Tampereen yliopisto

Andreas Valjakka

A REENGINEERING FRAMEWORK FOR THE MIGRATION OF A LEGACY FRONT END

Faculty of Information Technology and Communication Sciences
Master's Thesis
December 2019

ABSTRACT

Andreas Valjakka: A Reengineering Framework for the Migration of a Legacy Front End
Master's Thesis
Tampere University
Master's Degree Programme in Software Development
December 2019

JavaScript has evolved into the most popular programming language in the world with an ecosystem of code libraries. One of the first major web development frameworks, AngularJS, is still the most used JavaScript framework for web applications. However, due to its rigidity and the introduction of modern solutions, AngularJS is no longer a viable option for web applications. So, a need arises to call for a software migration methodology for phasing AngularJS out incrementally. This thesis devises and implements a framework for the phase-out process of an application that focuses on migrating its user interfaces from AngularJS to React - a popular modern open-source library.

The migration framework is an adaptation of an existing model. It describes a process where each major activity is viewed as a reengineering process that consists of a reverse engineering, a restructuring and a forward engineering phase. The framework is applied in a case study to analyze and illustrate the detailed steps of migrating a web user interface from AngularJS to React. To minimize the effort, eliminating unnecessary code and salvaging reusable logic is taken into account in different phases of the migration process, and tools are constructed for that purpose. Furthermore, the entire application structure is reverse engineered into a tree structure that assists in recognizing user interface components, their composition and interrelatedness. This tree structure facilitates evaluating the user interface components with which to begin reengineering activities.

This thesis contributes to web development by providing a reengineering framework for migrating client-side solutions to new technology platforms. Additionally, it provides a sequence of reengineering activities to follow. For AngularJS solutions, the activities introduce concepts and guidelines to utilize.

Key words: JavaScript, web development, migration, reengineering.

TABLE OF CONTENTS

1 Introduction

JavaScript, alongside the ecosystem surrounding it, has experienced a significant amount of change since the introduction of the language. As an example, the language has left the confinements of web browser environments during the 2000's and become one of the most widely used programming language in the world. Libraries that once were ubiquitous to the point of being synonymous with the language have been replaced with an abundance of more comprehensive libraries and frameworks for specific purposes.

The purpose of this thesis is to present and discuss the process of transforming a legacy JavaScript solution from year 2014 into one of contemporary time. Software migration consisting of processes of reengineering, forward engineering, restructuring and reverse engineering forms the focus of the discussion. The goal of the research is to devise a systematic approach than can be utilized to modernize client-side technology and to contribute to the practices of web development as well as software migration methodologies as a whole.

Wagner [2014] describes twelve criteria for evaluating legacy systems including the lack of up-to-date documentation and the presence of duplicate code. Following these criteria, the AngularJS project under inquiry only satisfies half of the criteria unambiguously. However, legacy software is not necessarily old. Tools that allow rapid development in combination with frequent personnel change can turn software into legacy rapidly [Demeyer et al. 2003]. Regardless, the starting state of the project will be referred to as legacy for the sake of brevity and ease of communication.

The thesis structure is as follows. The second chapter will delve into the details of the application under study. The motivation for performing the migration is discussed as well as the resulting technology stack. Additionally, remarks on the limitations presented by the application architecture are overviewed.

The third chapter provides an introduction to JavaScript as it is used in web browsers and desktop applications alike. Furthermore, a detailed overview of AngularJS and React is presented. The overview also provides an aid to the migration process by mapping analogous concepts between these libraries.

The fourth chapter introduces the software migration framework. It describes migration as a process of sequential or overlapping reengineering steps. Reengineering, in turn, consists of reverse engineering, restructuring and forward engineering.

The fifth chapter examines how the migration framework can be applied to the knowledge on the differences and similarities of React and AngularJS as well as studying design choices in the legacy application.

The sixth chapter is dedicated to discussing remarks on the results of the migration and reengineering processes, points that can be made from the architecture of the legacy application as well as AngularJS as a tool.

The seventh chapter concludes this thesis.

2 The subject application

The application under study is called Wheel. It is a dashboard application developed using a JavaScript framework called AngularJS [AJS Guide 2018]. Wheel is used to interactively display data that have been funneled into data containers which can hold, for example, a set of numeric values. The data inside these containers can then be visualized by creating a custom layout of differing data modules, ranging from a regular one-dimensional bar chart to a group that can hold several data modules which collectively follow one set of data filtration. Wheel conforms to a component-based architecture which is how React applications – and web applications in general – are typically built.

Wheel is a single-page application (SPA) [MDN 2019]. Single-page applications are websites that utilize JavaScript to load content dynamically. In contrast, regular websites require page loads in order to fetch static HTML documents from the server; single-page applications load all of the markup when a website is first accessed and displays it dynamically on demand when, for instance, user interaction occurs.

2.1 Motivation

According to Stack Overflow trends (Figure 2.1), the ecosystem of JavaScript – with regards to the technologies relevant to this thesis – has experienced major shifts starting from 2014 [SO Trends 2019] which is the year when Wheel was conceived. As the figure shows, the downward trend of AngularJS began in year 2016 with React sustaining a stable growth from around its release in 2013 to today. Alongside the downward trend of AngularJS is also the decrease in popularity of the once popular jQuery [2019] which further illustrates a major shift in the de facto standards of client-side web development.

The negative trend in the growth of AngularJS is partly explained by the introduction of Angular [2019]. Angular is a successor to AngularJS, however, it differs from its predecessor significantly. As a result, they are completely separate entities for all practical purposes. Nevertheless, the decline of AngularJS correlates with a rapid adoption of Angular in Figure 2.1 which seems to suggest that AngularJS developers were shifting focus on the newer framework. Furthermore, while Figure 2.1 seems to suggest that Angular and React have a similar level of interest, usage statistics from BuiltWith.com indicate that Angular is used by 92 000 websites [BuiltWith Angular 2019] while React is used by 958 000 [BuiltWith React 2019]; a difference of approximately 1041 percent. Additionally, AngularJS is used by 3.4 million websites [BuiltWith AngularJS 2019] which further illustrates the status it used to have.

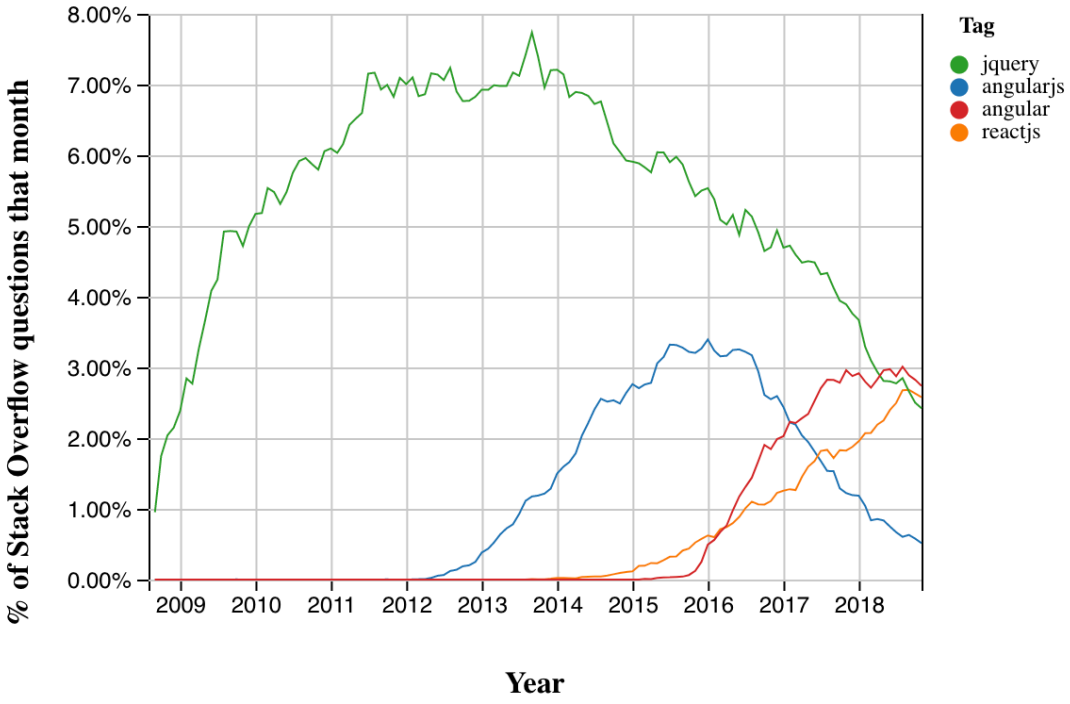

Figure 2.1 Stack Overflow trends of popular JavaScript libraries and frameworks of the last decade [SO Trends 2019].

A survey of AngularJS developers [Ramos et al. 2016] reveals that over 50 percent of the respondents agree on four complex AngularJS concepts: performance degradation, transclusion, differing directive scopes and using the built-in controller, link and compile functions. The study also points out that these are closely related due to being linked to defining custom directives and also shows that the most difficult components to test are complicated directives. Considering that AngularJS is built with a focus on declarative templating aimed to extend the behavior of regular hypertext markup language (HTML) and directives being the way to achieve this [AJS Guide 2018], the study seems to suggest that problems of the framework lie in the core of its design paradigm. Moreover, the survey considered problems related to placing business logic inside HTML templates. It concludes that, while it may result in silent failures and reduced separation of concerns among other things, over a half of the respondents noted that the design of AngularJS itself is the reason for placing a considerable amount of logic inside templates. Alongside the aforementioned difficult AngularJS concepts, the improper use of logic in templates seems to indicate a steep learning curve, a problem occasionally cited by AngularJS developers [AltexSoft 2018; mnemon1ck 2015]. However, developer experience is only anecdotal evidence. As such, it only serves illustrative purposes.

Performance degradation is also cited as a common problem [Ramos et al. 2016; Ramos et al. 2018]. For instance, a Medium article on the topic [mnemon1ck 2015] lists several blog posts and Stack Overflow questions pertaining to this issue. It stems from an internal process of the framework and is discussed in detail in the next chapter. Furthermore, a performance test conducted by Garuda [2017] concludes that React surpasses the performance of AngularJS in test cases related to changing the state of a user interface by switching pages, updating the views of those pages based on asynchronously fetched data and removing elements.

2.2 Technology stack

Modern web application projects typically follow a structure akin to one described by Ahmed [2018]. First, a package manager software is used to download code libraries or frameworks to be used in a project. Second, there are tools, methodologies and libraries utilized to help with cascading stylesheets (CSS) that offer additional syntax, conventions for code structure and common styling options, respectively. Third, build tools can be used aid in common tasks. Build tools include linters which analyze code to detect programming errors or unfavorable patterns. The term originates from a command called lint which is used to detect bugs in programs written in the programming language C [Johnson 1978]. Build tools also include task runners that perform specified tasks such as code compilation and bundlers that turn a modular codebase into a single file for a client's browser to download. Fourth and finally, there are tools used to test the application.

Wheel was scaffolded using Yeoman [YO 2019], a project generator. More specifically, the stack was generated with the AngularJS project generator [YO AJS Generator 2019] that had Grunt [2019] as its build system tool and Bower [2019] as the package manager. In addition to Bower, the project also utilized Node Package Manager [npm 2019] (npm [sic]) for some dependencies in the build system. Node Package Manager is a part of Node.js runtime environment [Node 2019] which is discussed in Chapter 3. Furthermore, the build system included Less [2019] as its CSS preprocessor A CSS preprocessor is a tool which allows the use of custom, often more legible, syntax that can be compiled to regular CSS [MDN 2019]. For testing purposes, Protractor [2019] was used for end-to-end testing. A unit testing framework was not used, although the generated project included Karma [2019] which is a unit testing framework for AngularJS.

The structure of an AngularJS project created with the AngularJS project generator is presented in Figure 2.2. Additionally, technologies which do not directly relate to the

front end include git [2019] which is used for version control and BitBucket [2019] that houses the git repositories. BitBucket also provides a continuous integration and delivery (CI/CD) pipeline.

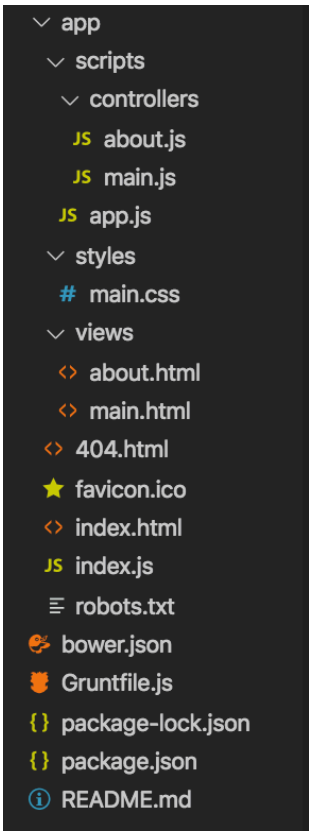

Figure 2.2 A blank AngularJS project generated with the Yeoman AngularJS generator. The dependency and test folders are omitted.

The technology stack for the resulting application was discussed on several occasions and ended up following the aforementioned structure described by Ahmed [2018]. The plan was to use Node Package Manager, keep Less as the CSS preprocessor and switch the build system to use Webpack [2019] as the bundler and npm scripts as task runners. In addition, testing would be made more comprehensive using Cypress [2019] as the tool for end-to-end tests with unit and integrations tests being done using Jest [2019] accompanied with libraries dedicated to testing React components specifically: Enzyme [2019] and React Testing Library [RTL 2019].

Other JavaScript frameworks were proposed as alternatives to replace AngularJS. These were Angular, Aurelia [2019] and Vue [2019]. The decision to use React stemmed from two factors. First, Wheel is a medium-sized application which benefits from flexibility. Second, the avoidance of solutions that make it difficult to effectively utilize labor.

In other words, it was considered important that current and possible upcoming employees are able to begin development work as quickly as possible. For these reasons, Aurelia was not chosen in order to avoid exoticism and Angular was not chosen in order to minimize architectural rigidity. Vue was a formidable option because it is lightweight and designed to be adopted incrementally. However, it was considered too exotic when compared to React since, for example, university courses are offered in React application development whereas Vue, while not obscure, seems to be familiar mainly to experienced developers.

Coincidentally, Facebook provides a piece of software called Create React App (CRA) [CRA] which ships with every tool accepted for the final stack with the exception of SASS being used as the CSS preprocessor instead of Less. Since CRA is an actively maintained open-source project, it grants the advantage of being regularly maintained and updated. As a result, every Less file was converted into SASS and a new scaffolding for the legacy project was instantiated using Create React App. Create React App can be installed using the Node Package Manager. The comparison between the initial and the result technology stack is presented in Table 2.1.

In an effort to streamline building applications, CRA provides default configurations for every build tool which cannot be altered via conventional means. This is one of the drawbacks of using this approach which makes it difficult to, for instance, include tools that are not included in Create React App. As an example, an application written in a language that requires compiling such as Dart [2019] or Elm [2019] will not work since the compiler of CRA is configured to understand only certain syntaxes such as that of TypeScript [2019] – a statically typed alternative that compiles to regular JavaScript.

The default configurations are not an issue with Wheel with the minor exception of a feature that Webpack can support but is not included in Create React App: the importing of HTML template files – used by user interface elements defined in AngularJS – as plain strings. While not required, it has two advantages. First, it allows utilizing the pattern of imports that works with other static assets such as CSS files and images. Second, because the HTML files are converted to strings, they will be directly included in the builds as-is as opposed to static assets which are requested asynchronously from the server which has a negative effect on performance.

Type	In legacy	In result
Version control	Git BitBucket	Git BitBucket
Prerequisites	Node.js & npm Bower Grunt	Node.js & npm
Scaffolding	Yeoman	Create React App
Package manager	Bower	Npm
CSS	Less	SASS
Task runner	Grunt	Npm scripts
Bundler	-	Webpack
Linter	-	ESLint [2019]
Unit & integration tests	Karma (not used)	Jest Enzyme React Testing Library
End-to-end tests	Protractor (initial) Cypress (replacement)	Cypress

Table 2.1 The technology stacks of the legacy and the resulting application platform.

The scenarios above – using an alternative programming language instead of JavaScript and importing HTML files as strings – can be achieved by modifying the Webpack configuration and there are three ways to achieve this when using Create React App. The first is the direct modification of the source code of CRA in the project dependencies folder. This approach can be helpful when one wants to quickly test out a solution but is very impractical due to the requirement of adjusting a dependency outside of the regular workflow. The second approach is to utilize the project ejection script provided by Create React App [CRA 2019]. The script moves every dependency included in CRA alongside their configurations as a dependency of the target project. Ejecting might be desirable in some situations where, for instance, an application has evolved enough to warrant the need for more flexible customizations. However, the consequence of ejecting is that the responsibility for dependency maintenance shifts from the community to the developers

of the target system and it cannot be reverted. If the customizations are as minimal as simply including a feature to allow using HTML imports, the third option, partial customization, is the most optimal. Create React App is a collection of packages, one of which is called react-scripts [react-scripts 2019]. It hosts the configurations whose modification is required for the scenarios above, and a React application can be scaffolded with CRA using a custom version of it [CRA 2019]. For Wheel, custom scripts [html-loader-react-scripts 2019] were made in order to import HTML files used as templates for AngularJS elements, however, the package does not require the presence of AngularJS.

The drawback of each of the three customization approaches is the requirement for additional maintenance on the part of the developers of the target system. As a result, the decision of using customized configurations is a decision that concerns maintainability. The third option, partial customization, requires the least maintenance effort and thus is the most favorable option whereas for the other two approaches it is debatable which one is the least favorable. Summary of the customization options is presented in Table 2.2.

The back-end side of Wheel provides a private application programming interface (API) built using a combination of Clojure and Java. It is implemented using the Representational State Transfer architecture (REST) which provides data in the JavaScript Object Notation (JSON) format.

Customization type	**Description**	**Advantages**	**Drawbacks**
none	Using Create React App as-is without customized configurations.	Needs no additional maintenance.	Having to compromise on parts where an application could have benefitted from customization.
Direct modification	Altering the source code as it is stored in the project dependencies folder.	Quick to use for testing purposes.	Difficult to maintain and to integrate into the regular workflow.
Ejection	Removing Create React App entirely which moves the configurations directly to the target project.	Makes the project fully customizable.	There was never a purpose to using CRA in the first place. Maintenance is delegated from a community to the developer.
Partial customization	Customizing only the part of Create React App that houses configurations.	Makes the project customizable to a limited extent.	Might not be suitable for all situations. Requires additional maintenance.

Table 2.2 The customization approaches for Create React App.

2.3 Limitations

Initially, AngularJS was chosen as the development framework due to its status as the de facto standard and not due to the presence of particularly experienced AngularJS developers. This factor in combination with the aforementioned steep learning curve of AngularJS has resulted in a code base that utilizes only a handful of features that AngularJS provides. For example, transclusion is a powerful feature that allows composing custom user interface elements of another elements that is nonetheless never utilized in Wheel. Instead, some interface element templates in Wheel have logic inside them to determine correct layouts – an antipattern transclusion is used to mitigate. As was discussed earlier, the result of the survey conducted by Ramos et al. [2016] concluded that transclusion is a difficult feature to understand and placing logic inside templates is a prevalent yet maligned pattern. In Wheel, it seems evident that the lack of transclusion usage is related to the prevalence of logic in templates; since using logic in templates has worked in solving a problem, there was never a need to investigate an alternative solution.

Moreover, there are several ways to implement a custom user interface element in AngularJS. Despite this, the code base consists almost entirely of just a single implementation type, the element directive, and the rest are implemented using one other type which is the attribute directive. Directives are discussed in detail in the upcoming chapter.

The legacy application does not utilize a sophisticated state container typical of modern web applications such as Redux [2019] that is included in Create React App. In order to keep the scope of this study more succinct, the implementation of a state container is not discussed.

Wheel uses the version 1.5.8 of AngularJS whereas the latest stable release version is 1.7.8 which is in long term support status with no further development other than critical fixes [Darwin 2018]. Details on the missing features will not be discussed or taken into consideration when discussing the reengineering of the legacy application even in situations where a problem addressed in this thesis has a solution in a later version of the framework. This is due to the judgment that it is not necessary to account for behavior that the application has not exhibited nor will ever exhibit.

Create React App supports the creation of TypeScript projects out of existing JavaScript projects which allows developers to seamlessly opt in to using the language. This is an option that could be pursued simultaneously with the migration project; however, it might introduce additional requirements for the setup of the application. For the sake of

simplicity, the migration project of this thesis will not include the transition into using TypeScript and thus will not discuss any consequences such a choice might entail.

The solutions presented do not take into consideration the browser compatibility such as the version, if any, of Internet Explorer supported. Browser compatibility is deemed as a consideration with ties to business demands and the decision regarding it should arise from business requirements. Suffice it to mention that both React and AngularJS are able to support Internet Explorer versions 9 and upwards [React 2019; AJS Guide 2018].

3 JavaScript

JavaScript is one of the core technologies used in the World Wide Web (WWW) alongside HTML and CSS. It is a programming language that is an implementation of the ECMAScript specification [ECMA 2019]. According to the Octoverse report [GitHub 2019], JavaScript has been the most popular language on the platform from the year 2014 onwards and its superset TypeScript is the third fastest growing language. While JavaScript was originally intended for web development only, it has been implemented into runtime environments that can run desktop and server-side software. One such runtime environment is the aforementioned Node.js, henceforth referred to as Node.

JavaScript is a dynamically and weakly typed programming language that is based on inheritable object prototypes. In order to facilitate interactivity with web pages, browsers have to use an engine to compile and run JavaScript code [Simpson 2014]. One such engine is V8 [2019], an open source JavaScript engine also used by Node. Additionally, browsers need to provide interfaces that provide access to the required artifacts in the browser context such as the window in which the browser is running. For this purpose, browsers provide a Browser Object Model (BOM) that, regardless of the browser, contains a global identifier called window through which the browser viewport can be manipulated [W3S JS 2019]. Furthermore, there is a related concept called the Document Object Model (DOM) which is a tree of objects created by a rendering engine of a browser that represents any individual web page. It also serves as a programming interface to manipulate HTML elements dynamically in JavaScript code [W3S JS 2019].

With Node, the notion of the Browser Object Model is not applicable since it is not a web browser but a runtime environment with its own global scope [Node 2019]. In computing generally, a scope is defined as the region of source code where an identifier – such as a variable – is valid [Backus et al. 1960]. An identifier that resides in the global scope is accessible anywhere in an application. Due to the use of the Webpack compiler that will be part of the resulting technology stack and which runs in Node environment, the distinction between these differing global scopes is relevant especially for developers that are accustomed to taking the browser environment for granted. On one hand, the application will be able to utilize objects in the global context of Node implicitly such as the process object which contains information on the currently running process [Node 2019] while on the other hand, the global context of a browser – the window object – has

to be stated explicitly. How this affects an application using AngularJS specifically is discussed in detail in Chapter 5.

As ECMAScript standard – and JavaScript alongside it – has experienced significant evolution between the inception of AngularJS and today, a distinction has to be made to meaningfully refer to features not available in JavaScript prior to the ECMAScript specifications that led to their development. Henceforth, modern JavaScript will refer to JavaScript that implements the ECMAScript 2015 (ES6) specification or newer while non-modern JavaScript conforms to a specification described by ES5.1.

3.1 AngularJS

AngularJS is a model-view-controller (MVC) framework for JavaScript projects [AJS Guide 2018]. Its initial release was in 2010 and it was still the most popular JavaScript MVC framework in 2018 [Ramos et al. 2018]. The core principle behind AngularJS is in using declarative code to build user interfaces and it can be understood as a tool for extending regular HTML documents, for instance, by defining custom elements. AngularJS applications consist of components which utilize dependency injection (DI) in their construction. Dependency injection is a software design pattern in which dependencies of a software component are passed to it in its constructor or its setter functions [Shore 2006]. AngularJS applications have an injector subsystem which handles dependencies [AJS Guide 2018].

Since AngularJS is a framework, it exhibits features that are typical to frameworks. Frameworks create an abstraction of a domain which determines how the components are interrelated and should be used [Riehle 2000]; the paradigm of AngularJS is to create web applications through extendable HTML which is achieved with custom user interface elements that facilitate the binding between the views and the controllers. The architecture of an application is determined by a framework [Gamma et al. 1995]; in AngularJS, the developers create modules into which constructs like custom user interface elements are registered. Frameworks provide capabilities for reusing code and design in a large scale [Riehle 2000] which leads to inversion of control (IoC). Inversion of control happens when the framework mandates the control flow and imposes rules for naming and function calling conventions [Gamma et al. 1995]. In AngularJS, inversion of control can be seen when, for instance, using internal APIs automatically fire HTML template compilation to update view states or when two-way data binding causes the updates in models to be synchronized with updates in views and vice versa.

The benefit of frameworks is that they offer functionality that is commonly used in their problem domains. For instance, AngularJS is a web application development framework, and web applications often require a means to communicate with an external back end API. For this purpose, AngularJS offers an internal service called $http [AJS API 2018] which, upon use, can also trigger HTML template compilation, further exemplifying the inversion of control AngularJS imposes. Another benefit comes from the fact that the architecture – and, thus, the design patterns as well – are dictated by the framework which means that the structure of an application is easy to understand once the developer is familiar with the framework.

3.1.1 AngularJS constructs

Following is a list of AngularJS concepts that need to be understood: modules, scopes, directives and services.

An AngularJS module is a container into which a developer can register their own directives, services and other AngularJS components [AJS Guide 2018]. One method of accomplishing this is through an interface provided by the internal Module API [AJS API 2018]. It is recommended that an AngularJS application has an application level module as the mounting point and a series of other modules that contain, for example, reusable components [AJS Guide 2018]. However, in Wheel there is only the application level module with every custom component registered into it.

A scope is a special type of an object that serves as a reference to the underlying application model [AJS Guide 2018]. Similar to how the term scope refers to a valid region of source code for referencing an identifier [Backus et al. 1960], an AngularJS scope can be thought of as a region of the DOM that is being, for example, manipulated by a directive. In more practical terms, attributes stored to a scope in a controller are accessible by the view and can be displayed in DOM by using expressions that AngularJS evaluates. In other words, the scope of a view serves as its model, forming one of the constituents of an MVC framework. The synchronicity between models and their respective views is achieved via data bindings [AJS Guide 2018] and a special feature of AngularJS, when compared to React, is the utilization of two-way data bindings. They enable the view and the model to be synchronized without explicit template compilation. While recompilation happens automatically on model mutation, those changes can also be observed programmatically by using separate watcher expressions ($watch). Directives that are used as user interface components typically have their own isolated scopes which

means that a hierarchy of directives creates a hierarchy of scopes. Therefore, scopes have parent and child scopes according to the hierarchy. The root level of every scope is called the root scope ($rootScope). Directives can also opt to use the scope of their parent in case they do not require their own underlying model in which case they lack an isolated scope.

Directives are the elements used to extend regular HTML by decorating regular DOM elements with custom behavior that the AngularJS compiler can process [AJS Guide 2018]. As a general term in computing, a directive refers to a construct utilized to instruct a compiler. An example of a conventional directive introduced in non-modern JavaScript is the strict mode expression (Figure 3.1). When strict mode is applied, the JavaScript engine of a browser interprets it to mean that the code being executed in the scope of the directive has to adhere to certain rules imposed by the strict mode, for example, by necessitating the use of variable declarations [W3S JS 2019]. In a similar manner, AngularJS uses its own HTML compiler which utilizes directives defined by or registered into AngularJS to insert custom behavior to an object in the DOM. This is done by treating DOM elements as declarations that can contain directives that can be matched to the ones registered to the AngularJS module. A typical AngularJS directive is defined as a custom stand-alone HTML element or an attribute for a regular element, but it is also possible to register a directive that is used as a comment or as a class name attribute. Moreover, AngularJS has the concept of interpolation which refers to a special type of directive reserved for data binding between the views and the models [AJS Guide 2018]. An example of each type of AngularJS directive is presented in Figure 3.2.

```
/* example.js
*/

"use strict";

// invalid since the variable isn't declared explicitly
x = "foo";

// valid
var y = "bar";
```

Figure 3.1 An example of strict mode, a conventional JavaScript directive.

```
<!-- example.html -->

<!-- a div element that matches to an attribute directive -->
<div my-attribute-directive></div>

<!-- a custom element that matches to an element directive -->
<my-custom-element></my-custom-element>

<!-- a div element that matches to a class directive -->
<div class="my-class-directive"></div>

<!-- a comment declaration that matches to a comment directive -->
<!-- directive: my-comment-directive -->

<!-- an interpolation, assuming the model has a myText attribute -->
<span>{{myText}}</span>
```

Figure 3.2 Different types of AngularJS directives in an HTML file.

Services are AngularJS constructs that contain business logic or values that can be shared and accessed throughout an application [AJS Guide 2018]. There are four types of services and each of them can be instantiated by using the internal $provide [AJS API 2018] service or the Module API. Each of them follows the singleton design pattern [AJS Guide 2018] in which only a single instance of a service is ever created; every subsequent request for the service obtains a reference to the existing instance [Gamma et al. 1995; Osmani 2017]. In AngularJS this is accomplished by wrapping each registered service inside a provider constructor that contains a getter for the service factory [AJS API 2018] which ultimately creates the service to be handled by the dependency injection subsystem [AJS Guide 2018]. Thus, services can be described as being a part of a provider ecosystem available via use of the internal $provide service [AJS API 2018]. In the ecosystem, the provider is responsible for serving (providing) services to be used throughout the application. Additionally, providers can be used to configure the default behavior of services.

The four types of services are values, constants, factory functions and constructors. The types can further be categorized into two groups according to how their usage can be characterized: values and constants form the value-oriented group since they are both services whose purpose is to serve values – such as strings, numbers or objects – but their usage and functioning within the AngularJS framework differs in two significant ways: values can be modified on per-service basis while constants are immutable which means that they can be used for configurations. In every other respect, they serve an identical purpose. Service factories and constructors form the function-oriented group because they

are intended for more complex shared business logic. The service factory function should not be confused with the service factory utilized in dependency injection. Rather, it is in contrast to service constructors which are instantiated and injected as new objects with the new operator as opposed to service factory functions which are injected as references to the value obtained when the target factory function is run. Unlike with the value-oriented services where the differences may matter in specific situations, service factories and service constructors are identical for all practical purposes except for their declaration. [AJS API 2018]

In addition to the four service types, developers are able to register filters [AJS API 2018]. Filters are value formatters which take an expression as an input [AJS Guide 2018]. As an example, the in-built date filter [AJS API 2018] can be used to format an arbitrary date type or UNIX timestamp input to always match, for instance, a year-month-day form. The notable feature of filters is that they can be used directly in view templates which allows the value to remain in its original form in the model, thus only affecting how the value is presented. For the purposes of this thesis, filters will be referred to as service-like and form a third category in addition to the value and function-oriented ones mentioned before.

For the remainder of the thesis the dependency injection subsystem will not be a major topic. Therefore, whenever factory functions are discussed in the context of AngularJS services, they will always refer to factory-type services. The differences of factory and constructor services will be an important factor in Chapter 5 where refactoring strategies are discussed to facilitate code reuse. A summary of services divided by their categorizations is presented in Table 3.1.

Collective	Subtype	Characterization	Name	Description
Services	Proper services	Function-oriented	Factory	Contains shared functionality defined as a factory function.
			Service	Contains shared functionality defined as an instance.
		Value-oriented	Value	Values that are modifiable on a per-service basis.
			Constant	Immutable values which can be used for configurations.
	Quasi services	Service-like	Filter	Formats values for the view without modifications to underlying model.

Table 3.1 A summary of different types of services in AngularJS.

3.1.2 The digest cycle

As was mentioned in Chapter 2, the poor performance of AngularJS is a common problem developers encounter. A survey of AngularJS developers [Ramos et al. 2018] reveals several causes of this and most of them relate to the digest cycle.

The digest cycle is a process in AngularJS which is initiated on model mutation [AJS Guide 2018] and its purpose is to keep the view state synchronized with its underlying model by continuously observing changes in the variables of the scope [Ramos et al. 2018]. Internally, each scope has an $apply function whose purpose is to evaluate a function parameter it is given and to start the digest cycle [AJS API 2018]. The internal APIs of AngularJS fire it automatically. Since each scope exposes said $apply function, devel-

opers can run it programmatically at any time. However, as the API documentation suggests, the purpose of this function is to evaluate expressions that originate from outside of the framework. Supplying a parameter is optional which means that executing $apply without one is functionally equivalent to simply initiating a digest cycle. Since React is an external library from the point of view of AngularJS, understanding the digest cycle and utilizing $apply will be integral to the migration process in order to avoid scenarios where AngularJS does not seem to behave as it should.

3.2 React

React [2019] is a JavaScript library used to declaratively build user interfaces with a component-based approach. According to download statistics starting from May 1st, 2015 to March 31st, 2019 [Vorbach 2018], React has been the most popular JavaScript package of the ones discussed in this thesis for almost the entire period (Figure 3.3)

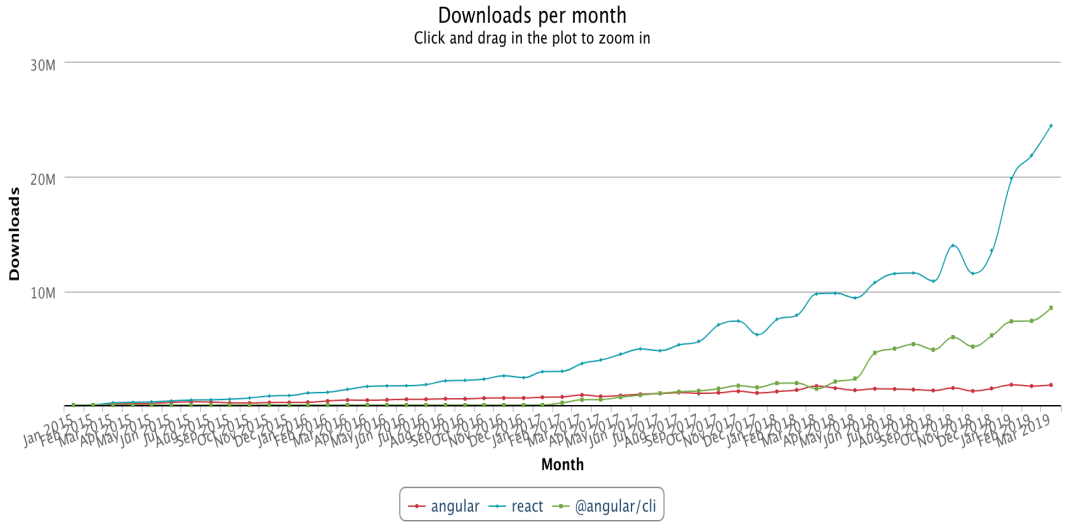

Figure 3.3 The popularity of NPM packages according to downloads for React, AngularJS (angular) and Angular (@angular/cli).

Both AngularJS and React emphasize defining user interfaces declaratively, however, their approaches differ significantly. AngularJS achieves this by compiling a separate HTML template file that is decorated with custom markup while the rendering tool of React provides a declarative API which can be called to get the view state after a manipulation. When the rendering API is called, React initiates a step called reconciliation. During this step, a representation of the DOM – called the virtual DOM (VDOM) – is mapped into the actual DOM in a browser window [React 2019].

A React element is a representation of a singular DOM element and thus a part of the virtual DOM [React 2019]. React elements are constructed as plain immutable JavaScript objects and changes in these representative elements are reflected on the DOM itself in the reconciliation step. Since computing the difference in user interface state is done by using the object representation of the DOM rather than DOM elements themselves, React is more performant than AngularJS – a conclusion supported by the performance study by Garuda [2017] discussed in Chapter 2.

React elements can be created by using JSX, a syntax extension for regular JavaScript. An example of a JSX for a text input inside a form is presented in Figure 3.4. Using it is the recommended practice; another method of creating React elements is to utilize the createElement API provided by React directly [React 2019]. The name JSX is a reference to extensible markup language (XML) since the syntax of the former is influenced by that of the latter (see Figure 3.4). Despite this, JSX does not comply with XML specifications [JSX 2014] nor is it a markup language per se. Rather, it is a template-like syntax that is a substitute for creating React elements as plain objects [React 2019].

```
const inputClasses = "text-input red-text";

const jsxForm = (
  <form>
    <label htmlFor="text">Text</label>
    <input name="text" type="text" className={inputClasses} />
  </form>
);
```

Figure 3.4 A JSX example. Note how using HTML attributes that are reserved key words in JavaScript (*for* and *class*) is circumvented.

A React component is a unit of the user interface that is composed of one or more React elements [React 2019]. They are analogous to element directives in AngularJS which are constructions of regular HTML elements. Data binding between components is achieved via inputs that are given as JSX attributes, similar to how attributes are used in conventional HTML. Unlike in AngularJS where it is possible to create two-way bindings between element directives, data flow in React is always unidirectional [React 2019], from top level components down to their children. Consequently, only one-way data bindings are used in React.

React is a code library which only provides means to build user interfaces. Thus, the migration out of AngularJS needs to also take into consideration, for example, situations where the inversion of control takes place and where a substitute for an in-built service is needed. In the intermediary phases of bottom-up migration approach, developers also need to consider occurrences where a state controlled by AngularJS needs to be displayed using a React component.

3.3 Mapping analogous AngularJS and React concepts

The differences between AngularJS and React are highlighted in Table 3.2. Instead technical details, the table focuses on the concepts in question. The first column contains a concept in AngularJS whose counterpart in React is presented in the second column. Some items in the React column are in parentheses, meaning that the concept is a substitute for the AngularJS counterpart while not being specific to React itself. The last column contains a short description for both the AngularJS and React concepts in order to highlight differences between them.

AngularJS	**React**	**Differences**
Element directive, component	React component	Element directives are a part of and may have decorated HTML templates. React components are constructs of React elements that depict a DOM element.
Module	The root of an application (A plugin)	Module is a container of components registered to it. If it is not the application itself, it can be used as a plugin. A root element of a React application is typically a DOM element in a static HTML document.
Service	(A module with general purpose functionality)	AngularJS services conform to the singleton pattern.
Scope	State Props	Scope is a part of the view-model combination and is utilized via mutation. State and props are immutable properties of a component
Element directive's isolated scope, component bindings	Props	Bind scopes together with possibilities for different types of bindings that can be utilized, for example, via mutation. Props only allow one-way binding and are immutable.
Digest cycle	Reconciliation	Digest cycle iterates through the model hierarchy of the application and compiles the HTML templates. Reconciliation heuristically analyzes a representation of the DOM and updates the actual DOM only where changes occur.
Two-way data binding	(none)	Causes the synchronicity of the model and the view.

Table 3.2 Mapping the analogous concepts of AngularJS and React.

4 Software migration and reengineering methods

Software migration is process of transforming software to a different platform or accommodating new technologies into it. Migrations are done in order to, for instance, improve adaptability to changing requirements. They are a core part of software maintenance which is a constituent of software lifecycle, i.e. the life span of a piece of software, and are performed on a live system. In addition, the system should behave the same despite the changes in underlying technology. [Wagner 2014]

The standard for software maintenance describes migration as a process where a migration plan is made after which users are notified of the process and trained to use it. Furthermore, when the process is complete, the users should be notified, the effects of the migration are assessed, and data is archived. Among other items, the migration plan should contain the definition of migration, how the migration is executed and verified and how the old environment is supported in the future. The intention behind the migration should also be communicated to the affected parties. [ISO/IEC 14764, 2006]

In this thesis, software migration is facilitated by consecutive reengineering activities. Each reengineering activity consists of the processes of reverse engineering, restructuring and forward engineering. However, these processes are interrelated and thus are not necessarily tasks done as separate action points in a roadmap of migration activities. For example, Canfora et al. [2001] suggest integrating reverse engineering to development process in order to create feedback that can aid in forward engineering.

The framework for understanding the interrelatedness of the high-level concepts of the engineering activities is outlaid in Figure 4.1. In the figure, software migration consists of a set of reengineering tasks that occur over time and can overlap. The tasks, in turn, consist of actions on different levels of abstraction. The most refined level is the implementation, i.e. source code, while the most abstract level consists of conceptual ideas that describe the intention behind the software. Forward engineering is depicted as a set of actions that refine abstractions. In practice, the depiction can refer to, for example, turning a business idea into an implementation. Restructuring can occur at varying levels of abstraction. For example, when an implementation is changed to improve the quality of the code, restructuring is done at the implementation level. Reverse engineering is the inverse of forward engineering where refined artifacts are abstracted. Reverse engineering can be performed by, for instance, mapping an implementation to its documented requirement to understand business needs at that time.

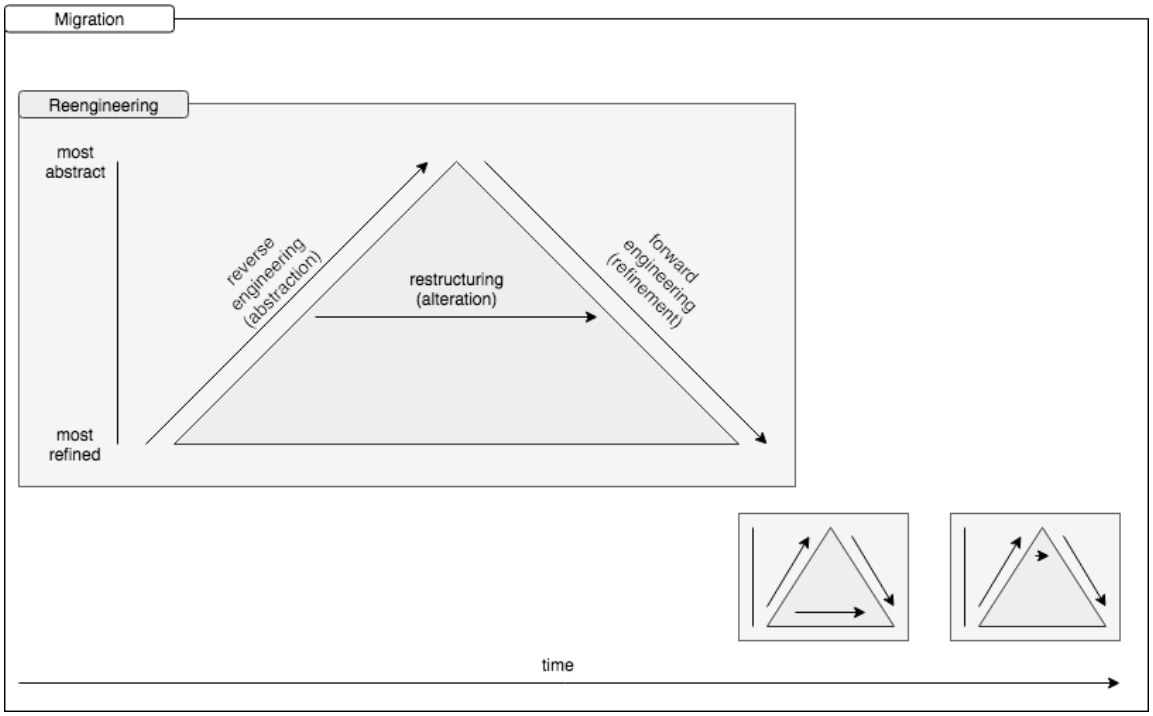

Figure 4.1 How the high-level concepts in this chapter are related. Adapted from a similar construction by Tripathy and Naik [2014].

4.1 Software migration

Wagner [2014] summarizes three methods of software migration. First, migration can be done by developing a new system that, when completed, replaces the old system all at once. This approach has several risk factors; for instance, a system developed in parallel might not account for all the requirements that the existing system satisfies. Second, wrapping can be used to transform components in the legacy system to be compatible with the new environment. While wrapping may seem like an effective strategy since it provides a means to reuse code, the wrapped components need to be migrated further down the line anyway. Furthermore, it might not be an option if the legacy system cannot be disassembled which may be the case for code already wrapped into a framework. Third, which is simply labeled migration, is a method to move the existing system incrementally without affecting its functionality.

The third method is discussed in detail by Demeyer et al. [2003]. They propose a solution in which it is suggested to identify components of the legacy system in order to address the migration task piece by piece. Wagner [2014] describes this as a bottom-up

approach since the source code is the starting point. The migration may also require wrapping or completely replacing legacy components. The advantage of this incremental approach is that a live version of the software product is running the entire time. Incremental changes also make it possible to receive feedback during the process and to retract breaking changes as soon as possible.

In addition, Demeyer et al. [2003] propose several patterns that should be incorporated into the process of migration. For example, prototyping the solution before starting the migration should be considered. During the process itself it is advised to grow the test base incrementally and ensure the system behaves the same after a change in order to have a system that is always running. To determine which components to migrate first, developers should prioritize aspects that are the most valuable to the users since it increases the commitment of the involved parties and has the potential to bear results that are perceived as positive which further validate the necessity of the performed reengineering.

4.2 Reengineering

Software reengineering is a process that is a part of software maintenance and evolution [Wagner 2014]. It aims to understand existing software artifacts and to improve the functionality and quality attributes of a system [Tripathy & Naik 2014; Wagner 2014]. Reengineering can be characterized as a combination of forward engineering, reverse engineering and restructuring [Pérez-Castillo et al. 2011] (see Figure 4.1). According to Chikofsky & Cross [2014], reengineering also seeks to alter the subject system substantially by implementing it in a new form, often with additional functionality, however, Tripathy and Naik [2014] state that, in general, the purpose of reengineering is not to support more functionalities but to enhance existing ones. In addition, reengineering process attempts to transform the system into a form that is further evolvable [Wagner 2014]. Reengineering is a risky process since it subjects the target system to alterations that can cause the system to behave differently or even lower its overall quality. Moreover, the benefits of reengineering might not actualize in a timely fashion [Tripathy & Naik 2014]. In addition, circa 50 percent of reengineering projects fail due to cost overruns or dissatisfying outcomes when using traditional ad hoc solutions where reengineering should be treated as a large change management project [Pérez-Castillo et al. 2011].

The need for reengineering can arise from two types of observations [Demeyer et al. 2003]. For one, there are needs that arise from a desire to better the overall environment

or architecture. The needs include modularizing a monolithic system, improving performance, porting to a new platform and exploiting new technology. For another, there are symptoms that are indicative of a need for reengineering. Usually several symptoms exist at once and they range from missing documentation, tests and original developers to difficulty of maintenance due to inabilities to perform simple changes, a constant need for bug fixes and code smells.

Understanding software development process is a backbone for reengineering. In software development, there are three key principles: the principle of abstraction, the principle of refinement and the principle of alteration [Tripathy & Naik 2014]. Each of the principles maps to an arrow depicted in Figure 4.1. The principles of abstraction and refinement are concerned with the representations that a system can have and are essentially the opposites of each other. A system is at its most refined when in source code form and at its most abstract when described in high-level concepts such as its reasons for existence. Thus, the principle of refinement is realized when producing source code from abstractions such as requirements, a process also known as forward engineering. Tilley [1998] calls this the top-down cognitive model of software understanding. The principle of abstraction is carried out whenever the refined source code is analyzed to, for example, determine original design decisions. As a contrast to the top-down cognitive model, this is the bottom-up model [Tilley 1998]. This process is called reverse engineering. When a system is modified without affecting the abstraction level, the process of restructuring occurs in which the principle of alteration is being followed.

There are six guidelines for a successful reengineering project. First, reengineering should be regarded as a change management project which entails establishing, for example, a concrete budget. Second, the methods should be formalized and use standards. Third, well-known solutions with regards to tools and techniques should be used. Fourth, the project should be in line with the overall strategy of the company. Fifth, decision support mechanisms ought to be used to focus reengineering efforts into the most beneficial activities. Sixth and finally, good software engineering processes, such as documentation, should be followed. [Pérez-Castillo et al. 2011]

Wagner [2014] describes reengineering as a three-step process. In the first step, information is reverse engineered out of the target system. Second, transformations are applied to change the software and finally those transformations are incorporated via forward engineering. These three steps can be summarized as a pattern in the form of extract-

transform-implement. This follows the conceptual model based of the three principles of software engineering described by Tripathy and Naik [2014].

Reengineering has four objectives [Tripathy & Naik 2014]: improving maintainability, improving quality, functional enhancement and migrating to a new technology. A reengineering process can be grounded on all of them or just one. The first objective, improving maintainability, focuses on actively combatting software entropy. Software entropy is the increase in disorder in a software system as it is modified, a concept borrowed from thermodynamics where entropy can be understood as the amount of disorder in a physical system [Jacobson et al. 1992]. In the second objective – improving quality – decreasing quality is understood as the result of change requests that deteriorate the system overall unless reengineering is applied. Functional enhancement is the third objective and it refers to the addition of functionalities that were not required earlier. Finally, the fourth objective is migrating to new technologies in which old platforms are adapted to be compatible with modern technology to reduce maintenance costs as well as costs that occur due to the support being discontinued from older technologies. Migration as an objective in a reengineering process should not be confused with the overarching software migration process which can consist of multiple distinct reengineering processes.

Tripathy and Naik [2014] describe five approaches to reengineering, the choice of which depends on considerations such as the project goals and the risks involved. The first is called the Big Bang approach where an entire system is replaced all at once. Performing a Big Bang might be necessary when, for example, the entire architecture is changed so that every component must be modified to fit the new environment. The second approach is the incremental approach in which the change occurs gradually by producing intermediary operational hybrid systems. The increments are substituted components. The third approach is called the partial approach and it refers to partitioning the system to portions that are to be reengineered which leaves parts of the system untouched on purpose. This might be desirable if the benefits of having a hybrid system outweigh the risks involved in a more complete reengineering. In the fourth approach, the iterative approach, the increments can be as small as a few subroutines, compared to entire components that are replaced in the incremental approach. It entails that the system will have to accommodate four types of components with varying degrees of reengineering performed: completely legacy, hybrid, completely reengineered and brand new. The fifth and final approach is called the evolutionary approach which is similar to the incremental

approach except that functionally similar components are identified and grouped beforehand in order to produce more focused components.

4.3 Forward engineering

Chikofsky and Cross [1990] distinguish forward engineering from reverse engineering and define it as turning high-level abstractions into a software system, following the principle of refinement described by Tripathy and Naik [2014] (see the downward arrow in Figure 4.1). In a more practical manner, Wagner [2014] describes it as a process that turns requirements all the way to source code while Baxter and Mehlich [2000] portray it as a manual construction by a creative agent guided by specifications. It is one constituent of reengineering [Tripathy & Naik 2014].

Including the concept of forward engineering into a migration process is meaningful since it provides a means to clearly communicate phenomena not related to maintaining existing systems. For instance, Favre et al. [2001] discuss how forward engineering has experienced changes via evolution of programming language paradigms. Furthermore, they propose that a component-based architecture is one such forward engineering trend that has had an impact on, for example, reverse engineering technologies. The title of an article by Baxter and Mehlich – Reverse engineering is reverse forward engineering – alludes to this notion [Baxter & Mehlich 2000]. The article further concludes that the knowledge and infrastructure required in forward engineering will be necessary when doing reverse engineering since forward engineering is a process of implicitly applying design choices, such as which structures to generalize and which algorithms to use, that may not be obvious or can be completely hidden. Moreover, the code evolves over time via the inclusion of error coverage and requirement changes which increase the gap between the implementation and its specification, resulting in unreliable specifications and initial documentation.

4.4 Restructuring

Restructuring is a process of transforming representations into others that share the same abstraction level [Pérez-Castillo et al. 2011]. Therefore, it is a realization of the principle of alteration [Tripathy & Naik 2014] (see the horizontal arrow in Figure 3.1).

Depending on the level of abstraction restructuring is concerned with, Tripathy and Naik [2014] categorize the change associated with restructuring into four groups with

decreasing levels of abstraction: rethinking, respecifying, redesigning and recoding. Rethinking occurs at the most abstract level and can lead to fundamental changes as it affects the concepts of the problem domain. For instance, a complete shift to a new problem domain can be the result of rethinking. Respecifying is concerned with the requirements and involves changing the form of existing requirements as well as modifying the project scope by adding, removing and altering requirements. Redesigning affects the design characteristics of overall architecture and individual algorithm choices. Finally, recoding alters the software at its most refined level and is further divided into translation and rephrasing. A translation is a complete change in language such as a compilation to machine code while rephrasing is a process of alteration where the language is not changed. Code optimization and refactoring are examples of rephrasing. The modifications in the migration process mainly occur at the implementation (most refined) level without changing the programming language and are therefore processes of refactoring.

4.4.1 Refactoring

Refactoring is a process of applying a series of small changes to source code to improve its overall quality [Fowler 2019]. Refactoring is performed like forward engineering with the difference that it has to follow a modified software process life cycle where the initial analyses are replaced by a discovery process where the developers learn about the requirements relevant to the targeted code in conjunction with the overall structure. Furthermore, a design phase is replaced with a redesign step to implement the refactored code sections, for example, by following a software design pattern. Moreover, the bulk of the actual coding can happen via copying or transforming the existing code while tests exist to validate the success of the refactoring done. [Mancl 2001]

If a software system is in use, it needs to adapt to changing requirements which increases its inner complexity, resulting in the deterioration of its structure [Wagner 2014]. As a consequence, it will experience an increase in software entropy and refactoring can be understood as a measure against it.

Since refactoring should not alter the behavior of a software system, automated tests are crucial for any refactoring process. Tests are important for catching faults introduced in the refactoring process due to human error which is why it should be done in small steps while the tests validate the source code continuously [Fowler 2019; Mancl 2001]. Three types of tests are taken into consideration. First, unit tests that are designed to test

the smallest individual parts – i.e. units – of a software system in isolation. Second, integration tests which verify the interaction between software components. In other words, integration testing aims to test the interoperability of units [Butterfield et al. 2016]. Third, end-to-end tests that test an application – or some specific feature of it – in its entirety by simulating user behavior. End-to-end testing are a type of system testing that ensures that the components of an application work in unison when a certain task is being undertaken. [Butterfield et al. 2016]

Tools that automate refactoring tasks are commonly built into integrated development environments (IDEs) such as Webstorm [2019]. According to Murphy-Hill and Black [2008], such tools are fast and mitigate the risks associated with human error but are often underutilized due to misunderstanding the ways programmer do refactoring. For example, Fowler [2019] suggests that refactoring is not a focused action but rather a practice to be incorporated into regular programming. Similarly, using dental floss is an activity that should be performed on a regular basis which is why Murphy-Hill and Black [2008] define floss refactoring as being refactoring done in small increments to maintain a healthy piece of software. Root-canal refactoring, in contrast, is a larger activity aimed at correcting unhealthy software at a stage where symptoms of poor maintenance begin showing up. They conclude that refactoring tools should focus on making floss refactoring align themselves with the workflow of a programmer and propose five principles for an efficient floss refactoring tool: it should be quick to use, provide an easy way to switch from the tool to the editor, make browsing the code effortless, avoid configurations and keep other tools accessible. Automated refactoring tools known to be reliable reduce the need to rely on unit tests while refactoring [Fowler 2019].

Code smells indicate a point in source code that may benefit from refactoring [Fowler 2019]. Demeyer et al. [2003] also point out that they indicate a need for reengineering since they signal that a software system has been expanded or adapted without considerations for reengineering. An example of a code smell is a long list of parameters which are hard to understand and tend to be inconsistent [Fowler 2019]. Saboury et al. [2017] studied code smells that occur in JavaScript projects and report up to 65 percent lower hazard rates for projects without code smells.

Conversely, there are cases where refactoring is not necessary. For example, on some occasions it may be easier to rewrite code entirely. Other times, code is readily available and functional despite needing refactoring; if the called code itself does not need to be modified, there is no need to refactor it [Fowler 2019].

4.5 Reverse engineering

Reverse engineering refers to a process of examining an existing software system to extract alternative representations of it [Canfora et al. 2011] and it aims at understanding systems with inadequate documentation [Baxter & Mehlich 2000]. In Figure 4.1, reverse engineering is depicted as an upward arrow, or a process of abstracting artifacts at a more refined level. The origins of reverse engineering are in software maintenance [Canfora et al. 2011; Wagner 2014]. It is also described as a computer-aided support mechanism that aims to better program understanding alongside other mechanisms such as knowledge leveraging and code browsing which are human-oriented [Tilley 1998].

The need for reverse engineering arises from several factors, one of which is the gap between the specifications and the actions of creative agents: the software developers [Baxter & Mehlich 2000]. Furthermore, Tripathy and Naik [2014] list factors that signal the necessity of reverse engineering such as the absence of the original developers and insufficient documentation. These factors have overlap with the symptoms that Demeyer et al. [2003] describe as indicative of a need for reengineering which is to be expected since reverse engineering is a subprocess in reengineering (see Figure 4.1).

According to Wagner [2014], reverse engineering is the foundation of a software reengineering process since producing information is necessary for understanding a system. In their classification and the classification of Beck et al. [2011], reverse engineering is a part of reengineering. The goal of reverse engineering is to perform examination without applying changes and its purpose is the increase in ability to comprehend the system [Chikofsky & Cross 1990]. Tilley [1998] specifies that the increased comprehension arises from artifact identification, relationship discovery and abstraction generation. Recovering these artifacts assists in reducing indirect cost of developers having to understand a system before being able to extend it [Chikofsky & Cross 1990]. Mancl [2001] describes this process of gathering understanding as the discovery process that is a necessary step before a developer can perform refactoring, thus creating a link to reverse engineering. Information produced when reverse engineering can have varying balances of abstractions and details depending on which aspect a representation is focused on which leaves room for the developer to interpret the information obtained in a most suitable way [Wagner 2014]. While the aim of reverse engineering is to understand the target system, Baxter and Mehlich [2000] point out that understanding is only a prerequisite; not a means unto itself. The actual target of reverse engineering is the ability to modify the system.

4.5.1 Reverse engineering in practice

Chikofsky and Cross [1990] identify six main objectives for reverse engineering. The first, coping with complexity, refers to being able to extract information out of a system despite its complexity. Second, alternative views are visual representations of some aspect of the system, such as graphical depictions of control flow. The third objective is recovering lost information. It is a process where programmatic reasoning about the system is applied in order to address the lack of documentation. Fourth, detecting side effects is a step to alleviate the effects of forward engineering that has been performed to apply changes without consideration for subtle unintended consequences. Fifth, synthesizing abstractions refers to a process where the level of abstraction of generated alternative views can be controlled. Finally, the goal of the sixth objective, reuse facilitation, is to detect reusable software assets.

Tilley [1998] describes five tasks that reverse engineering accomplishes in the order of increasing abstraction: program analysis, plan recognition, concept assignment, redocumentation and architecture recovery. In program analysis, the source code itself is studied in various levels of abstraction to extract different types of representations for differing purposes. Plan recognition focuses on the semantic aspect in contrast to the syntactical approach of program analysis. Semantic aspects include identifying patterns of, for instance, repetition and algorithm use the source code employs. Concept assignment further proceeds to understand software by discovering where artifacts visible to the end user are implemented in the source code. In other words, it is a mapping between an implementation and the concepts humans use to communicate about it. Redocumentation is a weak type of refactoring where retroactive documentation is supplied such as comments added to the source code. Finally, architecture recovery is concerned with the overall structure of a software system rather than any individual component. Isolated parts that are well documented might exist without offering valuable insight into the system overall. Architecture can also be viewed from a multitude of different angles which makes it impossible to have an exhaustive documentation. [Tilley 1998]

Tilley [1998] describes three reverse engineering activities, the first of which is called data gathering. Data gathering is usually the first phase of reverse engineering where raw data is collected to identify interrelated parts of a system. As such, data gathering serves as the foundational activity and maps directly to program analysis – the first of the reverse engineering tasks described earlier. Data can be gathered from examining systems by

performing analyses. Canfora et al. [2011] identify four types of software analysis: static, dynamic and hybrid analysis as well as historical analysis. First, static analysis is performed on a snapshot of a system without the need for execution. An example of a static analysis is a program that can create a tree representation out of the component structure found in source code. Second, dynamic analyzers utilize the execution environment of a piece of software to extract information out of program execution traces. A code coverage tool of a test runner is an example of a dynamic analyzer. Third, hybrid analyzers are a mixture of static and dynamic analysis tools. Finally, historical analyzers use information from systems that store traces of the evolution of a program such as a version control system. In addition to system examination, data can be gathered from documents – including comments in source code – and from knowledgeable people [Tilley 1998]. Biggerstaff [1989] also mentions descriptive variable names and characterizes each of these as informal information.

Knowledge management is the second reverse engineering activity. The goal of it is to achieve an active repository of shared knowledge. It relates closely to the third data gathering activity where information is drawn from experienced, knowledgeable people. Knowledge management follows as an extension and is a process that requires organizing knowledge in order to store and retrieve meaningfully structured data that is classified, aggregated and generalized. Furthermore, it requires knowledge discovery to facilitate understanding, for example, via multiple perspectives on data artifacts. One last final requirement for knowledge management is evolution in order to keep the knowledge base updated during the addition and modification of information.

The last reverse engineering activity is called information exploration and is deemed to be the most important one of all the activities since it is the phase during which most of the understanding of a software system happens [Tilley 1998]. It comprises three parts, first of which is information navigation which further comprises of selection of data artifacts that match their counterparts in reverse-engineered abstractions, editing these artifacts and traversal between them. The second constituent of information exploration is analysis, the levels of abstraction an analysis is concerned with, whether these analyses are automated and what type of an analysis they are. The analysis types here are the same ones that were discussed as a part of the data gathering activity: static, dynamic, hybrid and historical analysis. Finally, the third part of the information exploration activity is the

data presentation mechanism in which visual metaphors are utilized in assisting understanding. Data can be presented on a user interface from multiple angles utilizing many visualization mechanisms.

Analysis techniques are required to realize data gathering. Tripathy and Naik [2014] describe seven techniques: lexical analysis, syntactic analysis, control flow analysis, data flow analysis, program slicing, visualization and metrics. First, lexical analysis uses pattern matching and regular expressions to extract valid source code tokens, such as variable type definitions, to be processed in the syntax analysis phase. In the second technique, syntactic analysis, the compiler parses the tokens produced in the lexical analysis phase according to a context-free grammar that is able to detect more sophisticated structures such as the use of parenthesis in mathematical expressions. The third technique, control flow analysis, is divided into intra- and interprocedural analyses where the former is a depiction of the ways in which a subprogram can be executed – i.e. the order of the execution of statements within a procedure – while the latter describes how different procedures call each other which is depicted as a directed call graph. Control flow analyses be used to, for instance, detect loops. Data flow analysis is the fourth technique and it aims to examine the use of variables as they are used throughout the application in order to detect cases where, for example, a variable may be undefined or defined but not referenced. The fifth technique, program slicing, is used to split a program at any given point to obtain a portion of the program to, for instance, determine the values of a variable at that moment. A backward slice can be used to reason about the conditions that have led to a variable value at the slice point while the purpose of a forward slice is to detect which upcoming computations are dependent on the conditions present at the slice point. The sixth analysis technique is visualization where the software system is presented as depictions of singular components (representations) and the conglomerate of these depictions (visualizations). One of the most important factors in representations is their effectiveness when used in communication which can be influenced by, among other things, reducing visual complexity, increasing the information content and providing a user interface. The seventh and final technique is called program metrics which refers to the practice of extracting hard data out of the source code. Counting the lines of code (LOC) in a software is one of the most easily obtainable program metrics.

4.5.2 Design recovery

Design recovery is a subarea of reverse engineering where a combination of external knowledge and imperfect information is matched with data from examination of the system itself. Furthermore, it extends the observations made of a system by incorporating differing sources of information extensively such as domain knowledge. Therefore, it cannot be described as a task per se [Chikofsky & Cross 1990]. This is a more holistic view of a system whereas individual data gathering activities such as plan recognition [Tilley 1998] focus on generating abstractions from realizations of plans and design choices.

Baxter and Mehlich [2000] describe the problems with this approach which begin with the fact that some knowledge of the most important plan realizations must be present from the beginning. Furthermore, designs and plans are realized in multiple ways in conjunction with other design choices. This leads to the problem of needing to detect several plan realizations whilst utilizing multiple patterns in which the realizations have been made. Moreover, a singular implementation is not necessarily a realization of only one plan. Design recovery requires deeper understanding of the intention behind the structure which in turn relates to the choices made by the designer who originally implemented a structure. They conclude that the focus of reverse engineering needs to be in design recovery essentially as a process of applying forward engineering in reverse.

Design recovery models are defined by two properties [Biggerstaff 1989]: first, their use of informal human-oriented information and second, a domain model that serves as a basis for interpreting structures and conventions present in code. The use of informal information helps in defining systems in terms that are directly accessible to human beings. This can be achieved by identifying conceptual abstractions and their instances in the domain model. A conceptual abstraction is analogous to a class in an object-oriented system and comprises of a structural pattern that describes an entity as a set of details (i.e. attributes or fields of a class construct) as well as a web of associations to related concepts in the real world. The model by Tilley [1998] describes the data gathering activity as comprising of document and knowledge exploration which means that the model fulfills the described properties. However, Biggerstaff [1989] goes further and emphasizes the importance of informal information by using an example of a deliberately obfuscated code and by pointing out that the computational intent behind source code might not be

unambiguous if the programming style has caused a disconnection between the problem domain semantics and the code.

4.6 Summary

As a summary, software migration is a process that consists of sequential or overlapping reengineering steps. Reengineering is a process where activities of forward engineering, reverse engineering and restructuring occur at varying levels of abstraction. Figure 3.1 is a compilation of these concepts that describes a framework for understanding how these processes relate to each other.

Migration is a process of software transformation that accommodates a running system to using new technologies and platforms [Wagner 2014]. A migration project should have a plan and methods of communicating its state before, during and after the project is completed as well as the reason for the migration [ISO/IEC 14764, 2006]. The three methods of software migration are the development of a new system, the wrapping approach and the incremental migration that does not affect the functionality of the system [Wagner 2014]. Incremental migration should be addressed piece by piece [Demeyer et al. 2003] in a bottom-up approach that begins in the source code [Wagner 2014]. Software migration should incorporate patterns such as a prototyping platform and an incremental increase of test coverage [Demeyer et al. 2003]. The components that are found to be most valuable by the users should be prioritized [Demeyer et al. 2003].

Reengineering is a software maintenance process of gathering understanding about a software system in order to be able to make it more evolvable and to alter it by enhancing its existing functionality [Wagner 2014; Tripathy & Naik 2014]. It requires understanding the three principles of software development: the principles of abstraction, refinement and alteration [Tripathy & Naik 2014]. The need for reengineering can arise from needs pertaining to the betterment of its architecture or from symptoms of a deteriorated system such as code smells [Demeyer et al. 2003]. The six guidelines for reengineering instruct to treat reengineering as a change management project, to formalize methods, to use well-established techniques, to align reengineering with company strategy, to use decision support mechanisms and to follow good software development practices [Pérez-Castillo et al. 2011]. A reengineering process has three steps, four objectives and five approaches. The three-step process [Wagner 2014] – summarized as extract-transform-implement – follows the key principles of software development [Tripathy & Naik 2014] and are depicted as arrows inside the triangles in Figure 3.1. The four objectives are improving

maintainability, improving quality, enhancing functionality and migrating to a new platform and the five approaches are the Big Bang, incremental, partial, iterative and evolutionary approach [Tripathy & Naik 2014].

Forward engineering transforms abstractions, such as ideas, to refined implementations and is therefore a realization of the principle of refinement [Tripathy & Naik 2014]. Understanding the design choices applied in forward engineering is a prerequisite for design recovery [Baxter & Mehlich 2000].

Restructuring follows the principle of alteration and is associated with four change types: rethinking, respecifying, redesigning and recoding [Tripathy & Naik 2014]. Refactoring is a type of recoding and is the most important subtype of restructuring. The purpose of refactoring is to improve the source code without affecting the behavior of the system [Fowler 2019]. It is preceded by a discovery process where the requirements for the targeted part of source code is obtained [Mancl 2001]. Floss refactoring is a class of refactoring that describes small routine refactoring activities and its counterpart is root-canal refactoring which is a process of mandatory corrective refactoring [Murphy-Hill & Black 2008]. Tests have to be implemented before refactoring can be performed [Fowler 2019].

Reverse engineering is a process where understanding of inadequately documented systems is obtained by extracting abstractions of the target system [Baxter & Mehlich 2000; Canfora et al. 2011] through identifying artifacts, discovering relationships and generating abstractions [Tilley 1998]. It embodies the principle of abstraction [Tripathy & Naik 2014]. There are several indications of the need for reverse engineering [Tripathy & Naik 2014] which overlap with the symptoms of a deteriorated system [Demeyer et al. 2003]. The six objectives of reverse engineering are coping with complexity, producing alternative views, recovering lost information, detecting side effects, synthesizing abstractions and facilitating reuse [Chikofsky & Cross 1990]. There are five reverse engineering tasks: program analysis, plan recognition, concept assignment, redocumentation and architecture recovery [Tilley 1998]. Design recovery is effectively a process of applying forward engineering, including underlying assumptions and patterns, backwards [Baxter & Mehlich 2000]. The three activities of reverse engineering are data gathering, knowledge management and information exploration [Tilley 1998]. Information exploration further divides into navigation, analysis and presentation. Data gathering activity and the analysis in information exploration can be performed via four types of software analysis which are static, dynamic, hybrid and historical analysis [Canfora et al. 2011].

5 Analysis and implementation

According to Tilley [1998], a good software engineer is able to keep track of fifty thousand lines of code in their head. At the stage where the build system migration was initialized, Wheel comprised of exactly 38 813 lines of code – with an additional 6191 lines with the HTML template files included, totaling 45 004 lines – which makes it possible for a single person to be able to understand at least a significant portion of the application by heart. Reliance on such a developer always being around may partly explain the lack of thorough documentation. This view is not nuanced since it does not consider that programming languages have differing levels of verbosity and developers have their own preferences whether, for instance, the opening bracket of a code block requires its own line of code.

The migration project is laid out in the upcoming subchapters. The first one contains an overall description of the undertaken process. The subchapters that follow serve as step-by-step process by which the legacy application will undertake its migration. In the second subchapter, a sufficient regression test suite is built to prepare the software for the build system reengineering that is prepared and actualized in the succeeding steps. The third subchapter describes the elimination of implementations of unused or unwanted features that leads to a process of scouring the source code for any dead code. In the fourth one, the build system migration is done. The fifth subchapter is a major refactoring phase in which differing types of AngularJS service constructs are analyzed for salvageable code. In the sixth subchapter, knowledge about the user interface is reverse engineered.

5.1 Description of the migration

As per the requirements presented by the standard for migration [ISO/IEC 14764, 2006], the description of the migration process and a migration plan will be presented.

The process is a single migration project that consists of several reengineering processes occurring within it (see Figure 4.1). The target software system is a web application called Wheel and the migration will affect all the files that are contained in its version control repository such as configurations and the actual source code itself. The core technologies addressed are related to building user interfaces where the legacy solution is called AngularJS while the modern solution is called React.

The migration plan is as follows. The term migration is defined as a change to a new technology platform that includes modern development and build tools as well as a new modern core technology that will completely replace the legacy one. This is achieved

incrementally in a way that supports the existence of legacy technology until it is phased out completely. The state of completion is achieved when the number of occurrences of the text pattern Angular (case insensitive) in the version control repository of Wheel is exactly zero. Four tools have been developed in order to assist in the migration which are called NgTreeify, Elim, Elim2 and Temptor. The source code for these tools – with the exception of Temptor – is closed since the practical part of the Thesis is done for a company. NgTreeify is a tool for structuring a tree representation in JSON format out of used custom AngularJS element directives. It is used in conjunction with an open-source tool called VTree [2019] which is used to visualize the structure. Elim and Elim2 are used to discover dead code. The former detects unused functionalities provided by function-oriented service dependencies and the latter is able to find unused custom directives. Temptor is a shell script (Appendix A) used to find orphaned templates, i.e. those which do not have a construct to utilize it.

To address discrepancies introduced by the newly introduced web design paradigm, the code base is converted to follow a pattern of modularity. The execution of the migration is described in the following subchapters. Since the system is running in a regular manner during the migration, the verification of the migration is achieved by regular operationality of the application. The old environment will not need supporting once the project is complete except for the core technology.

Moreover, as suggested by Demeyer et al. [2003], a prototyping platform is utilized during the project for testing solutions. In this project, the internal administrator application – called Tools – serves as that platform. The benefit of using Tools is that it was initiated as a copy of the legacy application. As a result, the core technologies and the structure of the application is the same. Moreover, using an existing platform means that an additional project does not need to be hosted, although the Wheel application utilizes staging and smoke testing environments that will be utilized in tandem. The drawback of this approach is that Tools is vital for the work purposes of other staff in the company. For this reason, the prototyping cannot leave the platform into a non-functional state and should avoid remnants of partial solutions to avoid interfering with the daily work-related tasks.

5.2 Regression testing

In regression testing, the overall behavior of a software is tested to ensure that it is not affected by changes [Butterfield et al. 2016]. The first major changes will not be on the

unit level since the only difference between the old and the new build system is how the JavaScript files are included into the system. Therefore, there is no need to edit the existing unit tests, however, as was pointed out Chapter 2, a unit testing framework was not implemented for Wheel. Therefore, regression tests – mainly in the form of end-to-end tests – will be relied upon heavily and are made to test the application as thoroughly as possible. For the purposes of the project under study, user stories were created both ex nihilo and from observations on how one set of end users, the customer success staff of the company, used the application on a daily basis. In addition, some end-to-end tests were created as substitutes for integration tests where a test suite was defined for each collection of user interface components that the users perceived as being a singular component.

Additionally, Wheel offers a functionality where automated screenshots of dashboards are sent periodically. For this purpose, a separate visual regression test functionality was built that compares a screenshot to a reference image which automatically alerts the developers if there are discrepancies between the two. Moreover, visual regression can be spotted by people using the software. Therefore, it is beneficial to have a pre-production stage for inspection purposes. In Wheel, a smoke testing platform is used. In smoke testing, software is tested for a passing build with a chance to correct faults before an actual production release [Memon et al. 2005].

5.3 Preparation phase

The preparation phase is concerned with setting up the code base into a state where the work required to perform the largest individual step – the build system migration – is minimized. In practice, this is achieved by reducing the lines of code to a number as minimal possible with the application continuing to perform exactly as it did in the legacy environment excluding those implementations that are deliberately removed. This is achieved in two steps which are feature removal and dead-code elimination.

5.3.1 Feature removal

In the early phases of Wheel, many feature requests were implemented on-demand. Some of these served niche needs of clients who might have churned later on and some of them are ideas that no longer serve the strategic goal of the company. Here, the purpose of feature removal is to cause non-functional code to exist for the dead-code elimination

steps to discover. Therefore, there is an implied assumption that removing implementations will leave instances of dead code that are not detected during the removal phase itself.

Feature removal can be seen as reengineering that takes a requirement for removing a feature as one of its inputs. The main focus will be on reverse engineering; more specifically, a data gathering activity [Tilley 1998] has to take place to collect informal information mainly from documentation and people who have knowledge on the target features [Biggerstaff 1989] followed by a concept assignment activity [Tilley 1998] to understand what the actual implementations are in the source code. One of the most important concerns are the original requirements. Hence, as a restructuring change type, feature removal is mainly concerned with respecifying [Tripathy & Naik 2014]. Forward engineering is simply a matter of actualizing the modified specification. Additionally, as Baxter and Mehlich [2000] point out with regards to design recovery that it is crucial to understand the intention behind the affected feature and to view the removal as applying reversed forward engineering since there was a point in time when a requirement was refined into an implementation. As a visual metaphor, removing a feature can be seen as tearing down a house. In order to perform a recovery, the trees that were cut in order to build the house need to be replanted instead of leaving behind a construction site.

Performing system examination [Tilley 1998] via the use of static and historical analysis [Canfora et al. 2011] may also be beneficial. For example, the code base can be searched for patterns matching the phrase *todo* (as in something someone has *to do*) or variations thereof since it is often used as an annotation to mark sections of source code where an implementation has been left unfinished. In Wheel, only one variation of the pattern is used (TODO) and it is found 24 times in 16 files. For historical analysis, the version control system can be used to trace modification history at some source code location. This can be cumbersome unless using appropriate tools. Since Wheel uses git for version control, developers are able to use the *blame* command to retrieve information on when and by whom the last modification of a file occurred [Git 2019]. To use historical knowledge more effectively, the use of plugins for integrated development environments is recommended.

An observation that can be made regarding this step is that the discovery of these features might be happenstance. Therefore, the discovery of more of such features is expected. As an actual example from the application, one configuration (declared as a JSON structure) used to display a customized structure for a data module supported an attribute

that added an additional element. This information could be found on the documentation of the configuration while the documentation for the affected module was missing entirely and the tool for defining this configuration was a mere JSON editor that did not produce an example configuration as a basis. Therefore, it is possible to speculate that forgetting this feature is a result of the disconnection between the documentations and/or the insufficiency of the tool in an environment where internal tools are relied upon heavily. However, it is also possible to conclude that forgetting a singular feature is simply a result of a form of natural selection where the use of unneeded aspects naturally dies out.

5.3.2 Dead-code elimination

Removing dead code is a refactoring tactic where code that may have once been in use is deleted from the code base entirely [Fowler 2019]. Compilers are usually able to leave such code out of a build automatically which means that bloating the code base is not necessarily a considerable issue. In JavaScript context, some tools such as Webpack are able to perform automatic dead-code elimination in a process called tree shaking where dead code – analogous to dead leaves on a tree – are left out (shaken off) of the final code bundle, resulting in a tree with only live leaves [MDN 2019]. Since the exclusion of non-functional code occurs automatically, the foremost problem with dead code is that it might not signal its obsolescence to the developer. This makes it more difficult to reason about the software and may result in the developer attempting modifications that have no effect [Fowler 2019]. For example, detecting unused AngularJS directives is a difficult task to automate with the provided tools while searching for them manually with a pattern matching technique is trivial which is one of the reasons why the removal step precedes the current one.

In Wheel, the absence of a linter is going to necessitate two separate phases of eliminating dead code. The first one, described here, is concerned with minimizing the code base by removing large-scale functional units that will not be required in the future. Since it is refactoring done for the sake of itself, it is a root-canal refactoring operation [Murphy-Hill & Black 2008]. An example of such a unit is an unused custom AngularJS service definition. In other words, the first dead-code elimination step does not concern itself with minor cases of dead code such as unused variables and functions. The second dead-code elimination step occurs during build system migration where the use of a linter is possible due to its inclusion in Create React App. A linter is a tool of static analysis and is therefore able to automatically reverse engineer the code base in order to detect the presence of the

minor cases of dead code. For this reason, the first elimination phase is geared towards the parts of source code where dead code is detectable via reverse engineering patterns utilized in building AngularJS applications.

One of the tools developed to assist in the migration project is called Elim. It is a static analysis tool whose purpose is to reverse engineer how function-oriented AngularJS services are used throughout the source code. Elim achieves this by leveraging how services are utilized via the name of the service combined with a dot notation (see Figure 5.1). At its core, it is a pattern matching tool that reads source code as raw text and its most important goal is to find dead services.

First, Elim goes through every AngularJS file to collect the list of services that are injected into that construct after which the rest of the file is read to find every instance of a function that is provided through those services. In Figure 5.1, Elim would discover that the controller uses two services, each of which is used for one function. During the analysis, the occurrences of each service and service function are counted globally and on a per-file basis. The result of this yields program metrics called fan-in and fan-out. For a target function, fan-in is the number of functions that call it and fan-out is the number of functions that are called by it [Tripathy & Naik 2014]. For the purposes of this discussion, the target can also be a service: a service with a large number of dependencies has a large fan-out and a frequently used service has a large fan-in. The global counts are useful for determining the importance of a service. For example, in Wheel, the service whose purpose is to facilitate communication with the private API has the largest fan-in and is therefore ranked as the most important. Per-file counts are utilized to examine the fan-out which signals the difficulty of the restructuring effort; a fan-out of zero is the optimal case where the service is trivial to restructure. Restructuring services is a step in the process of migration that will be discussed later.

After processing every file, the services themselves are analyzed for the interface they provide. In Figure 5.1, aService only provides one function whereas bService provides two. When the analysis of service use is reflected against the interfaces, it is discovered that one of the bService functions is not used and is therefore a candidate for removal. Each such occurrence is collected into a list; the removal itself is a manual task. In this project, the analysis yielded a total of 74 functions with 16 false positives. The false positives occurred due to a limitation of Elim: they were functions that were used in the services themselves; for example, if b1Function in Figure 5.1 used b2Function, the function would not be dead but would have been picked up by Elim as a false positive. The

tool could be modified to address this problem automatically, however, since the analysis was run only once and the removal required manual labor that is able to detect these false positives, this was not considered important.

A similar analysis can be performed on directives. In this project, every directive is only used in HTML templates and every element directive has a template associated with it. Therefore, there are two simple cases which can be analyzed for with pattern matching: orphaned templates and dead directives. Detection of orphaned templates is performed before and after the analysis for dead directives and is performed with Temptor (Appendix A) which searches template file names that are missing in the source code. Initial run of the tool yielded a result of sixteen templates, five of which were subsequently removed since the analysis yielded eleven false positives. The false positives were templates used in the application router definition rather than the directives.

Detecting dead directives is more complicated due to directive name normalization [AJS Guide 2018] in which AngularJS automatically transforms the dash-delimited names used in HTML to their camel case alternatives that are used in directive definitions (see *dataModuleElement* in Figure 5.2). The Elim2 tool is similar to Elim: first, every HTML template is scanned for patterns matching non-standard HTML elements after which the element names are normalized the way they are normalized by AngularJS and compared against a list of every custom element directive definition found. Elim2 is designed to find only element directives since the other types are used infrequent. For the code in Figure 5.2, Elim2 would detect that the normalized directive name *dataModuleElement* is used while *deadDirective* is not, thereby making it a candidate for deletion. For Wheel, the Elim2 analysis yielded a result of nine dead directives with one false positive that was caused by a custom directive whose template was not declared in an HTML file but as a string. A Temptor analysis after removing the directives with their templates resulted in the elimination of two more templates when accounting for the false positives.

```
/* mainController.js
*/
angular.module("app")
  .controller("mainController", function (aService, bService) {
    aService.a1Function();
    bService.b1Function();
  });

/* aService.js
*/
angular.module("app")
  .service("aService", function() {
    this.a1Function = () => console.log("a1");
  });

/* bService.js
*/
angular.module("app")
  .service("bService", function() {
    this.b1Function = () => console.log("b1");
    this.b2Function = () => console.log("b2");
  });
```

Figure 5.1 An AngularJS controller with two injected dependencies that provide a total of three functions with two being in use.

```
<!-- containerTemplate.html -->
<!-- this dash-delimited declaration is normalized to camel case -->
<data-module-element> </data-module-element>

/* dataModuleElement.js
*/
angular.module("app")
  // directive declaration with a normalized name form
  .directive("dataModuleElement", () => {
    /* an element directive */
  });

/* deadDirective.js
*/
angular.module("app")
  .directive("deadDirective", () => {
    /* a dead element directive */
  });
```

Figure 5.2 An example of directive name normalization and a dead directive.

5.4 Build system migration

Changing the build system is the largest individual reengineering procedure in the process as a whole. More specifically, it can be classified as a Big Bang reengineering approach since the entire system is affected in a way that requires every component to be modified for the new environment [Tripathy & Naik 2014] because new build system is going to introduce complications as well as modern web design conventions and paradigms that need to be addressed. In addition, a compiler will be introduced with a linter tool, both of which were absent in the legacy build system which only relied on the browser to enforce the rules of the strict mode. Furthermore, the compiler runs in Node instead of the browser environment which causes a minor disconnection, however, under most circumstances it is not significant.

Due to the introduction of the compiler and the linter, steps of necessitated root-canal refactoring [Murphy-Hill & Black 2018] will take place which will likely require reverse engineering in a process of discovery [Mancl 2001]. Moreover, after the application is refactored to make it compile successfully, AngularJS is likely going to face runtime errors.

5.4.1 Application setup

The application setup phase contains four steps. The first is annotating dependencies of AngularJS constructs. The second step is scaffolding a blank project using Create React App. In the third step, existing files are included into the blank project. In the fourth step, dependencies from the legacy package manager are installed via the new package management system.

In the first step, the dependency injections of AngularJS constructs are modified to enable running the application after building a production version of it. The dependency injection system on AngularJS relies on annotating the dependencies to the injector subsystem [AJS Guide 2018]. There are three annotation methods, one of which cannot be used: the implicit annotation method. In implicit annotation, the function parameter names are used for acquiring a dependency. In Figure 5.1, mainController has two implicit dependencies (aService and bService) whose identifiers as parameters for the controller definition match the names given in the service definitions (the first parameter). However, parameters in function definitions can have arbitrary identifiers which is exploited by minification – a sub-process used in building production code bundles that reduces the file size to enable quicker download [MDN 2019]. To achieve this, one of the

procedures of minification is shortening identifiers to arbitrary single-character names. With implicit annotation, the injector subsystem is not able to recognize dependencies since minified identifiers do not match the names defined by developers. The legacy build system of Wheel included a process minification in conjunction with automated explicit annotation which enabled developers to utilize implicit annotation despite minifying it. Since the production build process in Create React App includes minification and cannot be configured trivially, implementing an automated explicit annotation step is more difficult than modifying the code manually to employ explicit annotation. There are two methods to achieve this with no significant differences. For Wheel, the inline array annotation is used since it is the recommended method [AJS Guide 2018]. In inline array annotation, the dependency names are explicitly listed alongside the function definition (Figure 5.3). This should be combined with the directive that forces strict – i.e. explicit – dependency injection, causing a runtime error when implicit annotation is used [AJS API 2018].

```
/* mainController.js
*/
angular.module("app")
  .controller("mainController", [
    "aService",
    "bService",

    function (aService, bService) {
      aService.a1Function();
      bService.b1Function();
    }
  ]);
```

Figure 5.3 The controller from Figure 5.1 defined with explicit dependency annotation.

The second step in the application setup is the creation of a new application scaffolding. This requires npm which includes npx [npx]. It is a software package that is used is used to run npm packages such as Create React App through a command-line interface (CLI). In Figure 5.4, two commands are introduced. The first one creates a regular project whereas the second one is a variation that installs a project with customized scripts. In the Figure, the customized script package introduced in Chapter 2 is used which allows importing HTML files as strings. A structure of a blank project generated by CRA is presented in Figure 5.5.

```
# creates a project called wheel
npx create-react-app wheel

# the same as above with customized react-scripts
npx create-react-app wheel --scripts-version html-loader-react-scripts
```

Figure 5.4 Two methods of scaffolding a new React project.

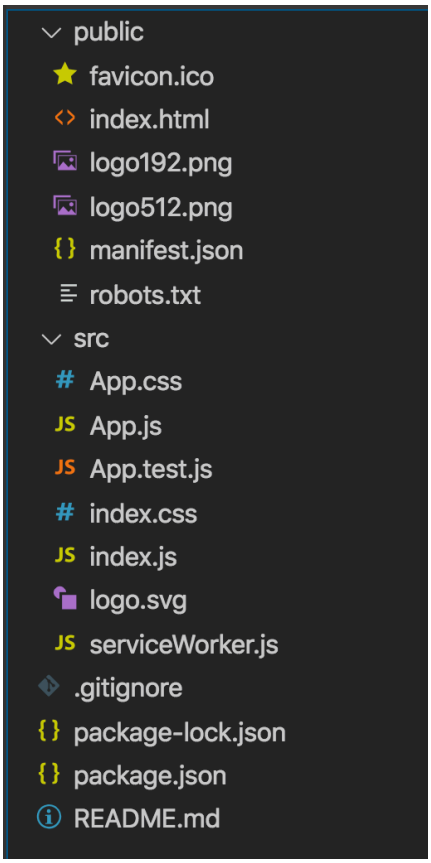

Figure 5.5 The folder structure of a new Create React App project. The dependency folder is omitted.

The third step is transferring files of the legacy platform into the newly scaffolded project. Create React App mandates three rules for a project. First, static assets that are not used modularly must reside in the public folder [CRA 2019]. The second rule is that source code has to be inside the source code folder (src in Figure 5.5). The third rule is that the starting point of an application is a specific file: index.js (see Figure 5.5) [CRA 2019]. To adhere to the first rule, every HTML template file used for AngularJS directives has to be moved to the public folder unless dynamic HTML imports are accessible. In addition, the contents of the new index HTML file need to be replaced with the contents

of the equivalent document in the AngularJS project. To satisfy the second rule, the entire scripts folder or its contents (see Figure 2.2) can be moved inside the new source code folder. In addition, the public folder has become the root folder of static files and therefore the file paths to AngularJS templates have to be modified to reflect this change. For the third rule, the first step is to remove the contents of index.js and every other file in the folder[1]. By default, the index.js file contain a React application renderer which will not be utilized. Instead, the file will serve as an alternative to script inclusion in the browser (Figure 5.6). In the figure, imports are used without storing values to variables. It is a process of side-effect inclusion via running the module code which emulates how script inclusion functions in a browser. The result of the third step when applied to the blank projects can be seen in Figure 5.7.

In the fourth and last step, the project dependencies are reinstalled using the new package management system. The dependencies downloaded using Bower are listed in bower.json file. This step is straight-forward; however, it is important to pay attention to the version numbers to avoid introducing incompatibilities. Additionally, some dependencies might be included via content delivery networks (CDN). A CDN is a system of computers designed to service copies of data that an end-user requires [MDN 2019]. An approach to this problem is presented in the following subchapter.

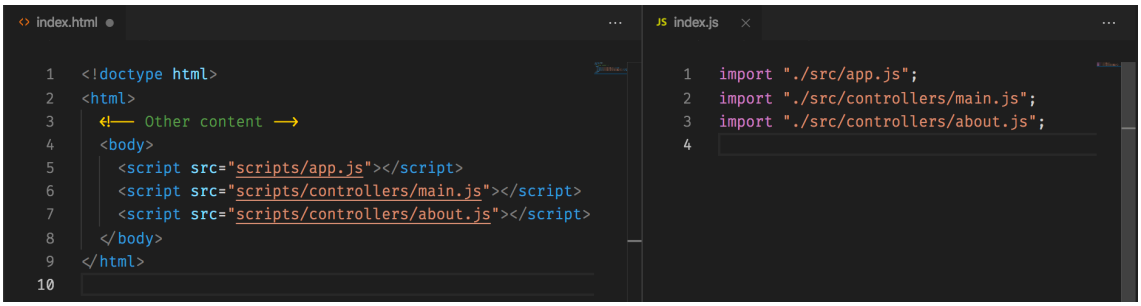

Figure 5.6 Moving script inclusions from a browser to a Webpack entry file.

[1] The service worker file can be preserved alongside the manifest file in public folder. They relate to progressive web applications (PWAs) which are websites that behave like native applications [MDN 2019]. In this thesis, PWAs are not discussed further.

Figure 5.7 From left: the blank AngularJS project (Figure 2.2) migrated into the blank CRA project (Figure 5.5) and the end result. Red overstricken files are removed, brown underlined files are replaced with their counterparts and/or modified and green files are newly introduced.

5.4.2 Post-setup refactoring

When included in a page, AngularJS attaches itself to the global window context after which it can be referenced from the scripts that follow without a separate declaration. For this to function, AngularJS has to be included before the scripts that reference it. In projects generated with the Yeoman generator for AngularJS [YO Angular Generator 2019], this is the default case. Since the new build system will run in Node, the code will not be in browser context (window) where such global variables are implicitly understood. Therefore, compilations of the code base will fail. Instead, the global context of Node is used. However, Node supports access to the global variables of conventional browser-based JavaScript [Node 2019]. Since global variables of a browser are properties of the window object, AngularJS can be accessed by using regular dot notation (see Figure 5.8). Since CDNs are included as scripts in the same manner, the same pattern can be followed for them.

In modern JavaScript projects that include a linter, identifiers need to be declared explicitly. The solution is to have AngularJS installed as a dependency via a package manager which is then imported into every file that needs to access it. Since the advantage of a CDN is a potential increase in page load speed, using it may be the preferred option in which case AngularJS can be accessed as a property of the window object.

However, if jQuery is used as a dependency, it needs to be included before AngularJS. This mandates a hybrid solution where it needs to be attached to the global context to be referenced by AngularJS. This can be achieved via a CDN or by installing it as a dependency and utilizing an import to run its side effects similar to how the scripts are included in Figure 5.6.

In Wheel, every file contained a reference to the implicit global AngularJS and therefore required more root-canal refactoring [Murphy-Hill & Black 2008] to enable compilation. As such, each occurrence was refactored to contain an explicit import instead. This was achieved with the help of the newly introduced linter. Moreover, the linter enables performing the second dead-code elimination step mentioned in the preparation phase.

```
/* Assuming that AngularJS is in the window object,
   this would work if included as a script in a browser but
   causes the Webpack compiler to fail since angular is not declared.
*/
angular.module('app').service('fooService', [function() { ]}

/* The compilation fails only because of the linter.
   Therefore, instructing the linter to ignore a line or the file
   will result in a working compilation.
*/
// eslint-disable-next-line
angular.module('app').service('fooService', [function() { ]}

/* The valid method of accessing the global context of a browser.
*/
window.angular.module('app').service('fooService', [function () {
   /* working compilation */
]}

/* Here angular is declared explicitly
*/
import angular from 'angular';

angular.module('app').service('fooService', [function() {
  /* working compilation */
]}
```

Figure 5.8 Illustrating the concept of the global scope of a browser in Node.

5.4.3 The second dead-code elimination phase

The linter is a static analysis (i.e. reverse engineering) tool. It can cause two types of outputs: errors and warnings. Errors cause compilations to fail. For example, the usage of an undeclared variable is considered an error. This feature aids in detecting where implicit global variables – such as AngularJS – are utilized. Warnings allow the compilation to happen but will prohibit building a production release version of an application in a CI/CD pipeline. An unused variable declaration, for instance, will cause a warning. The errors are convenient for detecting and removing dead code after which forward engineering can be performed to remove it.

However, the linter cannot detect all cases of dead code. Most notably, it is not able to recognize unused function parameters in CRA-based projects and, therefore, is not able to detect if an injected dependency is utilized. This causes inaccuracies when counting the fan-ins and fan-outs of services. A method to mitigate this problem is to make services importable. This can be achieved by leveraging the AngularJS injector service [AJS API 2018] to store injectables into exported variables. An additional benefit of having importable services is the ability to write tests for them using the new unit testing framework. This method is presented in Appendix B.

The linter is not aware of certain coding styles and will indirectly enforce the use of other styles which might inadvertently cause issues. For example, JavaScript contains two operators for comparing equality: regular equality comparison (weak) and the comparison of the value and the type (strong) [W3S JS 2019] (Figure 5.9). The weak comparison is performed with two equals signs and does not consider the types of the operands. Therefore, a number is equal to its stringified counterpart. The strong comparison uses three equals signs and will consider a number to be different from a string representation of that number. The linter will enforce the use of the strong comparison in almost every case. In Wheel, every comparison was performed with the weak operator and was changed to the strong one due to the linter. As a result, some user interface elements malfunctioned due to cases where numbers were compared to strings of those numbers. Therefore, the linting would have benefitted from reverse engineering how data is obtained from the private API compared to how it is stored and used in the front end.

```
x = 3;   // x is of type number
y = "3"; // y is of type string

x == y;  // true
x === y; // false
```

Figure 5.9 Illustrating the difference of JavaScript equality comparison operators.

5.5 Salvaging reusable code

Facilitating reuse is one of the main objectives of reverse engineering [Chikofsky & Cross 1990]. The output of the Elim tool is therefore useful here as it provides a list of services with their fan-in and fan-out counts. As a general rule, services have potential for reuse due to being designed for reuse. On the contrary, directives and controllers cannot be reused due to being integral constituents of the MVC pattern of AngularJS. Therefore, the following subchapters will solely focus on services. The subchapters are divided according to the service characterization presented in Table 3.1.

Two general types of refactoring methods are presented to support reuse: the teardown and the rewrite method. The teardown method refers to keeping the business logic intact while removing the AngularJS structure around it. The rewrite method has two approaches: paralleling and coupling. Paralleling refers to creating a parallel implementation in which a new file is created that mimics its AngularJS counterpart. In coupling, a dependency is moved to where it is used utilizing a refactoring tactic called Move Method [Fowler 2019].

5.5.1 Function-oriented services

Services used to house reusable functionality are characterized as function-oriented services and can then be used by injecting them as dependencies for other components. In Wheel, almost every function-oriented service is a factory, i.e. they return an object in contrast to the service-type counterparts which are instantiated. Regardless of the type, function-oriented services mostly expose methods that are plain JavaScript; as an example, Figure 5.10 contains a factory called datesFactory that has a public method daysBetweenDates aliased as daysInBetween. Methods such as these are trivial to move out of AngularJS components, however, the design pattern needs to be chosen by the developer. The most naïve solution is to adapt the singleton pattern used by AngularJS in delivering services [AJS Guide 2018] (Figure 5.11).

```
/* datesFactory.js
*/
angular.module("app")
  .factory("datesFactory", [function () {
    const somePrivateProperty = false;

    function daysBetweenDates(date1, date2) {
        // .. compute and return the result
    }

    return {
      daysInBetween: daysBetweenDates
    )
  }]);

/* someController.js
*/
// usage - inject as a dependency
let days = datesFactory.daysInBetween (/* parameters */)
```

Figure 5.10 An AngularJS factory exposing a method for calculating the number of days between two dates.

Singleton pattern refers to a design pattern where a code module has only one instance with a global point of access [Gamma et al. 1995]. Singleton pattern is useful in situations where exactly one instance is required, such as a file system in an operating system. Its benefits include easy extensibility to, for example, support multiple instances alongside characteristics such as strict control over how the instance can be used.

The benefit of the singleton pattern for the legacy application is that, once imported and instantiated, there is no need to modify how the service is used from within the business logic itself, that is, using the namespace with dot notation[2]. The end result, then, is a matter of switching the dependency injection to a standard import. However, in conventional JavaScript, this pattern is verbose. Additionally, Osmani [2017] notes that often the use of this pattern in JavaScript signifies a need for redesign since their use can be a sign that modules are too tightly coupled, or that business logic is too widely spread in the source code. Since the pattern is what AngularJS framework imposes on the developer, one can make the argument that it also enables the manifestation of these problems. In Wheel, it is common to see these antipatterns.

[2] The instance variable has to be named according to the AngularJS namespace rather than the import. Therefore, one must either use a default export or alias any named imports.

In modern JavaScript, importing modular code is a feature of the language and will, therefore, be utilized here as well. According to the types of patterns described by Osmani [2017], the module imports follow the revealing module pattern. It shares similarities with how AngularJS services are used. First, the pattern is similar to a factory object pattern used in factory-type AngularJS services. Second, it allows the developer to opt for using the services with a dot notation. However, unlike in the pure factory pattern, exports cannot be aliased. This can be worked around by using an explicit middle variable in the module. Each of these features is demonstrated in Figure 5.11.

```
/* datesFactory.js – using ES6 imports/exports
*/
const somePrivateProperty = false;

function daysBetweenDates(date1, date2) {
  // .. compute and return the result
}

// aliasing an export with middle variable
const daysInBetween = daysBetweenDates;

export {daysInBetween}

/* someController.js
*/
// usage – import either with a name
import {daysInBetween} from "./datesFactory";
// .. or as a namespace
import * as datesFactory from "./datesFactory";

let days1 = daysInBetween(/* params */);
let days2 = datesFactory.daysInBetween(/* params */);
```

Figure 5.11 An AngularJS service modified to reflect the revealing module pattern.

Osmani [2017] points out that committing to any single pattern is not recommended since different approaches may work better in different situations. However, the objective here is not to redesign but to disentangle logic from AngularJS structure. Therefore, opting for the path of least resistance – one that resembles the existing structure as much as possible – is the optimal solution. Therefore, the suggested course of action for migrating AngularJS services is to apply the revealing module pattern. To accomplish this, the teardown and the rewrite methods are both used accompanied with unit tests. The services

that are not trivial to tear down or rewrite should be declared strictly deprecated. In other words, their use as-is is not allowed in any newly produced code.

When teardown is utilized for factory-type services which have no dependencies (fan-out is equal to zero), the result is no longer tied to AngularJS and is therefore pure (compare datesFactory implementations between Figures 5.10 and 5.11). If injections are modified to be importable (Appendix B), AngularJS can be torn down from almost any service, however, services utilizing imported injectables are still impure by association. For instantiated services, the approach is not as straight-forward. However, transforming them to factories is simple and is therefore recommended (Figure 5.12).

The parallel rewrite method can be performed during the upcoming reengineering tasks since they automatically target only the required services. However, services that are often used throughout the application (a large fan-in) may need to be rewritten in their entirety in one go. Coupling should be performed in cases where fan-in count of a service or just one function in a service is exactly one.

```
/* instantiatedService.js
*/
angular.module("app")
  .service("aService", function() {
    this.aFunction = () => console.log("aFunction");
  });

/* asFactory.js
*/
angular.module("app")
  // despite being a factory function,
  // this service does not need to be an AngularJS factory construct
  .service("aService", function() {
    const aFunction = () => console.log("aFunction");

    return {
      aFunction: aFunction
    }
  });
```

Figure 5.12 An instantiated service is turned into a factory function.

5.5.2 Value-oriented services

Refactoring AngularJS values and constants can be performed similar to a service teardown since their fan-out is necessarily zero as they cannot have dependencies. The revealing module pattern is also applicable (Figure 5.13). Since constants are by their

nature immutable, this approach has no major drawbacks. Values, on the other hand, support the decorator pattern [AJS API 2018] and can thus be modified dynamically before they are injected as dependencies. In Wheel, values are used as constants which cannot be determined when the application is bootstrapped. For example, a value that stores user information is empty by default. After a login, the value is used to store the user's email address. In such a case, a means to use another type of value storage is needed. One option is the browser storage.

```
/* angularNumberConstants.js
*/
angular.module("app")
  // .constant could also be .value
  .constant("listOfNumbers", {
    one: 1,
    two: 2
  });

/* numberConstants.js
*/
const listOfNumbers = {
  one: 1,
  two: 2
}

export { listOfNumbers }
```

Figure 5.13 A constant object with and without AngularJS.

5.5.3 Service-like services

Service-like services consist of filters. Filters allow formatting values in templates without affecting the model. This characteristic makes them special in an AngularJS application despite resembling regular functions. If filters are exclusively used in HTML templates, the best course of action is to ignore them. Eventually, they might be required in the JSX code intended to replace legacy templates. Where filters are used as if they were function-oriented services, the methods of teardown and rewrite can be applied. Their classification provides grounds to reimplement them, for example, as one code module which is a collection of filters.

5.6 Reverse engineering the user interface

Regular HTML documents employ hierarchies of elements by nesting them. In the same manner, a typical AngularJS application has a hierarchy of element directives. Reverse engineering this hierarchy provides a view of the overall software structure. The NgTree-ify tool mentioned earlier is able to perform this by constructing a JSON structure that can be visualized as a tree of components. This is achieved via leveraging how Wheel is constructed as a single-page application that has a singular mounting point in which the application router is defined. AngularJS router matches URLs to controllers that have their associated templates [AJS API 2018]. The templates in the router, in turn, are constructed out of regular HTML elements as well as element directives. Therefore, the router definition is the root of the tree, the first level consists of routes. Below the router level, directives exist in arbitrary orders. To illustrate this, the migrated project in Figure 5.7 (the rightmost structure) was modified to include two element directive definitions – Hello and World – where latter is nested inside the former (Figure 5.14). Additionally, the World directive is used in one route definition by itself. The JSON structure generated according to this project is depicted in Figure 5.15 and a tree visualization of it in Figure 5.16.

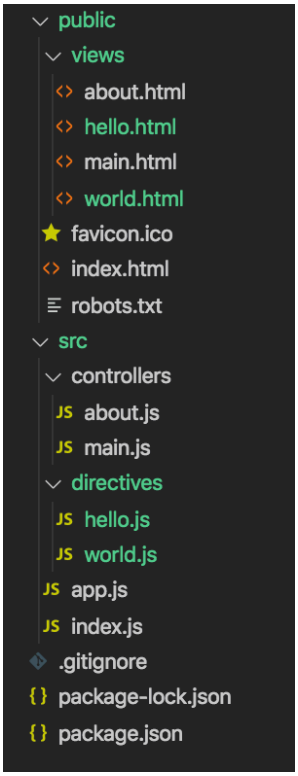

Figure 5.14 The structure in Figure 5.7 modified to include element directive definitions (highlighted in green).

```
{
  "root": "app.js",
  "routes": [
    {
      "route": "/",
      "ctrl": "MainCtrl",
      "template": "main.html",
      "components_MainCtrl": [
        {
          "name": "hello",
          "template": "hello.html",
          "components": [
            {
              "name": "world",
              "template": "world.html",
              "components": null
            }
          ]
        }
      ]
    },
    {
      "route": "/about",
      "ctrl": "AboutCtrl",
      "template": "about.html",
      "components_AboutCtrl": [
        {
          "name": "world",
          "template": "world.html",
          "components": "(already done)"
        }
      ]
    },
    {
      "type": "otherwise",
      "ctrl": null,
      "route": "/"
    }
  ]
}
```

Figure 5.15 A JSON structure generated from the empty project with additional element directives.

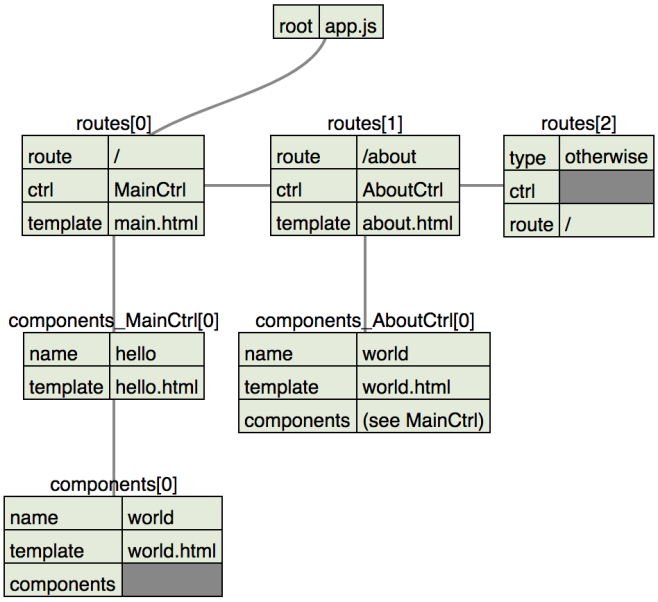

Figure 5.16 The JSON structure from Figure 5.15 presented as a tree using the VTree tool.

5.6.1 Individual user interface elements

The benefit of a tree depiction is realized when approaching the reengineering of user interface elements with the bottom-up approach [Wagner 2014]. The approach requires creating React component mounting points on the basis of individual directive elements. It will follow the incremental reengineering approach where intermediary steps yield a hybrid application [Tripathy & Naik 2014]. Moreover, the bottom-up approach avoids having layers of legacy implementations in the midst of new React implementations. To address the requirement of non-layered end result, one rule is proposed which states that data to AngularJS implementations should never flow through React components. This rule will be referred to as The Rule since it is the most important one.

However, the legacy code that contains React components may still need to be aware of events triggered inside React context. For instance, a click event inside a simple component that determines the color scheme of the entire application needs to be broadcast to the entire AngularJS context. In plain AngularJS, this is performed by mutating a two-way bound state variable that is an attribute in the model (scope). It follows an observer pattern where objects are subscribed to changes in another object they depend upon [Gamma et al. 1995]. While this pattern needs to be exploited, mutations of the state or properties should not occur within React component [React 2019]. Therefore, an intermediary stage is required which necessitates the presence of an intermediary directive.

The existing directives will be repurposed for this role and will be referred to as limbo directives. For instance, a structure to support turning the world directive in to a React component is depicted in Figure 5.17.

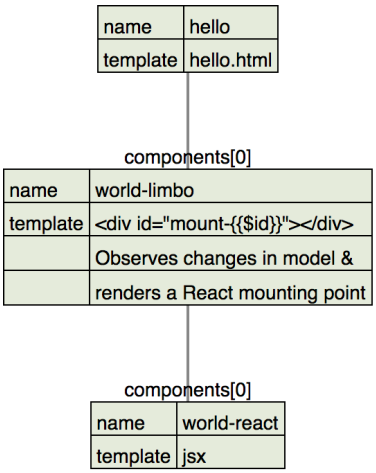

Figure 5.17 The World directive repurposed into a limbo directive that renders a React version of itself. Parts of the tree structure are omitted on purpose.

A limbo directive is a husk of the original implementation and is required to keep the behavior of the framework consistent. It should contain a controller whose only purpose is to render the React component and observe relevant changes. Moreover, they represent roots of subtrees that consist of only React components. As such, the limbo directive status is shifted if every following condition is met: another directive is turned into a React component, that directive has a limbo subtree and it is closer to the root of the entire tree than the current limbo directive. In accordance with The Rule, the initial limbo directive is removed entirely. Moreover, aspects of the limbo directives can be mapped the React component: the isolated scope is equivalent to React component properties and the scope (model) in the controller is equivalent to the state (see Table 3.2). Additionally, the template is converted into JSX.

The HTML template of a limbo directive can be replaced with a template string that contains a singular HTML element into which the React element is rendered. Moreover, the element should contain an interpolation of a unique identifier. For that purpose, the identifier automatically provided in the scope ($id) is exploited by interpolating it into the ID property. Ordinarily, the identifier is used by AngularJS internally and is not needed by developers. However, it is useful for this purpose since it is needed to identify individual instances of React components. Without the identifier, every instance of the

world directive would render their respective versions of the React component into the first instance of the mounting point in the document.

Due to the introduction of the interpolated value in the AngularJS template string, template compilation is required to replace the interpolation with an actual value. AngularJS template compilation can be slow and, therefore, a method of waiting for the completion of a compilation needs to be taken into account. The most optimal solution is to utilize the timeout service ($timeout) provided by AngularJS [AJS API 2018]. The service runs delayed functions. If a digest cycle is active, $timeout will wait for the cycle to finish before initiating the delayed execution. As such, a function to render a React component can be passed as the delayed function and the delay does not need to be defined as the rendering should happen immediately after a template is compiled. Another solution is to use the native timeout [W3S JS 2019] function to wait arbitrary amounts of time. Native timeouts may be useful for explicitly waiting for animations to complete and can be used in concert with the $timeout service.

As was discussed in Chapter 3, React elements are rendered using a declarative API that initiates reconciliation. Inside React components, this API is called with the setState function or is automatically run when a component receives properties from its parent component. In a limbo directive, modifying the properties requires mounting the component again, however, React is able to recognize this as a re-render of an existing component. Therefore, the function used to render the component initially can utilized. This can be combined with the observer pattern by having explicit watcher expressions inside the limbo directive whose purpose is to fire a re-render after the model (scope) is mutated. Unfortunately, due to the design of AngularJS, watcher expressions are always run on initial loads which can cause premature rendering. In Wheel, this is prevented using boolean variables that are set to allow re-rendering only after the initial render.

Figure 5.18 presents a summary of the process. In the figure, analogous concepts are mapped in an implementation level: a transformation of HTML into JSX (yellow), the new React mounting point (red), an isolated scope as component properties (green) and the model as the state (blue). Moreover, the controller definition includes the use of $timeout to trigger the initial render and to set the boolean to allow re-rending. Finally, a watch expression for the two-way bound model attribute is used to render the React component again when it is mutated. As is seen in the figure, the function that initializes the React component only has the directive scope (i.e. model) as a parameter. This pattern is followed for each user interface component.

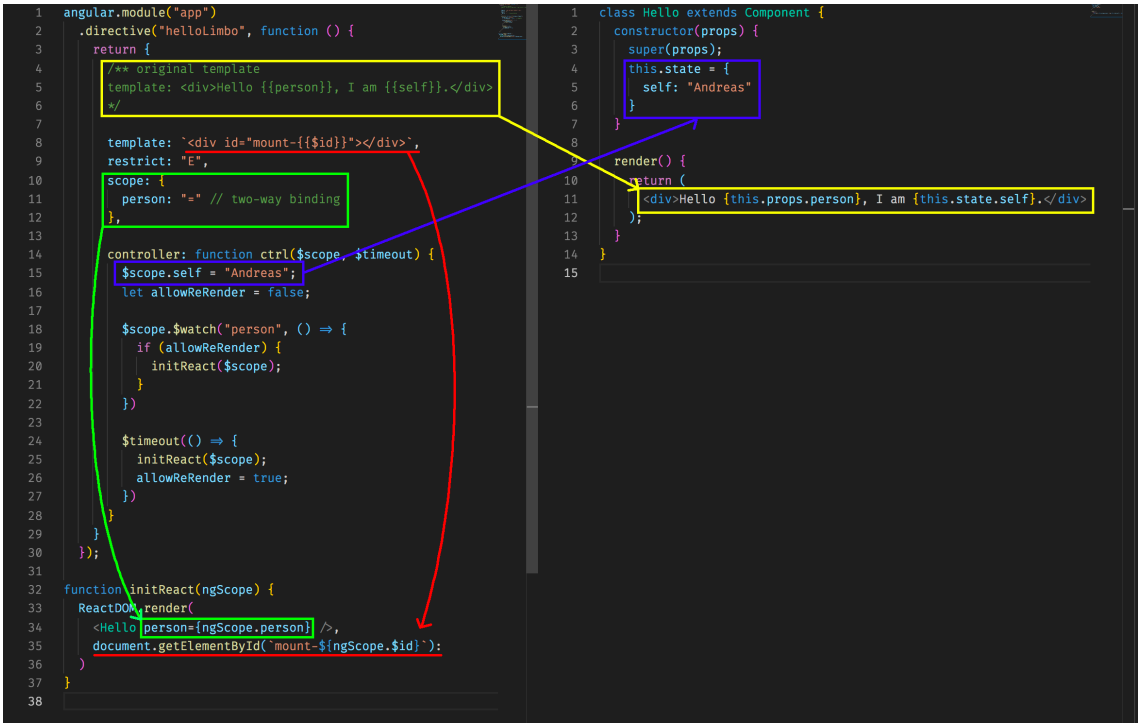

Figure 5.18 A directive that has been transformed into a limbo directive alongside the React implementation of it.

If an element directive does not utilize a model or an isolated scope – such as a typical loading indicator – a limbo directive is not needed. Rendering it is the responsibility of the parent directive. However, these types of user interface elements typically rely on their context of use. For example, an element that fetches data asynchronously to be displayed in a chart may include a loading indicator to signal an ongoing process of acquiring data. Therefore, the indicator is integral to the chart and, as such, they should be reengineered together. Recognizing element directive entities that belong together is a matter of understanding the problem domain that benefits from reverse engineering informal knowledge as well as historical data to see where implementations emerge simultaneously.

5.6.2 Recovering the design of Wheel

An abstraction of the overall design of a Wheel dashboard is depicted in Figure 5.19. It represents one of the routes defined in the application (compare to Figure 5.16) where each individual block may consist of one or more directives. The realization of this abstraction is presented in Figure 5.20. The underlying structure is defined as a JSON object (Figure 5.21) that is stored and fetched using the private API.

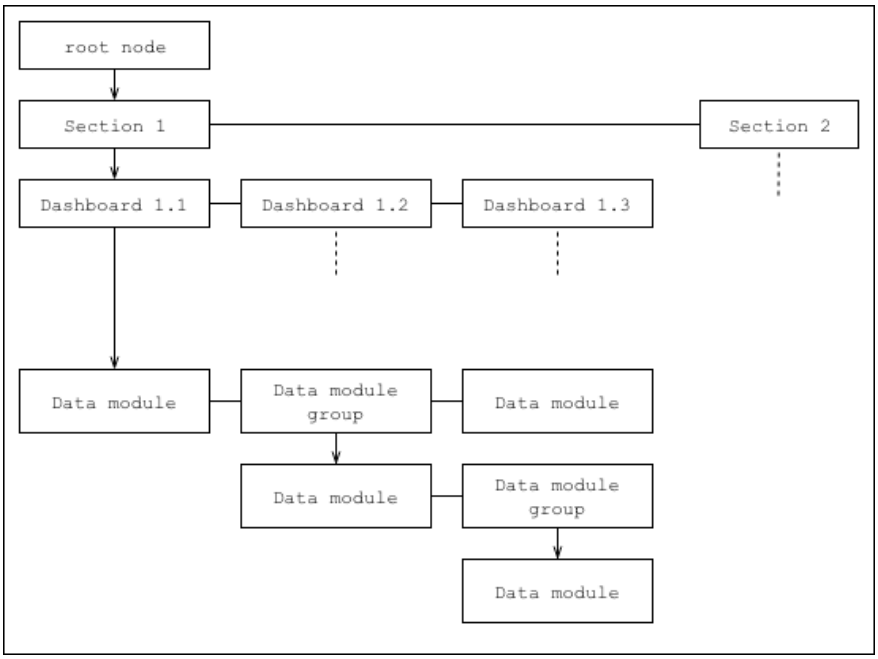

Figure 5.19 An example of a dashboard structure; downward arrows describe hierarchy (through nesting) while regular lines denote parallel components.

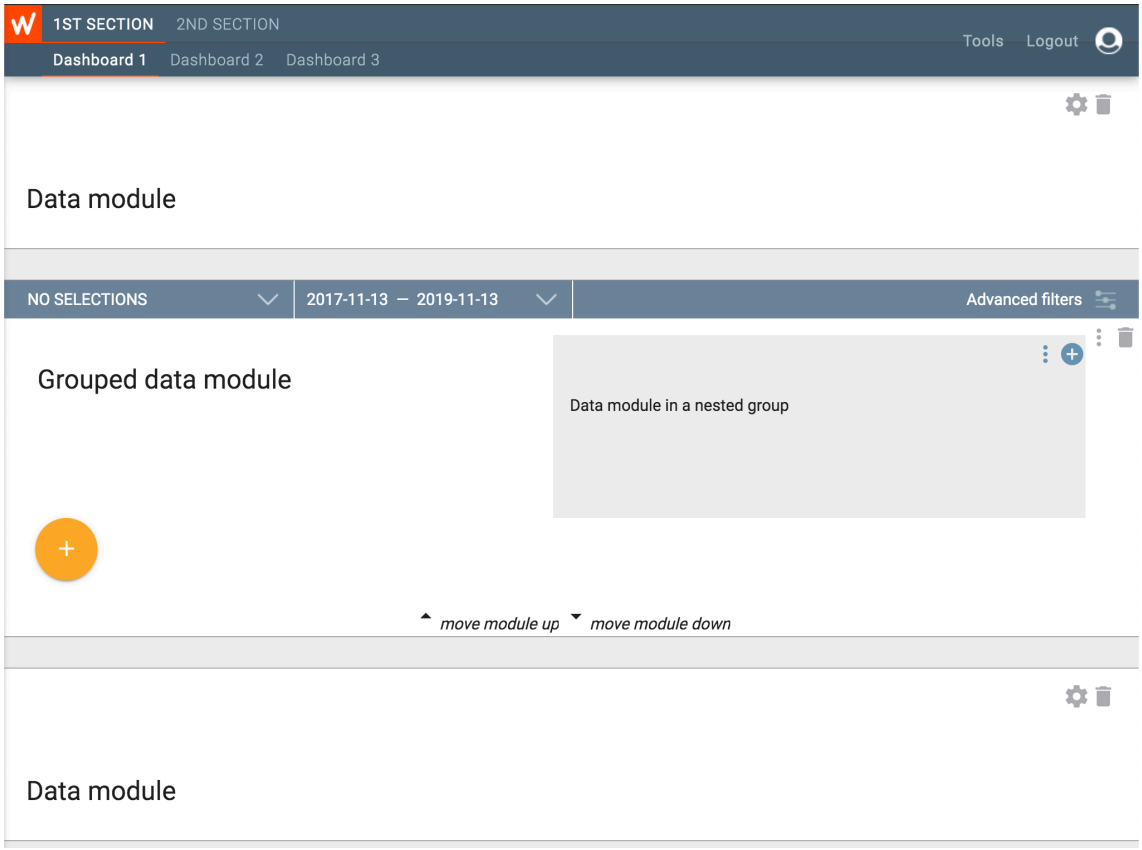

Figure 5.20 The realized version of the abstraction in Figure 5.19

```
{
  "sections": [
    {
      "name": "1st Section",
      "dashboards": [
        {
          "name": "Dashboard 2",
          "modules": [
            {
              "type": "data module"
            },
            {
              "type": "group",
              "modules": [
                {
                  "type": "data module",
                  "title": "Grouped data module",
                },
                {
                  "type": "group",
                  "modules": [
                    {
                      "type": "data module",
                      "title": "Data module in a nested group"
                    }
                  ]
                }
              ]
            },
            {
              "type": "data module"
            }
          ]
        },
        {
          "name": "Dashboard 2"
        },
        {
          "name": "Dashboard 3"
        }
      ]
    },
    {
      "name": "2nd Section"
    }
  ]
}
```

Figure 5.21 The underlying JSON structure of the dashboard view in Figure 5.20 (simplified from the actual).

As was described in Chapter 3, nested element directives that have isolated scopes create a hierarchy of models. When the JSON structure in Figure 5.21 is studied and

mapped to its visual counterpart in Figure 5.20, it is revealed that the dashboard structure contains information that is used in the model of the directives. For instance, the data modules that are not grouped seem to be using a default title derived from the type attribute whereas the grouped data modules have custom titles in the view as well as the structure. From this observation, it can be extrapolated that the JSON structure is travelled in a tree-like fashion and split into directive scopes at correct levels. The React implementations, therefore, should be synchronous with the models as they are provided by limbo directives and with the structure stored in the private API as well.

Data module groups are one of the most important features for the end user and reengineering them should have a high priority [Demeyer et al. 2003]. However, since the data module groups are designed to be collections of more specified data modules, the directive reengineering tasks cannot begin from them or data from React implementations would have to flow to AngularJS implementations which is a violation of The Rule. One way to address this is study the group module structure further using the output of NgTreeify to see if directives parallel to the data modules can be found. For instance, in Figure 5.20, the blue bar on top of the group module is implemented as a React component since it used to be a parallel directive. Moreover, since it is a stateless component, it does not need a limbo directive.

5.7 Summary

Two tables are presented as a summary of two aspects of the migration process. Table 5.1 contains the results of each performed analysis. The first four analyses were performed as one-off runs in the preparation phase to reduce the total amount of code. The ESLint analysis is integrated into the development environment and is therefore constantly active. An analysis with NgTreeify can be performed on demand and is done after the completion of each user interface component reengineering. Table 5.2 summarizes key principles or methods that are introduced in their related subchapters.

Tool	Purpose	Result	False positives
Elim	Detects dead services. Counts fan-in and fan-out of each service.	74 functions. Fan-in and fan-out counts of services as a whole & each function of each service.	16 functions
Temptor (1st)	Detects orphaned templates.	5 file removals.	11 files
Elim2	Detects dead directives.	9 file removals.	1 file
Temptor (2nd)	Detects orphaned templates.	2 file removals.	11 files
ESLint	Detects unused variables and enforces a coding style.	Modifications to every file.	Weak equality comparisons.
NgTreeify	Creates a tree representation of the application.	An overview of the application that is used in reengineering individual interface components.	N/A

Table 5.1 Results of the analyses in the order they were performed during the project.

Phase	Principles/methods	Description
Preparation	Minimizing	Reduce the amount of migrated code to minimize the required labor.
Build system migration	Accommodation	Create a new application scaffolding and adjust the legacy implementation to adhere to its rules.
Service reuse	Teardown	Services with zero fan-out can be used as-is by tearing down the AngularJS structure around them.
	Rewrite	Refactoring code by either copying it into a new file on demand (paralleling) or by using the Move Method refactoring tactic (coupling).
User interface reengineering	The Rule	Data from React implementations should never flow into AngularJS implementations.
	Limbo directives	Intermediary element directives that handle correct model usage in AngularJS.

Table 5.2 The key principles and methods introduced for the migration phases.

6 Discussion

Due to the classical software reengineering approach to the migration process, many of the artifacts produced – such as the component tree models of the application structure – run a risk of becoming obsolete right after actions are taken based on them. Furthermore, when a reengineering process needs to be applied again in the future, artifacts of the system built during the process described in this thesis need to be discovered in an equally labor-intensive manner. Wagner [2014] states that this problem arises from the loose coupling of multiple layers of software artifacts, namely the models and the code. Some solutions exist to alleviate the problem; for instance, a model-driven approach whose goal is to produce the model describing the application specification as its primary artifact. In model-driven software evolution, the model is an output that is understandable by every stakeholder and which can be used as a basis for generating source code [Wagner 2014].

Furthermore, reverse engineered artifacts are only representations which entails that, by their nature, they are meant to illustrate some particular aspect while obscuring others which would be evident in some altogether different type of a representation. An an example, the component tree structure is useful for a basic overview of the application, but it obscures implementation details such as the component interfaces that might be necessary to acknowledge for any arbitrary task. For instance, a developer may need to know that some components are interrelated in their purpose for the end user which might translate into the module having complex simultaneous interactions despite them seeming like isolated entities. For that reason, Mencl [2001] places the discovery process (i.e. reverse engineering) as the first step in refactoring and Baxter and Mehlich [2000] emphasize that information derived from the source code is only plan recognition and does not signal anything about the intentions of the developers.

When it comes to recovering design from the overall state of the legacy application, it should be noted that since AngularJS is a framework, application build using it utilize inversion of control. As a consequence, AngularJS necessitates the usage of certain patterns and naming conventions which occasionally makes recognizing patterns straightforward such as in the case of service functions which are all implemented as singletons. Therefore, there is little ambiguity in design of the legacy version of the application when it comes to how the source code is structured. On the contrary, it became a forward engineering issue, and thus a future reverse engineering concern, when alternative solutions

were conjured to replace the dependency injection system. This in an interesting demonstration of the fact that the overarching engineering process undertaken was not a process of driving Wheel to a completed state.

Implementations in the legacy application are described with the implicit notion that the developers have been acting as fully informed and rational agents. In practice, some of the design choices are most likely made in ignorance of the capabilities of the used tools, namely AngularJS. This does not necessarily mean that the developers have not been competent; it is also fair to argue that this is a failing of AngularJS as a tool. The survey conducted by Ramos et al. [2016] discussed in the introduction provides insight into this with its conclusion that directives, alongside concepts related to it, are the most difficult to comprehend despite them being the core concept in AngularJS.

In Wheel, using AngularJS due to its popularity in conjunction with the steep learning curve has left remnants of trial-and-error solutions especially the oldest parts of the source code. These solutions further evolved into patterns when functional solutions were discovered and applied continuously. This type of development has resulted in the code base exhibiting uniform design choices due to copying and pasting existing constructs which, while not necessarily the most optimal or appropriate for the circumstance, have nonetheless fulfilled a business need. An example of this from Wheel is a set of element directives that utilize a shared template despite being separate entities otherwise. As a direct consequence, the template is laced with complicated logic to determine the correct layout depending on the particular needs for specific directives. Another example of this mode of operation are element directives that utilize an isolated scope despite not requiring it, such as loading indicators whose activeness can be toggled via a passed property (the correct pattern is to have the loading state known only in the parent component which either renders the loading indicator or not). Passing such unnecessary properties is signals the lack of understanding of the construction of AngularJS directives. The most likely explanation for their presence is that the structure was copied from elsewhere and applied as-is or with minimal modifications such as renaming functions.

Furthermore, AngularJS enables developers to perform tasks in multiple ways. For example, there are two interchangeable methods of annotating dependencies explicitly [AJS Guide 2018]. Moreover, despite constructor-type services being differentiated from factory-type services (Table 3.1), they can be defined as factories as well. Finally, the regular method of defining element directives is complemented with the AngularJS component constructor that automatically applies characteristics of modular user interface

components such as an isolated scope and the requirement to be used as an element [AJS Guide 2018]. Key characteristic of components is that they use unidirectional data binding and require less setup than a regular element directive. However, in practice, neither is necessarily true as bindings of an AngularJS component (equivalent to isolated scopes) can have two-way bound variables which function exactly as they do in element directives. Moreover, the model of a components is utilized through the keyword this which always refers to the owner object [W3S JS 2019]. As functions are objects as well, the keyword will refer to the functions when used inside them. Therefore, when the model needs to be modified inside a function, it has to be passed as a parameter or referenced through an intermediary variable. The component constructor seems to have been an attempt to streamline the creation of modular user interface elements as competing libraries – such as React – offered more straight-forward solutions.

7 Conclusion

In this thesis, a software migration approach was devised for client-side JavaScript solutions which employ a component-based structure. In the studied project, AngularJS is replaced with React as the primary user interface technology solution although the main migration principles may be followed with any combination of software libraries. For the project specifically, a mapping of concepts between AngularJS and React was constructed that could be utilized when reengineering user interface components. It was demonstrated that Create React App can be used for a React project scaffolding with limited flexibility.

Software migration is a combination of sequential or overlapping reengineering activities that consist of processes of reverse engineering (abstraction), restructuring (alteration) and forward engineering (refinement). Beforehand, a migration plan was defined for the project and the application was equipped with an extensive regression test suite. Then, reengineering was applied for application preparation, build system migration, code reuse and for individual user interface elements.

Application preparation minimized the amount of labor required in upcoming tasks and requires reverse engineering informal knowledge and AngularJS patterns to remove or deprecate existing features as well as eliminate implementations which have already been deprecated or are instances of dead code. The Elim, Elim2 and Temptor tools were constructed for this purpose and resulted in the removal of several files and code lines. The build system migration accommodated the legacy structure to the rules of the new platform as well as another dead-code elimination step assisted by the linter tool introduced by Create React App, resulting in further removal of dead code. The service reuse phase introduced methods of teardown and rewrite to facilitate reusing entire services or functions they provide as-is. Finally, the user interface reengineering step utilized a tree structure of the entire application to recognize individual user interface components in order to perform component reengineering via a bottom-up approach. The principles of The Rule and limbo directives were introduced to aid in the reengineering of components.

Since a migration process is a large-scale change management project, having a sound definition of migration and a plan for executing it are crucial in addition to the aforementioned practical guidelines. This thesis contributes not only to the readily utilizable practices but to defining plans with considerations in mind that are not directly related to software implementation.

References

K. Ahmed. 2018. Learn to become a modern Frontend Developer in 2019. Available (accessed April 7, 2019): *https://medium.com/tech-tajawal/modern-frontend-developer-in-2018-4c2072fa2b9c*.

AJS Guide. 2018. Guide to Angular 1 Documentation, Google LLC, website. Available (accessed April 7, 2019): *https://code.angularjs.org/1.5.8/docs/guide*.

AJS API. 2018. AngularJS API Docs, Google LLC, website. Available (accessed April 7, 2019): *https://code.angularjs.org/1.5.8/docs/api*.

AltexSoft. 2018. The Good and the Bad of Angular Development, AltexSoft, website. Available (accessed April 10, 2019): *https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/*.

Angular. 2019. Angular, Google LLC, website. Available (accessed April 7, 2019): *https://angular.io/*.

Aurelia. 2019. Aurelia, Blue Spire Inc, website. Available (accessed October 23, 2019): https://aurelia.io/.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger. 1960. Report on the algorithmic language ALGOL 60. *Commun. ACM* 3 (5), 299-314.

I. D. Baxter and M. Mehlich. 2000. Reverse engineering is reverse forward engineering. *Science of Computer Programming* 36 (2-3), 131-147.

M. Beck, J. Trümper and J. Döllner. 2011. A visual analysis and design tool for planning software reengineerings. In: *6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 1-8.

T. J. Biggerstaff. 1989. Design recovery for maintenance and reuse. *Computer* 22 (7), 36-49.

BitBucket. 2019. BitBucket, Atlassian Inc, website. Available (accessed October 26, 2019): *https://bitbucket.org/product/*.

Bower. 2019. Bower, website. Available (accessed October 2, 2019): *https://bower.io/*.

BuiltWith Angular. 2019. Angular Usage Statistics. Available (accessed April 10, 2019) *https://trends.builtwith.com/framework/Angular*.

BuiltWith AngularJS. 2019. Angular JS Usage Statistics. Available (accessed April 10, 2019): *https://trends.builtwith.com/javascript/Angular-JS*.

BuiltWith React. 2019. React Usage Statistics. Available (accessed April 10, 2019): *https://trends.builtwith.com/javascript/React*.

A. Butterfield, G. E. Ngondi and A. Kerr. 2016. *A Dictionary of Computer Science*. Oxford University Press, 2016.

G. Canfora, M. Di Penta and L. Cerulo. 2011. Achievements and challenges in software reverse engineering. *Commun. ACM* 54 (4), 142-151.

E. J. Chikofsky and J. H. Cross II. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw*are 7 (1), 13-17.

CRA. 2019. Create React App, Facebook Inc., website. Available (accessed October 20, 2019): *https://create-react-app.dev/docs/getting-started*.

Cypress. 2019. Why Cypress?, Cypress, website. Available (accessed September 27, 2019): *https://docs.cypress.io/guides/overview/why-cypress.html*.

Dart. 2019. Dart, Google LLC, website. Available (accessed October 20, 2019): *https://dart.dev/*.

P. B. Darwin. 2018. Stable AngularJS and Long Term Support. Available (accessed October 22, 2019): *https://blog.angular.io/stable-angularjs-and-long-term-support-7e077635ee9c*.

S. Demeyer, S. Ducasse and O. Nierstrasz. 2003. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers, 2003.

ECMA. 2019. ECMAScript® 2020 Language Specification, ECMA International, website. Available (accessed April 10, 2019): *https://tc39.github.io/ecma262/*.

Elm. 2019. An Introduction to Elm, Elm Software Foundation, website. Available (accessed October 20, 2019): *https://guide.elm-lang.org*.

Enzyme. 2019. Enzyme, AirBnb, website. Available (accessed October 2, 2019): *https://airbnb.io/enzyme/*.

ESLint. 2019. ESLint, JS Foundation, website. Available (accessed November 15, 2019): *https://eslint.org/*.

J. -. Favre, F. Duclos, J. Estublier, R. Sanlaville and J. -. Auffret. 2001. Reverse engineering a large component-based software product. In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, 95-104.

M. Fowler. 2019. *Refactoring – Improving the Design of Existing Code. Second Edition*. Pearson Education (US), 2019.

E. Gamma, R. Helm, R. Johnson and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

G. Garuda. 2017. Migrating from AngularJS to React — how do you measure your performance gains?, website. Available (accessed April 10, 2019): *https://medium.freecodecamp.org/measuring-performance-gains-angularjs-to-react-with-redux-or-mobx-fb221517455*.

Git. 2019. Git, website. Available (accessed October 26, 2019): *https://git-scm.com/docs*.

GitHub. 2019. The State of the Octoverse, GitHub Inc., website. Available (accessed April 10, 2019): *https://octoverse.github.com/*.

Grunt. 2019. Grunt, website. Available (accessed October 2, 2019): *https://gruntjs.com/*.

html-loader-react-scripts. 2019. html-loader-react-scripts, npm Inc., website. Available (accessed October 20, 2019): *https://www.npmjs.com/package/html-loader-react-scripts*.

ISO/IEC 14764. 2006. ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes – Maintenance. In: ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998, 1-58.

I. Jacobson, M. Christerson, P. Jonsson and G. Övergaard. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, Addison–Wesley, 1992.

Jest. 2019. Jest, Facebook Inc., website. Available (accessed October 2, 2019): *https://jestjs.io/*.

S. C. Johnson. 1978. Lint, a C Program Checker. *COMP. SCI. TECH. REP*. Bell Laboratories.

jQuery. 2019. jQuery, The jQuery Foundation, website. Available (accessed October 23, 2019): *https://jquery.com/*.

JSX. 2014. Draft: JSX Specification, Facebook Inc., website. Available (accessed April 14, 2019): *https://facebook.github.io/jsx/*.

Karma. 2019. Karma, Google LLC, website. Available (accessed November 15, 2019): *https://karma-runner.github.io/latest/index.html*.

Less. 2019. Less, website. Available (accessed October 2, 2019): *http://lesscss.org/*.

MDN. 2019. MDN Web Docs Glossary: Definitions of Web-related terms, Mozilla, website. Available (accessed October 25, 2019): *https://developer.mozilla.org/en-US/docs/Glossary*.

A. Memon, A. Nagarajan and Q. Xie. 2005. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice* 17 (1), 27-64.

mnemon1ck. 2015. Why you should not use AngularJS. Available (accessed April 10, 2019): *https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99*.

E. Murphy-Hill and A. P. Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25 (5), 38-44.

Node. 2019. Node.js v11.15.0 Documentation, Node.js Foundation, website. Available (accessed April 10, 2019): *https://nodejs.org/dist/latest-v11.x/docs/api/*.

npm. 2019. About npm, npm Inc., website. Available (accessed October 2, 2019): *https://docs.npmjs.com/about-npm/*.

npx. 2019. npx, npm Inc., website. Available (accessed November 8, 2019): *https://www.npmjs.com/package/npx*.

A. Osmani. 2017. *Learning JavaScript Design Patterns*. O'Reilly Media, 2017.

R. Peréz-Castillo, I. G. d. Guzman, M. Piattini and C. Ebert. 2011. Reengineering technologies. *IEEE Software* 28 (6), 13-17.

Protractor. 2019. Protractor, Google LLC, website. Available (accessed October 2, 2019): *http://www.protractortest.org/#/*.

M. Ramos, M. T. Valente, R. Terra and G. Santos. 2016. AngularJS in the wild: a survey with 460 developers. In: *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016)*, ACM, 9-16.

M. Ramos, M. T. Valente and R. Terra. 2018. AngularJS Performance: A Survey Study. *IEEE Software* 35 (2), 72-79.

React. 2019. React, Facebook Inc., website. Available (accessed April 14, 2019): *https://reactjs.org/docs/getting-started.html*.

react-scripts. 2019. react-scripts, npm Inc., website. Available (accessed October 20, 2019): *https://www.npmjs.com/package/react-scripts*.

Redux. 2019. Getting Started with Redux, website. Available (accessed October 22, 2019): *https://redux.js.org/introduction/getting-started*.

D. Riehle. 2000. *Framework Design A Role Modeling Approach*. Ph. D. dissertation, Swiss Federal Institute of Technology.

RTL. 2019. React Testing Library, website. Available (accessed October 2, 2019): *https://testing-library.com/docs/react-testing-library/intro*.

A. Saboury, P. Musavi, F. Khomh and G. Antoniol. 2017. An empirical study of code smells in JavaScript projects. In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 294-305.

J. Shore. 2006. Dependency Injection Demystified, website. Available (accessed September 9, 2019): *https://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html*.

K. Simpson. 2014. *You Don't Know JS: this & Object Prototypes*. O'Reilly Media, 2014.

SO Trends. 2019. Stack Overflow Trends, Stack Exchange Inc., website. Available (accessed April 8, 2019): *https://insights.stackoverflow.com/trends?tags=jquery%2Cangularjs%2Cangular%2Creactjs*.

S. Tilley. 1998. *A reverse-engineering environment framework*. Carnegie-Mellon University Pittsburgh PA Software Engineering Inst., 1998.

P. Tripathy and S. Naik. 2014. *Software Evolution and Maintenance*. John Wiley & Sons, Incorporated, 2014.

TypeScript. 2019. TypeScript, Microsoft, website. Available (accessed April 10, 2019): *https://www.typescriptlang.org/index.html*.

V8. 2019. What is V8?, website. Available (accessed October 24, 2019): *https://v8.dev/*.

P. Vorbach. 2018. Download statistics for packages react, angular, @angular/cli, website. Available (accessed April 14, 2019): *https://npm-stat.com/charts.html?package=react&package=angular&package=%40angular%2Fcli&from=2015-01-01&to=2019-03-31*.

VTree. 2019. Online JSON to Tree Diagram Converter, website. Available (accessed November 13, 2019): *https://vanya.jp.net/vtree/*.

Vue. 2019. Vue, website. Available (accessed October 23, 2019): *https://vuejs.org/*.

W3S JS. 2019. JavaScript Tutorial, website. Available (accessed April 14, 2019): *https://www.w3schools.com/js/default.asp*.

C. Wagner. 2014. *Model-Driven Software Migration: A Methodology*. Springer, 2014.

Webpack. 2019. Webpack, website. Available (accessed October 2, 2019): *https://webpack.js.org/*guides/.

Webstorm. 2019. Meet Webstorm, JetBrains s.r.o., website. Available (accessed April 7, 2019): *https://www.jetbrains.com/help/webstorm/meet-webstorm.html*.

YO. 2019. What's Yeoman?, website. Available (accessed April 7, 2019): *https://yeoman.io*.

YO AJS Generator. 2019. Yeoman generator for AngularJS, website. Available (accessed November 11, 2019): *https://github.com/yeoman/generator-angular*.

Appendices

Appendix A: Temptor - a shell script for the detection of orphaned templates

The script (Figure A.1) detects orphaned templates by matching each entry in a list of file names with every file that is able to utilize templates. It assumes that the directive and template files of the project reside in their respective singular folders. The script also takes into account a possible router definition file that also contains references to template files.

```
#!/bin/bash

printf "List of orphaned templates\n”
printf "********\n"

# path to folder that contains the directive files
DIRECTIVES=~/path/to/application/directives

# path to file that contains the router definition
ROUTER=~/path/to/application/routerDefinition.js

# temporary file that is a list of template file names;
# ls is an utility for listing files in a directory
# ensure that this does not overwrite an existing templates.txt file
ls ~/path/to/application /views > templates.txt

# read the temporary file line by line
while read t; do
  # grep is a command line tool for pattern matching;
  # the -q option disables printing done by grep
  # and -r searches every file in a directory

  # finds the filename t from all directive files
  if grep -qr $t $DIRECTIVES;
  then
    # empty print indicating that nothing was found
    printf ""
  # finds the filename t from the router definition
  elif grep -qr $t $ROUTER;
  then
    printf ""
  else
    # file name was not found and its name is printed
    printf "$t\n"
  fi
# give the temporary file as an input to the loop
done < templates.txt

# remove the temporary file
rm templates.txt
```

Figure A.1 A shell script used to detect orphaned HTML templates.

Appendix B: Transforming dependency injection into an import pattern

This solution (Figure B.1) allows using custom AngularJS constructs with the import pattern. Only function-oriented services are considered; however, the pattern applies to any injectable construct. First, an index file is created. Second, exported variables are created and named after the services they are used to store. Finally, the AngularJS application module is used to run code that stores dependencies in the injection subsystem to the variables. As a result, the dependencies can be used without utilizing the dependency injection subsystem except for the ones provided by AngularJS such as $http (Figure B.2).

```
services/index.js

/* Assuming an Angular application called "app"
   and two services (aService and bService) exist.
*/

import angular from "angular";

export let aService, bService = undefined;

angular.module("app").run(["$injector", function ($injector) {
  aService = $injector.get("aService");
  bService = $injector.get("bService");
});
```

Figure B.1 A service index file where injectable services are initialized as exports.

```
someController.js

import angular from "angular";
import { aService, bService } from "./path/to/services/index.js";

angular.module("wheel")
  .controller("someController", ["$http", someController]);

function someController($http) {
  aService.aFunction();
  bService.bFunction();
}
```

Figure B.2 Demonstration of using injectables as imports where the dependency injection system is bypassed.