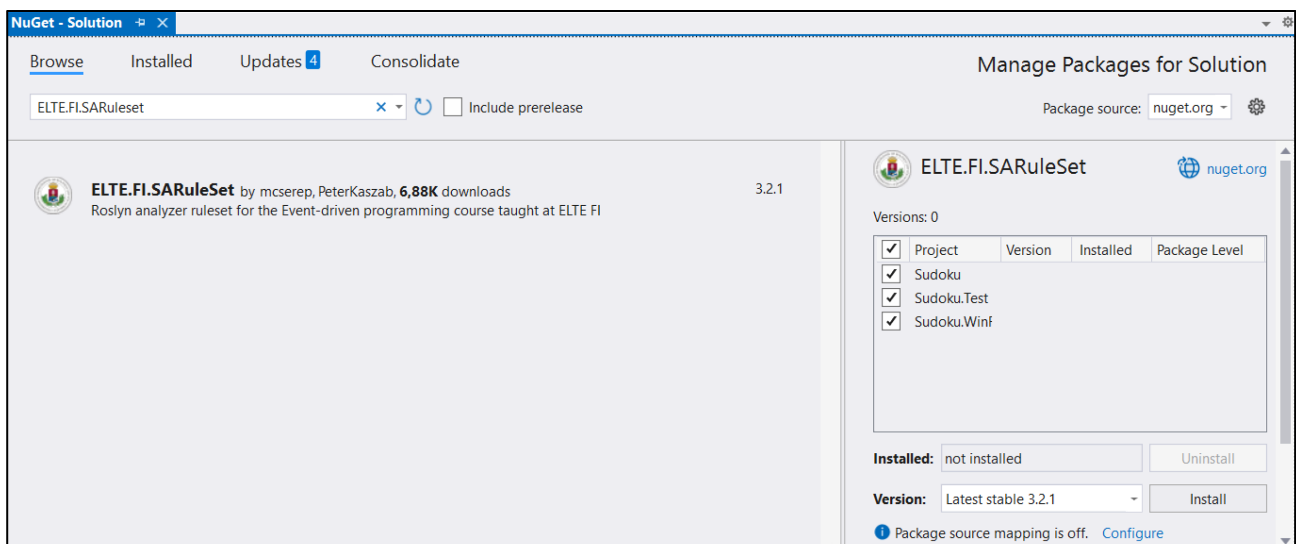
Eseményvezérelt alkalmazások – Statikus analízis

A feltöltött beadandó megoldások nem tartalmazhatnak fordítási hibákat, továbbá a fordító vagy a megadott statikus analízis szabályrendszer által generált figyelmeztetéseket (*warning*) sem. (Az ajánlás, azaz *suggestion* súlyosságú jelzések még megengedettek, de így is célszerű őket megvizsgálni.) Ezt a feltöltést követően a TMS is ellenőrzi és a webesfelületen is megtekinthető a talált hibák listája. Annak érdekében, hogy már fejlesztés közben kiderüljenek a problémák, javasolt a NuGet-csomag telepítése. A TMS által futtatott ellenőrzés hosszabb időt is igénybe vehet!

A fordító által adott figyelmeztetésekről leírás a Microsoft dokumentációjában, a szabályok tételes felsorolása pedig az ezen a linkeken érhető el. A NuGet-csomagban aktív további elemzők listája megtalálható a mellékelt táblázatban.

Használat

A *Solution manager*-ben válasszuk ki a *solution*-t és kattintsunk a *Manage Nuget Packages* menüpontra. Keressünk a rá az `ELTE.FI.SARuleset` csomagra és ügyeljünk arra, hogy minden projekthez adjuk hozzá. Amennyiben a későbbiekben új projekteket adunk a *solution*-hoz ismételjük meg a csomag telepítését azokhoz is.



A fordítás során (*Build*) a statikus ellenőrzés szabályai is automatikusan végrehajtásra kerülnek.

Ellenőrzések

Az alábbiakban kategorizáltan tekintsünk át tipikus előforduló hibalehetőségeket, amelyek elkerülése a beadandók esetén is elvárt és automatizáltan ellenőrzött.

1. Nullable típusok (*Nullable types*)

A táblázatban felsorolt elemzők mellett a beadandók esetében elvárt a *nullable* referencia típusokhoz kapcsolódó hibák javítása, amelyeket a fordító jelez. Ezen figyelmeztetések javításával már a kód írása közben elkerülhetőek azok a hibák, amelyek futási idejű `NullReferenceException` kivételek előfordulásához vezetnének.

2. Konvenciók (*Conventions*)

Ebbe a kategóriába tartoznak azok a figyelmeztetések, amelyek olyan hibákat jeleznek, amelyektől függetlenül a beadott megoldás működése helyes, de nem követi a C# nyelvi konvencióit. Például: nem javasolt a publikus mezők használata *property*-k helyett, az egyes típusokat névterekben kell definiálni, stb.

Szintén idetartoznak az elnevezéshez kapcsolódó problémák, de ezek betartása csak javasolt, nem kötelező. Például: *PascalCase* használata, a típusok ellátása megfelelő utótagokkal.

3. Helytelen értékadások (*Incorrect assignment to variable or field*)

Ezek a hibák már komolyabb problémákat jelezhet a beadott megoldásokban. Példák a jellemző hibákra:

- egész-osztás használata olyan környezetben, ahol lebegőpontos értéket várna a fordító;
- értékadás logikai kifejezésben a `=` és `==` operátorok felcseréle miatt;
- változó vagy mező értékül adása önmagának;
- a mellékhatás mentes függvények visszatérési értekének ignorálása.
- bináris operátorok mind a két oldalán ugyanaz a kifejezés áll (pl. `2 == 2`, `2 - 2`);

4. Teljesítmény és erőforrás-kezelés (*Performance and resource management*)

A C# *string*-ek módosíthatatlan (*immutable*) típusok, ezért minden módosító művelet egy új példány létrehozásával jár. Ha sok műveletet végzünk egy karaktersorozaton (pl. ciklusmagban), akkor javasolt a `StringBuilder` osztály használata.

Bizonyos osztályok használhatnak *unmanaged* erőforrásokat (pl. megnyitott fájlok), ezeket szükséges felszabadítani használat után. Az ilyen osztályok általában megvalósítják az `IDisposable` interfészt, amely megköveteli a `Dispose` (esetenként a `Close` is rendelkezésre áll) metódus meglétét és lehetővé teszi a `using` *statement/declaration* használatát, amelyek esetében a fordító gondoskodik a `Dispose` metódus hívásáról az objektum használata után. Fontos, hogy a felszabadítás után már nem szabad használni az adott objektumot.

5. Feltételes vezérlés, ciklusok és rekurziók (*Conditional executions, iterations and recursions*)

Ebbe a kategóriába tartoznak azok a problémák, amelyek hatására egy adott kódrészlet nem az elvárt számú alkalommal hajtódik végre. Tipikus problémák:

- végtelen rekurzió;
- a ciklusszámláló rossz irányba mozgatása;
- nem-elérhető feltételesen végrehajtott kódrészlet;
- túl korai `return` egy metódusban vagy `break` ciklusban, elérhetetlen kódrészletek;
- ismétlődő ágak `if-else` vagy `switch` esetében;
- Polimorfizmus

A C# csak a virtuális tagok (pl. metódus, *property*) felüldefiniálását teszi lehetővé, amit expliciten jeleznünk kell az `override` kulcsszó segítségével. Ha nem írjuk ki az `override` kulcsszót, akkor csak elrejtjük az eredeti tagot, így az osztályunk esetében a polimorfizmus nem az elvárt módon fog viselkedni. Ha szándékos volt a tag elrejtése, akkor azt szükséges jeleznünk a `new` kulcsszóval, ezzel megerősítve a nem megszokott programozói döntést.

6. Polimorfizmus és öröklődés (*Polymorphism and inheritance*)

Öröklődéskor a polimorfizmus várt működése érdekében kerülni kell az ősosztály metódusainak nem szándékos elfedését vagy a virtuális eljárások konstruktorban történő végrehajtását.

Az `Object` ősosztály `Equals()` metódusának felüldefiniálásakor kötelező a `GetHashCode()` eljárás felüldefiniálása is.

7. Aszinkron programozás (*Asynchronous programming*)

Az aszinkron műveleteket általában szükséges bevárni, – akkor is ha a metódus nem rendelkezik visszatérési értékkel vagy nem szeretnénk felhasználni azt, – hogy biztosak legyünk benne, hogy az objektumaink belső állapota konzisztens marad (pl. mentett játék betöltése közben).

Amennyiben saját aszinkron metódusokat írunk, amelyek nem rendelkeznek visszatérési értékkel, akkor is érdemes a `Task` visszatérési típust használni `void` helyett, mivel a `void` visszatérési típussal rendelkező aszinkron-metódusok nem várhatóak be a felhasználók által és a keletkező kivételek sem kezelhetőek le a hívásuk helyén. Kivételt képeznek ez alól bizonyos eseménykezelők (pl. WinForms és WPF esetében), ahol elvárt a `void` visszatérési érték kompatibilitási megfontolásból. Ezeket a metódusokat általában nem szoktuk használni az eseménykezelésen kívül.

8. Delegáltak és események (*Delegates and events*)

Az eseménykezeléshez (pl. `EventArgs.Empty` javasolt használata) és a delegáltakon végezhető műveletekhez kapcsolódó ajánlások.

9. Egyéb (*Miscellaneous*)

Általánosabb, a korábbi kategóriákba nem feltétlenül besorolható ellenőrzések.

Architekturális analízis

A klasszikus statikus analízis *checkerek* mellett a TMS automatizált architekturális analízist is végez a beküldött beadandókra. Ez az elemzés is a forráskód statikus analízisén alapul, és célja a MV, illetve az MVVM architekúrák, továbbá alapvető objektumorientált paradigmák potenciális megsértéseinek detektálása. Az architekturális analízis hibajelzései az **`ARCH`** prefixszel kezdődnek.

A vizsgálat könnyedén elvégezhető saját számítógépen is a beadandó megoldásra, az ellenőrzőt telepíteni konzolos eszközként (*.NET tool*) lehet, az alábbi utasítás kiadásával:

```
dotnet tool install --global ELTE.FI.ArchitecturalAnalyzer
```

(Frissítést keresni és azt telepíteni ugyanezen parancs megismétlésével lehet.)

Az eszköz telepítése után lokálisan is elvégezhető a TMS-ben is futó architekturális analízis:

- MV architektúra ellenőrzése:
  `ArchitecturalAnalyzer analyze /path/to/solution.sln --architecture MV --detailed`
- MVVM architektúra ellenőrzése:
  `ArchitecturalAnalyzer analyze /path/to/solution.sln --architecture MVVM –detailed`

Megjegyzés: az architekturális ellenőrző egy fejlesztés alatt álló, prototípus eszköz. Figyelmeztetései hasznosak lehetnek a helyes beadandó elkészítéséhez, de több *false positive* jelzést is adhat.