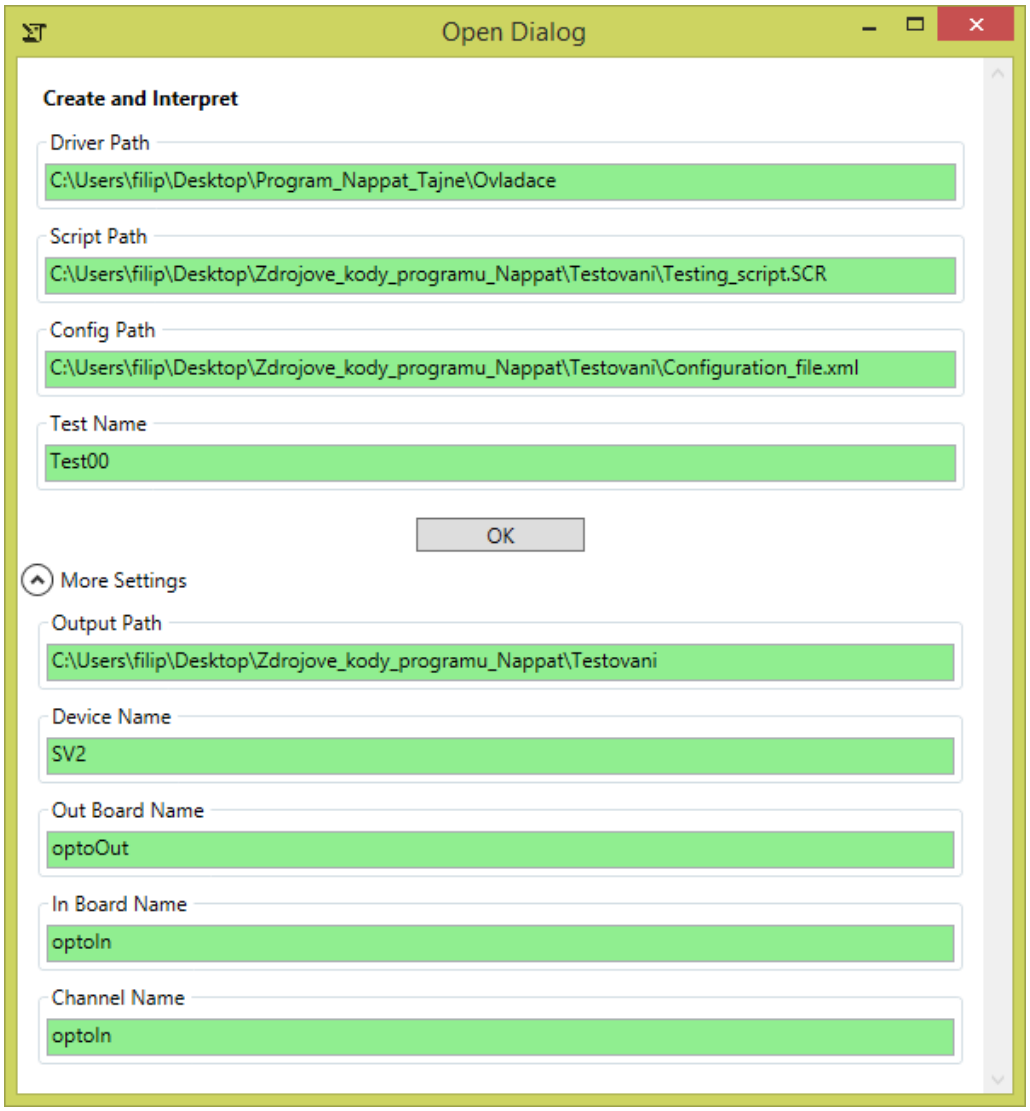
Pro praktické použití programu jsou zapotřebí ještě testovací soubory, které jsou uloženy ve složce „Testovani“ a fiktivní ovladače [1].

Jelikož na Vašem počítači pravděpodobně ještě nikdy nebylo testování prováděno, není zde ani přítomen inicializační soubor a při startu programu se objeví upozornění.

Pro začátek testování klikněte na nabídkové liště na záložku „Test“ a vyberte možnost „Create and Interpret“. V objeveném dialogovém okně zadejte umístění testovacích souborů a testovací parametry. Ovladače jsou ve složce „Ovladace“ a testovací soubory ve složce „Testovani“. Po rozevření zadávací nabídky zadejte dále umístění výstupních souborů, název testovacího zařízení, výstupní a vstupních komponent testovací fixtury a komunikační kanál. Tyto údaje musí přesně korespondovat s údaji v konfiguračním souboru, jinak test neproběhne v pořádku. Obsah textových polí je ukázán na obrázku A.1.

Alternativě lze využít možnosti načtení uloženého konfiguračního souboru s příponou nsf umístěného ve složce „Testovani“. V tomto souboru jsou již parametry

[1]Ovladače a některá sestavení jsou chráněna obchodním tajemstvím společnosti Honeywell International, Inc. V případě oprávněné potřeby zdrojových kódů, lze soubory získat v IS VUT Brno anebo kontaktováním vztyčné osoby: Radomír Svoboda, Honeywell HTS Brno, Tuřanka 96, 62700 Brno, e-mail: radomir.svoboda@honeywell.com.



Obr. A.1: Ukázka nastavení testování

přednastaveny. Soubor lze načíst kliknutím na položku „Load Test“ v záložce „Program“.

Po úspěšném zadání parametrů stiskněte tlačítko OK, čímž zavřete dialogové okno a můžete začít testování kliknutím na tlačítko „Start“. Využili-li jste možnosti načtení již uloženého souboru, pro povolení funkčnosti tlačítka „Start“ je nutno potvrdit údaje v dialogovém okně Test > Create and Interpret > OK.

B VIDEOUKÁZKA PROGRAMU NAPPAT

Na přiloženém CD/DVD nosiči se nachází videoukázka testování v souboru s příponou avi. Tato ukázka byla nahrána pro případy, kdy nebude možné získat utajené soubory potřebné k vykonávání testování. Pro omezenou velikost příloh v IS VUT v Brně nemohl být výše zmíněný soubor do systému IS nahrán a nachází se tak pouze na přiloženém médiu.

C SPECIFIKACE TESTOVACÍHO SKRIPTU

V textu této práce je několikrát odkazováno na interní specifikaci společnosti Honeywell, divize ACS, která obsahuje pokyny pro vytváření testovacích skriptů. Jelikož v této práci vytvořený program tyto skripty zpracovává, bylo nutné, aby jeho metody s touto specifikací plně korespondovaly.

V této příloze je originál specifikace, k jejímuž zveřejnění byl udělen explicitní souhlas. Text je v anglickém jazyce. Příloha existuje pouze v elektronické verzi, je však volně stažitelná z IS VUT Brno.