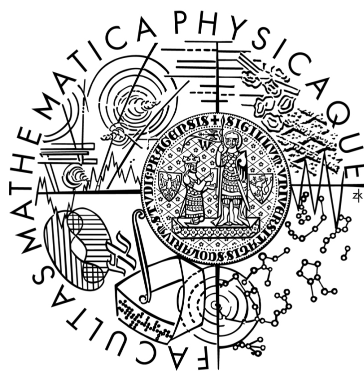
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jan Mareš

Podpora „Edit and Continue“ v prostředí SharpDevelop

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek

Studijní program: Informatika

Studijní obor: Programování

Praha 2012

Poděkování:

Rád bych poděkoval především vedoucímu bakalářské práce Mgr. Pavlu Ježkovi za trpělivost a čas, který mně a této práci věnoval. Dále bych chtěl poděkovat Davidu Srbeckému, vývojáři ladicího modulu pro SharpDevelop, který mi pomohl s integrací projektu do prostředí SharpDevelop, a Mgr. Michalu Kosovi, který mi pomohl s korekturou práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …...... dne............ podpis

Název práce: **Podpora „Edit and Continue“ v prostředí SharpDevelop**
Autor: **Jan Mareš**

Katedra / Ústav: **Katedra distribuovaných a spolehlivých systémů**

Vedoucí bakalářské práce: **Mgr. Pavel Ježek**

Abstrakt:

Vývoj velkých projektů vyžaduje ladicí nástroje, které usnadní vývojáři práci při hledání chyb. Pokud vývojář odhalí chybu v kódu během jeho krokování, je nejbezpečnější chybu opravit co nejdříve. Funkcionalita Edit and Continue je rozšíření ladicího nástroje pro .NET, která umožňuje opravit chybu bez nutnosti přerušit ladění aplikace. Tato práce se zabývá implementací této funkcionality pro nekomerční vývojové prostředí SharpDevelop.

Klíčová slova: **ladění, .NET, Edit and Continue, SharpDevelop**

Title: **“Edit and Continue” for the SharpDevelop IDE**

Author: **Jan Mares**

Department: **Department of Distributed and Dependable Systems**

Supervisor: **Mgr. Pavel Jezek**

Abstract:

Development of large projects requires usage of debugging tools, which makes developer‘s work – searching for bugs in code – easier. When developer discovers a bug in source code during debugging of application, the safest way is to fix this bug as soon as possible. The Edit and Continue functionality is a feature of .NET debugger that makes possible correction of source code of debugged application without the need of interrupting the debugging of application. The main concern of this thesis is to find a way to implement Edit and Continue for an OpenSource IDE SharpDevelop.

Keywords: **debugger, .NET, Edit and Continue, SharpDevelop**

Obsah

1.Úvod 3

1.1.EnC a .NET 4

1.2.Co je to CLI, CIL ? 5

1.3.Co je to SharpDevelop ? 6

1.4.Cíle práce 6

2.Analýza 7

2.1.Úvod 7

2.2.Problém vytvoření delta Metadat a delta IL kódu 9

2.2.1.Generování IL kódu 11

2.2.2.Generování Metadat 11

2.3.Rekonstrukce stavu ladicího nástroje 13

2.3.1.Úložiště symbolů 14

2.3.2.Přepočítání pozice IP 15

2.3.3.Obnovení Stepperů 16

2.4.Problém lokálních proměnných 17

2.5.Optimalizace, možnosti zrychlení 18

2.6.Podporované změny 18

2.7.Zapojení do SharpDevelopu 19

2.8.Shrnutí 19

3.Implementace 21

3.1.Rozdělení do komponent 21

3.1.1.Správce EnC 21

3.1.2.Komponenta pro generování IL kódu a Metadat 23

3.1.3.Komponenta pro správu zdrojů 24

3.1.4.Komponenta pro generování úložiště symbolů 24

3.1.5.Komponenta ke zpracování událostí z editoru 25

3.1.6.Komponenta pro obnovu ladicího nástroje 26

3.1.7.Objektový návrh 27

3.2.Změny v SharpDevelopu 27

3.2.1.Změny nutné k obnově ladícího nástroje 27

3.2.2.Kompilátor 28

3.3.Problémy spojené s implementací 28

3.3.1.Porovnávání entit 29

4.Srovnání s existujícími řešeními 30

4.1.Srovnání s podporou pro jazyk Java 30

5.Závěr **32**

5.1. Možná další využití jednotlivých částí projektu 32

5.2.Zhodnocení naplnění cílů 32

5.3.Možná rozšíření 32

5.3.1.Tlačítko Step Back 32

Seznam použité literatury **34**

Obsah CD **36**

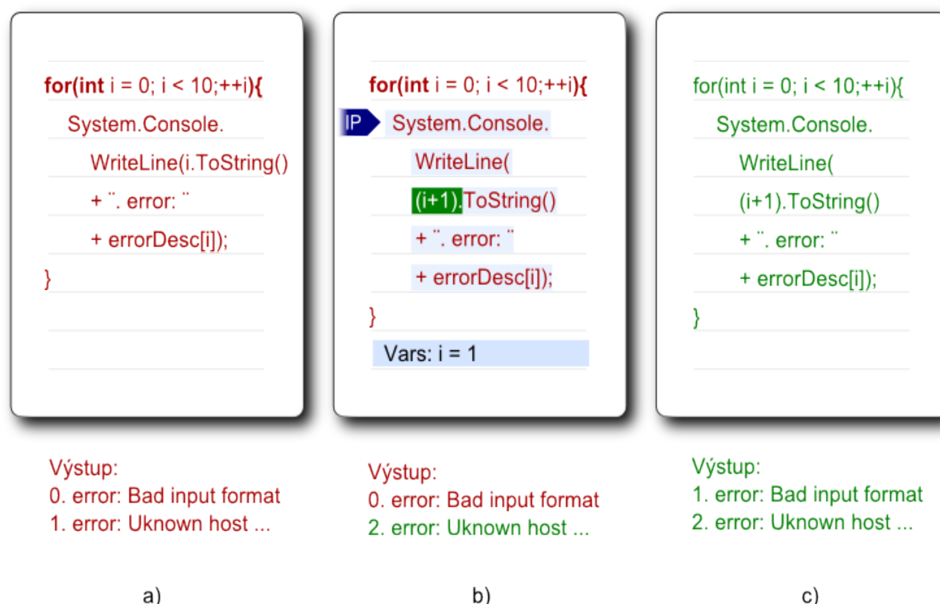
1. Úvod

Při vývoji velkých projektů se vývojář neobejde bez vývojového prostředí a ladicího nástroje, který by mu pomohl odhalovat chyby v kódu aplikace, kterou vytváří. Ladicí nástroje mohou disponovat, krom standardních příkazů ke krokování a výpisu proměnných, také dalšími nástroji, které jsou pro daný jazyk nebo vývojové prostředí typické. Jedním z takových je Edit and Continue (dále jen EnC).

EnC je nástroj, který vývojáři umožní změnit kód aplikace během ladění, bez nutnosti restartovat ladicí nástroj a složitě se dostávat zpět do míst, kde chyba vznikla.

Představme si následující situaci. Ladíme velký projekt, který je např. zásuvným modulem do nějaké ještě větší aplikace. Po spuštění ladění aplikace je potřeba její kód odkrokovat nebo se „proklikat" aplikací, abychom se dostali na místo, které chceme ladit. Nyní se však může stát, že zjistíme, že kód našeho zásuvného modulu fungovat nemůže, protože jsme zapomněli odečíst jedničku od některé proměnné. Toto je ideální případ, ve kterém se funkce EnC ukáže být jedinečným pomocníkem, neboť nám umožní realizovat drobné změny v kódu bez složitého restartování celé laděné aplikace a v ladění dále pokračovat.

S funkcí EnC se ovšem pojí některá programátorská rizika. Největší problémy mohou nastat při změně metod, ve kterých se nachází Instruction Pointer (dále jen IP). IP ve vývojovém prostředí ukazuje řádku, která má být spuštěna při dalším kroku nebo která se právě provádí. Pokud je IP uvnitř nějaké metody, znamená to, že došlo k volání této metody a její kód byl již z části proveden nebo se právě bude spouštět.



Obrázek 1

K demonstraci možného problému lze využít výše zmíněného příkladu a obrázku. Pokud by k dekrementaci, která byla vložena do kódu (původní verze – obrázek 1.a) pomocí EnC, došlo v cyklu, až při druhém průchodu – obrázek 1.b, dostaneme se ke stavu aplikace, který při dalších spuštěních nenastane – další spuštění obrázek 1.c.

Může se i stát, že stav, který je nereprodukovatelný, je stav, který bychom chtěli. Potom se uspokojíme s naší opravou, ale v budoucnu nás čeká nemilé překvapení, a sice to, že kód, který považujeme za otestovaný, nedělá to, co by měl. Pak nastává otázka, zda je funkce EnC užitečná, nebo spíše matoucí. Dle mého názoru je tato funkce velice dobrým pomocníkem, který usnadní vývojáři práci, a sám ji také často využívám. Její použití by se však mělo pojit s opatrností, ale to ostatně platí i o jiných ladicích nástrojích a o způsobu programování jako takovém. Asi nejbezpečnější použití je úprava metod, ve kterých se nenachází IP. U těch ostatních je třeba dávat pozor, aby se aplikace nedostala do stavu, který vznikl jenom díky náhlé změně kódu a je po dalších spuštěních nereprodukovatelným.

1.1. EnC a .NET

Cílem této práce je implementovat EnC pro prostředí, které slouží k vyvíjení aplikací v jazyce C# pro platformu .NET. Microsoft .NET Framework [1] je platforma od společnosti Microsoft, založená na standardu ECMA 335 [2]. Primární vývojové prostředí pro tuto platformu je dílem stejné společnosti a nazývá se Microsoft Visual Studio .NET [3]. Toto prostředí podporuje EnC, ale je komerční. Cílem této práce je tedy vytvořit podporu EnC pro nekomerční prostředí SharpDevelop [4].

1.2. Co je to CLI, CIL ?

CLI (Common Language Infrastructure) [2] je otevřená specifikace, vyvinutá společností Microsoft, jež je standardizována standardy ISO a ECMA [2]. Jedná se především o definici základního programovacího jazyka CIL (Common Intermediate Language) [2] a běhového prostředí VES (Virtual Execution System) [2]. Tento jazyk, jak napovídá anglický název, slouží jako prostřední jazyk, do kterého jsou překládány jazyky vyšší. Podobá se strojovému kódu, ale používá pokročilejší techniky, např. pracuje s objekty. Jazyk těchto vlastností bývá označován jako tzv. bytecode.

Existuje více implementací CLI. Společnost Microsoft, která je autorem standardů, má také svoji implementaci, která se nazývá CLR a je součástí komerčního balíku Microsoft .NET Framework. Ta je ovšem pevně spjata s operačním systémem Microsoft Windows. Další implementací je např. Mono [5]. Projekt rozšiřující CLI na jiné operační systémy jako je Linux, Unix a další.

S balíkem Microsoft .NET Framework je spjato komerční vývojové prostředí Microsoft Visual Studio .NET. Stejně jako implementace CLI existují však další, nekomerční, vývojová prostředí. Jedním z nich je vývojové prostředí SharpDevelop. Jedná se o OpenSource projekt, který se snaží být lehčí variantou prostředí Microsoft Visual Studio .NET. Prostředí podporuje vývoj jak v rámci Microsoft .NET Framework, tak na platformě Mono.

CLI obsahuje kompilátor používající technologii JIT (Just In Time) [6]. To znamená, že CLI převádí jednotlivé metody z CIL kódu do strojového kódu cílové platformy přímo za běhu spuštěné aplikace. Proto je také implementace této části CLI závislá na platformě, na které má běžet.

Aby bylo možné metodu EnC implementovat do vývojového prostředí, je potřeba podpora ze strany CLI. Proto je CLR vybavena některými prostředky, které aplikování změn na laděnou aplikaci umožňují. Tím základním je metoda, která umožňuje předat právě laděnému modulu tzv. delta Metadata a delta IL kód. To, co tyto struktury představují, bude přiblíženo v kapitole Analýza. Po zavolání této metody je potřeba správně obnovit kontext ladění. Získání těchto delta struktur a rekonstrukce stavu ladicího nástroje jsou dva hlavní úkoly této bakalářské práce.

Je důležité poznamenat, že podpora CLR pro EnC je zajištěna pouze pro ladění aplikací v režimu 32-bitů. Při ladění v 64-bitové verzi CLR 4.0, pro kterou je tato bakalářská práce určena, není EnC podporována. Společnost Microsoft oznámila, že hodlá tuto funkcionalitu implementovat i do 64-bitové verze CLR, ale není známo, pro kterou verzi CLR tak učiní a jestli bude používat stejné postupy k aplikaci změn jako nyní 32-bitová verze.

1.3. Co je to SharpDevelop ?

SharpDevelop je, jak bylo zmíněno výše, vývojovým prostředím pro jazyk C# a další. Prostředí je vytvořeno v C#. Cílovou platformou je MS Windows a jako výchozí technologie je zvolena technologie .NET, realizovaná firmou Microsoft. Prostředí ovšem podporuje také vývoj pro Mono. Jeho první beta verze byly uvolněny již v roce 2000. Za zmínku stojí i to, že odříznutím vývojové větve SharpDevelopu v jeho raných počátcích vzniklo vývojové prostředí MonoDevelop, které je dnes ovšem samostatným projektem.

SharpDevelop je vydáván pod licencí LGPL, a jedná se tedy o OpenSource projekt. Jeho stažení tedy není nijak zpoplatněno, a proto i výsledek této práce nabízí autor zdarma jako inspiraci ostatním vývojářům či k přímému použití uživatelům SharpDevelopu.

SharpDevelop je postaven na lehkém jádře, na které je spousta funkcionalit vrstvena pomocí zásuvných modulů. Některé moduly je třeba specifikovat tak, aby je prostředí dokázalo využít. Pro takové potřeby jsou v jádře zaneseny definice těchto modulů. Celé jádro se pak sestává ze služby, která se stará především o vhodné spouštění modulů a nabízí některé globální vlastnosti a zdroje.

1.4. Cíle práce

Hlavním cílem této práce je popsat možnosti jak vyvinout zásuvný modul pro nekomerční prostředí SharpDevelop, který umožní fungování metody EnC pro jazyk C# nad CLR a jednu z možností implementovat. Bude tedy umožňovat zjednodušení jak ladění, tak vývoje aplikací. Aby bylo možno tohoto cíle dosáhnout, je třeba prozkoumat možnosti napojení se na různá rozhraní CLR pro EnC a vyzkoušet některou z možných implementací, která bude spojovat tato rozhraní CLR a ladicí nástroje SharpDevelopu.

Výsledkem práce pak bude jak použitelná funkcionalita EnC, obohacující volně dostupné vývojové prostředí, tak popis problematiky, kterou bylo potřeba analyzovat při návrhu aplikace. Nástin možnoých způsobů implementace a popsání jejich výhod a nevýhod.

2. Analýza

2.1. Úvod

Aby bylo možné vysvětlit, co je ve skutečnosti potřeba pro aplikování změn pomocí EnC, je třeba pochopit stavbu aplikace, která vznikne po kompilaci projektu v prostředí SharpDevelop nebo MS Visual Studiu. Na platformě MS Windows je výsledkem kompilace soubor s příponou *.exe*. Tato přípona byla v minulosti vyhrazena aplikacím v binární podobě strojového kódu. A proto tyto soubory svojí příponou tak trochu klamou. V případě souborů vzniklých kompilací do jazyka MSIL strojový kód aplikace nevzniká. Co je tedy uloženo v souborech, které vznikají takovouto kompilací? Soubory s příponou *.exe* se označují jako PE (Portable Executable) soubory. S technologií .NET přišla i nová možnost využití těchto souborů jako úložiště instrukcí jazyka MSIL [7]. Operační systém pak, po uživatelově pokynu ke spuštění aplikace uložené v takovémto souboru, najde CLR a obsah souboru jí předá ke spuštění. MSIL kód je v souboru uložen ve speciálním formátu, jenž je definován standardy ECMA [2]. Tento soubor bývá označován také jako assembly. Díky tomu, že neobsahuje data, která by byla závislá na platformě, lze takový PE soubor spouštět i na jiných platformách, například pomocí kompilátoru virtuálního stroje projektu Mono.

Hlavním specifikem tohoto formátu je rozdělení MSIL kódu na dvě části: na Metadata a IL (Intermediate Language) kód. Metadata jsou databází obsahující definice hierarchie tříd v MSIL, včetně jejich metod, signatury (serializovaná pole bajtů), názvy metod, tříd a parametrů, definice typů, reference na jiné assembly a jejich třídy a metody, referenční offset začátku metody v IL kódu a další. Každý řádek některé tabulky z Metadat je opatřen jednoznačným identifikátorem, který se nazývá Metadata token. Metadata token je čtyřbajtové číslo. Nejvyšší bajt označuje tabulku, která je adresována. Zbylé tři bajty nesou číslo řádky v této tabulce. IL kód obsahuje serializované instrukce MSIL kódu všech metod v jednom dlouhém bajtovém proudu. Všude, kde je třeba, se odkazuje do Metadat pomocí Metadata tokenu. Definice metod v Metadatech se zase odkazuje do IL kódu pomocí bajtového offsetu, který je označován jako RVA. Ukázka Metadat je na obrázku 2. Obrázek zobrazuje pohled z programu MetadatReader na tabulku `MethodDef`, která obsahuje definice metod. Na obrázku jsou v seznamu nalevo vidět další tabulky Metadat, které zkoumaná assembly využívá.(viz konec kapitoly 2.2.2.).

- Table Stream(#~)
  - Module(0x0)
  - TypeRef(0x1)
  - TypeDef(0x2)
  - Field(0x4)
  - MehtodDef(0x6)
  - Param(0x8)
  - InterfaceImpl(0x9)
  - MemberRef(0xa)
  - Constant(0xb)
  - CostumAttribute(0xc)
  - DeclSecurity(0xe)
  - StandAloneSig(0x11)
  - EventMap(0x12)
  - Event(0x14)
  - PropertyMap(0x15)
  - Property(0x17)
  - MethodSemantics(0x18)
  - MethodImpl(0x19)
  - TypeSpec(0x1b)
  - Assembly(0x20)
  - AssemblyRef(0x23)
  - NestedClass(0x29)
  - MethodSpec(0x2b)
- String Heap(#Strings)
- UserString Heap(#US)
- Blob Heap(#Blob)

Metadata Token	RVA(4b)	ImplFlags(2b)	Flags(2b)	Name(2b)	Signature(2b)	ParamList(2b)
0x06000001	0x20d0	0x0	0x96	Initialize	0x2e	0x1
0x06000002	0x21c9	0x0	0x91	Processes_ProcessAdded	0x32	0x1
0x06000003	0x21e4	0x0	0x91	PostDebugEventHandler	0x39	0x3
0x06000004	0x221c	0x0	0x96	QuitCmd	0x40	0x5
0x06000005	0x23a8	0x0	0x96	HelpCmd	0x40	0x6
0x06000006	0x2688	0x0	0x96	RunCmd	0x40	0x7
0x06000007	0x2918	0x0	0x96	GoCmd	0x40	0x8
0x06000008	0x2940	0x0	0x96	KillCmd	0x40	0x9
0x06000009	0x29c8	0x0	0x96	SetIPCmd	0x40	0xa
0x0600000a	0x2bd8	0x0	0x96	WhereCmd	0x40	0xb
0x0600000b	0x2dac	0x0	0x91	InternalWhereCommand	0x45	0xc
0x0600000c	0x2f22	0x0	0x96	NextCmd	0x40	0xf
0x0600000d	0x2f4b	0x0	0x96	StepCmd	0x40	0x10
0x0600000e	0x2f74	0x0	0x96	StepOutCmd	0x40	0x11
0x0600000f	0x2f9c	0x0	0x96	ShowCmd	0x40	0x12
0x06000010	0x3138	0x0	0x96	ListBreakpoints	0x2e	0x13
0x06000011	0x31e0	0x0	0x96	BreakCmd	0x40	0x13
0x06000012	0x326c	0x0	0x96	DeleteCmd	0x40	0x14
0x06000013	0x32fc	0x0	0x96	ThreadCmd	0x40	0x15
0x06000014	0x34f8	0x0	0x91	GetThreadStateDescriptionString	0x4d	0x16
0x06000015	0x3634	0x0	0x96	SuspendCmd	0x40	0x17
0x06000016	0x3664	0x0	0x96	ResumeCmd	0x40	0x18
0x06000017	0x3684	0x0	0x91	SetDebugStateWrapper	0x53	0x19
0x06000018	0x377c	0x0	0x91	ThreadResumeSuspendHelper	0x5c	0x1c
0x06000019	0x393c	0x0	0x96	InterceptCmd	0x40	0x1f

Obrázek 2

Je důležité pochopit, jak se člení laděný proces v rámci implementace ladicího nástroje a jaké jsou možnosti přístupu k jeho součástem a práce s nimi. Jak bylo popsáno výše, je ke spuštění aplikace potřeba CLI, proto je tato specifikace nutná i pro její ladění. V případě CLR existuje velké množství nástrojů, jak aplikaci pro ladění spustit a jak její běh řídit. Součástí těchto nástrojů jsou mimo jiné nástroje pro aplikování změn v rámci EnC. CLR dává vývojovým prostředím k dispozici řadu rozhraní pro práci s laděnou aplikací. Základním rozhraním je rozhraní `ICorDebug`. To umožňuje správu několika procesů, jejich spouštění, procházení a přichytávání se na již spuštěný proces. Jednotlivé procesy lze řídit pomocí rozhraní `ICorDebugProcess`, která lze opatřit právě z instance rozhraní `ICorDebug`. Instance `ICorDebugProcess` vlastní několik instancí rozhraní `ICorDebugModule`, kde každá z těchto instancí je spjata právě s jednou assembly, kterou daný proces používá. Od verze CLR 2.0 včetně platí, že všechny instance `ICorDebugModule` implementují fakticky rozhraní `ICorDebugModule2`.

Nyní lze představit základní možnosti jak aplikovat změny na laděnou aplikaci. Klíčovou metodou je metoda `ApplyChanges` [8] rozhraní `ICorDebugModule2`. Tato metoda bere jako parametry dvě bajtová pole, označovaná jako delta Metadata a delta IL kód. Jedná se o útržek databáze, označované jako Metadata, a IL kódu. Oba útržky jsou převedeny na bajtový proud formátem velice podobným formátu, který je používán k vytvoření PE souborů [2]. Obsah útržků těchto struktur je omezen na kýžené změny v aplikaci a neobsahuje prakticky nic navíc.

Při návrzích prvních prototypů bylo potřeba odpovědět si na následující

otázky. Je metoda `ApplyChanges` skutečně spustitelná pro jiná vývojová prostředí, než je Microsoft Visual Studio .NET ? Jak opatřit testovací data pro aplikování změn pomocí této metody? Lze se v SharpDevelopu dostat k vhodné instanci rozhraní `ICorDebugModule2`? Je možné dané útržky vygenerovat? Existuje nástroj, který by generování útržků usnadnil nebo vyřešil kompletně, a jestli ano, tak za jakou cenu ?

Jako zdroj testovacích dat posloužil nástroj `ilasm` [7], který je určen ke kompilování MSIL kódu do výsledného PE souboru. Tento nástroj, který je součástí balíku Microsoft .NET Framework, dokáže jako vedlejší produkt kompilace se speciálními parametry a vstupem připravit potřebné výše zmíněné struktury delta Metadata a delta IL kód. K ověření vygenerovaných struktur a prozkoumání procesu aplikování změn se ukázal být velice vhodný CLR Managed Debugger, který je označován jako `MDbg` [9][10]. Jedná se o ladicí nástroj, kterým lze ladit aplikace napsané v jazyce MSIL a zkompilované pomocí `ilasm`. Nástroj disponuje příkazem, který aplikuje struktury vygenerované pomocí `ilasm`, a změní tak kód aplikace během jejího ladění, tedy realizuje EnC. Navíc je tento nástroj volně ke stažení, včetně jeho zdrojového kódu. Na změnu kódu aplikace používá právě metodu `ApplyChanges`. To odpovídá na dvě z položených otázek. Tedy lze vytvořit ladicí nástroj, který používá pouze rozhraní CLR určená k ladění a realizuje přitom změny EnC pomocí metody ApplyChanges. Navíc vhodné spuštění `ilasm` umožňuje vygenerovat testovací verzi útržku kýžených struktur.

Prvním prototypem se stala aplikace, která již měla podobu zásuvného modulu do prostředí SharpDevelop. Kód aplikace byl aktivován po stisku tlačítka. Z modulu pro ladění vývojového prostředí SharpDevelop byla získána instance rozhraní `ICorDebugModule2` pro laděnou assembly. A pomocí metody `ApplyChanges` byly úspěšně aplikovány struktury předem získané nástrojem `ilasm`.

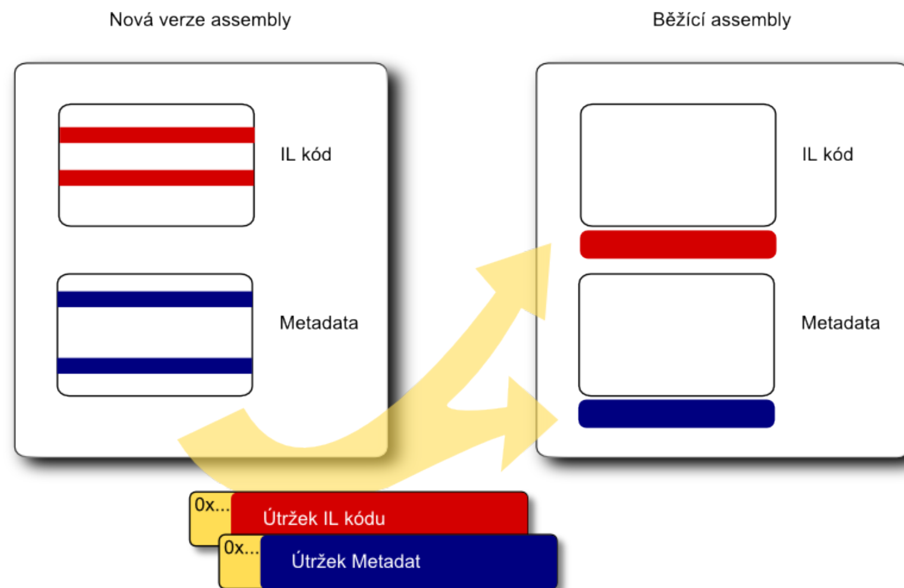
Další pokusy sestávaly ze snahy cíleně měnit vygenerované struktury. Na základě informací o formátu bajtového proudu Metadat a IL kódu jsem se snažil měnit útržky tak, abych mohl měnit jiné metody nebo vkládat jiný kód metod. Tyto snahy byly spíše neúspěšné, to také rozhodlo o dalším postupu k programovému získávání útržku.

2.2. Problém vytvoření delta Metadat a delta IL kódu

Jak bylo zmíněno výše, k aplikováni změn pro spuštěný proces, v režimu ladění, pomocí přípravků v CLR, je nutné mít připravenou část Metadat a část IL kódu, které tyto změny popisují. Nabízí se dvě alternativy, jak tyto struktury získat. První je vytvořit program, který je na základě změn ve zdrojovém kódu vygeneruje. Druhá je najít nějaký nástroj, který nám s generováním pomůže, případně udělá generování kompletně.

První možnost má výhodu v tom, že aplikace nebude závislá na nějakém nástroji a celý kód bude mnohem přenositelnější. Na druhou stranu je náročnost tohoto úkolu téměř stejná, jako je náročnost vývoje kompilátoru pro jazyk C# jako takového. Je třeba analyzovat syntaxi, na to SharpDevelop nástroje má. Kód se pak ale musí vhodně převést do jazyka MSIL. Ten se pak zase musí náležitě převést na bajty tak, abychom dostali IL kód a u toho je třeba zapsat všechny potřebné informace do Metadat..

Kompilátor je program, který je přesně stavěný k takovým úkolům. Jeho produktem je právě assembly obsahující IL kód a Metadata. Navíc každé vývojové prostředí by mělo mít nějaký kompilátor k dispozici. Proto se také kompilátor jeví jako dobrý nástroj pro realizování druhé možnosti. Aplikace změn by pak mohla probíhat tak, jak ilustruje obrázek 3. Tedy s pomocí kompilátoru bude vytvořena nová assembly a z ní pak budou



Obrázek 3

extrahovány útržky IL kódu a Metadat. Tato implementace má ale také svoje nevýhody. Jednou z nich je ta, že pro získání malé části obou struktur je třeba spustit kompilaci celého projektu. Nabízí se řešení – dát kód metody, která byla změněna, do menšího projektu a ten zkompilovat stranou. Taková metoda však teoreticky může používat všechny třídy z projektu a všechny jejich metody. A tak by musel projekt stranou obsahovat všechno z původního projektu. Další nevýhodou je, že vznikne kompletně nová assembly s novým IL kódem a novými Metadaty. Je téměř jisté, že Metadata nebudou generována pokaždé stejným způsobem. Proto může jeden token z Metadat v původní assembly reprezentovat jinou strukturu v nové assembly.

Nakonec jsem jako metodu k implementaci zvolil druhou možnost – vytvoření těchto struktur pomocí již hotového kompilátoru. To proto, že první možnost byla příliš náročná na čas, který by realizace potřebovala. Bylo však nutné vyřešit úskalí, která tato metoda přináší. Největším problémem byl přenos vygenerovaných struktur mezi novou a starou assembly.

2.2.1. Generování IL kódu

U IL kódu je potřeba odkazy do Metadat nové assembly, které představují Metadata tokeny, přeložit na odkazy na stejné struktury ve staré assembly. Tedy nahradit Metadata token použitý v nové assembly tokenem reprezentujícím stejnou strukturu v laděné assembly. Stačí rozdělit IL kód na instrukce a jejich operandy. Operandy, které představují Metadata token, přeložit výše zmíněným postupem a poté zas převést IL kód zpět na bajty. K tomuto účelu výborně posloužila knihovna `Mono.CECIL` [11], která je navržena na čtení PE souborů a která dokáže vytvořit reprezentaci IL kódu, jež daná kritéria naplňuje.

Na realizaci výše zmíněného postupu je potřeba nástroj, který dokáže překládat Metadata token z nové assembly na token z běžící assembly. Jedním z možných postupů, který se nabízí, je zapsat do Metadat, která byla vygenerovaná jako útržek, všechny tokeny, které útržek IL kódu využívá v jejich původní podobě. Je celkem zřejmé, že tahle metoda nemůže fungovat. Je to proto, že delta IL kód a delta Metadata jsou de facto připojeny na konec již existujícího IL kódu a Metadat v laděné aplikaci. Tedy token struktury, kterou pomocí tohoto tokenu adresoval původní IL kód, musí zůstat zachován, aby ji mohl adresovat i po změně. Navíc každá definice musí mít právě jeden token. Je tedy zřejmé, že abychom mohli najít odpovídající token ke struktuře z nové assembly v běžící assembly, musíme mít přístup k jejím Metadatům.

2.2.2. Generování Metadat

Otázka generování Metadat je složitější. Již při prvních pokusech s datovými útržky ve výše zmíněném prototypu šlo bez obtíží realizovat ruční změny IL kódu. Naopak ruční editace útržku s Metadaty se ukázala jako prakticky nemožná. Bylo to především proto, že způsob bajtové reprezentace útržku Metadat pro EnC je trochu jiný než způsob bajtové reprezentace standardních Metadat. V Metadatech pro EnC jsou navíc dvě tabulky, `ENCLog` a `ENCMap` [6], jejich schéma je také popsáno, ale jejich přesný význam popsán není. Mezi těmito dvěma druhy Metadat jsou ještě další rozdíly. Některé byly odhaleny experimentálně během vývoje.

Ke generování Metadat se nabízely dvě možnosti. První byla vytvořit vhodné bajtové pole z informací, které obsahuje nově vytvořená assembly. Všechny potřebné struktury nového kódu pak spolu s označením změněných metod zapsat, včetně všech doplňujících informací, vhodně do bajtového proudu, ale zároveň je také držet k dispozici tak, aby bylo možné tyto informace použít k překládání tokenů v IL kódu. Výhodou je nezávislost na dalších nástrojích, a tedy dobrá přenositelnost. Na druhou stranu slabá dokumentace formátu, který CLR u útržku Metadat očekává, je velkou překážkou při realizaci této možnosti.

Druhá možnost spočívala ve využití rozhraní IMetaDataEmit. Toto rozhraní je opět poskytováno CLR pro laděnou assembly. Rozhraní slouží k registrování entit do Metadat. Entitou myslíme třídu, vzdálený datový typ, metodu, vlastnost, ale i bajtovou reprezentaci řetězců, které byly uvnitř kódu použity. Navíc však toto rozhraní obsahuje metodu SaveDelta, resp. SaveDeltaToMemory, a metodu ResetENCLog. Metoda SaveDeltaToMemory, slouží k zapsání rozdílového útržku Metadat do operační paměti. Rozdílový útržek je útržek Metadat obsahující to, co bylo přidáno od posledního volání ResetENCLog. Výhodou je, že výstupem SaveDeltaToMemory je bajtové pole, jehož formát je shodný s formátem, který očekává metoda ApplyChanges. Nevýhodou je, že vzrůstá závislost na CLR, a je tedy omezena přenositelnost kódu.

Druhá možnost disponuje ještě jednou výhodou. Tou je možnost využití rozhraní IMetaDataEmit jako překladače Metadata tokenů z nové verze assembly na odpovídající tokeny v běžící verzi aplikace. Pokud je rozhraní IMetaDataEmit nastaveno v režimu MDUpdateENC, je možné „registrovat“ existující struktury do Metadat. V takovém případě je pouze vyhledán existující záznam a metoda rozhraní IMetaDataEmit, která byla k pokusu o registraci použita, vrátí Metadata token již existujícího záznamu. Pokud je entita v Metadatech s jiným parametry, její parametry jsou změněny a informace o těchto změnách je zanesena do útržku Metadat, který vygeneruje metoda SaveDeltaToMemory (nebo SaveDelta). Token takto změněné entity je zachován a opět navrácen po dokončení volání odpovídající metody pro registraci. Díky tomuto chování lze generování útržků Metadat a IL kódu postavit na následujícím algoritmu:

1. *procházení IL kódu, nejedná-li se o token, přepisuje se přímo do výstupního bajtového pole,*
2. *nalezení tokenu,*
3. *získání potřebných informací k registrování tokenu z nové verze assembly (např. pomocí IMetaDataImport pro novou assembly),*
4. *registrování získaných informací pomocí rozhraní IMetaDataEmit pro běžící verzi assembly, získání výsledného tokenu,*
5. *nahrazení načteného tokenu tokenem, který byl získán v předchozím kroku,*
6. *zapsání instrukce i s novým tokenem do výstupního bajtového pole.*

Je vidět, že tento algoritmus řeší problematiku generování obou struktur. Je také zřejmé, že po doběhnutí obsahuje nový IL kód odkazy na stejné entity jako IL kód získaný z nově zkompilované assembly. Navíc

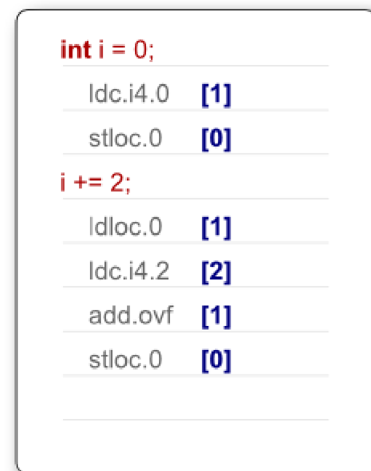
byly všechny tyto entity zaneseny do běžících Metadat. Tento algoritmus lze zefektivnit přidáním cache. Ta, díky jednoznačnosti tokenů v obou instancích Metadat, umožní registraci každé entity použité v IL kódu právě jednou.

Nakonec byla implementována druhá možnost. Pokud by byl k dispozici přesný formát útržku Metadat pro EnC, byla by přesto implementace první možnosti získávání Metadat pro EnC velice náročná. Druhá možnost nám navíc dovoluje spojit funkci generování Metadat s překládáním Metadata tokenů, a tedy i s generováním IL kódu.

Jako pomocný nástroj při analýze PE souborů a formátu Metadat posloužil nástroj, který byl k tomuto účelu vyvinut. Jedná se o program, který dokáže z PE souboru získat tabulky Metadat spolu s jejich řádky. Program byl poté rozšířen o schopnost načítat útržky Metadat připraveny pro EnC. Tento nástroj pomohl jak při ladění vyvíjeného modulu, tak při experimentálních rozvahách a snaze pochopit formát útržků Metadat.

2.3. Rekonstrukce stavu ladicího nástroje

Pro vysvětlení další problematiky je potřeba objasnit pojem sekvenční bod. CIL kód je objektově orientovaný jazyk založený na instrukcích a zásobníku (angl. stack-based). To znamená, že vstupní data pro instrukce jsou načítána ze zásobníku pro danou metodu a výsledek instrukce je opět uložen na zásobník. Sekvenční bod (angl. Sequence Point) je pozice v IL kódu, kdy je zásobník prázdný. Na obrázku 4 je červenou barvou jednoduchá část kódu v C#. Pod oběma příkazy v C# jsou tmavě šedou barvou zapsány instrukce v CIL kódu, které daný příkaz vykonají tak, jak je přeložil kompilátor. Číslo v hranatých závorkách udává počet prvků na zásobníku po vykonání instrukce uvedené nalevo od čísla. Je vidět, že po dokončení instrukcí pro příkaz je zásobník prázdný. To platí obecně. Díky tomu se používají sekvenční body pro vyjádření vztahů mezi původním zdrojovým kódem a instrukcemi.



Obrázek 4

Při aplikování změn pomocí EnC na zdrojový kód nějaké metody se způsob aplikace změny dělí na dva případy, a to podle toho, je-li při aplikaci změny IP uvnitř metody, nebo ne. V druhém případě se nejedná o nic složitého, protože kód metody, ve které není IP, není zrovna vykonáván. Potom není problém přepnout ze starého kódu na nový a ve chvíli, kdy bude metoda zavolána, spustit rovnou nový kód. V prvním případě běží ještě stará verze kódu, dokud se IP nedostane k sekvenčnímu bodu,

pak CLR přes rozhraní ICorDebugManagedCallback2 zavolá FunctionRemapOpportunity, čímž dá najevo, že kód může být vyměněn za nový. Na instanci rozhraní ICorDebugILFrame2 je třeba zavolat metodu RemapFunction a předat jí pozici v nové verzi IL kódu, na které se nachází nějaký sekvenční bod. Až po zavolání této metody dojde k samotné výměně IL kódů a IP je nastaven na pozici, která byla metodě předána.

Zpracováním poznatků diskutovaných v kapitole 2.2 vznikl prototyp, který dokázal vhodně aplikovat změny provedené v kódu aplikace. Prototyp byl napojen na události z editoru a ladicího nástroje prostředí SharpDevelop, tedy dokázal po zastavení na breakpointu, nebo mezi kroky při krokování, rozpoznat, zda došlo ke změně v kódu. Pokud ano, byla vytvořena nová assembly, obsahující novou verzi kódu. Poté byly výše zmíněným postupem získány útržky potřebných struktur a nakonec zavolána metoda ApplyChanges na modulu, který reprezentoval laděnou aplikaci. Po zavolání této metody pokračovalo prostředí SharpDevelop v ladění aplikace.

Pozorování chyb tohoto prototypu ukázalo, že po změně metody, ve které není IP, je možné kód dále krokovat, ale zvýrazňování kódu kopírovalo tvar (začátky a konce řádek) původní verze kódu. Po změně metody, ve které IP byl, ovšem dále krokovat nešlo. Pravděpodobně proto, že byla jako pozice v nové verzi kódu předávána pozice v původní verzi kódu, kterou poskytují parametry metody FunctionRemapOpportunity. Takto zvolená pozice téměř jistě nebyla sekvenčním bodem v nové verzi IL kódu, a tudíž se CLR dostala do nečekaného stavu a program doběhl sám až do konce. Je tedy potřeba vytvořit nástroj, který dokáže spočítat novou pozici v IL kódu, která je sekvenčním bodem, ale také odpovídá místu, od kterého se má v krokování kódu pokračovat.

2.3.1. Úložiště symbolů

Co se týká špatného zvýrazňování tvaru kódu, tak jsem z dalších experimentů s ilasm a MDbg zjistil, že pro úspěšné aplikování změn je potřeba ještě třetí struktura. Assembly jako taková v sobě zdrojový kód neobsahuje. Jak je možné, že lze ladit aplikace a krokovat jejich zdrojový kód i v těch případech, kdy jejich zdrojový kód přímo nemáme? Navíc tu je otázka, jak se zaznamenává vztah mezi řádky původního zdrojového kódu v C# a instrukcemi v IL kódu. V assembly pro takovéto informace není vyhrazeno žádné místo.

Odpovědí na tyto dvě otázky je existence třetí důležité struktury pro aplikace, které jsou ve fázi vývoje (připraveny k ladění). Tato struktura se nazývá úložiště symbolů (angl. Symbol Store). Úložiště symbolů obsahuje zdrojové kódy všech metod, vztah mezi IL kódem a zdrojovým kódem

aplikace, názvy, pozice a typy lokálních proměnných a další informace, které nejsou nutné ke spuštění aplikace, ale poskytují řadu informací při jejím ladění. Na platformě Microsoft Windows je toto úložiště ukládáno do samostatného souboru, proto nejsou tato data uvnitř assembly.

Pro správu tohoto úložiště dává CLR k dispozici rozhraní `ISymUnmanagedReader`. Ten je vázán k jedné assembly, a tedy i k jedné instanci rozhraní `ICorDebugModule`. V SharpDevelopu vlastní obě dvě instance instance třídy Module. Rozhraní `ISymUnmanagedReader` navíc disponuje metodou `UpdateSymbolStore` [12], která aktualizuje instanci tohoto rozhraní pomocí útržku úložiště symbolů (delta symbol store). V popisu této metody je přímo napsáno, že slouží pro potřeby EnC.

Při kompilaci projektu ve vývojovém prostředí je toto úložiště symbolů vytvářeno spolu s assembly, pokud je kompilátor spuštěn s příslušným parametrem. Dalším pokusem tedy bylo nechat při kompilaci nové verze assembly vytvořit i toto úložiště a předat ho celé, beze změn, metodě `UpdateSymbolStore` z instance rozhraní `ISymUnmanagedReader`, která patří k laděnému modulu. Tento pokus se nad očekávání vydařil. U nepříliš složitých metod bez proměnných bylo při krokování poznat, že zvýrazňování řádek skutečně odpovídá nově vytvořenému kódu.

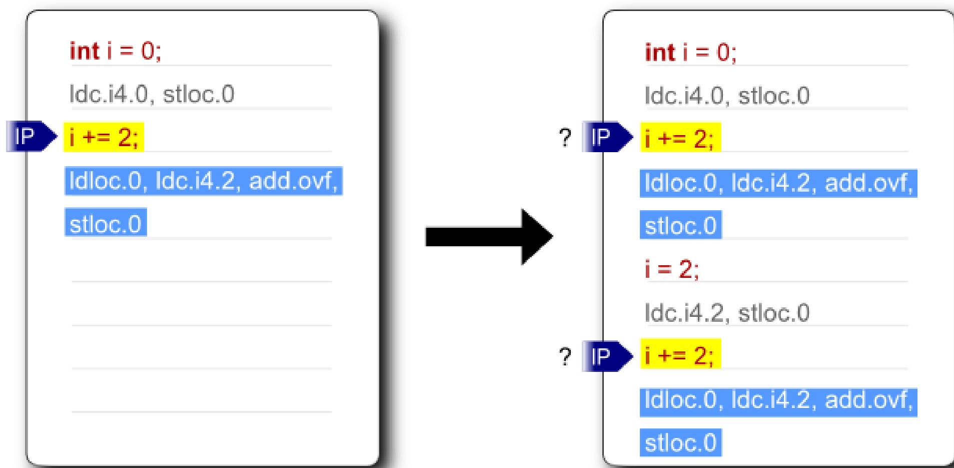
U metod, jejichž kód je komplikovanější, se ladicí nástroj dostal do neočekávaného stavu, a ladění tak skončilo výjimkou ze strany SharpDevelopu. Bližší zkoumání úložiště symbolů ukázalo, že obsahuje také definice lokálních proměnných a jejich typů. Z těchto důvodů se signatury, které ladicí nástroj používá, odkazovaly do Metadat nové assembly, a proto nemohlo být úložiště spojeno s běžící assembly. V tuto chvíli se opět nabízely dvě možnosti, jak problém řešit. Generovat úložiště symbolů celé nebo jen překládat tokeny do Metadat z nové assembly na tokeny v běžící assembly.

První možnost má opět výhodu větší přenositelnosti. Ovšem získání některých informací o metodě a jejím zdrojovém kódu je prakticky nemožné bez alespoň částečné kompilace metody. Velkým problémem by bylo například párování sekvenčních bodů v IL kódu k řádkám zdrojového kódu. Druhé řešení naopak nevyžaduje nic, co by už v současné verzi projektu nebylo. Nástroj na překládání tokenů je potřeba již ke stavbě IL kódu. Z těchto důvodu jsem implementoval možnost druhou.

2.3.2. Přepočítání pozice IP

Dalším úkolem bylo implementovat nástroj, který umožní spočítat pozici v IL kódu, která je potřebná pro zavolání metody `RemapFunction`. Obecněji řečeno je třeba mít nástroj, který umožní přepočítat pozici sekvenčního bodu ve staré verzi IL kódu na pozici odpovídajícího sekvenčního bodu v nové verzi IL kódu. To proto, že není garantováno volání `FunctionRemapOpportunity` od CLR při nejbližším sekvenčním bodu,

ale na nějakém dalším sekvenčním bodu. Nabízí se možnost provést porovnání obou IL kódů a najít stejné části, části smazané a části přidané. Tato možnost ovšem trpí nedostatkem, ten je znázorněn na obrázku 5.



Obrázek 5

Nalevo je stará verze kódu a napravo je verze po změně pomocí EnC. Jak je vidět na obrázku, pomocí porovnání, které je podloženo pouze původní a novou verzí IL kódu, nelze zjistit, kam je vhodné situovat novou pozici IP. Z tohoto důvodu je potřeba sbírat události při editaci zdrojového kódu a ukládat je. Především je důležité sledovat smazání a vytvoření nových sekvenčních bodů. To lze pomocí jednoduché heuristiky, nový sekvenční bod je zakončen jedním ze symbolů:

```
; { }
```

Díky tomu, že editor SharpDevelopu poskytuje informaci o každém vložení a smazání znaků v kódu, lze v průběhu editace kódu udržovat stále aktuální informaci o vložených a smazaných sekvenčních bodech, a mít tedy k dispozici přepočet z pozic sekvenčních bodů staré verze IL kódu na aktuální. Pro potřeby metody `RemapFunction` stačí vzít poslední verzi výše zmíněné datové struktury a přepočítat pozici v IL kódu, kterou dostaneme jako parametr události `FunctionRemapOpportunity` na pozici v novém IL kódu.

2.3.3. Obnovení Stepperů

Po implementování této metody se objevil další problém. Při úspěšném aplikování změn na metodu A, uvnitř které byl IP, a pokusu o „krok nahoru" (návrat do metody B, která spustila kód metody A) doběhl ladicí nástroj do konce programu a ukončil ladění, namísto toho aby IP zůstal na řádce, která spustila metodu A. Problém byl spojen s tím, že řada instancí ladicích rozhraní, jako je `IcorDebugStepper`, přestane být validní po aplikování

změny na metodu uvnitř vlákna, ke kterému tyto instance patří. Stepper je třída sloužící ke kontrole kroku v ladicím nástroji. Dává informaci o tom, zda byl krok proveden a umožňuje ho zrušit. Pro dokončení „kroku nahoru" je tedy třeba mít zprávu o tom, že „krok dovnitř" byl úspěšně proveden. Aby tuto zprávu prostředí dostalo, je potřeba znovu vytvořit instanci stepperu pro „krok dovnitř", protože ta původní už není platná. Podobný problém se vztahuje k breakpointům. I ty je po zavolání metody `ApplyChanges` potřeba obnovit.

2.4. Problém lokálních proměnných

Mike Stall na svém blogu [13] popisuje některá omezení EnC. Jedno z nich se týká změny metody s IP uvnitř. Pokud je změněn počet, nebo pořadí proměnných v nové verzi IL kódu, nesmí být žádná proměnná ze své pozice odstraněna. Toto omezení má zřejmou příčinu. Metodě je přidělena nějaká paměť, a to včetně jejích proměnných. Velikost paměti, která je přidělena nějaké proměnné, závisí na jejím typu. Pokud by se v důsledku změny IL kódu dostala na místo původní proměnné nějaká proměnná jiného typu, jenž vyžaduje jinak velký kus paměti, bude pro ní vyhrazené místo buď nedostatečné nebo zbytečně veliké. Navíc pokud pokračování v krokování kódu upravované metody poběží od místa, kterému v kódu předchází inicializace dané proměnné, bude na místě takové proměnné i špatný obsah (bude mít nesprávný typ a bude reprezentovat obsah staré proměnné, která byla smazána).

Nabízí se opět dvě možná řešení. První spočívá v tom, že se na místa proměnných, které byly smazány, ještě před kompilací doplní do správného úseku kódu (angl. scope) taková proměnná, která pouze zabírá místo původní proměnné, a to s názvem, který nekoliduje s žádným použitým v kódu. Tato metoda má několik nevýhod, první je otázka úpravy zdrojového kódu a nalezení správného úseku. Druhou je generování nových názvů proměnných tak, aby nekolidovaly s těmi použitými v kódu. Pak je tu problematika změny pořadí proměnných a poslední je nutnost zasáhnout do generování symbolů, abychom ony proměnné, jež pouze obsadily místa původních proměnných, nezobrazovali při ladění metody.

Druhou možností je při aplikaci vygenerovaného IL kódu porovnat seznam proměnných v běžící verzi a v nové verzi IL kódu. Ty, které si odpovídají, umístit na stejnou pozici, jako tomu bylo ve staré verzi. Ty, které byly smazány, ponechat v IL kódu na své pozici, ale nevygenerovat pro ně symboly. Potom je nutné vhodně držet informace popisující přesun struktur v nové verzi IL kódu a při zapisování instrukcí všechny instrukce, které pracují s lokálními proměnnými, vhodně přeložit, aby pracovaly se stejnou proměnnou na nové pozici. Toto řešení je robustnější, řeší i přeházení proměnných a snaží se maximálně využít prostor pro proměnné vyhrazený. Na druhou stranu je zde potřeba provádět změny v instrukcích IL kódu, byť jednoduché. Stejně jako u prvního řešení, je i zde třeba změnit i způsob

generování úložiště symbolů. Největším problémem je však rozpoznání odpovídajících si proměnných.

Díky větší robustnosti a tomu, že není nutné zasahovat přímo do zdrojového kódu nebo jeho kompilování, jsem zvolil druhou možnost. Rozpoznávání odpovídajících si proměnných jsem položil na kontrole názvu, typu a pozic v IL kódu, mezi kterými je proměnná platná. Tyto pozice je nutné přepočítat pomocí inverze funkce, kterou nám dal nástroj na překlad z pozic starých sekvenčních bodů na pozice bodů nových, zmíněný na konci kapitoly 2.3, abychom z pozic v nové verzi IL kódu dostali pozice ve staré verzi IL kódu a mohli porovnat, jestli proměnné platily před realizováním změny na stejném úseku kódu.

2.5. Optimalizace, možnosti zrychlení

Místo, které se zdá nejnáročnější na čas běhu při aplikováni změn pomocí EnC, je kompilace nové verze assembly. Ideálním řešením by bylo překládat pouze kód metody, která byla změněna. Jak však bylo popsáno výše, je tato metoda těžko realizovatelná, protože je těžké oddělit kód metody od zbytku projektu. Původním plánem bylo překládat celý projekt znova. Počítalo se s tím, že takový překlad bude velice náročný na čas. Při implementaci projektu bylo objeveno řešení, které tuto náročnost snižuje. V C# existuje klíčové slovo „extern“. U metody, která je uvozena tímto klíčovým slovem, kompilátor nevyžaduje zdrojový kód ve chvíli kompilace, kód je vyžadován až ve chvíli spouštění assembly. Díky tomu je možné vygenerovat kód, který před všechny metody, krom změněné, vloží toto klíčové slovo a de facto se překládá pouze kód metody, která byla změněna.

Dalším úskalím je srovnávání běžící a nové verze kódu. Protože CLR umožňuje realizovat jen některé změny ve zdrojovém kódu, je třeba zjistit, jestli nebyly v projektu učiněny změny, které restrikce porušují. Např. CLR umožňuje přidání soukromé metody, ale nedovoluje přidání metody veřejné. Je proto třeba srovnat starou verzi projektu s novou verzí a zjistit všechny přidané nebo smazané členy třídy. Jako řešení tohoto problému je porovnána každá třída nové verze projektu se starou verzí. Tento způsob hledání chyb lze určitě optimalizovat pomocí vhodné interpretace událostí z editorů.

2.6. Podporované změny

Jednou z důležitých otázek, které je třeba vyřešit před implementací, je otázka, které změny CLR pro EnC podporuje. Na stránkách MSDN [14] je popsáno, které změny lze realizovat pomocí EnC nad CLR. V základě se jedná o tyto změny:

- přidání privátní nevirtuální metody,
- přidání privátní členské proměnné,

- přidání privátní vlastnosti,
- změna kódu metody, která není generická,
- změna kódu vlastnosti.

Některé z nich mají ještě další omezení. Je to například ono omezení, které bylo zmíněno pro úpravu kódu metody viz. kapitola 2.4, kde nesmí být proměnná smazána. Další omezení se týkají spouštění aplikace. EnC nelze aplikovat na assembly, která byla připojena k ladicímu modulu po spuštění. Dále nelze EnC aplikovat na aplikaci, ke které je připojen profiler (nástroj pro testování výkonnosti).

2.7. Zapojení do SharpDevelopu

Způsobů zapojení modulu do prostředí SharpDevelop se nabízí několik. Je třeba vyřešit otázky, kdy modul spustit, jak mu předávat události z ladicího nástroje a jak předat řízení zpět. Nabízí se možnost vytvořit definice nového typu modulu pro SharpDevelop, ale protože tento modul pro EnC potřebuje přístup k prostředkům, které drží pouze ladicí modul a které jsou z jádra nepřístupné, je snazší je spolu provázat. Potom se nabízí možnost nedělat z EnC samostatný modul, ale začlenit ho přímo do modulu, který reprezentuje ladicí nástroj. Tato metoda má výhodu zjednodušení komunikace mezi komponentou realizující EnC a ladicím nástrojem. Ztratí se tím ovšem možnost EnC modul zcela deaktivovat. Další možností je použít jednoduchý typ modulu a připojit ho k SharpDevelopu hned po startu. Pouze se připojit k odběru událostí od ladicího nástroje a skončit. K dalšímu zavolání modulu dojde až ve chvíli, kdy je spuštěno ladění programu.

Rozhodl jsem se implementovat poslední možnost. Její nevýhodou je to, že prodlouží dobu spouštění SharpDevelopu, ale kód, který je třeba vykonat k připojení se na události ladicího nástroje, je krátký. Jenom je třeba se přesvědčit, zda byl před spuštěním tohoto modulu již spuštěn modul reprezentující ladicí nástroj. Výhodou je jednoduchá implementace a řešení, které umožňuje zpětnou podporu možnosti sloučení s ladicím nástrojem, která je zvažována na straně vývojářů SharpDevelopu. Další výhodou je možnost deaktivovat modul pro EnC.

Modul je propojen s ladicím nástrojem pomocí rozhraní `IEnCHook`, které bylo třeba přidat do ladicího modulu a které volá některé metody reprezentující zachycení událostí, které z bezpečnostních příčin ladicí nástroj nedává k připojení.

2.8. Shrnutí

Dosud se v této kapitole pro každý problém nabízí dva různé přístupy k řešení problému. Jeden je postavit ve své podstatě nový kompilátor, který za přístupu k běžící assembly zkompiluje nové tělo metody. Druhý je využít

stávající kompilátor a výsledek kompilace upravit tak, aby byl použitelný pro kritéria EnC.

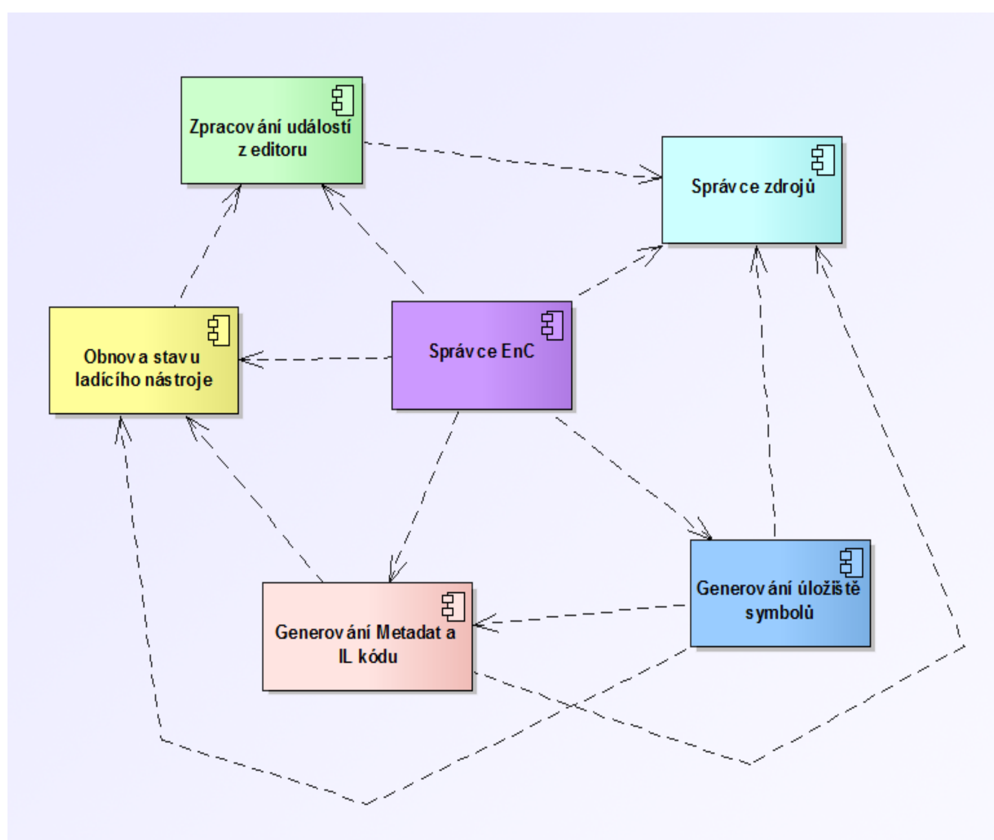
Je vidět, že jednotlivá rozhodnutí se vzájemně ovlivňují. Pokud je jako řešení při první otázce generování IL kódu zvolena metoda, která využívá stávající kompilátor, nedávalo by smysl vytvářet kompilátor pro generování Metadat. To je také důvod, proč není snadné zvolit správný způsob implementace funkcionality EnC před zahájením vývoje. Osobně jsem se implementace nového kompilátoru bál a odhadoval jsem náročnost zvoleného způsobu implementace jako jednodušší a rychlejší cestu. Při implementaci se však vynořovaly nové a nové problémy, až se při jejich řešení stal z nástroje na převod kódu z nové assembly do prostředí běžící assembly modul, který kompilátor připomíná a dělá část jeho práce.

Podle toho, co se mi podařilo zjistit, Microsoft Visual Studio .NET realizuje metodu EnC pomocí vlastního speciálního kompilátoru. Díky tomu je tato implementace důkazem, že to lze implementovat i jiným způsobem.

3. Implementace

3.1. Rozdělení do komponent

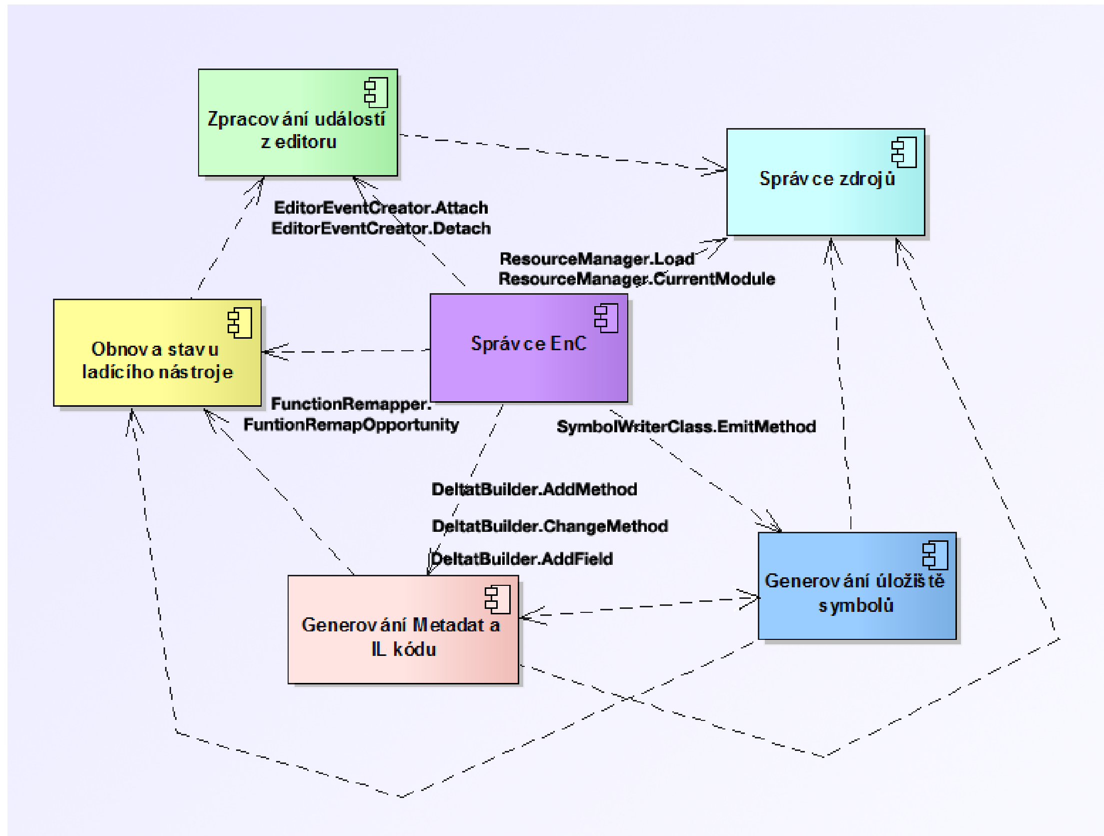
Projekt lze rozdělit na několik funkčních celků sestávajících ze tříd, které dohromady realizují některou z úloh, které jsou popsány v kapitole 2. Takové funkční celky budou nadále označovány jako komponenty. Tyto komponenty a vztahy mezi nimi jsou znázorněny na obrázku 6. V následujících kapitolách budou zmíněné vztahy a jejich úloha v projektu popsány.



Obrázek 6

3.1.1. Správce EnC

Celý projekt zastřešuje komponenta spravující zásuvný modul EnC, jež je reprezentována třídou `EnCManager`. Ta je dimenzována tak, že k jedné instanci laděného projektu je přiřazena jedna instance této třídy. Správce EnC se stará o správu a zprostředkování komunikace dalších komponent. Obrázek 7 ukazuje vztahy komponenty s ostatními komponentami, spolu s třídami a metodami, které z těchto komponent využívá.



Obrázek 7

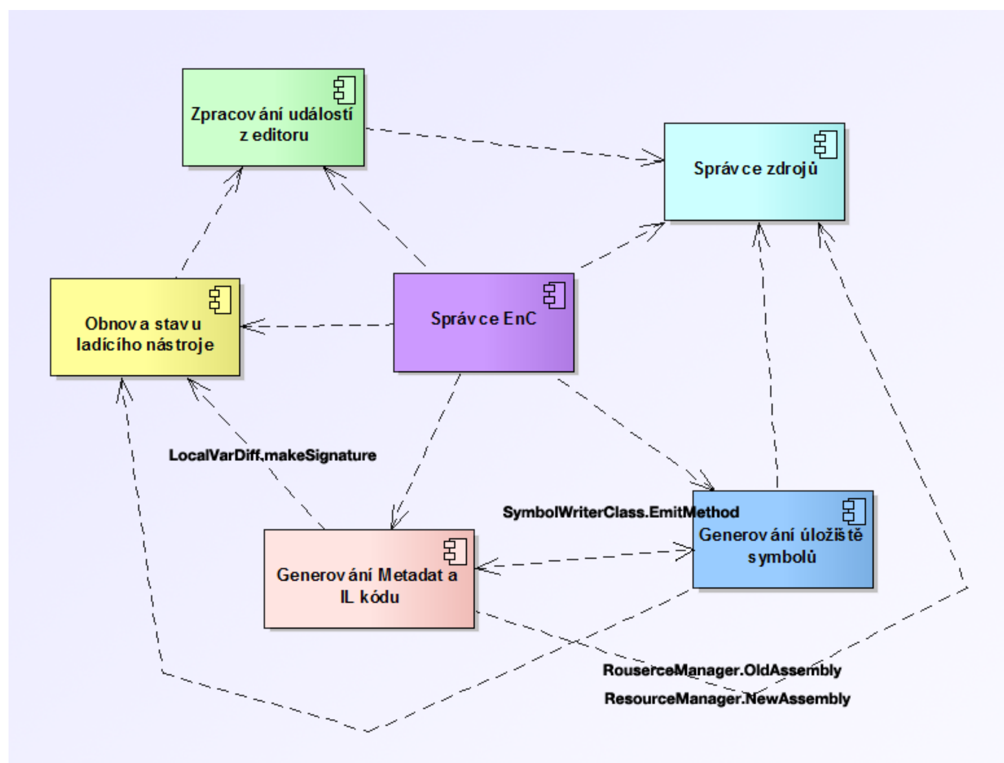
Samotný plugin pro SharpDevelop reprezentuje třída `EnCStarter`. Ta zachytává události o změně stavu ladícího nástroje. Při spuštění ladění některého z procesů je v případě, že je modul aktivován, vytvořena instance třídy `EnCManager`. Ta se pomocí rozhraní `IEnCHook` registruje jako příjemce událostí přímo z modulu ladícího nástroje. Mezi události, které zachycuje, patří například událost `FunctionRemapOpportunity`, která byla zmíněna v kapitole 2.3.2. Při zpracování této události volá stejnojmennou metodu instance třídy `FunctionRemapper` z komponenty pro obnovu stavu ladícího nástroje (viz kapitola 3.1.6.). Tuto instanci získá z komponenty pro zpracování událostí z editoru pro každou z upravených metod během doby zastavení ladění (viz kapitola 3.1.5.).

Správce EnC také aktivuje přichycení komponenty pro zpracování událostí z editoru na události editoru prostředí SharpDevelop pomocí metody `Attach` instance třídy `EditorEventCreator`, která komponentu pro zpracování událostí z editoru reprezentuje. To učiní ve chvíli, kdy komponenta správce EnC obdrží událost pozastavení ladění z ladícího nástroje. Po obdržení události pokračování ladění odpojí správce EnC komponentu pro zpracování událostí z editoru od událostí editoru pomocí metody `Detach` stejné instance. Z instance třídy `EditorEventCreator` potom pomocí metody `GetChanges` obdrží instanci třídy `EditorEvent`, která popisuje změny, které byly v projektu učiněny.

Pokud byly v projektu změny, spustí správce EnC kompilaci nové verze assembly pomocí metody `Load` instance třídy `ResourceManager`, která reprezentuje komponentu pro správu zdrojů (viz kapitola 3.1.3.). Pokud nastala chyba při kompilaci, je ladění ukončeno a chyby vypsány v seznamu chyb nastalých při kompilaci, který prostředí SharpDevelop zobrazuje. Pokud vše proběhlo v pořádku, předá správce EnC změny, které učinil uživatel v projektu, třídě `DeltaBuilder` pomocí metod `ChangeMethod`, `AddMethod`, `PreAddMethod` a `AddField`. Tato třída je jednou z páteřních tříd komponenty pro generování IL kódu a Metadat (viz kapitola 3.1.2.). Správce EnC poté z této třídy získá pole bajtů reprezentující delta IL kód a delta Metadata. Ty nakonec aplikuje pomocí metody `ApplyChanges` z rozhraní `ICorDebugModule2`, která byla zmíněna v kapitole 2.1. Referenci na instanci tohoto rozhraní získá pomocí vlastnosti `CurrentModule` z instance třídy `ResourceManager`.

3.1.2. Komponenta pro generování IL kódu a Metadat

Komponenta pro generování Metadat a IL kódu se stará o překládání IL kódu a Metadat z nové assembly do kontextu běžící assembly tak, abychom získali struktury popisující změny v projektu ve formátu, který vyžaduje CLR, jak bylo popsáno v kapitole 2.2. Komponenta je zastřešena třídou `DeltaBuilder`, které se stará o provedení změn v kódu. Třída `DeltaBuilder` předzpracuje změnu v kódu na vhodná volání metod `TranslateMethod`, `TranslateMethodNew` a `TranslateMethodWithoutBody` třídy `MethodTranslator`. Ty vrátí



Obrázek 8

hotový útržek IL kódu pro danou metodu. Z těchto útržků je sestaven finální delta IL kód, který je vrácen správci EnC. Potom je nová verze metody zanesena do úložiště symbolů pomocí metody `EmitMethod` instance třídy `SymbolWriterClass` z komponenty pro generování úložiště symbolů (viz kapitola 3.1.4.). Obrázek 8 ukazuje vztahy komponenty pro generování Metadat a IL kódu s ostatními komponentami, spolu s třídami, metodami a vlastnostmi, které z těchto komponent využívá.

Třída `MetaDataManager` slouží k převádění tokenů z assembly A do assembly B a v případě nutnosti vytváří nové definice, jak bylo popsáno v kapitolách 2.2.1. a 2.2.2. To vše pomocí instance rozhraní `IMetaDataImport` pro novou verzi assembly a instance rozhraní `IMetaDataEmit` pro laděnou assembly. Pro potřeby aplikování změn umožňuje kdykoliv získat aktuální útržek Metadat pomocí metody `SaveDeltaToMemory` v rozhraní `IMetaDataEmit`.

Překlad tokenů je realizován pomocí metody `TranslateToken`. Metoda dostane jako parametr token z assembly A a vrátí odpovídající token z assembly B. Některé tokeny potřebují pro svoji registraci další Metadata tokeny, v takovém případě překládá metoda rekurzivně. V případě, že by mohlo dojít k zacyklení a také z důvodu zvýšení efektivity, má třída `MetaDataManager` cache s již převedenými tokeny.

Třída `MethodTranslator` překládá IL kód metody z assembly A do assembly B. K překladu tokenů používá metodu `TranslateToken` instance třídy `MetaDataManager`. Pro přesun lokálních proměnných, který byl diskutován v kapitole 2.4, používá metodu `makeSignature` instance třídy `LocalVarDiff` z komponenty pro obnovení ladicího nástroje (viz kapitola 3.1.6.).

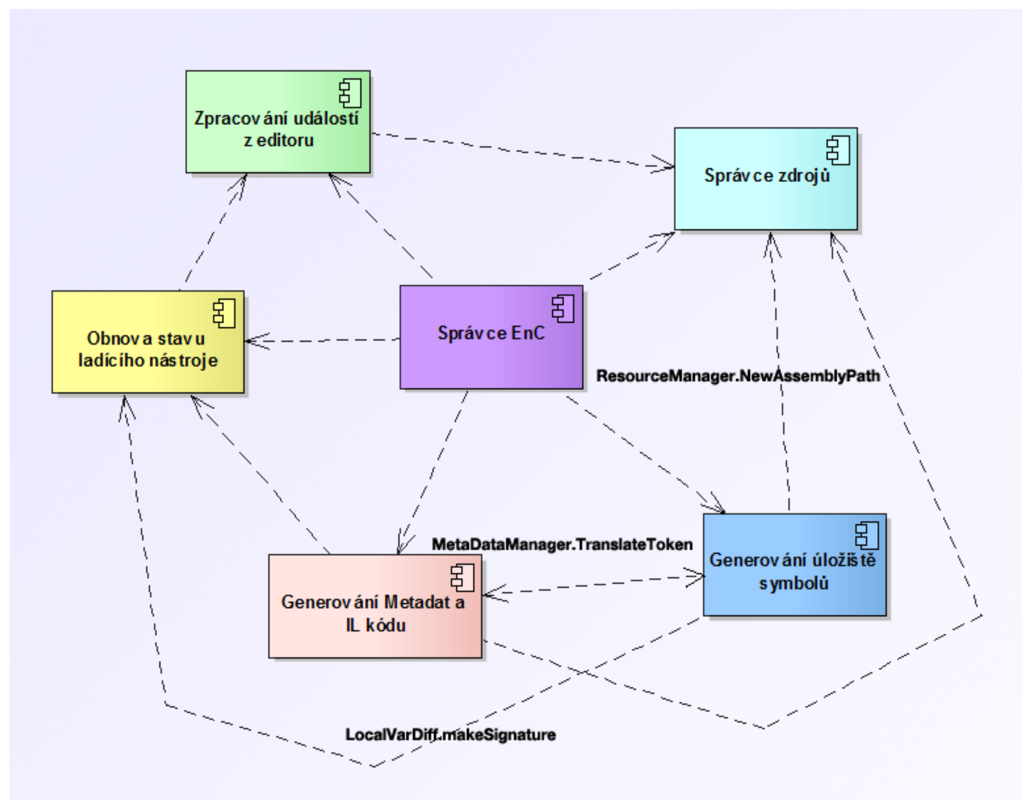
3.1.3. Komponenta pro správu zdrojů

Komponenta sestává z třídy `ResourceManager` a slouží k načtení běžící verze assembly pomocí knihovny `Mono.CECIL`. Drží reference na instance rozhraní `ICorDebugProcess` a `ICorDebugModule`, které odpovídají laděné assembly. Inicializuje kompilaci nové verze assembly pomocí metody `RecompileWithName` instance třídy `CSharpBackgroundCompiler` z projektu `CSharpBackendBinding`. Novou assembly, po úspěšné kompilaci, načte pomocí `Mono.CECIL`. Po neúspěšné kompilaci vypíše do seznamu chyb kompilace prostředí SharpDevelop chyby, které při kompilaci nastaly.

3.1.4. Komponenta pro generování úložiště symbolů

Komponenta k vytváření úložiště symbolů získává novou verzi úložiště

a pro požadované metody, které byly předány metodě `EmitMethod`, načte symboly a přeloží tokeny v signaturách proměnných pomocí instance třídy `Signature`, která využívá metodu `TranslateToken` z instance třídy `MetaDataManager`. Poté přesune proměnné na základě informace získané pomocí metody `makeSignature` z instance třídy `LocalVarDiff` z komponenty pro obnovu ladicího nástroje (viz kapitola 3.1.6.) a vygeneruje nové úložiště, obsahující jen změněné metody. Výsledné úložiště je uloženo do bajtového proudu v paměti, který je reprezentován pomocí implementace rozhraní IStream - proudu pro COM objekty. Touto implementací je třída `CorMemStream`, její instanci využije správce EnC (viz kapitola 3.1.1.) při volání metody `UpdateSymbolStore` z rozhraní `ISymUnmanagedReader` pro aktualizaci symbolů laděné assembly po aplikování, jak bylo zmíněno v kapitole 2.3.1. Obrázek 9 ukazuje vztahy komponenty pro generování úložiště symbolů s ostatními komponentami, spolu s třídami, metodami a vlastnostmi, které z těchto komponent využívá.

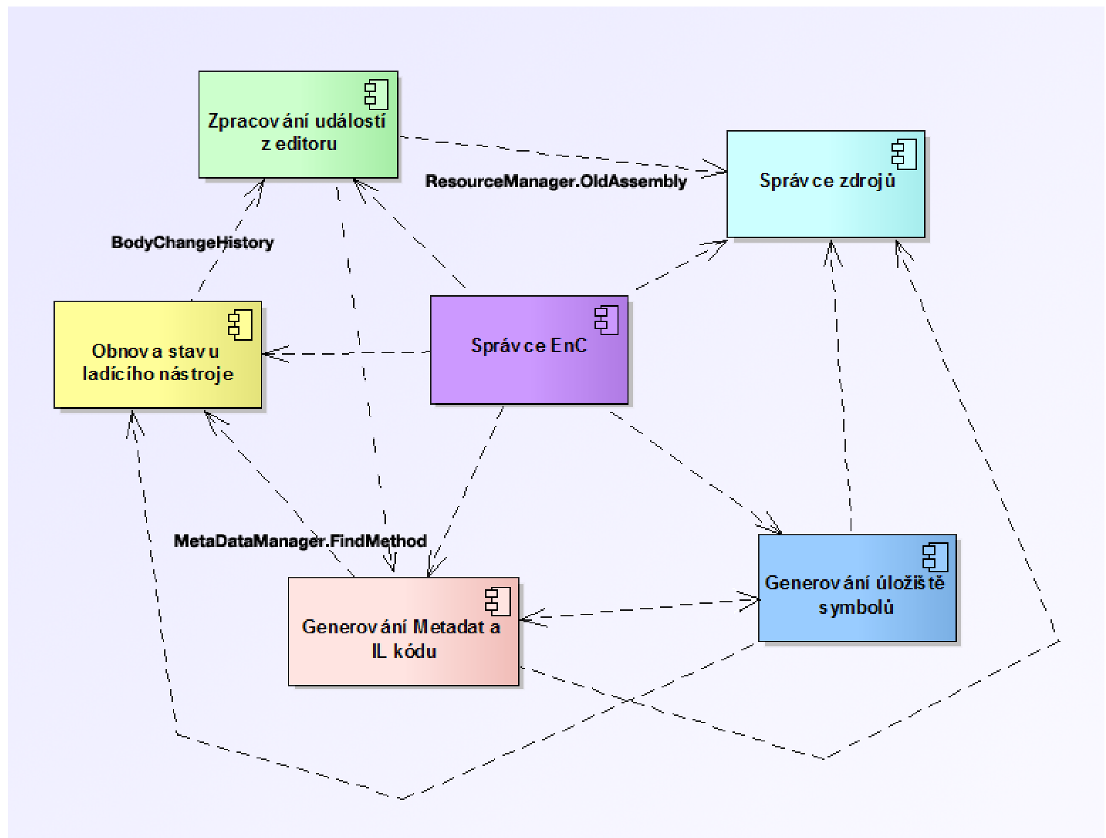


Obrázek 9

3.1.5. Komponenta ke zpracování událostí z editoru

Komponenta ke zpracování událostí z editoru je reprezentována třídou `EditorEventCreator`. Ta je po zavolání metody `Attach` registrována na události, které jí umožní zachytit všechny změny v kódu (včetně vybrání jiného souboru k editování apod.). Když editor zachytí událost pokusu změnit metodu, je třeba vytvořit záznam o posunu sekvenčních bodů (jak bylo

zmíněno v kapitole 2.3). Tuto změnu popisuje instance třídy `BodyChangeHistory`. Ta je přiřazena k tokenu změněné metody pomocí metody `FindMethod` z třídy `MetaDataManager` z komponenty pro generování IL kódu a Metadat (viz kapitola 3.1.2.) a ta vyžaduje instanci třídy `AssemblyDefinition` jako popis laděné assembly. Tu získává třída `EditorEvenCreator` z vlastnosti `OldAssembly` třídy `ResourceManager` z komponenty pro správu zdrojů (viz kapitola 3.1.3.). Obrázek 10 ukazuje vztahy komponenty ke zpracování událostí z editoru a komponenty pro obnovu ladicího nástroje s ostatními komponentami, spolu s třídami, metodami a vlastnostmi, které z těchto komponent využívají.



Obrázek 10

3.1.6. Komponenta pro obnovu ladicího nástroje

Komponenta pro obnovu ladicího nástroje se stará o přepočítávání z pozice sekvenčního bodu ve starém kódu na pozici stejného sekvenčního bodu v novém kódu, jak již bylo zmíněno v kapitole 2.3.2., a o přesun lokálních proměnných (viz kapitola 2.4.). Přesun lokálních proměnných zajišťuje metoda `makeSignature` z instance třídy `LocalVarDiff`. Přepočítání pozic sekvenčních bodů pro danou metodu zajišťuje metoda `TranslateILOffset` instance třídy `SequencePointRemapper`. K vytvoření této instance je potřeba instance třídy `BodyChangeHistory`, která je vytvořena pomocí komponenty pro zpracování událostí z editoru (viz

kapitola 3.1.5.).

Třída `FunctionRemapper` disponuje instancemi třídy `SequencePointRemapper` pro všechny změněné metody a pro metodu, u které došlo k události `FunctionRemapOpportunity`, volá metodu `TranslateILOffset` z instance třídy `SequencePointRemapper`, která k této metodě náleží.

3.1.7. Objektový návrh

Kompletní objektový návrh projektu je v příloze na CD spolu s dokumentací jednotlivých tříd v anglickém jazyce. Třídy v souboru `ObjektovyNavrh.png` ,který je na přiloženém CD, jsou podbarveny barvou, která odpovídá barvě komponenty, ke které patří, v souboru `NavrhKomponent.png`.

3.2. Změny v SharpDevelopu

Pro realizování EnC bylo potřeba provést některé změny i v prostředí SharpDevelop. Bylo například nutné opravit knihovnu Mono.CECIL. Knihovna ve svých strukturách zahazuje některé informace, které jsou pro generování kódu velice důležité. Pro metody bylo nutné získat token, který odkazuje na informace o lokálních proměnných dané metody. Potom bylo nutné zjistit, zda daná metoda používá zkrácený zápis, či nikoliv. Další částí prostředí, která vyžadovala úpravy, které popíšu v další podkapitole, byl kompilátor.

3.2.1. Změny nutné k obnově ladícího nástroje

Ladicí nástroj prostředí SharpDevelop nebyl připraven na to, že se struktura a kód aplikace může během ladění změnit. Proto byla potřeba jeho vnitřní reprezentaci těchto struktur obnovit. Jednalo se především o smazání pamětí cache, které ladicí nástroj používá pro optimalizaci rychlosti ladění. Bylo nutné přidat do třídy `AppDomain`, reprezentující aplikační doménu, veřejnou metodu `ResetCache`, která smaže cache členských entit tříd. To je nutné v případě přidání nové metody.

Dále bylo potřeba obnovit seznam lokálních proměnných metody. K tomu slouží nová veřejná metoda `ResetLocalCache` třídy `DebugMethodInfo`. Ta smaže cache s lokálními proměnnými, pro případ, že byl jejich seznam změněn po změně kódu metody. Další novou metodou je

veřejná metoda třídy `StackFrame` s názvem `RecreateSteppers`. Ta projde všechny staré, neplatné steppery a znovu je vytvoří tak, aby bylo možné pokračovat v krokování aplikace. Nutnost tohoto kroku byla popsána v kapitole 2.3.3.

Všechny tyto metody jsou volány po obdržení události `FunctionRemapOpportunity` na instancích získaných pro aktuální zásobník volání.

3.2.2. Kompilátor

Jednou z otázek při implementaci projektu byla ta jaký kompilátor použít ke kompilaci nové verze assembly. Po diskuzi s jedním z vývojářů SharpDevelopu jsem se dozvěděl, že v aktuální verzi je zahrnut pokus o kompilátor, který má sloužit k opakovanému kompilování aktuálního projektu na pozadí. Smysl takového kompilátoru tkvěl v možnosti nalézt chyby, které byly odhaleny až po kompilaci přímo při editování projektu.

Třída, která realizuje toto kompilování, se jmenuje `CSharpBackgroundCompiler`. Po analýze použitého kódu jsem zjistil, že kompilátor dělí projekt na dvě části. Zdrojový soubor, který byl právě editován, byl kompilován standardně. U ostatních zdrojových souborů byl kód vygenerován z informací parseru pro C#, který má SharpDevelop k dispozici. Jak bylo zmíněno v kapitole 2.5, byla v takto vygenerovaných zdrojových souborech nahrazena těla metod klíčovým slovem extern. Tedy hotovou assembly nelze ze zřejmých důvodů spustit, ale metody, které jsou v editovaném zdrojovém kódu, v ní jsou, včetně IL kódu a úložiště symbolů.

3.3. Problémy spojené s implementací

Implementace celého projektu je náročná především proto, že spousta volání je učiněno pomocí COM [15] objektů a tzv. Marshallingu. Díky tomu lze volat součásti CLR, které jsou přímo ve strojovém kódu. Problém je, že zachycení událostí nebo získání popisu chyby, která nastala při nebo po aplikaci změn je prakticky nemožné. Při volání metod objektů CLR je těžké určit, jestli je chyba ve způsobu volání, ve špatném formátu parametrů nebo Marshallingu dané metody.

Směr, kterým se projekt mohl ubírat, aby tento problém alespoň zčásti řešil, byl dán využitím knihovny `Mono.CECIL` pro úpravu a ukládání IL kódu a Metadat. Knihovna umožňuje načítání serializovaných Metadat a IL kódu v assembly, jejich reprezentaci a jejich opětovnou serializaci po změně. Nabízí se upravit ji pro potřeby EnC a využít. Nevýhoda této knihovny je v tom, že pracuje na vysoké úrovni abstrakce. Například v instrukcích IL kódu, kde jsou použity Metadata tokeny, knihovna přímo nahrazuje tyto tokeny instancemi tříd, které popisují objekt, na který se token odkazuje.

Některé informace knihovna dokonce vynechává, pravděpodobně proto, že mají význam především pro kompilaci než pro analýzu kódu.

Úprava knihovny byla na některých místech nutná, šlo například o zmiňované vracení tokenu, který odkazuje na úložiště informací o lokálních proměnných dané metody. Další úpravy této knihovny by mohly projekt osvobodit od závislosti na rozhraních CLR a COM objektech, které jsou těmito rozhraními reprezentovány. Tudíž zvýšit rychlost a přenositelnost kódu a snížit náročnost jeho ladění, viz. začátek této podkapitoly. Realizace těchto úprav by však byla nejspíš příliš náročná na čas.

3.3.1. Porovnávání entit

SharpDevelop disponuje vlastním nástrojem na analýzu zdrojového kódu. Ta popisuje entity ve zdrojovém kódu svým vlastním způsobem. Pokud dojde ke změně nějaké entity, je v informaci o změně, reprezentované instancí třídy `SourceChange`, uložena reference na změněnou entitu. Takovou entitu je pak potřeba najít v assembly, kterou popisuje instance třídy `AssemblyDefinition` knihovny `Mono.CECIL`. Tato instance obsahuje také popis všech entit v assembly, ale její formát je trochu odlišný.

Bylo tedy třeba vytvořit nástroj, který dokáže srovnávat tyto dva formáty entit, totiž aby k entitě získané při úpravě kódu bylo možné nalézt entitu pomocí `Mono.CECIL`. To není snadné, protože je potřeba porovnávat celou řadu entit, které souvisí s entitami, u kterých je změna povolena. Navíc je mezi oběma formáty celá řada drobných rozdílů, které se projeví až při použití všech možných konstrukcí jazyka C#.

4. Srovnání s existujícími řešeními

Jak už bylo nastíněno v kapitole 2.8, hlavním rozdílem mezi realizací tohoto projektu a způsobem, kterým Microsoft Visual Studio .NET (dále VS) realizuje EnC, je používání speciálního kompilátoru, který k těmto účelům VS má. Další rozdíl je v chování. VS po neúspěšném pokusu o změnu dovolí v pokračování ladění v „nepředvídatelném“ režimu. To znamená, že kód, kterým je krokováno, nemusí odpovídat kódu, který je spouštěn. Tento plugin se uvedené situaci vyhýbá tím, že ladění automaticky ukončí a ukáže, kde došlo k chybě.

Dalším rozdílem mezi podporou EnC ve VS a tímto projektem je schopnost VS učinit změny i v modulech, které jsou k právě laděnému modulu připojeny. Tuto možnost tento projekt nepodporuje. Její realizace by byla příliš náročná. Možným řešením je spouštět k ladění vždy ten modul, který bude potřebovat úpravy. Srovnání podporovaných změn je vidět na tabulce 1.

	Tento projekt	EnC ve VS 2010	Dispozice CLR
Přidání privátní nevirtuální metody	✔	✔	✔
Přidání privátní členské proměnné	✔	✔	✔
Přidání privátní vlastnosti	✘	✔	✔
Změna kódu metody, která není generická	✔	✔	✔
Změna kódu metody, která není generická, s IP uvnitř	✔	✔	✔
Změna kódu vlastnosti	✔	✔	✔

Tabulka 1

4.1. Srovnání s podporou pro jazyk Java

Podobně jako CLR disponuje i virtuální stroj jazyka Java (Java Virtual Machine, dále jen JVM) podporou pro změnu kódu během ladění. V kontextu jazyka Java se tato metoda nazývá HotSwap. Tato funkcionalita je v JVM

od verze 1.4. Podobně jako u CLR prolíná se tato funkcionalita až do API pro ladicí nástroj Javy. Funkce HotSwap je limitována pouze na změnu těla metody. Některé změny v těle metody bohužel vedou k vygenerování syntetických metod a členských proměnných, čímž znemožní aplikování takové změny.

Pro Javu existuje ještě knihovna `JRebel`, která umožňuje větší škálu změn. Tato knihovna stále monitoruje *.class* soubory projektu, jež jsou v Javě ekvivalentem assembly, vzniklou serializací IL kódu. Tím umožňuje podporu pro změny při ladění kterémukoliv vývojovému prostředí.

5. Závěr

5.1. Možná další využití jednotlivých částí projektu

Jedním z nástrojů potřebných pro aplikaci změn, je nástroj, který umožňuje přenos metody z jedné assembly do druhé. Tento nástroj má další možná využití. Jedním z nich by mohl být speciální druh verzování zaměřený na assembly, kdy by nově přidaná metoda do nové verze assembly mohla být zpětně přesunuta do starší verze. Díky tomu by bylo možné získat podobné delta útržky jako pro EnC. Tyto útržky by pak bylo lze použít pro reprezentaci změn v různých verzích aplikace, což by snížilo nároky na nutný prostor pro uložení několika verzí aplikace.

5.2. Zhodnocení naplnění cílů

Pokud se týká cílů průzkumných, tak se podařilo zjistit, že EnC lze implementovat i pro jiná prostředí než je Microsoft Visual Studio .NET. Dále je zřejmé, že EnC lze realizovat bez nutnosti speciálního kompilátoru pouze pro EnC, tedy jinak, než ho realizuje Microsoft Visual Studio .NET. Práce ukazuje jeden ze způsobů, kterým lze EnC implementovat a obsahuje i samotnou implementaci projektu. Její stabilita není stoprocentní a úzce souvisí s nástroji, na kterých je závislá. Mezi nejkritičtější patří kompilátor na pozadí z prostředí SharpDevelop.

5.3. Možná rozšíření

Práci je dále možné rozšířit na podporu všech změn, které CLR umožňuje, není to otázka klíčových zásahů do projektu jako spíše času k implementaci. Také lze zvýšit její stabilitu vhodnými úpravami kompilátoru SharpDevelopu nebo nalezením vhodnějšího kompilátoru pro potřeby EnC. Jak bylo zmíněno výše, lze práci „osvobodit“ od závislosti na rozhraních CLR pomocí efektivnějšího využívání knihovny `Mono.CECIL.`

5.3.1. Tlačítko Step Back

Jako jedno z možných rozšíření tohoto projektu se nabízí tlačítko „Step Back“. Tento nástroj by byl dalším druhem krokování v ladicím nástroji. Jednalo by se o krok vzad. Ten lze ovšem realizovat pomocí posunutí IP směrem nahoru v kódu. Přínos kroku zpět by byl v postupném navracení paměťového kontextu do výchozího stavu. Pro každou řádku, pro kterou by byl tento krok učiněn, by byly vráceny změny v paměti, které provedení této řádky způsobilo.

Díky tomu by bylo možné lépe realizovat změny pomocí EnC tak, aby se předešlo nereprodukovatelným stavům laděné aplikace. Pokud uživatel nalezne chybu na místě, kde by oprava způsobila nereprodukovatelný stav aplikace, může se pomocí tlačítka Step Back vrátit, aby mohl chybu opravit bezpečně.

Seznam použité literatury

1) Microsoft. *Microsoft .NET Framework 4 (webová instalační služba)*[online]. Dostupný z: http://www.microsoft.com/cs-cz/download/details.aspx?id=17851 [cit. 21.2.2012].

2) ECMA 335:2001. *Common Language Infrastructure (CLI)*,ECMA International.

3) Microsoft. *Microsoft Developer Network: Introducing Visual Studio*[online]. Dostupný z: http://msdn.microsoft.com/en-us/library/fx6bk1f4%28v=vs.100%29.aspx [ver. 2010].

4) ic#code. *SharpDevelop*[online]. Dostupný z: http://www.icsharpcode.net/OpenSource/SD/Download/ [ver. 4.2 beta 2].

5) Mono Project. *Mono*[online]. Dostupný z: http://www.mono-project.com/ [ver. 2.10.X].

6) Serge Lidin. *Expert .NET 2.0 IL Assembler*. Apress: 2006. ISBN 978-1-59059-646-3.

7) Serge Lidin. *Inside Microsoft® .NET IL Assembler*. Microsoft Press: 2002. ISBN 0-7356-1547-0.

8) Microsoft. *Microsoft Developer Network: ICorDebugModule2::ApplyChanges Method*[online]. Dostupný z: http://msdn.microsoft.com/cs-cz/library/ms231880.aspx [ver. .NET Framework 4].

9) Stall Mike. *Mike Stall's .NET Debugging Blog:3rd-parties and Edit And Continue (Part 2: Debuggers)*[online]. Dostupný z: http://blogs.msdn.com/b/jmstall/archive/2005/02/19/376666.aspx [cit. 19.2.2005].

10) Microsoft. *CLR Managed Debugger (mdbg) Sample 4.0*[online]. Dostupný z: http://www.microsoft.com/en-us/download/details.aspx?id=2282 [cit. 31.1.2011].

11) Evian Jb. *Mono.Cecil 0.9 "cecil/light"* [online]. Dostupný z: http://evain.net/blog/articles/2010/04/12/mono-cecil-0-9-cecil-light [cit. 12.4.2012].

12) Microsoft. *Microsoft Developer Network: SymReader.UpdateSymbolStore Method*[online]. Dostupný z: http://msdn.microsoft.com/cs-cz/library/system.diagnostics.symbolstore.symreader.updatesymbolstore.aspx [ver. .NET Framework 4].

13) Stall Mike. *Mike Stall's .NET Debugging Blog:3rd-parties and Edit-and-Continue (Part 1: Editors + Compilers)*[online]. Dostupný z: http://blogs.msdn.com/b/jmstall/archive/2005/02/17/375385.aspx [cit. 17.2.2005].

14) Microsoft. *Microsoft Developer Network: Supported Code Changes (C#)* [online]. Dostupný z: http://msdn.microsoft.com/en-us/library/ms164927.aspx [ver. .NET Framework 4].

15) Microsoft. *Microsoft Developer Network: Interop Marshaling*[online]. Dostupný

z: http://msdn.microsoft.com/en-us/library/eaw10et3.aspx [ver. .NET Framework 4].

Obsah CD

1) `repozitare\`

   Složka obsahující GIT repozitáře se změnami kódu v SharpDevelopu a s kódem modulu.

2) `repozitare\EnC`

   Složka obsahující GIT repozitář s kódem modulu pro EnC.

3) `repozitare\CSharpBinding`

   Složka obsahující GIT repozitář s kódem kompilátoru zachycující změny proti původní verzi v SharpDevelopu.

4) `repozitare\Mono.Cecil`

   Složka obsahující GIT repozitář s kódem knihovny Mono.Cecil zachycující změny proti původní verzi v SharpDevelopu.

5) `repozitare\NRefactory`

   Složka obsahující GIT repozitář s kódem modulu NRefactory, který slouží pro generování a parsování kódu a používá jej kompilátor SharpDevelopu. Repozitář zachycujíce změny proti původní verzi v SharpDevelopu.

6) `repozitare\Debugger.Core`

   Složka obsahující GIT repozitář s kódem modulu ladícího nástroje, který repozitář zachycujíce změny proti původní verzi v SharpDevelopu.

7) `repozitare\Umisteni v projektu SharpDevelopu.txt`

   Soubor obsahuje popis umístění výše zmíněných složek ve složce obsahující zdrojové kódy projektu SharpDevelop.

8) `sharpdevelop\`

   Složka obsahující spustitelnou verzi prostředí SharpDevelop s integrovaným modulem pro EnC.

9) `sharpdevelop\bin\SharpDevelop.exe`

   Spustitelný soubor reprezentující prostředí SharpDevelop.

10) `NavrhKomponent.png`

    Obrázek popisující vztah komponent

11) `ObjektovyNavrh.png`

    Obrázek obsahující objektový návrh projektu.

12) `UzivatelskaDokumentace.pdf`

Uživatelská dokumentace popisující používání modulu.

13) `dokumentace_doxygen\`

Dokumentace vytvořená z dokumentačních nástrojů pomocí nástroje doxygen.

14) `dokumentace_doxygen\index.html`

Vstupní bod do dokumentace.

15) `ElektronickaVerzePrace.pdf`

Elektronická podoba tohoto dokumentu.