A Scale First approach for Web API development

Carlos Martín Flores González School of Computer Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica 2550–2254
Email: martin.flores@computer.org
Student ID: 2015183528

***Abstract*—Most applications using the Web as a delivery mechanism have had to embrace practices in order to constantly adapt their software to ensure their applications will be up and running most of the time. This is not an easy task, and it becomes harder when applications are not designed to accommodate this change upfront. Web Application Programming Interfaces (APIs) have become a popular solution to deliver data and content through the Web, and since they are available through the Internet, they are exposed to the same concerns on how to manage change, particularly the way the application can scale and manage new resources. Scale First is an attempt to describe an approach where scalability topics are placed first when implementing a Web API. It proposes a set of architectural, design and coding principles that allow addressing change more effectively. These principles can be used in conjunction with software development methodologies to model the system's strategy in terms of scalability and evolution.**

I. Introduction

In software development, the use of APIs is crucial to define protocols, operations and tools for building applications. Software developers access APIs as interfaces for code libraries to speed up development and/or take advantage of existing low level tasks. With the advent of the World Wide Web a new kind of API has emerged in order to provide services in applications through a set of Hypertext Transfer Protocol (HTTP) messages, the Web API. But, as with APIs, most of today's Web APIs have a big problem: once deployed, they can't be changed. In fact, there are big-name APIs that stay static for years at a time, as the industry changes around them, because changing them would be too difficult [16]. A Web API's ability to adapt and evolve to new requirements is crucial to ensure its success under critical and high-demanding scenarios, such as the ones experienced by E-Commerce and data intensive applications.

Although we know software can change, many Web APIs can't accommodate change easily because they were not designed and developed with scalability in mind from the ground up [16]. A scalable Web API is an application that has the ability not only to allocate new resources in order to support new demanding growth, but also to decrease resource consumption when demand is down. Scalability is one of the main concerns that organizations have to keep in mind when implementing a software project. Without it, the ability to compete and bring services to a broader audience, and even reputation, are compromised.

Software methodologies don't propose a specific set of actions to consider in order to address scalability concerns. Some agile methodologies such as Scrum and Extreme Programming (XP) encourage software projects to hit production or production-like environments as soon as possible (one of these approaches is the "Walking Skeleton", described by Allistair Cockburn [5] where a tiny implementation of the system is supposed to perform a small end-to-end function.) Their main goal is to quickly get feedback about the deployment process and minimize further risks. In this aspect, we can say that these methodologies are intended to model the domain of the application rather than application growth.

On the other hand, modern engineering practices such as Continuous Integration and Continuous Delivery provide us with a reference on how to manage applications, their changes in the codebase and their deployment. In both practices, automation emerges as one of the key aspects since they rely on software to run test → build → acceptance → deploy cycles — In Continuous Delivery, this workflow is known as a deployment pipeline — ; this could allow not only faster response times during deployment but also a better sense of quality [11]. In Continuous Delivery, the deployment processes of applications might consider scalability concerns because the goal is to model a workflow where applications are always in a delivery-ready state and, in order to achieve it, software developers have to find a way to provide mechanisms where applications are available to be shipped to one or n servers at any time (ideally.) An important thing to note here is that these practices are agnostic to the application they serve. This makes sense at first glance, but at the same time it could lead to scenarios where monolithic applications are trying to be deployed. The term monolith has been in use by the Unix community for some time. It appears in The *Art of Unix Programming* [15] to describe systems that get too big. Despite the success of monolithic applications, frustration increases especially when they are being deployed. Change cycles are tightly coupled, small changes require the entire monolith to be rebuilt and deployed. Over time it is often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource [10]. An application could be delivered fast, but its internals do not allow for a better way to scale it unless you try to scale the whole application at the same time. This approach could be expensive in terms of money as well as resource consumption.

From the deployment standpoint, according to recent trends, organizations are planning for hybrid cloud computing solutions, meaning that they are increasingly relying their applica-

tion deployment on more than one cloud computing provider [17], making it even more important to have applications with cloud-friendly architectures. Here again, the eventual adoption of a development strategy that is aware of these kinds of deployment considerations could make application distribution easy and prevent further headaches.

II. Web APIs

Web API is a term that refers to functions/operations that can be programmatically invoked using Web protocols. The term is based on the concept of Service Oriented Architecture (SOA) which proposed an attempt to change how software functionality is exposed through well-defined interfaces that were formally registered and could be discovered on the fly.

As the SOA notions matured, especially in their web services style and XML format of data, SOA evolved into more Web-friendly technologies such as Representational State Transfer (REST) and JSON data format that greatly simplified the reusability of an enterprise's capabilities. We use a common term, Web APIs, to refer to them. The reason Web APIs are even more important today is that they allow organizations to make their business functionalities easily accessible to other entities (patterns, customers) and thus easily create new value-added services. Furthermore, users have more access than ever to a computing device in the context of their daily lives, i.e., mobiles. [21]

III. Introducing Scale First

Due the lack of a software methodology/approach that is able to embrace scalability concerns as a key role of the software development process of Web APIs from their initial designs, we introduce *Scale First*, an approach where scalability becomes a feature of the application itself. By sticking to this approach, Web APIs will be able to evolve more easily because growth aspects are attacked in the initial stages of the development process, making organizations and individuals even more aware of the entire Web API ecosystem.

In order to have *scale-ready* applications, *Scale First* proposes a set of principles for clean, well-structured and testable applications that will be able to evolve according their needs before a single line of code is written; here is where this approach differs from traditional methodologies, since concepts such as software design and deployment are tightly coupled.

A. Principles in Scale First

- Resizable deployment environments
- Technology agnostic design
- Modular, scalable application design
  - Decompose application domain
  - Split conceptual model into separate models for update and display
- Testability and maintainability
- Automated deployment process
- Define a scalability policy

B. Resizable deployment environments

Web APIs on *Scale First* are intended to provide an elastic architecture. This requires a suitable deployment environment that provides support for this elastic behavior, since applications and their resources can change, so the goal here is to be ready to handle this change as fast as possible. This will increase Web APIs' availability and resilience. One popular alternative for providing a flexible environment for Web APIs is the adoption of cloud[1] technologies. Software and infrastructure are increasingly consumed from the cloud. This is more flexible and much cheaper than deploying your own infrastructure, especially for smaller organizations.

Providers use three well-known models: IaaS (infrastructure as a service), PaaS (platform as a service) and SaaS (software as a service). As an example, in the case of IaaS, the model offers precise scalability. The cloud can outperform the physical hardware's classic scale-up or scale-out strategies. To gain as much from these features, applications have to be architected with as much as decoupling as possible, using SOA and queries between services [19].

C. Technology agnostic design

The aim of technology agnostic design(TAD) is to separate design and architecture from the technology employed and the specific implementation. This separation decreases both cost and risk while increasing scalability and availability of the Web API. With TAD, both organizations and individuals are forced to create disciplines around scales that are not dependent on any single provider or service. This discipline allows you to scale in multiple dimensions through multiple potential partners, the result of which is a more predictable scalable system independent of any single solution provider. One popular option

A common misperception is that by implementing a certain solution –for instance a database technology–, you are reliant on that solution. Just because the project makes use of a trendy new database technology does not mean that the Web API is dependent upon it alone for scale. TAD also impacts availability. The most obvious way is how it supports the ability to switch technology providers when one provider has significantly greater quality availability over another. [1]

1) TAD considerations for Web APIs: [1] proposes three steps for TAD designs. These steps are suitable for Web API projects as well, and can act as an initial help guide.

1) In the design itself, think in terms of boxes and wire diagrams rather than prose. Leave the detail of the boxes to the next step.
2) In defining boxes and flows, use generic terms. Application server, Web server, Relational Database Management Systems (RDBMS), and so on. Don't use vendor names.

[1]The most-used definition of cloud computing belongs to the US National Institute of Standards and Technology (NIST): *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [13]"*

3) Describe requirements in industry standards. Stay away from proprietary terms specific to a vendor.

If the design is being pulled toward a vendor in a description or design statement, try to "loose" that statement in order to make it agnostic.

D. Modular, scalable application design

Here the goal is to come up with an architecture that allows the Web API to become an autonomous component that communicates with a mechanism such as web service requests, or remote procedure calls. That is, make the Web API a service or a *microservice*, an architectural style to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [10]. From a scalability standpoint, the main reasons for using services as components are:

- Technology Heterogeneity: Web APIs can be developed using the best suitable tool or technology stack depending on their needs.
- Only those services that need scaling will scale.
- Services are independently deployable: a change that is made to a single service is deployed independently without impacting other systems. This allows for a faster deployment of the code. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve.
- Resilience: working as an isolated service; if a Web API fails, that failure will not be propagated to other systems.

Notice the above are not the only characteristics of a service-oriented architecture. More of this is described in [14] and [10].

1) Decompose application domain: The domain context of the Web API must be clearly defined. The aim here is not to motivate the creation of very specific applications doing just one thing. This could lead to management issues, and since we are addressing the Web API development from a scalability standpoint, we are more interested in the creation of applications that represent a functional unit, so that logic and features inside can be managed and shared easily.

The work on [8] provides a guide on how to create systems that model real-world domains. It introduces the concept of *bounded context*: "a specific responsibility enforced by explicit boundaries". The idea is that any given domain consists of multiple bounded contexts, and residing within each are things that do not need to be communicated to the outside, as well as things that are shared externally with other bounded contexts. Each bounded context has an explicit interface, where it decides what models to share with other contexts. To get information from a bounded context, or if you want to make functionality requests within a bounded context, a communication with its explicit boundary using models is performed.

When developing a Web API, the domain can be separated into bounded contexts, meaning that each context has the potential to become a separate Web API. It doesn't matter that the bounded contexts are not meant to be separate APIs in the initial stages of the Web API development process, because they can be extracted and implemented as separate APIs depending on the application's needs. This is why this separation process is crucial. With bounded contexts in place, future scalability efforts can be performed faster. This is therefore a very important activity to run when designing and developing applications that are meant to be subject of change. In the fictional application design on figure 1, although a Web API can be developed using the entire domain as a reference, both Sales Context and Support Context are potential separate Web APIs.

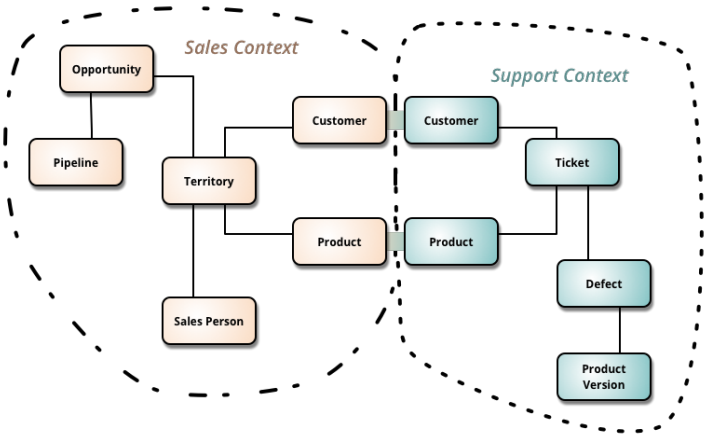

Figure 1: A fictional application domain with two bounded contexts: Sales Context and Support Context. [9]

2) Split conceptual model into separate models for update and display: Depending on the domain, some applications can be more intensive in reading (query) than in writing data and vice versa. Even so, both responsibilities have to be identified and separated into models for update and display. Ideally, each model will hold a set of autonomous components(such as databases) so each model is not depending on the other. This separation allows applications to scale based on the demand of each of its responsibilities; that way, if the most intensive part in a Web API is related to the query of data sources for display purposes, then this portion of the Web API can be isolated and segregated in another Web API whose only purpose will be to serve queries. Since queries are now being performed off of a separate data store from the master database, and assuming that the data that's being served is not 100% up to date, one can easily add more instances of these stores without having to worry that they don't contain the same exact data. This provides cheap horizontal scaling for queries. Also, since you are not doing nearly as much data transformation, the latency per query goes down as well. Simple code is fast code [6].

An often implemented solution to this is the three-tiered application, which has a separate data, logic and presentation tier. Within this solution, the database in the data tier is often seen as one CRUD (Create, Read, Update and Delete data) data store in which all commands and queries are performed on the same database. This can lead to locking, performance and scalability problems, especially with larger commands or queries, since all things have to be taken care of sequentially.

Distributing parts of the system in combination with selective locking of data provides a partial solution, but leads to a high probability of data inconsistency. An option is to split parts of the system that have an emphasis on consistency from parts that should have an emphasis on availability or partition tolerance. The Command Query Responsibility Segregation (CQRS) is a pattern in which all logic in a software product is separated based on whether it changes the application state (commands) or only queries it (queries). This means that executing commands is done by components different from the one responsible for executing the querying tasks, all of which can be done distributed and in parallel. Besides helping to solve the scalability problem of multi-tiered software products by enabling architects to distribute tasks of the system among an unlimited number of systems, CQRS also helps to implement a higher level of variability in online software products. The high level of variability is caused by the fact that the main pattern keeps commands strictly separated from queries and has a large collection of sub patterns using the distributed nature of the pattern to enable, among others, all sorts of tenant-dependent configurations, work flows, and business rules [12].

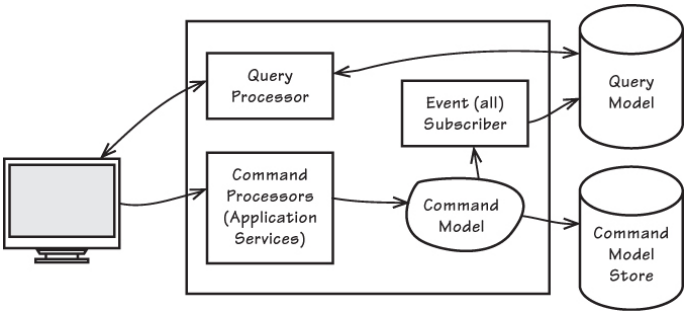

Figure 2: With CQRS, commands from clients travel one way to the command model. Queries are run against a separate data source optimized for presentation and delivered as user interface or reports [22].

With a CQRS approach like the one in figure 2, one can lead a Web API to be separated in two subapplications, each application can grow independently so resources can be assigned depending on its demand.

E. Testability and maintainability

A test strategy for a Web API that is intended to evolve quickly must provide the following:

1) Tests must run fast
2) Tests must cover as much of the application as possible
3) Tests must be automated
4) Continuous Integration

In the above list, items 1, 2 and 3 are intended to describe how a test should behave while item 4 is intended to provide a context for how tests should be executed. Although many kinds of testing exist [11], testing for fast and reliable scaling must have the automated component. For Web APIs, unit, integration and acceptance tests are the preferable set of tests to include.

The idea behind item 4, continuous integration, is that if regular integration of your codebase is good, why not do it all the time? In the context of integration, "all the time" means every single time somebody adds a change into the system [3]. The application of this concept in the Web API by the usage of continuous integration software such as Jenkins, Bamboo, Cruise Control, SnapCI and TravisCI, in the development process will allow delivering software much faster, and with fewer bugs (Or with no bugs at all).

To increase the Web API autonomy and increase overall availability, it is also necessary to identify and repair problems, and then notify the appropriate system operator of the service's current status. A Web API must provide support to actively monitor its internal state, act on potential trouble, try to heal itself, and continuously publish its status. For this, [18] proposes the implementation of a service watchdog[2], a pattern where a service actively monitors its internal state, acts on potential trouble, tries to heal itself, and continuously publishes its status. Web APIs on *Scale First* must contain a component in charge of monitoring the API's state. This component publishes the API's state periodically, and also when something meaningful occurs.

F. Automated deployment process

Scale First relies on the concepts and principles behind Continuous Delivery [11] in order to define processes that will allow having a Web API application in a delivery-ready state. To achieve this, continuous delivery introduces the concept of *deployment pipeline* which is an automated manifestation of the process for getting software from version control to the users. Every change in the software goes through a complex process on its way to being released. That process involves the building of the software, followed by the progress of these builds through multiple stages of testing and deployment. The deployment pipeline models this process, and its incarnation in a continuous integration and release management tool is what allows seeing and controlling the progress of each change as it moves from version control through various sets of tests and deployment to release to users.

An automated deployment process for a Web API in *Scale First* must include tasks in which new resources are able to be both, allocated and deallocated according to application demands. This ability to make applications and environments adjust to new conditions is what we refer to as the *scale-ready* attribute in this Web API development approach.

G. Define a scalability policy

As with every new paradigm, it's important to design Web APIs taking into account best practices and policies that will act as a body of knowledge for the application. The definition of policies and practices on *Scale First* must cover:

[2] *Watchdog* is a term borrowed from the embedded systems world. A watchdog is a hardware device that counts down to 0, at which point it takes action, such as resetting the device. To prevent this reset, the application has to "kick the dog" before the timer runs out. If the application doesn't reset the counter, it could mean that the application has stopped responding. A reset would fix that.

- Design for failure
- High availability
- Performance
- Security
- Monitoring

1) Design for failure: High scalability has limitations. Redundancy and fault tolerance are primary design goals. Applications have to be prepared for reboots and launches.

2) High availability: IT resource disruption has a huge negative impact on any business, this leads to design with outages and high availability in mind.

3) Performance: It is necessary to consider technology limitations regarding performance. A usage burst can affect available resources, notably compute units and disks' input/output operations. Bottlenecks might arise as a result of latency issues.

4) Security: Enforce well-established security practices around the Web API: firewalls, minimal server services to reduce attack vectors, up-to-date operating systems, key-based authentication, and so on.

5) Monitoring: The ease of deployment of new resources can make the number of servers grow exponentially. This raises new issues, and monitoring tools are vital to system management.

H. Web API Checklist on Scale First

	Implemented?	
Feature	**Yes**	**No**
Infrastructure		
Infrastructure is flexible?		
Application Domain		
Domain is delimited		
Display (Quey) model		
Update (Command) model		
Test Strategy		
Unit Tests		
Integration Tests		
Acceptance Tests		
Continuous Integration software		
Automated deployment		
Deployment pipeline is defined?		
Policies		
Failure		
High Availability		
Performance		
Security		
Monitoring		

Table I: Web API Checklist on Scale First

IV. Experimental Setup

The exploratory study of the impact of the *Scale First* approach for Web API development could be composed of four parts. In the first part we can collect a set of Web API projects in which one or more of the principles of the Scale First approach have not been adopted. In the second part, the development of a non-trivial Web API that follows the *Scale First* approach is required. This allow us to identify the potential of *Scale First*. In the third part, we need to create and perform scenarios where scalability is required on both types of applications: those that follow *Scale First* and those that don't. Lastly, we measure and interpret the impact of Scale First by analyzing the response time involved in performing the scalability task and the resources assigned to it, such as money, hardware and people.

A. First part: Web APIs don't follow Scale First principles

This experiment involves the gathering of Web API projects where some of the *Scale First* principles are not adopted. For starters, classic monolithical ("three tier") applications could be the target. These applications could be architected taking some, but not all, of the exposed principles into account.

B. Second part: Development of a non-trivial Web API following Scale First principles

A Web API project following the *Scale First* principles must be developed. Ideally this should be a project that will be used in a real-life scenario. It could mimic the size and complexity of one of the applications gathered in the first part. Another interesting approach would be to perform a migration project of one of the applications in the first part to the *Scale First* model.

C. Third part: Evaluation

In this part of the experiment we need to create and perform tests in which both types of applications need to react to change in some way. Some initial scenarios could involve:

- Scale a new instance of the Web API to a new server.
- Scale a new instance of the Web API into a new cloud provider.
- Once a new instance of the Web API is deployed, terminate it.
- Extract a portion of an existing Web API to an independent Web API.
- Add a new resource to the application (database, cache, third party solution)

The above list proposes some basic scenarios where scalability concerns are tested. Depending on the nature of the application and organization needs, these scenarios could be more and could include more complex specifications.

D. Fourth part: analysis and interpretation

Once we have selected a set of Web APIs, is time to measure, analyze and interpret the data we obtained from the previous part. The goal here is to evaluate whether the adoption of the *Scale First* principles in the development of the Web API causes a beneficial impact in comparison to the Web APIs that don't follow this approach.

Here, it is very important to select the most relevant metrics and then ask questions about the impact of those metrics in the scale process of a Web API:

- How much time does the scale process take?
- How many people were involved in the process?
- What was the user feedback (if any)?
- How much money is needed to scale the Web API resources?
- Which cloud provider has better response time?
- How were the memory consumption or disks' input/output operations impacted?

V. Conclusion

This paper proposes a development approach for Web APIs and creates an experimental setup. This proposal takes into consideration the constantly-changing environment in which Web APIs live, and then exposes a set of alternatives to make these kinds of applications more adaptive. This is achieved by putting together patterns and practices that already exist in software development, but are not part of a single body of knowledge for the context of Web APIs.

Today Web APIs are everywhere. They act as backing services behind desktop, mobile and Web applications, which is why these development efforts are more important than ever. *Scale First* is an attempt to address Web API software development from a scalability standpoint. And, although we are just scratching the surface regarding the eventual benefits of this approach, we see a lot of potential in it in comparison with current and old methodologies and approaches to create software for the Web.

References

[1] M. Abbot and M. Fisher, *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise*, 1st ed. Massachusetts, United States: Person Education, 2010.

[2] M. Abbot and M. Fisher, *Scalability Rules: 50 Principles for Scaling Web Sites*, 1st ed. Massachusetts, United States: Person Education, 2011.

[3] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change.* 2nd edition, Addison-Wesley, 2004.

[4] L. Chen, *Continuous Delivery: Huge Benefits, but Challenges Too*, IEEE Software, vol. 32, no 2, 2015, pp. 50-54.

[5] A. Cockburn, *Walking skeleton*. Available at: http://alistair.cockburn.us/Walking+skeleton

[6] U. Dahan. *Clarified CQRS*. December, 2009. Available at: http://udidahan.com/2009/12/09/clarified-cqrs/

[7] E. Espinha, A Zaidman and HG Gross *Web API Growing Pains: Stories from Client Developers and Their Code*, Delf University of Technology, The Netherlands. 2014

[8] E. Evans. *Domain-Driven Design*. Addison-Wesley. 2003.

[9] M. Fowler. *BoundedContext*. January, 2014. Available at: http://martinfowler.com/bliki/BoundedContext.html

[10] M. Fowler, *Microservices*. Available at: http://martinfowler.com/articles/microservices.html

[11] J. Humble and D. Farley, *Continuous delivery : reliable software releases through build, test, and deployment automation*, 1st ed. Massachusetts, United States: Person Education, 2011.

[12] J. Kabbedijk and S. Jansen and S. Brinkkemper. *A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software*. Proceedings of the 17th European Conference on Pattern Languages of Programs. 2012.

[13] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. US Nat'l Institute of Standards and Technology. 2011.

[14] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media Inc, 2015

[15] E. Raymond. *The Art of the UNIX Programming*. 1st ed. Addison-Wesley. 2003.

[16] L. Richardson and M. Amundsen, *RESTful Web APIs*, 1st ed. California, United States: O'Reilly Media Inc, 2013.

[17] RightScale. *RightScale 2015 State of the Cloud Report*. RightScale, Inc. 2005. Available at: http://www.rightscale.com/lp/2015-state-of-the-cloud-report?campaign=701700000012UP6

[18] A. Rotem-Gal-Oz. *SOA Patterns*. 1st ed. Manning Publications. 2012.

[19] N. Serrano, G. Gallardo and J. Hernantes, *Infrastructure as a Service and Cloud Technologies*, IEEE Software, vol. 32, no 2, 2015, pp. 30-36.

[20] T. Schlossnagle, *Scalable Internet Architectures*, O'Reilly Open Source Convention 2010. O'Reilly Media Inc, 2010.

[21] B. Srivastrava. *Composing Web APIs: State of the Art and Mobile Implications (Tutorial)*. Proceeding MOBILESoft 2014 Proceedings of the 1st International Conference on Mobile Software Engineering and Systems. 2014.

[22] V. Vernon. *Implementing Domain Driven Design*. 1st ed. Addison-Wesley. July, 2013.