Maintaining Parallel Realities in CQRS and Event Sourcing

Ehren Thomas Eschmann

Cleveland State University

Follow this and additional works at: http://engagedscholarship.csuohio.edu/etdarchive

Part of the Computer Engineering Commons, and the Electrical and Computer Engineering Commons

How does access to this work benefit you? Let us know!

Recommended Citation

http://engagedscholarship.csuohio.edu/etdarchive/962

This Thesis is brought to you for free and open access by EngagedScholarship@CSU. It has been accepted for inclusion in ETD Archive by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.
MAINTAINING PARALLEL REALITIES IN CQRS AND EVENT SOURCING

EHREN THOMAS ESCHMANN

Bachelor of Computer Science

John Carroll University

May 2008

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

August 2017
We hereby approve this thesis for

Ehren T. Eschmann

Candidate for the Master of Science in Software Engineering degree for the

Department of Electrical Engineering and Computer Science

and the CLEVELAND STATE UNIVERSITY

College of Graduate Studies

Thesis Chairperson, Dr. Nigamanth Sridhar

Thesis Committee Member, Dr. Yongjian Fu

Thesis Committee Member, Dr. Janche Sang

Students Date of Defense: (05/04/2017)
For my wife, Kristen ...
ACKNOWLEDGEMENTS

It is with great honor and pleasure that I acknowledge the faculty, staff and students at the Cleveland State University, during my tenure as a master’s student. My work with software architecture was partly inspired by the positive and talented people I surrounded myself with over the past few years. My thesis advisor, Dr. Nigamanth Sridhar, had a profound influence on my work from the day we met a week before I walked into my first classroom at CSU. Along those lines, I’m particularly grateful to my thesis committee of Dr. Nigamanth Sridhar, Dr. Yongjian Fu and Dr. Janche Sang. Furthermore, my passion for the design and implementation of quality software expanded under the talented instruction of Dr. Chien-Hua Lin and Dr. Sunni Chung.

The Electrical Engineering and Computer Science department is quite fortunate to have such a dynamic administrative assistant for their graduate school. Melanie Caves is dedicated to ensuring the success of her students. I attribute much of the success of this work and my success as a student to Melanie.

As life-long academics my parents, Tom and Pam Eschmann, always placed a substantial emphasis on education when I was growing up. They invested every dollar they earned in the growth and success of their children. My brothers and I were given every opportunity to be successful. That type of devotion was incredibly motivating when I set out to accomplish this thesis.

I want to thank all the good people at MRI Software and the Cleveland Clinic for making my weekdays enjoyable and giving me a reason to pursue my passion.
Finally, this thesis would not be possible without the support from my amazing wife. Kristen was chief in inspiring my research while also caring for our son, TJ. She and I shared many late nights pair-programming the Fictional Workflow Builder. Her encouragement, energy, patience and beautiful smile are the soul of this work.
MAINTAINING PARALLEL REALITIES IN CQRS AND EVENT SOURCING

EHREN THOMAS ESCHMANN

ABSTRACT

In today’s distributed software ecosystem, we have witnessed a broad exhibition of notable approaches to software architecture. Traditionally, these approaches have centered around persisting a system’s current state. Rather than adhere to these criteria, two modern architectures, Command Query Responsibility Segregation (CQRS) and Event Sourcing have inspired us to persist the interactions of the software actor as replayable events which describe the history of their input data.

While CQRS and Event Sourcing allow for considerable benefits in many types of systems, maintaining parallel realities (multiple snapshots of history deriving from a single parent history) is generally regarded as too complex for maintainability.

In our pursuit to achieve parallel realities in Event Sourcing systems, we established Command Sourcing, a superset of the two aforementioned architectures. Leveraging Command Sourcing, we effectively demonstrate maintainable parallel realities as part of a collection of architectural guidelines, data structures, and algorithms. By further applying Command Sourcing and researching the algorithms that belong in these systems, we present solutions to related complex milestones such as merging realities, reality optimization, conflict resolution, and aggregate duplication.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS........................................................................................................ iv

ABSTRACT............................................................................................................................. vi

LIST OF FIGURES .................................................................................................................. x

CHAPTER

I INTRODUCTION......................................................................................................................1

II GUIDING PRINCIPLES, PATTERNS, AND TERMINOLOGY ........................................ 4

  2.1. The Command Pattern.................................................................................................. 4

  2.2. Command-Query Separation......................................................................................... 5

  2.3. Bounded Context.......................................................................................................... 6

  2.4. Aggregate Root............................................................................................................ 6

  2.5. High Cohesion.............................................................................................................. 7

  2.6. Low Coupling.............................................................................................................. 7

III THE FICTIONAL WORKFLOW BUILDER ........................................................................ 9

  3.1. Consistent Terminology.............................................................................................. 9

  3.2. Overview..................................................................................................................... 9

  3.3. Getting Started.......................................................................................................... 10

  3.4. Consuming the Application......................................................................................... 10

  3.5. Implementation in TypeScript.................................................................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6. Testing the Fictional Workflow Builder</td>
<td>15</td>
</tr>
<tr>
<td>3.7. Sample Project: Approving PTO</td>
<td>16</td>
</tr>
<tr>
<td>IV CQRS AND EVENT SOURCING VS. TRADITIONAL CRUD APPROACH</td>
<td>18</td>
</tr>
<tr>
<td>V EXTENDING CQRS AND EVENT SOURCING</td>
<td>25</td>
</tr>
<tr>
<td>5.1. Parallel Realities</td>
<td>25</td>
</tr>
<tr>
<td>5.2. Command Sourcing</td>
<td>26</td>
</tr>
<tr>
<td>5.3. Databinding Aggregates</td>
<td>29</td>
</tr>
<tr>
<td>5.4. Command Sourcing Structure</td>
<td>30</td>
</tr>
<tr>
<td>5.4.1. QueryBus</td>
<td>31</td>
</tr>
<tr>
<td>5.4.2. CommandBus</td>
<td>31</td>
</tr>
<tr>
<td>5.4.3. DomainStore and DomainCache</td>
<td>32</td>
</tr>
<tr>
<td>5.4.4. CommandStore</td>
<td>32</td>
</tr>
<tr>
<td>5.4.5. Command and the Concrete Command</td>
<td>33</td>
</tr>
<tr>
<td>5.4.6. Instigator and Observer</td>
<td>33</td>
</tr>
<tr>
<td>5.5. Anatomy of a Command</td>
<td>33</td>
</tr>
<tr>
<td>5.6. Cutting / Copying / Pasting Aggregates</td>
<td>34</td>
</tr>
<tr>
<td>5.7. Testing with Command Sourcing</td>
<td>39</td>
</tr>
<tr>
<td>5.8. The New TypeStore Factory</td>
<td>42</td>
</tr>
<tr>
<td>VI MAINTAINING PARALLEL REALITIES WITH COMMAND SOURCING</td>
<td>45</td>
</tr>
<tr>
<td>6.1. Underlying Data Structures</td>
<td>46</td>
</tr>
</tbody>
</table>
6.2. Optimizing Command Stacks.................................................................47
6.3. Forking Strategy ..................................................................................50
6.4. Concurrency Strategy .........................................................................51
6.5. Conflict Resolution ............................................................................53
6.6. Merging Up ........................................................................................54
6.7. Merging Down ....................................................................................55
6.8. The Stack Dirty Flag ..........................................................................57
6.9. Hashing Aggregates ...........................................................................57

VII CONCLUSIONS ..................................................................................59

BIBLIOGRAPHY .......................................................................................61
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Command Pattern UML</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Cloning a GIT Repository</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Installing Package Dependencies</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Starting the Application</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Fictional Workflow Builder Root Menu</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Fictional Workflow Builder Parallel Reality</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Fictional Workflow Builder Conflict Modal</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Fictional Workflow Builder Aggregate Depiction</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>The PTO Approval Sample Project</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>CRUD Architectural Approach</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>CQRS</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Event Sourcing</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>Command Sourcing</td>
<td>29</td>
</tr>
<tr>
<td>14</td>
<td>Command Sourcing Class Diagram</td>
<td>31</td>
</tr>
<tr>
<td>15</td>
<td>Creating a Command to Duplication an Aggregate</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>Recursively Assigning Hashes to Future Aggregates</td>
<td>37</td>
</tr>
<tr>
<td>17</td>
<td>Aggregate Duplication With Macro Command</td>
<td>38</td>
</tr>
<tr>
<td>18</td>
<td>Aggregate Duplication With Serialized Blob</td>
<td>39</td>
</tr>
<tr>
<td>19</td>
<td>Testing Command State Consistency</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>State Mutation Test of the Fictional Workflow Builder’s ViewState Service</td>
<td>42</td>
</tr>
<tr>
<td>21</td>
<td>Caching Available Types For the TypeStore Factory</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>22</td>
<td>TypeStore Factory Creation Method</td>
<td>43</td>
</tr>
<tr>
<td>23</td>
<td>TypeStore Factory String Type Creation Method</td>
<td>43</td>
</tr>
<tr>
<td>24</td>
<td>Aggregate Cache Invalidation</td>
<td>43</td>
</tr>
<tr>
<td>25</td>
<td>Linked List Implementation in TypeScript</td>
<td>46</td>
</tr>
<tr>
<td>26</td>
<td>Depiction of the Command Optimization Algorithm</td>
<td>50</td>
</tr>
<tr>
<td>27</td>
<td>Optimizing The Command Stack In TypeScript</td>
<td>50</td>
</tr>
<tr>
<td>28</td>
<td>Discovering a Realities Full History</td>
<td>51</td>
</tr>
<tr>
<td>29</td>
<td>Forking Command Stacks</td>
<td>51</td>
</tr>
<tr>
<td>30</td>
<td>Merging A Reality Up With It’s Parent</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>Merging Up</td>
<td>55</td>
</tr>
<tr>
<td>32</td>
<td>Merging A Reality Down With It’s Children</td>
<td>56</td>
</tr>
<tr>
<td>33</td>
<td>Merging Down</td>
<td>57</td>
</tr>
<tr>
<td>34</td>
<td>Base-64 16-Digit Random Hashing Function</td>
<td>58</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

“I foresee two possibilities. One: coming face to face with herself thirty years older would put her into shock and she'd simply pass out. Or two, the encounter could create a time paradox, the result of which could cause a chain reaction that would unravel the very fabric of the space-time continuum and destroy the entire universe! Granted, that's worst-case scenario. The destruction might in fact be very localized, limited to merely our own galaxy.” [1]

-- Emmett Brown, Back to the Future 2.

The Command Query Responsibility Segregation (CQRS) and Event Sourcing architectural patterns fracture the traditional view of modeling software where we persist the state of our systems, not as facts within our object models, but as a collection of behaviors or events that have occurred in the past which describe our current state [2]. This history of immutable events can be replayed on any client to reconstruct a given set of aggregates. However, Greg Young claimed of Event Sourcing in 2010 [3] that “Parallel realities are far too complex and costly to model in most business systems.”

Parallel realities are a very powerful feature for modern software. While still upholding the architectural trademarks that compose CQRS and Event Sourcing, by
extending the architecture, we can achieve and maintain parallel realities in many deterministic systems.

This thesis explores a junction of the two modern software architectures, CQRS and Event Sourcing, and the circumstances by which we can extend them to achieve parallel realities. Through our work on the Fictional Workflow Builder application, we validate that through customization and standardization, parallel realities are achievable in deterministic systems.

The first two chapters of this work review the guiding principles and architectural background required for understanding our research. Common principles, patterns, architectures, and concepts are surveyed. Chapter III is an overview of our sample project, the Fictional Workflow Builder. We explore attaining the source code and running the project locally. A technical overview and brief tutorial on the application is provided. In Chapter IV, we introduce Command Sourcing, an architectural superset of CQRS and Event Sourcing. We then discuss the characteristics of the software and the architectural standards that must be followed to make parallel realities maintainable in Command Sourcing. As we’ve chosen our programming language to be TypeScript for our Fictional Workflow Builder, we will define how that architecture looks and explain some of the caveats of building these types of applications with a multi-paradigm language. One obstacle, the inability to dynamically generate domain objects, led to the creation of the TypeStore Factory, which is presented in this chapter.

CQRS and Event Sourcing offer us a template and guidelines to follow when building the Fictional Workflow Builder application, but they do not offer any suggestions
to creating or managing parallel realities. We review that superset of responsibilities in
Chapter V.

In summary, this thesis provides the following contributions to the field of Software
Engineering:

• Command Sourcing -- an architectural extension of CQRS and Event Sourcing.
• A set of constraints that warrant the use of Command Sourcing in software systems.
• The necessary algorithms required to achieve parallel realities in Command
  Sourcing and,
• A framework and implementation of Command Sourcing written in TypeScript
  along with the TypeStore Factory, a new factory pattern for dynamically loading
  module dependencies in TypeScript.
CHAPTER II
GUIDING PRINCIPLES, PATTERNS, AND TERMINOLOGY

2.1. The Command Pattern

Introduced in 1994 [4], the intent of the Command Pattern is to encapsulate the behavior of an operation as an object, allowing the engineer to either defer execution to a later time, undo its execution, log requests or parameterize clients with different requests. It is categorized as a behavioral pattern, one that is concerned with algorithms and the assignment of responsibilities between objects. Apart from the aforementioned benefits, the Command Pattern can be used when structuring some application around high-level operations built on primitive operations. As noted by Gamma et. al, this is common in information systems that support transactions such as a Database Management System (DBMS).

The structure of the pattern is relatively simple. Depending on the requirements for the system, a single Command abstraction should define an interface for executing an operation, while its derivatives define the behavior to complete the execution. Other participants in this pattern are the Receiver who understands how to carry out actions of
the Command, the Client who creates the Commands and alternatively set’s the Receiver of the Command, and the Invoker who asks the Commands to carry out its work.

![Command Pattern UML Diagram](image)

**Figure 1: Command Pattern UML**

Adding new Commands to a system is easy; you simply implement the Command interface. Since there is no need to modify existing classes, the Command Pattern supports the Open Closed Principle [5] and it supports the Single Responsibility Principle [6]. An unintended, yet fortunate, consequence of this pattern is that it decouples the execution behavior from the invoking class and it decouples the invoking class from the class that mutates the state of the system.

2.2. Command-Query Separation

In 1988, Bertrand Meyer [5] taught us that there are two distinct kinds of methods in object orientation: commands, which perform an action thereby mutating state, and queries, which answer a question and do not affect the state of the system. **Command-Query Separation** simply states that we should strive to maintain commands and queries as separate entities as much as possible in our methods by either returning data or
modifying state, but not both. There are two major benefits to following this principle. First, by separating the two, we can avoid side effects, or leaks, that would otherwise hide in functions violating this principle. Secondly, Command-Query Separation can reduce complexity in a system. Quite simply, it is challenging to reason about an object, if its state mutates whenever you observe it.

However, as Martin Fowler points out [7], there are some downsides to this principle and some reasons to not abide by it. He used a Stack’s Pop method as an example, which mutates the state of the data structure, then returns the data removed. There are other examples where Command-Query Separation might not be ideal, most notably in multithreaded systems.

2.3. Bounded Context

The Bounded Context is one of three central themes of Domain Driven Design (DDD) [8]. It acts as a conceptual boundary around related domain models. At each boundary, different architectural approaches can be applied. Domain Driven Design focuses on organizing and simplifying large models and the teams supporting and building those models. By separating domains into Bounded Contexts and managing their dependencies and interrelationships, we can more effectively reason about a system.

By organizing a system into multiple Bounded Contexts, we can employ different architectural approaches at each logical separation. As we identify different patterns and the cost of those patterns, the flexibility to employ them becomes increasingly valuable.

2.4. Aggregate Root
An aggregate is a group of disparate elements, that when combined, becomes a single cohesive unit. These elements take on a logical hierarchical structure with the root element having a special purpose in DDD. This Aggregate Root has one special characteristic: it has global identity within the system, whereas all other elements in the aggregate have identity local to its bounded context. [8]

Remember, a key benefit of applying the Bounded Context is the ability to apply various architectural patterns at each logical separation. Without the ability to reference aggregates globally within adjoining Bounded Contexts, the system cannot communicate across domain boundaries. In this case, we would consider all Bounded Contexts disparate systems. Each Aggregate Root acts as a connection for data flow across logical boundaries.

2.5. High Cohesion

Functional cohesion is a measure of how closely one item is related to another [9]. Moreover, it measures how focused the responsibilities of an item are. GRASP (General Responsibility Assignment Software Patterns) [10] states that a class has High Cohesion if it’s responsibilities are highly related and it does not perform a large amount of work. Conversely, a class with low cohesion will remain responsible for several unrelated things. It is an evaluative principle that cannot be applied in isolation of other principles, such as Low Coupling.

2.6. Low Coupling

Coupling measures the amount an item has knowledge of, is connected to, or relies on other items known as dependencies [11]. Briefly, we aim to minimize coupling so as to prevent a spontaneous reaction of changes if a single dependent item changes, thus
lowering the impact of that change. Items may refer to classes, systems, or even standards. It is one of the cardinal goals of building software.
CHAPTER III
THE FICTIONAL WORKFLOW BUILDER

3.1. Consistent Terminology

- **Forking** - To fork is to take a snapshot of the current state of the system and duplicate it somewhere else.
- **Depth** - In the process of forking, we create a child of the original set of aggregates. That child lives at a depth one greater than the depth of its parent. Creating a new reality from an initial state sets that reality at depth zero.
- **Stability** - We consider a reality less stable if it has a higher depth. Conversely, we consider a reality more stable if it maintains a lower depth.

3.2. Overview

Throughout this thesis, we reference our sample project, the Fictional Workflow Builder. It is intended for demonstration purposes only. The repository is publicly available for download on GitHub [12]. It is a single module, componentized web application written in TypeScript. Our TypeScript transpiles to ECMAScript 5 (ES5) [13] modules with the CommonJS module format [14]. This results in an application that can
be viewed on most modern browsers. The Fictional Workflow Builder uses SystemJS [15] to load external modules in the browser. We chose Angular v4.0 [16] as our frontend framework.

3.3. Getting Started

The Fictional Workflow Builder has a few global dependencies: NodeJS [17] and TypeScript [18]. Both must be installed on the development machine before running the code.

After the dependencies are installed, we must first clone the repository. We can do this by executing the following command in a terminal window:

```
# git clone https://github.com/EhrenEschmann/fictional-workflow-builder.git
```

**Figure 2: Cloning a GIT Repository**

From there we must use the NodeJS Package Manager (npm) [19] to install local dependencies for the Fictional Workflow Builder.

```
# npm install
```

**Figure 3: Installing Package Dependencies**

Finally, we can run a script to build the application, run our automated test suite, run our test web server, and open it in a browser with npm:

```
# npm start
```

**Figure 4: Starting the Application**

3.4. Consuming the Application
Upon loading, the actor is given the opportunity to create a new workflow. By selecting this button, the actor enables the Workflow Component which displays a newly created workflow and a new root reality. The Workflow Component initially displays the following menu:

![Figure 5: Fictional Workflow Builder Root Menu](image)

Since the Fictional Workflow Builder tracks parallel realities locally, each reality is assigned an identification number which is displayed on the far left (0, in Figure 5 above.) The *Fork* button will create a new parallel reality locally. That is modelled as a child of the root reality, seen here with identification number 1.

![Figure 6: Fictional Workflow Builder Parallel Reality](image)

You will notice that the *Merge Down* button on the root reality and the *Merge Up* button in its child reality are now enabled. These options will allow us to merge changes to our aggregates. Occasionally, when merging up, the Fictional Workflow Builder will detect a conflict. This means that the same aggregate property was modified in both realities after the fork and they contain different values. Those conflicts will manifest themselves in a modal:
Figure 7: Fictional Workflow Builder Conflict Modal

The Conflict modal title will present the actor with the merge direction and the target realities involved in the merge. For each conflict (two conflicts, seen above), the modal exhibits the uniquely identifying aggregate hash, the Command issue type, the From Value issued in the child reality, and the To Value issued in the parent reality. The conflict modal provides the actor with the fidelity to update the child reality with the new parent value or retain their existing value. If no action is taken on a conflict, the Fictional Workflow Builder will infer the child updates are more relevant and implicitly accept those for the merged reality.

The Optimize option manually runs our optimization algorithm against the Command Stack. More information on optimization is presented in Chapter V.

We can undo all of the modifications to a given reality with the Clear button. This won’t eliminate all state changes for that parallel state, just the changes made after it was forked. Similarly, we use the Undo and Redo buttons to reverse the previous execution and reverse the previous reversal, respectively. Depending on the state of the system, expect these buttons to disable when they are not available for use.

To add aggregates to the reality, we either select the Add Random Aggregate button for quick additions or we can create a specific aggregate by opening the attached split-
drop-down menu. The Fictional Workflow Builder supports four types of workflow: Send Email, Post REST API, Request Input, and Execute Compiled Binary.

The drop-down element beside the Add Random Aggregate button contains a list of all Commands executed against that reality. The list is for visual purposes only and provides no functionality. Adjacent to the Command drop-down are two numbers separated by a plus-sign. These are size indicators of our Command Stack. The first number represents the archive of Commands executed against the reality from its parent. The second number is a count of all Commands executed against the reality directly.

After an aggregate has been added to a reality, that aggregate will display in the Workflow Editor below. Seen here is a Post Rest API workflow step:

![Post Rest API Workflow Step](image)

**Figure 8: Fictional Workflow Builder Aggregate Depiction**

The workflow step will display the workflow step type, the uniquely identifying hash of the Aggregate, the Aggregates properties and values for those properties, and a list
of events with their attached child aggregates and a direct child size indicator. We can attach a new child Aggregate to an event by selecting the event and adding a new Aggregate with the *Add Aggregate* button on that realities menu. Each property can be modified directly. The property editor component maintains a ¾ second debounce time. This means that commands will execute against a reality to update that Aggregates property when the actor stops typing for ¾ of a second. As such, we might see multiple Commands added to our Stack count when modifying properties. The red X at the top of the Aggregate rendering will delete the Aggregate along with all of that Aggregate’s children.

The Fictional Workflow Builder supports duplication of aggregates. There are no visual indicators of this as we employ hotkeys to complete this user interaction. To copy an aggregate, the actor must first select the target aggregate, then invoke the in-memory duplication step by applying the keyboard shortcut, ctrl/cmd+c. This will store the aggregate on a virtual clipboard until the actor selects a target event on an aggregate and administers the ctrl/cmd+v keyboard shortcut. This sequence will execute the duplicated aggregate against the reality.

To assign aggregates to different parents, the actor simply drags the aggregate to the desired parent event. The Fictional Workflow Builder assigns simple error handling that prevents circular referencing of aggregates. It also supports dragging a nested child aggregate to the root of the workflow.

3.5. Implementation in TypeScript

We have chosen TypeScript as the programming language for the Fictional Workflow Builder for a few reasons. First, as mentioned, we want to keep the
implementation as succinct as possible and also want to demonstrate a realistic medium for parallel collaboration, namely the web. As TypeScript transpiles to JavaScript it is perfect for our use case. Second, we preferred a multi-paradigm programming language that would give us fidelity to leverage functional principles and also retain the Object-Oriented nature of the architecture.

Industry leading high-level programming languages such as C# and Java are also both excellent choices when building server-side applications, though very few languages are too restrictive in implementing this architecture.

We built the Fictional Workflow Builder project as a lightweight implementation of Command Sourcing. We’ve chosen this project due to the composite and hierarchical nature of its domain aggregates, and to highlight the parallel collaboration between several parties in branching and merging content often seen in code-editing and graphical drag/drop based applications.

3.6. Testing the Fictional Workflow Builder

The automation suite for the Fictional Workflow Builder is written in Jasmine [20], a behavior-based unit testing framework. Jasmine is fast, with no external JavaScript dependencies. The syntax is obvious so it is great for large teams or open source projects. We run all our tests for the Fictional Workflow Builder in the Karma [21] test runner. Karma spawns a webserver and executes source code against connected browsers. The results of our tests on each browser are displayed on the command line. Karma also listens to changes on each file and signals the webserver to notify each connected browser to re-run the automation suite. We include our unit test files directly in line with the source-
code for our project as siblings of the tested file, differentiating with a `.spec.ts` extension. By doing so, we prevent the need for a second deeply nested folder structure of code in our application. It primarily assists with long term maintainability. We have configured our module loader to ignore those files when serving the application to the browser.

3.7. Sample Project: Approving PTO

Periodically, we will reference a sample workflow project built with the Fictional Workflow Builder: The Paid Time Off (PTO) Approval Sample Project. The intent of this project is to track employee PTO approvals from their managers at some company. A manager can either approve or reject PTO and the workflow will send an email back to the employee notifying them of their managers response. If the manager does not respond in the requested duration (in this case, 1 day), the workflow will timeout, which forwards the input request to the director of the department. They, in turn, have the same options to accept or reject the PTO as the manager, however if they do not respond in the requested duration, the PTO is automatically rejected. This workflow is highlighted in Figure 9, below.
Figure 9: The PTO Approval Sample Project
CHAPTER IV

CQRS AND EVENT SOURCING VS. TRADITIONAL CRUD APPROACH

In typical entity create, read, update, delete (CRUD) architectural approaches, such as the Layered Architecture [22] or Onion Architecture [23], commands and queries are applied to the same object model. This object model (or some transformation of it) maps to fields in some relational database or other persistent store [22].

![CRUD Architectural Approach](image)

**Figure 10: CRUD Architectural Approach**

The traditional CRUD approach to software works well for simple business logic. The introduction of Object Relational Mapping (ORM) tools has made the case for building software following a CRUD approach more appealing. ORM’s such as
Microsoft’s Entity Framework [24] will scaffold data access code and a database schema from an object model. Additionally, we are only responsible for maintaining a single domain model when supporting a CRUD-based architectural approach.

A key drawback to the CRUD approach to software is data contention. This could occur in collaborative domains where multiple actors contend for the same resources in parallel. By reading stale data and modifying it, the actor might incidentally overwrite relevant data. CRUD architectures imply basic optimistic concurrency [25], which states that the system assumes that multiple transactions can frequently complete without interfering with each other. Furthermore, by sustaining a single object model, security maintenance may become complicated since each entity is subject to both read and write operations. Under some conditions, we risk exposing sensitive data. Also, since object-oriented CRUD based models mirror some relational database schema, when updating entities, we often modify more data than intended. By doing so, we further risk overwriting data in collaborative environments or unintentionally mutating state.

In some cases, we can improve on these drawbacks with Command Query Responsibility Segregation. CQRS expands the Command-Query Separation principle to the architectural level. In general, it divides the system or subsystem into two domain-centric stacks, one for state mutation and the other to answer questions about the state of system. CQRS can be applied at the system level, but was originally described by Udi Dahan to operate underneath a single Bounded Context [26].

In Figure 11 below, we see a separation of all write-based operations from all read-based operations. An actor will modify some data through the presentation layer and that change will propagate through a write interface, eventually mapping to some Command
**Model.** That model will travel down a bus where it will eventually be assigned a *Command Handler* for broadcasting and be persisted in some write-optimized store, usually a Third Normal Form relational database or a NOSQL document database.

Conversely, to read data, an actor would access a resource in the *presentation layer* which would make a request for data through the architecture’s *read interfaces*. A *Query Model*, either the models themselves or through a Facade Pattern [27] -- namely, an object which provides a simplified interface to a client object -- would get populated from the Domain Store, which exists as a read optimized datastore. The *presentation layer* would then become refreshed with the new models.

![Diagram of CQRS](image)

**Figure 11: CQRS**

For systems which do not require the additional complexity of two datastores, a single persistence medium is considered acceptable. Commands would execute behavior directly to the datastore and queries are executed directly against the same persistent store leveraging a thin data access layer.
In a variant of CQRS which maintains two datastores, the Commands manifest state into the *Domain Store* from the *Write Store*, either as a single coordinated transaction or as part of an eventual consistency pattern [28]. In this latter approach, the *Domain Store* may be out of sync temporarily. This discrepancy might last a few milliseconds. During this time, any call made through the read interfaces might return stale data. In connected systems, that data can be corrected when the read store comes to equilibrium with the write store. Maintaining a second datastore offers scalability and performance benefits [29]. Also, since queries should be optimized for reading data and commands should be optimized for writing data, CQRS enables us to build these optimizations in isolation. That logical separation can yield a less complex overall design.

In general, CQRS solves for collaboration and staleness [30]. By creating commands with fine granularity, we minimize the risk for merge conflict. Also, we can plan for specific scenarios and program our Commands to merge themselves. Furthermore, as the actor will invariably act upon stale data, some validation mechanism can reject conflicting commands but allow others, when our commands are created with a fine enough grain.

The architectural considerations for CQRS are deeply focused on collaborative environments and highly complex domains. If neither use case exists, then the benefits will not warrant the additional complexity and cost of distributing an application across multiple physical tiers.

Event Sourcing is an architectural pattern separate from CQRS. Though the two patterns complement each other, neither one implies the other. The main difference of Event Sourcing compared to CQRS is that instead of storing the current state of its entities,
it stores the state mutations of those entities over time as Events in a persistence medium known as the Event Store [31].

The Event Store is a database specifically optimized for the storage of Domain Events and complex processing on Event Streams. Separately, Event Store [32], a mature open source option, is a functional database which allows us to query the database in JavaScript. While Event Store offers us benefits in scalability, performance, and availability, it also provides us with Projections, which allows us to react to Domain Events as they are written.

While Commands request for state mutation and are dispatched to a single listener, Events are messages in the system that indicate something has happened and are broadcast to external listeners. An important difference between the two is that Commands can be rejected and Events are written history.

Consider Figure 12; in an Event Sourcing architecture, a consumer would request to mutate state in a system by dispatching a Command. That Command will either create a new aggregate or load one from the Event Store. The Command Handler will execute business logic against the aggregate. The aggregate in turn will create the relevant Events and apply any state mutation before calling on the Command Handler to persist those Events to the Event Store [33].

When the consumer queries the system, the request makes its way to the Event Store. There it returns the requested resource as a stream of Events that have occurred in the past, which yield the current state of the queried aggregate. We can achieve read optimization by employing Snapshotting [34], or caching the state of the system for a set
of aggregates at a current Event for quick retrieval. That way, the client system will not have to rebuild the entire state from an Event Stream, only from the latest snapshot.

Figure 12: Event Sourcing

There are two fundamental considerations for ensuring consistency. First, every change to state must be captured as an Event. Second, each Event must be partially ordered as the story Event Sourcing reveals is different when reordered [35].

When an aggregate is retrieved as a stream of Events, its historical record is replayed until it reaches its current state. Event Sourcing has some very powerful benefits in exchange for the extra complexity. First, we can reconstruct the state of any entity at any given time, which can be useful for testing, diagnostics, and debugging. That natural audit trail also offers insight to the actor’s intent. Also, by rejecting conflicting events, we can minimize or even eliminate data contention. Furthermore, by delivering potentially complex or hierarchical object models projected as a flat list of Events, we eliminate the need to map those object models to transfer objects when delivering them across physical
tiers. Finally, we can restore the system to any previous state by replaying just those Events, which assists with debugging.

There are some concerns to address with Event Sourcing. While Snapshotting can improve performance, that optimization comes at a complexity cost. Largely without Snapshotting, aggregate sets with massive amounts of Events will hinder performance. Also, since Events will execute functionality repeatedly against the same aggregate, careful thought must be placed on extending the system in the future. If the likelihood that Event contracts or implementations will change over time, each Event must be carefully versioned. Finally, though it is trivial in Event Sourcing to retrieve the current state of an aggregate, it can be onerous to query for data that spans all Events.
5.1. Parallel Realities

We define parallel realities as modelled in software as the past, future or any prior alternative junction of the current state of a system. Namely:

1. We can undo actions taken on the state.
2. We can replay actions that were previously undone on the state or take new actions.
3. We can fork into an independent reality, operating independently of its parent.
4. We can merge forked realities back with their parent.

Parallel realities are challenging to model in any type of information system. Though challenging, it’s not impossible. Martin Fowler points out that a common example of parallel realities is any version control system implementation [36].

CRUD-based systems have one unique advantage: the limited scope of the actions they can take on an aggregate. Our only obligation is the creation, modification and
deletion of aggregates. However, since we have no means of accessing iterative prior state, we would have to cache each individual state upon its mutation if we wanted to satisfy constraints (1) or (2), albeit inefficiently, above. We can satisfy the third constraint by duplicating the state of the aggregates, but merging forked realities would likely suffer data contention loss.

CQRS as an architecture distributed across multiple physical tiers solves different problems than that of parallel realities. By segregating write interfaces from the read interfaces we can focus on a single entry point for state mutation. Creating separate Commands and writing code to reverse the mutation step gets us a little closer. However, CQRS still centers on maintaining the current state of the system, so while the undo and redo constraints can be satisfied by maintaining an undo Stack, forking would inefficiently duplicate state and merging would likely suffer similar data loss as the CRUD-based approach.

Recall with Event Sourcing, a historical preview of Events is persisted in the Event Store as a record of how to bring a set of aggregates to their current state. Therefore, the second constraint of achieving parallel realities is supported out of the box. However, since the Event Store is an append-only datastore, we cannot reverse state natively in an Event Sourcing system. Forking into an independent reality is natively supported but merging forked realities back up to their parents can only occur as a coordinated transaction after the parents last committed Event.

5.2. Command Sourcing
We’ve seen the term Command Sourcing casually referenced on the internet [37] to describe Event Sourced systems that persist nondeterministic Commands which mutate the state of the system. Martin Fowler’s blog post on Event Sourcing [36] closely resembles this definition of Command Sourcing [38]. It is generally regarded as an inferior approach to Event Sourcing.

We have applied the term Command Sourcing to the architectural superset of CQRS and Event Sourcing which treats deterministic Commands as the sole persistence object. Command Sourcing inherits the predominant features of CQRS and Event Sourcing while consolidating complexity. It does this by eliminating most physical separation you find in CQRS and concentrate it in one location. In web-based applications like the Fictional Workflow Builder, it runs in the actor’s web browser. Like Event Sourcing, Command Sourcing persists a collection of behaviors and replays them to reveal the current state.

Whether they are Events or Commands, to ensure consistency, the persistence object must maintain determinism. Namely, for a given state of a system, inputs to a Command would behave the same each time that Command executes. This does not single out randomness in a Command. Commands must also maintain a partial ordering. For instance, Commands which reference aggregates must not rely on current state to apply a state change. That current state might be different in a parallel reality.

We settled on the term Command for our persistence object for two reasons. First, beneath the write interface in Command Sourcing, is a Command Pattern implementation. Second, while Events describe past actions that cannot be reversed in a
system, in a multi-reality system, Commands which exist on the stack of completed tasks can be undone.

The current state, generated by replaying a stack of Command objects, lives in a local domain model which is accessible from the read interfaces, but it also lives flattened in a hash table of references, so we can enjoy O(1) access to these entities. If this reality was forked from a parent reality, it is considered a parallel reality until it is later combined with the remaining data as part of a manual merge process. This implies we might have multiple consumers mutating the state of a system at any time.

There are other benefits to persisting the state of a system as a collection of behaviors. Unfortunately for complex view models, mapping data across physical tiers is tedious and complicated. Furthermore, we might have multiple projections of the data we must maintain at each layer. Storing a flat structure of Command objects is very appealing and the deserialization of a single abstraction is trivial.

The architectural overview of Command Sourcing in Figure 13 below appears similar to CQRS and Event Sourcing. The reason being, Command Sourcing is Event Sourcing and it is CQRS. We have moved the domain store into memory and created a cache for resolving aggregates within the system. The domain model is always hydrated as long as it lives in memory. That means the query interfaces interact with the cache of aggregates directly. At the core of this architecture we focus on the key facets of CQRS and Event Sourcing: Segregated read and write subsystems, distinct read and write models, and persistence of behaviors, which bring the system to its current state.
Command Sourcing can be leveraged as both a server-side and client-side architecture. While Command Sourcing does support Domain Driven Design and commands can describe behaviors, the architecture is intended for CRUD-Based commands. While CQRS and Event Sourcing both necessitate additional complexity, Command Sourcing is trivial to update and maintain.

Consider applying Command Sourcing in the following situations:

1. CRUD-based systems with complex domain models.
2. Web-based applications with complex view models.
3. Any application which requires undoing or redoing of actions.
4. Medium scale collaborative environments
5. Systems with a graphical user interface

5.3. Databinding Aggregates
Data-binding -- UI elements observing and displaying updates to their view models -- provides us an opportunity to build rich, aware, connected applications in the web. Command Sourcing itself does not focus on data-binding.

Most data-binding frameworks provide 2-way data-binding [39]. 2-way data-binding allows us to observe external changes to an aggregate from a UI element as they occur and directly update an aggregate from that same UI element. Since Command Sourcing requires a Command for every state mutation, and 2-way data-binding potentially mutates state from within a UI element, unless binding to a copy of the domain, 2-way data-binding is outlawed within a Command Sourcing system. Function (1-way) data-binding is the preferred approach.

5.4. Command Sourcing Structure

A typical UML diagram of a Command Sourcing implementation is presented in Figure 14, below. Like CQRS, it segregates read and write services. Command Sourcing relies on two key abstractions -- the CommandBus and the QueryBus -- which define the write and read interfaces, respectively. A Controller object (seen in Figure 14 as the Observer and Instigator) will typically provide a layer of indirection between these interfaces and the View. We use dependency injection [40] to bring the CommandBus and QueryBus as dependencies into the Controller object.
5.4.1. QueryBus

The *QueryBus* is one of two service interfaces which are injected into the high-level services and controllers. It is intended to act as the mechanism to read the current state of the system. It can maintain as little as one exposed method; one that accepts a uniquely identifying hash as its sole parameter and returns an Aggregate at that hash. Some implementations may query for objects more deeply nested than the root. For example, in the Fictional Workflow Builder application, we include a query method which allows the consumer to reference any nested member of an Aggregate recursively. Any form of open or proprietary querying language or syntax is fine as well. In the case of our Fictional Workflow Builder application, we leave it up to the high-level classes or Commands to do this work.

5.4.2. CommandBus
The *CommandBus* is the other service interface exposing methods to our high-level services and controllers. Its purpose is to act as the receiver of Command messages that ultimately mutate the state of the system. Depending on the functional requirements of the system, undo and redo may not be included in the design. The principle method of this service is the `executeCommand(Command):void` method. It accepts a Command intended to mutate the state of the system which is typically created in some high-level service class. The method does not return a value. It invokes the Commands execute method with the `QueryBus` and `TypeStoreFactory` dependencies and pushes it to the `CommandStore` for persistence.

5.4.3. DomainStore and DomainCache

The *DomainStore* simply maintains references to the state of each of our Aggregates. If we allow opening more than one collection of Aggregates, or in the case of our Fictional Workflow Builder -- Workflows, we must keep track of these collections with uniquely identifying information. We achieve this with a hashing function on the aggregate. An optional *DomainCache* gives us constant O(1) access to the Aggregates in the system.

5.4.4. CommandStore

The *CommandStore* maintains a historical view of every event up until the most recent state of the system. Due to its historical nature, the underlying data structure is perfectly suited for a Stack. If implementing undo/redo, choosing a Linked List as the Stacks underlying data structure will prevent the need to maintain both an undo and a redo stack. It does this by maintaining a pointer which enables fidelity to traverse the entire
Stack freely without mutating its data. By choosing a Stack, executing will be as simple as pushing a command onto the Stack and undoing is as simple as popping from the Stack. In TypeScript, the Array class offers Stack-like pushing and popping, but neither the Stack nor the Linked List is natively implemented, so we will have to build the data structure (in the Fictional Workflow Builder’s case, a Linked List) from scratch.

5.4.5. Command and the Concrete Command

The Command interface provides a contract for mutating the state of the system. The execution method on the contract is all that is required for simple systems. It accepts a service that gets data (QueryBus) and a service that creates new data (TypeStoreFactory.) Each concrete implementation encapsulates the code that it will use to mutate the state of an aggregate in the system.

5.4.6. Instigator and Observer

The Instigator creates an instance of Command and passes it to the CommandBus to mutate the state of the system and to be persisted. The observer binds a UI element to a domain model object provided by the QueryBus. The Instigator and the Observer are very likely the same controller class.

5.5. Anatomy of a Command

For parallel realities to be possible, all Commands must be deterministic. If they were non-deterministic, executing them in different circumstances would yield different state mutations.
Unlike Event Sourcing where “events that have happened are immutable [41],” Command Sourcing places no constraints on mutability, but the Command must be immutable after its execution and placement on the CommandStack.

Commands will likely require serialization, should they ever persist outside of memory. Therefore, all constructor parameters must be of primitive type. That way, we can leverage the *QueryBus* to retrieve the data from the *DomainCache* or *DomainStore*. When the *DomainStore* or *DomainCache* leave memory, any in memory object that lives with the Command, will lose its reference.

Command working methods require a service to query aggregates and a service to create new aggregates.

If supporting undo/redo, the undo method must put the system in a consistent state, before the Command was executed. We should isolate that test case with our automated testing. To redo a Command, re-invoking the execution method is sufficient.

Commands must be written in the finest grain. They should be limited to updating a single artifact of a single aggregate. Commands can accept additional updates as child Commands, executed after the single update to the aggregate. By maximizing granularity, we reduce the automation testing complexity since we are isolating a single property in the test. Also, we potentially lose merge functionality later by forcing more than one CRUD operation onto a Command's execute body.

5.6. Cutting / Copying / Pasting Aggregates

The functional requirements of cutting, copying and pasting are very common in today’s software systems. In a traditional CRUD based system, we might implement a
“copyable” interface on our aggregates which would duplicate them in memory. Remember though, that in Command Sourcing, we maintain a list of Commands which when executed, bring the system to its current state.

In Command Sourcing, if we follow an identical procedure for creating and updating an aggregate back to back, the result will be two different aggregates. Copying in Command Sourcing implies we are duplicating all the Commands which resulted in the creation of a particular set of aggregates. Depending on user requirements, cutting might be a different interaction. One option for a cut is a duplicate interaction (which creates the commands required to regenerate that sets aggregates) with a delete of the original. A second option is a delete with reference pointers to the original aggregate and the resulting paste would simply move that aggregate to the new parent. To the actor, the interaction would be the same but since it is a mutation to the composite structure itself, we could track and merge that as a parallel reality somewhere in the future.

In 1994, the Gang of Four gave us the concept of the Macro Command [42] -- a single Command which accepts many Command children with each mutating a particular grain of the state of the system. When duplicating a set of domain aggregates, we have two options. We can:

1. Create a Paste Macro Command.
2. Create a Serialized Blob Paste Command.

With a Paste Macro Command, we have more fidelity to optimize and merge the Commands down the road. It is also more consistent with the architecture. A high-level service would be responsible for the functionality to create these Commands. It would do
so by traversing the structure depth first and generating the Commands necessary to rebuild each aggregate. There is a complexity cost here, since each aggregate would need to be passed a unique hash. At copy time, none of those hashes exist. Usually a single target hash is available at the beginning of the Paste Command execution, and others would become available as new aggregates are created during execution. Therefore, the nested commands within the paste macro command must expose the ability to set their parents hashes if they don’t currently exist. We will need to extend the Command interface to set Command parents in the future. We should trust standard in-order tree traversal to set the parent hashes:

duplicate(aggregate: WorkflowAggregate, toParentEvent: string): CreateAggregateCommand {
    const command = new CreateAggregateCommand(aggregate.type, undefined, {
        new MoveAggregateToTargetCommand(undefined, toParentEvent)
    });

    for (let property in aggregate.properties) {
        command.updateCommands.push(new UpdatePropertyCommand(undefined, property, aggregate.properties[property].value));
    }

    for (let eventKey in aggregate.events) {
        for (let i = 0; i < aggregate.events[eventKey].length; i++) {
            command.updateCommands.push(this.duplicate(aggregate.events[eventKey][i], eventKey));
        }
    }

    return command;
}

**Figure 15: Creating a Command to Duplication an Aggregate**

If the Commands require deserialization, this service should have the ability to deserialize a Command without having all of the constructor parameters, and maintain the ability to deserialize a tree of Commands.

The Command itself will be mutable, but only right before it is executed so that it can set the parent. Caution must be taken with this approach so as to not couple your domain model to your command model. In composite structures, it might seem intuitive
to return a list of commands directly from an aggregate, but coupling of our two domains sets a dangerous precedent for the architecture. In the Fictional Workflow Builder, we delegate this work to a service, as seen in the duplicate method above. The duplication step occurs at the time the copy is executed, whereas hash assigning occurs when the paste is executed, in a similar tree traversal:

```javascript
populateHashes = (command: CreateNewAggregateCommand, toParentHash: string): void => {
  let targetHash = this.hashGenerator.createHash();
  command.targetHash = targetHash;
  for (let i = 0; i < command.updateCommands.length; i++) {
    if (command.updateCommands[i] instanceof MoveAggregateToTargetCommand) {
      command.updateCommands[i].movingHash = targetHash;
      command.updateCommands[i].parentHash = toParentHash;
    } else if (command.updateCommands[i] instanceof CreateNewAggregateCommand) {
      this.populateHashes(command.updateCommands[i], targetHash);
    }
  }
}
```

**Figure 16: Recursively Assigning Hashes to Future Aggregates**

Consider the PTO Approval Workflow Project from Chapter III. If we were to duplicate every aggregate beginning with the director approval, the result would resemble the command in Figure 17. We manipulated the hashes for readability. In Figure 17, hashes are represented by two-character strings. The first digit, a letter, denotes the breadth of the node in the tree and the second digit, a number, marks the depth of the node in the tree.

```javascript
const command =
  new CreateNewAggregateCommand("RequestInput", "b0", [
    new MoveAggregateToTargetCommand("a0", "onTimeout", "b0"),
    new UpdatePropertyCommand("b0", "user", "@employee.director"),
    new UpdatePropertyCommand("b0", "timeoutDuration", "1 Day"),
    new CreateNewAggregateCommand("PostRestApi", "c0", [
      new MoveAggregateToTargetCommand("b0", "onSuccess", "c0"),
      new UpdatePropertyCommand("c0", "url", "/api/pto/approve"),
      new UpdatePropertyCommand("c0", "body", "{employee: @employee}\"),
      new CreateNewAggregateCommand("SendEmail", "d0", [
        new MoveAggregateToTargetCommand("c0", "onSuccess", "d0"),
        new UpdatePropertyCommand("d0", "sendTo", @employee),
        new UpdatePropertyCommand("d0", "subject", "Your PTO Request"),
        new UpdatePropertyCommand("d0", "message", "... Was Approved."),
      ]),
    ]),
    new CreateNewAggregateCommand("RequestInput", "c1", [
      new MoveAggregateToTargetCommand("b0", "onTimeout", "c1"),
```
new UpdatePropertyCommand("c1", "url", "/api/pto/reject"),
new UpdatePropertyCommand("c1", "body", "{employee: @employee"},
new CreateNewAggregateCommand("SendEmail", "d1", [  
new MoveAggregateToTargetCommand("c1", "onTimeout", "d1"),  
new UpdatePropertyCommand("d1", "sendTo", "@employee"),  
new UpdatePropertyCommand("d1", "subject", "Your PTO Request"),  
new UpdatePropertyCommand("d1", "message", "... Was Rejected." )
],
new CreateNewAggregateCommand("RequestInput", "c2", [  
new MoveAggregateToTargetCommand("b0", "onFail", "c2"),  
new UpdatePropertyCommand("c2", "url", "/api/pto/reject"),  
new UpdatePropertyCommand("c2", "body", "{employee: @employee"},  
new MoveAggregateToTargetCommand("c2", "onTimeout", "d2"),  
new UpdatePropertyCommand("d2", "subject", "Your PTO Request"),  
new UpdatePropertyCommand("d2", "message", "... Was Rejected." )
]);

Figure 17: Aggregate Duplication With Macro Command

Alternatively, the Serialized Blob Paste Command is an attractive option for those who are more comfortable working with existing object copy patterns or wish to reuse library extensions which copy the state of an object. With this approach, an object is serialized and added as a member of the Command along with its new parent. Executing these commands would deserialize the blob or have special recreate extensions on each class to rebuild the aggregate from its serialized blob. Unfortunately, this does not mesh well with the Command-based architecture, yielding low cohesion and high coupling on our domain aggregates. We also lose fidelity in merging and optimizing as that process is highly dependent on fine-grained Commands. Finally, if our domain changes over time, those Commands would have to be upgraded on a “per-Command” basis. A Serialized Blob Command might resemble the code in Figure 18, below.

```javascript
const blob = {  
  type: "RequestInputCommand",  
  hash: "b0",  
  user: "@employee.director",  
  timeoutDuration: "1 Day",  
  events: {    
    onSuccess: [{    
      type: "PostRestApiCommand",    
      hash: "c0",    
      url: "/api/pto/approve",    
      body: "{employee: @employee"},    
      events: {        
        onSuccess: [{
```

38
5.7. Testing with Command Sourcing

A key benefit of Command Sourcing is that we inherit the testability benefits of a Command Sourced system from CQRS and Event Sourcing. Though the requirements regarding testability can range drastically from system to system, we aim for high testability among all our services, components and models in the Fictional Workflow Builder.

While we value automation across all facets of a system, a comprehensive suite of tests around the systems Commands provide guarantees of any state mutation. As
mentioned in Chapter V, Commands should ideally limit state mutation to the finest grain possible. That way, our tests only isolate mutation of that fine grain. Also, when supporting undo and redo, tests should confirm that undo leaves a system in a consistent state.

Figure 19 shows two tests of a Creation Command from the Fictional Workflow Builder. Since the `execute()` and `undo()` methods of a Command accept three arguments, we need some instance of a reality identifier (we can set this to some arbitrary primitive value), the TypeStoreFactory and the QueryBus. As it turns out, our unit testing framework Jasmine provides great support for mocking out our services. By creating a Jasmine Spy object, we can isolate which service methods are usable within our method under test. In the case of the `CreateNewWorkflowAggregate`, we only leverage the TypeStoreFactory’s `createAggregateByType()` and `invalidateCache()` methods, so we stub those out. In our `execute()` test, we assert that the TypeStoreFactory’s `createAggregateByType()` method gets called with the proper arguments and only gets called once when there are no child commands. In our `undo()` method test, we don’t bother with running the `execute()` method of the command. There is no requirement to do so, since we have already successfully tested the `execute()` method. To complete this test, we assert that the TypeStoreFactory’s `invalidateCache()` method is called once with the proper arguments, as that is the only way to reverse a creation of an aggregate.

```javascript
describe('In the CreateNewWorkflowAggregate Command', () => {
  let queryBus: any;
  let typeStoreFactory: any;
  const realityId = 0;
  const aggregateType = 'aggregateType';
  const targetHash = 'targetHash';
  let updateCommands: Array<Command>;

  beforeEach(() => {
    queryBus = jasmine.createSpy('queryBus');
  });
```
typeStoreFactory = jasmine.createSpyObj('typeStoreFactory', ['createAggregateByType', 'invalidateCache']);

describe('when there are no child commands', () => {
  beforeEach(() => {
    this.updateCommands = [];
  });

  it('Execute only creates one aggregate', () => {
    let command = new CreateNewAggregateCommand(aggregateType, targetHash, updateCommands);
    command.execute(realityId, queryBus, typeStoreFactory);
    expect(typeStoreFactory.createAggregateByType).toHaveBeenCalledTimes(1);
    expect(typeStoreFactory.createAggregateByType).toHaveBeenCalledWith(realityId, aggregateType, targetHash);
  });

  it('Undo leaves the system in a consistent state', () => {
    let command = new CreateNewAggregateCommand(aggregateType, targetHash, updateCommands);
    command.undo(realityId, queryBus, typeStoreFactory);
    expect(typeStoreFactory.invalidateCache).toHaveBeenCalledTimes(1);
    expect(typeStoreFactory.invalidateCache).toHaveBeenCalledWith(realityId, targetHash);
  });
});

Figure 19: Testing Command State Consistency

When designing a service layer, a domain model or UI components, careful consideration should be placed on low coupling of our dependencies to prevent unnecessary boilerplate within an automation suite in any type of system. Most of the critical services in Command Sourcing, like the CommandBus and the QueryBus, couple themselves to domain model objects or very low-level services and defer most or all of their work. This ensures high test maintainability for our most important services.

A common test theme for services, domain models and UI components is identifying state mutation. Recall that mutation only occurs by the execution or undoing of a Command object, which can only get executed by the CommandBus service. One way to test this is by mocking out every class dependency and calling every method on the subject under test. We would assert that no mutation occurred on any state-storing input
dependency. A trivial example is shown in Figure 20, a state mutation test of the \textit{ViewState} service in the Fictional Workflow Builder. No assertions are necessary as any read or write of our mutable aggregate would throw an exception.

```javascript
describe('In the View-State service', () => {
  let aggregate: any;
  let event = 'event';
  let realityId = 0;

  beforeEach(() => {
    aggregate = jasmine.createSpy('aggregate');
  });

  it('ensure the state of aggregate never mutates', () => {
    let ViewState = new ViewState();
    ViewState.setSelectedAggregate(aggregate, event, realityId);
    ViewState.clearSelectedAggregates(realityId);
    ViewState.getSelectedAggregate(realityId);
    ViewState.getSelectedEvent(realityId);
    ViewState.setDraggedAggregate(aggregate);
  });
});
```

\textbf{Figure 20: State Mutation Test of the Fictional Workflow Builder’s ViewState Service}

5.8. The New TypeStore Factory

Whether deserializing a large list of Commands or creating an Aggregate from its string type, maintaining factories of constructor mappings is tedious. For abstractions with potentially many implementations, like our Command abstraction, we risk defect injection from not properly maintaining the dictionary of types. We also violate the Open/Closed Principle.

Ideally, creating a Command or Aggregate object and loading that module should be enough to populate a factory’s dictionary.

The pattern itself consists of 2 parts: the TypeStore and the TypeStoreFactory. The TypeStore acts as a cache for mapping the name of an Aggregate to its constructor. The
TypeStore enables static placement and retrieval of these key/value pairs. When an aggregate is defined, a line of code afterward will place it in the TypeStore cache:

```javascript
TypeStore.put(SendEmailWorkflowAggregate);
```

**Figure 21: Caching Available Types For the TypeStore Factory**

The actual factory service is centralized around a single create method:

```javascript
createAggregate = <T extends WorkflowAggregate>({ aggregate: { new (hash: string): T; }, realityId: number, hash: string}): T => {
    if (this.domainCache.get(realityId, hash))
        throw new Error(`type already exists at ${hash}`);
    const newAggregate = new aggregate(hash);
    this.domainCache.insert(realityId, hash, newAggregate);
    return newAggregate;
}
```

**Figure 22: TypeStore Factory Creation Method**

This generic method accepts a type constructor and a hash used to create the aggregate. Since the strongly typed constructor may not be available at design time, a shim will map the string type to its constructor leveraging the mapping in the TypeStore:

```javascript
createAggregateByType = (realityId: number, stringType: string, hash: string): any => {
    const type = TypeStore.get(stringType);
    return this.createAggregate(type, realityId, hash);
}
```

**Figure 23: TypeStore Factory String Type Creation Method**

Since Command Sourcing may support undoing the creation of an aggregate, we must support the invalidation of the DomainCache:

```javascript
invalidateCache = (realityId: number, hash: string): void => {
    this.domainCache.remove(realityId, hash);
}
```

**Figure 24: Aggregate Cache Invalidation**
In TypeScript, we can auto-generate a dependency module containing all of our Commands and Aggregates with a JavaScript build automation tool like Gulp [43]. We can load that module with a module loader like SystemJS [15].
CHAPTER VI
MAINTAINING PARALLEL REALITIES WITH COMMAND SOURCING

In Chapter IV, we have listed a few key benefits of CQRS and Event Sourcing such as testing, diagnostics, and debugging. If our goal is to enable and support parallel states over time, we must be careful not to trample on the traits that make CQRS and Event Sourcing powerful.

Recall our definition of parallel realities: parallel realities as modelled in software as the past, future or any prior alternative junction of the current state of a system. Namely:

1. We can undo actions taken on the state.
2. We can replay actions that were previously undone on the state or take new actions.
3. We can fork into an independent reality, operating independently of its parent.
4. We can merge forked realities back with their parent.

Understanding what our approach might look like is the first step. In the Fictional Workflow Builder application two modified realities, or snapshots from a common archive
of commands, will merge a unique set of histories with a common parent. Those histories can last seconds, days, months, or years. Upon completion of both merges, their parent, the set of aggregates from which the snapshots were taken, yields a compound state of that set of aggregates.

For example, actor A can create a reality with x original aggregates. Actor B can fork that reality, modify details of each of the x original aggregates and also add aggregates of their own. During that same timeframe, actor A can add to their x original aggregates and also modify details of each of the x original aggregates. When actor A and actor B merge their forked domains, the final domain will maintain a compound state of changes made by both actor A and also actor B.

6.1. Underlying Data Structures

A reality in Command Sourcing is a Stack of Commands. A pop off the stack will apply an undo while a push will apply new behavior to the state. In TypeScript, we are given the option to use an array which natively supports Stack-like pushing and popping or implement a Linked List with a Stack interface. We opted for the latter approach in the Fictional Workflow Builder application. By leveraging a Linked List as our Stack implementation, we will not have to maintain two Stacks for undoing and redoing:

```typescript
class LinkedNode {
  previous: LinkedNode;
  next: LinkedNode;

  constructor(private readonly command?: Command) { }

  getCommand = () => {
    return this.command;
  }
}
```

Figure 25: Linked List Implementation in TypeScript
6.2. Optimizing Command Stacks

A goal of Command Sourcing was discovering the minimal set of Commands which yields an identical state in the domain. This can be beneficial for many reasons. First, reducing the size of a Command Stack yields a lighter footprint. Second, the number of conflicts that occur over time would be minimized as those conflicting Commands may no longer be relevant. Also, pruning irrelevant Commands will simplify upgrading Commands in the future as upgrading can be integrated into the same process. Optimization does not and should not exist in Event Sourcing or non-deterministic systems, since the Event Store is immutable. Our goal can yield small advantages in parallel realities, but while designing the Fictional Workflow Builder, we found that we could avoid having to optimize our Stack before forking.

We have identified three directives, listed below, to follow when optimizing Command stacks:

1. An aggregate deletion Command will nullify every Command which targeted that aggregate, including all other aggregate Commands that were moved under it or under one of its dependents, with the exception of the deletion if not paired with a creation.

2. Only the last update Command for an aggregates property is relevant.

3. Only the last move Command for an aggregate is relevant.

Based on these directives, we must design a data structure that combines “like” commands and prunes them if they are no longer relevant. We can apply a topographical sort (like Kahn’s algorithm [44]) to create a dependency tree. However, the simplest of
these runs asymptotically in $O(V+E)$, were $V$ is the number of vertices and $E$ is the number of edges in a graph. Additionally, we would have to create a graph from our current Stack of Commands. By maintaining a hash table of aggregate partitions, we are given $O(1)$ access to the partition references when we construct the tree. This dependency tree generation algorithm runs asymptotically in $O(n)$ time and requires $O(n)$ space to run. In the Fictional Workflow Builder, this was our partition step which enables us to easily prune Commands. We run the algorithm in this order:

1. Flatten the stack of Commands by removing nested Commands and storing them as siblings immediately after their parent.
2. Partition each Command into a structure and store those in a tree.
3. Apply Rule 1 and 3 by pruning all dependent Commands of all deleted Commands.
4. Apply Rule 2 by consolidating all the update Commands for a particular aggregate property to its final update Command.
5. Rebuild an optimized Command array applying each partition of Commands to the array in order (Create’s, Move’s, Update’s, Delete’s).
6. Return the intersection of the original flattened Stack (from step 1) with the optimized array of Commands (from step 5).

Figure 26 depicts this algorithm applying to a subset of the PTO Approval Sample Project from Chapter III, with a condensed Command syntax. Imagine if an original version of the workflow updated the PTO directory with a compiled binary as opposed to a REST API call and after it was completed, an additional email was sent to the companies.
HR manager. Also, the timeout duration was one week. The success path of the Command Stack would get optimized, like this:

Create("RequestInput"), Update("a0", "user", @employee.manager), Update("a0", "timeoutDuration", "1 Week"), Move("a0", "root"),
Create("ExecuteBinary"), Update("b0", "location", "C:\approve"), Update("b0", "parameters", "None"), Move("b0", "a0", "onSuccess"),
Create("SendEmail"), Update("c0", "sendTo", @employee), Update("c0", "subject", "Your PTO Request"), Update("c0", "message", "... Was approved."), Move("c0", "b0", "onSuccess"),
Create("SendEmail"), Update("c1", "sendTo", @hrManager), Update("c0", "subject", "@employee's PTO Request"), Update("c0", "message", "... Was approved."), Move("c1", "b0", "onSuccess"),
Create("PostRestApi"), Update("b1", "url", "/api/pto/approve"), Update("b1", "body", "{employee: @employee}"), Move("b1", "a0", "onSuccess"),
Move("c0", "b1"), Delete("b0"), Update("a0", "timeoutDuration", "1 Day")

```
a0 : {
  Create: Create("RequestInput"),
  Move: Move("a0", "root"),
  Update: [ Update("a0", "user", @employee.manager), Update("a0", "timeoutDuration", "1 Week"), Update("a0", "timeoutDuration", "1 Day") ],
  Delete: undefined
},
b0 : {
  Create: Create("ExecuteBinary"),
  Move: Move("b0", "a0", "onSuccess"),
  Update: [ Update("b0", "location", "C:\approve"), Update("b0", "parameters", "None") ],
  Delete: Delete("b0")
},
c1 : {
  Create: Create("SendEmail"),
  Move: Move("c1", "b0", "onSuccess"),
  Update: [ Update("c1", "sendTo", @hrManager), Update("c0", "subject", "@employee's PTO Request"), Update("c0", "message", "... Was approved." ) ],
  Delete: undefined
},
c0 : {
  Create: Create("SendEmail"),
  Move: Move("c0", "b0", "onSuccess"), Move("c0", "b1"),
  Update: [ Update("c1", "sendTo", @hrManager"), Update("c0", "subject", "@employee's PTO Request"), Update("c0", "message", "... Was approved." ) ],
  Delete: undefined
},
b1 : {
  Create: Create("PostRestApi"),
  Move: Move("b1", "a0", "onSuccess"),
  Update: [ Update("b1", "url", "/api/pto/approve"), Update("b1", "body", "{employee: @employee}" ) ]
  Delete: undefined
}
```

Create("RequestInput"), Move("a0", "root"), Update("a0", "user", @employee.manager),
Update("a0", "timeoutDuration", "1 Day"), Create("SendEmail"), Move("c1", "b0", "onSuccess"),
Update("c1", "sendTo", @hrManager"), Update("c0", "subject", "@employee's PTO Request"), Update("c0", "message", "... Was approved." ), Create("PostRestApi"),
Figure 26: Depiction of the Command Optimization Algorithm

The Workflow began with 25 Commands. We optimized the Command Stack down to 13 total Commands.

This algorithm runs asymptotically in $O(n)$ time, where $n$ is the number of Commands on the stack. By delegating most of the pruning and consolidation work to partitioning algorithms and data structures, the implementation is succinct and reads well:

```javascript
optimize = (originalCommands: Array<Command>): Array<Command> => {
  originalCommands = this.flattenCommandStack(originalCommands);
  let partitions = this.buildPartitionTree(originalCommands);
  partitions = this.pruneDeletedDependencies(partitions);
  let consolidatedPartitions = this.consolidate(partitions);
  let optimizedStack: Array<Command> = [];
  for (let hash in consolidatedPartitions) {
    if (consolidatedPartitions.hasOwnProperty(hash)) {
      optimizedStack = optimizedStack.concat(consolidatedPartitions[hash].getOrderedCommands());
    }
  }
  const intersection = originalCommands.filter((c: Command) => optimizedStack.indexOf(c) !== -1);
  return intersection;
}
```

Figure 27: Optimizing The Command Stack In TypeScript

6.3. Forking Strategy

A reality consists of two pieces: Its current history and all of its parents’ histories. In the Fictional Workflow Builder, we display this information summarized as two numbers; an archive count and a current count. When forking, the act of creating a new reality from an existing reality, we simply concatenate the Commands of each of the existing realities parents with the current set of Commands. In the newly created reality, we execute each Command in that concatenated list, yielding the exact state of its parent. At that point, each can be modified in parallel.
Forking a reality from depth 1 or more is no different than forking from depth 0. But since the reality at depth 1 or greater doesn’t store all the commands that make up its entire history (it only stores commands from a certain snapshot,) we will ask its parent reality to tell us its full history. We’ve demonstrated this, below.

```javascript
const reality = this.commandBus.getReality(realityId);
const childrenRealities = reality.getChildren()[0];
let commands = reality.getArchive().concat(reality.getCurrent());
```

**Figure 28: Discovering a Realities Full History**

Figure 29 below shows four different realities, each representing a parallel reality in time.

![Figure 29: Forking Command Stacks](image)

6.4. Concurrency Strategy

Merging histories adds some complexity, but is fully supported by Command Sourcing.

We have identified three basic merge ordering types. Those are:

1. Pre-Order: A Pre-Order merge suggests the changes at the less stable reality are less important than those at the more-stable reality. These might include visual changes to the domains mutated state, verbiage changes, or changes
the actor anticipates being added later. Remember, merging first enables any subsequent change to overwrite those changes later in history. If they already happened, those state mutations would be lost. One example in our Fictional Workflow Builder application, is updating an email sender node to display different text. A Pre-Order merge can be leveraged for defect remediation’s at the less stable branch.

2. In-Order: An In-Order merge requires UTC dates stored on the Command abstraction. That way, we know exactly when those Commands occurred and can merge each one in the order that it happened. However, we are still in a separate reality building off a snapshot of some previous state. By connecting each client and operating on a single reality we can work in a single connected reality, though Event Sourcing might be a better fit if this is a requirement.

3. Post-Order: A Post-Order merge is the most common merging strategy. It suggests that we are adding a set of behaviors to supplement the provided functionality or remove provided functionality. In our Fictional Workflow Builder application, this might include adding children nodes or deleting nodes. A Post-Order merge can also be leveraged for defect remediation’s on the more stable branch. In the Fictional Workflow Builder, we apply only Post-Order Merges.

Since the realities will be modified in parallel, and decisions might be made based on stale data, there is a risk in choosing one option and applying it without conflict resolution. This would yield a system where one actor always loses their changes (basic
optimistic concurrency.) With proper conflict resolution, the merge-type would become an implementation detail and would not affect the outcome.

6.5. Conflict Resolution

We have already discussed Command optimization, and conflict resolution in Command Sourcing takes advantage of those concepts. Conflict types correspond to Command types -- Creates, Move, Updates, Deletes -- with one exception. In Command Sourcing, Create Commands generate a unique hash, which implies their target is a unique aggregate. If, in parallel realities, two identical aggregates are created, we must determine whether they are conflicting. The Fictional Workflow Builder will treat them as different, but Command Sourcing can support merging of these aggregates.

The algorithm to find conflicts runs in O(n) time, where n is the number of aggregate partitions in one of the realities. The algorithm follows these steps:

1. Partition and Consolidate both stacks of Commands.
2. For each Partition in the originating reality (the reality we are merging from).
   a. If there was a change in both realities and it wasn’t an aggregate deletion.
      i. Create a conflict object if each partition has a move Command.
      ii. For each update Command in the partition.
         1. Create a conflict object if each partition has an identical update Command.
3. Return the list of conflicts.

After an actor decides on a desired Command, it can simply be applied at the end of the resulting merge stack, effectively overwriting both conflicting Commands.

6.6. Merging Up

Recall that merging commands up merges commands from the less stable reality to the more stable reality. Also, recall that in Command Sourcing, we always work from a given snapshot of Commands. The algorithm to merge Commands up is as follows:

1. Identify and resolve conflicts between the two realities.
2. Concatenate the current list of Commands we are merging from with the current list of Commands we are merging to.
3. Undo all current Commands on the reality we are merging to.
4. Clear all existing current and archived Commands on the reality we are merging from.
5. Apply concatenated list of Commands to the reality we are merging to as that realities current set of Commands.
6. Retrieve and apply all prior history as archived history to the reality we are merging from, including the newly merged history from this realities parent.

Steps two through six are displayed below, from the Fictional Workflow Builder application:

```javascript
postOrderMergeUpWorkflow = (fromReality: CommandReality, toRealityId: number) => {
  let toReality = this.commandStore.findReality(toRealityId);
  let fromCommands = fromReality.getCurrent();
  let toCommands = toReality.getCurrent();

  let allCommands = this.cloneCommands(toCommands.concat(fromCommands));
```

54
for (let command of allCommands) {
    this.commandBus.executeCommand(toRealityId, command);
}

let newArchive = toReality.getArchive().concat(toReality.getCurrent());
for (let command of newArchive) {
    this.commandBus.executeCommand(fromReality.getId(), command, true);
}
fromReality.setArchive(newArchive);

**Figure 30: Merging A Reality Up With It’s Parent**

Upon completion of this merge, we’ve successfully maintained a parallel reality for a given period and then terminated that reality by combining it with the parent from which it was created.

Figure 31 demonstrates an up-merge. We’ve focused on Realities 2 and 3 from Figure 29 above. After executing three Commands, we fork into a parallel reality. During that time, Reality 2 executes three more Commands and Reality 3 executes 7 Commands independent of Reality 2’s three parallel Commands. When Reality 3 merges up to Reality 2, the algorithm will apply ten total Commands as part of Reality 2’s current history.

![Figure 31: Merging Up](image)

6.7. Merging Down

The process of merging down in Command Sourcing is subtly different from merging up. When merging down, the history of Commands from the more stable reality
are applied to the less stable reality, before re-applying the less stable realities current Commands. The algorithm is slightly less complex than merging up:

1. Retrieve entire history by concatenating the archive with the current set of Commands in the more stable reality.
2. Copy current set of Commands in the less stable reality.
3. Clear all existing state by undoing the current set and the archived set of Commands.
4. Apply the Commands from step 1 as the less stable realities archive and re-apply the copied Commands from step 2.

In step 4, we must be careful when applying Commands since the realities history is now volatile: Commands might reference aggregates that may no longer exist. When applying Commands to a volatile history, we wrap these executions in a try/catch statement, capturing errors as warnings that the actor can usually safely ignore.

```javascript
mergeDown = (realityId: number) => {
  const reality = this.commandBus.getReality(realityId);
  const childrenRealities = reality.getChildren();

  for (let childReality of childrenRealities) {
    let commands = this.cloneCommands(reality.getArchive()).concat(reality.getCurrent());
    let childRealityId = childReality.getId();
    let originalCommands = this.cloneCommands(childReality.getCurrent());

    this.commandBus.clear(childReality.getId());

    for (let command of commands) {
      this.commandBus.executeCommand(childRealityId, command, true);
    }
    childReality.setArchive(commands);

    for (let originalCommand of originalCommands) {
      try {
        this.commandBus.executeCommand(childRealityId, originalCommand);
      } catch (e) {
        console.log(`Error: ${e}`);
      }
    }
  }
}
```

**Figure 32: Merging A Reality Down With It’s Children**
We can visualize in Figure 33, the change in history that occurred. We still maintain a parallel reality between reality 2 and reality 3, but apply the state changes from reality 2 to reality 3. We can liken it to re-hydrating [45] our aggregates; before we merged down, we were working off stale data.

![Figure 33: Merging Down](image)

6.8. The Stack Dirty Flag

For some systems, it might be valuable to compare two stacks of Commands to determine if changes were made. With Command Sourcing, determining if a stack is dirty is as simple as checking its size. If there are more than one Commands on the stack, or if the stack pointer is undefined, then the stack is clean. Otherwise it is dirty. This approach works because we don’t persist the entire stack of Commands, just the new modifications to the state.

6.9. Hashing Aggregates

Command Sourcing requires globally unique hashes for its aggregates. For most platforms, the Universally Unique Identifier or UUID is a suitable approach. The UUID gives us 128 bits of uniqueness, which is astronomically high. In the Fictional Workflow Builder, we used a 16-digit random number and encoded it as a base-64 ASCII string for our hashing function, since UUIDs don’t exist in TypeScript.
Figure 34: Base-64 16-Digit Random Hashing Function

It is considered best practice in Command Sourcing to check every newly generated hash against the current cache to ensure there are no collisions. A collision, while highly unlikely, will result in highly unusual behavior.
CHAPTER VII

CONCLUSIONS

At the beginning of our efforts in creating parallel realities with CQRS and Event Sourcing, we recognize the need for an extension to those architectures, to reduce the complexity that makes parallel realities unmanageable. Thus, we’ve designed a superset of the two architectures, which preserves the key ingredients of the original formula, while also simplifying the structure, and providing us the traits to build models of our software domain in parallel.

The Command Sourcing architectural pattern has been validated through one year of development of the Fictional Workflow Builder. The TypeStore factory complements the framework when delivered over platforms that required intelligent runtime loading of modules like the web.

Command Sourcing excels under deterministic systems where we allow additional clients to mutate the state of our systems. Each client can maintain its own parallel reality until they decide they want to merge their functionality.
Future work will focus on Command Sourcing framework implementations which would not require full implementations for every system. Also, we plan on applying Command Sourcing to other platforms and languages. Finally, we will test the merits of Command Sourcing across occasionally connected nodes.

To conclude, this thesis provides the following contributions to the field of Software Engineering:

- Command Sourcing -- an architectural extension of CQRS and Event Sourcing.
- A set of constraints that warrant the use of Command Sourcing in software systems.
- The necessary algorithms to achieve parallel realities in Command Sourcing and,
- A framework and implementation of Command Sourcing written in TypeScript along with the TypeStore Factory, a new factory pattern for dynamically loading module dependencies in TypeScript.
BIBLIOGRAPHY


[27] E. Gamma, R. Helm, R. Johnson and J. Vlissides, in Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994, pp. 185-193.


[42] E. Gamma, R. Helm, R. Johnson and J. Vlissides, in Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994, pp. 234-235.


