Bachelor/Master Thesis Project

An Investigation of Data Flow Patterns Impact on Maintainability When Implementing Additional Functionality

Author: David Grenmyr, Erik Magnusson
Supervisor: Tobias Ohlsson
Semester: VT/HT 2016
Subject: Computer Science

Abstract

JavaScript is breaking ground with the wave of new client-side frameworks. However, there are some key differences between some of them. One major distinction is the data flow pattern they applying. As of now, there are two predominant patterns used on client side frameworks, the Two-way data flow pattern and the Unidirectional data flow pattern.
In this research, an empirical experiment was conducted to test the data flow patterns impact on maintainability. The scope of maintainability of this research is defined by a set of metrics: Amount of lines code, an amount of files and amount of dependencies. By analyzing the results, a conclusion could not be made to prove that the data flow patterns does affect maintainability, using this research method.

Keywords: JavaScript, Framework, Client side, Unidirectional data flow, Two-way data flow, Data Flow Pattern, React, Flux, Angular 2 , Ember, Vue

Contents

1 Introduction

The range of JavaScript frameworks has increased over the past years [1]. JavaScript frameworks are used to develop anything from simple and complex web pages to mobile apps, forcing the frameworks to be versatile [2]. Two of the most common data flow patterns used in JavaScript frameworks are the two-way data binding and the unidirectional pattern.

The unidirectional data flow can be visualized as a circle, where the data flows in one direction through the application. The responsibility of updating the applications state is focused at one location.
On the other hand, a two-way data flow may transport the data in different directions and may also change the state in all locations where a reference to the data is used.

Our thesis means to test that the choice of data flow pattern could influence the maintainability of a system.

1.1 Background

Early client-side JavaScript framework, such as backbone, Ember, Vue and Angular, all shares the fact that they apply the two-way data flow pattern. However since 2013, the year Facebook presented React, one can see a change in the market. Angular started the Angular 2 project in 2014, and Ember published Ember 2.0 in 2015, both of which embraces a one-way data flow pattern, often described as a unidirectional data flow [3][4][5]. The fact that popular frameworks are embracing the unidirectional data flow in their newer releases inspired us to want to investigate if the data flow pattern of a JavaScript framework could have an impact on the maintainability of a system or a project.

To be able to go through with this experiment a deep dive into each framework was required. An evaluation will be conducted to ensure that the best practices are followed and that functionality that each framework provides is used. Furthermore, the concept of maintainability in the scope of this research needed to be defined.

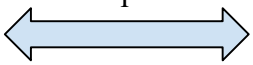
1.1.1 Two-Way Data Flow contra Unidirectional Data Flow

The main difference between the patterns is defined by where the responsibility for updating the state in the application is located and direction of which the data may be directed. The two-way data flow may have, for example, a Model and a user interface. Both of them may have its own instance of the state value and both have the ability to change the value. When a state is updated, the other components updates dynamically.

However, in a unidirectional pattern, the responsibility for changing the value remains in one place, and all parts have a reference to this value.

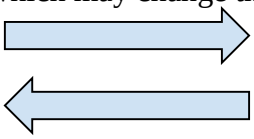
The unidirectional data flow could also be seen as a UI component with a subscription to one state and on an update, the application publishes the new value causing it to re-render the UI components. Where the two-way data flow can subscribe and publish to any component to update the state.

The examples of the data flow of an application are portrayed with arrows in the framework's chapter, where two-way data flow is described with a two-edged arrow. The meaning of this is that both of the components have the ability to access and manipulate each other's values.



Example: Suppose there are a Model component and a View Component. The Model component holds a "user" object, with a "name" property. If a new value is assigned to the property via the Model, this value would be updated on the View as well. But the View also owns the property value, if it were to be changed the model would also change its value according to the Views property value.

Where the one edged arrows means that the components have the ability to access references of data from each other. For example, the View may only call functions on the Model which may change the state.



Example: A View component holds a reference to a value which is stored somewhere else. If there is an intent to update a value, the intent would be sent to a function that performs the sought after change. When the View component notices that the application state has changed, it re-renders

1.1.2 Maintainability

Maintainability is an abstract and wide concept. There are many metrics, techniques, and properties that can be used to measure maintainability of code. Maintainability may also be completely different if the abstraction level is high or low [6].

In our research, it is necessary to find the definition of maintainability that can be measured by code churn [7]. Previous researches and come to the conclusion that a relative increase in code churn activities is accomplished by negative affect on maintainability [8][9].

Faragó, Csaba, Hegedűs, Péter, and Ferenc, Rudolf also come the conclusion that when comparing code churn activities, file additions within a commit has a positive effect on maintainability, updates has a negative effect on maintainability and delete has no significant effect regarding maintainability. Because of this, delete metric was not included in our code churn, only additions and updates were measured.

1.1.2.1 Defining maintainability

We define and measure maintainability using the following metrics:

1. Files added or updated.
2. Lines of Code within each file that are added or updated.
3. Amount of dependencies in each implementation with a relationship to the login state and function calls to the login and logout functions.

We decided to analyze the number of dependencies within each file is because it is a very important attribute of maintainability [10] [11]. The reason that the login state and the login and logout functions are defined as dependencies are because that the data flow patterns are dependent on them to handle the implemented authentication for the applications. Both a number of dependencies and their locations are important attributes when measuring maintainability [11].

1.1.2.2 Why maintainability is a concern

Maintainability is a widely observed and examined topic. Maintainability has a large effect on the development efforts and also a direct connection to resource costs. The importance of maintainability, therefore becomes an important concern for stakeholders when evaluating existing or future projects**.**

1.2 Previous research

Previous researches are focused on two main subjects, code churn- and maintainability. The analyzed researches helped us to get a perspective of which metrics that affects maintainability. We could also establish that file additions, updates or deletions have different impacts on maintainability.
C. Farago et al. [10] perform a code churn analysis on four software systems. The research analyses git commits on the four projects and extract the changes between each commit. The metrics are lines added, lines edited and lines created. By harvesting mentioned metrics the authors analyze it from a maintainability perspective. The research method has greatly inspired our method.

Faragó et al. [9] come to the conclusion that code churn generally has a negative impact on maintainability. However, the authors also come to the conclusion that not all code churn operations have a negative effect. File additions have a positive impact, file updates show a negative impact and delete no significant difference on maintainability.

N. Nagappan et al. [8] base their research on a code churn from Windows 2003 SP1. The authors experiment with different metrics to predict the system defects. The research comes to the conclusion that an increase in relative code churn measures increases the system defect density.

T. Bakota et al. [6] intend to analyze maintainability from two different perspectives. The first experiment is a low-level analysis of maintainability on 100 open source projects. While the second experiment focuses on maintainability analysis on two large systems from a high-level perspective. It is done so by combining metric properties from the first and second experiment. The result is then evaluated by a council of professional developers which give opinions of the maintainability.

1.3 Problem formulation

From a maintainability point of view, choosing a pattern can influence a whole project. Fewer dependencies may mean faster performance, but at a cost of more time spent implementing it.

Are there certain benefits to gain from choosing data flow patterns when developing a system or an application from a maintainability point of view? What are the benefits of the certain data flow pattern and what context would it be fitting for, from a system developer or project leaders point of view?

1.4 Motivation

Little research exists in the area of what the differences in implementation between the two patterns with JavaScript frameworks. This inspired us to investigate if it can, in fact, make a difference for a developer to take the data flow pattern of a framework into consideration when choosing a development procedure for a project.

Maintainability is an often sought after attribute in a system. We want to test if one can present data that point to the fact that a certain data flow pattern may affect the maintainability, it would be of great help to developers.
The results would be able to help the development to better suite their maintainability needs.

1.5 Research Question

RQ1	Are there differences between the two-way data flow pattern and the unidirectional data flow pattern from a maintainability perspective in JavaScript frameworks?

To be able to answer this question we had to specify the scope of maintainability. With maintainability defined we were able to construct three sub-questions (**RQ1.1, RQ1.2, RQ1.3**) which, when put together, would be able to answer **RQ1** as a whole.

RQ1.1. What *are the maintainability differences between the data flow patterns when considering the implementation number of code lines?*
RQ1.2. What *are the maintainability differences between the data flow patterns when considering the implementation of number of files?*
RQ1.3. What *are the maintainability differences between the data flow patterns when considering the implementation of number of dependencies?*

1.6 Scope/Limitation

Our limits mainly lie with the scope of time that the research needs to be executed within. We have selected a set of, what we deem to be the prominent framework.

1.7 Target group

From every day- to professional developers and architects or companies that are considering any of the data flow pattern for their upcoming projects. The results would also be able to provide an insight into maintainability costs for project leaders and other parties that are not as technically knowledgeable.

The results may be able to guide the reader to what pros and cons the data flow pattern may bring to their respective case.

2 Theory

2.1 Frameworks

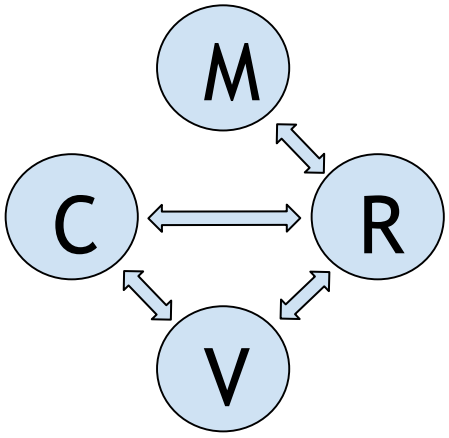
A framework is a software developing tool to provide generic functionality and solutions for developers [12]. The benefits of using frameworks are that it makes use of common principles that helps the development. Some the principles are modularity, reusability, extensibility, and inversion of control. It is common for frameworks to reuse common software development strategies with the focus on concrete designs, algorithms and implementations in particular languages. Frameworks also have tendencies to be less tightly coupled. For example, applications can reuse components without having to subclass from existing base classes[16].

2.1.1 Ember.js (1.10.0)

Ember has, until recent days been a solely supporting two-way data flow. But development has split into two individual branches. They are divided into the Ember 1.X and the Ember 2.X branches. Where the 2.X branch is influenced by the unidirectional data flow pattern. For this implementation, we will be using the main 1.X branch, which still practice two-way data flow.
From here on we will refer to it plainly as *Ember*.

Ember implements a two-way data flow between:

- Templates/Views and Controllers
- Controllers and the Router
- Router to the Models

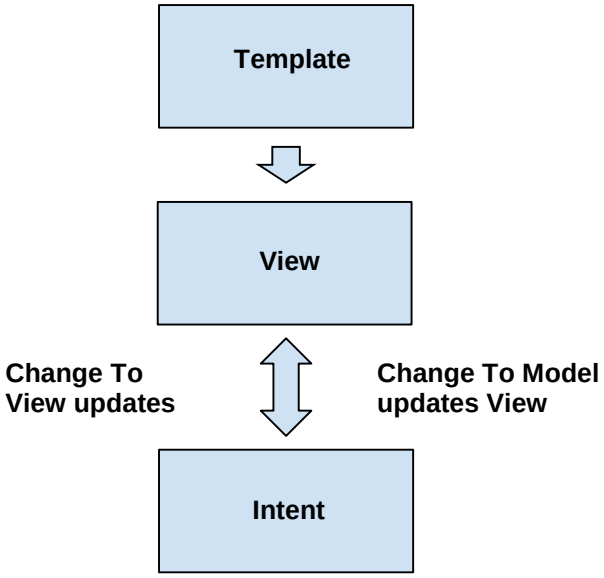


The dependencies and data flow are centralized at the Router, which holds the routes of the application, state and also updates the state or the model based on user interactions.

Ember applies the *convention over configuration* software design paradigm, meaning that they seek to reduce the amount of decisions a developer needs to make. However, *convention over configuration* may result in loss of flexibility when the convention is not applicable in certain scenarios.

2.1.2 Vue.js (1.0.1)

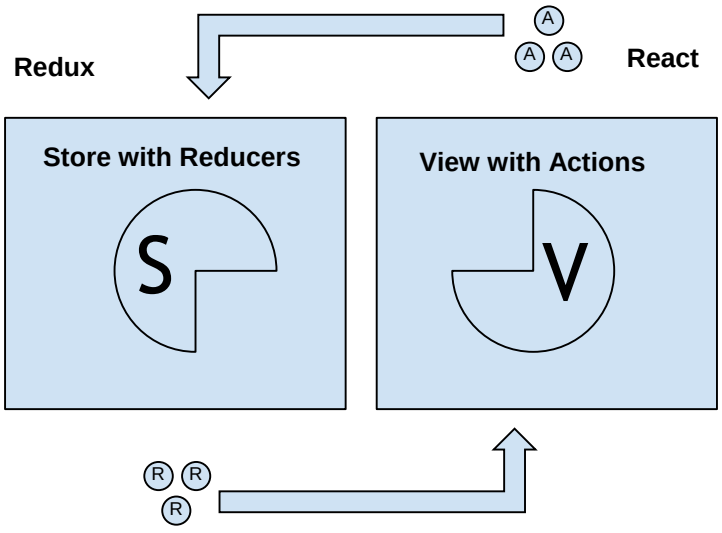
Compared to Ember, which is a full-featured framework with strict conventions, Vue offers a lot more flexibility. In theory, Vue consists of a View, a ViewModel and a Model. The View presents reactive data which are set in the ViewModel. The ViewModels are allowed to change data value, which in turn updates the Model value of the object, and vice versa for the Model.



What makes Vue interesting is that it could be used with a unidirectional pattern. But the documentation strictly applies the two-way data flow standards [13].

2.1.3 React.js + Flux (0.14.7)

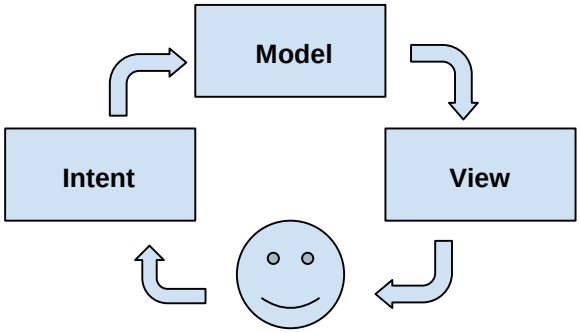
React is a framework for building user interfaces. It can be used to define the V in an MVC model, thus it handles the states provided without making any assumptions about the rest of the application. However, React is often paired with a state container solution for it to be more scalable and user-friendly in the long run. Flux provides just that for React. Actions are functions, called from the view components, which performed a task. When the task is done, it tells the reducers to update the state of the application with this update.



One can clearly see the difference of the data flow when comparing them to Ember and Vue, where the two-way data flow based frameworks use two-way data flow, the unidirectional pattern flows in a circular one direction motion to receive, handle and serve the data.

2.1.4 Angular 2 (2.0.0-beta.0)

The user gets presented with a view which provides, so-called intentions. When such an event is fired, there is an intent layer, much like Vues view model, which handles the input, it can inform the model what is supposed to update. When the Model has performed its change, the View re-renders. This way the intention layer may only inform the Model of what it should update and ask for the current values. The intentions may not change the value of the current state itself.



2.2 Designs for Maintainability

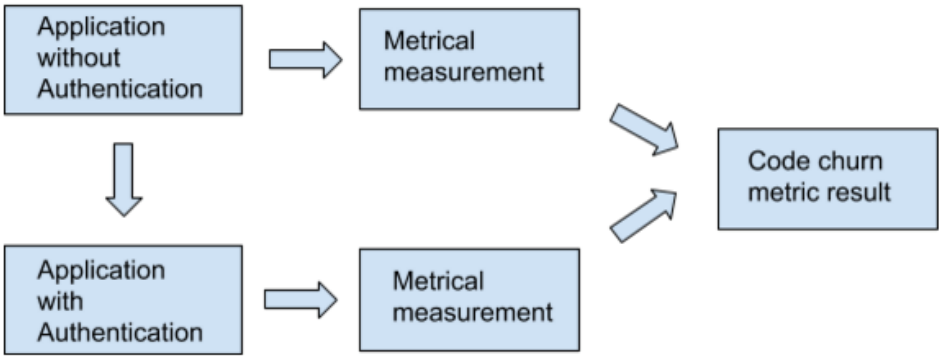
Cohesion and coupling are affected by multiple parameters. The coupling may be effected by the data flow patterns separation of responsibilities [17]. There are some differences in how unidirectional data flow and two-way data flow practice this. Unidirectional data flow updates all states at one location, therefore the coupling may be looser. Cohesion is harder to distinguish between the data flows. Frameworks standards as abstraction level, convention over configuration or the size of the frameworks may affect cohesion.

2.3 Measuring Maintainability

In order to measure maintainability, authentication functionality will be added to the set of frameworks. A code churn will then be performed analyzing the differences. The metrics that will be measured are amount of files added, lines of code added and amount of dependencies. By selecting these metrics an analysis of cohesion and coupling may be performed. This will in turn, hopefully, lead to a conclusion in how maintainability is affected by the two data flow patterns.

3 Method

The method measures the difference of each metric of the application in two steps. First step all metrics selected to calculate maintainability is measured before any changes are made to the applications. Secondly, the implementation is performed, adding login functionality to the set of applications. After implementation is done, the applications are once again measured by a code churn analyzing the metrics.



3.1 Scientific Approach

To test the thesis an empirical experiment is performed to collect quantitative data in an inductive approach. The data comprises three metrics and are collected from implemented functionality in a set of JavaScript frameworks applying the different data flow patterns. We observe and measure the data both manually and with help of git-diff to be able to analyze the results and find their impact on maintainability [14].

3.2 Method Description

The method conducted to answer the thesis will be done by implementing authentication functionality into already existing todo applications.

We have created a layer which simulates a database request with a login and logout functionality that will be used by the different frameworks.s

Based on previous analyzes we have selected a set of source code metrics:

- Amount of **rows** added/updated
- Amount of **files** added/updated
- Amount of **dependencies** dependent on the login state or the login and the logout functions.

3.2.1 Preliminary steps

The first phase takes place before the actual implementation. We analyze and measure the framework projects, which have been selected from the TodoMVC list of implementations, to ensure they are all containing the exact same functionality.

GitHub will be used for handling versions the project. The repository is public and can be found under the organization [15].

The version control will be essential in this method since it will show all the different results between the frameworks committed versions.

We have a set of rules that has to be followed continuously throughout all of the implementation phases. To assure the rules are followed we have been pair programming large parts of the implementations and inspected each other's implementations both separately and together.

- No white-spaces between rows
- ES5 standard
- No shorthanded statements

3.2.2 Method execution

Each framework instance will share the same login and logout functions that asynchronously return a true or false based on if authentication is correct. The code for this is located in the database.js file.

Before the method execution started the set of TodoMVC applications are added to version control. At this stage, they all share the exact same functionality.
The login functionality assures that an authentication is required to make use of these pre-existing features.
If a user is denied from authentication, the user will not be granted access to the TodoMVC functionality.

There will be no persistent data stored by the user. If a user is successfully authenticated and then reloads the page, the authenticated state will be lost.

3.2.3 Code churn

Code churning could be defined as the additions, modifications or deletions to a file or set of files from one version to another. This way of version

controlling helps to define assayable metrics to our research that has a direct relation to how two different versions of the same project are differing.

3.2.4 Calculation of code churn

The second phase takes place after the implementation is finished. We analyze and compare the data from the earlier analyses with the data from the updated version, which has the new functionality implemented.

All framework specific files will be excluded in the analysis since we are researching the parts which are of importance to the actual functionality and data flow pattern. Furthermore .css files will be excluded.

The extraction of data will be handled partly manually and by a help of the "gif-diff" measuring tool.

3.2.5 Lines of Code and Number of Files Added or Updated

To measure the lines of code and the amount of lines that has been handled, we have used the "git-diff"-tool [14]. This tool provides the functionality for counting what files has been handled during the implementation, and what has changed within the file depending on the options parameter provided to the git command.

3.2.5.1 Git Command Structure

git diff [options] <commit> <commit> [--] [<path>…]

3.2.5.2 Git Commando Examples

Example command input:

a) *git diff* ***--numstat*** *38f71b92c6b9bfcb84d5cda67333a9ac5d1c997b d974e4a62288bfeb3b8e16a8689b6bb9b4ffeca0 react-flux/js/*

b) *git diff* ***--shortstat*** *38f71b92c6b9bfcb84d5cda67333a9ac5d1c997b d974e4a62288bfeb3b8e16a8689b6bb9b4ffeca0 emberjs*

Example command output:

a)

```
23    0    react-flux/js/actions/UserActions.js
19    0    react-flux/js/actions/database.js
23    0    react-flux/js/components/AuthButton.react.js
22    0    react-flux/js/components/LoginForm.react.js
20    7    react-flux/js/components/TodoApp.react.js
5     0    react-flux/js/constants/UserConstants.js
43    0    react-flux/js/stores/UserStore.js
```

b) *4 files changed, 69 insertions (+), 1 deletion (-)*

3.2.6 Number of Dependencies present

The dependencies are counted manually since we have not had time to create a measurement tool which fits our definition of a dependency. There are two kinds of dependencies that are manually counted. The first dependency in our research is each time the application makes use of the login state value. The second dependency is each time the application use either of the login or logout functions.

3.3 Validity and Reliability

3.3.1 Internal validity risks

There are multiple internal validity risks. Firstly and perhaps the factor of highest risk is the different framework standards, for example, configuration over convention may affect how many files or a number of lines each set of framework need to implement for the same login functionality.

Although the framework instances share exactly the same functionality, they are implemented by different individuals. Which in turn may use different code standards that affect the difficulty for the researchers to implement the login functionality?

One of the greater internal risks is the fact that the researchers come from the same programming background, which may affect the implementation in certain ways. The same risk also applies to the researcher's view of the best practices of the frameworks. Based on the documentations we have implemented what we believe is the best practice for each framework. However, if a future attempt to conduct this method were made by developers with a different programming background, the implemented code may vary.

3.3.2 External validity risks

There is a risk that our simulation of login functionality does not represent a general case of how asynchronous request to a backend system to authenticate may behave.

Additionally, there is a risk that the scope of our research is too small, the set of implementations may be too small to represent a more general real case application. There is also a risk that the set of two frameworks implementing the unidirectional data flow respectively the two frameworks implementing two-way data flow pattern are too few to give reliable results.

3.3.3 Construct validity risks

Maintainability is an extensive area of subject and it is a risk that the metrics defining maintainability in this essay, as described in chapter 1.1.2, may not be the same definition as in other contexts.

3.3.4 Reliability risks

The research method has been produced to measure data flow patterns impact on maintainability. However, since the researchers only were able to comprise four frameworks into the experiment, combined with the quality of the pre-existing TodoMVC applications and the demography of the researchers, the results may not be reliable to the extent of making an assured conclusion.

3.4 Ethical Considerations

We are basing our research on pre-built applications of the frameworks. Even though they're under a free license, the people who have contributed to the sets of TodoMVC are listed in [Appendix C].

4 Implementation

This chapter will cover the implemented functionality in depth. This includes specific information about each implementation.

We have implemented the functionality from what we have concluded is the best practice for each framework. This is founded upon the frameworks documentation and the standard set by the TodoMVC application.

4.1 Preliminary steps

To set up a foundation for the actual implementation to be made there were some factors that had to be taken into consideration.

The choice of frameworks was made based on the frameworks ability to demonstrate the practice of its data flow pattern. It was to our belief that the choice of the framework would have little effect on the results since the goal of this method is to measure the data flow patterns. We did, however, look into the popularity of each framework to make sure that it had a well-based community to ensure the standards that can be requested from a framework.

To make the experiment generic the method easy for future utilizing. Authentication functionality was selected as implementation. It is well known to be used with many different programming languages, frameworks, use cases, and patterns.

To minimize external threats that might affect the experiment the best solution is to fake the database request and keep the applications state within the local environment.

4.2 The Functionality

The functionality will be simulating a login for the user. A simple *true* or *false* state that decides whether the user is authenticated to access the todo logic of the application.

We created a fake *database* to be able to simulate a server request, which authenticates the user's input. The file is called *"database.js"*. The reason this file is implemented is that the different frameworks have their own way of performing server requests to actual databases or API's. Since this could mean that the code may differ based on what framework is used, it could affect the results of the code churn.

4.2.1 database.js

The “database.js” file contains two functions, *login* and *logout*.
They both return a Promise, with a delay of 1000 milliseconds.
Promises help to handle an asynchronous action which is good since we are trying to simulate an actual server request.

Our initial thought was to use only one instance of the *database.js* file. But since the frameworks can’t operate outside the scope of their own root folder, we decided to keep one instance of the file within each of the frameworks.

```
'use strict';
module.exports = {
  login:function(username, password) {
    return new Promise(function (resolve, reject) {
      setTimeout(function () {
        if(username==="test" && password==="password"){
          resolve(true);
        }else {
          reject(false);
        }
      }, 1000);
    });
  },
  logout:function() {
    return new Promise(function (resolve) {
      setTimeout(function () {
          resolve(true);
      }, 1000);
    })
  },
};
```

4.3 EmberJS

Due to Embers TodoMVC application could not use "npm" lib “require” to import files, the same login and logout code, was instead added to an Ember model called “User”.

4.4 Vue

One of the functions *[File: Implementation/Vue/js/store.js, Rows: 29-31]* used in Vues store is never used but still got added to the version control system. The function spans over three rows which were excluded from the final result of a number of rows added.

5 Results

In this chapter, we will present raw data which have been retrieved from the implemented code. The data will be presented in tables divided by type of maintainability metric per framework and by which action that has been performed.

5.1 Table presentations

5.1.1 Lines

Framework	Lines Added	Lines Updated
Ember	68	1
React-Flux	154	7
Angular 2	88	31
Vue	65	2

Table 4.1: Describes the total amount of lines that has been added or updated in each framework. Detailed list of Lines of Code can be found under Appendix A.1

5.1.2 Files

Framework	Files Added	Files Updated
Ember	1	2
React-Flux	6	1
Angular 2	2	4
Vue	2	2

Table 4.2: Describes the total amount of files that has been added or updated in each framework. Detailed list of Files handled can be found under Appendix A.2

5.1.3 Dependencies

Framework	Dependencies	Files with Dependencies
Ember	7	2
React-Flux	6	3
Angular 2	7	3
Vue	7	3

Table 4.3: Describes the total Dependencies present in the project. Detailed list of Dependencies can be found under Appendix B

6 Analysis

The frameworks which apply the unidirectional patterns shows a trend of more code required to implement the same functionality compared to the two-way data flow pattern Table 5.1. When comparing React to Vue there is roughly a ~137% increase in code mass.
However, Angular 2 only required a 35% percent increase in code mass when compared to Vue.
Amount of lines required to be updated also tends to be greater for the unidirectional pattern. For Angular 2, there is a staggering difference of 1450% (Vue) and 3000% (Ember).
It is probable that the low amount of lines required to be updated in Ember is because it supports extensibility to a greater extent than the other frameworks.

When analyzing the Table 5.2, the significant result once again lies with React, which required the most additions for its implementation, though this affects the amount of lines required to be updated in already existing files, which is very minimal. While Angular 2 points to the exact opposite direction, needing more file updates than file additions.
The reason for this difference may be because of the framework standards of Angular 2, which may implement a stricter approach regarding modularity. This approach guides the developer to divide code into more modular pieces and places in the project.

The inspection of dependencies presented in each application, as shown in Table 5.3 shows minor difference regarding the amount of dependencies. Each set of the implementation contains from 6 to 7 dependencies. Furthermore, each set of implementation has 2 or 3 files where the dependencies are located within.

There are however differences when one compares the dependencies from a code churn point of view [Appendix B]. The majority of the unidirectional frameworks dependencies were implemented using additions. React has 5 out of 6, whilst Angular 2 uses 3 out of 7 dependencies implemented in new files as additions. On the other hand, the two frameworks using two-way data flow tends to update already existing files more frequently than adding new ones. All 7 of Embers dependencies and 6 out of 7 dependencies in Vue were located in existing files.

The trends indicate that the unidirectional pattern needed more files and code added to implement the authentication functionality, perhaps because of more loose coupling and stricter approach toward modularity. With the set of frameworks selected, the different best practices of each framework may

affect the developer into implement a certain way for its best practice. For example, the pattern chosen to implement the application in React, asks for Actions functions to call reducers to mutate the actual state of the application. If compared to Vue, which just performs a single call from the View to the Model to mutate the state, one could expect the unidirectional is more loose coupled and force more files and lines of code.

The fact that Angular 2 and React + Flux at times show such different results, and at some points even contradicts each other, leads to the conclusion that other factors than data flow related have affected the results.
This leads to the conclusion that other factors than data flow have affected the results. Our conclusions to the code churn results are that they are affected by the differences in framework standards, such as modularity, reusability, extensibility and inversion of control, to such extent that they are not reliable from a data-flow pattern point of view.
However, the interpretation could be more decisive with a broader spectra of frameworks comprised. The amount of data analyzed is not deemed enough to make any conclusions.

7 Discussion

Our results indicate that **RQ1.1** has ambiguous results. There are 154 additions and only 7 updates implemented in React, indicating a positive effect from a maintainability perspective. However, as Vue is implementing a two-way data flow and also indicates the same trend with 65 additions and only 2 updates. Considering this, it might be more probable that the results regarding **RQ1.1** derive from framework standards and practices rather than the data flow pattern the framework supports.

RQ1.2 has similar ambiguous results. The number of files added or updated shows no clear trend between the unidirectional data flow and two-way data flow. It is probable that once again, the framework standards and practices rather than the data flow may be affecting the results.

RQ1.3 The frameworks implementing the unidirectional data flow shows that more dependencies are added via new files [Appendix B].

- React has 7 of 8 additions in new files.
- Angular has 3 of 7 additions in new files.

Where the two frameworks using two-way data flow, has most dependencies updated in existing files instead.

- Ember has 0 of 7 additions in new files.
- Vue has 2 of 7 additions in new files.

As mentioned in the previous research, file additions have a positive impact on maintainability rather than updating files which have a negative impact [10]. From our results, we may come to the conclusion that the unidirectional data flow may have the positive impact on maintainability. However, the set of four frameworks is too small to draw any strong conclusions from.

Therefore the unidirectional data flow may affect maintainability in a positive way, but due to the frameworks potentially having a bigger impact on the results, no conclusions can be drawn.

8 Conclusion

The following chapter concludes the definitive answer to our thesis.

> **RQ1.1:** The results regarding the amount of rows implemented shows no significant difference between the data flow patterns
>
> **RQ1.2:** The results regarding the amount of lines implemented show no significant difference between the data flow patterns.
>
> **RQ1.3:** The results regarding the amount of dependencies implemented do show a slight difference between the data flow pattern. However, we deem our data to not be credible enough to make a conclusion.

The combined results of our sub-research questions point to the conclusion that maintainability is most likely not directly affected by the data flow pattern used. The results rather point to the fact that a certain framework and the way the framework chooses to implement its data flow pattern is the main reason of impact on maintainability.

Therefore we implicate that the answer to **RQ1** is that we cannot draw any conclusions from our method that there exist any differences between the two data flow patterns from a maintainability perspective.

Finally, we conclude that our research method does not apply well to the measurement of data flow patterns impact on maintainability, but rather with measuring JavaScript frameworks and their impact on maintainability.

8.1 Future Research

In future researches we would like to see the experiment conducted with a focus on JavaScript frameworks instead of data flow patterns.
The experiment should be performed with more programmers from different backgrounds and experiences; this would bring a wider more reliable perspective to the results. The research could also be carried out with a wider range of frameworks.

Since the research showed a stronger relation to the actual frameworks used, rather than the data flow patterns, it could also be interesting to see the data flow patterns used with different implementations using the same framework.

Example of future research questions might be:

- *What are the differences between certain JavaScript frameworks from a maintainability perspective*

References

[1] Andreas Gizas, Sotiris Christodoulou, and Theodore Papatheodorou. 2012. Comparative evaluation of javascript frameworks. In Proceedings of the 21st International Conference on World Wide Web (WWW '12 Companion). ACM, New York, NY, USA, 513-514. DOI=http://dx.doi.org/10.1145/2187980.2188103

[2] "Instagram",Instagram.com, 2016. [Online]. Available: https://www.instagram.com/. [Accessed: 19- May- 2016].

[3] "React (Javascript Library)". Wikipedia. N.p., 2016. Web. 19 May 2016.

[4] "Angularjs". Wikipedia. N.p., 2016. Web. 19 May 2016.

[5] "Ember.Js". Wikipedia. N.p., 2016. Web. 19 May 2016.

[6] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc and T. Gyimóthy, "A probabilistic software quality model," Software Maintenance (ICSM), 2011 27th IEEE International Conference on, Williamsburg, VI, 2011, pp. 243-252.
doi: 10.1109/ICSM.2011.608079

[7] J. C. Munson and S. G. Elbaum, "Code churn: a measure for estimating the impact of code change," Software Maintenance, 1998. Proceedings., International Conference on, Bethesda, MD, 1998, pp. 24-31.
doi: 10.1109/ICSM.1998.738486

[8] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, 2005, pp. 284-292.
doi: 10.1109/ICSE.2005.1553571

[9] Faragó, Csaba, Hegedűs, Péter, and Ferenc, Rudolf. The impact of version control operations on the quality change of the source code. In Computational Science and Its Applications–ICCSA 2014, pages 353–369. Springer, 2014.

[10] C. Farago, P. Hegedus and R. Ferenc, "Cumulative code churn: Impact on maintainability," Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on, Bremen, 2015, pp. 141-150.
doi: 10.1109/SCAM.2015.7335410, Ch 3

[11] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, Madrid, 2007, pp. 364-373. doi: 10.1109/ESEM.2007.13

[12] Vlissides, John M. and Mark A. Linton. "Unidraw: A Framework For Building Domain-Specific Graphical Editors". ACM Transactions on Information Systems 8.3 (1990): 237-268. Web. 19 May 2016.

[13] "Getting Started - Vue.Js". Vuejs.org. N.p., 2016. Web. 19 May 2016.

[14] "Git - Git-Diff Documentation". Git-scm.com. N.p., 2016. Web. 19 May 2016.

[15] E.Magnusson and D. Grenmyr, "Examensarbete 2DV50E/Implementation", GitHub, 2016. [Online]. Available: https://github.com/Examensarbete-2DV50E/Implementation. [Accessed: 23- May- 2016]

[16] Object-Oriented Application Frameworks. [Online]. Available: http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html [Accessed: 01- July- 2016]

[17] F. Bachmann, L. Bass and R. Nord, 2007. [Online]. Available: http://www.sei.cmu.edu/reports/07tr002.pdf. [Accessed: 16- Jul- 2016].

A Appendix: Detailed Results of Code Churn

A.1 Lines

The Vue implementation contains a method which is not used.(Line 29-31 in the Vue/js/store.js file). These rows is deducted from the amount of rows counted in that file.

Frameworks	Lines Added	Lines Updated
Ember		
emberjs/index.html	16	1
emberjs/js/controllers/todos_controller.js	23	0
emberjs/js/models/user.js	22	0
React-Flux		
react-flux/js/actions/UserActions.js	27	0
react-flux/js/actions/database.js	21	0
react-flux/js/components/AuthButton.react.js	23	0
react-flux/js/components/LoginForm.react.js	22	0
react-flux/js/components/TodoApp.react.js	13	7
react-flux/js/constants/UserConstants.js	5	0
react-flux/js/stores/UserStore.js	43	0
Angular 2		
angular2/app/app.html	11	21
angular2/app/app.js	14	4
angular2/app/bootstrap.js	1	2
angular2/app/services/database.js	21	0
angular2/app/services/userStore.js	38	0
angular2/index.html	1	1

Vue		
vue/index.html	11	0
vue/js/app.js	15	1
vue/js/database.js	22	0
vue/js/store.js	17	1

A.2 Files

The files that has had any modifications or has been added to the project is marked by an “x”.

Frameworks	File Added	File Updated
Ember		
emberjs/index.html		x
emberjs/js/controllers/todos_controller.js		x
emberjs/js/models/user.js	x	
React-Flux		
react-flux/js/actions/UserActions.js	x	
react-flux/js/actions/database.js	x	
react-flux/js/components/AuthButton.react.js	x	
react-flux/js/components/LoginForm.react.js	x	
react-flux/js/components/TodoApp.react.js		x
react-flux/js/constants/UserConstants.js	x	
react-flux/js/stores/UserStore.js	x	
Angular 2		
angular2/app/app.html		x
angular2/app/app.js		x
angular2/app/bootstrap.js		x
angular2/app/services/database.js	x	

angular2/app/services/userStore.js	x	
angular2/index.html		x
Vue		
vue/index.html		x
vue/js/app.js		x
vue/js/database.js	x	
vue/js/store.js		x

B Appendix: Dependencies to login state or asynchronous login/logout functions.

Total amount of dependencies within each file to login state or the model that has asynchronous login or logout functions. In React, Vue and Angular 2 the model is called db and is imported through require. In Ember it is called user and is added as Ember model, due to Ember TodoMVC application not supporting require.

Frameworks	Dependencies	On Line
Ember		
emberjs/index.html	3	35,39,65
emberjs/js/controllers/todos_controller.js	4	9,12,21,24
emberjs/js/models/user.js		
React-Flux		
react-flux/js/actions/UserActions.js	2	7,18
react-flux/js/actions/database.js	0	
react-flux/js/components/AuthButton.react.js	3	6,17,19
react-flux/js/components/LoginForm.react.js	0	
react-flux/js/components/TodoApp.react.js	1	32
react-flux/js/constants/UserConstants.js	0	
react-flux/js/stores/UserStore.js	0	
Angular 2		
angular2/app/app.html	2	4,28
angular2/app/app.js	2	57,62
angular2/app/services/database.js	0	
angular2/app/services/userStore.js	3	15,26,34
angular2/index.html	0	

angular2/app/bootstrap.js	0	
Vue		
vue/index.html	3	3,14,47
vue/js/app.js	2	69,77
vue/js/database.js	0	
vue/js/store.js	2	16,23

C Appendix: Developers of TodoMVC applications

Ember.js (1.10.0)
http://github.com/tomdale
http://github.com/addyosmani

Vue.js (1.0.1)
http://evanyou.me

React.js + Flux (0.14.7)
http://facebook.com/bill.fisher.771

Angular 2 (2.0.0-beta.0)
https://github.com/samccone