VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Aplikace pro automatizaci postupů vývoje průmyslových strojů

Application for Automation of Industrial Machine Development Procedures

2019 Petr Dvořáček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Zadání bakalářské práce

Student: **Petr Dvořáček**

Studijní program: B2649 Elektrotechnika

Studijní obor: 2612R041 Řídicí a informační systémy

Téma: Aplikace pro automatizaci postupů vývoje průmyslových strojů
Application for Automation of Industrial Machine Development Procedures

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Rozbor technologií využitých při řešení práce. (TIA Portal, TIA OPENNESS, Windows.Forms, ...)
2. Popis stávajícího řešení.
3. Možnosti vytvoření aplikace propojené s vývojovým prostředím TIA Portal.
4. Implementace vybrané části aplikace - Načtení datových typů z externího datového zdroje (Deflist) a z prostředí TIA Portal.
5. Implementace vybrané části aplikace - Porovnání datových typů načtených ze zdrojů uvedených v předchozím bodě.
6. Implementace vybrané části aplikace - Generování datových typů zpět do externího datového zdroje (Deflist) a do prostředí TIA Portal.
7. Integrace řešení do celkového systému.
8. Testování řešení.
9. Závěrečná zhodnocení.

Seznam doporučené odborné literatury:

[1] ROBINSON, Simon. *C#: programujeme profesionálně*. Vyd. 1. Brno: Computer Press, 2003. Programmer to programmer. ISBN 80-251-0085-5.
[2] NAGEL, Christian, Jay GLYNN a Morgan SKINNER. *Professional C# 5.0 and .NET 4.5.1*. Indianapolis, IN: John Wiley and Sons, 2014. ISBN 978-1-118-83294-3.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jaromír Konečný, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jiří Koziorek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019 .....

Abstrakt

Tato práce je zaměřena na rozšíření automatizační aplikace Elvac Makro o funkcionalitu umožňující operace s datovými typy při vývoji řídicí aplikace pro průmyslové stroje v prostředí TIA Portal. Díky této funkcionalitě bude programátorovi umožněno jednoduše vytvářet znovupoužitelné komponenty, které si uloží a bude je moci použít znovu v jiném projektu. Jako rozhraní prostředí TIA Portal byla použita knihovna TIA Portal Openness spolu s datovým formátem XML a jako rozhraní datového úložiště, které představuje soubor ve formátu XLSX, knihovna Spire.Xls. Přínosem této práce je zefektivnění vývoje řídicí aplikace v prostředí TIA Portal a tím i zefektivnění vývoje celého průmyslového stroje.

Klíčová slova: PLC; C#; TIA Portal Openness; XML; Automatizace

Abstract

This thesis focuses on the extension of the Elvac Makro automation software. The new functionality simplifies operations with datatypes during the development of a control application in the IDE TIA Portal. It enables the programmer to create reusable components, which can be saved in the data storage and reused in other projects. This was achieved using the TIA Portal Openness library together with the XML data format as an API for the IDE TIA Portal and the Spire.Xls as an API for the data structure represented by a file in the XLSX format. Successful implementation of the functionality will increase the efficiency of the control application development and thus the development of the whole industrial machine.

Keywords: PLC; C#; TIA Portal Openness; XML; Automation

Obsah

Seznam použitých zkratek a symbolů

IDE	–	Integrated Development Environment
COM	–	Component Object Model
TIA	–	Totally Integrated Automation
API	–	Application Programming Interface
CLR	–	Common Language Runtime
MSIL	–	Microsoft Intermediate Language
CLS	–	Common Language Specification
CTS	–	Common Type System
BCL	–	Basic Class Library
OS	–	Operating System
JIT	–	Just In Time
CIL	–	Common Intermediate Language
XML	–	eXtensible Markup Language
GC	–	Garbage Collector
JSON	–	JavaScript Object Notation
YAML	–	YAML Ain't Markup Language
PLC	–	Programmable Logic Controller
HMI	–	Human Machine Interface
SCADA	–	Supervisory Control And Data Acquisition
DLL	–	Dynamic-link library
DefList	–	Definiční Listina
GUI	–	Graphical User Interface
MFC	–	Microsoft Foundation Class Library

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Vývoj průmyslového stroje je komplexní proces, ve kterém je třeba zajistit efektivní komunikaci jednotlivých specializovaných pracovišť. Samotný vývoj obvykle probíhá v komerčním prostředí, kde je ze strany vedení kladen velký důraz jak na rychlost vývoje, tak na kvalitu finálního produktu. Aby byla zajištěna efektivní komunikace, rychlý vývoj a zároveň vysoká kvalita finálního produktu, je třeba vývoj rozdělit do několika částí a tyto předat ke zpracování jednotlivým specializovaným pracovištím. Pro zvýšení efektivity práce je nutné identifikovat části produktu, které se často opakují jak v produktu samotném, tak u různých produktů a tyto části nahradit předpřipravenými a znovupoužitelnými komponentami.

Tato práce se zabývá možnostmi automatizace rutinních postupů při vývoji průmyslových strojů [3, 10]. Práce je zaměřena především na zefektivnění návrhu řídicí aplikace, ke kterému je potřeba zpracovat velký objem dat (stovky až tisíce hodnot). Pro vývoj aplikace byl zvolen jazyk C# mimo jiné i proto, že cílovou platformou je operační systém (dále jen OS) Windows a jazyk C# se pyšní skvělou podporou pro rychlý vývoj desktopových aplikací právě pro tuto platformu. Pro propojení aplikace s prostředím TIA Portal byla jako rozhraní použita knihovna TIA Portal Openness.

Cílem této práce je implementovat novou funkcionalitu do již existující aplikace, která umožní operace s *datovými typy* v aplikaci a v prostředí TIA Portal podle standardů firmy ELVAC a.s. Tato nová funkcionalita by měla na stisknutí tlačítka generovat různé datové typy v prostředí TIA Portal podle předem definovaného vzoru a tímto zefektivnit celý proces vývoje, od návrhu po testování správné funkčnosti, mimo jiné i tím, že omezí chyby způsobené lidským faktorem.

Bakalářská práce je rozdělena do následujících sekcí. Sekce 2 se zabývá popisem postupu při vývoji průmyslových strojů a stávajícího řešení pro problematiku řešenou v této práci. Dále jsou zde vytyčeny základní pojmy (Deflist). V sekci 3 jsou podrobně rozebrány technologie použité ke zhotovení praktické části práce, jedná se především o programovací jazyk C# a jeho knihovny, s jazykem spojenou platformu .NET Framework, vývojové prostředí TIA Portal a datový formát XML. Dále je v této sekci popsána aplikace Elvac Makro, která se vyvíjí právě pro účel automatizace postupů vývoje průmyslových strojů. Do této aplikace bude implementována nová funkcionalita jako řešení praktické části práce. Samotná implementace řešení praktické části práce je popsána v sekci 4. V sekci 5 je popis integrace řešení z praktické části práce do aplikace Elvac Makro. V sekci 6 je rozebráno testování řešení praktické části práce a v sekci 7 jsou shrnuty výsledky práce.

2 Rozbor řešení

V této sekci je podrobně rozebraná problematika vývoje průmyslových strojů a současný stav. Dále zde bude definován pojem Deflist (definiční listina). Vývoj průmyslových strojů je obvykle rozdělen do následujících pěti fází [10]:

1. Definice požadavků zákazníkem – počáteční fáze projektu. V úzké spolupráci se zákazníkem jsou definovány jeho požadavky, finanční možnosti a termíny dodání, dále je zhotovena smlouva o dílo a technická specifikace. Jedná se o nejstěžejnější část vývoje, i drobné pochybení při definici požadavků může mít fatální důsledky, jako například překročení finančního limitu nebo nedodržení lhůty dodání. K tomuto účelu se obvykle používá jazyk UML [1]. Některé části této fáze mohou být vypracovány přímo zákazníkem (například technická specifikace).

2. Mechanický návrh – strojní projektant má za úkol vytvořit MC&I (Motor Control and Instrumental) listinu a model stroje, ze kterého je možné vytořit technické výkresy specifické dle zadání konkrétní zakázky. Jako vstup zde figuruje technická specifikace z bodu předchozího.

3. Elektrický návrh – hlavním úkolem projektanta elektrického návrhu je doplnit a vhodně modifikovat MC&I listinu a vytvořit elektrotechnické schéma, které slouží jako podklad pro automatické generování IO (Input Output) listiny. Vstupem pro tuto fázi jsou názvy konstrukčních částí, typové a technologické označení vycházející z návrhu v prostředí EPLAN a katalogové parametry jednotlivých konstrukčních částí.

4. Návrh řídicí aplikace – zahrnuje jak návrh logiky řídicího systému, tak návrh zpracování signálů. K tomuto je nutné mít výstup z předešlé fáze (elektrického návrhu) například v podobě listu vstupů a výstupů (IO list). K návrhu řídicí aplikace je v našem případě použito prostředí TIA Portal.

5. Návrh vizualizace – finální fáze projektu. K návrhu vizualizace je potřebný výstup z předešlé fáze (návrh řídicí aplikace) v podobě seznamu tagů. V této fázi je v našem případě opět používáno vývojové prostředí TIA Portal.

Fáze vývoje průmyslového stroje 1. – 3. je nutné dělat postupně, jejich vstupy a výstupy na sebe úzce navazují. 4. a 5. fázi vývoje je možné vypracovávat současně. Všechny fáze vývoje průmyslového stroje jsou pod kontrolou zadavatelské firmy, která kontroluje stav řešení a diskutuje o možném postupu.

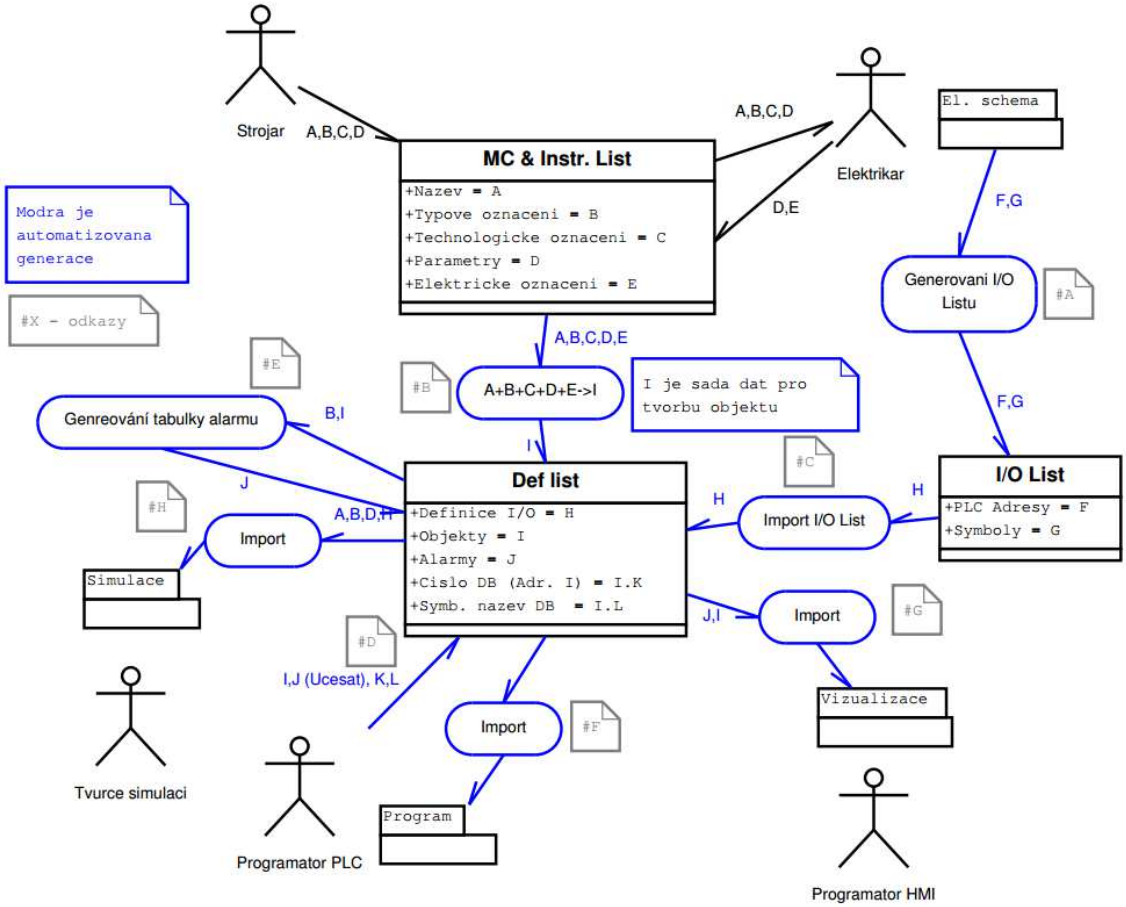
Tato práce je zaměřena především na bod 4, tedy na návrh samotné řídicí aplikace, ke kterému je potřeba výstup z bodů předchozích.

Na obrázku 1 jsou znázorněny toky dat při návrhu řídicí aplikace. Celý proces začíná fází mechanického návrhu (strojař), jehož hlavním výstupem je MC&I list, do kterého strojař přidá

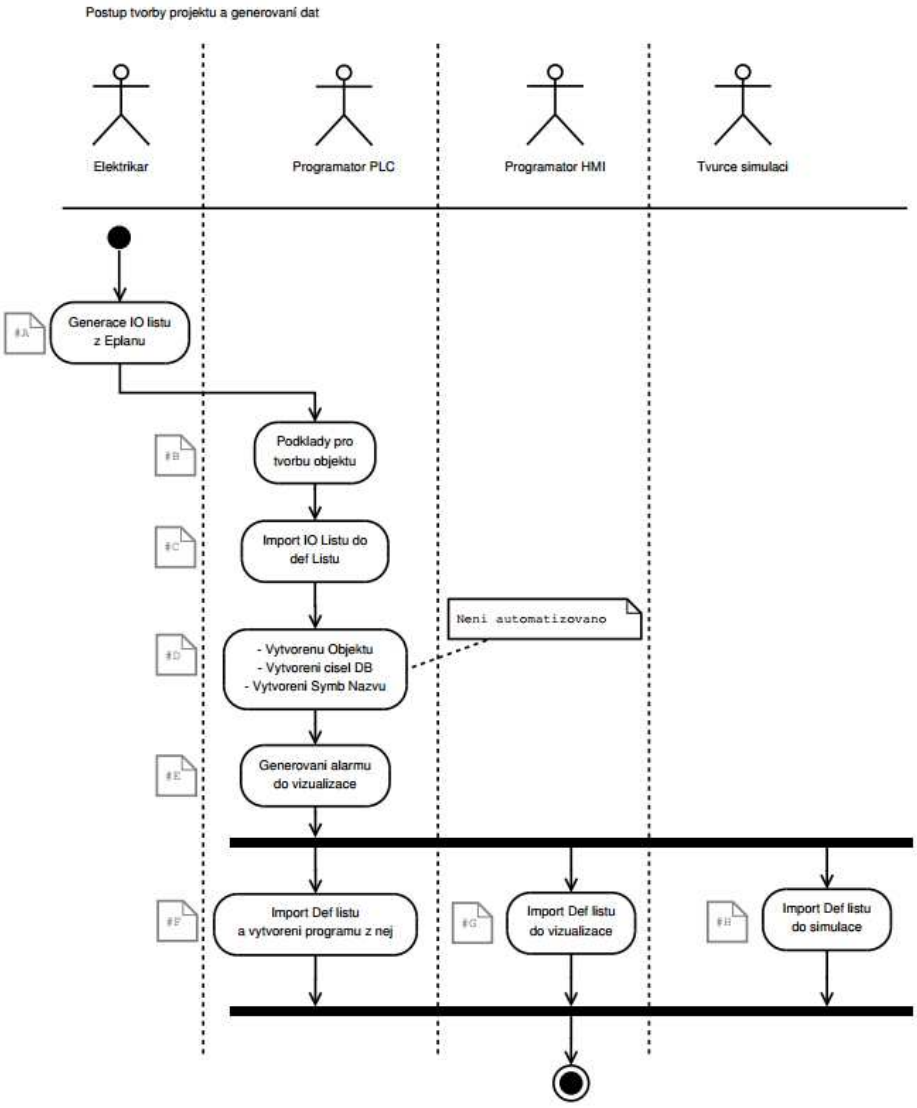
názvy, typová označení, technologická označení a parametry objektů. Tento dále figuruje jako vstup pro fázi elektrického návrh (elektrikář), který MC&I list doplní o rozšířené parametry a elektrická označení objektů. Výstupem fáze elektrického návrhu není jen MC&I list, ale také elektrické schéma, ze kterého je pomocí prostředí EPLAN vygenerován I/O list.

I/O list a MC&I list představují vstupní parametry pro vytvoření Deflistu. Deflist je dále použit k vytvoření alarmů pro vizualizaci, samotné vizualizace, simulace, ale hlavně programu pro PLC. Tato práce je zaměřena právě na proces importu dat z Deflistu do programu, kde program představuje řídicí aplikaci v prostředí TIA Portal.

Obrázek 2 blíže popisuje kroky potřebné k vytvoření řídicí aplikace z Deflistu. Tato práce se zabývá fází D, konkrétně *vytvořením objektu*. K vytvoření objektu v prostředí TIA Portal je třeba v tomto prostředí také vytvořit k němu vázané datové typy, toto bude dále rozebráno v dalších kapitolách práce.



Obrázek 1: Diagram znázorňující datové toky při tvorbě Deflistu.



Obrázek 2: Diagram doplňující diagram na obrázku 1

2.1 Popis stávajícího řešení

Stávající řešení pro operace s datovými typy při tvorbě řídicí aplikace v prostředí TIA Portal je založeno na ručním kopírování datových typů ze souboru ve formátu XLSX (Deflist, blíže popsáno v sekci 2.2). Při stávajícím řešení je tedy třeba otevřít prostředí TIA Portal a v něm vytvořit nový datový typ. Dále je třeba otevřít Deflist, vyhledat příslušný datový typ, označit jej a řádek po řádku jej kopírovat do vytvořeného datového typu v prostředí TIA Portal.

2.2 Deflist

Deflist je strukturovaný, sofistikovaně popsaný soubor ve formátu .xlsx, který obsahuje data potřebná k vytvoření řídicí aplikace pro PLC. K vytvoření Deflistu je třeba výstup z bodů 2 a 3, tedy z mechanického a elektrického návrhu. Jedná se o klíčový dokument definující řízený systém z pohledu PLC, jsou v něm obsaženy informace ve formátu skupinových datových typů, které se dále skládají z malých datových typů. Jeden skupinový datový typ se může skládat celkem z pěti malých datových typů a tyto malé datové typy se dále skládají z *triviálních* datových typů (Int, Real, Bool atd.). Rozdělení malých datových typů má své opodstatnění v jejich významu, ten je rozděluje následovně:

- sp - setpoint (*žádaná hodnota*) představuje požadovanou hodnotu regulované veličiny. Obvykle je tato hodnota vyjádřena reálným (*Real*) nebo přirozeným (*Int*) číslem či logickou hodnotou (*Bool*). Může se jednat o rychlost otáčení motoru, teplotu, hladinu tekutiny nebo i čas vykonávání určitého procesu obvykle ve vteřinách.

- cmd - command (*příkaz*) je obvykle tvořen výhradně polem bitových hodnot datového typu logické hodnoty (*Bool*) a představuje rozhraní pro ovládání stroje, kdy zapsání logické hodnoty 1 nebo 0 na určitý bit v paměti PLC představuje vyslání příkazu pro provedení určité akce ovládanému zařízení. Obvykle se jedná o akci typu spuštění/vypnutí motoru, změna směru jízdy, otevření/zavření ventilu atd.

- sts - status (*stav*) představuje stav regulované veličiny řízeného systému. Tento datový typ se může skládat jak z hodnot číselných, tak z hodnot logických nebo i časových (*DateTime*). Může reprezentovat například aktuální hladinu kapainy v nádrži, aktuální teplotu, směr jízdy, rychlost otáčení motoru atd.

- flt - fault (*porucha*) je tvořen výhradně logickými hodnotami a navíc musí obsahovat hlášku ve tvaru textového řetězce. Tento datový typ informuje operátora, pokud se nepodaří vykonat některý z příkazů (cmd), například po vyslání příkazu *jeď vpřed* není stav (sts) objektu *jede vpřed*, v takovém případě je hodnota proměnná *porucha* odpovídající tomuto poruchovému stavu změněna z 0 na 1 a o zobrazení chybové hlášky a její zpracování se postará vizualizace.

- msg - message (*zpráva*) podobně jako *porucha* je tvořen výhradně logickými hodnotami a hláškou ve tvaru textového řetězce. Na rozdíl od datového typu *porucha* slouží k informování operátora, pokud není možné provést některý *příkaz* a zároveň se nejedná o poruchu. Typicky může jít o stav, kdy se operátor pokusí spustit robot v době, kdy je otevřená bezpečnostní klec.

Dále Deflist obsahuje v listu *objects* tzv. *objekty*. Objekt představuje určitou komponentu řízeného systému (například motor, ventil nebo pumpa). Ke každému objektu je nutné přiřadit

symbol, datový typ, datový prostor, popis a navíc je možné přiřadit i programový kód. Objekty ovšem nejsou v přímém vztahu s touto prací, proto nebudou dále podrobněji rozebírány.

3 Rozbor použitých technologií

V této sekci jsou podrobně popsány technologie použité pro vypracování praktické části práce. Jedná se o programovací jazyk C# a jeho knihovny TIA Portal Opennes, TIA Opennes Helper, Windows.Forms a Spire.XLS, dále je zde popsán .NET Framework, vývojové prostředí TIA Portal a datový formát XML.

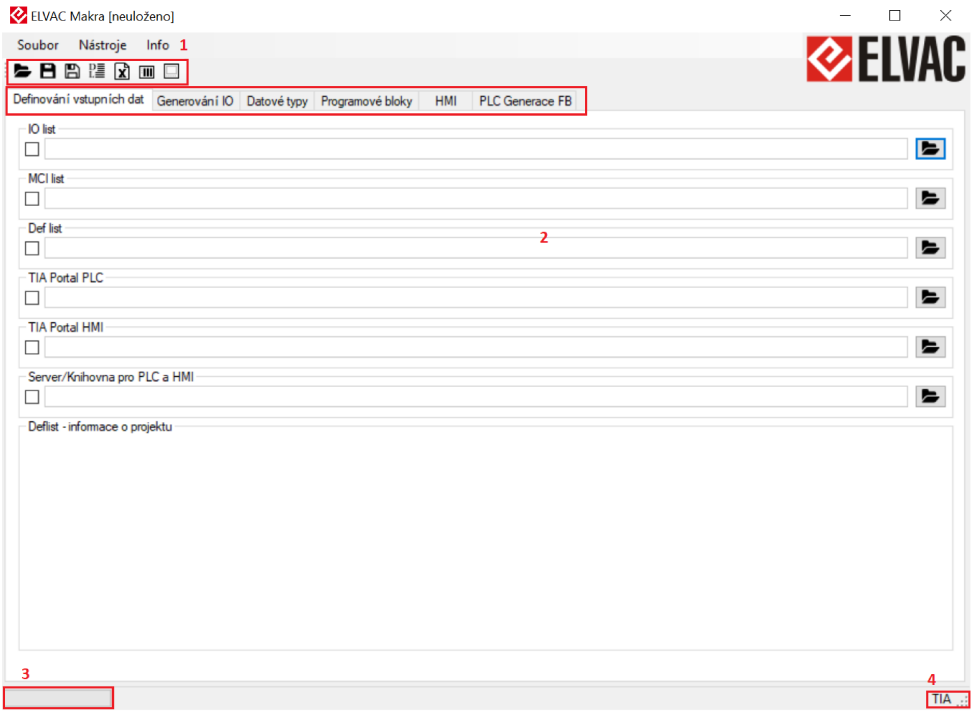
3.1 Aplikace Elvac Makro

Aplikace Elvac Makro (obrázek 3) je vyvíjena za účelem zautomatizování toku dat vznikajících při realizaci automatizačního projektu a pro automatizované generování struktury a kódů projektu v prostředí pro programování řídicích aplikací dle standardu firmy Elvac. Jako programovací jazyk pro tuto aplikaci byl zvolen jazyk C# pro jeho pokročilou podporu tvorby GUI (Graphical User Interface) na platformě Windows OS (Operating System).

Tato aplikace může být použita pro vývoj řídicích aplikací nejen v prostředí TIA Portal, ale také v prostředí Bosh Rexroth, tato práce se ovšem zabývá částí pro prostředí TIA Portal, další popis se tedy bude zabývat pouze částí aplikace komunikující s prostředím TIA Portal.

Cílem vývoje této aplikace je zefektivnit proces návrhu řídicí aplikace a její vizualizace. Základem pro zefektivnění procesu návrhu řídicí aplikace je implementace funkcionality pro operace s datovým typy, od které se odvíjejí další pokročilejší funkcionality, jako je funkcionalita pro zautomatizování generace programových a funkčních bloků podle předpřipraveného vzoru, která je v době odevzdání práce ve fázi testování, a funkcionalita pro automatizované generování kostry programu, která bude implementována dle plánu v řádech dvou měsíců. Poslední zmíněná funkcionalita je zároveň hlavním cílem při zefektivnění návrhu řídicí aplikace.

Aplikace by měla také zefektivnit vývoj vizualizace pro vytvořenou řídicí aplikaci. Toho je dosáhnuto exportem datových typů z Deflistu v předem specifikovaném formátu do souboru .xlsx, který je možné načíst v programu pro tvorbu vizualizace (WinCC). Tato funkcionalita ovšem přímo nesouvisí s touto prací, proto dále nebude podrobněji rozebírána.



Obrázek 3: Hlavní panel aplikace Elvac Makro. V boxu 1 jsou (zleva) prvky pro: otevření uloženého projektu aplikace, uložení, uložení jako, načtení Deflistu, otevření Deflistu, otevření PLC projektu, otevření HMI projektu. V boxu 2 jsou záložky s jednotlivými panely pro zpracování toku dat, v boxu 3 se nachází prvek pro zobrazení aktuálního stavu aplikace a v boxu 4 je informace o ovládaném rozhraní (TIA a nebo Rexroth).

3.2 Programovací jazyk C#

Jedná se o moderní programovací jazyk běžící na .NET (čteno *dotnet*) platformě (sekce 3.2.1), vyvíjený firmou Microsoft. Je založený na jazyku Java a C++, syntaxe vychází z C. Jazyk je vysokoúrovňový objektově orientovaný, nese ovšem prvky funkcionálního jazyka (např. klíčové slovo *static*[1], lambda výrazy[2]).

Programovací jazyk C# je úzce pojený s platformou .NET, ze které přebírá určité funkce (například automatický garbage collector), ale obsahuje i funkce, které v prostředí .NET dostupné nejsou (například možnost přetěžování operátorů). Jazyk je každopádně překládán do jazyka .NET platformy, a sice jazyka MSIL (sekce 3.2.1)[13, 18, 19]. V následujících odstavcích budou vysvětleny pojmy vstahující se k programování v jazyku C#[9, 13, 18, 19]

- OOP - OOP je paradigma programování stojící na třech pilířích, a sice *zapouzdření, dědinost* a *polymorfismus*.

[1] Deklaruje funkci volanou na třídě, nikoliv na její instanci.
[2] Zavedeny v C# 3.0, zkracují zápis anonymní metody, neuvádí se datové typy proměnných, tyto jsou určeny až při kompilaci.

  - Zapouzdření - Objekty nemohou být vzájemně modifikovány jinak, než přes rozhraní, které samy poskytují. Toto zaručuje konzistenci dat.
  - Dědičnost - Objekty mohou získat a rozšířit implementaci jiných objektů a takto vzniklé nové objekty jsou uspořádány ve stromové struktuře. Díky dědičnosti je možno omezit duplicitní kód.
  - Polymorfismus - Obecně se jedná o možnost sjednocení různých tříd na základě společných znaků (společná metoda, vlastnost). Toho se dosahuje u různých jazyků různým způsobem, u dynamicky typovaných jazyků (Python[5]) stačí společnou metodu zavolat, u staticky typovaných jazyků se toto provádí pomocí dědičnoti, kdy každý potomek může být načten do instance rodičovské třídy (C#[13, 18, 19]).

- Třída - Představuje vzorový návrh objektu, zapouzdřuje jeho vlastnosti a metody. Z jiného úhlu pohledu se dá třída považovat za datový typ. Hodnota třídního objektu je v jazyce C# předávána odkazem.

- Metoda - Metoda je funkce svázaná s instancí třídy. V jazyku C# může být zavolána pouze na konkrétní předem vytvořené instanci. Může mít vstupní parametry a může vracet jednu hodnotu jakéhokoliv datového typu.

- Konstruktor - Metoda volaná vždy spolu s klíčovým slovem *new* při vytváření instance třídy. V konstruktoru jsou obvykle volané metody, které vytvářejí vlastnosti daného objektu.

3.2.1 Microsoft .NET Framework

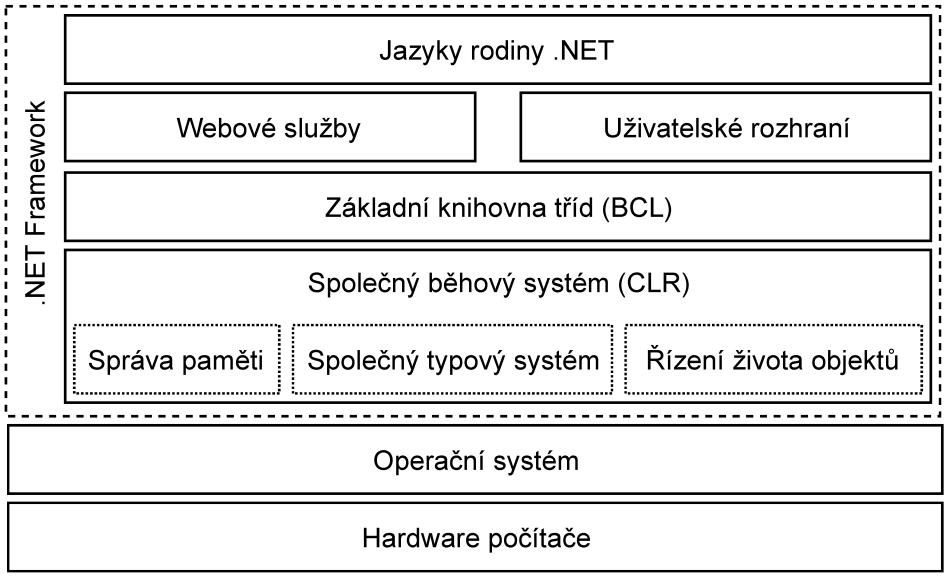
Na programovací jazyk C# v žádném případě nelze nahlížet izolovaně, ale vždy v kontextu platformy .NET Framework.[13] Platforma .NET Framework byla představena firmou Microsoft v roce 2002, kdy nahradila dosud používaný COM[13] (Component Object Model[3]) pro svou jednoduchost a flexibilitu.[13, 18] Platforma představuje nové rozhraní pro programování aplikací běžících na operačním systému Windows [13]. Je nezbytná jak pro programování v C#, ale také pro spouštění v něm vytvořených aplikací. Jedná se o prostředí běžící nad operačním systémem Windows[4], která se skládá z několika částí [19]:

- Společné běhové prostředí (dále jen CLR, zahrnuje společnou jazykovou specifikaci CLS a společný typový systém CTS) má na starost běh programů přeložených z jazyků rodiny .NET[5] do mezijazyka MSIL. Protože se různé jazyky běžící v prostředí .NET Frameworku překládají do jednoho mezijazyka MSIL, je možné napsat různé části programu v jakémkoli

[3] COM je distribuovaný, objektově orientovaný a na platformě nezávislý systém pro tvorbu binárních softwarových komponent schopných spolupracovat mezi sebou. Byl vyvinutý firmou Microsoft v roce 1993.[13, 17]

[4] Upravená komunitní verze prostředí .NET Core je kompatibilní nejen s OS Windows, ale i s OS Linux a jinými OS.

[5] C#, VB F# a další.



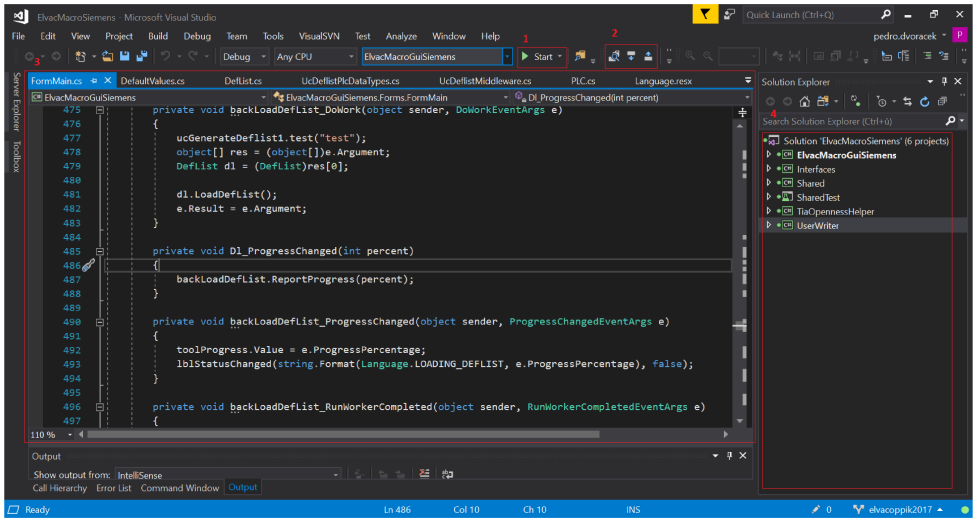
Obrázek 4: Struktura prostředí .NET Frameworku [19].

z těchto jazyků, v CLR budou tyto části navzájem propojeny. Dále se CLR stará o správu paměti, řídí život objektů[6], zpracovává výjimky (*exception handling*), řídí chod vláken aplikace a zajišťuje základní zabezpečení aplikace.[18]

- Základní knihovna tříd *System* (dále jen BCL) je jmenný prostor obsahující základní a bázové třídy, které jsou společné pro jazyky rodiny .NET. V těchto třídách jsou definovány často používané konstanty, referenční datové typy, události, metody reagující na události (tzv. *event handler*), rozhraní, atributy a zpracování událostí (např. *System.DateTime, System.String, System.EventArgs, System.EventHandler, System.IEquatable*). Tyto prvky není třeba implementovat pro jednotlivé jazyky rodiny .NET zvlášť, protože se jedná o předkompilované knihovny. BCL zahrnuje také knihovny pro tvorbu grafického uživatelského rozhraní (dále jen GUI, např. *System.Windows.Forms*) a pro webové služby (např. *System.Web*).

- Překladače různých jazyků rodiny .NET a JIT compilátory překládající přenositelný kód CIL (soubor s koncovkou .exe, nebo .dll) do strojového kódu specifického pro danou hardwarovou konfiguraci. Kompilátory JIT jsou celkem tři, liší se podle toho, kdy probíhá kompilace, ale také výpočetní a paměťovou náročností.

[6] Alokování paměti na *haldě* při vytváření nového objektu a uvolnění paměti mechanismem *Garbage Collector* (dále jen GC). GC počítá množství referencí u každého objektu, pokud se tento rovná nule, je objekt neaktivní a GC jej z paměti uvolní. O spuštění GC se stará .NET Framework, je ovšem možné jej zavolat funkcí *GC.Collect()*



Obrázek 5: Vývojové prostředí Visual Studio Community 2017. V boxu 1 je tlačítko pro spuštění debuggování aplikace, v boxu 2 jsou prvky TortoiseSVN pro verzování kódu, v boxu 3 se nachází editor kódu a v boxu 4 Solution Explorer, který reprezentuje souborový systém aplikace.

3.2.2 Microsoft Visual Studio 2017

Visual Studio je IDE (Integrated Development Environment) vyvinuté firmou Microsoft primárně určené k vývoji aplikací běžících na platformě .NET Frameworku. Mezi programátory je považováno za jedno z nejlepších dostupných vývojových prostředí pro svou jednoduchost, editor podporující refraktorování kódu a IntelliSense[7], ale hlavně debbuger, který dokáže pracovat nejen na úrovni kódu, ale i na úrovni strojového kódu.[8, 14]

Pro vývoj aplikace bylo použito Microsoft Visual Studio Community 2017 v anglické verzi (viz. Obrázek 5), kvůli značně větší komunitě mluvící anglicky v porovnání s česky mluvící komunitou. Community verze byla zvolena, protože je zcela zdarma, ale i přesto obsahuje všechny nástroje potřebné k vytvoření aplikace jako debugger, nástroje pro tvorbu unit testů, editaci kódu, IntelliSense a refractor[8], nástroje pro tvorbu UI (User Interface), grafu závislosti a mapu kódu, pro analýzu kódu a týmovou spolupráci. [14]

3.2.3 Knihovna Windows.Forms

Knihovna Windows.Forms je součástí .NET Frameworku. Je využívána při tvorbě GUI pro aplikace běžící na této platformě[15]. Představuje náhradu MFC (Microsoft Foundation Class Library)[15]. Knihovna byla od svého vydání v roce 2002 do roku 2018 spravována firmou Microsoft, v roce 2018 ji firma Microsoft zveřejnila jako otevřený projekt na platformě GitHub[7], nyní je tedy knihovna spravována komunitou.

[7] IntelliSense je obecný pojem pro automatizovanou pomoc a akce v aplikacích firmy Microsoft. Zřejmně nejznámějším příkladem je detekce a podtržení chybně napsaných slov v Microsoft Word[8], ve Visual Studiu pomáhá odhalit možné chyby při kompilaci již při psaní kódu.

[8]

```
//Direktiva pro přístup do jmenného prostoru knihovny
using Spire.Xls;
//Inicializace nové proměnné workbook
Workbook workbook = new Workbook();
//Načtení dat ze dokumentu.xlsx, který reprezentuje jeho cesta
string filename = "C:\cesta\k\dokumentu.xlsx";
workbook.LoadFromFile(fileName);
/*Vytvoření nové instance listu Excelu, který odpovídá listu s názvem "sts" v
    načteném dokumentu .xlsx (metoda Worksheets není citlivá na velká a malá pí
    smena)*/
Worksheet status = workbook.Worksheets["sts"];
//Pro načetní dat
DataTable dtStatuses = status.ExportDataTable(Status.Range, false, true);
```

Výpis 1: Načtení dokumentu ve formátu XLSX s použitím knihovny Spire.XLS [4].

```
//Direktiva pro přístup do jmenného prostoru knihovny
using Spire.Xls;
//Inicializace nové proměnné workbook
Workbook workbook = new Workbook();
//Inicializace nové proměnné sheet, vytvoření nového listu v proměnné workbook
Worksheet sheet = workbook.Worksheets[0];
//Vložení textu "Ahoj, XLSX!" do vytvořeného listu na pozici A1
sheet.Range["A1"].Text = "Ahoj, XLSX!";
\\Uložení vytvořeného dokumentu na zvolenou cestu
workbook.SaveToFile("C:\cesta\k\dokumentu.xlsx");
```

Výpis 2: Vytvoření dokumentu ve formátu XLSX s použitím knihovny Spire.XLS [4].

3.2.4 Knihovna Spire.XLS

Knihovna Spire.XLS je součástí .NET Framework a nabízí API pro práci se soubory ve formátu XLSX v prostředí aplikací .NET [4]. Knihovna zajišťuje parsování těchto dokumentů a umožňuje jejich načtení do objektové datové struktury v programu.(výpis 1).

Na výpisu 1 je kód pro načtení dokumentu ve formátu XLSX. Knihovna je nejprve načtena použitím klíčového slova *using*, dále je vytvořena nová instance třídy *Workbook*, na které je zavolána metoda *LoadFromFile*, do které vstupuje proměnná datového typu *string* reprezentující cestu k dokumentu. V této fázi je soubor načten do objektové datové struktury definované třídou *Workbook* a jednotlivá data z dokumentu jsou reprezentována jako vlastnosti instance proměnné *workbook*. Pro přístup k jednotlivým řádkům listu souboru je tento list uložen do proměnné *status*, která je datového typu *Worksheet*. Samotné řádky budou reprezentovány proměnnou *dtStatuses*, do které jsou načteny metodou *ExportDdataTable*, která je zavolána na instanci proměnné *status*.

Na výpisu 2 je kód pro vytvoření dokumentu ve formátu XLSX. Nejprve je načtena knihovna

Spire.Xls a vytvořena promměnná *workbook* datového typu *Workbook* (stejně jako na výpisu 1). S instancí proměnné *workbook* je svázána další proměnná *sheet* datového typu *Worksheet*, která reprezentuje list dokumentu. Proměnná *sheet* je s proměnnou *workbook* svázaná, protože proměnná *sheet* byla inicializována jako vlastnost proměnné *workbook*. Příkaz *sheet.Range["A1"].Text* přiřadí do vlastnosti proměnné *sheet* reprezentující text v buňce *A1* hodnotu *"Ahoj, XLSX!"*. Data z proměnné *workbook* jsou exportována do souboru zavoláním metody *SaveToFile* a předáním cesty, na která se má soubor nacházet.

V aplikaci byla použita knihovna Spire.XLS, protože umožňuje nejen základní operace s daty[9], ale také pokročilejší funkce, jako je změna formátu dokumentu samotného[10].

3.2.5 API TIA Portal Openness a TIA Openness Helper

Součástí instalace prostředí TIA Portal (viz. sekce 3.3) je i knihovna TIA Portal Openness představující API pro ovládání prostředí TIA Portal z aplikací běžících na platformě .NET verze 4.6.1. Aby bylo možné tuto knihovnu použít, je nutné do referencí *Siemens.Engineering* a *Siemens.Engineering.Hmi* přidat cestu k TiaPortalOpenness, která je při defaultní instalaci prostředí TIA Portal `C:\Program Files\Siemens\Automation\Portal V14\PublicAPI\V14SP1` S protředím TIA Portal se pracuje v těchto po sobě jdoucích krocích [16]:

1. Přidání referencí do programovacího prostředí a cest k jejich DLL (viz. výše)
2. Přidání direktiv pro přístup do jmenného prostoru knihoven TIA Portal Openness
3. Vytvoření instance třídy TiaPortal, propojení s prostředím TIA Portal
4. Otevření projektu na dané cestě
5. Provedení požadovaných příkazů
6. Uložení a uzavření projektu
7. Ukončení spojení s prostředím TIA Portal

```
//Direktivy pro přístup do jmenného prostoru knihoven
using System;
using System.IO;
using Siemens.Engineering;
using Siemens.Engineering.Hmi;

//Vytvoření nové instance prostředí TIA Portal, vytvoření spojení s prostředím
TiaPortal tiaPortal = new TiaPortal();
```

[9] Načítání dat, jejich přepisování atd.
[10] Barva záložek, řádku, tučné písmo atd.

```
ProjectComposition projects = tiaPortal.Projects;
FileInfo projectPath = new FileInfo(@"C:\cesta\k\projektu\projekt.ap14");
Project project = null;
try
{
    //Otevření projektu na dané cestě
    project = projects.Open(projectPath);
    //Nyní můžeme pracovat s prostředím
    //Příklad vložení datového typu do prostředí TIA Portal
    FileInfo dataType = new FileInfo(@"C:\cesta\k\datovemu\typu\dataType.xml");
    PlcTypeComposition types = plcSoftware.TypeGroup.Types;
    types.Import(dataType, ImportOptions.Override);
    //Uložení a uzavření projektu
    project.Save();
    project.Close();
}
catch (Exception)
{
    throw new Exception("Nepodařilo se otevřít projekt");
}
finally
{
    //Ukončení spojení s prostředím TIA Portal
    tiaPortal.Dispose();
}
```

Výpis 3: Příklad kódu pro vložení datového typu ve formátu xml do prostředí TIA Portal s použitím TIA Portal Openness.

API TIA Portal Openness umožňuje přístup k následujícím datům[16]:

- Data projektu
- Data PLC programu
- Data HMI programu

Tato práce je však zaměřena pouze na data projektu, dále tedy bude rozebrán pouze přístup k těmto datům.

Na výpisu 3 je kód pro importování datového typu do prostředí TIA Portal. Nejprve jsou načteny potřebné knihovny klíčovým slovem *using*, poté je vytvořena proměnná *tiaPortal* reprezentující instanci prostředí. Z vlastnosti *Projects* proměnné tiaPortal je vytvořena proměnná

projects datového typu *ProjectComposition*, která reprezentuje jednotlivé projekty v prostředí. Dále je vytvořena proměnná *projectPath*, která reprezentuje cestu k projektu, do kterého si přejeme datový typ importovat. Následně je vytvořena proměnná *project* datového typu *Project*, jejíž počáteční hodnota je hodnota *null*, do této proměnné bude následně načten projekt. V bloku *try* je kód, který může vyhodit výjimku, pokud se tak stane, provádění bloku *try* se ukončí a provede se blok *catch*, kteý vyhodí vyjímku s textem "Nepodařilo se otevřít projekt". V bloku *try* je nejprve načten projekt na cestě reprezentované proměnnou *projectPath*, poté je vytvořena proměnná *dataType* reprezentující cestu k souboru XML obsahující načítaný datový typ (struktura souboru XML je dále rozebrána v sekci 4.1.1). Následně je vytvořena proměnná *types* datového typu *PlcTypeComposition*, která reprezentuje jednotlivé datové typy v projektu. Metodou *Import* zavolané na proměnné *types*, která přijímá proměnnou reprezentující cestu k souboru XML (*dataType*) a proměnnou reprezentující režim nahrání (*ImportOptions.Override*), je datový typ nahrán do projektu v prostředí TIA Portal. Následně je projekt uložen a zavřen. Blok *finally* slouží pro ukončení spojení aplikace s prostředím TIA Portal a je volán vždy (po provedení bloku *finally* i bloku *catch*).

Knihovna TIA Openness Helper jako tzv. *helper* usnadňuje práci s knihovnou TIA Portal Openness tím, že obsahuje některé často používané funkce předimplementované a odladěné.

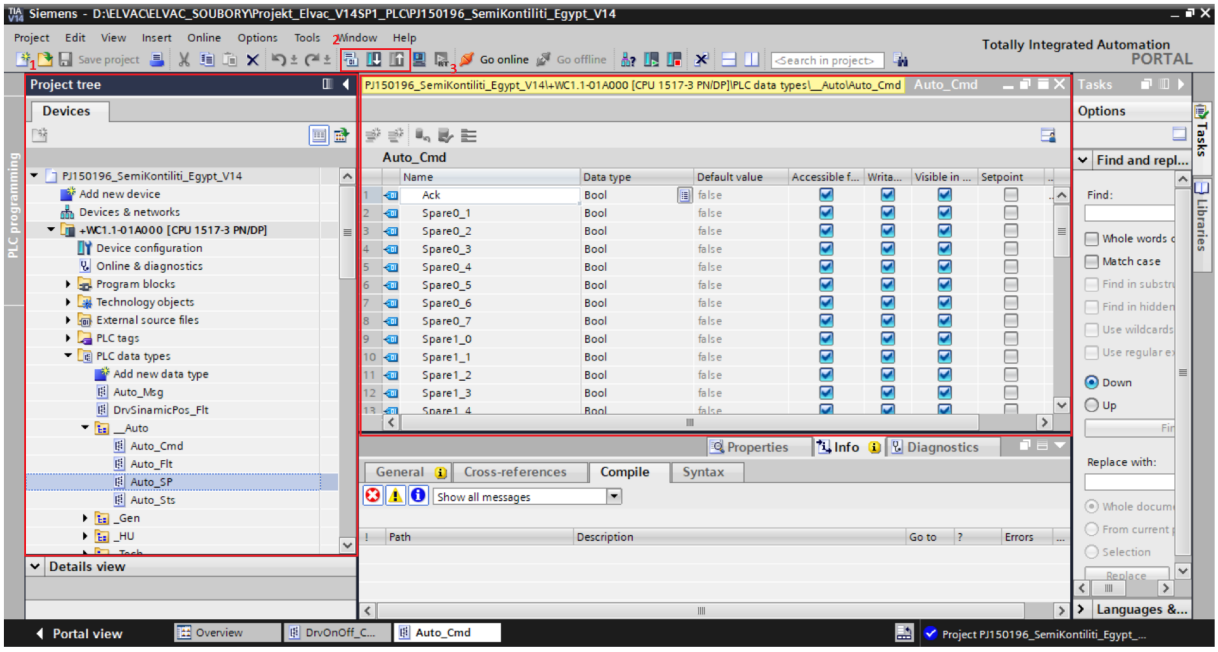
3.3 Vývojové prostředí TIA Portal

Vývojové prostředí TIA Portal je určeno pro návrh, realizaci a provoz průmyslově automatizačních strojů. Jedná se o produkt firmy Siemens, který je součástí komplexní strategie TIA pro minimalizaci nákladů na vývoj těchto strojů. Jeho předchůdci jsou WinCC pro tvorbu vizualizace, STEP 7 pro tvorbu řídicí aplikace a STARTER pro parametrování pohonů. Prostředí TIA Portal představuje sjednocené prostředí vycházející ze tří zmíněných prostředí, jako jedno z mála umožňuje jak samotné programování řídicí aplikace pro PLC, tak vytvoření vizualizace HMI [11] (Human Machine Interface) či SCADA[12] (Supervisory Control And Data Acquisition) pro tuto aplikaci.

Pro účely této práce bylo použito prostředí TIA Portal verze 14 (dále jen V14), nicméně aplikace Elvac Makro je kompatibilní i s novější verzí V15.

[11] Slouží k ovládání konkrétního stroje, obvyle v provedení ovládacího panelu v blízkosti stroje.

[12] Slouží k monitorování, ovládání a sběru dat z centrálního pracoviště, obvykle běží na počítači na řídicím můstku.



Obrázek 6: Vývojové prostředí TIA Portal V14. V boxu 1 se nachází souborová struktura programu, v boxu 2 se nacházejí tlačítka pro spuštění kompilace kódu a pro nahrávání a stahování programu z PLC a v boxu 3 je zobrazený vybraný datový typ *Auto_Cmd*.

3.4 Datový formát XML

"Přestože jsou dnes počítače schopny pracovat s elektronickými dokumenty, které obsahují obrázky, hudbu či video, je nutné je organizovat do určitého druhu infrastruktury. Standart XML takovou platformu nabízí."[2] XML (v překladu *rozšiřitelný značkovací jazyk*) je jazyk a zároveň datový formát primárně určený k serializaci věcných dat a k jejich výměně mezi aplikacemi. Obsah XML dokumentu je uložen jako čitelný text, jeho struktura je lehce srozumitelná. Uživatel jej může upravovat z libovolného textového editoru (viz. výpis 5). Syntaxe jazyku rozlišuje malá a velká písmena a zahrnuje:

- *tagy*, které slouží jako značky pro parsování dokumentu. Začínají znakem < a končí > (viz. výpis 5). Tyto se dále dělí na:
  - *start-tag* (např. <name>) obsahující pouze charaktery písmen značící začátek *elementu* v dokumentu.
  - *end-tag* (např. </name>) začínající lomítkem značící konec *elementu* v dokumentu
  - *empty-element tag* (např. <linebreak />) definující *element* bez *obsahu*
- *elementy*, které začínají značkou *start-tag* a končí značkou *end-tag*, anebo se sestávají pouze ze značky *empty-element tag*. Jsou to nositelé dat. Jako *obsah elementu* označujeme vše co je mezi značkami *start-tag* a *end-tag*, může to být text, ale i další *elementy* (viz. výpis 5)

- *attributy*, což jsou proměnné *elementu*. Tyto se vpisují do těla značky *start-tag*, a nebo *empty-element tag* (viz. výpis 4)
- *deklarace* obsahuje informace o dokumentu, jako je verze xml a použitá znaková sada (viz. první řádek ve výpisu 5)

```
<name id="10">Petr Dvoracek</name>
```

Výpis 4: Příklad *elementu* **name** s *attributem* **id**, jehož *hodnota* je **10**. *Obsah elementu* je **Petr Dvoracek**

```
<?xml version = "1.0"?>
<contact-info>
   <name>Petr Dvoracek</name>
   <company>VSB</company>
   <phone>123 456 789</phone>
   <line-break />
</contact-info>
```

Výpis 5: Příklad souboru XML, *contact-info, name, company, phone* představují *elementy* uvozené značkou *start-tag* (např. <name>) a ukončené značkou *end-tag* (např. </name>). *line-break* představuje *element* uvozený a zároveň ukončený značkou *empty-element tag*

Další datové formáty určené pro přenos dat mezi aplikacemi jsou např. JSON[13] a YAML[14]. V této práci byl použit formát XML, protože právě tento využívá pro výměnu dat prostředí TIA Portal.

[13] Nejpoužívanější formát odvozený z javascriptu. Tím, že syntaxe jazyka používá méně znaků, je zpracování dat ve formátu JSON rychlejší, ale také je jeho zápis pro člověka čitelnější.

[14] Nejčastěji používaný pro konfigurační soubory. Jeho syntaxe je podobná syntaxi JSON, je lépe čitelný než XML.

4 Implementace vybraných částí aplikace

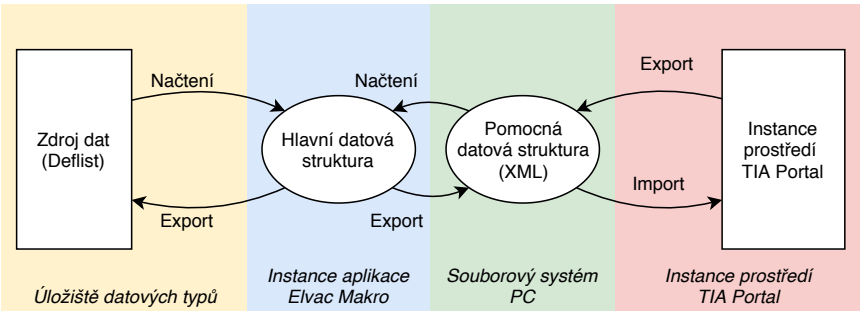
Tato sekce se zabývá implementací vybraných částí aplikace. Je zde popsán způsob, jakým se načítají datové typy jak z prostředí TIA Potal, kde se s těmito pracuje při vývoji stroje, tak z externího zdroje, který představuje standard firmy při programování různých strojů. Dále je v této sekci popsán způsob porovnávání datových typů načtených z prostředí TIA Portal a z externího zdroje a jejich generování z prostředí TIA Portal do externího datového zdroje a naopak.

4.1 Vzájemné propojení aplikace, prostředí TIA Portal a zdroje dat

Pokud chceme propojit dvě nezávislé aplikace, máme na výběr z několika možností. V případě webové aplikace se nabízí možnost využití webového API (Application Programming Interface), v případě desktopové aplikace (což je i případ vypracování praktické části práce) můžeme toto propojení založit na výměně souborů. Obě zmíněná řešení mají jedno společné, a sice vyměňovaná data musejí být v obou aplikacích ve stejném formátu. Je možné vytvořit si vlastní formát dat, toto se ale obvykle nevyplatí, zvláště pokud na projektu pracuje omezený počet vývojářů. Druhou možností je využít formát již existující (XML, JSON). Pro účely praktické části bakalářské práce byl zvolen první zmíněný formát, tedy formát XML, který je detailně popsán v kapitole 3.4. Tento formát byl zvolen právě proto, že jej využívá knihovna TIA Openness pro export dat z prostředí TIA Portal.

Pro získání dat z prostředí TIA Portal ve formátu XML je nutné s tímto prostředím nějakým způsobem komunikovat. V případě přístupu uživatele k tomuto prostředí to umožňuje GUI (Graphical User Interface), v případě přístupu jiné aplikace je nutná existence API k tomuto prostředí. Naštěstí je nativní součástí instalace prostředí TIA Portal i knihovna TIA Portal Openness (viz. sekce 3.2.5, která představuje API mezi jakoukoliv aplikací napsanou v jazyce, který umožňuje použití .dll knihoven, a prostředím TIA Portal. V ideálním případě by toto API mělo být schopno načíst data z prostředí přímo do objektové struktury programu. Bohužel toto knihovna neumožňuje, namísto toho data exportuje do souboru ve formátu XML a teprve tento je možné načíst jinou aplikací (obrázek 7).

V případě exportu dat z jakékoliv jiné aplikace do prostředí TIA Portal je situace podobná, data je třeba uložit do souboru ve formátu XML podle předem daného schématu, které odpovídá schématu dat exportovaných samotným prostředím TIA Portal. Tento soubor je následně nahrán do prostředí přes API TIA Portal Openness vysláním žádosti o načtení souboru na určené cestě (výpis 6).



Obrázek 7: Datové toky v aplikaci Elvac Makro

```
private static void ImportUserCreatedDataType(PlcSoftware plcSoftware)
{
 FileInfo fullFilePath = new FileInfo(@"C:\path\to\newDataType.xml");
 PlcTypeComposition types = plcSoftware.TypeGroup.Types;
 IList<PlcType> importedTypes = types.Import(fullFilePath, ImportOptions.
     Override);
}
```

Výpis 6: Import vytvořeného datového typu na cestě do prostředí TIA Portal pomocí TIA Portal Openness[16].

K načítání Deflistu ani k exportu dat do něj není třeba využívat jako prostředníka formát XML. O nahrání dat z Deflistu do objektové reprezentace aplikace Elvac Makro se postará knihovna Spire.XLS. Způsob zpracování objektové reprezentace Deflistu je rozebrán v následujících sekcích, stejně jako export dat z aplikace do této listiny.

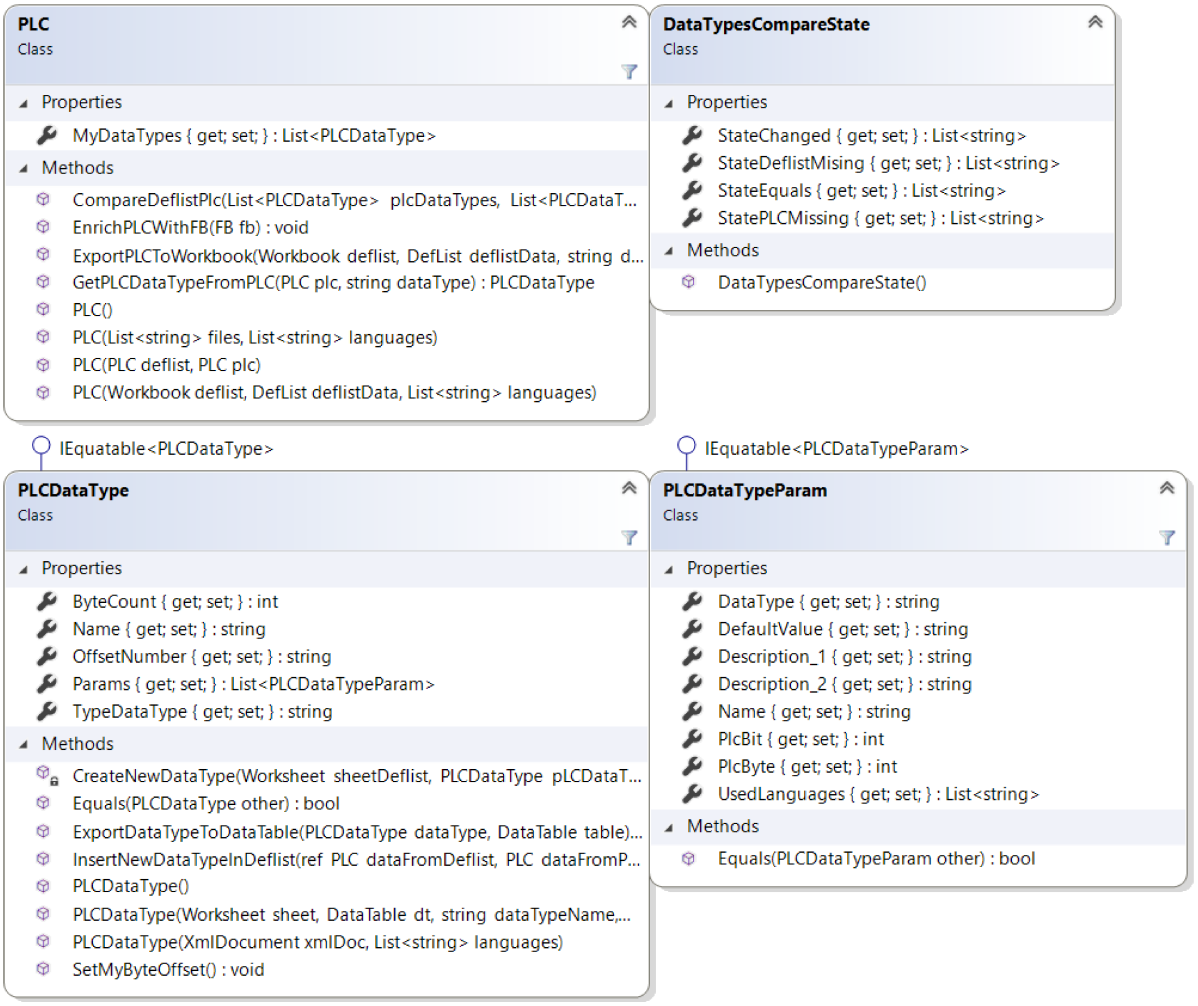
Data jsou z externího zdroje[15] vždy načítána do totožné objektové struktury reprezentované třídou PLC, liší se pouze funkcemi, které provádějí samotné načítání, tedy konstruktory (viz obrázek 8). Toto je velmi výhodné při porovnávání dat v různých strukturách (viz sekce 4.2).

Z obrázku 8 je patrné, že každá instance třídy PLC obsahuje property MyDataTypes, která je datového typu List<PLCDataType> a představuje jednotlivé datové typy.

Třída PLCDataType obsahuje property ByteCount reprezentující velikost datového typu v bytech, Name reprezentující jméno datového typu, OffsetNumber obsahující informaci o bytovém posunu v plc, TypeDataType nesoucí informaci o tom, o jaký datový typ se jedná[16], a konečně property Params datového typu List<PLCDataTypeParam> nesoucí parametry datového typu.

Třída PLCDataTypeParam obsahuje property týkající se parametrů datového typu (DataType, DefaultValue, Description_1, Description_2, Name, PlcBit, PlcByte) a property potřebnou pro porovnávání datových typů UsedLanguages.

[15] Definiční listina, nebo prostředí TIA Portal.
[16] Status, command, setpoint, fault nebo message.

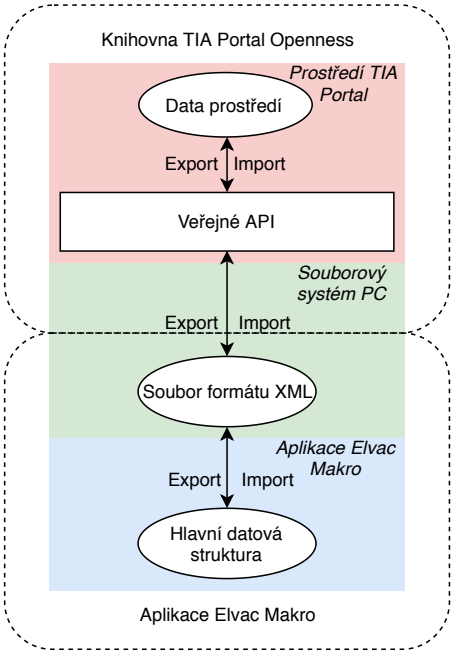


Obrázek 8: Třídní diagram struktury datových typů v aplikaci. V diagramu jsou uvedeny pouze property, konstruktory a rozhraní, ze kterých tyto třídy dědí.

4.1.1 Propojení s prostředím TIA Portal

V této sekci bude popsán způsob propojení aplikace Elvac Makro s prostředím TIA Portal, konkrétně bude popsaný způsob exportu dat z prostředí TIA Portal do aplikace a naopak.

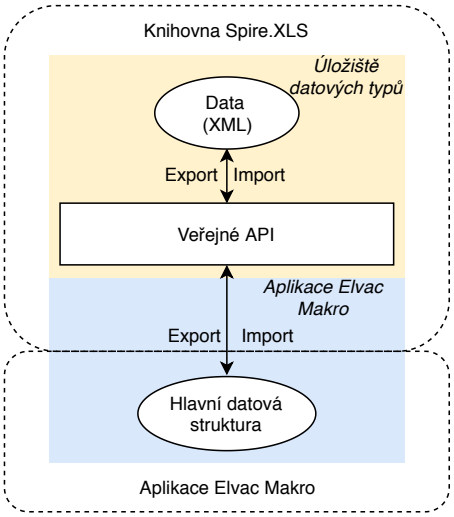
Samotná funkcionalita získávání datových typů z prostředí TIA Portal byla implementována do funkce GenerateTreeView[17]. Do této vstupuje proměnná *project* datového typu Project reprezentující v aplikaci celý projekt z prostředí TIA Portal. Proměnná *project* obsahuje kolekci proměnných datového typu PlcSoftware, tyto jednotlivé proměnné reprezentují jednotlivé programy v projektu z prostředí TIA Portal. Z kolekce je vybrána proměnná reprezentující uživatelem zvolený program a s touto se nadále pracuje pod jménem projectPlc. K vybrání této proměnné je využita knihovna TIA Openness Helper (viz. sekce 3.2.5), konkrétně funkce *OpennessHelper.GetAllPlcSoftwares*, do které vstupuje proměnná *project*).



Obrázek 9: Výměna dat mezi aplikací Elvac Makro a prostředím TIA Portal je založena na datovém formátu XML.

Jednotlivé datové typy z prostředí TIA Portal ve zvoleném PLC programu jsou uloženy v proměnné *projectPlc*, konkrétně v její vlastnosti *projectPlc.TypeGroup.Types*. Tato vlastnost je datového typu *PlcTypeComposition* a reprezentuje všechny datové typy daného PLC programu. Přes vlastnost *projectPlc.TypeGroup.Types* se dále iteruje tak, že jednotlivé prvky *PlcTypeComposition* jsou uloženy v proměnné *plcDataType* datového typu *Siemens.Engineering.SW.Types.PlcType*. Proměnná plcDataType je exportována do souboru XML metodou *Export*, která je volána na její instanci, a vstupuje do ní instance proměnné *FileInfo* se zvolenou cestou pro export a poměnná

[17]Zde je také funkcionalita vizualizace.



Obrázek 10: Výměna dat mezi aplikací Elvac Makro a Deflistem. Veřejné API představuje knihovna Spire.XLS.

ExportOptions.WithDefaults popisující způsob exportu. Všechny datové typy jsou exportovány ve formátu XML do souboru *%temp%\ElvacMacro\readerPlc*, kde jsou dále načteny do objektové reprezentace v aplikaci.

Načtení datových typů do objektové reprezentace aplikace Elvac Makro ze souboru XML je implementováno v konstruktoru třídy *PLC*, do kterého vstupuje proměnná *files* datového typu *List<string>* obsahující cesty k datovým typům, které byly vyexportovány z prostředí TIA Portal. Dále do konstruktoru vstupuje proměnná *languages* datového typu *List<string>* s informacemi o použitých jazycích. Datové typy se načítají v následujících případech:

- porovnávání datových typů v definiční listině a v prostředí TIA Portal, konstruktor je volaný funkcí *Compare_DoWork*
- převod datových typů z prostředí TIA Portal do definiční listiny, konstruktor je volaný funkcí *GenPlcToDeflist_DoWork*

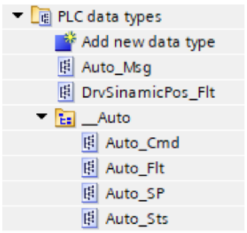
Z přípony *_DoWork* u obou funkcí je patrné, že proces načítání datových typů využívá komponenty background worker ke spuštění procesu na jiném vláknu, než ve kterém běží GUI. Díky tomu je možné aplikaci používat i při načítání datových typů, aplikace nezamrzne (viz sekce 5.2).

Zmíněný konstruktor třídy *PLC* postupně vytváří instance třídy *PLCDataType* konstruktorem této třídy, do kterého vstupuje konkrétní datový typ (z prostředí TIA Portal) načtený do instance třídy *XmlDocument* a proměnná *languages*. Vytvořená instance třídy *PLCDataType* je následně přidána do vlastnosti *MyDataTypes* typu *List<PLCDataType>*. Konstruktor třídy *PLCDataType*, do něhož vstupuje instance třídy *XmlDocument* se jménem *xmlDoc* a proměnná

DrvOnOff_Cmd

	Name	Data type	Default value	Accessible f...	Writa...	Visible in ...	Setpoint	Comment
1	Ack	Bool	false	☑	☑	☑	☐	Acknowledge
2	Str	Bool	false	☑	☑	☑	☐	Start
3	Stp	Bool	false	☑	☑	☑	☐	Stop
4	Spare0_3	Bool	false	☑	☑	☑	☐	
5	Spare0_4	Bool	false	☑	☑	☑	☐	
6	Spare0_5	Bool	false	☑	☑	☑	☐	
7	Spare0_6	Bool	false	☑	☑	☑	☐	
8	Spare0_7	Bool	false	☑	☑	☑	☐	
9	Spare1_0	Bool	false	☑	☑	☑	☐	
10	Spare1_1	Bool	false	☑	☑	☑	☐	
11	Spare1_2	Bool	false	☑	☑	☑	☐	
12	Spare1_3	Bool	false	☑	☑	☑	☐	
13	Spare1_4	Bool	false	☑	☑	☑	☐	
14	Spare1_5	Bool	false	☑	☑	☑	☐	
15	SetWh	Bool	false	☑	☑	☑	☐	Set workhours
16	RstWh	Bool	false	☑	☑	☑	☐	Reset working hours

Obrázek 11: Datový typ DrvOnOff_Cmd v prostředí TIA Portal



Obrázek 12: Pokud datový typ v prostředí TIA Portal existuje, je přehrán (adresář *___Auto*), pokud neexistuje, je nahrán do kořenového adresáře *PLC data types* jako datový typ *Auto_Msg*.

languages postupně přiřadí hodnoty všem vlastnostem své instance podle jím odpovídajících hodnot v proměnné *xmlDoc*.

Generování datových typů do prostředí TIA Portal je založeno na tvorbě a výměně souborů formátu XML (stejně jako získávání datových typů z prostředí TIA Portal). K tomuto účelu byla implementována třída *TiaXmlEngineFactory*, která definuje proceduru pro převod datového typu z objektové reprezentace aplikace do souboru formátu XML (serializace), který je možné importovat do prostředí TIA Portal.

Samotná funkcionalita pro generování datových typů z aplikace do prostředí TIA Portal byla implementována vzhledem k časové náročnosti operace do metody *BckWrkrGenDeftoPlc_DoWork*. Zde jsou z GUI získány datové typy vybrané k exportu do prostředí TIA Portal. Vybrané datové typy jsou následně exportovány do souboru ve formátu XML na cestě *%temp%\ElvacMacro\readerPlc*, tento soubor nese jméno datového typu, z toho plyne, že toto jméno musí být unikátní. Následně jsou tyto vyexportované datové typy načteny do prostředí TIA Portal pomocí knihovny TIA Portal Openness tak, že pokud načítaný datový typ v prostředí již existuje, je přepsán a je zachována jeho pozice ve skupinovém datovém typu, a pokud v prostředí není, je do něj nahrán (výpis 6) do kořenového adresáře datových typů (obrázek 12).

4.1.2 Propojení s Deflistem

V této sekci bude popsán způsob propojení aplikace Elvac Makro s Deflistem. Bude zde probráno jak načítání dat z Deflistu do aplikace, tak export dat z aplikace do Deflistu.

Propojení aplikace Elvac Makro s Deflistem vyžadovalo v porovnání s propojením aplikace s prostředím TIA Portal menší úsilí a to z důvodu použití knihovny, která je schopna data ze souboru XLSX načíst do objektové reprezentace v aplikaci přímo. Odpadá tedy nejen nutnost exportování, ukládání, načítání a zpracování soborů ve formátu XML (viz obrázek 10)

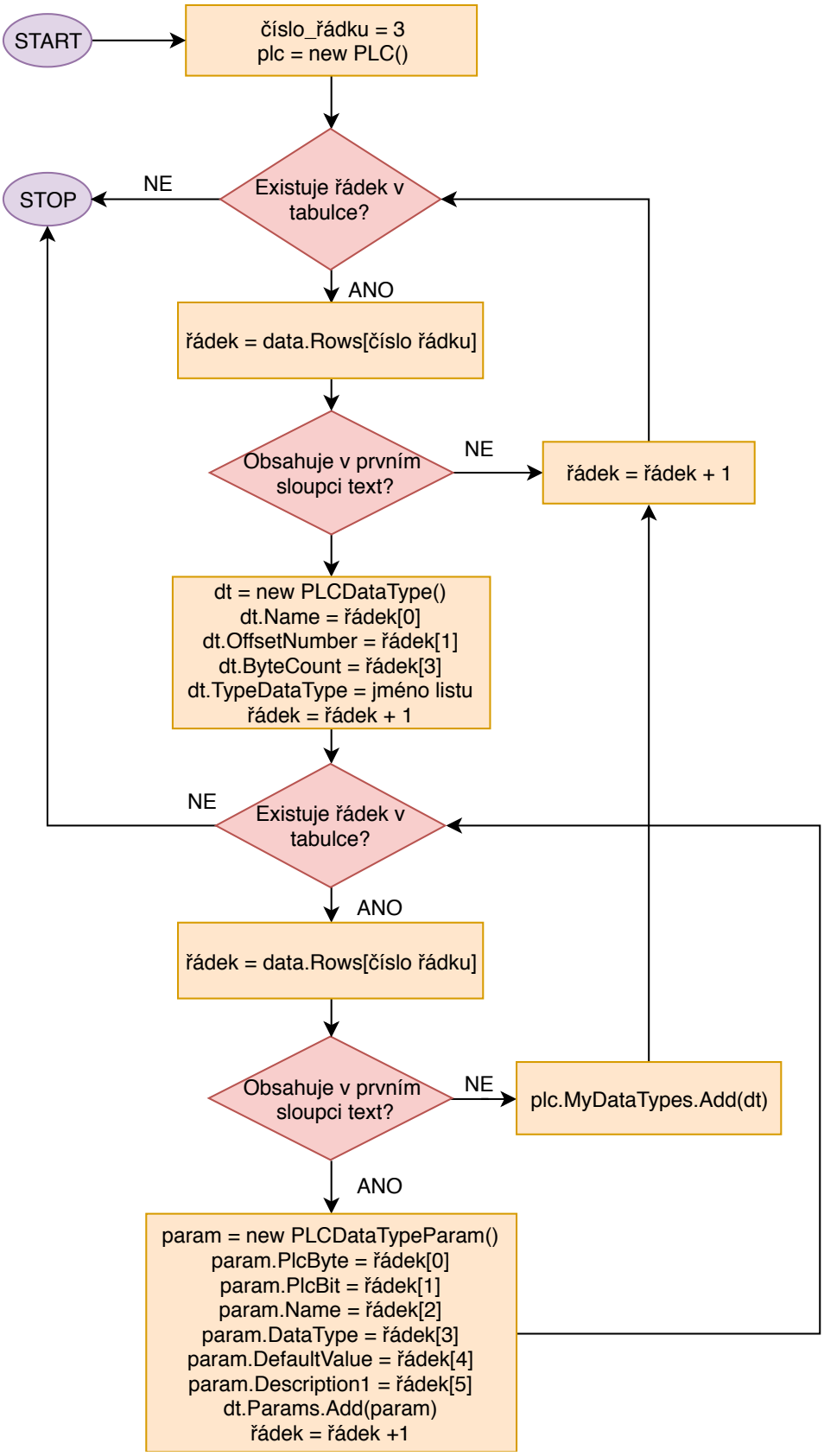
Datové typy jsou v Deflistu uloženy v listech STS, CMD, SP, FLT a MSG. Struktura definiční listiny je daná standardem firmy, struktura datového typu uloženého v Deflistu je patrná z obrázku 15, z tohoto je zřejmé, že v jednom sloupci jsou různé typy dat (např. datový typ je ve stejném sloupci jako Bytová pozice jeho parametrů). Data tedy nelze explicitně načíst jako sloupce, převést na pole datového typu string a vyfiltrovat prvky string.empty.

Pro správné načtení datových typů z Deflistu bylo třeba implementovat sofistikovaný algoritmus 13, 14. Tento algoritmus iteruje přes všechny řádky instance DataTable (iteruje tedy přes vlastnost DataTable.Rows, která je datového typu DataRowCollection, a její prvky jsou datového typu DataRow), přičemž začíná na řádku, který je na 3. pozici[18]. Algoritmus v prvním kroku nastaví inicializační hodnotu pro pozici řádku v DataRowCollection (3) a vytvoří instanci třídy PLC (proměnná plc). V dalším kroku algoritmus zjistí, zda inicializační řádek vůbec existuje, pokud neexistuje, algoritmus se zastaví, pokud existuje, tak zjistí, zda je ve sloupci na pozici 0 v daném řádku text. Pokud zde text není, algoritmus přičte k řádkové pozici 1 a vrátí se o krok zpět. Pokud zde text je, vytvoří instanci třídy PLCDataType a načte do ní hlavičku, tedy jméno datového typu, počet bytů, jeho grupový typ a číslo posunu, dále přičte k řádkové pozici 1. V dalším kroku algoritmus zjistí, zda nový řádek existuje, pokud neexistuje, algoritmus se zastaví, pokud existuje, zjistí, zda je v řádku ve sloupci 0 text, pokud zde text není, algoritmus přidá datový typ do vlastnosti *plc.MyDataType*, přičte řádek a vrátí se (viz obrázek **??**), pokud zde text je, algoritmus vytvoří proměnnou datového typu PLCDataTypeParam a načte do ní hodnoty bytové a bitové pozice parametru datového typu, jeho jméno, datový typ, výchozí hodnotu a jeho popis. Následně tento parametr přidá do pole vlastnosti Params a zvětší číslo řádku o 1.

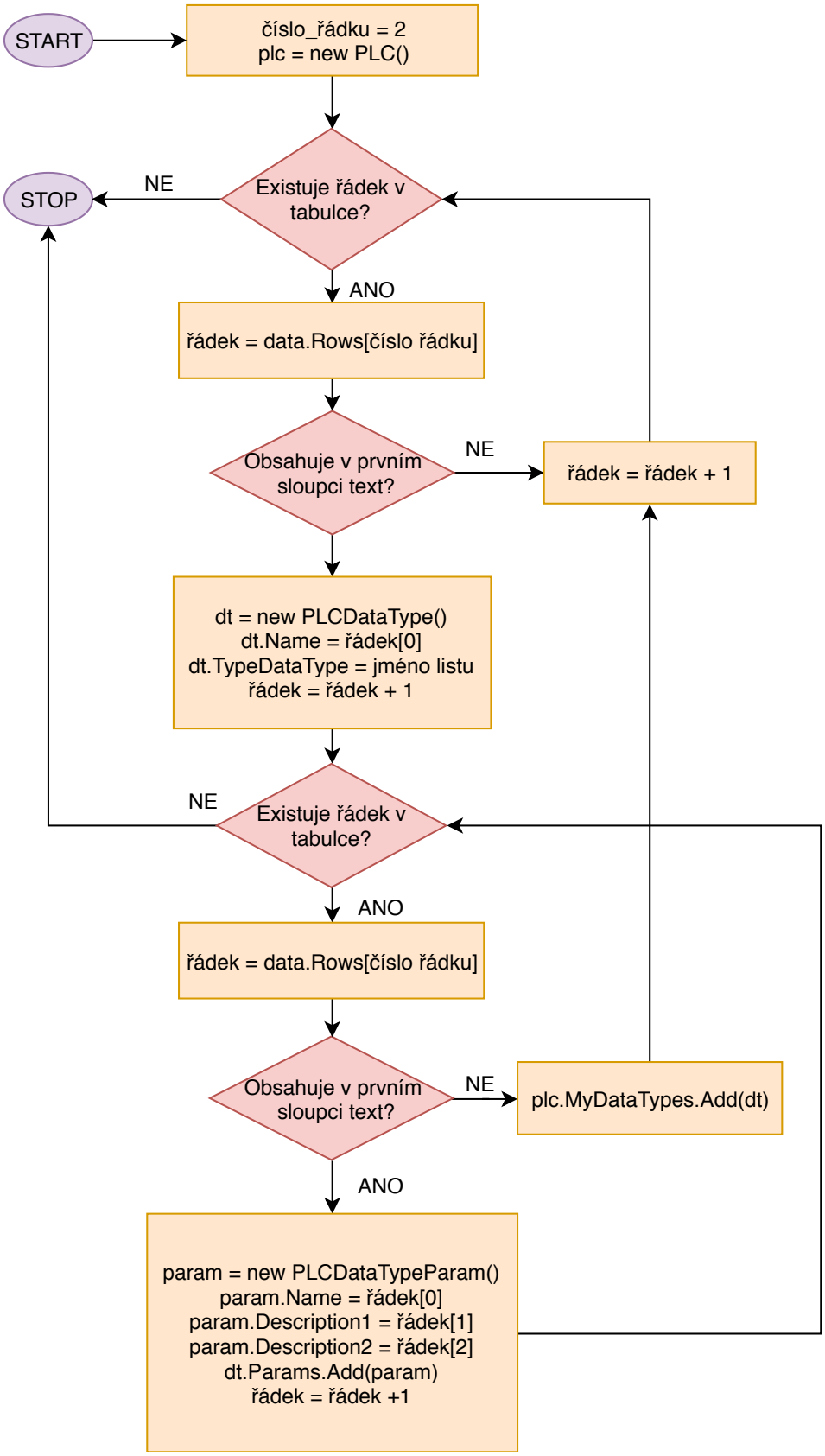
Tento algoritmus byl implementován do konstruktoru třídy PLCDataType, do něhož vstupují proměnné datového typu WorkSheet reprezentující list souboru .xlsx, string reprezentující název datového typu, Deflist reprezentující uživatelské nastavení a List<string> reprezentující v projektu použité jazyky. Konstruktor třídy PLCDataType je volán z konstruktoru třídy PLC, do něhož vstupuje navíc proměnná datového typu PlcSoftware.

V následujících odstavcích bude popsán export datových typů z objektové reprezentace v aplikaci do Deflistu. Pro každý list v Deflistu (STS, CMD, SP, FLT a MSG) je vytvořena jedna proměnná datového typu *DataTable*. Do těchto proměnných jsou postupně načteny datové typy

[18]Indexace DataTable začíná od nuly, narozdíl od třídy WorkSheet, ve která začíná index jedničkou.



Obrázek 13: Algoritmus pro načtení datových typů skupiny sts, cmd a sp



Obrázek 14: Algoritmus pro načtení datových typů skupiny flt a msg

	A	B	C	D	E	F	G
1	**HMI**	**Struc Name**	**Adr. Offset**		**Length**	**Unit**	
2	**atribut**	**Byte**	**Bit**	**Tag Name**	**Type**	**Def. Value**	**Description**
3	b - bits (default); w - words; a - array;						
296		**VlvPropOneDir**	222		**32**	**BYTE**	
297		0	0	LostDly	Real	0.0	Lost position fault filter time
298		4	0	TmReach	Real	0.0	Maximal time for reach position
299		8	0	PosMin	Real	0.0	Valve is considered as CLOSE when acutal position is less than this value [%]
300		12	0	PosMax	Real	0.0	Valve is considered as OPEN when acutal position is greather [%]
301		16	0	PosSp	Real	0.0	Position setpoint in manual mode [%]
302		20	0	DeadBand	Real	0.0	Position deadband [%]
303		24	0	RampUP	Real	0.0	Ramp up output signal
304		28	0	RampDwn	Real	0.0	Ramp down output signal
305							

Obrázek 15: Datový typ *VlvPropOneDir* v Deflistu.

odpovídajících skupin metodou textitInsertDataTypeInDataTable do které vstupuje proměnná *item* představující datový typ a proměnná *dt* představující proměnnou datového typu *DataTable*. O samotné vložení datového typu se stará zmíněná metoda *InsertDataTypeInDataTable*, která vrací obnovenou proměnnou datového typu *DataTable*.

Metoda *InsertDataTypeInDataTable* přidává jednotlivá data po řádcích. Nejprve vytvoří metodou *NewRow* nový řádek v proměnné datového typu *DataTable*, tento nový řádek je uložen do proměnné datového typu *DataRow* a jménem *param* a do toho jsou postupně nahrána odpovídající data indexací.

Z jednotlivých proměnných datového typu *DataTable* jsou následně vytvořeny proměnné datového typu *Worksheet* z proměnné *deflist* datového typu *WorkBook* a to tak, že nejprve je vytvořena samotná proměnná datového typu *Worksheet* a poté je do ní nahrána odpovídající proměnná datového typu *DataTable* zavoláním metody *InsertDataTable*. Po provedení těchto kroků obsahuje proměnná *deflist* data určená k exportu a je tedy možné na ni zavolat metodu *SaveToFile* pro uložení ve formátu XLSX.

4.2 Porovnání datových typů

Protože jsou obě načtené struktury (jak z prostředí TIA Portal, tak z Deflistu) načteny do totožného datového typu (PLC) aplikace Elvac Makro, nabízí se možnost implementovat generické rozhraní *IEqatable*. Tento přístup zpřehlední kód a zároveň bude kód čitelnější pro jakéhokoliv programátora, který je s tímto rozhraním obeznámen a umí jej používat.

Rozhraní *IEquatable* implementují třídy PLCDataType a PLCDataTypeParam. To, že třída implementuje rozhraní *IEquatable* znamená, že je v této třídě naimplementovaná metoda Equals. Ve třídě PLCDataType je metoda implementovaná tak, že vrátí hodnotu *true*, pokud mají obě porovnávané instance stejný počet parametrů a zároveň pokud metoda *Equals* zavolaná na všech jejich parametrech vrátí vždy *true*. Metoda *Equals* ve třídě PLCDataTypeParam je implementovaná tak, že porovná jméno, typ datového typu, popis a výchozí hodnotu, a pokud jsou tyto shodné, vrátí hodnotu **true**, pokud nejsou shodné, vrátí hodnotu *false*.

Tyto třídy tedy obsahují metodu Equals, která je dále využita ve funkci CompareDeflistPLC porovnávající jednotlivé struktury. Tato funkce vrací objekt datového typu DataTypesCompareState (viz výpis **??**). Tento obsahuje 4 listy datových typů string, které reprezentují názvy těch datových typů, jež jsou stejné, odlišné, chybí v deflistu, a nebo chybí v prostředí TIA Portal.obsah těchto listů je vhodným způsobem zobrazen uživateli (viz. obrázek 3).

Algoritmus pro porovnání struktur datových typů, tedy instancí tříd *PLC* je na obrázku 18 a 19. Algoritmy jsou dva, algoritmus na obrázku 18 porovná, zda jsou datové typy shodné, odlišné, a nebo zda datový typ existující v prostředí TIA Portal (*plc*) chybí v Deflistu. Algoritmus na obrázku 19 určí, zda datový typ existující v Deflistu chybí v prostředí TIA Portal (*plc*). Pro správné porovnání datových typů je tedy třeba použít oba dva algoritmy.

Do obou algoritmů vstupují proměnné *plc* a *deflist* reprezentující datové struktury a proměnná *stavy* datového typu *DataTypesCompareStates*. Její vlastnosti (viz. obrázek 8) jsou datového typu *List<string>* a obsahují jména v podobě textového řetězce těch datových typů, které jsou po porovnání v odpovídajícím stavu (stejné, odlišné, chybějící v Deflistu, chybějící v PLC).

Dále je na obrázku 18 a 19 pro názornost vyznačen cyklus *foreach*, a to čárkovanou čarou a písmeny *A* a *B*. V cyklu *A* na obrázku 18 algoritmus iteruje přes prvky kolekce plc.MyDataTypes, která představuje datové typy načtené z prostředí TIA Portal, a to tak, že nejdříve ověří, zda v této kolekci jsou nějaké prvky, pokud je počet prvků v kolekci větší než nula, uloží první prvek do proměnné *a* a nastaví hodnotu proměnné *nenalezen* na *pravda*. Pokud v kolekci nejsou žádné prvky, je algoritmus ukončen. Algoritmus dále pokračuje cyklem *B*, stejně jako u cyklu *A* je nejdříve ověřen počet prvků v kolekci deflist.MyDataTypes, pokud je tento počet větší než nula, je prvek na pozici nula z kolekce deflist.MyDataTypes načten do proměnné b. Pokud je počet prvků roven nule, je celý algoritmus ukončen. V cyklu *B* je dále ověřeno, zda datové typy z prostředí TIA Portal (*a*) a z Deflist (*b*) mají stejné jméno. Pokud tyto dva mají stejné jméno, je hodnota proměnné *nenalezen* nastavena na *pravda* a následuje porovnání samotných parametrů těchto datových typů. Pokud se datové typy *a* a *b* jmenují rozdílně, cyklus *B* pokračuje dalším prvkem bez změny jakékoliv proměnné. Pokud se datové typy *a* a *b* ukážou jako stejné, je stav datového typu s daným jménem *stejný* a toto jméno je uloženo do kolekce *stavy.StateEquals*, pokud jsou rozdílné, je jméno datového typu uloženo do kolekce *stavy.Changed*. V této fázi algoritmus pokračuje buďto vybráním dalšího prvku z kolekce (pokud existuje) a opakokváním cyklu *B*, a nebo ukončením cyklu *B* a pokračováním cyklu *A*. Cyklus *A* pokračuje ověřením, zda je hodnota *nenalezen* pravdivá, pokud ano, znamená to, že datový typ neexistuje v Deflistu, jeho jméno je tedy uloženo do kolekce *stavy.StateDeflistMissing* a algoritmus dále pokračuje ve stejné větvi, jako když je hodnota proměnné *nenalezen* nepravdivá, a sice ověřním, zda kolekce *plc.MyDataTypes* obsahuje další prvky. Pokud kolekce *plc.MyDataTypes* obsahuje další prvky, je načten další prvek a cyklus *A* se opakuje, pokud další prvky neobsahuje, cyklus a celý algoritmus je ukončen.

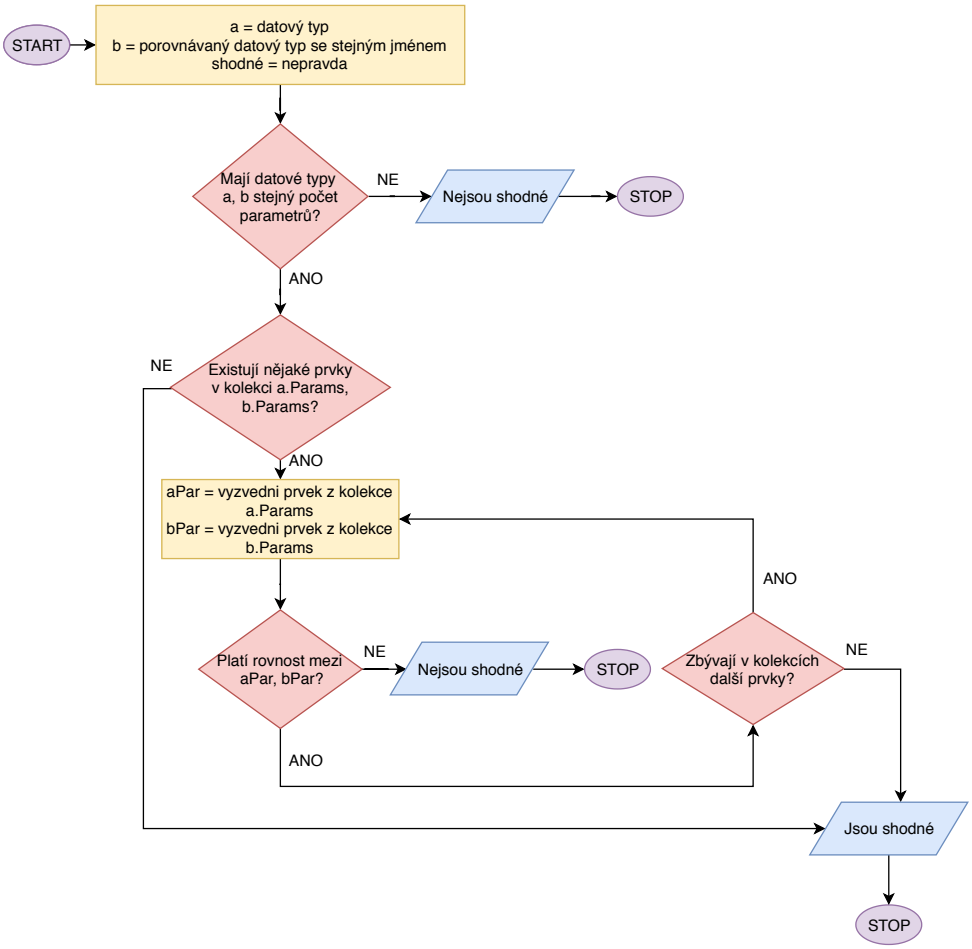
Algoritmus na obrázku 19 pracuje na stejném principu jako algoritmus na obrázku 18, jsou

zde stejné vstupní proměnné a dva cykly *A* a *B*. Rozdíl v algoritmech na obrázku 18 a 19 je v tom, že zatímco algoritmus na obrázku 18 zjišťuje, zda je daný datový typ z prostředí TIA Portal v deflistu a pokud ano, tak zjistí zda je stejný, a nebo se liší, algoritmus na obrázku 19 zjišťuje pouze to, zda datový typ z Deflistu existuje v prostředí TIA Portal. Výsledek z algoritmu je uložen, stejně jako v předchozím případě, do kolekce *stavy.StatePLCMissing*.

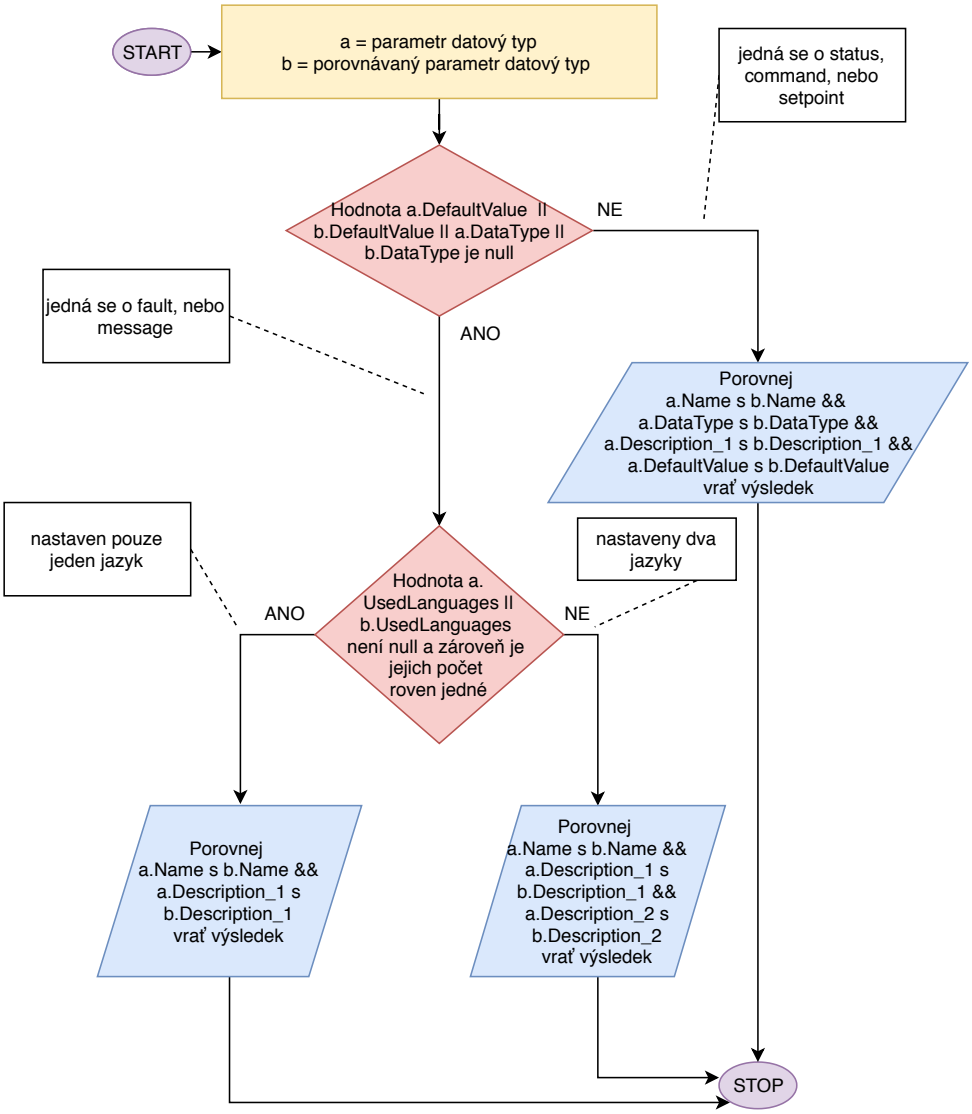
Výstupem obou algoritmů je proměnná *stavy*, ve které jsou v odpovídajících vlastnostech (viz 8) zapsány ve formě jména datového typu. Obsah této proměnné je posléze vhodným způsobem zobrazen uživateli (viz sekce 5.2).

Dále je ještě třeba objasnit jeden prvek z algoritmu na obrázku 18 a to je samotné porovnávání datových typů. Rozhodnutí *jsou datové typy a a b stejné?* odkazuje na algoritmus na obrázku 16. Tento algoritmus porovnává instance třídy *PLCDataType*, tedy samotné datové typy. Vstupují do něj dvě proměnné *a* a *b*, představující porovnávané datové typy se stejným jménem. Jako první je porovnán počet parametrů, pokud není shodný, jsou datové typy automaticky považovány za rozdílné a porovnávání je ukončeno. Pokud datové typy mají shodný počet parametrů (tento může být i nulový), je ověřeno, že je tento počet parametrů větší než nula a tyto parametry jsou následně cyklicky porovnány a v případě, že je jeden z parametrů odlišný od druhého, datový typ označen jako neshodný a algoritmus je zastaven.

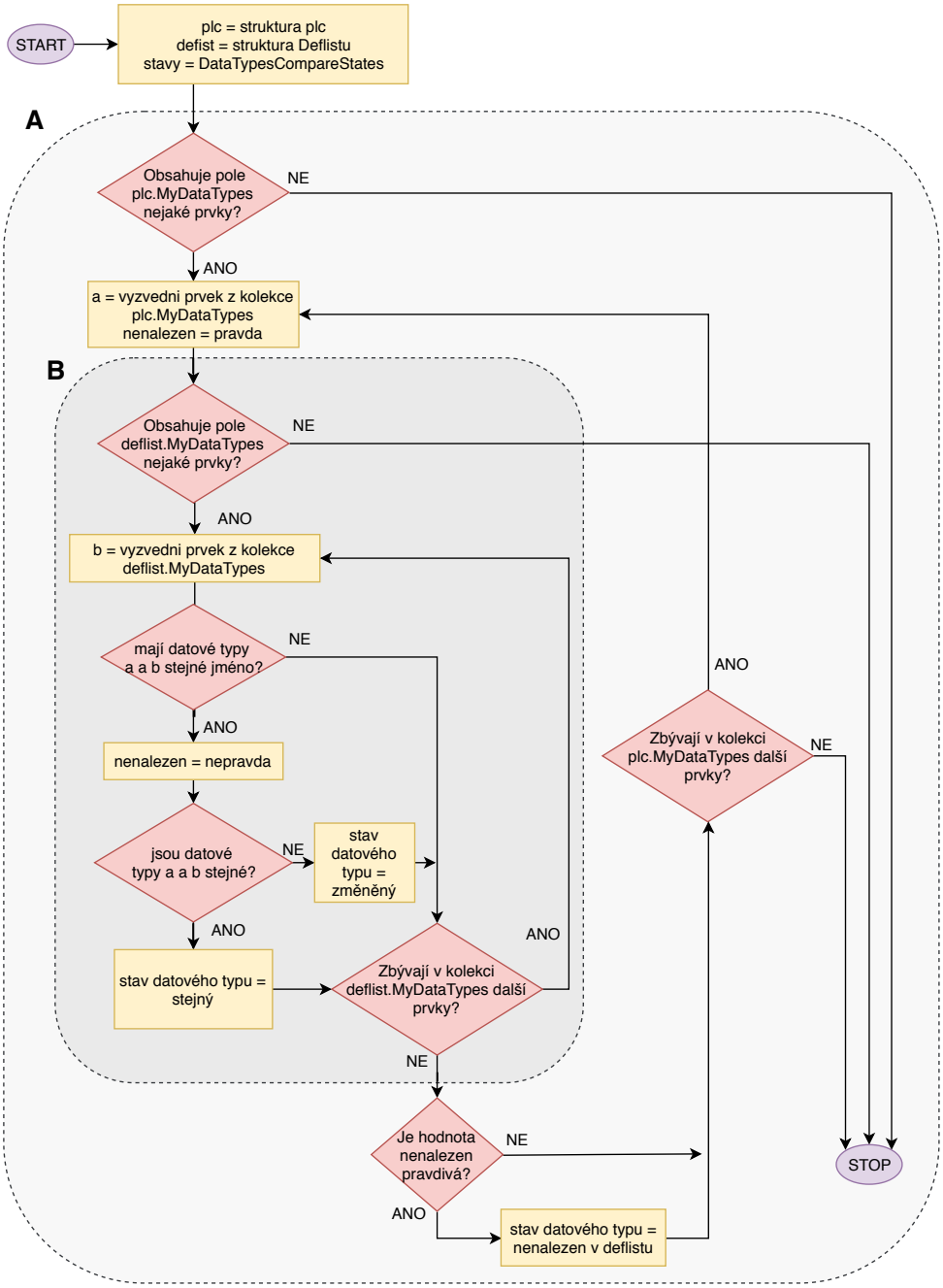
Srdcem algoritmu pro porovnávání struktur s datovými typy je algoritmus pro porovnávání samotných parametrů datových typů, tedy instancí třídy PLCDataTypeParam. Tento algoritmus je na obrázku 17, vstupují do něj dvě proměnné reprezentující samotný parametr datového typu. Nejprve je ověřeno, zda se jedná o parametr datového typu skupiny fault, nebo message (flt, msg), pokud ne, je parametr implicitně považován za parametr datového typu skupiny status, command, nebo setpoint (sts, cmd, sp) a jsou porovnány jména těchto parametrů, datový typ, popis a výchozí hodnota a je vrácen výsledek tohoto porovnání a algoritmus je ukončen. Pokud je parametr původem z datového typu skupiny fault, nebo message, je dále zjištěno, zda jsou v projektu použity dva jazyky, nebo pouze jeden. pokud je použit jeden jazyk a tento není u obou parametrů nulový, jsou navzájem porovnány první popisy obou parametrů, pokud byly použity dva jazyky, jsou navzájem porovnány oba první popisy i oba druhé popisy parametrů. Vždy je porovnán i název parametr. Hodnota porovnání je zase vrácena a algoritmus je končen.



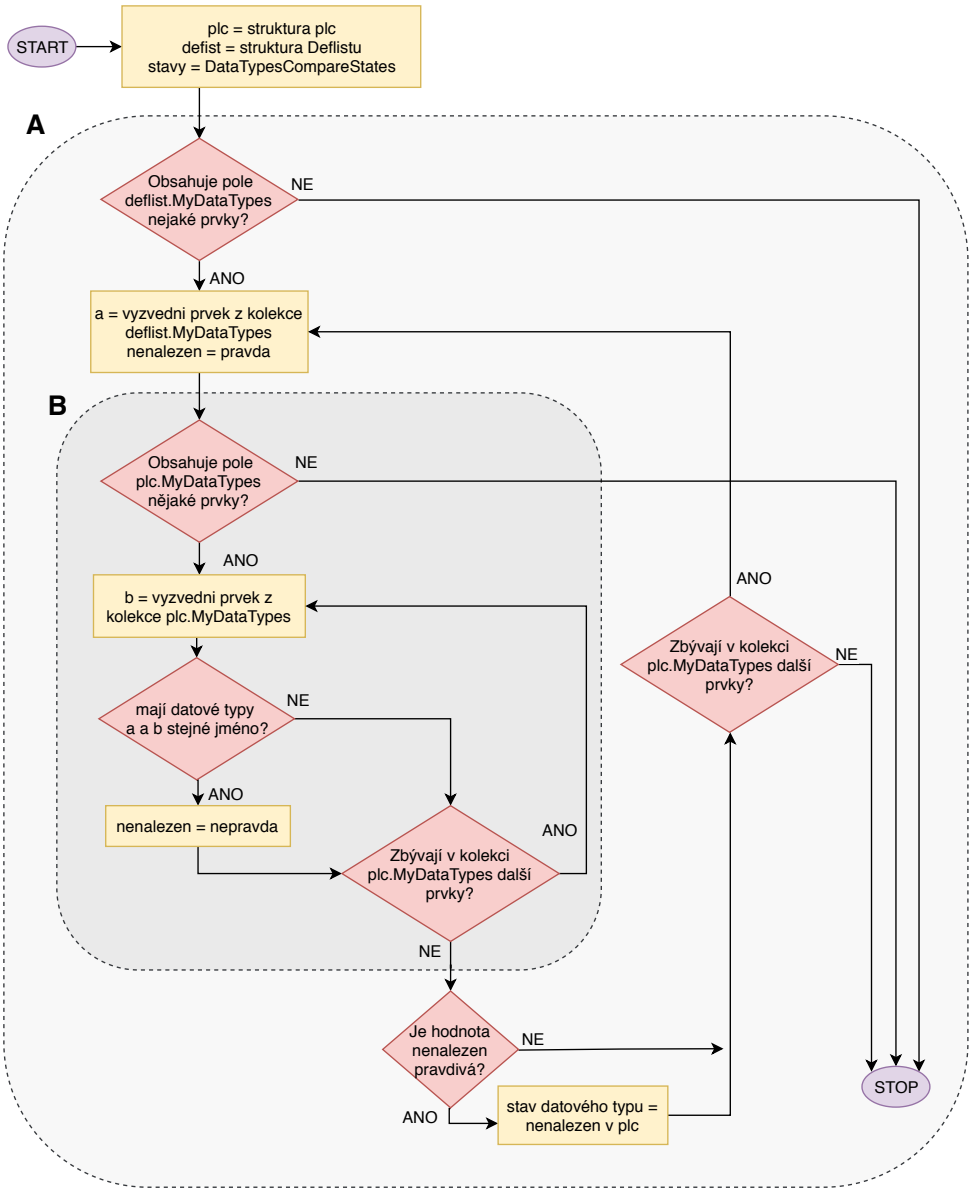
Obrázek 16: Algoritmus pro porovnání instancí třídy *PLCDataType*. Na tomto algoritmu je založena metoda *Equals* třídy PLCDataType.



Obrázek 17: Algoritmus pro porovnání instancí třídy *PLCDataTypeParam*. Na tomto algoritmu je založena metoda *Equals* třídy PLCDataTypeParam.



Obrázek 18: Algoritmus po porovnání struktury datových typů, třídy *PLC*, první část.



Obrázek 19: Algoritmus pro porovnání struktury datových typů, tedy třídy *PLC*, druhá část.

5 Integrace řešení do systému

V této sekci bude popsána integrace funkcionality umožňující operace s datovými typy a jejího GUI (Graphical User Interface). K tomuto účelu byla využita třída UserControl, která umožňuje zapouzdřit určitou funkcionalitu i s GUI a toto následně vyžívat i v jiných projektech. Použití této třídy v kombinaci se správnými programovacími technikami nám zaručí zapouzdřenost a znovupoužitelnost kódu.

Aplikace jako taková se skládá ze čtyř projektů v prostředí Visual Studio 2017. Jedná se o projekt ElvacMacroGuiSiemens obsahující prvky GUI, projekt Shared obsahuje logiku programu, projekt TiaOpennessHelper obsahuje funkce usnadňující práci s knihovnou TIA Portal Openness představující rozhraní pro prostředí TIA Portal a projekt SharedTest obsahující jednotkové testy.

5.1 Integrace logiky

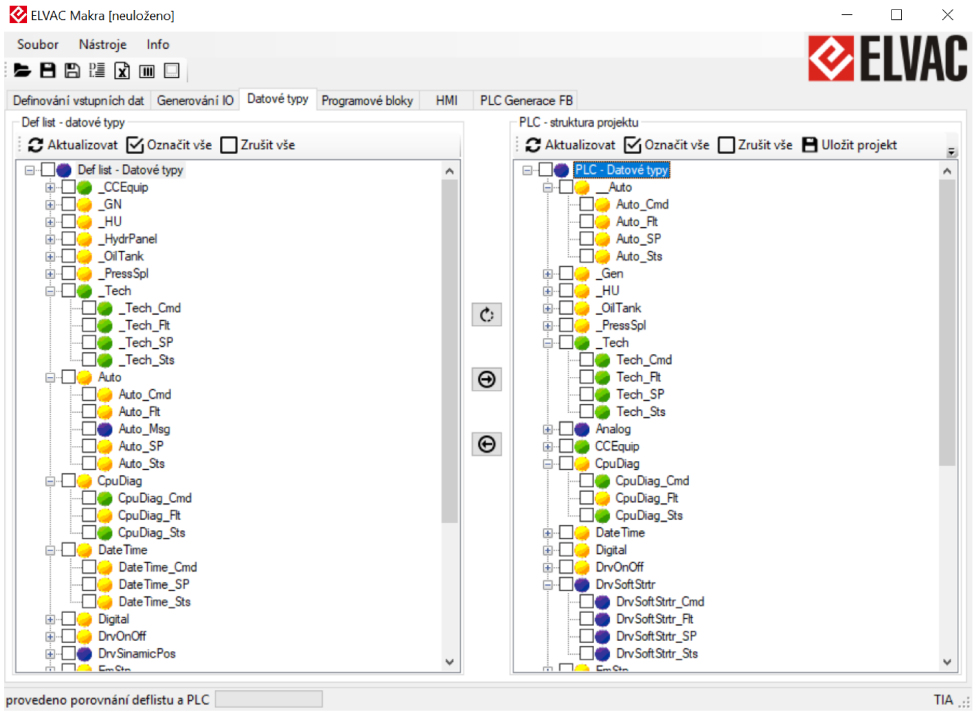
Samotná logika funkcionality pro operace s datovými typy byla implementována v projektu Shared, konkrétně do nové složky PLC byly přidány třídy PLC, PLCDataType, PLCDataTypeParam a DataTypesCompareStates. Všechny metody zajišťující operace s datovými typy v aplikaci Elvac Makro jsou zapouzdřeny v těchto třídách.

5.2 Vytvoření GUI

Při tvorbě GUI funkcionality pro operace s datovými typy bylo využito třídy UserControl, která umožňuje vytvoření znovupoužitelné komponenty grafického rozhraní. Komponenta vytvořená pomocí třídy UserControl je také zapouzdřená a s vnějším grafickým rozhraním (aplikací Elvac Makro) komunikuje přes tzv. *delegáta*[13, 18]. Samotná implementace komponenty se nachází v soubor UcDeflistPlcDataTypes v projektu ElvacMacroGuiSiemens ve složce Controls, která obsahuje všechny komponenty dědící ze třídy UserControl. V této třídě jsou zapouzdřeny všechny metody potřebné pro správný chod GUI, tedy jak pro získávání událostí a dat od uživatele, tak pro jejich správné zobrazení. Samotné zpracování dat je implementováno v projektu Shared (viz. sekce 5.1), který je importován do třídy UcDeflistPlcDataTypes klíčovým slovem using.

Koncept grafického rozhraní funkcionality pro porovnávání datových typů se nachází na obrázku 20. V samotné aplikaci Elvac Makro je pod záložkou *Datové typy*. Základem komponenty jsou dvě okna jiné komponenty *TreeView*[19]. První okno s názvem *Deflist - datové typy* (vlevo) obsahuje seznam datových typů načtených z Deflistu a druhé okno s názvem *PLC - struktura projektu* obsahuje zase datové typy načtené z prostředí TIA Portal. Obě okna jsou založena na stejné logice, liší se pouze v datovém zdroji. Nad těmito okny se dále nacházejí tlačítka pro aktualizování dat, pro označení všech načtených datových typů a také pro zrušení výběru, nad oknem *PLC - struktura projektu* je navíc tlačítko *Uložit projekt* sloužící k uložení načteného projektu v prostředí TIA Portal.

[19]Komponenta *TreeView* slouží k zobrazení stromových struktur.



Obrázek 20: Implementace grafického rozhraní funkcionality pro práci s datovými typy.

Tabulka 1: Stavy, které může datový typ nabýt po porovnání datových struktur.

barva	stav
zelená	shodný
žlutá	rozdílný
modrá	chybějící

Následující odstavce budou zaměřeny na samotnou komponentu *TreeView* a způsob, jakým jsou datové typy zobrazeny. Jak bylo řečeno výše, datové typy jsou uspořádány ve stromové struktuře. V komponentě představuje kořen stromu uzel reprezentující datovou strukturu (viz obrázek 20). Kořen této datové struktury obvykle obsahuje více potomků - *skupinové* datové typy. Tyto *skupinové* datové typy se dále skládají z *malých* datových typů. Jedná se tedy o stromovou strukturu o výšce 2.

Výše zmíněné názvosloví *skupinový* a *malý* datový typ bylo vytvořeno právě pro potřeby této práce, aby bylo možné rozlišit skupinový datový typ, který se obvykle skládá ze skupiny malých datových typů sts, cmd, sp, flt a msg.

Uprostřed grafického rozhraní se nacházejí tři tlačítka pro porovnávání (stočená šipka, nahoře), převedení z Deflistu do prostředí TIA Portal (šipka zleva doprava, uprostřed) a z prostředí TIA Portal do Deflistu (šipka zprava doleva, dole). Po porovnání struktur je barva kruhu u datových typů v obou strukturách *TreeView* změněna dle stavu datového typu.

6 Testování řešení

Testování produktu je stěžejní částí jeho vývoje, kdy je potvrzena správná funkce produktu. V případě, kdy produkt představuje hardware, je testování založeno na statistické analýze výsledků jedné úlohy která byla opakována dle potřeby testu (obvykle více než tisíckrát). Tento způsob testování je zvolen proto, že hardware představuje nedeterministický systém, u kterého není vždy stoprocentní jistota stejného výsledku při stejných vstupních parametrech. Důvodů, proč hardware nelze považovat za deterministický systém, je mnoho, jedny z nich jsou rušení, vliv vnějších vlivů a určité aproximace při tvorbě modelu reálného světa, které zjednodušují výpočty.

Software je brán jako deterministický systém, po přivedení určitých hodnot na vstup systému budou hodnoty výstupních vždy totožné. Tento systém se tedy nechová náhodně a odpadá zde potřeba statistické analýzy výsledků testů. I přesto je ovšem nutné testovat správnost implementace softwaru a jeho správné fungování.

Správnost implementace softwaru se testuje několika způsoby[6].

- Jednotkové testy - obvykle tvořené samotným programátorem. Určené k testování jednotlivých funkcí a metod. Založené na principu bílé skříňky, testovací kód tedy píšeme na základě implementace, kdy funkce nebo metoda při stejných parametrech vždy vrátí stejnou hodnotu. Tyto testy jsou velmi užitečné při případné změně implementace, kdy odhalí, zda tato změna nezasáhla fungování dalších funkcí či metod.

- Funkční testy - testují aplikaci jako celek, zejména správné fungování grafického rozhraní. Tyto testy jsou obvykle do určité rozsáhlosti aplikace prováděny ručně, je ovšem možné je automatizovat s využitím k tomu určených automatizačních nástrojů.

- Integrační testy - tvořené testovacím týmem, nikoliv programátorem samotným jako jednotkové testy. Integrační testy jsou určené k ověření správné komunikace jednotlivých komponent aplikace mezi sebou, případně komunikace komponent s operačním systémem. Tato fáze testování je stěžejní pro rozsáhlé projekty, ale u malých projektů je obvykle vynechána, protože při provedení předešlých testů je chyba odhalena těmito testy.

- Systémové testy - ověřují, že implementací nových funkcionalit nebyly změněny funkcionality již implementované. Obvykle se provádí při vytvoření nové verze apikace.

Při testování aplikace Elvac Makro byly, vzhledem k malému rozsahu aplikace, provedeny pouze jednotkové a funkční testy, přičemž jednotkové testy byly použity ke stejnému účelu jako testy systémové.

Jednotkové testy byly realizovány s použitím nástroje *UnitTesting*, který je nativní součástí Visual studia a nachází se ve jmenném prostoru *Microsoft.VisualStudio.TestTools.UnitTesting*. Pro účel jednotkových testů byl vytvořen nový projekt v řešení aplikace se jménem SharedTest a v tomto byla vytvořena třída pro testování implementované funkcionality pro porovnávání datových typů *PLCDataTypeParamTest*. Při tvorbě testovací třídy byla dodržena standardní

jmenná konvence testovací třída se jmenuje stejně jako třída, kterou testuje, a navíc má příponu *Test*. Funkční testy byly prováděny v průběhu vývoje aplikace ručně programátorem, ale také ve polupráci se zákazníkem.

7 Závěr

Podařilo se implementovat kýženou funkcionalitu pro operace s datovými typy pro prostředí TIA Portal a pro Deflist. Tímto bylo zároveň dosáhnuto sjednocení formátu dat z Deflistu a prostředí TIA Portál.

Tato funkcionalita zkrátí čas nutný pro převod datových typů mezi jednotlivými datovými zdroji (Deflist a prostředí TIA Portal) přibližně o faktor 1/5. Vzhledem k tomu, že tato činnost se provádí až 30x při práci na jednom projektu a trvá v průměru 10 minut, může být s využitím nové funkcionality dosáhnuto úspory až 4 hodiny práce na jednom projektu.

Další výhodou, kterou přináší implementovaná funkcionalita, je zjednodušení procesu vývoje řídicí aplikace, na vývoji se tedy může podílet i méně kvalifikovaný zaměstnanec, což ve výsledku může přinést další finanční i časové úspory.

Implementace funkcionality pro operace s datovými typy byla stěžejní částí projektu. Cílem vývoje aplikace Elvac Makro je vytvořit aplikaci, která bude schopna automaticky generovat kostru programu jak v prostředí TIA Portal, tak v prostředí IndraWorks a tímto ušetřit značné množství času potřebného pro ruční vytvoření kostry programu. Funkcionalita pro operace s datovými typy umožňuje automatizovat operace s dalšími prvky prostředí TIA Portal, jako jsou programové bloky, funkční bloky a kostra programu. Funkcionalita pro operace s programovými a funkčními bloky již byla implementována, na funkcionalitě pro generování kostry programu se stále pracuje.

Dosažené cíle bakalářské práce, ale také vytyčené cíle vývoje aplikace Elvac Makro se promítnou, v případě jejich úspěšné implementace, v nižších nákladech na vývoj průmyslového stroje, tímto se zlepší konkurenceschopnost firmy, ale také se zrychlí a zefektivní vývoj průmyslových strojů.

Literatura

[1] ARLOW, Jim a Ila NEUSTADT. *UML a unifikovaný proces vývoje aplikací: průvodce analýzou a návrhem objektově orientovaného softwaru.* Brno: Computer Press, 2003. ISBN 80-7226-947-X.

[2] BRADLEY, Neil. *XML kompletní průvodce.* Přeložil Jiří BRÁZDA. Praha: Grada, 2000. ISBN 80-7169-949-7.

[3] DANILEVIČIUS, Gediminas & EZERSKIS, Darius & BALAŠEVIČIUS, Leonas. *Data structures and interfaces for automated PLC applications development process* Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems, Prague, 2011, pp. 546-549. 10.1109/IDAACS.2011.6072826

[4] E-ICEBLUE. *Spire.XLS for .NET Quick Start* [online]. Chengdu, China [cit. 30.4.2019]. Dostupné z: `https://www.e-iceblue.com/Tutorials/Spire.XLS/Getting-started/Spire.XLS-Quick-Start.html`

[5] GAURA J. *Studijní opora k předmětu Skriptovací programovací jazyky a jejich aplikace* [online]. Ostrava [cit. 30.4.2019]. Dostupné z: `http://mrl.cs.vsb.cz/people/gaura/spja/skripta.pdf`

[6] GRAHAM, Dorothy & VAN VEENEDAAL, Erik & EVANS, Isabel. *Foundations of Software Testing: ISTQB Certification.* Boston: Cengage Learning EMEA, 2008. ISBN 1844809897

[7] JEFF, Martin (4 December 2018). "Microsoft Open Sources WPF, WinForms, and WinUI". InfoQ. Retrieved 2018-12-06.

[8] JOHNSON, Bruce. *Professional Visual Studio 2017.* Indianapolis: John Wiley & Sons, 2018. ISBN 978-1-119-40458-3.

[9] KIMMEL, Paul. *Advanced Csharp Programming.* New York: McGraw Hill Professional, 2002. ISBN 0072224177.

[10] KOZIOREK, Jiri & KONECNY, Jaromir & GAVLAS, Antonin & KRAUT, Radim & WALDER, Petr. (2018). Analysis of dataflows within industrial control system design. MATEC Web of Conferences. 210. 02030. 10.1051/matecconf/201821002030.

[11] NAGEL, Christian & GLYNN, Jay & SKINNER, Morgan. *Professional C# 5.0 and .NET 4.5.1.* Indianapolis, IN: John Wiley and Sons, 2014. ISBN 978-1-118-83294-3.

[12] OSHEVORE, Roy. *The Art of Unit Testing With Examples in .NET.* Greenwich: Manning Publications Co.,2009. ISBN 978-1933988276

[13] ROBINSON, Simon. *C#: programujeme profesionálně*. Brno: Computer Press, 2003. ISBN 80-251-0085-5.

[14] RITCHIE, Peter. *Practical Microsoft Visual Studio 2015*. 1. New York: Apress, [2016]. ISBN 1484223128.

[15] SELLS, Chris. *Moving from MFC, Windows Forms 2.0 Programming*. Boston: Addison-Wesley Professional. ISBN 0321116208

[16] SIEMENS, *SIMATIC Openness: Automating creation of projects System Manual* [online]. Mnichov: SIEMENS [cit. 30.4.2019].Dostupné z: `https://cache.industry.siemens.com/dl/files/163/109477163/att_926042/v1/TIAPortalOpennessenUS_en-US.pdf`

[17] SIEMENS, *The Component Object Model* [online]. Redmond: Microsoft, 1994 [cit. 30.4.2019].Dostupné z: `http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1994/9416/9416c/9416c.htm`

[18] TROELSEN, Andrew & JAPIKSE, Philip. *Pro C# 7: with .net and .net core*. 1. New York, NY: Springer Science+Business Media, 2017. ISBN 9781484230176.

[19] VIRIUS, Miroslav. *C# pro zelenáče*. 1. Praha: Neocortex, 2002. ISBN 80-86330-11-7.