TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

SYSTÉM PRO ANALÝZU A TRANSFORMACI DATOVÉ KOMUNIKACE ZAŘÍZENÍ POUŽÍVANÝCH VE SPORTU

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Pavel Novák**
Vedoucí práce: doc. Ing. Jiřina Královcová, Ph.D.

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

SYSTEM FOR ANALYSIS AND TRANSFORMATION OF DATA COMMUNICATION OF DEVICES APPLIED IN SPORTS

Diploma thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Pavel Novák**
Supervisor: doc. Ing. Jiřina Královcová, Ph.D.

Liberec 2014

Tento list nahraďte
originálem zadání.

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Poděkování

Tímto krátkým odstavcem bych rád poděkoval vedoucí této práce doc. Ing. Jiřině Královcové, Ph.D. za poskytnuté rady, připomínky a celkové vedení práce. Další velké poděkování bych směřoval společnosti ST Software s.r.o. (především pak konzultantovi této práce – Bc. Martinovi Froschovi) za prostor a možnost spolupráce na zajímavém projektu v rámci diplomové práce. Na závěr bych rád poděkoval celé své rodině a přítelkyni za podporu při řešení a vypracovávání této práce.

Abstrakt

Tato diplomová práce se zabývá problematikou vývoje aplikací ve sportovním prostředí. Hlavním cílem práce je návrh a implementace grafické uživatelské aplikace, která slouží jako nástroj usnadňující vývoj. Primárním úkolem výsledné aplikace je zefektivnit vývoj v aplikačním prostředí. Umožňuje manipulaci s daty v rámci komunikačních rozhraní, datovou transformaci a analýzu. Mezi hlavní možnosti využití aplikace patří přesměrování dat mezi jednotlivými komunikačními rozhraními, transformace či analýza dat, transformace protokolů, záznam a simulace datového provozu. Výsledná aplikace umožňuje uživatelům pracovat s množinou komponent. Komponenty je možné umístit na pracovní plochu aplikace a dle potřeby je propojovat a konfigurovat. Důležitým aspektem výsledné aplikace je její modularita. Výstup práce poskytuje mechanismus, pomocí kterého je možné snadno a efektivně implementovat nové komponenty, které jsou automaticky načítány. V základní verzi aplikace se nachází množina základních obecných komponent pokrývajících základní protokoly.

Tvorba výsledné aplikace je vázána na platformu .NET verze 4. Její grafická podoba je vytvořena pomocí technologie WPF (nástupce Windows Forms). Při vývoji byla snaha o maximální univerzálnost, proto jsou v aplikaci použity návrhové vzory jako Inversion of Control, Dependency Injection, MVVM a další. Aplikace je psána proti rozhraní a podle doporučovaných praktik pro vývoj software.

Přínosem této diplomové práce, respektive výsledné aplikace, je její univerzálnost a všestranné využití při vývoji aplikací ve sportovním prostředí, popřípadě i jinde, kde se vyskytuje nějaký datový provoz. Aplikace není omezena na konkrétní rozhraní nebo protokoly, jako většina případných konkurenčních aplikací. Hlavním přínosem pro vývojáře je možnost jednoduše a efektivně tvořit vlastní komponenty. Výsledná aplikace je také vhodná pro testování při vývoji, například pomocí simulace datového provozu.

Klíčová slova:

modulární aplikace, .NET, sportovní prostředí, datová analýza a transformace, komunikační rozhraní, protokoly

Abstract

This diploma thesis deals with the development of applications in the sports environment. The main goal of this work is the design and implementation of a graphic user application. This application serves as a development tool which facilitates the development. The primary task of the application is to make the development in the application environment more effective. This allows data manipulation within the communication interfaces, data transformation and analysis. The main features of the application is data routing between communication interfaces, data transformation and analysis, transformations of protocols, recording and simulation of data traffic. The final application enables users to work with a set of components. These components are possible to drop to an application desktop, connect and configure. The important feature of the application is its modularity. The output of the study provides a mechanism through which it is possible easily and effectively to implement new components. The final application contains main general components that handle the basic protocols.

The application is developed on .NET 4 platform. The graphic user interface is created on WPF technology (successor of Windows Forms). The goal was to achieve maximum of universality during the development. It contains design patterns such as Inversion of Control, Dependency Injection, MVVM, Dispose Pattern, etc. The application is written against interfaces and with the best practices for software development.

The main attribute of this thesis is its versatility in application development in a sports environment or another environment with data traffic. The application is not limited to specific interfaces or protocols, such as other potential competing applications. The main benefit for developers is the ability to easily and efficiently create custom components. The application is also suitable for testing, for example, by data simulation.

Keywords:

modular application, .NET, sports environment, data analysis and transformation, communication interface, protocols

Obsah

Seznam obrázků

Seznam výpisů

Seznam symbolů, zkratek a termínů

.NET	Vývojový framework a platforma
DI	Dependency Injection, návrhový vzor
IoC	Inversion Of Control, návrhový vzor
MVVM	Model View ViewModel, návrhový vzor
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language, značkovací jazyk
XML	Extensible Markup Language, obecný značkovací jazyk
XSD	XML Schema Definition, typ XML schématu

Úvod

Tématem této diplomové práce je systém pro analýzu a transformaci datové komunikace zařízení používaných ve sportu. Důvodem volby tohoto tématu byl zájem o vývoj aplikací, které jsou využívány ve sportovním prostředí. Dalším důvodem byl zájem o technologie .NET, které jsou moderním, efektivním a rychle se rozvíjejícím vývojářským nástrojem. Sportovní prostředí je v tomto směru zajímavé především svou pestrostí používaných zařízení, rozhraní a protokolů. Tato práce je vyvíjena ve spolupráci s libereckou firmou ST Software s.r.o., která je dceřinou společností švýcarské společnosti Swiss Timing Ltd. Tyto společnosti se zabývají vývojem měřící techniky a aplikací pro sportovní prostředí. Hlavním cílem této diplomové práce je návrh a implementace aplikace v jazyce .NET, která bude sloužit jako vývojový nástroj v aplikačním prostředí zabývajícím se sportem. Použití pouze ve sportovním prostředí však není podmínkou a výslednou aplikaci by mělo být možné využívat kdekoliv, kde se vyskytuje nějaký datový provoz. Hlavním smyslem a přínosem této práce je možnost následně využívat vytvořenou aplikaci k efektivnějšímu vývoji v aplikačním prostředí. Aplikace by měla být inovativní v její maximální univerzálnosti, nezávislosti na rozhraní a protokolech. Měla by disponovat modulárním mechanismem, pomocí kterého by mělo být snadné přidávat novou funkcionalitu aplikace, jako například obsluhu dalších rozhraní, protokolů, případně zavedení nových datových transformací. Přínosem výsledné aplikace je zvýšení efektivity vývoje, odpadá nutnost vlastnit specifická zařízení po celou dobu vývoje, mechanismus pro testování atd.

Hlavní funkce vyvíjené cílové aplikace jsou předávání dat mezi jednotlivými rozhraními, transformace protokolů a dat, jejich analýza, archivace a následná simulace. Aplikace by měla umožňovat jejím uživatelům pracovat s komponentami, které by měly být vstupní, transformační nebo výstupní. Hlavními uživateli budou především vývojáři aplikací, musí pro ně být snadné implementovat komponenty s novou funkcionalitou a jejich následné zavedení do celého systému. Pro tvorbu komponent musí být poskytnuta rozhraní, která umožní jejich okamžitou implementaci, ale musí existovat další podpůrné prostředky umožňující jejich složitější funkcionalitu. Komponenty by měly disponovat mechanismem pro jejich konfiguraci, poskytování vlastního okna, záznam stavů a další obslužné nástroje.

Aplikace by měla dále obsahovat možnost pracovat s projekty – sestavy propojených komponent se specifickou konfigurací. Pro vývojáře využívající aplikaci musí existovat možnost efektivní implementace nových komponent, které by mělo být možné do aplikace přidat jako samostatné plug-in moduly.

Samotný text práce je kromě úvodu a závěru strukturován do tří částí (kapitoly 1 až 3). První část práce seznamuje s problematikou vývoje aplikací ve sportovním prostředí, srovnává výslednou aplikaci s konkurenčními nástroji a definuje její stanovené cíle a možnosti. Druhá část práce se věnuje návrhu výsledné aplikace, kde jsou rozebrány některé použité návrhové vzory a popsána důležitá rozhraní a použité formáty. Poslední důležitá část práce se zabývá implementací aplikace pomocí technologií .NET, konkrétně implementací některých zajímavých rozhraní a kritických částí kódu. Dále jsou ve třetí kapitole popsány využité knihovny třetích stran, základní tvorba komponent a grafické prostředí včetně jednotlivých grafických ovládacích prvků.

1 Seznámení s problematikou

1.1 Vývoj SW pro sportovní prostředí

Tato část práce přibližuje vlastnosti vývoje aplikací používaných ve sportu a problémy, které je v tomto odvětví třeba často řešit. Jsou zde demonstrovány některé modelové situace. Problematika vývoje software pro sportovní prostředí má svá specifika a je v určitých praktikách odlišná od vývoje aplikací s jiným účelem použití. Jedná se především o různorodost zařízení a protokolů, se kterými se programátor takto orientovaných aplikací setká a musí je efektivně řešit. Závěrem této části je definování cílů výsledné aplikace, která by měla splňovat požadavky pro efektivní řešení problematických a klíčových situací. Jejím hlavním cílem je ulehčení práce vývojářům v tomto odvětví. Příloha C demonstruje možná využití na hotových schématech z výsledné aplikace.

1.1.1 Sportovní prostředí

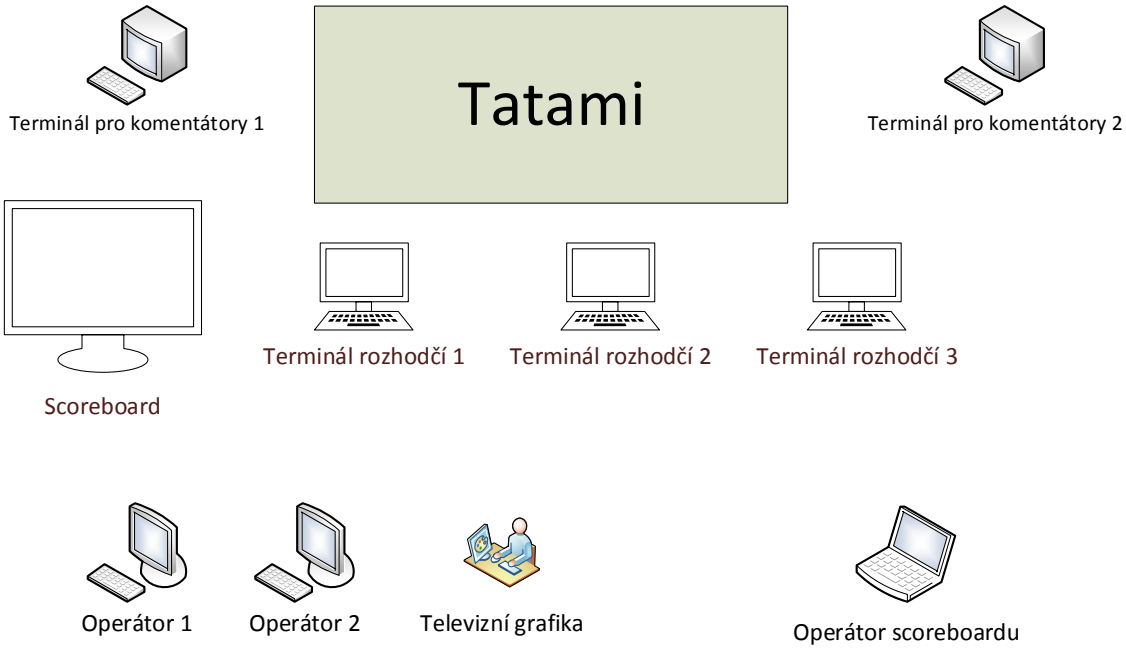
Pojem sportovní prostředí je poměrně široký. V kontextu této práce se jedná především o prostředí vyšších sportovních akcí, konkrétněji světových her, pohárů, mistrovství a šampionátů. Zmíněné sportovní akce jsou naprosto nezávislé na jednom konkrétním sportu, nebo nějakém úzkém okruhu sportů. Do sportovního prostředí také patří data o sportovcích, měřící technika na sportovních utkáních, poskytování dat s výsledky na internet nebo do dalších specifických sítí, televizní grafika, operátoři utkání, informační tabule (scoreboard) a další elementární prvky.

V tomto prostředí vystupuje velká skupina lidí. Jedná se o sportovce, rozhodčí, ale také komentátory, operátory, režiséry, zaměstnance televizní grafiky, osobnosti ze sportovních federací a další. Většina ze zmíněných má v průběhu akce přímý styk s používanou technikou. Rozhodčí ovládají terminály pro záznam výsledků a průběžných informací, komentátoři používají terminály s detailními informacemi o průběhu utkání a s daty od rozhodčích, operátoři kontrolují celkový stav a prohlašují výsledky za oficiální. Lidé pracující s televizní grafikou následně poskytují oficiální data do přehledových tabulek a informačních stavových lišt, které jsou zobrazovány ve sportovních přenosech, které jsou živě vysílané v televizních stanicích.

Měřící technika na sportovních utkáních je velice rozmanitá. Pod měřicí technikou si lze představit jakékoliv zařízení, které má spojitost s vyhodnocením výsledků daného utkání či turnaje. Může se jednat o klasickou automatickou časomíru, GPS tracker, či specifické zařízení. Mezi měřící techniku je také možné zařadit terminály pro rozhodčí, kteří je používají k hodnocení a určování průběhu zápasu. Veškerá zmíněná měřící technika komunikuje s dalšími zařízeními pomocí různorodých rozhraní (sériová linka, ethernet, Wi-Fi, …) a používá různé, často nestandardní, komunikační protokoly.

Samotná data, naměřená a vyhodnocená měřicí technikou, je třeba distribuovat dále. Zde se v tomto řetězci ve většině případů objevují operátoři (ve výjimečných případech nejsou třeba a celý proces může být plně automatizován). Operátor je člověk, který kontroluje běh celkového systému jako celku a sleduje korektnost dat. Mnohdy musí operátor výsledky utkání či zápasu potvrzovat za správné. Potvrzením, že jsou data správně, jsou označena jako oficiální a mohou se distribuovat dále, konkrétně například pro televizní grafiku a internet.

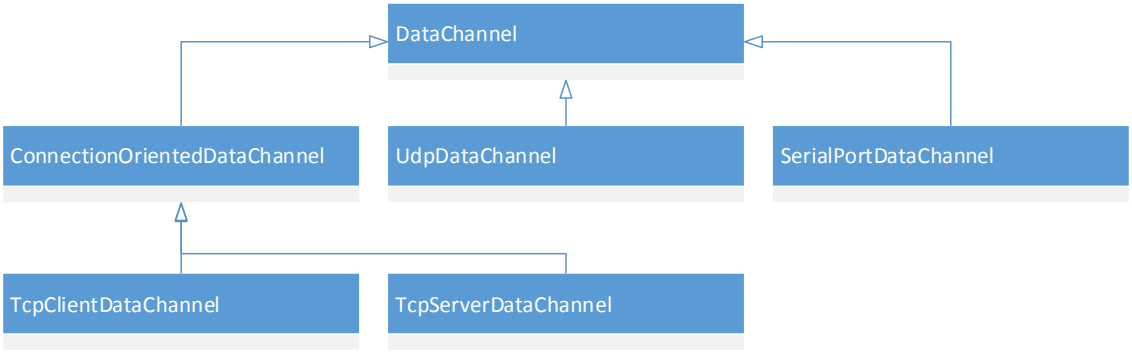
Obrázek 1 zjednodušeně demonstruje malou sportovní akci, konkrétně jde o zápas v bojovém sportu. Tatami označuje oblast, kde účastníci vybraných bojových sportů zápasí. Nejblíže tatami jsou umístěny terminály pro rozhodčí. Na obrázku jsou dále vidět terminály pro komentátory, ty ale typicky sídlí dále od hrací plochy ve specializované buňce s akustickou izolací. Kolem tatami se také nacházejí informační tabule (scoreboard) pro diváky sledující přenos. Informační tabule zobrazují aktuální skóre a průběh zápasu a bývá jich dostatečný počet, aby je mohli sledovat diváci ze všech stran. V delší vzdálenosti od tatami se nachází počítače operátorů akce, televizních grafiků a operátora veřejných informačních tabulí. Všechna zmíněná zařízení jsou spolu spojena v rámci jedné, či více sítí a komunikují spolu. Obrázek je pouze orientační, pro každý sport a každé utkání se variabilně mění podle konkrétních potřeb. Demonstruje jednoduchou variantu, kde nejsou znázorněny televizní kamery a další měřicí a vyhodnocující přístroje.



Obrázek 1: Demonstrační pohled na menší sportovní akci

1.1.2 Různorodost rozhraní a protokolů

Hlavním problémem vývoje aplikací ve sportovním prostředí je právě různorodost komunikačních rozhraní a protokolů. Z používaných rozhraní se nejčastěji vývojář setká se sériovou komunikací, ethernetem či Wi-Fi. Ze sériové komunikace se nejvíce používá klasická sériová linka RS-232 a více průmyslově zaměřený standard sériové komunikace – RS-485. U ethernetu se typicky využívá přenosových médií kroucené dvoulinky, někdy se v tomto prostředí vyskytuje i koaxiální kabeláž. Různorodost rozhraní může mít vliv na efektivitu vývojáře. Z tohoto důvodu existuje ve společnosti ST Software knihovna RLib, která tvoří abstrakci napříč těmito rozhraními. Této knihovny bude využito v některých síťových komponentách v cílové aplikaci práce. Obrázek 2 znázorňuje abstrakce knihovny RLib na několika základních rozhraních. Samotných datových kanálů se v knihovně nachází více. Jednotlivé datové kanály mají společného předka *DataChannel*, se kterým v aplikaci vývojář může snadno operovat a nemusí se zabývat konkrétní implementací. Tu je samozřejmě možné nastavit pevně v aplikaci, nebo je možné ji variabilně měnit. Toho lze docílit například za pomoci konfiguračních souborů a aplikačního frameworku Spring [1], který je k dispozici pro většinu běžně používaných programovacích jazyků.



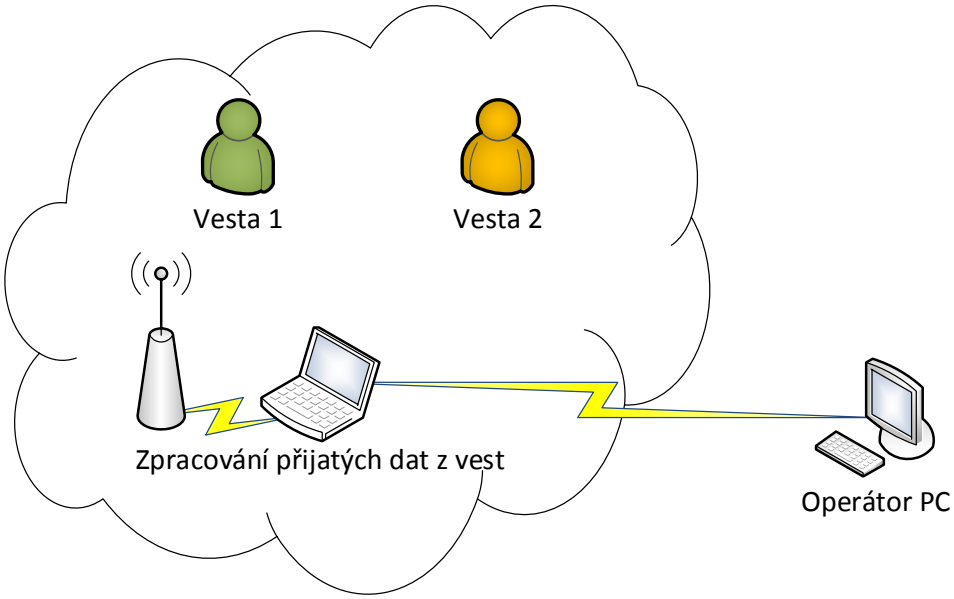
Obrázek 2: Ukázka abstrakce knihovny RLib

Dalším problémem při vývoji je množství protokolů, které se ve sportovním prostředí vyskytují. Každá společnost v tomto odvětví se pro své výrobky (měřicí přístroje, video-servery, …) snaží prosazovat jejich protokoly. Díky různorodosti zařízení ve sportu ani není možné definovat jednotný protokol, protože jedním protokolem není možné, kupříkladu, ovládat veřejnou informační tabuli a přenášet data z GPS trackerů ze závodů koní v dostizích. Tento fakt se projevuje na efektivitě vývojářů, protože s každým dalším projektem, ve kterém se pracuje s nějakým novým protokolem, musí vývojář takový protokol studovat a umět ho implementovat. Velká rozmanitost protokolů je vidět například u video serveru XT2 od společnosti EVS [10][1]. Tento video server je možné ovládat pomocí šesti odlišných komunikačních ovládacích protokolů. Konkrétně se jedná o Sony, DD35, Odetics, VDCP, AVSP a protokol LinX.

1.1.3 Modelové situace

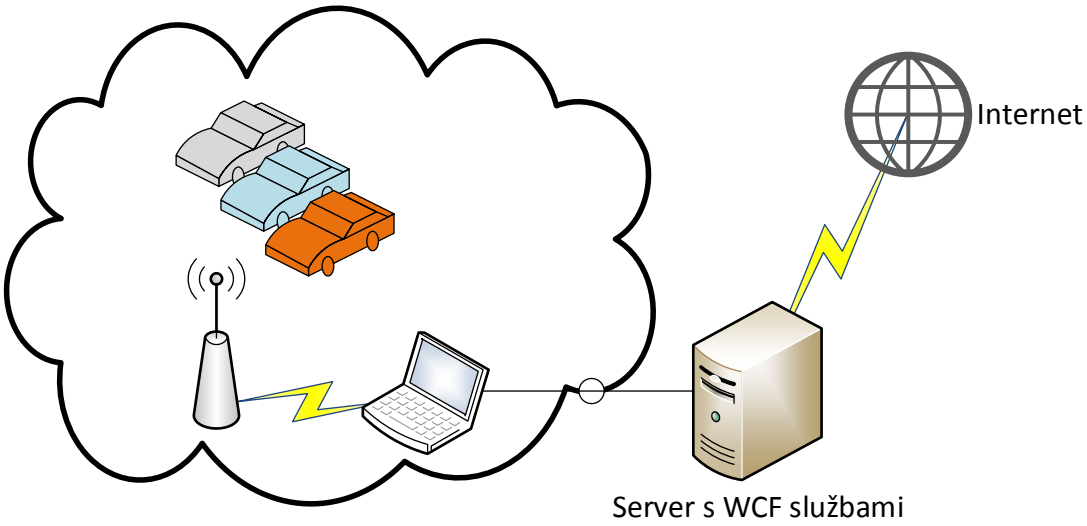
V této části práce bude demonstrováno několik modelových situací, které je nutné ve sportovním prostředí velmi často řešit. Situace budou přiblíženy formou jednoduchých ilustrací s konkrétními problémy. Různých situací nastává v reálném životě mnohem více, zde se jedná především o demonstraci pro bližší představu a pohled na pestrost vývoje aplikací ve sportu. Sportovní prostředí je z pohledu vývoje velice odlišné od ostatních prostředí právě proto, že sport jako takový je velice variabilní a jednotlivých sportů existuje nespočet. Z toho také vyplývá používání různorodé techniky, ať už měřící, tak té, která slouží pro úchovu dat, případně jejich distribuci dalším systémům.

Obrázek 1 znázorňuje první modelovou situaci. Jedná o zápas v bojovém sportu, například v thajském boxu. V tomto případě je nutné řešit mnoho elementárních i komplexních problémů. Nejprve je nutné mít funkční registrační systém, do kterého jsou registrováni jednotliví zápasníci a další významné osoby. Veškerá technika zobrazená na obrázku musí být správně strategicky spojena v několika sítích. První privátní síť slouží k propojení počítačů operátorů s terminály rozhodčích, televizní grafikou a s informačními tabulemi včetně operátora těchto tabulí. V další síti jsou z důvodu bezpečnosti odstíněni od první sítě například terminály komentátorů, případně mohou být data z této sítě rovnou poskytována do sítě internet pro sledování aktuálního průběhu utkání. Tato data mohou sloužit například zpravodajským serverům jako zdroj aktuálních informací. Z této akce je patrné, že je nutná poměrně rozsáhlá příprava, kterou musí společnost zajišťující zpracování výsledků obstarat a vykonat. Musí být vymyšlena komunikace mezi jednotlivými technickými elementy (počítače, terminály, switche, spojení, …). Také musí být připraven software usnadňující práci operátorům a dalším účastníkům. Je nutné mít vytvořený robustní registrační systém a aplikaci, která je ovládána operátory, kteří pomocí ní kontrolují průběh a stav vyhodnocování výsledků. Také musí být naprogramovány klientské aplikace obsluhující terminály pro rozhodčí, komentátory a veřejné informační tabule. Proces vývoje takto velkého množství aplikací, které spolu komunikují v rámci sítě, je poměrně složitý na testování. Zde by výsledná aplikace této práce byla nápomocna hned v několika případech. Pokud by některé prvky komunikovaly pomocí sériové linky (nebo nějakého méně obvyklého rozhraní), ne vždy je jednoduché a žádoucí při vývoji celou dobu být připojen pomocí specifického spojení. Zde by bylo výhodné, kdyby jeden vývojář, který tvoří aplikaci, která data vysílá, data odchytával pomocí komponenty sériové linky a data přímo přeposílal (případně i logoval pro následnou simulaci provozu) například po TCP/IP vývojáři, který tvoří aplikaci, která data pouze čte a zobrazuje (například veřejný informační panel). Druhý programátor by cílovou aplikaci pouze použil pro odchycení dat z TCP/IP, která by pomocí aplikace přesměroval například na virtuální sériový port. Tím by vypadla při vývoji nutnost spojení pomocí sériové linky u všech vývojářů.



Obrázek 3: Modelová situace - vesty na taekwondo

Obrázek 3 je specifičtější částí předchozí popsané situace. V tomto případě se jedná o zápas v taekwondu. V těchto zápasech se používají k vyhodnocení výsledku elektronické vesty s helmou a rukavicemi, které umožňují detailnější statistiky ze zápasu a přesnější hodnocení. Tyto vesty snímají a odesílají informace o místě a síle úderu. Tento specifický měřící systém poskytuje společnost TKD Score [29]. Vesty komunikují s řídící jednotkou a přenáší data přes bezdrátovou technologii Wi-Fi. Na řídící jednotce jsou data vyhodnocována a přes specifický protokol společnosti TKD Score se mohou poskytovat dále. Na popisovaném modelu jsou zpracovaná data předávána po ethernetu právě do počítače operátora. V tomto místě opět vzniká situace, kdy vývojář aplikace pro operátora musí implementovat protokol společnosti TDK Score. Zde by mohla být cílová aplikace této práce pomocná tím, že by vývojář mohl vytvořit novou komponentu, která by byla určená k simulaci tohoto protokolu. Například by do komponenty mohl nadefinovat tlačítka, která by při stisku měla za následek odeslání různých stavových kódů protokolu, případně odesílání některých testovacích dat. Výstup z této komponenty by byl směrován do komponenty obstarávající TCP/IP výstup a odtud by data doputovala přímo do vyvíjené aplikace pro operátora. Taková komponenta by vývojáři velice zjednodušovala průběh vývoje aplikace a umožňovala efektivní testování přímo při vývoji. Další využití je možné pro logování datového provozu a jeho následná simulace.



Obrázek 4: Modelová situace - měření rychlosti kol aut

Obrázek 4 demonstruje zjednodušený systém měření rychlosti kol aut při závodech. Každé auto má na všech kolek implementováno zařízení, které je schopno měřit rychlost otáčení kola a data odesílat. Toho může být podle aktuálních podmínek docíleno různě, například přenosem dat přes GSM síť, případně v závodech, kde se jezdí nějaký okruh na více kol, se mohou data odeslat při průjezdu klíčovými místy. Cílem je, co nejaktuálnější naměřená data, distribuovat na internet, kde jsou poskytována přes webové služby, buď přes REST či SOAP. Poskytování těchto služeb lze docílit pomocí WCF [20]. V tomto případě může výsledná aplikace této práce pomoci obdobně, jako v předchozích modelových situacích, při vývoji obsluhy protokolů, či odchytu a následné simulaci datového provozu. Další významnou pomocí může být implementace komponenty, která by dokázala konzumovat data z webové služby WCF a data vizualizovat, archivovat a testovat. Při vývoji by bylo možné do komponenty také zaintegrovat například graf průběhu rychlosti. Zmiňovaná vizuální komponenta by byla využitelná jako přehledný a okamžitý testovací prvek.

1.2 Cíle práce

V předchozím textu byly uvedeny tři modelové situace, ve kterých by bylo vhodné mít k dispozici nástroj usnadňující vývojářům tvorbu softwaru. Hlavním cílem této práce je návrh a implementace aplikace, která bude využitelná a prospěšná při vývoji softwaru ve sportovním prostředí. Aplikace ovšem může nalézt uplatnění při vývoji jakýchkoliv aplikací, i mimo sportovní prostředí, které

nějakým způsobem pracují s daty. Aplikace by měla umožňovat jejímu uživateli na pracovní plátno umístit požadované komponenty, které budou mít různou funkci a tyto komponenty pospojovat. S aplikací by měla být dodána základní sada komponent, která bude pokrývat nejvíce využívané prvky. V případě, že vývojář bude potřebovat komponentu, která nebude součástí, musí být zajištěn mechanismus pro její snadnou implementaci. Hlavním cílem je tedy univerzálnost a průhlednost aplikace, díky které by vývojáři mohli vytvářet komponenty co nejefektivněji, a tím by šetřili čas při vývoji. Vyjma ušetřeného času je v určitých případech možnost danou datovou komunikaci lépe otestovat, a tím zajistit vyšší spolehlivost výsledného kódu. Tato práce je vyvíjena ve spolupráci se společností ST Software s.r.o., ve které veškerý vývoj probíhá na produktech společnosti Microsoft. Z tohoto důvodu je kladen požadavek na to, aby byla výsledná aplikace napsána v jazyce C# a běhovém prostředí .NET. Z cílových požadavků na aplikaci vyplývá, že musí podporovat:

- Tři základní typy komponent. Jedná se o komponenty vstupní, transformační a výstupní. Vstupní komponenta nemá žádný vstup, ale jen výstup či výstupy. Reprezentuje nějaký datový vstup, typicky z nějakého síťového rozhraní, souboru, uživatelského vstupu a tak dále. Komponenty transformační disponují jak vstupem/vstupy, tak výstupem/výstupy. Ačkoliv jsou v této práci tyto komponenty nazývány jako transformační, jelikož to bude jejich nejčastější účel, není transformace dat v této komponentě podmínkou. Může se jednat i o komponenty analyzující datový provoz. Výsledky analýzy se mohou poté například kreslit do grafu. Výstupní komponenty reprezentují nějaké výstupní zařízení, nebo přesměrování na zvolené rozhraní. Mají pouze vstup a jsou protikladem komponent vstupních.
- Možnost konfigurace jednotlivých komponent, kde by měl být zajištěn mechanismus, který bude konfiguraci jednotlivých komponent ukládat pro další použití. Každá komponenta musí mít možnost vlastnit jiné konfigurační atributy, než mají ostatní komponenty.
- Ukládání celých schémat (dále jen projekty). Sestavené projekty musí být možné ukládat a následně upravovat. Projekt je záznam obsahující množinu

komponent, nastavení jednotlivých komponent, jejich vzájemné vazby a případné dodatečné informace.

- Mechanismus, který by umožňoval textový výstup, kde by mohla být zobrazována data, která komponentou proudí. Komponenta jako taková by neměla mít nutnost tento mechanismus implementovat a musí jít pouze o volbu vývojáře, zda tuto funkcionalitu využije.
- Možnost vytvoření okna, které bude naprosto nezávislé na zbytku aplikace (ačkoliv bude v jejím vlastnictví). Takové okno komponenty může její vývojář definovat přesně podle svých potřeb. Může se jednat o složitější konfiguraci komponenty, ovládací prvky komponenty, nebo například grafy se stavovými informacemi atd. Toto okno, neboli vývojářem definovaný výstup komponenty je také pouze volitelnou vlastností komponenty.
- Základní bázová třída pro vývoj asynchronních komponent. Ty by bylo možné vyvinout i bez takového mechanismu, ale sjednocení v tomto směru přináší rychlejší implementaci a následnou lepší správu vláken z aplikace jako takové.
- Mechanismus logování dat. Logovat je nutné všechny chybové stavy, neobvyklé a nečekané situace, stavové a další informace. Zachycené záznamy v rámci aplikace by měly být archivovány v externím souboru v čitelném textovém formátu. Další možnost logování dat by měla obsahovat i cílová aplikace formou panelu, kde uživatel uvidí ihned případné stavové informace.

Základem pro uvedené vlastnosti je samostatná aplikace s grafickým uživatelským rozhraním, která vývojáři dovolí rychlé použití, propojení a správu jednotlivých komponent. Dále musí být její uživatel schopen komponenty konfigurovat a spravovat jednotlivé projekty. Aplikace musí automaticky načítat vytvořené komponenty, jejichž binární reprezentace je uložena v předem definovaném umístění. Nalezené komponenty by měly být zobrazeny v aplikační nabídce komponent. Vývojáři musí být poskytnuto API, pomocí kterého by mohl vytvářet nové komponenty. Toto API včetně vývoje komponent by mělo být absolutně odstíněno od samotné aplikace. Tento přístup zajistí vyšší univerzálnost výsledného celku.

1.3 Konkurenční/obdobný software

Existuje řada aplikací, které nějakým způsobem manipulují s daty. Některé jsou určeny k přeposílání dat, jiné k transformaci a další například k analýze. Existují i komplexnější nástroje, které se snaží spojit více zmíněných vlastností. Takové aplikace ovšem vždy plní jen jeden, či několik úkolů a nenabízí požadovanou univerzálnost. Většina takových aplikací také nedisponuje modulárním systémem, pomocí kterého by bylo možné aplikaci rozšiřovat o další komponenty.

Při hledání alternativy pro vyvíjený software v rámci této práce bylo nalezeno množství aplikací, které umožňují přesměrování dat mezi jednotlivými rozhraními. Typicky se bohužel jedná pouze o jednoúčelové aplikace, které splňují vždy obsluhu jen jedné kombinace rozhraní. Takové nástroje obvykle existují jak v placených, tak zdarma dostupných variantách. Do této skupiny lze zařadit například následující aplikace:

- tcp2com [26],
- TCP/Com [23],
- Comm Tunnel Pro [4].

Předchozí seznam zmiňuje pouze zlomek aplikací pro tento účel, které existují. První v seznamu, tcp2com, je k dispozici zdarma a slouží pouze k vytvoření datového přemostění mezi TCP/IP a sériovou linkou. Další dvě aplikace jsou distribuovány komerčně a umožňují přeposílání dat ze sériové linky na jeden či více TCP/IP klientů. Comm Tunnel Pro navíc umožňuje data redistribuovat mezi sériovou linkou, TCP/IP a UDP rozhraními. Problém s těmito aplikacemi nastává, pokud by byla potřeba využít jiného rozhraní, případně nějak data transformovat nebo analyzovat. Žádná ze zmíněných aplikací nenabízí možnost pro tvorbu vlastních modulů.

Existují i přímo systémové nástroje, které umožňují manipulaci s daty. V Linuxu lze například použít aplikaci *route*, která umožňuje přesměrovávat datový provoz ethernetového spojení. Zde opět vzniká problém, že se jedná jen o jeden úkol cílové práce a ostatní úkoly by se musely zajistit separátně.

Zmíněné aplikace a možnosti je možné využít, ovšem ve většině případů, které se ve sportovním prostředí vyskytují, jsou nedostatečné. Mnohdy je nutné

využít kombinaci více aplikací, často ani tato varianta není například z důvodu datové transformace/analýzy proveditelná. V tomto ohledu je aplikace, která je cílem této práce, inovativní a snaží se vývojáři poskytnout jedno prostředí, ve kterém může být využito všech zmiňovaných vlastností a snadné implementace dalších funkcí.

2 Návrh

Primárním cílem této diplomové práce je návrh a implementace aplikace v jazyce C#, která bude sloužit jako vývojový nástroj pro aplikační oblast, která se zabývá vývojem aplikací pro sportovní prostředí. Aplikace by měla být také použitelná kdekoliv mimo sportovní prostředí, kde se vyskytuje nějaká manipulace s daty. Návrh aplikace je tvořen sadou rozhraní. Aplikace by měla umožňovat rozsáhlou možnost zacházení s daty, konkrétně umožnit vývojáři přesměrovávat proud dat z rozhraní na jiná rozhraní, data případně transformovat, či analyzovat. Zároveň musí být pro vývojáře zajištěn mechanismus, kterým budou moci aplikaci rozšiřovat o další komponenty. V tomto směru je také nutno zajistit maximální univerzálnost pro tvorbu komponent, aby vývojáři v tomto směru nebyli jakkoliv omezeni. Výsledná aplikace byla pojmenována **CommSpy**. Název byl odvozen dle jejího potencionálního využití napříč komunikačními rozhraními. Tato část práce seznamuje se samotným návrhem aplikace, konkrétně s jejím celkovým sestavením, komunikací mezi jednotlivými částmi a univerzálním modulárním systémem komponent. Návrh disponuje především sadou rozhraní, která jsou dále při implementaci využitelná pro:

- realizaci komponent,
- konfiguraci komponent,
- záznam stavů komponent,
- nástroje manipulující s komponentami,
- práci s projekty,
- sestavení aplikace.

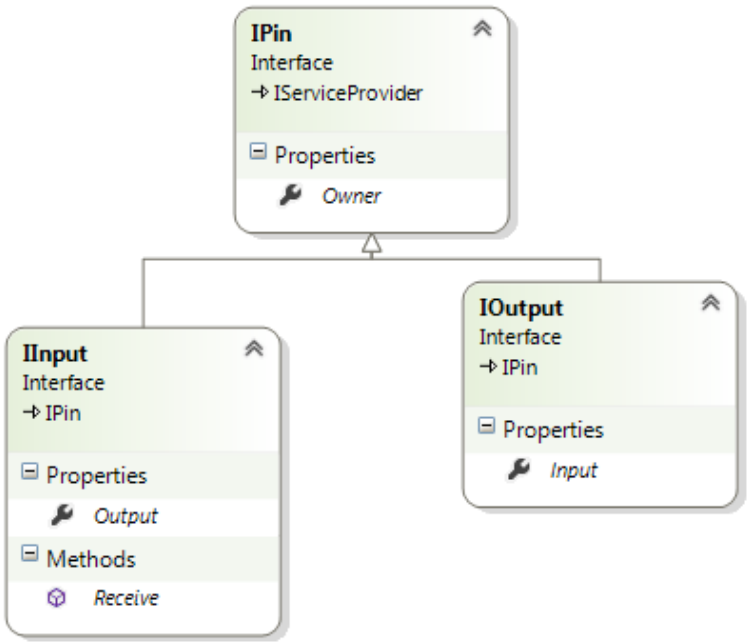
Prvky návrhu jsou uvedeny v následujících podkapitolách.

2.1 Komponenty

V prvním kroku návrhu aplikace bylo nutné vymyslet mechanismus, který umožní tvorbu komponent. U těch je nutné, aby měly určitou abstrakci, díky které je bude možné v aplikaci načítat jako jednotlivé moduly. Komponenty musí splňovat několik důležitých vlastností. Mezi tyto vlastnosti patří univerzálnost. Tím je myšleno, že vývojář by neměl být při tvorbě komponenty jakkoliv omezen a mohl by do ní naimplementovat v podstatě cokoliv. Tento přístup by nemusel být ve většině běžných aplikací zcela bezpečný, ale pro vývojový nástroj používaný především

v rámci jedné společnosti, pouze vývojáři, se jeví jako nejoptimálnější. Další důležitou vlastností je rychlost a komfortnost implementace nové komponenty. To při návrhu struktury rozhraní pro komponenty hrálo velice významnou roli. Základní rozhraní pro tvorbu komponent tedy obsahuje jen ty vlastnosti, které jsou pro její existenci opravdu nutné. Je zbytečné implementátora zatěžovat zbytečnými detaily. Pokud takové detaily potřebuje více programátorů, mohou použít rozšířená rozhraní nebo nějakou třídu ve formě předka a tím si práci usnadnit. Součástí aplikace by měla být také bázová třída, která bude většinu obecných vlastností implementovat a tím opět šetřit vývojářům čas.

Při návrhu komponenty byla nejprve navržena její nejelementárnější součást – konektor neboli pin. Konektor zprostředkovává komponentě datový vstup nebo výstup (dle typu). Přes objekt konektoru se data do komponenty podsouvají. Komponenta je dle své implementace zpracuje a posílá dále na její výstupní pin. Konektory také budou v konečné grafické aplikaci sloužit k snadnému propojování a určování datového toku mezi komponentami. Každá komponenta disponuje alespoň jedním vstupním, či výstupním pinem. Vstupní komponenta obvykle vlastní jen konektor/konektory výstupní, zatímco výstupní komponenta obsahuje pouze pin/piny vstupní. Transformační komponenty obsahují vstupní i výstupní piny.

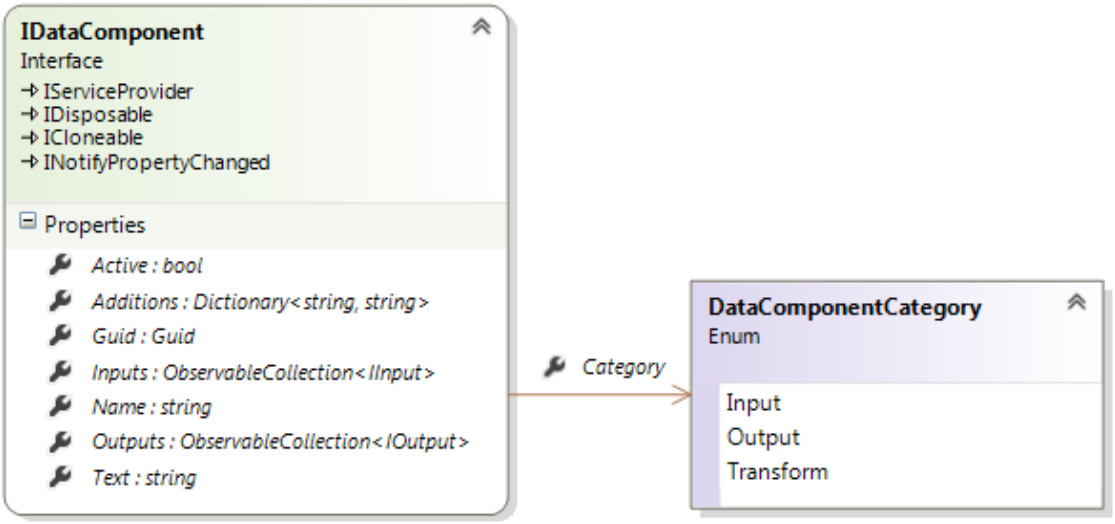


Obrázek 5: UML diagram konektorů komponenty

Obrázek 5 znázorňuje UML diagram s rozhraními vstupních a výstupních pinů a jejich společného předka. Základní rozhraní *IPin* obsahuje pouze vlastnost *Owner*. *Owner* je typem rozhraní komponenty. V určitých situacích je vhodné, když samotný konektor nese informaci, které konkrétní komponentě patří. Důvody budou rozebrány v části věnující se implementace aplikace. *IOutput* obsahuje vlastnost *Input* typu *IInput*. Z pohledu aplikace se jedná o referenci na následující vstupní konektor komponenty, která po ní následuje. Pokud tato vlastnost nabývá hodnoty *null*, značí to, že ke konkrétnímu pinu není nic připojeno. Stejnou vazbu obsahuje i *IInput* na *IOutput*. *IInput* vlastní metodu *Receive*. Tato metoda slouží k předávání bloku dat komponentě. Jejím vstupním parametrem je pole bytů. Principiálně se v každé komponentě, která má vstupní pin, vytváří interní třída implementující rozhraní *IInput*. V této interní třídě se při implementaci metody *Receive* předává přijatý blok dat komponentě, která s ním dále manipuluje.

Obrázek 5 dále znázorňuje, že rozhraní *IPin* je rozšířeno o rozhraní *IServiceProvider* [15]. Toto rozhraní má pro celkovou univerzálnost aplikace velký význam. Rozhraní obsahuje pouze jednu metodu, konkrétně *GetService*, jejímž parametrem je typ, který je při volání metody vyžadován. *IServiceProvider* definuje mechanismus sloužící k návratu tzv. servisních objektů. Tyto objekty slouží k poskytování specifické podpory dalším objektům. Komponenty jsou principiálně navržené tak, aby si mezi sebou posílaly data ve formě pole bytů, jelikož se jedná asi o nejuniverzálnější možný způsob. Může ovšem nastat situace, kdy bude potřeba vytvořit komponentu, přes jejíž konektor bude například posílána instance nějaké třídy. Jejím obvyklým protějškem může být nová komponenta, která takový vstup očekává. V tomto případě by *IServiceProvider* nebylo nutné použít (ačkoliv jeho použití není na škodu). Jeho potenciál nastává v případě, že by se bylo nutné připojovat k nějaké komponentě, která již vlastní vstup typu *IInput* s metodou *Receive* s parametrem pole bytů. V tomto případě by stačilo definovat třídu, jejíž instance přebere referenci na konkrétní implementaci konektoru a zároveň implementuje vstupní rozhraní s metodou *Receive* s parametrem požadované instance. Třídě by poté stačilo, aby obsahovala mechanismus, kterým převede data instance po jejím přijetí metodou *Receive* na pole bytů, které přepošle implementaci *IInput*, na kterou si drží referenci. Tato třída poté slouží jako konverzní nástroj mezi

různými rozhraními pinů. Díky této funkcionalitě je zajištěn čistší a především více znovupoužitelný kód. Znovupoužitelnost je dána tím, že takovou konverzní třídu je možné použít ve více dalších komponentách, do kterých se možnost konverze přidá velice snadno pouze pomocí modifikace metody *GetService*. Pomocí těchto vlastností nejsou konektory nucené používat pouze univerzální pole bytů, ale je možné využití i složitějších datových typů či objektů. Další výhodou je, že komponenty nemusí všechny jejich služby implementovat jako rozhraní, ale mohou je pouze poskytovat jako služby.



Obrázek 6: UML diagram rozhraní komponenty

Obrázek 6 demonstruje následné použití konektorů v komponentě. Jedná se o vlastnosti *Inputs* a *Outputs*. Protože každá komponenta může mít libovolný počet vstupních a výstupních pinů, jsou definovány jako kolekce, konkrétně jako *ObservableCollection*. Důležitou vlastností je generičnost tohoto typu. Tento typ kolekce byl navržen také z důvodu vlastnictví událostí, které umožňují reagovat na změny kolekce – přidání a odebrání prvku. Tato funkcionalita je následně v samotné aplikaci využívaná a umožňuje lepší oddělení grafické části aplikace od modelu, se kterým za běhu může být manipulováno odjinud, například pomocí nějakého protokolu.

IDataComponent je hlavním rozhraním komponent a tvoří jejich nejvyšší úroveň abstrakce. Při návrhu tohoto rozhraní se dbalo na pokrytí nejnutnějších funkcionalit komponenty a zároveň na minimální soubor vlastností. Čím méně povinných vlastností, které musí vývojář komponenty implementovat, tím lépe.

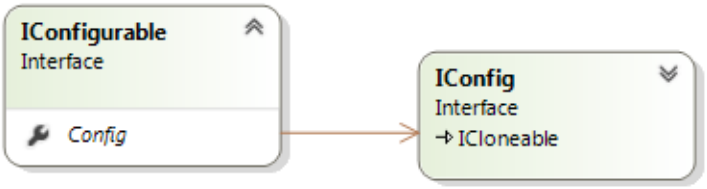
Každá komponenta implementuje vlastnost *Active*, která je typu *boolean*. Ta značí aktuální stav komponenty. Konkrétně zda komponenta běží, nebo je neaktivní. Pomocí této vlastnosti (jejího „*setteru*“) se komponenta spouští a zastavuje. Každá komponenta obsahuje textový řetězec *Name*. Ten pro ostatní uživatele vystihuje funckionalitu komponenty. Ve vývoji nad jedním repositářem by se mohlo stát, že by dva vývojáři vytvořili různé komponenty se stejným jménem a tím by vznikl problém, protože komponenty by byly v rámci aplikace nerozlišitelné a celý systém by dokázal pracovat jen s jednou z nich. Z tohoto důvodu komponenta vlastní globální unikátní identifikátor *Guid*, který je datového typu *Guid*. Tento identifikátor je tvořen hexadecimálními skupinami oddělených spojovníkem, například: *45AD1307-21AA-4e0b-863C-B28EAE04337E*. Pomocí tohoto identifikátoru při náhodné shodě jmen více komponent nenastane žádný problém. Komponenta dále implementuje vlastnost *Text*, která je typu textový řetězec. Tato vlastnost může být definována konstantně, případně může vyjadřovat stavové informace komponenty. V případě síťové komponenty může například obsahovat formátovaný řetězec s její definovanou IP adresou a portem. Její použití je opět univerzální a je na tvůrci komponenty, jak s touto vlastností naloží. Komponenta také implementuje asociační pole, kde klíč i hodnota jsou typu textového řetězce. Tento slovník je pojmenován *Additions* a slouží jako univerzální kontejner pro stavové hodnoty komponent. Například může sloužit k ukládání aktuální pozice komponenty na návrhářském plátně v grafické aplikaci.

IDataComponent je rozšířeno o několik dalších rozhraní. Jedná se rozhraní *IDisposable*, *ICloneable*, *INotifyPropertyChanged* a *IServiceProvider*. *IDisposable* je zahrnuto do datové komponenty z důvodu možnosti podchytit v každé komponentě okamžik, kdy má být zrušena – nastavena na neaktivní a předána systému *garbage collector* k odstranění z paměti. *ICloneable* vlastní metodu *Clone*, která po jejím zavolání vrací klon objektu (komponenty), na kterém je volána. Klonování komponent je důležité při samotném vytváření komponent. Rozhraní *INotifyPropertyChanged* je implementováno především z důvodu využití mechanismu *data binding* ve výsledné grafické části aplikace a také ke snadnější správě a informovanosti částí aplikace na změnu vlastností v rámci komponenty. Toto rozhraní vlastní událost *PropertyChanged*, která je vyvolána při změně určitých

vlastností. Ty musí při změně jejich hodnoty událost *PropertyChanged* vyvolat, přičemž do argumentů této události je předán název vlastnosti, která je měněna. Komponenty implementují pro jejich vyšší univerzálnost *IServiceProvider*. Důvody použití tohoto rozhraní jsou popsány výše u popisu konektorů komponent. Pro usnadnění práce vývojářům by měly být vytvořeny základní implementace (synchronní a asynchronní) rozhraní *IDataComponent*, které mohou sloužit jako předci pro konkrétní komponenty. Tyto základní implementace jsou popsány v kapitole 3.1.1.

2.1.1 Konfigurace

Při návrhu komponent bylo nutné vymyslet systém, díky kterému bude moci každá komponenta využívat své vlastní definice nastavení. Konfiguraci komponenta nemusí využívat (ačkoliv to není obvyklá situace). Konfigurace komponenty je poskytována jako servisní objekt, jinak řečeno, komponenta ho může poskytovat pomocí metody *GetService* z rozhraní *IServiceProvider*. Konfigurace komponent je možné ukládat na disk uživatele a následně je číst. Tato vlastnost vývojářům přináší výhodu v tom, že nemusí vždy při použití jedné komponenty provádět nastavení, které bývá typicky po většinu času vývoje jednoho projektu stejné.



Obrázek 7: UML diagram rozhraní pro konfiguraci

Obrázek 7 demonstruje jednoduchý systém rozhraní pro konfiguraci komponent. Komponenta při volání metody *GetService* s typem *IConfigurable* vrací null v případě, že komponenta konfigurační objekt neposkytuje, v opačném případě vrací instanci implementace *IConfigurable*. Rozhraní *IConfigurable* slouží k obalení instance typu *IConfig*. Tento obal je důležitý k dosažení možnosti komponentě konfiguraci nastavit z vnějšku. Obsahuje tedy vlastnost *Config* typu *IConfig*, která představuje již samotnou konfiguraci. Rozhraní *IConfig* je odvozeno od rozhraní *ICloneable*, takže je vývojář nucen implementovat metodu *Clone*. Toho je využíváno

při klonování komponent, kdy je nutné, aby každá komponenta vlastnila svou konfiguraci a ne jen referenci na konfiguraci jiné komponenty.

Pro konfigurace komponent bylo také nutné navrhnout rozhraní, které bude obsahovat metody pro jejich obsluhu. Konkrétně se jedná o jejich ukládání a nahrávání. Pokud by takové rozhraní neexistovalo, komponenty by musely takovouto funkcionalitu přímo implementovat, což by bylo značně neefektivní a přinášelo by to velké množství duplicitního kódu. Řešení tohoto problému je dosaženo pomocí definovaného rozhraní s názvem *IConfigManager*. Toto rozhraní obsahuje metody pro ukládání a nahrávání konfigurace, které jsou nazvány *Save* a *Load*. Obě metody mají několik přetížení, podle požadovaného použití. Jednotlivé verze metod umožňují jako parametr předat cestu ke konfiguraci, parametr samotné komponenty, nebo pouze její unikátní identifikátor *Guid*.

2.1.2 Specifické výstupy

Pro komponenty byla navržena rozhraní, která tvoří specifické výstupy. Těmito výstupy nejsou myšleny typické datové výstupy komponenty (přes konektory), ale informativní či ovládací výstupy. Ty jsou dále v aplikaci poskytovány uživateli jako ovládací prvky, které jsou distribuovány v rámci vlastního okna. Tato rozhraní nejsou při implementaci komponenty povinná a je jen na vývojáři, zda je využije. Tyto možné výstupy jsou následně poskytovány jako servisní objekty pomocí mechanismu rozhraní *IServiceProvider*.

Pro specifické výstupy byla navržena konkrétně rozhraní *ITextOutput* a *ICustomOutput*. Rozhraní *ITextOutput* je určeno pro textový výstup, jehož okno včetně ovládacího prvku bude poskytovat aplikace a bude pro všechny komponenty totožné. Obsahuje pouze metodu *WriteData*, která přebírá formou parametru pole bytů. Pomocí tohoto rozhraní lze implementovat například textový výstup informující o datovém provozu uvnitř komponenty. Pro rozhraní *ITextOutput* bylo také navrženo obalové rozhraní *ITextOutputAware*. Rozhraní *ICustomOutput* definuje pouze metodu *GetComponentControl*, která vrací vizuální prvek typu *Control*. Toto rozhraní je určeno pro komponenty, které vlastní specifický ovládací prvek, který je komponentou poskytován formou okna. Takový prvek může sloužit

jako informační, konfigurační nebo ovládací okno komponenty. Případně pomocí tohoto prvku komponenta může vizualizovat nějakou analýzu průběhu dat.

2.1.3 Logování

Tato kapitola pojednává o zaznamenávání specifických stavů (dále jen logování). V aplikaci je pro logování stavů možné využít logovacích nástrojů. Mezi stavy patří například stavy chybové, informační a případně různá upozornění. Chybové stavy by bylo možné registrovat v aplikaci pomocí odchytávání výjimek, ale upozornění a informační stavy takto ošetřit nelze. Možnost tu existuje, ale vyvolávat výjimky pro každý informační stav, jejichž množina se také může neustále měnit a rozšiřovat, není z vývojářského pohledu přípustná. Z tohoto důvodu byl navržen mechanismus, který umožňuje logování informací v rámci aplikace, jež mohou využívat komponenty. Komponenta není nucena mechanismus implementovat a je pouze na konkrétním vývojáři, zda komponentu logovacím systémem vybaví. Obdobně jako u konfigurace komponenty je možné se na existenci objektu pro logování dotázat pomocí metody *GetService*, která vrací instanci obalující logovací objekt. Navržený logovací systém dokáže pracovat se třemi stavy – informace, upozornění a omyl. Informace slouží k informačním účelům uživateli aplikace, může jí být využito například k oznámení spojení komponenty při její aktivaci. Upozornění slouží k oznámení neobvyklých stavů, které nelze vyloženě klasifikovat jako chybové. Jedná se kupříkladu o ukončení navázaného spojení druhou stranou. Stav oznamující omyl/problém je možné použít například v případě, že se nezdaří pokus o spojení s cílovou stanicí v přiřazeném časovém úseku. Mechanismus pracuje na jednoduchém principu. Komponenta, případně jiná část aplikace, vlastní obalovou instanci logovacího objektu. Pokud logovací objekt existuje (není hodnoty null), znamená to, že ho nějaká část aplikace přiřadila, respektive jeho požadovanou implementaci. Logovací objekt poté umožňuje volat metodu pro zápis stavu, konkrétně se jedná o metodu *WriteMessage*. Jejím prvním parametrem je identifikace stavu definovaného jako výčtový typ. Druhým parametrem je textový řetězec, který obvykle nese informační zprávu. Obrázek 8 demonstruje UML diagram s mechanismem rozhraní pro logování stavů.

Obrázek 8: UML diagram rozhraní pro logování

2.1.4 Nástroje komponent

Základní funkcionalita komponent je dána komponentami samotnými. Pro jejich životní cyklus z pohledu aplikace jsou ovšem nutné další obslužné rutiny, které s nimi pracují. Mezi takové nástroje patří načítání komponent, poskytování komponent aplikaci, prostor pro komponenty a manažer k propojování komponent. Všechny tyto rutiny slouží k usnadnění práce s komponentami, k zapouzdření často prováděných operací a zpřehlednění kódu. Pracují obvykle jen s komponentami (a jejich součástmi), takže nemají žádné další závislosti.

Prvním a nejzákladnějším mechanismem je načítání komponent. Rozhraní pro tento úkon se nazývá *IPluginLoader*. Toto rozhraní obsahuje pouze jednu metodu. Ta má název *LoadPlugins*, jejím vstupním parametrem je textový řetězec s cestou ke složce, ve které se nacházejí binární reprezentace komponent. Vrací seznam (konkrétně typ *List*) s genericky definovaným typem *IDataComponent*. Metoda má sloužit k tomu, aby vyfiltrovala z cílové složky binární soubory obsahující implementace rozhraní komponent. Konkrétní popis řešení nahrávání jednotlivých modulů je popsán v kapitole 3.3.

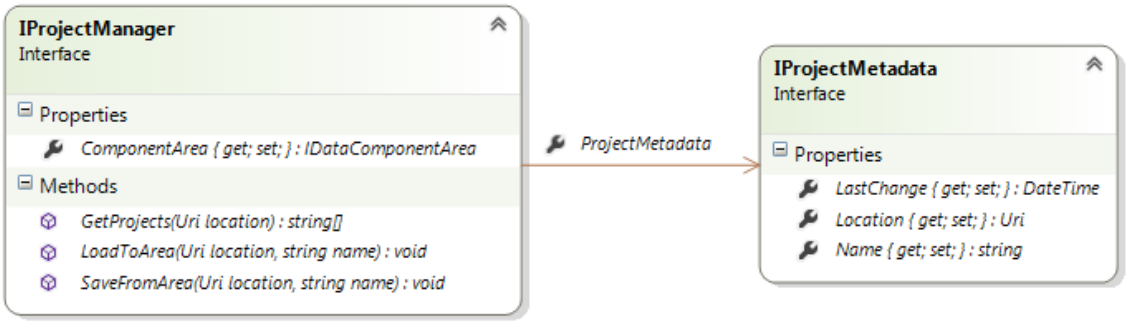
Dále bylo navrženo rozhraní, které slouží k poskytování komponent. Instance třídy, která tato rozhraní implementuje, by poté měla být schopna vytvářet instance komponent a následně je vracet. Tato třída v aplikaci funguje jako takzvaná továrna (*Factory*) pro tvorbu komponent. Rozhraní pro tuto funkcionalitu bylo pojmenováno *IDataComponentProvider*. Toto rozhraní obsahuje vlastnost *Components*, která je typu seznam (*List*) s definovaným parametrem typu *IDataComponent*. V rámci aplikace se typicky jedná o výsledek volání metody

LoadPlugins z rozhraní *IPluginLoader*. Rozhraní dále obsahuje přetíženou metodu *CreateComponent*, která v obou případech vrací danou komponentu (instanci potomka rozhraní *IDataComponent*). Prvním parametrem metody je identifikátor komponenty *Guid*, v případě přetížené formy se jedná přímo o typ komponenty. Druhým parametrem je hodnota typu *boolean*, která definuje, zda se má k vytvářené komponentě připojit/nahrát její konfigurace. Ta se obvykle nahrává ze souboru, ale není to podmínkou a je možné vytvořit odlišnou implementaci. Poté se může konfigurace nahrávat například z databáze či přes nějaký síťový protokol.

Komponenty je vhodné uchovávat (z pohledu aplikace) na nějakém konkrétním místě, ze kterého bude možné provádět vybrané operace nad všemi komponentami. Z tohoto důvodu vznikla třída *DataComponentCollection*, která je potomkem kolekce *ObservableCollection*. Ta umožňuje reagovat na změnu kolekce při přidání nebo odebraní komponenty. Jejím hlavním účelem je, že přetěžuje metody pro odebrání položky z kolekce a smazání jejího obsahu. Při přetížení těchto metod je přidána funkcionalita, která odebíraným komponentám volá metodu *Dispose*, která ukončí akce komponenty a její životnost v paměti. Pro tuto třídu byly při návrhu také definovány rozšiřující metody [12], které umožňují spuštění a zastavení komponent uložených v této kolekci. Třídy *DataComponentCollection* je následně využito u vlastnosti *Components* v rozhraní *IDataComponentArea*, které definuje samostatný prostor pro aktuálně používané komponenty.

2.2 Projekty

Pro lepší obsluhu aplikace byly navrženy rutiny, které umožňují uživateli pracovat s projekty. Projekt je z pohledu aplikace soubor vlastností, které charakterizují konkrétní sestavení komponent s jejich konfigurací a propojením. Projekt jako takový může obsahovat také další informace, jako svůj název, datum a čas poslední změny a další detaily. Projekty by mělo být v základním sestavení aplikace možné ukládat na lokální úložiště uživatele, ale rozhraní by měla být univerzální pro dovolení případné změny a odlišné implementace. Pro lokální ukládání projektů bylo třeba vymyslet systém a formát uložení, který je univerzální pro všechny projekty a zároveň čitelný pro vývojáře.



Obrázek 9: UML diagram správce projektů

Obrázek 9 zobrazuje rozhraní *IProjectManager*, které definuje vlastnosti a metody pro používání aktuálního projektu. Z povinných vlastností tohoto rozhraní vyplývá, že pracuje nad prostorem komponent (*ComponentArea*), ve kterém jsou obsaženy všechny momentálně používané komponenty. Další vlastností je vlastnost *ProjectMetadata*, která představuje implementaci rozhraní *IProjectMetada*, které je na UML diagramu též zobrazeno. *IProjectManager* obsahuje také metody pro samotné načtení a uložení projektu. Tyto metody jsou pojmenovány *LoadToArea* a *SaveFromArea*. Jména jsou volena cíleně, aby bylo zřejmé, že pracují nad aplikačním prostorem komponent. Obě tyto metody nic nevrací a přebírají parametry s cestou umístění a názvem projektu. *IProjectManager* dále obsahuje metodu *GetProjects*, která vrací pole textových řetězců, které reprezentují názvy projektů v konkrétním umístění. Umístění je předáváno, jako u metod pro uložení a načtení, pomocí typu *Uri*. Rozhraní *IProjectMetada* definuje vlastnosti jako jméno projektu, jeho umístění a datum poslední změny. Konkrétní implementace může definovat další vhodné vlastnosti, které by byly v rámci projektu vhodné a přínosné.

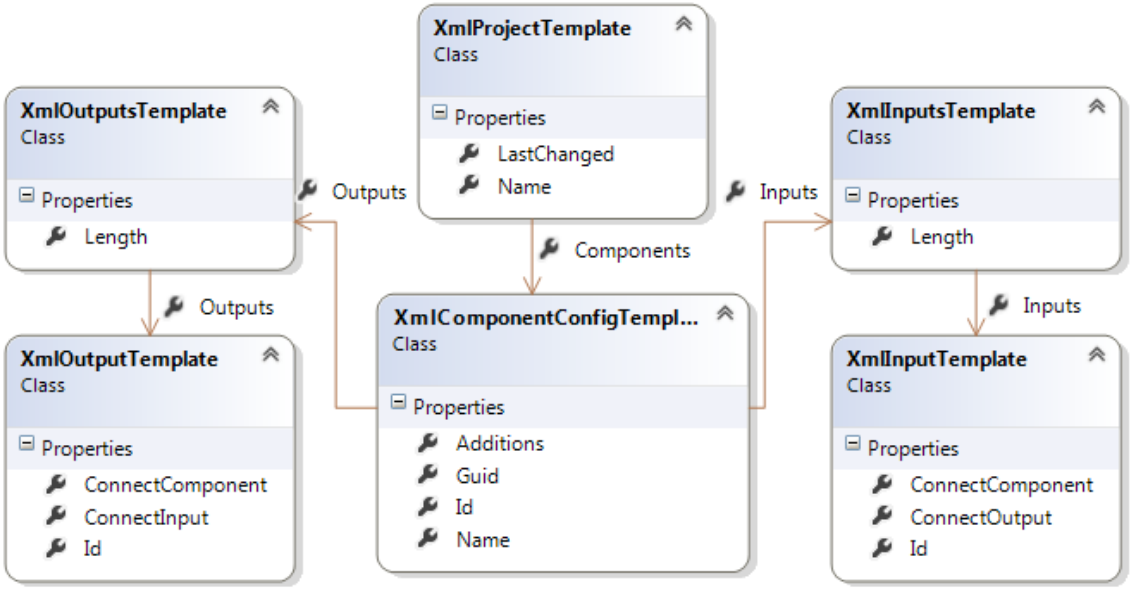
Při návrhu správce projektů bylo nutné definovat formát, v jakém bude projekt v úložišti ukládán. Zde bylo pro čitelnost využito jazyka XML. Využití XML je ovšem konkrétní implementací rozhraní *IProjectManager*, kterou je možno v budoucnu modifikovat, případně vytvořit novou. Ukládání projektu bylo navrženo tak, že komponenty jsou při ukládání vzestupně očíslovány. V umístění projektu je vytvořena složka s názvem projektu a sufixem *Conf*, ve které jsou umístěny soubory s číselným označením, které obsahují konfigurace jednotlivých komponent. Díky tomuto mechanismu je zajištěno načtení správné konfigurace komponenty. Ve složce umístění projektu se nachází soubor s názvem projektu a specifickou

příponou – *cspc* (CommSpy Project Configuration). Tento soubor je ve zmíněném XML formátu, konkrétní XSD schéma dokumentu se nachází na přiloženém CD k této práci. Výpis 1 zobrazuje zjednodušenou ukázku XML konfigurace projektu.

```
<?xml version="1.0" encoding="utf-8"?>
<XmlProject xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Components>
    <XmlComponentConfig>
      <Additions />
      <Guid>2daf420a-d05e-46d7-8c17-14058d59e8cd</Guid>
      <Id>0</Id>
      <Inputs i:nil="true" />
      <Name>File reader</Name>
      <Outputs>
        <Length>1</Length>
        <Outputs>
          <XmlOutput>
            <ConnectComponent>-1</ConnectComponent>
            <ConnectInput>-1</ConnectInput>
            <Id>0</Id>
          </XmlOutput>
        </Outputs>
      </Outputs>
    </XmlComponentConfig>
  </Components>
  <LastChanged>2014-02-28T22:48:44.6321986+01:00</LastChanged>
  <Name>project name</Name>
</XmlProject>
```

Výpis 1: Ukázka XML konfigurace projektu

V ukázce se nachází pouze jedna komponenta. Z ukázky je patrné, že tato komponenta nemá žádné vstupní konektory. Konkrétně se jedná o komponentu umožňující čtení ze souboru. Z uzlu *XmlOutput* vyplývá, že k výstupnímu konektoru není připojena žádná komponenta. Tato vlastnost je definována pomocí hodnoty mínus jedna. Pokud by v ukázce bylo komponent více a některé z nich by byly propojené, tak by v této oblasti bylo směrování na konkrétní komponentu a její číselné vyjádření konektoru. Samotné ukládání do formy XML je prováděno pomocí serializace. Tento přístup umožňuje vyšší variabilitu přidání nových vlastností bez nutnosti měnit další části kódu. Pro ukládání dat byly navrženy a vytvořeny třídy, které tvoří vzor pro uložení. To by pro samotné ukládání nebylo nutné, ale tento přístup přináší několik výhod. Mezi ně patří například to, že programátor nemusí definovat serializační atributy přímo do struktur, které jsou využívány v kódu, ale definuje je pouze na odděleném místě – v třídách šablon. Díky tomu je samotný kód přehlednější. Další výhodou je možnost separace jen určitých cílových dat, popřípadě přidaní nějakých dodatečných informací. Obrázek 10 znázorňuje UML diagram s třídami šablon pro uložení projektu.



Obrázek 10: UML diagram tříd šablon konfigurace projektu

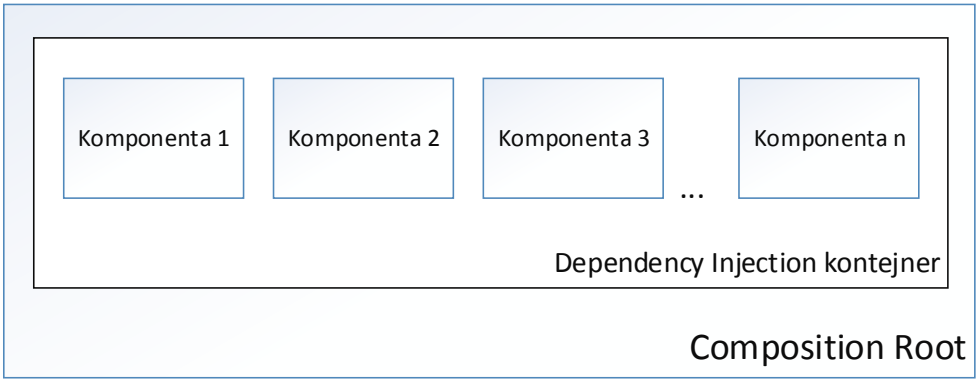
2.3 Aplikace

Tato část práce rozebírá návrh aplikace jako takové. Budou zde popsány jednotlivé části a použité návrhové vzory. Aplikaci bylo třeba navrhnout co nejuniverzálněji, aby mohlo být jednoduše použito jiné grafické rozhraní, popřípadě aby byla ovladatelná například z terminálu, nebo pomocí nějakého protokolu. Z toho důvodu byla při návrhu snaha, aby všechny části aplikace bylo možné používat samostatně a jejich závislosti byly minimální. Již při návrhu aplikace bylo počítáno s následnou implementací v jazyce C#. To umožnilo již při návrhu počítat se specializovanými technikami pro vývoj softwaru. Aplikace, včetně všech jejich částí, je kompletně programována proti rozhraní. Tato technika znamená, že je při návrhu hlavní prioritou správně navrhnout veškerá rozhraní a samotné implementace jsou vedlejší. Využití této techniky přináší především vyšší možnosti znovu-použitelnosti kódu a snižuje závislost na implementaci. To je v případě cílové aplikace jedna z důležitých vlastností, především z důvodu udržení maximální univerzálnosti. Mezi nejzákladnější a nejdůležitější princip přístupu v cílové aplikaci patří použití vzoru **Inversion Of Control** (IoC) [3]. IoC neboli obrácené řízení je poměrně moderní návrhový vzor, jehož primárním cílem je uvolnění vazeb mezi jednotlivými komponentami aplikace. Komponenty v aplikaci, která IoC nevyužívá, jsou typicky pevně svázány. To znamená, že jedna komponenta, která využívá nějakou další, ji typicky vytváří a zároveň vlastní, respektive může řídit její životnost. IoC přístup je

opačný. V IoC je snaha, aby všechny komponenty aplikace byly zřizovány samotnou aplikací a poté jedním z definovaných přístupů byly instance (případně jejich reference) podstrčeny dalším komponentám, které je vyžadují. Tímto způsobem jsou minimalizovány veškeré vazby mezi částmi aplikace. Cílová aplikace konkrétněji využívá návrhového vzoru **Dependency Injection** (DI) [7]. DI je česky často nazýván jako vkládání závislostí. Je téměř synonymem pro IoC a tyto návrhové vzory jsou často zaměňovány. Ve skutečnosti je DI pouze modernější název pro IoC a je zaměřen pro specifičtější použití. DI jako takové obsahuje množství výhod pro vývoj aplikací. Hlavní výhody, které vedly k použití Dependency Injection v cílové vyvíjené aplikaci, jsou následující:

- testovatelnost kódu,
- explicitní definice komponent (jsou definovány na jednom místě),
- minimální vzájemná závislost komponent,
- snazší údržba komponent.

Dependenci Injection může být v jistých případech i nevýhodou, ale jedná se spíše o situace, kdy je aplikace implementující DI zadána vývojáři, který tuto techniku nezná. Další možnou nevýhodou je zbytečné použití v malých aplikacích. V Dependency Injection je počítáno s využitím kontejnerů pro komponenty. Kontejnery uchovávají instance komponent/částí aplikace a mohou je poskytovat dalším částem, které je vyžadují. Tyto kontejnery se obvykle inicializují a registrují v části aplikace zvané **Composition Root** [5]. Composition Root je též návrhový vzor, který je velice úzce spjat s Dependency Injection. Definuje část aplikace, kde jsou vytvářeny a definovány vazby mezi jednotlivými částmi aplikace. V případě konzolové aplikace se typicky jedná o metodu *Main*, v případě WPF aplikace se jedná o metodu *OnStartup* třídy *Application*. Nástroje pro obsluhu DI obsahují hotové využitelné kontejnery pro komponenty, dále také umožňují takzvané injektování registrovaných komponent. Injektování je možné provádět více způsoby. Každý framework umožňuje injektování provádět jinak, nejčastěji se ovšem jedná o přiřazení komponent pomocí konstruktoru, případně pomocí nastavení veřejných vlastností. Obrázek 11 demonstruje umístění komponent v rámci Dependency Injection.



Obrázek 11: Schéma umístění komponent v rámci DI

Při návrhu bylo počítáno také s dalším návrhovým vzorem, který bude aplikace využívat. Bylo předem definováno, že aplikace bude psána v jazyce C# a bude využívat moderní uživatelské prostředí psané pomocí knihovny WPF [21]. Při vývoji takových aplikací je doporučováno společností Microsoft využití návrhového vzoru Model View ViewModel (MVVM) [32]. Jedná se o návrhový vzor, který zároveň definuje architekturu vývoje aplikace. Je obdobou dalších vzorů, jako jsou Model View Controller nebo Model View Presenter, které jsou populární a používané v jiných technologiích. MVVM je zaměřeno na použití v jazyce XAML [33] a řídí se lehce odlišnou filozofií udržování dat, na rozdíl od zmíněných vzorů MVC a MVP. ViewModel je část aplikace, která tvoří jakýsi mezičlánek mezi modelem aplikace a její grafickou částí. ViewModel poskytuje data pro grafickou část (View), ovšem podle dodržení pravidel tohoto vzoru, o grafické části absolutně nic neví, nemá na ni referenci, a tudíž nezná její definici. To představuje vyšší znovupoužitelnost kódu a umožňuje testování dat (pomocí jednotkových testů), které jsou následně poskytovány do grafické části. Ideální ViewModel obvykle pouze zveřejňuje a obaluje data z modelu, která jsou následně využívána z grafické části. Výhodou prvku ViewModel, která bude v cílové aplikaci využívána je možnost datových vazeb. Grafická část komunikuje pouze s mezičlánky ViewModel a se samotným modelem přímo vůbec nekomunikuje. Prvek View může své grafické elementy datově spojovat s veřejnými vlastnostmi ViewModel. Komunikace mezi prvkem View směrem k prvku ViewModel je zajištěna pomocí tzv. příkazů, které jsou potomky rozhraní *ICommand* [14]. ViewModel ke komunikaci s grafickými částmi využívá mechanismu událostí.

Při návrhu prvku ViewModel pro hlavní okno aplikace bylo definováno rozhraní s názvem *IMainViewModel*. Toto rozhraní je dále rozšířeno o rozhraní *INotifyPropertyChanged*, což je nepsaný standard při návrhu prvku ViewModel. Obvykle je rozhraní *INotifyPropertyChanged* nutné z důvodu použití datových vazeb s grafickou částí. *IMainViewModel* obsahuje velké množství členů a definuje velké množství příkazů (potomci *ICommand*). Mezi ně patří například příkazy k přidání či odebrání komponenty, spojení či rozpojení daných komponent, změna konfigurace komponenty, ukončení aplikace, aktivace/deaktivace komponent, uložení, vytvoření či načtení projektu a další. *IMainViewModel* také definuje vlastnosti, které musí jeho implementace poskytovat uživatelskému rozhraní. Jedná se kupříkladu o seznam komponent, titulek okna a další informace. Rozhraní dále obsahuje událost *ProjectLoaded*, která konzumentovi tohoto prvku může oznámit, že byl kompletně nahrán projekt.

Aplikace obsahuje navržené rozhraní *ICommSpyEnvironment*. Toto rozhraní nese základní informace o aplikaci a jejích vlastnostech. Všechny vlastnosti, které rozhraní definuje, jsou typu textových řetězců. Obsahuje vlastnosti definující cestu k binární reprezentaci aplikace, cestu ke složce s komponentami, umístění složky s konfiguracemi pro komponenty, výchozí cestu ke složce s projekty a příponu konfiguračních souborů projektů. Implementace tohoto rozhraní se bude využívat jako komponenty aplikace, bude tedy zahrnuta v aplikačním IoC kontejneru, takže budou moci tyto informace být snadno šířeny do dalších komponent celého systému.

3 Implementace

3.1 Sestavení aplikace

Základní funkční kód aplikace byl rozdělen do sedmi vývojových projektů .NET, které jsou v rámci jednoho řešení (Solution). Další projekty v sestavení aplikace jsou určeny pro jednotkové testy a samotné implementace komponent. Celá aplikace je tvořena více než 150 rozhraními a třídami. Tato kapitola bude seznamovat jen s nejzajímavějšími z nich. Mezi hlavní moduly aplikace patří výčet projektů definovaných v následujícím seznamu:

- CommSpy,
- CommSpy.Common,
- CommSpy.Core,
- CommSpy.PluginProvider,
- CommSpy.Project,
- CommSpy.ViewModel,
- CommSpy.View.

Projekt CommSpy je spustitelnou aplikací, ostatní projekty jsou definovány jako knihovny. CommSpy je projekt sloužící ke startu celé aplikace. Protože se jedná o grafickou aplikaci, která využívá knihovny WPF, je celá aplikace spouštěna v hlavním STA vlákně. Obsahuje pouze dvě třídy. První je pojmenována *App* a je potomkem třídy *Application*. Metoda *OnStartup* třídy *App* slouží v této aplikaci jako Composition Root. Je zde vytvořen kontejner Dependency Injection a jsou na něj registrovány jednotlivé komponenty celé aplikace a vytvářeny jejich vazby. Konkrétní ukázka sestavení a popsání implementace DI nástroje se nachází v kapitole 3.4. Metoda *OnStartup* po vytvoření a inicializaci kontejneru vytvoří instanci hlavního okna a to zobrazí. Tím je připravena celá aplikace k použití. Druhou obsaženou třídou v tomto projektu je třída *IocExtensions*, která obsahuje definici, pomocí které je možné využít automatického injektování vlastností opatřených vlastním atributem, například při registraci komponent na kontejner. Projekt CommSpy.Common obsahuje nástroje pro práci s komponentami, konkrétně se jedná o implementaci nástrojů popsaných v kapitole 2.1.4. CommSpy.Core je nejzákladnějším projektem celé aplikace, který je zároveň jediným projektem, který je nutný pro vývoj nových komponent. Obsahuje všechna

potřebná rozhraní pro jejich vývoj a další konkrétní implementace, které pomáhají šetřit čas vývojářům. Projekt CommSpy.PluginProvider obsahuje mechanismus, který umožňuje dynamické načítání binárních reprezentací komponent a jejich následné poskytování v aplikaci. CommSpy.ViewModel je projekt s rozhraními a implementacemi prvků ViewModel, konkrétně obsahuje ViewModel pro hlavní okno a univerzální textový výstup komponent, definice příkazů a další mezi-prvky. CommSpy.View obsahuje kompletní obsluhu grafického prostředí. Jedná se o definice oken, uživatelských komponent a dalších grafických elementárních prvků. Více informací o tomto projektu a jeho obsahu se nachází v kapitole 3.6.

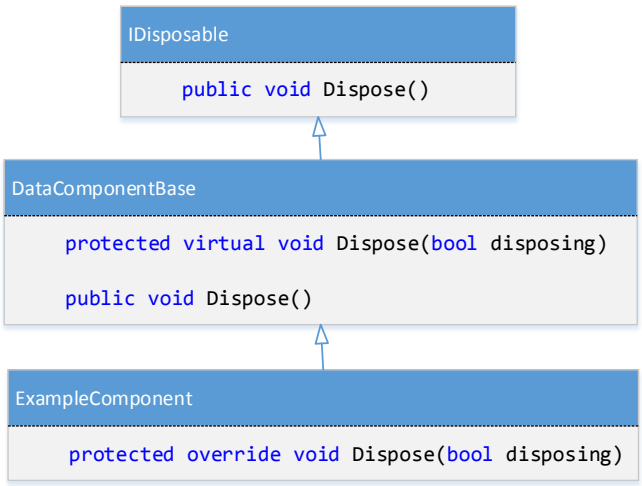
Sestavení řešení aplikace dále obsahuje dvě specifické složky sestavení, tzv. Solution Folder. Ty nesou názvy Tests a Components. Složka Test slouží k oddělení projektů obsahující jednotkové testy od zbytku aplikace. Složka Components se snaží obdobně vyseparovat implementaci jednotlivých komponent. Tyto složky nemají žádný vliv na chod aplikace, pouze zpřehledňují celkové sestavení a řešení aplikace a oddělují od sebe části, které k sobě logicky nepatří.

Samotnou aplikaci lze po kompilaci spustit pomocí souboru CommSpy.exe. Aplikace při prvním spuštění pod daným uživatelem systému Windows vytvoří složku CommSpy, která je umístěna ve složce dokumentů aktuálně přihlášeného uživatele. Tato složka slouží jako typické umístění pro ukládání projektů, posledních konfigurací komponent, nebo například logů aplikace.

3.1.1 Bázová třída DataComponentBase

DataComponentBase implementuje základní mechanismy pro práci s komponentami. Je potomkem rozhraní *IDataComponent*. Definuje metodu *GetService* z rozhraní *IServiceProvider*. Tato metoda je implementována jako virtuální, aby bylo možné z konkrétní komponenty dopsat podporu pro další servisní objekty. Tato bázová třída počítá s použitím konfigurace a logování, proto jsou obslužné objekty poskytovány přes *GetService*. Využití těchto vlastností není podmínkou. Pokud komponenta neimplementuje servisní objekt, metoda vrací hodnotu null. To aplikace chápe tak, že komponenta danou funkcionalitu nepodporuje. Výhodou je absence nutnosti implementace totožné metody *GetService* ve vícero komponentách. Bázová třída také implementuje a obsluhuje

vlastnosti z *IDataComponent* jako *Text*, *Active* a další. V definici přiřazení těchto vlastností je využito volání události *PropertyChanged*. Tyto a všechny další implementované vlastnosti jsou v rámci této třídy definovány, z důvodu univerzálnosti, jako virtuální. Třída implementuje přes rozhraní *IDataComponent* také rozhraní *IDisposable*. Bázová třída využívá při použití tohoto rozhraní návrhový vzor Dispose pattern [9]. Obrázek 12 demonstruje využití tohoto vzoru v komponentách. Rozhraní *IDisposable* definuje pouze veřejnou metodu *Dispose*. Myšlenkou využití tohoto rozhraní a vzoru Dispose Pattern je, že bázová třída, u které je očekávatelné, že bude často používána jako předek dalších komponent, bude definovat dále přetíženou variantu metody *Dispose*, která bude virtuální a viditelná pouze pro její potomky. Pokud potomek chce pozměnit nebo doplnit akce při ukončování životnosti, tak virtuální metodu přepíše, jako je demonstrováno na obrázku třídou *ExampleComponent*. Přetížená metoda *Dispose* obsahuje vstupní parametr typu *boolean*, kterým lze identifikovat, zda je ukončení existence způsobeno voláním z aplikačního kódu, nebo jej vyžaduje interní mechanismus *garbage collector*. Tato vlastnost přináší v tomto vzoru hlavní přínos a je kvůli tomu možné reagovat na ukončení komponenty odlišně, například různě nakládat se zdroji.



Obrázek 12: Demonstrace využití Dispose Pattern

Třída *DataComponentBase* také obsahuje mechanismus, který umožňuje celkové zpřehlednění implementovaných komponent. Jedná se XML zápis základních vlastností komponenty, které by se jinak typicky musely definovat například v konstruktoru třídy. To je stále možné, ale definice mnoha vlastností,

nebo delších textových řetězců působí v kódu rušivě. Z tohoto důvodu byla implementována třída *XmlComponentMetadata*, která vytváří definici pro oddělený XML zápis vlastností. Pomocí této třídy je možné určit jméno komponenty, její kategorii a její textový popis. Bázová třída *DataComponentBase* jejím potomkům umožňuje využití metody *LoadMetadaFromXml*, která načte XML data ze zdroje, jehož cesta je předána jako vstupní parametr. Tyto soubory je vhodné označit v rámci projektu typem „Embedded Resource“. Tím je zajištěno, že budou distribuovány uvnitř výsledného binárního souboru, dále je nebude možné modifikovat bez vlastnictví zdrojových kódů a následné rekompilace. Výpis 2 demonstruje formát a použití zmiňovaného zdroje vlastností komponenty.

```
<?xml version="1.0" encoding="utf-8" ?>
<XmlComponentMetadata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Component 1</Name>
  <Description>Description of component 1!Description>
  <Category>Input</Category>
</XmlComponentMetadata>
```

Výpis 2: Ukázka popisu vlastností komponenty pomocí XML

3.2 CommSpy.Core

CommSpy.Core je základním projektem celé aplikace. Obsahuje nejdůležitější rozhraní a prvky, díky kterým je možné vytvářet nové komponenty. V tomto projektu jsou také zahrnuta další rozhraní, která mohou komponenty využívat a být jimi rozšiřovány. Jedná se například o rozhraní *ILogger*, *IConfig*, *ITextOutput* a mnoho dalších, které byly popsány v kapitole věnující se návrhu. Projekt také obsahuje bázové třídy pro tvorbu komponent. Jedná se o potomky rozhraní *IDataComponent*. Bázové třídy jsou pojmenovány podle použití, jedná se o *DataComponentBase* a *AsyncDataComponentBase*. Vývojář komponenty není na těchto bázových implementacích závislý, nemusí je využít vůbec, ale ve většině případů tvorby komponenty bude použití těchto bázových tříd výhodné a efektivní. První z nich, *DataComponentBase*, je určena pro všechny komponenty, u kterých není nutné využít asynchronního provozu, tedy mohou pracovat na vlákně, ze kterého do nich data doputují. *AsyncDataComponentBase* dědí ze třídy *DataComponentBase* a pouze ji rozšiřuje o asynchronní mechanismus. Vývojář komponenty není ovšem v tomto směru omezen a může využít neasynchronní verze

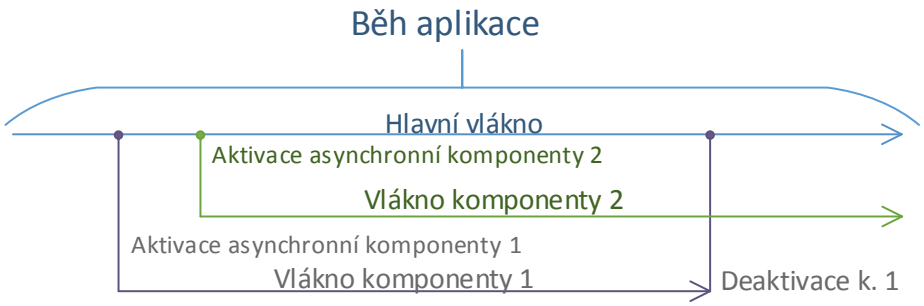
bázové třídy a v ní implementovat vlastní specifický mechanismus pro asynchronní komunikaci.

3.2.1 Bázová třída AsyncDataComponentBase

Bázová třída pro tvorbu komponent *DataComponentBase* je pro většinu implementovaných komponent vhodná a dostačují, kromě případů, kdy je po komponentě vyžadován asynchronní provoz. Z tohoto důvodu vznikla druhá bázová třída, která je ekvivalentem k *DataComponentBase*, ale je určena pro asynchronně operující komponenty. Asynchronní bázová třída je pojmenována podle jejího určení – *AsyncDataComponentBase*. Asynchronní varianta je definována jako abstraktní a obsahuje kompletní funkcionalitu jako *DataComponentBase*, je jejím potomkem. Asynchronní varianta navíc obsahuje metody viditelné pouze pro potomky, konkrétně se jedná o *Execute* a *ExecuteStep*. Metoda *Execute* je vyvolána po aktivaci komponenty a v základní implementaci volá ve smyčce metodu *ExecuteStep*. Metoda *Execute* je definována jako virtuální, takže je možné ji v případě nutnosti přepsat. Pokud je komponenta následně deaktivována, tak je pomocí objektu *AutoResetEvent* [2] spuštěn časový interval o délce jedné vteřiny, kdy musí komponenta dokončit svůj provoz. Pokud nastane interní chyba komponenty a nedokáže v daném časovém intervalu ukončit svůj provoz, tak bude vyvolána výjimka *TimeoutException*, kterou může aplikace odchytit a náležitě na ni reagovat. Celý tento provoz je možné kontrolovat z hlavního vlákna aplikace, na novém vlákně je spouštěna až metoda *Execute*.

Z důvodu obsluhy vláken komponent bylo vytvořeno rozhraní *IComponentSyncContext*, které definuje metody *Send*, *Post* a *Abort*. Metody *Send* a *Post* přebírají formou parametru reference na metody, které se mají vykonat na novém vlákně. *Send* zpracovává delegátskou metodu synchronně, zatímco *Post* je určen k asynchronnímu použití. Název rozhraní a definované metody mohou připomínat třídu *SynchronizationContext* z frameworku .NET. Tato třída byla dokonce nejprve využívána pro tyto účely, ale nakonec byla nahrazena vlastním řešením. Důvodem bylo, že třída *SynchronizationContext* u metody *Send* nespouští zpracování v samostatném vlákně. Metoda *Post* sice ano, ale pouze vyzvedává vlákno z thread poolu. Z důvodu vyšší úrovně využití vláken v aplikaci bylo předem počítáno s tím, že bude využita fronta zpráv, konkrétně za pomoci instance typu

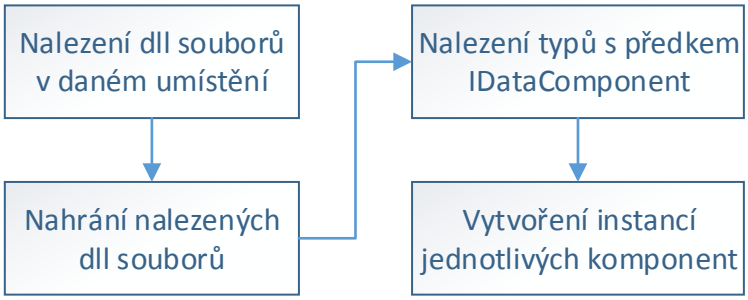
Dispatcher [8]. Zde by byla další nevýhoda v použití *SynchronizationContext*, i pro případ využití jako předka, protože metody této třídy jsou typu *void*, takže není možné vracet žádné stavové objekty nesoucí informace o prováděné operaci. Základní využívaná implementace rozhraní *IComponentSyncContext* v aplikaci je pojmenována *DataComponentSyncContext*. Tato třída při inicializaci nového objektu vytvoří nové vlákno a *Dispatcher*, kterému toto vlákno předá. Tím je vytvořen objekt pro synchronizaci kontextu a může být aplikací využíván. Tato instance je registrována na kontejner IoC. V okamžiku, kdy implementace rozhraní *IDataComponentProvider* vytváří novou komponentu, nechá ji zpracovat DI kontejnerem. V případě, že daná komponenta obsahuje atributovanou vlastnost typu *IDataSyncContextAware*, je kontejnerem vytvořena nová instance *DataSyncContextAware*, která je následně přiřazena do konkrétní vlastnosti komponenty. *IDataSyncContextAware* je rozraní, které v sobě nese pouze vlastnost s rozhraním *IComponentSyncContext*. Jeho využití je pouze obalové, aby při dotázání na tuto funkcionalitu přes metodu *GetService* bylo možné komponentě synchronizační kontext přenastavit. Obrázek 13 demonstruje běh hlavního vlákna aplikace a spuštění dvou asynchronních komponent využívajících bázová třídy *AsyncDataComponentBase*. Demonstrace simuluje časový průběh z pohledu vláken v aplikaci. Dále ukazuje, že vlákno vlastněné komponentou je ukončeno v momentě, kdy je komponenta deaktivována. Pokud deaktivována není, může vlákno pracovat až do ukončení aplikace. Ukončení aplikace je odchyceno a v tomto stavu se všechna vlákna komponent ukončují, respektive se komponenty nastavují jako neaktivní, čímž se uzavřou používané datové proudy, soubory, případně další využívané zdroje.



Obrázek 13: Demonstrace běhu asynchronních komponent

3.3 Nahrávání modulů

Jedním z cílů aplikace je modulárnost. Jednotlivé komponenty je nutné dynamicky načítat z daného umístění. Tato problematika lze řešit pomocí několika variant. Mezi tyto varianty patří například využití frameworku MEF [18] nebo lze tento úkon realizovat přímo pomocí systémových prostředků. Řešení bylo navrženo a implementováno na prostředcích, které jsou poskytovány přímo v prostředí .NET. Z toho vyplývá, že aplikace nemusí být zatěžována další knihovnou třetí strany. Dynamické nahrávání modulů je ve výsledné aplikaci CommSpy řešeno v projektu CommSpy.PluginProvider.



Obrázek 14: Postup dynamického nahrávání modulů

Obrázek 14 demonstruje obecný postup, jak jsou dynamicky moduly v metodě *LoadPlugins* třídy *PluginLoader* nahrávány. Nejprve je prohledáno cílové umístění a jsou z něj vyseparovány všechny knihovní soubory, které mají příponu dll. Ty jsou následně nahrány a jsou iterovány všechny jejich typy. Při iteraci typů se testuje, zda se jedná o potomka rozhraní pro komponenty – *IDataComponent*. Z typů, které jsou konkrétními komponentami, se vytvoří jejich instance. Ty jsou následně poskytovány jako prvky seznamu návratové hodnoty metody *LoadPlugins*.

V případě třídy *PluginLoader* se jedná o konkrétní implementaci, kterou je možné v budoucnu přepsat. V takovém případě by stačilo změnit při registraci volenou implementaci v Composition Root. Vhodným rozšířením/změnou implementace by bylo dynamické nahrávání za běhu aplikace. Tím by bylo možné přidávat moduly i za běhu aplikace s tím, že by se automaticky nahrály.

3.4 Dependency Injection framework

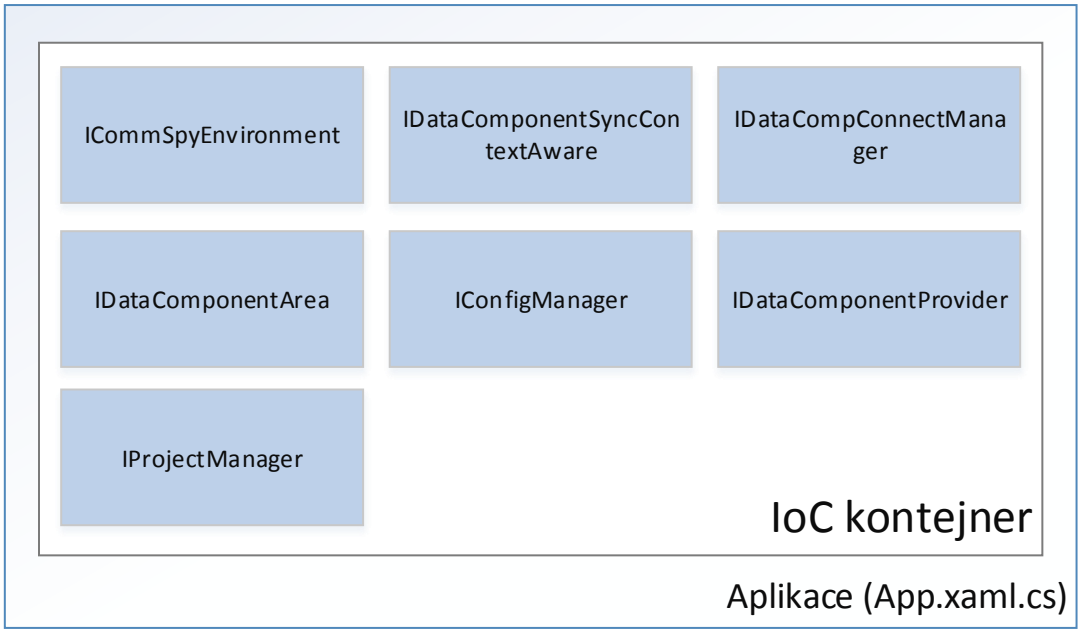
V kapitole 2.3, která se věnovala návrhu aplikace, byly zmíněny a popsány důvody, proč aplikace bude postavena na návrhovém vzoru Inversion Of Control, respektive Dependency Injection. Pro zmírnění vazeb mezi částmi aplikace při použití návrhového vzoru Dependency Injection je potřeba speciálního kontejneru, který umí tyto části uchovávat a pracovat s nimi. Ten musí disponovat mechanismem, pomocí kterého je schopen vlastnosti injektovat. V průběhu implementace bylo rozhodnuto, že bude využito existujícího řešení, které je poskytováno pro dané účely zdarma. Volně využitelného softwaru pro tyto úkoly existuje množství. Mezi nejznámější se řadí například nástroje Unity [30], Spring.NET [25], či SimpleInjector [24]. Z mnoha kandidátů byl nakonec pro nasazení v cílové aplikaci vybrán právě SimpleInjector. Důvodem byla především lehkost frameworku, dostačující funkcionalita pro cílové úkoly a snadnost použití. SimpleInjector je šířen pod velice volnou licencí MIT [27]. Tato licence umožňuje použití zcela zdarma a to, za splnění určitých podmínek, i v komerčně využívaném softwaru.

Pro označování vlastností, které mají být injektovány IoC kontejnerem, byl v aplikaci vytvořen atribut. Tento atribut byl pojmenován *InjectableProperty*. Je umístěn v základním projektu celé aplikace – CommSpy.Core. Vytvoření atributu v jazyce C# lze provést pomocí pouhého odvození od třídy *Attribute*, která je součástí .NET. Atribut je omezen pouze na použití u vlastností, jinde totiž postrádá význam. V projektu CommSpy byla dále vytvořena třída *IocExtensions*, která pro IoC kontejner definuje rozšiřující metody, které umožňují injektování atributem označených vlastností. IoC kontejner nástroje SimpleInjector má název *Container*. Je definován staticky ve třídě *App* projektu CommSpy. Její metoda *OnStartup* slouží jako Composition Root. V tomto místě se registrují komponenty/části aplikace na kontejner. Při registraci je také nutné určit životnost registrovaného objektu.

```
var container = new Container();
container.Options.AutowirePropertiesWithAttribute<InjectableProperty>();
container.Register<ICommSpyEnvironment, CommSpyEnvironment>
(Lifestyle.Singleton);
container.Register<IDataComponentSyncContextAware,
DefaultComponentSyncContextAware>(Lifestyle.Transient);
```

Výpis 3: Ukázka registrace částí aplikace do kontejneru

Životnost může být buď typu singleton, popřípadě se může vytvořit nová instance komponenty při každém jejím vyžádání. Obrázek 15 demonstruje všechny registrované části aplikace v kontejneru. Při registraci dochází k případnému injektování předem registrovaných závislostí. Výpis 3 znázorňuje ukázku zdrojového kódu s registrací částí aplikace do IoC kontejneru.



Obrázek 15: Obsah IoC kontejneru

3.5 ViewModel

ViewModel je, v aplikaci orientované podle vzoru MVVM, mezičlánek mezi modelem a grafickou formou aplikace. Více o vzoru MVVM bylo uvedeno v kapitole 2.3. Prvky typu ViewModel jsou ve výsledné aplikaci implementovány v projektu CommSpy.ViewModel. Tento projekt obsahuje rozhraní a jejich implementace pro ViewModel hlavního okna a ViewModel pro univerzální textový výstup. Tento výstup je také oknem, které je zobrazováno jako plovoucí panel nad hlavním oknem. Rozhraní pro oba zmíněné typy ViewModel obsahují mnoho vlastností typu příkaz – potomek rozhraní *ICommand*. Tyto příkazy poskytují mechanismus, pomocí kterého je možné provádět z grafické části volání akcí prvku ViewModel. *ICommand* má kromě spouštěcí metody *Execute* také metodu *CanExecute*, která prvku View dokáže oznámit, zda je možné daný příkaz aktuálně vyvolat. Všechny použité prvky ViewModel jsou potomky rozhraní *INotifyPropertyChanged*.

Hlavní ViewModel, respektive ViewModel pro hlavní okno, obsahuje příkazy k ovládání celé aplikace. Jedná se o příkazy pro přidávání/odebírání komponent, změnu konfigurace, uložení/načítání projektů a ukončení aplikace jako takové. ViewModel hlavního okna dále obsahuje vlastnost obalující informaci, zda je projekt uložen. Tato vlastnost je vizualizována hlavním oknem, kde se projevuje v titulku okna zobrazením hvězdičky za názvem projektu.

V metodách obsahujících obsluhu jednotlivých akcí je zajištěno odchytávání a zpracování případných chybových stavů (výjimek) zahrnutím potenciálně chybového kódu do bloku try-catch. V případě zachycení výjimky je informace o ní předána konkrétní implementaci rozhraní *ILogger* jako chybový stav. Rozhraní *ILogger* je detailněji popsáno v kapitole 2.1.3. Konkrétní použitá implementace navíc využívá logovacího objektu nástroje Log4net [1]. Ten ukládá stavy aplikace do souboru *CommSpy.log* umístěném ve složce CommSpy, která se nachází ve složce dokumentů aktuálně přihlášeného uživatele. Konfigurace nástroje Log4net je uložena a definována v konfiguračním souboru aplikace *App.config*, který je možné modifikovat i po kompilaci aplikace.

3.6 Grafické prostředí

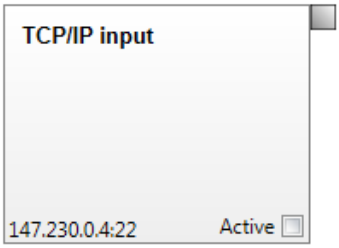
Tato kapitola se věnuje samotnému grafickému návrhu aplikace a to jak implementačním záležitostem grafické stránky aplikace jako celku, tak jednotlivým ovládacím prvkům. Grafické prostředí aplikace je kompletně naprogramováno ve WPF. WPF je nástupcem Windows Forms [31] a je totožně, jako jeho předchůdce, podmnožinou .NET frameworku. Poprvé se WPF objevilo na platformě .NET verze 3. Oproti jeho předchůdci přináší řadu výhod, vylepšení a změn. Asi nejrazantnější změnou z pohledu vývojáře je možnost návrhu grafických prvků pomocí značkovacího jazyka XAML [33]. Všechny grafické definice a třídy se nacházejí v projektu CommSpy.View.

3.6.1 Grafická podoba komponenty

Aplikace je založena na práci s komponentami. Komponenty je možné propojovat, přidávat, odebírat, číst z nich stavové informace, aktivovat je a tak dále. Pro komponenty tedy bylo třeba navrhnout a vytvořit jejich grafickou podobu. S grafickou variantou komponenty může následně uživatel na grafickém plátně

manipulovat a řídit tak její chod. Grafický prvek obstarávající podobu komponenty je potomkem třídy *UserControl*. Tato třída je součástí běhového frameworku a je určena jako základ pro nové grafické komponenty. Samotný ovládací prvek byl nazván *VisualDataComponent*. Jedná se o ovládací prvek, který navíc zobrazuje informace o komponentě. Zobrazuje konkrétně název komponenty, její aktuální hodnotu vlastnosti *Text* a aktuální hodnotu *Active*. Všechny tyto vlastnosti mají vytvořené datové vazby na skutečný objekt komponenty. Objekty komponent u svých vlastností využívají metody *PropertyChanged*. To zajišťuje, že pokud se změní nějaká vlastnost komponenty, tak se změna projeví přímo v grafické komponentě bez nutnosti nějaké další programové obsluhy. Využití datových vazeb se ve frameworku WPF, obdobně jako v jiných obdobných systémech, nazývá termínem *Data Binding*. Uživatel může komponentu aktivovat pomocí zaškrtávacího pole (*Checkbox*) datově svázaného s vlastností Active. Pomocí tohoto pole může být také uživatel informován o jejím běhu. Konkrétně se jedná o její aktivaci či deaktivaci.

Komponenta umožňuje propojení s dalšími komponentami. Z tohoto důvodu byly vytvořeny dva další ovládací prvky – *VisualPin* a *PinStack*. *VisualPin* graficky obaluje konektor komponenty. Konektorů typicky vlastní komponenta více, proto se umísťují do grafické komponenty, která slouží jako obal pro konektory, zvané *PinStack*. Každá vizuální komponenta obsahuje dva objekty *PinStack*, jeden pro vstupní konektory a jeden pro výstupní. Vizuální komponenta využívá toho, že konektory objektu komponenty jsou ukládány v kolekci typu *ObservableCollection*. Pomocí události sleduje změnu kolekce a v případě přidání, nebo odebrání pinů, vizuální konektory vytvoří nebo odebere. Obrázek 16 zobrazuje ukázku výsledné grafické komponenty včetně prvku *PinStack*.



Obrázek 16: Ukázka vizuální komponenty

3.6.2 Editor vlastností

V kapitole věnující se návrhu bylo popsáno, že každá komponenta může mít k dispozici vlastní konfiguraci. Tento problém v rámci grafické aplikace není úplně snadné řešit. Bylo by možné, aby vývojář při implementace komponenty také navrhl

a vytvořil okno, pomocí kterého by bylo možné editovat vlastnosti konfigurace. Takové řešení je ovšem značně neefektivní a zbytečně komplikované. Další možností je vytvoření specializovaného univerzálního editoru vlastností, který by vlastnosti a jejich hodnoty načítal z konfigurační třídy pomocí reflexe, ty zobrazoval, a v případě změny uživatelem aplikace by je uložil. Takové řešení je již efektivní a vývojář komponent se o takové záležitosti nemusí starat. Zmiňované řešení by bylo možné navrhnout a implementoval vlastní, ale prvek obstarávající danou funkcionalitu již existuje. Jedná se o ovládací prvek zvaný *PropertyGrid*. Ten je dostupný v balíku WPF prvků s názvem Extended WPF Toolkit [11]. Tento balík je volně dostupný pod licencí Ms-PL [19].

Prvku *PropertyGrid* je možné předložit jakýkoliv objekt. Z objektu jsou pomocí reflexe vybrány veřejné vlastnosti známých datových typů a ty jsou ve vizuální části prvku *PropertyGrid* zobrazeny. Zobrazení připomíná tabulku, jejíž první sloupec uvádí název vlastnosti a druhý je editační pole dané vlastnosti. *PropertyGrid* také umožňuje zobrazené vlastnosti filtrovat a řadit. Pro přehlednější zobrazení vlastností v prvku je možné využít atributů v definici třídy konfigurace. Pomocí atributů lze definovat zobrazený název vlastnosti, její popis a další informace. Pro složitější datové typy a objekty je možné pro *PropertyGrid* vytvářet vlastní editory, které poté lze s danou vlastností spárovat pomocí atributů.

3.6.3 Plátno aplikace

Plátno aplikace slouží k základním operacím s grafickou variantou komponent. Pro tento účel byl vytvořen potomek grafického plátna *Canvas* z WPF. Potomkem byl nazván *DesignCanvas*, protože bude sloužit jako konstrukční plátno. *DesignCanvas* je z pohledu aplikace grafickým ovládacím prvkem. Stará se o přejímání komponent při jejich přetažení z menu (Drag and Drop), jsou nad ním vytvářeny propojení komponent a umožňuje poziční manipulaci s komponentami. *DesignCanvas* je také spojen s událostí *CollectionChanged* z *DataComponentArea*. Tím je zajištěno, že dokáže přidávat a odebírat komponenty z plátna, i kdyby byly přidány/odebrány odjinud. Komponenta *DesignCanvas* využívá grafických datových komponent popsaných v kapitole 3.6.1. Ty jsou v této komponentě dále zabaleny do prvku Thumb [28]. Ten zajišťuje možnost pohybu jednotlivých komponent na plátně.

Pro lepší ovladatelnost a přehlednost uživatelského plátna byl vytvořen ovládací prvek *ZoomControl*. Tento prvek umožňuje pracovat s prvkem WPF *ScrollViewer*. *ZoomControl* umožňuje uživatelské plátno, které je prvkem *ScrollViewer*, přibližovat, oddalovat a v případě přiblížení zobrazuje malou orientační mapu pro přehlednost plátna a rychlou orientaci. Tento ovládací prvek je v aplikaci využit v plátně vpravo nahoře. Příloha B znázorňuje v ilustracích jeho vzhled. Pro možnost přibližování plátna je využito grafické transformace *ScaleTransform* [22]. Tento typ transformace je součástí frameworku WPF. I při opravdu velkém přiblížení nedochází ke ztrátě grafické kvality, protože všechny používané prvky z WPF i vlastní grafické prvky jsou vektorové. Při použití *ScaleTransform* dojde pouze k překreslení. *ZoomControl* obsahuje vlastnost závislosti (Dependency property [6]) nazvanou *ScrollViewerProperty*. Do této vlastnosti musí být po inicializaci prvku zaregistrována instance třídy *ScrollViewer*, se kterou bude následně spojena. Možnost změny pozice přímo z prvku *ZoomControl* je možná pomocí zvýrazněného elementu tvaru obdélník. Tento element má nastavenou průhlednost a znázorňuje, co je aktuálně vidět na uživatelském plátně. Lze s ním pohybovat a tím měnit pozici na uživatelském plátně. Této funkcionality je docíleno, obdobně jako u přemístění komponent, pomocí prvku *Thumb*. Výpis 4 demonstruje použití ovládacího prvku *ZoomControl* v jazyce XAML.

```
<ScrollViewer x:Name="ConstructionScrollViewer">
   <s:DesignCanvas x:Name="ConstructionCanvas" Height="2000" Width="3200"/>
</ScrollViewer>
<s:ZoomControl ScrollViewerContainer = "{x:Reference
ConstructionScrollViewer}"/>
```

Výpis 4: Použití ovládacího prvku ZoomControl

3.6.4 Nabídka komponent

Pro menu komponent byl navržen a implementován vlastní ovládací prvek. Tento prvek se skládá ze tří kategorií, podle kterých komponenty rozlišuje. Jedná se o již zmíněné kategorie vstupních, výstupních a transformačních komponent. Prvek má přístup k seznamu komponent přes hlavní ViewModel, který je nastaven jako datový kontext hlavnímu oknu a jeho potomkům. Implementovaná nabídka komponent zobrazuje vždy jen jednu kategorii. Při změně kategorie je změna doplněna krátkou animací. Animace nemá vliv na funkčnost, ale plynulý rychlý přechod působí pro uživatele více přirozeně. Animace jsou zprostředkovány pomocí

animačního mechanismu z WPF. Je využito třídy *DoubleAnimation*, která animuje hodnotu typu double mezi dvěma body pomocí lineární interpolace. Konkrétní průběh animace je zajištěn třídou *CubicEase*, která animuje průběh podle vzorce $f(t) = t^3$. Animace je provedena v časovém intervalu 400 milisekund.

3.6.5 Hlavní okno

Datovým kontextem hlavního okna a většiny jeho potomků je instance hlavního prvku ViewModel. Konkrétně se jedná o implementaci rozhraní *IMainViewModel* nazvanou *MainViewModel*. Veškerá data a vlastnosti grafické formě poskytuje právě ViewModel. Při implementaci aplikace bylo dbáno na její snadnou ovladatelnost a přizpůsobivost. Bylo využito rozšíření, které umožňuje připínání jednotlivých komponent. Pomocí tohoto rozšíření je možné si přizpůsobit grafický vzhled aplikace dle potřeb, jednotlivé aplikační komponenty připnout na různé pozice v hlavním okně, případně je od hlavního okna oddělit. To je optimální například v případě větších projektů. Pokud vývojář využívá dvou, či více, displejů, tak může části aplikace dle své potřeby rozdělit mezi oba displeje a tím docílit například přehlednější analýzy. Doplněk, který byl pro tento mechanismus v aplikaci použit, se nazývá *AvalonDock* a je dostupný z balíku grafických prvků Extended WPF Toolkit. Práce s tímto doplňkem je snadná. Výpis 5 zobrazuje XAML kód obsahující jeho základní inicializaci a vytvoření panelu s možností přichycování.

```
<xcad:DockingManager x:Name="DockManager">
   <xcad:DockingManager.Theme>
      <xcad:GenericTheme />
   </xcad:DockingManager.Theme>
      <xcad:LayoutRoot>
         <xcad:LayoutPanel Orientation="Vertical">
            <xcad:LayoutAnchorablePane DockWidth="180">
             ...
```

Výpis 5: Vytvoření panelu pomocí AvalonDock

Hlavní okno aplikace se po spuštění skládá ze čtyř hlavních částí. Jedná se o menu s komponentami, logovací panel, editor vlastností a plátno aplikace. Příloha B v ilustracích demonstruje konkrétní rozvržení. Každou část, kromě plátna, je možné skrýt a následně zobrazit pomocí menu aplikace, konkrétně přes položku View. Menu aplikace dále obsahuje položku File, která obsahuje možnosti uložení, načtení a vytvoření nového projektu a ukončení aplikace. Všechny tyto možnosti mají také vazbu na klávesové zkratky. Tyto vazby jsou vytvořeny pomocí praktik

doporučovaných pro WPF aplikace. Tyto praktiky využívají *CommandBinding* a *KeyBinding* v kontextu hlavního okna. Výpis 6 znázorňuje XAML kód s jejich použitím. Z výpisu je možné vyčíst kombinace kláves pro vyvolání dané akce.

```
<Window.CommandBindings>
    <CommandBinding Command="New" Executed="OnNewCall" />
    <CommandBinding Command="Open" Executed="OnOpenCall" />
    <CommandBinding Command="Save" Executed="OnSaveCall" />
    <CommandBinding Command="SaveAs" Executed="OnSaveAsCall" />
</Window.CommandBindings>
<Window.InputBindings>
    <KeyBinding Key="N" Modifiers="Control" Command="New"/>
    <KeyBinding Key="O" Modifiers="Control" Command="Open"/>
    <KeyBinding Key="S" Modifiers="Control" Command="Save"/>
    <KeyBinding Key="S" Modifiers="Control+Shift" Command="SaveAs"/>
</Window.InputBindings>
```

Výpis 6: Použití CommandBinding a KeyBinding

3.7 Základní kolekce komponent

Do aplikace bylo implementováno několik základních komponent, které jsou obecné a budou ve vývoji pro sportovní prostředí často používané. Jedná se především o vstupní a výstupní komponenty pro obsluhu běžně používaných komunikačních rozhraní. Dále se v základním balíku vyskytují komponenty obstarávající průběh dat, jako například rozbočovač. Všechny tyto komponenty jsou distribuovány přímo s aplikací. Jejich zdrojové kódy jsou umístěny ve složce projektu aplikace – Components.

TCP/IP Input, Output

Vstupní a výstupní komponenty pro obsluhu TCP/IP komunikace. Obě komponenty jsou klientské. Obsahují vlastní konfiguraci s možností nastavení IP adresy a portu. Pro komunikaci využívají zmíněné knihovny *RLib*. Příjem zpráv z využívaných objektů knihovny RLib je prováděn asynchronně. Pouze spojení probíhá synchronně, proto je odstíněné od hlavního vlákna pomocí třídy *Task*.

UDP Input, Output

Opět se jedná o dvě komponenty, které využívají datových kanálů knihovny RLib. Jedná se o komponenty obstarávající vstup a výstup nestavového protokolu UDP. V tomto případě je přenos plně asynchronní a je takto rovnou poskytován z použité knihovny. Komponenta vstupu má konfiguraci pouze na port, na kterém bude naslouchat. Komponenta výstupu vlastní již IP adresu a port cílové služby.

Serial Input, Output

Vstupní a výstupní komponenty pro obsluhu sériové linky. Pro datovou komunikaci je opět využito knihovny RLib. Obě komponenty využívají komponentu aplikace pro logování jejich běhu a stavů. Konfigurace se skládá z názvu portu, baudrate, parity a počtu koncových bitů.

File Reader, Writer

Komponenty zajišťující primitivní čtení a zápis souborů. Vstupní komponenta, poskytující čtení souboru po blocích dat a jejich následnou distribuci do dalších komponent, je potomkem bázové třídy *AsyncDataComponentBase*. To znamená, že jsou data dále zasílána v kontextu jiného vlákna, tedy asynchronně.

Null Output

Výstupní komponenta Null output nedisponuje žádnou speciální funkcí. Je obdobou zařízení */dev/null* používaném v unixových systémech. Všechna data, která do komponenty doputují, jsou zahozena a ignorována. Komponenta poskytuje servisní objekt pro textový výstup, takže je možné pomocí této komponenty snadno sledovat datový provoz, který do ní vstupuje.

User Input

User Input je vstupní komponenta uživatelského vstupu. Tato komponenta poskytuje servisní objekt pro vlastní okno. Z tohoto okna je poté možné zasílat data do dalších komponent. Uživatelským vstupem mohou být data v čistém textovém formátu, ve formátu se speciální mapou znaků, nebo je možné data zadávat pomocí hexadecimálních hodnot. Komponenta dále ve svém okně obsahuje vstupy pro definici maker, která mohou být nápomocna při testování specifických protokolů.

Splitter, Joiner

Komponenty Splitter a Joiner jsou obě transformační. Splitter slouží k rozdělení jednoho toku dat do více komponent. Joiner naopak umožňuje spojit více datových proudů do jednoho. Obě komponenty mají konfiguraci, ve které je možné definovat počet výstupních, respektive vstupních konektorů.

Data Logger, Data Simulator

Tyto komponenty jsou umístěny v jednom projektu, protože spolu úzce souvisí. Výstupní komponenta Data Logger umožňuje archivaci obdržených dat. Každý archivovaný blok dat nese také informaci o přesném čase, kdy ho komponenta přijala. Těchto specifických souborů využívá vstupní komponenta Data Simulator. Ta poskytuje servisní objekt vlastního výstupu. V poskytovaném okně je možné průběh datové komunikace simulovat s přesnými časovými prodlevami, nebo s uživatelsky definovaným intervalem mezi jednotlivými bloky dat. Další možností je ruční simulace, kdy jsou jednotlivé bloky odesílány na aktuální reakci uživatele (kliknutí na tlačítko). Příloha B ilustruje, mimo jiné, ukázku okna komponenty Data Simulator.

WCF Demo

WCF Demo je pouze demonstrační komponentou. Využívá WCF mechanismu k využití veřejné volně dostupné SOAP služby, která poskytuje zeměpisnou délku a šířku pro konkrétní stát (pevně definovaný ve zdrojovém kódu) ze Spojených států amerických. Komponenta po odpovědi služby přepošle přijatá data na svůj výstup. WCF Demo nemá vyvinut asynchronní mechanismus a možnost konfigurace, jedná se pouze o demonstraci využití vyšší úrovně spojení a komunikace v komponentě.

3.8 Ukázka tvorby komponenty

Při tvorbě komponenty je možné využít definovaných rozhraní, bázových tříd a dalších rozšiřujících servisních objektů. Každá komponenta musí být potomkem rozhraní *IDataComponent*. Může ho dědit přímo, nebo může být potomkem třídy, která *IDataComponent* implementuje. Implementované bázové třídy se jmenují *DataComponentBase* a *AsyncDataComponentBase*. Mezi volitelné servisní objekty patří implementace následujících rozhraní:

- ILoggerAware,
- IConfigurable,
- ITextOutputAware,
- ICustomOutput.

Datovou komponentu následně využitelnou jako plug-in v aplikaci je možné vytvořit několika způsoby. Nejzákladnějším a zároveň zdlouhavým řešením je přímá

implementace rozhraní *IDataComponent*. V tomto případě je nutné implementovat všechny vlastnosti a metody od základu. Vhodnějším řešením je využití rozšíření bázové třídy *DataComponentBase* nebo *AsyncDataComponentBase*. Tato kapitola bude demonstrovat vytvoření úplně jednoduché komponenty. Příloha A následně obsahuje podrobnější popis tvorby komponent.

Výpis 7 demonstruje vytvoření jednoduché transformační komponenty s jedním vstupem a výstupem. Činnost této komponenty zahrnuje pouze přeposlání přijatých dat ze vstupu na výstup. Komponenta je potomkem bázové třídy *DataComponentBase*. Implementuje interní třídu *Input* pro vstupní konektor. V této třídě je implementována metoda *Receive*, která přijatá data předá metodě *OnDataReceive*. Ta je již členem ukázkové implementované komponenty. V této metodě se jen data předají na vstupní konektor komponenty spojené s výstupním konektorem ukázkové komponenty. V konstruktoru komponenty je pouze přiřazen unikátní identifikátor Guid a jsou zde definované vstupní a výstupní piny. Ukázková komponenta neobsahuje žádné specifické servisní objekty určené například pro logování, konfiguraci, textový nebo vlastní výstup, ani žádné další.

```
public class Component : DataComponentBase
{
   private void OnDataReceive(byte[] data)
   {
      if(Active)
         Outputs[0].Input.Receive(data);
   }
   private class Input : ByteInputBase
   {
      private readonly Component _component;
      public Input(Component component)
      {
           component = component;
          Owner = component;
      }

      public override void Receive(byte[] data)
      {
          _component.OnDataReceive(data);
      }
   }
   public Component()
   {
      Guid = new Guid("15BD1307-21AA-4e0b-863C-B28EAE04337E");
      LoadMetadaFromXml("Metadata.xml");
      Inputs = new ObservableCollection<IInput> { new Input(this) };
      Outputs = new ObservableCollection<IOutput> { new ByteOutputBase() };
   }
}
```

Výpis 7: Vytvoření jednoduché komponenty

Závěr

Výsledkem této diplomové práce je univerzální aplikace, která byla navržena a implementována pomocí moderních technologií na platformě .NET. Aplikace se v konečné podobě skládá z více než 150 rozhraní a tříd. Jejími primárními cíli jsou přeposílání dat mezi různými rozhraními, transformace datového provozu a jeho analýza. Aplikace slouží jako vývojový prostředek pro tvorbu aplikací ve sportovním prostředí, případně i jinde, kde se pracuje s datovým provozem. Uživatel může pracovat s komponentami, které je možné umístit na pracovní plochu aplikace a dle cílových požadavků je spojit. Po aktivaci požadovaných komponent jsou na daných rozhraních inicializována spojení a propojené komponenty si mezi sebou vzájemně vyměňují data. Komponenty se dělí na tři typy – vstupní, výstupní a transformační. Vstupní a výstupní komponenty reprezentují jednotlivé vstupy či výstupy různých komunikačních rozhraní, případně alternativních vstupů/výstupů. Transformační komponenty uživateli umožňují provádět různé datové transformace, například transformace protokolů. Další možností transformačních komponent je možnost datové analýzy, kupříkladu formou grafů či tabulek. Aplikace byla implementována v prostředí .NET verze 4 a její grafická podoba je vytvořena pomocí WPF. Při vývoji bylo dále využito efektivních návrhových vzorů, jako jsou Inversion Of Control, MVVM a dalších.

Aplikace obsahuje mechanismus pro snadnou implementaci nových komponent. Toho lze docílit pomocí implementace rozhraní *IDataComponent*, nebo pomocí bázových tříd *DataComponentBase*, popřípadě *AsyncDataComponentBase* pro asynchronní komponenty. Se základní verzí aplikace je distribuován balík základních komponent. Tento balík obsahuje přibližně patnáct komponent. Jedná se především o komponenty pokrývající obecná rozhraní, jako sériovou linku, TCP/IP, UDP a další. Dále obsahuje komponenty pracující s daty, jako rozdělovač či slučovač, komponenty umožňující záznam datové komunikace a její simulaci s přesnými časovými rozestupy mezi jednotlivými bloky dat.

Rozšiřitelnost aplikace o novou funkcionalitu je zaručena především pomocí modulárního mechanismu, který umožňuje automatizované načítání komponent z předem definovaného umístění. Aplikace je kompletně programována proti

rozhraní, takže jakákoliv její část může být snadno nahrazena jinou konkrétní implementací. Pokud by se jednalo o implementaci klíčového rozhraní aplikace, postačila by změna pouze na jednom místě v aplikaci, v tzv. Composition Root. Do budoucna by bylo možné přidat možnost načtení jednotlivých základních implementací pomocí vnější konfigurace aplikace. Toho by bylo možné realizovat například pomocí nástroje Spring.NET. Při vývoji aplikace bylo dbáno na její maximální univerzálnost a následnou rozšiřitelnost. Do budoucna je plně otevřena vůči změnám a nové funkcionalitě. Za nevýhodu lze považovat omezenost běhu aplikace pouze na platformě .NET, takže ji není možné typicky spouštět na jiných platformách jako Linux, Android a další.

Seznam použité literatury

[1] Apache log4net. APACHE SOFTWARE FOUNDATION. *Apache logging services* [online]. © 2004-2013 [cit. 2014-05-03]. Dostupné z: http://logging.apache.org/log4net

[2] AutoResetEvent. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/cs-cz/library/zd6a283y(v=vs.110).aspx

[3] BURNS, Kyle. Inversion of Control. *Beginning Windows 8 Application Development: XAML Edition*. Apress, 2012. s. 165-174. ISBN 978-1430245667.

[4] Comm Tunnel Pro. *Serial Port Tool* [online]. © 2014 [cit. 2014-05-10]. Dostupné z: http://www.serialporttool.com/CommTunnelPro.htm

[5] Composition Root. SEEMANN, Mark. *Mark Seemann's blog* [online]. © 2014 [cit. 201405-03]. Dostupné z:http://blog.ploeh.dk/2011/07/28/CompositionRoot

[6] Dependency Properties Overview. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms752914(v=vs.110).aspx

[7] Design Patterns: Dependency Injection. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/magazine/cc163739.aspx

[8] Dispatcher Class. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher(v=vs.110).aspx

[9] Dispose Pattern. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx

[10] EVS BROADCAST EQUIPMENT. *EVS | Enriched. Live* [online]. © 2014 [cit. 2014-03-21]. Dostupné z: http://www.evs.com

[11] *Extended WPF Toolkit* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://wpftoolkit.codeplex.com

[12] Extension Methods (C# Programming Guide). MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/cs-cz/library/bb383977.aspx

[13] GOOK, Michael. *Hardwarová rozhraní: průvodce programátora*. Vyd. 1. Brno: Computer Press, 2006, 463 s. ISBN 80-251-1019-2.

[14] ICommand Interface. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/vstudio/system.windows.input.icommand

[15] IServiceProvider Interface. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/cc678965(v=vs.85).aspx

[16] KABELOVÁ, Alena a Libor DOSTÁLEK. *Velký průvodce protokoly TCP/IP a systémem DNS*. 5., aktualiz. vyd. Brno: Computer Press, 2008, 488 s. ISBN 978-80-251-2236-5.

[17] MACDONALD, Matthew. *Pro WPF in C#2008: Windows presentation foundation with .NET 3.5*. 2nd ed. Apress, 2008. ISBN 978-159-0599-556.

[18] *Managed Extensibility Framework* [online]. © 2014 [cit. 2014-05-08]. Dostupné z: http://mef.codeplex.com

[19] Microsoft Public License (MS-PL). *The Open Source Initiative* [online]. © 2013 [cit. 2014-05-03]. Dostupné z: http://opensource.org/licenses/MS-PL

[20] PATHAK, Nishith. *Pro WCF 4: practical Microsoft SOA implementation*. 2nd ed. New York, NY: Distributed to the book trade worldwide by Springer Science Business Media, 446 s. ISBN 978-143-0233-688.

[21] PETZOLD, Charles. *Mistrovství ve Windows Presentation Foundation*. Vyd. 1. Brno: Computer Press, 2008, 928 s. ISBN 978-80-251-2141-2.

[22] ScaleTransform Class. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library system.windows.media.scaletransform(v=vs.110).aspx

[23] Serial to Ethernet Converter Software TCP/Com. *TAL Technologies, Inc.* [online]. © 2014 [cit. 2014-05-10]. Dostupné z: http://www.taltech.com/tcpcom

[24] *Simple Injector* [online]. © 2006-2014 [cit. 2014-05-03]. Dostupné z: http://simpleinjector.codeplex.com

[25] *Spring.NET - Application Framework* [online]. © 2004-2014 [cit. 2014-03-21]. Dostupné z: http://springframework.net

[26] *TCP2COM* [online]. © 2004 [cit. 2014-05-10]. Dostupné z: http://tcp2com.sourceforge.net

[27] The MIT License (MIT). *The Open Source Initiative* [online]. © 2013 [cit. 2014-05-03]. Dostupné z: http://opensource.org/licenses/MIT

[28] Thumb Class. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/ system.windows.controls.primitives.thumb(v=vs.110).aspx

[29] TKD SCORE. *Daedo TrueScore E-Foot Gear Sparring* [online]. © 1999-2013 [cit. 2014-03-21]. Dostupné z: http://www.tkdscore.com

[30] *Unity* [online]. © 2006-2014 [cit. 2014-05-03]. Dostupné z: http://unity.codeplex.com

[31] Windows Forms. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx

[32] WPF Apps With The Model-View-ViewModel Design Pattern. MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

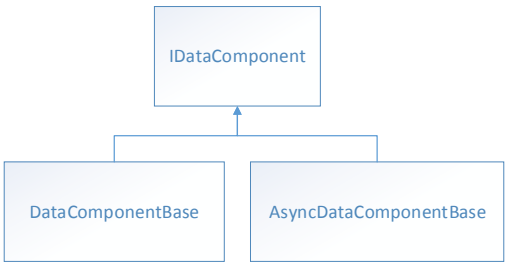
[33] XAML Overview (WPF). MICROSOFT. *MSDN-the microsoft developer network* [online]. © 2014 [cit. 2014-05-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/ms752059(v=vs.110).aspx

A Tvorba komponent

Tato příloha rozebírá možnosti vývoje jednotlivých komponent. Základní popis tvorby komponenty byl popsán v kapitole 3.8. Oproti kapitole 3.8 tento text vytváří větší přehled ohledně servisních objektů a demonstruje zdrojový kód složitější komponenty.

Předci komponent

Každá komponenta musí být předkem rozhraní *IDataComponent*. Toho je možné docílit přímou implementací tohoto rozhraní, popřípadě odvozením bázové třídy *DataComponentBase* nebo *AsyncDataComponentBase*. Další možností je vytvoření dalších bázových tříd, které budou vhodné pro použití v určité množině komponent.



Obrázek 17: Hierarchie bázových tříd komponent

Servisní objekty

Servisní objekty komponent mohou být v aktuální verzi aplikace rozděleny do tří pomyslných skupin. První skupinou je konfigurace, která v sobě nese rozhraní *IConfigurable*. Toto rozhraní poskytuje mechanismus, díky kterému mohou mít jednotlivé komponenty různé možnosti nastavení/konfigurace. Více o tomto rozhraní se nachází v kapitole 2.1.1. Za druhou skupinu lze považovat logování dat. Tato skupina disponuje rozhraním *ILoggerAware*. To obaluje implementaci rozhraní *ILogger*, takže je z aplikace možné komponentně implementaci *ILogger* podstrčit. Poslední skupina zaobaluje alternativní výstupy dat. Do nich lze zařadit univerzální textový výstup a vlastní textový výstup. Rozhraní *ITextOutputAware* poskytuje komponentě možnost posílat textová data na konkrétní implementaci *ITextOutput*. Rozhraní *ICustomOutput* umožňuje komponentě poskytovat vlastní okno, které může obsahovat například ovládací prvky, nebo grafy znázorňující nějakou datovou

analýzu. Vývojář musí při použití *ICustomOutput* vytvořit vizuální ovládací prvek typu *Control* a ten vracet v implementaci metody *GetComponentControl*.

Obrázek 18: Skupiny servisních objektů komponent

Ukázka komponenty

Tato ukázka demonstruje zdrojové kódy komponenty obstarávající UDP vstup. Komponenta využívá rozšíření bázové třídy *DataComponentBase*, takže se není ve vývoji komponenty třeba věnovat obecným implementačním záležitostem. Tato komponenta využívá vlastní konfiguraci a textový výstup, tudíž implementuje rozhraní *ITextOutputAware*. Nejprve bude uveden kód s třídou definující konfiguraci komponenty:

```
public class UdpInputConfig : IConfig
{
        private int _port = 9999;

        [Category("Basic")]
        [DisplayName("Port")]
        public int Port
        {
            get { return _port; }
            set { _port = value; }
        }

        public object Clone()
        {
            var clone = new UdpInputConfig();
            clone.Port = Port;
            return clone;
        }
}
```

Výpis 8: Ukázka konfigurační třídy komponenty

Při tvorbě definice konfigurace pro komponentu je nutné, aby vytvářená třída implementovala rozhraní *IConfig*. Toto rozhraní je rozšířeno také rozhraním *ICloneable*. Název, popis a kategorie komponenty jsou pro přehlednost definovány v XML souboru (v tomto případě InputMetadaData.xml). Toho je docíleno pomocí mechanismu zahrnutém v *DataComponentBase*. Výpis 9 znázorňuje použití pro komponentu UDP.

```
<XmlComponentMetadata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>UDP input</Name>
  <Description>UDP input enables to receive UDP packets and distribute data to other
components.</Description>
  <Category>Input</Category>
</XmlComponentMetadata>
```

Výpis 9: XML konfigurace vlastností komponenty

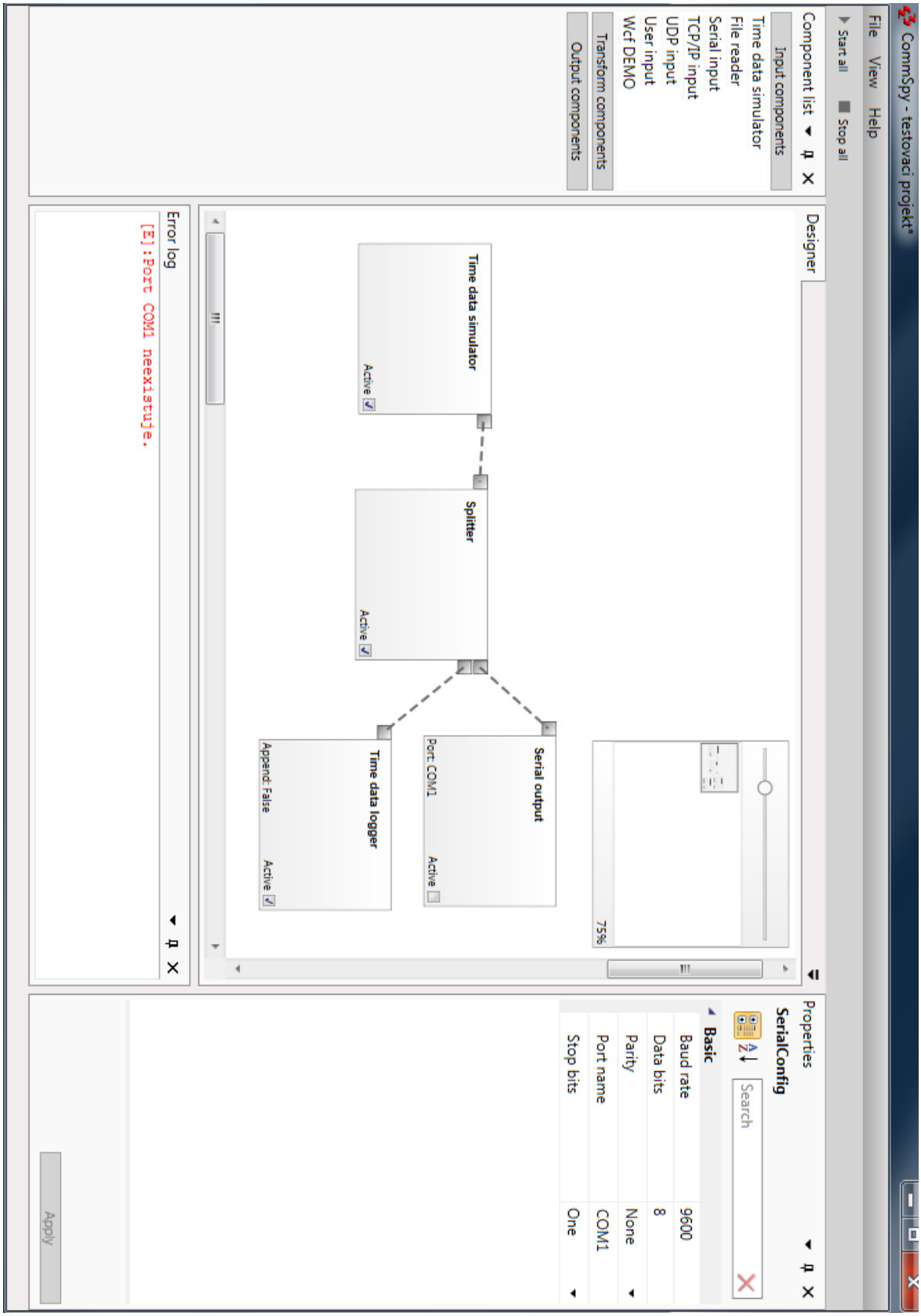
Výpis 10 demonstruje již kompletní implementaci komponenty s použitím bázové třídy *DataComponentBase* a rozhraní *ITextOutputAware*.

```
public class UdpInput : DataComponentBase, ITextOutputAware
{
        private UdpInputConfig _config = new UdpInputConfig();
        private UdpDataChannel _dataChannel;

        protected override IConfig GetConfig()
        {
            return _config;
        }

        private void InitIO()
        {
            Outputs = new ObservableCollection<IOutput>();
            Outputs.Add(new ByteOutputBase() { Owner = this });
        }

        public UdpInput()
        {
            Guid = new Guid("2A83DE07-CB3B-4B59-8B9D-B868A40DAC96");
            LoadMetadaFromXml("InputMetadata.xml");
            InitIO();
        }

        public override bool Active
        {
            get { return base.Active; }
            set
            {
                base.Active = value;
                if (value)
                {
                    _dataChannel = new UdpDataChannel(_config.Port);
                    _dataChannel.DataReceived += OnDataReceived;
                    _dataChannel.Open();
                }
                else
                {
                    _dataChannel.DataReceived -= OnDataReceived;
                    _dataChannel.Close();
                    _dataChannel = null;
                }
            }
        }

        private void OnDataReceived(object sender, DataReceivedEventArgs e)
        {
            if (Active && Outputs[0].Input != null)
            {
                Outputs[0].Input.Receive(e.Buffer.Array.Take(e.Buffer.Count).ToArray());
                if(TextOutput != null)
                    TextOutput.WriteData(e.Buffer.Array.Take(e.Buffer.Count).ToArray());
            }
        }

        protected override void RefreshText()
        {
            Text = String.Format("Port: {0}", _config.Port);
        }

        protected override void OnConfigChanged(IConfig config)
        {
            _config = (UdpInputConfig)config;
            RefreshText();
            if (Active)
                Active = false;
```

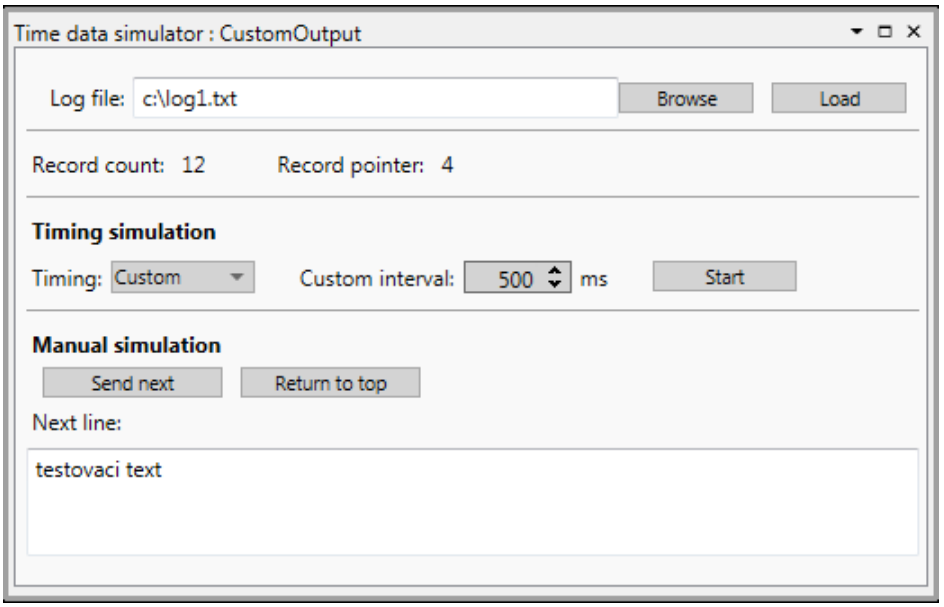
```
        }

        protected override void Dispose(bool disposing)
        {
            if (_dataChannel != null)
            {
                _dataChannel.DataReceived -= OnDataReceived;
                _dataChannel.Close();
            }
            base.Dispose(disposing);
        }

        public ITextOutput TextOutput { get; set; }
}
```

Výpis 10: Ukázka implementace komponenty

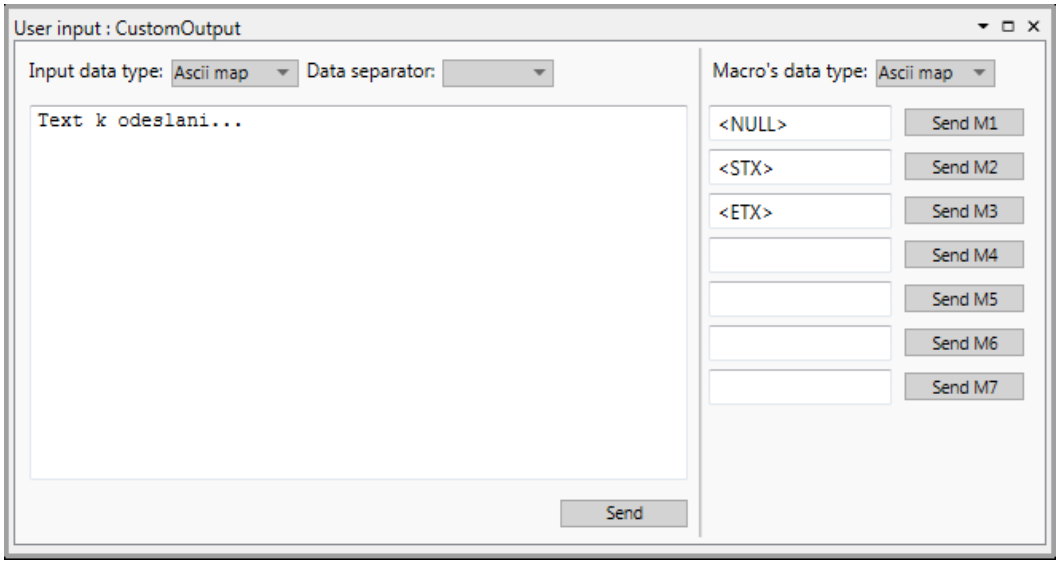
B Vizuální prvky aplikace



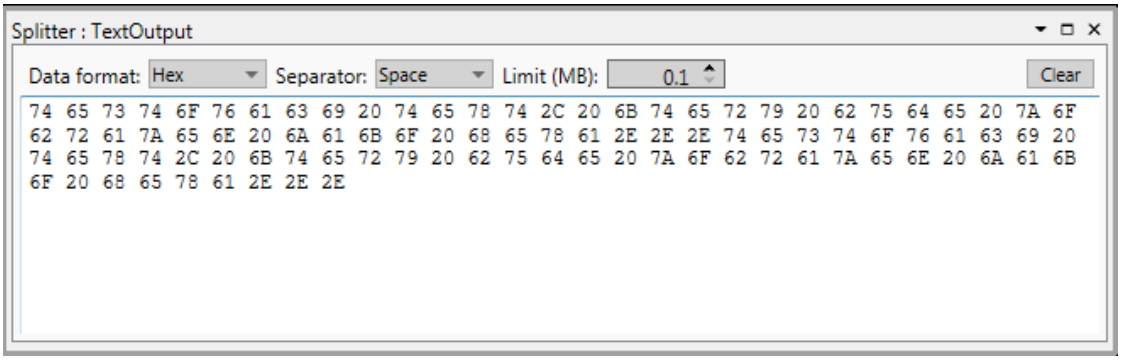
- Hlavní okno aplikace.



- Ukázka vlastního okna komponenty Data Simulator.

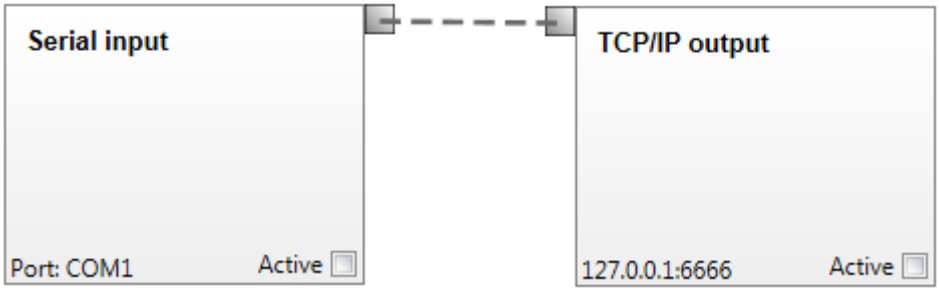


- Vlastní okno komponenty User Input.

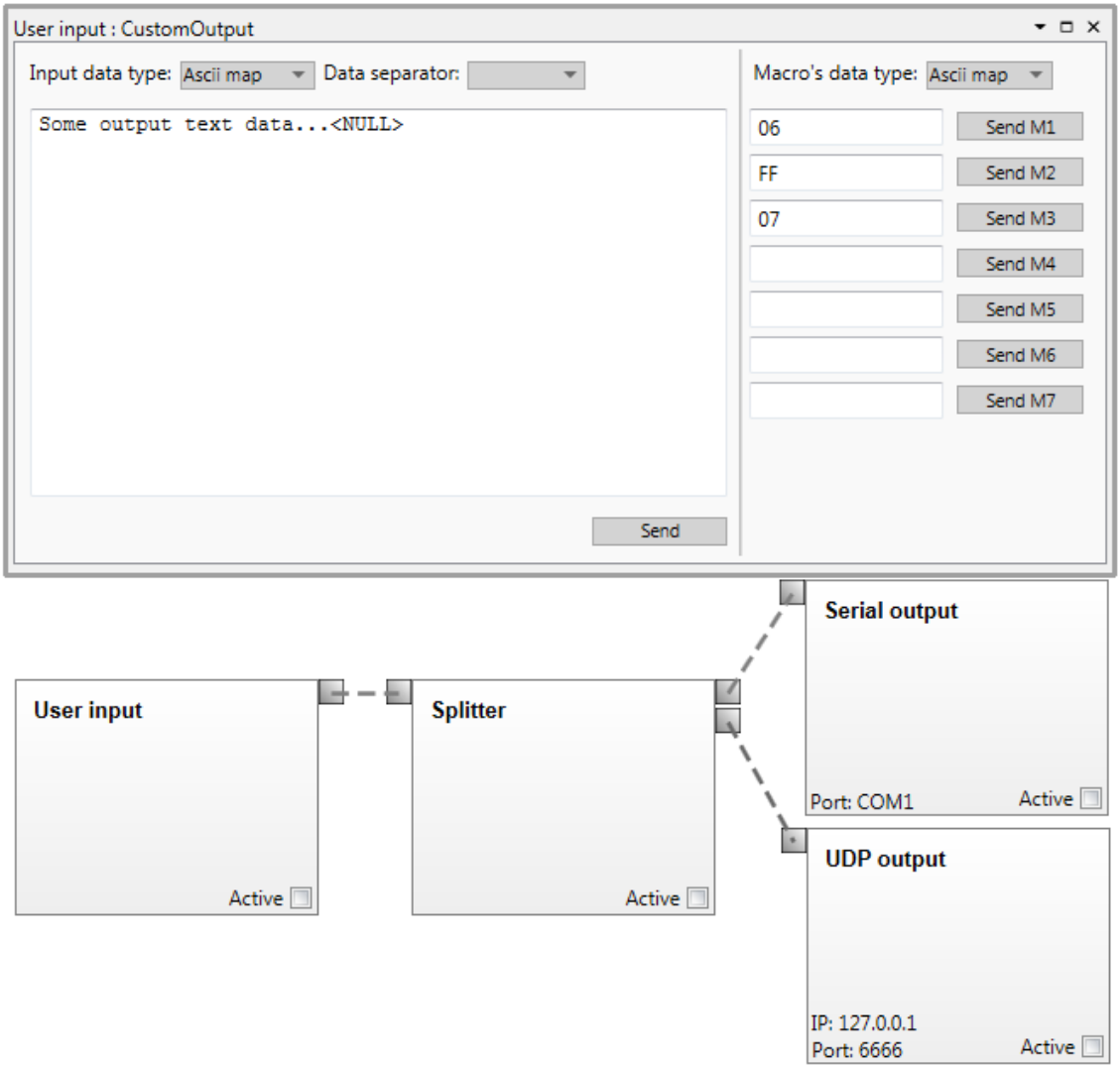


- Základní implementace obecného okna textového výstupu.

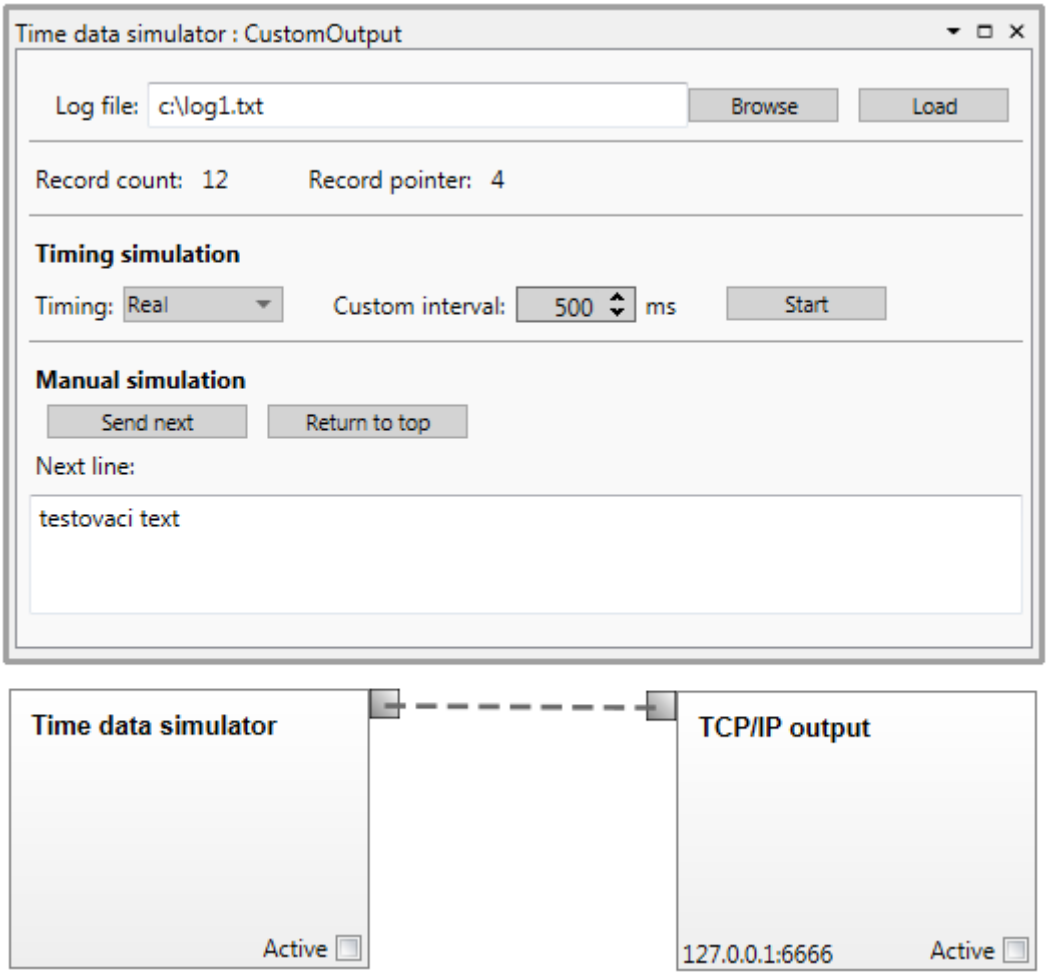
C Příklady použití



- Přesměrování datového vstupu sériové linky na TCP/IP výstup.



- Využití uživatelského vstupu k odesílání dat na výstup sériové linky a UDP.



- Simulace archivované datové komunikace odesílané na TCP/IP výstup.

- Transformace dat z protokolu TDK na protokol SDP na TCP/IP kanálu.

D Obsah přiloženého CD

- text diplomové práce
  - diplomova_prace_2014_Pavel_Novak.pdf
  - diplomova_prace_2014_Pavel_Novak.docx
  - kopie_zadani_diplomove_prace_2014_Pavel_Novak.pdf
- zdrojové kódy výsledné aplikace
  - kódy aplikace CommSpy v C# a XAML (umístění: */CommSpy/src*)
- binární reprezentace výsledné aplikace
  - binární data a spustitelná aplikace (umístění: */CommSpy/bin*)
- knihovny/závislosti výsledné aplikace
  - binární data a spustitelná aplikace (umístění: */CommSpy/comp*)
- ostatní kódy
  - XSD konfigurace projektu (umístění: */CommSpy/other/project_conf.xsd*)