Celkové zhodnocení

Na úvod malý disclaimer: stejná práce byla pisateli tohoto review předána i v rámci úkolu #3, takže se zde budeme částečně opírat o předchozí zkušenosti.

Zdůvodnění návrhu

Vysvětlení struktury a návrhu řešení se oproti úkolu #3 rozhodně zlepšilo a nyní z něj již jsem schopen vyčíst významné skutečností, než se podívám na kód samotný. Celkovému dojmu však výrazně pomáhá, že jsem se s prací autorů již setkal a nemusím tedy řešení objevovat od nuly. I přes moje připomínky však zůstávají některé krkolomné fráze, resp. jisté záměry/výroky, kterým buďto nejsem schopen porozumět nebo s nimi nemohu souhlasit. Některé začnou dávat smysl až po dlouhém uvažování (případně review samotného kódu), jiné jsou tak nesouvisející či samozřejmé, až mě přepadá dojem, že mají prostě jen zabírat místo. Například vůbec nechápu naznačovaný kompromis v tomto výroku:

```
Celý projekt jsme se rozhodli rozdělit do 3 souborů jako kompromis
mezi snadným nasazením do jiných projektů a rozšiřitelností jednotlivých
částí práce s konfiguracemi.
```

Snad každá dobrá a užívaná knihovna by měla být nasazována do projektů prostřednictvím package managerů, resp. zbuildovaných DLL knihoven, nikoli manuálně na bázi jednotlivých zdrojových souborů…

Přestože se zdůvodnění nesporně zlepšilo, není úplné, neboť podává víceméně jen velmi základní strukturální přehled, který by měl být snadno patrný už ze samotného kódu a komentářů (případně z jednoduchého diagramu hierarchie typů). Chybí zde zmínky o mnoha detailech (návrhových a implementačních rozhodnutích), díky čemuž je už předem poměrně jasné, že:

- Implementace bude sestávat z nemála nedotažených částí.
- Při review mě čeká další martyrium, podobné tomu v úkolu #3.

Zdůvodnění deklaruje rozšiřitelnost jako vůbec nejvýraznější princip užitý při vývoji. Na jednu stranu je třeba chválit snahu o všeobecné pokrytí tříd rozhraními. Na stranu druhou ale implementace v oblasti rozšiřitelnosti až nečekaně moc selhává. Několik takových problémů bude později v review vypíchnuto.

Okamžitě jsem byl zklamán triviálním prohlášením, že operace načtení konfigurace končí nalezením jednoho jediného problému a vyhozením příslušné výjimky. Přitom jsem naposledy připomínkoval, že ve zdůvodnění chybí zmínka a vysvětlení této části návrhu, a dokonce v duchu zadání nabídl k zamyšlení více alternativ. Jednotlivé problémy rozhodně LZE uživatelům předávat v reálném čase a bez nutnosti definice spousty specializovaných typů. Když už nic jiného, alespoň lze ty nejzávažnější problémy vypisovat na konzoli, nebo je sbírat v kolekcích a po skončení operace předat uživateli. Spousta dílčích problémů je naprosto zotavitelná, a autoři to ignorují.

S naposledy řečeným se pojí i tato zvláštnost: `IOtherItem: datová položka vytvořená pro nevalidní řádky při čtení v relax módu`. Jednak nezní zrovna “rozšiřitelně” (co striktní mód?) a jednak je slovo “nevalidní” zvoleno naprosto špatně. Správně by tam mělo být: “řádky s neznámou syntaxí”.

Dále se ve zdůvodnění vůbec nepíše, že konfigurace jsou při striktním módu validovány ihned po načtení (vyčteno z implementace). Proč? Mně osobně připadá, že zadání nic takového ani nenaznačuje, neřkuli, že se to zrovna moc neslučuje s odkazy, o nichž ostatně zdůvodnění možná až úzkostlivě mlčí. Později uvidíme proč.

Rovněž mě velice mrzí, že se ve zdůvodnění nevyskytuje ani slovo o formátu pro definice konfigurací.

Knihovna

Implementace nesplňuje zadání v následujících bodech:

1. Knihovna neřeší escapované bílé znaky na konci identifikátorů nebo voleb.
2. Objektová reprezentace voleb obsahuje jen jednu hodnotu (chyba již v samotném návrhu). Oddělovače (čárka nebo dvojtečka) jsou tedy naprosto ignorovány a v důsledku toho nefunguje správně ani escapování metaznaků hodnot (`,`, `:` a `;`).
3. Do struktury konfigurace je sice začleněno něco s názvem “odkaz”, ale jedná se o naprosté nedochůdče a v praxi nejsou odkazy stejně nikdy parsovány.
4. Celočíselné elementy pracují pouze s desítkovou soustavou (vstup i výstup).
5. Třída pro konfiguraci sice obsahuje neprázdnou metodu `Check()` (validace), nicméně jako by prázdná byla. Knihovna zkrátka validaci vůbec neimplementuje.
6. Knihovna sice definuje jakýsi formát pro definici konfigurace (viz později), ale nespecifikuje popisy sekcí/voleb a omezení na celočíselné hodnoty (intervaly).
7. Knihovna selhává v co největším zachovávání původní podoby konfigurací.
8. Knihovna neumožňuje vytvářet defaultní konfigurace z definic.

Z hlediska návrhu je základní použití sice OK, ale ďábel se skrývá v chybějící funkcionalitě, detailech a implementaci. Příklady konkrétních problémů lze nalézt dále v dokumentu. V některých ohledech knihovna použitelná rozhodně není a celkově bych ji k užití ani trochu nedoporučil. Když pominu nesplnění zadání a poměrně časté implementační chyby, některé části návrhu na mě působí značně nepromyšleně či nedomyšleně a silně mě při review rozčilovaly, neboť jsem je autorům v rámci úkolu #3 hojně připomínkoval. Návrh konzistencí se zadáním zrovna neoplývá a občas se mi dokonce jeví nekonzistentní vůči sobě samému. Podobně mě velmi zklamalo, jak špatně je na tom implementace knihovny s rozšiřitelností, kterou ironicky deklaruje jako hlavní princip návrhu a vývoje.

Podobně jako v úkolu #3, i nyní si nejsem jistý, co si o autorech mám myslet. Buďto trpí nedostatkem dovedností a zkušeností při vývoji SW, neměli dostatek času či se prostě moc nesnažili. Osobně se nemohu zbavit dojmu, že si svou roli zahrály všechny tři faktory.

Implementace

Pokrytí testy

Je definováno sedm testovacích vstupů, z nichž:

- `BigConfig.ini` je prázdný, navzdory specializovanému vstupu `Empty.ini`.
- Čtyři obsahují ukázku ze stránek předmětu, případně trošičku upravenou, a prakticky se neliší. Například vstup `MultiSectionsWithReferences.ini` hravě strčí do kapsy vstupy `SimpleReference.ini` a `SingleSection.ini`.

- Vstup `SingleSectionWithComments.ini` obsahuje zakázené duplicitní sekce a pokus o načtení vstupu by tedy měl skončit s chybou, což však název vstupu vůbec nenaznačuje. Byly vstupy vůbec testovány?

Dále tu máme dva soubory s testy:

- `UnitTests.cs`
- `IntegrationTests.cs`

Všechny testy de-facto prázdné (nic netestují). Ve zkratce k žádnému testování reálně nedošlo, navzdory tomu co tvrdí zdůvodnění. Navíc nemáme k dispozici ani jedinou testovací definici pro konfigurace.

Kvalita a styl kódu

Jednoduše řečeno: kvalita mizivá a styl špatný. Lze tam nalézt nebývale mnoho prohřešků vůči DPP a kdybych je měl všechny vypisovat, cvičící by čekalo pořádné počtení. Jako příklad nyní uvedu několik návrhových prohřešků (implementační ponechám druhé části dokumentu):

Regiony
Přestože je veškerý kód rozdělen pouze do tří souborů, marně hledám regiony pro snadnější orientaci a analýzu.

Komentáře
Oproti úkolu #3 jsou na trochu vyšší úrovni, ale:

- Komentáře chybí až příliš často, nejen pro metody a třídy, ale i pro kód samotný.
- Když už knihovna nějaký ten komentář obsahuje, z obsahové stránky se jedná o holé minimum, ze kterého člověk nic významného nevyčte.

Komentáře lze tedy zcela ignorovat.

Duplikátní fieldy
Nechápu, co dělají v C# konstrukty typu:

```
public bool IsStrict
{
        get
        {
                return _isStrict;
        }
}
private readonly bool _isStrict;
```

neboť jsou de-facto ekvivalentní implicitním getterům a setterům:

```
public bool IsStrict { get; private set; }
```

Private metody a fieldy
Když se autoři ve zdůvodnění tolik dušují rozšiřitelností, očekával bych, že metody či fieldy budou `private` jen ve zcela odůvodněných případech (naopak jsou úplně běžné). Podobně není prakticky žádná metoda označena slovem `virtual`, takže je ani nejde v rámci dědičnosti přepsat.

Vyhazování výjimek v konstruktorech
Přestože jsem to připomínkoval, nic se nezměnilo. Mnohdy by se tomu dalo snadno vyhnout odlišným návrhem.

Vždy chybějící volání string.format()
Jako příklad uvádím:

```
throw new ConfigException("Unexcepted line – '{0}'", line);
```

Angličtina na podprůměrné úrovni
Z jednoho bonbónku je například poněkud obtížné s jistotou usoudit, na kolika prvcích validace konfigurace selhala:

```
if (!config.Check())
{
        throw new ConfigException("All items of configuration is not valid.");
}
```

Všimněte si, kolik chyb a nedostatků se na těchto dvou řádcích nachází…

TODO:
Také zhodnoťte, jak snadné nebo těžké bylo knihovnu rozšířit, tj. jak byla knihovna připravena na budoucí rozšíření.

Hlubší analýza chyb a nedostatků řešení

Přestože to zdůvodnění nezmiňuje, řešení je rozděleno do tří projektů:

1. Ukázka práce s knihovnou.
2. Knihovna sama.
3. Testování knihovny.

Třetí už jsme prozkoumali, takže se postupně se vyjádřím k prvním dvěma.

Ukázková práce s knihovnou

Jedná se o hezký nápad s potenciálem urychlit mi práci. Projekt sestává z jednoho zdrojového souboru, který nicméně nejde spustit, neboť neexistují vytyčené soubory `config.ini` a `definition_file.def`. Ukázkové vstupy se vyskytují až v testovacím projektu, takže by bylo dobré takovou skutečnost alespoň zmínit ve zdůvodnění, případně rovnou společně s existencí tohoto projektu.

Mám důvodné podezření si myslet, že následující úryvky ze zdůvodnění a ukázkové práce s knihovnou nejsou kompatibilní:

> Relax mód umožňuje plnohodnotnou práci s konfiguračním souborem bez nutnosti využívat výše zmíněné definice podoby.

```
static void RelaxExample(string configFilePath, string definitionFilePath)
{
```

```
        var defFile = new IOConfigDefinition(definitionFilePath);

        IConfigDefinition definition = defFile.Load();

        var configFile = new IniIOConfig(configFilePath, definition, false);

        IConfig config = configFile.Load();

        var item = config["section 1"]["id1"];
        if (item is DoubleValueItem)
        {
                (item as DoubleValueItem).SetValue(5.0);
        }

        configFile.Save(config);
}
```

Očekával bych, že se zde vůbec nebude načítat definice a konstruktoru `IniIOConfig` bude místo ní předán `null`.

Knihovna

Už jen rychlým čtením kódu jsem schopen odhalit mnoho chyb či potenciálních problémů, včetně téměř neutuchajících provinění vůči best-practices, které nám mají být v rámci tohoto předmětu vštěpovány.

Zbytečně složitý a nikterak snadno udržitelný kód, který lze poměrně jednoduše zlepšit. Hojně se vyskytuje duplicitní kód a dokonce se najde i kód, který se nikdy nespustí či nemá význam.

Obšírná analýza kódu by byla všestranným plýtváním času, a v dalším textu se proto pokusím zaměřit hlavně na závažné problémy návrhu a implementace. Nemám pochyb, že i tak jich bude dost.

Soubor `IniConfig.cs` (IO operace)

Třída `IniIOConfig`

Přestože jsem to připomínkoval, třída `FileIOConfig` i nadále dědí od rozhraní `IDisposable`, jeho hlavní metoda `Dispose()` zůstává prázdná a příčina opět není k nalezení.

O rozhodnutí validovat konfigurace ihned po načtení už jsem mluvil.

Nechápu, proč není metoda `LoadRegex()` eliminována ve prospěch konstruktorů.

Jak jsem se obával už od přečtení zdůvodnění, prázdné řádky (členění) nejsou v konfiguracích zachovávány.

Knihovna při parsování neodstraňuje bílé znaky u identifikátorů a už vůbec neřeší escapované bílé znaky (jak nám přikazuje zadání).

Ukázka toliko oslavované “rozšiřitelnosti”, “modularity” a dělení na rozhraní:

```
private void Save(IConfigItem item, TextWriter writer)
{
        if (item == null)
        {
                return;
```

```
        }
        if (item is ISection)
        {
                Save(item as ISection, writer);
        }
        else if (item is IValueItem)
        {
                Save(item as IValueItem, writer);
        }
        else if (item is IComment)
        {
                Save(item as IComment, writer);
        }
        else if (item is IOtherItem)
        {
                Save(item as IOtherItem, writer);
        }
        else
        {
                throw new ConfigException(item, "Unknown type of item for save.");
        }
}
```

Jednoduchým pozorováním regulárních výrazů pro sekce a volby jsem schopen vymyslet zcela validní konfiguraci o jednom či dvou řádcích, která parserem autorů neprojde. Studenta MFF přece musí napadnout, že neexistuje jen jeden bílý znak. A to ani nemluvím o tom, že literál pomlčky není uvnitř hranatých závorek escapován (resp. správně použit jako metaznak), takže při vytváření instance třídy `Regex` musí dojít k chybě. Dokonce již před vlastním spuštěním knihovny tedy vím, že nikdy spuštěna nebyla.

Třída `IOConfigDefinition`

Toto je jediné místo, odkud lze alespoň trochu vyčíst textový formát definic. Pro úplnost si vyhotovím vlastní příklad:

```
SECTION
identifikátor1
REQUIRED
// neprázdné řádky s komentářem
// ať si tu ale napíšu co chci, parser to ignoruje a nijak neukládá

VALUE
identifikátor2
REQUIRED
// tady má být typ hodnoty; v kódu je to prostě string, takže nevím jak má vypadat
SINGLE_VALUE
// tady mohou být jakési "parametry" (jeden string na jeden řádek)

VALUE
identifikátor3
NOT_REQUIRED
// tady má být typ hodnoty; v kódu je to prostě string, takže nevím jak má vypadat
MULTI_VALUE
// tady mohou být jakési "parametry" (jeden string na jeden řádek)

SECTION
identifikátor4
NOT_REQUIRED
// neprázdné řádky s komentářem
// ať si tu ale napíšu co chci, parser to ignoruje a nijak neukládá
```

```
// a tak dále
```

Obávám se, že zbytek formátu budu muset luštit v jiných částech projektu. To be continued.

Soubor `IConfig.cs` (rozhraní pro definice a konfigurace)

V úkolu #3 jsem připomínkoval mapování prvků definic a konfigurací v poměru 1:1. Nic se nezměnilo. Uživatel může i nadále upravovat obě struktury nezávisle na sobě. Například tedy může odstranit definici sekce, a ona přitom bude i nadále spárovaná s živoucí instancí sekce v konfiguraci. Následná validace konfigurace tedy vůbec nepomůže a v horším případě dokonce úspěšně projde. Jak vidno, knihovna validuje podle instancí definic spárovaných při parsování (mám silné podezření, že i nadále budou v implementaci read-only), nikoli podle aktuální podoby definice jako takové. Autoři nenabízejí žádné nastavení udržování mapování nebo jiné možnosti. Serializace obou struktur je navíc zcela nezávislá a s validací nemá vůbec nic společného. Jinými slovy, autoři nechávají uživatele naprosto ve štychu ohledně ošetření nekonzistencí a validace konfigurace před zapisováním.

Přestože jsem to připomínkoval, zůstal tu pokus o diktaturu zachovávání pořadí prvků konfigurace v podobě metod pracujících s obsahem na základě indexu. Přitom knihovna dané metody nikdy nevyužívá, a jsou tedy zbytečné.

Rozhraní `IValueDefinition<T>`

Přestože jsem to připomínkoval, definuje pouze jednu výchozí hodnotu.

Rozhraní `IValueDefinition`

Hmm, “parametry” (viz ukázka definice výše)… Tváří se jako výchozí hodnoty předávané implementacím rozhraní `IValue`, ale `IValue` přitom obsahuje jen jednu hodnotu… Vážně by mě zajímalo, co z toho nakonec vyleze.

Ani zde se z komentářů nedozvídám, jak má v definici vypadat typ hodnoty. Zřejmě se budu muset podívat až do implementace a nějak ho vyluštit.

Rozhraní `IConfig`

Jsou tu dva indexery: jeden vysloveně pro sekce a druhý pro všechny prvky obsahu. V implementaci bych zdvojený indexer ještě pochopil, ale v rozhraní nikoli.

Přestože jsem to připomínkoval, zůstalo pojmenování metody `Check()`.

Rozhraní `IConfigItem`

Když už se u takto vysoce postaveného rozhraní vyskytuje metoda `IsValid()`, nedává mi moc smysl, aby neobsahovala také identifikátor. Ano, musely by ho obsahovat i komentáře, ale za stávajících okolností budou obsahovat i příznak určující jejich validitu, a ten se nikdy nevymaní ze statického světa.

Díky tomu dostává implementace pomocí slovníků nepříjemnou podpásovku a bod dolů ve sloupečku s rozšiřitelností. Navíc jsem tento konkrétní problém, myslím, přímo či nepřímo naznačoval už v úkolu #3.

Rozhraní `ISection`

Nechápu metodu `SetComment(IComment)`, resp. proč není eliminována ve prospěch Property:

```
IComment Comment { get; }
```

Rozhraní `IValueItem`

Stejná poznámka jako u předchozího rozhraní `ISection`.

Přestože jsem to velmi rozhořčeně komentoval, i nadále zůstává šílenství v podobě:

```
/// <summary>
/// Whether item has defined value.
/// </summary>
bool HasValue { get; }

/// <summary>
/// Remove value from item.
/// </summary>
void RemoveValue();
```

Rozhraní `IComment`

Nechápu přínos Property:

```
/// <summary>
/// Whether is comment empty.
/// </summary>
bool IsEmpty { get; }
/// <summary>
/// Whether is comment read only.
/// </summary>
bool IsReadOnly { get; }
```

První se MOŽNÁ hodí jen pro serializaci a velmi by mě zajímalo, k čemu by kdo potřeboval tu druhou…

Rozhraní `IIOConfig` a `IIOConfigDefinition`

Jak jsem již komentoval v tomto dokumentu a připomínkoval i dříve, naprosto zbytečná a nenaplněná dědičnost od `IDisposable`.

Rozhraní `IReferenceValue`

Rozhraní pro odkazy, které od ničeho nedědí a může být do datových struktur konfigurace zařazováno jen jako obsah obálky (zbytečné instance hodnoty příslušného typu, např. `BoolValueItem`). Rozhodnutí podporovat u voleb jen jednu hodnotu a ještě do stejné instance cpát odkazy a definice se vymyká mému chápání, především z perspektivy rozšiřitelnosti.

Rozhraní opět deklaruje jen jedinou hodnotu, nedeklaruje její původ a je velice hubené. Navzdory mým připomínkám…

Jak již víme, knihovna odkazy nijak neparsuje a tedy ani nezachovává.

Soubor `Config.cs` (vzhůru do implementace)

Počet řádků: 1310. Počet řádků s komentáři: 34 (reálný obsah po vydělení třemi). Palec dolů, a ještě ho zadupat do země.

V zájmu naší příčetnosti budu již popsané problémy znovu vytahovat na povrch jen minimálně a zaměřím se na problémy nové, tentokráte i implementační. Každý nový problém bude agregován pouze do svého prvního výskytu.

Přes čirou velikost implementace zvoní už první řádky na červený poplach.

Třída `Config`

Zaprvé:

```
// hmm, nestačil by náhodou jeden return?
ISection section;
if (_sections.TryGetValue(sectionId, out section))
{
        return section;
}
return null;
```

Zadruhé:

```
// přestože rozhraní je bez setteru...
public IComment Comment { get; set; }
```

Zatřetí:

```
// no jistě... směle pokračujeme v čertovské tradici read-only definic
private readonly IConfigDefinition _definition;
```

Začtvrté:

```
// nešlo by to náhodou snadno smrsknout do jedné datové struktury?
private List<IConfigItem> _items = new List<IConfigItem>();
private Dictionary<string, ISection> _sections = new Dictionary<string, ISection>();
```

Inu, šlo. To by si ale autoři museli dát tu práci a:

1. Odhalit existenci `System.Collections.Specialized.OrderedDictionary`.
2. Návrh by nesměl být tak necitlivý vůči implementaci, musel by se řídit některými mými radami z minula a hlavně by se musel snažit eliminovat problémy, které tu teď musím porůznu opakovat.

Zapáté:

```
// parser ukládá řádky s neznámou syntaxí jako instance třídy 'OtherItem'
public void Add(IConfigItem item)
```

```
{
        _items.Add(item);

        // co to... přidávání ne-sekcí končí výjimkou!
        var section = item as ISection;
        if (section != null && !_sections.ContainsKey(section.ID))
        {
                _sections.Add(section.ID, section);
        }
}
```

Zašesté, validace konfigurace vůbec nevaliduje sekce, resp. volby.

Nedošli jsme ani na řádek 151.

Třída `Section`

Zasedmé:

```
// není to při read-only definicích nějaké zbytečně složité?
public string ID
{
        get
        {
                if (Definition != null)
                {
                        return Definition.ID;
                }
                return _id;
        }
}
```

Zaosmé:

```
public bool IsValid
{
        get
        {
                // implicitně validní sekce... NICE!
                // setkali se již autoři s NotImplementedException?
                // vyjde to totiž nastejno :)
                return true;
        }
}
```

Zadeváté:

```
public void Add(IConfigItem item)
{
        _items.Add(item);
        var valueItem = item as IValueItem;

        // jak říkává můj táta: když si čert uprdne, je to vždycky Tichan Záludný!
        if (valueItem != null && !_valueItems.ContainsKey(valueItem.ID))
        {
```

```
                _valueItems.Add(valueItem.ID, valueItem);
        }
}
```

Nedošli jsme ani na řádek 317.

Třída `ValueItem<T>`

Zadesáté:

```
// Pozor! Pozor! Svatá Trojice přichází na Zemi!

public readonly IValueDefinition<T> TypedDefinition;
protected IReferenceValue<T> RefValue { get; private set; }
private T _value;

public void SetValue(T value)
{
        HasValue = true;
        _value = value;
        RefValue = null;
}
public void SetValue(IReferenceValue<T> reference)
{
        if (reference == null)
        {
                throw new ArgumentNullException("reference");
        }
        HasValue = true;
        _value = default(T);
        RefValue = reference;
}
public void RemoveValue()
{
        HasValue = false;
        _value = default(T);
        RefValue = null;
}
public virtual string GetStringValue()
{
        var value = RefValue != null ? RefValue.Value : Value;
        if (TypedDefinition != null)
        {
                return TypedDefinition.ConvertValueToString(value);
        }

        // zajímalo by mě, jak se na to budou tvářit float hodnoty :)
        return value.ToString();
}
public bool IsValid
{
        get
        {
                // případ pro Sherlocka Holmese nebo doktora Chocholouška?
                return HasValue || !Definition.IsRequired;
        }
}
```

Nedošli jsme ani na řádek 532.

Třída `EnumValueItem`

Zajedenácté:

```
public override string GetStringValue()
{
        // ale, ale... odkazy nás už nebaví?
        return Value;
}
```

Nedošli jsme ani na řádek 725.

Třída `ConfigDefinition`

Zadvanácté, tahle “třída” prakticky jen všechno deleguje na vnitřní slovník. Stačilo by od něj podědit, všechno zbylé vymazat, vyskočit zpátky na pec a pokračovat ve válení.

Nedošli jsme ani na řádek 851.

Třída `ValueDefinition`

Zatřinácté, konečně se dovídáme, že knihovna v definici konfigurace očekává na pozici typu hodnoty prostě nějaký hardkódovaný string. Možnosti úpravy? Nulové. Tři body dolů v kolonce s rozšiřitelností:

```
public static bool TryParse(string typeId, string id, bool isRequired, bool isSingleValue, IList<st
{
        switch (typeId)
        {
                case BoolValueDefinition.TYPE_ID:
                        valueDefinition = new BoolValueDefinition(id, parameters);
                        break;

                case Int64ValueDefinition.TYPE_ID:
                        valueDefinition = new Int64ValueDefinition(id, parameters);
                        break;

                case UInt64ValueDefinition.TYPE_ID:
                        valueDefinition = new UInt64ValueDefinition(id, parameters);
                        break;

                case DoubleValueDefinition.TYPE_ID:
                        valueDefinition = new DoubleValueDefinition(id, parameters);
                        break;

                case StringValueDefinition.TYPE_ID:
                        valueDefinition = new StringValueDefinition(id, parameters);
                        break;

                case EnumValueDefinition.TYPE_ID:
                        valueDefinition = new EnumValueDefinition(id, parameters);
                        break;

                default:
                        valueDefinition = null;
                        return false;
        }
```

```
        valueDefinition.IsRequired = isRequired;
        valueDefinition.IsSingleValue = isSingleValue;
        return true;
}
```

Začtrnácté, právě uvedená metoda je “úžasně” doplňována touto:

```
public abstract bool TryParse(string value, out IValueItem item);
```

Nedošli jsme ani na řádek 929.

Třída `BoolValueDefinition`

Zapatnácté, parametry z ukázkové definice konfigurace najednou nabývají významu a přecházejí nejen ve výchozí hodnoty, ale také výchozí `true` a `false` stringy! Na obecné rovině je nápad třeba chválit, ale jeho provedení je mimo moje chápání:

```
private readonly bool _defaultValue;
private readonly string _valueFalse;
private readonly string _valueTrue;

public BoolValueDefinition(string id, IList<string> parameters)
        : base(id)
{
        if (parameters == null)
        {
                throw new ArgumentNullException("parameters");
        }
        _defaultValue = (parameters.Count >= 1 && parameters[0] == "true") ? true : false;
        _valueTrue = parameters.Count >= 2 ? parameters[1] : DEFAULT_TRUE;
        _valueFalse = parameters.Count >= 3 ? parameters[2] : DEFAULT_FALSE;
}
```

Zašestnácté, že by po předchozím hezkém nápadu ohledně rozšiřitelnosti konečně svítalo na lepší časy? Ani omylem… je třeba se ihned vrátit králičí norou zpět do světa hardkódovaného šílenství:

```
public override bool TryParse(string value, out IValueItem item)
{
        bool bValue;

        if (value == _valueTrue)
        {
                bValue = true;
        }
        else if (value == _valueFalse)
        {
                bValue = false;
        }
        else
        {
                switch (value)
                {
                        case "1":
                        case "t":
```

```
                        case "y":
                        case "on":
                        case "yes":
                        case "enabled":
                                bValue = true;
                                break;

                        case "0":
                        case "f":
                        case "n":
                        case "off":
                        case "no":
                        case "disabled":
                                bValue = false;
                                break;

                        default:
                                item = null;
                                return false;
                }
        }

        item = new BoolValueItem(this);
        (item as BoolValueItem).SetValue(bValue);
        return true;
}
```

Už raději ani nestrhávám body v kolonce s rozšiřitelností.

Nedošli jsme ani na řádek 1023.

Třída `DoubleValueDefinition`

Zasedmnácté:

```
// jednou správně a jednou špatně... nechápu
public override IEnumerable<string> GetParameters()
{
        return new string[] { _defaultValue.ToString(CultureInfo.InvariantCulture) };
}
public string ConvertValueToString(double value)
{
        return value.ToString();
}
```