O'REILLY®

Head First

Software Architecture

A Learner's Guide to Architectural Thinking

Raju Gandhi,
Mark Richards
& Neal Ford

A Brain-Friendly Guide

Head First Software Architecture

A Learner's Guide to Architectural Thinking

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Raju Gandhi, Mark Richards, Neal Ford

Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First Software Architecture

by Raju Gandhi, Mark Richards, and Neal Ford

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Melissa Duffield
- Development Editor: Sarah Grey
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- December 2023: First Edition

Revision History for the Early Release

- 2023-03-22: First Release
- 2023-04-19: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098134358 for release details.

licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-13429-7

Chapter 1. Software Architecture Demystified: *Let's Get Started!*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor (*sgrey@oreilly.com*)

Figure 1-1.

Software architecture is fundamental to the success of your system. This chapter demystifies software architecture. You'll gain an understanding of architectural dimensions and understand the differences between architecture and design. Why is this important? Because understanding and applying architectural practices helps you build more effective and correct software systems—those that not only work better functionally, but also meet the needs and concerns of the business and continue to operate as your business and techni-

cal environments undergo constant change. So, without further delay, let's get started.

Building your understanding of software architecture

To better understand software architecture, think about a typical home in your neighborhood. The structure of the home is its **architecture**—things like its shape, how many rooms and floors it has, its dimensions, and so on. A house is usually represented through a building plan, which contains all the lines and boxes necessary to know how to build the house. Structural things like those shown below are hard to change later and are the *important* stuff about the house.

NOTE

The building metaphor is a very popular one for understanding software architecture.

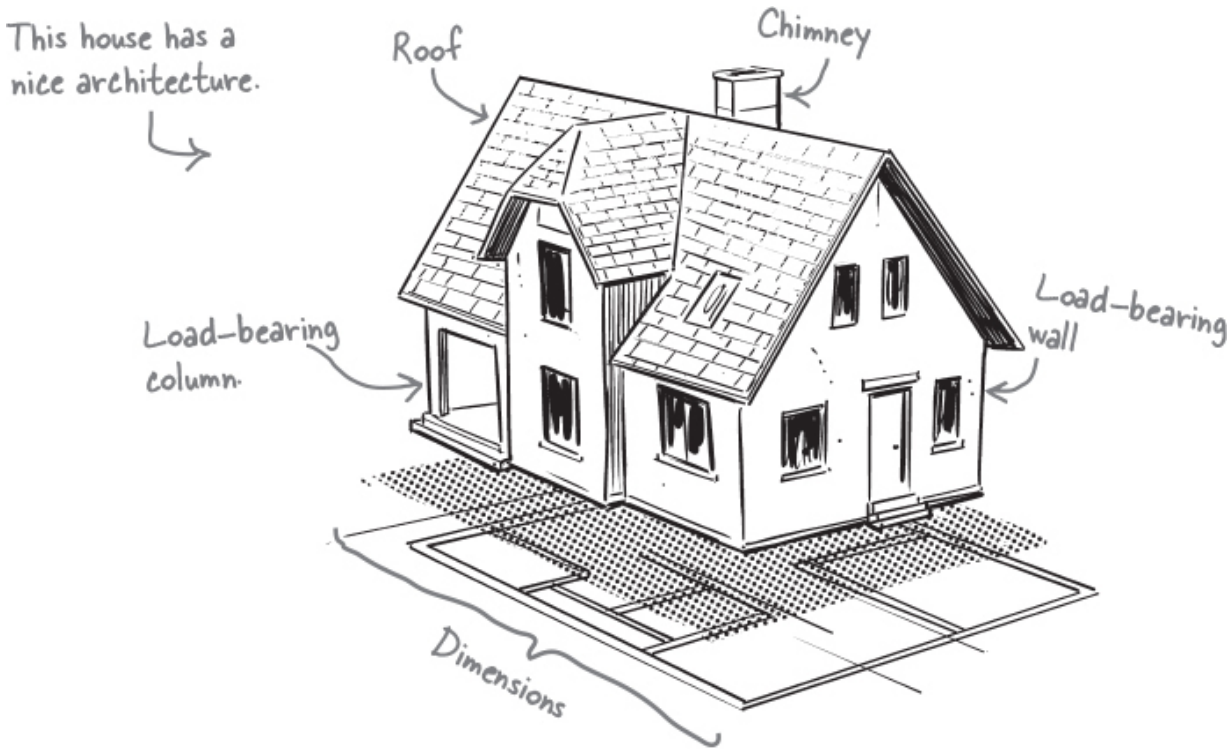


Figure 1-2.

Architecture is essential for building a house. Can you imagine building one without an architecture? It might turn out looking something like the house on the right.

Like physical structures, architecture is essential for building software systems as well. Have you ever come across a system that doesn't scale, is not relaible, or is too difficult to maintain? It's likely not enough emphasis was placed on its architecture.

Figure 1-3.

EXERCISE

Figure 1-4.

Gardening is often used as a metaphor for describing software architecture. Using the space below, can you describe how planning a garden might relate to software architecture? You can see what we came up with at the end of this chapter.

Building plans and software architecture

You might be wondering how the building plans of your home relate to software architecture. Each is a representation of the thing you're building. So what does a "building plan" of a software system look like? Lines and boxes of course.

A building plan specifies the structure of your home—the rooms, walls, stairs, and so on—in the same way a software architecture diagram specifies its structure—user interfaces, services, databases,

and communication protocols. Both artifacts provide guidelines and constraints, as well as a vision of the final result.

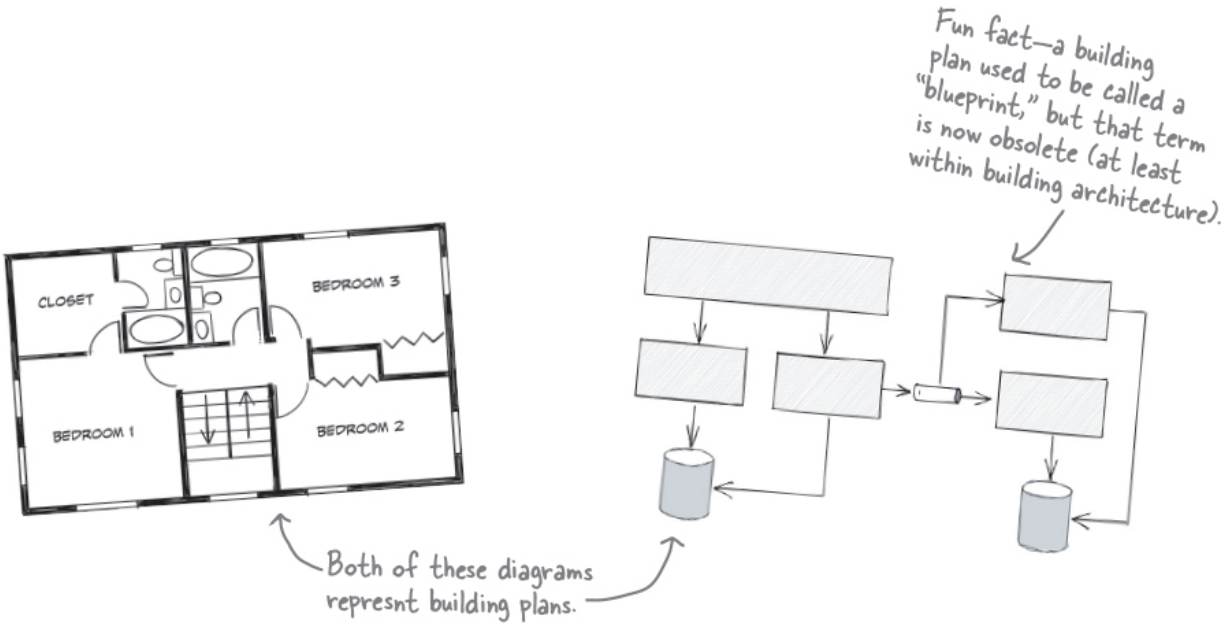


Figure 1-5.

SHARPEN YOUR PENCIL

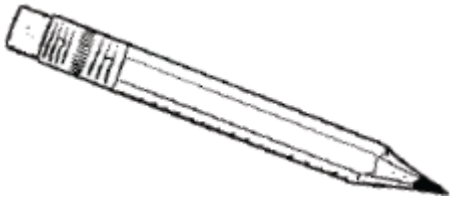


Figure 1-6.

What features of your home can you list that are *structural* and related to its *architecture*? You can find our thoughts at the end of this chapter.

NOTE

Use this space to write down your ideas.

Did you notice that the floor plan for the house above doesn't specify the details of the rooms—things like the type of flooring (carpet or hardwood), the color of the walls, and where a bed might go in a bedroom? That's because those things aren't **structural**. In other words, they don't specify something about the architecture of the house, but rather its design.

NOTE

Don't worry—you'll learn a lot more about this distinction later in this chapter. Right now, just focus on the structure of something—in other words, its architecture.

The dimensions of software architecture

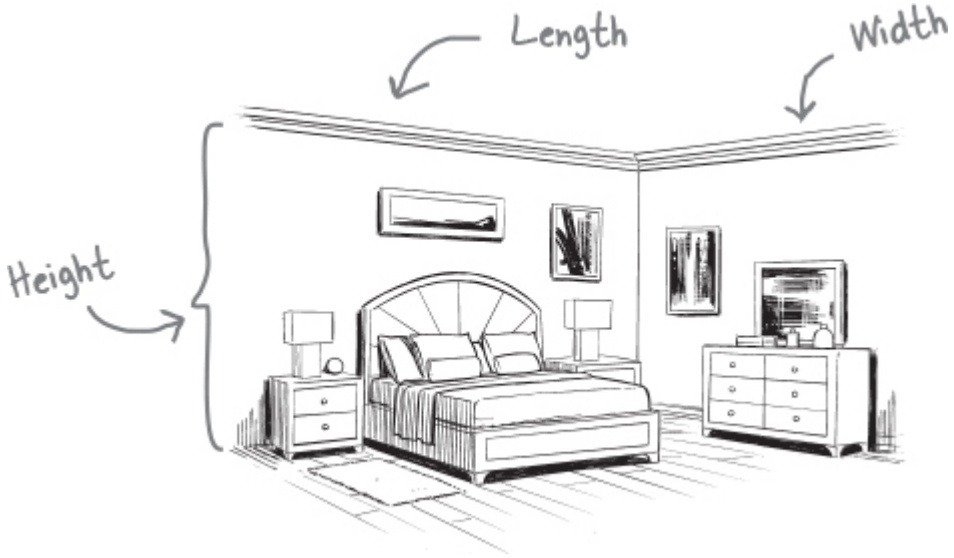


Figure 1-7.

Most things around us are multidimensional. For example, you might describe a particular room in your home by saying it is 5 meters long and 4 meters wide, with a ceiling height of 2.5 meters. Notice that to properly describe the room you needed to specify all three dimensions—its height, length and width.

Software architecture is no different. Like a room in your house, you can describe a software architecture by its dimensions. The difference is that software architecture has **four dimensions**.

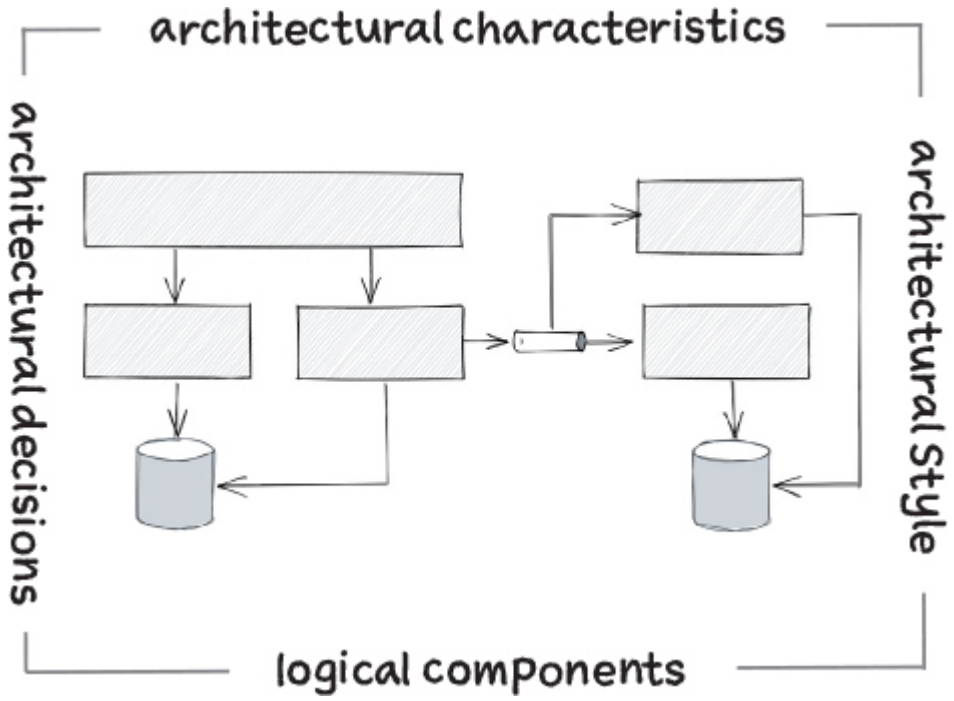


Figure 1-8.

❶ Architectural characteristics

This dimension describes what aspects of the system the architecture needs to support—things like scalability, testability, availability, and so on.

❷ Architectural decisions

This dimension includes important decisions that have long-term or significant implications for the system—for example, the kind of database it uses, the number of services it has, and how those services communicate with each other.

❸ Logical components

This dimension describes the building blocks of the system's functionality and how they interact with each other. For example, an e-commerce system might have components for inventory management, payment processing, and so on.

❹ Architectural style

This dimension defines the overall physical shape and structure of a software system in the same way a building plan defines the overall shape and structure of your home.

NOTE

You'll learn about five of the most common architecture styles later in this book.

Puzzling out the dimensions

You can think of software architecture as a puzzle, with each dimension representing a separate puzzle piece. While each piece has its own unique shape and characteristics, they must all fit together and interact to build a complete picture. And that's exactly what we're going to do—help you put together a much more complete picture of software architecture.

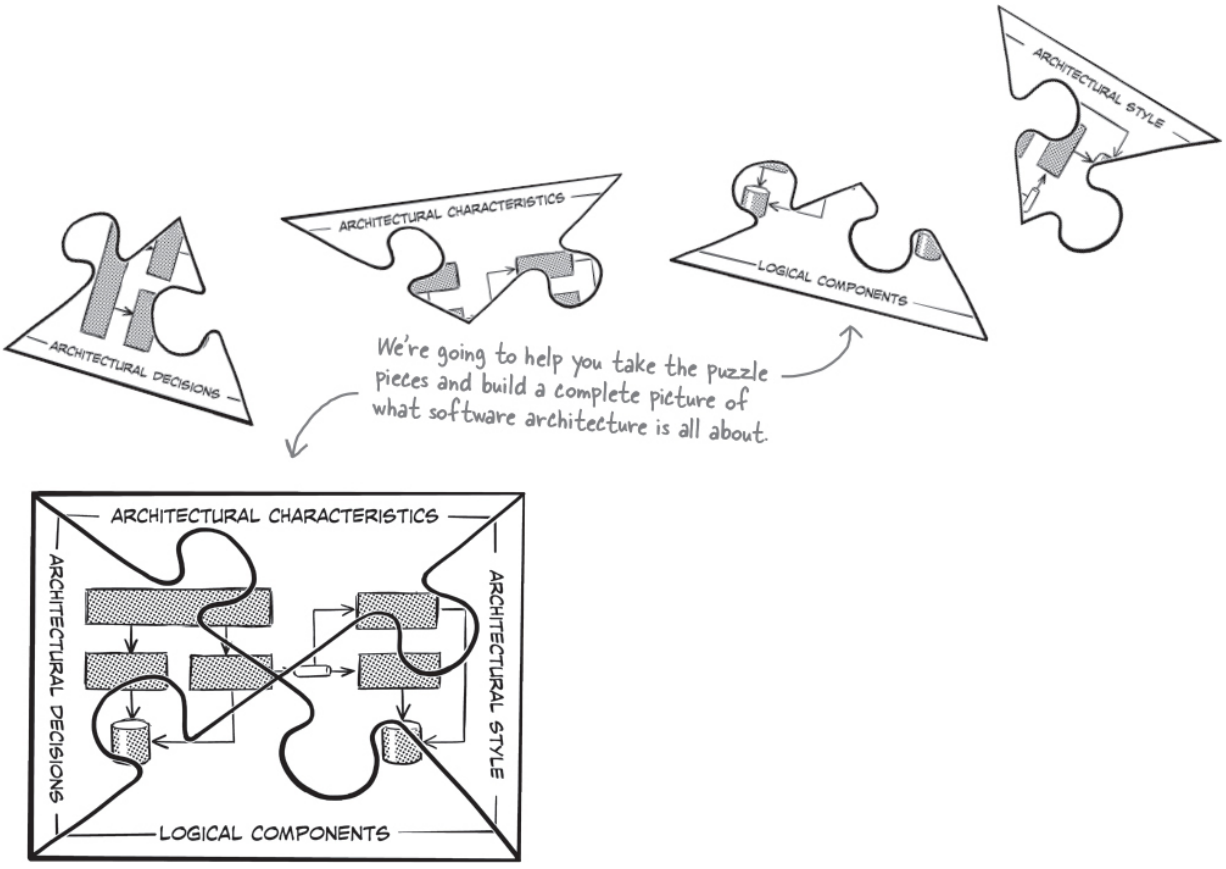


Figure 1-9.

Everything is interconnected.

Do you notice how the pieces of this puzzle are joined in the middle? That's exactly how software architecture works: each dimension must align.

The architecture style must align with the architectural characteristics you choose as well as the architectural decisions you make. Similarly, the logical components you define must align with the characteristics and the architectural style as well as the decisions you make.

All of this interconnectivity represents the *domain*—the problem you are actually trying to solve.

THERE ARE NO DUMB QUESTIONS

Q: Do you need all four dimensions when creating an architecture, or can you skip some if you don't have time?

A: Unfortunately, you can't skip any of these dimensions—they are all required to create and describe an architecture. One common mistake software architects make is using only one or two of these dimensions when describing their architecture. "Our architecture is microservices" describes a single dimension—the architectural style—but leaves too many unanswered questions. For example, what architectural characteristics are critical to the success of the system? What are its logical components (functional building blocks)? What major decisions have you made about how you'll implement the architecture?

The first dimension: Architectural characteristics

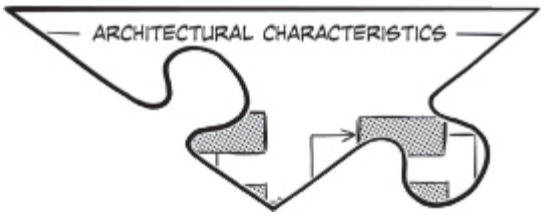


Figure 1-10.

Architectural characteristics form the foundation of the architecture in a software system. Without them, you cannot make architectural decisions or analyze important trade-offs.

Imagine you're trying to choose between two homes. One home is roomy but is next to a busy, noisy motorway. The other home is in a nice, quiet neighborhood, but is much smaller. Which characteristic is more important to you—home size or the level of noise and traffic? Without knowing that, you can't make the right choice.

The same is true with software architecture. Let's say you need to decide what kind of database to use for your new system. Should it be a relational database, a simple key-value database, or a complex graph database? The answer will be based on what architectural characteristics are critical to you. For example, you might choose a graph database if you need high-speed search capability (we'll call that *performance*), whereas a traditional relational database might be better if you need to preserve data relationships (we'll call that *data integrity*).

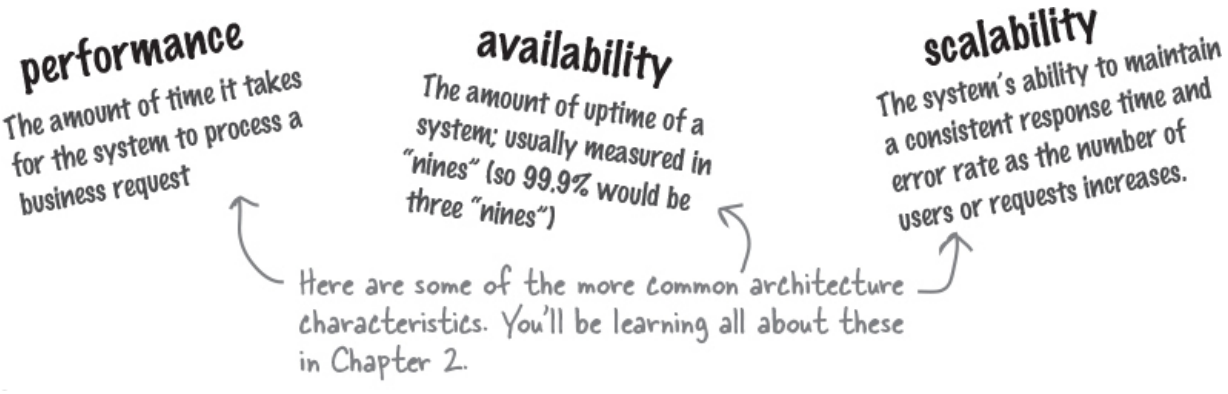

performance
The amount of time it takes for the system to process a business request
availability
The amount of uptime of a system; usually measured in "nines" (so 99.9% would be three "nines")
scalability
The system's ability to maintain a consistent response time and error rate as the number of users or requests increases.
Here are some of the more common architecture characteristics. You'll be learning all about these in Chapter 2.

Figure 1-11.

EXERCISE

Figure 1-12.

Check the things you think might be considered architectural characteristics—something that the *structure* of the software system supports. You can find the solution at the end of the chapter.

☐ **Changing the font size in a window on the user interface screen**

☐ **Making changes quickly**

☐ **Handling thousands of concurrent users**

☐ **Encrypting user passwords stored in the database**

☐ **Interacting with many external systems to complete a business request**

The term **architectural characteristics** might not be familiar to you, but that doesn't mean you haven't heard of them before. Collectively,

things like performance, scalability, reliability, and availability are also known as non-functional requirements, system quality attributes, and simply "the ilities" because most end with the suffix *-ility*. We like the term *architectural characteristics* because these help define the characteristics of the architecture and what it needs to support.

NOTE

Architectural characteristics are capabilities that are critical or important to the success of the system.

MAKE IT STICK

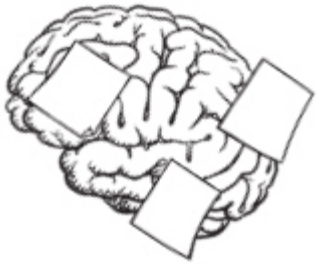


Figure 1-13.

To architect software you must first address

Capabilities key to the new app's success

WHO DOES WHAT?

Here's your chance to see how much you already know about many common architectural characteristics. Can you match up the architectural characteristic on the left with the definition on the right? You'll notice there are more definitions than characteristics, so be careful—not all of the definitions have matches. You can find the solution at the end of the chapter.

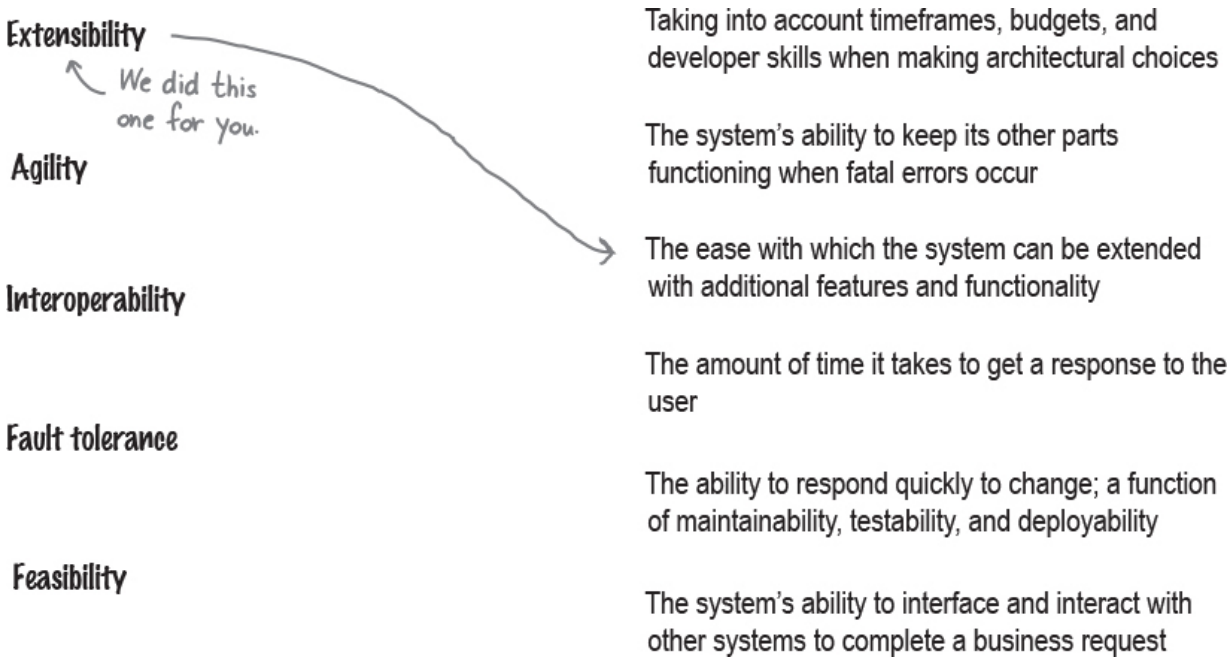


Figure 1-14.

The second dimension: Architectural decisions

Architectural decisions are choices you make about structural aspects of the system that have long-term or significant implications. As constraints, they'll guide your development teams in planning and building the system.

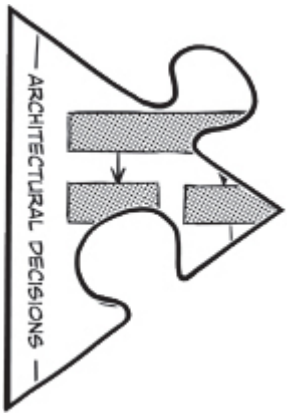


Figure 1-15.

Should your new home have one floor or two? Should the roof be flat or peaked? Should you build a big, sprawling ranch house? These are good examples of architecture decisions because they involve the *structural* aspect of your home.

Figure 1-16.

You might decide that your system's user interface should not communicate directly with the database, but instead must go through the underlying services to retrieve and update data. This architectural decision places a particular constraint on the development of the user interface, and also guides the development team about how other components should access and update data in the database.

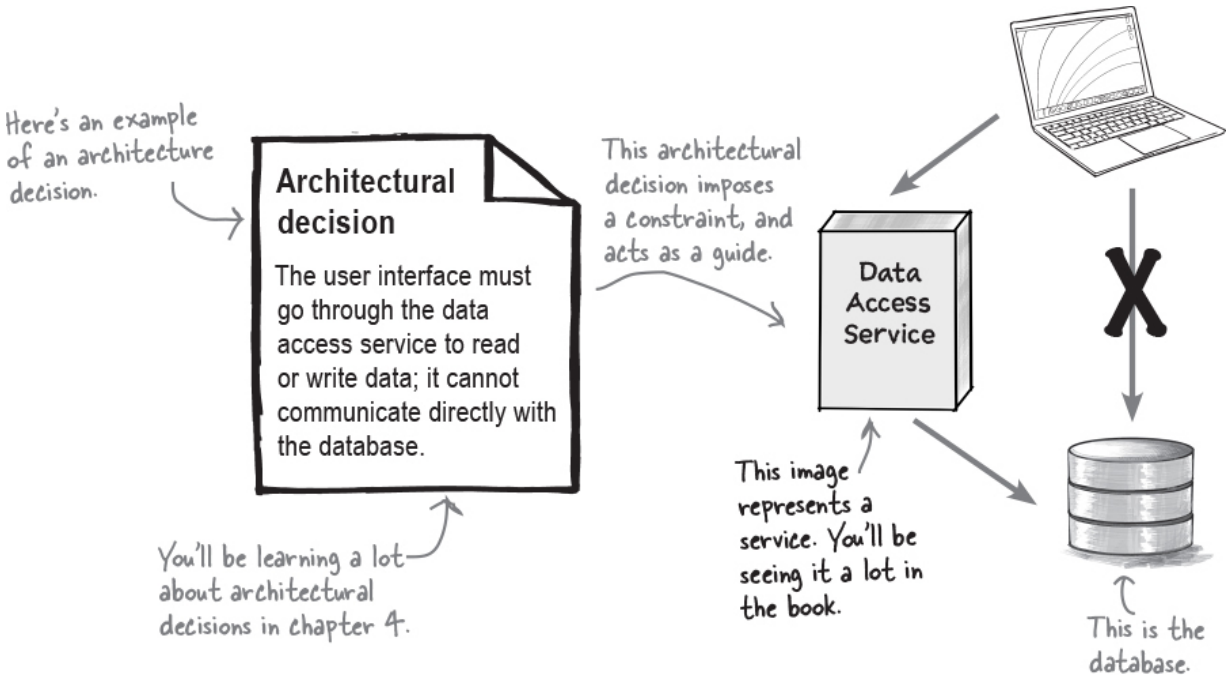


Figure 1-17.

MAKE IT STICK

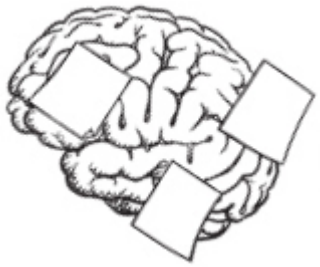


Figure 1-18.

Decisions are structural guides for dev teams

Significant trade-offs are mostly their themes

It's not uncommon to have several dozen or more architectural decisions within any system. Generally, the larger and more complicated the system, the more architectural decisions it will have.

BE the architect

Figure 1-19.

Your job is to be the architect and identify as many architectural decisions you can in the diagram below. Draw a circle around anything that you think might be an architectual decision and write what that decision might be. After you're done, look at the end of the chapter for the solution.

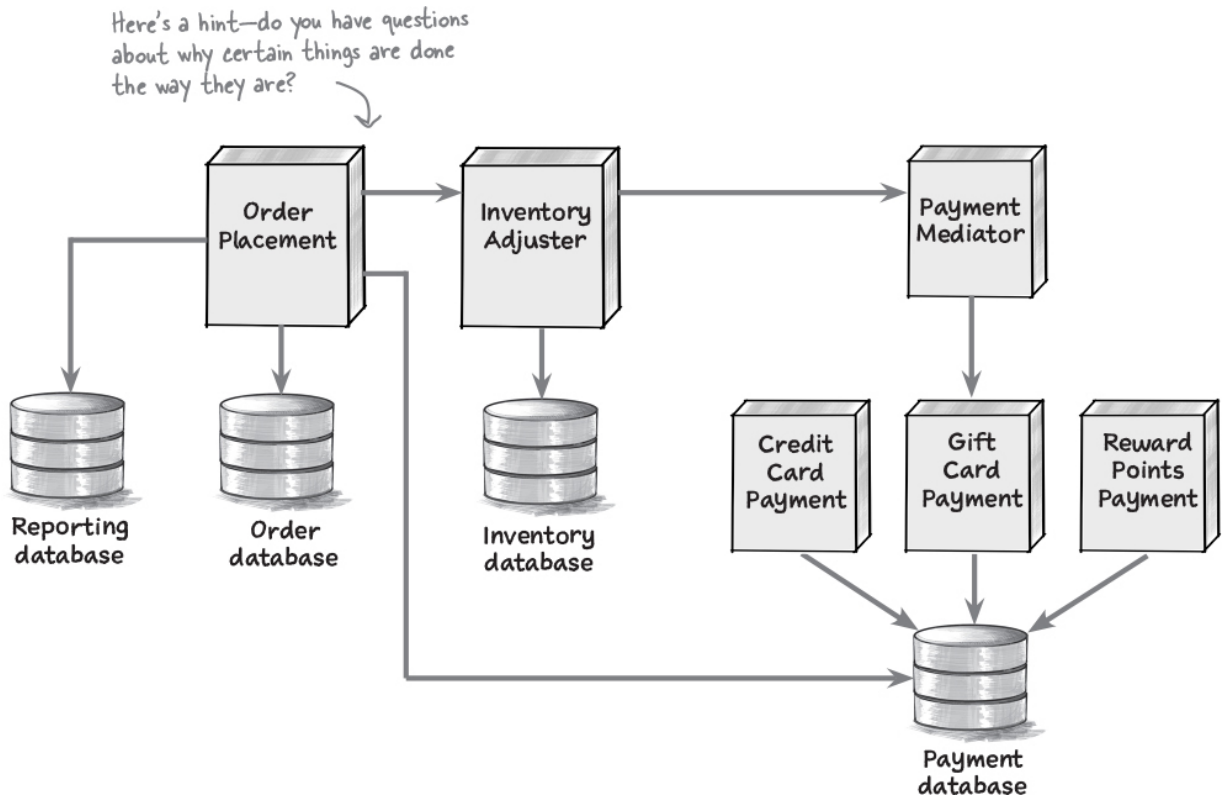


Figure 1-20.

The third dimension: Logical components

Logical components are the building blocks of a system, much in the same way rooms are the building blocks of your home. A logical component performs some sort of function—such as processing the payment for an order, managing item inventory, and tracking orders.

Figure 1-21.

Logical components in a system are usually represented through a directory or namespace. For example, the directory `app/order/payment` with the corresponding namespace `app.order.payment` identifies a logical component named `Payment Processing`. The source code that allows users to pay for an order is stored in this directory and use this namespace.

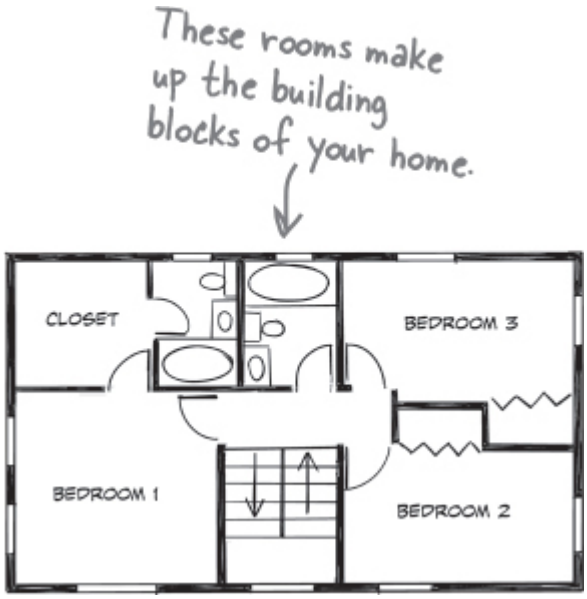


Figure 1-22.

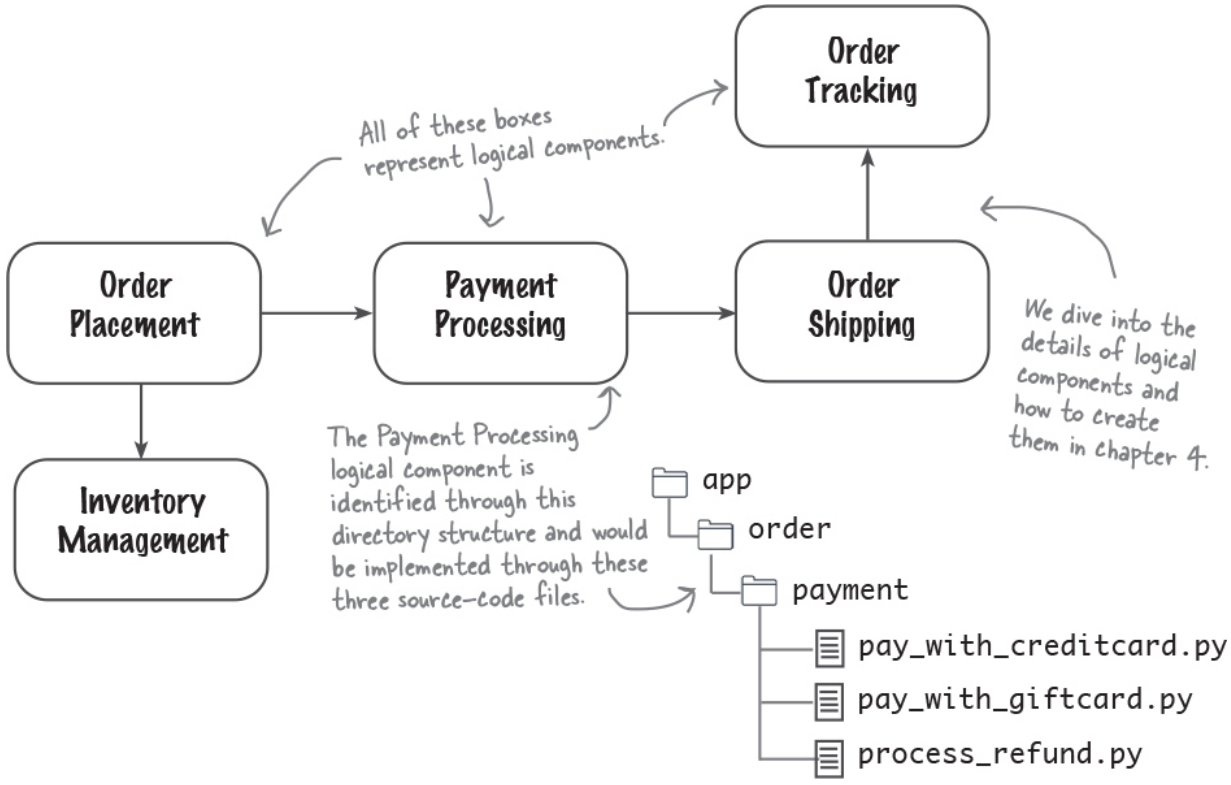

Order Tracking
All of these boxes represent logical components.
Order Placement
Payment Processing
Order Shipping
We dive into the details of logical components and how to create them in chapter 4.
Inventory Management
The Payment Processing logical component is identified through this directory structure and would be implemented through these three source-code files.
app
order
payment
pay_with_creditcard.py
pay_with_giftcard.py
process_refund.py

Figure 1-23.

SHARPEN YOUR PENCIL

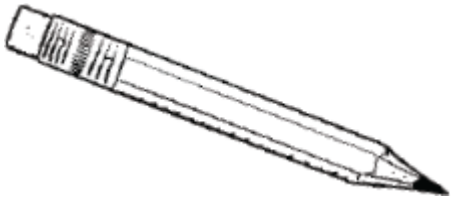


Figure 1-24.

You've just created the following two components for a new system called **BuyFromUs**, and your development team wants to start writing class files to implement them. Can you create a directory structure for them so they can start coding? Flip to the end of the chapter for our solution.

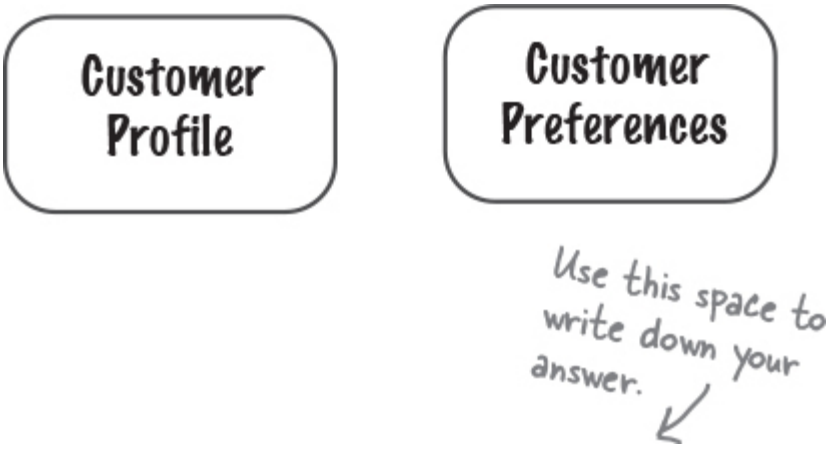


Figure 1-25.

A logical component should always have a well-defined role and responsibility in the system—in other words, what it does.

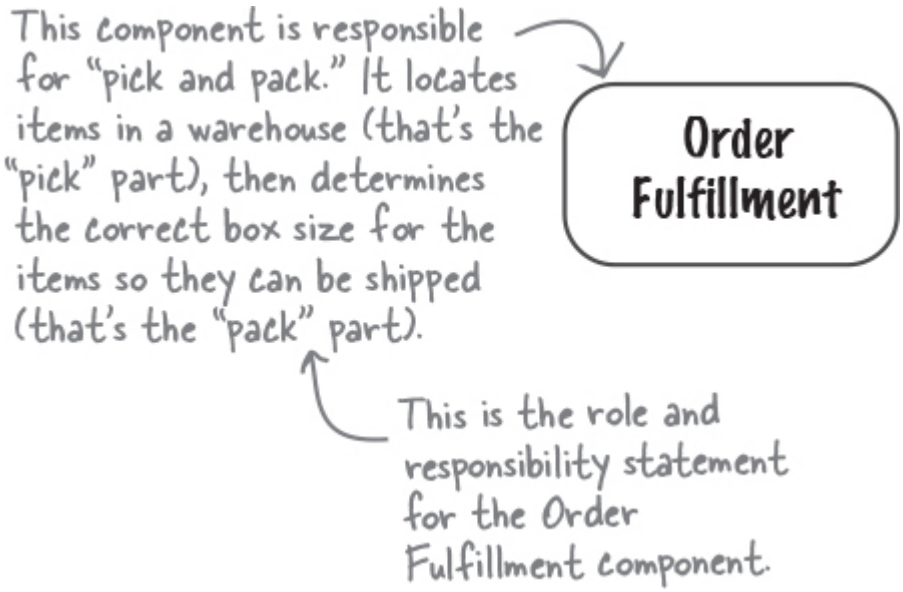


Figure 1-26.

MAKE IT STICK

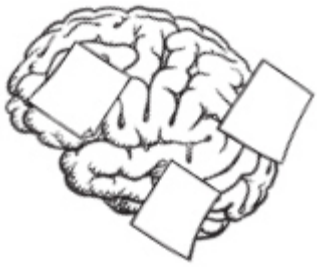


Figure 1-27.

Logical components are blocks in conjunction

They hold the source code for each business function

THERE ARE NO DUMB QUESTIONS

Q: What is the difference between system functionality and the domain?

A: The domain is the problem you are trying to solve, and the system functionality is how you are solving that problem. In other words, the domain is the "what," and the system's functionality is the "how."

The fourth dimension: Architectural styles

Homes come in all shapes, sizes, and styles. While there are some wild-looking houses out there, most conform to a particular style, such as Victorian, ranch, or Tudor. The style of a home says a lot about its overall structure. For example, ranch homes typically have only one floor; colonial and Tudor homes typically have chimneys; contemporary homes typically have flat roofs.

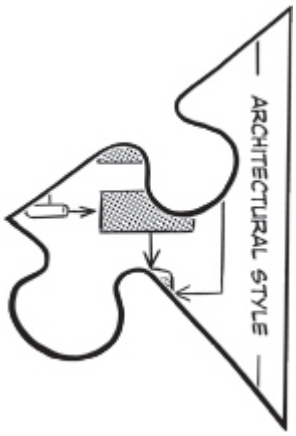


Figure 1-28.

Figure 1-29.

Architectural styles define the overall shape and structure of a software system, each with its own unique set of characteristics. For example, the microservices architectural style scales very well and provides a high level of *agility*—the ability to respond quickly to change—whereas the layered architecture style is less complex and less costly. The event-driven architectural style provides high levels of scalability and is very fast and responsive.

NOTE

Don't worry—you'll be learning all about these architectural styles later on in the book. We've devoted chapters to each of them.

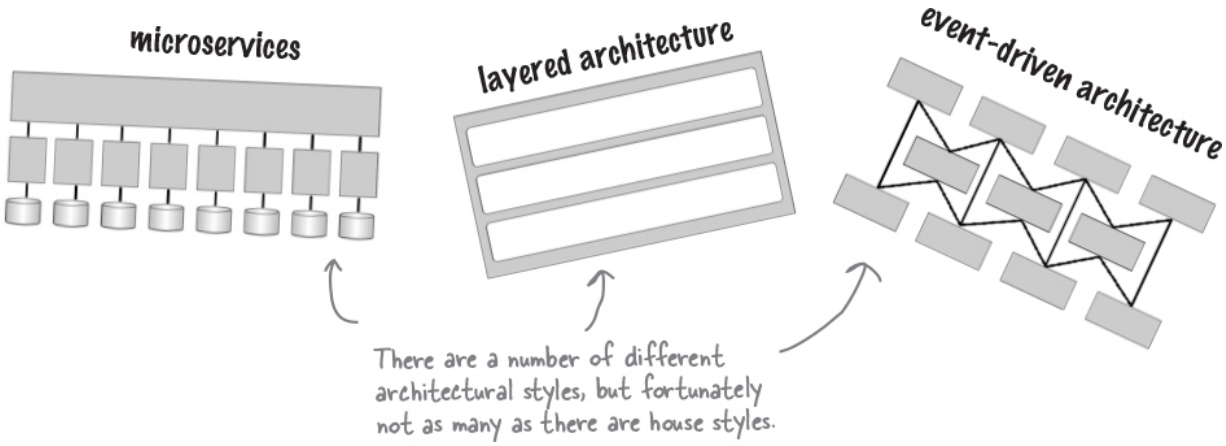


Figure 1-30.

Because the architecture style defines the overall shape and characteristics of the system, it's important to get it right the first time. Why? Can you imagine starting construction on a one-story ranch home, and in the middle of construction changing your mind and wanting a three story Victorian house instead? That would be a major undertaking, and likely exceed your budget as well as when you want to move into the house.

Software architecture is no different. It's not easy changing from a monolithic layered architecture to microservices. Like the house example, this would be quite an undertaking.

MAKE IT STICK

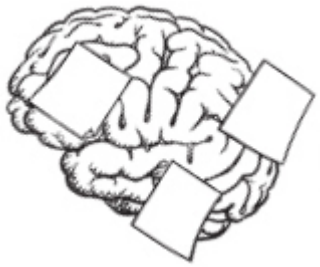


Figure 1-31.

Styles shape the system and help serve its purposes–

You might choose a monolith or microservices.

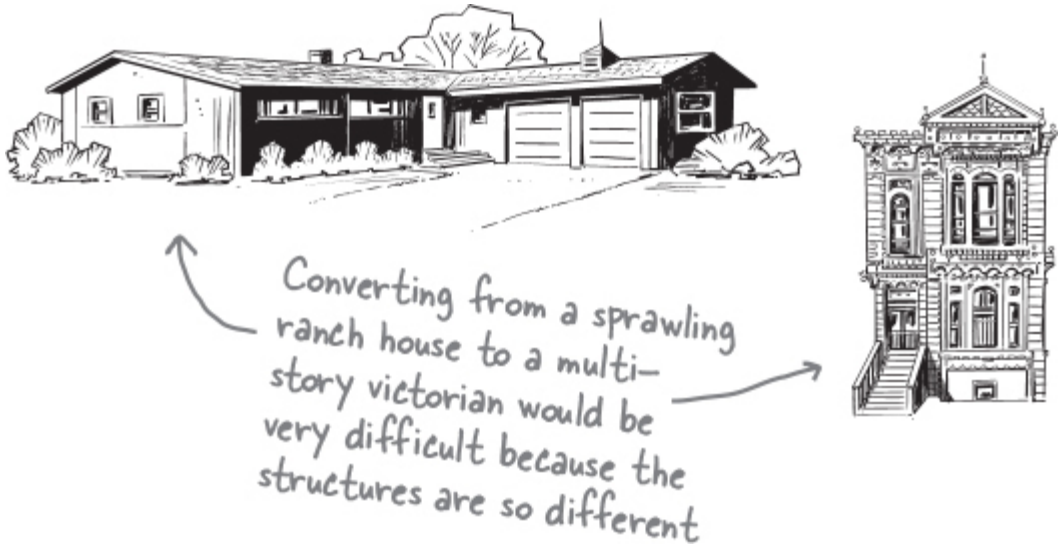


Figure 1-32.

Later in the book, we'll show you how to properly select an architecture style based on architectural characteristics that are important to

you.

Which brings us back to an earlier point—all of the dimensions of software architecture are interconnected. You can't select an architectural style without knowing what's important to you.

BRAIN POWER

Figure 1-33.

The tightly wound tendons and muscles in a lion's legs enable it to reach speeds as fast as 80 kilometers per hour and leap up to almost 11 meters in the air. This characteristic allows lions to survive by catching fast prey.

Look around you—what else has a structure or shape that defines its characteristics and capabilities?

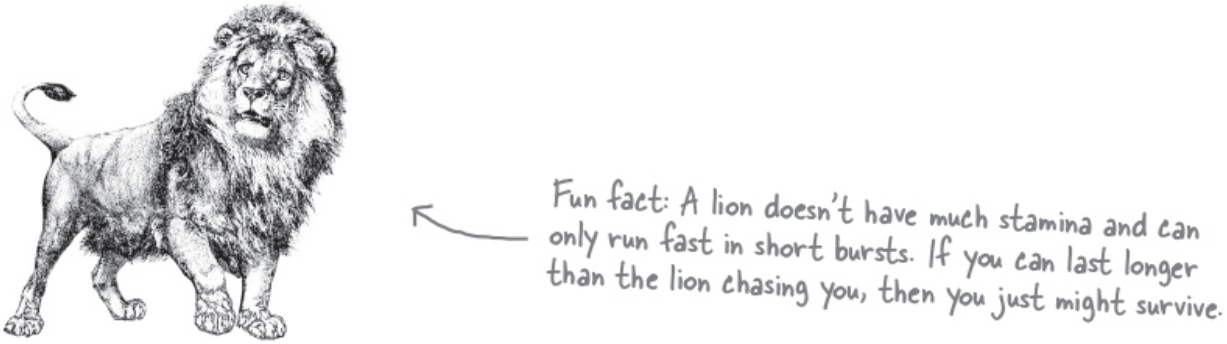


Figure 1-34.

WHO DOES WHAT?

We were trying to describe our architecture, but all the puzzle pieces got mixed up. Can you help us figure out which dimension does what by matching the statements on the left with the software architecture dimensions on the right? Be careful—some of the statements don't have a match because they are not related to *architecture*. You can find the solution at the end of this chapter.

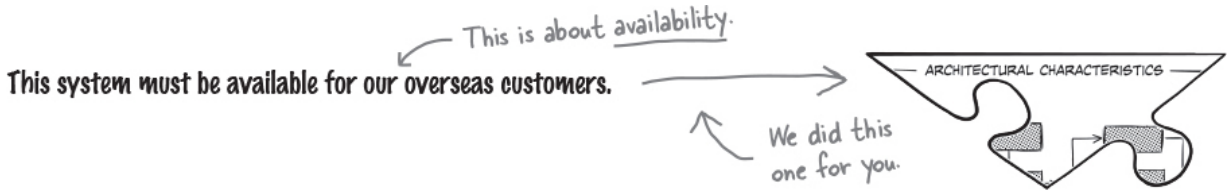


Figure 1-35.

Customers are complaining about the background color of the new user interface.

The product owner insists that we get new features and bug fixes out to our customers as fast as possible.

Our system uses an event-driven architecture.

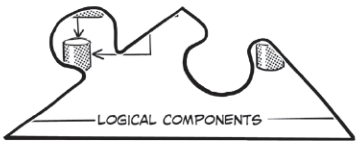
We need to support up to 300,000 concurrent users in this system.

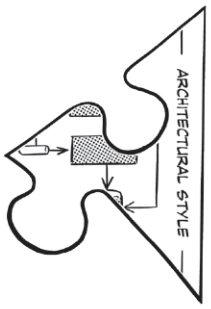
The single payment service will be broken apart into separate services, one for each payment type we accept.

We are going to start offering reward points as a new payment option when paying for an order.

We are breaking up the Order class into three smaller class files.

The user interface shall not communicate directly with the database.





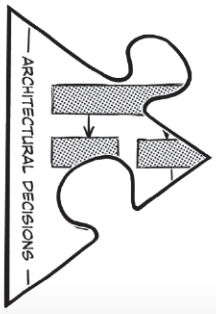


Figure 1-36.


If I'm responsible for the design of a software system, does that mean I am responsible for the architecture of the system as well? Aren't these the same thing?

Figure 1-37.

No, architecture and design are different.

You see, architecture is less about appearance and more about structure, where is design is less about structure and more about appearance.

The color of your walls, the placement of furniture, and the type of flooring you use (carpet or wood) are all aspects of design, whereas the physical size of the room and the height of the ceilings are all part of architecture—in other words, the **structure** of the room.

Think about a typical website . The architecture, or structure, is all about how the web pages communicate with backend services and databases to retrieve and save data, whereas design is all about what each page looks like: the colors, the placement of the fields, and so on. Again, it becomes a matter of structure versus appearance.

Your question is a good one, because sometimes it gets confusing trying to tell what is considered architecture and what is considered design. So let's investigate these differences.

A design perspective

Let's say your company wants to replace its outdated order-processing system with a new custom-built one that better suits its specific needs. Customers can place orders and can view or cancel orders once they have been placed. They can pay for an order using a credit card, a gift card, or both payment methods.

Figure 1-38.

From a **design perspective,** you might build a UML class diagram like this one to show how the classes interact with each other to implement the payment functionality. While you could write source code to implement these class files, this design says nothing about the **physical structure** of the source code—in other words, how these class files would be organized and deployed.

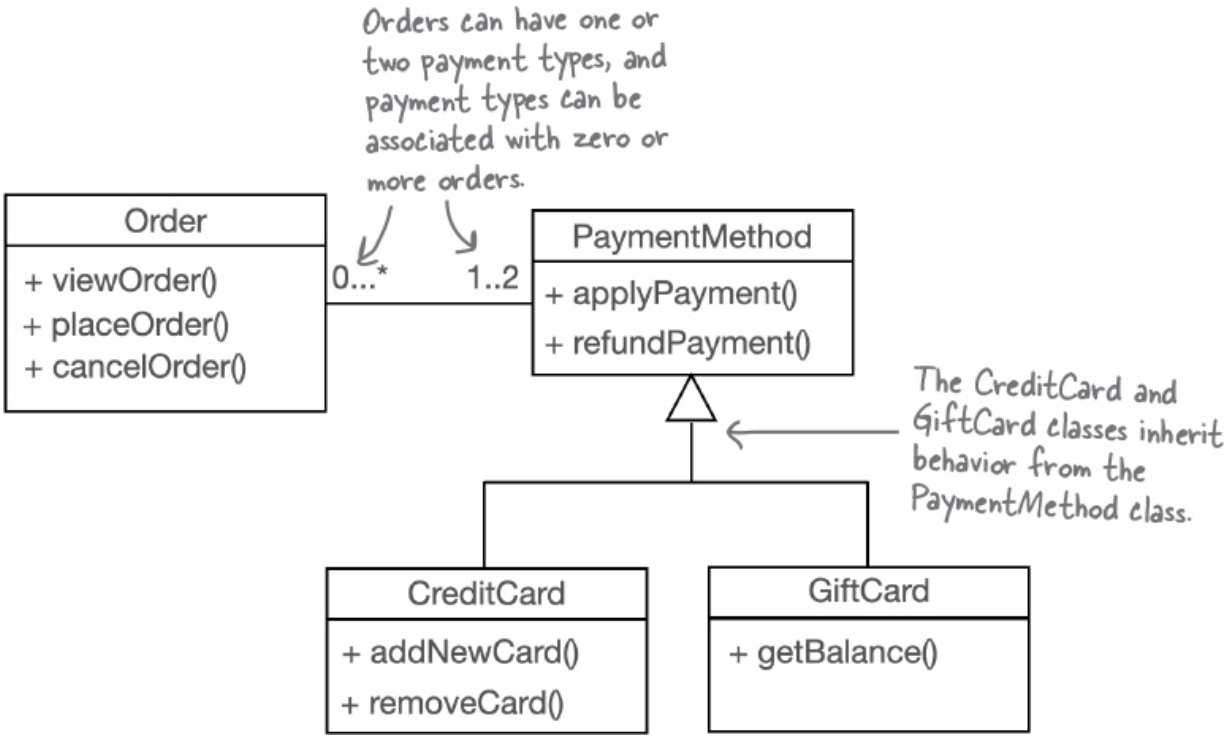


Figure 1-39.

An architectural perspective

Unlike design, architecture is about the physical structure of the system—things like services, databases, and how services communicate with each other and to the user interface.

Let's think about that new order-processing system again. What would the **system** look like? From an **architectural perspective** you might decide to create separate services for each payment type within the order payment process, and have a payment orchestrator service to manage the payment processing part of the system like the diagram below.

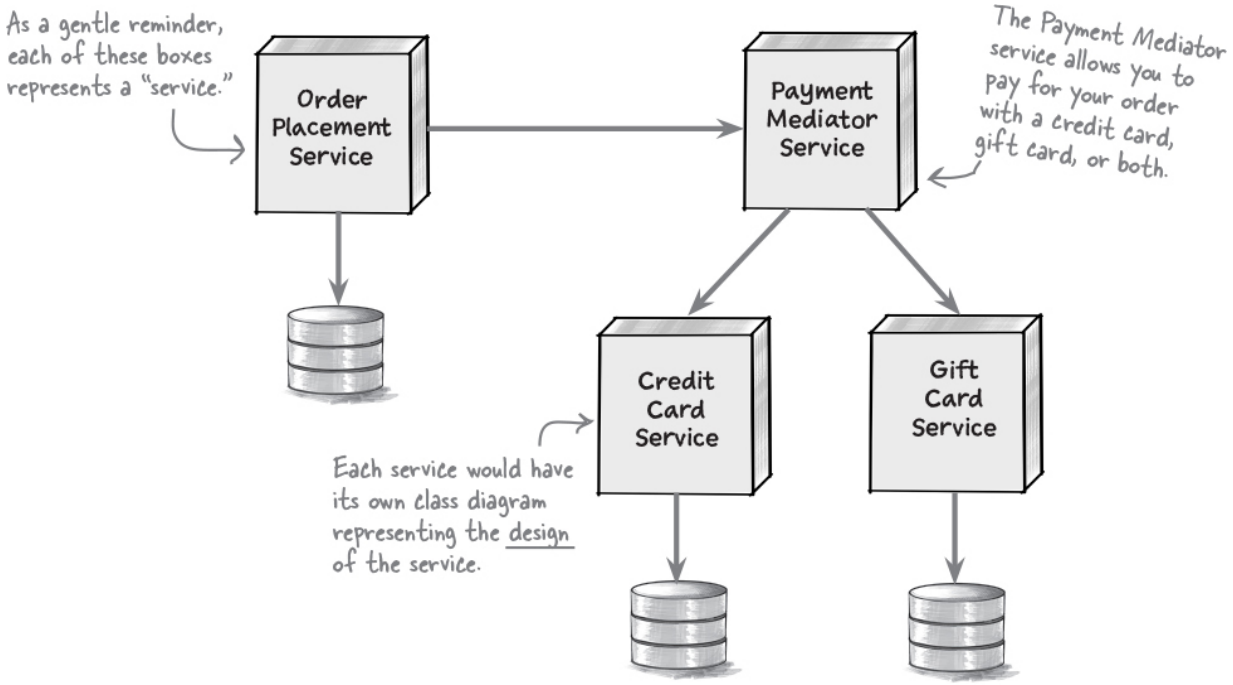

As a gentle reminder, each of these boxes represents a "service."
Order Placement Service
Payment Mediator Service
The Payment Mediator service allows you to pay for your order with a credit card, gift card, or both.
Credit Card Service
Gift Card Service
Each service would have its own class diagram representing the design of the service.

Figure 1-40.

EXERCISE

Figure 1-41.

Check all of the things that should be included in a diagram from an **architectural perspective**. You can find the solution at the end of the chapter.

☐ **How services communicate with each other**

☐ **The platform and language in which the services are implemented**

☐ **Which services can access which databses**

☐ **How many services and databases there are**

The spectrum between architecture and design

Some decisions are certainly architectural (such as deciding which architectural style to use), and others are clearly design related (such as changing the position of a field on a screen). Unfortunately, most decisions you encounter fall between these two examples within a **spectrum** of architecture and design.

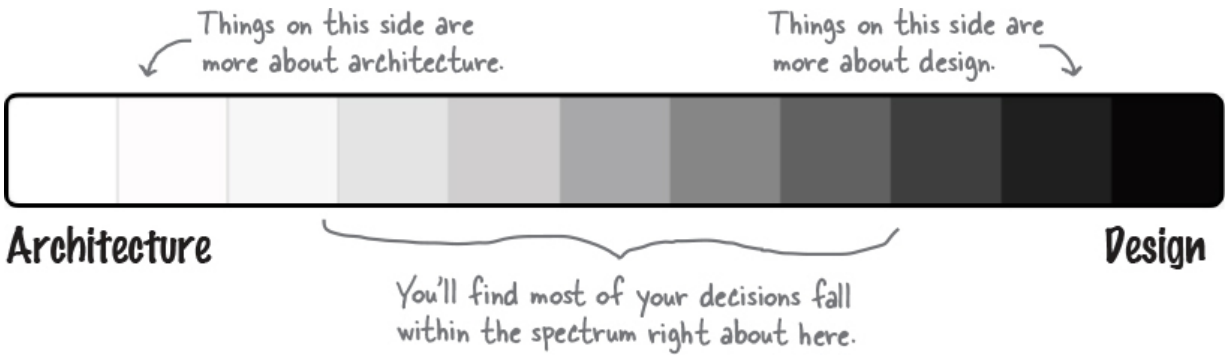


Figure 1-42.

NOTE

Don't worry if you don't know all the answers to this exercise—you'll learn more about this topic on the next page.

SHARPEN YOUR PENCIL

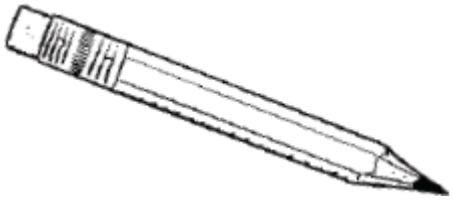


Figure 1-43.

Circle all of the things that you think fall somewhere in the middle of the spectrum **between** architecture and design. You can find the solution in “Sharpen your pencil Solution”.

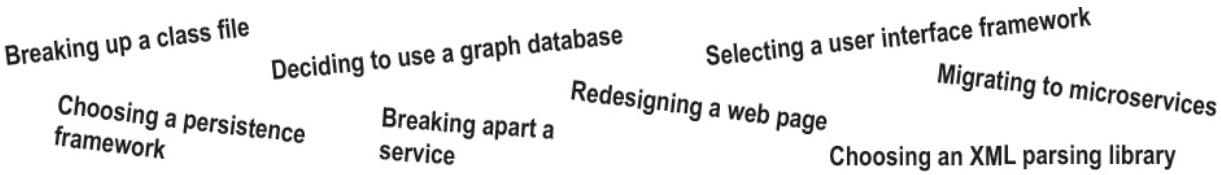


Figure 1-44.

Figure 1-45.

Yes, it matters a lot. You see, knowing where along the spectrum between architecture and design your decision lies helps determine **who** should be responsible for ultimately making that decision. There are some decisions that the development team should make (such as designing the classes to implement a certain feature), some decisions that an architect should make (such as choosing the most appropriate architecture style for a system), and others that should be made together (such as breaking apart services or putting them back together).

Where along the spectrum does your decision fall?

Is it strategic or tactical?

Strategic decisions are long-term and influence future actions or decisions. *Tactical* decisions are short-term and generally stand independent of other actions or decisions (but may be under the context of a particular strategy). For example, deciding how big your new home will be influences the number of rooms and the size of those rooms, whereas deciding on a particular lighting fixture won't affect decisions about the size of your kitchen or dining room table. The more strategic the decision, the more it sits toward the architecture side of the spectrum.

Figure 1-46.

How much effort will it take to construct or change?

Architectural decisions require more effort to construct or change, while design decisions require relatively less. For example, building an addition to your home generally requires a high level of effort and would therefore be more on the architecture side of the spectrum, whereas adding an area rug to a room requires much less effort and would therefore be more on the design side.

NOTE

Sometimes waking up in the morning requires a lot of effort—we'll call those "architecture" mornings.

Figure 1-47.

Does it have significant trade-offs?

Trade-offs are the pros and cons you evaluate as you are making a decision. Decisions that have significant trade-offs require much more time and analysis to make and tend to be more architectural in

nature. Decisions that have less significant trade-offs can be made quicker with less analysis and therefore tend to be more on the design side.

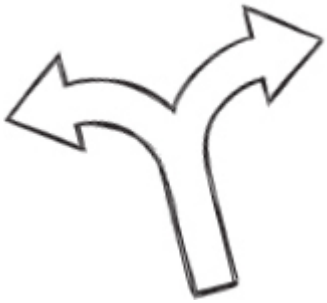


Figure 1-48.

NOTE

We're going to walk you through the details of all three of these factors in the next several pages.

BRAIN POWER

Figure 1-49.

Can you think of a decision that doesn't involve a trade-off, no matter how small or insignificant? Here's a hint: if you think you've found a decision that doesn't involve a trade-off, keep looking.

Strategic versus tactical

The more strategic a decision is, the more architectural it becomes. This is an important distinction, because decisions that are strategic require more thought and planning and are generally long-term.

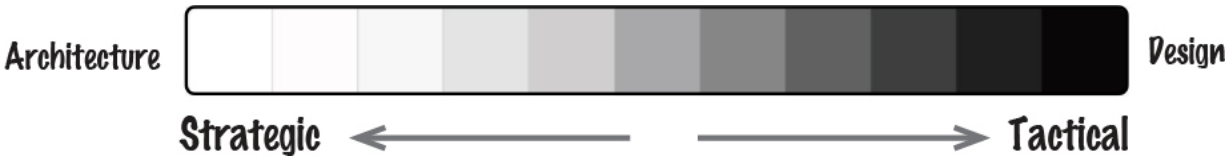


Figure 1-50.


How should I determine whether a decision is more strategic or more tactical?

Figure 1-51.

Indeed there are. You can use these three questions to help determine if something is more strategic or tactical. Remember, the more strategic something is, the more it's about architecture.

1. **How much thought and planning do you need to put into the decision?**

   If making the decision takes a couple of minutes to an hour, it's more tactical in nature. If thought and planning require several days or weeks, it's likely more strategic (hence more architectural).
2. **How many people are involved in the decision?**

   The more people involved, the more strategic the decision. A decisions you can make by yourself or with a colleague is likely to be tactical. A decision that requires many meetings with lots of stakeholders is probably more strategic.
3. **Does your decision involve a long-term vision or a short-term action?**

   If you are making a quick decision about something that is temporary or likely to change soon, it's more tactical and hence more about design. Conversely, if this is a decision you'll be living with for a very long time, it's more strategic and more about architecture.

SHARPEN YOUR PENCIL

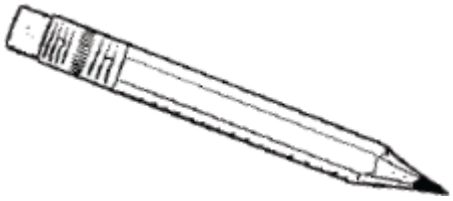


Figure 1-52.

Oh dear. We've lost all of our marbles and we need your help putting them back in the right spot. Using the three questions on the prior page as a guide, can you figure out which jar each marble should go in? You can find the solution at the end of the chapter.

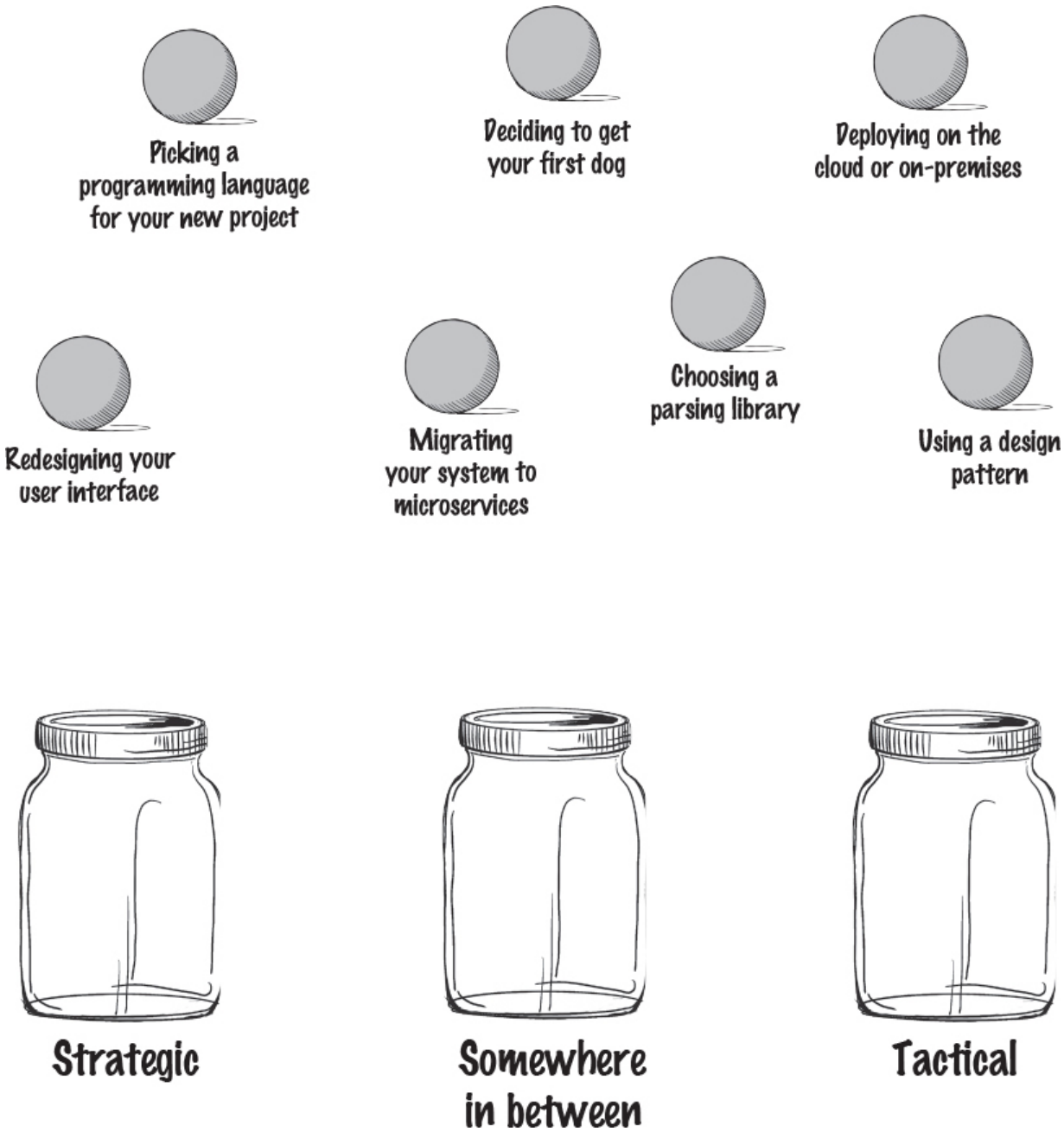


Figure 1-53.

High versus low levels of effort

Martin Fowler, a software architect and author, once wrote that "software architecture is the stuff that's hard to change." You can use Martin's definition to help determine where along the spectrum your decision lies. You see, the harder something is to change later, the further it falls toward the architecture side of the spectrum. Conversely, the easier it is to change later, the more it's probably related to design.

NOTE

Martin Fowler's website (https://martinfowler.com/architecture/) has lots of useful stuff about architecture.

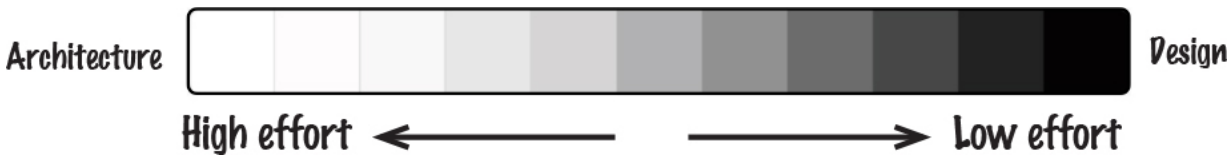


Figure 1-54.

Let's say you are planning on moving from one architecture style to another; say, from a traditional n-tiered layered architecture to microservices. This migration effort is rather difficult and will take a lot of time. Because the level of effort is high, this would be on the far end of the **architecture side** of the spectrum.

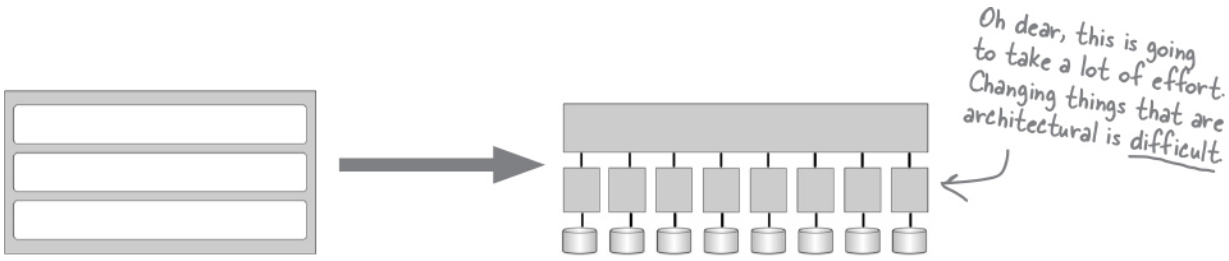


Figure 1-55.

However, let's say you're rearranging fields on a user interface screen. This task takes relatively less effort, so it resides on the far end of the **design side** of the spectrum.

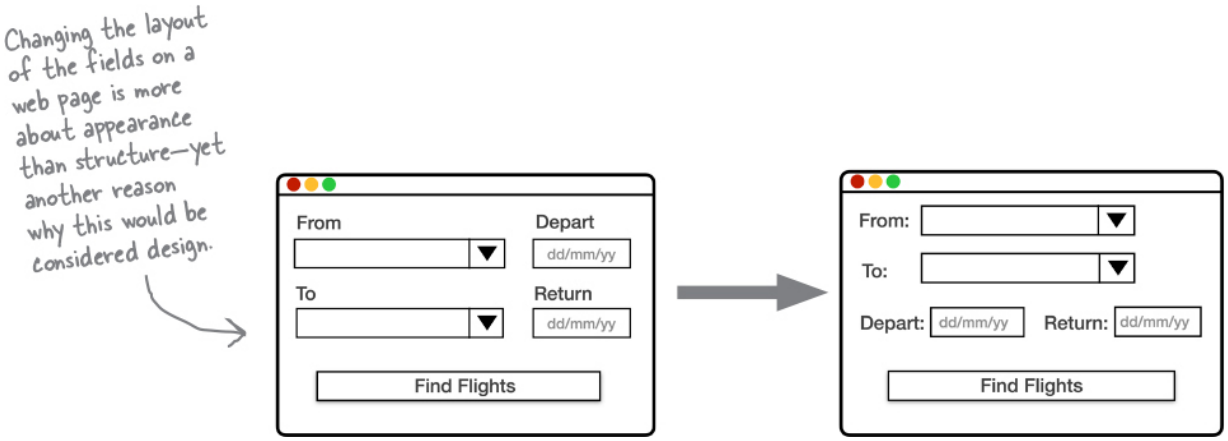


Figure 1-56.

Code Magnets

Figure 1-57.

Oh no. We had all of these magnets from our to-do list arranged from high effort to low effort, and somehow they all fell on the floor and got mixed up. Can you help us put these back in the right order based on the amount of effort it would take to make each change? You can find the solution at the chapter's end.

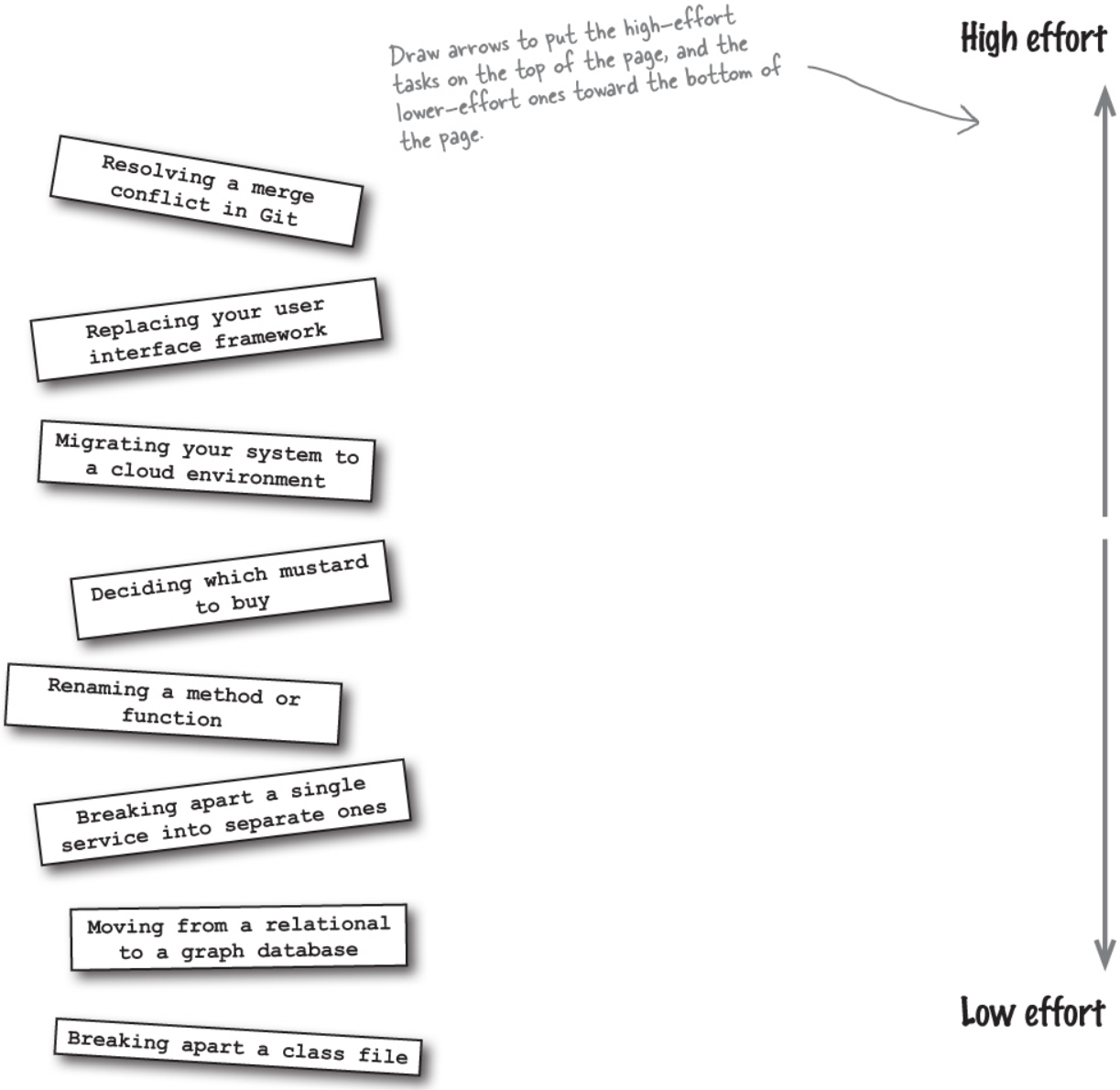


Figure 1-58.

Significant versus less significant trade-offs

Some decisions you make might have significant trade-offs, such as choosing which city to live in. Others might have less significant trade-offs, like deciding on the color of your living-room rug. You can use the level of significance of the trade-offs in a particular decision to help determine whether that decision is more about architecture or design. The more significant the trade-offs, the more it's about architecture; the less significant the tradeoffs, the more it's about design.

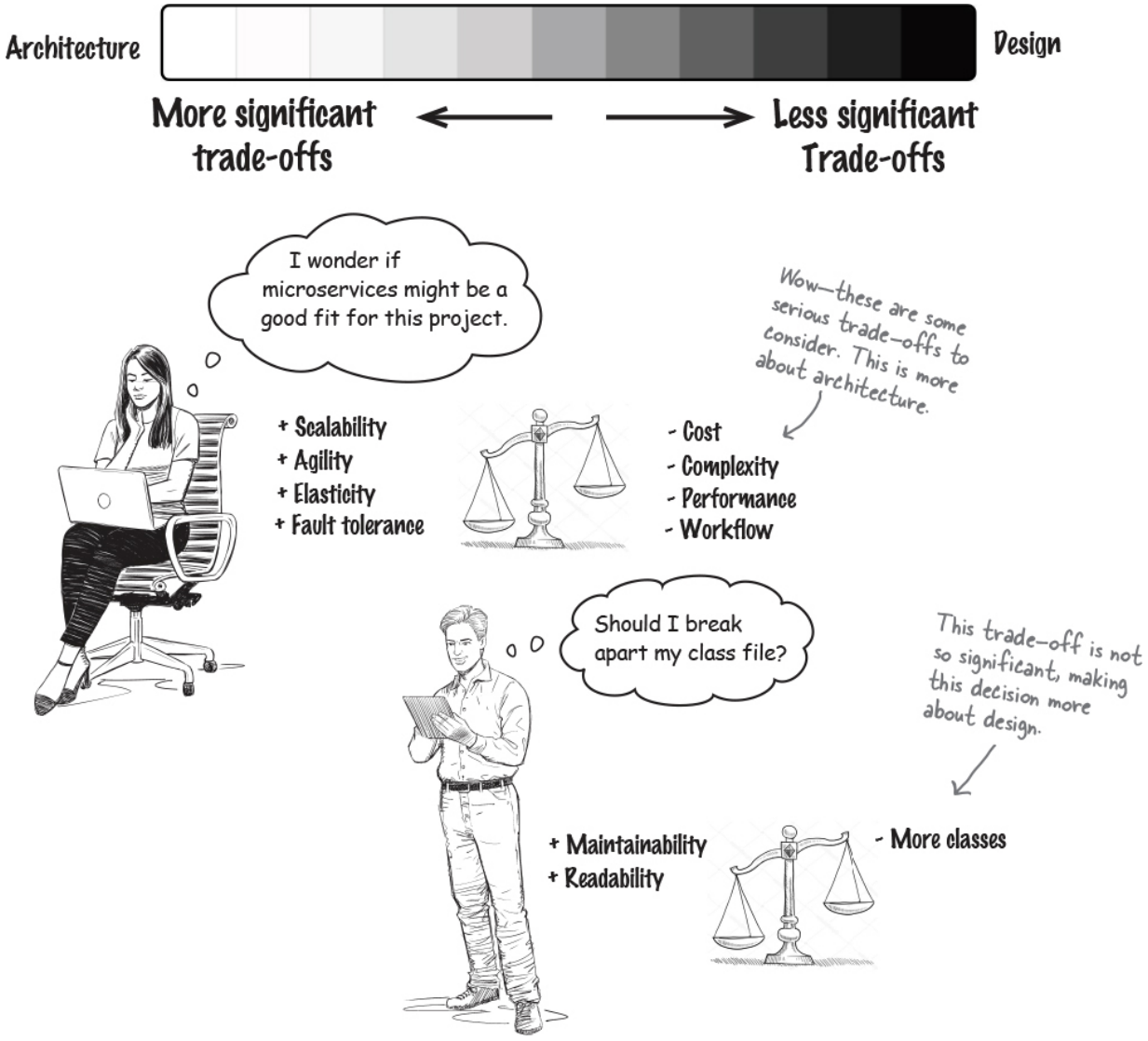

Architecture
Design
More significant trade-offs
Less significant Trade-offs
I wonder if microservices might be a good fit for this project.
+ Scalability
+ Agility
+ Elasticity
+ Fault tolerance
- Cost
- Complexity
- Performance
- Workflow
Wow—these are some serious trade-offs to consider. This is more about architecture.
Should I break apart my class file?
+ Maintainability
+ Readability
- More classes
This trade-off is not so significant, making this decision more about design.

Figure 1-59.

EXERCISE

Figure 1-60.

Decisions, decisions, decisions. How can we ever tackle all of these decisions? One thing we think might help is to identify all of those decisions that have significant trade-offs, since those will require more thinking and will take longer. Can you help us by identifying which decisions have significant trade-offs and which don't? You can find the solution at the end of the chapter.

Is this a significant trade-off?

☐ Yes	☐ No	Picking out what clothes to wear to work today
☐ Yes	☐ No	Choosing to deploy on the cloud or on premise
☐ Yes	☐ No	Selecting a user interface framework
☐ Yes	☐ No	Naming a variable in a class file
☐ Yes	☐ No	Choosing between vanilla and chocolate ice cream
☐ Yes	☐ No	Deciding which architecture style to use
☐ Yes	☐ No	Choosing between REST and messaging
☐ Yes	☐ No	Using full data or only keys for the message payload
☐ Yes	☐ No	Selecting an XML parsing library

☐ **Yes**	☐ **No**	**Deciding whether or not to break apart a service**
☐ **Yes**	☐ **No**	**Choosing between atomic or distributed transactions**
☐ **Yes**	☐ **No**	**Deciding whether or not to go out to dinner tonight**

Putting it all together

Now it's time to put **all three** of these factors to use. Let's figure out whether a decision is more about architecture or more about design, which will tell us who should be ultimately responsible for the decision.

Let's say you decide to use asynchronous messaging between the `Order Placement` service and the `Inventory Management` service to increase the system's responsiveness when customers place orders. After all, why should the customer have to wait for the business to adjust and process inventory? Let's see if we can determine where in the spectrum this decision lies.

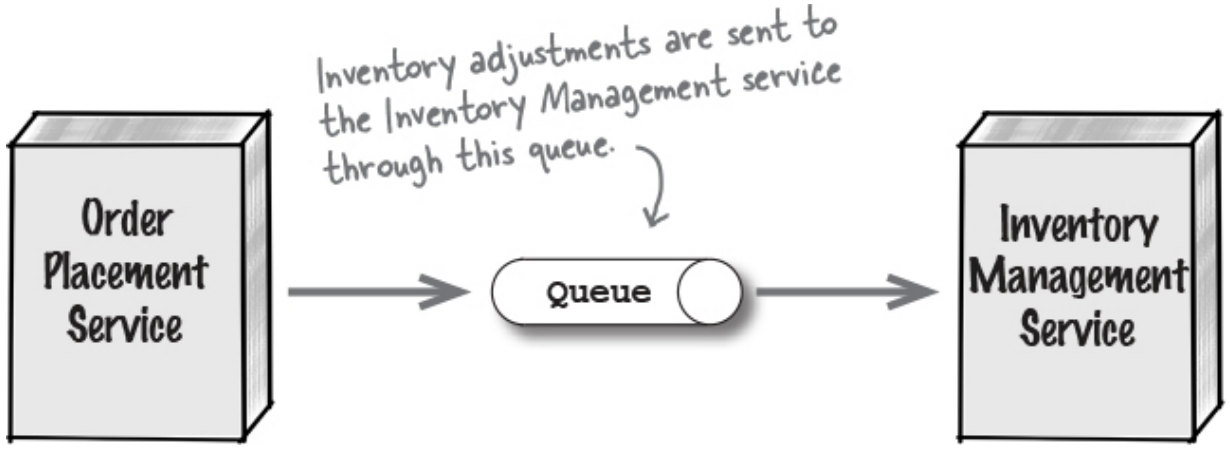


Figure 1-61.

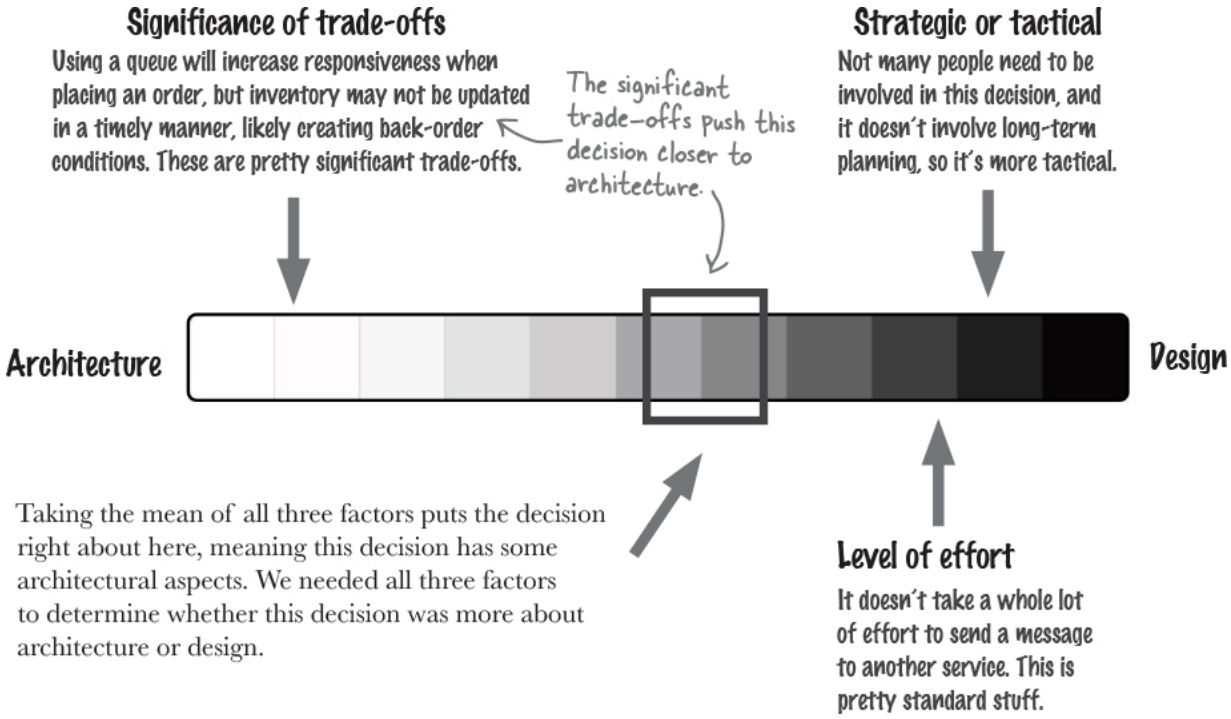


Figure 1-62.

You made it!

Congratulations—you made it through the first part of your journey to understanding software architecture. But before you roll up your sleeves to dig into further chapters, here's a little true-and-false quiz for you to test your knowledge so far. For each of the statements below, circle whether it is true or false. You can find the solution in "True or False"..

TRUE OR FALSE

True	False	Design is like the structure of a house (walls, roof, layout, and so on), and software architecture is like the furniture and decoration.
True	False	Most decisions are purely about architecture or design. Very few exist along a spectrum between architecture and design.
True	False	The more strategic your decision, the more it's about architecture; the more tactical, the more it's about design.
True	False	The more effort it takes to implement or change your decision, the more it's about design; the less effort, the more it's about architecture.
True	False	*Trade-offs* are the pros and cons of a given decision or task. The more significant the trade-offs become, the more it's about architecture. True False

BULLET POINTS

- Software architecture is less about appearance and more about structure, whereas design is more about appearance and less about structure.
- You need to use four dimensions to understand and describe software architecture: architectural characteristics, architectural decisions, logical components, and architectural style.
- *Architectural characteristics* form the foundational aspects of software architecture. You must know which architecture characteristics are most important to your specific system, so you can analyze trade-offs and make the right architectural decisions.
- *Architectural decisions* serve as guideposts to help development teams understand the constraints and conditions of the architecture.
- The *logical components* of a software architecture solution make up the building blocks of the system. They represent things the system does and are implemented through class files or source code.
- Like house styles, there are many different *architectural styles* you can use. Each style supports a specific set of architectural characteristics, so it's important to make sure you select the right one (or combination of them) for your system.
- It's important to know if a decision is about architecture or design because that helps determine who should be responsible for the

decision and how important it is.

Software Architecture Crossword

Figure 1-63.

Congratulations. You made it through the first chapter and learned about what software architecture is (and isn't). Now, why don't you try architecting the solution to this crossword?

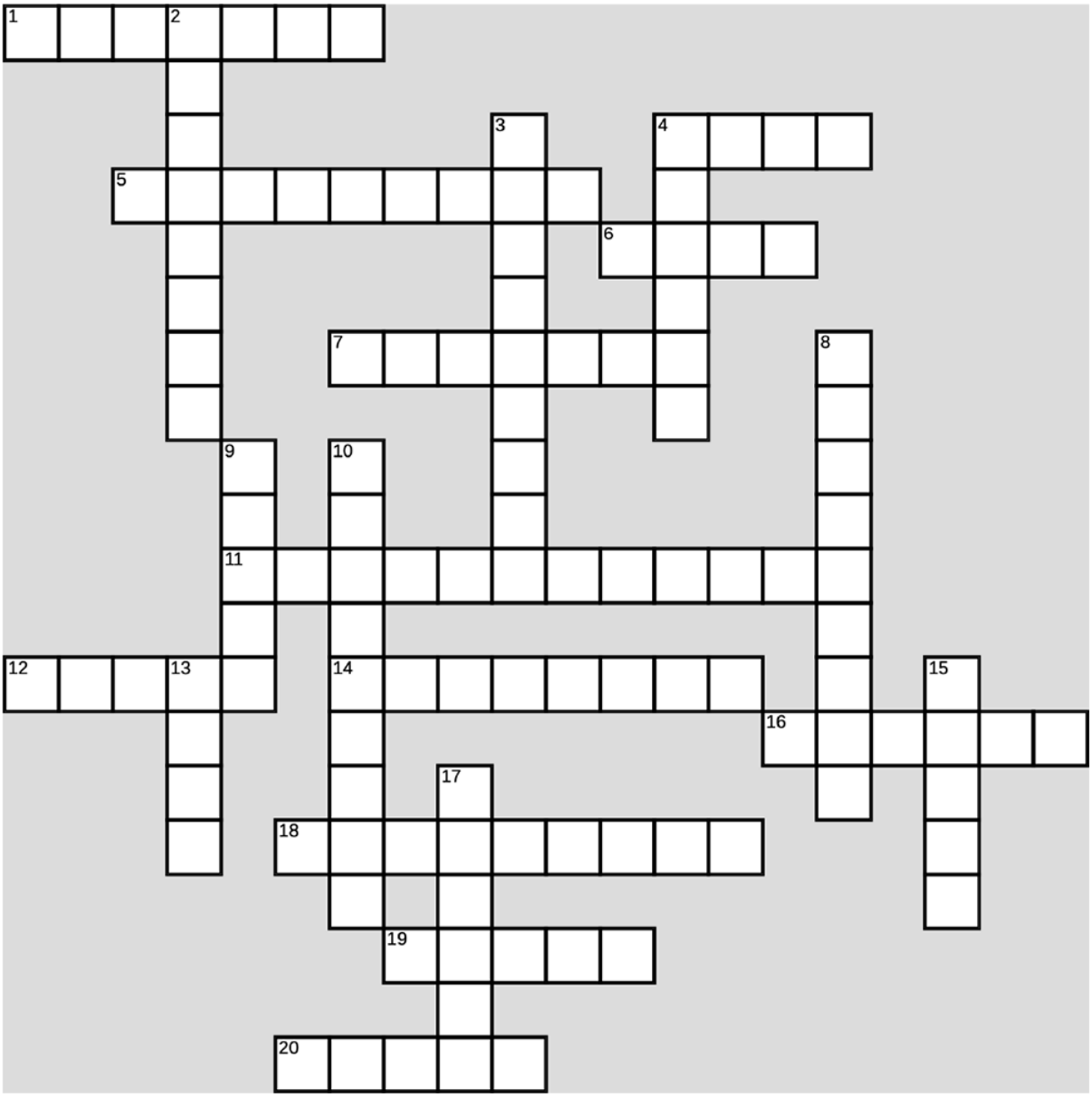


Figure 1-64.

Across

1 Architectural characteristics are sometimes called this

4 Tradeoffs are about the ___ and cons

5 The number of rooms in your home is part of its ___

6 How many dimensions it takes to describe a software architecture

7 A system's ___ components are its building blocks

11 You're learning about software ___

12 The overall shape of a house or a system, like Victorian or microservices

14 Architecture and design exist on a ___

16 If something takes a lot of ___ to implement, it's probably architectural

18 A website's user ___ involves lots of design decisions

Down

2 Decisions can be strategic or ___

3 You might want to become one after reading this book

4 Strategic decisions typically involve a lot of these

8 You analyze these when making an architectural decision

9 An architectural style determines the system's overall ___

10 You'll make lots of architectural ___

13 Architectural decisions are usually ___ -term

15 Building this can be a great metaphor

17 It's important to know whether a decision is about architecture or this

19 You can’t step in the same one twice

20 ___ -driven is an architectural style

From “Exercise”

EXERCISE SOLUTION

Figure 1-65.

Gardening is often used as another metaphor for describing software architecture. Using the space below, can you describe how a garden might relate to software architecture? You can see what we came up with at the end of this chapter.

NOTE

The overall layout of a garden can be compared to the architectural style, whereas each grouping of like plants (either by type or color) can represent the architectural components. Individual plants within a group represent the class files implementing those components.

NOTE

Gardens are influenced by weather in the same way a software architecture is influenced by changes in technology, platforms, the deployment environment, and so on.

From "Sharpen your pencil"

SHARPEN YOUR PENCIL SOLUTION

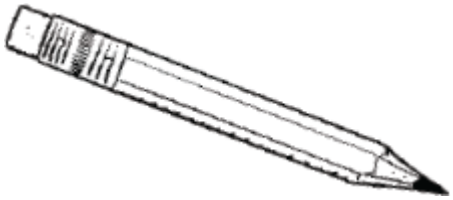


Figure 1-66.

What features of your home can you list that are *structural* and related to its *architecture*? You can find our thoughts at the end of this chapter.

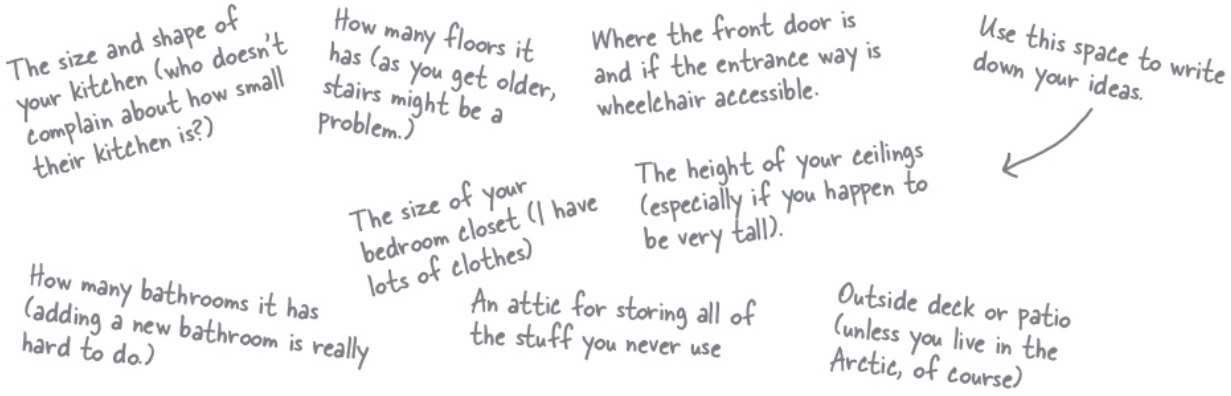


Figure 1-67.

From "Exercise"

EXERCISE SOLUTION

Figure 1-68.

Check the things you think might be considered architectural characteristics—something that the *structure* of the software system supports. You can find the solution at the end of the chapter.

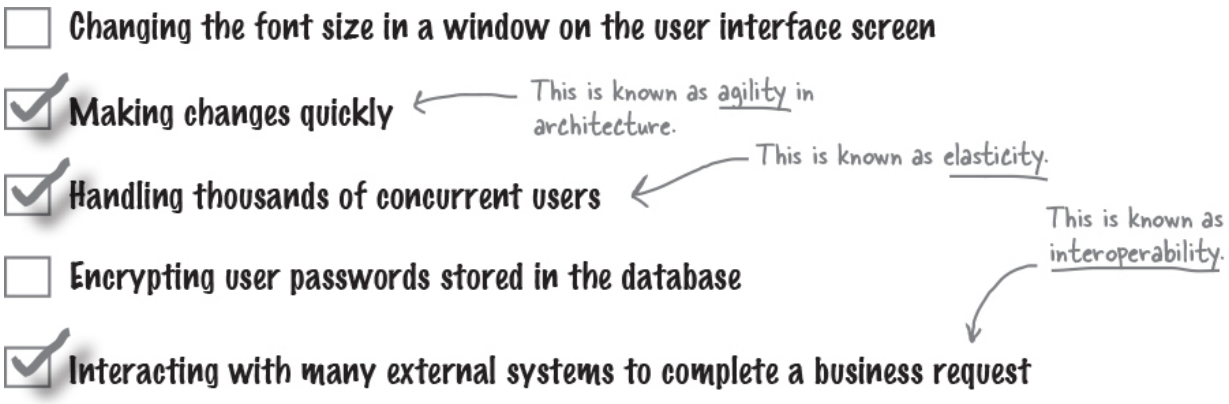


Figure 1-69.

From "Exercise"

EXERCISE SOLUTION

Figure 1-70.

Check all of the things that should be included in a diagram from an **architecture perspective**. You can find the solution at the end of the chapter.

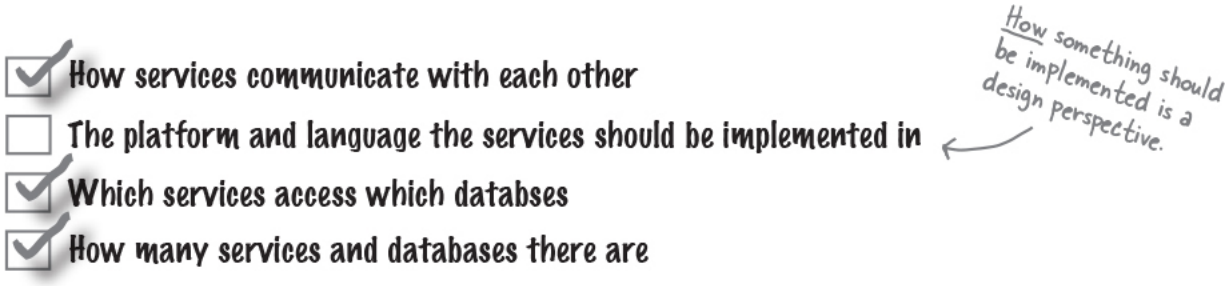


Figure 1-71.

From "Who Does What?"

WHO DOES WHAT? SOLUTION

Here's your chance to see how much you already know about many common architectural characteristics. Can you match up the architectural characteristic on the left with the definition on the right? You'll notice there are more definitions than characteristics, so be careful—not all of the definitions have matches. You can find the solution in "Sharpen your pencil Solution".

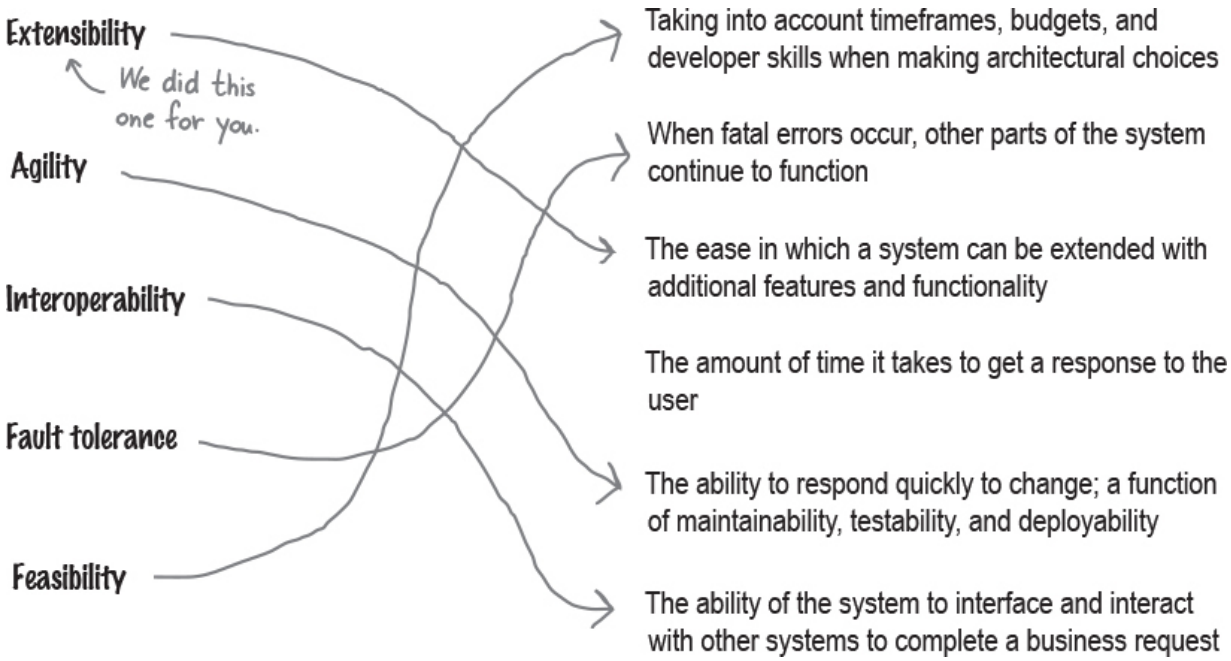


Figure 1-72.

From "Sharpen your pencil"

SHARPEN YOUR PENCIL SOLUTION

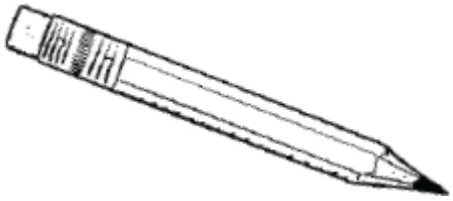


Figure 1-73.

Circle all of the things that you think fall somewhere in the middle of the spectrum **between** architecture and design. You can find the solution in “Sharpen your pencil Solution”.

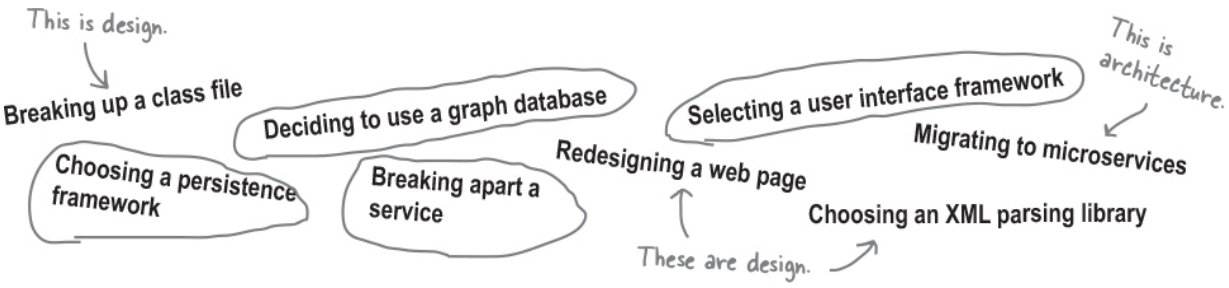


Figure 1-74.

From “BE the architect”

BE the architect solution

Your job is to play like the architect and identify as many architectural decisions you can in the diagram below. Draw a circle around any-

thing that you think might be an architectual decision and write what that decision might be. After you're done, look at the end of the chapter for the solution.

Figure 1-75.

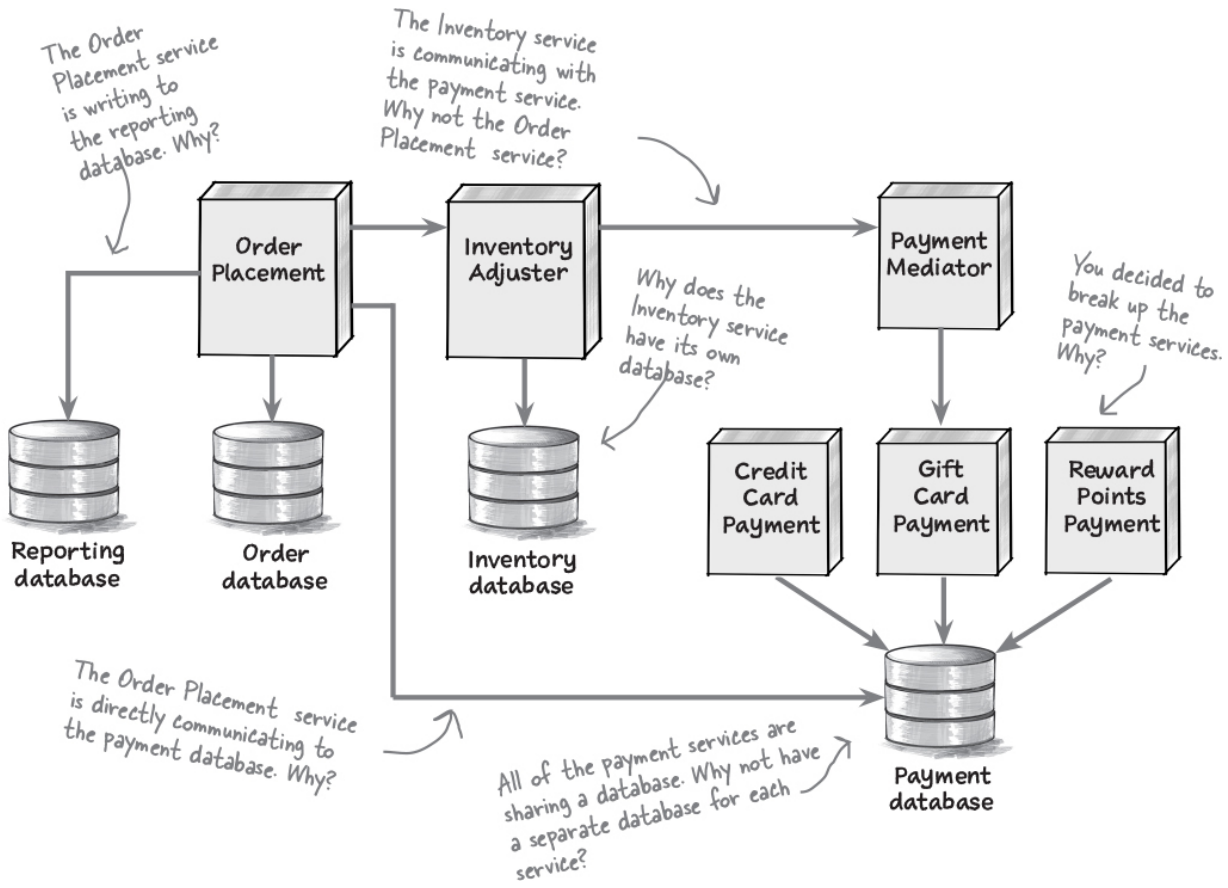


Figure 1-76.

From [“Sharpen your pencil”]

SHARPEN YOUR PENCIL SOLUTION

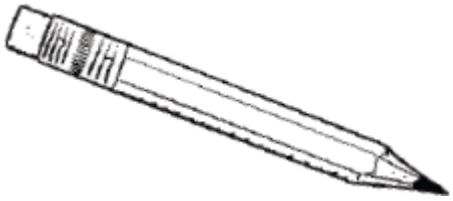


Figure 1-77.

You've just created the following two components for a new system called **BuyFromUs**, and your development team wants to start writing class files to implement them. Can you create the directory structure for them so they can start coding? Flip to the end of the chapter for our solution.

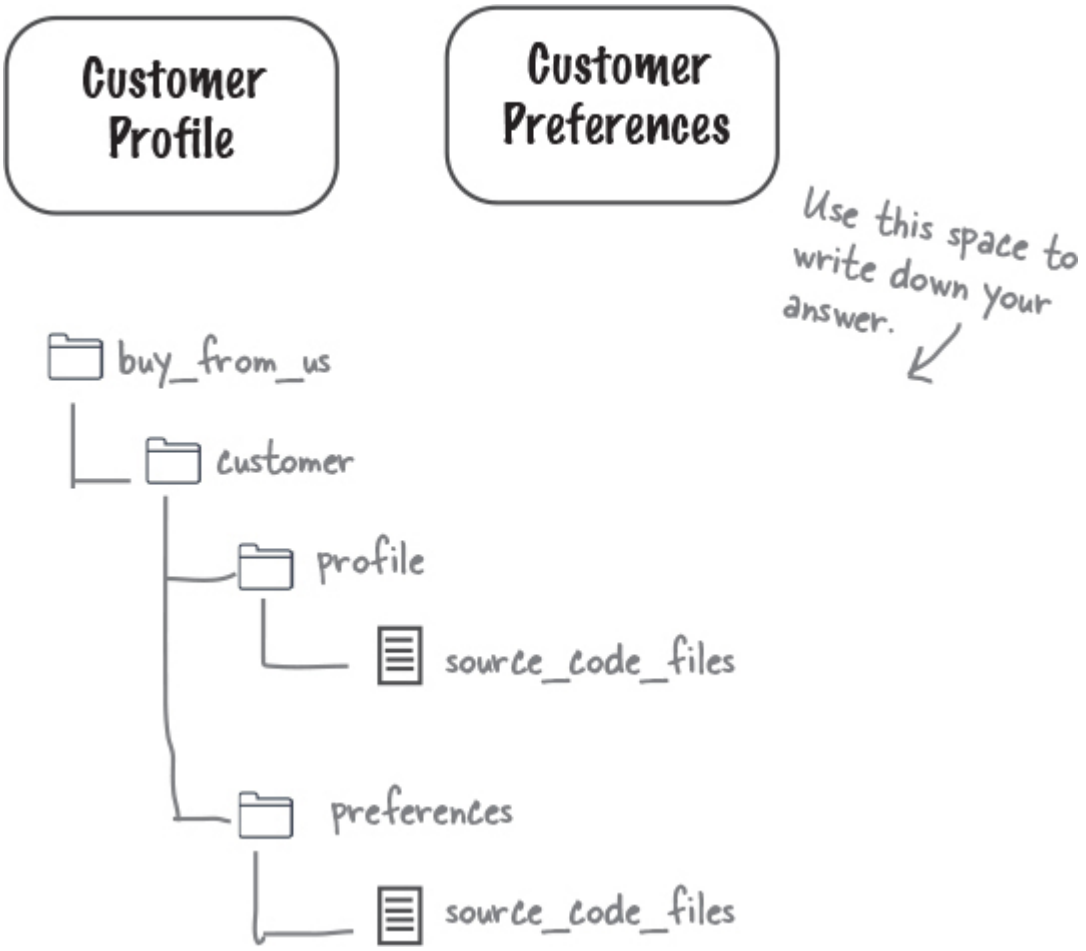


Figure 1-78.

From "Who Does What?"

WHO DOES WHAT? SOLUTION

We were trying to describe our architecture, but all the puzzle pieces got mixed up. Can you help us figure out which dimension does what by matching the statements on the left with the software architecture dimensions on the right? Be careful—some of the statements don't have a match because they are not related to *architecture*.

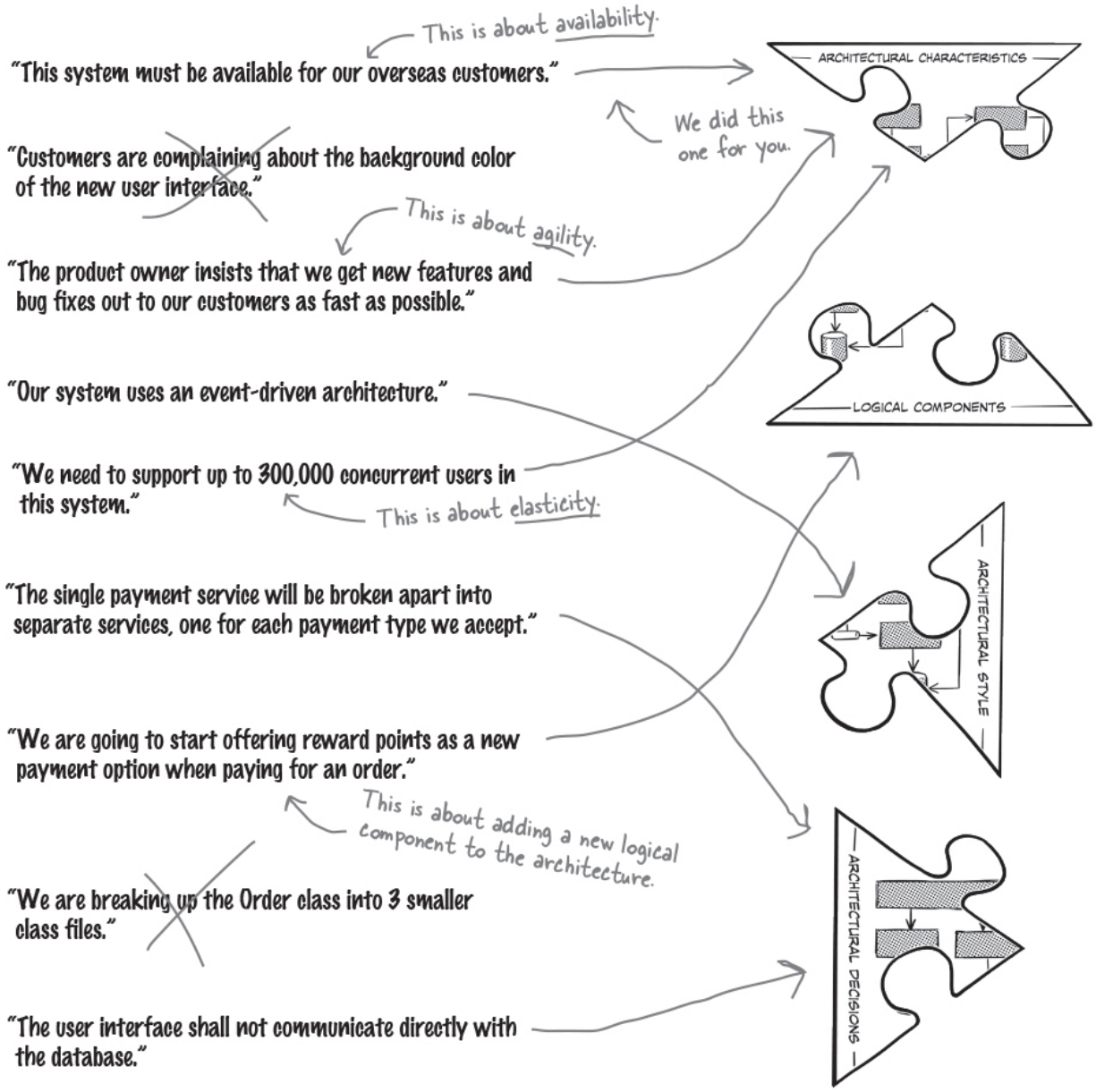


Figure 1-79.

From "Sharpen your pencil"

SHARPEN YOUR PENCIL SOLUTION

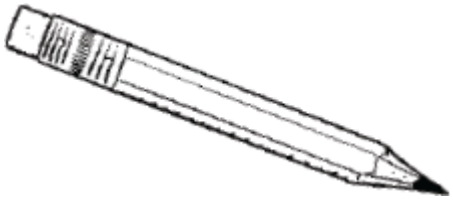


Figure 1-80.

Oh dear. We've lost all of our marbles and we need your help putting them back in the right spot. Using the three questions on the prior page as a guide, can you figure out which jar each marble should go in? You can find the solution at the end of the chapter.

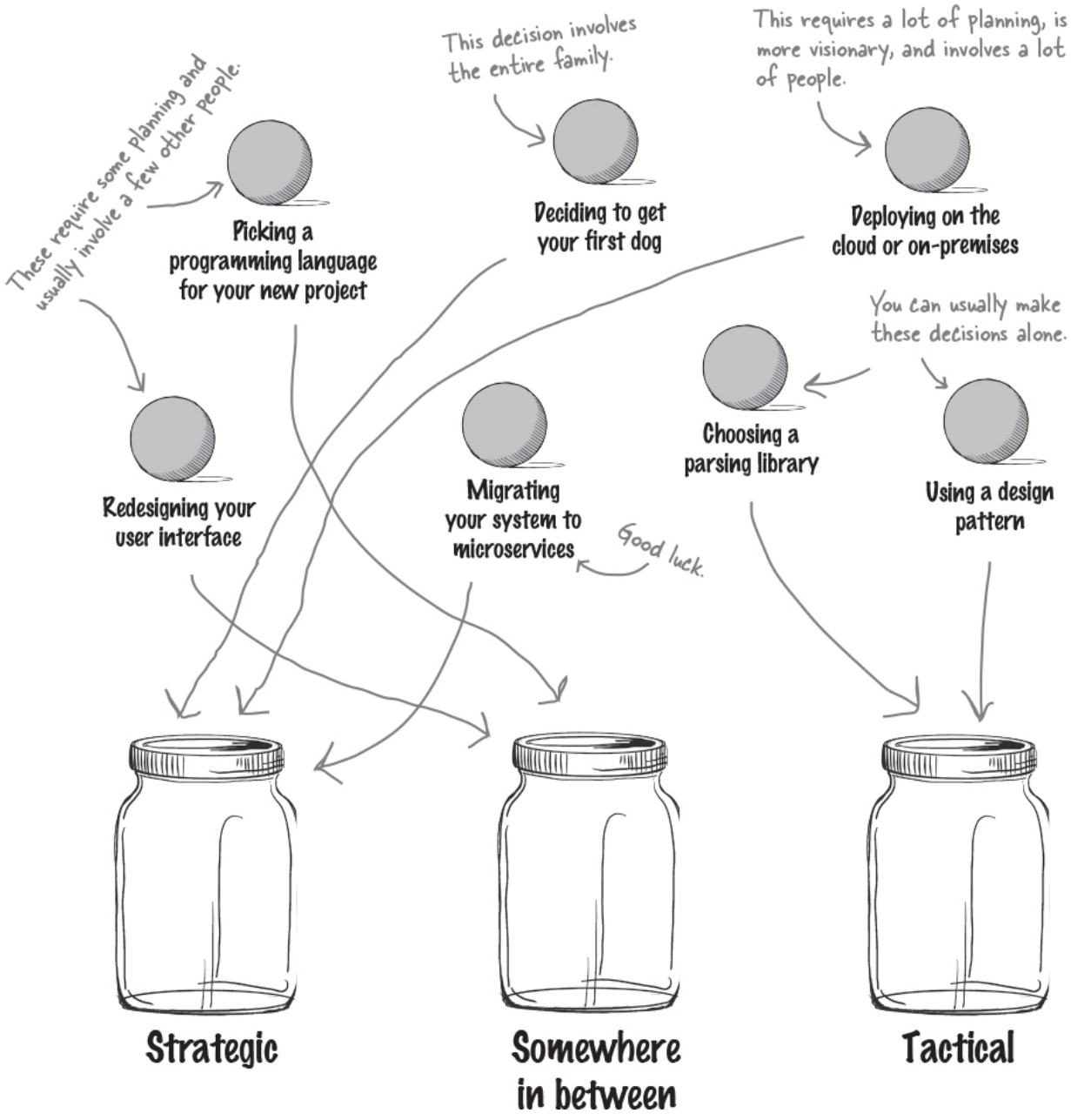


Figure 1-81.

From [“Code Magnets”]

Code Magnets Solution

Figure 1-82.

Oh no. We had all of these magnets from our to-do list arranged from high effort to low effort, and somehow they all fell on the floor and got all mixed up. Can you help us put these back in the right order based on the amount of effort it would take to make each change? You can find the solution at the chapter's end.

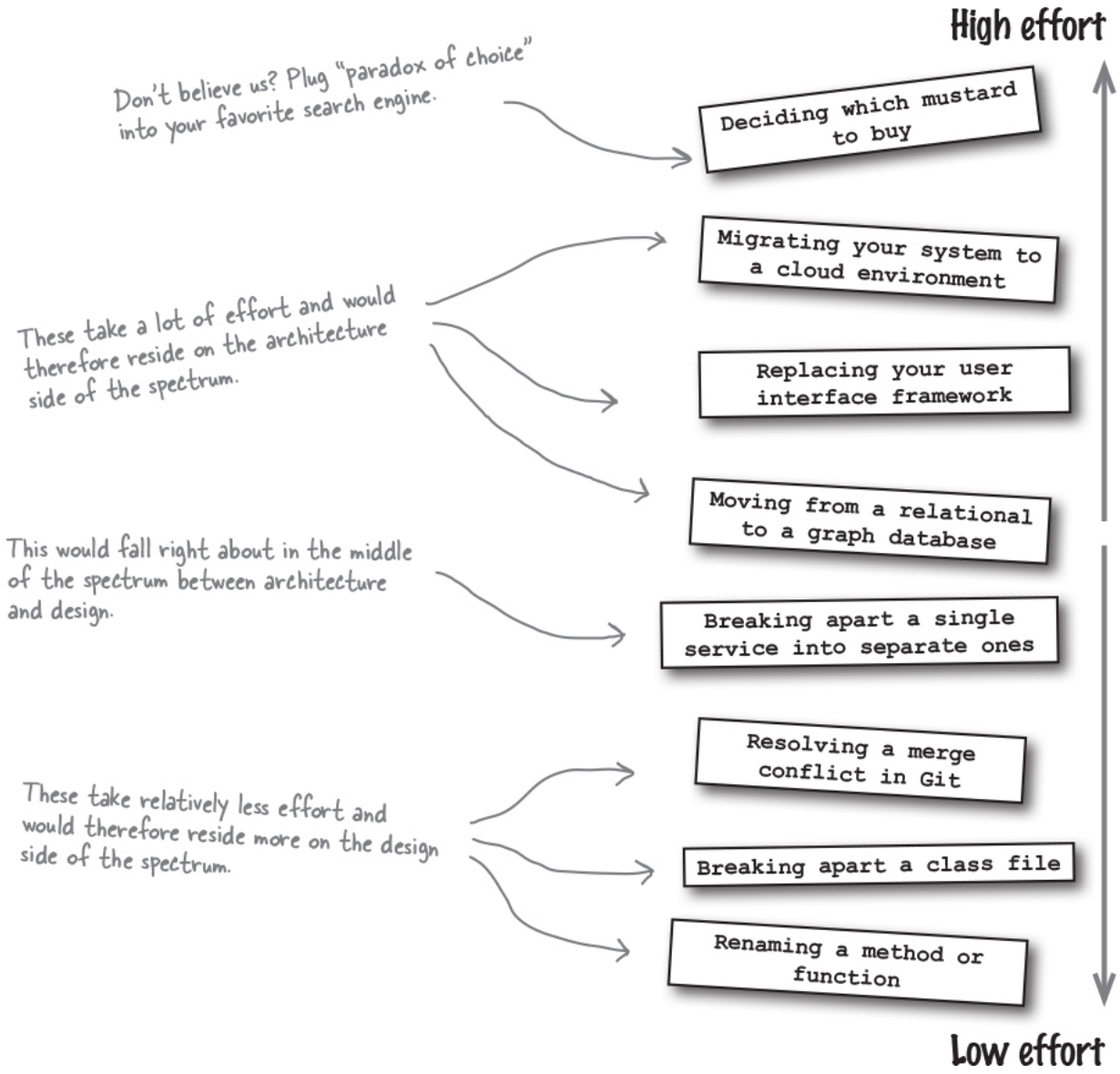

High effort
Don't believe us? Plug "paradox of choice" into your favorite search engine.
Deciding which mustard to buy
Migrating your system to a cloud environment
These take a lot of effort and would therefore reside on the architecture side of the spectrum.
Replacing your user interface framework
Moving from a relational to a graph database
This would fall right about in the middle of the spectrum between architecture and design.
Breaking apart a single service into separate ones
Resolving a merge conflict in Git
These take relatively less effort and would therefore reside more on the design side of the spectrum.
Breaking apart a class file
Renaming a method or function
Low effort

Figure 1-83.

EXERCISE SOLUTION

Figure 1-84.

From “Code Magnets”

Decisions, decisions, decisions. How can we ever tackle all of these decisions? One thing we think might help is to identify all of those decisions that have significant trade-offs since those will require more thinking and will take longer. Can you help us by identifying which decisions have significant trade-offs and which ones don’t? You can find the solution at the end of the chapter.

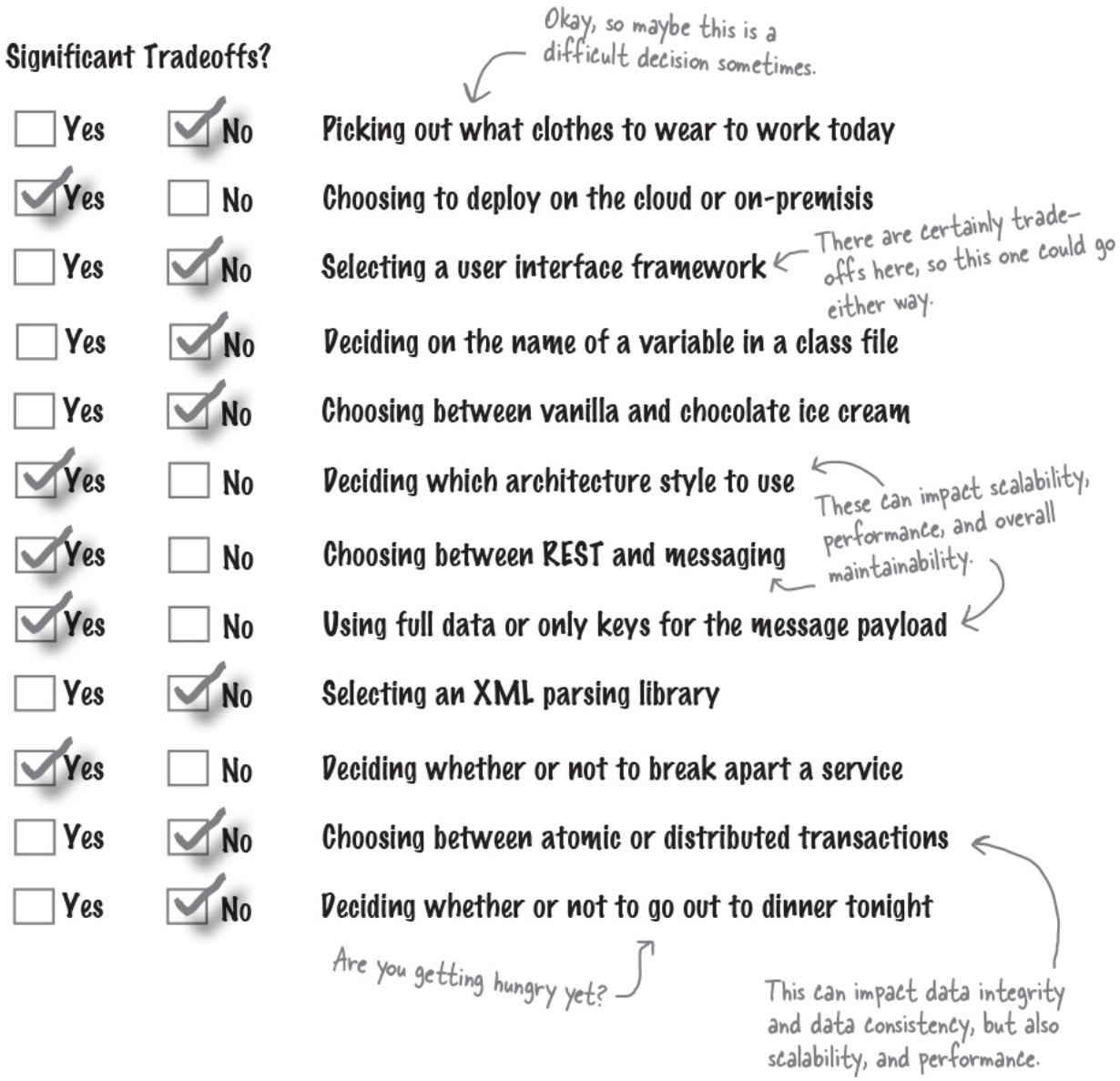


Figure 1-85.

From "Exercise"

EXERCISE SOLUTION

Figure 1-86.

Decisions, decisions, decisions. How can we ever tackle all of these decisions? One thing we think might help is to identify all of those decisions that have significant trade-offs since those will require more thinking and will take longer. Can you help us by identifying which decisions have significant trade-offs and which ones don't? You can find the solution in "Exercise Solution".

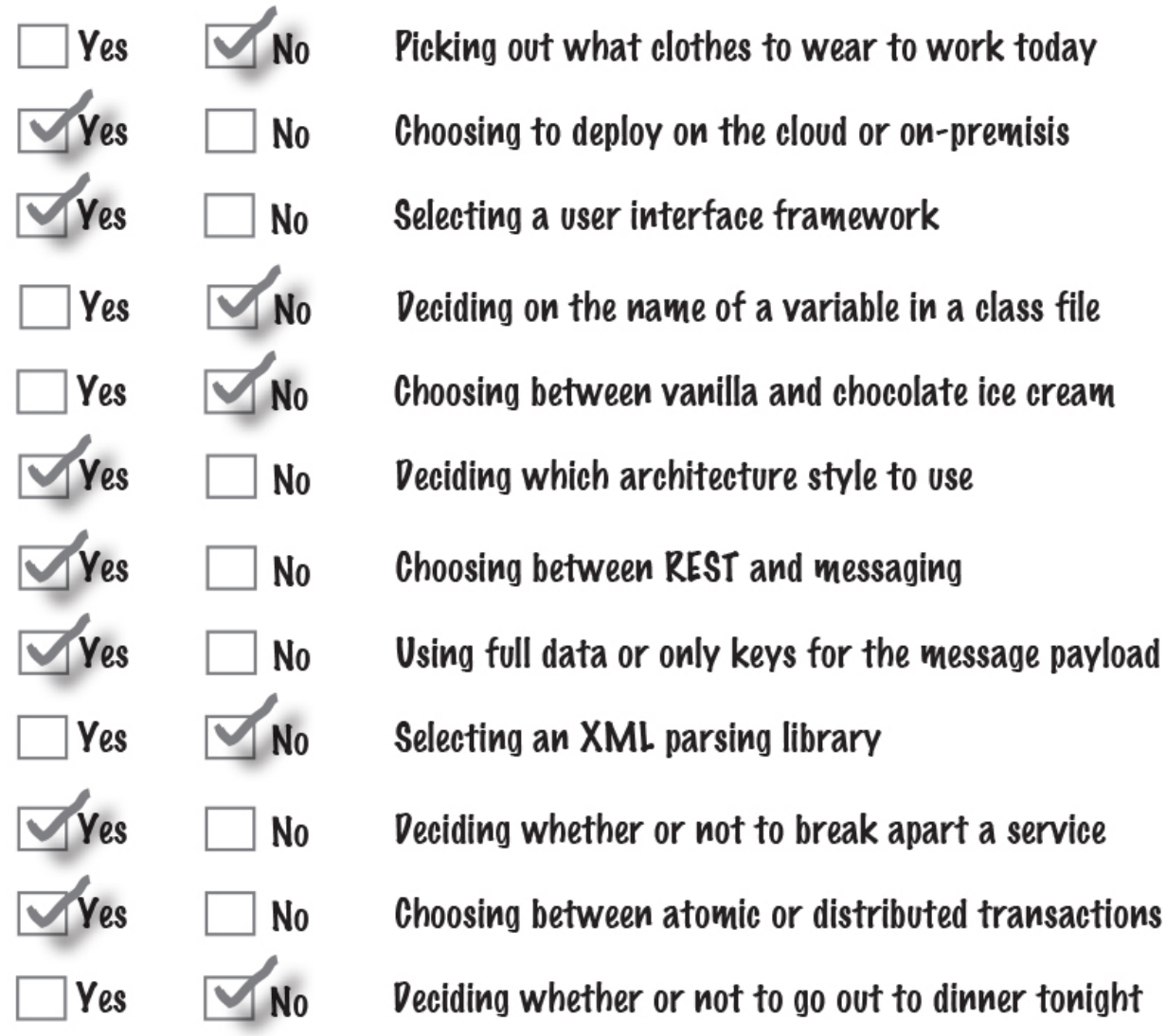


Figure 1-87.

From "True or False"

TRUE OR FALSE SOLUTION

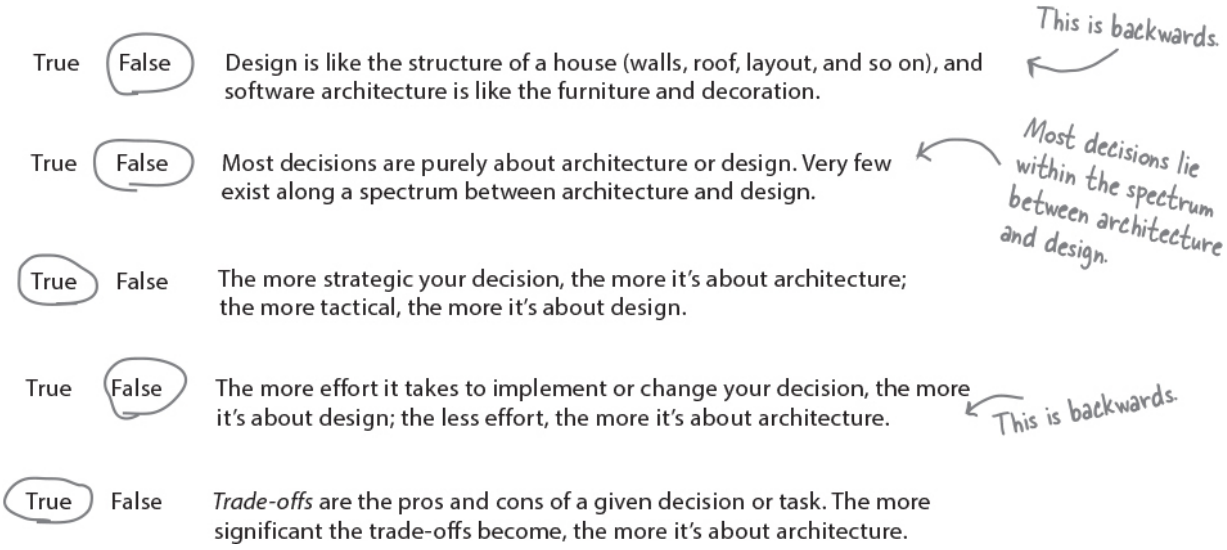


Figure 1-88.

Software Architecture Crossword Solution

Figure 1-89.

Congratulations. You made it through the first chapter and learned about what software architecture is (and isn't). Now, why don't you try

architecting the solution to this crossword?

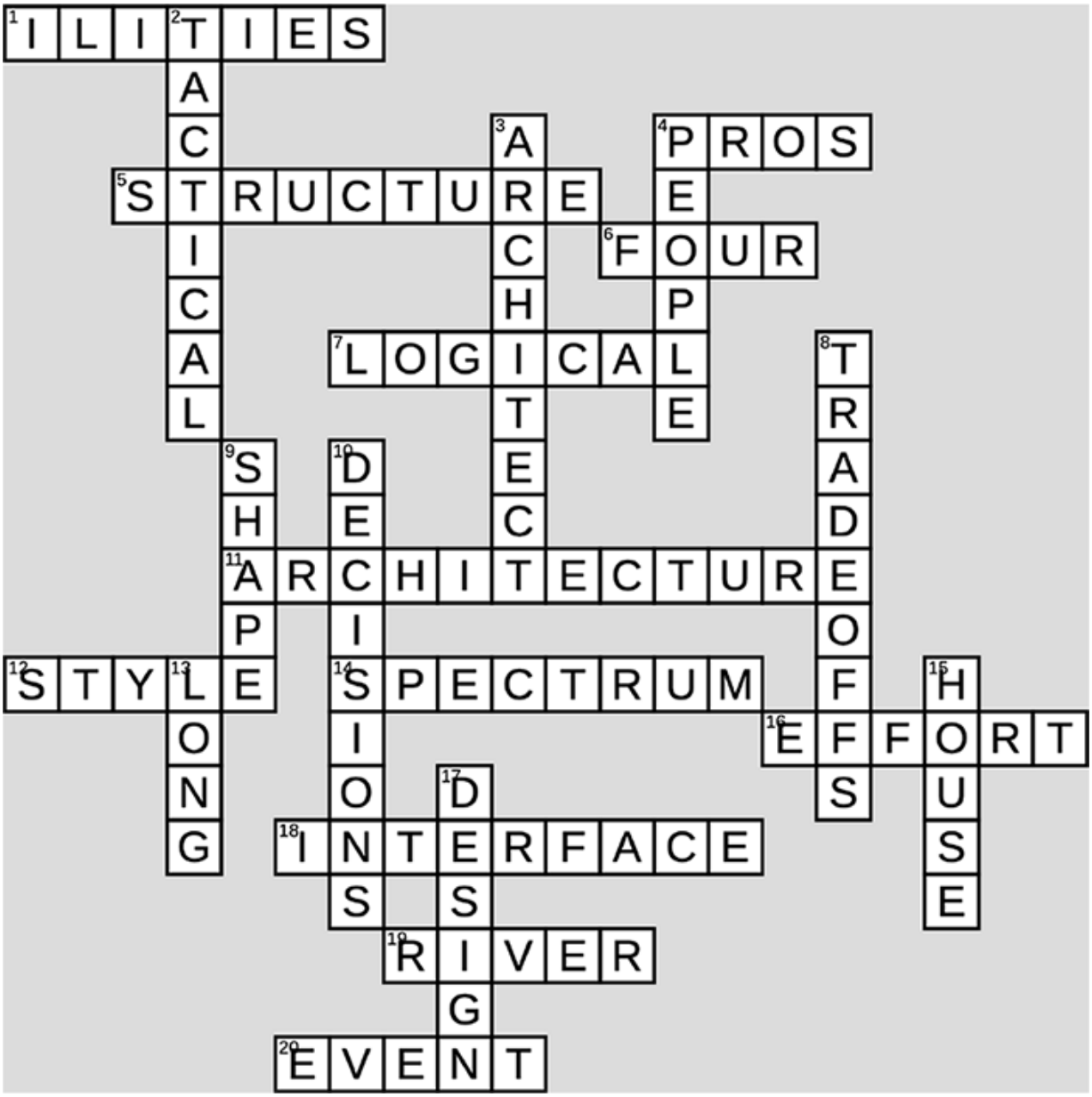


Figure 1-90.

Across

1 Architectural characteristics are sometimes called this

4 Tradeoffs are about the ___ and cons

5 The number of rooms in your home is part of its ___

6 How many dimensions it takes to describe a software architecture

7 A system's ___ components are its building blocks

11 You're learning about software ___

12 The overall shape of a house or a system, like Victorian or microservices

14 Architecture and design exist on a ___

16 If something takes a lot of ___ to implement, it's probably architectural

18 A website's user ___ involves lots of design decisions

Down

2 Decisions can be strategic or ___

3 You might want to become one after reading this book

4 Strategic decisions typically involve a lot of these

8 You analyze these when making an architectural decision

9 An architectural style determines the system's overall ___

10 You'll make lots of architectural ___

13 Architectural decisions are usually ___ -term

15 Building this can be a great metaphor

17 It's important to know whether a decision is about architecture or this

19 You can’t step in the same
one twice
20 ___ -driven is an architectural
style

Chapter 2. Everything's a Trade-off: *The Two Laws of Software Architecture*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor (*sgrey@oreilly.com*)

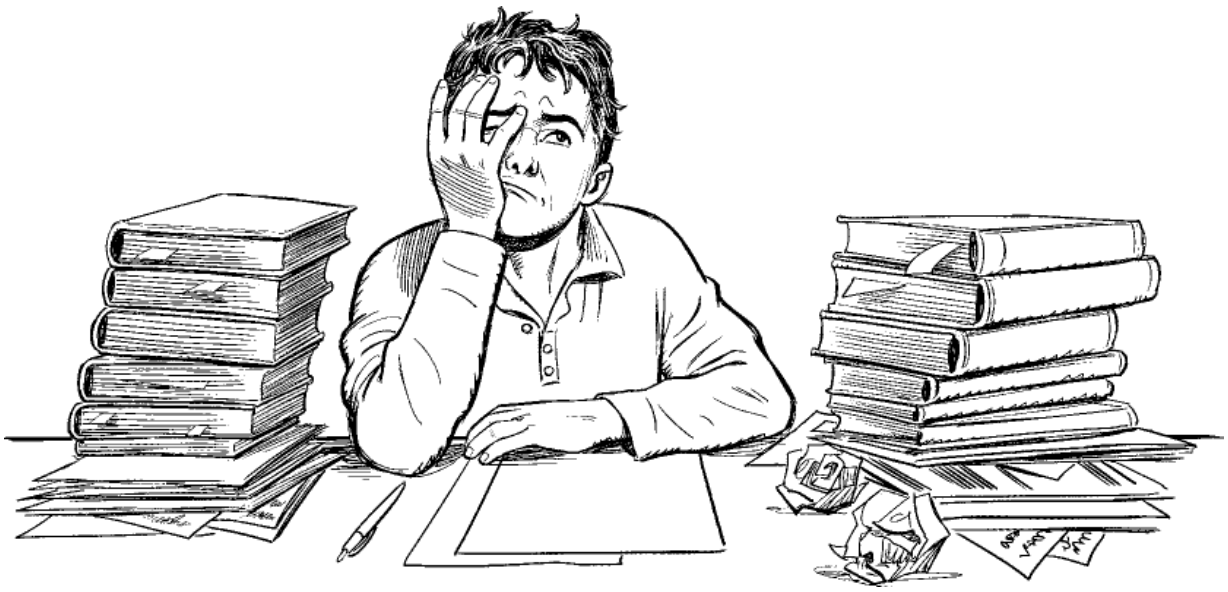


Figure 2-1.

What happens when there are no "best practices"? You see, the nice thing about having best practices is that you can simply use them to achieve a goal. They're called "best" (not "better" or "good") for a reason—you know they work, so why not just use them? But thing you'll quickly learn about software architecture is that it has no best practices. You'll have to analyze every situation carefully to make a decision, and you'll need to communicate not just the "what" of the decision, but the "why."

So, how **do** you navigate this new frontier? Fortunately, you have the laws of software architecture to guide you. This chapter shows you how to analyze trade-offs as you make decisions. We'll also show you how to create architectural decision records to capture the "hows"

and “whys” of decisions. By the end of this chapter, you’ll have the tools to navigate the uncertain territory that is software architecture.

It starts with a sneaker app

Archana works for Two Many Sneakers, a company with a very successful mobile app where shoe collectors (“sneakerheads”) can buy, sell, and trade collectible sneakers. With millions of shoes listed, customers can find the shoes they really want or upload photos to help sell the ones they don’t.

The app’s initial architecture was a single service:

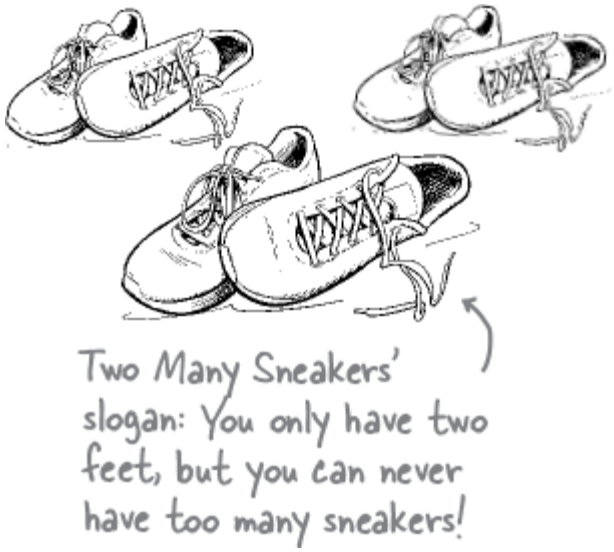


Figure 2-2.

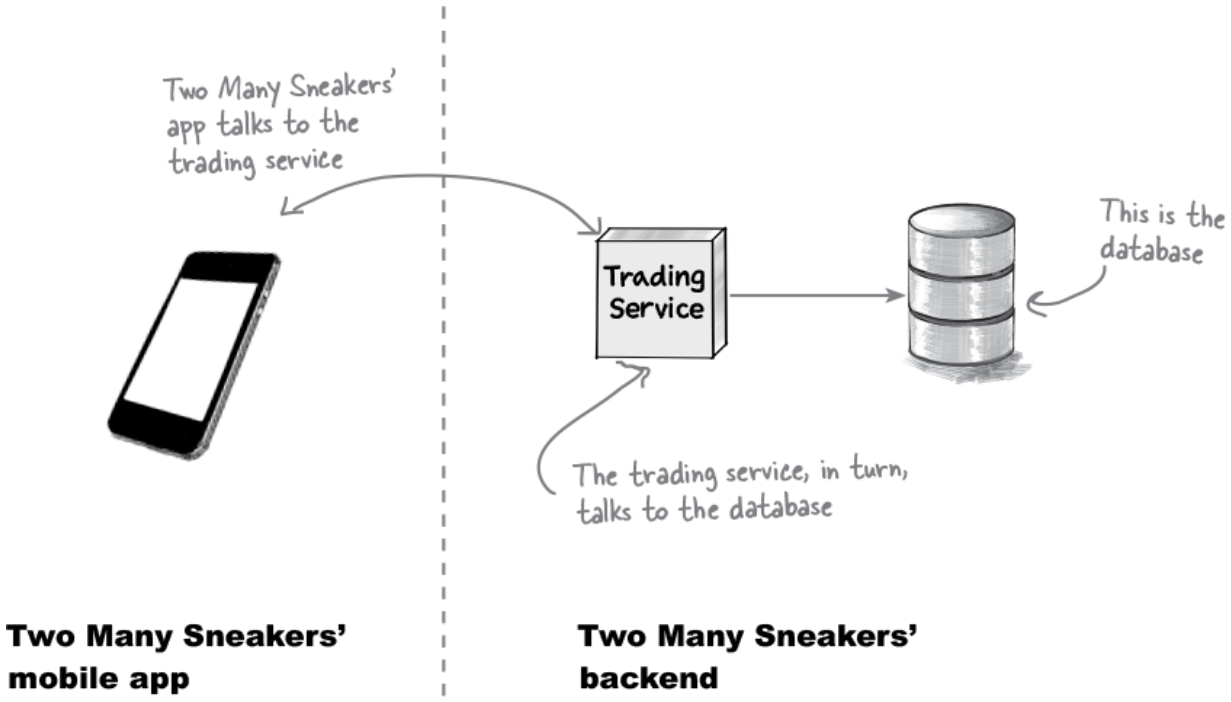


Figure 2-3.

The Two Many Sneakers app knows to talk to the trading service to fetch and update data (like a photo of a mint-condition pair of Nikes). The trading service, in turn, fetches data and updates the database.

Business is booming. Sneakerheads are always willing to change up their collections, and Two Many Sneakers' customer base has grown quickly. Now customers are demanding realtime notifications, so they'll know whenever someone lists a pristine pair of those Air Jordans they've been pining for.

Security is always a concern in online sales. Nobody wants knock-offs, and credit-card numbers need to be protected. To stay a few steps ahead of any scammers, Two Many Sneakers' management

team wants to prioritize improving the app's fraud detection capabilities. They plan to use data analytics to help detect fraud by spotting anomalies in user behavior and filtering out bots.

Work has already begun—all the team needs to do now is set up the trading service to notify the new notification and analytics services anytime something of interest happens in the app. Piece of cake! I'll just use messaging to inform the notification and analytics services every time a new pair of shoes is listed on the app.


Piece of cake! I'll just use messaging to inform the notification and analytics services every time a new pair of shoes is listed on the app.
Genius!
Archana

Figure 2-4.

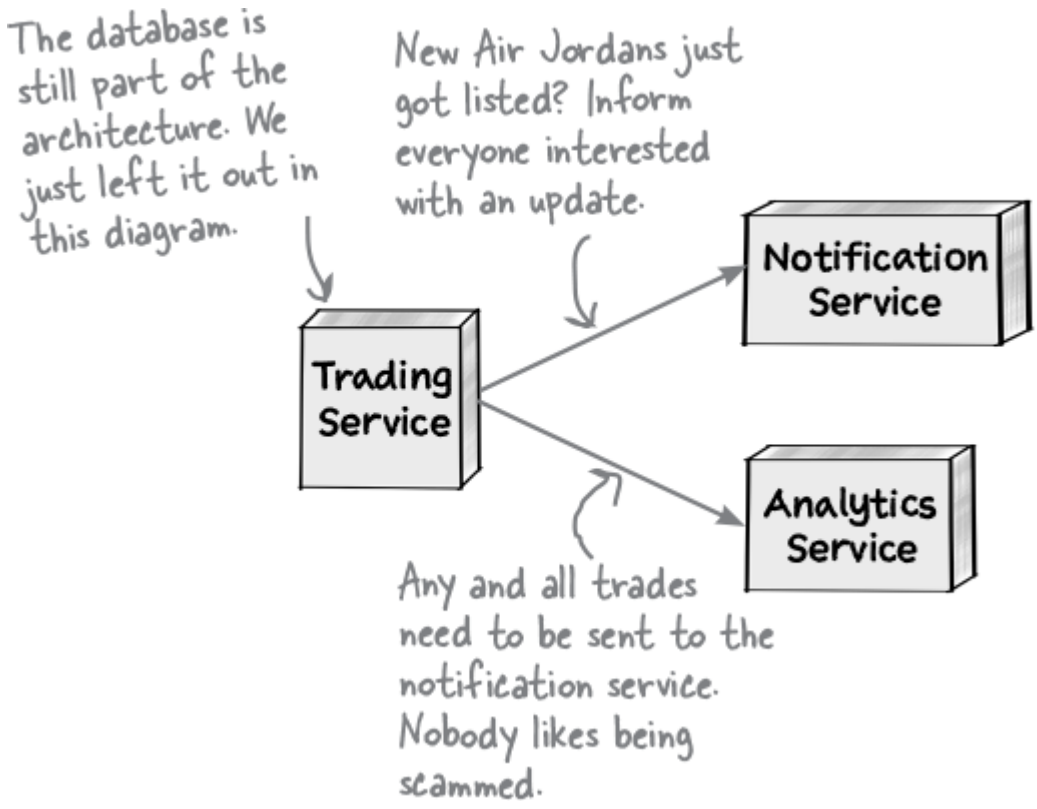


Figure 2-5.

What do we know so far?

We need to figure out how these services will communicate with one another. Let's recap what we know so far:

☑ *The current architecture is rather simple—the trading service talks to its own database, and that's that. We need the trading service to send information to the notification service and the analytics service.*

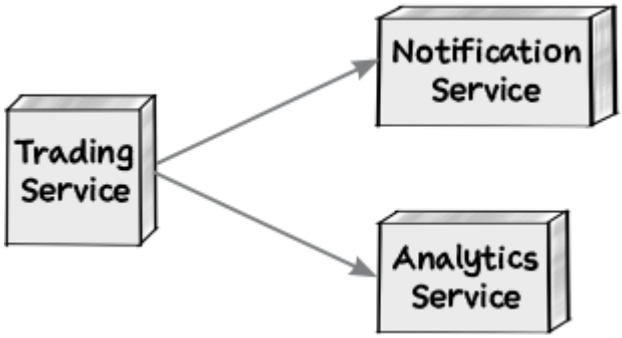


Figure 2-6.

☑ *Word in the office is that there's a chance that the Finance department will want updates from the trading service. In other words, whatever architecture we come up with will need to be* **extensible**.

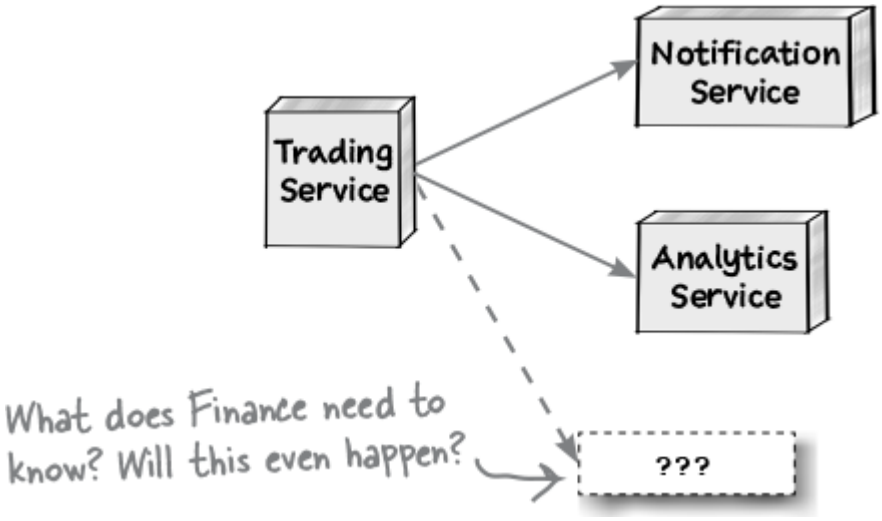


Figure 2-7.

☑ *We don't know what data to send the notification and analytics services—do the two services get the same kind of data, or wildly different data? And we don't know where things stand with the finance department, so that's another unknown.*

```
{
  "sellerId": 12345,
  "buyerId": 6789,
  "itemId": 1492092517,
  "price": "$125.00"
}
```

NOTE

What should this look like?

To be clear, there are some things we know and plenty that we don't. Welcome to the world of software architecture.

NOTE

As the system's architects, we need to identify its architectural characteristics. (You learned about those in Chapter 2.)

Speaking of architecture, we'll be done, say, next Thursday—right?

EXERCISE

Figure 2-8.

Which of the following architectural characteristics stand out as important for this particular problem? **Hint:** There are no right answers here, because there is a lot we don't know or aren't sure of yet. Take your best guess—we have provided our solutions at the end of this chapter. We'll get you started:

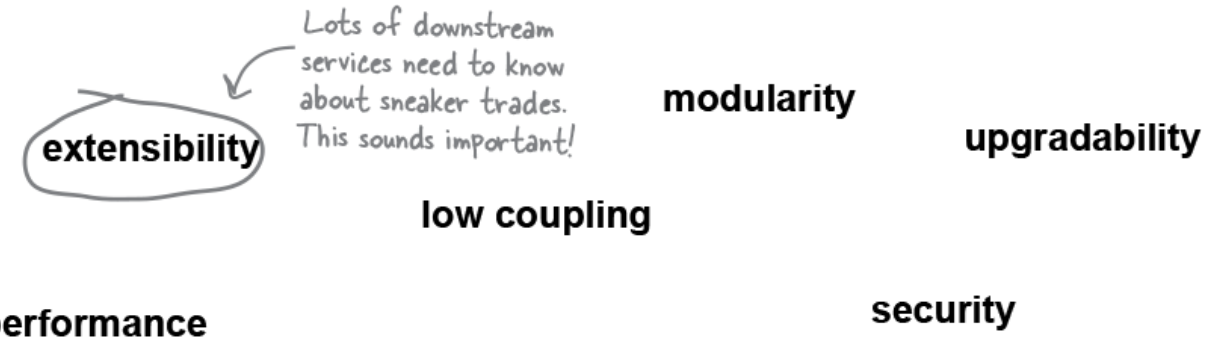


Figure 2-9.

BRAIN POWER

Figure 2-10.

All the characteristics in the previous exercise sound pretty good, right? Seriously, who'd say no to upgradability?

But for each one, ask yourself—is this characteristic critical to the project's success? Or is it a nice-to-have?

What's more, some characteristics conflict. A highly secure application with loads of encryption and secure connections probably won't be highly performant. Go back and see if any of your choices are at odds. If so, you can only pick one.

NOTE

Flashback to Chapter 2? You bet it is!

THERE ARE NO DUMB QUESTIONS

Q: Even this simple exercise seems to have a lot of moving parts. We know some things, we think we know some other things, and there's a lot that we certainly don't know. How do we go about thinking about architecture?

A: You're right. In almost all real-life scenarios, your list of architectural characteristics will probably have a healthy mix of "this is what we want" and "this is something we might want." Even your customers can't answer the question of what they will eventually want. (Wouldn't that be nice?) This is the "stuff you don't know you don't know," also known as the "unknown unknowns."

It's not unusual for an "unknown unknown" to rear its head midway through a project and derail even the best-laid plans. The solution? Embrace agility and its iterative nature. Realize that nothing, particularly software architecture, remains static. What worked today might prove to be the biggest hurdle to success tomorrow. That's the nature of software architecture: it constantly evolves and changes as you discover more about the problem and as your customers demand more of you.

Having the trading service communicate with downstream services

Our goal is to get the trading system to notify the reporting and analytics systems automatically. For now, let's assume we decide to use messaging. But that presents a dilemma—should our messaging use queues or topics?

NOTE

It's OK if you don't know much about messaging, queues, or topics. We'll tell you what you need to know.

Before we go further, let's make sure we're on the same page about the differences between queues and topics. Most messaging platforms offer two models for a *publisher* of a message (in this case, that's the trading service) to communicate with one or more *consumers* (the downstream services).

The first option is a *queue*, or a point-to-point communication protocol. Here, the publisher knows who is receiving the message. To reach multiple consumers, the publisher needs to send a message to

one queue for each consumer. If the trading app wants to use queues to tell the analytics service and the reporting service about trades, this is what the setup would look like:

NOTE

If it helps, think of queues as being like a group text—you pick everyone you want to inform, type your message, and hit "send."

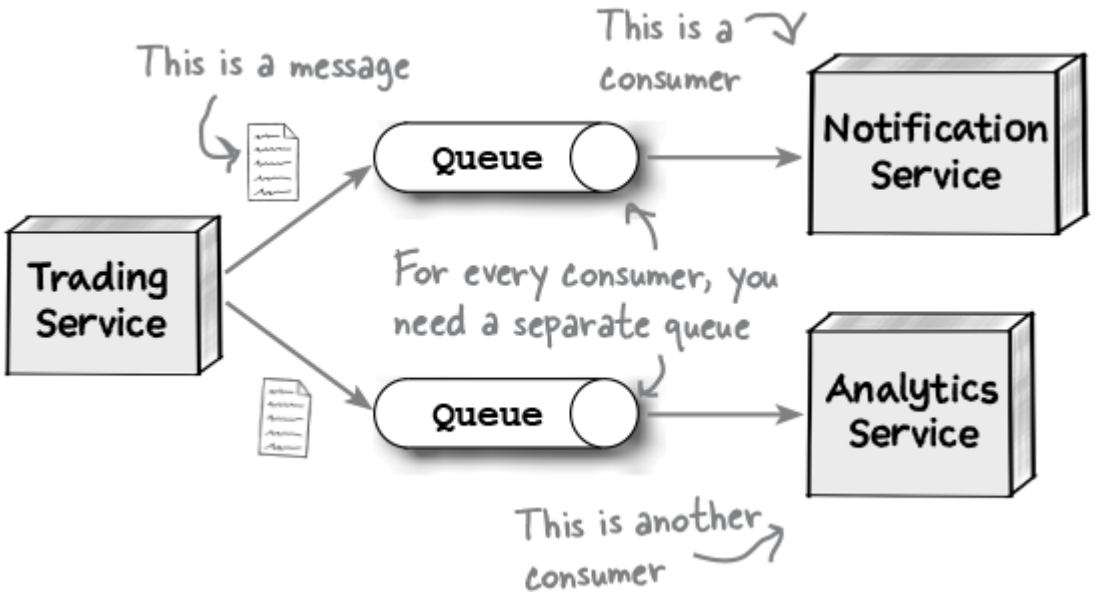


Figure 2-11.

When using the second option, *topics*, you are signing on for a *broadcasting* model. The publisher simply produces and sends a message. If another service downstream wants to hear from the publisher, it can subscribe to the topic to receive messages. The publisher doesn't know (or care) how many services are listening.

NOTE

Topics are similar to posting a picture on your go-to social networking site. Anyone following you will see that picture, since they've "subscribed" to your timeline.

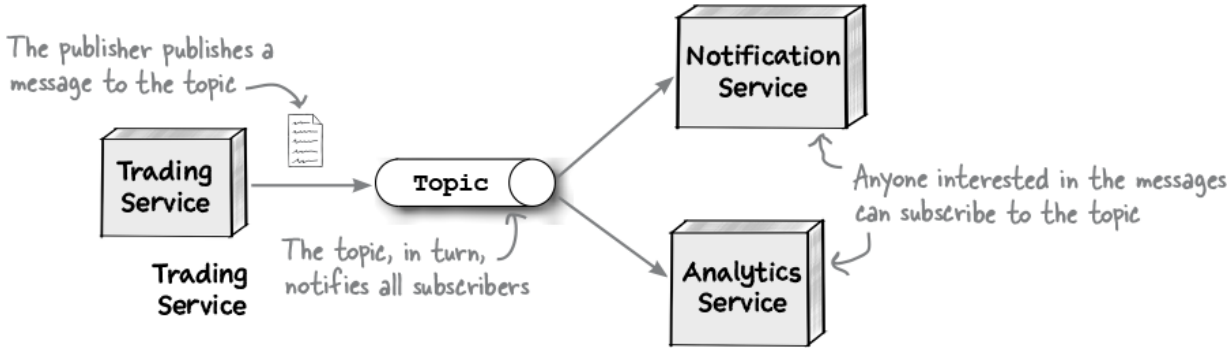


Figure 2-12.

Both options sound good—so how do we pick? Let's find out.

Analyzing trade-offs

You can't have your cake and eat it too. The world is full of compromises—we often optimize for one thing at the cost of another. Want to take and store lots of pictures on your phone? Either get more storage, which costs more, or compress them, which lowers the image quality.

Figure 2-13.

Software architecture is no different. Every choice you make involves significant compromises or, as we like to call them, **trade-offs**. So what exactly does this mean for you?

NOTE

If this sounds familiar, it should be! It was part of our discussion of significant versus less significant trade-offs in Chapter 1.

If you know which architectural characteristics are most important to your project, you can start thinking of solutions that will maximize some of those attributes. But if a solution lets you maximize one characteristic (or more), it will come at the cost of other characteristics. For example, a solution that allows for great scalability might also make deployability or reliability harder.

No matter what solution you come up with, it will come with trade-offs—upsides *and* downsides.

Your job is twofold—know the trade-offs associated with every solution you come up with, and then pick the solution that best serves the most important architectural characteristics.

NOTE

Rich Hickey, creator of the Clojure programming language, once said, "Programmers know the benefits of everything and the trade-offs of nothing." We'd like to add: "Architects need to understand both."

> **You can't have it all. You'll have to decide which architectural characteristics are most important, and choose the solution that best allows for those characteristics.**

Trade-off analysis: Queue edition

Trade-off analysis isn't just about finding the benefits of a particular approach. It's also about seeking out the negatives to get the full picture. Let's look at each option in turn, starting with queues.

With queues, for every service that the trading service needs to notify, we need a separate queue. If the notification service and the analytics service need different information, we can send different messages along each queue. The trading service is keenly aware of every system to which it communicates, which makes it harder for an-

other (potentially rogue) service to “listen in.” (That’s useful if security is high on our priority list, right?) Oh, and since each queue is independent, we can monitor them separately and even scale them independently if needed.

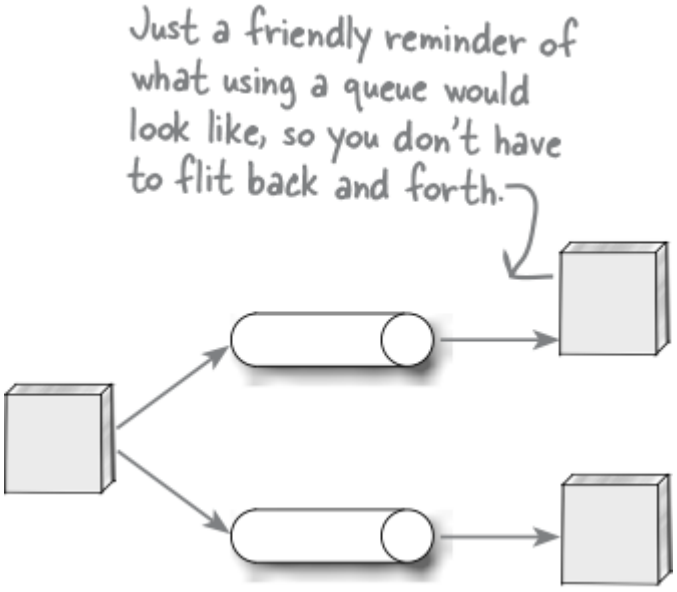


Figure 2-14.

The trading service is tightly coupled to its consumers—it knows exactly how many there are. But we’re not sure if we’ll need to sending messages to the compliance service, too. If that happens, we’ll have to rework the trading service to start sending messages to a third queue. In short, if we choose queues, we’re giving up on extensibility.

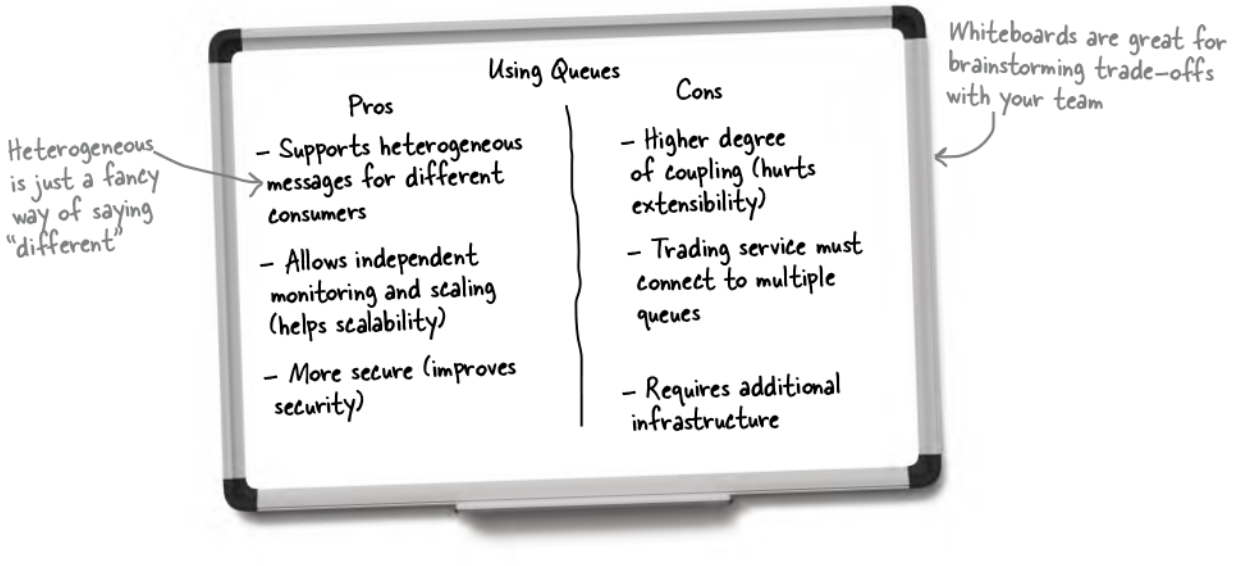


Figure 2-15.

See what we mean when we say "trade-off analysis"?

Trade-off analysis: Topic edition

What about using topics? Well, the upside is clear—the trading service only delivers messages to a topic, and anyone interested in listening for a message from the trading service simply subscribes to that topic. Compliance wants in? They can simply subscribe: no need to make any changes to the trading service. Low coupling for the win.

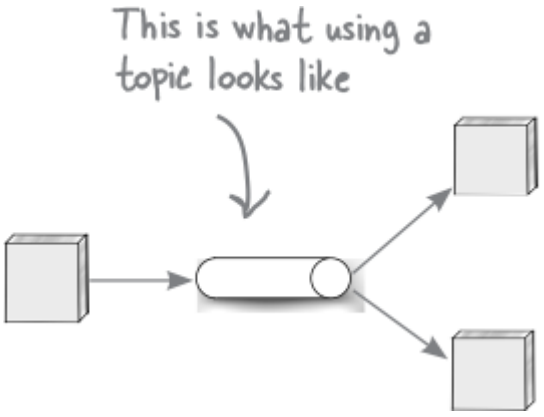


Figure 2-16.

But topics have a few downsides, too. For one thing, you can't customize the message for any particular service—it's a one-size-fits-all, take-it-or-leave-it proposition. Scaling, too, is one-size-fits-all, since you have only one thing to scale. And anyone can subscribe to the topic without the trading service knowing—which, in some circumstances, is a potential security risk.

Back to the whiteboard!

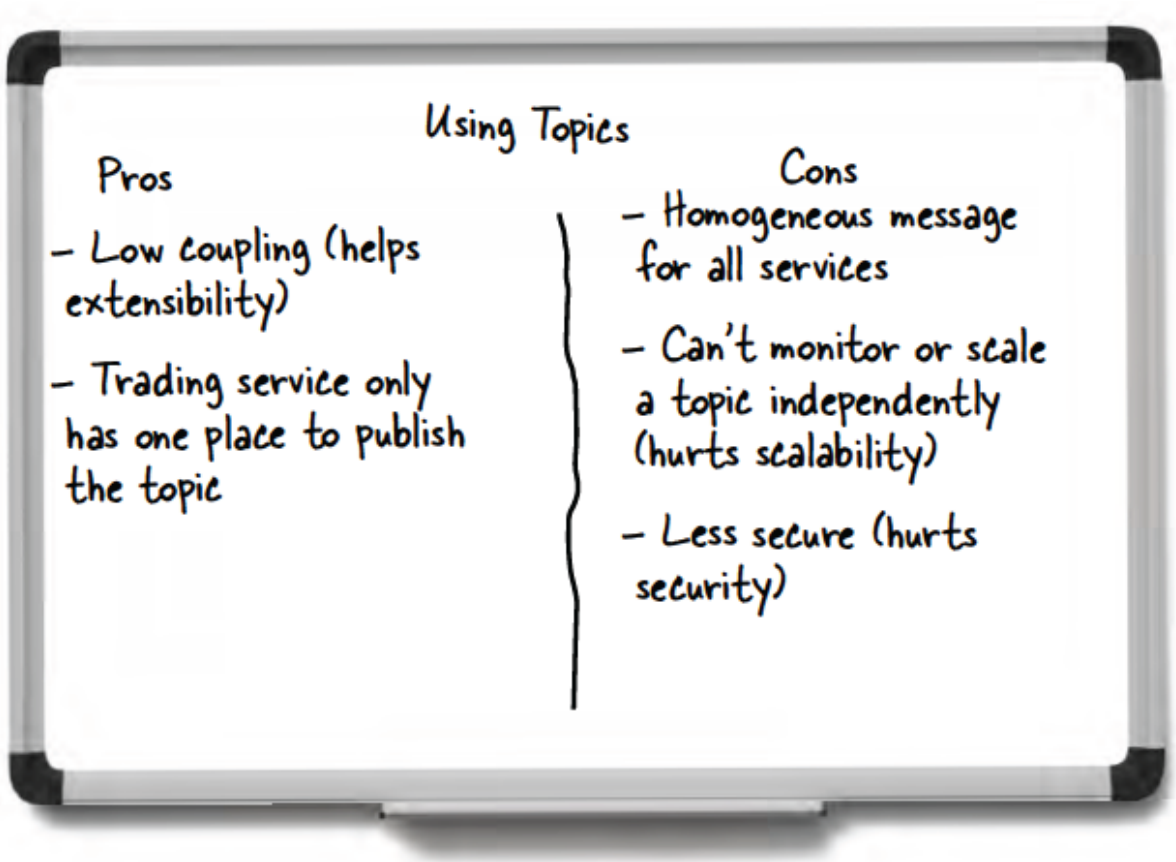

Using Topics
Pros
- Low coupling (helps extensibility)
- Trading service only has one place to publish the topic
Cons
- Homogeneous message for all services
- Can't monitor or scale a topic independently (hurts scalability)
- Less secure (hurts security)

Figure 2-17.

SHARPEN YOUR PENCIL

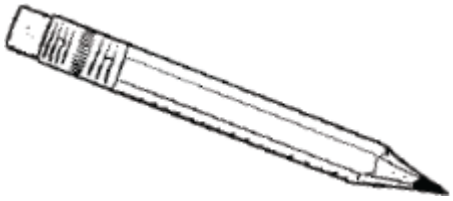


Figure 2-18.

Spend a few minutes comparing the results of our trade-off analysis. Notice how both options support some characteristics but trade off on others? Now we're going to present you with some requirements—see if you can decide if you'd pick queues or topics. Then check our answers at the back of this chapter.

Requirements	
"Security is important to us"	**Queues** / **Topics**
"Different downstream services need different kinds of information"	**Queues** / **Topics**
"We'll be adding other downstream services in the future"	**Queues** / **Topics**

The first law of software architecture

Queue or topic? Enough with the suspense already. The answer is—**it depends**!

What's important to the business? If security is paramount, we should probably go with queues. Two Many Sneakers is growing by leaps and bounds and has loads of other services interested in its sneaker trades, so extensibility is its biggest priority. That means we should pick the topic option.

Time is also a factor: if we need to get to market quickly, we might pick a simpler architecture (simplicity) over one that offers high availability. (Having an application that guarantees three "nines" of uptime only matters if you have customers, right?)

The key takeaway is that in software architecture, you'll always be balancing trade-offs. That leads us to **the First Law of Software Architecture**.


it depends
One of your authors often sports this T-shirt in public. (If you get any printed, please send us one medium and two extra-large!)

Figure 2-19.

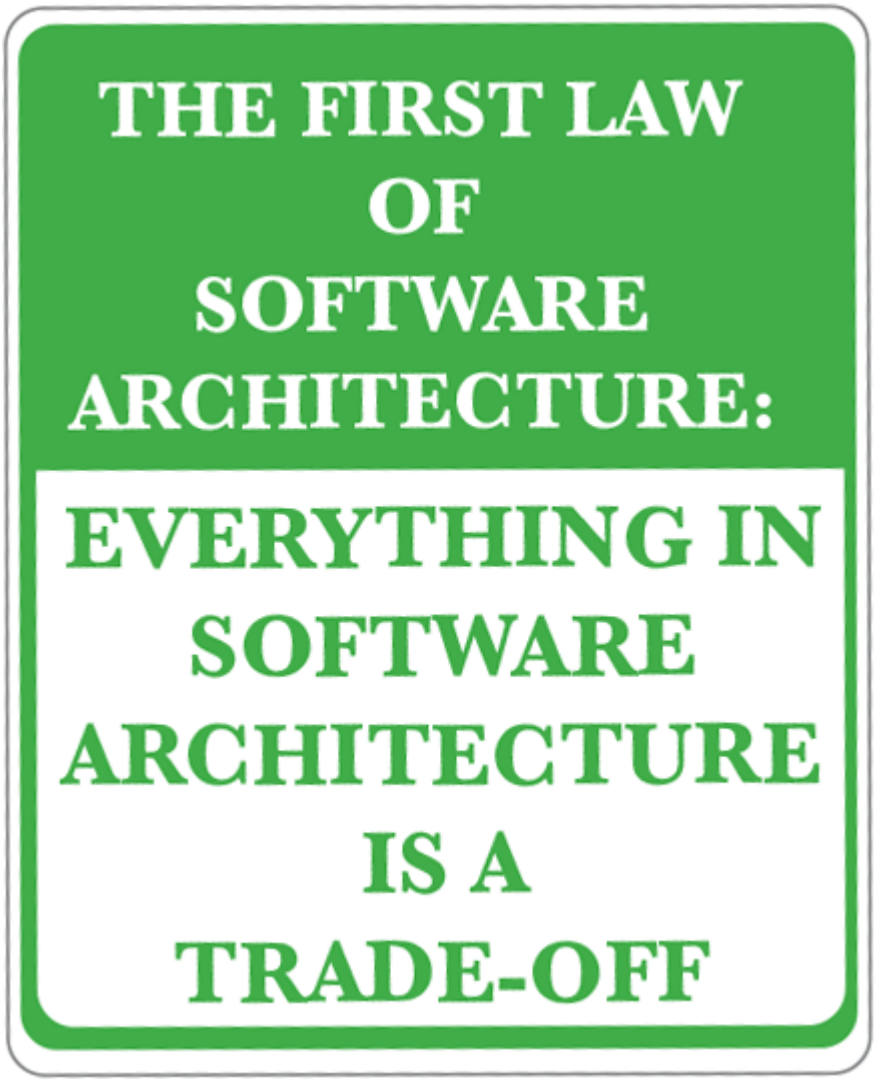


Figure 2-20.

In software architecture, nice, clean lines and “best practices” are rare. Every choice you make will involve many factors—often conflicting ones. The First Law is an important lesson, so take it to heart. Write it down on a sticky note and put it on your monitor. Get a backwards tattoo of it on your forehead so you’ll see it in the mirror! Whatever it takes.

NOTE

If you find a decision in software architecture that doesn't have a trade-off, you haven't looked at it hard enough.

SHARPEN YOUR PENCIL

Figure 2-21.

This time, we'd like you to do some trade-off analysis on your own. We chose messaging as the communication protocol between our trading service and its consumers. Messaging is asynchronous. Choosing between asynchronous and synchronous forms of communication (like REST and RPC) comes with its own set of trade-offs! We've given you two whiteboards, one for each form of communication, and we've listed a bunch of "-ilities." We'd like you to consider how each architecture characteristic would work in both contexts. Is this characteristic a pro or a con (or neither) in synchronous communications? What about in asynchronous communications? Place each "-ility" in the appropriate column. **Not all of them apply to this decision**. We put the first pro on the whiteboard for you. When you're done, you can see our answers at the end of the chapter.

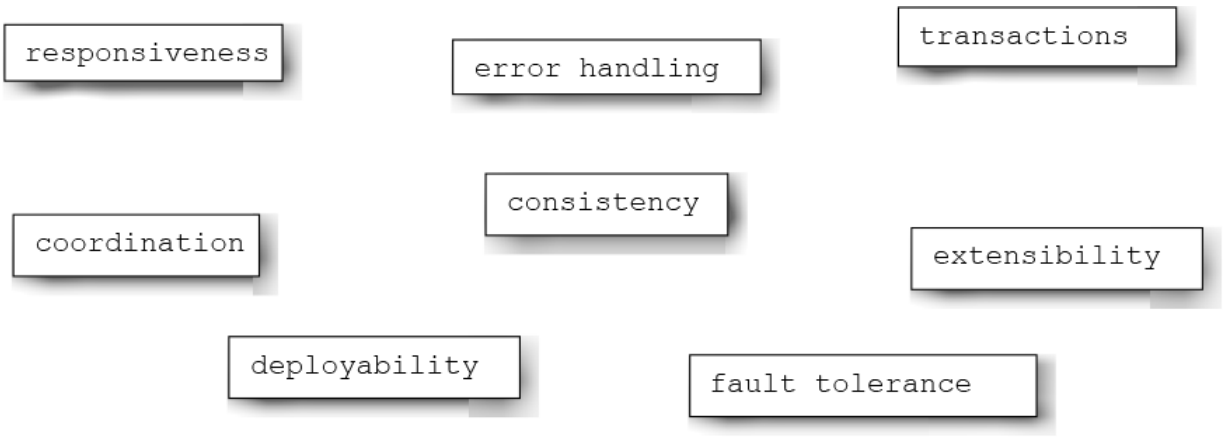


Figure 2-22.

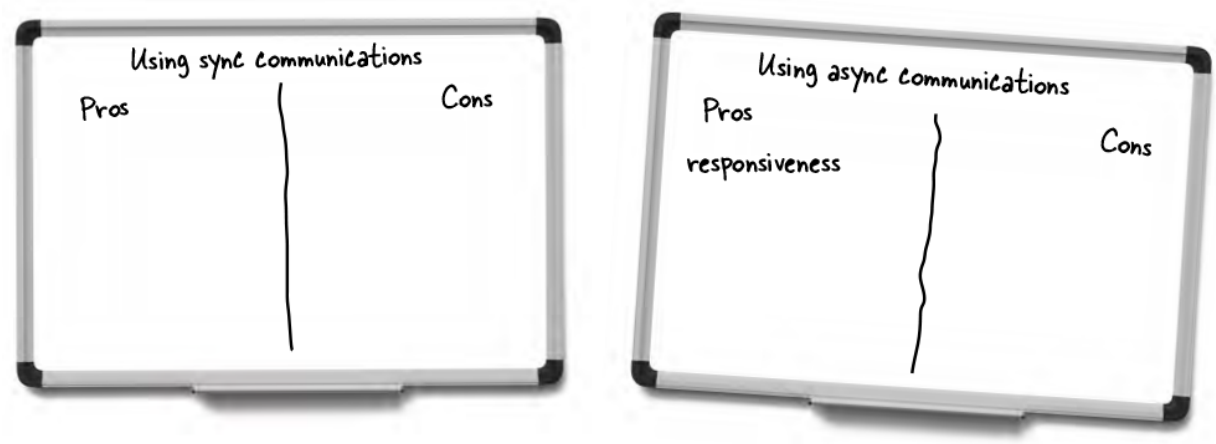


Figure 2-23.

THERE ARE NO DUMB QUESTIONS

Q: I've heard of the Architecture Tradeoff Analysis Method (ATAM). Is that what you're talking about?

A: ATAM is a popular method of trade-off analysis. With ATAM, you start by considering the business drivers, the "-ilities," and the proposed architecture, which you present to the stakeholders. Then, as a group, you run through a bunch of scenarios to produce a "validated architecture." While ATAM offers a good approach, we believe it comes with certain limitions—one being that it assumes the architecture is static and doesn't change.

Rather than focusing on the process of ATAM, we prefer to focus on results. The objective of any trade-off analysis should be to arrive at an architecture that best serves your needs. You'll probably go through the process several times as you discover more and more about the problem and come up with different scenarios.

Another popular approach is the Cost-Benefit Analysis Method (CBAM). In contrast to ATAM, CBAM focuses on the cost of achieving a particular "-ility."

We recommend you look at both methods and perhaps consider combining them—ATAM can help with trade-off analysis, while CBAM can help you get the best return on investment (ROI).

Just remember—the process is not as important as the goal, which is to arrive at an architecture that satisfies the business's needs.

It always comes back to trade-offs

Some people always pick a particular technique, approach, or tool regardless of the problem at hand. Often they choose something they've had a lot of success with in the past. Sometimes they have what we affectionately call "shiny object syndrome," where they think that some new technology or method will solve all their problems.

Regardless of past achievements or future promises, just remember—for every upside, there's a downside. The only questions you need to answer are: Will the upsides help you implement a successful application? And can you live with the downsides?

Whenever someone sings the praises of a certain approach, your response should be: "What are the trade-offs?"

NOTE

To be clear, we aren't saying you shouldn't use new tools and techniques. That's progress, right? Just don't forget to consider the trade-offs as you decide.

Making an architectural decision

Debating the pros and cons with your team in front of a whiteboard is fun and all, but at some point, you *must* make an **architectural** decision.

We mentioned architectural decisions in Chapter 1, but let's dive a little deeper. As you architect and design systems, you will be making lots of decisions, about everything from the system's overall structure to what tools and technologies to use. So what makes a decision an *architectural decision*?

In most cases, any choice you make that affects the *structure* of your system is an architectural decision. Here are a couple of example decisions:

NOTE

To jog your memory, picking whether you'd like a one- or two-story house would be an architectural decision.

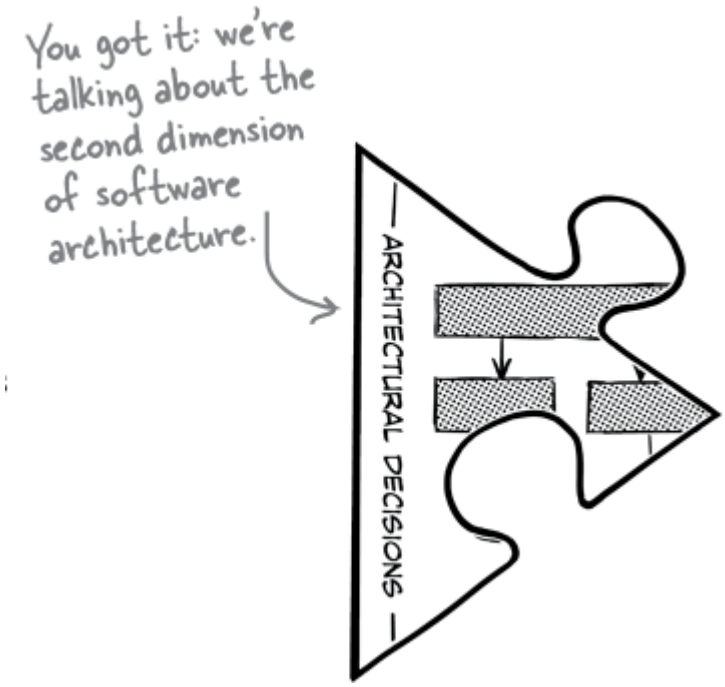

You got it: we're talking about the second dimension of software architecture.
ARCHITECTURAL DECISIONS

Figure 2-24.


I believe we have a decision—we're going to split the order-shipping service apart from the order-tracking service.

Figure 2-25.

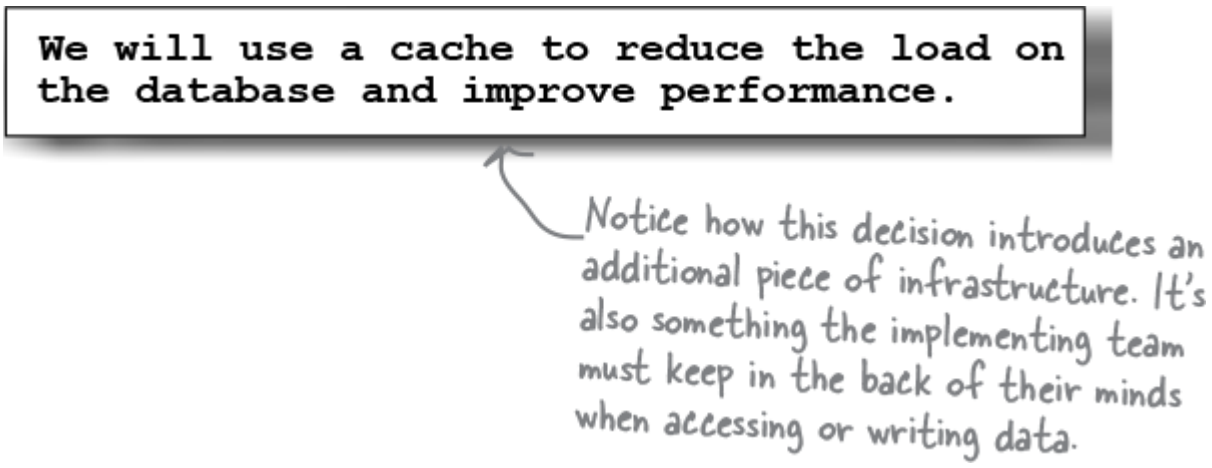


Figure 2-26.

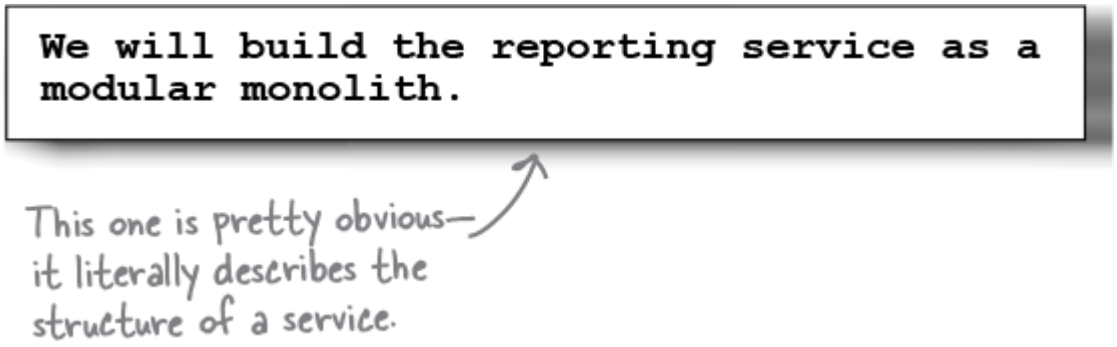


Figure 2-27.

Notice how these decisions act as guides rather than rules. They aid teams in making choices, without being too specific. Most (but not all) of the architectural decisions you'll make will revolve around the structure of your systems.

NOTE

As we put it in Chapter 1: "Architectural decisions serve as guideposts and help development teams make the right technical choices."

What else makes a decision architectural?

Usually, architectural decisions affect the structure of an architecture—are we going with a monolith, or will we leverage microservices? But every so often, you might decide to maintain a particular architectural characteristic. If security is paramount, Two Many Sneakers might make a decision like this:

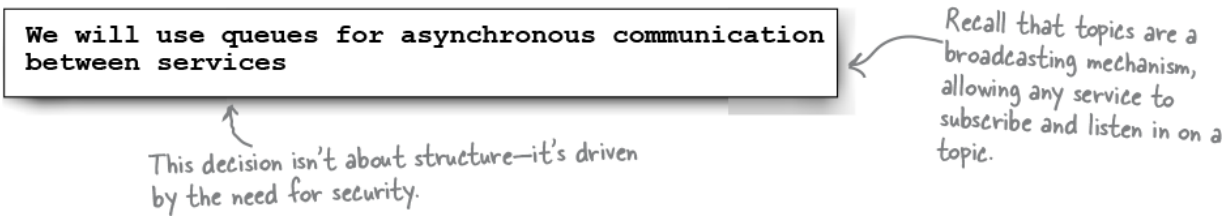


Figure 2-28.

At other times, you might decide on a *specific* tool, technology, or process if it affects the architecture or indirectly helps you achieve a particular architectural characteristic. For example:

Figure 2-29.

Everything in this chapter so far leads to this important moment—making an architectural decision. You start with a trade-off analysis. Then you consider the pros and cons of each option in light of other constraints, like business and end-user needs, architectural characteristics, technical feasibility, time and budgetary constraints, and even development concerns. Then, **finally,** you can make a decision.

SERIOUS CODING

Figure 2-30.

Michael Nygard, author of the book *Release It!* (Pragmatic Programmer, 2018), defines an architecturally significant decision as "something that has an effect on how the rest of the project will run" or that can "affect the structure, non-functional characteristics, dependencies, interfaces, or construction techniques" of the architecture. To learn more, we recommend this blog post: https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions.


Hold up.
Whiteboards are great,
but there has to be a more
permanant way of recording the
trade-off analysis, the decision, and
most importantly, why that choice
was made. Whiteboards seem
awfully temporary, no?

Figure 2-31.

You bring up several good points. Trade-off analysis is an involved process. It'd be a real waste if we lost all that work just because someone got a little hasty with the eraser. It's important that to record our decision in a more permanent way. But you subtly suggest something else. While the decision itself is important, *why* we made that decision might be even more important. Which leads us to...

The second law of software architecture

Making decisions is one of the most important things software architects do.

Let's say you and your team do a trade-off analysis and conclude that you're going to use a cache to improve your application's performance. The result of your analysis is that your system starts using a cache somewhere. The **what** is easy to spot.

That decision is important, but so are the circumstances in which you made that decision, its impact on the team implementing it, and **why**, of all the options available to you, you chose what you did.

This leads us to **the Second Law of Software Architecture**.

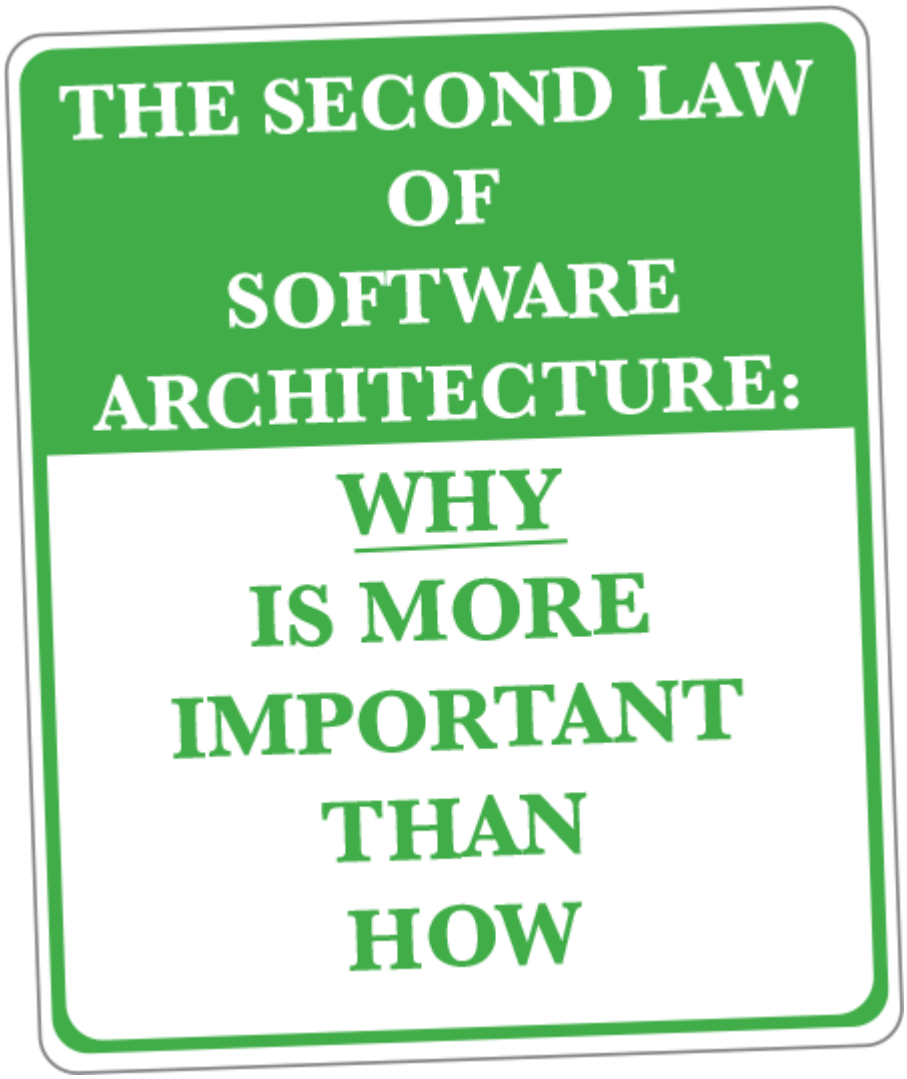


Figure 2-32.

You see, future architects (or even "future you") might be able to discern what you did and even how you did it—but it'll be very hard for them to tell *why* you did it that way. Without knowing that, they might waste time exploring solutions you've already rejected for good reasons, or miss a key factor that swayed your decision.

This is why we have the Second Law. You need to understand **and** record the "why" of each decision so it doesn't get lost in the sands of time.

So how do we go about capturing architectural decisions? Let's dive into that next.

Architectural Decision Records (ADRs)

Do you remember everything you did last week? No? Neither do we. This is why it's important to document stuff—especially the important stuff.

Thanks to the Second Law of Software Architecture, we know we need a way to capture not just the decision, but the reason we made it. Architects use *architectural decision records* (ADRs) to record such decisions because it gives us a specific template to work with.

NOTE

We cannot emphasize enough how important keeping these records is.

An ADR is a document that describes a specific architecture decision. You write one for every architecture decision you make. Over time, they'll build up into an architecture decision log. Remember that architecture decisions form the second dimension to describe your architecture. ADRs are the documentation that supports this dimension.

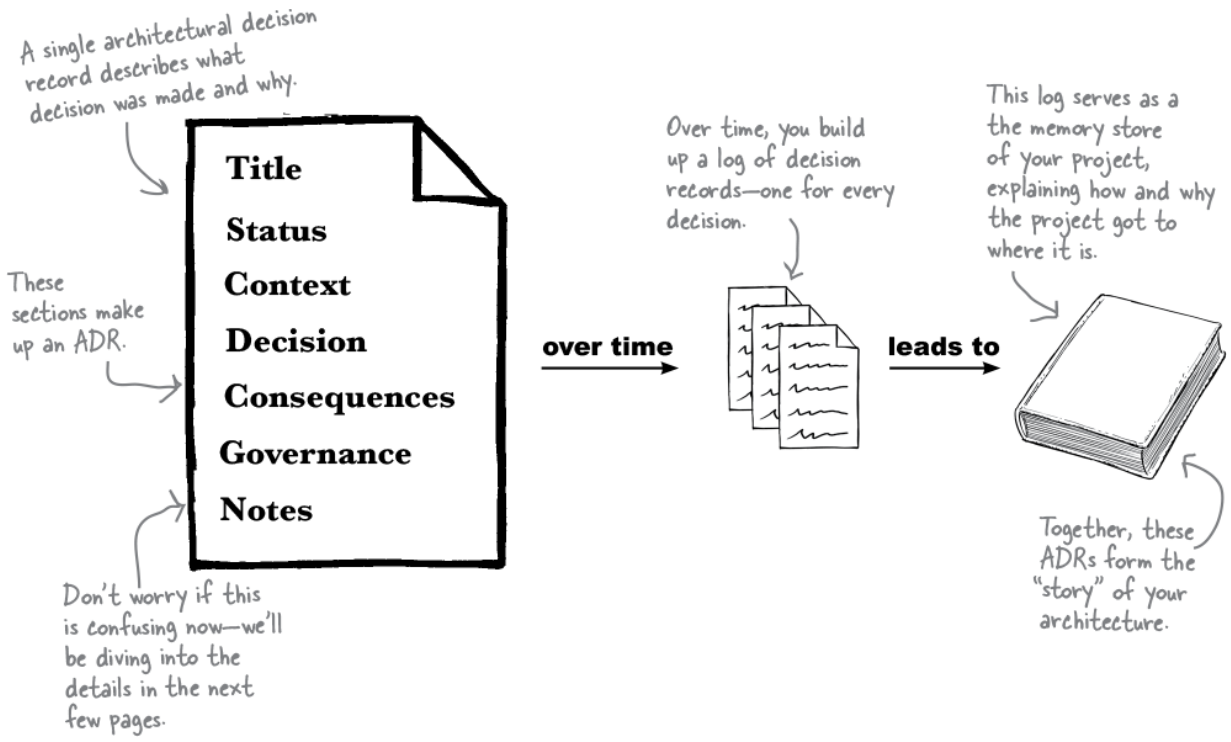


Figure 2-33.

An ADR has seven sections—Title, Status, Context, Decision, Consequences, Governance, and finally Notes. Every aspect of an architectural decision, including the decision itself, is captured in one of these sections. Let's take a look, shall we?

Cubicle conversation

Figure 2-34.

Mara: Doing the trade-off analysis between queues and topics took it out of me.

Sam: Me too. Trade-off analysis can be arduous, but I'm glad we got it done. This is a big architectural decision. It's crucial that we understand the pros and cons of every choice.

Alex: Yeah, yeah. So we decided to use queues, right? Now can we get back to programming?

NOTE

Guess what? You're going to be helping the team write their ADR. Keep an eye out for those exercises.

Sam: Slow down a second. You're right—we've made a decision. Now we should record our decision in an ADR.

Alex: But why? We already know what we're going to do. That seems like a lot of work.

Mara: Look, we know why we chose to go with queues. It's the option that best supports the architectural characteristics we want to maximize in the system, right?

Sam: Correct. And while *we* know why we made that decision, what about anyone else who might come along and wonder why we chose queues over topics, like future employees? *That's* why we should record our thinking.

Alex: I can see that being useful.

Mara: Great! So can we start drafting our ADR?

Writing ADRs: Getting the title right

Every ADR starts with a title that describes the decision. Craft this title carefully. It should be meaningful, yet concise. A good title makes it easy to figure out what the ADR is about, which is especially handy when you're frantically searching for an answer!

Let's dive deeper into what a good ADR title looks like. Imagine a team is writing a service that provides surveys to customers. They've done a trade-off analysis and have decided to use a relational database to store survey results. Here's what their ADR title might look like:

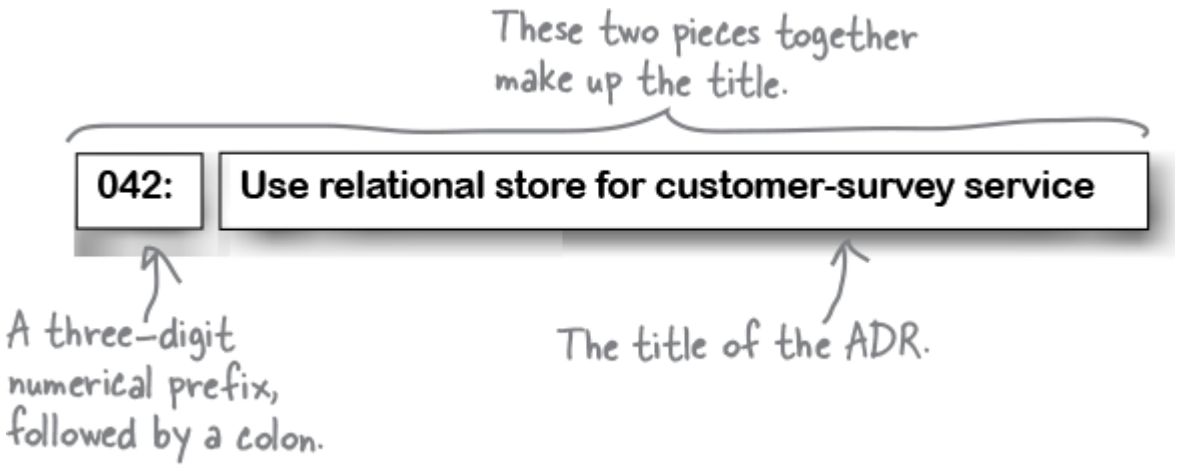


Figure 2-35.

The title should consist mostly of nouns. Keep it terse: you'll have plenty of opportunities to go into detail later. It should describe what the ADR is about, much like the headline of a news article or blog post. Get that right, and the rest will follow.

The title should start with a number—we suggest using three digits, with leading zeroes where needed. This allows you to number your ADRs sequentially, starting with 001 for your first ADR, all the way to 999. Every time you add a new ADR, you increment the number. This makes it easy for anyone reading your records to know which decisions came before others.

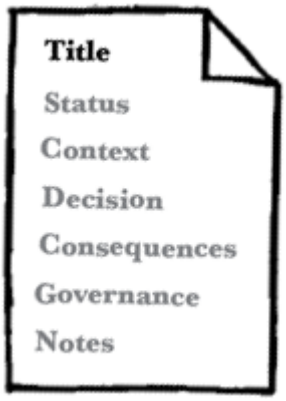


Figure 2-36.

THERE ARE NO DUMB QUESTIONS

Q: What happens if we end up writing more than 999 ADRs?

A: That's a lot of ADRs! If that were to happen, you'd need to revise a bunch of titles (and potentially filenames). In our experience, a three-digit prefix is plenty.

Q: Can I ever reuse a ADR number?

A: Every ADR gets a unique identifier. This makes it easier to reference them without confusion.

NOTE

More about this when we discuss the Status section.

EXERCISE

Figure 2-37.

For the rest of this chapter, you'll be helping the team at Two Many Sneakers write out their ADR. They've decided to use asynchronous messaging, with queues between the trading service and downstream services. Here, you'll start with the title. Assume this is the **12th** ADR the team is writing. What title would you give this ADR? Don't forget to number it! Use this space to jot down your thoughts, then see the Solutions page at the end of this chapter for a few of our ideas.

Writing ADRs: What's your status?

Great! You've settled on a descriptive title. Next, you'll need to decide on the status of your ADR. The status communicates where the team stands on the decision itself.

But wait—isn't the point of the ADR to record a decision? Well, kinda. But making decisions is a process.

ADRs do record architectural decisions, but they also act as documentation, making it easier to share and collaborate. Others might need to look at or even sign off on an ADR. Let's start by looking at all of the statuses an ADR can have.

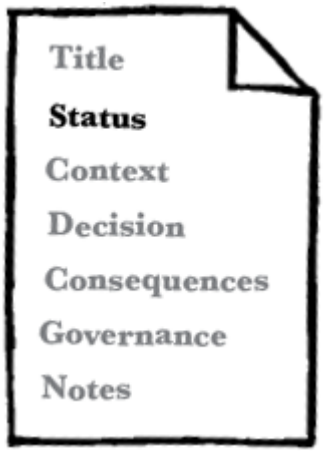


Figure 2-38.

Request for Comment (RFC)

Use this status for ADRs that need additional input—perhaps from other teams or some sort of advisory board. Usually, these ADRs affect multiple teams or address a cross-cutting concern like security. An ADR in RFC status is typically a draft, open for commentary and critique from anyone invited to do so. An ADR in RFC status should **always** *have a "respond by" deadline.*

NOTE

This is like planning an evening out. You know you'd like to go out and which friends you want to invite, but you hope they'll suggest a restaurant.

NOTE

You ask everyone to respond by Tuesday so you can make reservations. (The deadline is important, since Ted can never make up their mind about anything.)

Proposed

After everyone has a chance to comment, the ADR's status moves to Proposed. This means the ADR is waiting for approvals. You might edit it or even overhaul the decision if you

discover a limitation that makes it a no-go. In other words, you still haven't made a decision, but you're getting there.

NOTE

You have a plan for the evening, but you haven't hit "Send" on the invite yet—just in case the weather turns.

Accepted

Does exactly what it says on the tin . A decision has been made, and everyone is on board who needs to be. An Accepted status also tells the team tasked with implementing this decision that they can get started.

NOTE

Oh yeah. Everyone has RSVPed. Time to find a cool outfit!

If there's no need for feedback from others, you can set the ADR's status to Accepted as soon as the decision is made. Most ADRs stay at Accepted, but there is one more status: Superseded.

You've arrived at a decision, which you diligently record in an ADR. Sealed, signed, delivered—you're done.

But then things change.

Maybe the business is growing and the board decides to focus more on scalability than on time to market. Maybe the company is entering international markets and needs to comply with EU data privacy and retention regulations. Whatever the reason, the decision you made is no longer relevant. What now?

Well, you write another ADR. The old ADR is *superseded* by the new one, and you record it as such. Suppose the customer-survey team realizes that a relational database is no longer fulfilling their needs—so they do another trade-off analysis and decide to switch to a document store. Here are the titles and statuses of the old and new ADRs:

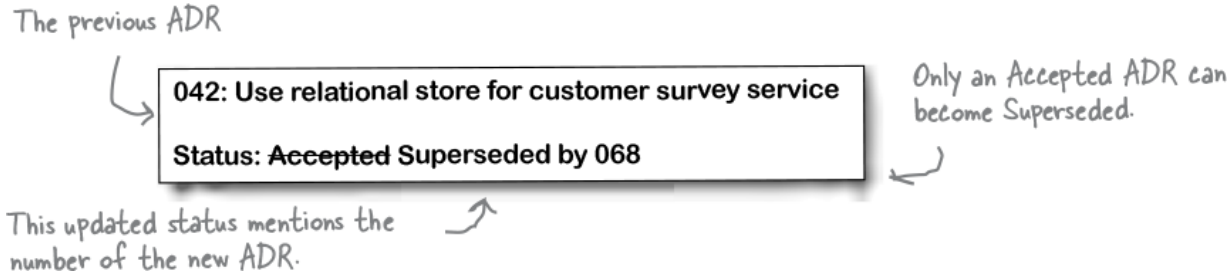


Figure 2-39.

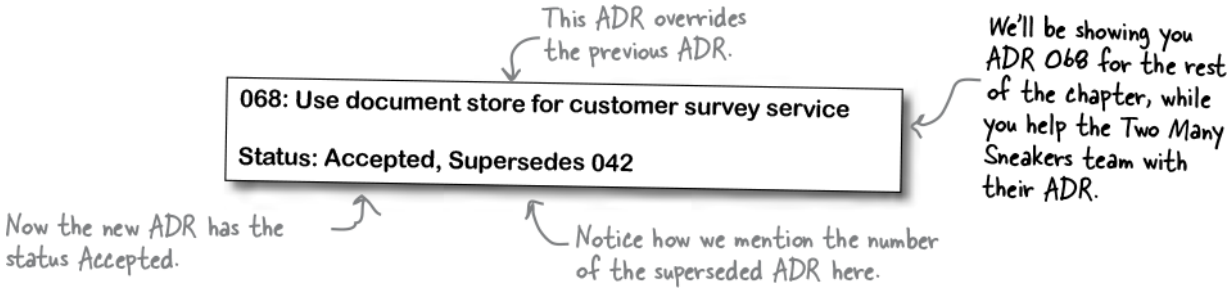


Figure 2-40.

An accepted ADR can move into Superseded status if a future ADR changes the decision it documents. It's important for both ADRs to highlight which ADR did the *superseding* and which ADR has been *superseded*. This bidirectional linking allows anyone looking at a superseded ADR to quickly realize that it's no longer relevant, and tells them exactly where to look. Anyone looking at the superseding ADR can follow the link back to the superseded ADR to understand everything involved in solving that particular problem.

NOTE

Linking ADRs is an important part of a project's "memory." It helps everyone remember what has already been tried.

THERE ARE NO DUMB QUESTIONS

Q: All this superseding and numbering seems overly complicated. Why not just edit the original ADR?

A: We use a three-digit prefix in the ADR title because it helps sequence things. Let's say ADR 007 no longer applies to your situation, but you've made a bunch of architectural decisions in the meantime. The last ADR in your architectural decision log is ADR 013.

Now you need to reevaluate ADR 007. Say you choose to edit it, as opposed to superseding it with ADR 014. What would happen?

Chronologically speaking, you amend ADR 007 *after* accepting ADR 013. But if someone tries to follow the decision process by reading the ADRs, they'd be seeing them in the wrong order! Readers might think that the new decision came first. It wouldn't convey that you made one decision and then had to change it for some reason. In other words, the old ADR 007 was no longer relevant after 013, which makes the new ADR number 014. Confused yet?

Q: So you're telling me that an Accepted ADR is immutable: once accepted, it is not permitted to change. Is that right?

A: Look at you! That's exactly it. Except for when the status of an ADR goes from Accepted to Superseded, a decision recorded in an ADR is immutable. Sure, you might edit the ADR to include additional

information, but for the most part, other than the status, things don’t change much.

EXERCISE

Figure 2-41.

In the previous exercise, you hashed out the title of Two Many Sneakers' ADR about using queues for messaging. Let's say your title gets the green light. Write down the title you chose in the space below and give your ADR a status:

Title: _____

Status: _____

Three months later:

Whoops! The requirements have changed. Your latest trade-off analysis reveals that topics would be a better fit. Everyone has signed off on this, so you need to supersede your ADR with a new ADR. This is the 21st ADR your team has worked on. Write down the title and the new status of the old ADR:

Title: _____

Status: _____

Now write down the title and status of the newly introduced ADR:

Title: _____

Status: _____

Writing ADRs: What's your status? (recap)

There's a lot going on with ADR statuses, so we've created a handy visualization to help you out.

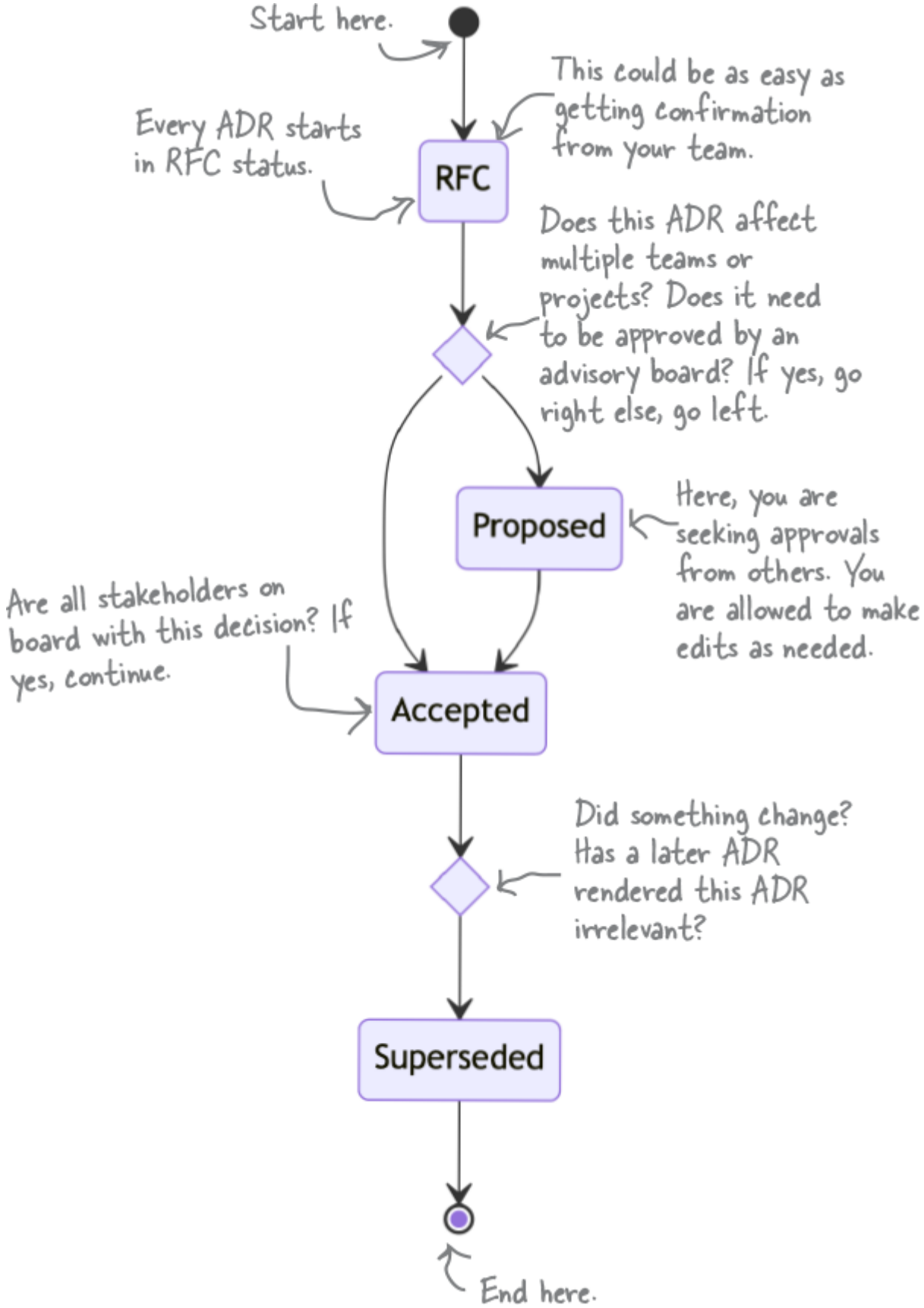

Start here.
This could be as easy as getting confirmation from your team.
Every ADR starts in RFC status.
RFC
Does this ADR affect multiple teams or projects? Does it need to be approved by an advisory board? If yes, go right else, go left.
Proposed
Here, you are seeking approvals from others. You are allowed to make edits as needed.
Are all stakeholders on board with this decision? If yes, continue.
Accepted
Did something change? Has a later ADR rendered this ADR irrelevant?
Superseded
End here.

Figure 2-42.

Writing ADRs: Establishing the context

Context matters. Every decision you've ever made, you made within a certain context and with certain constraints. When you chose what to have for breakfast this morning, the context might have included how hungry you were, how your body felt, your lunch plans, and whether you're trying to increase your fiber intake. It's no different for software architecture.

The Context section in the ADR template is your place to explain the circumstances that forced you to make the decision the ADR is capturing. It should also capture any and all factors that influenced your decision. While technological reasons will usually find their way onto this list, it's not unusual to include cultural or political factors to help the reader understand where you're coming from.

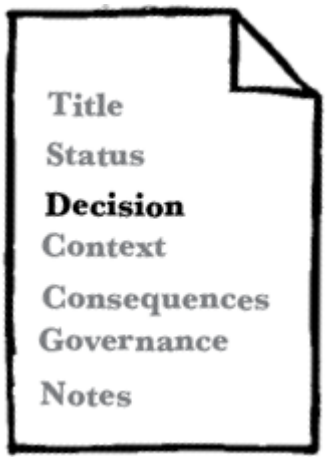


Figure 2-43.

NOTE

Let's continue working on the ADR we started with in the Status section.

Context

We need to simplify how we store customer survey responses. The data currently resides in a relational store, and its rigid schema requirements have become challenging as we evolve the surveys (such as introducing different or extended surveys for our premium customers).

There are various options available to us, like the JSONB datatype in PostgreSQL or document stores like MongoDB.

NOTE

The Context section answers the question "Why did we have to make this choice to begin with?"

SHARPEN YOUR PENCIL

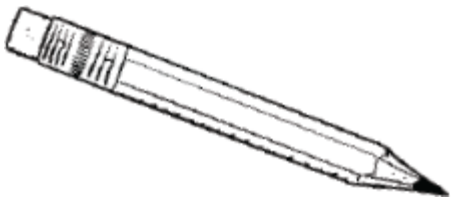


Figure 2-44.

Continue building out the ADR for Two Many Sneakers. Use the space below to write a Context section for the team's decision to use queues for communication between the trading service and other services. (Then compare our take in the Solutions section at the end of this chapter.)

THERE ARE NO DUMB QUESTIONS

Q: What about all that time and effort I spent on the whiteboard? Is that part of the context?

A: If you need to document your trade-off analysis, we suggest you introduce a new section called “Alternatives.” In it, list all the alternatives you considered, followed by your lists of pros and cons.

Using a separate section to list the trade-off analysis delineates it cleanly, and avoids cluttering the Context section.

Writing ADRs: Writing the decision

We’ve finally arrived at the actual decision. Let’s start by looking at the customer survey team’s completed Decision section:

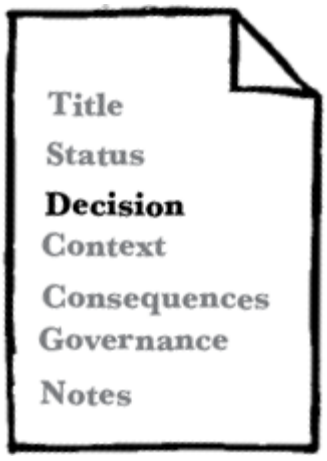


Figure 2-45.

Decision

We will use a document database for the customer survey.

NOTE

Notice the authoritative voice!

The Marketing Department requires a faster, more flexible way to make changes to the customer surveys.

Moving to a document database will provide better flexibility and speed, and will better facilitate changes by simplifying the customer-survey user interface.

NOTE

The Decision section covers the "why" of the decision itself: remember the Second Law?

If this ADR's status is RFC or Proposed, the decision hasn't been made (yet). Even so, the Decision section starts by clearly expressing the decision being made. The tone of the writing should reflect that. It's best to use an authoritative voice when stating the decision, with

active phrases like “we will use” (as opposed to “we believe” or “we think”).

The Decision section is also the place to explain why you’re making this decision, paying tribute to the Second Law of Software Architecture: “Why is more important than how.” Future you, or anyone else who reads the ADR, will understand not just the decision but the justification for it.

NOTE

In the Context section, you explained why this decision was on the table. The Decision section, which immediately follows it, explains the decision itself. Together, they allow the reader to frame the decision correctly.

NOTE

This is also a great place to list others who signed off on this decision. For example, “the Marketing department requires...” is an example of CYA.

NOTE

“Cover Your Assets”! :)

WATCH IT!

Figure 2-46.

The ADR is not an opinion piece

Remember that the ADR is not a place for anyone's opinions on the state of things. It's easy to slip into that mode, especially when justifying a decision. Even explaining context can sometimes make it hard to stay objective.

Treat an ADR like a journalist treats a news article—stick to the facts and keep your tone neutral.

EXERCISE

Figure 2-47.

Time for you to write the Decision section of the ADR for Two Many Sneakers. Here are the main factors the team considered when making their decision:

- Queues allow for heterogeneous messages.
- Security is an important architectural characteristic for the stakeholders.

We've given you some space to write out a Decision section, including the corresponding justification. Your section should answer the question "Why queues?" **Hint:** Be sure to focus on the decision and the "why." See the Solutions at the end of the chapter for our own take.

NOTE

Feel free to glance back at the trade-off analysis we did earlier in the chapter to refresh your memory.

THERE ARE NO DUMB QUESTIONS

Q: I am not entirely clear on the difference between the context and the "justification" we provide in the decision section. Aren't those the same thing?

A: Maybe an example will help. Say it's your best friend's birthday, and you and a few others decide to go out to a fancy dinner to celebrate. That's the *context*—the circumstances surrounding the decision you have to make.

Before you decide on the details, you might make a list of possible restaurants (the *alternatives* available to you), thinking about how well the cuisines they offer match everyone's preferences. This would be akin to a trade-off analysis.

You pick a pan-Asian bistro: that's the *decision*. You chose that particular restaurant because its menu has vegetarian and gluten-free options, and it allows anyone with dietary restrictions to make substitutions. That's the *justification* for your decision.

Writing ADRs: Considering the consequences

Every decision has consequences. Did you work out extra hard yesterday? If so, you might be sore this morning. (But maybe a little bit proud of yourself, too!)

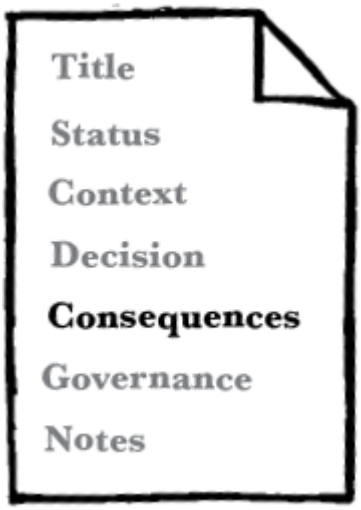


Figure 2-48.

It's important to realize the consequences—good and bad—of architectural decisions and document them. This increases transparency by ensuring that everyone understands what the decision entails, including the team(s) affected by it. Most importantly, it allows everyone to assess whether the decision's positive consequences will outweigh its negative consequences.

The consequences of an ADR can be limited in scope or have huge ramifications. Architectural decisions can affect all kinds of things—teams, infrastructure, budgets, even the implementation of the ADR itself. Here's an incomplete list of questions to ask:

☐ How does this ADR affect the implementing team? For instance, does it change the algorithms? Does it make testing harder or easier? How will we know when we're "done" implementing it?

☐ Does this ADR introduce or decommission infrastructure? What does that entail?

☐ Are cross-cutting concerns like security or observability affected? If so, what effects will that have across the organization?

☐ How will the decision affect your time and budget? Does it introduce costs or save money? Will it take arduous effort to implement or make things easier?

NOTE

Time and money are big—be sure to think this one through!

☐ Does the ADR introduce any one-way paths? (For example, using queues means we can't control the order of messages.) If so, elaborate on this.

NOTE

Of course, the ADR might make things simpler and more cost-effective. If so, that's definitely worth highlighting.

Collaborating with others is a great way to make sure your assessment is thorough. No matter how hard you think through the consequences of the ADR, you're likely to miss a few things; multiple perspectives will reveal more potential consequences. Here's a sample Consequences section:

Consequences

Since we will be using a single representation for all surveys, multiple documents will need to be changed when a common survey question is updated, added, or removed.

NOTE

Highlight the consequences of the decision for the implementation team.

The IT team will need to shut down survey functionality during the data migration from the relational database to the document database.

NOTE

This might affect customer experience—be sure to mention that.

SHARPEN YOUR PENCIL

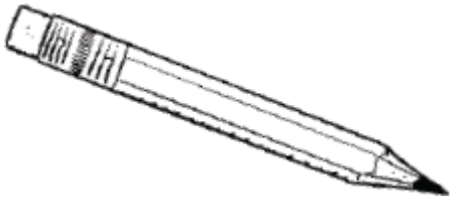


Figure 2-49.

Help the Two Many Sneakers team iron out the Consequences section of their ADR. Here are a few things to think about:

- ➢ A queue introduces a new piece of infrastructure.
- ➢ The queues themselves will probably need to be highly available.
- ➢ Queues mean a higher degree of coupling between services.

NOTE

There are no right or wrong answers, but if you'd like to see how we approached this, glance at the solution at the end of the chapter.

BRAIN POWER

Figure 2-50.

Think about an architectural decision made in your current project, or one you've worked on in the past. That might be the choice of the programming language used, the application's structure, or even the choice of database. Can you think of at least two intended consequences and two unintended consequences of that decision?

Writing ADRs: Ensuring governance

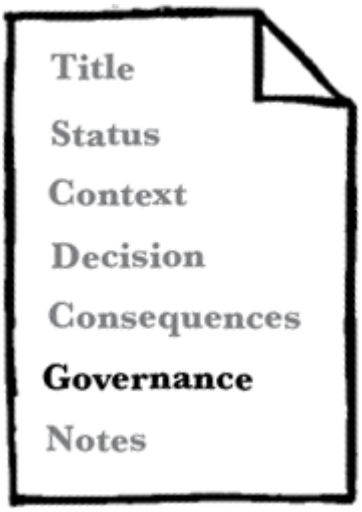


Figure 2-51.

Have you ever made a New Year's resolution that fizzled out before the end of February? Maybe you joined a gym, only to end up paying but never working out? Us too. A decision is only good if you act on it, and if you don't accidentally stray away from it in the future.

Sure, you and your team spent a bunch of time analyzing trade-offs and writing an ADR to record the decision. Now what? How do you ensure that the decision is correctly implemented—and that it stays that way?

This is why the Governance section plays a vital role in any ADR. Here, you outline how you'll ensure that your organization doesn't deviate from the decision—now or in the future. You could use manual techniques like pair programming or code reviews to ensure compliance, or you could use automated means like specialized testing frameworks.

NOTE

One of your authors has written a book called "Building Evolutionary Architectures" that shows you how to use "fitness functions" for architectural governance. Be sure to pick up a copy. (After you're done with this book, of course!)

NOTE

These two sections aren't part of the standard ADR template, but we think they add a lot of value.

NOTE

If the word "governance" conjures up ideas of regulatory compliance, well, this isn't that.

Writing ADRs: Closing notes

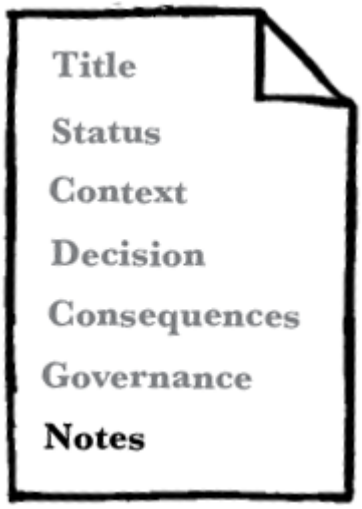


Figure 2-52.

The notes section is simply just that—metadata about the ADR itself. Here's a listing of fields we like to include in our ADRs:

NOTE

This section is handy even if the tool you use to store your ADRs automatically records things like creation and modification dates. Yes, including this information may be repetitive, but making it part of the ADR makes it easier to discover.

- ☐ Original author
- ☐ Approval date
- ☐ Approved by
- ☐ Superseded date

- [ ] Last modified date
- [ ] Modified by
- [ ] Last modification

EXERCISE

Figure 2-53.

Let's bring it all together! You've been working piecemeal on the ADR for the Two Many Sneakers team. We'd like you to flip back over the past few exercises and copy your ADR sections onto this page to create a full ADR. We've given you the section titles—all you have to do is fill 'em out. (Assume the status to be "Accepted.")

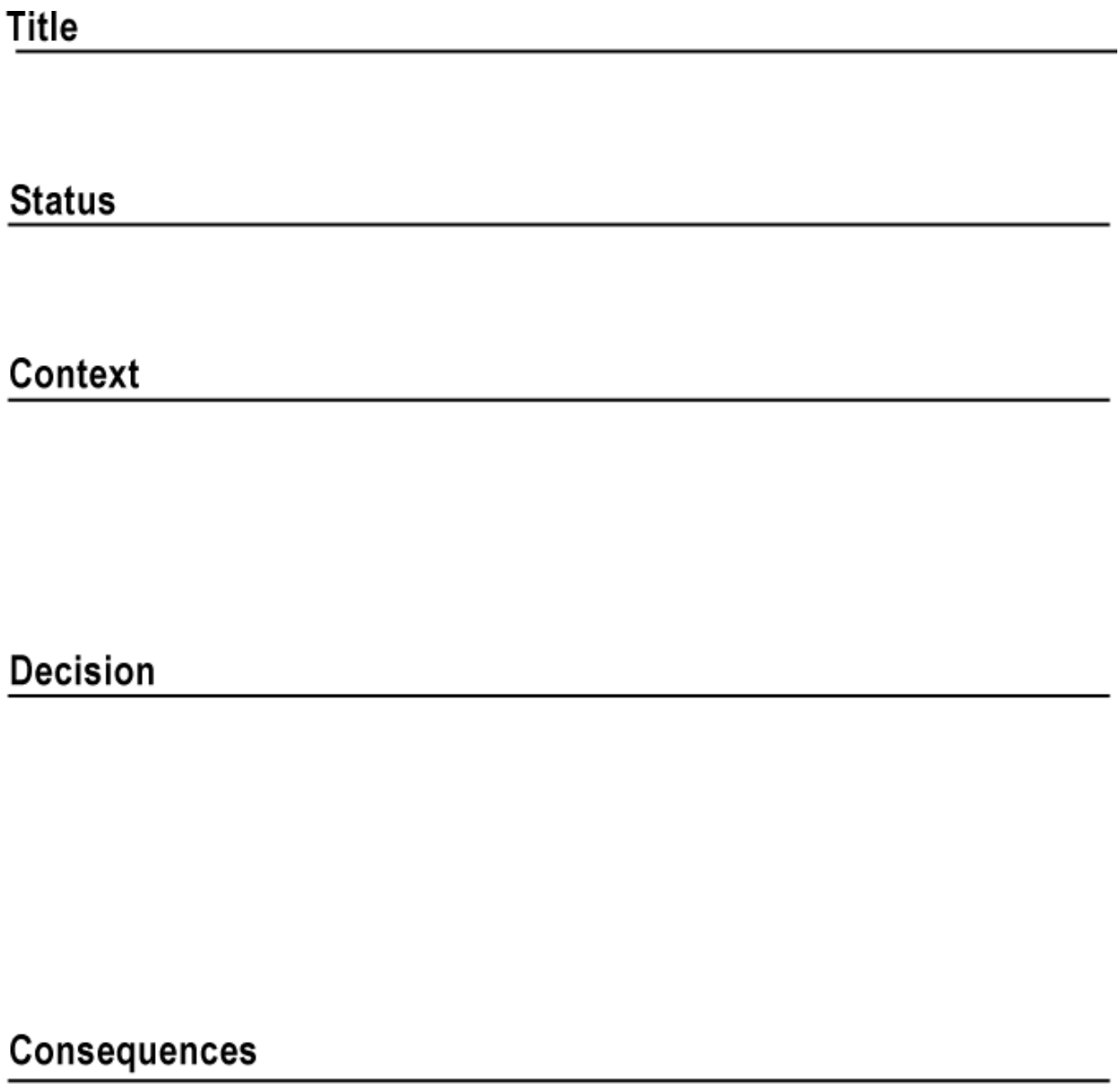


Figure 2-54.

And there you have it: a complete ADR in its full glory.

NOTE

They grow up so quickly! :') We're so proud of you all.

THERE ARE NO DUMB QUESTIONS

Q: I really like the ADR template. But where am I supposed to store my ADRs?

A: There are lots of options—it all depends on what you and your team are comfortable with, and who else might be interested in reading or contributing to the ADRs.

One option is to store ADRs in plain-text files (or maybe Markdown or AsciiDoc files) in a version-control system like Git. This way, there's a commit history showing any changes to the ADRs. The downside is that non-developers don't always know how to access version-controlled documents. If you do choose to store your ADRs this way, we recommending keeping them in a separate repository (as opposed to stuffing them in with your source code). You'll thank us later.

Alternatively, you could use a wiki. Most wikis use a WYSIWYG ("what you see is what you get") editor, so they're accessible to more people. Just be sure that your choice of wiki can track changes. You wouldn't want someone to edit an ADR accidentally without everyone knowing.

Whatever you choose, make sure it's easy to add, edit, and search for ADRs. We've seen too many honest efforts at recording ADRs die just because no one could find the ADRs again if their lives depended on it.

Q: My whole team loves Markdown. (Plain text for the win!) Any advice on file-naming conventions?

A: Recall that ADR titles have a three-digit prefix, followed by a very succinct description of the ADR. If you store your ADRs as plain-text files, we recommend using the title as your filename, including the prefix. For example, an ADR with the title "042: Use queues between the trading and downstream services" should be stored in a file named `042-use-queues-between-the-trading-and-downstream-services.md`. We like using all lowercase letters, which avoids any confusion between different operating systems. Replace spaces with hyphens to avoid whitespace.

This forces you to come up with good titles! And the three-digit prefix means you can simply sort the files in a folder by name to put them in the right order.

Q: Do you recommend any tools that make it easier to write and manage ADRs?

A: Oh, sure! There are many options, from command-line tools to language-specific tools that allow you to record ADRs directly in your source code. You can see a list of available tools at https://adr.github.io/#decision-capturing-tools.

Most third-party tools make assumptions about the format of the ADR—perhaps they generate Markdown files or store the files in a specific directory structure. Test-drive a tool a few times to get a feel for it.

Finally, some age-old advice: keep it simple, silly. We suggest you start by writing out ADRs without any complicated tooling or automation. Get a sense of what works best for your team. Then, as your needs grow, go find a tool that fits those needs.

Q: Do ADRs always belong to a single project, or can they affect multiple projects and teams? How about the whole organization?

A: Yes, yes, and yes. ADRs can be as narrow or as broad as you'd like them to be. Some ADRs are project-specific, affecting only one team. Other ADRs affect many or all teams in an organization. At the online retailer Amazon, there's an ADR affectionately referred to as "the Jeff Bezos API mandate." It records a decision that company founder Jeff Bezos once made: that all services within Amazon can only talk to other services via an API. Naturally, this affected the entire organization—no small feat, given Amazon's size.

Most cross-project or cross-team ADRs require a lot of collaboration, and often the blessing of a central architecture-review board. Such ADRs tend to affect cross-cutting concerns, like how services should communicate with one another or which data-transfer protocol to use.

ADRs related to security or regulatory compliance often cut across multiple teams or a whole organization.

The benefits of ADRs

We hope we've convinced you by now that recording your decisions in ADRs need not be a long, arduous process. We really like the format we've shown you in this chapter, but feel free to tweak or modify it.

Is recording architectural decisions really that important? We certainly think so! There are tons of benefits to recording architectural decisions—not just for you and your team, but for your entire organization. Let's quickly recap:

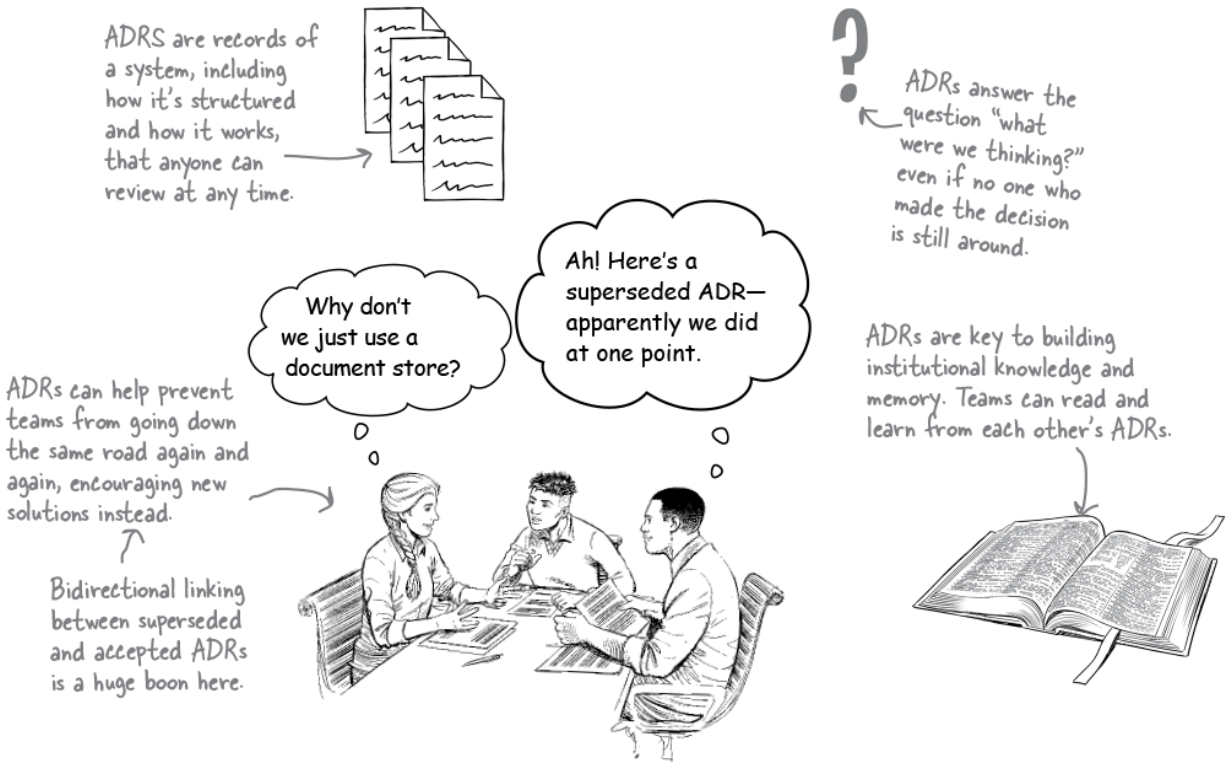

ADRS are records of a system, including how it's structured and how it works, that anyone can review at any time.
ADRs answer the question "what were we thinking?" even if no one who made the decision is still around.
Why don't we just use a document store?
Ah! Here's a superseded ADR—apparently we did at one point.
ADRs can help prevent teams from going down the same road again and again, encouraging new solutions instead.
Bidirectional linking between superseded and accepted ADRs is a huge boon here.
ADRs are key to building institutional knowledge and memory. Teams can read and learn from each other's ADRs.

Figure 2-55.

WATCH IT!

Figure 2-56.

Keep the ADR process as frictionless as possible

It's tempting to add sections to the ADR template in the hope of being comprehensive. While that's a noble goal, it adds work. If you keep "feeding the beast," the documentation process gets harder. That can discourage people, and some might stop writing ADRs altogether.

Focus on concision and brevity. Keep it simple. You'll thank us later.

Two Many Sneakers is a success

The team at Two Many Sneakers are ecstatic. Their customers love getting realtime notifications about new offerings in the app, and the improved analytics are giving the security team the information they need to sniff out any and all sneaker scams from a mile away.

Figure 2-57.

Grokking the two laws of software architecture will serve you well. Now you know that there are no "best practices" in software architecture—just trade-offs. It's up to you (and your team) to find the most viable and best-fitting option. And don't forget to record your decision in an ADR!

Onward and upward.

BULLET POINTS

- There is nothing “static” about architecture. It’s constantly changing and evolving.
- Requirements and circumstances change. It’s up to you to change your architecture to meet new goals.
- For every decision, you will be faced with multiple solutions. To find the best (or least worst), do a trade-off analysis. This collaborative exercise helps you identify the pros and cons of every possible option.
- The **First Law of Software Architecture** is: Everything in software architecture is a trade-off.
- The answer to every question in software architecture is “it depends.” To learn which solutions are best for your situation, you’ll need to identify the top priorities and goals. What are the requirements? What’s most important to your stakeholders and customers? Are you in a rush to get to market, or hoping to get things stable in growth mode?
- The product of a trade-off analysis is an architectural decision: one of the four dimensions needed to describe any architecture.
- An architectural decision involves looking at the pros and cons of every choice in light of other constraints—such as cultural, technical, business, and customer needs—and choosing the option that serves these constraints best.

- Making an architectural decision isn't just about choosing; it's also about *why* you're choosing that particular option.
- The **Second Law of Software Architecture** is: Why is more important than how.
- To formalize the process of capturing architectural decisions, use Architectural Decision Records (ADRs). These documents have seven sections: Title, Status, Context, Decision, Consequences, Compliance, and Notes.
- Over time, your ADRs will build into a log of architectural decisions that will serve as the memory store of your project.
- An ADR's title should consist of a three-digit numerical prefix and a noun-heavy, succinct description of the decision being made.
- An ADR can be assigned one of many statuses, depending on the kind of ADR and its place in the decision workflow.
- Once all parties involved in the decision sign off on the ADR, its status becomes Accepted.
- If a future decision supplants an accepted ADR, you should write a new ADR. The supplanted ADR's status is marked as Superseded and the new ADR becomes Accepted.
- The Context section of an ADR explains why the decision needed to be made to begin with.
- The Decision section documents and justifies the actual decision being made. It always includes the "why."

- The Consequences section describes the decision's expected impact, good and bad. This helps ensure that the good outweighs the bad, and aids the team(s) implementing the ADR.
- The Governance section lists ways to ensure that the decision is implemented correctly and that future actions do not stray away from the decision.
- The final section is Notes, which mostly records metadata about the the ADR itself—like its author and when it was created, approved, and last modified.
- ADRs are important tools for abiding by the Second Law of Software Architecture, because they capture the "why" along with the "what."
- ADRs are necessary for building institutional knowledge and helping teams learn from one another.

The "two laws" crossword

Figure 2-58.

Think you've mastered the two laws of software architecture? Why don't you document your knowledge by completing this crossword?

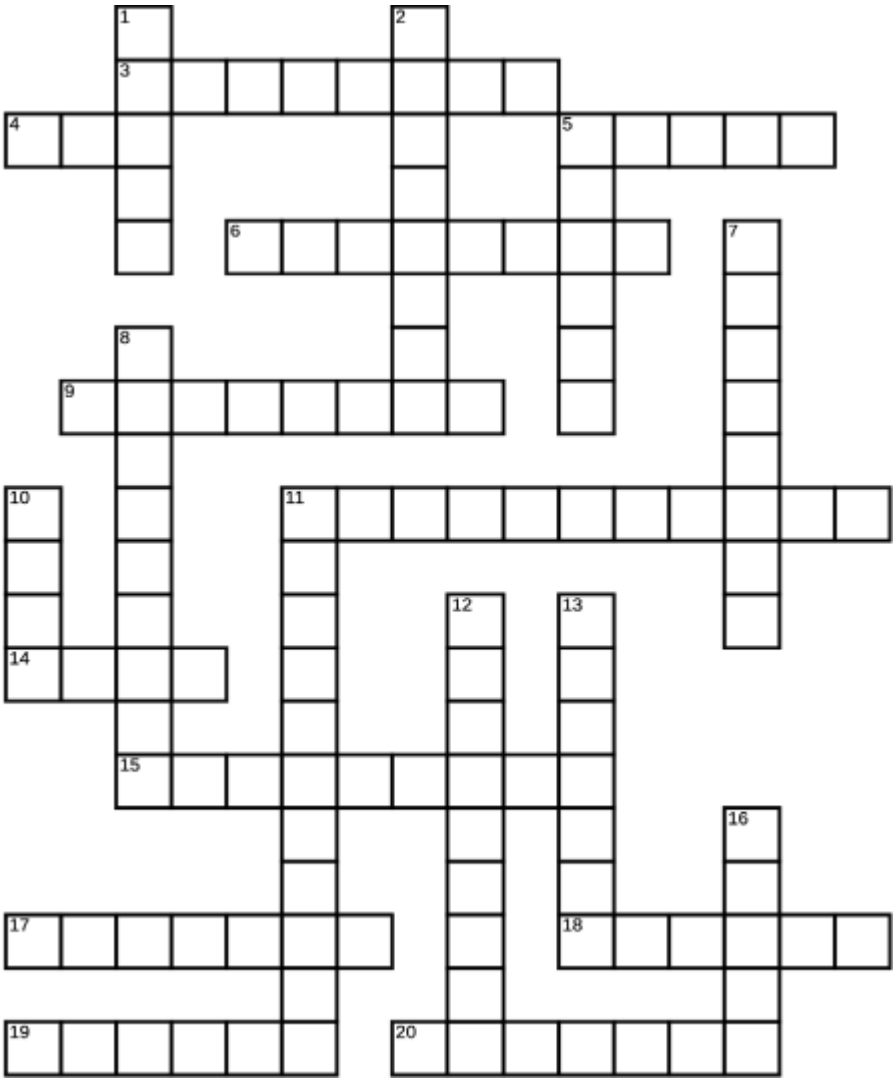


Figure 2-59.

Across

3 Two Many ____

4 More important than how, according to the Second Law

5 If you're too excited about a new tool, you might have ___ Object Syndrome

6 "Everything in software architecture is a trade-off" is the __ __ of software architecture (2 wds.)

9 High or low interdependence

11 Important architectural characteristic for a fast-growing business

14 Documents made up of seven sections (abbr.)

15 Heterogeneous

17 Best tone to use when writing an ADR

18 Examples of messaging mechanisms include queues and ____

19 Topics use a fire-and-____ system

20 Two Many's mobile app communicates with the trading ____

Down

1 Short way to say 'not at the same time'

2 You should record every architectural ____ you make

5 The ___ of an ADR might be “accepted”

7 An architecture characteristic that’s especially important for financial transactions

8 Topics can be independently _____

10 A top-heavy Swedish ship

11 Architects are responsible for making architecturally ____ decisions

12 A new ADR can ___ an old one

13 ADR section that tells you why a decision needed to be made

16 You can list pros and cons on a ____ board

EXERCISE SOLUTION

Figure 2-60.

Which of the following architectural characteristics stand out as important for this particular problem? **Hint:** There are no right answers here, because there is a lot we don't know or aren't sure of yet. Take your best guess—we have provided our solutions at the end of this chapter. We'll get you started:

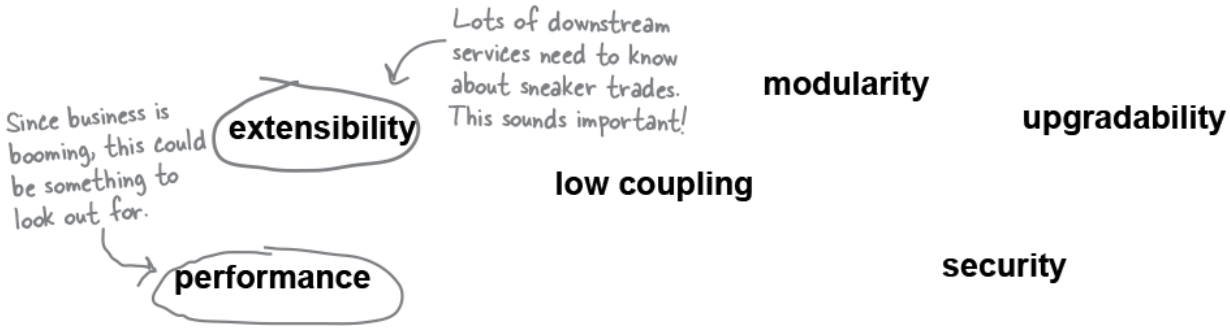


Figure 2-61.

SHARPEN YOUR PENCIL

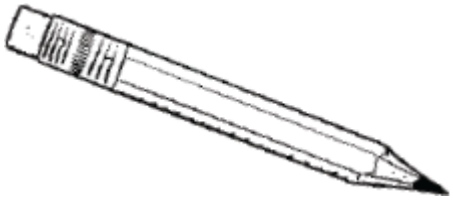


Figure 2-62.

Spend a few minutes comparing the results of our trade-off analysis. Notice how both options support some characteristics but trade off on others? Now we're going to present you with some requirements—see if you can decide if you'd pick queues or topics. Then check our answers at the back of this chapter.

Requirements

"Security is important to us" — Queues / Topics

"Different downstream services need different kinds of information" — Queues / Topics

"We'll be adding other downstream services in the future" — Queues / Topics

Figure 2-63.

SHARPEN YOUR PENCIL SOLUTION

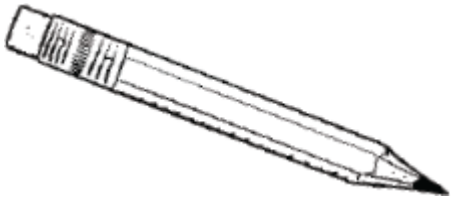


Figure 2-64.

This time, we'd like you to do some trade-off analysis on your own. We chose messaging as the communication protocol between our trading service and its consumers. Messaging is asynchronous. Choosing between asynchronous and synchronous forms of communication (like REST and RPC) comes with its own set of trade-offs! We've given you two whiteboards, one for each form of communication, and we've listed a bunch of "-ilities." We'd like you to consider how each architecture characteristic would work in both contexts. Is this characteristic a pro or a con (or neither) in synchronous communications? What about in asynchronous communications? Place each "-ility" in the appropriate column. **Not all of them apply to this decision**. We put the first pro on the whiteboard for you. When you're done, you can see our answers at the end of the chapter.

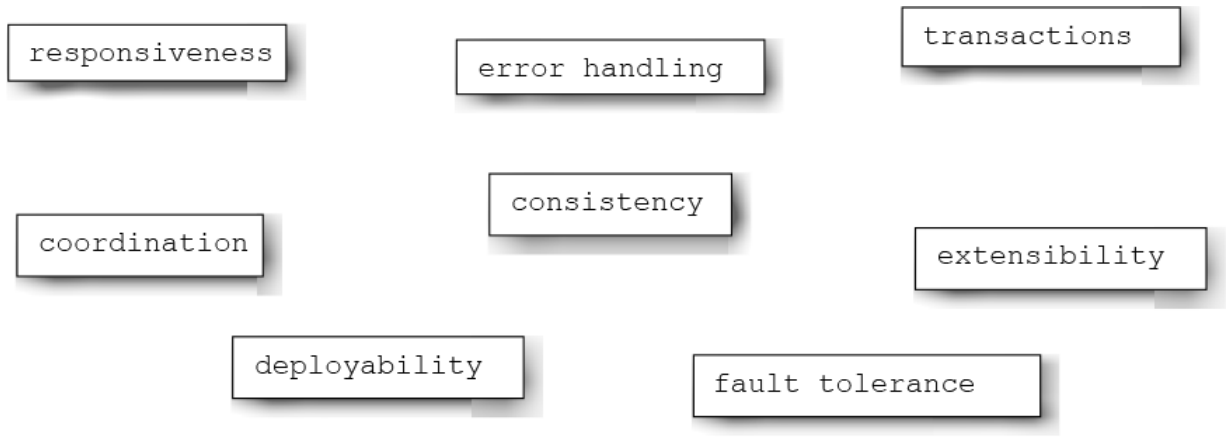


Figure 2-65.

Figure 2-66.

EXERCISE SOLUTION

Figure 2-67.

For the rest of this chapter, you'll write your first ADR. Here, you'll start with the title. The team at Two Many Sneakers has decided to use asynchronous messaging, with queues between the trading service and downstream services. They need to write an ADR for this decision. This is the **12th** ADR the team is writing. What title would you give this ADR? Don't forget to number it! Use this space to jot down your thoughts, then see the Solutions page at the end of this chapter for a few of our ideas.

NOTE

012: **Use of queues for asynchronous messaging between order and downstream services**

EXERCISE SOLUTION

Figure 2-68.

In the previous exercise, you hashed out the title of Two Many Sneakers' ADR about using queues for messaging. Let's say you get the greenlight. Write down the title you chose in the space below and give it a status:

Title: 012: Use of queues for asynchronous messaging between order and downstream services

Status: Accepted

Figure 2-69.

Whoops! It's been a few months, and the requirements have changed. Your latest trade-off analysis reveals that topics would be a better fit. Everyone has signed off on this, so you need to supersede your ADR with a new ADR. This is the 21st ADR your team has worked on. Write down the title and the new status of the old ADR:

Title: 012: Use of queues for asynchronous messaging between order and downstream services

Status: Superseded by 021

Figure 2-70.

Now write down the title and status of the newly introduced ADR:

Title: 021: Use of topics for asynchronous messaging between order and downstream services

Status: Accepted, Supersedes 012

Figure 2-71.

SHARPEN YOUR PENCIL SOLUTION

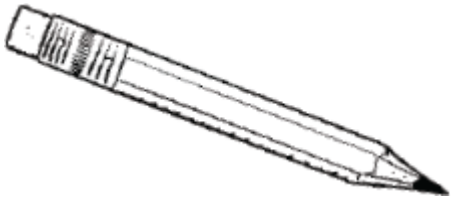


Figure 2-72.

Continue building out the ADR for Two Many Sneakers. Use the space below to write a Context section for the team's decision to use queues for communication between the trading service and other services. (Then compare our take in the Solutions section at the end of this chapter.)

NOTE

The trading service must inform downstream services (namely the notification and analytics services, for now) about new items available for sale and about all transactions. This can be done through synchronous messaging (using REST) or asynchronous messaging (using queues or topics).

EXERCISE SOLUTION

Figure 2-73.

Time for you to write the Decision section of the ADR for Two Many Sneakers. Here are the main factors the team considered when making their decision:

- Queues allow for heterogeneous messages.
- Security is an important architectural characteristic for the stakeholders.

We've given you some space to write out a Decision section, including the corresponding justification. Your section should answer the question "Why queues?" **Hint:** Be sure to focus on the decision and the "why." See the Solutions at the end of the chapter for our own take.

NOTE

We will use queues for asynchronous messaging between the trading and downstream services.

NOTE

Using queues makes the system more extensible, since each queue can deliver a different kind of message. Furthermore, since the trading service is acutely aware of any and all subscribers, adding a new consumer involves modifying it—which improves the security of the system.

SHARPEN YOUR PENCIL SOLUTION

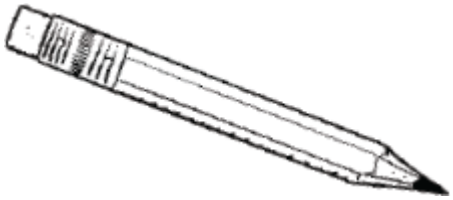


Figure 2-74.

Help the Two Many Sneakers team iron out the Consequences section of their ADR. Here are a few things to think about:

- ➢ A queue introduces a new piece of infrastructure.
- ➢ The queues themselves will probably need to be highly available.

Queues mean a higher degree of coupling between services.

We will need to provision queuing infrastrastructure. It will require clustering to provide for high availability.

If additional downstream services (in addition to the ones we know about) need to be notified, we will have to make modifications to the trading service.

EXERCISE SOLUTION

Figure 2-75.

Let's bring it together! You've been working piecemeal on the ADR for the Two Many Sneakers team. We'd like you to flip back over the past few exercises and copy your ADR sections onto this page to create a full ADR. We've given you the section titles—all you have to do is fill 'em out. (Assume the status to be "Accepted.")

Title

012: Use of queues for asynchronous messaging between order and downstream services

Status

Accepted

Context

The trading service must inform downstream services (namely the notification and analytics services, for now) about new items available for sale and about all transactions. This can be done through synchronous messaging (using REST) or asynchronous messaging (using queues or topics).

Decision

We will use queues for asynchronous messaging between the trading and downstream services.

Using queues makes the system more extensible, since each queue can deliver a different kind of message. Furthermore, since the trading service is acutely aware of any and all subscribers, adding a new consumer involves modifying it—which improves the security of the system.

Consequences

Queues mean a higher degree of coupling between services.

We will need to provision queuing infrastrastructure. It will require clustering to provide for high availability.

If additional downstream services (in addition to the ones we know about) need to be notified, we will have to make modifications to the trading service.

Figure 2-76.

And there you have it! A complete ADR in its full glory.

The “two laws” crossword solution

Figure 2-77.

Think you've mastered the two laws of software architecture? Why don't you document your knowledge by completing this crossword?

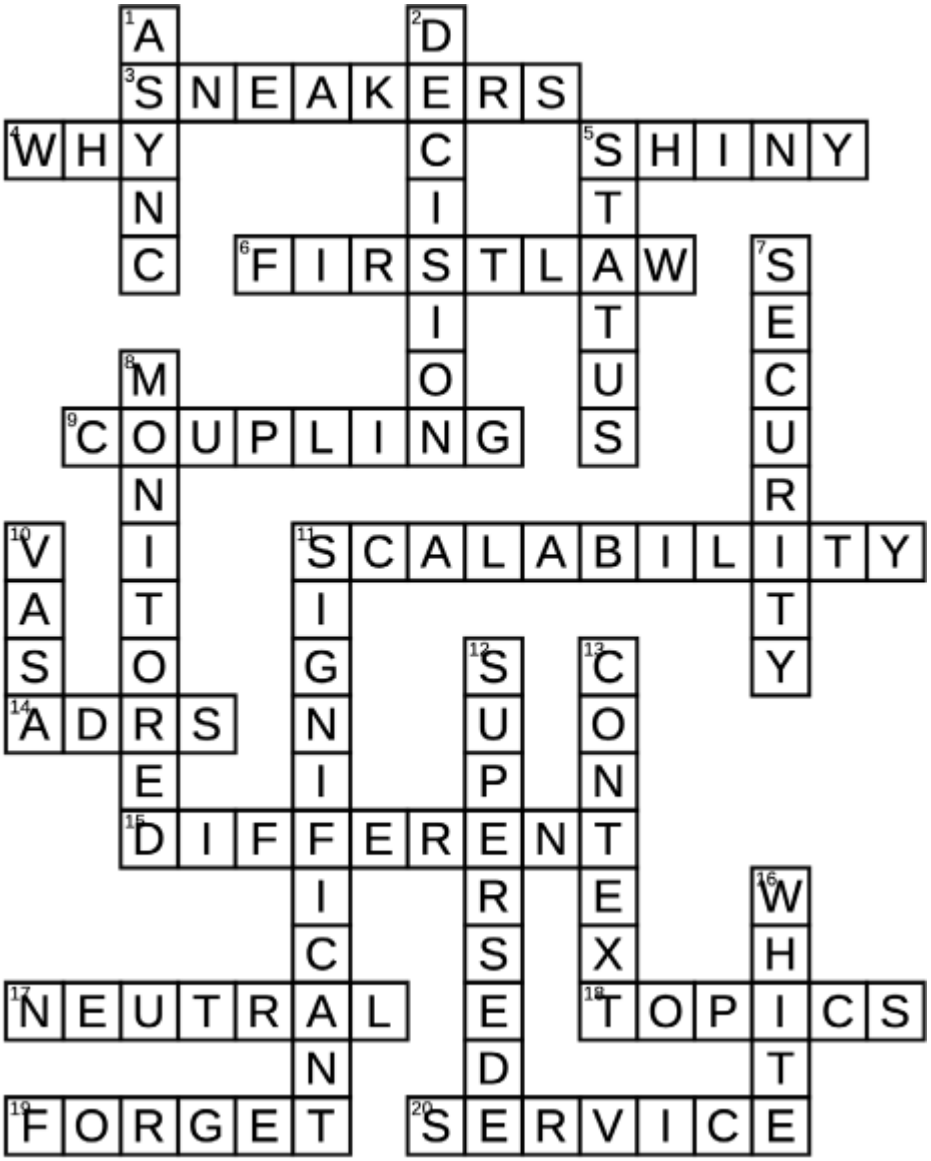


Figure 2-78.

Across

3 Two Many ___

4 More important than how, according to the Second Law

5 If you're too excited about a new tool, you might have ___ Object Syndrome

6 "Everything in software architecture is a trade-off" is the __ __ of software architecture (2 wds.)

9 High or low interdependence

11 Important architectural characteristic for a fast-growing business

14 Documents made up of seven sections (abbr.)

15 Heterogeneous

17 Best tone to use when writing an ADR

18 Examples of messaging mechanisms include queues and ____

19 Topics use a fire-and-____ system

20 Two Many's mobile app communicates with the trading ____

Down

1 Short way to say 'not at the same time'

2 You should record every architectural ____ you make

5 The ___ of an ADR might be “accepted”

7 An architecture characteristic that’s especially important for financial transactions

8 Topics can be independently _____

10 A top-heavy Swedish ship

11 Architects are responsible for making architecturally ____ decisions

12 A new ADR can ____ an old one

13 ADR section that tells you why a decision needed to be made

16 You can list pros and cons on a ____ board

About the Authors

Raju Gandhi is a software craftsman with almost 20 years of hands-on experience scoping, architecting, designing, and implementing full stack applications. A full-time consultant, published author, internationally known public speaker, and trainer, he provides a 360-degree view of the development cycle. He's proficient in a variety of programming languages and paradigms, experienced with software development methodologies, and an expert in infrastructure and tooling. His long pursued hermeticism across the development stack by championing immutability during development (with languages like Clojure), deployment (leveraging tools like Docker and Kubernetes), and provisioning and configuration via code (using toolkits like Ansible, Terraform, Packer, and "everything as code"). In his spare time, you'll find Raju reading, playing with technology, or spending time with his wonderful (and significantly better) other half.

Mark Richards is an experienced hands-on software architect involved in the architecture, design, and implementation of microservices architectures, service-oriented architectures, and distributed systems. He's been in the software industry since 1983 and has significant experience and expertise in application, integration, and enterprise architecture. He's the author of numerous O'Reilly technical books and videos, including several books on microservices, the

Software Architecture Fundamentals video series, the Enterprise Messaging video series, and Java Message Service, second edition, and was a contributing author to 97 Things Every Software Architect Should Know. A speaker and trainer, he's given talks on a variety of enterprise-related technical topics at hundreds of conferences and user groups around the world.

Neal Ford is a director, software architect, and meme wrangler at Thoughtworks, a software company and a community of passionate, purpose-led individuals who think disruptively to deliver technology to address the toughest challenges, all while seeking to revolutionize the IT industry and create positive social change. He's an internationally recognized expert on software development and delivery, especially in the intersection of Agile engineering techniques and software architecture. Neal's authored 9 books and counting, a number of magazine articles, and dozens of video presentations (including a video on improving technical presentations) and spoken at hundreds of developer conferences worldwide. His topics of interest include software architecture, continuous delivery, functional programming, and cutting-edge software innovations. Check out his website, Nealford.com.