Eigen Statische Code Analyse maken

Proof of Concept

[**Wat is de huidige situatie?**](#_47af38u7nj3g) **3**

[**Hoe kan het gerealiseerd worden?**](#_cerkpkz8c3yy) **4**

[De mogelijke oplossing](#_78fb1jhp2rdq) 4

[Uitvoering en documentatie](#_jwyg9xxf30zm) 4

[Input en Output](#_z4j037izevgz) 4

[Code Structuur](#_gphiuhd9nacx) 6

# Wat is de huidige situatie?

Gedurende de tijd van KB-41 Research & Innovation 1 in 2017 is er een onderzoek gedaan naar de mogelijke oplossingen voor de Gamification Project van de Haagse Hogeschool. Een van de problemen dat het project heeft is dat er statische code analyse gebruikt moet worden om er zeker van te zijn dat de studenten hun opdrachten op een goede manier hebben gemaakt. Momenteel kijkt de leraar dit handmatig na en is er de wens dat het geautomatiseerd kan worden door middel van statische code analyse.

Hieruit is een onderzoek gestart met als doel om een toepasselijke oplossing te vinden die binnen de criteria van het project past. Het resultaat was echter dat geen van de mogelijke software toegepast kon worden voor die doeleinden. De gekozen software moest tijdens programma looptijd requirements inladen en een analyse uitvoeren op de opdracht van de student en een diagnose geven. Het inladen van de requirements tijden looptijd bleek onmogelijk en ook het uitvoeren van een analyse had haken en ogen. Het toevoegen van een regel betekende dat er code gemaakt moest worden voor die specifieke requirement. Voor elke requirement per opdracht moest er een analyser geschreven worden, dit is een niet onderhoudbare uitdaging.

Op basis van onderzoeksresultaten zijn we tot de conclusie gekomen dat er geen kant-en-klare oplossingen beschikbaar zijn. Als oplossing moet er een eigen functies gemaakt worden.

# Hoe kan het gerealiseerd worden?

In het vorige hoofdstuk hebben we de huidige situatie beschreven. Hieruit is er als conclusie gekomen dat het maken van eigen functies een van de enigste mogelijke oplossing is. In dit hoofdstuk kijken we hier verder op in en beschrijven we hoe het toegepast kan worden om het gewenste resultaat te behalen.

## De mogelijke oplossing

Het maken van eigen functies om statische code analyse uit te voeren is een uitdaging. Het idee is dat we de requirements kunnen onderverdelen in categorieën die dezelfde soort functie moeten uitvoeren. Zo is het handig dat een categorie voor variables en if statements zijn. Je wilt weten of een variabele bestaat en een waarde heeft en of een if statement gebruikt wordt en de juiste inhoud heeft.

Wanneer deze functies per categorie gemaakt worden kunnen er makkelijker analysers gemaakt worden door simpel functies aan te roepen. Een makkelijk voorbeeld is het zoeken naar een variabele met een specifieke waarde, deze functie hoeft alleen een waarde te weten maar het is ook mogelijk om uit te breiden door een variabele naam.

In eerste instantie moeten deze functies eerst de basis requirements kunnen uitvoeren zoals casting identificeren en met elkaar kunnen samenwerken om complexere requirements op te lossen. Een requirement zoals het vinden of een console log daadwerkelijk met behulp van een lus 5 keer is uitgevoerd kan hiermee geanalyseerd worden. Zo moet er eerst gekeken worden of er een instantie of variatie van een for loop is gemaakt waarna je dan kijkt of er een console log in zit met bepaalde inhoud.

Al deze functies kunnen gebruik maken van de Roslyn API’s die geleverd worden door de Microsoft Analysers NuGet package. Door middel van het aanroepen van CSharpCompilation of CSharpScript kan je een syntax boom maken. Het gebruik van een sytnax boom zal naar verwachting essentieel zijn.

## Uitvoering en documentatie

### Input en Output

De analyser moet de input in het volgende tabel ontvangen.

|  |  |
| --- | --- |
| **Naam Input** | **Type Input** |
| source-code | string |
| requirements | json |

De analyser moet ook output geven in een array formaat.

|  |  |
| --- | --- |
| **Type Output** | **Formaat** |
| String | JSON |

Elke requirement in de output heeft het volgende formaat:

|  |  |  |
| --- | --- | --- |
| **Name** | **Optioneel** | **Inhoud** |
| name | Nee | Requirement naam zoals ontvangen |
| description | Nee | Requirement beschrijving zoals ontvangen |
| status | Nee | Status of requirement is voldaan |
| additional\_information | Ja | Extra informatie zoals lijnnummer etc |

{

"requirement\_one": {

"requirement\_name": "\"Name of Requirement\"",

"requirement\_description": "\"Description of Requirement\"",

"status": "false",

"additional\_information": "\"Failure indicated on ln 12\"",

"": {}

},

"requirement\_two": {

"requirement\_name": "\"Name of Requirement\",",

"requirement\_description": "\"Description of Requirement\",",

"status\"": "true"

}

}

De requirement JSON is nog niet geheel af, zo is er nog gediscussieerd over de toevoeging van een pairing id en een pairing type. Een pairing id en pairing type komen optioneel bij requirements. Sommige requirements bestaan uit meerdere eisen, als voorbeeld kan er in een for loop een variabele gecast worden. We kunnen beiden checken, maar dat betekent niet dat de gecast variable in een for loop zit, het kan eveneens buiten een for loop zitten. Door requirement 1 en 2 te linken met dezelfde pairing id is het mogelijk om ze te linken.

Requirement 1: pairing id 1.0

Requirement 2: pairing id 1.1

Requirement 3: pairing id 1.11

Requirement 3 is gelinked aan 2 en die aan 1. Het idee van een pairing id is dus om een connectie te maken tussen requirements die het nodig hebben. De type link kan gedefinieerd worden met behulp van een pairing type. Een voorbeeld van een pairing type kan zijn “inside”. Als requirement 1 een for loop requirement is en requirement 2 een console write met een pairing type van “inside” dan moet requirement 2 binnen het blok van requirement 1 zitten.

Dit idee is nog niet in zijn geheel uitgewerkt in betrekking met de realiseerbaarheid.

### Code Structuur

De makkelijkste manier om de oplossing te realiseren is om generieke functies te maken en deze aan te roepen per opdracht. Het idee is dat er, afhankelijk van de opdracht, gemiddeld 5 regels aan echte code geschreven moet worden om een analyse uit te voeren. Elke opdracht heeft zijn eigen analyser, deze roept generieke functies aan om het gewenste resultaat te verkrijgen. Als voorbeeld:

*// Opdracht 915*

*public string analyse(var compiledCode, string requirements) {*

*// check loop 5 times*

*var result = GenericLoopAnalyser.getLoopAmount(compiledCode.syntaxTree, requirements, 5);*

*return result;*

*}*

De GenericLoopAnalyser kan het volgende bevatten:

*public string getLoopAmount(var syntaxTree, requirements, amount) {*

*// Gebruik Roslyn API om de for loop te identificeren*

*var forLoopMatches = syntaxTree.getForLoops();*

*// Check hoe vaak die looped mbv Roslyn API*

*var loopResult = forLoopMatches[0].getLoopAmount();*

*// Return resultaat in gespecificeerde requirement output formaat*

*string result = responseService.createRequirementOutput(requirements.name, requirements.description, status, loopResult.additional\_information)*

*return result;*

*}*

De code structuur is natuurlijk heel erg simpel en moet alleen het idee weergeven. Elk categorie heeft zijn eigen Generic Analyser die aangeroepen kan worden door andere analysers of door opdracht analysers. De output van een Generic Analyser kan een resultaat vanuit de responseService zijn of van Roslyn zelf.

Een voorbeeld van het Proof of Concept zit in het mapje “Project/POC - Analyser” van de github repository https://github.com/RichTee/KB-41-StaticCodeAnalyser.

Opdracht Analysers maken gebruik van een interface zodat functies niet veranderen en dus makkelijk aan te roepen zijn door alleen de klasse naam te veranderen tijdens instantiëring. De Generic Analysers moeten geleidelijk gemaakt worden, zo kan er begonnen worden door een functie te maken die bekijkt hoe vaak het loopt of dat een bepaalde variabele gecast wordt binnen een methode. Voor de JSON output formaat kan een NuGet package gebruikt worden zoals Newtonsoft.Json.