Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Jarno Rantanen

Isolation Mechanisms for Web Frontend Application Architectures

Master's Thesis
Espoo, August 10, 2015

Supervisors: Professor Heikki Saikkonen
Instructor: Risto Sarvas, docent, D.Sc.(Tech)

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Jarno Rantanen

Title:
Isolation Mechanisms for Web Frontend Application Architectures

Date: August 10, 2015 **Pages:** 87

Professorship: Software Systems **Code:** T-106

Supervisors: Professor Heikki Saikkonen

Instructor: Risto Sarvas, docent, D.Sc.(Tech)

Traditional backend-oriented web applications are increasingly being replaced by frontend applications, which execute directly in the user's browser. Web application performance has been shown to directly affect business performance, and frontend applications enable unique performance improvements. However, building complex applications within the browser is still a new and poorly understood field, and engineering efforts within the field are often plagued by quality issues.

This thesis addresses the current research gap around frontend applications, by investigating the applicability of isolation mechanisms available in browsers to frontend application architecture. We review the important publications around the topic, forming an overview of current research, and current best practices in the field. We use this understanding, combined with relevant industry experience, to categorize the available isolation mechanisms to four classes: state and variable isolation, isolation from the DOM, isolation within the DOM, and execution isolation. For each class, we provide background and concrete examples on both the related quality issues, as well as tools for their mitigation. Finally, we use the ISO 25010 quality standard to evaluate the impact of these isolation mechanisms on frontend application quality.

Our results suggest that the application of the previously introduced isolation mechanisms has the potential to significantly improve several key areas of frontend application quality, most importantly compatibility and maintainability, but also performance and security. Many of these mechanisms also imply tradeoffs between other quality attributes, most commonly performance. Future work could include developing frontend application architectures that leverage these isolation mechanisms to their full potential.

Keywords: web, browser, frontend, architecture, quality, isolation

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Jarno Rantanen		
Työn nimi:	Eristämismekanismeja selainpohjaisille ohjelmistoarkkitehtuureille		
Päiväys:	10. elokuuta 2015	**Sivumäärä:**	87
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvojat:	Professori Heikki Saikkonen		
Ohjaaja:	Dosentti Risto Sarvas, TkT		

Perinteisiä palvelinorientoituneita verkko-ohjelmistoja korvataan kiihtyvällä vauhdilla selainpohjaisilla ohjelmistoilla. Verkko-ohjelmistojen suorituskyvyn on osoitettu vaikuttavan suoraan yritysten tulokseen, ja selainpohjaiset ohjelmistot mahdollistavat huomattavia parannuksia suorituskykyyn. Monimutkaisten selainpohjaisten ohjelmistojen rakentaminen on kuitenkin uusi ja huonosti ymmärretty ala, ja sillä tapahtuva kehitystyö on ollut laatuongelmien piinaamaa.

Tässä diplomityössä täydennetään puutteellista tutkimusta selainpohjaisista ohjelmistoista tutkimalla selaimista löytyvien eristysmekanismien soveltuvuutta näiden ohjelmistojen arkkitehtuurin parantamiseen. Käymme läpi tärkeimmät alan julkaisut muodostaen yleiskuvan tutkimuksen tilasta ja parhaiksi katsotuista käytännöistä alan harjoittajien keskuudessa. Yhdistämällä kirjallisuuskatsauksen tulokset omaan työkokemukseemme alalta, luokittelemme selainten käytettävissä olevat eristysmekanismit neljään kategoriaan: tilan ja muuttujien eristäminen, eristäminen DOM:ista, eristäminen DOM:in sisällä sekä suorituksen eristäminen. Käsittelemme tämän jälkeen löydetyt kategoriat sekä esitämme niihin liittyviä konkreettisia laatuongelmia sekä työkaluja näiden ongelmien ratkaisuun. Lopuksi arvioimme näiden eristysmekanismien vaikutusta selainpohjaisten ohjelmistojen laatuun ISO 25010 -laatustandardin avulla.

Tuloksemme osoittavat että työssä esitettyjen eristysmekanismien käyttö saattaisi parantaa ohjelmistojen laatua usealla tärkeällä alueella. Näistä merkittävimpiä ovat yhteensopivuus ja ylläpidettävyys, mutta hyötyjä voitaisiin saada myös suorituskyvyn sekä tietoturvan parantumisella. Toisaalta monet esitellyistä mekanismeista myös vaativat kompromisseja muiden laatuvaatimusten osalta. Jatkotutkimusta tarvittaisiin selainpohjaisista arkkitehtuureista, jotka hyödyntäisivät paremmin työssä esitettyjä eristysmekanismeja.

Asiasanat:	web, selain, frontend, arkkitehtuuri, laatu, eristys
Kieli:	Englanti

Acknowledgements

Well, this took a bit longer than I originally expected... Still, good things come to those who wait. Even though — as it turned out — more than waiting was actually required.

In all seriousness, I want to thank my professor Heikki Saikkonen, for helping me iterate on the topic, and cut it down into something manageable.

I want to thank my employer Futurice, both for providing a Thesis Boot Camp for getting started with this project, and then providing such darn interesting work opportunities that I promptly forgot about my thesis for a few years. But the Boot Camp was the start of a journey, in more ways than one.

I also want to thank Olli Jarva, and especially Marja Käpyaho, for your persistence in ~~kick~~gently reminding me about getting back on track, and your help in reviews and everything.

KIITOS 18.6.2012–10.8.2015

Jarno Rantanen

Abbreviations and Acronyms

WWW	World Wide Web
URL	Uniform Resource Locator
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
IP	Internet Protocol
SPA	Single-Page Application
RIA	Rich Internet Application
DOM	Document Object Model
API	Application Programming Interface
IEEE	Institute of Electrical and Electronics Engineers
SOP	Same-Origin Policy
UX	User Experience
IIFE	Immediately-Invoked Function Expression
UI	User Interface
GPU	Graphics Processor Unit

Contents

Chapter 1

Introduction

During the last two decades, the Internet has arguably been the largest single disruption in how the civilized world exchanges information. For the masses, its main facet is the World Wide Web (WWW): a massive, interlinked collection of interactive documents. Towards the latter part of the past decade, the interactivity of these documents has increased dramatically, to the point where the term "document" no longer accurately describes them; indeed, they are increasingly becoming interlinked *applications*, which are installed to and executed in a web browser when the user types in their URL.

While these browser-based *web frontend applications* are already taking over the World Wide Web (and by extension the world), they are only very recently being recognized as applications in their own right [28]. It is becoming evident that the same rigor should be applied to their design and architecture, as has been to traditional applications in the decades of software development efforts that preceded them. This neglect has lead to a wealth of quality problems with frontend applications, and it seems what can be achieved purely within the browser is not limited by the technologies available, but by the lack of architecture allowing these applications to grow in complexity.

It turns out many of the aforementioned quality issues relate to the lack of isolation between various components of these applications. This work identifies the 4 major classes of isolation mechanisms available to frontend application architectures, discusses the concrete quality issues related to each, and introduces the most relevant mechanisms for addressing those quality issues. To substantiate how these isolation mechanisms can positively affect overall application quality, we also discuss their effects using a well-known quality standard, namely the ISO 25010 [18].

1.1 Background and Motivation

This section will briefly introduce the environment in which this work is relevant: web frontend applications, which run purely[1] in a web browser, and are part of the whole that is the World Wide Web. Finally, we enumerate the concrete classes of isolation mechanisms this work will address.

1.1.1 The Web

At the foundation of the scope of this work is the World Wide Web, commonly referred to as simply *the web*. The web is a global system of interconnected hypertext documents, that may contain various types of resources (images, for example), links to other such documents, styling information, and — especially importantly in the scope of this work — executable code that provides capabilities for interacting with the user. [41] These documents are typically accessed using a *web browser*, and an application which is composed of (or is capable of producing) such documents is commonly known as a *web application*.

There are numerous layers below the web facilitating its functions, such as HTTP, the Internet, the TCP/IP Protocol Suite, and the physical infrastructure making the globally interconnected web a possibility. They are, however, also far beyond the scope of this work, and a basic understanding of them is assumed from the reader.

1.1.2 The Browser

The way a typical user interacts with the web is through a web browser, commonly known as simply a browser. The role of a browser is to navigate to a specific URL, retrieve its HTML document and related resources, interpret and visually display them to the user, and execute any JavaScript[2] the document instructs it to. [41]

An important distinction between the components of a web application can be made by looking at where application code is executed, specifically whether this happens before or after delivering the code over the network to the browser. When before, we often talk about the backend part of the web

[1] By "purely" we do not preclude the involvement of systems beyond the browser; see the upcoming definition of web frontend applications.

[2] There are other languages which modern browsers can execute such as Dart (https://www.dartlang.org/), but their prevalence on the web is dwarfed by that of standard JavaScript.

application. When after, we talk about the frontend part. Using this definition, even though code (such as JavaScript) or resources (such as images) are often delivered to the browser by backend code, we still refer to such code or resources as pertaining to the frontend, as they are executed (or interpreted) by the browser.

1.1.3 Web Frontend Applications

Communication between the browser and the backend has inherent delays due to networking, and avoiding these delays can result in better application responsiveness and performance. For example, this can be achieved by:

- **Performing navigation within the browser.** When the user navigates to another document for which data is already available in the browser, instead of requesting the new document over the network, that document can be instantaneously assembled and displayed by JavaScript.
- **Updating the document speculatively.** When the user takes an action that needs to be performed by the backend, the frontend can immediately update the document as if the action already succeeded, while it is carried out in the background. If the background action can be assumed to usually succeed, this results in the perception of instantaneous response.
- **Performing actions purely within the browser.** In some cases the backend need not be involved at all in carrying out actions. For instance, if the required computation and logic can be performed by JavaScript in the frontend, the response to such actions can be near-instantaneous, without the potential complications of speculative updates.

User-perceived performance is of increasing importance to modern web applications. Amazon famously experimented with artificially slowing down their page load times, and observed a 1% loss of sales for each added 100 milliseconds. Similar experiments by Google revealed that an artificial delay of 500 milliseconds decreased their revenue by 20%. [23] These and similar observations have resulted in the conclusion that web application performance is of great importance. They have also driven many web applications to shift their focus to their frontend, where many classes of performance improvements can only be obtained (such as the ones listed above). Taking this trend to its extreme, the frontend part of the application contains the

majority of application logic, and the backend part is relegated to the role of delivering resources and data storage. Such applications are referred to as *web frontend applications*, or just frontend applications in this work.

Another common term for applications fitting the previous description is a Single-Page Application (SPA).[3] While perhaps a more commonly used term in the industry, it carries a connotation that makes it unsuitable for defining the scope of this work. The "Single-Page" part refers to a specific kind of frontend application, which is usually contained in a single HTML document, delivered with associated resources and JavaScript, and this document is typically never loaded again. While an SPA definitely fits the previous description of frontend applications, the latter also includes web applications which are composed of several HTML documents, with possible reloads during their execution. The difference is subtle, but the isolation mechanisms considered in in this work benefit a broader class of frontend applications than just the SPA. That is, we consider the former a superset of the latter, and thus we will use the term (web) frontend application for the rest of this work.

1.1.4 Isolation in Frontend Application Architectures

In the rush to benefit from the performance improvements made possible by frontend applications, these applications grew quickly in size. There was little precedent on how to build large applications for the frontend, however, and quality issues abounded (see subsection 2.5.1 Spaghetti Code).

In developing large applications, architectures that focus on isolating application components from one another have traditionally been considered valuable (see section 2.4 Isolation in Architectures). Frontend applications are especially susceptible to quality issues from lack of isolation, due to several reasons:

1. The lingua franca of frontend applications — JavaScript — has traditionally lacked many features related to enforcing isolation within architectures, such as language-level modules, or visibility modifiers for variables. This class of issues is discussed in depth in section 3.1 State and Variable Isolation.

2. By tradition, frontend applications have closely intertwined their state and application logic with their presentation logic — the DOM API

[3]Rich Internet Application (RIA) was also a popular, related term for a time, but its common use also encompassed proprietary technologies such as Flash and Silverlight, thus making it an unsuitable term for our standards-oriented discussion.

(see subsection 3.2.1 DOM Introduction) of the browser especially. This makes it hard to perform changes to either part without accidentally affecting the other. This class of issues is discussed in depth in section 3.2 Isolation from the DOM.

3. By technical necessity, most components of an application need to coexist within a single, shared DOM, with little to no access control between them. This means any faults within a component are mostly free to propagate to their neighbors, potentially causing additional faults that can be hard to diagnose. This class of issues is discussed in depth in section 3.3 Isolation within the DOM.

4. By technical necessity, most computation needs to be performed on a single thread of execution, within a single shared memory space (see subsection 3.4.1 Evented Programming on a Single Thread). As application logic is increasingly moved to the frontend, these limited computing resources become increasingly saturated, up to the point where the very benefits of moving application logic to the frontend are negated by the resulting performance problems. Due to traditional lack of memory isolation, parallelizing these workloads is complicated. This class of issues is discussed in depth in section 3.4 Execution Isolation.

This is to say the lack of isolation in architectures is both a major issue for frontend applications today, and one of the significant detractors to frontend applications reaching new levels of complexity, significance and quality.

1.2 Research Questions

The concrete questions this work addresses are as follows:

1. **What kinds of isolation mechanisms are available for web frontend application architectures?**

   This question is addressed in chapter 3 Isolation Mechanisms, where we introduce the 4 major classes of isolation mechanisms we have identified in literature, give concrete examples of related quality issues, and present mechanisms for addressing those issues through enforcing isolation.

2. **How do those mechanisms benefit web frontend application quality?**

This question is addressed in section 4.1 Relationship to Quality, where we evaluate the impact of applying the isolation mechanisms through the quality attributes defined by the ISO 25010 standard.

1.3 Research Methods

The methods of research in this work can be categorized as follows:

- **Academic literature review.** Academic research into web frontend application architecture is limited. The most relevant works — perhaps interestingly — do not have to do with the structure and architecture in frontend applications, but the lack thereof. Security and performance are also discussed, though their treatment of architectural issues is often indirect. However, application architecture and the related concept of software quality are well-researched topics on their own. Relevant work in academia is introduced in chapter 2 Previous Work.

- **Industry literature review.** Compared to academic work on the topic, both in software development industry and in various Open Source Software communities publications are frequent. In our experience, such publications are also often highly influential among developers, much more so in fact than formal research. Thus, this work often also cites technical reports, white papers, Open Source Software projects, books, technical documentation and even blog posts, where appropriate. In the absence of publications peer-reviewed by the scientific community, this work relies on recognized industry thought leaders[4], but also on organizations and popular bodies of code[5]. Relevant work in the industry is introduced in section 2.5 Frontend Application Architecture, and informs much of the main contribution of this work, chapter 3 Isolation Mechanisms.

- **Personal experience.** I, the author, have been involved in web development in one form or another for more than 15 years[6], close to 10 of that professionally[7]. The last four years I have specialized in web

[4]In the context of frontend architectures, two often cited ones are Nicholas Zakas and Addy Osmani (of Yahoo! and Google fame, respectively).

[5]Such as in widely used Open Source Software packages.

[6]Non-professional work has included various kinds of projects, from small experiments to developing and maintaining several mid-sized web sites.

[7]See http://fi.linkedin.com/in/jarnorantanen for an overview of professional work.

frontend applications, often working in the capacity of a lead developer or architect. The projects I have worked on range from single developer efforts to projects of close to a 100 000 lines of code with more than 10 full-time developers. This experience has given me insight into the topic of this work, and helps, for instance, in assessing the credibility of sources in the industry literature review. This experience also especially informs chapter 4 Discussion.

1.4 Structure of Work

In chapter 2 Previous Work, we first introduce relevant concepts and publications from the context of software architecture and software quality research. We then move on to explore the relationship between the two, and also touch upon previous work on isolation in application architectures. Finally, we present the most important publications relevant to frontend application architecture.

In chapter 3 Isolation Mechanisms, we present the main contribution of this work: the enumeration of the 4 major classes of isolation mechanisms for frontend application architectures. For each we also introduce relevant background and concepts, give examples of related quality issues, and list the concrete mechanisms available for implementing isolation within that class.

In chapter 4 Discussion, we evaluate the potential impact of the presented isolation mechanisms on application quality, using the ISO 25010 [18] quality standard as a frame of reference. We conclude by presenting a summary of our findings.

Finally, in chapter 5 Conclusions, we briefly discuss the broader implications of the topic of this work, and offer some avenues for future research.

Chapter 2

Previous Work

This chapter will first cover relevant previous work on classical software architecture, mainly to adequately define what we mean by software architecture. We then cover classical methods of modeling software quality, in preparation for the eventual discussion relating our isolation mechanisms to concrete quality improvements. Having defined architecture and quality, we move onto investigating the relationship between the two, covering some established methods of affecting quality through architectural decisions. Finally, we briefly visit previous work on using isolation in architectures.

Having adequately covered the concepts of architecture, quality and isolation in general, we move onto presenting previous publications relating specifically to the domain of this work. We cover the state of research on topics such as frontend application security, architecture, maintainability and performance. All of the topics are related to our main focus of isolation in architecture, yet there is a clear gap in research on the exact topic, as no publications were found directly discussing isolation in this context. The main body of this work will contribute directly towards narrowing that gap.

2.1 Classical Software Architecture

2.1.1 Defining Architecture

As the topic of this work relates to architecture in the context of software systems, a brief treatment of its meaning is in order. Architecture is a concept which is easy to grasp by intuition, yet difficult to formalize or define accurately, although there have been countless attempts over the years [29, p. 2]. The Institute of Electrical and Electronics Engineers (IEEE) defines it in a standard titled "Systems and software engineering — Architecture

description" as:

> Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution. [19]

It is hard to argue for or against such a definition, as it seems nearly all-encompassing, and makes it hard to say what is architectural, and even harder to say what is not.

For another take on the subject, the industry giant IBM, for their Rational Unified Process software development model at the turn of the millennium, approaches the concept through distinct views:

> Architecture is represented by a number of architectural views. These views capture the major structural design decisions. In essence, architectural views are abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside. [25]

This seems more reasonable, as depending on the viewpoint — of a developer, of a user, of a project manager — the simplified version of the system might look very different. It also brings up the interesting concept of *structure:* how things fit together to form a whole.

Martin Fowler, author of important architectural work in the context of large enterprise applications [11], offers a very simple definition of "things that people perceive as hard to change" [12]. While concise and intuitive, a simple counterexample is the choice of programming language: surely simple matters of syntax are not architecturally significant choices? Yet, often the only way to migrate a software project to a different programming language is a complete rewrite.

For the purposes of this work, let us formulate a definition that is a combination of each of the above:

> Architecture is the fundamental way in which the application is decomposed into smaller parts, how they fit together to form a whole, and how resilient that whole is in the face of inevitable change.

2.1.2 Architectural Design Research

As varied as the definitions of software architecture, are the avenues of research on the topic. While a comprehensive review of the state of software

architecture research is well beyond the scope of this work, one area is too relevant to be omitted: architectural design research.

Research into architectural design focuses on understanding early on the decisions that lead to particular architectural results at a later stage of a software development project. Recalling the earlier definition of architecture as "things that are hard to change later", and keeping in mind there are often commercial interests in software development projects, the motivation of architectural design research is to guide projects into getting their architecture right as early as possible, thus minimizing costly rework.

One classical method for bringing such architectural design and understanding into a software development project is the Architecture Tradeoff Analysis Method (ATAM) [22]. The method focuses on analyzing the main business drivers of the project, extracting and documenting the quality attributes (introduced in subsection 2.2.1 Quality Attributes) that embody those drivers, and then modeling the proposed architecture as a set of tradeoffs affecting the known quality attributes. By iterating on the architectural choices and analyzing each iteration, a suitable balance of the desired quality attributes can be achieved before the implementation phase of the application begins. Even though the Agile development movement has successfully questioned the feasibility of heavy up-front design [4] , understanding the causalities between architectural choices and the resulting quality attributes of the software system remains a worthy goal.

2.2 Classical Software Quality

2.2.1 Quality Attributes

Quality attributes are a taxonomy of desirable software characteristics, often taking the form of various "ilities", such as reliability, extensibility and portability. This taxonomy is often nested to contain various sub-characteristics: portability, for instance, can be thought to consist of sub-characteristics such as adaptability, coexistence and replaceability. [29, p. 3]

The usefulness of quality attributes in software development is broad: they can assist in understanding a software system, in facilitating discussion and trade-off analysis (see ATAM above), and in allowing derivation of concrete models for quality assessment.

2.2.2 Abstract Models

When talking about software quality, it is hard to sidestep one of the most recognized standards on the topic: the ISO 25010:2011 [18]. The standard proposes two quality models, namely Quality in Use and Product Quality. Each model defines a set of quality attributes[1], through which various properties of the system can be evaluated.

The Quality in Use model defines the attributes of the system in the context of interacting with a user. Its five top-level attributes are:

- **Effectiveness:** How accurately and completely the user is able to achieve his/her goals
- **Efficiency:** How much resources are consumed, relative to the degree to which the users' goals are met
- **Satisfaction:** How well the system satisfies the needs of the user during use (subdivided into *Usefulness*, *Trust*, *Pleasure* and *Comfort*)
- **Freedom from risk:** How well the system limits imposed risks to its users and environment (subdivided into *Economic risk mitigation*, *Health and safety risk mitigation* and *Environmental risk mitigation*)
- **Context coverage:** How well the above quality attributes are fulfilled in relation to known and unexpected contexts of use (subdivided into *Context completeness* and *Flexibility*)

The Product Quality model, on the other hand, defines the attributes for evaluating the static and dynamic quality properties of the system itself. Its eight top-level attributes are:

- **Functional Suitability:** How well the system provides functionality to satisfy pre-defined needs (subdivided into *Functional completeness*, *Functional correctness* and *Functional appropriateness*)
- **Performance Efficiency:** How well the system performs, relative to the degree of resource use (subdivided into *Time behaviour*, *Resource utilization* and *Capacity*)

[1]The standard uses the term "characteristic" in place of "quality attribute" (which is more often used in related literature); for the purposes of this work, the terms are interchangeable.

- **Compatibility:** How well the system is able to exchange information with other systems, and how it performs while operating in a shared software/hardware environment (subdivided into *Co-existence* and *Interoperability*)
- **Usability:** How well the system behaves with respect to effectiveness, efficiency and satisfaction during use (subdivided into *Appropriateness recognizability*, *Learnability*, *Operability*, *User error protection*, *User interface aesthetics* and *Accessibility*)
- **Reliability:** How well the system is able to carry out its intended function, with respect to specific conditions or the passing of time (subdivided into *Maturity*, *Availability*, *Fault tolerance* and *Recoverability*)
- **Security:** How well the system protects information so that only the intended persons or systems are granted access (subdivided into *Confidentiality*, *Integrity*, *Non-repudiation*, *Accountability* and *Authenticity*)
- **Maintainability:** How efficiently the system can be modified in the face of changing requirements (subdivided into *Modularity*, *Reusability*, *Analysability*, *Modifiability* and *Testability*)
- **Portability:** How efficiently the system can be transferred between different hardware, software or usage environments (subdivided into *Adaptability*, *Installability* and *Replaceability*)

Especially the latter model will be useful later in this work in section 4.1 Relationship to Quality, providing a framework for discussion on the quality implications of the proposed isolation mechanisms.

2.2.3 Applied Models

As useful as high-level models are for building a frame of reference, ISO 25010 operates on such an abstract level of description that it may not be easy to apply to day-to-day work. For that reason, the model proposed by the standard is often applied to specific contexts by defining context-specific quality attributes, which reduce the level of abstraction and thus make the attributes more practically useful.

One such applied quality model has been presented for the context of Service-Oriented Architectures (SOA) [14]. It evaluates related literature, extracting quality attributes[2] specifically relating to SOA, and enumerates

[2]The paper uses the term "Quality Activity"; we treat it here equivalently to quality attributes for reasons of brevity.

them citing the ISO 25010 counterparts. The presented quality attributes are:

1. **Understanding:** How easily the service is understood by a potential user (corresponds to *Usability [Appropriateness Recognizability]* in ISO 25010)
2. **Consumption:** How well the service covers its intended functionality (corresponds to *Functional Suitability [Functional Appropriateness]* in ISO 25010)
3. **Support:** How easy it is for service providers to support their users (the work does not identify a counterpart in ISO 25010)
4. **Service Reuse:** How well the service composes with other services (corresponds to *Reusability* in ISO 25010)
5. **Extension:** How easily new functionality can be added to the service (corresponds to *Reusability* in ISO 25010)

Another applied quality model has been presented specifically for web applications [34]. It especially stresses the potential for practical use during design, development and maintenance of a web site. The presented quality attributes, their proposed sub-characteristics and ISO 25010 counterparts (if any) are:

- **Architecture:** Information architecture and Navigation
- **Communication:** Brand identity, Visual design, Typography and Multimedia usage
- **Functionality:** Functional adequacy, Functional correctness and Security/privacy (corresponds to Functional Suitability in in ISO 25010)
- **Content:** Categorization/labeling, Conformity to style guide, Information/data quality, Content timeliness and Content localization
- **Community:** User relations and Community management
- **Platform:** Platform adequacy, Site availability, Site performance and Access monitoring
- **Accessibility:** Findability, Band requirements, Client independence and Usability requirements (corresponds to *Usability [Accessibility]* in ISO 25010)

- **Usability:** Effectiveness, Efficiency and User satisfaction (corresponds to *Usability [Accessibility]* in ISO 25010)
- **Software code:** Reliability, Compliance to standards and Maintainability (partially corresponds to *Reliability* in ISO 25010)

Several other works on the topic of applied quality models in the specific context of web applications also exist. [30, 31]

All of the works cited above serve as examples of the applicability of abstract quality models into more concrete contexts. A clear need for context-specificity is identified in each. The major contribution of this work — a list of isolation mechanisms for frontend applications — can also be understood as a list of highly context-specific quality attributes. This interpretation is further discussed in section 4.1 Relationship to Quality.

2.3 Relating Quality and Architecture

Architectural design decisions have direct influence on the quality of a software system [29, p. 3]. On the other hand, making decisions about the desired qualities of the software system affect architectural design [22]. While the two concepts are fundamentally linked, architectural design decisions rarely hold intrinsic value. For instance, choosing a client-server architecture does not usually produce value in itself, but rather, the qualities which derive from that choice do. For example, these qualities might present themselves as increased decoupling and interoperability. Thus, research into the relationship between quality and architecture often focuses on how to derive an architecture out of chosen quality requirements. [29]

2.3.1 Attribute Driven Design

In addition to ATAM, other examples of this research include the Attribute Driven Design method (ADD) [51]. ADD follows a Plan-Do-Check cycle in recursively decomposing and designing the architecture of the system:

- In the **Plan** phase, the desired quality attributes and other design constraints are considered to produce a set of roles, responsibilities, properties and relationships between software elements in the system.
- In the **Do** phase, the architect produces a list of patterns[3] for fulfilling the previously identified roles, properties and relationships. This list

[3]The term pattern is loosely related to the concept of Design Patterns (treated thoroughly in [11]).

can be derived from the architect's personal experience, books, papers, commercial products, et cetera.

- In the **Check** phase, the whole of the architecture is verified against the original quality constraints, to make sure the most recent iteration of the Do phase has not introduced regressions.

The process is then repeated recursively to further decompose the architecture, until a satisfactory level of detail is achieved.

2.3.2 Architectural Tactics

Another method for deriving a concrete, partial architecture out of the desired quality attributes exists in the form of Architectural Tactics [2]. Architectural Tactics are concerned with taking a concrete scenario relating to a quality attribute, and applying a repeatable and predictable process for turning the scenario into design fragments. The application architecture emerges from the combination of multiple such design fragments. As a simplified example, if a performance scenario dictates some hard limit for system responsiveness, through application of known methods for affecting responsiveness, a tactic of *bound execution times*[4] might be chosen. The concrete design fragment derived of this might be the use of a scheduler for arbitrating the execution times of system functions.

2.4 Isolation in Architectures

This work deals extensively with isolation in the context of software architecture. Little work exists specifically on the topic of isolation, but its value in software development is often discussed indirectly.

One aspect which can greatly benefit from isolation is fault tolerance. A classical example of this is isolating the address spaces of cooperating software modules. [49] The purpose is to ensure a failure in a single module does not corrupt the address space of other modules. While this will not make the root cause of the failure (such as a bug in application code) go away, proper isolation makes analysing the failure more tractable for humans, and makes it easier for other modules to reason about the failure (for example by retrying the operation that failed). There is often some overhead associated with

[4]In this context, being bound is understood as having a well-defined upper limit. For instance, a given operation is allowed to execute for at most 10 milliseconds before it is either terminated, or suspended while other operations are allowed to execute.

enforcing isolation between modules, but in many cases it can be negligible, considering its benefits.

Another beneficial aspect is security. A topical example are modern web browsers, which place increasing importance on isolating the various web applications they execute from each other. Despite the benefits of address space isolation [36], isolating each application using this method is not always possible. This is due to many existing features in the web platform that necessitate breaking this kind of isolation. But even in such cases several mechanisms exist in browsers to ensure proper isolation of applications, the most important of which is the Same Origin Policy (SOP). The SOP ensures that web applications that were started from two separate domains have limited facilities of affecting each other. [1] This feature is further elaborated on in subsection 2.5.2 Security.

Finally, isolation benefits maintainability and understandability of large software systems. Human beings have finite capacity for understanding software complexity, and sufficiently large systems can no longer be completely understood by any single individual. In such cases, isolated modules that mask complex implementation details behind a well-defined, simplified interface are one of the few tools developers have at their disposal to keep the system maintainable. [54]

2.5 Frontend Application Architecture

There is a clear research gap around frontend application architecture (as originally brought up in section 1.2 Research Questions). Thus, when discussing previous work on the subject, we are mostly limited to publications that discuss the topic indirectly.

2.5.1 Spaghetti Code

As suggested in section 1.3 Research Methods, formal research into frontend application architecture is scarce. Notably, one topic on which such research does exist is the lack of proper architectures for frontend applications.

Mikkonen and Taivalsaari note how web applications are the "spaghetti code for the 21st century" [28]. They argue many of the well-established software engineering principles contributing to software quality have been lost with the rise of web applications, specifically:

- **Modularity and related principles:** The browser platform does not adequately support separation of concerns. For instance, procedural

JavaScript and declarative HTML and CSS are mixed and matched at will, and thus also application and presentation logic. Also, besides the DOM, no well-defined interfaces exist for cooperation between UI components and the browser. Finally, while privacy and information hiding can be emulated with JavaScript, no such mechanisms are built into the language.

- **Consistency, simplicity, and elegance:** Due to the dynamic nature of the environment, mixing and matching programming languages, and the convoluted history of browser support for various API's, there are often several ways to achieve the same end-result. Also, the current generation of web applications are often unstructured and difficult to read and understand.

- **Reusability and portability:** References to resources are often hard-coded into an application, making it harder to use a part of it elsewhere. As most frontend code traditionally deals with user experience, the reliance on UI toolkits makes it difficult to transfer that user experience to other applications, as it would often imply carrying over (potentially incompatible) UI toolkits as well.

In addition to these violations of established software engineering principles, they identify two other important categories of challenges:

- **Usability:** The browser has many historical features which are poorly suited for the context of application interactions (as opposed to interacting with traditional web documents). The standard reload, stop and back buttons do not have immediately obvious and portable semantics in the context of applications, which may host complex internal state and communicate with servers in the background, often with no indication to the user of an ongoing operation.

- **Development style:** The support for static verification or type-checking of JavaScript applications is lacking, if not nonexistent. The dynamic nature of the programming environment makes it hard to apply traditional quality control methods to source code.

Mikkonen and Taivalsaari conclude, however, by noting that there are no fundamental reasons for web applications to be any worse than traditional applications. While application architectures are not explicitly discussed, in light of the previous discussion in section 2.3 Relating Quality and Architecture, it would seem many of these issues are best addressed through application of proper architectural design principles.

2.5.2 Security

Another topic on which formal research relating to frontend application architecture has been published is security. In a paper titled "Privilege Separation in HTML5 Applications" [1], Akhawe et al contrast the established security mechanisms for traditional applications with the lack of such mechanisms for web frontend applications. They note how traditional use of the Same Origin Policy — one of the major methods of privilege separation for frontend applications — is ill-suited for providing security for complex applications, due to the monetary and administrative overheads associated with domain management[5]. Instead, they propose an architecture where the SOP is used with temporary origins, allowing strong isolation and privilege separation among application components.

Application components running in temporary origins execute without the privileges granted to the main application. These privileges may include things like access to cookies associated with some server, access to browser API's such as storage or geolocation, or navigating the browser to a different URL. In order to use such privileged API's, the unprivileged components communicate with the main application via message passing. The main application acts as a mediator, implementing a security policy which dictates which components are allowed to exercise which privileged API's, and fulfills or rejects their requests accordingly.

Akhawe et al proceed to apply this security-minded architecture to a number of existing web frontend applications. They note that they are able to achieve a drastic reduction in the amount of code executing with full privileges, while requiring only minor changes to most applications. The performance penalties of the approach are also deemed negligible. In a related study on applying similar changes to browser extensions [5], the authors show a drastic reduction not only in the amount of code executed with full privileges, but in the number of concrete, exploitable vulnerabilities. The results of the studies indicate there are significant — yet underutilized — security benefits to applying the proposed architecture to web frontend applications. This proposed architecture relies heavily on the ability to provide

[5]This is due to how the Same-Origin Policy requires for either the protocol or the host to be different in order to isolate origins. Generating a new port for every component that needs isolation (for example `:8080`, `:8081`, `:8082` and so on) would not work with many firewalls, plus generates infrastructure overhead. Generating a new subdomain for every component (for instance `c1.example.com`, `c2.example.com`, and so on) would not provide complete isolation, as setting the `document.domain` property to `''example.com''` would allow the components to communicate directly. This property has no effect on top-level domains (such as setting `document.domain` to `''com''` would not allow access), but using top-level domains for containment would be prohibitively expensive.

isolation between application components.

2.5.3 Patterns and Architecture

A wealth of literature — in the form of books, articles and blog posts — exists on the topic of patterns and architectures for web frontend applications. As argued before in section 1.3 Research Methods, this work is often influential among practitioners, and thus must be considered. While many cited works date back several years[6], they remain both influential and relevant. The former is due to how they have essentially shaped the collective thinking of the industry. The latter is due to how architectural patterns stand the test of time: while implementations are outdated quickly in the context of frontend applications, ideas significantly less so. This section introduces some of the most relevant authors on the topic of patterns and architecture, and some their most influential works.

Nicholas Zakas was one of the globally visible front-runners in developing and championing proper architectural design in the emerging context of frontend applications. Zakas was professionally part of Yahoo![7] for a long time, and attributes much of his insight on the topic to his experiences in leading large-scale frontend development projects there [54]. One particularly influential presentation given in 2009 on the topic of "Scalable JavaScript Application Architectures" [52] remains an often-cited source of inspiration, even among other industry thought leaders (cited as such in [32], for instance). In the presentation, Zakas talks about structuring Single-Page Applications (a notably recent trend at the time), treating concepts such as modularity, loose coupling, security through message passing, and importantly, sandboxing. Sandboxing, as per Zakas, has to do with isolating components of the application so that they cannot interfere with one another during application execution. This includes isolating the state, variables and the DOM of components from each other. These same concepts are treated in great detail later in this work, in chapter 3 Isolation Mechanisms.

Another important figure in the context of frontend architectures is Addy Osmani, who has — like Zakas — contributed several books and dozens of

[6]The frontend application development community is perhaps notorious for how often it changes its collective views on best practices, et cetera; we cannot resist the urge to point the interested reader to http://www.reddit.com/comments/2kl88s, where several frontend practitioners react to the news (2014-10-29) that one of the largest frontend frameworks of the time — AngularJS — is going to completely break backwards compatibility with its next major version, even as many have only recently invested heavily into the current version.

[7]http://www.linkedin.com/in/nzakas

blog posts to improving the craft. Osmani's position in developer relations at industry giant Google[8] has recently lent him additional visibility. Osmani's 2011 write-up on the topic of "Patterns for Large-Scale JavaScript Application Architectures" [32] brings together many related publications and concepts, and frames the discussion with relevant Design Patterns. The patterns Osmani considers most important for large-scale frontend applications are:

- **The Module Pattern:** The primary purpose of the pattern is to provide encapsulation and code organization for the architecture. This includes emulating privacy and using namespace objects for organizing variables (see section 3.1 State and Variable Isolation for related discussion in this work).
- **The Facade Pattern:** A Facade provides an abstraction layer on top of a concrete module implementation, making it possible for that module to change or be completely replaced, without affecting other modules in the system. It is somewhat analogous to the function of interfaces in less dynamic languages.
- **The Mediator Pattern:** To tie the proposed architecture together, a Mediator provides methods for various modules to communicate, without having direct access to each other (see subsection 3.4.5 Message Passing for related discussion in this work).

Osmani expands on the topic in a full-length book titled "Learning JavaScript Design Patterns" [33], which covers non-architectural patterns as well.

Other noteworthy literature on the topic of patterns and architecture includes books such as Stoyan Stefanov's "JavaScript Patterns" [38]. The book goes into great detail especially on practical implementations and implications of using many patterns in the browser, not just in theory, or in JavaScript environments in general.

Finally, this list of authors and works makes no attempt at being exhaustive: there is a wealth of important authors and publications left out. Osmani provides a high-quality starting point [32] to exploring this space further.

2.5.4 Maintainability

As previously argued in section 2.3 Relating Quality and Architecture, the two concepts are fundamentally linked. One important aspect of quality is

[8] http://uk.linkedin.com/in/osmani

maintainability, and perhaps due to its traditional negligence in frontend applications, several well-known books and other publications have been devoted the treatment of maintainability in this context.

One of the seminal works on maintainable JavaScript is Douglas Crockford's 2008 book "JavaScript: The Good Parts" [7]. Indeed, it is still often cited as one of the best starting points to learning professional-grade JavaScript development. The author argues that the language contains many features which should be avoided in order to produce JavaScript code that is free from maintainability issues. These claims are codified into a subset of the language, which includes features like simple objects, closures and higher-order functions. The parts of the language the author advises to avoid include global variables, which are discussed at length later, in section 3.1 State and Variable Isolation.

"Maintainable JavaScript" [54] by Zakas also contains a thorough treatment of programming practices for building robust and maintainable frontend applications. He discusses topics such as loose coupling, separation of concerns, and the tendency of JavaScript towards global variables and state. This discussion is foundational to several of the isolation mechanisms presented in chapter 3 Isolation Mechanisms.

2.5.5 Performance

As frontend applications have an intimate relationship with User Experience (UX), and as performance is an especially important part of good UX (see subsection 4.1.2 Performance Efficiency for related discussion), several books have been devoted to the topic.

One of the classic works on the topic is, again, by Zakas, titled "High Performance JavaScript" [53]. It contains a holistic overview of the fundamentals in achieving high performance on the browser platform. Even though the performance of web browsers continues to improve rapidly, much of the book is as relevant today as it was when published in 2010. This is because the fundamentals of frontend performance — avoiding round-trips[9], DOM-thrashing and expensive work on the UI-thread[10] — have improved through gradual evolution, rather than a revolution making previous advice obsolete. These performance fundamentals lay the groundwork for much

[9]Network access is still many orders of magnitude more expensive than local computation for example; see subsection 3.4.1 Evented Programming on a Single Thread for related issues.

[10]Fundamental limitations with DOM access preclude revolutionary improvements to DOM performance; see subsection 3.4.2 Responsibilities of the UI-thread for details.

of the performance-related discussion in section 3.4 Execution Isolation, for instance.

A collection of articles from several notable authors on the topic of frontend performance was also released in 2012, titled "Web Performance Daybook Volume 2"[11] [39]. As it is a collection of independent articles, if offers a less holistic view of the field, but contains both more recent and in-depth advice on the issues it does cover.

[11]Curiously, a Volume 1 has never been published, and according to a representative of the publisher, perhaps never will be: http://support.oreilly.com/oreilly/topics/is_there_a_web_performance_daybook_volume_1

Chapter 3

Isolation Mechanisms

This chapter defines the 4 major classes of isolation mechanisms identified in this work, namely:

- State and Variable Isolation in section 3.1
- Isolation from the DOM in section 3.2
- Isolation within the DOM in section 3.3
- Execution Isolation in section 3.4

Each section introduces the specific concepts involved, contains concrete examples of related quality issues, and covers the available mechanisms for enforcing isolation within that class. This includes discussing the relevant best practices in the field, as in most cases the isolation mechanisms are not purely about using specific technology, or purely about implementing a specific architectural pattern, but a combination of both.

3.1 State and Variable Isolation

The first class of isolation mechanisms is state and variable isolation. It pertains to the common lack of namespacing in frontend applications: unless the developer takes explicit measures to implement isolation for state and variables, collisions (accidental sharing of a variable for example) are likely to occur. These collisions often manifest as bugs that are hard to diagnose.

3.1.1 Information Hiding

JavaScript, as a language, was born in haste and out of necessity. Thus, it does not contain many features a professional programmer might expect from a language used to implement industrial-grade applications. [7, p. 1] One such language feature is the ability to restrict the *visibility of variables*[1]. In other common languages — such as Java — it is possible to declare specific fields as private. This forces any access to them to take place within a restricted subset of the application — in contrast to being accessible from everywhere.

Restricting the visibility of variables is one common tool for implementing *information hiding* in an application. An application is ideally a collection of smaller parts, each of which take care of a specific task, and do little else. Information hiding refers to implementing those parts so that each exposes only enough features as is necessary. Since the parts are working together to form a larger system, some features are necessarily exposed so that the application can work as a whole, but everything else becomes an implementation detail. Such implementation details should not be of interest to any other part of the application, and can thus be changed freely, as long as the exposed features are not affected. [35, p. 268] [52, p. 30] Declaring fields or methods `private` in Java effectively marks them as implementation details, whereas declaring them `public` marks them as exposed interaction points.

All variables created outside an enclosing function body[2] are public by default in JavaScript [10, p. 56]. This means — in a simplified case — that every variable defined (including functions) is automatically accessible from every other part of the application. An experienced programmer will know to avoid such arbitrary access to maintain proper information hiding, but not all programmers benefit from formal training and years of industry experience. This is especially true in the case of JavaScript, due to the relative youth of the language, and the near-zero upfront investment in starting development [6]. Having language-level features for limiting access to specific variables alleviates the need for constantly consciously maintaining these limitations by the application developer. Also, in an application with more than one

[1] There are many terms with subtly differing meanings which could be used here: variable, identifier, symbol, name, et cetera. For the purposes of this discussion, variable will be used. In addition to the titular `var` statement, this covers well the visibility semantics of named functions, as they are very close to those of assigning anonymous functions into named variables [10, p. 98]. Assignments to the Global Object are also semantically close enough to using top-level `var` statements [10, p. 103].

[2] The upcoming revision of the language introduces the `let` keyword, though, which allows limiting the visibility of variables to scopes other than function bodies.

```
1 var addItem = (function() {
2   var private = [];
3   return function(newItem) {
4     private.push(newItem);
5   };
6 })();
7
8 addItem("foobar");
```

Listing 3.1: By wrapping the variable `private` in an Immediately Invoked Function Expression (IIFE), it becomes only visible to the function assigned to variable `addItem`, thus protecting it from uncontrolled access.

developer, these features become tools for communicating which parts of application code are meant to remain as implementation details. Having all variables public by default is often considered one of the worst features of the language [7, p. 40] [38, p. 11], and many of the following isolation mechanisms will involve information hiding in one form or another.

3.1.2 Emulating Privacy

The notable exception to the previously discussed global visibility of variables are function bodies. While variables that were originally created as global will remain accessible from everywhere[3], new variables defined within a function body are only accessible from within that specific function [10, p. 59]. This additional privacy can simply be coincidental: the primary purpose of functions is to represent an executable block of code, and the reduced visibility of new variables can be just a side-effect. But many JavaScript best practice guides ([33, p. 29] [7, p. 37], [54, p. 77]) place high value on this ability to provide information hiding, going as far as to advocate the use of functions solely for that purpose. To achieve this effect, the application code that would otherwise be in the global scope is placed within a function body, which is in turn immediately invoked (that is, the lines of code it contains are executed). This technique — also known as Immediately Invoked Function Expression (IIFE) [33, p. 115] — makes all newly introduced variables private to the enclosing function, while leaving the execution order of individual lines of code unchanged. This provides an effective emulation of private variables to the language [32]. Listing 3.1 demonstrates this effect.

[3] With specific exceptions such as a closer variable with the same name eclipsing it [10, p. 59].

3.1.3 Variable Scope

Variable scope refers to the parts of application code where a variable can be accessed [7, p. 36]. It is closely related to the previously discussed concept of visibility: declaring variables as private limits their scope, and declaring them as public extends their scope to the entire application.

Since JavaScript lacks language-level visibility modifiers (such as Java's `private` and `public`), the scope of a variable is defined by its enclosing function body[4]. These scopes can be nested, so that when variable references are bound, the named variable is first looked for from the function where the access takes place, then from the function that encloses that function, and so on. These function bodies form a *scope chain*, the root of which is the global scope, from which the variable reference is finally looked for if none of the previous scopes declared a variable by that name. [10, p. 51]

3.1.4 Naming Collisions

This nesting of scopes, with chained access to outer scopes, is a very useful feature of the language. It allows convenient sharing of application variables with select subsections of the same application. [7, p. 37] But it also has an insidious side-effect: when assigning to or reading from a variable which has not been declared in any outer scope, the variable is created to or read from the global scope[5]. This is a natural consequence of the previously discussed chained variable resolution, but can cause bugs that are hard to diagnose. The global variable being affected may control completely unrelated aspects of the application, a third-party library, or even the browser itself. [54, p. 69] An accidental sharing of a variable with the same name is called a *collision* [33, p. 114].

The most commonly recommended methods of protecting against collisions are always declaring variables [38, p. 11] and static code analysis [7, p. 116] [54, p. 70]. Collisions are most likely to happen in the global scope, since it is always shared by all parts of the application. Always explicitly declaring variables — as opposed to allowing them to be created implicitly — makes sure the global scope is never reached when traversing the scope chain, and thus the global state is never affected. Static code analysis tools

[4] With specific — and largely irrelevant in the context of this discussion — exceptions such as the `with/catch` clause [10, p. 51], and the previously mentioned upcoming `let` keyword.

[5] With the exception of the opt-in "strict mode" [10, p. 4].

```
1  // In the global scope:
2  var myApp = {
3    log: function(message) {
4      // process the log entry
5    },
6    alert: function(message) {
7      // show a customized alert dialog
8    }
9  };
10
11 (function() {
12   // In a local scope:
13   myApp.log("User logged in");
14   myApp.alert("Welcome to My Application!");
15 })();
```

Listing 3.2: Variables introduced by the application are organized under the namespace object `myApp`. While that object remains in the global scope, its properties `log` and `alert` are protected from collisions with other globals. For instance, the global `alert` is usually reserved by the browser, and assigning the custom implementation to that global might have undesired consequences for other parts of the application.

such as JSHint[6] and ESLint[7] can help in detecting implicit variable declarations, as well as many other common programming mistakes associated with a loosely typed language such as JavaScript.

3.1.5 Namespace Objects

Even though the use of global variables is often categorically discouraged [38, p. 10], many real-world applications end up sharing select variables and functions as globals. Generic variable names such as `log` are likely to be defined by many source files (either from the same application or from 3rd party components), and are thus prime candidates for collisions. To reduce the chances of collisions with other variables populating the global scope, many best practice guides suggest organizing all global variables into a global *namespace object*, whose name has a low likelihood of colliding with other globals [7, p. 25] [33, p. 114] [54, p. 71]. Specific names such as the name of the application are good candidates for the name of the namespace object. This technique is demonstrated in Listing 3.2.

[6] http://www.jshint.com/
[7] https://github.com/eslint/eslint

The properties of the namespace object are still effectively globals within the context of the application. While all arguments about avoiding shared global state and information hiding still apply, the namespace object makes collisions between the application, its libraries and the browser environment less likely.

Internal collisions are still possible within the namespace object: in a large application, the same property of the shared namespace object can be inadvertently used by unrelated parts of the code. This is in fact a manifestation of the same problem originally solved by the namespace object, only limited to the context of the application (as opposed to the global scope). It can be mitigated by applying the namespace object pattern recursively: for each subsection of the application, a separate property is reserved from the root namespace object, which is then used as the root namespace object for that subsection. Thus the namespace of the application can be subdivided as much as needed to avoid collisions. This subdivision also has the potential to help in understanding a large application, as functionality relating to specific subproblems the application solves can be found from specific parts of the namespace tree. This nesting approach is recommended by many best practice guides for namespacing complex applications [33, p. 114] [38, p. 88] [54, p. 72].

3.1.6 Modules

Recent years have seen a proliferation of *module systems* for JavaScript, the most prevalent of which are AMD[8], CommonJS[9] and, very recently, ES6 Modules[10].

While these systems contribute tremendously to the interoperability between JavaScript applications and components, they fundamentally offer little extra in terms of isolation between modules. The module systems contain many useful features related to locating and loading additional JavaScript code, and ensuring each additional module is loaded only once. However, in terms of isolation guarantees, they simply assume the role of a top-level namespace object, and offer little else than the techniques presented in Listing 3.1 and Listing 3.2 before.

That is, the importance of the various module systems warrant a mention, but their treatment here is left brief.

[8] https://github.com/amdjs/amdjs-api/wiki/AMD

[9] http://wiki.commonjs.org/wiki/CommonJS

[10] http://people.mozilla.org/ jorendorff/es6-draft.html#sec-modules, though the specification is yet to be finalized.

3.2 Isolation from the DOM

The second class of isolation mechanisms is isolation from the DOM. It pertains to the issues inherent in the tradition of tightly coupling application state and logic with the presentation logic of a frontend application. The major forces driving developers towards this are sheer productiveness, the ease of interweaving HTML, CSS and JavaScript, and traditions emerging from a culture of treating frontend applications more as the domain of graphic designers, not of trained programmers [28].

3.2.1 DOM Introduction

The DOM (Document Object Model [46]) is the in-memory representation a browser uses to construct, render and maintain the web page as defined by its HTML content. The DOM is exposed to the JavaScript executing on a page as an API for reading and modifying the web page, and for subscribing to events generated by the browser or the user interacting with the web page. [46]

On the evolutionary path of the frontend application, the DOM has always been a central component. The DOM API is the primary means for JavaScript to affect the contents of the web page it runs on, so to add any functionality to a static HTML document, the DOM API must be used. The DOM is also a large shared data structure always accessible by all JavaScript executing on the page — effectively a mandatory shared global variable for all frontend applications [28]. In light of the previous discussion in section 3.1 State and Variable Isolation, this is not an optimal foundation on which to build large, production quality applications.

3.2.2 Naive DOM Use

Using the DOM as the shared data structure that maintains the complete application state is very tempting. When some change to application state takes place, the UI will have to be updated to reflect that change, and the visible UI changes can only be effected through the DOM API[11]. Listing 3.4 presents an application which does the following:

1. Performs a `POST /login` HTTP request.

[11] There are specific exceptions such as updating the UI through the recently introduced `<canvas>` element, though frontend applications only using that element for UI updates are rare.

```
<script src="/lib/jquery-2.1.1.js"></script>
<header style="display: none"></header>
<ul></ul>
```

Listing 3.3: HTML document associated with the JavaScript application presented in Listing 3.4.

```
$.post("/login").then(function(userName) {
  $("header").text("Logged in as: " + userName).show();
}).always(function() {
  $.get("/posts").then(function(posts) {
    $("ul").append(posts.map(function(post) {
      var $li = $("<li>").text(post);
      if ($("header").is(":visible")) { // login check
        $li.append(" <a href=#>[edit post]</a>");
      }
      return $li;
    }));
  });
});
```

Listing 3.4: A sample JavaScript application which maintains its state exclusively in the DOM.

2. If the request succeeds (that is, correct credentials were provided), the server-provided username is made visible on the UI. If not, this step is skipped, and the relevant part of the UI remains hidden.

3. Performs a `GET /posts` HTTP request.

4. For each post received from the server, adds it to the UI into the list of posts.

5. For each such post, based on whether or not the user is logged in, optionally displays an link for editing the post (as for the purposes of this example editing posts is reserved for users who are logged in).

The application presented in Listing 3.4 is in fact a fully functioning implementation of the listed requirements, all in 13 lines of JavaScript code, and 3 lines of supporting HTML (Listing 3.3). It contains non-blocking network operations, conditional branching based on the success or failure of those operations, updating the UI and storing and retrieving the state of the previous login request (as the visibility of the `<header>` element). The

application only depends on a single external library, jQuery[12].

Indeed, what makes this method of storing application state in the DOM tempting is that there is no duplication of effort: the UI changes can only be effected through the DOM API, so the state will have to exist in the DOM. It thus stands to reason said state should not have to be duplicated elsewhere. Furthermore, modern DOM manipulation libraries (such as jQuery) can make the developer very productive in working with the DOM, and under the constant pressure to ship code [4], getting features implemented quickly sounds very attractive. Finally, while the example application in Listing 3.4 is contrived, this has, up until recent times, been a very common way to model the frontend code of web applications: keeping most application state in the DOM [28].

3.2.3 Storing State in the DOM

Unfortunately, as application complexity increases, this method of maintaining application state mostly within the DOM starts to break down [21]. Firstly, let us assume other conditional actions (similar to the previous login check) will have to be performed in other parts of the application: some other information or functionality can only be available to logged in users, for example. The same DOM-based check will be implemented for those cases. The application code ends up riddled with references to specific elements in the DOM. But the requirements for software systems need to cope with change — so much so in fact that requirements churn is classically attributed as one of the top reasons for software project failure [15]. Should the application requirements change so that the login status be shown in a `<footer>` element instead of the current `<header>` element, all of the DOM-based conditional checks would break. This is because the element from which the state is retrieved is specified by its type, and the type has to be updated everywhere where there is a login status check.

However, there are several ways to work around this problem:

- Reference the element through an ID-attribute, so that the actual element type can change without affecting the conditional check:
  `<header id="login-state">` and `$("#login-state")`
- Use a CSS class for storing the login state, and use the class for rendering the element visible/hidden as needed:
  `<header class="is-logged-in">`

[12] http://jquery.com/

- Create a function which will encapsulate retrieving the state from the DOM, so that the way it is stored in the DOM can be freely changed without affecting more than one location in the application code: `function isLoggedIn() { return $("..."); }`

Yet, let us assume that the requirements change so that the login state need not be displayed to the user at all. The intuitive solution might be to keep the `<header>` element permanently hidden in the UI. All of the conditional checks will keep working as before[13], and hiding an element is usually a single line of added styling, thus very straightforward to implement. But as the application evolves, maintaining UI elements in the DOM which have no contribution to the actual UI becomes questionable, for several reasons:

1. Firstly, DOM access is one of the slowest aspects of a modern JavaScript engine [53, p. 35], and accumulating unnecessary DOM accesses will begin slowing the application down.

2. Secondly, another developer working on the application may see a part of the UI which is never shown to the user, and decide to remove it as unnecessary. This would again break the conditional checks.

3. Finally, the example of a boolean login state is an oversimplification — in a frontend application of non-trivial size, the types of state that will have to be maintained include complex data structures such as objects, collections of objects, and references between them.

Thus, we conclude that using the DOM as both the description of the UI and the sole storage of application state is problematic, in all but the most trivial frontend applications. That is, we want to isolate application state from the DOM, so that concerns of visual representation do not interfere with concerns of application logic.

3.2.4 Separation of Concerns

Separation of Concerns was originally described by Edsger W. Dijkstra in 1974 as follows:

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake

[13] Assuming something else than element visibility is used as the property storing the login state.

> of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called **"the separation of concerns"**, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. [9]

Besides rooting itself permanently into computer science terminology, the concept is often discussed alongside JavaScript and frontend application architecture. Its primary technical interpretation is maintaining a clear separation between the HTML, CSS and JavaScript of an application — that is, its *content, presentation and behaviour* [38, p. 181] [54, p. 53]:

- HTML defines the *content* of a web page, by providing the building blocks and their associated semantics from which a web page is assembled. It is possible to embed both CSS and JavaScript within HTML, but either is considered bad separation of concerns [54, p. 57]. An exception to this are some advanced optimization techniques, but they are generally applied using build automation during deployment, rather than being maintained as part of the codebase [53, p. 163].

- CSS defines the *presentation* of a web page, using declarative rules which apply a specific visual appearance to matching DOM elements. This application is automatic, in the sense that an update to the DOM will cause a web browser to automatically consult the associated CSS for relevant changes in appearance. [42] It is not possible to embed HTML or JavaScript within CSS[14].

- JavaScript defines the *behaviour* of a web page, by executing code against the scripting environment provided by the browser. It is possible to embed both HTML and CSS within JavaScript, though again, both are considered bad separation of concerns [54, p. 56].

[14] An exception to this are CSS Expressions (http://msdn.microsoft.com/en-us/library/ms537634%28v=vs.85%29.aspx) which are not supported by any modern browser. Methods for achieving similar end-results, such as the CSS `calc()` function (https://developer.mozilla.org/en-US/docs/Web/CSS/calc) and CSS pseudo-elements (https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements) exist, but do not allow direct embedding of either JavaScript or HTML into CSS.

In addition to the above technical interpretation, the more abstract interpretation of the separation of concerns has to do with the internal structure of the JavaScript of the application. Many JavaScript best practice guides give a thorough treatment to related concepts, such as:

- Code base organization [54]
- Design Patterns [33]
- Code maintainability [54]
- Application architecture [32]

Yet, while these concepts have important applications in the domain of frontend applications, they are — in their general form — ultimately applicable to any software development effort. Thus, their treatment is limited to citing relevant sources, in favor of focusing on topics specific to frontend applications.

3.2.5 The MVC Pattern

From its beginnings, up to recent times, web frontend development efforts have lacked structure. Instead of proper software development (which could and would be taken seriously), the prevailing attitude towards frontend development has been one of hacker-culture and ignorance: even if web applications contained significant frontend components, they have been mostly treated as a necessary evil, or at least the domain of graphic designers and the like, not proper programmers. [28] [54, p. ix]

With the increasing importance of the web frontend, and indeed the rise of the frontend application, the frontend developer community has found itself revisiting established architectural patterns for software development, to allow their applications to grow in size and complexity. Perhaps the most important of these patterns is the Model-View-Controller (MVC) pattern, originally introduced with the Smalltalk-80 programming environment [24] in the early 1980's, and later popularized by the classic Design Patterns work by the "Gang of Four" in 1994 [13].

In the form the pattern has since been adopted by frontend application developers [33], it splits the application into three distinct parts:

- **Model:** Encapsulating and managing the data and state of the application.

```
1 <script src="/lib/jquery-2.1.1.js"></script>
2 <script src="/lib/underscore-1.7.0.js"></script>
3 <script src="/lib/backbone-1.1.2.js"></script>
4
5 <header style="display: none"></header>
6
7 <ul></ul>
```

Listing 3.5: HTML document associated with the application presented in Listing 3.6.

- **View:** Providing a visual representation of related Models (using the DOM API).
- **Controller:** Working as an intermediary between the Models and the Views, effecting updates to Model state as the user interacts with a View, and updating relevant Views when changes to Model state are detected.

In the context of this discussion, the most important property of the MVC pattern is that it allows for a separation of concerns between the state, UI and logic of a frontend application (Models, Views and Controllers, respectively). Revisiting the simple application introduced in subsection 3.2.2 Naive DOM Use and addressing the issues brought up in subsection 3.2.3 Storing State in the DOM, we can apply the MVC pattern for separating the concerns of the state and the UI. Backbone.js[15] is used for the demonstration of the principle. Backbone.js was arguably one of the most prominent libraries in the beginnings of the movement from the unstructured frontend code of the past to the modern frontend application era, and is also heavily inspired by the MVC pattern[16]. Thus, it is a well-suited library for the sample presented in Listing 3.6.

The sample application presented in Listing 3.6 addresses the issues of its predecessor (presented in Listing 3.4) in several important ways:

1. It maintains the data and state of the application in a **Model** (variable `data`), separate from the DOM. This allows performing the login check (on line 15) without any assumptions towards how the data is visually represented. Should the requirements change to display the login status in a `<footer>` element instead, the conditional checks would remain

[15] http://backbonejs.org/
[16] http://backbonejs.org/#FAQ-mvc

```
var MainView = Backbone.View.extend({
  initialize: function() {
    this.listenTo(this.model, "change", this.render);
  },
  render: function() {
    var userName = this.model.get("userName");
    var posts = this.model.get("posts");
    this.$("header")
      .text("Logged in as: " + userName)
      .toggle(!!userName);
    this.$("ul")
      .empty()
      .append(posts.map(function(post) {
        var $li = $("<li>").text(post);
        if (!!userName) { // login check
          $li.append(" <a href=#>[edit post]</a>");
        }
        return $li;
      }));
  }
});

var data = new Backbone.Model();
var ui = new MainView({
  el: document.body,
  model: data
});

$.post("/login").then(function(userName) {
  data.set({ userName: userName });
}).always(function() {
  $.get("/posts").then(function(posts) {
    data.set({ posts: posts });
  });
});
```

Listing 3.6: The application originally presented in Listing 3.4, rewritten according to the principles of the MVC pattern. The state of the application is stored in `data` (Model), the DOM is managed exclusively through `ui` (View), and the root scope performs the functions of the Controller.

unaffected, and only the View code would need to be changed. Similarly, the login status can be removed from the visible UI altogether, without affecting the conditional checks.

2. It manipulates the DOM exclusively through a **View** (variable `ui`). One of the important implications is that if any of the aforementioned changes to the appearance of the UI have to be implemented, the View alone needs to be changed.

3. It establishes a persistent binding between the Model and the View, by the main script acting as a **Controller**. This is an important improvement over the previous implementation, which was built to handle responses to two HTTP requests, and to then update the UI accordingly. The version presented in Listing 3.6 raises the level of abstraction so that any subsequent operation which changes the state in `data` will also be reflected by the UI automatically. For instance, evaluating `data.set({ userName: null });` at any later point during the application execution will hide the "[edit post]" labels and the "Logged in as:" text.

Applications of the MVC Pattern have been an important stepping stone in the evolution of modern frontend application development [33, p. 79]. While it has benefitted the craft in numerous ways, its most important aspect while discussing isolation from the DOM is allowing for an explicit separation of DOM manipulation from the other concerns of the application (such as state manipulation or domain logic).

3.3 Isolation within the DOM

The third class of isolation mechanisms is isolation within the DOM. It pertains to the common lack of isolation between components that need to coexist within a shared DOM. Unless explicitly addressed by the developer, this allows for accidental access across components, with the potential to affect both application logic and appearance in an unexpected manner. Mechanisms for dealing with these issues exist, but require various tradeoffs between effectiveness and feasibility.

3.3.1 Introduction

The DOM can become a very large data structure, shared across the entire frontend application. In addition to keeping the DOM and the application state separate (as previously discussed in section 3.2 Isolation from

the DOM), providing isolation between parts of the DOM can be equally valuable.

The DOM is a tree structure. Regardless of the plethora of potential problems the browser may encounter while constructing the tree (such as improperly nested elements, missing required elements or attributes, et cetera), modern browsers are very resilient in HTML parsing, and its result is always a directed acyclic graph of DOM nodes, of varying types. Most of these types correspond to specific types of HTML elements (such as a button or a text input). So while the HTML parsed by the browser may contain errors or omissions, it is always normalized into a standardized tree structure. [45]

3.3.2 CSS Selectors

Specific elements of the DOM tree have to be targeted for various reasons. The Cascading Stylesheets (CSS) specification [42] defines *selectors,* using which one or more DOM elements can be selected to receive specific visual styling by the browser. JavaScript code often needs to select specific DOM elements as well. Common examples include being able to subscribe to DOM events [46] generated at a specific element, programmatically changing the styling applied to an element, or adding or removing an element altogether. Updating the contents of a specific element also requires first selecting the element to be updated. This is needed, for example, when a View updates the HTML contents of a specific part of the page, as discussed in section 3.2 Isolation from the DOM, and exemplified in Listing 3.6.

CSS selectors are a compact and expressive way to select elements. In modern browsers, the CSS selector engine used to apply styling to the DOM tree can also be accessed from JavaScript using the Selectors API [44]. Even without the convenience of the Selectors API, the basic DOM API provides methods — albeit cruder ones — for traversing the tree and selecting elements. Finally, abstractions exist which will use either method of element selection, depending on what is available in the browser environment. These abstractions are also able to combine both approaches for increased performance, and implement selector syntaxes beyond what is defined by the CSS specification alone[17].

[17]jQuery (http://jquery.com/) is one such abstraction, leveraging the Selectors API where available, but implementing several non-standard selector syntaxes on top (such as `:visible`) using JavaScript (http://sizzlejs.com/).

```
<header>
  Followers:
  <ul id="followers">
    <li>John
    <li>Alice
    <li>Maurice
  </ul>
</header>

<main>
  List of posts:
  <ul id="posts">
    <li>The Art of Isolation
    <li>The Perils of Globals
    <li>The Root of all Selectors
  </ul>
</main>
```

Listing 3.7: The HTML of a sample application which lists followers and posts, demonstrating the dangers of relying on global CSS selectors for effecting UI updates: naive selections such as `"ul"` would match both lists, which is rarely intended.

3.3.3 Selection Collisions

Element selection — whether realized through selector queries or raw DOM access — is essential to all applications which interact with the DOM. Elements may be selected by querying the entire DOM tree for elements matching a given selector, but also to query a specific subtree of it[18]. This makes selector queries easy to understand: they always start from a specific element, working their way down the tree and looking for matches to the executing selector. The most commonly used DOM manipulation libraries (such as jQuery) and the Selectors API support both types of queries — rooted and global ones — but always default to the latter.

In light of the arguments against globals presented in section 3.1 State and Variable Isolation, global selectors seem to violate previously established best practices. To demonstrate this, Listing 3.7 presents an application which lists followers and posts. Assuming for the moment the `<header>` and its contents (that is, the list of followers) do not exist, the list of posts could be updated with:

```
$("ul").append("<li>New post</li>");
```

[18]Querying the entire document is in fact equivalent to querying the subtree rooted at the document root.

This performs a global selector query[19] which matches the list of posts, into which a new item is then appended. This is very easy to implement and works well in a simple application. Should the requirements of the application change, however, so that the `<header>` needs to be included (and into it rendered the list of followers), the above code for adding a post would cause the new item to appear into both lists. This is not desired behaviour, and is analogous to the preceding discussion in subsection 3.1.4 Naming Collisions, in that following a reference (in this case the selector `"ul"`) leads to an unexpected object (in this case the set of both lists, instead of just the intended one).

The naive solution to this predicament is to give unique names to the target elements. In Listing 3.7 this has already been done, so that both lists can be selected by their `id` attribute as follows:

```
$("ul#posts").append("<li>New post</li>");
$("ul#followers").append("<li>New follower</li>");
```

The above code will keep matching the intended list, regardless of how many other lists will be added to the application. Yet the solution still suffers from the same underlying issue: relying on names in a global namespace. As a frontend application grows — in lines of code and in the number of developers working on it — the chances for selector collisions in a global namespace become increasingly likely [16].

3.3.4 Rooted Selectors

As previously discussed, selector queries can be either global or rooted. Rooted queries start their DOM tree traversal from a given DOM element, and will never match elements which are not descendants of the given element [44]. Holding a reference to a DOM element effectively creates a *selector query namespace*:

```
var $namespace = $("main");
$namespace.find("ul").append("<li>New post</li>");
```

The above code can again use the simple `"ul"` selector without mixing up the list elements, as the element stored at `$namespace` only contains the intended list element, but not the other one.

Recalling Listing 3.6, a View in an MVC-like architecture often allows managing these selector query namespaces automatically. The sample application in fact makes use of this namespacing by referring to `this.$()`

[19] The `$` variable refers to jQuery's Selector API (http://api.jquery.com/category/selectors/). Unless otherwise noted, this applies to other code samples as well.

instead of `$()` for selecting elements. The former performs a rooted selector query within the confines of the View instance, whereas the latter performs a global selector query. This technique — regardless of which frameworks or libraries are being used — is a powerful method for providing separation between components which inhabit the shared DOM. [16]

3.3.5 Style Isolation

While the preceding discussion has treated JavaScript exclusively, let us recall subsection 3.3.2 CSS Selectors, and the original purpose of CSS selectors in effecting visual styling into a web page. Indeed, the same dangers of global selections apply in the context of styling: an accidental reuse of an existing CSS selector may visually affect an unrelated part of the application in an undesired way.

The concept presented in subsection 3.3.4 Rooted Selectors can be applied to provide a degree of style isolation as well. When developing styles for an element, the developer can limit their applicability by choosing a parent element which is relevant to the context: instead of using `"ul"` as the selector, using `"main ul"` ensures only the `<ul>` element within the `<main>` element is applicable. This is referred to as increasing the *specificity* [42] of the selector.

Increasing selector specificity, however, is not a panacea for providing style isolation within the DOM. To the contrary, many notable works on managing CSS in complex frontend applications treat overly specific selectors as an antipattern. While other reasons such as performance are also cited, the most prominent argument is one of maintainability: overly specific selectors make the structure of the application hard to change without causing unexpected changes to styling as well. These works instead suggest using fairly unspecific selectors based on unique, globally reserved names for each component. [17, 37] In our opinion, this is merely a stopgap measure of managing styling complexity in a large frontend application, since many of the fundamental problems of using a single namespace for global names (see subsection 3.3.3 Selection Collisions) remain unsolved by the approach.

3.3.6 Depthwise Isolation

The preceding discussion on isolation within the DOM — both isolating JavaScript element selections and isolating CSS styling — has concentrated on limiting the effects DOM siblings have on each other. Being siblings is not the only way DOM elements can be related, however, and much of DOM terminology [46] revolves around these concepts of ancestry:

- The **parent** of a DOM element is another element which directly contains the element in question
- A **child** of a DOM element is another element for which the element in question is the parent
- Two or more elements in the DOM are considered **siblings** if they share the same parent element

A condition not yet explicitly discussed arises when two UI components of an application are not siblings but instead in a *parent-child* relationship. Considering our aim is to provide isolation between UI components in the DOM, the previously discussed methods for achieving that isolation are not equally effective anymore. To demonstrate this, recalling Listing 3.7, let us assume the application needs to be changed so that each post also contains a list of the authors who have worked on it. The results of such amendments are presented in Listing 3.8, and they present potential violations of cross-component isolation in several scenarios:

- If using a selector such as `$("main ul")` to effect changes to the list of posts, the newly added list of authors would inadvertently receive the same changes. The syntactic alternative of using `var $namespace = $("main");` would be equally defective.
- If JavaScript was using a View in an MVC-like architecture (as in Listing 3.6) and selecting elements with properly rooted selectors such as `this.$("ul")`, the issue would persist. This is regardless of whether the inner list (that is, the list of authors) was managed by another independent View or not.
- If the page stylesheets were using CSS rules such as `ul li { color: red; }` to make the titles of posts appear in with specific visual styling, the same styling would — most likely accidentally — be applied to the list of authors as well.

The root cause for all of the issues listed above is the use of *descendant selectors* for specifying the relationships between the various referenced DOM elements. The CSS specification defines the selector as follows:

> At times, authors may want selectors to match an element that is the descendant of another element in the document tree (for example, "Match those EM elements that are contained by an H1 element"). Descendant selectors express such a relationship

```
<main>
  List of posts:
  <ul id="posts">
    <li>The Art of Isolation, by:
      <ul>
        <li>John
        <li>Donny
        <li>Alice
      </ul>
    </li>
    <li>The Perils of Globals, by:
      <ul>
        <li>Maurice
      </ul>
    </li>
    <li>The Root of all Selectors, by:
      <ul>
        <li>John
        <li>Alice
      </ul>
    </li>
  </ul>
</main>
```

Listing 3.8: The HTML of a sample application which lists posts and their authors, demonstrating the dangers of unfettered depthwise CSS selection.

> in a pattern. A descendant selector is made up of two or more selectors separated by white space. A descendant selector of the form `"A B"` matches when an element `B` is an arbitrary descendant of some ancestor element `A`. [42]

While specific figures on their prevalence were not obtained for this work, in our experience, descendant selectors are the overwhelmingly most common form of CSS selectors on the web. Alternative selector forms exist which define stricter rules for the parent-child relationship (such as the child selector[20]), but their widespread use is both rare and against recent works on CSS best practices (as discussed in subsection 3.3.5 Style Isolation).

Thus, we conclude that providing depthwise isolation for either JavaScript element selections or CSS styling using the current, widely available technologies remains challenging. The single exception[21] to this are *frames*.

3.3.7 Frames and the iframe

The concept of frames on web pages dates back to the 90's and the HTML 4.01 Specification, which introduces them as:

> HTML frames allow authors to present documents in multiple views, which may be independent windows or subwindows. Multiple views offer designers a way to keep certain information visible, while other views are scrolled or replaced. For example, within the same window, one frame might display a static banner, a second a navigation menu, and a third the main document that can be scrolled through or replaced by navigating in the second frame. [40]

That is, a frame essentially creates a view into another web page from within the current web page. As of HTML5 [45], all other frame types except the *iframe* have been deprecated, so the following discussion will focus solely on the iframe element. The iframe, however, has several properties which make it very different from most other DOM elements, such as:

1. **Creating nested browsing contexts:** A web page loaded into an iframe behaves as it were a stand-alone page loaded directly into the

[20] https://developer.mozilla.org/en-US/docs/Web/CSS/Child_selectors

[21] `<svg>` elements are another, but largely uninteresting in the context of this discussion, as while an `<svg>` element has many of the same properties of a frame, they are meant for representing graphics, not general-purpose web page components.

browser[22], except for being visually rendered inside its parent web page. This includes a separate JavaScript environment from the parent page.

2. **Generating a new viewport:** Media Queries [42] within an iframe will react to the dimensions of the iframe, not the web page containing the iframe.

3. **Restricting JavaScript DOM access:** A web page using an iframe to load another page from a differing origin[23] is not allowed to access the content of the loaded page with JavaScript[24]. The same applies the other way around, as the loaded page is not allowed to access its parent page, either. This includes stopping the propagation of element selection using selector queries.

4. **Stopping the application of CSS styling:** Style rules which have been defined for a web page do not[25] affect the web page loaded into an iframe.

Considering the context of our discussion — isolation within the DOM — the iframe seems like a perfect unit of composition for building large frontend applications, as it specifically addresses the primary issues brought up thus far:

- In subsection 3.3.3 Selection Collisions: When performing element selection through JavaScript, selector queries will never match elements within an iframe. Those elements are thus safe from accidental modification, such as in Listing 3.7.

- In subsection 3.3.5 Style Isolation: CSS styles defined for an element within an iframe will never affect the styling of its siblings.

- In subsection 3.3.6 Depthwise Isolation: Leaks between the parent and the child — both in CSS styling and JavaScript element selection — are prevented, such as the ones in Listing 3.8.

[22]There are other subtle differences as well, such as how the browser handles navigation events within a nested browsing context, but their full enumeration is beyond the scope of this work. The interested reader is referred to https://developer.mozilla.org/en/docs/Web/HTML/Element/iframe for a high-level explanation of an iframe's properties.

[23]https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[24]There are several exceptions to this, however, such as `window.postMessage` (https://developer.mozilla.org/en-US/docs/Web/API/window.postMessage), which allow selective flexibility in isolating the JavaScript environments.

[25]The experimental `seamless` attribute (https://developer.mozilla.org/en/docs/Web/HTML/Element/iframe#a seamless) allows selectively relaxing this, however.

```
<style>
  li {
    font-weight: bold;
  }
</style>

<main>
  List of posts:
  <ul id="posts">
    <li>The Art of Isolation, by:
      <iframe src="author-list.html"></iframe>
    </li>
  </ul>
</main>
```

Listing 3.9: The HTML of a sample application which lists posts and their authors. The list of authors is included using an iframe to ensure its isolation from the rest of the application. The list of authors is loaded as a separate web page from the URL `"author-list.html"`, and its contents are presented in Listing 3.10.

```
<style>
  li {
    color: red;
  }
</style>

<ul>
  <li>John
  <li>Donny
  <li>Alice
</ul>
```

Listing 3.10: The HTML which is assumed to be available at the URL `"author-list.html"`, and which is included by the main application presented in Listing 3.9 to list the authors for a post.

A sample application listing posts and their authors is presented in Listing 3.9 and Listing 3.10. The isolation properties of the iframe ensure that while both components use the simple `"li"` selector, the `font-weight: bold;` styling only applies to the list of posts, and conversely, the `color: red;` styling only affects the list of authors. Similarly, should either part of the application use selector queries such as `$("li")`, the former application would only match elements representing posts, while the latter application would only match elements representing authors.

While iframes exhibit extremely desirable characteristics for providing isolation between UI components in the DOM, those capabilities come at a cost — iframes come with severe performance penalties:

- Resources needed by the web page within the iframe (including the page itself) cause additional HTTP requests to the server[26]. As the list of needed resources is often unknown until the page within the iframe has started loading, this limits the browser's ability to load the resources in parallel, inducing unwanted latency [39, p. 31].

- Additional HTTP requests caused by loading the iframe consume an HTTP connection pool shared with the rest of the web page. What this means is that after opening a specific number of HTTP connections, the browser will start enqueuing subsequent connection attempts until capacity is returned to the connection pool, again introducing unwanted latency to loading the page. [39, p. 31] This number has traditionally been as low as 2-4 connections per hostname[27], though modern browsers have continuously increased the number, 10-15 connections being a reasonable assumption at the time of writing[28].

- Creation of iframes into the DOM is costly in terms of execution time. As the most recent figures by a reputable online source[29] were already over 5 years old, and browsers are updated with an ever-increasing

[26]Advanced optimization techniques such as aggressive resource inlining helps with the performance penalties, but are both non-trivial to implement and beyond the scope of this work.

[27]http://www.stevesouders.com/blog/2009/06/03/using-iframes-sparingly/ lists some classical guidelines on the topic.

[28]http://www.browserscope.org/?category=network contains relevant statistics from many contemporary browsers, and is frequently updated as browsers update; the truly interested reader is directed to https://insouciant.org/tech/connection-management-in-chromium/ for a fascinatingly thorough treatment of the connection management in the Chromium project.

[29]http://www.stevesouders.com/blog/2009/06/03/using-iframes-sparingly/ is by a well-known web performance researcher, but being from 2009, already outdated.

frequency[30], this assumption was verified experimentally. The findings are presented in Table 3.1, and indicate a creation cost 1-2 *orders of magnitude* greater than that of simpler DOM elements.

- Maintenance of iframes within the DOM is costly in terms of retained memory. As relevant existing figures were not obtained for this work, this assumption was also verified experimentally. The findings are presented in Table 3.3, and indicate a maintenance cost 2-3 *orders of magnitude* greater than that of simpler DOM elements.

In light of these performance characteristics, using iframes as the only — or even the primary — unit of composition in a large frontend application does not seem feasible. While the performance penalties are significant, the usage of iframes is still extremely common across the web. This is due to the prevalence of contexts such as online advertising, where the performance tradeoff for isolation and security is desirable. In conclusion, iframes are an indispensable tool for providing strong isolation guarantees within a frontend application, but have to be used sparingly due to their high runtime costs.

3.3.8 Web Components

While composing large frontend applications entirely out of iframes (to leverage their favorable properties for isolation within the DOM) remains prohibitively expensive performance-wise, a collection of recent standard proposals — collectively known as Web Components [47] — introduces mechanisms for achieving some of the same benefits, without the steep performance penalties of iframes. In this regard, the most interesting part of said specifications is the Shadow DOM, the abstract of which reads:

> This specification describes a method of combining multiple DOM trees into one hierarchy and how these trees interact with each other within a document, thus enabling better composition of the DOM. [48]

Contrasting its offering with the beneficial properties previously introduced in subsection 3.3.7 Frames and the iframe, we find several similarities:

[30]Indeed Chrome, one of the major browsers of today had only been in existence for a year at the time of Steve Souders' related article, and has since received major updates almost monthly. It is reasonable to assume the performance characteristics of today's browsers may have changed, even drastically, from those of 2009.

```
1  // <a>
2  var el = document.createElement("a");
3  el.href = "http://google.com/";
4  el.appendChild(document.createTextNode("Some textual content"
       ));
5  document.body.appendChild(el);
6  document.body.removeChild(el);
7
8  // <div>
9  var el = document.createElement("div");
10 el.appendChild(document.createTextNode("Some textual content"
       ));
11 document.body.appendChild(el);
12 document.body.removeChild(el);
13
14 // <script>
15 var el = document.createElement("script");
16 el.appendChild(document.createTextNode("var foobar = 123;"));
17 document.body.appendChild(el);
18 document.body.removeChild(el);
19
20 // <style>
21 var el = document.createElement("style");
22 el.appendChild(document.createTextNode("p { color: red; }"));
23 document.body.appendChild(el);
24 document.body.removeChild(el);
25
26 // <iframe>
27 var el = document.createElement("iframe");
28 el.srcdoc = "<p>Hello World</p>";
29 document.body.appendChild(el);
30 document.body.removeChild(el);
```

Listing 3.11: Source code for testing the performance of the creation of `<iframe>` elements into the DOM. Several other common element types are created similarly to provide context. The execution time of each code block should be measured separately. For a sample of such measurements, see Table 3.1.

```
1  // <div>
2  var el = document.createElement("div");
3  var p = document.createElement("p");
4  p.innerText = "Hello World";
5  el.appendChild(p);
6  document.body.appendChild(el);
7  document.body.removeChild(el);
8
9  // Shadow DOM
10 var el = document.createElement("div");
11 var root = el.createShadowRoot();
12 var p = document.createElement("p");
13 p.innerText = "Hello World";
14 root.appendChild(p);
15 document.body.appendChild(el);
16 document.body.removeChild(el);
17
18 // <iframe>
19 var el = document.createElement("iframe");
20 el.srcdoc = "<p>Hello World</p>";
21 document.body.appendChild(el);
22 document.body.removeChild(el);
```

Listing 3.12: Source code for testing the performance of the creation of Shadow DOM elements into the DOM. Several other common element types are created similarly to provide context. The execution time of each code block should be measured separately. For a sample of such measurements, see Table 3.2.

```
// Testbench for comparing heap allocations between creating
// and appending various types of DOM elements.

var EL_COUNT = 100;
var EL_TYPE = "shadow"; // one of "div", "iframe" or "shadow"

// Wait for a bit after main document load:
window.setTimeout(function() {
  // Mark the start of the interesting part in the timeline:
  console.timeStamp("Start: " + EL_TYPE);
  // Create and append EL_COUNT elements of given type:
  var range = new Array(EL_COUNT + 1).join(" ").split("");
  range.forEach(function(x, i) {
    var el, text;
    if (EL_TYPE === "shadow") {
      el = document.createElement("div");
      var p = document.createElement("p");
      text = document.createTextNode(EL_TYPE + "#" + i);
      p.appendChild(text);
      el.createShadowRoot().appendChild(p);
    } else {
      el = document.createElement(EL_TYPE);
      if (EL_TYPE === "div") {
        text = document.createTextNode(EL_TYPE + "#" + i);
        el.appendChild(text);
      } else if (EL_TYPE === "iframe") {
        el.srcdoc = EL_TYPE + "#" + i;
      }
    }
    document.body.appendChild(el);
  });
}, 3000);
```

Listing 3.13: Source code for testing the memory retention of `<iframe>` and Shadow DOM elements. Regular `<div>` elements are created similarly to provide context. Measuring the memory retention for each element type should be carried out using the profiling facilities of the web browser. For a sample of such measurements, see Table 3.3.

UserAgent	<a>		<iframe>	<script>	<style>	Runs
Chrome 37.0.2062	130,477	209,187	1,316	101,874	79,117	1
Chrome 38.0.2125	68,291	137,432	1,855	65,181	37,010	3
Chrome 40.0.2182	64,994	184,361	982	59,954	35,949	3
Chrome 40.0.2183	146,735	290,175	1,460	145,301	72,146	1
Chromium 37.0.2062	25,811	39,079	1,070	25,491	21,469	1
Firefox 20.0	65,352	100,751	280	8,270	19,891	1
Firefox 31.0	114,256	153,427	265	18,498	26,334	1
Firefox 32.0	103,343	161,694	502	20,889	25,917	2
Opera 12.17	57,492	134,042	2,713	17,860	59,852	1
Opera 24.0.1558	109,205	206,955	1,211	84,694	51,175	1
Other	5,613	6,424	1,288	4,434	2,577	2
Safari 7.1	322,741	720,387	2,629	190,859	71,051	1
Safari 8.0	200,581	617,288	1,294	139,872	62,998	2

Table 3.1: This table presents the raw results of evaluating the runtime performance of Listing 3.11. Numeric columns (except runs) indicate operations per second, that is, how many times the browser was able to perform the creation of the element within one second. Thus, higher is better. The tests were carried out using http://jsperf.com/, a service specializing in comparative performance evaluation of JavaScript code samples. During a period of approximately 12 hours, the test page was visited by 20 separate web browsers, each of which was benchmarked for performance (the runs column indicates the visit frequency per browser). The technical implementation and the reliability of the results of such benchmarking are discussed at length in http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/.

UserAgent		<iframe>	Shadow DOM	Runs
Chrome 37.0.2062	135,555	908	94,137	1
Chrome 38.0.2125	144,250	1,834	84,231	2
Chrome 40.0.2182	100,339	789	62,055	1
Chrome Mobile 38.0.2125	21,764	268	17,200	1
Firefox 32.0	78,872	244	33,111	1
Mobile Safari 8.2	75,381	48		2
Opera 12.17	111,434	2,706		1

Table 3.2: This table presents the raw results of evaluating the runtime performance of Listing 3.12. The results were obtained using the methodology described in Table 3.1. During a period of approximately 12 hours, the test page was visited by 9 separate web browsers.

1. **Creating nested browsing contexts:** While technically not equivalent to the browsing context defined by the HTML5 specification [45] — most notably because the JavaScript environment is not isolated — the end-result is similar: the Shadow DOM allows maintaining a DOM subtree in isolation, while still visually rendering it as it were part of the parent page which hosts it.

2. **Generating a new viewport:** A Shadow DOM does not generate a new viewport.

3. **Restricting JavaScript DOM access:** Selector queries will not match elements within a Shadow DOM by default. The specification does allow access when explicitly requested, but considering such access requires specific alterations to the way the DOM is accessed[31], it stands to reason such access would unlikely be accidental.

4. **Stopping the application of CSS styling:** As with selector queries, CSS styling does not have effects on a Shadow DOM by default. Again, explicit methods[32] exist for circumventing this protection, but they are less likely to happen unintentionally.

Based on the above evaluation, the Shadow DOM does not provide equal isolation properties compared to iframes. While #1 would be beneficial for previously discussed properties (see section 3.1 State and Variable Isolation), and #2 would greatly benefit Responsive Web Design[33], #3 and #4 are strikingly similar to the corresponding properties of iframes. Indeed, as one of the design goals of the Shadow DOM specification is "enabling better composition of the DOM" [48], this seems reasonable.

Striking a balance between the isolation guarantees of iframes and the performance of a globally shared DOM, it would seem the performance characteristics of the Shadow DOM should compare favorably to those of iframes. No existing work directly contrasting their performance was obtained, so this assumption was verified experimentally, using the same method as previously. The results are presented in Table 3.1 and Table 3.3. These findings indicate performance *several orders of magnitude* better than that of iframes,

[31]The element has to be accessed through a special `shadowRoot` property (http://www.w3.org/TR/shadow-dom/#attributes-1).

[32]Such as the `::shadow` pseudo element selector or the `/deep/` combinator (http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-201/#toc-style-cat-hat).

[33]The interested reader is directed to http://www.smashingmagazine.com/2013/06/25/media-queries for a treatment of the elusive "element-based media queries".

UserAgent	`<div>`	`<iframe>`	Shadow DOM
Chrome 38.0	0.088	21.246	0.145
Firefox 32.0	0.450	28.947	0.280

Table 3.3: This table presents the raw results of evaluating the memory retention of Listing 3.13, using the memory profiling facilities of Chrome (https://developer.chrome.com/devtools/docs/javascript-memory-profiling) and Firefox (https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler). The experiment was carried out manually, at least 3 times for each combination of browsers and element types, using a sample size of 100 elements, and finally taking the average of the obtained samples. It should be noted, however, that the measurements are highly inaccurate for relatively small changes in memory usage. This is visible in the data, as in Firefox the retained memory for Shadow DOM in fact appears to be *less* than that of regular `<div>` elements. While this seems unintuitive (and is likely in error), the value of the experiment is in showcasing the differences in orders of magnitude between an `<iframe>` and other types of elements, not exact values.

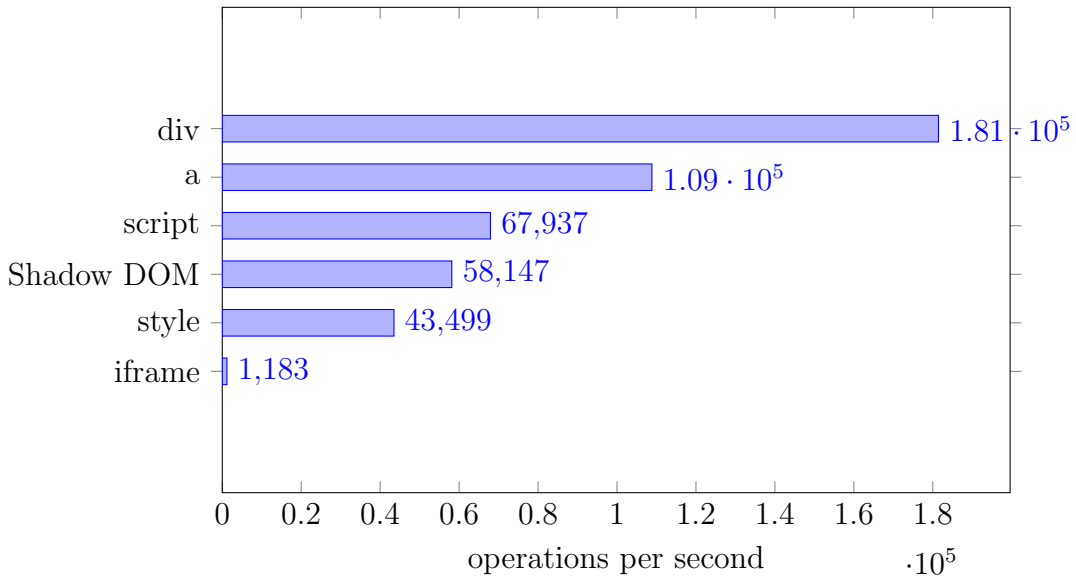

Figure 3.1: Relative creation performance for various element types, summarized from Table 3.1 and Table 3.2. Higher is better.

both in terms of creation cost and memory retention. The differences are summarized in Figure 3.1 and Figure 3.2.

In conclusion, Web Components, and especially the Shadow DOM, are powerful tools for providing isolation within the DOM. Their current support in mainstream browsers is limited[34], though, and thus their impact on frontend application development remains limited, for the time being. In our opinion, however, they are one of the technologies which will likely serve as the foundation of web frontend development when they mature.

3.4 Execution Isolation

The fourth and final class of isolation mechanisms is execution isolation. It pertains to the issues that result from the common lack of separation between functions and the data they operate on. Due to their tight coupling — partly by tradition, partly by language design — it is hard to leverage the potential for parallelization of JavaScript execution in the browser. As frontend application complexity increases, being constrained to a single thread of execution becomes increasingly difficult to cope with.

3.4.1 Evented Programming on a Single Thread

Evented programming is the method of constructing an application so that instead of blocking calls to long-running operations (such as relating to disk or network access), callbacks and an event queue are used. In contrast, in non-evented environments, a blocking call suspends the executing thread until the long-running operation completes. This allows another thread to take over, to make sure the CPU remains utilized. Once the operation completes, the original thread resumes, possibly *pre-empting* (also known as *evicting*) another thread which was in the middle of something else. Thus, the developer needs to take great care in protecting resources that are shared between the threads, as other threads may make modifications to those shared resources at unexpected times. For decades, elaborate concurrency control mechanisms have been implemented to alleviate these problems, but due to the nondeterministic nature of any nontrivial multi-threaded application, they remain a common source of application errors. [8]

With evented programming, on the other hand, there is often only one thread executing the user-land portion of the application[35]. When a call to

[34] http://caniuse.com/#feat=shadowdom reports (and will continue to report) on their support among mainstream browsers.

[35] The expression "user-land" is used here to distinguish between the programming model

a long-running operation is made, instead of yielding the CPU to another thread, a callback is registered. The same thread then continues execution, picking the next event from the queue of events waiting to be processed. Once the long-running operation completes, its callback is added to the event queue, to be processed as soon as the main thread works its way through the preceding queue. The notable difference is that there is no pre-emption: each callback is allowed to execute as long as needed, and to yield the CPU when it is done. This means that there is little need for complex concurrency control: each executed callback may work on shared resources without interruption as long as it needs to. Thus, evented programming has the potential of greatly simplifying an application that deals with long-running operations. [8] The downside to this simplification is that any computationally expensive operations in the user-land thread will block any other callbacks in the event queue from executing.

JavaScript — as a language — is not particularly biased towards either approach: depending on the host environment, some function calls may be blocking and some non-blocking (that is, using callbacks), and in some cases this distinction can even be made during call-time[36]. But as JavaScript execution environments are almost exclusively single-threaded (with mostly experimental exceptions[37]), blocking calls would be very wasteful of CPU. Network calls for example — the bread and butter of a web browser — may take up time on the order of several seconds (or tens of seconds under unfavorable network conditions), during which the JavaScript application would be unable to carry out any meaningful work. Since web browsers also model user interactions as events, the only thread capable of responding to user input would be blocked on the network call, and the application would be completely unresponsive for the duration of that call. For these reasons, JavaScript host environments — web browsers especially — tend to expose the majority of their API calls as non-blocking. The language also makes it relatively easy to deal with higher-order functions, which in turn makes it easy to register callbacks for events. Therefore, most JavaScript is written in an evented manner. [38, p. 66]

---

exposed to the developer, and the underlying mechanisms used to power that model. Indeed, the underlying model may often use several threads for carrying out work, before exposing the results of that work to the single-threaded user-land.

[36] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Req

[37] https://github.com/audreyt/node-webworker-threads is one such exception, which exposes system-level threads into node.js user-land code.

3.4.2 Responsibilities of the UI-thread

While there are slight differences in implementation, and the term thread may not always exactly correspond to system-level threads[38], it is commonplace to refer to the *UI-thread* when discussing JavaScript execution in the context of a web browser. The tasks this thread is burdened with usually include: [53, p. 50] [50]

- **JavaScript execution:** That is, executing JavaScript callbacks from the event queue. This includes all work which is performed in JavaScript (such as arithmetic, data processing), and includes blocking browser API calls (DOM access and most data storage for example), but excludes any work which is performed through non-blocking browser API calls (such as network requests, waiting on timers). User interactions such as reacting to clicks on DOM elements are also processed through the event queue.

- **Reflows:** A reflow takes place when the geometry or layout of DOM elements changes (for example an element is added or removed, or the size of an element changes). During a reflow, the browser recalculates the positions at which each visible element should appear on the page. Depending on the change that triggered the reflow, these recalculations may be small and localized, or affect a large portion of the web page.

- **Repaints:** Whenever the visual appearance of an element is changed (such as when updating the text color of an element), the browser needs to invoke its graphics implementation to redraw the element. A change to the styling of one element may also cause repaints of other elements (for instance, adding a box-shadow[39] to an element may require repainting elements beneath it).

- **Compositing:** After a reflow has established the visible positions of elements, and a repaint has established their appearance, the browser will need to combine the visual representations of each element into the final web page which can be displayed to the user. This process is referred to as compositing. While in some browsers this work can be partially offloaded to a Graphics Processing Unit (GPU), the task of coordinating work with the GPU still falls on the UI-thread [50].

[38] As the smallest schedulable unit of work from the point of view of the Operating System.

[39] https://developer.mozilla.org/en-US/docs/Web/CSS/box-shadow

In light of the above (non-exhaustive!) list of responsibilities, it is unsurprising that for more complex frontend applications, the UI-thread may become oversaturated. [53, p. 52] Even with the advent of mainstream multi-process browsers[40], most of this work is still bound to the UI-thread. This is often attributed to the fact that the necessarily central shared data structure — the DOM — has no built-in mechanisms for thread-safety, and thus mandates confining any related functionality into a single thread. [20, 27] In addition, even in multi-process browser architectures, any web pages with potential for JavaScript access[41] between them need to be hosted within the same process. [36]

3.4.3 Yielding Batch Operations

As previously established, performance is important for modern frontend applications (see subsection 1.1.3 Web Frontend Applications). Traditionally, there has been little the developer has been able to do to alleviate the oversaturation of the UI-thread: after making sure all API calls are non-blocking where possible, and optimizing the access patterns to necessarily-blocking API's[42], the remaining work will have to be performed on the UI-thread. Assuming that even after optimization there are operations which make the frontend application unresponsive for significant time periods, the common best practice is to split those operations into a series of smaller operations (later referred to as *partial operations*), between which other callbacks from the event queue can be processed. This helps in maintaining the appearance of an application responsive to user input, as user input is also processed through the event queue. The recommended approach is: [53, p. 116]

1. When the operation is to be started, instead of starting the operation, create a workspace[43] for it, and schedule a timeout[44] for starting the first partial operation. The duration of this timeout should be very

[40]The canonical example of which is perhaps Google Chrome (https://www.google.com/chrome/).

[41]While some classes of non-blocking access would allow separating these pages into separate processes (see subsection 3.4.5 Message Passing below), as long as there is potential for blocking access between the pages, separating them remains infeasible.

[42]The DOM API is a classic example of an API where different access patterns to achieve the same visible result may have wildly differing performance characteristics; see https://github.com/wilsonpage/fastdom for an example.

[43]The workspace can be simply a set of protected variables which hold the data the operation is to process; see subsection 3.1.2 Emulating Privacy for discussion on the available mechanisms for such protection.

[44]https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers.setTimeout

small, on the order of tens of milliseconds. Store a callback in the workspace which is to be called when the entire operation is finished.

2. Once the event queue gets to the callback of the timeout, look into the workspace of the operation for the data needed by a partial operation, and execute the partial operation. Store the results of the partial operation in the workspace.

3. If there are no more partial operations to carry out, the entire operation is finished (as in #5).

4. If there are more partial operations to carry out, look at how much time has elapsed since #2 started. If less than an allotted time window, process the next partial operation (as in #2). If more than the allotted time window, schedule a timeout (as in #1). The choice of this time window will vary between applications, but as it will dictate the maximum period of time the UI-thread will be unresponsive, it is recommended to be on the order of a hundred milliseconds.

5. If there are no more partial operations to carry out, the entire operation is finished. Invoke the callback from the workspace (see #1) and pass the final results as arguments. Dispose of the workspace[45].

The virtue of this approach is that it allows interleaving the processing of other events from the event queue, maintaining the appearance of responsiveness, while still carrying out the same amount of work. It also makes it possible to make the operation cancelable by the user, which is not possible with a fully blocking implementation. The downsides of the approach are still numerous, however:

- The operation as a whole will take more time. This due to the overhead of scheduling timers, setting up a workspace, et cetera.

- Whether or not an operation can easily be broken down into partial operations is highly dependent on the problem at hand [53, p. 122]. An algorithm that accumulates data from a linear array of datums may be simple to break into partial operations. A recursive algorithm that traverses the DOM may not be.

[45] As JavaScript is a garbage-collected language, if the workspace is properly protected from outside references, simply removing the last reference to the workspace will effectively release its retained resources.

- Yielding the UI-thread between timeout callbacks exposes potential for race conditions. If the partial operations work on a shared, mutable data structure, concurrent access to that data structure may have unexpected effects on the outcome of the operation. Creating a protected copy of the data structure into the workspace protects against concurrent modification, but can be expensive in terms of both memory and processor time.
- Yielding the UI-thread between timeout callbacks makes it possible to re-start the operation before the previous invocation has finished. Assuming proper protection of the workspace, this will not interfere with the previous invocation, but still results in unnecessary work carried out on the UI-thread. To protect against concurrent re-starts, an external locking mechanism can be implemented, though as mentioned in subsection 3.4.1 Evented Programming on a Single Thread, locks as concurrency control primitives are not without problems of their own.

Finally, breaking an operation into its partial operations cannot work around the fundamental limitation of being bound to the UI-thread: given enough work, there will not be enough CPU cycles to complete it all in time. This is where exploiting parallelism necessarily comes into play, regardless of programming language or runtime. In the case of frontend applications, it comes in the form of Web Workers.

3.4.4 Web Workers

Originally introduced as part of HTML5 [45], Web Workers have since been moved to their own specification, which describes them as follows:

> This specification defines an API for running scripts in the background independently of any user interface scripts.
>
> This allows for long-running scripts that are not interrupted by scripts that respond to clicks or other user interactions, and allows long tasks to be executed without yielding to keep the page responsive. [43]

Browsers implementing the specification allow the inclusion of scripts that will be started and executed in a separate thread. This is both in contrast and similar to how regular scripts are loaded: both are fetched by a URL, parsed and automatically executed by the browser. For Web Workers, however, this simply happens without affecting the UI-thread in any way. This means they

will not have access to many of the API's the UI-thread traditionally does; there is no DOM API for a Web Worker, for instance.

Running Web Workers that never end up having effects outside their own thread is comparable to the proverbial tree that falls without anyone there to hear it: without going into metaphysics (and whether it actually executes or not), it is safe to say such a Web Worker is not very useful. In order to have useful, observable effects, the Web Worker has to communicate with the UI-thread[46], which is achieved by *passing messages* between the threads.

3.4.5 Message Passing

Message passing is a well-established method for implementing cooperation between nodes of a distributed system. It has been a subject of study for more than 50 years, since the times of the first computers capable of parallel processing. A 1994 paper providing an overview of message passing environments introduces it as follows (emphasis added):

> **Processors** in distributed **memory** systems have no direct access to the memory of other processors. In order to utilize multiple processors on one **task** it is necessary to exchange information between processors by sending packets of data — **messages** — between them using an available **communication network**. Software libraries to facilitate such exchange of data are called Message Passing Environments. While communication networks vary enormously in detail, they tend to provide broadly similar capabilities for exchanging data. As a result, message passing environments are often **remarkably similar across architectures** [...]. [26]

As message passing is a broadly applicable concept, it is unsurprising that this description also maps cleanly to communication between the UI-thread and Web Workers in a web browser:

- **Processor** maps to a thread executing JavaScript.
- **Memory** maps to the JavaScript heap of a thread. This includes the DOM in the UI-thread, to which other threads have no direct access.

[46] A Web Worker may also have externally observable effects through other means, network communication for instance, though it is questionable whether it is sensible to carry out such work in the browser to begin with (that is, work with only network-observable effects).

- **Task** maps to — for example — the concept of operation (previously discussed in subsection 3.4.3 Yielding Batch Operations).
- **Messages** map to messages as defined by the Web Workers specification [43] — usually strings or simple objects.
- **Communication network** maps to the internal mechanisms of the browser which facilitates message exchange across threads[47].
- **Remarkably similar across architectures** maps to other communication mechanisms available in a modern web browser — such as XHR[48], WebSockets[49] and WebRTC[50] — which offer remarkably similar message passing semantics. That is, the mechanics of passing messages to another thread on the same browser process is in fact remarkably similar to passing messages to another physical machine across the network.

In concrete terms, passing messages takes place by invoking `postMessage()` on the sending side, and registering interest on incoming messages takes place by invoking `addEventListener("message")` on the other side. As messages are received, the relevant callback is executed on the receiving thread. [43] Notably, this API is fully compatible with that of iframes communicating across domains (see subsection 2.5.2 Security).

Contrasting this to the previous discussion in subsection 3.4.3 Yielding Batch Operations, even with the use of Web Workers and message passing, similar issues remain:

- The operation as a whole will still take more time (as measured in CPU cycles spent solving the problem). This is due to the overhead of setting up and communicating with a Web Worker. The specification itself states that "[Web Workers] are expected to be long-lived, have a high start-up performance cost, and a high per-instance memory cost" [43]. The communication overhead relates to the need to perform cloning on any data structures passed between threads[51]: sharing data structures is disallowed to protect against concurrent modification errors.

[47] http://www.chromium.org/blink/web-workers is a concise introduction into how the Chromium project implements Web Workers.

[48] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

[49] https://developer.mozilla.org/en/docs/WebSockets

[50] https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC

[51] https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm

- Whether or not the operation can be easily transferred to another thread remains dependent on the problem at hand: work that involves heavy computation with simple inputs and outputs is easy to transfer, but a recursive algorithm that traverses the DOM may not be.

- Performing the operation on another thread still exposes potential for race conditions: while the messaging mechanism disallows sharing data structures between threads, coordination with another thread often needs additional concurrency control mechanisms. These mechanisms may appear, for instance, in the form of locks to protect against duplication of work, or to maintain a specific ordering of operations.

That is to say, while Web Workers and message passing alleviate the fundamental limitation of performing all work on the UI-thread, they necessarily also increase application complexity. In other words, their use should be limited to cases where the rewards outweigh the related overhead and complications. Instead of defaulting to the use of Web Workers, a frontend application should be architected to allow it, where a need is identified. This need may not be initially obvious: as previously argued in subsection 3.2.3 Storing State in the DOM, the requirements — and thus the application itself — may change often during an application's lifetime.

3.4.6 The Perils of References

Allowing parallelization of the application is not trivial: different kinds of workloads have highly differing potential for parallelization. However, recent work on exploiting parallelism for increased JavaScript performance seems to identify complex jungles of object references as the prime culprit for the inability to parallelize workloads. [20, 27]

As discussed in section 3.1 State and Variable Isolation, both the development traditions and host environments of JavaScript encourage shared global state. The primary mechanism for sharing state is by object references: in JavaScript, everything except simple primitive values[52] are objects. This includes things like arrays and functions. Objects are always *passed by reference*, meaning an assignment of an object to a variable will not duplicate that object, but will instead create a new reference to the same object. [7, p. 20] This is in contrast to primitive values, which are duplicated upon assignment. Since there are no concurrency control features in JavaScript, any — even potential — access to a shared object precludes parallelization

[52]Numbers, strings, booleans, `null` and `undefined` are considered the primitive types of JavaScript.

of affected code paths [27], and as most data is passed by reference, these shared objects are everywhere.

To allow retroactively[53] moving select parts of application execution to another thread, execution has to be as isolated from the UI-thread as possible. In addition to minimizing references to other shared objects (as argued before in section 3.1 State and Variable Isolation), this also means isolation from the DOM is a necessity. Parts of the application which can be modeled with the previously introduced concept of message passing in mind are particularly simple to move to another thread if necessary. It should be noted this does not necessitate modeling such parts as calls to `postMessage()` and `addEventListener()`, but rather making sure the execution of said part is purely dependent on a well-defined set of inputs, and does not affect or rely on shared external data. Modeling real-world JavaScript applications as such is not trivial, however, and the tendency of the language towards shared objects and references does not help. Still, this style of programming — more generally known as Functional Programming [3] — has made inroads into JavaScript mainstream in recent years[54], and the properties it promotes are especially suited for implementing execution isolation in frontend applications.

[53]The need for doing this retroactively stems from the argument that software will inevitably change during its lifetime — and often in unforeseen ways. Thus, places where the benefits of parallelization end up being needed may not be initially obvious.

[54]The popularity of the functional helper libraries Underscore (http://underscorejs.org/) and its fork Lo-Dash (https://lodash.com/) should be sufficient evidence of the mainstream embrace of the basics of Functional Programming.

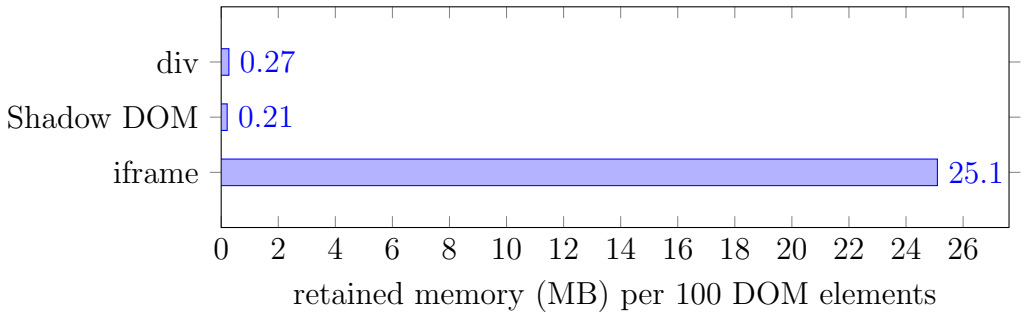

Figure 3.2: Relative memory retention for various element types, summarized from Table 3.3. Lower is better.

Chapter 4

Discussion

4.1 Relationship to Quality

Since we have previously argued that architecture and quality are two fundamentally intertwined concepts (see section 2.3 Relating Quality and Architecture), it is valuable to discuss the quality implications of the previously introduced isolation mechanisms. That is, if we wish to claim employing isolation mechanisms in frontend application architectures has beneficial effects on application quality, we should be able to show concrete connections between said mechanisms and the quality attributes of the application.

To provide a frame of reference for this discussion, the quality attributes defined by the ISO 25010 standard are used (the standard was introduced in subsection 2.2.2 Abstract Models). For each attribute, we hope to either show it is unaffected by the isolation mechanisms, or benefits from them in some way.

Another way of looking at this discussion is as an applied quality model (see subsection 2.2.3 Applied Models), specifically for the context of frontend application architectures. The use cases for such a model could include, for instance, guiding architectural decision-making through previously introduced methods such as ATAM, Attribute Driven Design or Architectural Tactics.

4.1.1 Functional Suitability

This quality attribute was defined as:

> How well the system provides functionality to satisfy pre-defined needs (subdivided into Functional completeness, Functional correctness and Functional appropriateness). [18]

While attributes such as *functional correctness* can be seen as being affected by the application of isolation mechanisms in frontend application architecture, that effect is mainly indirect. For instance, effecting good fault tolerance on the application will likely also positively affect its functional correctness (that is, doing what the application is expected to do), but for the purposes of this discussion the correlation between isolation mechanisms and quality attributes is interesting, and not so much the correlations between quality attributes themselves. No such interesting correlations could be identified.

In conclusion, functional suitability does not seem to be directly affected by the previously introduced isolation mechanisms.

4.1.2 Performance Efficiency

This quality attribute was defined as:

> How well the system performs, relative to the degree of resource use (subdivided into Time behaviour, Resource utilization and Capacity). [18]

Isolation from the DOM can be seen as positively affecting *time behaviour*, as slow DOM access is traditionally a source of performance issues in web frontend applications. Maintaining state separately from the DOM will inevitably require more memory, and thus can be taken as negatively affecting the *resource utilization* of the application, but this loss seems negligible compared to the positive effects it has.

Isolation within the DOM can also have positive effects on *time behaviour*, for instance by limiting the amount of necessary DOM traversal: rooted selectors can dramatically reduce the search space for selector queries. On the other hand, more robust isolation mechanisms such as the iframe can have significant negative effects on both the *time behaviour* (such as having to fetch dependent resources much after page load) and *resource utilization* (for example the retained memory) of an application.

Execution isolation has several beneficial effects on performance efficiency. It positively affects *time behaviour* and *capacity*, as the application can do more computation with more than one thread. It also allows for a more responsive user experience, even without actual additional threads (as in subsection 3.4.3 Yielding Batch Operations). It can also be seen as positively affecting *resource utilization*, as using the techniques discussed in subsection 3.4.5 Message Passing can make garbage collection easier, since references across large object pools become less likely. On the negative side,

almost all methods for implementing execution isolation have additional computational and/or memory costs associated with them, Web Workers especially so.

In conclusion, performance efficiency is directly affected by many of the previously introduced isolation mechanisms. The effects are not entirely positive, either, and require careful analysis before application, to determine their inherent tradeoffs. This is unsurprising in the sense that most improvements in abstraction level, robustness et cetera traditionally carry a performance penalty.

4.1.3 Compatibility

This quality attribute was defined as:

> How well the system is able to exchange information with other systems, and how it performs while operating in a shared software/hardware environment (subdivided into Co-existence and Interoperability). [18]

State and variable isolation can benefit the *co-existence* capabilities within an application by making it less likely for components to interfere with each other, through emulation of privacy and namespace objects, for instance.

Isolation within the DOM has a tremendous beneficial effect on *co-existence*. Rooted selectors, iframes, Web Components and their kin all work towards an architecture where the presence (or absence) of neighboring or nested components have less unexpected side-effects.

Execution isolation also lends *co-existence* capacity to a frontend application, as limiting references across the application (see subsection 3.4.6 The Perils of References) makes it less likely to encounter unexpected side-effects across components through references. Web Workers offer especially good guarantees for this, as they operate within their own memory space, enforced by the browser.

In conclusion, compatibility is directly affected by many of the previously introduced isolation mechanisms, and the effects are overwhelmingly positive.

4.1.4 Usability

This quality attribute was defined as:

> How well the system behaves with respect to effectiveness, efficiency and satisfaction during use (subdivided into Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics and Accessibility). [18]

Much as with functional suitability, usability can be seen as being affected by the application of isolation mechanisms, but only indirectly.

In conclusion, usability does not seem to be directly affected by the previously introduced isolation mechanisms.

4.1.5 Reliability

This quality attribute was defined as:

> How well the system is able to carry out its intended function, with respect to specific conditions or the passing of time (subdivided into Maturity, Availability, Fault tolerance and Recoverability). [18]

Isolation within the DOM can be seen as contributing towards *fault tolerance* of a frontend application, as errors in DOM manipulation are less likely to propagate to the domain logic of the application, and vice versa. Components hosted in iframes can also benefit *recoverability*, as such components can be externally and reliably restarted if they are detected to be in an erroneous state.

Execution isolation, especially through its stronger forms (such as with Web Workers) can contribute to *fault tolerance*, as an error taking place behind a message passing interface will only affect the execution on its other side if an explicit error handler has been registered. That is, errors will either appear at an expected location, or not at all. While the latter solution — essentially ignoring errors — is often not an optimal solution, for a specific class of tasks these kinds of failures can be modeled as timeouts, and simply retried automatically. As with iframes above, Web Workers can also be externally restarted if faulty behaviour is detected.

In conclusion, reliability is directly affected by the previously introduced isolation mechanisms, and the effects are positive.

4.1.6 Security

This quality attribute was defined as:

> How well the system protects information so that only the intended persons or systems are granted access (subdivided into Confidentiality, Integrity, Non-repudiation, Accountability and Authenticity). [18]

State and variable isolation can be seen as affecting security in multiple ways. Emulating privacy allows for *confidentiality* (read access to data can be controlled), *integrity* (write access to data can be controlled) and *non-repudiation* (the sole access point to the data in question can produce an audit trail).

Isolation from the DOM can benefit *confidentiality* and *integrity*, as while the DOM is a necessarily shared data structure across an application (save for some cases discussed in section 3.3 Isolation within the DOM and section 3.4 Execution Isolation), if the authoritative data is maintained elsewhere and only rendered to the DOM as necessary, it remains more secure. In practice, however, an attacker who has access to the DOM seems likely to be able to both read and write data in application memory, regardless of isolation from the DOM, so its effects will be considered negligible.

Isolation within the DOM can also benefit *confidentiality* and *integrity*, in the case where iframes are applied across domains: in that case, the browser will enforce a message passing interface between them, and thus obtaining direct references to the DOM on the other side becomes impossible. This well-defined and controlled interface can also be audited for *non-repudiation*.

Execution isolation also allows for the above properties (when using strong isolation, as in subsection 3.4.4 Web Workers for example).

In conclusion, security is directly affected by the previously introduced isolation mechanisms, and the effects are positive. In the traditional sense of web security (that is, the server being the authority) the client can never be trusted, but in modern and complex web frontend applications meaningful privilege boundaries can exist even within the client-side code.

4.1.7 Maintainability

This quality attribute was defined as:

> How efficiently the system can be modified in the face of changing requirements (subdivided into Modularity, Reusability, Analysability, Modifiability and Testability). [18]

The topic of this work — isolation mechanisms for frontend application architectures — all contribute towards lower coupling between different parts

of the application. Being able to create and modify parts of the application so that the modifications have minimal unexpected effects on other parts of the application is often cited as one of the tentpoles of maintainability in the context of software systems.

In conclusion, maintainability is directly affected by all of the previously introduced isolation mechanisms, and the effects are overwhelmingly positive.

4.1.8 Portability

This quality attribute was defined as:

> How efficiently the system can be transferred between different hardware, software or usage environments (subdivided into Adaptability, Installability and Replaceability). [18]

Isolation within the DOM can be seen as benefiting *installability* and *replaceability*, as self-contained components are easier to transfer between environments. Leveraging iframes make this particularly simple (as do Web Components, to a lesser extent), as their strong isolation guarantees make it possible to transfer components with minimal likelihood of unexpected side-effects.

Execution isolation can also positively affect *adaptability*, as it makes it easier to transfer parts of the application to different environments.

In conclusion, portability is directly affected by some of the previously introduced isolation mechanisms, and the effects are positive.

4.2 Summary

The previous section contrasted the isolation mechanisms presented in this work with the ISO 25010 quality standard. The results of the discussion are summarized in Table 4.1. The key insights are:

1. **Performance Efficiency** was identified as the most controversial of the considered quality attributes. Especially isolation within the DOM and execution isolation have the potential to cause significant performance issues. At the same time, however, they contribute quality improvements across the board.

2. **Compatibility and Maintainability** were (perhaps unsurprisingly) identified as benefitting greatly from the presented isolation mechanisms. This confirms the applicability of isolation in architectures (as

presented in chapter 2 Previous Work) to frontend applications, and is significant since especially maintainability has traditionally been poorly addressed in frontend applications (as discussed in subsection 2.5.1 Spaghetti Code).

3. **Security** was also identified as benefitting from the isolation mechanisms. This is significant, since the traditional discussion on the topic has concentrated on the boundary between the frontend application and the rest of the Internet (as discussed in subsection 2.5.2 Security), whereas here we identified several benefits to security within the architecture of a single frontend application.

Most importantly, we conclude that the application of the presented isolation mechanisms to frontend application architecture has broad, beneficial effects on quality, far outweighing their negative counterparts.

	State and Variable Isolation	Isolation from the DOM	Isolation Within the DOM	Execution Isolation
Functional Suitability				
Performance Efficiency		+ (–)	– (+)	+ –
Compatibility	+		+	+
Usability				
Reliability			+	+
Security	+	(+)	+	+
Maintainability	+	+	+	+
Portability			+	+

Table 4.1: This table summarizes the discussion in section 4.1 Relationship to Quality. "+" signifies positive effects, and "–" negative effects having been identified. In cases where positive or negative effects were identified, but considered negligible, the symbols are in parentheses.

Chapter 5

Conclusions

There are plenty of features in web browsers for enforcing or improving the isolation between the various components that compose a modern frontend application. Their adoption among the mainstream of developers, however, has significant room for improvement. Often this is not wholly the fault of the developers themselves, or even framework authors, as the web frontend platform itself is still fairly immature in its capability to support the development of complex applications. However, we see great promise in the increasing availability of features such as Web Components, Web Workers and countless others. They are laying the foundations for the future architectures of robust, interoperable, high-quality frontend applications. In the meantime, developers should leverage the isolation mechanisms present in today's browsers to already enjoy many of the same benefits.

Isolation mechanisms hold no intrinsic value, but our evaluation indicated that they can have significant positive effects on frontend application quality. There is still a performance penalty to applying the stronger forms of isolation on the architecture of such applications, and in many cases this penalty can even be prohibitive. But web browsers continue to evolve at a staggering pace, and it will not be long until many of these performance penalties have become insignificant enough to make the tradeoffs worth it.

In conclusion, software is eating the world,[1] and a large portion of that software is web applications. Their frontend plays a crucial role in providing the performant user experience modern consumers have come to expect. These same expectations also increasingly apply to enterprise software due to its ongoing consumerization.[2] Web frontend applications enable addressing these performance expectations, and thus have great industry significance.

[1] http://www.wsj.com/articles/SB10001424053111903480904576512250915629460

[2] http://www.forbes.com/sites/darianshirazi/2013/05/08/the-consumerization-of-enterprise-software/

A concrete example of this significance is the money invested in the development of frontend applications. These applications have traditionally had a wealth of quality issues, and bad software quality tends to lead into lost revenue, costly rework, or both. As we have shown, the application of isolation mechanisms to frontend application architecture has the potential to improve upon the status quo.

Most frontend applications are also based on one or more frameworks, and those frameworks tend to dictate many of the architectural design choices. Thus, it is not always fully up to the developer to decide on the application of the presented isolation mechanisms. Assessing the isolation capabilities of existing frontend frameworks is an interesting avenue for future research. Furthermore, in contexts where a specific type of isolation is not only beneficial but mandatory, due to for instance performance or security, a web frontend application framework designed specifically for enforcing these isolation mechanisms could prove valuable.

Bibliography

[1] Akhawe, D., Saxena, P., and Song, D. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 23–23.

[2] Bachmann, F., Bass, L., and Klein, M. Deriving architectural tactics: A step toward methodical architectural design. Tech. rep., DTIC Document, 2003.

[3] Backus, J. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[4] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. Manifesto for agile software development, 2001.

[5] Carlini, N., Felt, A. P., and Wagner, D. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 7–7.

[6] Crockford, D. Javascript: The world's most misunderstood programming language. `http://www.crockford.com/javascript/javascript.html` (retrieved 2014-10-03), 2001.

[7] Crockford, D. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[8] Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., and Morris, R. Event-driven programming for robust software. In *Pro-*

ceedings of the 10th Workshop on ACM SIGOPS European Workshop (New York, NY, USA, 2002), EW 10, ACM, pp. 186–189.

[9] Dijkstra, E. W. On the role of scientific thought. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science. Springer New York, 1982, pp. 60–66.

[10] ECMA. Standard ECMA-262 - ECMAScript Language Specification. Tech. rep., ECMA International, June 2011.

[11] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[12] Fowler, M. Who needs an architect? *IEEE Softw. 20*, 5 (Sept. 2003), 11–13.

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[14] Goeb, A., and Lochmann, K. A software quality model for soa. In *Proceedings of the 8th International Workshop on Software Quality* (New York, NY, USA, 2011), WoSQ '11, ACM, pp. 18–25.

[15] Group, T. S. The chaos report, 1994.

[16] Hardy, A. Javascript architecture. `http://aaronhardy.com/javascript/javascript-architecture-the-basics/` (retrieved 2014-10-10), 2012. Series of posts detailing the author's experiences in architecting enterprise-grade SPA's.

[17] Harisov, V., Berezhnoy, S., and Grinenko, V. BEM - Technology for creating web applications. `https://bem.info/method/definitions/` (retrieved 2014-10-12), 2014.

[18] ISO, I. Iec 25010: 2011: Systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models. *International Organization for Standardization* (2011).

[19] ISO/IEC/IEEE. Iso/iec/ieee systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (Dec 2011), 1–46.

[20] Jones, C. G., Liu, R., Meyerovich, L., Asanovic, K., and Bodik, R. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism* (2009).

[21] Joreteg, H. Human JavaScript. `http://read.humanjavascript.com/` (retrieved 2014-10-07), 2014. JavaScript best practices manual published by &yet, a consultancy company specializing in web applications.

[22] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. The architecture tradeoff analysis method. In *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on* (1998), IEEE, pp. 68–78.

[23] Kohavi, R., and Longbotham, R. Online experiments: Lessons learned. *Computer 40*, 9 (2007), 103–105.

[24] Krasner, G. E., Pope, S. T., et al. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of object oriented programming 1*, 3 (1988), 26–49.

[25] Kruchten, P. *The Rational Unified Process: An Introduction*, 3 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[26] McBryan, O. A. An overview of message passing environments. *Parallel Computing 20*, 4 (1994), 417–444.

[27] Mehrara, M., Hsu, P.-C., Samadi, M., and Mahlke, S. Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (2011), IEEE, pp. 87–98.

[28] Mikkonen, T., and Taivalsaari, A. Web applications - spaghetti code for the 21st century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications* (Washington, DC, USA, 2008), SERA '08, IEEE Computer Society, pp. 319–328.

[29] Mistrik, I., Bahsoon, R., Eeles, P., Roshandel, R., and Stal, M. *Relating System Quality and Software Architecture*. Morgan Kaufmann, 2014.

[30] Offutt, J. Quality attributes of web software applications. *IEEE Softw. 19*, 2 (Mar. 2002), 25–32.

[31] Olsina, L., and Rossi, G. Measuring web application quality with webqem. *IEEE MultiMedia 9*, 4 (Oct. 2002), 20–29.

[32] Osmani, A. Patterns For Large-Scale JavaScript Application Architecture. `http://addyosmani.com/largescalejavascript/` (retrieved 2014-10-03), 2011.

[33] Osmani, A. *Learning JavaScript Design Patterns*. O'Reilly Media, Inc., 2012.

[34] Polillo, R. Quality models for web [2.0] sites: A methodological approach and a proposal. In *Proceedings of the 11th International Conference on Current Trends in Web Engineering* (Berlin, Heidelberg, 2012), ICWE'11, Springer-Verlag, pp. 251–265.

[35] Pressman, R. *Software Engineering: A Practitioner's Approach*, 7 ed. McGraw-Hill, Inc., New York, NY, USA, 2010.

[36] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 219–232.

[37] Snook, J. Scalable and Modular Architecture for CSS (SMACSS). `https://smacss.com/book/` (retrieved 2014-10-12), 2011.

[38] Stefanov, S. *JavaScript Patterns*. O'Reilly Media, Inc., 2010.

[39] Stefanov, S. *Web Performance Daybook Volume 2*. O'Reilly Media, Inc., 2012.

[40] W3C. HTML 4.01 Specification. Tech. rep., Word Wide Web Consortium, 1999.

[41] W3C. Architecture of the World Wide Web, Volume One. Tech. rep., Word Wide Web Consortium, 2004.

[42] W3C. Cascading Style Sheets (CSS). Tech. rep., Word Wide Web Consortium, 2010.

[43] W3C. Web Workers. Tech. rep., Word Wide Web Consortium, 2012.

[44] W3C. Selectors API. Tech. rep., Word Wide Web Consortium, 2013.

[45] W3C. A vocabulary and associated APIs for HTML and XHTML (HTML5). Tech. rep., Word Wide Web Consortium, 2014.

[46] W3C. Document Object Model (DOM). Tech. rep., Word Wide Web Consortium, 2014.

[47] W3C. Introduction to Web Components. Tech. rep., Word Wide Web Consortium, 2014.

[48] W3C. Shadow DOM. Tech. rep., Word Wide Web Consortium, 2014.

[49] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review* (1994), vol. 27, ACM, pp. 203–216.

[50] Wiltzius, T., and Kokkevis, V. GPU Accelerated Compositing in Chrome. `http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome` (retrieved 2014-10-16), 2014.

[51] Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., and Wood, B. Attribute-driven design (add), version 2.0. Tech. rep., DTIC Document, 2006.

[52] Zakas, N. C. Scalable javascript application architecture. `http://www.slideshare.net/nzakas/scalable-javascript-application-architecture` (retrieved 2014-10-03), 2009. Presented at the BayJax group meeting at Yahoo! on 2009-09-07.

[53] Zakas, N. C. *High Performance JavaScript*, 1st ed. Yahoo! Press, USA, 2010.

[54] Zakas, N. C. *Maintainable JavaScript*. O'Reilly Media, Inc., 2012.