ABSTRACT

Title of Dissertation: SOFTWARE-DEFINED SOFTWARE

Moshe M. Katz
Doctor of Philosophy, 2023

Dissertation Directed by: Professor Ashok Agrawala
Department of Computer Science

Modern software systems are extremely complex, and this complexity makes them difficult to create, debug, maintain, update, and secure. Because software systems are an increasingly vital part of just about everything that goes on in the world today, it is imperative that we make it easier to develop and maintain them in order to prevent a breakdown of critical systems, whether due to developer error, cyberattack, failure to apply updates, or any other reason. While there are several existing software development frameworks that are commonly used to split a large application into components, these frameworks can be difficult for developers and administrators to implement, and do not provide detailed visibility into the application's state.

We present a new software engineering paradigm which we call "Software-Defined Software" which can make it easier to work with such complex software systems. We break down complex systems into a set of components and define the methods by which these components communicate with each other and how the overall program state is maintained in a way that allows for significant improvements in the software creation, debugging, maintenance, updating, and

security processes. Following the model of Software-Defined Networking, we separate the application into two layers, the *execution layer* in which the component code is executed, and the *monitoring/control layer* which manages the components and keeps track of the *decision functions* that determine when each component should be executed in response to changes in the application's state information. We discuss how Software-Defined Software can alternatively be implemented at multiple levels instead of one large central state store, allowing existing applications to adopt the benefits of the Software-Defined Software in a gradual process.

We demonstrate two applications of the Software-Defined Software technique. Our first application is a model example of a large-scale online marketplace. For this application, we describe how a software development team would use our techniques to apply Software-Defined Software to an entire application. Our second application is a live production application that is currently used by thousands of customer users. For this application, we describe application of Software-Defined Software techniques to only a portion of the application using hierarchical decomposition. We also explain how our techniques allow for easier maintenance and expansion of this application in the future.

We also discuss several methods of graph analysis to detect potential problems in the application's decision functions, using loop and cycle detection over directed graphs.

Finally, we discuss several future applications which have great potential to benefit from the use of Software-Defined Software.

SOFTWARE-DEFINED SOFTWARE

by

Moshe M. Katz

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor Ashok Agrawala, Chair/Advisor
Professor Adam Porter
Professor James Purtilo
Professor Hanan Samet
Professor Min Wu

© Copyright by
Moshe M. Katz
2023

DEDICATION

לה' הארץ ומלואה

—

שלי ושלכם שלה הוא

ACKNOWLEDGEMENTS

When I was around five years years old, my father brought home our family's first computer. It was running MS-DOS and Windows 3.1. My father mostly used WordPerfect 5.1 for DOS, but he also bought a few games for myself and my brothers. Besides playing the games, I was immediately fascinated by the computer itself, and quickly started investigating as much as I could about how this computer worked. At some point around 11-12 years old, I discovered the computer section at the White Oak branch of the Montgomery County Public Library, and I was immediately hooked. In summer 2005, I further strengthened my desire to study Computer Science when I had an incredible experience in the *Passport* program for high school students. Of course, I never could have gotten from there to here without the help of many individuals.

First, I would like to thank my advisor, Dr. Ashok Agrawala, who also founded the company that made it possible for me to apply my research in the real world. Dr. Agrawala supported and encouraged my research, and we spent many hours discussing research topics, at both a theoretical and a practical level, as well as many other topics ranging from technology to politics and just about everything else either of us could think of. For me, those free-association discussions often produced valuable insights about research, work, or life in general. I started working with Dr. Agrawala when he supervised my CMSC498A credit for working on WaterShed, University of Maryland's prize-winning entry to the USDoE Solar Decathlon in 2011, and it has always been great to work with him. Along with Dr. Agrawala, I would like to thank the members of my defense committee, Dr. Porter, Dr. Purtilo, Dr. Samet, and Dr. Wu. I would also like to thank Dr. Keleher who was a member of my preliminary committee. A special thanks to Dr. Purtilo, whose CMSC435

class became one of my reasons for wanting to stay at the University of Maryland for graduate school, as well as for his willingness to talk about diverse topics, and his patience with me when I was attempting to resurrect work by some of his earlier students on "Dynamic Reconfiguration." While this dissertation does not directly cite that earlier work, reading those papers definitely influenced my thought process during my research. A special thank you also goes to Dr. Cleaveland, who taught CMSC330 in Spring 2011. The discussions we had before, during, and after that class were what planted the seeds of my desire to apply to graduate school and perform research about the ways in which software is developed. Thank you as well to Dr. Foster who taught the other section of that class, who piqued my interest about software maintainability and debugging when he responded "I wrote the entire project in one line" to a student who asked "around how many lines of OCaml code should it take me to write this project?" Thanks to Nelson Padua Perez, who taught the Passport program, and to Fawzi Emad, who taught my first CS class as an undergraduate student, both of who provided a strong foundation for my further experience in the Computer Science Department. Thanks also go to Geoff Ransom, who worked tirelessly in the Computer Science Department's IT offices for many years, for his willingness to talk about real-world software upgrades and systems management. A final CS Department thank you to all the professors with open doors who allowed me to stick my head in and talk to them about their work, my work, or general computer science topics, and to all of the administrative staff who have kept the department running since I first set foot in CSIC in summer 2005. I apologize that I cannot mention every single one of you by name, but you have all made a big difference to my UMD experience.

Next, I would like to thank the people at SaferMobility, who have supported me during my time as a graduate student and have allowed me to implement my research ideas. Specifi-

cally, thanks to Ravindra Passan for all of his feedback and development planning, as well to Dr. Matthew Mah, both of whom spent many hours with me at various whiteboards discussing application structure and maintainability. Thanks as well to the guys at Bridge Technology at South East European University's Tech Park, in Tetovo, North Macedonia, who rigorously pushed our carefully crafted software development plans to the breaking point and beyond.

Thank you to my parents who unwittingly started my fascination with computers, and then allowed me to continue to cultivate it, despite filling large parts of their house with collected computing cast-offs from other people. Thank you also to my grandparents on both sides who allowed me full access to their computers and happily gave me their old hardware to use and experiment on. Special thanks to my twin brother, Yehuda, for working with me for so many years and for sharing so much of the discovery process with me. I don't think words exist to express my thanks to my wife, Michal, for all of her support, especially while I was writing this dissertation. Thanks also to my kids for understanding that I was working on a big, important, and time-sensitive project, and trying to avoid disturbing me if possible.

Thanks as well to Nechemia Mond and Mordechai Hyatt for providing me with my first software development job, working on complex e-commerce software and third-party integrations. This experience provided significant insight for many of the examples in this dissertation.

Writing this dissertation would have been a lot more complicated without the work of Dorothea Brosius at University of Maryland's Institute for Research in Electronics and Applied Physics, who maintains a LaTeXtemplate for dissertations. This dissertation was written in Markdown [33], invented by John Gruber, and was compiled into a finished document using Pandoc [55], invented and maintained by John MacFarlane. It makes use of the `memoir` LaTeXdocument class [99], maintained by Peter Wilson and Lars Madsen. It is typeset in 12pt Gentium Plus [27],

distributed by SIL International, with Hebrew phrases in the acknowledgements and dedication typeset in Keter YG, created by Yoram Gnat.

Last, but certainly not least, thank you to God, who is with me at all times, and who has guided me to reach this moment. אין אנחנו מספיקים להודות לך ה' אלקינו ואלקי אבותינו, ולברך את שמך על אחת מאלף אלף אלפי אלפים ורבי רבבות פעמים הטובות שעשית עם אבותינו ועמנו.

TABLE OF CONTENTS

Dedication **ii**

Acknowledgements **iii**

Table of Contents **vii**

List of Figures **ix**

List of Tables **ix**

1 Introduction **1**
- 1.1 Simplifying Complex Software 3
- 1.2 Organization of this Dissertation 4

2 Other Solutions **5**
- 2.1 (Enterprise) Service Bus (ESB) 7
- 2.2 Service-Oriented Architecture (SOA) 7
- 2.3 Microservices 8
- 2.4 Containerization 10
- 2.5 “Serverless” Architecture 10
- 2.6 Service Discovery and the *Service Mesh* 11
- 2.7 Back to the Monolith 12
- 2.8 Remaining issues 13

3 Simplifying Complex Software Projects **15**
- 3.1 Specification 16
- 3.2 Creation 19
- 3.3 Debugging 21
- 3.4 Maintenance 22
- 3.5 Upgrading 24
- 3.6 Security 25
- 3.7 Problem Summary 27

4 “Software-Defined” **29**
- 4.1 What is Software-Defined? 29
- 4.2 Software-Defined *Everything* (SDx) 31

5 Introducing Software-Defined Software **33**
- 5.1 Components and Decision Functions 35
- 5.2 The shared state 36
- 5.3 Maintaining Program Correctness 39

5.4 Maintaining State Consistency 39
5.5 Component Maintenance 43
5.6 Execution as an Event-Driven System 44
5.7 State Variable Tags 44
5.8 Software-Defined all the Way Down 45
5.9 Security of State Information 49
5.10 Software or Techniques? 51

6 A Complete Application of Software-Defined Software **53**
6.1 Developing a Specification 53
6.2 Component Coding and Testing 56
6.3 Benefits 59

7 Real-world practice of Software-Defined Software **65**
7.1 Case Study - SafeBanker™ 65
7.2 Takeaways 74

8 Analysis Tools **76**
8.1 Program Correctness Using Component Graphs 76
8.2 Program Correctness Using Data Structure Graphs 79
8.3 Program Correctness Using a Hybrid Graph 80

9 Conclusion **82**

Bibliography **84**

LIST OF FIGURES

1.1 Example graphs showing a simple application (left) and a complex one (right), illustrating the difficulty of "seeing" all modules at once. 3

2.1 Example illustrating the relationship between events and notifications. 6

5.1 Example illustrating the shared state being used to perform operations. 38
5.2 Sequence of operations for concurrency control mechanisms. 42

6.1 Example application data model graphs of online store information 54

8.1 Graphs illustrating how component dependencies and information flow can be determined. 81

LIST OF TABLES

4.1 Software-Defined system resources and state for various systems 32

Chapter 1: Introduction

In short, software is eating the world.
– Marc Andreessen (coauthor of early web browser Mosaic)

In the modern world, software systems are increasingly becoming central to the continued functioning of society. Most of these systems are complex, having many components or modules which interact and carry out a variety of processing tasks on multiple inputs the system receives, and generating multiple outputs. Some inputs and outputs are intended for human processing and others are to be fed back into other system components or external systems.

Before the rise of online shopping and social media in the dot-com boom, the vast majority of software applications were fairly simple, typically performing only a single function per application. More complex applications were typically designed and run as a small collection of tightly-coupled modules (fig. 1.1a), often compiled together and executed as a single binary. Applications would typically run as a single instance on a single server, with larger companies performing horizontal scaling by deploying additional servers running additional instances of the same application. Depending on the chosen programming languages and server platforms, this application might be multi-threaded within itself (such as a Java or Microsoft .NET application) or it might require a web server or connection pooling application of some kind to manage multiple single-threaded instances (such as a PHP application or a CGI-executed binary). Applications would typically be very simple and small, and would be the work of a single developer or a small handful of developers. Developers would be expected to keep a "picture" of the application's structure and flow in their minds and to remember most or all of the necessary documen-

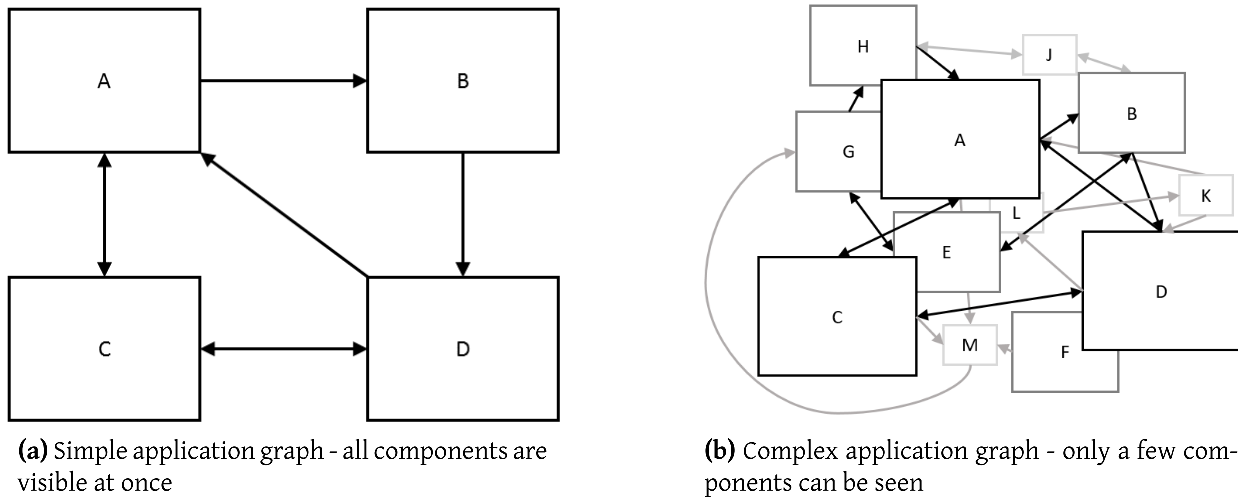
tation for application development and maintenance. The philosophy of the time is illustrated by designers of the IBM System/360 architecture in 1997:

> *...if, after all techniques to make the task manageable by a single mind have been applied, the architectural task is still so large and complex that it cannot be done in that way, the product conceived is too complex to be usable and should not be built. In other words, the mind of a single user must comprehend a computer architecture. If a planned architecture cannot be designed by a single mind, it cannot be comprehended by one.* [8:14–15]

While there were clearly some complex applications before and during the early days of the Internet, they were mostly limited to scientific research and military purposes.

As network connectivity and computing technology advanced, large numbers of users began to connect to the Internet to accomplish various tasks. Companies that already provided services online had to significantly enlarge their online presence, and many more companies joined in building enormous new software systems to handle the influx of new traffic. As a result, the software systems we have today usually have many more modules than did the systems of old, and these modules are less tightly coupled to each other. Along with this increased size and complexity comes increased development and management burden. Modern systems also have many more stakeholders in the application development, including designers, developers, testers, and managers, for each module. It also becomes difficult, even impossible, for a single developer or a small group to keep that mental picture and to be involved in every single aspect of every change or update to the application (fig. 1.1b). Additionally, applications are often scaled by being split across multiple servers such that different components run in different computing environments which may be more closely tailored to their specific workloads.

Figure 1.1 Example graphs showing a simple application (left) and a complex one (right), illustrating the difficulty of "seeing" all modules at once.



(a) Simple application graph - all components are visible at once

(b) Complex application graph - only a few components can be seen

1.1 Simplifying Complex Software

Today, banking, manufacturing, education, transportation, government, and almost all other areas of modern life are using computers for everything, and software systems are growing and becoming more entrenched in society. As this happens, systems become increasingly difficult to maintain at the same time as maintenance becomes more critical and downtime, data loss, and cyberattacks all become more impactful. To combat this issue, it would seem that it is necessary to find a way to simplify these complex systems. It is tempting to suggest that we return to text-only interfaces and binaries that follow the classic Unix Philosophy that each program should "do one thing well" [58:1902], but doing this would put computing back out of reach of most of the world's population who expect to interact with their computers using modern interfaces and to have access to software that performs a wide range of functions in an integrated fashion. Instead, we propose a better way to architect large software systems, combining a new way to model data flow between components of an application with the flexibility and observability of

other modern service-oriented architectures.

1.2 Organization of this Dissertation

The rest of this dissertation is organized into the following chapters. In chapter 2, we describe other historical and current attempts to provide a robust framework and/or methodology for developing large software systems. In chapter 3, we discuss the difficulties involved in large software projects, and describe several specific problem areas in such projects. Chapter 4 provides background definitions as a prerequisite to understanding our approach. Chapter 5 describes our approach, which we call *Software-Defined Software*. Chapter 6 provides a discussion of a theoretical complete implementation of our techniques for an online shopping marketplace, and chapter 7 details a real-world software implementation as part of the backend of a real product with thousands of daily users. Chapter 8 discusses program analysis methods that are possible for software systems that implement our approach. We finally conclude with a recap of our contributions and a discussion of future work in chapter 9.

Chapter 2: Other Solutions

The nice thing about standards is that you have so many to choose from.
– Andrew S. Tanenbaum [89:702]

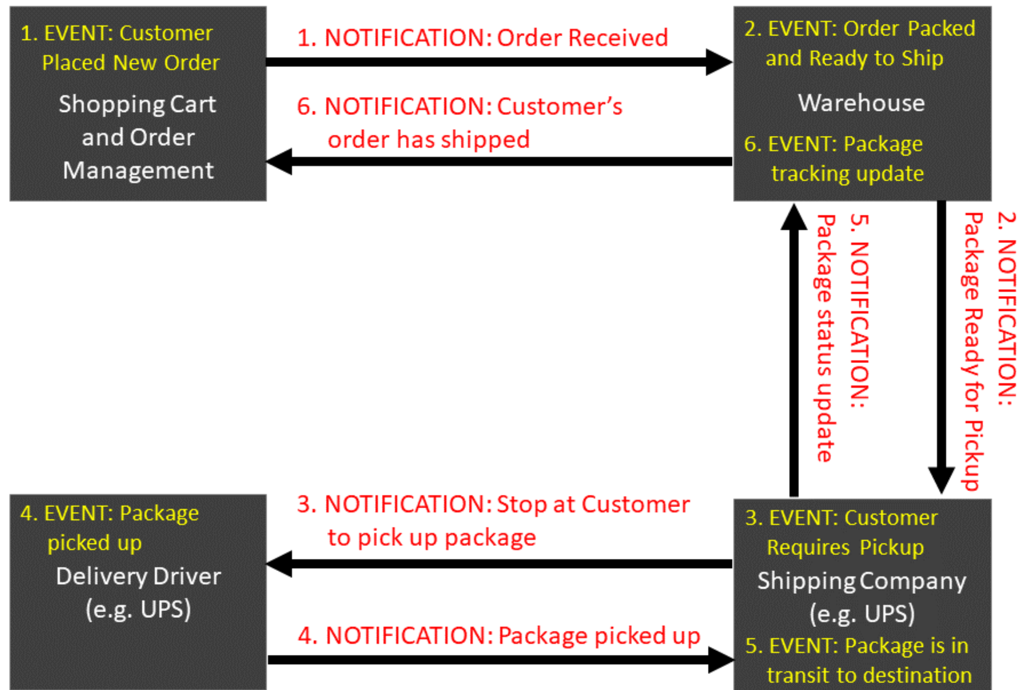
A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. – Leslie Lamport [51]

Problems with monolithic software architectures have been recognized for a long time. There is a strong consensus that the best way to design large systems is to decouple the application components from each other. Stevens et al. [87] first defined *coupling* as "the measure of the strength of association established by a connection from one module to another." Shereshevsky et al. [75] and others have identified the difficulties involved in maintaining, modelling, and understanding highly-coupled systems due to the complexity that high degrees of coupling introduces into a system.

The simplest – and most popular – way to handle communication between these decoupled components is an event-based system, which uses event notifications sent between a set of otherwise-independent components. At its simplest level, we can use the following definitions from Faison [20:71] to describe the communication between components (see fig. 2.1):

- **Event** - a detectable condition within a component that can trigger a notification to other components
- **Notification** - an event-triggered signal sent to a run-time-defined recipient component (or set of recipient components)

Figure 2.1 Example illustrating the relationship between events and notifications.



Event-based systems are among the most ubiquitous software architecture patterns that exist today because of the significant benefits they bring. Besides the simplicity enabled by decoupling components, using events also allows for better responsiveness of graphical components (hence their use in all major modern Graphical User Interface (GUI) toolkits across all modern Operating System platforms), and for ease of scaling systems across multiple separate hardware appliances. Chandy et al. [13] also describe how event-driven systems are particularly useful for large-scale "dynamic applications" which may need to react quickly to sudden changes of requirements.

Over time, a number of architectural solutions have been proposed centered around the concept of Event-Driven systems. Early examples include Smalltalk's Model-View-Controller pattern [50], shared-memory "blackboard" systems [34], modular interconnection languages (MIL) such as MIL75 [17] and POLYLITH [68], and architecture description languages (ADL) such as Rapide [54]. The rest of this chapter describes a selection of solutions that are in common use today.

2.1 (Enterprise) Service Bus (ESB)

A Service Bus is a software communication system based on the common idea of a *bus* in computer hardware architecture. Simply put, a bus allows all connected components to put messages onto it which can be read by all other components. The term *Service Bus* denotes a bus that is designed to allow separate *service* components to communicate with each other. The *Enterprise* Service Bus adds additional components to help manage the needs of enterprise-level organizations to manage the bus itself and its connected services. The service bus concept was one of the first industrially successful and popular paradigms for development of component-based software systems, and the same underlying ideas continue to be used today as the basis for many modern componentized systems.

2.2 Service-Oriented Architecture (SOA)

The primary issue with a traditional bus is that all messages are broadcast to all connected clients. In a large system, this causes clients to be overwhelmed by handling messages that are not relevant to them. The traditional service bus is also designed assuming that each connected component has a unique type of message that it sends and therefore needs no management of component message traffic.

The SOA concept builds upon the primary innovation introduced in the "enterprise" components of ESB, called the *Message Broker*. This building block allows for the creation of complex integrations between dozens – or even hundreds – of components to create a truly large-scale component-based system. The Message Broker itself is a component which acts as the gatekeeper to the bus and routes messages based on some identifying characteristics. This ensures that com-

ponents only receive messages that are relevant to them and that they are not overwhelmed by junk they cannot use. A more advanced level of broker allows for authentication of access to the bus, something that was not necessary in the early days before network-connected systems. A broker also allows multiple components to send similar messages, setting up the ability to have components that split a message queue to balance a load between them.

One major variation on the SOA is the *Distributed Component Model*. Microsoft's Distributed Component Object Model (DCOM), Object Management Group's Common Object Request Broker Architecture (CORBA), and Oracle/Sun's JavaBeans are all popular examples of this model in wide use which allow the design of large software systems spread across a network. All of these systems were theoretically designed to allow distributed component execution to happen automatically and be (almost) as easy as writing the components as if they were running on the same hardware together, automatically routing messages over the network to the components that need them.

2.3 Microservices

A popular complaint about the "traditional" enterprise SOA architecture is that it is too complicated for many projects. Instead of being a simple message-moving skeleton framework, ESB has itself become a component of the application [26], and has gathered so many additional components around itself over time to support the large enterprise application model that it has become its own self-supporting ecosystem. In effect, developers and administrators may spend more effort dealing with the SOA implementation than with the business logic of the application components. While the business components are now much simpler (at least in theory), the SOA foundation itself has become the massive application that is difficult to work with, upgrade, and

secure. For this reason, SOA-based application development was commonly reserved for really large applications written by large companies, while most smaller applications remained monolithic.

Despite those issues, the benefits of an event-based, componentized application as discussed above are well known and developers have tried to figure out other techniques to break up applications to reap those benefits. A new way of doing that is with Microservices.

A typical microservice-based application uses a large number of small components which communicate with each other using a set of published communication endpoints known as an *Application Programming Interface* (API). This usually starts with the application's graphical or other user-facing components, which call APIs on various microservices to execute business logic, either directly or via some kind of API *gateway* that routes the requests to the correct services. Often, applications are designed with microservices that call back to other microservices. Considering an online store as an example, the website or mobile app may call back to a "purchasing" microservice when the user submits an order, while the "purchasing" service calls back to the "credit card processing" microservice before returning a response to the user.

This highly decentralized design is supposed to encourage applications to be built using very large numbers of very small components, providing ease of management and development, ease of troubleshooting, good locations at the boundaries between services to inspect traffic, and high fault tolerance.

2.4 Containerization

As an additional security and ease-of-management measure, many microservices are designed to run in *containers*, which run each component in isolation from the rest, almost as if running each one on its own hardware or virtual machine but without as much overhead. Containers are supposed to be connected to each other and to the Internet by explicitly linking them together in a minimal dependency graph, such that only communication necessary for the application's function is allowed.

2.5 "Serverless" Architecture

The ease of microservice creation and containerization has lead to the creation of a new form of cloud-hosted applications called "serverless" components. Instead of provisioning physical or virtual servers to run microservices, applications can be deployed by telling a cloud provider (or multiple cloud providers) to host the containers and allowing the hosting provider to manage the hardware and software provisioning.

These innovations allow for creation of globally scalable, redundant/fault-tolerant, and secure applications more easily than ever before in the history of computing. However, the price of such myriad levels of containerization and indirection is that it becomes even harder than before to keep track of all the components. For example, how does the "purchasing" component described above know which of a half-dozen "credit card processing" microservices are available to be used? This is true from both a service availability perspective – checking which microservices are running and accepting requests – and from a policy and business logic perspective – checking which microservices give the best rate for a given card type or which ones are "sand-

box" endpoints that only simulate a card transaction when testing the application. This lack of awareness of the structure of the application is a critical problem when fully embracing microservices for application architecture.

2.6 Service Discovery and the *Service Mesh*

Internet forums are filled with stories of server hardware forgotten in a closet but quietly keeping large companies running (until one day the server suddenly fails or someone turns it off because they don't know what it does). If that can happen for physical servers that take up space, consume power, and usually produce sound, how much more are virtualized software services susceptible to being forgotten or misplaced. Microservice-based systems will typically use a *Service Registry* that maintains a list of all running services that are part of the system, and tracks the operational status of each one. To combat the difficulty of finding needed services on demand from among a dynamically changing set, two common methods are used. *Server-side Service Discovery* protocols allow a *request router* or *API gateway* service to query the *Service Registry*, which returns a reference to the component that should be used. A classical example of Server-side Service Discovery is a load balancing application which monitors the state of multiple downstream servers and routes incoming connections only to the ones that are available. Clients make their requests to the router/gateway which forwards them to the correct microservice as determined by querying the service registry. However, this design means that the router/gateway component (which itself may be a microservice) becomes a single point of failure without which clients cannot communicate with the microservices, even if they themselves are still up. To avoid this point of failure, *Client-side Service Discovery* allows the client to query the service registry directly

in order to discover the correct microservices to use. While this removes one component from the path, it transfers the single point of failure from the router/gateway to the service registry. If the registry goes down, the client is helpless, even if the services themselves are still available. While these availability issues can be solved with redundant/failover gateway/router or registry components (respectively), those components must synchronize their state with each other in order to provide a unified view of the available services. All of these components add additional complexity to application deployment and management.

Building on service discovery, a *Service Mesh* also combines load balancing, rate limiting, statistics, logging, and management features into a single foundation (also popularly referred to as a "substrate") that can be used to build a microservice-based application [48]. While this provides valuable services to the application components, components are still supposed to remain largely independent of each other. The primary goals of service mesh systems are usually performance and observability of collected statistics, logs, and performance traces.

2.7 Back to the Monolith

Just like microservices became popular because SOA was "too complicated," there has also been a backlash against microservices, service discovery, and all of the related technologies. While Fowler cautioned in 2015 to "[not] even consider micro services unless you have a system that's too complex to manage as a monolith" [24], that warning was mostly ignored in favor of the perceived ease of microservice development. Others, even proponents of microservices such as Newman [62] (who wrote a book in favor of building microservices [61]), also suggest that microservices are too complicated for a new project and should only be used to split up existing

monoliths that have grown too large. In more recent years, many (e.g. [36,70]) have rediscovered these warnings and argued that because the vast majority of software systems will never need the auto-scaling features enabled by a microservice architecture, the overhead of managing such an architecture is actively detrimental to producing a good product. They liken use of such architectures to using a Formula 1 race car for pizza delivery, which would be an expensive and difficult idea due to the costs of operating and maintaining such a vehicle.

Improper implementation of microservice architectures also can result in ending up with a *Distributed Monolith* [63]. This term refers to a software system that is ostensibly designed as a set of multiple independent services but the services are so tightly coupled that they must be deployed and managaged as if their were a single component. Such systems typically have all the disadvantages of a single-process monolith, as well as the disadvantages of a complex distributed system.

2.8 Remaining issues

One of the reasons for use of microservice architecture over traditional ESB/SOA architecture was supposed to be that a microservice-based architecture is simpler and easier to set up and manage. As we have seen through the issues with Service Discovery, it is apparent that a fully "microserviced" application is every bit as complex as a traditional SOA to maintain. For this reason, many applications today use a hybrid approach - a medium-sized monolithic application surrounded by a cloud of microservices that perform specific tasks. Like the *distributed monolith*, this approach also suffers from the disadvantages of *both* methodologies.

Additionally, there is an issue of visibility into the application state. Applications that end

up as a *Distributed Monolith* have no better visibility than if the application had been kept as a monolith in the first place. Even if an application *does* use microservices properly, unless all microservices are set up to ship their logs and/or metadata back to some central location for analysis, determining an overall picture of application function will still be a significant challenge. Data sources used by each microservice may be difficult to track, connect, and reconcile with each other. With an ESB, it is possible to see large parts of the application state by watching all messages flowing on the bus, something that is impossible in the distributed bus-free world of microservices.

Finally, there is the issue of managing service dependencies. In a complex microservice based application, maintaining a graph of dependencies can be difficult, even with assistive technologies such as Service Mesh. Determining, for example, whether a component can be brought down for upgrades or maintenance requires complete knowledge of which other components depend on this component and how those components will react to the change.

Now that we have seen existing solutions for building large software systems, let us take a step back to clarify the question these solutions are trying to answer. What makes it so hard to build software systems? And what can we do to address that difficulty?

Chapter 3: Simplifying Complex Software Projects

> *[Software] complexity is the measure of the resources expended ... in interacting with a piece of software.* [5]

As a young software developer having gotten into computers in the tail end of the dot-com era, I have lost count of the number of times I was asked to "develop this quick project for me – it's really simple and will take no time at all!" It turns out that it's never actually as simple as they think.

Software projects are hard. Jorgensen et al. [43] found that only 16% of all projects are classified "successful", and only 5% of "large" projects are classified as such. Additionally, while 50-55% of small-medium projects were classified as "acceptable", only 42% of large projects were the same. As early as 1995, Lederer and Prasad [52] reported that only one of every four systems development projects was completed within estimates. While Jorgensen also found that "Agile" methodologies [1] can improve the likelihood of a "successful" or "adequate" software project, the project success rate remains disappointingly low.

Based on the "Agile Manifesto" assertion that "[we value] **individuals and interactions** over processes and tools," [6] many software project management tools and methodologies seek to improve the success of software development projects by directing the behavior of the development teams. There is also a large body of research (e.g. surveyed by Saeed et al. [73]) about estimating development timelines based on these methods. However useful this focus on the human angle of the problem has been, there is still a lot of space to improve the software processes and workflows used by developers in the creation of large software systems.

We have identified six specific problem areas which contribute to the difficulty of development and operation of large software projects: specification, creation (initial development and testing), debugging, maintenance, upgrades, and security. The following sections describe each of these problem areas, including illustrative examples for `Nile`, a vast online product shopping and delivery service named after a powerful water collection and delivery service, accessible at the domain `nile.shop`. The software system for `Nile` must perform many functions, including data entry for new products, business-side customer and order management, a public-facing catalog with shopping cart and ordering capabilities, including options for browsing and searching for products, customer account management, credit card processing, warehouse and shipping management, integration of third party product feeds (such as selling products on Amazon or eBay and advertising on Google or Facebook), and possibly many of other requirements.

3.1 Specification

> *Failing to write a spec is the* single biggest unnecessary risk *you take in a software project.*
> – Joel Spolsky [84]

When developing a large software system, one of the most difficult tasks begins before any code is written - the task of *designing* the system. This is critical since a poorly designed system will often become unmanageable in the future as technologies and requirements inevitably change. The *Joel Test* [83] is a simple test that can be used to roughly evaluate the quality of a software development team. It includes "do you have a spec?" as one of the 12 questions that can be used to define whether a software team is likely to provide a good developer experience and produce a quality product. According to Spolsky [84], non-trivial projects lacking a specification written in plain language "will *always* take more time and create lower quality code." Often, this

will be because the developers working without a specification will end up making unconscious design choices that affect later steps of the software development process. Additionally, without a specification, developers will not know whether they have planned out how to handle all of the business requirements of the software. Other, more complex Software Engineering project management approaches [40] also emphasize the importance of a written specification, but often with a different and more complex focus on metrics and measurability.

Because a "specification" can mean different things to different people, Spolsky [85] contrasts a "functional specification" that describes in plain language how a product will work from the user's perspective, against a "technical specification" that describes the internal implementation of the product in minute detail.[1] (These specifications may also be described as a "top down" view vs. a "bottom up" view of the software project, respectively.) Spolsky notes (based on his years of industry experience, as well as reader feedback on his blog posts) that many software developers conflate the two types of specifications, leading to the conclusion that *all* specification-writing is too difficult and is "a luxury reserved for NASA space shuttle engineers, or people who work for giant, established insurance companies" [84]. Both types of specification share two major weaknesses. First, specifications are guaranteed to change over time. As the product is developed, new decisions are made that may clarify (or completely change) earlier understanding of the product. All non-trivial projects take time; the larger the product, the more time there will be for the understanding – and thus the specification – to change. The environment in which the software will operate is also subject to change, which can also cause changes to the require-

[1] Another type of specification is the "formal specification," which uses mathematical principles and formal specification/verification languages [2,5]. While these are also an important tool, they are techniques that are beyond the scope of most business software development teams, and are therefore outside the scope of this work.

ments and the specification. Second, both types of specification can easily become too complex to be useful. Spolsky [86] describes his experience creating a 500-page *functional specification* for Microsoft's Visual Basic (VBA) implementation for Microsoft Excel, which was implemented by a team of 250 people, including software developers, technical writers, and project managers – all for a *single feature* of Microsoft Excel. A single complete functional specification for the entire Microsoft Excel would be both massive and useless. Purely *technical specifications* for such large systems also run into similar issues of overcomplexity. Tepper [91] explains that one reason for this is that technical specifications are written by developers for whom *writing the software* is more important than *using the software*. Additionally, Tepper describes how software developers often design the logical structure of the software (either explicitly or implicitly including the specification) based on perceived technical merits, without regard to how the end users want to use the product. Fitting the users' desired features into the developers' desired structure can be a significant source of complexity. Another cause of complexity is that a specification filled with minutiae lends itself to *bikeshedding*, a process in which practical planning devolves into endless and meaningless debate over small changes [44].

The `Nile` store will have a number of complex features with non-obvious implementation details. Examples include filtering products by multiple attributes (e.g. clothing by size and color), products being able to be classified in multiple categories, whether multiple coupons can be simultaneously applied to a single order, and whether coupons are in dollar or percentage amounts. While some of these features will be easier to implement than others, all of them have subtle edge cases and questions that must be answered. Without a specification, it is almost certain that the developers will guess the wrong answer to one of those questions.

Properly designed specifications of both types, functional and technical, clearly contribute

to the success of the software development process. How can we enable the creation of better, more useful, software specifications?

3.2 Creation

> *Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.*
> – Alan Kay [23]

> *There are two methods in software design. One is to make the program so simple, there are obviously no errors. The other is to make it so complicated, there are no obvious errors.*
> – C. A. R. (Tony) Hoare [37]

> *The real value of tests is not that they detect bugs in the code, but that they detect inadequacies in the methods, concentration, and skills of those who design and produce the code.*
> – C. A. R. (Tony) Hoare [38]

Even when a software system has a proper design specification, that specification is only useful if developers are able to understand it and properly implement it.

In our `Nile` shopping application, a developer tasked with building a component for managing shipping out of the warehouse will need to interface with the components of the application that handle products, customers, payments, and printing shipping labels, as well as possibly many others. For that component to be developed properly, the developer must understand the design of and interaction with all of those other components.

Another problem that arises is that of testing. Modern software development methodologies emphasize the importance of writing software tests alongside the initial development of the application. Some, such as Test-Driven Development [57], go even further, requiring the tests to be written *before* the application code, followed by writing the code to pass the tests. Many projects use multiple types of testing, including Unit Tests (tests of individual components) and

End-to-end Tests (tests of a complete execution path through the application) in their software development process. However, testing a single component with these methods can be difficult for a number of reasons. First, Unit Tests are designed to evaluate extremely small portions of the codebase, which means that they often do not evaluate the correctness of the interfaces that components provide to other components. Attaining *full coverage* (ensuring that tests cover every conditional branch) using Unit Testing is also difficult, requiring many slightly-overlapping test cases. Additionally, because of the difficulty of coming up with a wide variety of test cases, many tests are written with Congruence Bias [53] - a type of confirmation bias in which testing is done only to confirm that the process deals with correct information but not how the process handles incorrect or invalid information [4,97,98]. Even when tests do include coverage for handling incorrect or invalid information, there still may be many situations that the tests do not cover.[2] End-to-end tests also cannot be used to validate the functioning of a single component, because they do not directly test that component without also being affected by the other connected components. Finally, writing tests for large applications is often seen as too difficult, time-consuming, or otherwise not worth doing for many projects [12], despite being considered critical for software project success [7]. Even for projects which do have tests, Kochhar et al. [49] found that there is a negative correlation between the lines of code and the number of test cases per line of code, showing that testing may become more difficult or less effective as projects grow.

Without development, there would be no software, and without testing, software would be

[2]There is a popular joke about when a software tester walks into the bar, runs into the bar, jumps into the bar, and then crawls into the bar. He orders a beer, then two beers, 1.5 beers, 0 beers, -5 beers, a lizard, and "asnwoudfhalwioej". He leaves, satisfied that the bar is functioning properly. The first real customer walks into the bar and asks for the restroom. The bar immediately bursts into flames.

bug-ridden and unusable. What can we contribute to the software development and testing process to make it easier to create software systems?

3.3 Debugging

> *The problem with obscure code is that debugging and modification become much more difficult, and these are already the hardest aspects of computer programming. Besides, there is the added danger that a too clever program may not say what you thought it said.*
>
> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
>
> *A program usually has to be read several times in the process of getting it debugged. The harder it is for* people *to grasp the intent of any given section, the longer it will be before the program becomes operational.*
> – Brian Kernighan [45]

As we saw described in the IBM developer manual in chapter 1 about designing software, the larger a software system becomes, the harder it is for any one developer to understand it. Because the execution path can wind back and forth through many different components, a developer debugging an issue with one component must necessarily be familiar with the other components reached on this execution path. Visualizing a diagram of the interrelationships between components in mind as well as remembering the values of dozens or even hundreds of different variables often becomes a necessity for fixing even trivial issues.

For most developers, this kind of work requires a high degree of concentration and focus. The concept of "flow" describes "a state of concentration so focused that it amounts to absolute absorption in an activity" [14]. Murphy [59] describes the importance of flow to software developers specifically as applied to debugging complex software systems, as do dozens of blog posts and questions on StackOverflow (e.g. [88]) by software developers wanting to know how to get into a flow state or lamenting the fact that their coworkers bother them and cause them to lose

their flow state. Parnin and Rugaber [67] found that programmers often take 10-15 minutes to start coding again after losing flow state, and that developers often manage to have only a single uninterrupted 2-hour coding session per day. Survey data cited by Griffith [32] indicates that it may take even longer than that to fully regain flow after an interruption, and that 18% of tasks are not revisited at all on the same day after being interrupted.

An additional difficulty in debugging large systems is that it is often required to "step" through the code line by line using a debugging tool. In a complex system, the debugger may need to step through hundreds, or even thousands, of lines of code in order to reach certain critical points. However, in most debuggers, stepping back to a previous line cannot be done – that would be "unexecuting" the program – and an accidental step past the critical line can waste a large amount of time restarting from the beginning of the whole debugging process.

Logically, smaller and simpler programs should be easier to debug because they require less cognitive effort to understand. How can we make debugging large software systems as easy as smaller ones?

3.4 Maintenance

Repeat after me: "Change is inevitable"
– Jim Purtilo, CMSC 435

A good software architecture recognizes that software requirements are constantly changing and designs the software in a way that allows changes to be made easily [29:37]. However, in practice, adding new features to a legacy application can be very difficult for several reasons. First, for many applications without automated testing it can be too easy to accidentally break existing features without noticing that they have been broken. This concern can cause management-level

decisions that actively discourage the addition of new features to "legacy" applications unless there is no way to avoid them, solely in fear of the mess that might result. (While this might be avoided by adding additional testing to the application, creating testing for an existing complex application is itself a difficult, expensive, and time-consuming task.) Additionally, even when new features are added, understanding how to modify the existing application and where are all the locations in the code that must be touched is often a heavy cognitive load for the developers, as described in the previous section.

Another issue is that it is also often difficult to be aware of the state of the application. Most applications act as a "black box" which does not allow for easy inspection of internal state. This lack of visibility makes it hard to understand what is going on inside the application when errors occur, when memory usage is higher than expected, or when trying to plan for increased capacity. While there are many tools that instrument software to help handle these issues, such as software for error reporting that captures variable values and attaches them to the call stack trace created when an error occurs, or special "profiling" builds of the application to record CPU, memory, and/or disk usage parameters, these tools are often difficult to integrate into an existing legacy application and can sometimes have a significant impact on application performance.

Finally, traditional application design often treats the application's own internal variable state as the primary source of truth. Sometimes applications can lose state information when they are restarted as part of scheduled maintenance or due to a software or hardware crash. Often, this causes applications to require long maintenance windows and/or complex startup procedures to maintain data integrity. Applications with this design are also unable to be horizontally scaled (i.e. scaling the application by splitting it to run across multiple separate hardware instances) because each instance is unaware of the internal state of the other instances. This

limitation also prevents running multiple instances of the application on separate hardware for fault tolerance to guard against hardware or network connection failure.

Application stagnation due to the difficulty of making changes prevents the application owner from adapting to changing requirements and environmental factors. How can we enable constructing applications in a way that encourages maintenance instead of discouraging it?

3.5 Upgrading

Defect-free software does not exist.
– Wietse Venema (creator of Postfix)

As we have discussed, software systems require changes over time. Once these changes have been developed and tested, they need to be deployed. For most systems, this is true at multiple levels - the application software must be upgraded, as must its dependencies, all the way down to the Operating System upon which it is running and the firmware that powers individual hardware components. These lower-level upgrades often affect the application, sometimes because they cause a change in computer system behavior that the application must handle, or sometimes simply because they require rebooting the hardware in order to apply the change.

Unfortunately, the difficulty of the process of upgrading an application can vary widely. Some applications are designed to have updates deployed dozens of times a day [18], while others require a complex multi-step workflow with multiple approval, testing, and planning stages before changes can be made, often based on the ISO/IEC 20000 "IT Service Management (ITSM)" standard [41]. The reason for requiring a multi-step workflow for software upgrades is that upgrades of complex systems are themselves complex and can often require downtime and/or changes to other systems as a prerequisite or a result of the upgrade. A failed upgrade process or an up-

grade that introduces problems into the system can lead to downtime, lost data, and unhappy administrators and users.

Due to the great difficulty of such complex upgrades, many organizations upgrade much less often than they might otherwise like to do [15,65,96]. (A well known case of this occurred in 2014 when Microsoft finally stopped supporting Windows XP, which was then 12 years old [46,95]. Thousands of businesses, schools, and government agencies had to begin rush programs to upgrade their systems to Windows 7 or newer, and some - including the British and Dutch governments [25] - had to pay Microsoft for extra support time.) Because upgrades often require downtime, scheduling them for off-peak work times is necessary. In the increasingly globalized environment of today, finding this off-peak time is much more difficult than it used to be. Because the steps are so complex and time consuming, lots of small feature upgrades or bug fixes are often bundled into large bulk updates. In the fast-paced business world today, users are easily upset when the feature they requested takes so long to be added to the product because companies are reluctant to perform updates frequently.

Regular updates are critical to the continued functioning of modern computer systems. Can we provide a framework that eases the difficulty of performing upgrades?

3.6 Security

> *An attacker needs to find only one vulnerability, while a defender must find them all.*
> – Anonymous

The larger and more complex a system is, the harder it is to have confidence that as many vulnerabilities as possible have been found and removed from its codebase. The BugBox project [64] catalogs vulnerability history from several large web-based open-source software projects.

Analysis of data from the project shows that vulnerabilities often survive for many years and through multiple refactorings and rearrangements of the codebase. The observation that software complexity contributes to insecurity is also evident from the OpenSSL "Heartbleed" bug of 2014, which resulted in a number of very high profile "forks" of OpenSSL (including LibreSSL [66] and BoringSSL [30]) specifically meant to remove code and features and make the codebase simpler – and therefore easier to secure. Techniques such as Symbolic Execution [9,39,47] exist to help find subtle-but-dangerous security issues in code, but symbolic execution is slow, memory-intensive, too expensive to run on larger sections of code and with large numbers of variables due to path explosion [3], and not useful if the program depends on multiple input sources not all of which can be symbolically modeled [11,93].

Additional security difficulties exist within the running state of the application itself. First, it is often helpful to classify legitimate or malicious behavior by inspecting the state of the application, as a normal user will usually try to work "with" the application while an attacker will be trying to work "against" it. In a traditional application, it is very difficult to get a view into the application state. Often this is because it is hard to retroactively instrument programs (especially those written in a compiled language) but these needs were not originally foreseen when the program was written. Even if the program can be modified to add instrumentation that captures state, the sheer volume of information coming out all at once from the entire system that needs to be analyzed often means that important datapoints are lost in the noise.

A final security issue occurs when some information in the system must be treated with a different level of security than the rest of the information. For the `Nile` online store, this could include customers' credit card numbers and addresses, which should be available only for the explicit components in which they are needed. Another example is the modern automobile "in-

fotainment" system, which often needs to send information to and receive information from the vehicle's other systems (such as speed data for speed-based volume adjustment or providing the next GPS navigation instruction to the screen in the instrument cluster). However, the infotainment system should not be able to modify sensitive information (such as allowing illegal odometer modification or allowing external access to the vehicle's safety systems via a mobile data connection). It is difficult to design a monolithic application that guarantees this kind of data protection.

Software security is critical to ensure that customer data remains safe and the software remains available for use. With black-hat hacking groups growing in number, size, and proficiency, software teams must remain constantly vigilant to keep their systems adequately secured. How can we make it easier to secure software applications, including both prevention and detection of attempted intrusion?

3.7 Problem Summary

We have described six areas that are common pain points for software development in general and large software projects in particular. This is, of course, not a new problem, and there have been many solutions proposed in previous work. An old riddle[3] provides some insight into the correct approach, asking "how do you eat an elephant?" It certainly seems like an insurmountable task. The answer, of course, is simply to "take one bite at a time." If a large software system can be broken into bite-sized[4] components, it should be possible to address these problem areas

[3] Origin unknown; often attributed to famous figures including Francis of Assisi, Bishop Desmond Tutu, General Creighton Abrams, and various unnamed African, Chinese, or Indian philosophers.

[4] Or, perhaps, *byte-sized*? :-)

individually in the design of the components instead of trying to fix the entire system all at once. However, at the same time it is important to remember that all of those parts came from a single unit, with a central component that directed all of their operations. In the rest of this work, we present our software development framework which is designed to provide better project outcomes for software development projects.

Chapter 4: “Software-Defined”

> *Abstractions are the key to dealing with complexity. Networking is complex because the appropriate abstractions have not yet been defined.*
> – Matt Davy et al. [16]

As indicated on the title page, our solution to the problems described in the previous chapter is called **Software-Defined Software.** On the surface, this term seems redundant – of course software is defined in software! How else could it be defined? To explain this term, we must first start by explaining the general concept of **Software-Defined** systems.

4.1 What is Software-Defined?

At the simplest level, “software-defined” means that a system has a *control plane* and a *data plane* that are decoupled from each other. The data plane is designed in a generic way without any built-in control logic but with building blocks available to perform the required control functions. The control plane is programmed to use those building blocks to perform the necessary functions for the system.

The most well known example of this concept, and the first to use the term “Software-Defined”, is Software-Defined Networking (SDN) [21]. Traditional network architecture has a tight coupling between the control and data planes, which makes debugging configuration problems and predicting or controlling routing behavior much more difficult. This necessitates a mostly-static configuration of routers and switches to direct traffic along predefined network paths that do not change frequently. In contrast, Software-Defined Networking allows network

operators to provision generic routing and switching hardware and to centrally define the routing and switching capabilities using complex software rules that can be easily and dynamically changed at any time. Software-Defined Networking also provides visibility into the network's routing behavior, and allows the control plane to monitor the flow of information across the entire network. The separation of the planes enables network operators to make their network configurations more flexible and configurable in order to support the proliferation of mobile devices, virtualized servers, and cloud services, all of which require more complex routing and other network processing rules than ever before.

Shortly after the popularization of Software-Defined Networking, the term Software-Defined Radio (SDR) began to be used to describe advancements in digital radio processing. Traditional radio hardware is purpose-built for specific frequencies and encodings, necessitating expensive hardware swaps when technology changes or upgrades are necessary. A good example of the difficulty posed by traditional hardware is the FCC's 2010 ban on wireless microphones operating the in 700MHz band (which had been reassigned to public safety radio networks), followed by an additional ban on most of the 600MHz band in 2020 (which had been rearranged to support 5G networks and broadcast television) [22]. These bans led to media production companies having to replace millions of dollars of microphone hardware that could not be reprogrammed to use different frequencies. In contrast, Software-Defined Radio allows for the use of generic/commodity hardware to provision mobile networks, with raw signals fed through analog-to-digital conversion and all processing handled in software [72]. For example, a single cellular device can now be used on multiple carrier networks across the world, despite different countries using different mobile frequencies.[1] Carriers can also dynamically change bandwidth allocations for data or

[1]Note that this still requires appropriately-sized antennas, and may be restricted for marketing and/or regula-

voice communication as needed, and the customers' devices can automatically and transparently handle the change.

4.2 Software-Defined *Everything* (SDx)

More broadly, *Software-Defined* refers to a class of approaches for *dynamic control of a system based on some data within the system.* In the case of SDN, the data used for controlling the system comprises the business requirements for data packet routing. Because of the power and flexibility of the Software-Defined paradigm, it is showing up everywhere in the creation and management of large-scale systems. Examining a few more examples may help us further clarify the meaning and power of "Software-Defined".

- **Software-Defined Data Center (SDDC)**[2] - Popularized by offerings such as Amazon Web Services' "Elastic Compute Cloud" (EC2), this term refers to using automated provisioning to provide all elements of the data center infrastructure – including computing (CPU/GPU), networking, storage, and security services, based on sets of requirements provided by various customers.
- **Software-Defined Mobile Networks (SDMN)** - Popularized along with the creation of 4G (WiMax/LTE) mobile networks, SDMN leveraged SDR to allow for improved scaling of mobile network capacity, no longer needing separate technologies for voice and data communications, higher-quality phone calls, and easier creation of Mobile Virtual Network Operators (MVNO's) which provide services using a larger carrier's physical hardware.

tory reasons; nevertheless, the radio hardware is still capable of this type of communication.

[2]While SDDC is primarily a marketing and not a technical term, the concepts described therein are still a part of the greater "Software-Defined" class of solutions.

Looking at the parallels between these Software-Defined technologies, we arrive at the following definition: **A Software-Defined system provides dynamic allocation, management, and control of resources based on the current state of the system.** Table 4.1 describes the resources and state for each of the four technologies described above. In the following chapters, we will demonstrate how software itself can be *Software-Defined.*

Table 4.1: Software-Defined system resources and state for various systems

System	Resources	Example State
SDN	Network routers, switches, and links	Routing and firewall rules sufficient to deliver packets while preventing network congestion and intrusions.
SDR	Radio transceivers and frequencies	Frequency license information, signal propagation characteristics, Signal-to-Noise Ratios (SNR), and current load.
SDDC	Virtualization hosts, network-attached storage systems, and network configurations	CPU, GPU, and RAM utilization statistics, customers' desired configurations and tasks to be executed.
SDMN	Network and radio infrastructure (as described above)	MVNO allocations, active customer plans, SNR, expected high-density events (e.g. large gatherings requiring additional coverage), and other traffic patterns.

Chapter 5: Introducing Software-Defined Software

> *A Software-Defined system provides dynamic allocation, management, and control of resources based on the current state of the system.*
> (previous chapter)

To improve the software development experience and increase the likelihood of successful outcomes, we present a new paradigm for software development, which we call **Software-Defined Software**. Following the definition of *Software-Defined* from the previous chapter, in Software-Defined Software the "resources" being controlled are *pieces of code* that comprise the application, and the "current state of the system" is the *data that the code is processing*.

A simplistic view of any software system is that it consists of straight-line code interspersed with various decision points which determine the execution path that is followed at runtime. Decisions at these points are based on the state of the system as captured by a number of variables defined for the purpose of guiding the flow of the software. (We note that debugging the straight-line portions of the code is usually rather straightforward.) Most of the complexities in software arise due to either erroneous decision functions (see sec. 5.1) or the use of an incomplete set of variables that does not fully describe the state of the system necessary to make the proper decision.

Instead, let us model a software system as a collection of components (sometimes alternatively referred to as modules). The dividing lines between components are typically chosen to split the "business logic" of the application's code based on its features. Returning to our `Nile` store example, the online store application might have a component that provides an admin-

istrative interface for marketing and sales people, a component that provides an interface for warehouse management, another that provides the public-facing website, another that processes credit cards, a component that communicates with shipping providers, one that provides product feeds to third-party sales platforms, and one that provides product feeds to advertisers.[1]

As in all Software-Defined systems, Software-Defined Software views the software application as two different levels: the *data plane* or execution level (EL) and the *control plane* or monitoring/control level (MCL). The actual code of the application which carries out the functionality of the system operates at the execution level and makes the state of components involved in decision making available to the monitoring/control level. By examining the state of the application in the MCL, it becomes possible to have a complete view of what is going on inside the EL. Using this information along with configuration by the operator, the MCL can add, remove, and/or rearrange components in the EL to alter the program execution as required. (It is possible to apply this concept recursively and have multiple monitoring levels, which we will discuss further in section 5.8. We note here that this allows a system to be designed such that a human operator need monitor only the top level unless it is necessary to drill down into a specific component. This can prevent operator overload through problems such as alarm fatigue from needing to individually monitor all system components.) The visibility into the data structures themselves enabled by the MCL provides a much more useful and clear view of what the application does

[1]Note that the difference between the two final components on the list is in the format of the feeds they provide. The former provides product listings, for example, for the company's own eBay store or Amazon Marketplace account, where the transactions happen only on that platform, while the latter provides feeds of links that direct the user back to the company's main website. Additionally, the former component must accept feedback from the sales platforms about new sales made, while the latter does not have that requirement (because actual sales happen on the company's own site).

and how it works than the opposite view of being able to see the code but not the information, as poetically described by Fred Brooks:

> *Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables , and I won't usually need your flowcharts; they'll be obvious.*
> [10,as explained by 81:111]

5.1 Components and Decision Functions

In the previous section, we initially talked about straight-line code and decision points. The same idea can be applied at a higher level to discrete individual software components which may have one entry point but may exit at multiple possible points depending on values in the information coming in to the entry point. (A component may also have connections to outside systems, such as integrations with third parties or communication with hardware devices.) Let us define the "state" of the application as the collection of variables critical to the outcome of all of the decision points of the application. The conditions to initiate a call into such a software component are captured in this state space and monitored by the level above, and each component is executed if and when the state of the application requires it. Each component has a "decision function" which defines the necessary state changes which are required to cause its execution - each time the state is mutated causing these conditions to be met, the component is executed (see lst. 5.1). When a component is executed, it may have multiple effects, including local effects within the component, such as writing files to disk or making third-party API calls, as well as changes to variables that matter to other components for which updates must be sent back into the state in order to allow other components to use them. While it may be noted that some variables may be changed which will affect the global state while other variable changes may seem to only be

locally relevant, we suggest that practically all state variables may be important to the global state for multiple reasons; first, because they may need to be used by future components added to the system, and second, because monitoring the state space is critical for detecting security and performance issues and leaving information out of the state space hampers those efforts.

Listing 5.1 A sample "decision function" for whether a new order in the online store should be sent to the warehouse management component for picking from the shelves.

```
{
    Order.Paid = true
    AND
    Product.In_stock_warehouses CONTAINS THIS.Warehouse_id
    AND {
        Product.Released_to_public = true
        OR
        Order.Customer_type = RESELLER
    }
    AND {
        Warehouse.Available_shipping_contractors CONTAINS (UPS, Fedex)
        OR
        Order.Customer_will_pickup = true
    }
}
```

5.2 The shared state

Because visibility into the state of the application is so critical to the proper function of Software-Defined Software, it becomes necessary to create a mechanism of collecting the application's state information from the disparate components and centralizing it in a shared state store. In this store, the information can be analyzed to direct the operation of the system. Following the Software-Defined paradigm, it is also necessary to keep storage of application state information out of the application components themselves. Any component that stores its own information effectively hides that information from the rest of the system. That information may be impor-

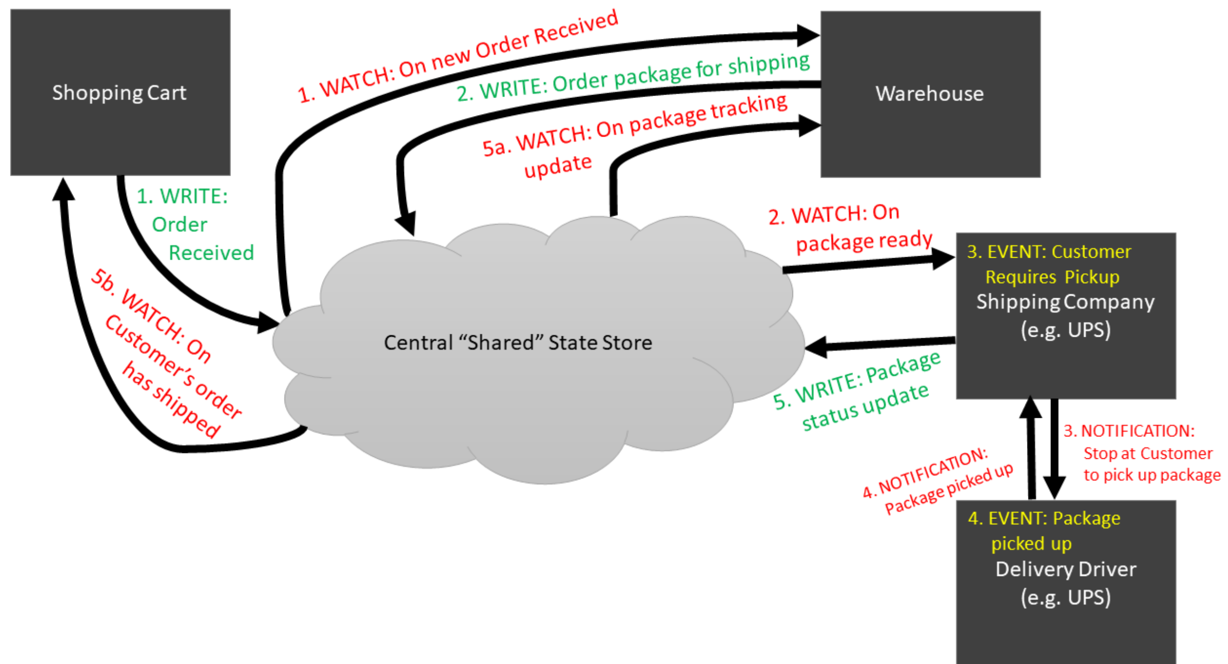
tant to the functioning of the system as a whole, but the monitoring/control components would not be able to use that state information. Therefore, when each component registers itself with the central store, it includes a list of state information that is required for it to function as part its registration. Once a component is registered, there are three operations that it can perform to interact with the state information (example in fig. 5.1):

1. **Read**. A component can request information from the store
2. **Write**. A component can provide updates to the items in the store. This could include the addition of new information, replacement of existing information, or removal of information.
3. **Watch**. A component can request that it be notified of any changes made to specific areas of the state. When another component performs a **write** operation that changes that information, the central store notifies the watcher of the changes.

With these three primitive operations, it is possible to build all manner of complex operations needed for program execution. In addition to the state of the program (i.e. the business logic), the central store also contains the state of the complete execution level itself, such as the types, locations, and names of each of the components, as well as the decision functions for component execution, which can be simply modeled as sets of **Watch** operations. This allows the visibility provided by the monitoring/control level to extend to observing the health of the application itself which is vital to ensure continued application performance.

With all state gathered into a central location, it becomes necessary to define a way to model it to prevent a “mess” (a technical term to describe something you don’t want to work with because it is too hard to make sense of what’s inside it). While many applications model their datastores

Figure 5.1 Example illustrating the shared state being used to perform the same operations as shown in fig. 2.1. (Note that Notifications 3 and 4 are left as they were in the previous figure, as they represent operations performed by an external component – the shipping company – and are therefore not within the design purview of our application.)



as relational (or NoSQL) databases, using simple key-value stores, or using more complex hierarchical key-value representations, the most general data model that can describe all of those types within it is a graph. An operator who wishes to have the store act as if it is a relational database can model the graph as collections of "rows" with the "foreign keys" being edges to other elements, while someone desiring a simple key-value store or a hierarchical tree can easily design those as simple forms of graphs. Another benefit of this architecture is that it has become one that people are very used to interacting with, in the form of the World Wide Web. As Web users, developers today can implicitly understand the concept of the "web" of links that connect web pages (and other information) on the Internet together, and they can easily apply this "web" model to the data structures in their own applications [81:93–94].

5.3 Maintaining Program Correctness

One of the primary responsibilities of the architect of an application which uses the central state storage model is the creation of the decision functions which trigger component execution. If these functions are incorrectly defined or undefined, components will not execute when they are needed or might execute unnecessarily. At best, this may cause self-Denial-of-Service because the application is not performing the operations that need to be done. At worst, it may cause data corruption because components are executing the wrong order and overwriting previous information. However, the most insidious case of an incorrect definition is when the decision functions of two components contradict each other. For example, the `Nile` developers may decide that an order is only presented to the warehouse to be shipped once the credit card is charged, but might separately determine to charge the credit card only after checking for available stock in the warehouse. If these functions are not written properly, they will end in a deadlock and never execute, each waiting in vain for the other's update to the order record.

As we will discuss further in chapter 5.9, we require components to have their read and write permissions explicitly defined. Using those definitions, we can draw connections between the decision functions of all components in application the and analyze the graph of those connections. Graph analysis techniques will be described in further detail in chapter 8.

5.4 Maintaining State Consistency

Another issue that arises when multiple components are writing to the same state storage location is proper management of locking and precedence. If multiple components try to update the same state variable at the same time without some kind of locking or transaction mechanism,

whichever update is received first will be overwritten by the second and data loss will occur.

There are two well-known methods of locking the information store to prevent these issues. In order to allow developers to build a wide variety of applications, we describe implementation of both mechanisms and allow the implementer to choose which one is better for any specific application.

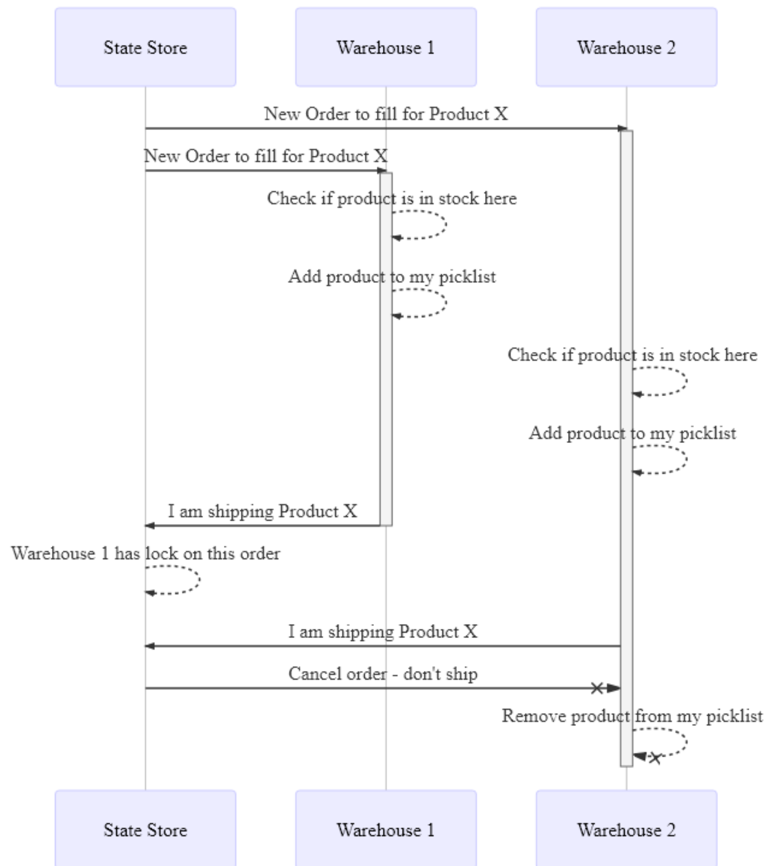
1. **Transactions with rollback** (fig. 5.2a). If the operation to be performed is cheap to execute and it can be easily rolled back if it should not have been done, it may be most efficient to use a transaction-based locking mechanism. In this mechanism, the central state starts a "transaction" for each component at the time the component's entrypoint is called. When the first component to finish returns a result back to the state store, that component's transaction is committed and the changes it provided are applied to the central state. The variables that have been changed are also tagged (see section 5.7) as having been modified. When the second component returns, the state store sees that the variable tags show that they have been modified already since the transaction began, which causes it to reject the transaction and send a message to the second component to roll back any changes that it has made internally and return to its previous state. An example of a case where this should be used might be using multiple services to perform a lookup of an item name based on some other piece of identifying information. In this case, whichever lookup service returns a result first is used and all the others are ignored and discarded.

2. **Explicit Locks** (fig. 5.2b). If the operation to be performed is expensive, time-consuming, or difficult (perhaps impossible, such as instructing a machine to perform a physical action) to roll back, an explicit locking mechanism can be used. With this mechanism, the central

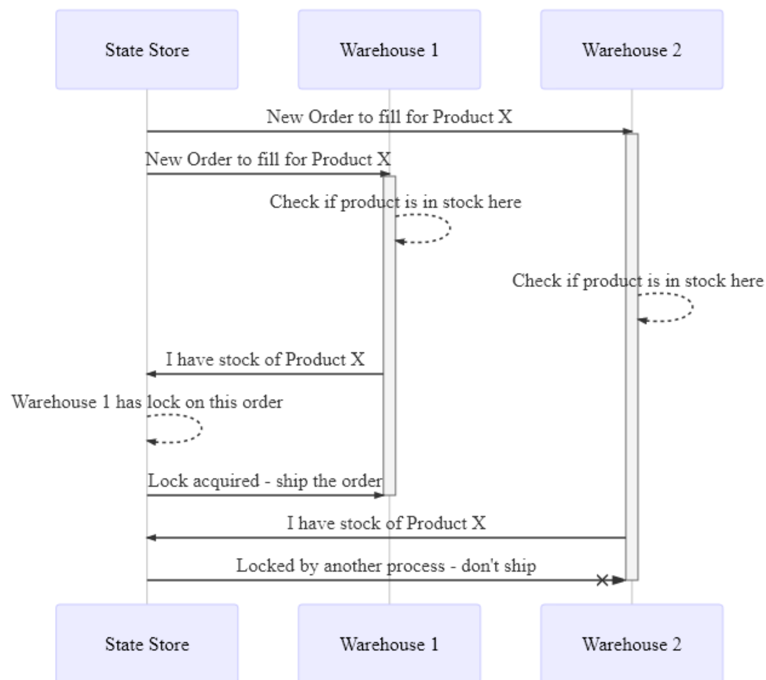
state notifies each component that it should start executing. Each component requests a LOCK on the information it plans to modify. Only the first lock for each variable is accepted, and that component is allowed to send back modifications to that part of the state when it completes. All other components which did not receive the lock may choose to wait for the lock to be released or may choose to cancel execution entirely (assuming that component execution is mutually exclusive such that only one of the components need run). An example of a case where this might be used could be where multiple warehouses are sent order information for shipment. Each warehouse internally verifies whether the item is located there and all of the ones that have the item in stock request a lock on the "order shipment status" variable. The first warehouse to respond will be awarded the lock and be told to pull, pack, and ship the item while the others will be told to disregard the request.

Despite the provision of these two concurrency mechanisms for the state store, we suggest that most applications will make very limited use of these constructs. These constructs should be reserved only for the case of multiple components performing identical (or otherwise equivalent) operations where the reason multiple components exist is to allow for load balancing and/or high-availability/failover capabilities. If there are multiple separate operations which must be run which all modify the same variables, analysis tools (to be discussed in chapter 8) will reveal that multiple functions are being triggered by identical conditions and will suggest that the decision functions be examined to see if they can be made more specific to avoid contention. In a broad sense, three components with the identical decision function `CONDITION A = TRUE` might be able to be modified such that the function for the second one is `CONDITIONS A & B BOTH = TRUE` and the third is `CONDITION A & C BOTH = TRUE` - then the completion of the first func-

Figure 5.2 Sequence of operations for concurrency control mechanisms.



(a) Using transactions with rollback



(b) Using explicit locks

tion sets `CONDITION B` to `TRUE` thus allowing for the execution of the second component – which sets `CONDITION C` to `TRUE` thereby allowing for the execution of the third component. While this example is admittedly trivial, there is no end to the flexibility of decision functions to efficiently direct the flow of execution.

5.5 Component Maintenance

Unlike maintenance of a traditional system, which typically occurs at the code or configuration file level, maintenance of a Software-Defined Software application is primarily about addition, replacement, or removal of components. While components still need to be worked on at the code level, administrators at higher levels do not need to concern themselves with those modifications. (Further, in the case of recursively nested levels of components to be described in section 5.8, the administrator responsible for each level need worry only about the changes at that level without the levels above or below it as long as the overall guarantees of expected input and output types do not change.) While this is also true of other types of component-based systems, what those systems lack is the ability to look at the overall state to get a clearer picture of the impact of any modifications performed. For example, Heineman and Councill [35:528–529] explicitly define most components as black boxes about which nothing is known internally and over which the administrator has no control other than reconfiguration within the system as a whole. Rombach [71] and Erdogmus and Vandergraaf [19] also warn about the difficulty and complexity of component maintenance in traditional componentized software systems.

5.6 Execution as an Event-Driven System

As we have seen, unlike traditional software systems in which the developer defines the critical paths on which component execution occurs, in this model, the components of the system are dynamically executed based on the state changes. In effect, this event-driven design takes away the notion of a single path of execution that runs through the components and binds them together into a single unit and replaces it with nothing more than the collection of components themselves. The necessary conditions for component execution are embedded directly in the component definitions.

Returning to the definitions in the description of traditional event-based systems in chapter 2, we can now explain the terms “event” and “notification” in a manner specific to this framework:

- **Event** - a program state in which the decision function for a component has its conditions met and determines that component should have a notification triggered.
- **Notification** - a set of program state variables necessary for execution of a component, which are sent to the component when the requirements of its decision function have been met.

5.7 State Variable Tags

We note that when the computer executes traditional software, it only manipulates the representations of variables in the form of raw bits. These bits do not contain any description of their meaning which is derived only by the ways they are defined and manipulated in the code of the program. This information, instead of being encoded and contained in the state of the application, is actually contained in the mind of the application designer, who may or may not have

entered some of it in the form of variable names and comments in the source code. Software-Defined Software developers may choose to extend the idea of variable representation within the central state management component by associating tags with all variables of significance in the software. When a variable is captured by the monitoring level, all of its associated tags come with it, thereby making its meaning clear. This data can be used by data analysis tools, intrusion/exfiltration detection systems, and/or performance metric collectors to provide significant added insight into program execution. Tags may also be used to indicate data classification levels for privacy protection, to allow the state store to transparently anonymize or block access to sensitive data.

5.8 Software-Defined all the Way Down

> ***It's turtles all the way down!***
> – Anonymous rebuttal to a well-known scientist[2]
>
> *[A programmer has] the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large. When you're writing a program, you're saying, "Add one to the counter," but you know why you're adding one to the counter. You can step back and see a picture of the way a process is moving. Computer scientists see things simultaneously at the low level and the high level.*
> – Donald Knuth [100]

Without the central state structure described in chapter 5.2, it would be impossible for a Software-Defined Software system to have the visibility necessary to make execution decisions. However, complex information data structures lead to complex graphs, which are difficult for

[2]This phrase is the punch line to a famous joke, in which a renowned professor has just finished a lecture in Cosmology, or some other related area of physics. He is approached by an audience member, who declares "all of what you said was very interesting, but none of it is true. The earth is actually supported on the back of the enormous *World Turtle!*" The scientist replies, "oh, really? What is supporting the turtle?" The audience member replies "another turtle, even bigger." The now-exasperated scientist replies, "and what is *that* one standing on?" The audience member retorts, "you foolish man, it's turtles all the way down!"

developers to visualize. As we discussed in chapter 3.3, one of our goals is to reduce the size of the mental picture of the software that developers need to maintain.

We can significantly simplify the graph structure by first observing that most components need to access only small portions of the complete state. Typically, components focus on particular data structures within the overall model. Therefore, we design individual components such that they view the state space as a projection of the graph into a tree-ish structure instead of as a complete graph. (Unlike a pure tree, there can be connections from a leaf to multiple parent nodes; the point is that we focus on a single node as the root and choose what we want to see below it solely based on what data this component needs for its execution.)

Once using these projections, we can observe that some information does not matter to the global state. For example, the `Nile` developers observe that most components do not need to know anything about payment transaction details. Processing payment in the Credit Card component generates data such as authorization codes and transaction IDs, which no other component needs. Most other components that care about payment information only need to see success or failure messages. Even in the case of failure messages, payment processor instructions say that some types of credit card failures (such as stolen cards) should *not* be shown to the user, instead being replaced by a generic error.[3] Although we originally asserted in chapter 5.1 that *all* infor-

[3]We also illustrate an example situation in which the `Nile` developers try to decompose the state space, only to learn that this part of the space needs to remain globally available. The developers observe that the Warehouse component needs the shipping address, the Credit Card component needs the billing address, but the full list of addresses should be kept at a lower level within the User component so the user can add, remove, or modify address information. When they suggest that this means the address information should be moved out of the central store, they realize that the Shopping Cart Checkout component also requires the full list of addresses to be able to choose specific addresses for each order. Additionally, they would like to implement fraud detection that checks whether

mation *might* be relevant to the global state, we modify that assertion here to say that as long as the information is available at *some* level of the hierarchy, it is a small exercise to move it to a higher level (if other components need it) or a lower level (if other components no longer need it) of the state space.

In addition to decomposing the state information, we can also decompose the components themselves into subcomponents. For example, the `Nile` Warehouse management component may handle multiple tasks, including choosing which warehouse(s) will fulfill an order, picking the correct packaging size based on product dimensions, printing labels, and directing employees to fill and seal the packages. Instead of a single complex component that handles all of these jobs together, the Warehouse component will itself act as a central state for subcomponents that each perform a single function. None of the other system components are concerned with internal warehouse operations, but the individual warehouse components still follow Software-Defined Software design principles. This design also helps prevent congestion at higher levels because irrelevant information no longer bubbles all the way to the top level.

With these observations, we refine the component input-output rules as follows. Components do not communicate with siblings or cousins, only with parents and children. Components receive state updates from above and below. Components send information up the hierarchy to the parent component when this component or a descendent requests to modify the state. Components send information down to children when this component receives updated information from above or changes at its own level trigger child decision functions. To enable observability, execution of a component *always* causes a changed state to be sent to the parent component in

the address is far away from other addresses the user has used before as one of the fraud scoring factors. For these reasons, they determine that this information *does* need to remain in the global state.

order to make a record of execution, even if no additional data changes are caused by this component's execution. If all components were only communicating to the central store, such updates might overwhelm the actual application information. With hierarchical state, a component that collects execution information from its child components can summarize that information for the parent component, and operators desiring additional details can drill down into the child components based on the summary. In this way, hierarchical decomposition can also provide performance benefits for large applications. Effectively, we have made every lower level of the application able to act as its own execution layer and monitoring/control layer for the components in the level below it.

Given the ability to have a hierarchical component layout, we tried to determine whether it makes more sense to define the components and describe the state from the top down (i.e. starting from a more general level) or from the bottom up (starting with individual operations). In real-world planning meetings with software developers discussing the implementation described in chapter 7, we found that different development team members find different perspectives more useful. However, we found when using a bottom up approach that some developers got lost in the weeds, arguing about minutiae and bikeshedding. We also found when using a top down approach that some developers were ready to stop the conversation too early before enough detail had been worked out. We suggest that a less-formal organic approach to the discussion that grows from both ends to meet in the middle may provide the best insight into a well designed application structure. We note that such an approach would not be viable in a traditional distributed system, in which state is maintained only within components and not centrally. It only works for Software-Defined Software because of the ease with which we can move information between hierarchical levels.

5.9 Security of State Information

Information security is an essential requirement for all modern software applications. This requirement includes security of information both in transit and at rest. The central store is able to provide guarantees of both of these properties.

Information security in transit is ensured using existing security standards for interprocess communication, including enforcing that all connections to the state store happen over protected channels, usually using TLS, DTLS, and/or QUIC (choosing the appropriate technology as required for a specific application). Authentication of the components themselves (to prevent spoofing) may also be done through standard TLS Client Certificate mechanisms or other public/private key technologies, which are managed through the component registration process itself.

Two other information security mechanisms related to protecting data in transit are Intrusion Detection/Prevention (IDS/IPS) and Data Exfiltration Prevention (DEP). The well-defined connections between the central state management and the small, specialized execution components allow Software-Defined Software systems to provide IDS/IPS and DEP systems with full access to look for suspicious data access patterns or other indicators of attacks.

Security at rest has two components - that the information is encrypted on disk and that access to it is controlled. The level of data encryption required (as well as the algorithms used) will vary widely depending on the user (for example, depending on whether FIPS-certified cryptography is required for a government application), and so can be done using standard practices for the Operating System platform on which the software is deployed. Access control is done through the component registration process. In order for a component to be allowed to register

with the central store to handle information, the operator must have defined a security profile for that component (or for the group or type of component that it is). A security profile provides a specific description of the type of information that the component is allowed to read and write. Read and write attempts to information not on this list for a specific component will be denied. Again, because all of the information about security is also part of the application state accessible to the MCL, it becomes very easy to spot potential compromises as monitoring alerts can be generated for attempts to access information without permission.

A common security problem found in many modern systems is that configuring the system following the principle of least privilege [74] for all components can be such a daunting task that operators give much more access than the components actually require. If and when a component is compromised, the attacker has access to read all of the extra information that should not have been available and sometimes even gains write access into other parts of the system. Because of the flexible graph-based structure of the state store, it is possible to define the type of access allowed for each component using tree-like projections of the graph. This is especially useful given that the intent of splitting the system into components is that each component perform a specific action or set of actions; therefore each component can be given a view of the information that is most convenient for the purpose it serves. Providing the explicit map of which information from the graph is required for each component (including whether such access is read-only or read+write) means that it is much easier to visualize the permission requirements and therefore much easier to ensure that they are met.

5.10 Software or Techniques?

While we considered developing and offering a pre-made open-source "central state management" component that would serve as a foundation for Software-Defined Software Development, we determined that different business requirements such as database/storage needs (e.g. relational or document databases), types of hosting (e.g. public/private/multi-cloud, "bare metal"), application data structure, resiliency requirements (e.g. hot-spare/failover, multi-master), and many other factors mean that any premade software component would be too limiting to be widely useful. Any library or framework that we could provide would necessarily include our own assumptions about how the application will be structured around it. Our own choices of message formats or storage systems could seem perfectly reasonable for our own test cases, while being a poor fit for other applications. As evidence of this, we note that chapter 6 and chapter 7, while both being example implementations of Software-Defined Software, are quite different from each other and a single one-size-fits-all state management component would not be able to meet the needs of both applications. Instead, we present Software-Defined Software as a set of techniques that can be applied to any software project. The greatest benefit of this approach is that Software-Defined Software is not an all-or-nothing proposition. As the Netscape Navigator team discovered in the late 1990's, it is a very risky business decision to completely rewrite working software from scratch [82]. We also recognize that the industry has found that building a modular *greenfield* system from the ground up can be much more difficult than upgrading and modularizing an existing *brownfield* system [62]. While we encourage the use of Software-Defined Software for a complete application, hierarchical decomposition of the state space described in section 5.8 means that the techniques can be applied equally well to small parts of the appli-

cation at a time, either in the case of migrating an existing traditional application or for a new application in which the developers want to introduce new techniques slowly and cautiously. For these reasons, chapter 6 will describe a theoretical complete implementation of Software-Defined Software, while chapter 7 will describe an incremental implementation that we have brought to production use at a real-world software company.

Chapter 6: A Complete Application of Software-Defined Software

Let us return to our example online store, `Nile`, and describe how developers for this application would implement it from the ground up using Software-Defined Software.

6.1 Developing a Specification

6.1.1 Writing a *Functional Specification*

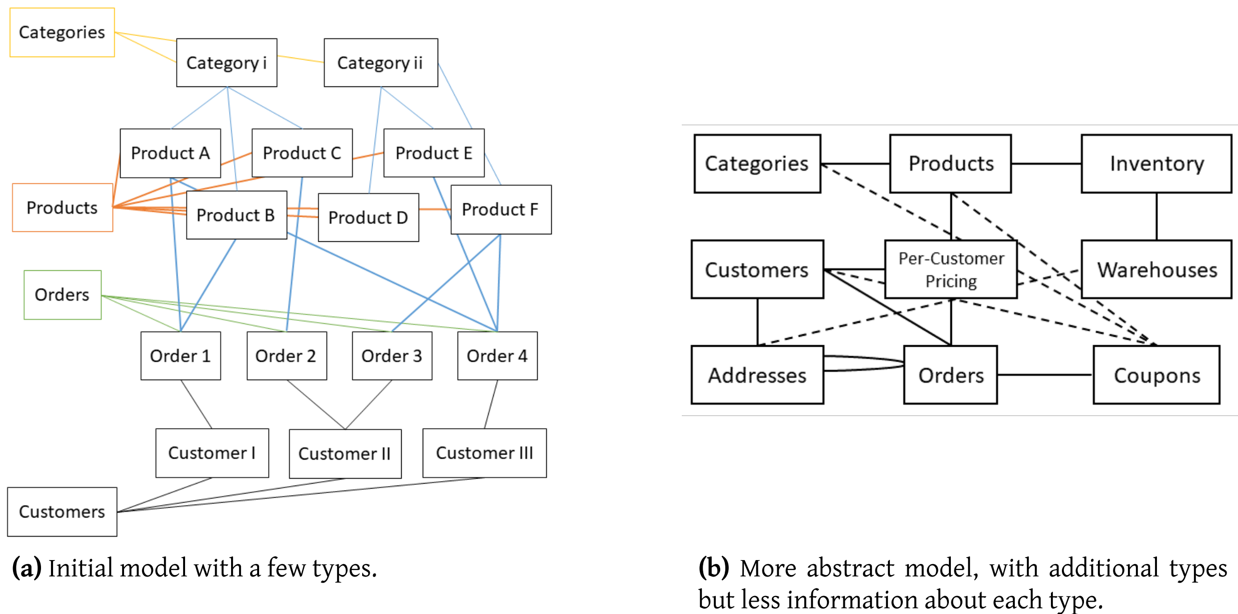
The `Nile` developers start by writing a simple *functional specification* of their product. This document provides an overview of what the software is expected to do, examples of how users will interact with it, and simple descriptions of the designs for various pages. Listing 6.1 shows a portion of this functional specification.

6.1.2 Modeling the Application Information

Next, the `Nile` developers use the functional specification to build a rough model of the application's data structures, as described in fig. 8.1b. As shown in fig. 6.1a, an early version of their model might start by describing the relationships between categories<->products, products<->orders, and orders<->customers. Further refinements of the model add descriptions of additional relationships, such as links to product brands, product variants (e.g. colors, sizes), related/suggested products, warehouse stock, credit cards, and shipping providers. As these further refinements are added to the model, the model becomes too complex to represent in a simple

two-dimensional graph as shown in this figure. The developers may try to continue building the model by increasing the level of abstraction, as shown in fig. 6.1b. However, even with a higher abstraction level, the model still becomes too complicated to represent as a comprehensible graph drawing.

Figure 6.1 Example application data model graphs of online store information



(a) Initial model with a few types.

(b) More abstract model, with additional types but less information about each type.

6.1.3 Component-specific views of the model

Although we have seen that a full application data model may be too complex for a single flat representation, it was necessary for the `Nile` developers to perform this exercise in order to familiarize themselves with the basic structure of their data model and give themselves the opportunity to discuss and refine the model. The next step they take is to build the tree-like projections of the data model that individual components will use. While we have seen that the full application state model forms a graph, each component of the application will be designed to focus on particular parts of the model with particular relationships. For example, the component which handles

credit card processing has no need to be aware of details about products and categories. For security, the catalog and warehouse components should not be able to access the customers' credit card information. While both of these are examples of information that is *not needed*, it would be dangerous to develop the software using `denylisting` of information that should *not* be sent to each component, because future data model additions might be sensitive and because sending extra information needlessly fills network bandwidth that could be better utilized for other things. Additionally, each component must only be allowed to write specific portions of the data model, even if it is allowed to read from larger portions. For example, the components that are used by shoppers visiting the website should not be allowed to modify inventory listings, although they can be allowed to see aggregate item availability. To meet these access control requirements, the `Nile` team designs the application using `allowlisting` by making each component describe exactly what information it needs to read and write in order to perform its function. Listing 6.2 shows the lists of information needed for two example components, the credit card processor and the warehouse order packing manager. As illustrated here, many components require only a few elements of the complete state in order to perform their functions. Others, such as the product catalog shown in listing 6.3, require much more information, but that information is still only a small subset of the full graph and can be easily represented and understood.

6.1.4 Data Transfer Objects

There are many possible software development techniques to ensure that proper access control is maintained for such complex data structures. The `Nile` team uses Google's Protocol Buffers (often called `protobuf`) [31], which provide a language-neutral, bandwidth-friendly, and easily-

extensible method of transferring information between the central state store and the application components. The team first defines protobuf `message` types that include a full representation of the information, which can be used internally inside the central state store. Listing 6.4 shows an initial implementation of the `Customer` and `Order` protobuf message definitions.

With the complete definitions in hand, the team creates specific message types for requests and responses for each type of component. Listing 6.5 shows the message definitions that are used by the credit card processor component. It is here that the extensibility of the protobuf message format is particularly important. As we discussed in chapter 5.9, it is critical that components follow the principle of least privilege, only having access to the specific information needed to perform their functions. At the same time, components must be able to be updated with new features that may require new information. The extensibility of protobuf messages is specifically designed to allow components to maintain both forward and backward compatibility as message types are changed to meet changing application requirements.

6.2 Component Coding and Testing

With the specification and planning process complete, the `Nile` developers start to develop individual application components.

6.2.1 The Central State Store

The `Nile` developers create a software component that will act as the central repository for all significant application state, as described in chapter 5.2. In a nod to the Isaac Asimov classic short story, "Catch that Rabbit" – in which a single robot manages and analyzes the state of a

whole group in order to direct the actions of its subsidiary robots which actually perform the work – they decide to name this component **DAVE** - which also represents the functions it performs - **D**efinition (of the software program), **A**nalysis (of the state information), **V**erification (of incoming and outgoing information), and **E**xecution (i.e. managing the components in the execution layer). As described in the previous section, this component communicates with the other application components by sending and receiving `protobuf` messages. The developers decide that the central state for this project will be backed by a PostgreSQL relational database, with a table corresponding to each of the "base" protobuf `message` types. Using PostgreSQL's built-in replication along with popular open-source failover/high-availability tools allows the developers to focus on developing business logic and allows the IT team to keep working with familiar tools. For each component interface (i.e. set of `Request/Response` protobuf messages), the developers decide whether to create one or both of the following interfaces:

1. **Unary request/response** - This interface performs the **Read** and **Write** functions described in chapter 5.2, allowing the component to query the central state from **DAVE** and write updates back to it. Each request is processed individually and receives a single response.
2. **Streaming request/response** - This interface performs the **Watch** and **Write** functions, allowing the component to be notified by **DAVE** of changes caused by other components. Each component opens a channel to **DAVE** and waits for messages to be received on that channel.

For security, the developers decide that initially the **decision functions** (chapter 5.1) will be set inside the **DAVE** configuration. While this sacrifices some flexibility by not allowing compo-

nents to dynamically define their own decision functions, it also prevents a poorly-designed or rogue component from requesting more information than intended, which could lead to Data Exfiltration or Denial of Service vulnerabilities. Testing of decision functions is provided by connecting the test code to the **Watch** channel(s) and sending a **Write** request to ensure that the appropriate **Watch** channels are notified.

As noted above, **DAVE** is built using a PostgreSQL database as its backing storage. In addition to the database tables needed for the application's business data, additional tables are created for application metadata. The metadata stored in these tables is used for monitoring and controlling the connected components. It includes information about currently connected components and configuration, component invocation history (with configurable input/output logging levels, allowign increased visibility or reduced data transfer/storage requirements), and error logging, which allows the team to monitor the application health in real-time.

6.2.2 Application Components

Now that they have a specification and a central state management component, the `Nile` developers create the individual application components. Each component is developed independently, and must be able to function in isolation. For each component, developers write two types of tests. *Unit tests,* which test individual pieces of code, are built using common testing frameworks available in the programming languages used for each component. *Integration tests,* which test the inputs and outputs of the component, are built by providing a set of inputs and expected outputs. The test framework will run the tests by providing the input and checking that the output matches the expected value. Depending on which component is being tested, the exact form of

the input will vary. Inputs can include information coming from the central state and/or connections to other sources (such as *mocked* 3rd-party APIs or incoming HTTP requests from users), and outputs can also include information to be sent back to the central state and/or information to be sent to other targets.

6.3 Benefits

In chapter 3, we described six problem areas in the development and maintenance of complex software projects. Throughout this chapter, we have described the application of Software-Defined Software techniques to the `Nile` online store application. We observe that this application provides improvements in all six problem areas:

1. **Specification** - The specification development process is broken into logical chunks to produce a usable application specification with clear and concise descriptions of all components and the interactions between them.
2. **Creation** - The development and testing process is significantly simplified because each component can be built and tested completely independently from all other components.
3. **Debugging** - Because components are simple and single-purpose, each component can be debugged in isolation, using sets of known inputs and expected outputs. Because the central state store tracks component execution and allows logging component input and output values, it is easy to identify problem components and create additional component test cases based on the logged data.
4. **Maintenance** - Because components have clearly defined request and response message formats, changes can be made to individual components without concern for cascading

changes into other components. If a component does need new information in a request or response, the `protobuf` format is designed to help ensure that there will be backward compatibility between old and new versions of the messages.

5. **Upgrades** - Because all components run as independent services, upgrades to individual components can be made quickly and easily without affecting other parts of the system. Because components do not maintain internal state, it is easy to move them to different hardware in order to perform Operating System or hardware upgrades.
6. **Security** - Because components are simple and single-purpose, application of Symbolic Execution and other Dynamic Analysis tools is more likely to provide full coverage of the application's code. Because each component can only communicate with the central state using predefined message types, an attacker who compromises the security of one component is limited to accessing the only information that specific component is able to access.

Listing 6.1 A portion of the functional specification for `Nile`

```
# Functional Specification for Nile.shop
## Overview
`Nile.shop` is a marketplace that connects retailers and customers
using a shared online catalog and purchasing workflow, and
delivers products from shared warehouse spaces as soon as the
same day and typically not more than three days later.

## Scenarios

1. Joe owns a business that manufactures toys. Joe would like to sell
   toys all over the world, but he is mystified by the complexities of
   sales tax laws in other states and countries and he is haunted by
   nightmares of incorrectly-calculated postage preventing items from
   reaching his customers. Joe is terrified that his website will be
   hacked, and knows that it will be very expensive for him to market
   his shop directly to customers. Joe would be happy to pay a portion
   of his sales revenue to a company that will display his products in
   an online marketplace, accept payments, and handle shipping.
2. Jane wants to buy toys as presents for her grandchildren. She
   knows that they all like different types of toys, but she is unable
   to drive around to multiple physical stores to search for toys that
   would make them happy. Jane is scared of having her identity stolen
   online so she prefers to buy everything from one store in order
   to minimize exposure of her personal information. Jane has a small
   apartment without room to hide the presents, but she does not want
   them arriving at her grandchildren's house too early, so she would
   like to be confident that the delivery will arrive on a specific day.

## Non Goals
1. This application will not prevent harvesting user information to sell
   to advertisers.
2. This application will not validate the provenance or safety of
   marketplace listed products.

## Example Screens
### Home Page
On this page, users will see lists of products that are algorithmically
chosen to be products this user is likely to want to purchase or
products related to other products the user has recently purchased.
At the top of the page, there will be a navigation bar with popular
shopping categories, a search box, and links to the user's account
and shopping cart pages.
```

Listing 6.2 `Allowlists` for information (read/write) for order processing components

```
# Component: Credit Card Processor
## Receives Information:
- Customer:
  - Name
  - Billing Address
  - Credit Card
- Order:
  - Product total ready to bill
  - Transaction Type: AUTH/CAPTURE/REFUND
## Returns Information:
- Order:
  - Amount Billed
  - Transaction ID

# Component: Warehouse Order Packing
## Receives Information:
- Customer:
  - Shipping Address
- Order:
  - Products:
    - SKU
    - Size/Weight
    - Inventory Locations
  - Shipping Method
## Returns Information:
- Order:
  - Products:
    - Ready to bill
    - Shipping Status

## Notes:
To make sure customers are not charged for out-of-stock items,
credit cards are only charged after the warehouse staff have
"picked" the item and prepared it for shipment.
```

Listing 6.3 `Allowlist` for information (read-only) for catalog component

```
# Component: Product Catalog
## Receives Information:
- Categories
  - > Products (see below)
  - Subcategories -> Name
- Brands
  - > Products (see below)
  - Storefront Information
    - > Custom Categories
- Products
  - Product Variants (size/color)
  - Reviews
  - Related Products
  - Special Deals
```

Listing 6.4 `protobuf` messages representing customer and order objects

```
message Customer {
   string id = 1;
   string name = 2;
   string email = 3;
   repeated Address address_book = 4;
   repeated ProductList wishlists = 5;
   repeated PaymentMethod stored_cards = 6;
   repeated Order orders = 7;
   repeated Pricing special_pricing = 8;
}

message Order {
   string id = 1;
   string coupon = 2;
   sint32 total = 3;
   Address billing_address = 4;
   Address shipping_address = 5;
   message LineItem {
      string sku = 1;
      int32 quantity = 2;
      sint32 item_base_price = 3;
      map<string, string> options = 4;
   }
   repeated LineItem items = 6;
   repeated string transaction_ids = 7;
}
```

Listing 6.5 `protobuf` message representing information for credit card processing

```
message CreditCardRequest {
   string customer_name = 1;
   string customer_email = 2;
   Address billing_address = 3;
   PaymentMethod payment_method = 4;
   sint32 amount_in_cents = 5;
   enum TransactionType {
      UNSPECIFIED = 0;
      AUTH = 1;
      CAPTURE = 2;
      REFUND = 3;
   }
   TransactionType action = 6;
}

message CreditCardResponse {
   sint32 amount_in_cents = 1;
   string transaction_id = 2;
   string error_code = 3;
   string error_message = 4;
}
```

Chapter 7: Real-world practice of Software-Defined Software

In chapter 6, we described a complete application that has been built from scratch using Software-Defined Software techniques. However, we recognize that established development teams may not want to learn a completely new development paradigm. We also would like to provide the opportunity to apply the benefits of Software-Defined Software to existing applications. For these reasons, we now describe how Software-Defined Software techniques can be applied to individual components of a larger software system, including as an upgrade to parts of existing applications. To do this, we will take advantage of the decomposability of state space described in chapter 5.8.

7.1 Case Study - SafeBanker™

SafeBanker[1] is a mobile and web application designed for bank employee safety and regulatory compliance. The Bank Protection Act of 1968 [94] and subsequent additional laws and regulations require banks to establish procedures for opening and closing for business, along with other security requirements to discourage robberies, burglaries, and larcenies. SafeBanker tracks employee performance of these procedures, which it calls *tasks*, and sends alarm signals to a security monitoring center in the event of any trouble during the task. Tasks must be completed before a timer expires, otherwise an alarm signal will be sent to the monitoring center, to all other employees at that location, and to regional and/or corporate managers. Customers can define checklists of

[1]The same product is also marketed under the name SafeResource™ for Credit Unions and other non-bank customers.

items that must be done during the task, and the employee performing the task must check off all of the checklist items. Employees can add additional time to the timer if more time is needed to complete a specific task. SafeBanker also allows audiovisual and chat connections to the monitoring center, whether during task performance, after task expiration/alarm, or any time a user decides there is a need to contact the monitoring center. SafeBanker notifies all other employees of the bank branch, both for successful task performance and alarms, in order to keep the other employees safe and aware of what is going on in their location. Other product functions include Emergency Notifications (ENS), daily employee surveys, and various account management and report generation capabilities.

The original SafeBanker server-side infrastructure was developed in 2015 as two separate systems: a single component, called the "controller", that handled audiovisual connections using WebRTC, and a monolithic application, called the "webapp", that handled all other functions. Because the original specification did not fully describe the requirements for multiple security companies and product lines, both the controller and the webapp require separate instances of the application to run for each monitoring center and product brand. The original designs of both components also maintain significant application state within the application itself, precluding the ability to run multiple copies for load balancing or high availability. The connection between the two components was also not fully defined, and the web and mobile frontend applications need to communicate separately with both backend components. The two components have needless duplication of various features, such as two-way text message chat, which is implemented once on the "controller" for text messages that accompany video calls, and separately in the "webapp" for silent text-only conversations. Legacy code in both components is structured in a way that makes it difficult to add new features without breaking existing features. Many

functions are tightly coupled to other parts of the components, making individual component testing almost impossible. Due to historical design choices, most functions in the webapp reimplement their own versions of the authentication and authorization code with minor tweaks for each function, meaning that most changes to permissions management must be made in several dozen places in the codebase. Both components handle foreground and background processing, meaning that updates to any feature within the component require a full component restart which affects the background jobs as well.

The company decided that it would be necessary to redesign the server-side applications using a modern approach that is more decoupled and testable. We worked with the company to develop a specification for the new backend services, and to choose implementation technologies which allow them to realize their goal of having more maintainable and expandable products. The company would have liked to use the complete Software-Defined Software approach, but had tight deadlines for implementation that would not have allowed junior developers to learn and become familiar with the full Software-Defined Software technique. Instead, the company chose to implement all non-real-time (e.g. user/location management and report generation) features using an existing open-source framework which includes reliable implementations for authentication, authorization, database operations, and other common application features, and encourages modularity and testability. Recalling from chapter 5.8 that Software-Defined Software can be applied at multiple levels of the application, the company decided to apply these techniques for critical subcomponents of the server components that handle real-time (also called "live") features of the application, including running opening, closing, and service tasks and other timed tasks, as well as audio, video, and text chat functions. Specifically, the Software-Defined Software techniques are applied for the "rule engine" that decides whether conditions are met to

allow execution of an incoming request from a user's device.

7.1.1 Specification

A portion of the specification for handling real-time events is shown in listing 7.1. This portion specifically deals with starting performance of a bank branch opening task. Other portions are very similar, and are omitted for brevity.

This implementation will focus on one key phrase from the "example endpoints" section of the specification:

> *The system must check the rules for whether this task can be performed right now as requested.*

Rules for task performance can vary significantly, depending on many factors. Rule sets for many common task types include some of the following example rules:

- The location cannot currently have an alarm. If the location has an alarm, it must be resolved before normal operations can be resumed.
- The location must not already have another task in progress.
- The location must be in a specific state. For example, a location can only be opened if it is currently closed.
- The user must be within a certain distance ("geofence") from the location. Users should not be able to run most tasks unless they are physically present.
- The user must have permission to perform this task. For example, only certain users may be authorized to open the vault.

- The task may need to be performed within a certain time frame. For example, closing may be allowed only between 5:00-5:30 PM on Monday through Thursday, and 5:30-6:00 PM on Friday.
- The user's device clock must be close to correct. When working with timed tasks, the user's device needs to accurately show how much time is remaining.
- Each checklist item can only be performed once per task.
- Checklist items might be able to be performed in any order, or might be required to be done in a specific order.
- Checklist items may need to be performed by multiple users, such a vault opening which must be done by a team of two users who each perform part of the process.
- Most tasks require entering a PIN to end the task, though some may be allowed to end without a PIN. Most tasks also allow a "duress" pin to be entered that looks like it ended the task successfully but also sends a silent alarm signal.

Individual customers can customize the rules for specific tasks, as well as customizing the tasks themselves. The rule engine needs to look at the status of the user and location, as well as the task definition and customer settings, to determine whether a task is allowed to be started, updated, or ended.

As mentioned earlier in this chapter, we must explain how this implementation takes advantage of state decomposition. We do this by viewing the non-real-time part of the application as the top level central state management component which holds the single source of truth for the application. For this reason, we also refer to this component as the "central" server. The live server performs **read** operations from the central server by doing HTTP API calls, and **write**

operations back to the central server by `INSERT`ing data into a database table. At the next lower level, the mobile applications and rule validation components treat the live server as the state management component which holds the single source of truth for the user, location, and task that are currently being handled. Mobile applications perform **write** operations to the live server to request changes, and **read** the responses from the live server to update the user interface in the app. Rule validation components perform **watch** operations on the live server by implementing a `Condition interface` which the live server can call with the state data. Rule validation components perform **write** operations on the live server by returning a response to the **watch** call; this response can either be an error message, or a `nil` value which indicates that the current data conforms to the rule's condition.

7.1.2 Development and Testing

To ensure that the live server accurately represents the actual state data, all live server operations for a single client request are performed inside a single database transaction[2]. During this transaction, the live server first obtains a lock[3] on the location ID that is being worked on. This informs other instances of the application that this location *potentially* has a task in progress, and they must wait for the lock to be released because the truth about this location may change.

Each rule component must conform to a simple API interface: it receives required information from the current state data and the input data from the mobile device, and returns an error if

[2]To be clear, "client request" here means "a single API call." Starting a task performance, checking each checklist item, and ending the task performance are all separate transactions; the lock is not held for the entire duration of the task timer, only for the duration of the rule processing.

[3]Using the PostgreSQL feature "Advisory Locks" [92].

the rule fails or `nil` if the rule passes. Some rules may also make additional **read** requests back to the live server to request additional data that would have been costly to look up and is therefore only retrieved and calculated on-demand if the rule requires it. Additionally, two “meta-rules” exist to combine other rules. Both of them take a list of other rules as the only argument. The `AllOf` rule returns `nil` only if all of its child rules return `nil`. The rule can optionally “fail fast” on the first error it encounters, or it can process the full list of rules and then return an error that encompasses all of the errors that were found. The `OneOf` rule returns `nil` if *any* of its child rules returns `nil`. If none of them returns `nil`, it returns an error that encompasses all of the errors that were found.

7.1.3 Benefits

Because the code for each rule must contain only what is necessary for that rule, and all rules must explicitly declare which data from the central state is needed, it is simple to write unit tests for all of the rule implementations. Integration tests for the live server endpoints can treat all of the lower-level rules as a black box, and ignore the specific implementations, or can do things like testing the rule choice process itself by providing a sample input and verifying that the list of rules that is generated for that input includes all of the correct and relevant rules. During this implementation process, most of the initial implementations of the rules were based on code copied and translated from the old backend. During the testing for some rules, it became apparent that there was a subtle edge case that had been handled incorrectly in the old backend code. Due to the inability to test the old code at the component level, there had been no easy way to detect this bug. Enabling testing at the component level has already shown the benefits that

Software-Defined Software techniques have brought to this project. Additionally, we discovered that one of the tests was incorrectly passing when it should have failed because the developer transposed greater-than (>) and less-than (<) symbols in both the rule and the test itself. In this case, it was tests at the higher level of the live server that revealed the error. This was a clear example to us that there are benefits to the decomposition approach. We had originally planned to deliberately introduce some subtle bugs in order to have developers find them and use this exercise to test and demonstrate the usefulness of Software-Defined Software over a traditional approach. Instead, we found that no such deliberate introduction would be needed, as we found and fixed these existing bugs immediately upon application of Software-Defined Software to this application.

7.1.4 Swapping Components

One benefit of the simple component-based architecture is that components can be designed so they conform to the same interface but have different underlying implementations. This allows components to be replaced with minimal effort. SafeBanker uses a *media server* that manages WebRTC connections for audio/video calls. The old "controller" component was hard-coded to communicate with a media server running on the same computer. It was unable to do load-balancing of simultaneous calls or failover if there was a problem with the media server. One of the `Rule` components of the new system checks whether the media server is available before it allows an audiovisual call to start. If no media server is available, the caller will receive an error message and be told to call a phone number to speak to an operator by telephone. We have provided two implementations of this component, one that communicates with a single chosen media server,

and the other that maintains a list of multiple media servers and chooses one of them based on current availability. Using Software-Defined Software, it is easy to choose which component to use (such as choosing to use one server for staging/demo servers and multiple servers for production). A future extended version of this component will choose a media server based on CPU load and/or physical location, not only service availability. This future version will be equally as easy as the existing two versions to drop into the system because it conforms to the same interface.

7.1.5 Adding a Feature

One feature that is currently missing from the SafeBanker product is the ability to track multiple users performing a task together. To maintain employee accountability, tasks such as opening the bank vault typically require two employees to work together, each performing a specific part of the task. Using the old backend components, adding such a feature would have required rewriting most of the code that handles task performance, as the code was originally written to support only a single user per task. Using the new component-based architecture, the process for adding this feature is simple:

1. **Identify which data structures must be modified.** For this feature, the `Task` structure must have information added about how many users are expected to perform each task. For compatibility with existing tasks, this number will default to `1`.
2. **Identify new or changed conditions.** Only some of the `Rule` components deal with the information about the user who is performing a task. Any components that do not need this information can continue to exist without changes. Any components that do process the user information will need to be evaluated to see if they need changes to handle more

than one user.

3. **Add any new components.** There will be new rules that need to be added to handle the new feature. For example, some multi-user tasks will allow any of the users to perform any of the checklist items, while others will require specific users to perform specific items. We will add a rule that checks if a specific user is required and then checks that the correct user is performing each part of the task.

7.2 Takeaways

While the resulting software code in this application is completely different from the example described in chapter 6, this real-world example implementation clearly demonstrated the benefits of using Software-Defined Software techniques to improve software development quality and ease the processes of maintaining the application and adding new features. We plan to continue working with the SafeBanker product to adopt Software-Defined Software techniques across additional parts of the application.

Listing 7.1 Specification for SafeBanker™ "Live" components

```
# Specification for Live Components
## Overview
The "Live" server handles all real-time communication features of
SafeBanker and related products, including timers/checklists and
media calls (audio/video/chat).

## Scenarios

1. Joe is a bank branch manager. He needs to start the opening procedure,
   While the procedure is running, he needs to be able to update the
   status of the task, such as checking off checklist items, adding extra
   time, and ending the task. If there is an emergency, he needs to be
   able to start a call with the monitoring center.

## Non Goals
1. This application does not handle user authentication directly.
   Instead, it passes the user's credentials to the non-real-time part
   of the application, then temporarily caches the results. This ensures
   that the same authentication and authorization procedures will always
   be followed, and that access control methods for the different parts
   of the application will not diverge.
2. This application does not handle out-of-band notifications, such as
   email and mobile push notifications. When such notifications are
   required, a message is dispatched to the non-real-time part of the
   application which processes the notifications in a queue.
3. This application does not have a graphical user interface, only an
   RPC API.

## Example Endpoints
### Start performing a task
The user's device sends a request which includes the user authentication
information, location ID, user's position (lat/lng), and which task the
user would like to perform. The system must check the rules for whether
this task can be performed right now as requested. If yes, the server
must save the information in the database and provide the user's device
with a "task performance ID" and a timer expiration timestamp (UTC). If
no, the server must provide an error message, including human-readable
and machine-parseable error messages.
```

Chapter 8: Analysis Tools

In chapter 5.1, we discussed how Software-Defined Software requires us to define *decision functions* that will determine when a model is executed. In chapter 5.9, we discussed how Software-Defined Software requires us to explicitly define the data structures that each component will use for reading and writing information. We can use both of these design elements to analyze the data flow through the components and find potential problems before they lead to application failures or data loss.

8.1 Program Correctness Using Component Graphs

We can use a directed graph to ensure that component execution does not cause an infinite loop. We begin with adding all components (other than the central state storage) as nodes of a graph. On each node, we list the data elements that are used by the decision function that triggers that component to execute. For each node, we take the data elements modified by the **write** operations of that node and use them to draw directed edges to each component which uses that data in its decision function. We label each edge with the data elements that are written by the source node and acted upon by the target node.

8.1.1 Loop Detection for Individual Component Complexity

The first analysis we apply to the graph is simple *single-node loop detection*, which finds any nodes with edges that point directly back to the same node. If the application state is complex, devel-

opers may accidentally design a component that acts on data that it modifies itself, either in the initial application design or when features are added later. We make a list of all nodes and visit each one individually, checking all edges to see if the target is the same node. There are two types of procedures for handling such loops if they are found:

1. **Add label detail** - A component may only handle certain values of the element that caused the loop. For example, the "Warehouse" component of the `Nile` application described in chapter 6 may **watch** a variable named `order->status` but also update that same variable. In this case, the graph can be repaired by adding the important data *values* to the edge labels, which makes the labels more specific and removes the loop. For the `Nile` Warehouse component, the graph will be updated to say that the decision function uses the data element `order->status:ready_to_ship` while the component writes data elements `order->status:shipped`, `order->status:backordered`, etc. Any other component that uses the `order->status` element *without a specific value* will be a target of edges from all of the `order->status` writers, but any component that receives only specific values will only be the target of edges that write those specific values.
2. **Rebalance component logic** - Loops that cannot be resolved by adjusting the labeling are an indication that the component with the loop is too complex. The system designers should see this as a warning that the component is doing more than one job, which defeats the purpose of splitting the application into components. If the overall system design is sound and only a few components need to be changed, there are a few ways the jobs of the components can be "balanced" to make the components fit together without loops. One option is to implement hierarchical decomposition of the state space (as described in chap-

ter 5.8), and break the component into subcomponents. Again using the `Nile` Warehouse component as an example, this component may perform both checking stock levels in different warehouse locations as well as directing warehouse employees to pack and ship the order, which means the component is both writing and reading stock availability information. This can be resolved by dividing the component into subcomponents, one handling stock management and the other handling picking and packaging, so the shipping process will be fully handled at this lower level and the combined output of all subcomponents will be communicated back up to the central state. Another option is to split a single component into two (or more) components at the same level. For example, when the `Nile` developers originally implement order fulfillment, they assume that orders will only contain physical items that need to be shipped. At some point later, they add digital orders, such as e-books and music downloads. Because these products do not have an inventory count, they decide that the Warehouse component will immediately charge the credit card and make the files available for the user to download, which changes the criteria for triggering execution of the Warehouse component. In this case, they can resolve the issue by splitting the Warehouse component into separate "Physical Warehouse" and "Digital Warehouse" components with different internal logic. Each of those components will operate independently to fulfill their respective parts of the order.

8.1.2 Cycle Detection for Decision Function Correctness

With simple loops out of the way, we can move on to more advanced cycle detection. We first find all *strongly connected components* (SCC) of the graph. This is more difficult than a simple DFS-

based search (using e.g. Tarjan [90] or Johnson [42]) because those algorithms do not account for edge labels. To work around this, we first search the graph to make a list of all labels. We then apply an existing SCC-finding algorithm once for each label. The algorithm will only follow edges that are labeled with the chosen label *or any label that is less precise than the chosen label.* (For example, from the previous section, a search using the label `order->status:shipped` will also follow edges labeled `order->status`.) We note that although Depth First Search algorithms are normally difficult to parallelize, running the same algorithm multiple times over the same graph with different chosen labels each time is trivially parallelizable. We expect that the graphs of even the largest applications will only contain tens or hundreds of nodes, meaning that performance of each individual algorithm run should not be a problem. Any cycles that are found indicate the potential for either a deadlock [77:301] or an infinite loop within the components, suggesting that decision functions need to be adjusted. As above, this may involve making the function operate on more specific data, involve adding or removing components, or changing the hierarchical level at which some components operate.

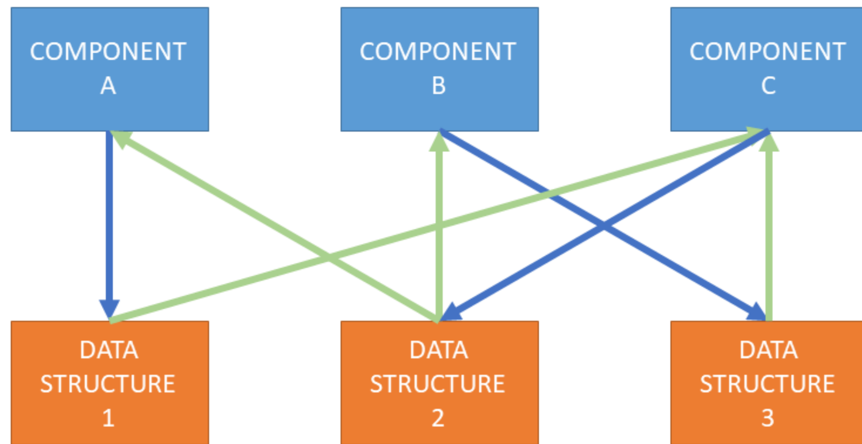
8.2 Program Correctness Using Data Structure Graphs

An alternate way to model the application is to use the *data structures* as the graph nodes instead of the components. With this method, we model each **decision function** as a node, and each set of **write permissions** as a node. We draw directional edges (unlabeled this time) from every **write** node to the **decision** node that makes use of the same information. This provides a view of which information changes cause other information changes.

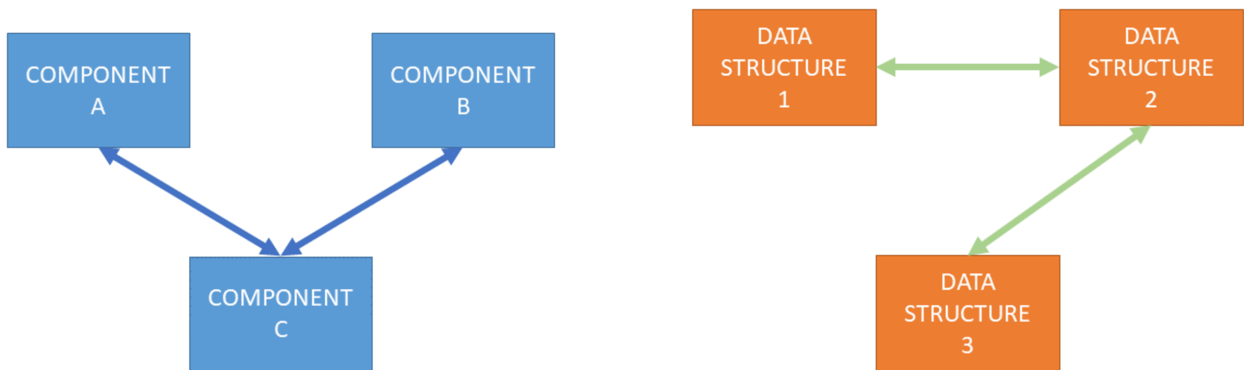
8.3 Program Correctness Using a Hybrid Graph

We can also describe the application using a graph that combines both types of graph previously discussed. First, we add all application components as one type of node. Then we add all data structures as another type of node. Edges are drawn from a component node pointing to an information (data structure) node if the component is able to write information to that node, and edges are drawn from an information node to a component node if the component's decision function is based on the state of that information node (fig. 8.1a). By removing the information nodes and collapsing the edges where they had been, a graph showing only which components are dependent on each other is formed (fig. 8.1b). Likewise, by removing the component nodes and collapsing the appropriate edges, a graph is formed showing which data structures can cause changes to each other data structure (fig. 8.1c). Either of these simplified graphs can be searched for labeled cycles and then cross-referenced with the full graph to show how the cycle occurs.

Figure 8.1 Graphs illustrating how component dependencies and information flow can be determined.



(a) Graph of components connected to data structures



(b) Graph of components only, showing which components can cause other components to be triggered.

(c) Graph of data structures only, showing how changes to one area can cause changes to another.

Chapter 9: Conclusion

In this dissertation, we have presented the complexity of large software systems, and specific problem areas that occur in building and maintaining these systems. We have also presented several existing methodologies for handling this complexity, using Service Buses, Service-Oriented Architecture, and Microservice technologies. We have found that practice of these methodologies still requires a lot of human intervention for management and deployment of large systems. These technologies also require a lot of implicit knowledge that exists only in the minds of the software developers and maintainers.

We introduced the concept and application techniques of Software-Defined Software, which we believe offers a unique and powerful approach to handle the problem of managing large, component-based software systems. This can include traditional software applications broken up into multiple components for scaling and redundancy purposes as well as distributed systems in which a number of data streams from many different components are received, filtered, registered, combed, stored, and distributed. Incoming streams may carry with them tags that specify the essential description of the stream, and these tags may be taken into account in creating execution paths through the system. System monitoring may be done with a new level of insight never before gained with such ease, and the simple connection mechanism from components to the central store allows for simple and easy development and testing.

We have demonstrated that Software-Defined Software techniques can be applied in practice in production-grade software. In the process, we have also shown how we can take information

that was previously implicitly included in the code and expose it in ways that make the software development and management processes clearer and easier to understand. In the open-source software world, Linus' Law states that "given enough eyeballs, all bugs are shallow" [69]. With Software-Defined Software, we have provided a framework that makes bugs shallower by simplifying the software, thus lowering the number of eyeballs required.

The future for Software-Defined Software is bright. We expect to build on our successful implementation of the technique and expand to additional modules of the existing product, as well as implementing new products and new features. More broadly, we would like to investigate application of the central state management concept at a lower level, such as in the Operating System or even directly in hardware. Such expansion would bring improvements, both for development and runtime of applications that use these techniques. Providing active state and component monitoring as a first-class feature will allow for significant benefits in application security, performance, troubleshooting, and management.

We began with the premise that software runs just about everything in the modern world, and we expect that nothing short of a world-wide cataclysmic event will change that. Future software systems will only become more complex, with integrations ever deeper into society and new developments in all manner of scientific fields. We provide Software-Defined Software, not only as a tool for addressing today's software development challenges, but as a foundation for enhancements that can answer the call of tomorrow's software developers as well.

> *If we think seriously about these problems, we find that we cannot work with procedures alone, since they are sequential. We need to define the problem instead of the procedures. ... Clearly, we must break away from the sequential and not limit the computers. We must state definitions and provide for priorities and descriptions of data. We must state relationships, not procedures.*
> – Rear Admiral Grace Murray Hopper [56:273]

Bibliography

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile Software Development Methods: Review and Analysis. (2017). DOI:https://doi.org/10.48550/ARXIV.1709.08439

[2] V. S. Alagar and K. Periyasamy. 2011. *Specification of Software Systems*. Springer London, London. DOI:https://doi.org/10.1007/978-0-85729-277-3

[3] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381. DOI:https://doi.org/10.1007/978-3-540-78800-3_28

[4] Jonathan Baron, Jane Beattie, and John C Hershey. 1988. Heuristics and biases in diagnostic reasoning. *Organizational Behavior and Human Decision Processes* 42, 1 (August 1988), 88–110. DOI:https://doi.org/10.1016/0749-5978(88)90021-0

[5] Victor R. Basili. 1979. Quantitative Software Complexity Models: A Panel Summary. In *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*, IEEE, Kiamesha Lake, New York, 243. Retrieved from https://www.cs.umd.edu/~basili/publications/proceedings/P14.pdf

[6] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. 2001. Manifesto for Agile Software Development. Retrieved February 19, 2023 from https://agilemanifesto.org/

[7] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE '07)*, IEEE, Minneapolis, MN, USA, 85–103. DOI:https://doi.org/10.1109/FOSE.2007.25

[8] Gerrit A. Blaauw and Frederick P. Brooks. 1997. *Computer architecture: Concepts and evolution*. Addison-Wesley, Reading, Mass.

[9] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software -*, ACM Press, Los Angeles, California, 234–245. DOI:https://doi.org/10.1145/800027.808445

[10] Frederick P. Brooks. 1995. *The mythical man-month: Essays on software engineering* (Anniversary ed.). Addison-Wesley Pub. Co, Reading, Mass.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, San Diego, CA, 209–224. Retrieved from https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf

[12] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. 2010. An Industrial Survey on Contemporary Aspects of Software Testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, Paris, France, 393–401. DOI:https://doi.org/10.1109/ICST.2010.52

[13] K Mani Chandy, Brian Emre Aydemir, Elliott Michael Karpilovsky, and Daniel M Zimmerman. 2003. Event-driven architectures for distributed crisis management. *Computer Science* 256, (2003), 80. Retrieved from https://www.academia.edu/download/61879376/EDA-DCM-03.pdf

[14] Mihaly Csikszentmihalyi. 1990. *Flow: The psychology of optimal experience*. Harper and Row, New York.

[15] Joseph Dadzie. 2005. Understanding software patching. *Queue* Vol 3, (March 2005), 24. DOI:https://doi.org/10.1145/1053331.1053343

[16] Matt Davy, Guru Parulkar, Johan van Reijendam, Dan Schmiedt, Russ Clark, Chris Tengi, Ivan Seskar, Patrick Christian, Ivan Cote, and George China. 2012. A Case for Expanding OpenFlow/SDN Deployments On University Campuses. Retrieved from https://spaces.at.internet2.edu/display/sdn/Whitepaper

[17] F. DeRemer and H. H. Kron. 1976. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* SE-2, 2 (June 1976), 80–86. DOI:https://doi.org/10.1109/TSE.1976.233534

[18] Jake Douglas. 2012. Deploying at GitHub. Retrieved from https://blog.github.com/2012-08-29-deploying-at-github/

[19] H. Erdogmus and J. Vandergraaf. 1999. Quantitative approaches for assessing the value of COTS-centric development. In *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*, IEEE Comput. Soc, Boca Raton, FL, USA, 279–290. DOI:https://doi.org/10.1109/METRIC.1999.809749

[20] Ted Faison. 2006. *Event-Based Programming: Taking Events to the Limit* (1st ed.). Apress.

[21] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2013. The Road to SDN: An intellectual history of programmable networks. *Queue* Vol 11, (December 2013), 20–40. DOI:https://doi.org/10.1145/2559899.2560327

[22] Federal Communications Commission. 2019. Operation of Wireless Microphones. Retrieved from https://www.fcc.gov/consumers/guides/operation-wireless-microphones

[23] Stuart Feldman. 2004. A Conversation with Alan Kay: Big talk with the creator of smalltalk - and much more. *Queue* Vol 2, (December 2004), 20–30. DOI:https://doi.org/10.1145/1039511.1039523

[24] Martin Fowler. 2015. Microservice Premium. Retrieved from https://martinfowler.com/bliki/MicroservicePremium.html

[25] Sean Gallagher. 2014. Not dead yet: Dutch, British governments pay to keep Windows XP alive. *Ars Technica*. Retrieved December 20, 2018 from https://arstechnica.com/information-technology/2014/04/not-dead-yet-dutch-british-governments-pay-to-keep-windows-xp-alive/

[26] David Garlan and Mary Shaw. 1993. An Introduction to Software Architecture. In *Series on Software Engineering and Knowledge Engineering*. WORLD SCIENTIFIC, 1–39. DOI:https://doi.org/10.1142/9789812798039_0001

[27] Victor Gaultney. 2023. *Gentium Plus*. SIL International. Retrieved from https://software.sil.org/gentium/

[28] Boby George and Laurie Williams. 2004. A structured experiment of test-driven development. *Information and Software Technology* 46, 5 (April 2004), 337–342. DOI:https://doi.org/10.1016/j.infsof.2003.09.011

[29] Pete Goodliffe. 2009. A Tale of Two Systems: A Modern-Day Software Fable. In *Beautiful architecture* (1st ed), Diomidis Spinellis and Georgios Gousios (eds.). O'Reilly, Sebastopol, Calif, 25–42.

[30] Google, Inc. *BoringSSL*. Google, Inc. Retrieved from https://boringssl.googlesource.com/boringssl/

[31] Google, Inc. *Protocol Buffers*. Retrieved from https://protobuf.dev/overview/

[32] Terri Griffith. 2014. Help Your Employees Find Flow. *Harvard Business Review*. Retrieved from https://hbr.org/2014/04/help-your-employees-find-flow

[33] John Gruber and Aaron Swartz. *Markdown*. Retrieved from https://daringfireball.net/projects/markdown/

[34] Barbara Hayes-Roth. 1985. A blackboard architecture for control. *Artificial Intelligence* 26, 3 (July 1985), 251–321. DOI:https://doi.org/10.1016/0004-3702(85)90063-3

[35] George T. Heineman and William T. Councill (Eds.). 2001. *Component-based software engineering: Putting the pieces together*. Addison-Wesley, Boston.

[36] Kelsey Hightower. 2020. Monoliths are the future. Retrieved from https://changelog.com/posts/monoliths-are-the-future

[37] C. A. R. Hoare. 1981. The emperor's old clothes. *Commun. ACM* 24, 2 (February 1981), 75–83. DOI:https://doi.org/10.1145/358549.358561

[38] C. A. R. Hoare. 1996. How did software get so reliable without proof? In *FME'96: Industrial Benefit and Advances in Formal Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

[39] William E. Howden. 1976. Experiments with a symbolic evaluation system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition on - AFIPS '76*, ACM Press, New York, New York, 899. DOI:https://doi.org/10.1145/1499799.1499922

[40] Jerome Hugues and Joseph D. Yankel. *Model-Based Systems Engineering Meets DevSecOps*. Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=877043

[41] International Organization for Standardization. 2018. *Information technology - Service management - Part 1: Service management system requirements.* International Organization for Standardization. Retrieved from https://www.iso.org/standard/70636.html

[42] Donald B. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 4, 1 (March 1975), 77–84. DOI:https://doi.org/10.1137/0204007

[43] Magne Jorgensen. 2019. Relationships Between Project Size, Agile Practices, and Successful Software Development: Results and Analysis. *IEEE Softw.* 36, 2 (March 2019), 39–43. DOI:https://doi.org/10.1109/MS.2018.2884863

[44] Poul-Henning Kamp. 1999. A bike shed (any colour will do) on greener grass... Retrieved from https://bikeshed.org/

[45] Brian W. Kernighan and P. J. Plauger. 1978. *The Elements of Programming Style* (2d ed ed.). McGraw-Hill, New York.

[46] Chris King. 2014. Like tears in rain: The death of Windows XP. *ACM SIGUCCS plugged in* 2, 4 (April 2014), 9–9. DOI:https://doi.org/10.1145/2661747.2661755

[47] James C. King. 1976. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (July 1976), 385–394. DOI:https://doi.org/10.1145/360248.360252

[48] Matt Klein. 2017. Envoy: 7 months later. Retrieved from https://eng.lyft.com/envoy-7-months-later-41986c2fd443

[49] Pavneet Singh Kochhar, Tegawende F. Bissyande, David Lo, and Lingxiao Jiang. 2013. Adoption of Software Testing in Open Source Projects–A Preliminary Study on 50,000 Projects. In *2013 17th European Conference on Software Maintenance and Reengineering*, IEEE, Genova, 353–356. DOI:https://doi.org/10.1109/CSMR.2013.48

[50] Glenn E. Krasner and Stephen T. Pope. 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP* 1, 3 (August 1988). Retrieved from https://www.lri.fr/~mbl/ENS/FONDIHM/2013/papers/Krasner-JOOP88.pdf

[51] Leslie Lamport. 1987. Distribution. Retrieved from https://lamport.azurewebsites.net/pubs/distributed-system.txt

[52] Albert L. Lederer and Jayesh Prasad. 1995. Causes of inaccurate software development cost estimates. *Journal of Systems and Software* 31, 2 (November 1995), 125–134. DOI:https://doi.org/10.1016/0164-1212(94)00092-2

[53] Laura Marie Leventhal, Barbee M. Teasley, Diane S. Rohlman, and Keith Instone. 1993. Positive test bias in software testing among professionals: A review. In *Human-Computer Interaction*, Leonard J. Bass, Juri Gornostaev and Claus Unger (eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 210–218. DOI:https://doi.org/10.1007/3-540-57433-6_50

[54] D. C. Luckham and J. Vera. Sept./1995. An event-based architecture definition language. *IEEE Transactions on Software Engineering* 21, 9 (Sept./1995), 717–734. DOI:https://doi.org/10.1109/32.464548

[55] John MacFarlane, Albert Krewinkel, and Jesse Rosenthal. *Pandoc*. Retrieved from https://github.com/jgm/pandoc

[56] Massachusetts Institute of Technology. School of Industrial Managemen. 1962. *Management and the Computer of the Future*. M.I.T. Press and Wiley, New York.

[57] E. M. Maximilien and L. Williams. 2003. Assessing test-driven development at IBM. In *25th International Conference on Software Engineering, 2003. Proceedings.*, IEEE, Portland, OR, USA, 564–569. DOI:https://doi.org/10.1109/ICSE.2003.1201238

[58] Doug McIlroy, E. N. Pinson, and B. A. Tague. 1978. UNIX Time-Sharing System. *The Bell System Technical Journal* 57, 6 (July 1978), 1899–1904. Retrieved from https://archive.org/details/bstj57-6-1899/page/n3/mode/2up

[59] Gail C. Murphy. 2014. Getting to Flow in Software Development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! '14*, ACM Press, Portland, Oregon, USA, 269–281. DOI:https://doi.org/10.1145/2661136.2661158

[60] Glenford J. Myers, Tom Badgett, and Corey Sandler. 2012. Introduction. In *The art of software testing: Now covers testing for usability, smartphone apps, and agile development environments* (3. ed). Wiley, Hoboken, NJ, ix.

[61] Sam Newman. 2015. *Building microservices: Designing fine-grained systems* (First Edition ed.). O'Reilly Media, Beijing Sebastopol, CA.

[62] Sam Newman. 2015. Microservices For Greenfield? Retrieved from https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/

[63] Sam Newman. 2019. *Monolith to microservices evolutionary patterns to transform your monolith* (First edition ed.). O'Reilly Media, Inc., Beijing [China].

[64] Gary Nilson, Kent Wills, Jeffrey Stuckman, and James Purtilo. 2013. BugBox: A vulnerability corpus for PHP web applications. In *6th workshop on cyber security experimentation and test (CSET 13)*, USENIX Association, Washington, D.C. Retrieved from https://www.usenix.org/conference/cset13/workshop-program/presentation/nilson

[65] Jon Oberheide, Evan Cooke, and Farnam Jahanian. 2009. If it ain't broke, don't fix it: Challenges and new directions for inferring the impact of software patches. In *Proceedings of the 12th conference on Hot topic in operating systems*, USENIX, Monte Verità, Switzerland, 17. Retrieved from http://vhosts.eecs.umich.edu/fjgroup/pubs/hotos09-patchadvisor.pdf

[66] OpenBSD Project Developers. 2014–2023. *LibreSSL*. Retrieved from https://www.libressl.org/

[67] Chris Parnin and Spencer Rugaber. 2011. Resumption strategies for interrupted programming tasks. *Software Qual J* 19, 1 (March 2011), 5–34. DOI:https://doi.org/10.1007/s11219-010-9104-9

[68] James M. Purtilo. 1994. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems* 16, 1 (January 1994), 151–174. DOI:https://doi.org/10.1145/174625.174629

[69] Eric S. Raymond. 1999. *The cathedral & the bazaar: Musings on Linux and open source by an accidental revolutionary* (1st ed ed.). O'Reilly, Beijing ; Cambridge, Mass.

[70] Rick. 2021. Don't even consider micro services unless you have a system that's too complex to manage as a monolith. Retrieved July 28, 2021 from https://sennalabs.com/en/blogs/don-t-even-consider-micro-services-unless-you-have-a-system-that-s-too-complex-to-manage-as-a-monolith

[71] H. D. Rombach. 1990. Design measurement: Some lessons learned. *IEEE Software* 7, 2 (March 1990), 17–25. DOI:https://doi.org/10.1109/52.50770

[72] M. N. O. Sadiku and C. M. Akujuobi. 2004. Software-defined radio: A brief overview. *IEEE Potentials* 23, 4 (October 2004), 14–15. DOI:https://doi.org/10.1109/MP.2004.1343223

[73] Ayesha Saeed, Wasi Haider Butt, Farwa Kazmi, and Madeha Arif. 2018. Survey of Software Development Effort Estimation Techniques. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ACM, Kuantan Malaysia, 82–86. DOI:https://doi.org/10.1145/3185089.3185140

[74] Jerome H. Saltzer. 1974. Protection and the control of information sharing in multics. *Commun. ACM* 17, 7 (July 1974), 388–402. DOI:https://doi.org/10.1145/361011.361067

[75] M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, and H. H. Ammar. 2001. Information theoretic metrics for software architectures. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, IEEE, Chicago, IL, USA, 151–157. DOI:https://doi.org/10.1109/CMPSAC.2001.960611

[76] Nataliya Shevchenko. 2020. An Introduction to Model-Based Systems Engineering (MBSE). Retrieved from https://insights.sei.cmu.edu/blog/introduction-model-based-systems-engineering-mbse/

[77] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2009. *Operating system concepts* (8th ed ed.). J. Wiley & Sons, Hoboken, NJ.

[78] Software Engineering Institute. 2016. Goal-Driven Measurement (IGDM) SEMA Course. Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=635664

[79] Software Engineering Institute. 2020. Designing Products and Processes Using Six Sigma (DPPSS) SEMA Course. Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=646819

[80] Software Engineering Institute. 2020. Improving Process Performance Using Six Sigma (IPPSS) SEMA Course. Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=646828

[81] Diomidis Spinellis and Georgios Gousios (Eds.). 2009. *Beautiful architecture* (1st ed ed.). O'Reilly, Sebastopol, Calif.

[82] Joel Spolsky. 2000. Things You Should Never Do, Part I. Retrieved from https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

[83] Joel Spolsky. 2004. The Joel Test: 12 Steps to Better Code. In *Joel on Software*. Apress, Berkeley, CA, 17–30. DOI:https://doi.org/10.1007/978-1-4302-0753-5_3

[84] Joel Spolsky. 2004. Painless Functional Specifications Part 1: Why Bother? In *Joel on Software*. Apress, Berkeley, CA, 45–51. DOI:https://doi.org/10.1007/978-1-4302-0753-5_5

[85] Joel Spolsky. 2004. Painless Functional Specifications Part 2: What's a Spec? In *Joel on Software*. Apress, Berkeley, CA, 53–64. DOI:https://doi.org/10.1007/978-1-4302-0753-5_6

[86] Joel Spolsky. 2004. Painless Functional Specifications Part 3: But ... How? In *Joel on Software*. Apress, Berkeley, CA, 65–68. DOI:https://doi.org/10.1007/978-1-4302-0753-5_7

[87] W. P. Stevens, G. J. Myers, and L. L. Constantine. 1974. Structured design. *IBM Systems Journal* 13, 2 (1974), 115–139. DOI:https://doi.org/10.1147/sj.132.0115

[88] András Szepesházi. 2011. How to explain a layperson why a developer should not be interrupted while neck-deep in coding? Retrieved from https://softwareengineering.stackexchange.com/q/46252/58952

[89] Andrew S. Tanenbaum and D. Wetherall. 2011. *Computer networks* (5th ed ed.). Pearson Prentice Hall, Boston.

[90] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (June 1972), 146–160. DOI:https://doi.org/10.1137/0201010

[91] Michele Tepper. 2002. Why software still stinks. *netWorker* 6, 3 (September 2002), 40. DOI:https://doi.org/10.1145/569207.569223

[92] The PostgreSQL Global Development Group. 2023. Explicit Locking: Advisory Locks. Retrieved from https://www.postgresql.org/docs/current/explicit-locking.html#ADVISORY-LOCKS

[93] Jonathan Turpie, Elnatan Reisner, Jeffrey S. Foster, and Michael Hicks. 2011. *MultiOtter: Multiprocess Symbolic Execution*. University of Maryland, College Park, MD. Retrieved from https://www.cs.umd.edu/~mwh/papers/multiotter.pdf

[94] United States Government. 1968. *Bank Protection Act*.

[95] US-CERT. 2014. *Microsoft ending support for Windows XP and Office 2003*. US-CERT. Retrieved July 20, 2014 from http://www.us-cert.gov/ncas/alerts/TA14-069A-0

[96] Kami Vaniea and Yasmeen Rashidi. 2016. Tales of Software Updates: The process of updating software. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems - CHI '16*, ACM Press, Santa Clara, California, USA, 3215–3226. DOI:https://doi.org/10.1145/2858036.2858303

[97] P. C. Wason. 1960. On the failure to eliminate hypotheses in a conceptual task. *Quarterly Journal of Experimental Psychology* 12, 3 (July 1960), 129–140. DOI:https://doi.org/10.1080/17470216008416717

[98] P. C. Wason. 1968. Reasoning about a rule. *Quarterly Journal of Experimental Psychology* 20, 3 (August 1968), 273–281. DOI:https://doi.org/10.1080/14640746808400161

[99] Peter Wilson and Lars Madsen. *Memoir*. Retrieved from https://www.ctan.org/tex-archive/macros/latex/contrib/memoir

[100] Jack J. Woehr. 1996. An Interview With Donald Knuth. *Dr. Dobb's Journal* Vol 21, (April 1996), 16–22. Retrieved March 9, 2023 from https://drdobbs.com/an-interview-with-donald-knuth/184409858