Návrhová dokumentace – ConfigRW

Při tvorbě API pro knihovnu *ConfigRW* byl kladen důraz na co nejjednoduší použití na straně uživatele knihovny. Proto jsme se rozhodli pro minimalistické rozhraní, které vypadá následovně:

- Třída *Configuration* – slouží k vytváření konfiguračních objektů
- Interface *IConfiguration* – používaný pro zpřístupnění služeb na konfiguračních objektech
- Třída *QualifiedName* – slouží pro jednoznačné pojmenování elementů konfiguračního souboru
- Specifikační Attributy – umožňující rozšířené možnosti specifikace struktury konfiguračního souboru
- Výjimky – používané pro upozornění na chybové situace a špatná použití knihovny

Návrh třídy Configuration

Abychom mohli pracovat s konfiguračním souborem, musíme nejprve specifikovat jeho strukturu. Způsobů jak specifikaci provést je několik. Například zavést formální gramatiku pro popis optionů a sekcí, zadefinovat XML popis souboru, případně bychom mohli vytvořit builder jímž strukturu specifikujeme. My jsme se však rozhodli pro umožnění popisu konfigurační struktury pomocí .NET interface. Tím získáme možnost typové kontroly již za překladu. Samotné využití ve zdrojovém kódu navíc vypadá velmi intuitivně.

Motivační příklad:

Určení struktury přístupem „Builder“

```
var structureBuilder = new ConfigBuilder();

//Structure is hidden in code :-(
structureBuilder.AddOption<string>("Player1", "Name")
    .SetDefault("NoName").SetReadonly();
structureBuilder.AddOption<int>("Player1", "SavedLevel");

structureBuilder.AddOption<string>("Player2", "Name")
    .SetDefault("NoName").SetReadonly();
structureBuilder.AddOption<int>("Player2", "SavedLevel");
```

stejná struktura přes interface

```
public interface Players : IConfiguration
{
    Player Player1 { get; }
    Player Player2 { get; }
}

public interface Player
{
    [OptionInfo(DefaultValue = "NoName")]
    string Name { get; }
    int SavedLevel { get; set; }
}
```

Použití konfigurace, která není silně typovaná

```
var configuration = structureBuilder.Build("players.ini");

//we has to explicitly now what type which value hase
var savedLevel=configuration.ReadValue<int>("Player1", "SavedLevel");
++savedLevel;

//we need to explicitly write changes into configuration object
configuration.WriteValue<int>("Player1", "SavedLevel", savedLevel);

//...other work with configuration data...

//save results
configuration.SaveTo("changedConfig.ini");
```

typ je jednoznačně určen pomocí interface

```
var players = Configuration.CreateFromFile<Players>("players.ini");

//now we can work with players like with usuall .NET object
++players.Player1.SavedLevel;

//...other work with configuration data...

//we can save configuration easily
players.SaveTo("changedConfig.ini");
```

Jak je vidět z motivačního příkladu, naše knihovna po uživateli nevyžaduje žádné explicitní získávání nebo měnění hodnot konfigurace, ani jejich přetypování. Použití načtených hodnot je tedy velmi jednoduché a nepřidává zbytečnou zavlečenou složitost.

Způsoby vytváření konfiguračních objektů:

Naše knihovna nabízí uživateli možnost vytvořit konfigurační objekt kromě standardních způsobů jako je ze souboru nebo streamu také z defaultních hodnot. Pro tuto možnost jsme se rozhodli, neboť uživatel může do struktury popisující konfigurační soubor jednoduše specifikovat defaultní hodnoty a komentáře k jednotlivým optionům. Jejich vypsáním do souboru pak získá předpis konfigurace aniž by ji musel například vytvářet ručně.

Návrh interface IConfiguration

Popis struktury konfiguračního souboru pro *ConfigRW* knihovnu musí být potomkem interface *IConfiguration*. Pro tuto povinnost jsme se rozhodli kvůli snadnému zpřístupnění služeb na konfiguračním objektu. Kdybychom totiž chtěli jiným způsobem zpřístupnit třeba ukládání konfiguračních objektů, vypadalo by to nějak takto:

```
Configuration.Save(configObj, "output.ini");
```

Problémem je, že pokud by *configObj* nebyl oděděn od nějaké společné třídy nebo interface, musela by metoda *Save* přijímat objekty typu *System.Object*. Tím bychom však ztratily možnost ubránit uživatele špatnému použití knihovny typovou kontrolou. Proto jsme se rozhodli požadovat po struktuře konfiguračního souboru rozhraní typu *IConfiguration*. Použití konfiguračního objektu v kódu se tím také zpřehlední:

```
configObj.SaveTo("output.ini");
```

Návrh třídy QualifiedName

Pro pojmenování elementů konfiguračního souboru jsou používány objekty typu *QualifiedName*. Díky tomu, že *QualifiedName* ukrývá implementační detaily struktury jmen, umožňuje budoucí rozšiřitelnost knihovny (například o podsekce, či jinou hierarchii elementů). Navíc je nutnost udávání *QualifiedName* do metody *IConfiguration.SetComment* určitou ochranou proti špatnému použití. Pokud bychom použili metodu se signaturou obsahující například dva argumenty typu *string*, může se snadno stát, že zaměníme jejich pořadí. Z těchto důvodů jsme zvolili *QualifiedName* pro reprezentaci jména elementu konfiguračního souboru.

Návrh specifikačních atributů

Funkční požadavky na formát konfiguračního souboru přesahují možnosti, které nám dávají samotné .NET interface. Určitá možnost, jak například specifikovat defaultní hodnoty, omezení rozsahů,... by byla v dodání patřičných objektů do metody při vytváření konfiguračního souboru. Tím bychom ale rozdělili popis struktury do více míst v kódu. Proto jsme zvolili pro C# typičtější přístup a definovali jsme atributy, které mohou dekorovat jednotlivé sekce a optiony přímo v interface určujícím strukturu. Na tento přístup jsou programátoři C# zvyklí. Výsledný kód je navíc výrazně přehlednější.

Možnost určení rozsahů, defaultních hodnot a komentářů je celkem přímočará, avšak kvůli omezení na jména memberů používaných v C# jsme museli umožnit pomocí atributu nastavit i jméno identifikátoru optionu či sekce. Funkční požadavky totiž vyžadují možnost netradičních identifikátorů. Naproti tomu, běžné použití takové identifikátory využívat nebude, proto je defaultní název identifikátoru odvozen od názvu property pro sekci či option. Pokud by tento název nevyhovoval, můžeme určit ID pomocí atributu následovně:

```
//Set special ID for section
[SectionInfo(ID = "Special $ - name")]
Section1 Sec1 { get; }
```

V běžných případech tak nemusíme psát žádný kód navíc. Přesto však ve zvláštních případech můžeme určit identifikátor s plnou syntaxí dle funkčních požadavků.

Návrh výjimek

Pro snadné ošetřování výjimek jsou všechny výjimky vyhazované naší knihovnou potomky *ConfigRWException*. To umožní oddělit výjimky vzniklé v uživatelském kódu od výjimek z naší knihovny. Dále jsou odděleny výjimky vzniklé během parsování tím, že dědí od *ParserException*. Ostatní výjimky se týkají validace typu struktury popisující konfigurační soubor a její použití.

Každá výjimka obsahuje zjednodušenou uživatelskou zprávu a pak zprávu pro vývojáře. Toto rozdělení umožní detailní pohled na vzniklý problém a zároveň umožní srozumitelnou formou informovat například uživatele aplikace. Některé výjimky navíc obsahují membery s informacemi týkající se místa chyby kde vznikla (číslo chybné řádky parsovaného souboru, property u které byl nalezen neplatný rozsah..). Tím dává knihovna uživateli možnost tvorby vlastních detailních chybových hlášení nebo automatizovaných oprav chyb.

Omezená funkčnost pro Mono

Protože runtime Mono v některých verzích neobsahuje implementaci několika využívaných funkcí ze *System.Reflection*, nemusí být v tomto běhovém prostředí knihovna použitelná. A to i přesto, že se naše implementace striktně drží standardů .NET 3.5. Do budoucna však očekáváme lepší pokrytí .NET frameworku v Mono. Bylo nám proto učitelem cvičení předmětu Doporučených postupů v programování sděleno, že současná nekompatibilita s tímto běhovým prostředím není v rozporu se zadáním úkolu. Knihovna pomocí uvedeného Makefile totiž lze bez problémů v Mono zkompilovat. Výsledný binární soubor je pak v .NET frameworku funkční.