Proof of concept: De update automatiseren van Angular versie 16 naar versie 20 in de applicaties van een end-to-end kredietdienstverlener.

Wauters Sander.

Scriptie voorgedragen tot het bekomen van de graad van
Professionele bachelor in de toegepaste informatica

Promotor: Mevr. I. Malfait
Co-promotor: Dhr. P. De Seranno
Academiejaar: 2024–2025
Eerste examenperiode

Departement IT en Digitale Innovatie .

HO
GENT

Woord vooraf

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Samenvatting

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van codefragmenten

1

Inleiding

Softwareframeworks vereenvoudigen het maken van dynamische webapplicaties. Het bedrijf Stater gebruikt voor deze toepassing het Angular framework. Net zoals de meeste software krijgt Angular geregeld updates. Deze updates komen met verschillende voordelen, zoals: nieuwe functionaliteiten, betere performantie, bug-fixes, ... Het toepassen van deze updates is niet altijd even vanzelfsprekend. Nieuwe functionaliteiten dienen soms als vervanging voor oudere. Het gevolg hiervan is dat de broncode moet veranderen van de applicaties die Angular gebruiken.
Stater is een end-to-end dienstverlener voor zowel hypothecaire als consumenten-kredieten. Ze ondersteunen de kredietverstrekker voor de dienstverlening aan consumenten. Stater heeft intern meerdere applicaties die gebruikmaken van het Angular-framework. Specifiek gebruiken deze applicaties Angular versie 16 (v16). Stater zou graag al deze applicaties updaten naar de meest recente versie, Angular versie 20 (v20).

1.1. Probleemstelling

De sprong van 4 versies betekent dat er wellicht veel aanpassingen aan de broncode nodig zijn. Dit probleem vermeerdert zich met de grootte van de broncode en nogmaals met het aantal applicaties dat deze updates nodig heeft. Het manueel uitvoeren van al deze veranderingen neemt veel tijd in beslag. Tenslotte is dit geen eenmalig probleem. Angular krijgt volgens Callaghan (2023) een nieuwe versie om de 6 maanden. Het tijdig uitvoeren van deze updates is in de praktijk niet altijd mogelijk. De ontwikkelaars hebben meer verantwoordelijkheden dan enkel software te onderhouden. Ondertussen worden nieuwe applicaties ontworpen of worden huidige applicaties uitgebreid. Uiteraard kan dit soort onderhoud niet eeuwig uitgesteld worden. Software-updates zijn volgens Vaniea en Rashidi (2016) noodzakelijk om de cyberveiligheid van een applicatie te garanderen. De huidige

versies van Angular zijn momenteel zonder veiligheidsrisico's, maar dit betekent niet dat er geen zijn. Cyberveiligheid is geen opgelost probleem. Het is daarom van belang om alles up-to-date te houden.
Al dit tezamen zorgt ervoor dat de onderhoudskosten snel oplopen. De studie door U. Kaur en Singh (2015) beweert dat het onderhouden van een softwareproject gemiddeld 60% van de totale kostprijs in beslag neemt. Om deze redenen is het vereenvoudigen van het updateproces best interessant. Voor de programmeurs die de updates toepassen, vermindert de werkdruk. En voor het bedrijf betekent dit dat de onderhoudstijd/-kosten voor hun applicaties lager kunnen liggen.

1.2. Onderzoeksvraag

Op basis van de bovenstaande probleemstelling is de volgende onderzoeksvraag geformuleerd: in welke mate kan de automatisering van het updateproces van Angular v16 naar v20, bij meerdere applicaties, de onderhoudstijd voor de ontwikkelaars verlagen?
Om deze onderzoeksvraag te beantwoorden, zijn de volgende deelvragen opgesteld:

- Hoeveel veranderingen moeten uitgevoerd worden om Angular van v16 naar v20 te updaten?
- Welke manieren bestaan er om code automatisch aan te passen zonder ongewenste veranderingen uit te voeren?
- Welke manier om code automatisch aan te passen is het meest geschikt om in deze casus toe te passen?
- Wat zijn statistisch gezien de meest voorkomende problemen bij het updaten van code?

1.3. Onderzoeksdoelstelling

Om de onderzoeksvraag te beantwoorden, wordt als proof of concept een applicatie ontwikkeld die de programmeurs ondersteunt in het updateproces. In de rest van dit onderzoek zal naar deze applicatie verwezen worden als de "updater".
Buiten de functionele requirements van de updater zal dit onderzoek proberen rekening te houden met de ruimere bedrijfscontext. Dit houdt in dat de gekozen implementatie rekening houdt met de huidige doelgroepen en de middelen/noden van het bedrijf.
Concreet betekend dit dat de updater aan de volgende criteria moet voldoen:

- De updater is van de ontwikkelaars voor de ontwikkelaars. De bedoeling is dat de persoon die de update uitvoert de updater kan instellen.

- De updater moet aanpasbaar zijn aan de uit te voeren update. Het moet kunnen gebruikt worden bij de volgende update.
- De updater mag geen nieuwe bugs introduceren. Gegeven dat de configuratie correct is, mag het geen ongewenste fouten maken.
- De updater mag niet gekoppeld zijn aan Angular. Dit zorgt ervoor dat de updater zelf niet geüpdatet moet worden bij een nieuwe Angular-versie.
- De updater stuurt geen informatie door aan derde partijen. Het bedrijf bevindt zich in de financiële sector, waardoor confidentialiteit een prioriteit is.

1.4. Opzet van deze bachelorproef

De rest van deze bachelorproef is opgebouwd als volgt:
In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Hier geven we een omschrijving van wat Angular is en hoe een Angular-project is opgebouwd. Verder overlopen we wat refactoring is en welke manieren reeds bestaan om dit te automatiseren.
In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen. De methodologie begint met het toelichten van de gekozen refactoringtechnieken uit de literatuurstudie. Hierna volgt een korte oplijsting van welke veranderingen concreet uitgevoerd moeten worden om een Angular-applicatie van v16 naar v20 te updaten. Vervolgens wordt als proof of concept de updater uitgewerkt op basis van de gekozen technieken. Tegelijk wordt een gecontroleerde omgeving gemaakt die dient om de effectiviteit van de updater te testen. Tenslotte geven we de resultaten van de updater.
In Hoofdstuk 5, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Alle deelvragen worden beantwoord. Om het onderzoek af te ronden, worden enkele nieuwe inzichten gegeven. Deze kunnen dienen als aanzet tot verder onderzoek.

2

Stand van zaken

In dit hoofdstuk bespreken we de verschillende technologieën die betrekking hebben op dit onderzoek. Deze literatuurstudie start met een omschrijving van het Angular-framework en hoe een Angular-project gestructureerd is. Vervolgens wordt uitleg gegeven over de TypeScript-programmeertaal, specifiek hoe Angular deze gebruikt. Ten slotte volgt een overzicht van verschillende gekende manieren om code automatisch aan te passen.

2.1. Angular

Angular is een user interface (UI) framework ontwikkeld door Google in 2016 (Cincovic e.a., 2019). Het is open-source onder de MIT-licentie en wordt onderhouden door een diverse groep van ontwikkelaars. Angular is de directe opvolger van AngularJS. Hoewel ze dezelfde naam delen, is Angular een volledig nieuw framework met een andere architectuur.

Angular wordt gebruikt voor het ontwikkelen van dynamische single-page webapplicaties. De code van een Angular applicatie volgt een eenduidige structuur. Om deze reden wordt het omschreven als een opinionated framework. Parker (2017) definieert een framework als opinionated als het de ontwikkelaar aanstuurt om op een specifieke manier te werken. Opinionated frameworks houden zich aan strikte conventies die dicteren hoe een project is opgesteld en geschreven. Dit helpt de broncode van een applicatie consistent te houden.

Buiten UI-functies komt Angular met verschillende functionaliteiten die het ontwikkelingsproces ondersteunen (Wilken, 2018). Het komt ingebouwd met een Hyper Text Transfer Protocol (HTTP) client voor een applicatie te verbinden met een backendservice over het internet. Hiernaast heeft Angular een collectie van Command Line Interface (CLI) tools die de ontwikkelaars helpt bij het maken van een applicatie, bijvoorbeeld het genereren van een blanco component met bijhorende

testen in één commando. Tenslotte komt het met functies die toelaten om testen te schrijven voor de UI.
Angular is gebaseerd op TypeScript en gebruikt dit in combinatie met andere technologieën. In een Angular-project zijn de volgende bestanden terug te vinden:

- TypeScript: de TypeScript-programmeertaal wordt gebruikt voor de implementatie van de bedrijfslogica en testen.
- HTML: HTML wordt gebruikt om de structuur van de UI te omschrijven.
- CSS: CSS wordt gebruikt om de visuele representatie van de UI te omschrijven
- JSON: JSON wordt gebruikt voor het configureren van Angular en TypeScript.

In de volgende secties van dit hoofdstuk bespreken we de werking van Angular in meer detail.

2.1.1. TypeScript

Het Angular framework gebruikt en is geschreven in TypeScript. Als gevolg neemt TypeScript het grootste deel van een Angular applicatie in. Zoals omschreven door Bierman e.a. (2014) is TypeScript een programmeertaal ontworpen door Microsoft in 2012. Het is een extensie van JavaScript die een statisch typesysteem toevoegt. Verder heeft het betere ondersteuning voor objectgeoriënteerd programmeren dan JavaScript. TypeScript is een multiparadigma-programmeertaal. Dit betekent dat het verschillende programmeerstijlen ondersteunt, zoals: functioneel, procedureel, objectgeoriënteerd, ...
TypeScript code is niet uitvoerbaar. Om een TypeScript applicatie uit te voeren, moet deze eerst gecompileerd worden. Angular voorziet een compiler TypeScript, met behulp van de TypeScript compiler, naar JavaScript. Volgens Ramos (2024) zijn twee verschillende manieren waarop een Angular applicatie gecompileerd kan worden. Just-in-Time (JIT) en Ahead-of-Time (AOT). JIT compileert de applicatie in de webbrowser tijdens runtime. AOT daarentegen compileert de applicatie op voorhand en stuurt de output naar de webbrowser. Deze studie focust op applicaties die AOT-gecompileerd zijn. Dit is de standaardmanier van werken sinds Angular v9.
Angular gebruikt TypeScript op een objectgeoriënteerde manier. Het is niet strikt objectgeoriënteerd. Angular komt naast klassen met een collectie aan losstaande functies. Een Angular applicatie bestaat voornamelijk uit klassen in combinatie met TypeScript decorators. Martins e.a. (2025) omschrijft decorators als een manier om extra data te koppelen aan een klasse. Decorators kunnen meegegeven worden aan methodes, members, properties of de klassedefinitie. Deze functie is niet uniek aan TypeScript; in Java noemt men dit annotaties en in C# attributes. Angular maakt gebruik van verschillende decorators voor verschillende doeleinden.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-component',
  templateUrl: './my-component.html',
  styleUrl: './my-component.css',
})
export class MyComponent { ... }
```

Codefragment 2.1: Voorbeeld van hoe een Angular-component gedeclareerd wordt in TypeScript.

Welke decorators en hun werking worden in de volgende secties van dit hoofdstuk in meer detail omschreven.

2.1.2. Componenten

Angular applicaties volgen een componentgebaseerde architectuur. Deze architectuur breekt een webpagina op in verschillende bouwblokken, genaamd componenten. Een component kan een klein deel van de UI vormen, zoals een knop of een invoerveld. Of het stelt een groot deel van de UI voor, zoals een navigatiebalk of een volledig formulier. Eén van de belangrijkste eigenschappen van een component is dat deze andere componenten kan bevatten. De compositie van meerdere componenten vormt uiteindelijk de webpagina die de eindgebruiker te zien krijgt. Een component encapsuleert het uiterlijk van een UI-element samen met de achterliggende logica. Voorbeelden van achterliggende logica zijn: valideren van input, data opvragen van een server, ...
In Angular zijn componenten TypeScript-klassen met de `@Component` decorator. De `@Component` decorator verwacht drie parameters. Een “selector” die de naam bepaalt van de component. Een “template” dat de structuur van de UI bepaalt. Dit kan ofwel een verwijzing zijn naar een HTML-bestand of een hardgecodeerde string met HTML-code. Tenslotte is er de “style” parameter die het uiterlijk van de UI bepaalt. Dit kan ofwel een verwijzing zijn naar een CSS-bestand of een hardgecodeerde string met CSS-code. Codefragment 2.1 geeft een voorbeeld van hoe deze syntax eruitziet.

2.1.3. Templates

Angular-componenten maken gebruik van templates om de structuur van de UI te definiëren. Templates zijn syntactisch gelijkaardig aan HTML, maar er zijn enkele verschillen. HTML is de standaardtechnologie die webbrowsers gebruiken om de structuur van een webpagina te interpreteren. Templates breiden de mogelijkheden van HTML uit met nieuwe syntax specifiek aan Angular. Om HTML dynamischer te maken, voorziet het de `@if`, `@else` en `@for` syntax. Deze syntax is

```
<h1>User profile</h1>
@if (isAdmin()) {
  <h2>Admin settings</h2>
  <!-- ... -->
} @else {
  <h2>User settings</h2>
  <ul>
    @for (badge of badges(); track badge.id) {
      <li class="user-badge">{{badge.name}}</li>
    }
  </ul>
}
```

Codefragment 2.2: Voorbeeld van conditionele condities in Angular-templates.

functioneel identiek aan hun TypeScript-equivalent. Codefragment 2.2 geeft een voorbeeld van hoe deze syntax eruitziet. Buiten conditionele syntax is er nieuwe syntax om data uit de component klasse te koppelen aan de template. Dit proces noemt men data binding.

2.1.4. Data binding

Angular gebruikt data binding als manier om data uit te wisselen tussen TypeScript en HTML. Data binding is een manier om een databron met een bestemming te verbinden. In de context van Angular is een TypeScript-klasse de databron en de template de bestemming. Conceptueel zijn er twee soorten van data binding: one-way binding en two-way binding. One-way binding zorgt dat een verandering aan de databron gesynchroniseerd wordt met de bestemming. Veranderingen in de bestemming daarentegen worden niet gereflecteerd in de databron. Two-way binding zorgt dat een verandering aan zowel de databron als de bestemming gereflecteerd wordt in de andere.

Angular heeft vier verschillende soorten syntax voor data binding. De eerste noemt "interpolatie". In de context van templates is interpolatie een manier om dynamische tekst te tonen. Het is one-way binding; de tekst kan via de UI niet aangepast worden. De syntax voor interpolatie is `{{ }}`, bijvoorbeeld:
`<p>Welcome {{ name }}</p>`.

De tweede soort noemt "property binding". Property binding is een manier om dynamisch data aan een HTML-element of component mee te geven. Het is one-way binding; het HTML-element of component kan de data enkel lezen. De syntax voor propery binding is `[ ]`, bijvoorbeeld:
`<button [disabled]="hasErrors()">Save</button>`.

De derde soort noemt "event binding". Hier is de data een referentie naar een func-

```
import { Injectable, Component, inject } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class MyService { ... }

@Component({ ... })
export class MyComponent {
  private service = inject(MyService);
}

```

Codefragment 2.3: Voorbeeld van hoe een service in Angular gedeclareerd en geïnjecteerd wordt.

tie. Deze functie wordt meegegeven aan een HTML-element of component. De bestemming van de functie roept de functie op als een bepaalde conditie bereikt is. Deze functie wordt om deze reden een "event listener" genoemd. Het is one-way binding; het HTML-element of component kan de functie enkel oproepen. De syntax voor event binding is `( )`, bijvoorbeeld:
`<button (click)="saveChanges()">Save</button>`.
De laatste soort is two-way binding. Two-way binding is Angular is hetzelfde als property binding. Met het verschil dat het HTML-element of de component nu de data kan aanpassen. De syntax voor two-way binding is `[( )]`, bijvoorbeeld:
`<input type="text" [(ngModel)]="firstName" />`.

2.1.5. Services

Services zijn herbruikbare stukken code die gedeeld kunnen worden over meerdere componenten. Het zorgt ervoor dat verschillende componenten dezelfde logica delen. Dit maakt een applicatie modulair en zorgt ervoor dat code herbruikbaar blijft.
Services worden door de ontwikkelaars van de applicatie gemaakt. Angular verbindt de services met de componenten door middel van dependency injection. Razina en Janzen (2007) omschrijft dependency injection als een ontwerppatroon om van buitenaf logica aan een object mee te geven. Het zorgt voor een losse koppeling tussen objecten, waardoor de code modulair blijft.
Angular injecteert services in componenten tijdens hun instantiatie. Om Angular dit voor ons te laten doen, moeten alle services de `@Injectable` decorator hebben. Deze service wordt geinjecteert in een klasse via de constructor of de `inject()` functie. Hoewel beide opties mogelijk zijn, wordt de `inject()` functie als de standaardmanier beschouwd sinds Angular v18. Codefragment 2.3 geeft een voorbeeld van hoe dit eruit ziet.

2.1.6. Testen

Testen spelen een belangrijke rol in het ontwikkelingsproces. Testen, valideren en verifiëren dat een applicatie voldoet aan de eisen van de gebruiker (Jamil e.a., 2016). Er zijn verschillende vormen van testen, elk met een ander doel. In deze context zijn unit-testen de voornaamste. Angular komt ingebouwd met functionaliteiten om unit-testen te schrijven voor componenten. Olan (2003) omschrijft unit-testen als een manier om een "unit" in isolatie te testen. In objectgeoriënteerde programmeertalen is een unit vaak een klasse. Unit-testen roepen de methodes van deze klasse automatisch één voor één aan en vergelijken de output. In Angular worden componenten als units beschouwd.
Angular komt met CLI-tools om automatisch unit-testen te genereren. Achterliggend wordt het Jasmine-testing-framework gebruikt. Tijdens het opzetten van een Angular-project wordt Jasmine automatisch geconfigureerd. Het is echter mogelijk om een ander testing-framework te gebruiken. Startende vanaf Angular v20 is het Vitest-testing-framework de standaard. Jasmine is nog steeds volledig ondersteund en is wat gebruikt wordt in dit onderzoek.
Angular voorziet in speciale functies om unit-testen te schrijven voor componenten. Deze functies maken het mogelijk om meer dan alleen TypeScript-klassen te testen. Ze bootsen de werking van de applicatie in een webbrowser na. Het laat toe om templates en de data bindings te testen door interacties met de UI te simuleren.

2.2. Automatisch refactoren

Refactoren, zoals omschreven door A. Kaur en Kaur (2016), is het proces om de broncode van een applicatie te passen zonder de functie te veranderen. Het doel van refactoren is om de interne structuur van een applicatie te verbeteren. Concreet betekent "verbeteren" voor deze casus dat de applicatie up-to-date is en de conventies blijft volgen van de laatste nieuwe Angular-versie.
De studie door Hodovychenko en Kurinko (2025) vergelijkt verschillende gekende manieren om het refactoringproces te automatiseren. In hun studie worden de manieren onderverdeeld in de volgende categorieën:

- Toolgebaseerd: manuele of semi-automatische technieken ingebouwd in Integrated Development Environments (IDE's).
- Algorithmisch: regel-, patroon- of graaf-gebaseerde algoritmen.
- AI-gebaseerd: machine- of deep learning-modellen.

De volgende secties van het onderzoek geven concrete voorbeelden voor elke categorie. Per voorbeeld worden inzichten gegeven in de werking en de voor- en nadelen. Achteraf wordt de beslissing genomen over welke manier het meest geschikt is om toe te passen in deze casus.

2.2.1. Zoek & vervang

Eén van de simpelste vormen van toolgebaseerd refactoren is de zoek- en vervangfunctie. Zoals de naam het zegt, zoekt dit naar instanties van een bepaalde sequentie van karakters in een tekst om deze vervolgens te vervangen.
De simpelste vorm zoekt op basis van een woord of een vaste sequentie van karakters. Meer complexe implementaties maken gebruik van reguliere expressies (regex). Regex, als omschreven door Goyvaerts (2006), is een speciale sequentie van karakters die een zoekpatroon omschrijft. Het laat toe om naar complexe patronen in een tekst te zoeken.
Zoek- en vervangfuncties hebben meerdere voordelen. Het is goed gekend en gedocumenteerd. Deze functies worden in veel applicaties gebruikt, zoals webbrowsers, tekstverwerkers, IDE's, ... De meeste programmeertalen komen met functies om dit te implementeren, zowel tekst- als regex-gebaseerd.
Zoek- en vervangfuncties gebruiken op code is niet zonder nadelen. Omdat het enkel zoekt op tekst, heeft het geen vat op de syntax of de semantiek van de programmeertaal. De syntax omschrijft de structuur en volgorde van verklaringen in de programmeertaal. De semantiek daarentegen omschrijft de betekenis achter deze structuur en verklaringen. Hierdoor is het mogelijk om ongewenste aanpassingen uit te voeren op de broncode. Bijvoorbeeld: stel dat er twee klassen zijn met de naam A en B, en dat beide een methode foo bevatten. Als we alle instanties van de methode foo in klasse A willen veranderen naar bar met behulp van zoek- en vervangfuncties, zullen alle instanties van foo in klasse B ook vervangen worden.
Hoewel het theoretisch mogelijk zou zijn om een reguliere expressie te schrijven die met deze specifieke syntax rekening houdt, zijn hier verschillende praktische problemen mee. Uit de studie door Michael e.a. (2019) blijkt dat regexes moeilijk leesbaar, vindbaar, valideerbaar en documenteerbaar zijn.

2.2.2. Language server

Het Language Server Protocol (LSP) is een open protocol ontwikkeld door Microsoft voor Visual Studio Code. Code editors en IDE's gebruiken LSP's om te communiceren met een language server (Bork & Langer, 2023). Een language server is een programma dat programmeertaalspecifieke functionaliteiten aanbiedt, zoals: automatisch aanvullen van code, code-diagnostiek, code-navigatie, ... Sinds de ontwikkeling van het LSP is het de facto standaard geworden om deze functies te implementeren in code-editors en IDE's.
Voor dit onderzoek zijn de TypeScript en Angular language servers relevant. Volgens de TypeScript Language Server medewerkers (2025) biedt de TypeScript language server functies aan die broncode aanpassen, zoals: verwijderen van ongebruikte variabelen, organiseren van imports, verwijderen van ongebruikte imports, ... De Angular language server focust volgens de Angular Language Server medewerkers (2025) op automatisch aanvullen en diagnostiek voor Angular-specifieke

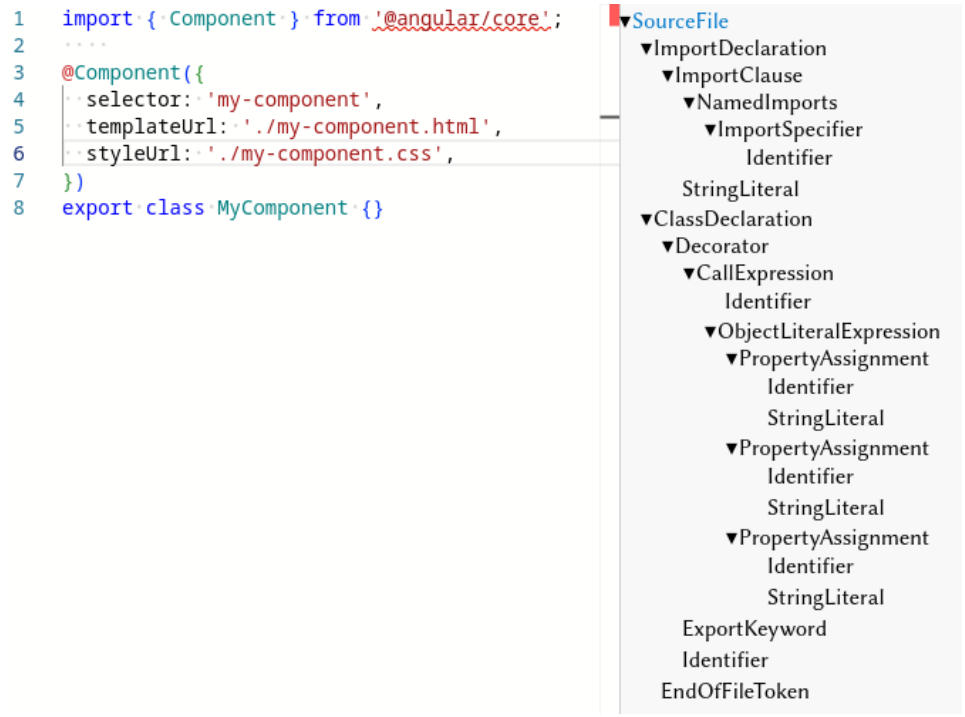
syntax in HTML.
Het voordeel van language servers is dat ze vat hebben op de specifieke syntax van de programmeertaal. In combinatie met een zoek- en vervangfunctie kunnen we gericht code aanpassen. En omdat language servers komen met code-diagnostiek, is het mogelijk om na een verandering direct fouten in de syntax op te sporen.
Language servers zijn echter niet perfect. Ze hebben vat op de syntax, maar niet op de semantiek van de applicatie. Verder zijn we gelimiteerd aan de functies van de language server. Hoewel het mogelijk is om hun functies uit te breiden, is dit in de praktijk niet evident. Het is een zeer niche domein, ondanks dat het de facto standaard is in code-editors en IDE's Een grondige kennis van het LSP is nodig om een language server aan te spreken of uit te breiden.

2.2.3. Compiler

De studie door Wright e.a. (2013) presenteert een tool voor het refactoren van grote C++ broncodes bij Google. Deze tool maakt gebruik van de Clang C++ compiler om broncode om te zetten naar een Abstract Syntax Tree (AST) en zo de code te doorlopen met vat op de syntax. Een AST, als omschreven door Sun e.a. (2023), is een datastructuur die de structuur en syntax van een stuk code weergeeft. Dezelfde bron noemt ATS's een fundamentele eigenschap van code. Om deze reden wordt deze datastructuur vaak gebruikt om codegerelateerde problemen op te lossen. Figuur 2.1 geeft een vereenvoudigde representatie van hoe een AST van een stuk TypeScript code eruit ziet.
Herinner dat Angular gebaseerd is op TypeScript en dat TypeScript gecompileerd moet worden. De TypeScript compiler kan programmatisch aangesproken worden met de TypeScript compiler API. Een studie door Reid e.a. (2023) gebruikt de TypeScript compiler voor het opsporen van foutieve code-elementen, met positieve resultaten.
Net zoals LSP's heeft de compiler vat op de syntax van de programmeertaal. Op het vlak van correctheid is dit één van de meest betrouwbare opties. Verder is de werking van AST's goed gedocumenteerd. Tenslotte bestaan er voor TypeScript verschillende packages die helpen met het opstellen en doorlopen van een AST.
Maar net zoals LSP's heeft de compiler geen vat op de semantiek van de applicatie. Werken met een AST vraagt kennis van boomstructuren en een goede kennis van de nuances van de syntax.

2.2.4. Artificiële intelligentie

Met de recente opkomst in populariteit van artificiële intelligentie (AI) zijn reeds verschillende tools ontwikkeld om deze technologie in te zetten voor het schrijven van code.
Een studie door Polu (2025) gebruikt AI om automatisch code te refactoren om de performantie van een applicatie te verbeteren. Het toont aan dat deze AI-tools vat



Figuur 2.1: Voorbeeld van een vereenvoudigde AST-representatie (rechts) van een stuk TypeScript-code (links).

hebben op zowel de syntax als de semantiek van een applicatie. Uit dezelfde studie blijkt dat het automatisatieproces correct was in 98% van de gevallen.
AI-gebaseerd refactoren is veelbelovend, maar niet perfect. De studie door Hodovychenko en Kurinko (2025) identificeert enkele praktische problemen met deze aanpak. Om dit soort AI-tools te ontwikkelen, is een grote hoeveelheid kwalitatieve data nodig. Het verzamelen en verifiëren van deze data is praktisch niet altijd haalbaar. Tenslotte is er een probleem met transparantie; de interne werking kan niet geverifieerd worden.

2.2.5. Gekende problemen

Een applicatie refactoren is niet zonder risico's. Bij het aanpassen van software bestaat altijd het risico dat er nieuwe bugs ontstaan. Het is niet altijd even evident om dit op te sporen. Indien een applicatie over voldoende testen beschikt, kunnen bugs snel opgespoord worden. Dan nog is het een meerwaarde om deze bugs op voorhand te vermijden.
De studie door Di Penta e.a. (2020) onderzoekt wat de kans is dat een refactoringactie op objectgeoriënteerde applicaties een bug introduceert. Uit deze studie blijkt dat de meest voorkomende acties, zoals een methode of variabele van naam of type veranderen, de meeste kans hebben om nieuwe bugs te introduceren. Verder blijkt dat elke actie een kans heeft om een nieuwe bug te introduceren.

3

Methodologie

In dit hoofdstuk maakt het onderzoek een beslissing over hoe de updater ontwikkeld wordt. Deze beslissing is gebaseerd op de huidige stand van zaken en de noden van het bedrijf, zoals besproken in Hoofdstuk 1.3. We bespreken de verschillende soorten aanpassingen die moeten gebeuren om een Angular applicatie van v16 naar v20 te updaten. Vervolgens bespreken we de proof of concept. Hoe de updater geëvalueerd zal worden aan de hand van een testomgeving. Wat de updater kan en doet. En tenslotte de opstelling van deze testomgeving. In Hoofdstuk 4 wordt de technische uitwerking van de updater in meer detail besproken.

3.1. Plan van aanpak

Op basis van de huidige stand van zaken en de noden van het bedrijf, kiest het onderzoek voor het volgende plan van aanpak. We ontwikkelen een collectie aan helperfuncties om een TypeScript command line applicatie te maken die onderliggend gebruikmaakt van zoek- en vervangfuncties op basis van regex in combinatie met de TypeScript compiler API. Met de hulp van deze functies kan een ontwikkelaar programmatisch een updater maken.
De nodige aanpassingen aan een applicatie zijn afhankelijk van de Angular versie. Deze kunnen complex of simpel zijn. Door de ontwikkelaar programmatisch de updater te laten configureren, is er een hoge flexibiliteit en uitbreidbaarheid van de updater. De helperfuncties zorgen voor een extra abstractie laag, wat toelaat om snel nieuwe updates te automatiseren. Al dit geschreven in TypeScript, de programmeertaal waar de ontwikkelaars bekend mee zijn.
Om dit te implementeren wordt van de TypeScript compiler API gebruikgemaakt. Door te programmeren op de compiler die normaal gebruikt wordt om de applicaties te compileren, krijgen we toegang tot dezelfde error detectie als de compiler. Dit zorgt ervoor dat we nieuwe bugs snel en accuraat kunnen opsporen. Verder is

de interne werking, op basis van een AST, goed gedocumenteerd. Een basiskennis van boomstructuren is wel vereist om hiermee vlot aan de slag te gaan. Tenslotte zijn we niet rechtstreeks gekoppeld aan Angular, enkel aan de programmeertaal.
Om de implementatie voor de ontwikkelaar te vereenvoudigen, wordt de TypeScript compiler API gebruikt in combinatie met zoek- en vervangfuncties op basis van regex.
Angular komt reeds met CLI-tools; door de updater een CLI-applicatie te maken, past het in de huidige workflow. Verder geeft dit de mogelijkheid om alle commando's samen te voegen in één script om de updater op meerdere projecten te laten uitvoeren.

3.2. Angular aanpassingen

Volgens de Angular update handleidling door Google (2025) zijn er in totaal 80 verschillende stappen nodig om een applicatie van v16 naar v20 te updaten. Dit onderzoek verdeelt deze stappen in verschillende "categorieën". Deze onderverdeling geeft een beter overzicht van wat veranderd moet worden aan een Angular v16 applicatie. Tenslotte geeft deze onderverdeling in combinatie met de resultaten van de updater een beter inzicht in waar de updater meer geschikt voor is. De categorieën zijn opgesteld als volgt:

- Veranderingen aan TypeScript. Dit is het grootste deel van alle aanpassingen.
- Veranderingen aan HTML templates. Dit zijn aanpassingen aan Angular specifieke code in HTML templates.
- Veranderingen aan unittesten. Dit zijn aanpassingen aan de unittests die afhankelijk zijn van Angular.
- Veranderingen aan JSON. Dit zijn aanpassingen aan JSON bestanden die de applicatie configureren.
- Uit te voeren commando's. Dit zijn commando's die uitgevoerd moeten worden in de command line. Meestal gaat dit om Angular packages of dependencies te updaten.
- Veranderingen aan syntax. Dit zijn aanpassingen aan syntax die de werking van de applicatie niet aanpassen.
- Veranderingen aan semantiek. Dit zijn aanpassingen in de achterliggende werking van Angular. En/of veranderingen die ervoor zorgen dat de huidige werking van de applicatie moet veranderen.
- Veranderingen die niet van toepassing zijn. Dit zijn aanpassingen aan functies toegevoegd na v16. Het is dus onmogelijk dat de applicaties binnen Stater hiervan gebruikmaken.

Buiten de veranderingen dat niet van toepassing zijn, zijn deze categorieën niet wederzijds exclusief. In één stap kunnen meerdere categorieën van toepassing zijn. Een verandering kan impact hebben op zowel TypeScript als HTML, syntax als semantiek, ...

3.3. Opzet proof of concept

3.3.1. Opzet testomgeving

Om de effectiviteit van de updater te meten, zet dit onderzoek een testomgeving op. De testomgeving is een applicatie gemaakt met Angular v16. Het bestaat uit verschillende klasse en componenten specifiek geschreven met code dat moet veranderen in de update naar v20. De applicatie is enkel syntactisch correct, verder heeft het geen doel. Dit wil zeggen dat het enkel moet kunnen compileren zonder fouten, meer niet. Er is bewust gekozen geen verdere semantiek aan de testomgeving te koppelen, omdat de huidige aanpak er geen rekening mee kan houden.
Dit onderzoek kiest ervoor om de effectiviteit van de updater te testen in een gecontroleerde omgeving om een totaalbeeld te krijgen van alle stappen. In de praktijk is het niet zeker of een Angular applicatie alle code gebruikt dat aangepast wordt in de updates. Het opzetten van een testomgeving geeft de mogelijkheid om een ruimer beeld te schetsen van wat mogelijk is met de updater.
Het volgende proces wordt gehanteerd in het opstellen van de testapplicatie. We doorlopen de Angular update handleidling van Google (2025) en evalueren elke stap of deze in aanmerking komt voor automatisatie. Zoals eerder besproken zijn niet alle stappen aanpassingen aan TypeScript bestanden. Verder is de aard van de verandering belangrijk. De updater kan syntax interpreteren, maar geen semantiek. Tabel 3.1 geeft een overzicht van welke soort veranderingen in de testomgeving opgenomen worden. Voor elke verandering opgenomen in de testomgeving voorzien we een codefragment dat geüpdatet moet worden. Alle codefragmenten zijn semi-realistisch en volgen de Angular best-practices waar mogelijk. Tenslotte geven we per codefragment minstens één stuk code mee dat gelijkaardig is aan de code die veranderd moet worden. Dit dient als controle om na te gaan of de updater specifiek genoeg ingesteld is.

3.3.2. Opzet updater

Dit onderzoek stelt een updater op die de testapplicatie van Angular v16 tot v20 autonoom tracht te updaten waar mogelijk. Buiten de updates uitvoeren zal deze updater bijhouden wat het kan en niet kan. Elke stap in het updateproces wordt manueel gecategoriseerd volgens de categorieën besproken in Hoofdstuk 3.2. De updater probeert voor elk codefragment in de testapplicatie de nodige aanpassingen te detecteren en vervolgens te automatiseren.
Om een beter overzicht te krijgen van de capaciteiten van de updater worden de-

Soort verandering	In testomgeving	Reden
Bestandstype waar de verandering plaatsvindt		
In HTML/JSON	Nee	De updater heeft geen vat op HTML/JSON.
In TypeScript	Ja	De updater is gemaakt om TypeScript syntax te interpreteren. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
In TypeScript & HTML/JSON	Ja	De updater heeft geen vat op HTML/JSON maar wel op TypeScript. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
Aard van de verandering		
Aan semantiek	Nee	De updater heeft geen vat op semantiek. Dit soort aanpassingen zijn afhankelijk van hoe een applicatie Angular gebruikt.
Aan syntax	Ja	De updater is gemaakt om TypeScript syntax te interpreteren. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
Aan syntax & semantiek	Ja	De updater heeft geen vat op semantiek, maar wel op de syntax. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
Randgevalen		
CLI-operaties	Nee	Voor deze stappen is geen nood aan een speciaal stuk code.
Functionaliteiten toegevoegd na v16	Nee	Het is onmogelijk dat deze functionaliteiten gebruikt worden in de applicaties van Stater.

Tabel 3.1: Omschrijft welke stappen uit het updateproces al dan niet opgenomen worden in de testopgeving. Een stap wordt opgenomen in de testomgeving indien deze voldoet aan het bestandstype en de aard van de verandering.

	Toegewezen automatiseerbaarheid		
Conditie	**Volledig**	**Deels**	**Niet**
output = controle	Waar	Fout	Fout
output ≠ controle	Fout	Waar	Waar

Tabel 3.2: Omschrijft wanneer de output van de updater correct of foutief is. De automatiseerbaarheid wordt tijdens het updateproces toegewezen per stap. Een stap is volledig of deels automatiseerbaar als we weten dat er code aangepast is.

tectie en automatisatie elk onderverdeeld in drie categorieën: niet, deels en volledig. Niet detecteer-/automatiseerbaar zijn aanpassingen die niet autonoom uitvoerbaar zijn. Dit kan zijn door de limitaties van de updater of de complexiteit van de aanpassing. Dit onderzoek beschouwt een aanpassing als te complex indien er nood is om meer dan één AST te doorlopen om de aanpassing te detecteren. Per TypeScript bestand behoort één AST. Indien er meer dan één AST doorlopen moet worden, betekent dit dat de aanpassing betrekking heeft op meerdere bestanden binnen de applicatie. Deels detecteer-/automatiseerbaar zijn aanpassingen die enkel deels autonoom uitvoerbaar zijn. Bijvoorbeeld een functie die een nieuwe naam en extra parameters krijgt. De naam kunnen we automatisch veranderen, maar de extra parameters moeten handmatig ingevuld worden, omdat ze afhankelijk zijn van de context. Daarom kunnen uit de automatisatie van deze aanpassingen compiler errors ontstaan. Volledig detecteer-/automatiseerbaar zijn aanpassingen die autonoom uitvoerbaar zijn. Hier kan de aanpassing volledig autonoom gedetecteerd en/of uitgevoerd worden zonder compiler errors te veroorzaken.
Om de updater te maken, voorziet het onderzoek verschillende helperfuncties om de implementatie te vereenvoudigen. Deze helperfuncties maken gebruik van de ts-morph package. ts-morph is een open-source package ontwikkeld door Sherret (2025). Het is een wrapper bovenop de TypeScript Compiler API die het mogelijk maakt een TypeScript project om te zetten naar een AST. Verder biedt het verschillende functies aan om een AST te navigeren en te manipuleren. Onze helperfuncties vormen een extra laag hierbovenop. Het doel hiervan is om een eenvoudige syntax te creëren, specifiek naar de noden van de updater.

3.3.3. Evaluatie updater

Om de correctheid van de updater te testen, stelt dit onderzoek een controleomgeving op. De controleomgeving is opgesteld door de testomgeving handmatig te updaten naar v20. We evalueren de updater door de output te vergelijken met de controleomgeving. Deze vergelijking gebeurt door alle TypeScript bestanden van beide projecten in te lezen en karakter per karakter te vergelijken. Tabel 3.2 geeft een overzicht van hoe de vergelijking in werking treedt.

4

Proof of concept

In dit hoofdstuk geven we een technische omschrijving van hoe de updater werkt. We beginnen met een omschrijving van de helperfuncties. Vervolgens geven we enkele voorbeelden van hoe deze functies samen gebruikt worden om een aanpassing aan broncode te automatiseren.
De eerste functie in codefragment 4.1 helpt om de AST op een uniforme manier te navigeren. Het doorloopt de AST vanaf een gegeven node. Onderliggend gebruikt het een diepte eerst in orde zoek algoritme. De parameters van deze functie specificeren twee callbackfuncties. De eerste is een predicaat dat nagaat of een node voldoet aan een bepaalde omschrijving. Indien dit waar is, wordt de tweede callbackfunctie opgeroepen, die vervolgens een bepaalde operatie uitvoert op deze node. De helperfunctie geeft het aantal nodes terug dat aan het predicaat voldoet. Dit maakt het mogelijk om deze functie als een predicaat mee te geven en zo de functie te nesten.
Codefragment 4.2 omschrijft een ander manier om de AST te navigeren. Hier doorlopen we de AST van een gegeven node terug naar de root van de AST. De functie roept zichzelf recursief op tot de gewenste diepte bereikt is of de huidige node geen ouder bevat.
De rest van de helperfuncties zijn predicaten voor codefragment 4.1. Startend met codefragment 4.3. Deze helperfunctie gaat na of de tekstrepresentatie van een AST-node een gegeven regexpatroon bevat. De tekstrepresentatie van een node is simpelweg hoe een stuk code eruitziet in de broncode. Bijvoorbeeld, deze functie oproepen op de root node van een AST is hetzelfde als zoeken naar een patroon in het eigenlijke bestand. Wordt de functie opgeroepen op een node die een klassedeclaratie voorstelt, is dit hetzelfde als zoeken naar een patroon in deze klasse. Houd er rekening mee dat de tekstrepresentatie van een node ook de tekstrepresentatie bevat van alle kinderen.
Codefragment 4.4 is een extensie op functie 4.3. Hier gaan we na of de gegeven

```
/**
 * Finds the number of nodes that match the predicate in the given AST.
 *
 * @param {Node} root - The root node of the tree.
 * @param {(node: Node) ⇒ boolean | number} predicate - Callback function that evaluates each node.
 * @param {(node: Node) ⇒ void} onMatch - Callback function called if a node matches the predicate.
 */
export function findNodes(
  root: Node,
  predicate: (node: Node) ⇒ boolean | number,
  onMatch: (node: Node) ⇒ void,
): number {
  let matches = 0;
  root.forEachDescendant((node) ⇒ {
    if (predicate(node)) {
      matches += 1;
      onMatch(node);
    }
  });
  return matches;
}
```

Codefragment 4.1: Helperfunctie die de AST vanaf een gegeven node doorloopt. Op basis van de callbackfuncties kunnen gericht aanpassingen uitgevoerd worden op de AST.

```
/**
 * Gets the n'th ancestor of a node.
 *
 * @param {Node} node - The current node.
 * @param {number} count - The distance of the ancestor to the
     current node.
 */
export function getAncestor(node: Node, count: number): Node |
    undefined {
  const parent = node.getParent();
  if (count ⩽ 1 || !parent) return parent;
  return getAncestor(parent, --count);
}
```

Codefragment 4.2: Helperfunctie die de n'de voorouder van een AST node teruggeeft.

```
/**
 * Checks if the text of a node contains a given pattern.
 *
 * @param {Node} node - The node to check.
 * @param {string} pattern - The pattern to check against.
 */
export function containsPattern(node: Node, pattern: string):
    boolean {
  const matches = node.getText().match(pattern);
  return matches ≠= null && matches.length > 0;
}
```

Codefragment 4.3: Helperfunctie die nagaat of een patroon terug te vinden is in de tekstrepresentatie van een AST-node.

```ts
/**
 * Checks if a node contains the last occurance of a given string in it's own subtree.
 *
 * @param {Node} node - The node to check.
 * @param {string} pattern - The pattern to check against.
 */
export function lastInstanceInTree(node: Node, pattern: string): boolean {
  const matchesCurrent = containsPattern(node, pattern);
  const matchesChild = node.forEachChild((child) =>
    containsPattern(child, pattern),
  );
  return matchesCurrent && !matchesChild;
}
```

Codefragment 4.4: Helperfunctie die nagaat of een node de laatste instantie van een patroon bevat in de AST.

node de laatste instantie van een patroon bevat in de AST. Concreet kijkt de functie of de huidige node, en geen enkele van de kinderen, het patroon bevat.
Alle functies tot nu toe waren tamelijk abstract en kunnen voor meerdere doeleinden gebruikt worden. Nu volgen enkele functies die specifiek één doel hebben. De eerste van deze functies in codefragment 4.5 gaat na of een node in een bepaalde scope ligt. We doen dit door recursief het syntaxtype van de ouder te vergelijken tot een match is gevonden of de node geen ouder bevat. ts-morph evalueert het syntaxtype aan de hand van de “SyntaxKind” enumeratie. Voorbeelden van syntaxtype zijn: bestanden, importdeclaraties, klassedeclaraties, expressies, decorators, keywords, ... Als een node de syntaxtype matcht, dan wordt deze node teruggegeven.
Codefragment 4.6 definieert een helperfunctie om het type van een node terug te vinden. Dit doen we door de tekstrepresentatie van het type te vergelijken met een gegeven regex patroon. De tekstrepresentatie van het type is simpelweg hoe het type gebruikt wordt in de broncode.
De laatste helperfunctie in codefragment 4.7 dient om na te gaan of een node toegankelijk is vanaf een bepaald type. Deze functie kijkt of de gegeven node aangesproken wordt vanuit een klasse. Indien dit zo is, vergelijken we het type van de klasse via de functie in codefragment 4.6.
Deze functies maken het mogelijk om een AST te doorlopen in enkele lijnen code en gericht code te detecteren. Wat volgt zijn enkele voorbeelden van hoe deze functies tezamen werken. De aanpassingen die uitgevoerd worden, komen uit de

```ts
/**
 * Checks if a node is in a certain scope.
 * Returns the first node that matches the given scope, otherwise undefined.
 *
 * @param {Node} node - The node to check.
 * @param {SyntaxKind} kind - The kind of the scope.
 */
export function inScopeOf(node: Node, kind: SyntaxKind): Node | undefined {
  const parent = node.getParent();
  if (!parent) return undefined;
  if (parent.getKind() === kind) return parent;
  return inScopeOf(parent, kind);
}
```

Codefragment 4.5: Helperfunctie die nagaat of een AST node in een bepaalde scope zit.

```ts
/**
 * Checks if the type of the node matches the given pattern.
 *
 * @param {Node} node - The node to check.
 * @param {string} type - The pattern of the type.
 */
export function hasType(node: Node, type: string): boolean {
  const matches = node
    .getType()
    .getText(undefined, TypeFormatFlags.InTypeAlias)
    .match(type);
  return matches !== null && matches.length > 0;
}
```

Codefragment 4.6: Helperfunctie die nagaat of een AST node een bepaald type heeft.

```
/**
 * Checks if a node is accessed from a certain type.
 *
 * @param {Node} node - The node to check.
 * @param {string} type - The pattern of the type.
 */
export function accessedFrom(node: Node, type: string): boolean {
  const accessProp =
      node.getParentIfKind(SyntaxKind.PropertyAccessExpression);
  if (!accessProp) return false;
  return hasType(accessProp.getExpression(), type);
}
```

Codefragment 4.7: Helperfunctie die nagaat of een AST node opgeroepen wordt vanuit een bepaald type.

Angular update handleidling door Google (2025). Het eerste voorbeeld in codefragment 4.8 toont aan hoe we de methode van een bepaalde klasse van naam veranderen. Als predicaat zoeken we de naam van de methode op om vervolgens de naam van de klasse te vergelijken. Indien een node aan het predicaat voldoet, vervangt het de tekst met de nieuwe naam van de methode.
De updater kan meer dan methodes van naam veranderen. Ook alleenstaande functies zijn mogelijk. Neem codefragment 4.9 als voorbeeld. Hier veranderen we alle instanties van de async-functie uit Angular met waitForAsync. TypeScript gebruikt async als sleutelwoord; deze instanties mogen niet mee veranderen. Door één lijn code toe te voegen, is het mogelijk om alle instanties van het async-sleutelwoord uit te filteren.
Codefragment 4.10 toont een variatie op het vervangen van een alleenstaande functie.
Het is niet altijd mogelijk om de nodige aanpassingen te automatiseren. Dan nog kan het een meerwaarde zijn om deze op te sporen. Bijvoorbeeld, in één van de stappen in het updateproces moeten Angular componenten met de OnPush-verandering detectiestrategie nagekeken worden hoe ze interageren met templates. We weten op voorhand dat templates niet toegankelijk zijn voor de updater. Maar we kunnen wel verandering detectiestrategieën gaan opsporen. In codefragment 4.11 zoeken we alle componenten op met de OnPush-verandering detectiestrategie. Vervolgens geven we terug waar deze component zich bevindt binnen het project.
Dit waren enkele voorbeelden van hoe een updater met behulp van de helperfuncties ontwikkeld kan worden. De predicaten in deze voorbeelden zijn opzettelijk simpel gehouden. De specificiteit van een predicaat is afhankelijk van de complexiteit van het project en de manier waarop de broncode geschreven is. Verschillende

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      lastInstanceInTree(node, "mutate") &&
      accessedFrom(node, "WritableSignal"),
    (node) ⇒ node.replaceWithText("update"),
  ),
);
await saveProject(project);
```

Codefragment 4.8: Hernoemt alle instanties van de mutatemethode uit de WritableSignal-klasse met update.

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      lastInstanceInTree(node, "async") &&
      node.getKind() ≠= SyntaxKind.AsyncKeyword,
    (node) ⇒ node.replaceWithText("waitForAsync"),
  ),
);
await saveProject(project);
```

Codefragment 4.9: Hernoem alle instanties van de async-functie met waitForAsync.

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      lastInstanceInTree(node, "afterRender") &&
      (!!inScopeOf(node, SyntaxKind.CallExpression) ||
        !!inScopeOf(node, SyntaxKind.ImportDeclaration)),
    (node) ⇒ node.replaceWithText("afterEveryRender"),
  ),
);
await saveProject(project);
```

Codefragment 4.10: Hernoem alle instanties van de afterRender-functie en import met afterEveryRender.

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      lastInstanceInTree(node, "OnPush") &&
      accessedFrom(node, "ChangeDetectionStrategy") &&
      !!inScopeOf(node, SyntaxKind.Decorator),
    () ⇒ console.log(file.getBaseName(), file.getStartLineNumber()),
  ),
);
```

Codefragment 4.11: Zoekt naar elke instantie van de OnPush-methode opgeroepen uit ChangeDetectionStrategy in de scope van een decorator.

bedrijven hanteren intern verschillende manieren om code te schrijven en te structureren. Hoewel deze voorbeelden perfect werken in de testomgeving van dit onderzoek, zullen ze niet bruikbaar zijn voor elk project in elk bedrijf.

5

Conclusie

5.1. Test resultaten

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis

Categorie	Totaal		TypeScript		Unit-testen		Syntax		Semantiek	
#Stappen	80	100.00%	51	100.00%	16	100.00%	24	100.00%	39	100.00%
Automatiseerbaar										
Volledig	22	27.50%	10	19.61%	3	18.75%	9	37.50%	1	2.56%
Deels	9	11.25%	9	17.65%	0	0.00%	5	20.83%	6	15.38%
Niet	49	61.25%	32	62.75%	13	81.25%	10	41.67%	32	82.05%
Detecteerbaar										
Volledig	26	32.50%	26	50.98%	6	37.50%	12	50.00%	15	38.46%
Deels	4	5.00%	4	7.84%	0	0.00%	2	8.33%	3	7.69%
Niet	50	62.50%	21	41.18%	10	62.50%	10	41.67%	21	53.85%
Verandering										
TypeScript	51	63.75%	51	100.00%	16	100.00%	17	70.83%	38	97.44%
template	10	12.50%	3	5.88%	2	12.50%	6	25.00%	4	10.26%
Unit-testen	16	20.00%	16	31.37%	16	100.00%	2	8.33%	14	35.90%
JSON	12	15.00%	0	0.00%	0	0.00%	1	4.17%	0	0.00%
CLI	12	15.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
n.v.t.	9	11.25%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Syntax	24	30.00%	17	33.33%	2	12.50%	24	100.00%	4	10.26%
Semantiek	39	48.75%	38	74.51%	14	87.50%	4	16.67%	39	100.00%

Tabel 5.1: Voorbeeld van een tabel.

Categorie	Totaal		Templates		JSON		CLI	
#Stappen	80	100.00%	10	100.00%	12	100.00%	12	100.00%
Automatiseerbaar								
Volledig	22	27.50%	0	0.00%	11	91.67%	12	100.00%
Deels	9	11.25%	0	0.00%	0	0.00%	0	0.00%
Niet	49	61.25%	10	100.00%	1	8.33%	0	0.00%
Detecteerbaar								
Volledig	26	32.50%	0	0.00%	0	0.00%	0	0.00%
Deels	4	5.00%	1	10.00%	0	0.00%	0	0.00%
Niet	50	62.50%	9	90.00%	12	100.00%	12	100.00%
Verandering								
TypeScript	51	63.75%	3	30.00%	0	0.00%	0	0.00%
Templates	10	12.50%	10	100.00%	0	0.00%	0	0.00%
Unit-testen	16	20.00%	2	20.00%	0	0.00%	0	0.00%
JSON	12	15.00%	0	0.00%	12	100.00%	11	91.67%
CLI	12	15.00%	0	0.00%	11	91.67%	12	100.00%
n.v.t.	9	11.25%	0	0.00%	0	0.00%	0	0.00%
Syntax	24	30.00%	6	60.00%	1	8.33%	0	0.00%
Semantiek	39	48.75%	4	40.00%	0	0.00%	0	0.00%

Tabel 5.2: Voorbeeld van een tabel.

et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit lacus ut lorem. Sed luctus justo sed enim.
Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetuer libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem.
Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

A

Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1. Inleiding

Het bedrijf Stater is een end-to-end dienstverlener voor zowel hypothecaire als consumentenkredieten. Ze ondersteunen de kredietverstrekker voor de dienstverlening aan consumenten. Binnen het bedrijf zijn er verschillende applicaties die gebruikmaken van het Angular webframework. Angular is een open-source front-end framework, ontwikkeling door Google, gebaseerd op de TypeScript programmeertaal voor de ontwikkeling van dynamische webapplicaties (Cincovic e.a., 2019). Momenteel is Angular versie 20 (v20) de meest recente stabiele versie. Binnen Stater maken de applicaties gebruik van Angular versie 16 (v16). Het bedrijf is van plan de applicaties te updaten naar de recentste versie, Angular v20.
De updates niet uitvoeren is geen optie. Volgens de studie door Vaniea en Rashidi (2016) zijn software-updates nodig, omdat het nieuwe functionaliteiten introduceert, de performantie verbetert en verzekert dat de software compatibel blijft met andere nieuwe software. Verder omschrijft deze bron dat het up-to-date houden van software cruciaal is om de cyberveiligheid te garanderen.
Het grote verschil in versies zorgt ervoor dat het updaten van alle applicaties veel tijd in beslag neemt. Dit is geen eenmalig probleem; volgens Callaghan (2023) krijgt Angular een nieuwe versie om de 6 maanden. De studie door U. Kaur en Singh (2015) beweert dat het onderhouden van een softwareproject gemiddeld 60% van de kostprijs in beslag neemt. Een manier om de tijd voor software-onderhoud in te korten is daarom best interessant. Hieruit komt de vraag: in welke mate kan de automatisering van het updateproces van Angular v16 naar v20, bij meerdere

applicaties, de onderhoudstijd voor de ontwikkelaars verlagen? Om deze vraag te beantwoorden zijn de volgende deelvragen geformuleerd:

- Hoeveel veranderingen moeten uitgevoerd worden om Angular van v16 naar v20 te updaten?
- Welke manieren bestaan om code automatisch aan te passen zonder ongewenste veranderingen uit te voeren?
- Welke manier om code automatisch aan te passen is het meest geschikt om in deze casus toe te passen?
- Wat zijn statistisch gezien de meest voorkomende problemen bij het updaten van code?

Gedurende dit onderzoek zal als proof of concept een applicatie ontwikkeld worden die een softwareproject in Angular v16 automatisch updatet naar Angular v20. In de rest van dit document wordt naar deze applicatie verwezen als de “updater”. De updater doorloopt de broncode van een applicatie en maakt een oplijsting van alle nodige aanpassingen en tracht de aanpassing zelf uit te voeren indien mogelijk. Het doelpubliek van de updater zijn de personen die anders deze aanpassingen aan de broncode manueel uitvoeren. De effectiviteit van de updater wordt bepaald aan het aantal gedetecteerde en opgeloste aanpassingen tegenover het totaal van de nodige aanpassingen.
In de volgende sectie wordt een kort overzicht gegeven van de huidige stand van zaken binnen het probleem- en oplossingsdomein. Hierna volgt een beschrijving van de methodologie, waar de werking en evaluatie van de updater in meer detail beschreven is. Tenslotte worden de verwachte resultaten besproken, waarin een inschatting wordt gegeven van de bevindingen van het onderzoek.

A.2. Literatuurstudie

A.2.1. Uit te voeren veranderingen

De Angular update guide door Google (2025) geeft een uitgebreid overzicht van alle aanpassingen die nodig zijn om een Angular-applicatie van v16 naar v20 te updaten. Uit deze bron blijkt dat in totaal 80 verschillende stappen uitgevoerd moeten worden. Deze stappen gaan van het uitvoeren van commando’s tot verschillende aanpassingen aan de code.
Zoals omschreven in de studie door Cincović en Punt (2020), is de code in Angular applicaties onderverdeeld in 3 verschillende soorten bestanden:

- TypeScript-bestanden die de bedrijfslogica bevatten.
- HTML-bestanden die de structuur van de user interface (UI) omschrijven.
- CSS-bestanden die de visuele representatie van de UI omschrijven.

De studie door Di Penta e.a. (2020) onderzoekt welke veranderingen aan code de meeste kans hebben om nieuwe bugs te introduceren. Deze studie maakt het mogelijk om een geïnformeerde inschatting te maken van welke aanpassingen geautomatiseerd kunnen worden zonder ongewenste bijwerkingen te introduceren.

A.2.2. Het automatisatie process

Eén van de simpelste manieren om code in bulk aan te passen is het gebruikmaken van zoek- en vervangfuncties gebaseerd op text of reguliere expressies (Regex). De studie van Michael e.a. (2019) omschrijft Regex als een sequentie van karakters die een patroon in een tekst omschrijft. Uit dezelfde studie blijken enkele problemen bij de implementatie van Regex, namelijk dat het moeilijk leesbaar, vindbaar, valideerbaar en documenteerbaar is. Verder kan Regex geen rekening houden met de semantiek van de programmeertaal, aangezien het enkel op tekst gebaseerd is. Om met de semantiek van de programmeertaal rekening te houden, kan gebruikgemaakt worden van een abstract syntax tree. Zoals omschreven door Sun e.a. (2023), een abstract syntax tree is een datastructuur die de broncode van een applicatie illustreert en rekening houdt met de syntax en semantiek van de programmeertaal. In combinatie met zoek- en vervangfuncties laat dit toe om een stuk code aan te passen, enkel als het in een bepaalde context zit.
Herinner dat Angular gebaseerd is op TypeScript. Een bestaande tool voor TypeScript die gebruikmaakt van een abstract syntax tree is de TypeScript Compiler API. De studie door Reid e.a. (2023) onderzoekt hoe de TypeScript Compiler API gebruikt kan worden voor het corrigeren van foutieve codefragmenten. Deze studie raadt aan om de TypeScript Compiler API te gebruiken voor statische code-analyse vanwege de effectiviteit, accuraatheid en mogelijkheid om foutieve code te detecteren.
Een alternatief voor de TypeScript Compiler API dat ook gebruikmaakt van een abstract syntax tree is het Angular Language Server Protocol (LSP). Het LSP, als omschreven door Bork en Langer (2023), is een open protocol voor gebruik in verschillende code-editors of integrated development environments (IDEs) dat programmeertaalspecifieke functies voorziet zoals: automatisch code aanvullen en code-diagnostiek. Dezelfde bron omschrijft LSP's als het de facto standaardprotocol voor de implementatie van deze functies in IDE's.
Tenslotte is het mogelijk om artificiële intelligentie in te zetten om deze aanpassingen te maken. Met de recente opkomst van AI-tools die specifiek gemaakt zijn voor programmeren, is het mogelijk om deze taak uit te besteden aan AI. Er zijn echter problemen met deze aanpak voor deze casus. Uit de studie door Hodovychenko en Kurinko (2025) blijkt dat AI-gedreven tools een gebrek hebben aan transparantie en risico lopen de semantiek van de programmeertaal in de loop van de tijd fout te interpreteren. Verder maakt deze studie de bewering dat voor het maken van dit soort AI-tools er nood is aan een grote hoeveelheid betrouwbare trainingsdata.

Het bemachtigen van deze data is problematisch, vooral als het gaat om code die gebruikmaakt van de allernieuwste updates.

A.3. Methodologie

A.3.1. Literatuurstudie

Dit onderzoek start met een uitgebreide literatuurstudie naar de verschillen tussen Angular v16 en Angular v20. De nodige veranderingen worden opgelijst en onderverdeeld in verschillende categorieën. Deze oplijsting bepaalt de capaciteit van de updater. Verder geeft dit een beter overzicht van welke aanpassingen al dan niet geschikt zijn voor automatisatie. Tenslotte zal deze oplijsting dienen als maatstaf om de effectiviteit van de updater op te meten.
Vervolgens wordt onderzocht wat de verschillende manieren zijn om automatisch code te updaten. Eén of meerdere manieren worden verkozen om te implementeren op basis van de volgende criteria:

- Complexiteit van de implementatie. Een voorkeur wordt gegeven aan het gebruiken van bestaande tools boven het ontwikkelen van nieuwe algoritmes.
- Betrouwbaarheid van de output. Zolang de input hetzelfde blijft, mag de output niet veranderen.
- Gebruiksvriendelijkheid. Het moet bruikbaar zijn voor de persoon die normaal manueel de applicaties updatet.

A.3.2. Ontwikkeling van de proof of concept

In deze fase van het onderzoek zal de updater ontwikkeld worden op basis van de voorafgaande literatuurstudie.
De updater heeft als minimum de volgende twee functies: het detecteren van code die aangepast moet worden en de aanpassingen uitvoeren indien mogelijk. Het detecteren van de code speelt een dubbele rol. In eerste instantie is het nodig om de aanpassing op de correcte plaats uit te voeren. Indien de aanpassing niet geautomatiseerd kan worden, zorgt het voor een overzicht van waar alle nodige aanpassingen gemaakt moeten worden.
De updater wordt in een gecontroleerde omgeving getest om de stabiliteit te verzekeren. Deze gecontroleerde omgeving bestaat uit een testapplicatie gemaakt in Angular v16. De testapplicatie bevat een codefragment voor elke vooraf geïdentificeerde stap in het updateproces.

A.3.3. Evaluatie

De effectiviteit van de updater wordt bepaald aan het aantal gedetecteerde en opgeloste aanpassingen tegenover het totaal aantal aanpassingen.

Deze meting wordt uitgevoerd op de gecontroleerde omgeving die gemaakt is in de proof of concept. Dit geeft een totaalresultaat voor alle aanpassingen die theoretisch nodig zijn.

A.4. Verwacht resultaat, conclusie

Op basis van de literatuurstudie en de gehanteerde methodologie verwacht dit onderzoek dat minstens 65% van alle nodige aanpassingen automatisch uitgevoerd kan worden.
Het automatisatieproces zal naar verwachting de nodige tijd voor de applicaties te updaten verminderen. De totale hoeveelheid tijd die in werkelijkheid bespaard wordt, is afhankelijk van verschillende factoren: de ervaring van de programmeur, hun kennis van de broncode, de grootte van de applicaties, ... Deze oplossing zal wellicht meer tijd in beslag nemen als enkel één kleine applicatie geüpdatet moet worden.
Dit onderzoek tracht de beste methode te implementeren voor deze casus op basis van gekende literatuur. Echter, kan het interessant zijn om andere manieren te implementeren en te vergelijken. Verder onderzoek kan uitgevoerd worden naar de performantie, accuraatheid en complexiteit van de verschillende implementaties besproken in de literatuurstudie.

Bibliografie

Angular Language Server medewerkers. (2025). *Angular Language Server* [GitHub repository]. Verkregen oktober 31, 2025, van https://github.com/angular/vscode-ng-language-service

Bierman, G., Abadi, M., & Torgersen, M. (2014). Understanding typescript. *European Conference on Object-Oriented Programming*, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Bork, D., & Langer, P. (2023). Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, *18*, 9–1. https://doi.org/10.18417/emisa.18.9

Callaghan, M. D. (2023). Upgrading Angular. In *Angular for Business: Awaken the Advocate Within and Become the Angular Expert at Work* (pp. 95–118). Springer. https://doi.org/10.1007/978-1-4842-9609-7_8

Cincovic, J., Delcev, S., & Draskovic, D. (2019). Architecture of web applications based on Angular Framework: A Case Study. *methodology*, *7*(7), 254–259. https://www.eventiotic.com/eventiotic/files/Papers/URL/df6b5054-816e-4bee-b983-663fb87be2cd.pdf

Cincović, J., & Punt, M. (2020). Comparison: Angular vs. React vs. Vue. Which framework is the best choice? *Zdravković, M., Konjović, Z., Trajanović, M.(Eds.) ICIST 2020 Proceedings*, 250–255. https://www.eventiotic.com/eventiotic/files/Papers/URL/50173409-699e-4b17-8edb-9764ecc53160.pdf

Di Penta, M., Bavota, G., & Zampetti, F. (2020). On the relationship between refactoring actions and bugs: a differentiated replication. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 556–567. https://doi.org/10.1145/3368089.3409695

Google. (2025). *Angular update guide*. Verkregen september 8, 2025, van https://angular.dev/update-guide?v=16.0-20.0

Goyvaerts, J. (2006). Regular Expressions. *Regular Expression*.

Hodovychenko, M. A. H. M. A., & Kurinko, D. D. K. D. D. (2025). Analysis of existing approaches to automated refactoring of object-oriented software systems. *Вісник сучасних інформаційних технологій*, *8*(2), 179–196. https://doi.org/10.15276/hait.08.2025.11

Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2016). Software testing techniques: A literature review. *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, 177–182. https://doi.org/10.1109/ICT4M.2016.045

Kaur, A., & Kaur, M. (2016). Analysis of code refactoring impact on software quality. *MATEC web of conferences*, *57*, 02012. https://doi.org/10.1051/matecconf/20165702012

Kaur, U., & Singh, G. (2015). A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, *118*(1), 6–11. https://doi.org/10.5120/20707-3021

Martins, L., Francisco, P., & Meirelles, P. (2025). An Exploratory Study of Decorators in the TypeScript Programming Language. *Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, 58–68. https://doi.org/10.5753/vem.2025.14563

Michael, L. G., Donohue, J., Davis, J. C., Lee, D., & Servant, F. (2019). Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 415–426. https://doi.org/10.1109/ASE.2019.00047

Olan, M. (2003). Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, *19*(2), 319–328.

Parker, H. (2017). Opinionated analysis development. *PeerJ Preprints*, *5*, e3210v1.

Polu, O. R. (2025). AI-Driven Automatic Code Refactoring for Performance Optimization. https://doi.org/10.21275/SR25011114610

Ramos, A. (2024). Advanced Techniques for Angular Performance Enhancement: Strategies for Optimizing Rendering, Reducing Latency, and Improving User Experience in Modern Web Applications.

Razina, E., & Janzen, D. S. (2007). Effects of dependency injection on maintainability. *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 7. https://digitalcommons.calpoly.edu/csse_fac/34/

Reid, B., Treude, C., & Wagner, M. (2023). Using the TypeScript compiler to fix erroneous Node. js snippets. *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 220–230. https://doi.org/10.1109/SCAM59687.2023.00031

Sherret, D. (2025). *ts-morph* [GitHub repository]. Verkregen november 7, 2025, van https://github.com/dsherret/ts-morph

Sun, W., Fang, C., Miao, Y., You, Y., Yuan, M., Chen, Y., Zhang, Q., Guo, A., Chen, X., Liu, Y., e.a. (2023). Abstract syntax tree for programming language understanding and representation: How far are we? *arXiv preprint arXiv:2312.00413*. https://doi.org/10.48550/arXiv.2312.00413

TypeScript Language Server medewerkers. (2025). *TypeScript Language Server* [GitHub repository]. Verkregen oktober 31, 2025, van https://github.com/typescript-language-server/typescript-language-server

Vaniea, K., & Rashidi, Y. (2016). Tales of software updates: The process of updating software. *Proceedings of the 2016 chi conference on human factors in computing systems*, 3215–3226. https://doi.org/10.1145/2858036.2858303

Wilken, J. (2018). *Angular in action*. Simon; Schuster.

Wright, H. K., Jasper, D., Klimek, M., Carruth, C., & Wan, Z. (2013). Large-scale automated refactoring using ClangMR. *2013 IEEE International Conference on Software Maintenance*, 548–551. https://doi.org/10.1109/ICSM.2013.93