```
class A {
  //virtuální metoda, která
  //může být přepsána v potomkovi
  public virtual void M1()

  //metoda, která virtuální není
  public void M2()
}
```

```
class B : A {

  //přepíšeme metodu A.M1
  public override void M1()

  //tato metoda NEpřepíše A.M2
  public void M2()
}
```

```
void Test(A obj)
{
  //volaná metoda je určena
  //na základě typu obj
  //v našem případě buď
  //A.M1 nebo B.M1
  obj.M1();
}
public void Usage()
{
  //způsobí vyvolání A.M1
  Test(new A());
  //způsobí vyvolání B.M1
  Test(new B());
}
```

2-9 Ukázka vlastností virtuálních volání v jazyce C#.

Všimněme si metody `Usage`, která volá `Test` s různým typem objektů. Kvůli tomu dochází postupně k vyvolání metod `A.M1` a `B.M1`.

Rozlišování metod podle volaného objektu však nelze provádět vždy. Kdybychom v metodě Test použili `obj.M2`, dostali bychom v metodě `Usage` sekvenci volání `A.M2`, `A.M2`, neboť metoda `M2` není virtuální.

Z tohoto důvodu musíme mít možnost identifikátor označit příznakem, zda je nutné metodu vyhledávat na základě jejích argumentů až v průběhu interpretování. Poznamenejme, že objekt, na němž je metoda volána, je pro instrukci *call* také argumentem. Díky tomu dokážeme simulovat virtuální i nevirtuální volání.

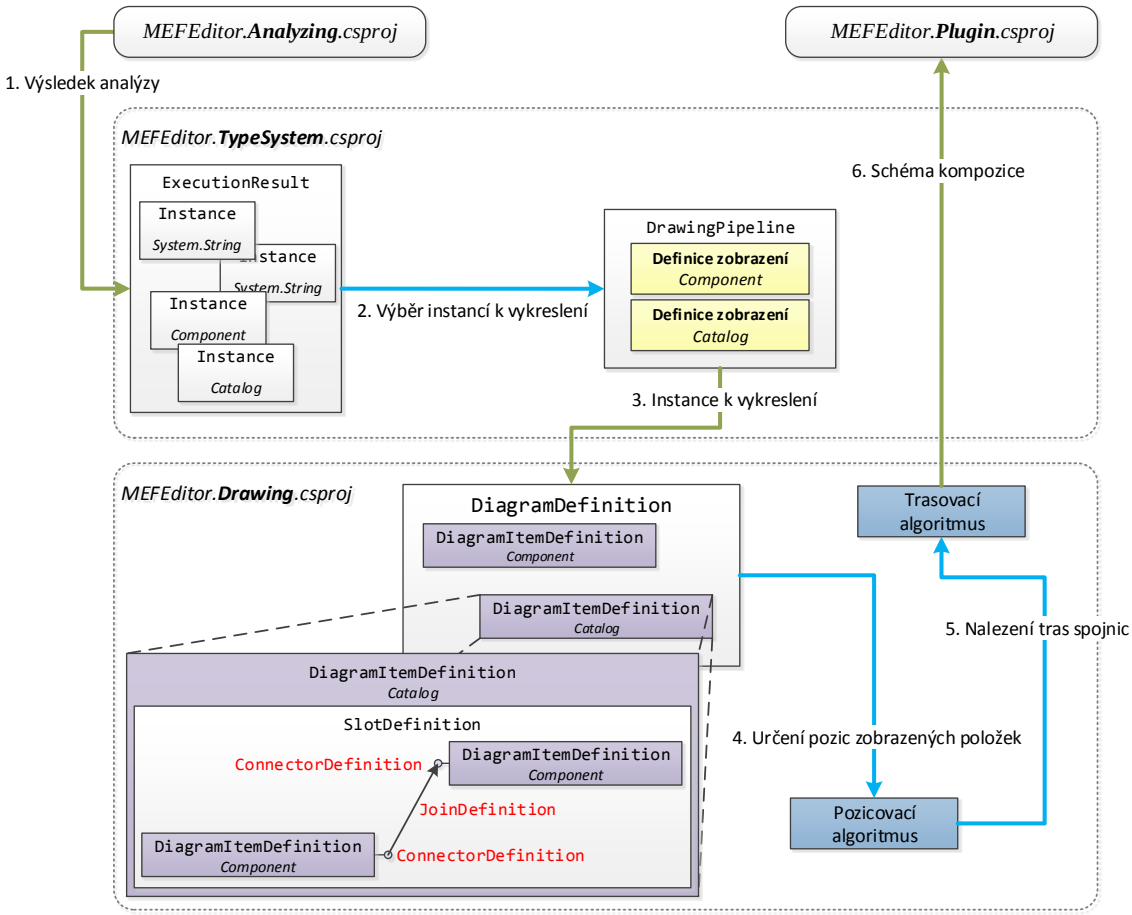
Instrukce pro statická volání

Statická volání jsou prováděna na objektu sdíleném mezi všemi statickými voláními na stejné třídě. Inicializace těchto objektů je prováděna statickým konstruktorem pouze jednou a to těsně před prvním voláním statické metody na třídě. Abychom mohli simulovat i tento druh volání, musíme být schopni zjistit, zda byla *sdílená instance* odpovídající sdílenému objektu již inicializována. Tak zajistíme, že se inicializace neprovede vícekrát. Pro tyto účely bude virtuální stroj nabízet globální proměnné, které budou na rozdíl od běžných proměnných uchovávat *instance* i napříč voláním metod. Díky tomu můžeme zjistit, zda byla *sdílená instance* již vytvořena, nehledě na metodu kde k tomu došlo.

Vlastní vytvoření *sdílené instance* je nutné provést těsně před instrukcí *call*, která reprezentuje nějaké statické volání. Pro tyto účely přidáme do analyzačních instrukcí instrukci *ensure_init*, která má operand určující cílovou globální proměnnou a operand s identifikátorem metody, která provede inicializaci v případě, že globální proměnná ještě inicializovaná není. Umístěním *ensure_init* před *call* se statickým voláním tedy můžeme simulovat statická volání tak, jak probíhají v .NET.

To by však nebylo dostatečné pro vykreslování komponent, které obvykle chceme zobrazit nezávisle na jejich typu. Proto *instance*, které jsou komponentami a nemají explicitně určenou definici zobrazení, vykreslíme pomocí obecného vykreslovacího rozšíření.

Než je možné výsledek zobrazit uživateli, je však nutné nejprve upravit pozice zobrazených položek a také nalézt neprotínající trasy pro spojnice. Popsaný proces je znázorněn na následujícím obrázku 4-8:

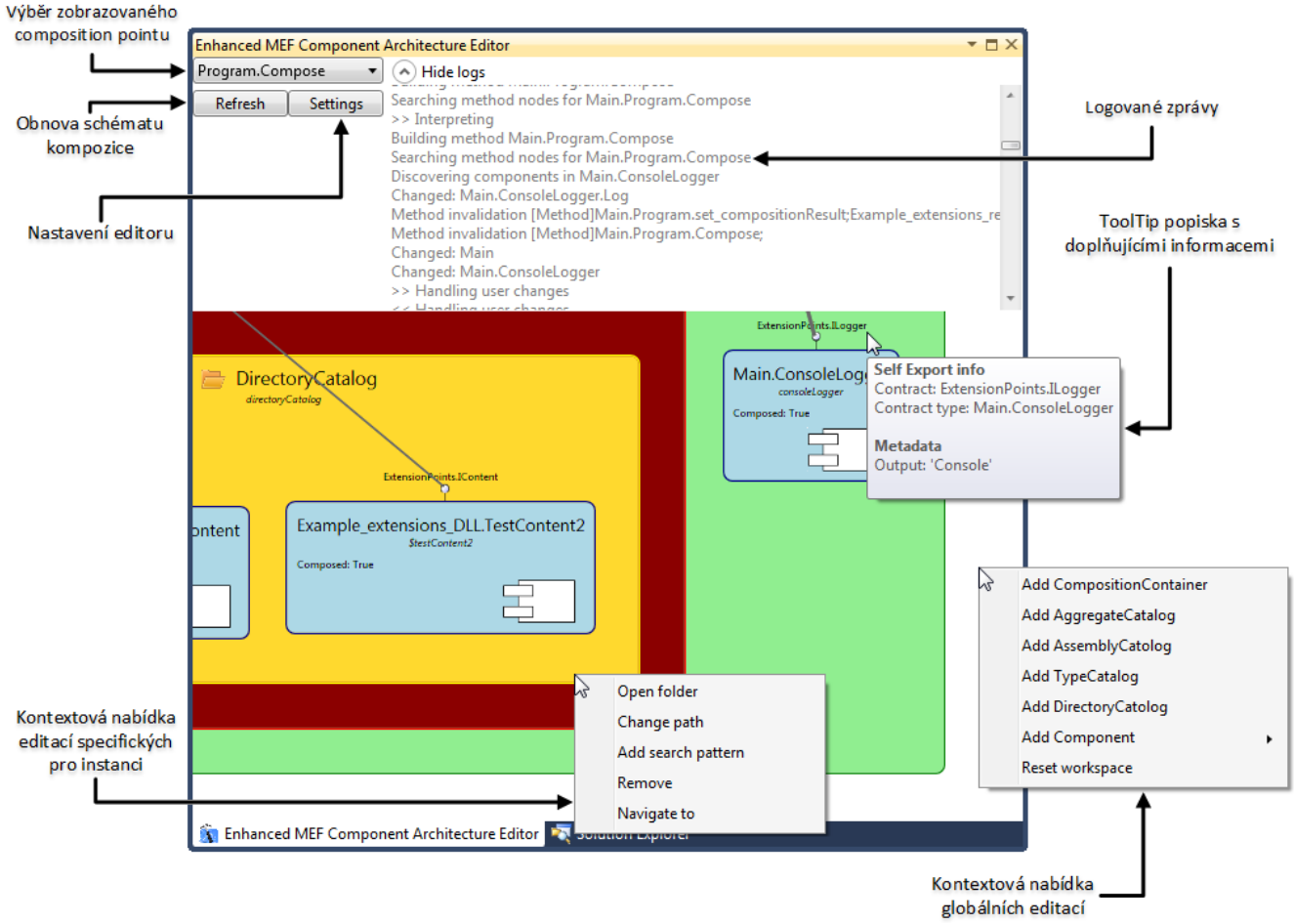


4-8 Proces vykreslení schématu kompozice z výsledku analýzy composition pointu.

Vykreslení *instance* v předchozí verzi editoru vycházelo přímo ze zkoumání stavu této *instance*. To s sebou však neslo problémy, které souvisely s přílišnou provázaností interní implementace definic typů s vykreslovacími rozšířeními. Editor kvůli tomu také nebyl schopen řešit případy s kruhovou závislostí mezi zobrazenými *instancemi*.

Z těchto důvodů je současná implementace navržena mnohem deklarativněji. Z *instancí*, které chceme vykreslit, nejdříve necháme vytvořit objekty třídy `DiagramItemDefinition`, které jsou poskytovány *typovou definicí* každé *instance*. Ve vytvořené definici zobrazení jsou uchovány informace, sloužící jako podklady pro práci vykreslovacího rozšíření. Tento přístup znamená, že schéma kompozice dokážeme vytvořit v implementačně nezávislé formě, ve které například vyřešíme kruhové závislosti, a až na jejím základě sestrojujeme samotné grafické reprezentace *instancí*.

5.2 Uživatelské rozhraní



5-1 Popis pracovní plochy uživatelského rozhraní

Editor zobrazuje schéma kompozice na základě právě vybraného *composition pointu*. Po vybrání se zobrazí patřičné schéma kompozice nebo bude vypsána chyba, ke které došlo v průběhu analýzy, a kvůli které nebylo možné schéma kompozice vykreslit. Kontextová nabídka u vypsané chyby obsahuje příkaz pro zkopírování textové reprezentace chyby do schránky, případně může také obsahovat příkaz pro navigaci na místo ve zdrojovém kódu, kde k chybě došlo.

Spouštění editací

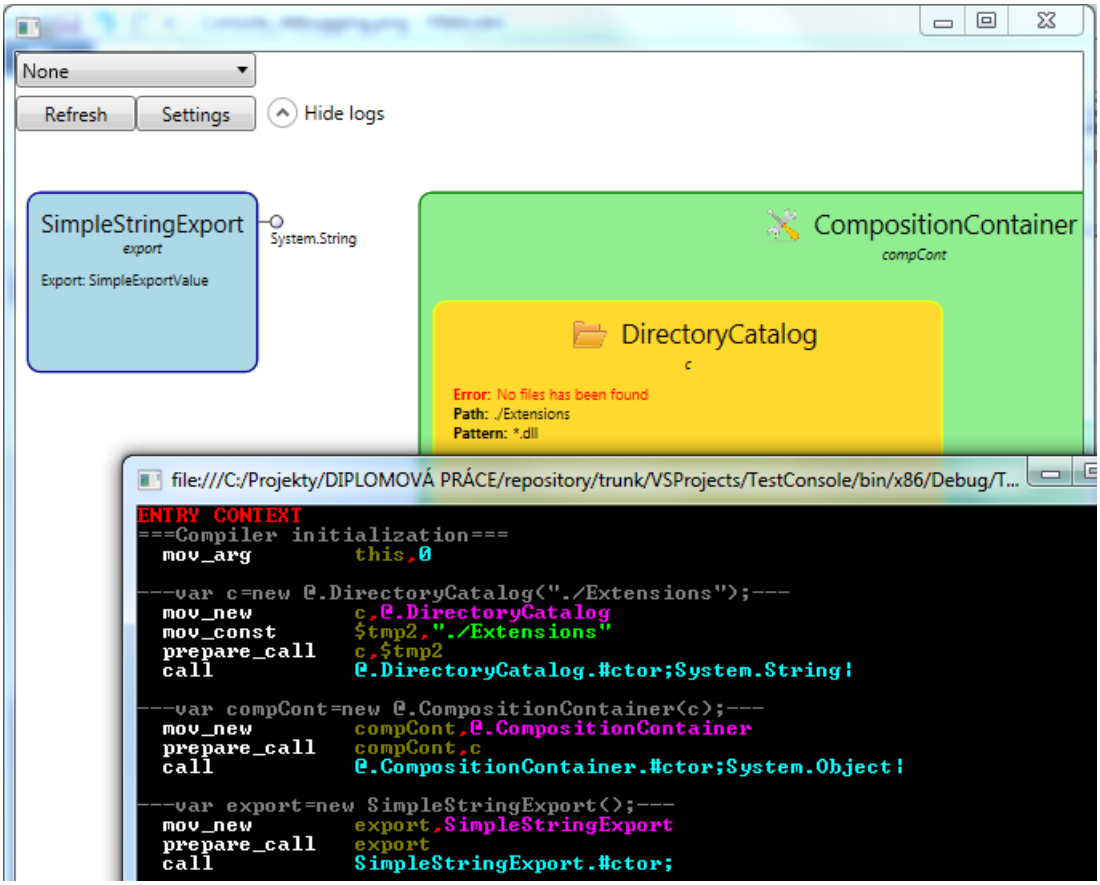
V zobrazeném schématu kompozice jsou editace prováděny pomocí kontextových nabídek zobrazených *instancí*. Dalším způsobem jak lze editace provádět je drag&drop akce, kdy požadovanou *instanci* zkusíme přesunout do prostoru nějakého katalogu, kontejneru nebo na volnou plochu schématu kompozice. Pokud je editace možná, dojde po drop akci k zapsání změn do zdrojového kódu.

Editace, které nejsou vázané na konkrétní *instanci*, se zobrazují v kontextové nabídce volné plochy schématu kompozice.

Změna zobrazení schématu kompozice

Schéma kompozice může uživatel přizpůsobovat pro lepší přehlednost. Je možné měnit přiblížení schématu kompozice pomocí rolování kolečka myši. Uživatel

Výstup konzole i uživatelského rozhraní můžeme vidět na následujícím obrázku:



6-31 Ukázka uživatelského rozhraní editoru (vzadu) spuštěného mimo Visual Studio a konzole (vpředu) s ladícími informacemi o analyzovaných metodách.

Jelikož je konzolová aplikace určená pro pomoc při vývoji rozšíření editoru, je způsob jejího použití založen na přímých úpravách zdrojových kódů. Díky tomu nepřijdeme o možnost ladění za pomoci běžného debuggeru *Visual Studia*, jako kdybychom načítali vyvíjená rozšíření v podobě zkompilovaných knihoven.

V této kapitole si předvedeme, jak projekt s konzolovou aplikací použít na ladění rozšiřující knihovny implementované v předcházejících kapitolách. Veškerý kód, který zde naimplementujeme je již dostupný v solution *TestConsole.sln* z přílohy [C].

Pro náš příklad budeme testovat, zda se instance reprezentující objekt typu `Diagnostic` zobrazí ve schématu kompozice a zda bude nabízet patřičné editace. Implementujeme tedy metodu, která patřičný test provede. K dispozici máme stejné nástroje, které jsou dostupné v testovacím frameworku popsaném v kapitole 4.6. Místo spouštění testu ale budeme v testovací metodě vytvářet objekt `TestingAssembly`.

Abychom mohli v projektu konzolové aplikace použít třídy implementované v rozšiřující knihovně, musíme na ni přidat referenci. Poté do statické třídy `TestCases` přidáme testovací metodu `TestExtensions`. Zde vytvoříme jednoduchou komponentu spolu s instancí testované `DiagnosticDefinition`.

Testovací metoda bude vypadat následovně: