Building maintainable web applications using React
- An evaluation of architectural patterns conducted on Canvas LMS

David Johansson

Supervisor : Jonas Wallgren
Examiner : Peter Fritzson
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© David Johansson
Abstract

Maintainability for web applications is increasingly important due to increasing demands for advanced functionality as well as a short time-to-market. Fixing errors, reusing functionality and adding new features efficiently are crucial for making the application profitable for the software organization as well as valuable for the end-user. Modern frameworks and libraries such as React assist web engineers in building sophisticated applications using high-quality solutions called architectural patterns. In this thesis architectural patterns have been evaluated by performing static code analysis using well-established metrics. The evaluation was conducted using a Design Science Research approach on the Learning Management System Canvas. The results showed large variations in maintainability depending on the architectural pattern used.
Acknowledgments

I would like to thank VISIARC for giving me the opportunity to write this thesis covering an interesting subject, giving me supervision and a place to do the work. Furthermore, I would like to thank my supervisor Jonas Wallgren and examiner Peter Fritzson for providing assistance and valuable guidance throughout the work keeping the thesis on the right path and help to achieve the best results possible. Last, but not least, a big thanks to my family and friends for the endless support during this thesis, my studies at Linköping University and every other challenge I encounter in life.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures viii
List of Tables ix
Listings x

1 Introduction 1
\hspace{0.5em}1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
\hspace{0.5em}1.2 Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
\hspace{0.5em}1.3 Research questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
\hspace{0.5em}1.4 Delimitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Theory 3
\hspace{0.5em}2.1 Canvas LMS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
\hspace{1em}Canvas LMS API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
\hspace{0.5em}2.2 Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
\hspace{1em}Patterns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
\hspace{1em}Quality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
\hspace{1em}Standards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
\hspace{1em}Measurements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
\hspace{1em}Automated static code analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
\hspace{0.5em}2.3 Maintainability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
\hspace{1em}Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
\hspace{1em}Lines Of Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
\hspace{1em}Cyclomatic Complexity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
\hspace{1em}Coupling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
\hspace{1em}Cohesion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
\hspace{1em}Design Complexity of Card and Agresti . . . . . . . . . . . . . . . . . . . . . . . . . 12
\hspace{1em}Software Science Effort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
\hspace{0.5em}2.4 Javascript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
\hspace{1em}ECMAScript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
\hspace{1em}Engines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
\hspace{1em}Node.js . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
\hspace{1em}Node Package Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
\hspace{0.5em}2.5 Web applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
\hspace{1em}Single Page Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
\hspace{1em}Fetching data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.6 Frameworks and Libraries
 React
 Component
 States and props
 Class-based components
 Life-cycle methods
 Function components
 React Hooks
 Virtual DOM
 Context API
 create-react-app
 2.7 Architectural patterns
 Model-View-Controller
 Model-View-ViewModel
 Flux
 Architectural pattern in Redux
 Redux in React
 Architectural pattern in Apollo Client
 Apollo Client in React
 2.8 Design Science Research
 3 Method
 3.1 Study object
 Users administration page architecture
 3.2 Preparation
 Dependencies
 Setting up Plato
 3.3 Initial implementation
 Source code
 Measurements
 3.4 Design Science Research approach
 Iteration 1 - Redux connect
 Awareness of Problem
 Suggestion
 Development
 Evaluation
 Iteration 2 - GraphQL, Apollo Client Cache and React Context API
 Awareness of Problem
 Suggestion
 Development
 Evaluation
 Iteration 3 - GraphQL and Apollo Client Cache without React Context API
 Awareness of Problem
 Suggestion
 Development
 Evaluation
 4 Results
 5 Discussion
# List of Figures

2.1 The control flow graph for a program with two if-else statements ........................................ 10
2.2 An example system with high fan-in and fan-out ................................................................. 12
2.3 Illustration of a graph .............................................................................................................. 19
2.4 NPM downloads for React, Angular and Vue ........................................................................ 20
2.5 Life-cycle methods in React .................................................................................................... 21
2.6 Communication between the React Virtual DOM and the Browser DOM ............................ 23
2.7 NPM downloads for Redux, Apollo Client, MobX and Relay Client .................................... 26
2.8 MVC architectural pattern ....................................................................................................... 27
2.9 Server-side MVC architectural pattern ................................................................................... 28
2.10 Client-side MVC architectural pattern .................................................................................. 28
2.11 MVC architectural pattern issue ........................................................................................... 29
2.12 MVVM architectural pattern .................................................................................................. 29
2.13 Flux architectural pattern ....................................................................................................... 30
2.14 Redux architectural pattern .................................................................................................... 32
2.15 Apollo Client architectural pattern ......................................................................................... 33
2.16 Design Science Research process .......................................................................................... 36
3.1 User administration page in Canvas ........................................................................................ 38
3.2 User administration page architecture ................................................................................... 39
3.3 Plato custom script output ......................................................................................................... 41
3.4 Data flow graph for dispatch functions in initial implementation ........................................... 44
3.5 Data flow graph for static data in initial implementation ......................................................... 45
3.6 Data flow graph for Redux store state data in initial implementation ...................................... 46
3.7 Data flow graph for Iteration 1 .................................................................................................. 51
3.8 Data flow graph for Iteration 2 ................................................................................................. 56
3.9 Data flow graph for Iteration 3 ................................................................................................. 59
List of Tables

2.1 ISO/IEC 25010 characteristics ............................................. 7
2.2 Architectural patterns used in popular frameworks and libraries ............................................. 25

3.1 Canvas API Users parameters .................................................. 39
3.2 Design Complexity measures for initial implementation .................................................. 44
3.3 Number of Operators and Operands for initial implementation ............................................. 47
3.4 Software Science Effort measures for initial implementation ............................................. 47
3.5 Halstead Time and Estimated Bugs for initial implementation ............................................. 47
3.6 LOC and CC measures for initial implementation ............................................. 47
3.7 Design Complexity measures for Iteration 1 ............................................. 50
3.8 Software Science Effort, LOC and CC measures for Iteration 1 ............................................. 52
3.9 Design Complexity measures for Iteration 2 ............................................. 55
3.10 Software Science Effort, LOC and CC measures for Iteration 2 ............................................. 57
3.11 Design Complexity measures for Iteration 3 ............................................. 58
3.12 Software Science effort, LOC and CC measures for Iteration 3 ............................................. 60

4.1 Results of the Design Science Research effort ............................................. 62
Listings

2.1 GraphQL Query type definition ........................................ 18
2.2 GraphQL query ......................................................... 18
2.3 GraphQL query result ................................................. 18
2.4 GraphQL mutation ..................................................... 18
2.5 GraphQL mutation result .............................................. 19
2.6 Class-based React component ........................................ 21
2.7 React function component ........................................... 22
2.8 React useState hook ................................................ 22
2.9 React useEffect hook ................................................ 22
2.10 JSX syntax example .................................................. 23
2.11 Compiled Javascript from JSX ...................................... 23
2.12 Accessing data using useContext hook ................................ 24
2.13 Accessing Redux store using Connect ............................ 32
2.14 Accessing Apollo Client using withApollo ...................... 34
2.15 Accessing Apollo Client using Apollo hook ..................... 34
2.16 Accessing Apollo Client using ApolloConsumer ............... 34
2.17 Querying using Apollo Client Query component ............... 35
3.1 External dependencies for study object specified in package.json 40
3.2 Initial Redux store state for study object ........................ 42
3.3 Providing Redux store to the UsersPane component .......... 49
3.4 Connecting the UsersPane component to the Redux store .... 49
1 Introduction

As web applications have evolved from simple web pages into sophisticated business products, architectural patterns have evolved to provide web engineers with high-quality solutions to recurrent problems. This thesis researches architectural patterns used for building complex web applications and investigates how they affect maintainability. This is done using a research approach named Design Science Research performed on a web application built using the React library which is extracted from the Canvas Learning Management System (LMS).

Maintainability is an important software quality with many sub-characteristics affecting the software organization as well as the end-user. It is part of the ISO/IEC 25010 standard and there exists a variety of metrics to evaluate it. In this thesis, well-established and validated metrics will be used to measure different architectures in regard to maintainability. Some metrics require automated static code analysis and a software program named Plato will be used to perform this on the web application. The essential problem for web applications that architectural patterns try to solve is how different parts within the application should be structured and how they should communicate. Modern web applications often place the architecture of the application on the client-side instead of the server-side and fetch data dynamically from a server-side data API. Depending on the design of these APIs the architectural pattern chosen for the client-side application may change.

1.1 Motivation

VISIARC is a company located in Linköping, Sweden which specializes in developing high-quality software solutions for various customers and industries. As with many software developing companies the environment is high-pace using agile development with short iterations. Maintainability of the software is essential for creating profitable projects satisfying the customer as well as the end-user. For an upcoming project, they will be developing web applications that will be integrated into the Canvas LMS. By investigating maintainability for different architectural patterns and using the results on future projects it will enable better reuse of functionality, easier testing, and faster bug fixes.
1.2 Aim

The aim of this research is to gain knowledge of what architectural patterns exist for web application and how well they perform in terms of maintainability. This will be done by investigating what metrics exist that are relevant for maintainability regarding the architecture of an application followed by quantitative measurements using them. Furthermore, the aim is to gain knowledge of how these measurements can be automated and integrated into the developing process of the application.

1.3 Research questions

1. Which metrics are suitable for measuring maintainability regarding the architecture of web applications, and can they be automated into the developing process?

2. What architectural patterns exist for building web applications using React, and which alternative gives the best effect considering maintainability?

1.4 Delimitations

This thesis will be limited to architectural patterns for React. Even though a majority of the patterns studied are framework and library agnostic they are implemented using React and the results are not guaranteed to be valid for other frameworks and libraries. The decision for using React in this thesis is based on the following:

- The Canvas LMS client-side application is developed primarily using the React library.
- React applications have an architecture which advocates reusability and composition, which align well with the goal of building maintainable software.
- It is the most popular library/framework for developing web application today which will be discussed in Section 2.6.
- Instructure, the company behind Canvas, offers an open-source library of React components that will speed up the development of future applications and make them look more integrated with the Canvas LMS.
In the following sections, the theory necessary to perform the research is covered.

2.1 Canvas LMS

Canvas is a Learning Management System (LMS). Learning Management Systems are part of the e-learning (electronic-learning) ecosystem which, with the help of computers and information delivery using the internet, provides traditional classroom experiences digitally. From the first LMS released in the late 1990s, they have evolved from being used to distribute and manage digital content and for keeping track of courses and test results to systems providing rich learning experiences with a large variety of functionality and integration to third-party software. Canvas LMS makes learning and teaching easier by allowing schools to customize digital learning environments to meet their unique challenges. It simplifies teaching, elevates learning and eliminates the complexity of supporting and growing traditional learning technologies. Canvas LMS is used by 30M+ users and over 4000 universities, schools, and institutions worldwide. Canvas is built with modularity in mind and has great support for integrating third-party applications into the platform. These third-party applications make the platform powerful because it allows institutions to build the platform with all the functionality they need. Canvas LMS is an open-sourced software developed by the company Instructure, Inc. founded in 2008 which is an educational technology company. Canvas LMS targets both K-12 (Kindergarten until 12th grade) education as well as higher education, such as collages.

Canvas announced in 2016 that they would move all the front-end development to React due to maintainability issues using older technologies. Using React with modular components the code became more maintainable and reusable. Instructure open-sourced their reusable React components in a library called instructure-ui which is available on Github. One isolated part of the Canvas LMS will later be used as the study object for this thesis.

1https://github.com/instructure/instructure-ui
Canvas LMS API

Canvas LMS provides a publicly available API that is used by the platform to provide the client-side with data dynamically. This API can be used from any web application using an “access token” for authentication. The API is accessible by adding /api to the end of any Canvas instance URL. For a Canvas instance hosted by Instructure the URL is https://<canvas-instance>.instructure.com/api/. This way the client-side application can communicate with the API. For a web application that does not have the same domain as the API this is prevented by default because of the same-origin policy [31] implemented in XMLHttpRequest and the Fetch API which is used in browsers to get resources over the network. This is an issue when building a third-party application that wants to access the API and was solved in this thesis by communicating with the API using a proxy server. Canvas LMS provides two types of APIs, a REST API, and a GraphQL API. In the documentation of the Canvas LMS API, it is stated that “New Canvas features will be developed primarily in GraphQL and may not be back-ported to the REST API.” [5].

2.2 Software

Software is one of the most important technologies in the world and in the latest decades it has evolved from being specialized problem-solving tools in other industries to the industry itself. Software provides the means for acquiring an important product of our time - information. It has changed the way we find, create, share and uses information. While the demand for business functions using information technology increases competitive pressure is put on every commercial organization. For this reason, software always has to evolve and change to fulfill the demands. Apart from demands from customers software has to change due to errors or to adapt to new environments.

Building software has been compared to traditional building-construction. When building a house one has to decide on what kind of house that should be built, this can be compared to the problem definition in software development. An architect then has to come up with a design for the house which is also necessary for software systems. Planning is an essential part of both building and software construction, without planning the software becomes hard to implement, test and debug. Similar to building construction the success of a project has been determined before starting the actual construction and with more experience with what is being built affects the ability to make good planning decisions. Similar to building advanced buildings like skyscrapers, advanced software has to use rigid solutions and approaches to achieve reliable, high-quality results. These are called patterns.

Patterns

Brad Appleton describes software patterns as follows:

The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users. [2]

In summary, Appleton describes patterns as a tool for software engineers to share knowledge which enables solutions to recurrent problems to be reused and help improve the understanding of good architectures that results in high-quality software that satisfies the end-user.

There are different categories of patterns. Architectural patterns describe problems and solutions on a structural level of the software, as to how to organize modules and how data
2.2. Software

flows through the system. In the context of web applications, there are several architectural patterns commonly used and will be covered in Section 2.7. Component patterns, also known as design patterns, address problems on a component-level of the software system. The book Gang of Four [16] was the first book introducing the concept of design patterns into software, previously used in other professions as for building architects.

Quality

Building high-quality software provides benefits for both the software organization as well as the end-user. The software organization profits because it enables software engineers to spend less time on rework and more time of creating new applications due to less maintenance needed, fewer bugs and less customer support. The end-user benefits by it because the software provides a better experience allowing them to more efficiently perform their task within the software. To achieve this, both the source code to construct the software and the design that models the problem has to be of high-quality.

The model specifies how the software architecture, data structures, and components should be designed to implement a system and is where software quality is established before any code is implemented. The architecture defines the major structural parts of the system: how the parts interact and how data flows. "Software architecture is the high-level part of software design, the frame that holds the more detailed parts of the design" [4]. It highlights decisions in an early stage that can be used for considering the benefits of alternative system structures. Managing the complexity of a system is the main goal for creating high-quality software, at an architectural level the complexity of a system is reduced by dividing it into smaller parts. The more independent the sub-parts are the easier it is to focus on one bit of complexity at a time, this is called modularity and is an indication of high-quality software design. In the book Software Engineering A Practitioner’s Approach, Roger S. Pressman [34] suggests guidelines for design concepts related to software architecture:

A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics […], and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

The first guideline suggests that recognizable architectural styles and patterns should be used. The second guideline is that the system should be composed of components that exhibit good design and characteristics, this is achieved by using design concepts for component-level design described below. The third guideline suggests that the design should be implemented in an evolutionary fashion which means it should be built with high modularity which makes it easy to maintain and test.

After the skeleton, the architecture, of a system is established component-level design should be considered. While the architecture specifies the overall structure of the system the way components are structured and how they interact, components are more detailed parts of a system one step closer to the actual implementation of the system. For high-quality component-design these should strive to be cohesive, meaning they should be as independent as possible performing a single task and requiring as little interaction with other components as possible. This follows a design concept called Separations of Concern suggested by Dijkstra [11]. A concern is a feature or behavior of the system that is required for the system to work, by separating these it takes less time and effort to maintain and complexity is reduced. By following this design concept the system is broken down to modular pieces and the modularity of the system is improving. Myers [32] states that “modularity is the single attribute of software that allows a program to be intellectually manageable”. The interconnection among modules is called coupling. High cohesion often leads to low coupling which is an indication of a high-quality software-design. As the amount of communication increases between modules the complexity of the system increases and the difficulty of implementing, maintaining
and testing it becomes more difficult. Communication between modules must occur for a system to work and therefore some degree of coupling is necessary, but it should be as low as possible.

Goals for the business are almost always related to specific goals at the software engineering level, especially if the business is the software product itself. Both the design models and the quality of the source code can be measured in an objective way rather than in a subjective way which is important to understand how to improve the software. Because the definition of quality differs depending on the system being constructed the software quality has to be explicitly defined from case to case. Some quality characteristics contradict others and tradeoffs have to be made from competing characteristics to achieve the business goals.

Software qualities can be classified as external or internal. External qualities are qualities that the user is aware of, for example, if the software is easy to use and if it works as expected. Internal qualities, on the other hand, are what the software engineers are concerned about, as the ease of changing the software, adding functionality, correcting errors or improving performance. Internal qualities often affect external qualities, software that is not internally understandable or maintainable will, in turn, lead to unreliable software which is hard to correct defects for. Furthermore, software that is not built for adapting to change will affect the usability negatively because user requests will require a lot of work and therefore take a long time to implement. This thesis is code-centric and therefore the internal quality characteristics will be studied while external qualities will be affected implicitly.

Standards

In an attempt to identify the key quality characteristics for software a standard was developed named ISO/IEC9126 [24] containing six attributes and these were: Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability. The ISO/IEC 9126 quality attributes provide indirect measures for assessing the quality of a system. ISO/IEC 9126 is a tool to develop a common understanding of the system’s objectives and priorities. ISO/IEC 9126 was 2011 replaced by a modern standard named ISO/IEC 25010 which has eight product quality characteristics and 31 sub-characteristics which was derived from the ISO/IEC 9126 standard. The goal of the standard, as with ISO/IEC 9126, was to “identify relevant quality characteristics that can be further used to establish requirements, their criteria for satisfaction and the corresponding measures.” [23]. ISO/IEC 25010 consists of the following eight characteristics: Functional Suitability, Reliability, Performance Efficiency, Usability, Security, Compatibility, Maintainability, and Portability. The sub-characteristics for these are shown in table 2.1 on page 7.
| Functional suitability | Functional completeness  
|                       | Functional correctness  
|                       | Functional appropriateness  
| Performance efficiency | Time behaviour  
|                       | Resource utilization  
|                       | Capacity  
| Compatibility | Co-existence  
|                   | Interoperability  
| Usability | Appropriateness recognizability  
|             | Learnability  
|             | Operability  
|             | User error protection  
|             | User interface aesthetics  
|             | Accessibility  
| Reliability | Maturity  
|             | Availability  
|             | Fault tolerance  
|             | Recoverability  
| Security | Confidentiality  
|            | Integrity  
|            | Non-repudiation  
|            | Accountability  
|            | Non-repudiation  
| Maintainability | Modularity  
|              | Reusability  
|              | Analysability  
|              | Modifiability  
|              | Testability  
| Portability | Adaptability  
|            | Installability  
|            | Replaceability  

Table 2.1: ISO/IEC 25010 characteristics
Measurements

To be able to measure the quality of a software system quantitative measurements of software quality are needed that will enable objective analysis. By using quantitative measurements quality control can be done for the quality characteristics that are important to the system. It is used to control the software development process and to analyze how the overall quality of the system is impacted by changes made. A metric is defined by the IEEE Standard Glossary of Software Engineering Terminology [21]: “a quantitative measure of the degree to which a system, component, or process possesses a given attribute.” An indicator is a metric or a collection of metrics that provide useful information on how to improve a software system for the better. Depending on which quality attributes that are considered important for the system metrics are derived and measurements are made on the system. The results of the measurements may be analyzed based on pre-established goals and/or past data. Furthermore, insight gained from analyzing the data can be used to improve the design model and the source code.

Ejigru [14] proposes a few attributes that characterize effective software metrics. He suggests that a metric and the measurement acquired from it should be:

- Simple and computable: The metric should be relatively easy to learn and use and the computation should not demand too much effort.
- Empirically and intuitively persuasive: The metric should satisfy the intuitive notions of how the metric affects a certain quality attribute, meaning a metric for measuring coupling shows an increase when more modules interconnect.
- Consistent and objective: The metric should result in the same value when conducted by an independent third party using the same information about the software.
- Consistent in its use of units and dimensions: The computation of the metric should not mix units that are not intuitively related.
- Programming language-independent: The metric should not be dependent on syntax or semantics of the programming that is analyzed.
- An effective mechanism for high-quality feedback: By analyzing the results of the measurement the metric should provide information that can lead to higher-quality of the software.

Measurements are used to establish high-quality products by using the resultant metrics to improve it. “If you do not measure, there no real way of determining whether you are improving. And if you are not improving, you are lost” [34].

Automated static code analysis  For large codebases, it may be impossible to calculate some code metrics manually. Many metrics are defined using mathematical equations, therefore these calculations can be automated by a static code analysis software. By automating the analysis process and setting explicit code metric boundaries problematic code can be detected by adding the analysis into the build process which reports on its quality and the system can be reviewed and redesigned.

Plato is an analysis tool for automating the calculation of source code metrics for Javascript. It generates reports of the results and visualizes the results. Plato calculates complexity data using the escomplex library. Escomplex is a complexity analysis tool for abstract syntax trees (ASTs) for Javascript which supports modern Javascript as well as the syntax extension JSX used in React that will be covered in Section 2.6. The AST for the code is generated using a Javascript parser and once the full AST is obtained static code analysis is performed to produce complexity data.

https://github.com/es-analysis/plato
2.3 Maintainability

Maintainability is a software quality characteristic included in the ISO/IEC 25010 standard. Maintainability includes five sub-characteristics as mentioned in Section 2.2, and these are Modularity, Reusability, Analysability, Modifiability, and Testability. The description of these sub-characteristics from the ISO/IEC25010 documentation is the following:

- **Modularity**: the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Reusability**: the degree to which an asset can be used in more than one system, or in building other assets.
- **Analysability**: the degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability**: the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability**: the degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

In summary, maintainability accounts for the ease with which software can adapt to new environments, be corrected if an error occurs and be improved due to changes in the requirements, environment or functional specifications.

Measures of maintenance activities such as the time required to make a change or fix an error in a system can be done after a system has been developed. To be able to evaluate maintainability during development models for maintainability exist which predicts the maintainability of a system using measures including complexity, size, cohesion, and coupling. Reducing complexity is the primary goal for improving the maintainability of a system. By dividing software into smaller manageable pieces it becomes easier to focus on one piece of complexity at a time and becomes easier to correct, adapt and enhance the software.

**Metrics**

In the paper “Software Metrics for Predicting Maintainability” published by Marc Frappier and his colleagues a set of metrics for measuring maintainability was evaluated based on empirical studies. The metrics were evaluated according to criteria derived from two studies in the field of software engineering: Characteristics of Software Quality and Software Engineering Metrics and Models. These studies conclude that metrics should be evaluated based on if they are “Objectivity and Algorithmic” measurements. Algorithmic means that a metric should be computed precisely according to an algorithm and should not change due to changes in time, place or observer. The metric should be objective because of subjective metrics prone to produce inconsistent results. Furthermore, a metric should be validated and therefore experimental results published by researchers were investigated in the paper for correlation between the metric and indicators of maintainability. Nineteen metrics were reviewed whereby fourteen of these were recommended for implementation. Out of these fourteen recommendations, six of them have been studied more closely. These were Lines Of Code, Cyclomatic Complexity, Module Coupling, Module Strength, Design Complexity of Card and Agresti and Software Science Effort. Some of these metrics were calculated manually while others were calculated using static code analyzing tools.
2.3. Maintainability

Lines Of Code

The number of Lines Of Code (LOC) in a program, excluding blank lines and comment lines, is a measure of the size of the system. A decrease of LOC is an indication of increased maintainability. The computation of LOC can be done automatically using static code analyzing tools. LOC is a straightforward measure of the size of a system and a high value is almost always an indicator of a system that is hard to maintain [15].

Cyclomatic Complexity

Cyclomatic Complexity is one of the most well-known metrics to determine the structural complexity of a program control flow. Cyclomatic Complexity is computed on a flow graph which is a representation of the program containing nodes and edges. Nodes are program statements and edges represent the control flow between statements which are determined by conditional statements. The formula for calculating the Cyclomatic Complexity is shown in equation (2.1), where $E$ is the number of edges, $N$ is the number of nodes and $\rho$ is the number of exit points in the program.

$$M = E - N + 2 \times \rho$$  \hspace{1cm} (2.1)

A program with two if-else statements results in the control flow graph shown in Figure 2.1. The Cyclomatic Complexity measure for this program results in equation (2.2).

$$M = 8 - 7 + 2 \times 1 = 3$$  \hspace{1cm} (2.2)

![Figure 2.1: The control flow graph for a program with two if-else statements](image)

The Cyclomatic Complexity metric was developed by Thomas McCabe in 1976 [27] who also described a number of important uses for complexity metrics:

Complexity metrics can be used to predict critical information about the reliability and maintainability of software systems from automatic analysis of source code. Complexity metrics also provide feedback during the software project to help control the design activity. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability [28].
2.3. Maintainability

Cyclomatic Complexity has a good correlation with maintainability and is extensively validated. When the Cyclomatic Complexity decreases maintainability of the software increases \[15\]. In the paper by Marc Frappier and his colleagues, a few studies indicated that programs with a higher value for Cyclomatic Complexity actually had fewer errors, this was explained with that larger modules often were coded using greater care than smaller modules.

Coupling

Coupling is a measure of the strength of the relationship between modules in a system, the degree of which modules depend or rely on other modules in the system. Coupling can occur directly to other modules, to global data or the outside environment. A module that takes a lower number of parameters and has fewer public methods are more “loosely coupled” than one of the opposite. “Tightly coupled” systems often have some disadvantages. A change in one module usually means that connected modules also have to change which makes it harder to reuse and test. Apart from the size, modules should have flexible relations to other modules by being modular. The term coupling was introduced in an article by Stevens, Myers and Constantine \[37\] and in a book by Yourdon and Constantine \[40\], and has become a standard term in software engineering.

Module coupling is one measure for coupling that was introduced by G.J. Myers \[32\] and is computed manually using a checklist. Myers specifies six different types of interconnections between modules which increase coupling:

- Content coupling: A module makes direct reference to another module using its internal data. This violates hiding information to other modules.
- Common coupling: Several modules have access to the same global data which can lead to uncontrolled side-effects when changes are made.
- External: Several modules share a common data item which is data of external tools and devices.
- Control: A control element is passed to a module which is controlling the flow of another module by passing information on what to do.
- Stamp: A data structure is passed to a module where only parts of it, possibly different parts, are used.
- Data: Modules share data through parameters that are of primitive type.

This metric has no correlation data available and therefore the validity has not been proved but has gained wide acceptance in the software engineering community \[15\].

Cohesion

Cohesion is a measure of how strongly related elements are within a module, also known as the cohesion of a module. Modules having strong cohesion have advantages on reusability and understandability which leads to better maintainability. Furthermore, changes to a module affect fewer other modules and it becomes easier to correct an error among cohesive operations. Cohesion is often correlated with coupling, strongly cohesive modules often result in loosely coupled modules. The term cohesion was also introduced in the article by Stevens, Myers and Constantine \[37\] and the book by Yourdon and Constantine \[40\]. Module Strength is measured using a checklist suggested by Myers \[32\] which consists of seven types of cohesion within a module:

- Coincidental cohesion: There is no meaningful relationship between operations in the module.
2.3. Maintainability

- Logical cohesion: Modules are grouped because they do similar things even though they operate differently, for example, a module handling both input and output.

- Classical cohesion: Modules are grouped similarly to logical cohesion but operations are grouped depending on the time in the program it is executed, for example, a module handling initialization that handles multiple different operations.

- Procedural cohesion: Modules are grouped because they follow a certain sequence of execution, for example, a module checking if a user is authorized then shows the user some information.

- Communicational cohesion: Modules are grouped because they use the same data.

- Informational cohesion: Connecting multiple modules acting on a common database.

- Functional cohesion: Operations of a module are grouped because they are carrying out a single well-defined task.

Design Complexity of Card and Agresti

Design complexity by Card and Agresti is a composed measurement of the complexity of procedures in modules and the connection between them in a system in terms of procedure calls, parameters passing and data use. The measure is based on the number of fan-outs and input/output variables from each module in the system as well as the number of modules in total [15]. This results in a metric measuring the coupling and cohesion of a system. The measurement was defined by D. N. Card, W.W. Agresti [7] and can be used to evaluate the difficulty of building a designed system before the implementation or to evaluate the complexity and maintainability of an implemented system. It allows complexity evaluation among different systems and sizes because it is a normalized measure calculating the average complexity of the system.

Fan-in and fan-out are structural metrics to measure complexity between modules in a system. Fan-in is the number of modules that call a module and fan-out is the number of modules called by the module. It provides insight into how complex the interlinking of modules is. A high value of Fan-out indicates a strongly coupled module while a low or zero value for fan-out indicates an independent, self-sufficient module that is easily maintainable. A high value of fan-in is an indication of a module that is extensively used by others meaning the system is using reusability efficiently [29].

Figure 2.2 illustrates a system where the modules are highly interconnected with high values for both fan-in and fan-out. This results in a complex system that can break or need modification when one of the modules is changed which makes it hard to maintain.

![Figure 2.2: An example system with high fan-in and fan-out](image_url)
2.3. Maintainability

Design complexity consists out of two parts: structural complexity ($S$), which is the external complexity between modules, and local complexity ($L$), which is the internal complexity of modules. It is calculated using equation (2.3)

$$DC = S + L$$  \hspace{1cm} (2.3)

$S$ is calculated using equation (2.4) where $f_m$ is the fan-out measure of module $m$ and $n$ the total number of modules in the system. $S$ is calculated using fan-out squared which leads to that a module calling many other modules in a system has high structural complexity.

$$S = \left( \sum_{m=1}^{n} f_m^2 \right)/n$$  \hspace{1cm} (2.4)

$L$ is calculated using equation (2.5) where $v_m$ is the number of input/output variables in module $m$ and $f_m$ and $n$ are the same as before. The more data the module has to handle based on the input and output parameters the higher local complexity it has, but the more modules connected to it (fan-out) the more likely it is to have delegated operations to other modules that are reflected in the equation. Minimizing $L$ may result in fewer operations in the module but more calls between the modules which in turn increases $S$. Each input and output variable is counted once, regardless if it used or not in the module.

$$L = \left( \sum_{m=1}^{n} v_m/(f_m + 1) \right)/n$$  \hspace{1cm} (2.5)

By distributing the number of other modules called from within modules (fan-out) evenly the structural complexity can be decreased. To minimize the local complexity, design decisions have to be made considering if data should be passed as parameters to modules or if the use of non-local variables should be used (for example global variables). Parameters should not be passed through modules without using their values and if the data is needed in several places, storing the data in a non-local variable is preferred.

Validation of the measure was done by the authors Card and Agresti [7] which involved seven large software systems containing an average of 265 modules and 64 000 lines of code and showing good correlation to the complexity and thus maintainability.

Software Science Effort

Software Science Effort is a measure published by M.H. Halstead [20] and is an estimation of programming effort based on the number of operators and operands in a system. The metric is well studied and shows a good correlation with that maintainability decreases as the Software Science Effort decreases. Halstead assigned primitive measures that could be derived from the source code to analyze the overall program length and volume, development effort, development time and projected number of bugs in the software. These measures are:

$$n_1 = \text{number of distinct operators in a program (+, =, IF, WHILE)}$$

$$n_2 = \text{number of distinct operands in a program (variables, constants)}$$

$$N_1 = \text{total number of operator occurrences}$$

$$N_2 = \text{total number of operand occurrences}$$

Software Science Effort is calculated using Halstead difficulty ($D$) and Halstead volume ($V$) which are based on program length ($N$) and program vocabulary ($n$) according to the equations shown in the equations (2.6 to 2.10)

$$E = V \ast D$$  \hspace{1cm} (2.6)
\[ D = \frac{n_1}{2} \times \frac{N_2}{n_2} \]  
\[ V = N \times \log_2(n) \]  
\[ N = N_1 + N_2 \]  
\[ n = n_1 + n_2 \]

Halstead effort can also be used to estimate the time it requires to implement the program \((T)\) according to equation \(2.11\) as well as predicting the number of bugs \((B)\) in the implementation according to equation \(2.12\).

\[ T = E/18 \]  
\[ B = E^{2/3}/3000 \]

### 2.4 Javascript

Javascript is a high-level programming language, meaning that it has strong abstraction from the details of the computer. It was initially used to write small scripts in the browser to add logic to web pages. Nowadays Javascript is used to build sophisticated web applications, both client- and server-side as well as other software applications such as desktop applications using Electron\(^3\) and mobile applications using React-Native\(^4\).

Javascript is a scripting language, which is a programming language designed specifically for acting on an existing entity or system and does not need an explicit compilation step. When using Javascript in the browser it interacts with the Document Object Model (DOM). Javascript relies on the host environment to provide functionality as networking, storage, and rendering graphics.

Javascript is a multi-paradigm programming language that supports event-driven, functional and imperative programming styles \(30\). Javascript has first-class functions that are necessary for functional programming because it supports passing functions as arguments to other functions, assigning them to variables and storing them in data structures \(1\). Higher-order functions are an essential part of functional programming which in mathematics and computer science is a function that follows two rules: It takes one or more functions as arguments, and it returns a function as its result. Javascript has several higher-order functions built into the language.

**ECMAScript**

Javascript is following the ECMAScript standard created by the Ecma International organization \(13\). The first specification of ECMAScript was released 1997 and several updated versions of the specifications have been published since then. Javascript has remained the far most popular implementation of ECMAScript since its foundation. Apart from being compatible with ECMAScript Javascript provide additional functionality. In June 2015, an extensive new specification was published named ECMAScript2015 (also called ES6) \(12\). The specification included two major updates for building more complex programs including syntactic sugar for class declarations and module sharing. Javascript is prototype-based object-oriented but with ES6 an alternative and more familiar way to create objects and deal with inheritance were added, the Javascript class\(5\).

---

3https://electronjs.org/
4https://facebook.github.io/react-native/

---

14
2.5 Web applications

ECMAScript2015 also included importing and exporting Javascript modules between files. This update made it possible to share modules between Javascript files which made reusability much simpler.

Engines

A Javascript Engine is responsible for interpreting Javascript. Different engines are used depending on the host environment and every major browser has its own. One of the most used Javascript Engine is V8 developed by Google. This engine is used by Chrome and many other Chromium-based browsers. Apart from being used in browsers, V8 is used in the popular library Node.js.

Node.js

Node.js, also known as Node, is a simple and effective runtime environment for running Javascript outside of the browser using the V8 engine. Node is often used as a server-side runtime but is also used client-side for building Javascript web applications. Node includes the Node Package Manager (NPM), which is used to set up a Node application and to manage dependencies in the application.

Node Package Manager  The Node Package Manager is set up in the project directory using a configuration file called package.json. By using NPM developers have access to an extensive library of open-source third-party packages. The package name is added to the package.json configuration file and the source code for the package is downloaded from the NPM library and installed locally in a folder called node_modules. The field ”scripts” in package.json is used for specifying certain operations usually it contains a start script, a build script and a test script. NPM is often used when developing React application which will be covered in Section 2.6.

2.5 Web applications

A web application, or “WebApp” for short, is a network-centric software which in its simplest form can be a static web page displaying information using text and limited graphics. This was how the majority of web applications were built in the early days of the World Wide Web. Web applications started to utilize programming languages such as Java which lead to web engineers having the capability to add computation along with informational content which made web applications evolve and having more than just presentational abilities.

When a user types in a URL in the browser window it is translated into a unique IP-address that belongs to a web server that serves the web page the user wants to access. Traditionally web applications had no client-side architecture, the browsers fetched static files from a web server and rendered it to the user. As technology evolved web browsers became more powerful which led to that more functionality was moved to the client-side.

WebApps have unique features and requirements. Powell describes web-based systems as a “mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology”. Building WebApps requires both technical and non-technical activities which include designing the application’s aesthetic layout, designing the user experience, defining the architectural structure and developing the functionality and content. According to Pressman what differs from conventional software applications is that WebApps often have faster iterations between releases, they evolve continuously specifically their content.

---
7https://www.npmjs.com/
and often have a short time-to-market which can be a matter of days or weeks. These characteristics of WebApps put a lot of pressure on high maintainability which leads to that changes can be made easier and errors can be corrected quickly.

Historically to provide rich and interactive experiences in web applications, client-side technologies as Adobe Flash[^8] and Microsoft Silverlight[^9] was extensively used. In recent years these solutions have faded away in interest and many web browsers do not support these any longer. Instead, Javascript-based solutions have evolved to make web pages interactive, one of the earliest and most popular libraries was jQuery which enabled developers to manipulate the DOM using Javascript to make the web page interactive. jQuery was cross-platform meaning that web browsers with different compilers and rendering engines still ended up rendering the same content which was beneficial. In recent years a paradigm shift has occurred, instead of injecting Javascript into HTML-pages as with jQuery, the application is built using Javascript which in turn generates HTML-pages and CSS. The final product that the web browser will interpret is always the same: HTML files that specify the structure of the elements, style sheets (CSS) that decides how the elements should look and finally plain Javascript files making the application interactive.

Frameworks and libraries have become essential parts of web engineering due to the often short time-to-market and the demand for complex functionality and high standards. Frameworks and libraries reduce the code the developers have to write by providing high-quality solutions to well-known problems.

**Single Page Application**

Traditionally, web applications consisted of static HTML-pages linked together by hyper-links. When clicking a hyper-link the browser fetched a new HTML page from the web-server and rendered it which made the application predictable. A downside to this approach was that rendering new HTML pages affects the user experience negatively and led to bad performance because it required a complete re-render for each change that was made. This is referred to as a “thick server architectures” where the client sends requests based on user requests and the server responds with HTML or Javascript that is necessary to bring the browser to the new state.

For Single-page applications (SPAs) only one HTML file exists and parts of the page are dynamically re-rendered using Javascript. To keep navigability between logical pages the HTML5 History API is used, it makes it possible to change the URL in the browser dynamically and enable the user to navigate using the controls in the browser as with traditional web applications. To update content, data is fetched dynamically from either the web-server or a third-party API. The data received can be either raw-data or pre-rendered HTML which is dynamically added to the DOM. This is referred to as a “thin server architecture”.

When initially loading a SPA the minified code for the framework is loaded, therefore the initial loading time is often a bit longer than for traditional web pages. By using caching in the browser and loading parts of the application when it is needed, called lazy-loading, the loading time of SPAs can be decreased. Even though the initial loading takes a bit longer subsequent content is loaded faster and the user experience will be improved.

**Fetching data**

For permanent data storage, data is often stored in a database on the server. For the client-side application to receive and update data from this database an API (Application Programming Interface) is used. The communication between the client and the API is done through HTTP requests.

[^9]: https://www.microsoft.com/silverlight/
An API can be designed in several ways, the two most commonly used are the Representational State Transfer (RESTful) API and a newer type of API called GraphQL API. Canvas LMS provides a REST API and a GraphQL API and all available API requests can be found in the Canvas LMS API documentation \[5\].

### REST API

The lack of detailed specifications for REST APIs has led to different approaches on how to design them. The core structure for REST APIs is the same: When using data from a REST API the request method in the HTTP-request describes what kind of action should be performed on the data and the URL decides which data the operation should be applied on. The URL matches an endpoint defined for the API containing route handlers where the API do the computation, get data from a database or call other APIs. The route handler returns data, often in JSON \[10\] format.

### GraphQL API

GraphQL was initially released in 2015 by Facebook and has become an alternative to REST APIs \[19\]. GraphQL is a query language for building and using GraphQL APIs. The client can query the exact data it wants to receive in contrast to REST APIs where each endpoint returns a fixed chunk of data. GraphQL APIs only contain a single endpoint where all data can be consumed from. With this approach, new queries can be made from the client without changing or creating new endpoints on the server-side. The communication between the client and the API are made through HTTP requests as with REST APIs.

When using data from a third-party API the developer has no control of the structure of the server-side endpoints. In REST APIs the application has to make many round trips to collect all the data needed resulting in a poor fit between the API and the application. The application also needs to have knowledge of the APIs responses for different endpoints to handle them correctly. These problems occur because REST-APIs are inflexible, GraphQL solves these problems by providing a flexible API where the client gets more control over the data. The drawbacks for GraphQL APIs is that it is more complex to set up and because it is not as widely understood as REST and has not existed as long there are less well-tested libraries and support to get.

To get the benefits of GraphQL when using data from a third-party that does not provide a GraphQL API or is in an early state not providing all the data needed the third-party API can be wrapped into a GraphQL API. There are a few tools available to simplify this process, one of these is Apollo Server \[11\] which has been used in this thesis to account for missing data provided by the Canvas LMS GraphQL API. The implementation details for the Apollo Server will not be covered in this thesis because it is not relevant to the research questions.

A description of what operations are available in the API is called a schema. It is a blueprint for the API of the types and their relationships. The schema is strongly typed using GraphQL schema definition language (SDL). Before the queries are executed by the GraphQL API it is validated against the schema. GraphQL APIs are structured using types and fields instead of endpoints. The most basic piece of a GraphQL API is a type that describes an object that can be fetched by the API. There are built-in scalar types, these primitive types are ID, String, Int, Float, and Boolean.

There is a special type named Query, this type describes which queries are available in API and what each query will return. An example of a Query type is shown in Listing 2.1.

---

\[10\]https://www.json.org/

\[11\]https://www.apollographql.com/docs/apollo-server/
2.5. Web applications

```graphql
type person {
  name: String
  age: Int
}
type Query {
  persons: [person]
}
```

Listing 2.1: GraphQL Query type definition

Square brackets indicate that it returns an array of a certain type. Another type is the Mutation type that defines which mutations, i.e., updates of data, that is available in the API. Each Query and Mutation has its own resolver where the logic for doing calculations, communicating with the database or calling other APIs is specified. The resolver returns data according to the schema specification. The simplest request is a query asking the API for a field inside an object, shown in Listing 2.2. A query is defined using the “query” keyword and the first pair of curly braces is called the root object, on the root object the field “person” is selected, for each person object the field “name” and “age” is selected.

```graphql
query GetPersonsQuery{
  persons {
    name,
    age
  }
}
```

Listing 2.2: GraphQL query

The result of the query is shown in Listing 2.3.

```json
{
  "data": {
    "persons": [{
      "name": "John",
      "age": "32"
    }]
  }
}
```

Listing 2.3: GraphQL query result

To update data a mutation is used. Mutations have the same structure as a query and modifying data could, in fact, be executed using a query, but by convention just like POST and GET in REST-apis these are distinguished. Variables in the mutation are used to tell the API which objects and fields that should be modified. A simple mutation is shown in Listing 2.4.

```graphql
mutation UpdatePersonMutation($name:String, $age:Int){
  updatePerson(name: $name, age: $age) {
    name,
    age
  }
}
```

Listing 2.4: GraphQL mutation

As with queries, the mutation returns data which is useful for updating the client-side application to a new state after an update, this is shown in Listing 2.5.
Apart from a query and a mutation there exist a third type of operation called a subscription. This operation is similar to queries but instead, it listens to changes in the data provided by the API and gets notified when changed.

The GraphQL Canvas LMS API [6] is designed according to a schema specification created by Facebook called Relay. It provides design guidelines for creating future-proof GraphQL APIs.

**Relay schema specification**  Relay uses terminology from graph theory. Figure 2.3 shows a graph which is a high-level concept of how to think about the data in an application. A graph contains Nodes which are the circles in the graph, the lines connecting Nodes are called Edges which represents relationships between the Nodes. When querying multiple Nodes connected to each other for large graphs it is not possible to query all results at the same time and therefore some kind of pagination is needed. Pagination enables requesting parts of the graph and keeps track of the start and end of the graph. Using a pagination method named cursor-based pagination this is done using Connections. The Relay schema specification contains rules for defining the Nodes, Edges, and Connections for the data in the graph.

![Illustration of a graph](image)

**Figure 2.3: Illustration of a graph**

### 2.6 Frameworks and Libraries

Frameworks and libraries essentially have the same purpose which is to provide solutions that speed up development. A framework is often more complex than a library and the architecture of the application is predefined. The developer only has to be concerned with implementing domain-specific functionality. The disadvantages of using a framework are the strict rules and patterns that have to be followed which leads to a steep learning curve and also impacts the size of the application negatively. Libraries on the other hand offer functionality that developers can use when needed and lets the developer control the architecture themselves. This results in an application that contains only as much functionality as
needed. A disadvantage of using libraries is that it can lead to less maintainable software if the application is not designed correctly.

There are plenty of libraries and frameworks available to assist the developer when developing complex web applications. At the time being the most popular frameworks and libraries are React, Angular and Vue. According to the number of downloads using NPM, React is the most popular, Angular the second most popular and Vue the third most popular, shown in Figure 2.4. The comparison was made on the “react-dom” package instead of “react” because “react” includes development for mobile devices as well. For Angular, the package @angular/core was used in the comparison which is the package used for all Angular applications.

![Figure 2.4: NPM downloads for React, Angular and Vue](image)

**React**

React is a Javascript library for building user interfaces [35], it can be used in isolated parts or for creating user interfaces for whole web applications. React does not enforce any specific architectural pattern for managing data, it focuses solely on the creation of Views. React utilizes composition to build complex user interfaces from small building blocks called Components.

**Component**

Each Component consists of a render method that returns a description of what to render and can either be HTML or other React components. The description of what to render uses a syntax called JSX which is a combination of HTML and Javascript. A Component can be defined either as a class or as a function. State and props are essential parts for building React applications.

**States and props** A Component’s state lets it “remember” things, the state of a Component may change based on user interaction or other actions within the application. The State is not mandatory, Components having a state is called stateful components while components not having a state is called presentational components.
Props, or properties, are immutable data passed into a component upon construction. Props make React components flexible and reusable because one component can behave and look different depending on the props passed into it.

In React, data flows downwards through the component tree using props. For a child component to interact with its parent callback functions are passed as props. In large React applications the component tree becomes deep which results in having to pass down callback functions and other data multiple levels. This is referred to as “props-drilling” and makes the components tightly coupled and makes the application less maintainable. This is one reason why architectural patterns are necessary for complex React applications.

Class-based components A class-based Component is created by extending the React.Component class as shown in Listing 2.6. The state of a class-based component is updated using the method setState() and read by using this.state within the class. Using the setState() method makes the component re-render which is not the case when mutating the state directly. Class-based components include life-cycle methods that can be used to create more complex behavior.

```javascript
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

Listing 2.6: Class-based React component

Life-cycle methods Life-cycle methods are built-in methods called when a state or prop updates, before or after a component has rendered or when a component is destroyed. An overview of the life-cycle methods of a class-based component is shown in Figure 2.5.

![Figure 2.5: Life-cycle methods in React](image)
2.6. Frameworks and Libraries

**Function components**  A function component is a pure function that accepts props as input and returns a JSX element. A simple function component is shown in Listing 2.7. To provide the function component with the possibility of using state and life-cycle methods as for class-based components React Hooks are used. Using function components with hooks can reduce the size and complexity of a React application in comparison to class-based components.

```javascript
const Greeting = (props) => {
  return <h1>Hello, {props.name}</h1>
}
```

Listing 2.7: React function component

**React Hooks**

React Hooks are used for more advanced function components, they “hook into” React features. React hooks follow a naming convention starting with the word “use”. State does not exist in a function component by default, instead, a React hook named useState can be used which preserves the state throughout the existence of the components. The useState hook, and all other hooks, can be used more than once inside the same component. A function component using the useState hook is shown in Listing 2.8.

```javascript
const Counter = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Listing 2.8: React useState hook

Built-in life-cycles methods do not exist for function components but can be created using the useEffect hook. The useEffect hook will execute a function for every re-render of the component by default but can be customized to only execute for certain changes. An example of the useEffect hook is shown in Listing 2.9.

```javascript
const Counter = () => {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = 'Count: ' + count;
  });
  ...
}
```

Listing 2.9: React useEffect hook

Another React hook is the useContext hook which will be covered in the Context API section. Apart from built-in hooks, custom hooks can be created which enables reusability of functionality between components.

**Virtual DOM**

React creates a new View as a function of immutable states and props for all components in the application, a change in either a state or a prop re-renders the View. This makes the Views predictable and testable. Re-rendering Views by replacing the DOM with a new version has a negative impact on the user experience because it is a time-consuming operation that loses input information and scrolling position. This is solved in React by using a Virtual DOM.
The Virtual DOM re-renders Views that need to change in an inexpensive way without affecting other DOM nodes by creating a new Virtual DOM subtree when data changes in the application and comparing it to the previous one. React computes the minimum set of DOM mutations necessary to make the real DOM look like the virtual DOM according to Figure 2.6. Manipulations of the DOM are made using Javascript and are put in a queue and are batch executed to save time. Developers use JSX to decide how components should render in different states and React takes care of manipulating the DOM to get to the different states.

![Figure 2.6: Communication between the React Virtual DOM and the Browser DOM](image)

The React library includes a compiler that compiles the declarative JSX syntax into Javascript that the browser engine can interpret. Listing 2.10 shows the JSX code before compiling and Listing 2.11 shows the code executed by the browser. The virtual DOM does not rely on the browser and can therefore be used in server-side rendering as well.

```
1 <div>
2   <Greeting name="John" />
3 </div>
```

Listing 2.10: JSX syntax example

```
1 React.createElement("div",null,
2   React.createElement(Greeting, {name: "John"})
3 )
```

Listing 2.11: Compiled Javascript from JSX

**Context API**

To share data between React components without using props-drilling a built-in solution is available named the Context API. A Context can be shared between multiple components in the component tree and is initialized using the createContext method available in the React library. To add data to the Context and make it available to children components a Provider available on the Context instance is used. The Provider is a component that takes one property named value where the data for the Context is specified which can be variables, functions or objects. By wrapping children components inside the Provider component the Context data can be accessed from any child component using a Consumer available on the Context instance as well. The application is not limited to one Context instance instead, multiple different contexts are used for different purposes.

Passing down callback functions to child components using Context API lets a child component modify the state of a parent component. According to the official documentation, Context is used to share data that is considered global for the tree of React components, for
example, if a user is logged in, the theme of the application or the language set in the application. Using Context may decrease the reusability of components because they rely on data provided through context from another component.

For a functional component wrapped in a Context Provider, useContext acts as the Consumer and the data can be accessed. An example using useContext hook is shown in Listing 2.12.

```javascript
const PersonsContext = React.createContext();

<PersonsContext.Provider value={{persons: persons}}>
  <Greetings />
</PersonsContext.Provider>

const Greetings = () => {
  const { persons } = useContext(PersonsContext);
  return persons.map(person =>
    <h1> Hello, {person.name} </h1>
  )
}
```

Listing 2.12: Accessing data using useContext hook

create-react-app

React provides a tool called create-react-app which sets up a React application from scratch using Node. It includes helpful tools that speed up development like hot-reloading and debugging, as well as tools for building an application for production. It also includes a package manager, a bundler and a compiler among other useful things.

The prerequisites for using create-react-app is Node and the Node Package Manager. The environment is set up by running the command npm init react-app <application-name> using a version of NPM which is of version 6 or higher [18].

The V8 Javascript engine cannot process syntax such as JSX, therefore a compiler is necessary that transforms JSX to normal Javascript that Node and all browsers can understand. The create-react-app toolchain includes a compiler named Babel. babel-core contains the necessary functionality for Babel to work and presets are used to compile different kinds of javascript, preset-env is used for compiling modern ECMAScript and preset-react is used for compiling JSX code.

create-react-app includes a bundler which bundles and minifies all application assets, as node_modules and required Javascript files named Webpack. Webpack is configured using a webpack.config.js file where the entry point of the files that should be bundled is specified and where the bundled files should be outputted. Furthermore, a development server that hosts the application locally is also configured in the config file which uses the web-pack-devserver library. For additional functionality, Webpack plugins can be used. One plugin used in create-react-app is called Hot Module Replacement which triggers a rebuild of modules when a change is made in the source code.

In package.json config file, a start script is present which starts the webpack development server and listens for changes. By running the start script the application is hosted locally using the port specified in the webpack config and can be viewed in the browser on the address http://localhost:<port>. package.json also includes a build script which bundles and minimizes the source files in the application into a build folder ready to be hosted on a web-server for production. A third script available is the test script which triggers a test runner that looks for files with the .test.js extension.

create-react-app abstracts a lot of functionality, to be able to tweak the configuration of the bundler or the compiler the application has to be “ejected”, this is done using the eject script. This is a one-way operation that exposes the underlying configuration for Webpack and Babel.
The React application generated contains the following directories:

- **node_modules**: containing all necessary dependencies for developing React applications.
- **public**: directory handling static assets in the application, including an index.html containing a `<div>` tag where the React application is rendered. Furthermore, it contains a link to the application icon, useful metadata, a title for the application as well as a link to a manifest.json file located in the public folder. The manifest.json file is a JSON file containing information on how it should behave when installed on a mobile phone as a progressive web app.
- **src**: Directory containing the source code, including CSS and Javascript files. The entry point that renders the application to the DOM is named index.js.

### 2.7 Architectural patterns

All functionality for a web application could, in theory, be placed in a single module handling DOM manipulation, data fetching and business logic but that leads to an error-prone application that is hard to maintain. Therefore architectural patterns are needed to build well-structured high-quality applications.

As mentioned in Section 2.6, the architectural pattern used depends on which framework or library the web application is built with. The most common architectural patterns used in web application frameworks are the Model-View-Control (MVC) pattern and the Model-View-ViewModel (MVVM) pattern. Table 2.2 summarizes the architectural patterns used in the most popular web application frameworks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Architectural pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angular</td>
<td>Framework</td>
<td>MVVM</td>
</tr>
<tr>
<td>Ember</td>
<td>Framework</td>
<td>MVVM</td>
</tr>
<tr>
<td>ExtJS</td>
<td>Framework</td>
<td>MVC or MVVM</td>
</tr>
<tr>
<td>Vue</td>
<td>Framework</td>
<td>MVVM</td>
</tr>
<tr>
<td>Ruby On Rails</td>
<td>Framework</td>
<td>MVC</td>
</tr>
</tbody>
</table>

Table 2.2: Architectural patterns used in popular frameworks and libraries

For web applications built using libraries, the architectural pattern is decided by the developer. React applications have a unidirectional data flow meaning that data flows downwards through the component tree using props as described in Section 2.6. Each component having its own data stored in its state. For larger React applications more advanced architectural patterns are needed to make the application maintainable which will be shown later in this thesis. The first one invented was Flux. Since then other architectural patterns have evolved as well as libraries that assist developers to implement these patterns. The most commonly used libraries are Redux, Apollo Client, MobX and Relay Client. A comparison between these based on the number of downloads using NPM is shown in Figure 2.7 on page 26. The numbers show that Redux is the most popular followed by Apollo Client which both will be studied further.
2.7. Architectural patterns

Figure 2.7: NPM downloads for Redux, Apollo Client, MobX and Relay Client
In the next sections, MVC and MVVM will be studied first to gain knowledge of how common architectural patterns work and why they are needed. Followed by the unidirectional architecture patterns Flux, Redux and Apollo Client used in React applications.

**Model-View-Controller**

The MVC pattern is one of the oldest architectural patterns and was traditionally used for building Graphical User Interfaces (GUI) for desktop applications \(^{25}\). It consists of three interconnected parts: The Model (M), the View (V) and the Controller (C). The ultimate goal of the MVC architecture is to separate the user interface from the underlying data that it represents \(^{26}\). The MVC pattern is illustrated in Figure 2.8.

![Figure 2.8: MVC architectural pattern](image)

The View contains all interface-specific functionality in the application for presentation of the content and user input. It has access to the model containing the data in the application and is responsible to inform the controller of any user interactions within the user interface. The appropriate View reacts to changes in the model and renders a new View based on it. The Model contains the data that the View presents and it can access data stored locally, fetch data from a third-party data API or from a database. The Controller is the central part of the MVC architecture and it communicates with both the view and the model. Controllers handle user-interaction made in the application and inform the model how the data should change to get to a new state. MVC applications often have multiple controllers each controlling different views.

The MVC pattern defines the three parts and describes how they communicate with each other. By having three separate parts each having different responsibilities emphasizes the Separation of Concern principle which is a widely accepted principle for increased code quality of software. By separating the model and the view, views can consist of different representations of the same data. The controller separates the logic of user interactions from the View, by doing this the same user-interface can act differently depending on which controller is used with it. The Model handles the data in the application and decides where data should be fetched from, a model can be replaced to fetch data from another source.

In early 2000’s several frameworks as Spring\(^{12}\), ASP.net\(^{13}\) and Ruby on Rails\(^{14}\) adopted the MVC architectural pattern to build web applications. These frameworks used the MVC pattern on the server-side. In this set up all the work is performed on the server and upon user interaction within the browser the controller on the server receives a browser request which in turn communicates with the model to perform a state change. A new View is generated and sent to the client as HTML. This implementation is illustrated in Figure 2.9 on page 28.

---

\(^{12}\)https://spring.io/
\(^{13}\)https://dotnet.microsoft.com/apps/aspnet
\(^{14}\)https://rubyonrails.org/
2.7. Architectural patterns

For a “thin server” architecture the complexity is moved to the client and the design for MVC is slightly changed. An initial GET request is sent to the server that responds with the SPA and the framework files. Subsequent requests to the server only receive pieces of raw data and the server acts as a data API. This is illustrated in Figure 2.10.

The MVC pattern has evolved to new patterns used in modern Javascript frameworks and libraries even though a few still support MVC. One of these is the Model-View-ViewModel pattern.

Model-View-ViewModel

The Model-View-ViewModel (MVVM) architectural pattern has evolved from the concepts of MVC. MVC has one critical drawback that the MVVM patterns try to solve: The communication between the View and the Model is bi-directional which means that the Model can update the View while the View also can grab data from the Model and update itself. This is problematic because it becomes difficult to predict what the state of the View is at a given time because changes can come from different sources. This makes the web application hard to maintain and test. This is illustrated in Figure 2.11 on page 29.

The solution to this suggested in the MVVM pattern is by tightly coupling views to a related ViewModel by having a two-way data-binding for getting and setting values. The ViewModel manages the Model data and the associated view using data bindings rather than through events. This makes the application more predictable and easier to test because a certain View-Model always results in the same View. The View is updated by observing changes in the ViewModel. The ViewModel also contains presentational logic that handles user interaction and calls other functionality for business logic processing from the Model. It never directly asks the view to display anything. Figure 2.12 on page 29 illustrates the structure of the MVVM architectural pattern.
2.7. Architectural patterns

Figure 2.11: MVC architectural pattern issue

Figure 2.12: MVVM architectural pattern
2.7. Architectural patterns

Flux

The flux architectural pattern was invented by Facebook and is, as MVC and MVVM, framework and library agnostic. It was derived from the MVC pattern with the purpose to have more control of the data flow of the application. The problem using MVC was the lack of control when updating Models and Views which also the MVVM pattern tries to solve. As described earlier Views and Models interact with each other in an error-prone manner. Models provide data to different views, views can update Models based on user interaction and Models can update other Models data. At Facebook, this led to problems debugging the data flow when developing web applications. Flux suggests a solution by using unidirectional data flow. The Flux architectural pattern consists of four pieces: Views, Actions, Dispatcher, and Stores.

Stores are where all the data in the application is located. Views that are interested in the data can subscribe to the stores and update accordingly. One important property of stores is that only the store itself can manipulate its data and exposes public getters for views to get the data. This is different from the MVC pattern where Views can update the data in the Model freely. To update the data in the store an Action is created. One Action exists for every change needed in the application affecting the store, it can be due to a user interacting with a View or a network request that has finished and got data as a response. Each Action is an object containing two fields: a type and a payload. The type field is a unique string describing what the action will do, for example, ADD_USER. The payload field contains the data that will be used to update the Store. For an Action to reach a Store there is one more piece in the Flux pattern, the Dispatcher. The Dispatcher is responsible for handling incoming actions before passing them to the Store and decides in which order the actions should be dispatched. By using actions the operations are decoupled from the Views and can be reused for multiple Views.

Analogies to MVC:

- C: Actions are similar to controllers in the MVC pattern. They are responsible for notifying the store what to change and contains logic for user interactions or any other change causing a state change in the application.
- M: The Store where the state of the application is located.
- V: The Views presenting the data.

The unidirectional data flow in Flux is illustrated in Figure 2.13.

There are a few libraries for implementing patterns deeply inspired by the Flux pattern. The most popular is Redux which is used in Canvas LMS and will be the one studied in this thesis.
Architectural pattern in Redux

Redux is a library used for implementing a pattern similar to Flux but with some additional features. In contrast to Flux, Redux uses a single store that acts as a “Single source of truth” for the whole application which is called the Redux store. Views subscribe to the Redux store and re-renders when it changes, which makes the data flow uni-directional as in the Flux pattern. Furthermore Redux adds one additional part to the Flux pattern which is Reducers. Reducers are pure functions that are used by Redux to decide how the state of the store should change for dispatched Actions. Pure functions are a special kind of function that have a set of characteristics:

- They do not rely on any other data than the data sent in as parameters.
- The arguments to a pure function are seen as immutable and should not be changed.

The Reducer functions take the previous state from the store as an argument and an Action. Using these two arguments it returns a new state object. A reducer is described by equation (2.13).

\[
f(state, [...actions]) = \text{newState}
\] (2.13)

The Redux library consists of five functions:

- createStore(reducer, [preloadedState], [enhancer])
- combineReducers(reducers)
- applyMiddleware(...middlewares)
- bindActionCreators(actionCreators, dispatch)
- compose(...functions)

createStore creates the Redux store taking a reducer as an argument. Reducers can become large and may need to be split up in smaller pieces each responsible for specific parts of the application, this is what the combineReducers function does. applyMiddleware is used for adding functionality that will run before an action is dispatched that can be used for asynchronous actions, a popular middleware library for handling asynchronous actions is the redux-thunk library. bindActionCreators bind action creators, which are functions that create actions, to the Redux dispatch function. Compose is a simple function composing functions from left to right.

The Redux store exposes four public methods that contain functionality for dispatching actions, subscribing to changes in the store and getting the current state of the store. These are the following:

- store.dispatch(action)
- store.subscribe(listener)
- store.getState()
- replaceReducer(nextReducer)

The unidirectional data flow using Redux is illustrated in Figure 2.14 on page 32.
Redux in React

Redux is commonly used as a data management solution for React applications and a library named react-redux\(^\text{15}\) can be used that makes the Redux store accessible from any React component. By connecting the Redux store to the components, updating and reading data from the Redux store can be done from any component in the application without using props-drilling. The react-redux library exposes a function called connect which is used to enhance React components with functionality for dispatching actions and reading the state from the Redux store. For the connect function to work it has to be able to access the instantiated Redux store in the application, this is done using the Provider component available in the react-redux library. This is shown in Listing 2.13.

```javascript
<Provider store={store}>
  <Greetings />
</Provider>

const Greetings = ({ persons, updatePersons }) => {
  return persons.map(person =>
    <h1>Hello, {person.name}</h1>
  )
}

export default connect(
  state => {return {persons:state.persons}},
  dispatch => {return {updatePersons:dispatch(/*...*/)}})(Greetings)
```

Listing 2.13: Accessing Redux store using Connect

Architectural pattern in Apollo Client

Apollo Client is a complete data management solution for updating and storing data in web applications as well as fetching data from GraphQL APIs. Using Apollo Client for data management results in an architecture with unidirectional data flow similar to the Flux architectural pattern.\(^\text{15}\)

\(^{15}\)https://react-redux.js.org/
2.7. Architectural patterns

A single source of truth is used containing the application state, this is called the Apollo Client Cache. The Cache stores all results from the fetched queries. By caching query results the Apollo Client prevents sending multiple network requests to the API. It is possible to bypass the cache if the data need to be re-fetched from the server by changing the fetching policy to “network-only”. Apollo client cache normalizes the data fetched so if a list of persons are fetched in a query every person is cached separately, therefore subsequent queries for a single user are fetched from the normalized cache.

Apart from querying remote data from an API, local data can also be queried and stored in the same manner. Local data is stored alongside remote data in the Cache and is queried with an additional decorator named @client placed after the query. Available local data types and queries are specified using a schema in the same way as for the GraphQL API on the server described earlier in this section. Updating local data in the Cache is done using client Resolvers as for remote data.

Analogies with Redux and Flux:

- **Mutation**: Created by the View to update the state of the application and contains variables with data. Similar to Actions in Flux and Redux.
- **Apollo Client dispatch methods**: The Apollo Client contains methods for dispatching mutations. This is similar to the Dispatcher in Flux and Redux.
- **Apollo Client Cache**: Stores local and remote data in the application. Similar to the Store in Flux and Redux.
- **Resolvers**: Functions updating the state of the Cache. Similar to Reducers in Redux.

The unidirectional flow using Apollo Client is illustrated in Figure 2.15.

Figure 2.15: Apollo Client architectural pattern

Apollo Client in React

The library apollo-react[^16] is used to provide the Apollo Client functionality to React Views. The Apollo Client is initialized in the root component of the application and to access it from any child component there are four available options apart from props-drilling. Every option provides the ability to perform queries and mutations from within React views.

[^16]: https://www.apollographql.com/docs/react/
1. Wrapping the root component with an ApolloProvider component from the react-apollo library. The instance of the Apollo client is set as a property for the ApolloProvider and components further down the component tree can access the client by using the withApollo method available from the react-apollo package. This is illustrated in Listing 2.14.

```javascript
<ApolloProvider client={client}>
  <Greetings />
</ApolloProvider>

const Greetings = ({ client }) => {
  const { loading, error, data } = client.query(GET_PERSONS)
  if (loading) return "Loading..."
  if (error) return error.message
  return data.persons.map(person =>
    <h1>Hello, {person.name}</h1>
  )
}
export default withApollo(Greetings);
```

Listing 2.14: Accessing Apollo Client using withApollo

2. Using React hooks to access the Apollo Client in functional components has a similar approach. The ApolloProvider is included from an additional library named @apollo/react-hooks and the Apollo Client is accessible from the child components using the useApolloClient hook. By using the useQuery and useMutations hooks, queries and mutations can be executed directly. This is illustrated in Listing 2.15.

```javascript
<ApolloProvider client={client}>
  <Greetings />
</ApolloProvider>

const Greetings = () => {
  const { loading, error, data } = useQuery(GET_PERSONS)
  if (loading) return "Loading..."
  if (error) return error.message
  return data.persons.map(person =>
    <h1>Hello, {person.name}</h1>
  )
}
export default Greetings;
```

Listing 2.15: Accessing Apollo Client using Apollo hook

3. Using the ApolloConsumer component from any component available in the react-apollo library by providing a render function with the Apollo client as its only argument. Illustrated in Listing 2.16.

```javascript
const Greetings = () => {
  <ApolloConsumer>
    {client => {
      const { loading, error, data } = client.query(GET_PERSONS)
      if (loading) return "Loading..."
      if (error) return error.message
      return data.persons.map(person =>
        <h1>Hello, {person.name}</h1>
      )
    }}
  </ApolloConsumer>
}
```

Listing 2.16: Accessing Apollo Client using ApolloConsumer
4. To perform a Query or Mutation in a component directly the components Query and Mutation can be exported from the react-apollo library and used in a declarative way in the render method. This is illustrated in Listing 2.17.

```jsx
<Query query={GET_PERSONS}>
  {({ loading, error, data }) => {
    if (loading) return "Loading..."
    if (error) return error.message
    return data.persons.map(person =>
      <h1>Hello, {person.name}</h1>
    )
  }}
</Query>
```

Listing 2.17: Quering using Apollo Client Query component

### 2.8 Design Science Research

Design Science Research is a set of guidelines for gaining and contributing knowledge. It is a problem-solving paradigm where innovative “artifacts” are built and analyzed to gain knowledge of how things work in practice that is not known before [39].

The five steps of the Design Science process are the following:

- **Awareness of Problem:** It may come from different sources, it can be from development in the industry or by indication of problems in the studied system. The outcome of this stage in the process model is a Proposal for a new research effort.

- **Suggestion:** It is the immediate stage after the proposal for a new research effort. Suggestion(s) on how to approach the problem is formed in a Tentative Design which is the output of this creative phase.

- **Development:** It is the phase where the Tentative design is implemented. The outcome of this phase is an “artifact” which is a realization of the solution in its environment.

- **Evaluation:** It is the phase where the artifact constructed is evaluated based on the criteria stated in the proposal. Rarely in design science research, the initial hypothesis is completely satisfied. Instead, additional information has been gathered in the development phase and together with the evaluation of the artifact a new round of suggestions are fed into the process model as shown in the figure.

- **Conclusion:** It is the last phase of the process and the end of a design science research effort. In this phase, it is determined why or why not certain artifacts worked within its environment. For a successful process when the hypothetical predictions fit the results well enough the Design Science Knowledge is reported, which is the contribution of knowledge.

The process is illustrated in Figure 2.16 on page 36 from the paper "Design Science Research in Information Systems" [39].
Figure 2.16: Design Science Research process
This chapter describes how the evaluation process of the architectural patterns for React covered in the Theory section has been carried out using the Design Science Research approach introduced in Section 2.8.

3.1 Study object

To choose a suitable study object from within Canvas LMS to perform the research on, it had to fulfill a few requirements:

- It should be using industry standards.
- It should be using modern technology.
- It should be of moderate size.

Canvas LMS recently moved from using the “server-thick” web framework Ruby on Rails\(^1\) to using React for all the client-side development. While some parts of the Canvas LMS are still using Ruby on Rails it contains over 50 parts built with React which is the majority of the user interface. Therefore the study object had to be built using React which is both an industry standard and a modern technology. Furthermore, the size of the study object is important because a small application does not necessarily need an architectural pattern because built-in functionality like State and Props in React may be enough. For medium and large applications the architecture becomes more complex and therefore architectural patterns are needed. To focus mainly on the architectural patterns instead of domain-specific functionality the study object had to be of moderate size.

The study object chosen based on these criteria was the page for administrators to handle users in Canvas LMS built with React and Redux. This module is an essential piece of the platform where administrators create and edit students, teachers and as well as other administrators in the platform. Users can be filtered using live-search with text input as well as categorized filtering using roles. Each user is represented with a row containing the avatar, name, email address, SIS (School Information System) ID and last login date. Each row also contains three links: A link to the “act as”-user page where administrators can masquerade

\(^1\)https://rubyonrails.org/
the user, a link to the Canvas mailbox for sending email to the user and a link for editing the user. The list of users can be sorted using the header for each table column. It also contains pagination showing 15 users per page as default. The Users page is shown in Figure 3.1.

![User administration page in Canvas](image)

Figure 3.1: User administration page in Canvas
The user data is fetched from the Canvas LMS REST API endpoint, located at /api/v1/accounts/:account_id/users using a GET request. The :account_id parameter can be any account id on the Canvas instance. Only administrators of the Canvas instance have permission to list all users. The API responds with a paginated list of users associated with the account. The request accepts four parameters: A search term, type of enrollment, column to sort the results by and finally if the results should be sorted in ascending or descending order. The detailed descriptions for the parameters can be found in Table 3.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search_term</td>
<td>string</td>
<td>The partial name or full ID of the users to match and return in the results list. Must be at least 3 characters.</td>
</tr>
<tr>
<td>enrollment_type</td>
<td>string</td>
<td>When set, only return users enrolled with the specified course-level base role. This can be a base role type of student, teacher, ta, observer, or designer.</td>
</tr>
<tr>
<td>sort</td>
<td>string</td>
<td>The column to sort results by. Allowed values: username, email, sis_id, last_login.</td>
</tr>
<tr>
<td>order</td>
<td>string</td>
<td>The order to sort the given column by. Allowed values: asc, desc.</td>
</tr>
</tbody>
</table>

Table 3.1: Canvas API Users parameters

Users administration page architecture

The user administration page architecture is illustrated in Figure 3.2.

![Figure 3.2: User administration page architecture](image-url)
The entry point of the application is the index component which renders the content to
the DOM using reactDOM.render method provided by the react-dom library. In the Canvas
LMS platform, this component was used in pages for administering accounts and courses
as well. This application only focuses on the User administration page and therefore the
functionality for rendering the account and course page was removed. The second com-
ponent in the tree was also used for the account page and course page and had the name
AccountCourseUserSearch, this component was renamed to UsersSearch and functionality
for supporting the account and the course page was removed.

For clarity leaf nodes of the component tree were excluded from Figure 3.2 because they
are not impacting the data-flow in the application. Leaf node components are for example
buttons, links, icons, loading indicators, and input fields. These components were provided
by the third-party library instructure-ui which is Instructure’s open-source library for React
components.

3.2 Preparation

To be able to perform the Design Science Research on the user administration page it had to
be extracted from the rest of the platform into an isolated application in a local development
environment. This was done by extracting the source code and its dependencies from the
Canvas LMS repository on Github.

Dependencies

The user administration page contained internal dependencies to other modules within the
repository as well as dependencies on external libraries. External dependencies are included
using NPM and these are specified in the package.json file and are shown in Listing 3.1

```
"dependencies": {
  "@instructure/canvas-theme": "^6.4.0",
  "@instructure/ui-ally": "^5",
  "@instructure/ui-billboard": "^5",
  "@instructure/ui-buttons": "^5",
  "@instructure/ui-core": "^5",
  "@instructure/ui-form-field": "^5",
  "@instructure/ui-forms": "^5",
  "@instructure/ui-icons": "^5",
  "@instructure/ui-layout": "^5",
  "@instructure/ui-menu": "^5",
  "@instructure/ui-overslays": "^5",
  "@instructure/ui-pagination": "^5",
  "@instructure/ui-svg-images": "^5",
  "axios": "^0.18.0",
  "immutability-helper": "^2",
  "jquery": "https://github.com/ryankshaw/jquery.git#dadf794ebc4d4c4d4c45dad4d4d4b6d7cc",
  "lodash": "^4.16.4",
  "page.js": "^4.13.3",
  "parse-link-header": "^1",
  "prop-types": "^15",
  "qs": "^6.6.0",
  "react": "^16.8.6",
  "react-dom": "^16.8.6",
  "react-scripts": "^3.0.1",
  "redux": "^4.0.1",
  "redux-thunk": "^2.3.0"
},
```

Listing 3.1: External dependencies for study object specified in package.json
3.2. Preparation

Setting up Plato

The static code analysis tool Plato mentioned in Section 2.2 was used to calculate some of the metrics. Plato was installed using the Node Package Manager. Plato offers multiple options to customize the process of the static analysis as setting a title and a date of the report, excluding certain files and specifying other configurations. For a more customized behavior of the analysis, Plato can be executed using a script. The reports generated by Plato by default do not calculate the average values for the Halstead metrics nor the Cyclomatic Complexity. They focus on file-by-file analysis to spot complexity issues in single files. Calculating the average values for the whole application was beneficial in this research where evaluation is performed for the same application using different implementations. Therefore a small program was developed using Node and it is located in analyze.js in the root folder of the application. An example output of the program is shown in Figure 3.3.

![Figure 3.3: Plato custom script output](image)

When calculating the average values of all present files in a system removing empty source code files results in a less maintainable system which is not correct. To have a correct comparison between the implementations in each iteration the number of files had to be consistent. Therefore empty Javascript files were added when needed having zero complexity.
3.3 Initial implementation

The tool create-react-app introduced in Section 2.6 was used to create the developing environment for the study object. All source files generated by the create-react-app were removed except index.html and index.js which was the entry point for the application. In the initial implementation, the source code for the users’ administration page was added to the generated src folder of the application.

Source code

The application is built using the Redux architectural pattern. The createStore method provided by the redux library is used to create the Redux store and is enhanced with the redux-thunk middleware for handling asynchronous actions. The createStore method takes two input arguments: an initial state for the store and a reducer.

The initial state consists of an object named userList which contains five fields: users, isLoading, errors, links, and searchFilter. The initial state is shown in Listing 3.2.

```
userList: {
  users: [],
  isLoading: true,
  errors: {search_term: ''},
  links: undefined,
  searchFilter: {search_term: ''}
}
```

Listing 3.2: Initial Redux store state for study object

Users are remote data that is fetched from the Canvas LMS API and the other fields are local data. A few fields concerned with switching between the user, account and course page was removed from the initial state because they were not relevant to this application.

The application contains one reducer updating the store based on dispatched actions and it is named rootReducer. The reducer returns a new immutable state based on the previous state and the action. Every action object contains a type and a payload according to the Flux pattern. The type decides which action to be executed and the payload includes the data that should change. The rootReducer contains functionality for handling five actions: GOT_USERS, GOT_USER_UPDATE, UPDATE_SEARCH_FILTER, SEARCH_TERM_TOO_SHORT, and LOADING_USERS.

- **GOT_USERS**: Updates the state with new users, sets the loading status to false and updates the links for pagination of the users.
- **GOT_USER_UPDATE**: Updates a single user in the users array.
- **UPDATE_SEARCH_FILTER**: Updates the search filter: search term, role, page, and order. It also resets the error state for a too-short search term.
- **SEARCH_TERM_TOO_SHORT**: Updates the errors state to contain a “term too short”-message.
- **LOADING_USERS**: Updates the isLoading state to true.

The Redux store is passed to the UsersSearch component as a property upon initialization. By doing this it is available to the user interface of the application. The Redux store is then passed further down the component tree to the UsersPane component.

The application is heavily dependent on the UsersPane component because it is the only component in the component tree connected to the Redux store. It is responsible for providing store data down the component tree, subscribing for changes in the Redux store as
3.3. Initial implementation

well as dispatching actions to update the store. In the constructor of the class-based component UsersPane the current state of the Redux store is extracted using the getState() method available on the store. The extracted state is saved into the UsersPane local state using the setState() method inherited from React.Component.

The UsersPane component defines two life-cycle methods apart from the constructor and the mandatory render method, these are componentDidMount and componentWillUnmount. In the componentDidMount method, the component subscribes to the Redux store and specifies a change listener that will be called every time the store state changes. The change listener is a class method named handleStateChange that updates the local state of the component to match the current Redux store state. In the other life-cycle method, componentWillUnmount, the component unsubscribes from the store changes.

After mounting the component and subscribing to the Redux store two actions are dispatched using the dispatch method on the Redux store object. The first action dispatched is created using the action creator updateSearchFilter which updates the searchFilter state in the store to reflect query params in the URL which are passed down as properties along with the store from the UsersSearch component. The second action creator applySearchFilter triggers the initial fetch for users.

The applySearchFilter action creator performs the following tasks:

1. Gets the current state of the search filter.
2. Checks if the search term is either empty or longer than a minimum search length.
   - If the search term is too short it dispatches an action called displaySearchTermTooShortError action which has the type SEARCH_TERM_TOO_SHORT and a payload containing an error message. This error message is used in the UsersToolbar component.
   - If the search term is valid it first dispatches an action called loadingUsers which has the type LOADING_USERS and an empty payload. Then it initializes the asynchronous fetching of users. When the fetching finishes it returns a response containing users which are then dispatched in a new action called gotUserList. gotUserList has the type GOT_USERS and a payload including the users and the XMLHttpRequest object containing the links used for pagination.

Fetching the users from the Canvas LMS API is done using the UsersStore object which contains methods for performing asynchronous network requests using jQuery library. To create the UsersStore the function createStore is used which adds necessary functionality can also be enhanced with custom properties. createStore is reused in several places in the Canvas platform with different purposes and therefore contains a lot of functionality. Functionality not used in the User administration page was removed to have the study object as small as possible for better comparisons to other implementations. When initializing UsersStore using the createStore constructor a function named getUrl is specified which specifies the URL to where the users should be fetched from. Furthermore, a function named normalizeParams is set which removes falsy values from the search filter URL as well as including default parameters to the fetching URL. The first default parameter specifies that each user row returned, apart from the essential information, should include last logged in information, an URL to the avatar, email address and the timezone for the user. The second default parameter specifies the number of users that should be returned which is set to fifteen.

The createStore function constructs an object containing a load method by using getUrl and normalizeParams specified. The load method fetches users using the Ajax method provided in the library jQuery. This function is extended to include error handling, authentication, and parsing of JSON responses.
3.3. Initial implementation

Measurements

Measurements were performed using the metrics introduced in Section 2.3 for maintainability. The selection of these was based on which metrics were the most relevant for evaluating the architecture of the application as well as which metrics that were supported by Plato. One of them is the Design Complexity of Card and Agresti which measures the structural complexity (coupling) and local complexity (cohesion) of the application. The second metric is the Software Science Effort which measures the complexity and thus the maintainability of the application. The third metric is the Cyclomatic Complexity which measures source code complexity. The fourth and last metric used in the evaluation is Lines of Code which measures the source code size of the application and thus the complexity.

The initial implementation results in the values shown in Table 3.2 for Design Complexity which was calculated manually using the illustration of the data flow in the application shown in Figure 3.4, Figure 3.5, and Figure 3.6. Figure 3.4 illustrates how functions for dispatching actions to the Redux store flows in the component tree, figure 3.5 illustrates how static data flows in the application and finally figure 3.6 illustrates how data stored in the Redux store state flows in the application. The data flow in the application was divided into three figures for a better overview and was used in the Design Science process to find improvements to the architecture.

<table>
<thead>
<tr>
<th>Sum of (v_m/(f_m+1))</th>
<th>34.33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of (f_m^2)</td>
<td>20</td>
</tr>
<tr>
<td>Number of modules ((n))</td>
<td>10</td>
</tr>
<tr>
<td>Structural complexity ((S))</td>
<td>2</td>
</tr>
<tr>
<td>Local complexity ((L))</td>
<td>3.43</td>
</tr>
<tr>
<td>Design complexity ((DC))</td>
<td>5.43</td>
</tr>
</tbody>
</table>

Table 3.2: Design Complexity measures for initial implementation

![Diagram](image-url)
3.3. Initial implementation

Figure 3.5: Data flow graph for static data in initial implementation
3.3. Initial implementation

Figure 3.6: Data flow graph for Redux store state data in initial implementation
3.4 Design Science Research approach

The Software Science Effort was measured using Plato and the customized script for calculating the average values per file. The measurement resulted in the number of operators and operands shown in Table 3.3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct operators ($n_2$)</td>
<td>16.09</td>
</tr>
<tr>
<td>Total operators ($N_2$)</td>
<td>104.32</td>
</tr>
<tr>
<td>Distinct operands ($n_1$)</td>
<td>50.59</td>
</tr>
<tr>
<td>Total operands ($N_1$)</td>
<td>129.09</td>
</tr>
</tbody>
</table>

Table 3.3: Number of Operators and Operands for initial implementation

This resulted in the average Software Science Effort values shown in Table 3.4.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vocabulary ($n$)</td>
<td>66.68</td>
</tr>
<tr>
<td>Observed Length ($N$)</td>
<td>233.41</td>
</tr>
<tr>
<td>Volume ($V$)</td>
<td>1544.75</td>
</tr>
<tr>
<td>Difficulty ($D$)</td>
<td>20.15</td>
</tr>
<tr>
<td>Effort ($E$)</td>
<td>53826.98</td>
</tr>
</tbody>
</table>

Table 3.4: Software Science Effort measures for initial implementation

Based on the value for Software Science Effort the estimated seconds it requires to implement the program as well as an estimation of the number of delivered bugs in the implementation results in the values shown in Table 3.5.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time ($T$)</td>
<td>2990.39</td>
</tr>
<tr>
<td>Delivered bugs ($B$)</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 3.5: Halstead Time and Estimated Bugs for initial implementation

Furthermore, measurements for the average Lines of Code and Cyclomatic Complexity also calculated using Plato are shown in Table 3.6.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines Of Code (LOC)</td>
<td>94.14</td>
</tr>
<tr>
<td>Cyclomatic Complexity (CC)</td>
<td>7.27</td>
</tr>
</tbody>
</table>

Table 3.6: LOC and CC measures for initial implementation

3.4 Design Science Research approach

In this section, the process for improving the maintainability of the user administration page using the Design Science Research approach will be described. Every iteration will go through four out of five process steps of the design science research approach: Awareness of Problem, Suggestions, Development and Evaluation. Finally, the fifth step conclusion will be made on the research.

The source code for the initial implementation, as well as the implementations for every iteration, is available on Github.

https://github.com/davjo664/thesis
Iteration 1 - Redux connect

Awareness of Problem

The user administration page is using Redux for data management and the Redux store is initialized in the index component. The store is passed down as a property to UsersSearch component along with the static data: permissions, rootAccountId, accountId and roles. Neither the Redux store nor the static data are used in the UsersSearch component, instead, they are passed down to the next component in the component tree named UsersPane. In the UsersPane component, the Redux store state is set as the local state for the component. This component is responsible for dispatching all actions in the application to the Redux store. It is also responsible for keeping track of each change made to the store state and provides its child components with the current state. Furthermore, it provides its child components with functions as updating the search filter and to execute a new fetch for users. Passing down functions and other data leads to an issue referred to as props-drilling where props are passed down through several components to reach the component it is used by. Props drilling affects coupling and cohesion of components negatively which affects maintainability as they get harder to reuse and become less modular and harder to test.

For example, the method handleApplyingSearchFilter declared in the UsersPane component is used in the component CreateOrUpdateUserModal. This method has to be passed down as a property to the component UsersList, then passed further down to UsersListRow, and finally passed down to CreateOrUpdateUserModal where it is used after updating or creating a new user. Another method used in multiple parts of the application is handleUpdateSearchFilter which updates the current state of the searchFilter, this is declared in the UsersPane and passed down as a property through UserList and used by UsersListHeader. This can be seen in Figure 3.4 on page 44.

Apart from dispatch functions, static data also uses props-drilling to get to the correct component. This can be seen in Figure 3.5 on page 45. This data are stored outside of the redux store which violates the single source of truth principle of the redux architectural pattern.

The redux store state consists of links, users, isLoading, errors, and searchFilter. The users state contains the users fetched and are used in UsersPane, UsersList, and SearchMessage. The links state which is used for pagination is used in the SearchMessage component. Furthermore, the isLoading state indicates if users are being fetched from the API and are used in UsersPane and SearchMessage component. The errors state containing the error message when the search term is to short is used in the UsersToolbar component. And finally, the searchFilter state containing the search term and other search filters is used in UsersToolbar and UsersListHeader. This leads to many properties being passed through components to reach its destination. This can be seen in Figure 3.6 on page 46.

Props-drilling makes the components more difficult to test independently because they require other components to pass down the properties needed. It also makes components harder to reuse because components passing properties through them without using them have low cohesion with many input and output variables. Finding bugs becomes harder because many components may be included in the data-flow and tracking the issue may require to debug each of them. Due to these reasons, maintainability is negatively affected and the proposal for improving this is by sharing data and dispatch function not using props-drilling and by storing all data in a single source of truth.

Suggestion

By storing all the application data in the Redux store it will be easier to access in the same manner in the whole application. To enable all components to access the application state as well as being able to dispatch actions for updating the state every component needs to have access to the redux store instance of the application. A library called react-redux enables
3.4. Design Science Research approach

this by connecting the global redux store to react components using React Context under the hood.

Passing props from parent to children components are the key part of building React applications but when the props have to be passed down multiple levels and used in multiple parts of the application the coupling increases and the code will become harder to maintain. Therefore the hypothesis is that coupling and cohesion in the application will improve by using the new architecture implemented with Redux and the react-redux library.

Development

Firstly the static data permissions, rootAccountId, accountId and roles were stored in the Redux store by adding these to the initialState object and thus following the single source of truth principle. The data was then accessible through the store and could be removed from the index component where it was previously defined. The package react-redux was imported which provides a component named Provider used to wrap the UsersPane component. The Provider component accepts one property which is the redux store instance of the application. This connected the Redux store to the user interface and enabled all components in the component tree to access it and is shown in Listing 3.3.

```jsx
const store = configureStore(initialState)
ReactDom.render(<Provider store={store}><UsersPane /></Provider>, document.getElementById("root")
)
```

Listing 3.3: Providing Redux store to the UsersPane component

To access the Redux store from a component in the application the react-redux library provides a function called connect which connects a React component to a Redux store. The connect function accepts four arguments as well as the component to be connected. The two most important arguments which were used in this application were mapStateToProps and mapDispatchToProps. All components needing data from the Redux store or needing to dispatch actions to update the Redux store state is connected to the store similarly using the connect function provided by the react-redux library. What differs is the mapStateToProps and mapDispatchToProps functions provided as arguments to the connect function. These are called the selectors because they select what parts of the Redux store that should be accessible to each react component. All state data and dispatch functions specified in the mapStateToProps and mapDispatchToProps functions are available as props in the component. The mapStateToProps uses the Redux state parameter to extract the parts of the state that the component needs and mapDispatchToProps use the dispatch variable available to dispatch actions to the store. Listing 3.4 shows the creation of the functions and how it is connected to the component in the UsersPane component.

```jsx
const mapStateToProps = state => {
  return (
    { users: state.userList.users,
      isLoading: state.userList.isLoading,
      searchFilter: state.userList.searchFilter
    })
}

const mapDispatchToProps = (dispatch, props) => {
  return {
    updateSearchFilter: (filter) => {
      dispatch(UserActions.updateSearchFilter(filter))
    },
    applySearchFilter: () => {
      UserActions.applySearchFilter()
    }
  }
}
```

Listing 3.4: mapStateToProps and mapDispatchToProps
Listing 3.4: Connecting the UsersPane component to the Redux store

Here, the mapDispatchToProps uses the two action creators updateSearchFilter and applySearchFilter that returns actions based on the parameters sent to them.

As mentioned UsersPane had a lot of responsibilities previously, it was responsible for dispatching actions for every state change in the application as well as providing all child components with application state data. These responsibilities have been removed and replaced by instead enabling each component to get the state data from the redux store and dispatching actions. The only method apart from the constructor and the render method that is still present in the UsersPane component is the componentWillMount lifecycle method which is responsible for the initial updating of the search filter based on the URL parameters, and for applying the filter which triggers a new fetch of users.

The UsersSearch component was no longer needed because no properties were passed through it and the function updateQueryParams previously located in this component which updated the URL parameters to reflect the current state of searchFilter was moved to the action creator applySearchFilter. Instead the index component uses the UsersPane component right away.

Evaluation

The data flow for the new implementation using react-redux is shown in Figure 3.7 on page 51. Components can now dispatch actions using the dispatch method as well as extract data from the state object both provided by the Redux store module. The data flow for updating the Redux store from within a component (View) was illustrated in Figure 2.14 on page 32. The Design Complexity was calculated manually which resulted in the values shown in Table 3.7.

| Sum of \( \frac{v_m}{f_m+1} \) | 11.5 |
| Sum of \( f_m^2 \) | 18 |
| Number of modules (n) | 10 |
| Structural complexity (S) | 1.8 |
| Local complexity (L) | 1.15 |
| Design complexity (DC) | 2.95 |

Table 3.7: Design Complexity measures for Iteration 1
3.4. Design Science Research approach

Figure 3.7: Data flow graph for Iteration 1
3.4. Design Science Research approach

In comparison to the initial implementation, the sum of fan-outs from the components remained the same but the sum of fan-out squared decreased by two, from 20 to 18. Furthermore, the number of input/output variables decreased from 80 to 22 and the number of modules in the system stayed the same due to that UsersSearch was removed and Redux store was added to the architecture as a separate global module. Compared to the initial implementation this led to a decrease of local complexity by 66.47%, a decrease of the structural complexity 10% and a decrease of the total Design Complexity by 45.67%.

The implementation using the react-redux library resulted in a reduced number of files from 22 to 21, due to the removal of the UsersSearch component. To keep the number of files consistent one empty Javascript file was added having a complexity of 0. Calculating the Software Science Effort, LOC and CC using the static code analysis tool Plato resulted in the values shown in Table 3.8.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct operators ($n_2$)</td>
<td>15.68</td>
</tr>
<tr>
<td>Total operators ($N_2$)</td>
<td>102.41</td>
</tr>
<tr>
<td>Distinct operands ($n_1$)</td>
<td>50.73</td>
</tr>
<tr>
<td>Total operands ($N_1$)</td>
<td>123.14</td>
</tr>
<tr>
<td>Vocabulary ($n$)</td>
<td>66.41</td>
</tr>
<tr>
<td>Observed Length ($N$)</td>
<td>225.55</td>
</tr>
<tr>
<td>Volume ($V$)</td>
<td>1513.41</td>
</tr>
<tr>
<td>Difficulty ($D$)</td>
<td>18.88</td>
</tr>
<tr>
<td>Effort ($E$)</td>
<td>53243.40</td>
</tr>
<tr>
<td>Time ($T$)</td>
<td>2957.97</td>
</tr>
<tr>
<td>Delivered bugs ($B$)</td>
<td>0.50</td>
</tr>
<tr>
<td>Lines Of Code (LOC)</td>
<td>91.36</td>
</tr>
<tr>
<td>Cyclomatic Complexity (CC)</td>
<td>7.32</td>
</tr>
</tbody>
</table>

Table 3.8: Software Science Effort, LOC and CC measures for Iteration 1

The react-redux library was used for connecting components directly to the Redux store which enabled components to dispatch actions and getting relevant data without using props-drilling between components. This resulted in a system having less complexity and therefore better maintainability.

Every metric measured apart from the Cyclomatic Complexity indicates an improvement of the maintainability of the system. The Software Science Effort metric resulted in an improvement from 53826.98 to 53243.40 which is a percentage decrease of the effort by 1.08%. As a result of the decreased Software Science Effort the estimated time required to implement the data flow was decreased by 1.08% to 2957.97 seconds per file on average as well as the estimated delivered bugs by 1.96% to a value of 0.5 bugs per file in average.

The average Lines Of Code was reduced from 94.14 to 91.36 which is a decrease in the size of the system by 2.95%. The average Cyclomatic Complexity resulted in increased complexity of the system from a value of 7.27 to 7.32 which is an increase of 0.69%.

The evaluation shows that by using react-redux library to handle fetching and updating data in the application and by storing all data in a single source of truth, structural complexity (coupling) and local complexity (cohesion) could be improved in comparison to using props drilling. The Software Science Effort using the Halstead measures, and the Lines Of Code metric showed only small improvements and the Cyclomatic Complexity was not improved at all. In an attempt to improve these metrics a new iteration cycle was conducted to improve the complexity of the application further and thus the maintainability by investigating another architecture.
3.4. Design Science Research approach

Iteration 2 - GraphQL, Apollo Client Cache and React Context API

Awareness of Problem

In the first iteration by adding the react-redux library and connecting components to a global Redux store the Design Complexity was decreased by 45.67%. The source code complexity and size was only improved slightly with an improvement of at most 2.95%.

The Redux library requires a few parts that are necessary for it to function that affects the source code complexity and size of the system negatively. These parts are Actions, Reducers, the configuration of the Redux store, custom fetching logic as well as functionality for connecting the Views to the Redux store. The Reducers are responsible for normalizing data fetched from the REST API to Views in parts that they can handle efficiently. If the API would provide more flexible data the reduc ers would not be necessary.

The manual implementation for fetching data adds large amounts of complexity and size to the application. By eliminating this the maintainability would be improved. The configuration and initialization of the Redux store are located in five files each handling essentials parts for the Redux store to function. A simpler setup would benefit the maintainability of the system.

To access the global Redux store from a component the connect method provided by the react-redux library is necessary to use. The connect function takes two arguments: mapStateToProps and mapDispatchToProps. Both these functions require certain amounts of code to map the Redux store state to properties to the component as well as mapping dispatch functions to the component properties. Again, if accessing the data and updating it would be more flexible mapping would not be necessary. Finally, class-based react components require more code than function components and the size of the application is impacted negatively.

The proposal is therefore to use a more flexible way of fetching and storing data and connecting this to the components. As well as using function components instead of class-based components.

Suggestion

The suggested approach for reducing complexity and thus maintainability for the application was by using an architecture that is more lightweight and flexible. Complexity handling data fetching from the API could be improved by using the GraphQL API instead of the REST API available for Canvas LMS. The hypothesis is that by using GraphQL and requesting the necessary data from the component reduces the complexity that the client will have to handle. The GraphQL client that will be used to consume the GraphQL API is Apollo Client which works for any GraphQL API and is framework agnostic. To connect the Apollo Client with React the library @apollo/react-hooks could be used to perform queries and mutations from within the react components using React Hooks.

There are three types of data in the application: local application state data, remote data, and static data. The remote data are the users fetched from the Canvas LMS API. The local application state data are searchFilter and errors. Static data as mentioned earlier is accountId, rootAccountId, permissions and roles. These will be updated using mutations and accessed using queries. The searchFilter is updated when a user changes the sorting using the UsersListHeader component or by changing the search term or the role filter in the UsersToolbar. Therefore these two components have to be able to access the Apollo Client instance for making a mutation that changes the search filter. Both remote data and local application state data will be stored and accessible in the Apollo Client Cache.

The static data needs to be accessible from multiple Views as well. To be able to pass data down the component tree without using props-drilling React’s Context API will be used. Context API is included in the react library and can be used by importing the useContext hook. To be able to use the React hooks provided for Context API and Apollo Client the
class-based components will be rewritten into function components which should reduce the size of the components and thus the complexity.

Development

Firstly, the redux specific dependencies for the application specified in package.json was removed. These were: redux, redux-thunk and react-redux and replaced with the Apollo Client dependencies: apollo-client, apollo-cache-inmemory which is a cache configuration for Apollo Client and apollo-link-http which is used for making HTTP requests. Apollo Client requires that the graphql library is available as well and the graphql-tag library to define queries and mutations. Network communication with the API does not require any manual implementation using jQuery, instead, it is handled automatically using Apollo Client. These dependencies were therefore removed as well.

The Redux store is replaced by the apollo client cache, therefore all files for implementing it are removed. The Redux store which was initialized in the index component was replaced by an instance of the Apollo Client using the apollo-client library. The Apollo Client was configured using four settings: the cache used to store the data, the http link to the Canvas LMS GraphQL API, the client resolvers for handling local cache queries and mutations and lastly the local schema including type definitions.

The client schema specifies two queries: One for getting the state of the searchFilter and one for getting the state for the errors from the cache. It also specifies one mutation named updateSearchFilter which updates the search filter using a client resolver. The searchFilter type used as input for updating the filter consists out of four fields: sort, search_term, role_filter_id and order. The errors type is an object containing one field called search_term.

There is one client resolver defined for the updateSearchfilter mutation which checks if the new search filter is correct and sets the new searchFilter. If the search term is too short it updates the error state data and includes an error message. Updating the searchFilter is done by combining the old filter and the new filter and writing the result to the cache. After the new searchFilter data is written to the cache it updates the window URL to reflect the current search filter. Writing to the cache is done using the method cache.writeQuery. The cache object is available by default in all resolvers.

After initializing the Apollo Client the initial state of the cache was set by using the writeData method for the InMemoryCache object. The search filter was set to empty strings for its fields, the page is set to the first and the errors to null. To make the Apollo Client available to all React components a component named ApolloProvider was used imported from the apollo-client library. This works similarly to the Redux store provider used in the previous implementation which internally uses the Context API to pass data down the hierarchy. The ApolloProvider component takes one property which is the instance of the Apollo Client for the application. The ApolloProvider wraps the UsersPane component in the ReactDOM.render method to make it accessible to all components.

To pass the static data down the component tree a new context object was initialized and exported in a separate file. It was initialized by using the createContext function available in the react library and contains a context provider and a context consumer. In the index component the instance of the context was imported and the context provider wraps the UsersPane component in the same way as the ApolloProvider. The context provider takes a property named value which was set to an object containing the static data. To access the static data in the UsersListHeader, UsersListRow, UsersToolbar and CreateOrUpdateModal components the useContext hook was used which accepts the context instance as a parameter.

To make local and remote queries and mutations using the Apollo Client the useQuery and useMutation hooks were used in the components available from the @apollo/react-hook library. The queries for fetching and updating data from the Canvas LMS GraphQL API were specified and exported from two additional files.
The remote data was fetched from the Canvas LMS API GraphQL endpoint available at https://<canvas-instance>.instructure.com/api/graphql. As stated in the official API documentation for Canvas LMS [6]: “Fields are being added to the GraphQL API on an as-needed basis. The GraphQL API does not include everything that is currently in the REST API.”. Therefore there are a few things missing for the users’ search page to work using the GraphQL API right away. The GraphQL API should have had provided an Account node containing a usersConnection with a filter where the search term, sorting, ordering, and role filter id could be specified as well as having paging functionality. The current GraphQL API contains an Account node where it is possible to specify the account id and a userConnection having paging functionality but these are not connected and the userConnection lacks filtering options.

This problem was solved in this thesis by instead of communicating straight with the Canvas LMS REST API an Apollo Server is used that wraps the REST API into a GraphQQL API to get the missing functionality using the REST API but still be able to consume GraphQL from the client. Because this study focuses on measuring maintainability regarding architectural patterns for client-side data flow this will not impact the results and is not relevant for the research questions. Therefore the implementation details for the Apollo Server have been left out.

Evaluation

The data flow of the application is shown in Figure 3.8 on page 56. Components can now get data using the useQuery hook and update data using the useMutation hook both provided by the Apollo Client module. The data flow for getting and updating data from within a component (View) was illustrated in Figure 2.15 on page 33. The calculation for the Design Complexity of the Apollo Client and Context API implementation was conducted manually and resulted in the values shown in Table 3.9.

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of $v_m/(f_m+1)$</td>
<td>12</td>
</tr>
<tr>
<td>Sum of $f_m$</td>
<td>24</td>
</tr>
<tr>
<td>Number of modules ($n$)</td>
<td>11</td>
</tr>
<tr>
<td>Structural complexity ($S$)</td>
<td>2.18</td>
</tr>
<tr>
<td>Local complexity ($L$)</td>
<td>1.09</td>
</tr>
<tr>
<td>Design complexity ($DC$)</td>
<td>3.27</td>
</tr>
</tbody>
</table>

Table 3.9: Design Complexity measures for Iteration 2
3.4. Design Science Research approach

![Data flow graph for Iteration 2](image)

Figure 3.8: Data flow graph for Iteration 2
3.4. Design Science Research approach

The implementation has 11 modules which were one more than the initial implementation, fan-outs resulted in 12 which was 2 more than the initial implementation and for fan-out squared the value became 24 which was 4 more than the initial implementation. Due to the increased number for fan-out squared the structural complexity was increased by 9% to a value of 2.18. Compared to the initial implementation of the application the number of input/output variables were decreased from 80 to 24 which decreases the intramodule complexity of the system by 68.22% with a value to 1.09. The resulting Design Complexity value becomes 3.27 which is a 39.78% decrease of complexity compared to the initial implementation.

The Apollo Client and Context API implementation resulted in 18 files instead of the initial 22. To keep the number of files consistent 4 empty files were added having the complexity of 0. By using Plato and the customized script for calculating the average values for Software Science Effort, Lines of Code and Cyclomatic Complexity the values in Table 3.10 were obtained.

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct operators ($n_2$)</td>
<td>12.05</td>
</tr>
<tr>
<td>Total operators ($N_2$)</td>
<td>80.73</td>
</tr>
<tr>
<td>Distinct operands ($n_1$)</td>
<td>41.82</td>
</tr>
<tr>
<td>Total operands ($N_1$)</td>
<td>94.91</td>
</tr>
<tr>
<td>Vocabulary ($v$)</td>
<td>53.86</td>
</tr>
<tr>
<td>Observed Length ($N$)</td>
<td>175.64</td>
</tr>
<tr>
<td>Volume ($V$)</td>
<td>1180.21</td>
</tr>
<tr>
<td>Difficulty ($D$)</td>
<td>14.06</td>
</tr>
<tr>
<td>Effort ($E$)</td>
<td>36708.07</td>
</tr>
<tr>
<td>Time ($T$)</td>
<td>2039.34</td>
</tr>
<tr>
<td>Delivered bugs ($B$)</td>
<td>0.39</td>
</tr>
<tr>
<td>Lines Of Code (LOC)</td>
<td>72.00</td>
</tr>
<tr>
<td>Cyclomatic Complexity (CC)</td>
<td>5.68</td>
</tr>
</tbody>
</table>

Table 3.10: Software Science Effort, LOC and CC measures for Iteration 2

The Software Science Effort metric resulted in a value of 36708.07 this is a decrease of 31.80% compared to the initial implementation. The estimated average time required to implement a file equals 2039.34 seconds which is also a 31.80% decrease. The number of average estimated delivered bugs per file resulted in 0.39, which is a decrease of 23.53%.

The number of lines of code resulted in a value of 72.00 which is a decrease of 23.52% from the initial implementation. The Cyclomatic Complexity for the source code resulted in a value of 5.68 was decreased by 21.87% from the initial implementation.

Using static code analysis showed that the hypothesis was correct, the source code complexity and size was improved. The result of the Design Complexity metric was not satisfying this was due to the high value of structural complexity which is caused by high fan-out values. In more detail, it is caused by the modules UsersToolbar and CreateOrUpdateUserModal both having a fan-out value of 2 while in the initial implementation had fan-out values of 1 respectively 0. This due to that these modules are dependent on both the Apollo Client module and the Context module. This issue will be addressed in the next iteration.

Iteration 3 - GraphQL and Apollo Client Cache without React Context API

Awareness of problem

As noted in the evaluation of Iteration 2 the architecture using Apollo Client and Context API for the data flow of the application resulted in an increase of the coupling between modules. This is caused by the components UsersToolbar and CreateOrUpdateModal being dependent on both the Context module and the Apollo Client module.
Suggestion

By changing the architecture and use the Apollo Client cache as the single source of truth for all application data, including static data, the coupling for the application will be decreased and in turn the Design Complexity. To access the static data from within the Views, new queries have to be defined in the client schema.

Development

Firstly, in the index component the static data permissions, rootAccountId, accountId and roles previously used as value property to the Context API provider were stored in the InMemoryCache using the writeData method in the same manner as the searchFilter and errors. The context instance was removed as well as the context provider and the useContext hook in the components UsersListHeader, UsersListRow, UsersToolbar, and CreateOrUpdateModal. To be able to access the static data stored in the Apollo Client cache, type definitions for those queries were created as well as defining the fetch queries. GET_ERRORS, GET_SEARCH_FILTER, GET_PERMISSIONS, GET_ROOT_ACCOUNT_ID, GET_ACCOUNT_ID, and GET_ROLES queries are created querying the static data from the cache using the @client decorator. The static data was accessed from within the Views using the useQuery hook provided by the @apollo/react-hooks as for the other data.

Evaluation

By using Apollo Client Cache as a single source of truth results in a similar architecture as iteration 1 with a few exceptions. Instead of dispatch, useMutation is used and instead of state, useQuery is used. This resulted in the same values of fan-out for the modules, the same amount of input/output variables and the same number of modules in the system. Therefore the Design Complexity resulted in the same value as for iteration 1. The data flow for getting and updating data using the useMutation and useQuery hooks from within a component (View) is the same as for the previous iteration and was illustrated in Figure 2.15 on page 54. Calculations was made manually using the data flow graph shown in Figure 3.9 and resulted in the Design Complexity values shown in Table 3.11.

| Sum of \(v_m/(f_m+1)\) | 11.5 |
| Sum of \(f_m^2\) | 18 |
| Number of modules \((n)\) | 10 |
| Structural complexity \((S)\) | 1.8 |
| Local complexity \((L)\) | 1.15 |
| Design complexity \((DC)\) | 2.95 |

Table 3.11: Design Complexity measures for Iteration 3
Figure 3.9: Data flow graph for Iteration 3
By removing the Context definition file the number of files became 17, to keep the number of files consistent 5 empty files with complexity 0 was added before performing the static code analysis. Using Plato to obtain the source code complexity and size using Software Science Effort, Lines of Code and Cyclomatic Complexity resulted in the values shown in Table 3.12.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct operators ($n_2$)</td>
<td>11.77</td>
</tr>
<tr>
<td>Total operators ($N_2$)</td>
<td>81.59</td>
</tr>
<tr>
<td>Distinct operands ($n_1$)</td>
<td>42.09</td>
</tr>
<tr>
<td>Total operands ($N_1$)</td>
<td>95.45</td>
</tr>
<tr>
<td>Vocabulary ($\eta$)</td>
<td>53.86</td>
</tr>
<tr>
<td>Observed Length ($N$)</td>
<td>177.05</td>
</tr>
<tr>
<td>Volume ($V$)</td>
<td>1190.20</td>
</tr>
<tr>
<td>Difficulty ($D$)</td>
<td>13.90</td>
</tr>
<tr>
<td>Effort ($E$)</td>
<td>37071.94</td>
</tr>
<tr>
<td>Time ($T$)</td>
<td>2059.55</td>
</tr>
<tr>
<td>Delivered bugs ($B$)</td>
<td>0.40</td>
</tr>
<tr>
<td>Lines Of Code (LOC)</td>
<td>73.73</td>
</tr>
<tr>
<td>Cyclomatic Complexity (CC)</td>
<td>5.68</td>
</tr>
</tbody>
</table>

Table 3.12: Software Science effort, LOC and CC measures for Iteration 3

The Software Science Effort resulted in a value of 37071.94 which is a decrease of 31.13%, correlated to the Software Science Effort the average estimated time to implement a file was 2059.55 seconds which is also a 31.13% decrease and the estimated average number of delivered bugs results in a value of 0.40 which is an decrease of 21.57%. The measure for Lines Of Code resulted in a value of 73.73 lines which is a decrease of 21.68% from the initial implementation and cyclomatic complexity of 5.68 which is a decrease of 21.87%.

The source code complexity and size was increased slightly compared to the previous artifact due to the additional functionality needed for storing and fetching the static data from Apollo Client Cache. The structural complexity was decreased and therefore the Design Complexity as well. The results of this artifact had satisfying results with improved Design Complexity as well as improved source code complexity and size compared to the initial implementation. No more architectural improvements could be found and the most commonly used architectural patterns for React have been studied. The results will be presented in the following chapter and later used to answer the research questions.
By changing the design of the application using architectural patterns the overall maintainability was improved with a decrease of design complexity and source code complexity.

Using the react-redux library in Iteration 1 resulted in an improvement of the Design Complexity by 45.67% but the source code complexity remained high. This was addressed in the second iteration. The source code complexity and size were improved in Iteration 2 by using the GraphQL API for Canvas LMS which made fetching and updating data more flexible by making specified queries for data instead of getting a fixed chunk of data from the REST API. To consume the GraphQL API, Apollo Client was used together with the InMemoryCache which stored local and remote data. This replaced the Redux store used in Iteration 1 and all custom fetching logic and Redux specific code which resulted in an improvement from the initial implementation of the Software Science Effort by 31.80%, the Lines of Code by 23.52% and Cyclomatic Complexity by 21.87%. Sharing static data between components was made with React Context API which led to increased coupling by 9% compared to the initial implementation. To cope with this in Iteration 3 static data was stored inside the Apollo Client Cache and React Context API was removed. This led to a small increase in source code complexity compared to iteration 2 because of the queries and type definitions required but the coupling was decreased by 10% from the initial implementation.

The largest percentage improvement was made for cohesion which was improved by 66.47% in the final implementation. The issue in the initial implementation was a large number of input and output variables passed through the component tree. Instead by storing data in a global Apollo Client Cache accessible from all components using React Hooks this was improved. The new architecture led to less coupling between the modules as well and the Design Complexity metric was improved by 45.67%.

In Table 4.1 on page 62 the results of the Design Science Research effort are summarized. Each measure having a percentage value showing the difference to the initial implementation of the study object. All calculations and details for the initial implementation and each iteration can be found in Section 3.4 and Section 3.5. 
<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Itr. 1</th>
<th>Itr. 2</th>
<th>Itr. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural complexity</strong></td>
<td>2</td>
<td>1.8</td>
<td>2.18</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-10.00%</td>
<td>+9.00%</td>
<td>-10.00%</td>
</tr>
<tr>
<td><strong>Local complexity</strong></td>
<td>3.43</td>
<td>1.15</td>
<td>1.09</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-66.47%</td>
<td>-66.22%</td>
<td>-66.47%</td>
</tr>
<tr>
<td><strong>Design complexity</strong></td>
<td>5.43</td>
<td>2.95</td>
<td>3.27</td>
<td>2.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-45.67%</td>
<td>-39.78%</td>
<td>-45.67%</td>
</tr>
<tr>
<td><strong>Software science effort</strong></td>
<td>53826.98</td>
<td>53243.40</td>
<td>36708.07</td>
<td>37071.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.08%</td>
<td>-31.80%</td>
<td>-31.13%</td>
</tr>
<tr>
<td><strong>Halstead’s time</strong></td>
<td>2990.39</td>
<td>2957.97</td>
<td>2039.34</td>
<td>2059.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.08%</td>
<td>-31.80%</td>
<td>-31.13%</td>
</tr>
<tr>
<td><strong>Halstead’s delivered bugs</strong></td>
<td>0.51</td>
<td>0.5</td>
<td>0.39</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.96%</td>
<td>-23.53%</td>
<td>-21.57%</td>
</tr>
<tr>
<td><strong>Lines of code</strong></td>
<td>94.14</td>
<td>91.36</td>
<td>72.00</td>
<td>73.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-2.95%</td>
<td>-23.52%</td>
<td>-21.68%</td>
</tr>
<tr>
<td><strong>Cyclomatic complexity</strong></td>
<td>7.27</td>
<td>7.32</td>
<td>5.68</td>
<td>5.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+0.69%</td>
<td>-21.87%</td>
<td>-21.87%</td>
</tr>
</tbody>
</table>

Table 4.1: Results of the Design Science Research effort
In this chapter, the results and the method used will be discussed. Furthermore, source criticism, as well as the ethical and social aspects of this thesis will be discussed in the final two parts of this chapter.

5.1 Results

Architectural patterns used in React has been implemented and evaluated using the iterative Design Science Research process. Starting with the built-in architecture in React using a unidirectional data flow with props-drilling which was shown to have a negative impact on especially the cohesion of the application. This was because a large number of input and output variables passed several levels in the component tree. For smaller applications, the amount of data that flows in the application is less and the component tree is not as deep. Because implementing the architectural patterns used in this thesis requires additional source code complexity the results for small applications are not necessarily the same.

In the second iteration, the library react-redux was introduced which allowed data to be passed directly to the React Views from a global Redux store. This approach decreased the cohesion and coupling of the application resulting in a decrease in the Design Complexity. Using Redux and the react-redux library led to high source code complexity and was addressed in the third iteration replacing Redux and react-redux with Apollo Client and Context API. Apollo Client uses built-in functionality for fetching data from the GraphQL API, it also requires less code to configure and less code to interpret the API responses. This led to improved source code complexity measured using Software Science Effort, Lines of Code and Cyclomatic Complexity.

Using Context API to pass static data down the component tree had a negative effect on the coupling of the application due to that some React Views were dependent on both the Apollo Client module and the Context API module. This issue was addressed in the final iteration where Apollo Client Cache was used as a "single source of truth" managing all data. If every measure has the same priority Iteration 3 gives the best overall maintainability. Depending on the priorities these can be weighted differently and for some scenarios, the architecture of Iteration 2 may be more suitable than the architecture of Iteration 3 because the cohesion and source code complexity was slightly lower.
What stands out in the results apart from the large differences in maintainability is that the most popular architectural pattern in React which is Redux are less maintainable than the second most popular alternative which is Apollo Client. This may have to do with that Redux existed for a longer time and has become the standard approach when building complex applications in React. Another reason could be that Redux is often used together with a REST API on the back-end and if more APIs start adapting the GraphQL technology Apollo Client may become more popular.

5.2 Method

The method used to implement and evaluate the maintainability of different architectural patterns was the Design Science Research process. This approach was suitable for studying the benefits and issues for each of the patterns by improving the application iteratively. The study object was initially implemented using props-drilling which had some negative side-effects. If the study object chosen would have been implemented using the react-redux library right away the process would not have covered the issues of using props-drilling. Therefore the choice of the study object was of great importance.

Furthermore by choosing a study object used by thousands of people every day built by experienced software engineers the study has more credibility than if all work was done by a single person. The extensive theory behind each architectural pattern was necessary to fully understand each pattern and how they would be implemented. Even though the guidelines for implementing each pattern were followed carefully, some implementation details will vary depending on the person implementing it and the complexity of the source code will differ slightly. This will not affect the conclusion drawn from this study because the final results show large differences depending on the architectural pattern used.

In the paper “Software Metrics for Predicting Maintainability” [15] the authors suggested fourteen metrics that had good correlation to maintainability and had been validated through experimental results published by researchers. Out of these fourteen, six of them were used in this thesis where cohesion and coupling were part of the Design Complexity metric. These metrics were chosen because they were the most relevant when studying the architecture of an application and they were also supported by Plato.

Using Plato with a custom script enabled measurements of the source code complexity to be done automatically. For design complexity calculated using Design Complexity of Card and Agresti, no tool exists that automates the measurements for React applications. This lead to that manual calculation was necessary that cannot be automated into the developing process. As mentioned in Section 2.2 Plato uses a library named escomplex to do measurements based on the abstract syntax tree (AST) of an application. The AST could be used to automate the measurements for design complexity as well but requires a lot of work and was out of the scope for this thesis.

Due to the fundamentals of how React applications are built it is recommended to use unidirectional data flow [36] and therefore the most popular unidirectional architectural patterns were evaluated in this thesis. The popularity was based on the number of downloads from the NPM Registry which is the standard approach for getting third-party libraries. Even though third-party libraries can be imported using other methods this is the most common way which makes the statistic reliable.

The whole method process or parts of it can be applied in other applications and would give similar results, at least for application with similar size or larger as discussed in Section 5.1. When applying the method on other applications the iterative Design Science Research process may not be necessary, instead, the architectural pattern used in the final iteration could be applied right away and measurements can be performed using the metrics used in this thesis.
5.3 Source criticism

The sources of this thesis have mainly been found using the web library of Linköping University as well as Google scholar. Well cited literature is a good indication of a reliable source and therefore this has been taken into account when choosing the sources. Software quality and software quality metrics are well-studied topics with a lot of research and publications made. The metrics used in this thesis were chosen based on the relevance for maintainability as well as the validity and reliability of the metric.

Software Science Effort suggested by M.H. Halstead in the article "Elements of software science" [20] is a well-established metric that has been cited 3984 times. Another source code complexity metric used was Cyclomatic Complexity published by T.J. McCabe in the article "A Complexity Measure" [27] which has 7201 citations. The metric Lines of Code were suggested in the book "Software engineering metrics and models" [9] by S. D. Conte, H. E. Dunsmore, and Y. E. Shen which has been cited 1845 times. Furthermore the article "Measuring Software Design Complexity" by D.N. Card and W.W. Agresti suggesting the Design Complexity metric has been cited 159 times. These metrics have later been validated for relevance for maintainability in the study “Software Metrics for Predicting Maintainability” [7] by M. Frappier, S. Matwin and A. Mili.

For React, which is a quite new technology, only a few books are available covering the concepts of it and even less regarding architectural patterns used. Therefore the official React documentation has been a great resource to grasp the fundamentals of React as well as forums discussing different architectures used. Using statistics from NPM the popularity of different techniques and architectural patterns could be verified.

5.4 The work in a wider context

The work done in this thesis has no direct impact on any people but applying the results on a web application will impact everyone involved with it. Due to improved maintainability, the end-user will have better accessibility to the application because it will contain fewer bugs and bugs would be faster to resolve. The complexity of the application is decreased and becomes easier to maintain by the software engineers. This will, in turn, lead to more resources to build other applications which will be beneficial for the software organization building the applications. One drawback of this could be the decreased demand for software engineers. Furthermore, because this thesis suggests that a single architectural pattern gives the best maintainability other architectural patterns and libraries implementing these may become less popular and people profiting from these will be affected negatively.
In this thesis, metrics have been investigated that measure maintainability for architectural patterns and how these measurements can be automated using static code analysis. The measurements have then been used to perform a quantitative comparison between the most commonly used architectural patterns for building web applications using React. The results can be used to gain knowledge of how different architectural patterns perform in terms of maintainability and make design decisions based on them when building or refactoring web applications. Furthermore, the metrics and the static analysis tool used in this thesis can be used in other studies for measuring architectural maintainability.

The research questions formulated in Section 1.3 will be answered in the next section followed by a section covering future work.

6.1 Research questions

1. Which metrics are suitable for measuring maintainability regarding the architecture of web applications, and can they be automated into the developing process?

The choice of the architectural pattern used in a web application affects both the design complexity as well as the source code complexity which both have a great impact on maintainability. Thus, metrics measuring these two types of complexity are relevant when measuring the maintainability of architectural patterns. In this thesis Design Complexity of Card and Agresti, Software Science Effort, Cyclomatic Complexity, and Lines of Code has been used. These metrics have been validated and are well-established in the field of software science. Measurements for source code complexity can be automated using the static code analysis tool Plato. Plato works for web applications written in Javascript and supports the JSX syntax used in React applications. Furthermore, Plato can be executed with customized scripts using Node which enables it to be automated into the build process of the application.

2. What architectural patterns exist for building web applications using React, and which alternative gives the best effect considering maintainability?

React applications consist of a composition of React components, the Views of the application. The components pass data downwards through the component tree using props which is the built-in unidirectional architecture for React applications. For more
complex React applications an architectural pattern is necessary to improve the maintainability of the application. Flux is one architectural pattern used in React that has a similar architecture as MVC with a few improvements to make it more predictable. Additional architectural patterns have evolved from Flux and libraries such as Redux and Apollo Client implement these. This research concludes that the architectural pattern used for building the most maintainable React applications is implemented using Apollo Client.

6.2 Future work

As discussed in Section 5.1 the size of the application may affect how maintainability change for different architectural patterns. Therefore it would be interesting to apply the same method on different application sizes to investigate when using an architectural pattern is most beneficial. The hypothesis being that larger applications benefits more than smaller applications. Another interesting aspect is how other architectural patterns that are not unidirectional, like MVVM, would perform in comparison to the ones studied in this thesis. Finally to be able to automate all measurements, calculating design complexity using a static analysis tool would be necessary and would be interesting to study further.
Bibliography


