

O'REILLY®

Head First Software Architecture

A Learner's Guide to
Architectural Thinking

Raju Gandhi,
Mark Richards
& Neal Ford

Early
Release
RAW &
UNEDITED



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

Copyright © 2024 Defmacro Software L.L.C., Mark Richards and Neal Ford. All rights reserved.

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
- 2023-07-24: Third Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Head First Software Architecture, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source 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)

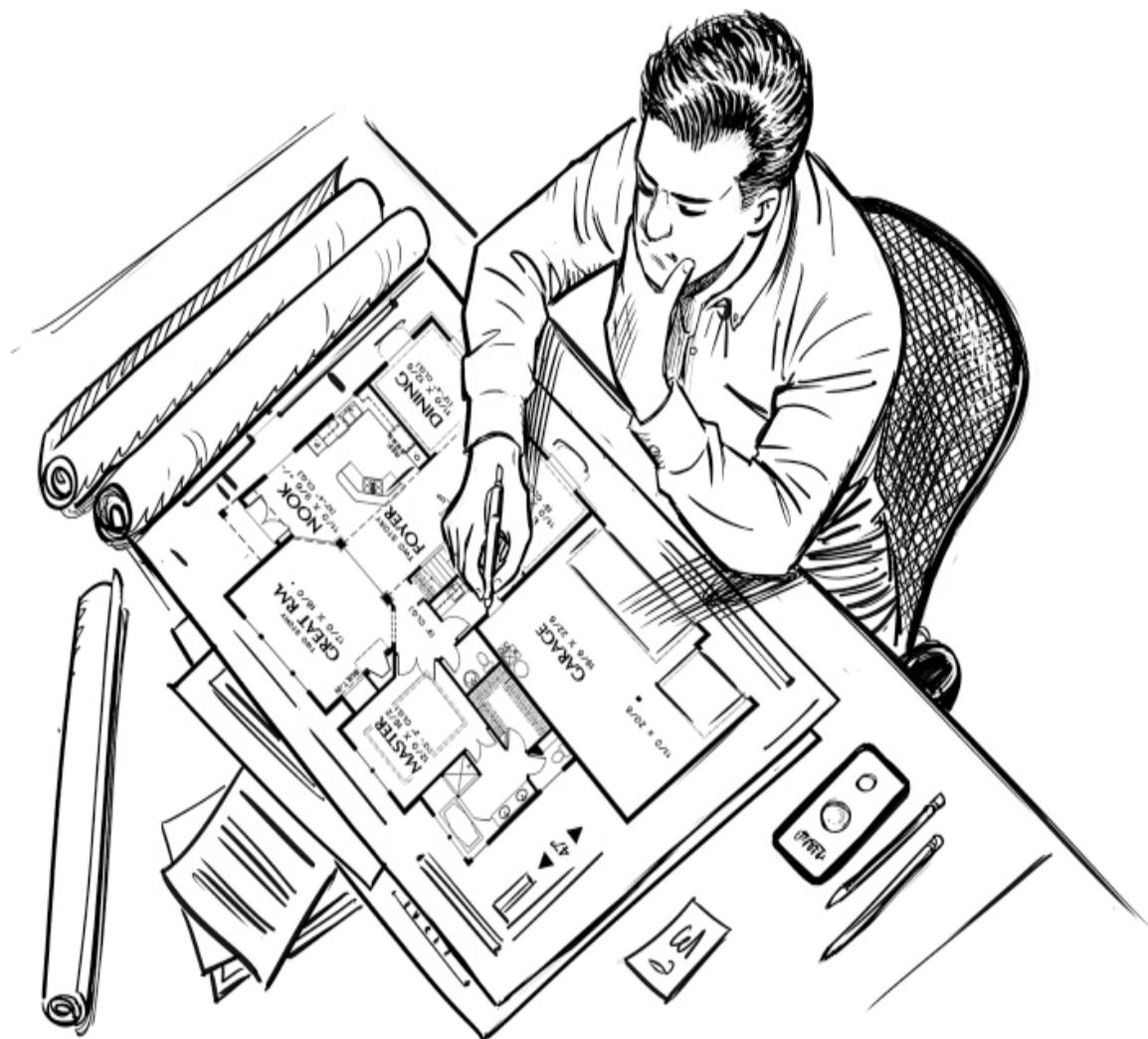


Figure 1-1.

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 technical 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.

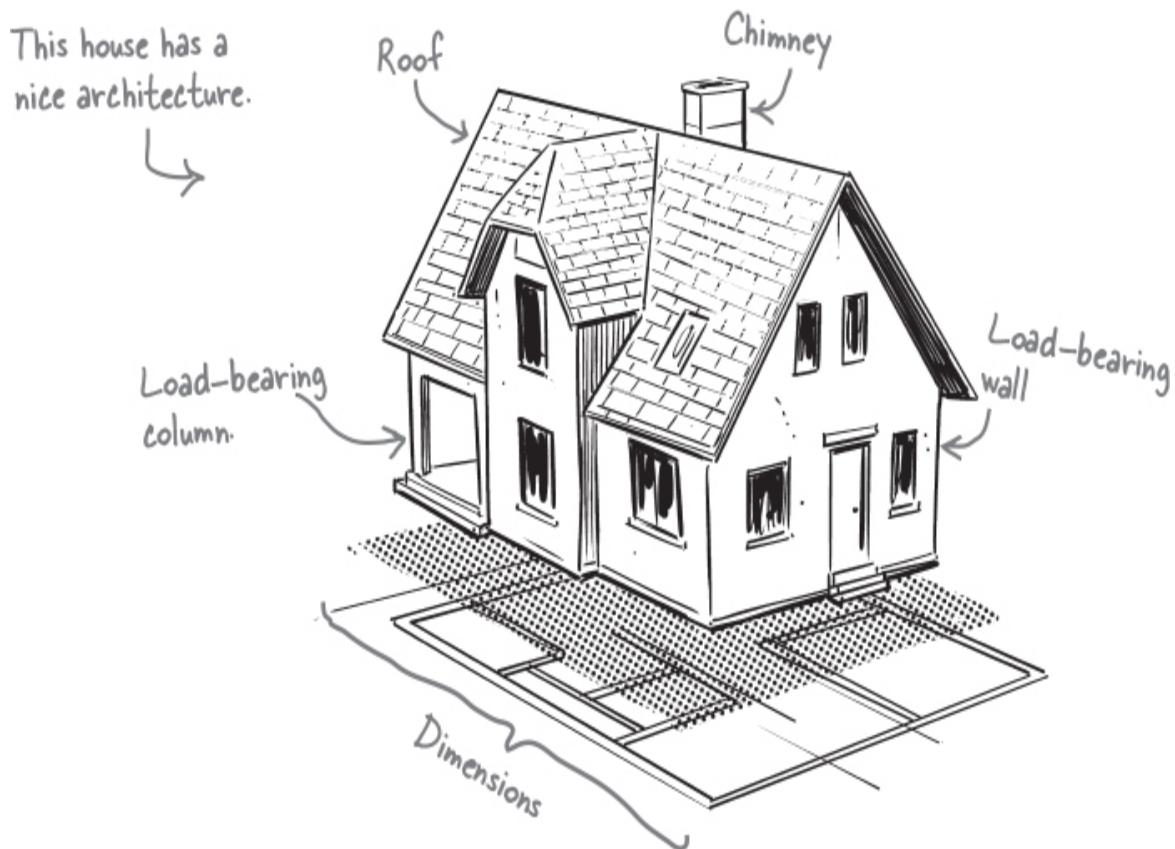


Figure 1-2.

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 reliable, or is too difficult to maintain? It's likely not enough emphasis was placed on its architecture.



Figure 1-3.

EXERCISE



Figure 1-4.

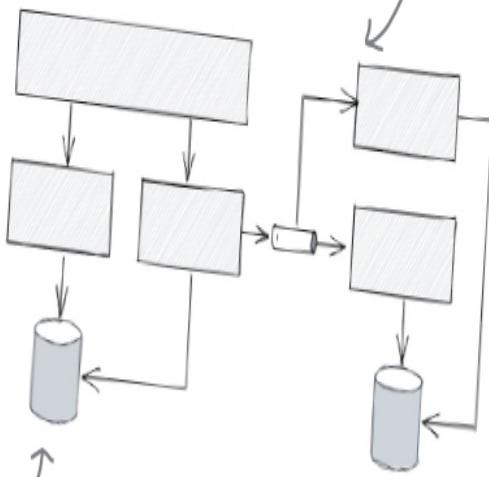
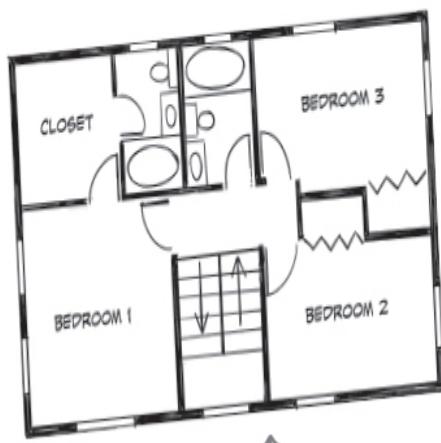
Gardening is often used as a metaphor for describing software architecture. Using the space below, can you describe how planning a garden might relate to software architecture? You can see what we came up with at the end of this chapter.

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.

Fun fact—a building plan used to be called a “blueprint,” but that term is now obsolete (at least within building architecture).



Both of these diagrams represent building plans.

Figure 1-5.

SHARPEN YOUR PENCIL

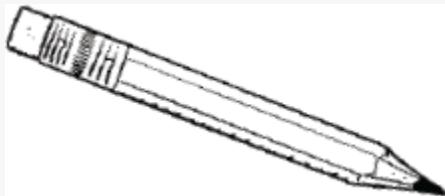


Figure 1-6.

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

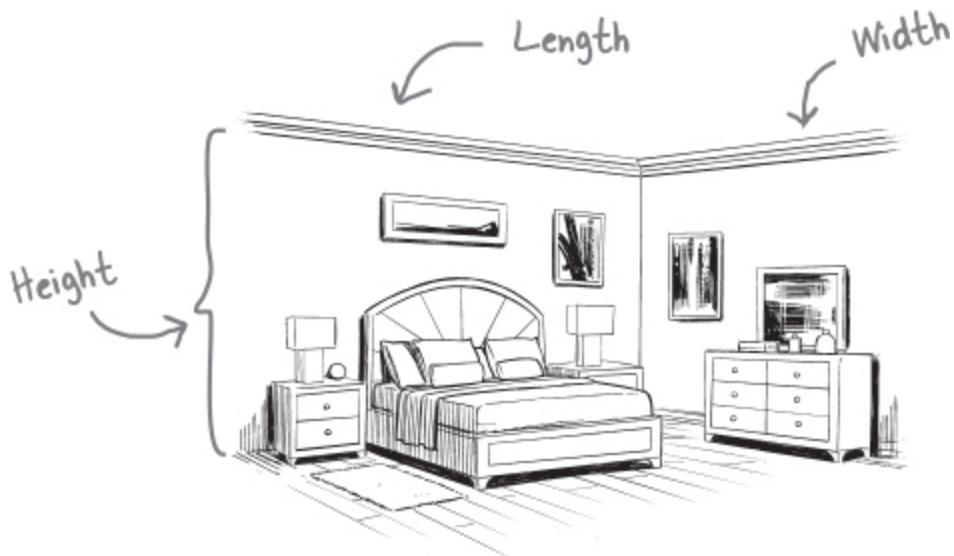


Figure 1-7.

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***.

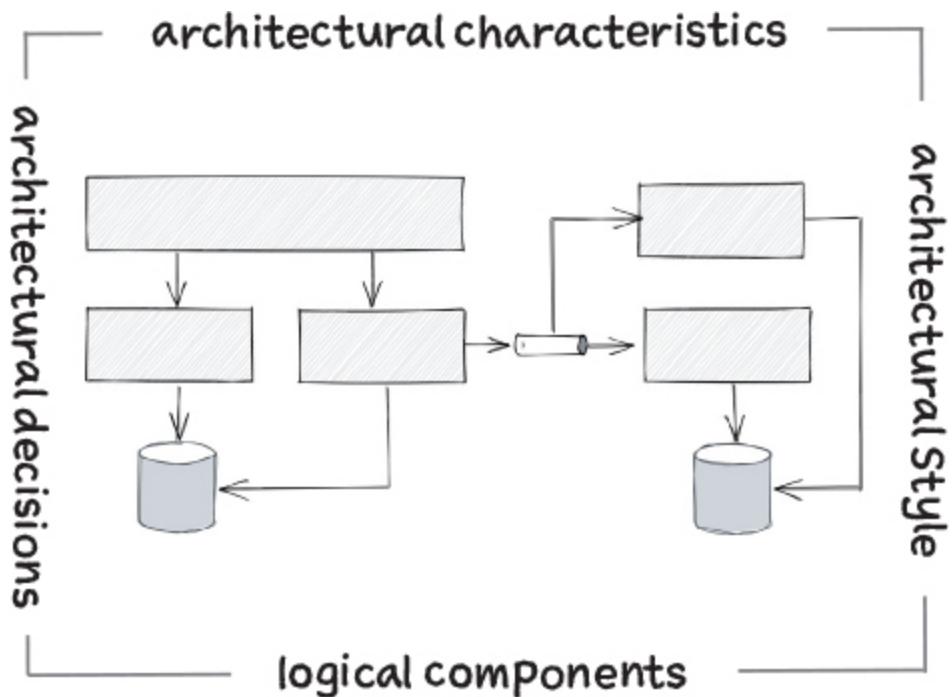


Figure 1-8.

① Architectural characteristics

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

② 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.

③ 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.

④ 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.

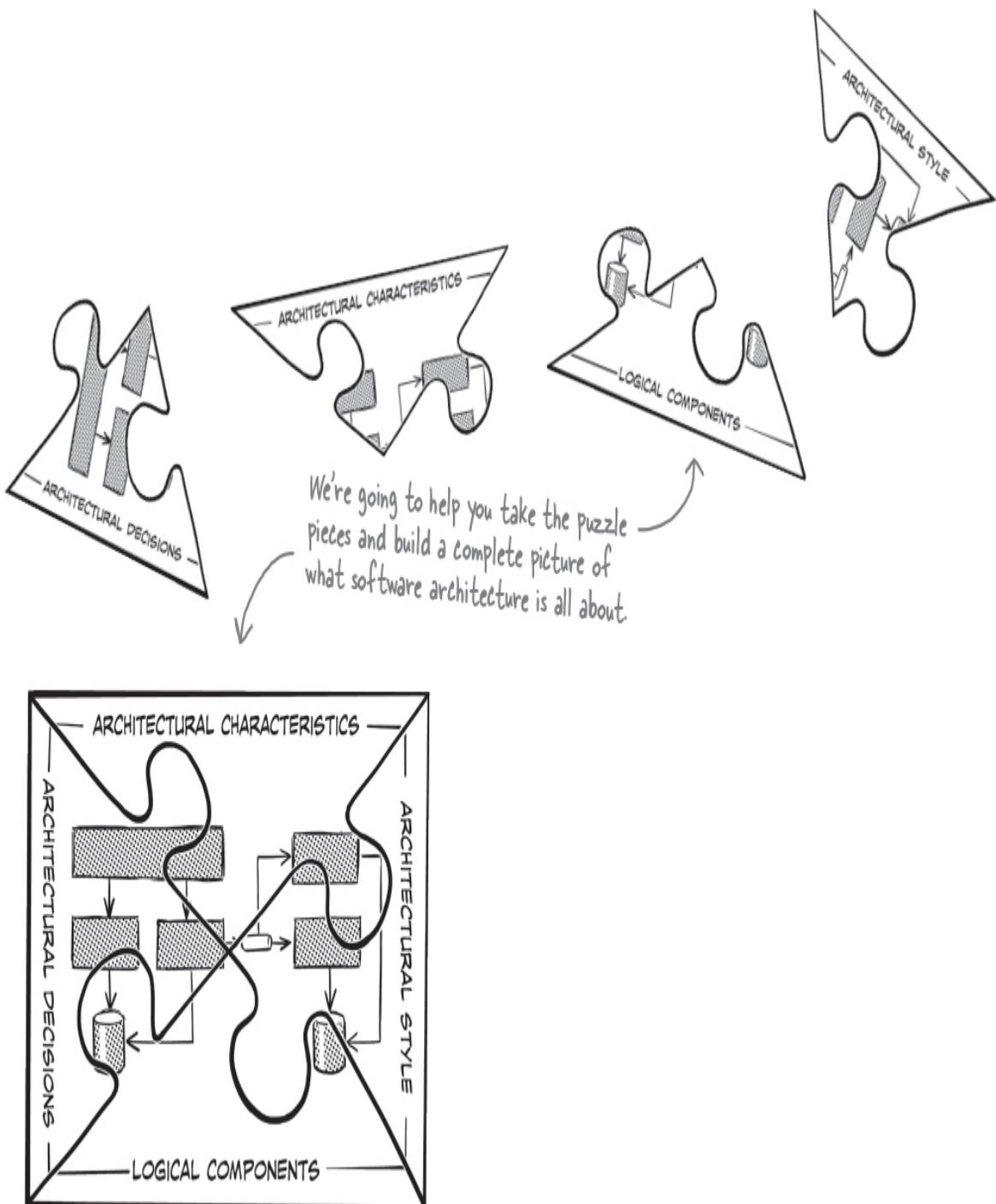


Figure 1-9.

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

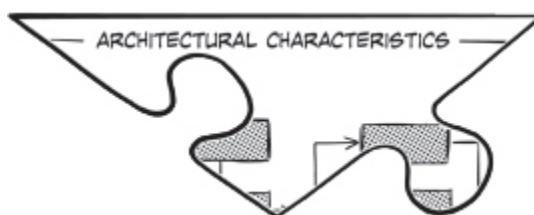


Figure 1-10.

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*).

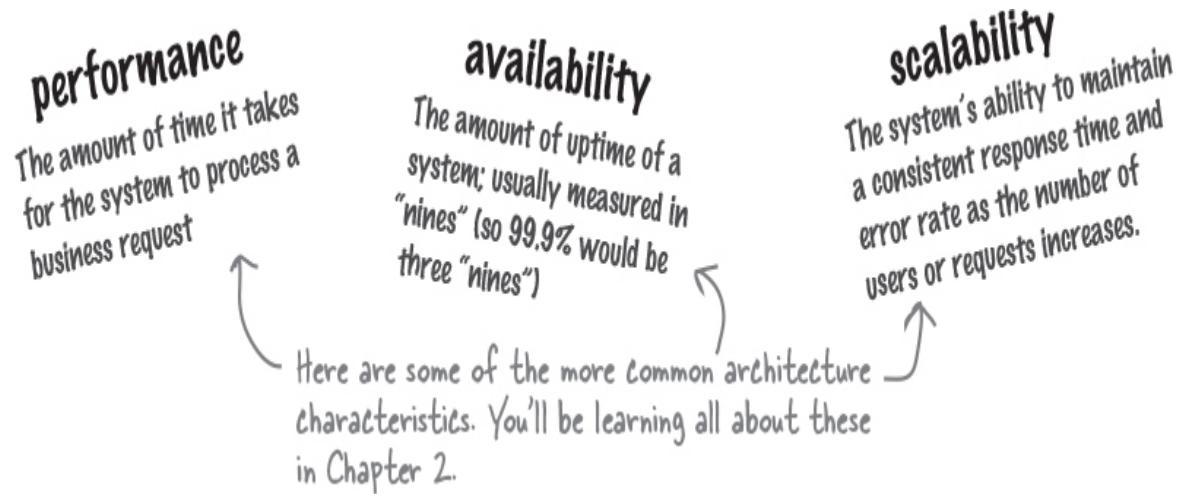


Figure 1-11.

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

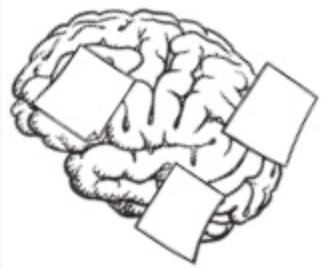


Figure 1-13.

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.

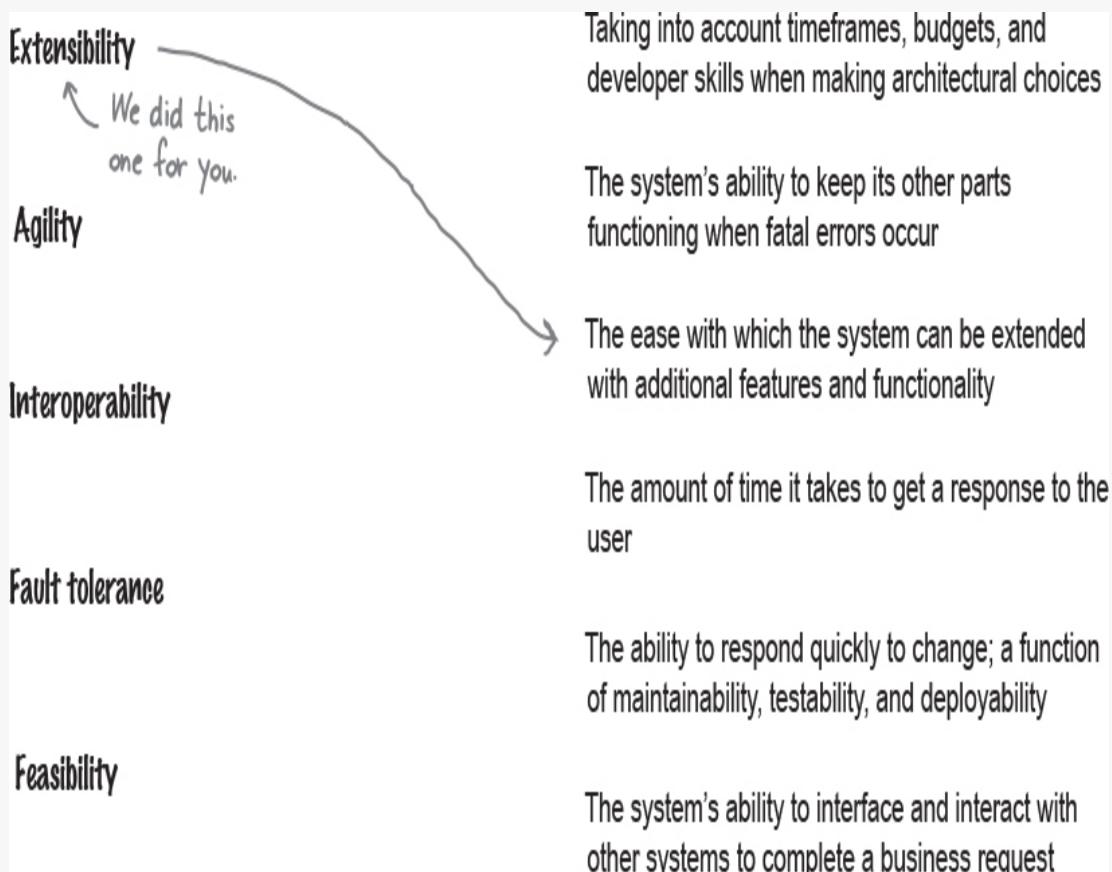


Figure 1-14.

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.

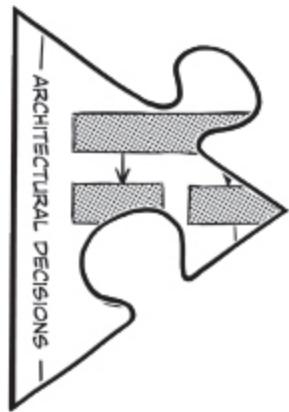


Figure 1-15.

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.

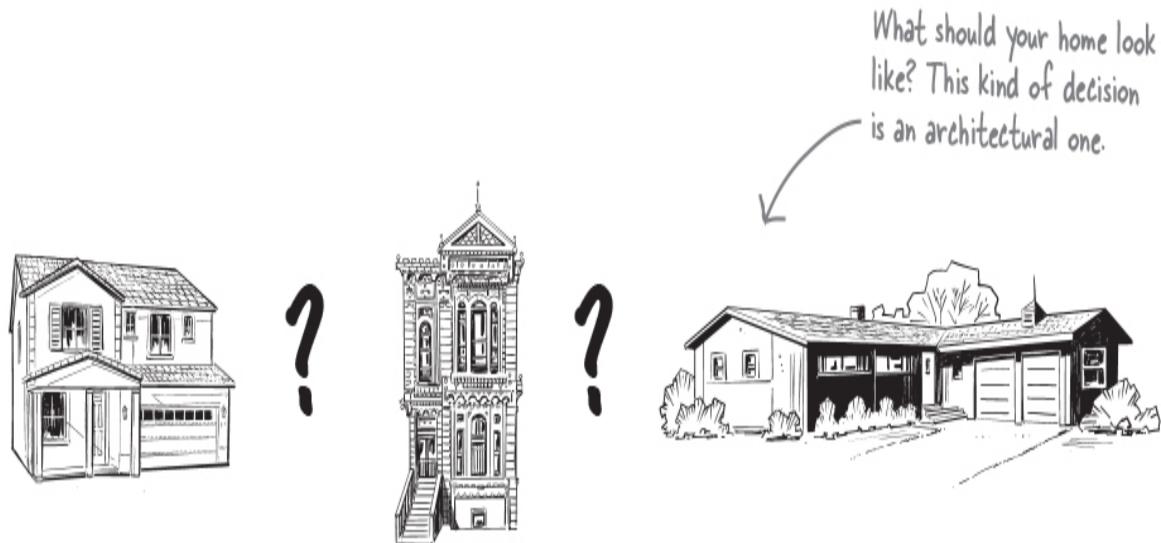


Figure 1-16.

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.

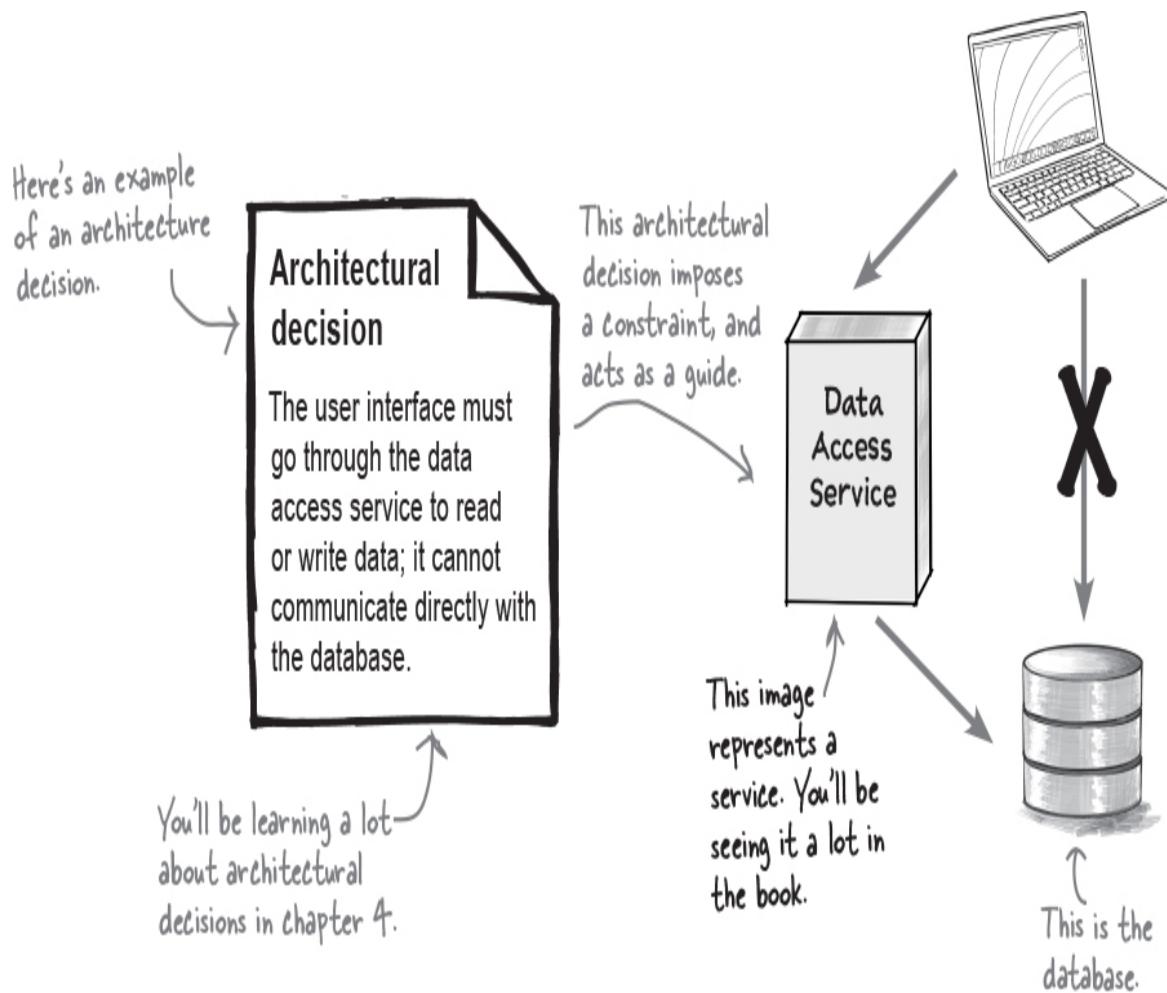


Figure 1-17.

MAKE IT STICK

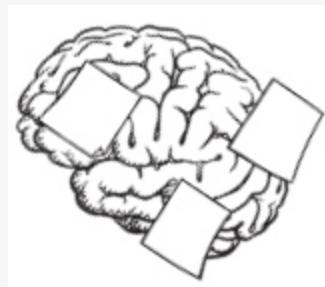


Figure 1-18.

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

Here's a hint—do you have questions
about why certain things are done
the way they are?

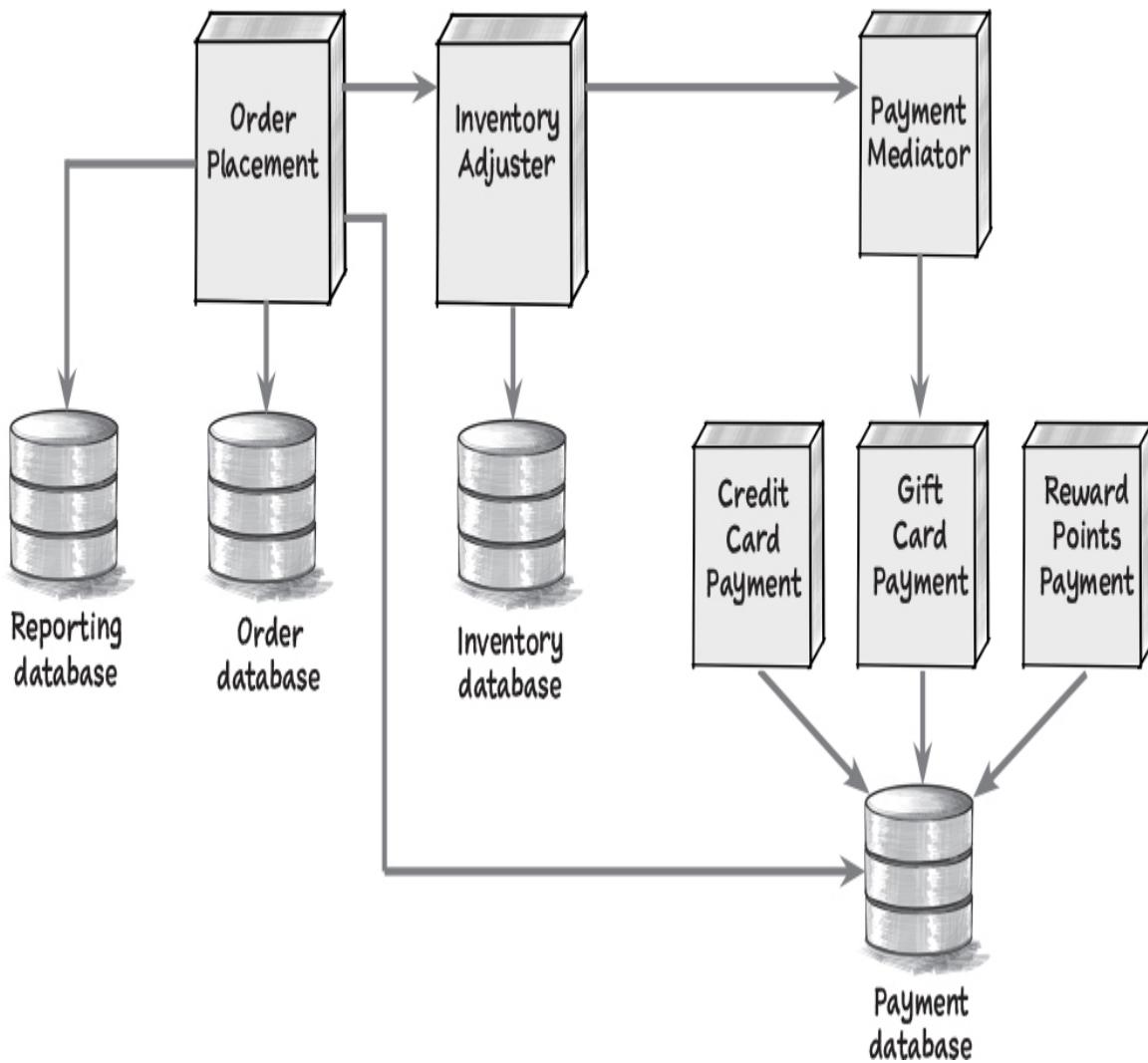


Figure 1-20.

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.

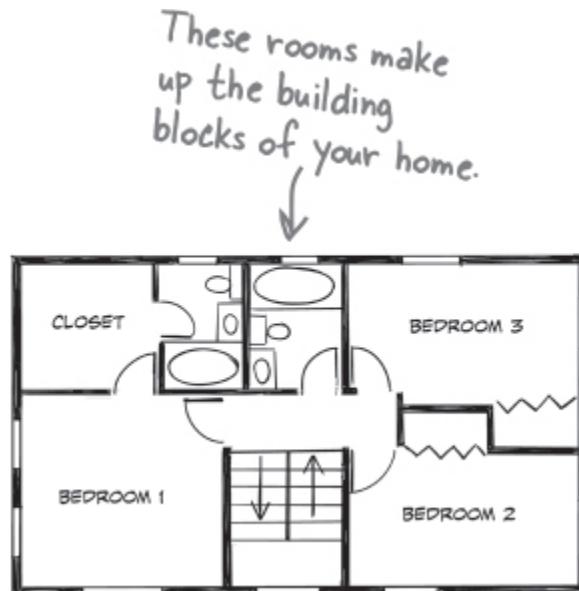


Figure 1-22.

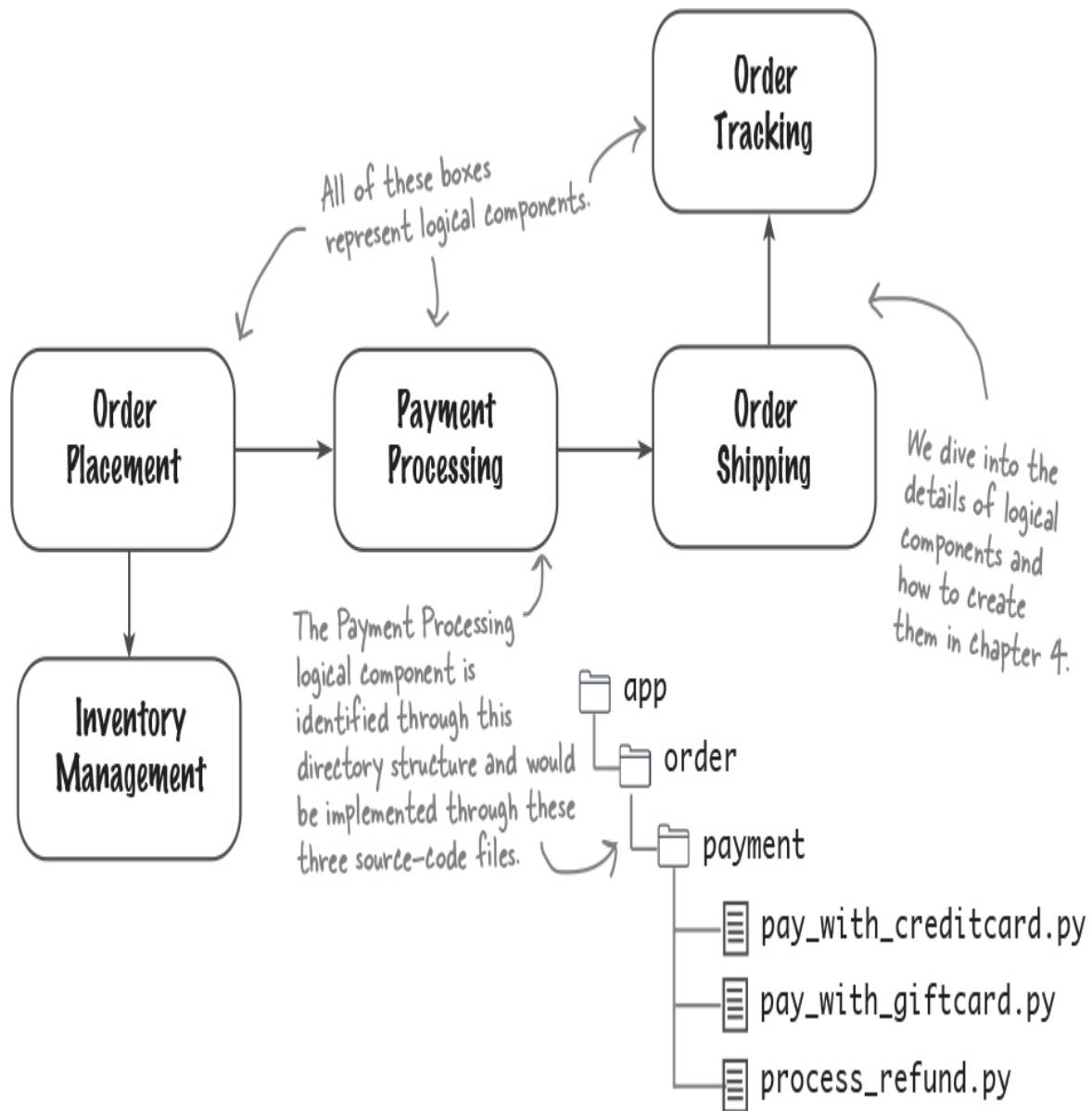


Figure 1-23.

SHARPEN YOUR PENCIL

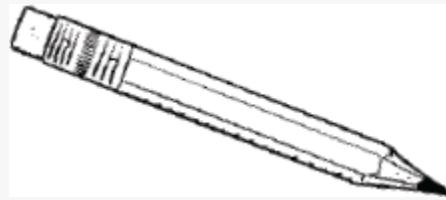


Figure 1-24.

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.

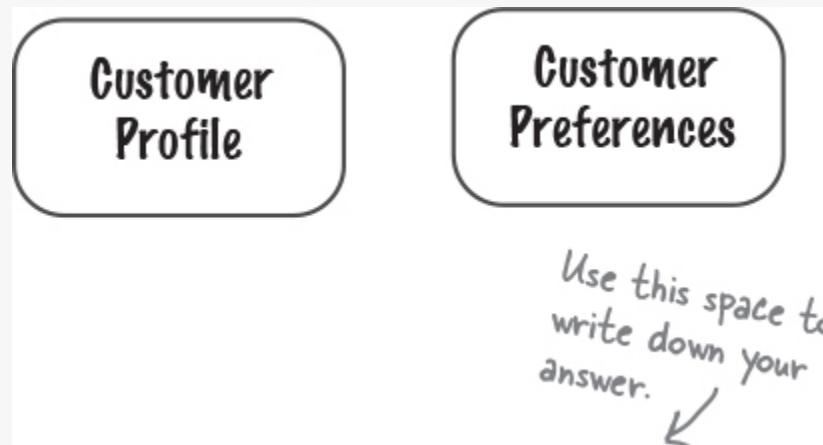


Figure 1-25.

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

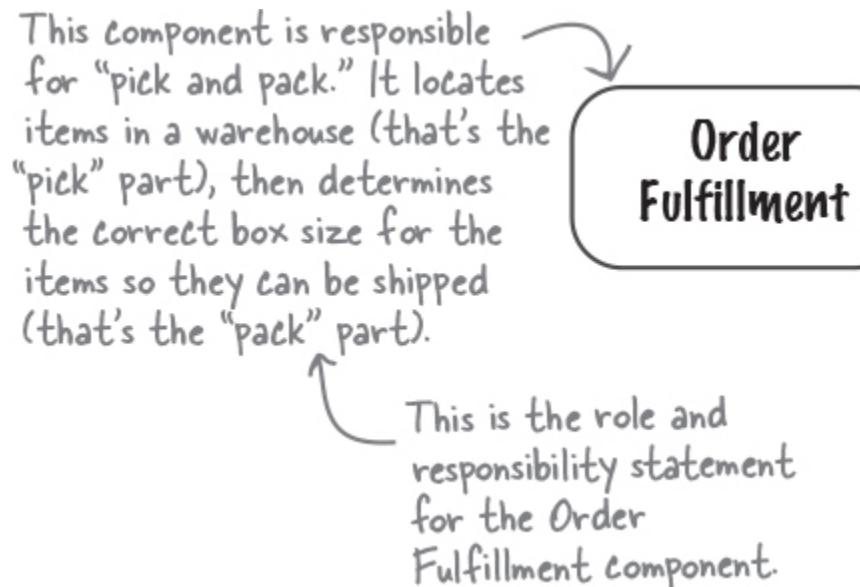


Figure 1-26.

MAKE IT STICK

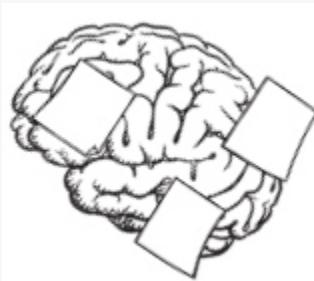


Figure 1-27.

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.

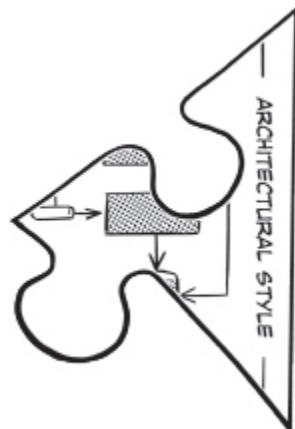


Figure 1-28.



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.

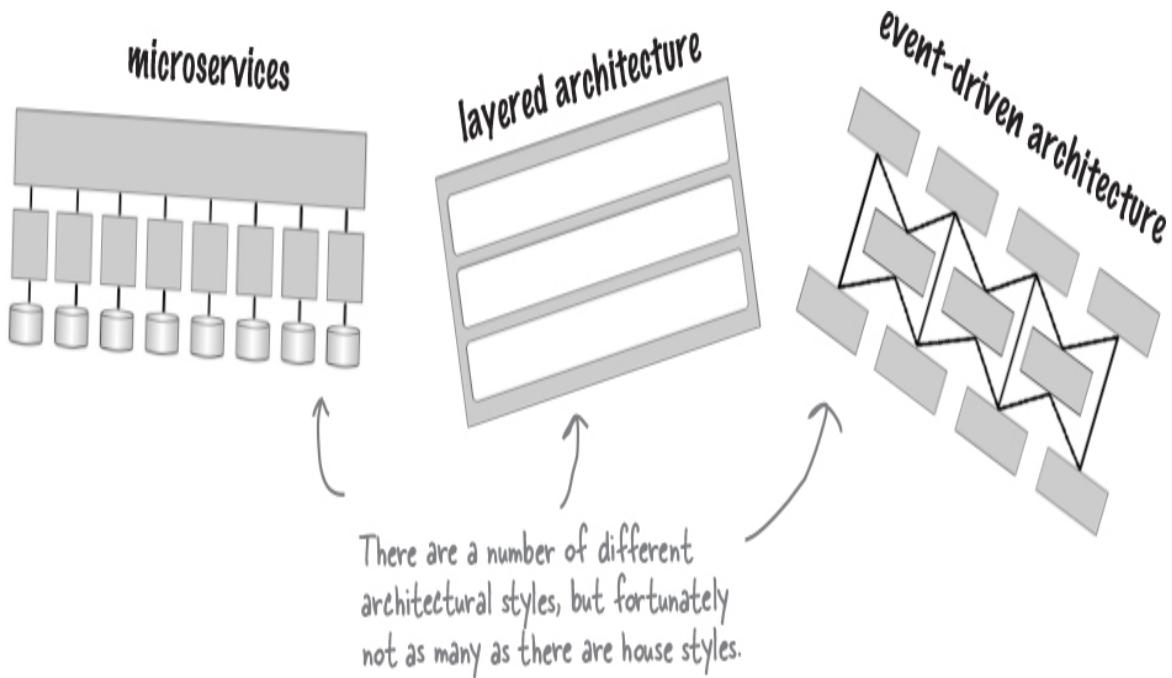


Figure 1-30.

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

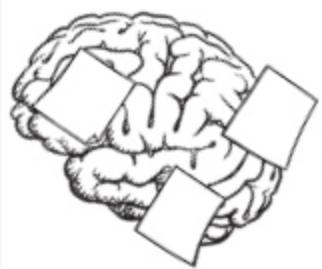


Figure 1-31.

Styles shape the system and help serve its purposes—
You might choose a monolith or microservices.

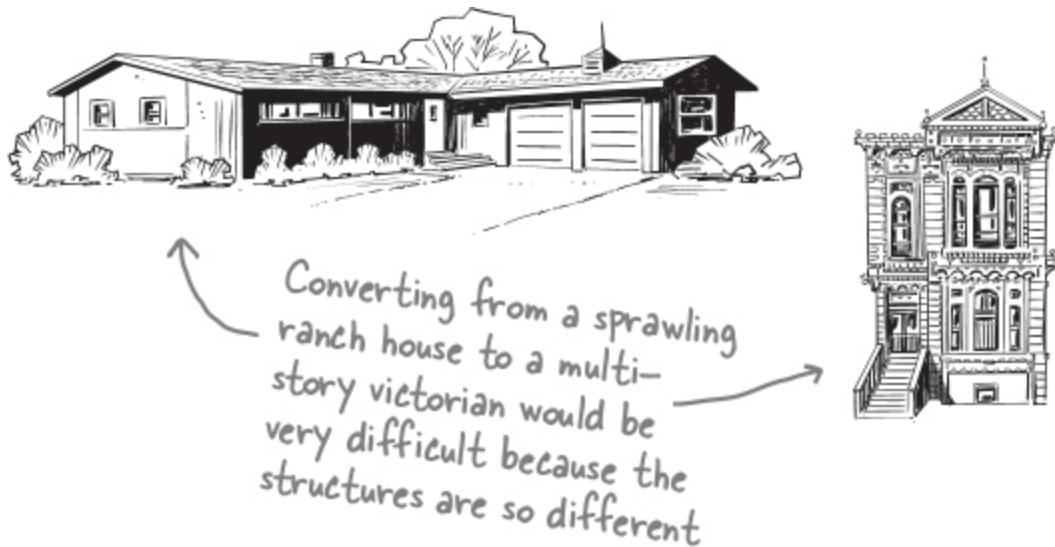


Figure 1-32.

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

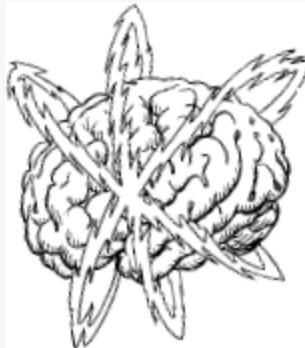


Figure 1-33.

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?



Fun fact: A lion doesn't have much stamina and can only run fast in short bursts. If you can last longer than the lion chasing you, then you just might survive.

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.

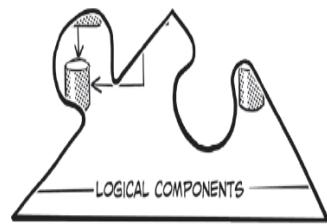


Figure 1-35.

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

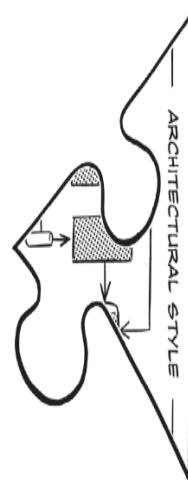
The product owner insists that we get new features and bug fixes out to our customers as fast as possible.

Our system uses an event-driven architecture.



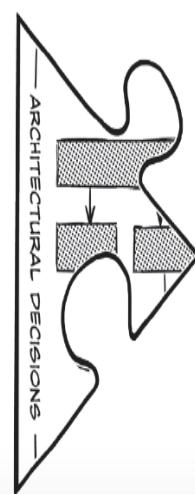
We need to support up to 300,000 concurrent users in this system.

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, whereas 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.

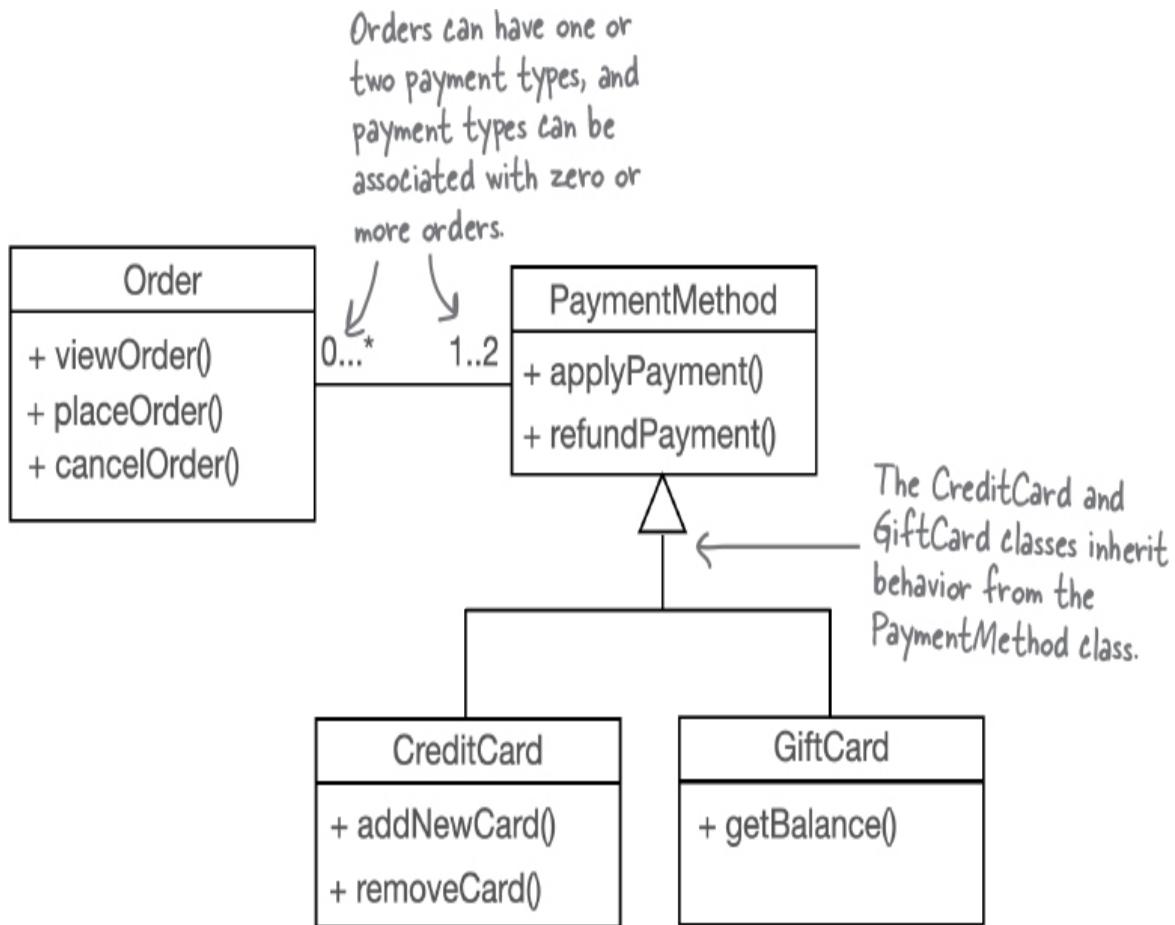


Figure 1-39.

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.

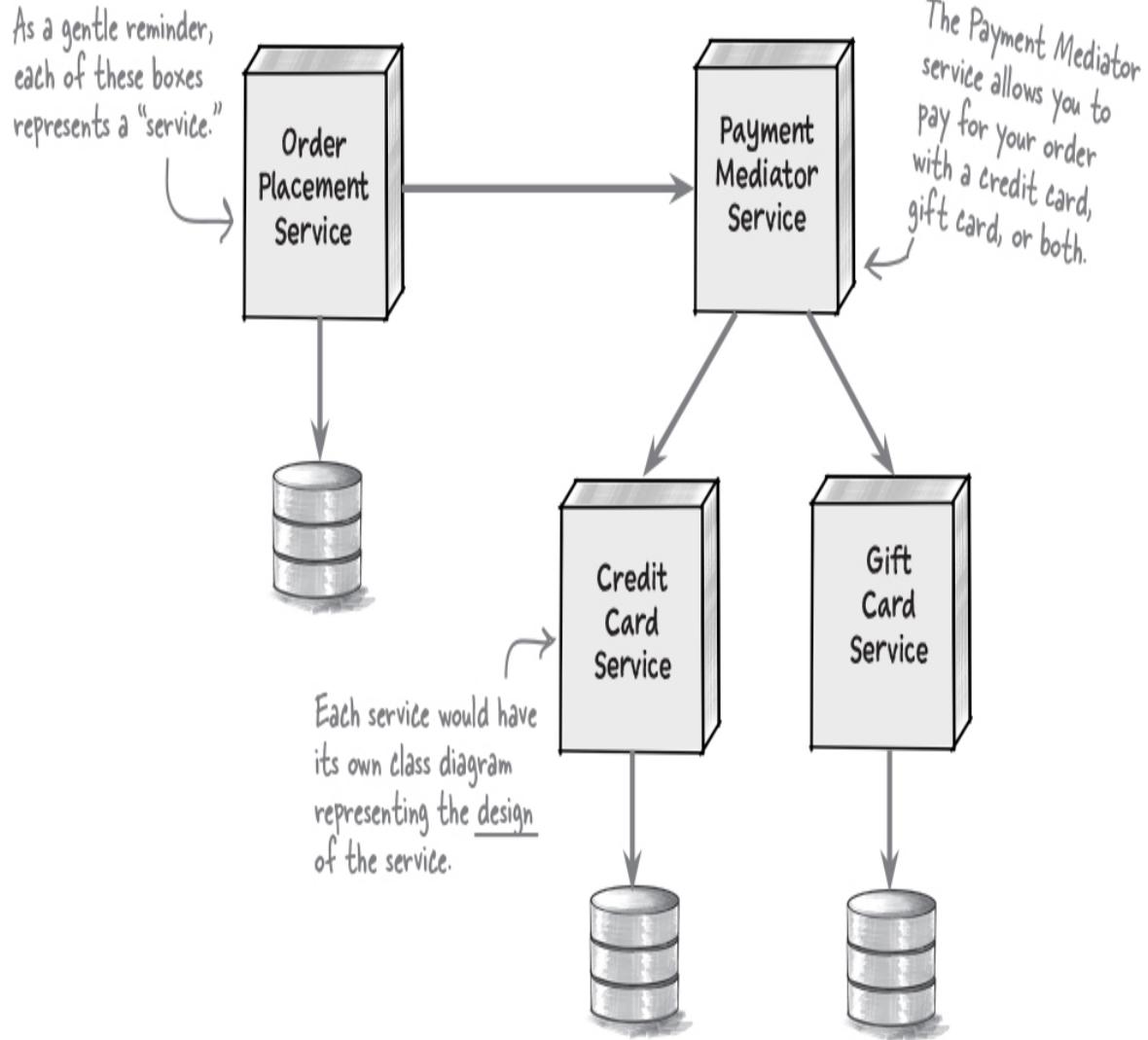


Figure 1-40.

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 databases**
- 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.

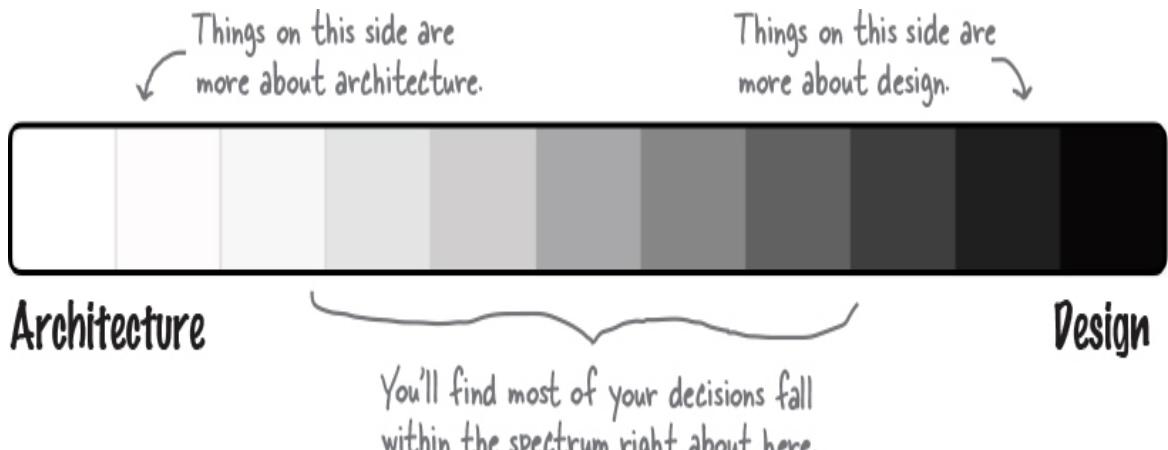


Figure 1-42.

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

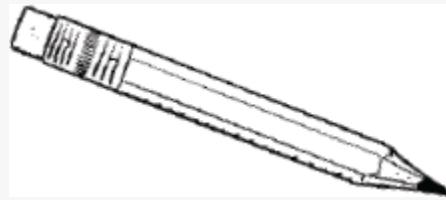


Figure 1-43.

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”.

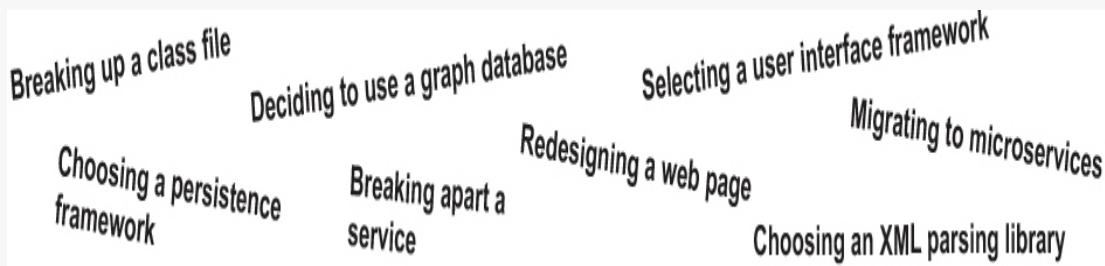


Figure 1-44.



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.

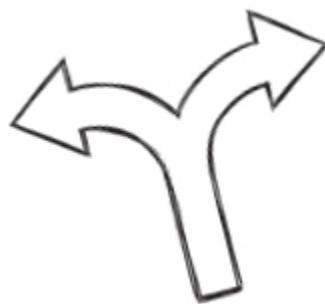


Figure 1-48.

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.

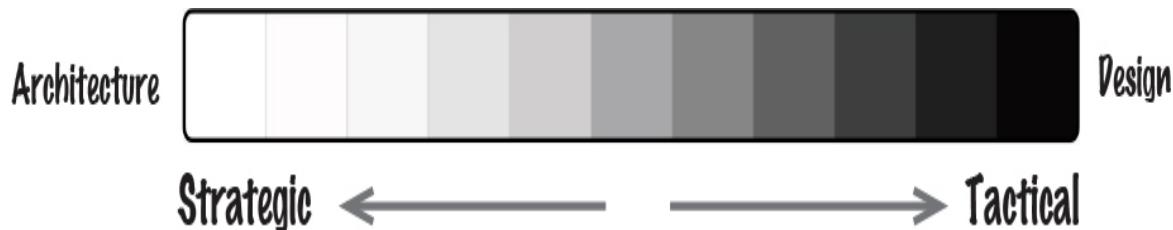


Figure 1-50.



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

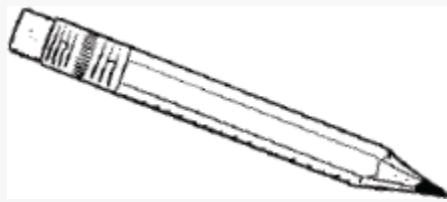
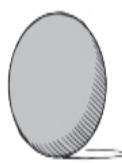


Figure 1-52.

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.



Picking a
programming language
for your new project



Deciding to get
your first dog



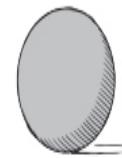
Deploying on the
cloud or on-premises



Redesigning your
user interface



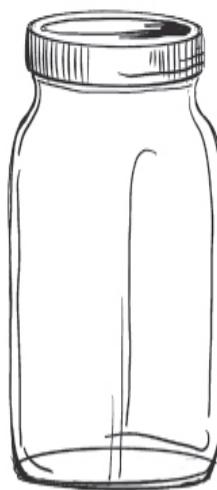
Migrating
your system to
microservices



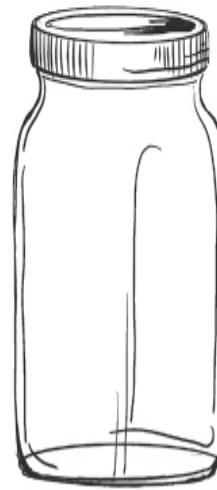
Choosing a
parsing library



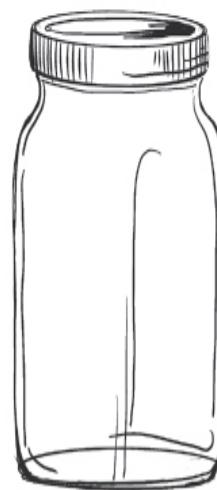
Using a design
pattern



Strategic



Somewhere
in between



Tactical

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.

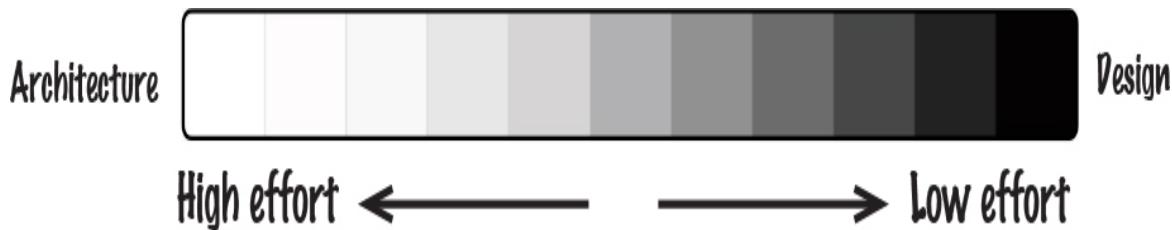


Figure 1-54.

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.

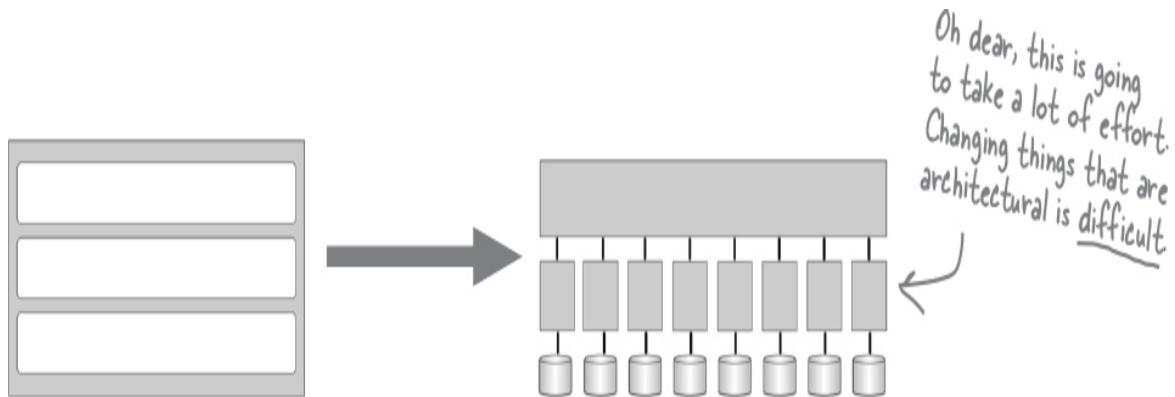


Figure 1-55.

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.

Changing the layout
of the fields on a
web page is more
about appearance
than structure—yet
another reason
why this would be
considered design.

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.

Draw arrows to put the high-effort tasks on the top of the page, and the lower-effort ones toward the bottom of the page.

High effort

Resolving a merge conflict in Git

Replacing your user interface framework

Migrating your system to a cloud environment

Deciding which mustard to buy

Renaming a method or function

Breaking apart a single service into separate ones

Moving from a relational to a graph database

Breaking apart a class file

Low effort

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.



More significant trade-offs ← → Less significant Trade-offs

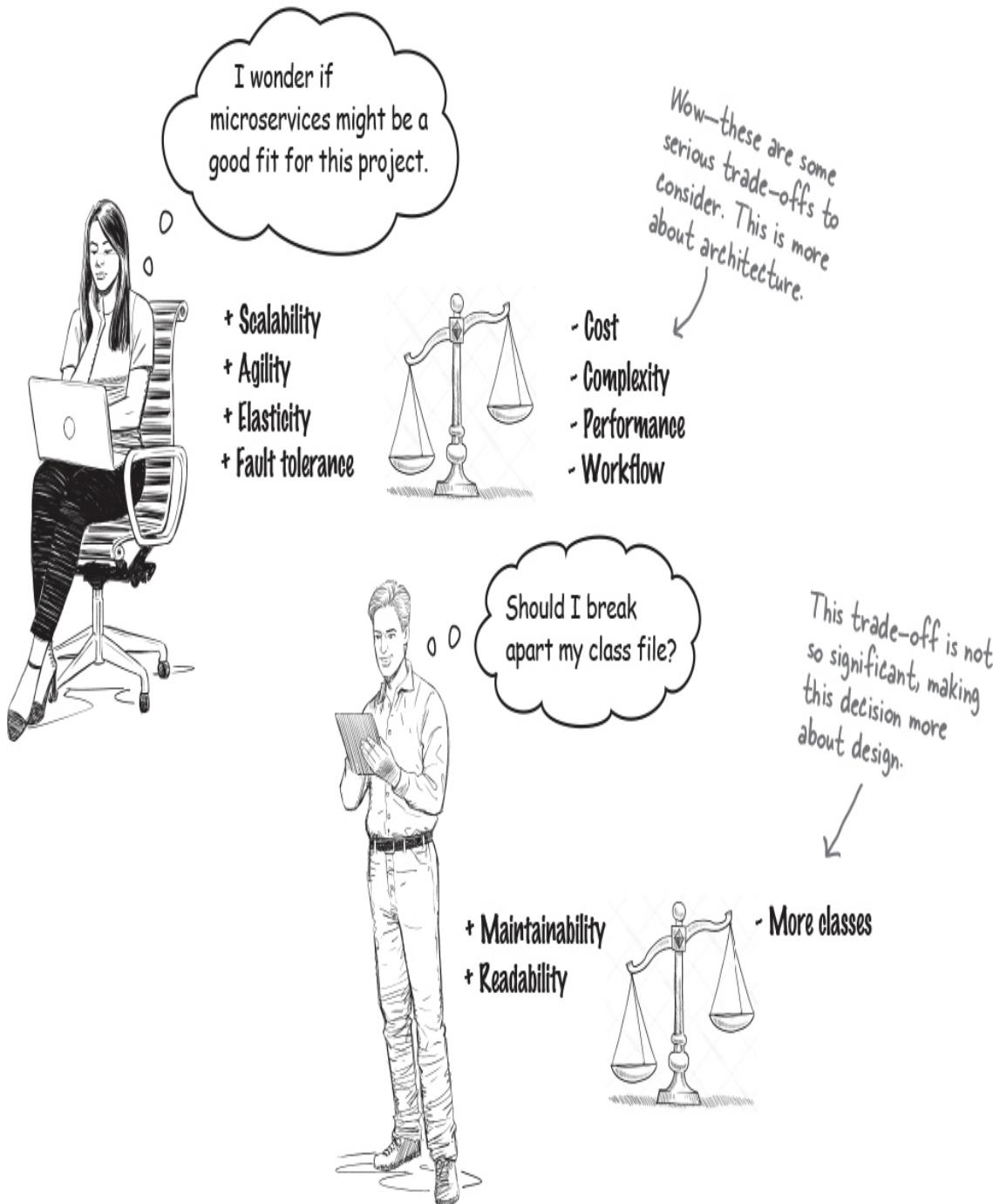


Figure 1-59.

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?

<input type="checkbox"/> Yes	<input type="checkbox"/> No	Picking out what clothes to wear to work today
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing to deploy on the cloud or on premise
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Selecting a user interface framework
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Naming a variable in a class file
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing between vanilla and chocolate ice cream
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Deciding which architecture style to use
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing between REST and messaging
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Using full data or only keys for the message payload
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Selecting an XML parsing library
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Deciding whether or not to break apart a service
<input type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing between atomic or distributed transactions
<input type="checkbox"/> Yes	<input type="checkbox"/> 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.

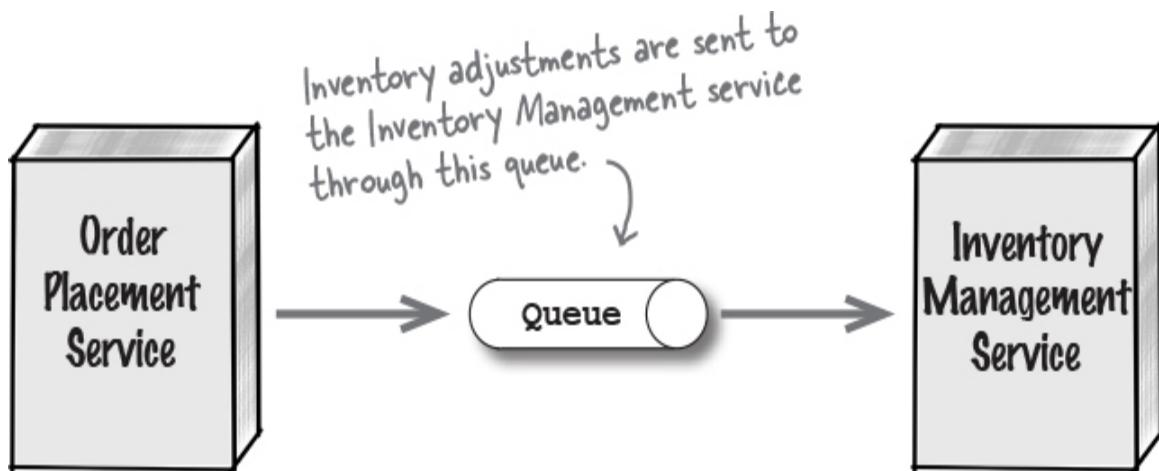


Figure 1-61.

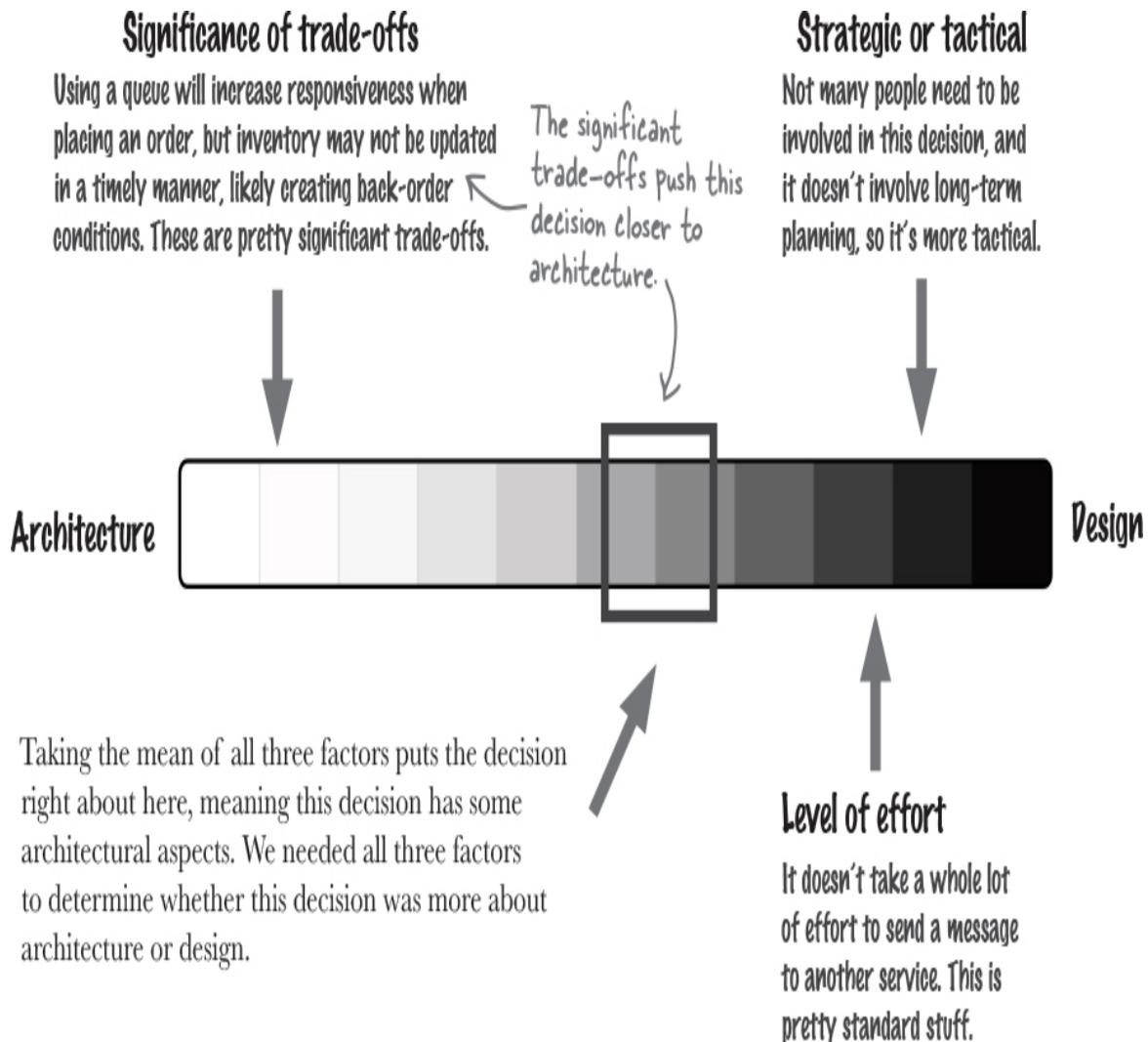


Figure 1-62.

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	<i>Trade-offs</i> 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?

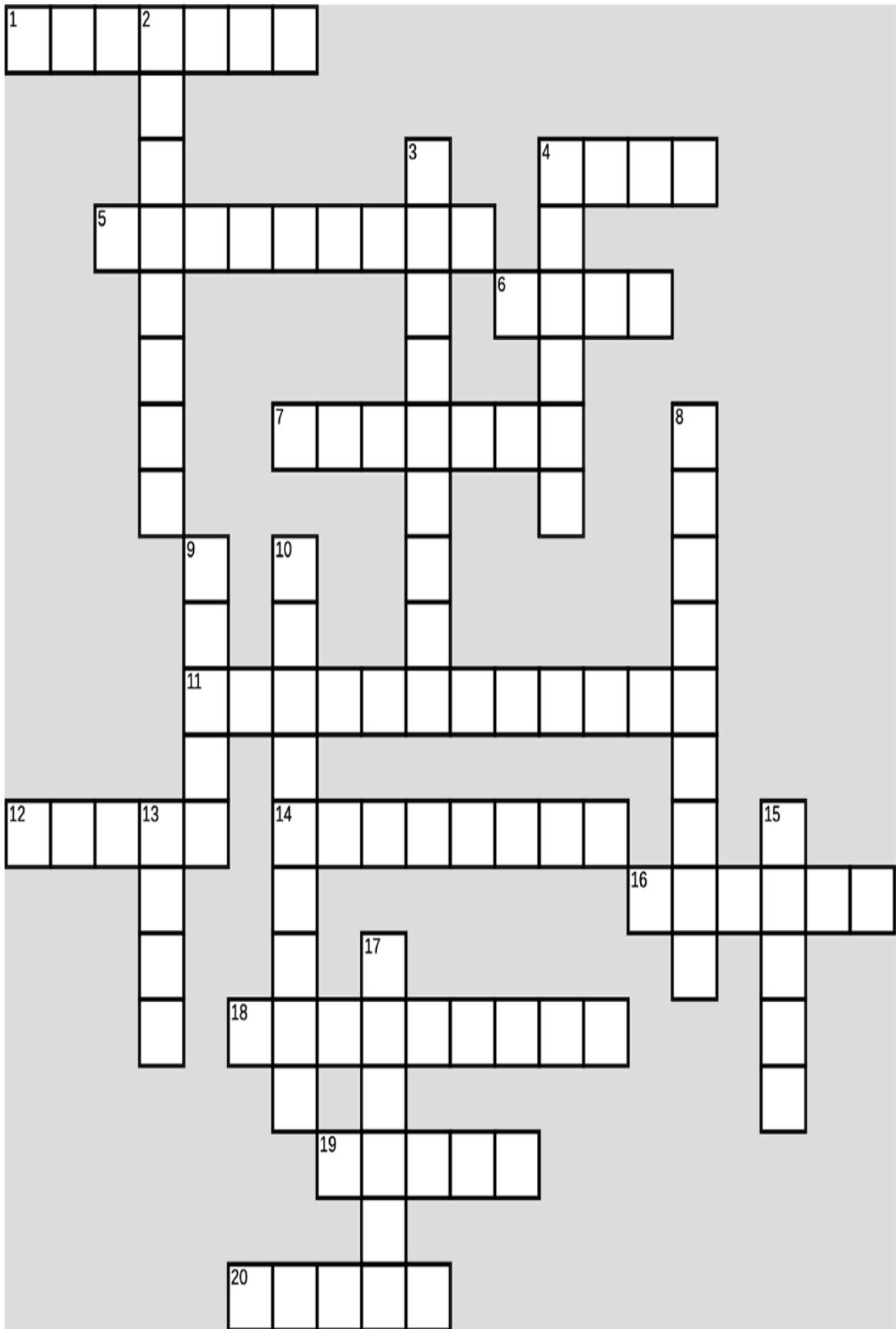


Figure 1-64.

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

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

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

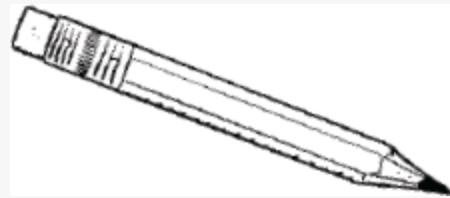


Figure 1-66.

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.

- Changing the font size in a window on the user interface screen
- Making changes quickly ← This is known as agility in architecture.
- Handling thousands of concurrent users ← This is known as elasticity.
- Encrypting user passwords stored in the database
- Interacting with many external systems to complete a business request ← This is known as interoperability.

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.

- How services communicate with each other
 - The platform and language the services should be implemented in
 - Which services access which databases
 - How many services and databases there are
- How something should be implemented is a design perspective.*

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](#)”.

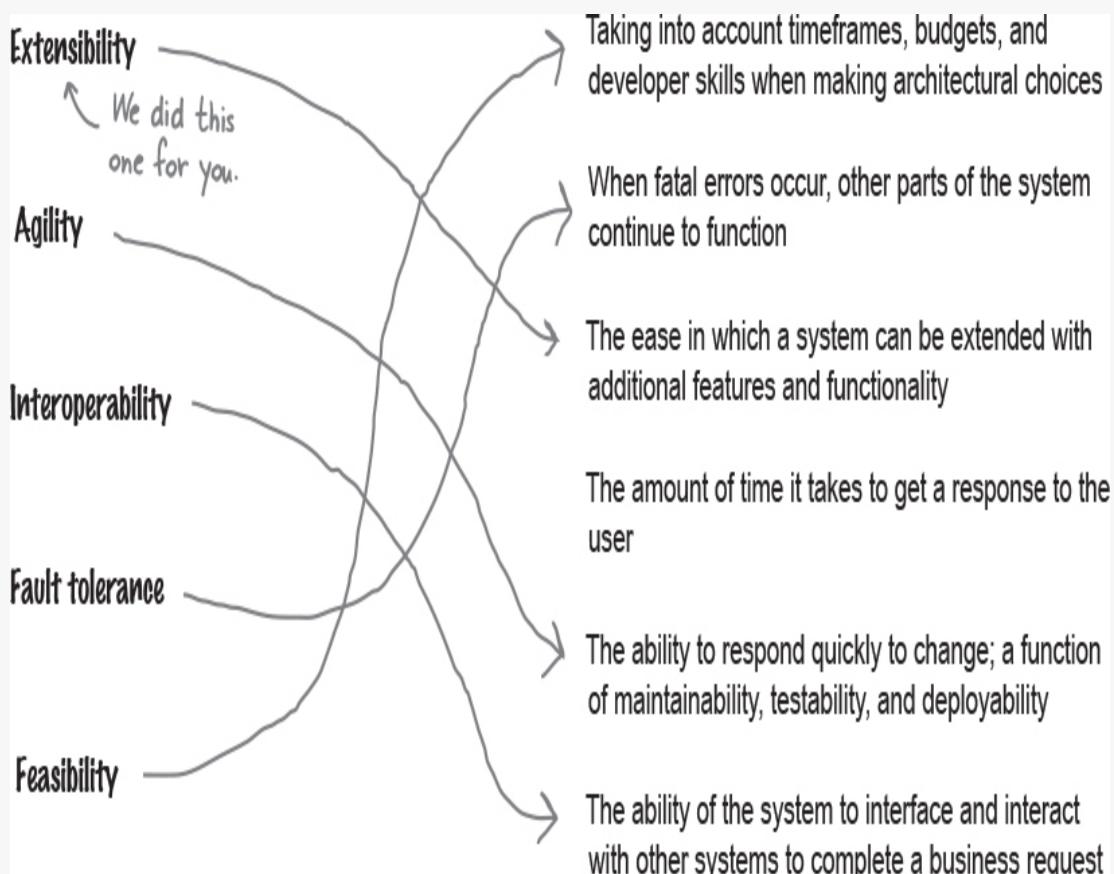


Figure 1-72.

From “[Sharpen your pencil](#)”

SHARPEN YOUR PENCIL SOLUTION

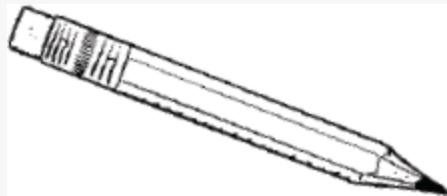


Figure 1-73.

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”.

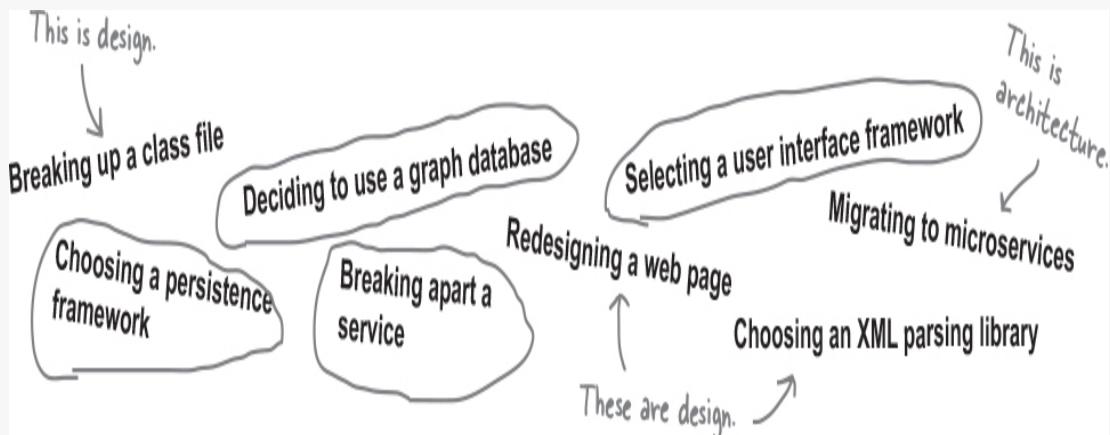


Figure 1-74.

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 anything that you think might be an architectural 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.

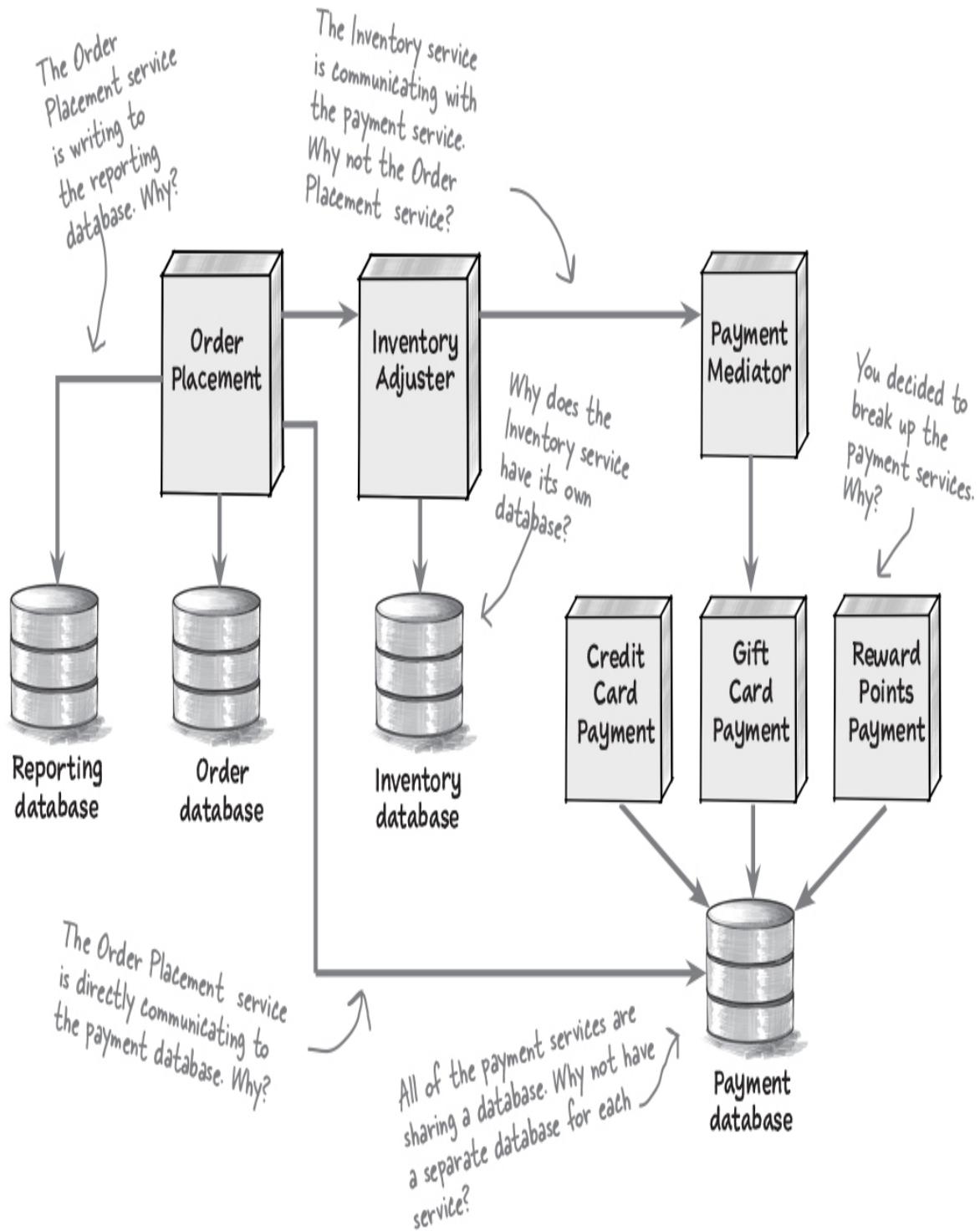


Figure 1-76.

From “Sharpen your pencil”

SHARPEN YOUR PENCIL SOLUTION

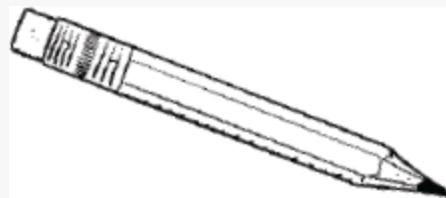


Figure 1-77.

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.

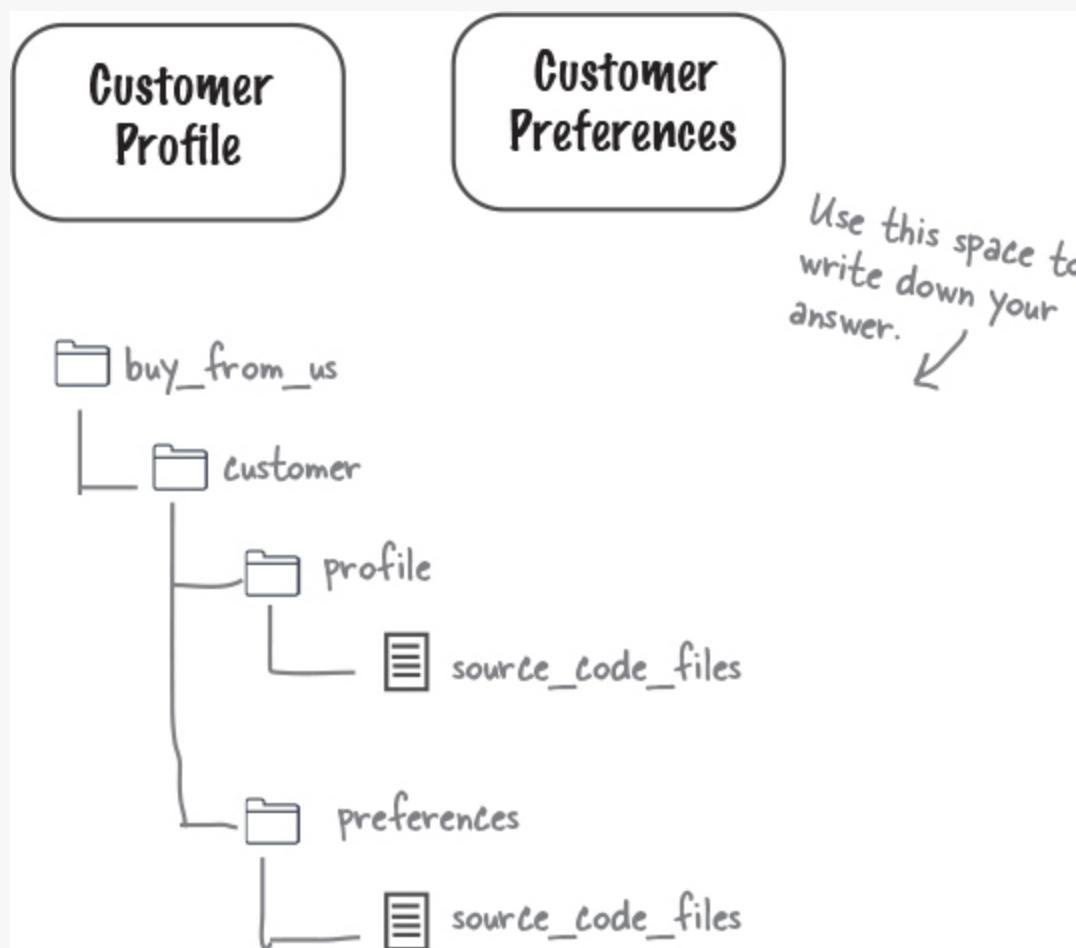


Figure 1-78.

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*.

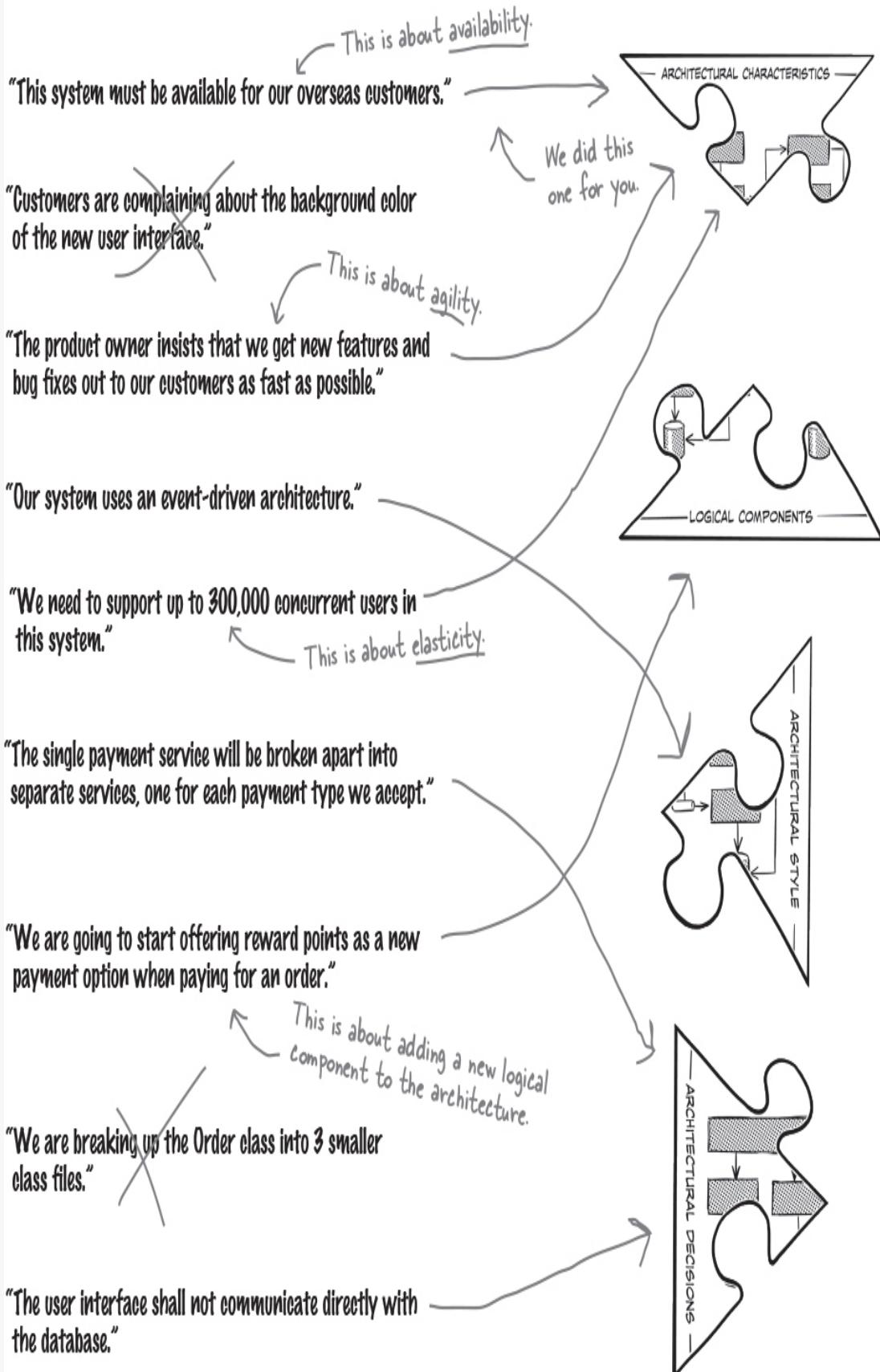


Figure 1-79.

From “Sharpen your pencil”

SHARPEN YOUR PENCIL SOLUTION

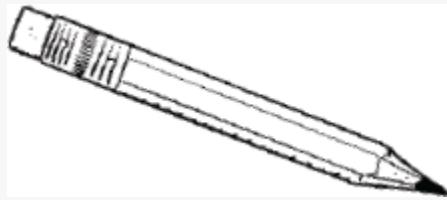


Figure 1-80.

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.

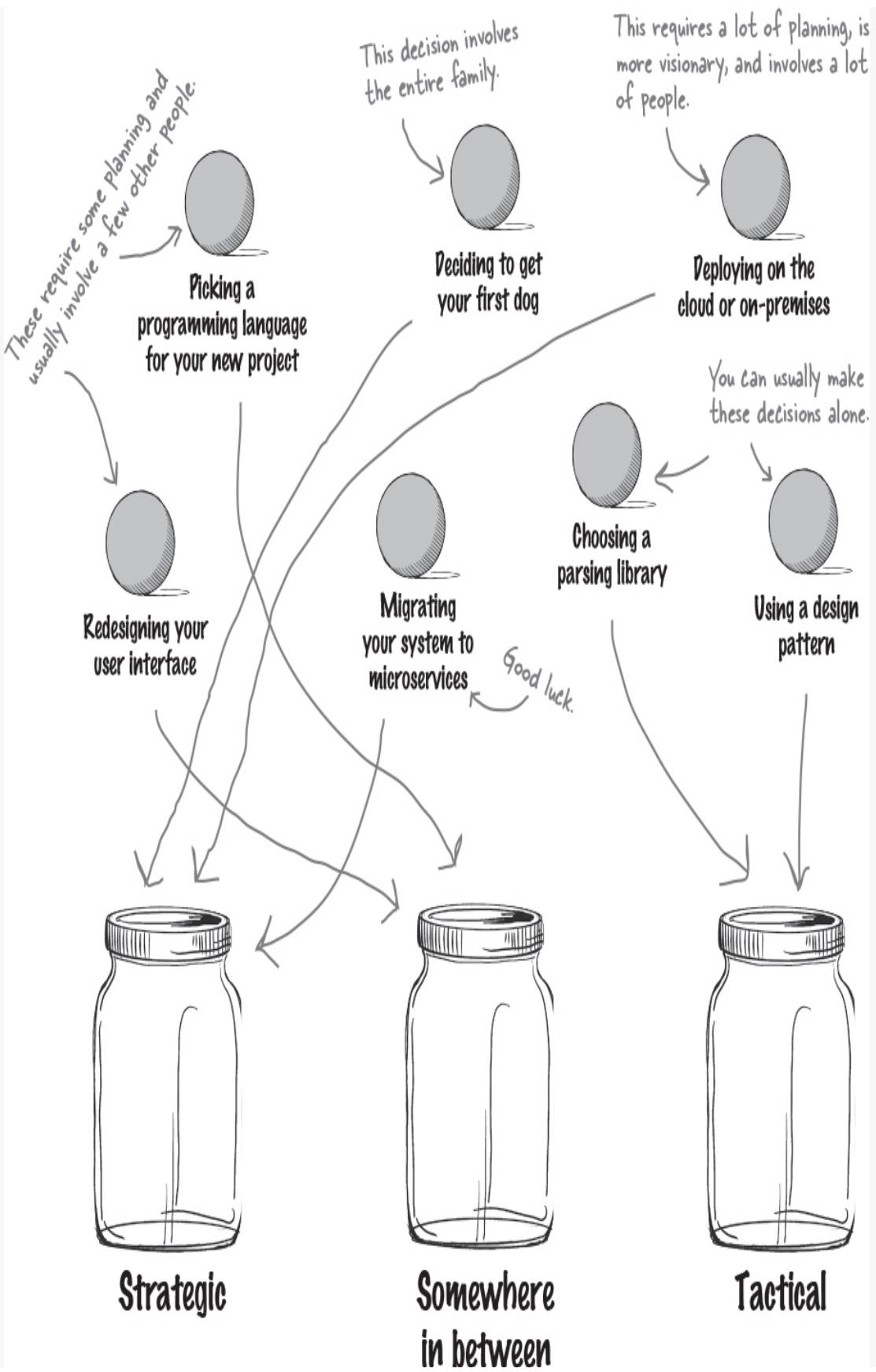


Figure 1-81.

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.

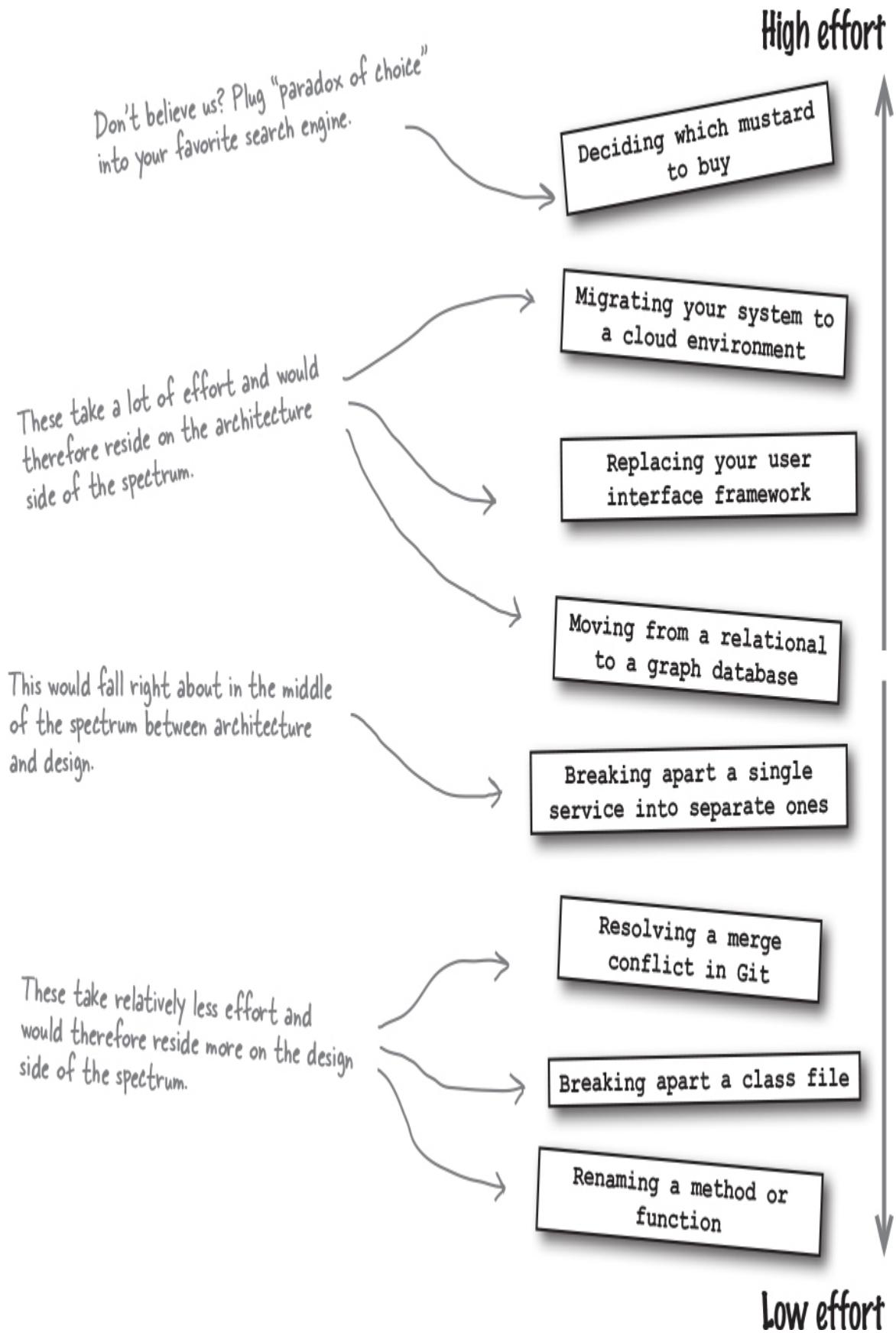


Figure 1-83.

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.

Significant Tradeoffs?

Okay, so maybe this is a difficult decision sometimes.

- | | | |
|---|--|--|
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Picking out what clothes to wear to work today |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | Choosing to deploy on the cloud or on-premisis |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Selecting a user interface framework |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Deciding on the name of a variable in a class file |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Choosing between vanilla and chocolate ice cream |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | Deciding which architecture style to use |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | Choosing between REST and messaging |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | Using full data or only keys for the message payload |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Selecting an XML parsing library |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No | Deciding whether or not to break apart a service |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Choosing between atomic or distributed transactions |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No | Deciding whether or not to go out to dinner tonight |

There are certainly trade-offs here, so this one could go either way.

These can impact scalability, performance, and overall maintainability.

Are you getting hungry yet?

This can impact data integrity and data consistency, but also scalability, and performance.

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”.

Significant Tradeoffs?

<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	Picking out what clothes to wear to work today
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing to deploy on the cloud or on-premisis
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Selecting a user interface framework
<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	Deciding on the name of a variable in a class file
<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	Choosing between vanilla and chocolate ice cream
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Deciding which architecture style to use
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing between REST and messaging
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Using full data or only keys for the message payload
<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	Selecting an XML parsing library
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Deciding whether or not to break apart a service
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	Choosing between atomic or distributed transactions
<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	Deciding whether or not to go out to dinner tonight

Figure 1-87.

From “True or False”

TRUE OR FALSE SOLUTION

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.

This is backwards.

True

False

Most decisions are purely about architecture or design. Very few exist along a spectrum between architecture and design.

Most decisions lie within the 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.

This is backwards.

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.

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?

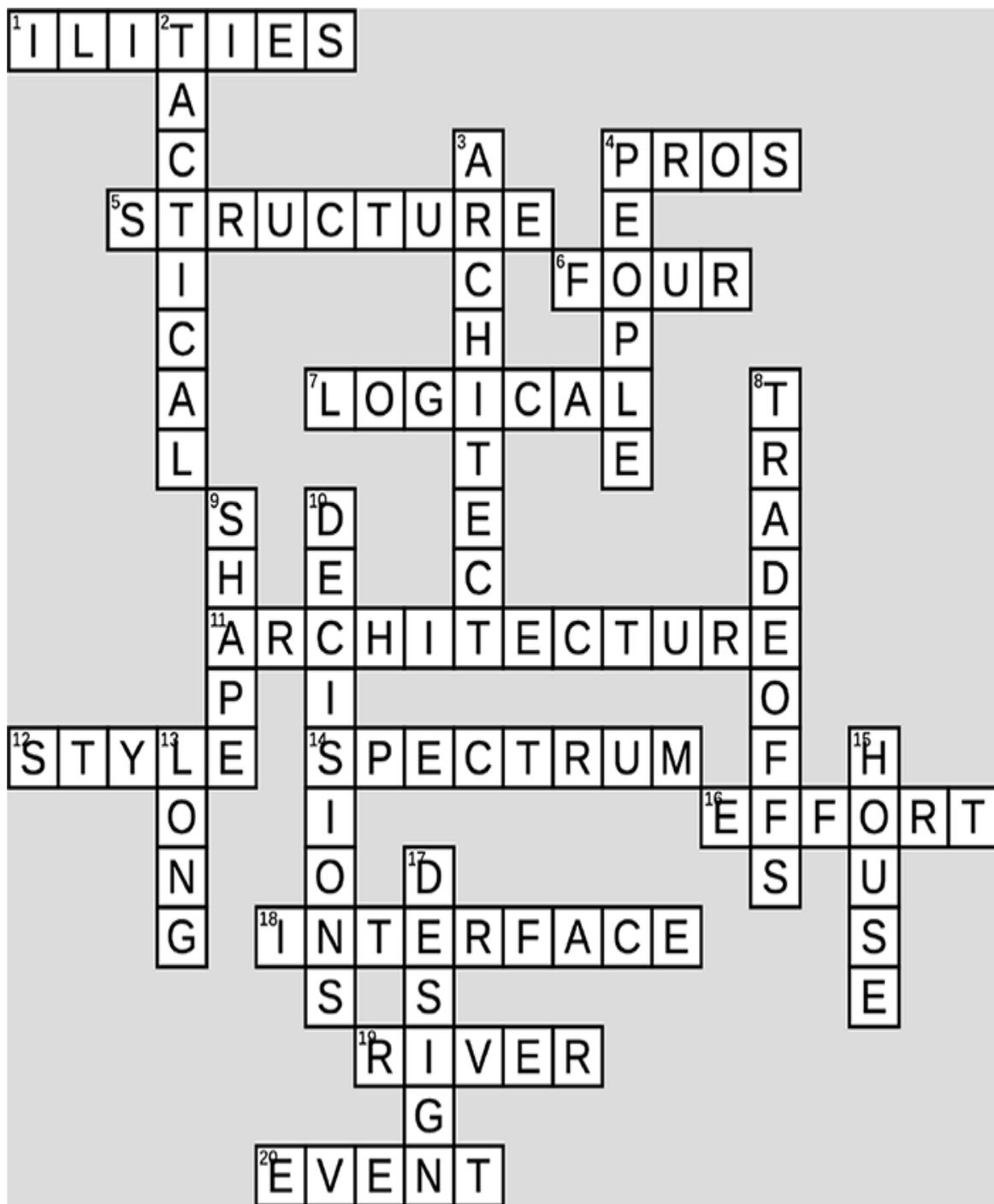


Figure 1-90.

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

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

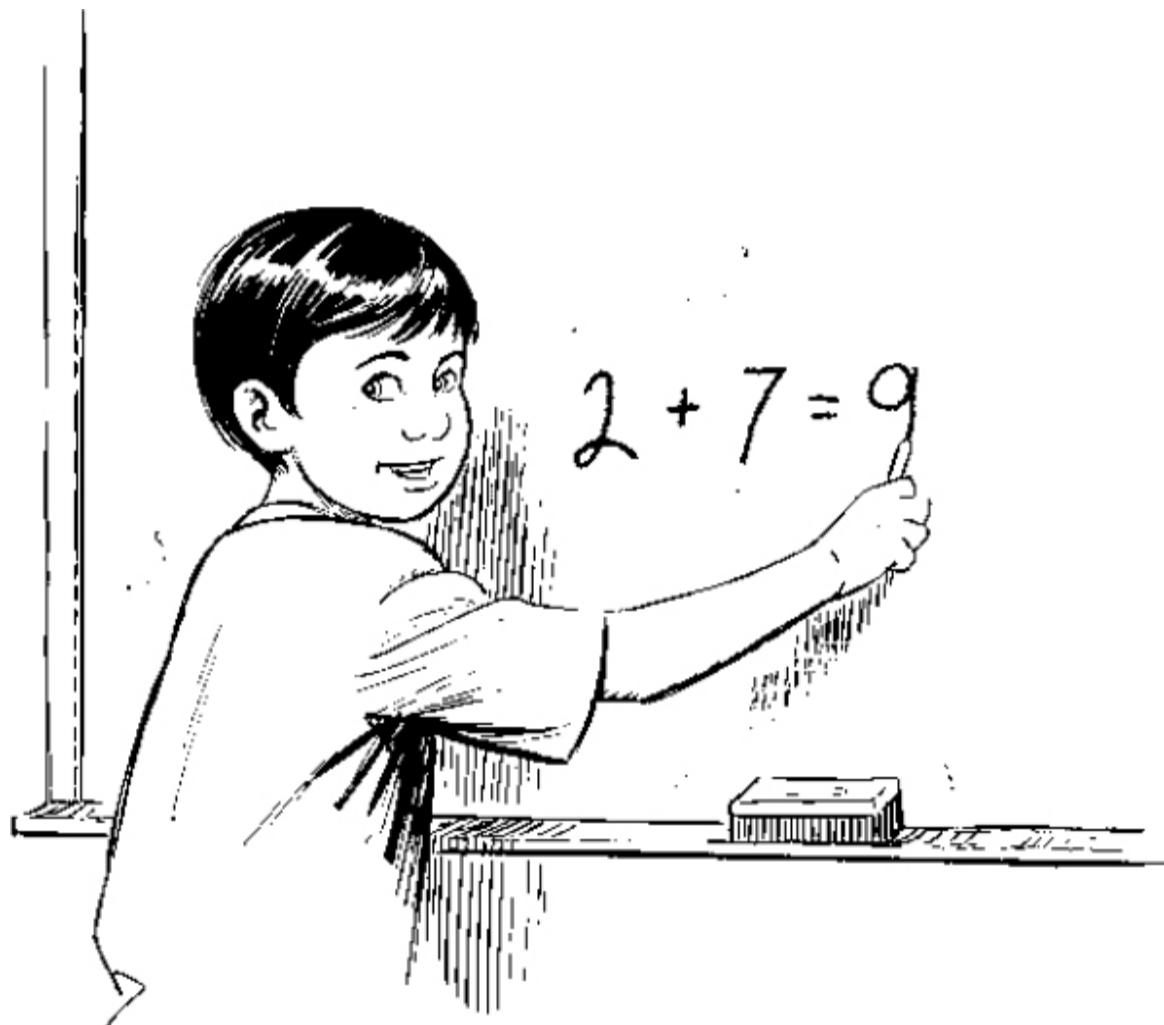
Chapter 2. Architectural Characteristics: *Putting the “Function” into Nonfunctional*

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 2nd 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)



Why care about something called non-functional? An architect designs a software system to meet requirements. However, other factors many influence the design of the software system. However, some requirements focus on the capabilities of the system rather than its behavior. For example, if it must support a large number of users at once and it fails, then the system isn't a success.

The other criteria an architect must consider when crafting a software system are called *architectural characteristics* (or *non-functional requirements*), which are fundamental building blocks for understanding and implementing architecture. They are the pathway to supporting large numbers of users, attaining a fast time to market, and other important priorities.

Let's consider architectural characteristics: how to define them, how to discover them, and how they influence software architecture.

Laffter, the best medicine?

Sillycon Symposia is a startup with a Bay Area feel whose business plan combines technology-themed conferences with comedy. By gathering like minds, Sillycon provides unique offerings for each group and keeps them engaged by keeping them laughing.

Part of the business plan includes building Laffter, a social media network related to (but not limited to) the conferences Sillycon hosts. The business stakeholders put together a requirements document for it:

NOTE

How hard could it be to start a social networking site?!

A pretty generic
requirements document

Sillycon Symposia is hosting a social media network of like-minded technologists.

Users: Hundreds of speakers, thousands of users

Requirements:

Users can register for user names and approve privacy policy

Users can add new content on Laffter as a "Joke" (long-form post) or "Pun" (short-form post)

Followers can "HaHa" (indicating strong approval) or "Giggle" (a milder approval message) content they like

Speakers at Sillycon Symposia events have a special icon

Speakers can host forums on the platform related to their content

Users can post messages of up to 281 characters

Users can also post links to external content

One better than
some competing
sites!

Additional Context:

International support

Very small support staff

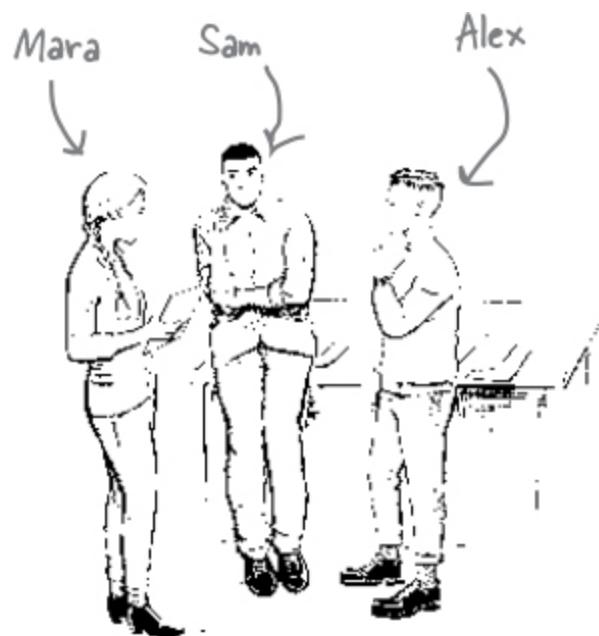
'Bursty' traffic: extremely busy during live conferences

BRAIN POWER



You have no doubt used a social media site, but you may not have thought about it like an architect. For example, these sites must handle large numbers of concurrent users (a characteristic known as *scalability*). What other kinds of capabilities (outside of the basic social-media site behaviors) would the architect of a system like this care about? Don't worry about technical terms; just brainstorm.

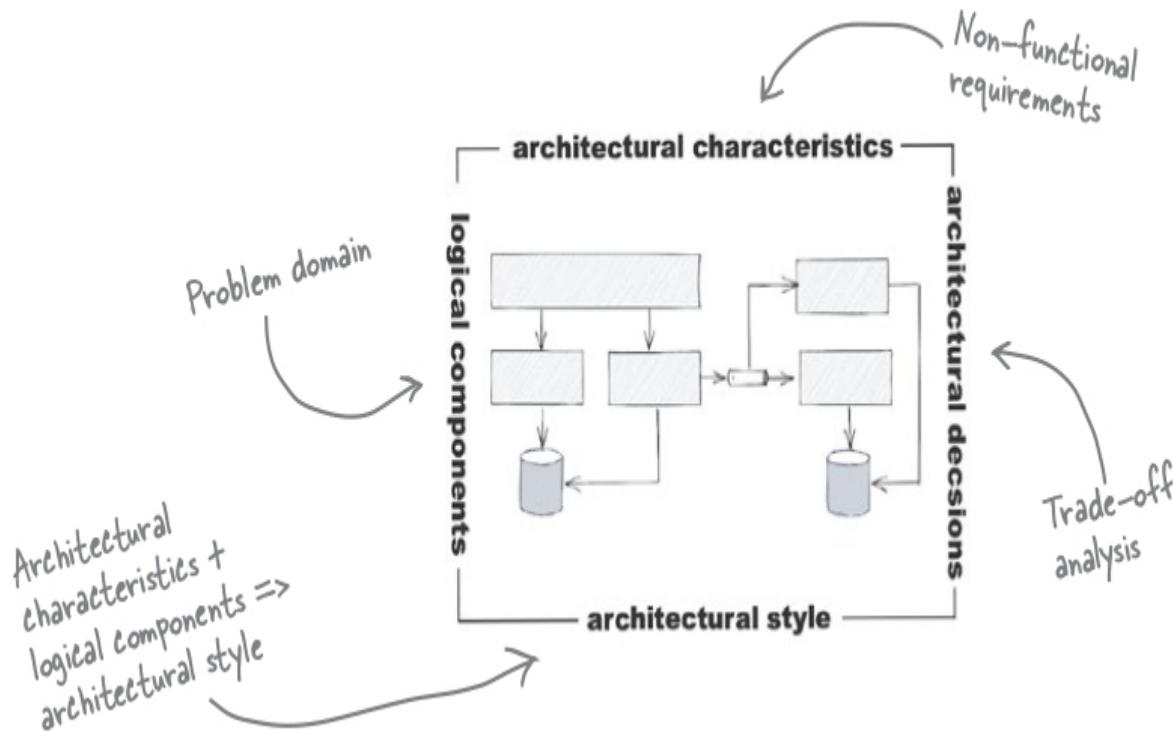
Cubicle Conversation



Mara: Look what just landed in my inbox--the Powers That Be want me to be the architect for the Sillycon Symposia social media app, Laffter.

Alex: You have the requirements? You should jump right into the design of the system—it seems really straightforward.

Sam: Well, you can really only do that for the simplest of applications, and I don't think this one qualifies. Remember the diagram I drew on the whiteboard the other day?



You need to analyze both architectural characteristics and logical components before you can choose an architecture style as a starting point.

You can implement just about any application in any architecture style, but some make it much easier than others. Choosing the style before performing this type of analysis is a classic case of putting the cart before the horse.

Alex: Can't we just be super-agile, start with something tiny, and then keep iterating on it until we have the entire system?

Sam: The iterative approach you talk about doesn't quite work like that for architectural characteristics analysis. For example, it's difficult to make a

system highly scalable if it wasn't designed for that.

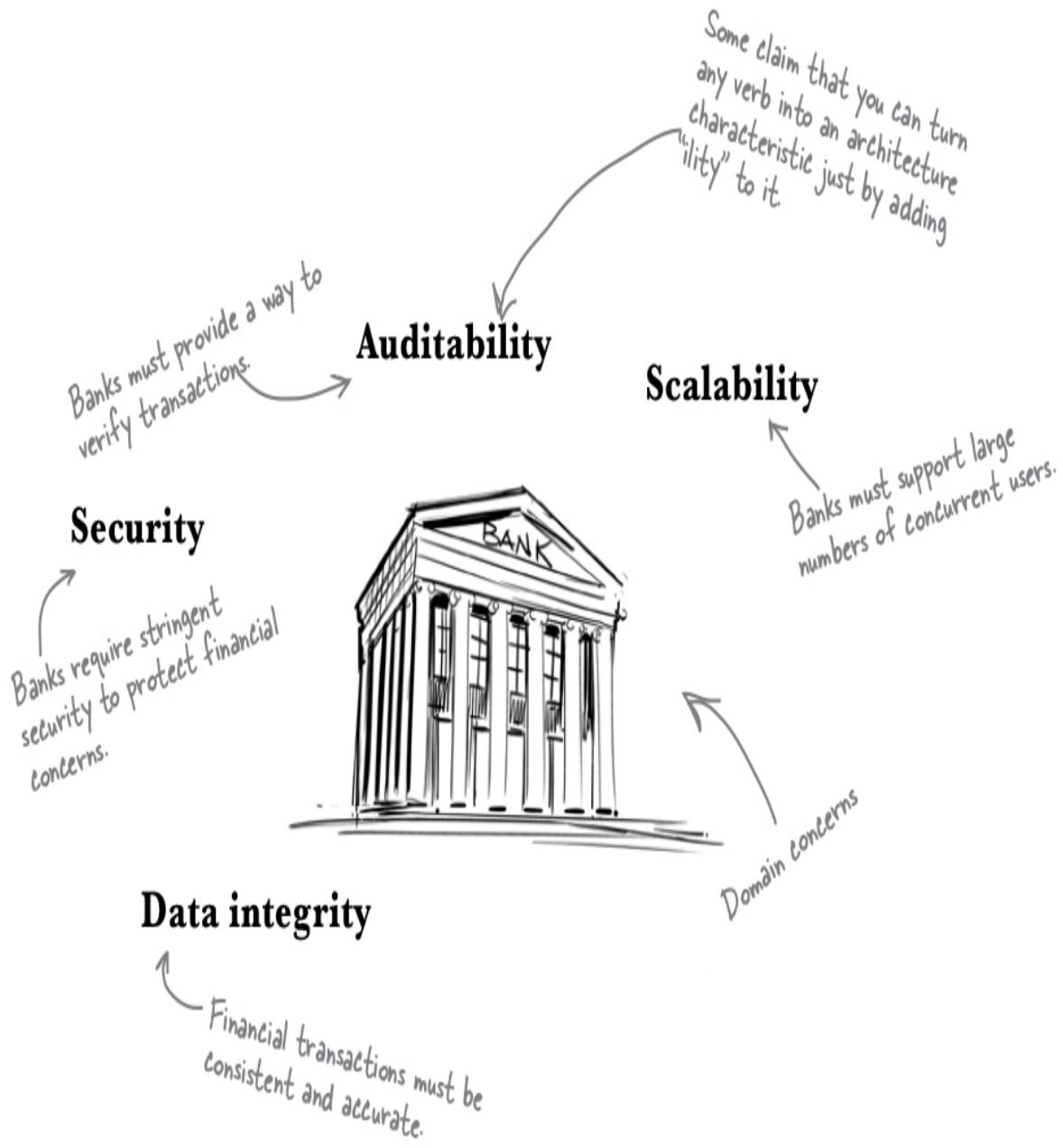
Mara: That makes sense. I guess I need to roll up my sleeves and analyze some architectural characteristics—thanks!

What are architectural characteristics?

You have a problem and decide “I’m going to write some software to solve this problem!” The *thing* you’re writing software about is called the *domain*, and designing for it will occupy much of your effort—that is after all why you’re writing software. However, it’s not the only thing an architect must consider--they must also analyze *architectural characteristics*.

NOTE

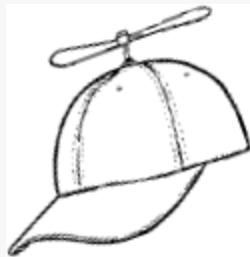
Now you have 2! problems?



NOTE

The many names of “architectural characteristics”

SERIOUS CODING



The software architecture world lacks a standard term for what we call *architectural characteristics*. Here are some of the terms people often use, and why we don't care for them.

Most teams still call them *non-functional requirements*, which is misleading because architectural characteristics are indeed functional—they just don't concern the domain. Calling them *non-functional* downplays their importance. Other teams call them *system quality attributes*, which implies an activity that happens at the end of the project rather than the beginning. Another common name is *cross-cutting requirements*, which is the one we dislike the least—but it contains the word *requirement*, which entangles it with domain behaviors, which come from requirements, as opposed to capabilities, which are defined by architectural characteristics.

Defining architectural characteristics

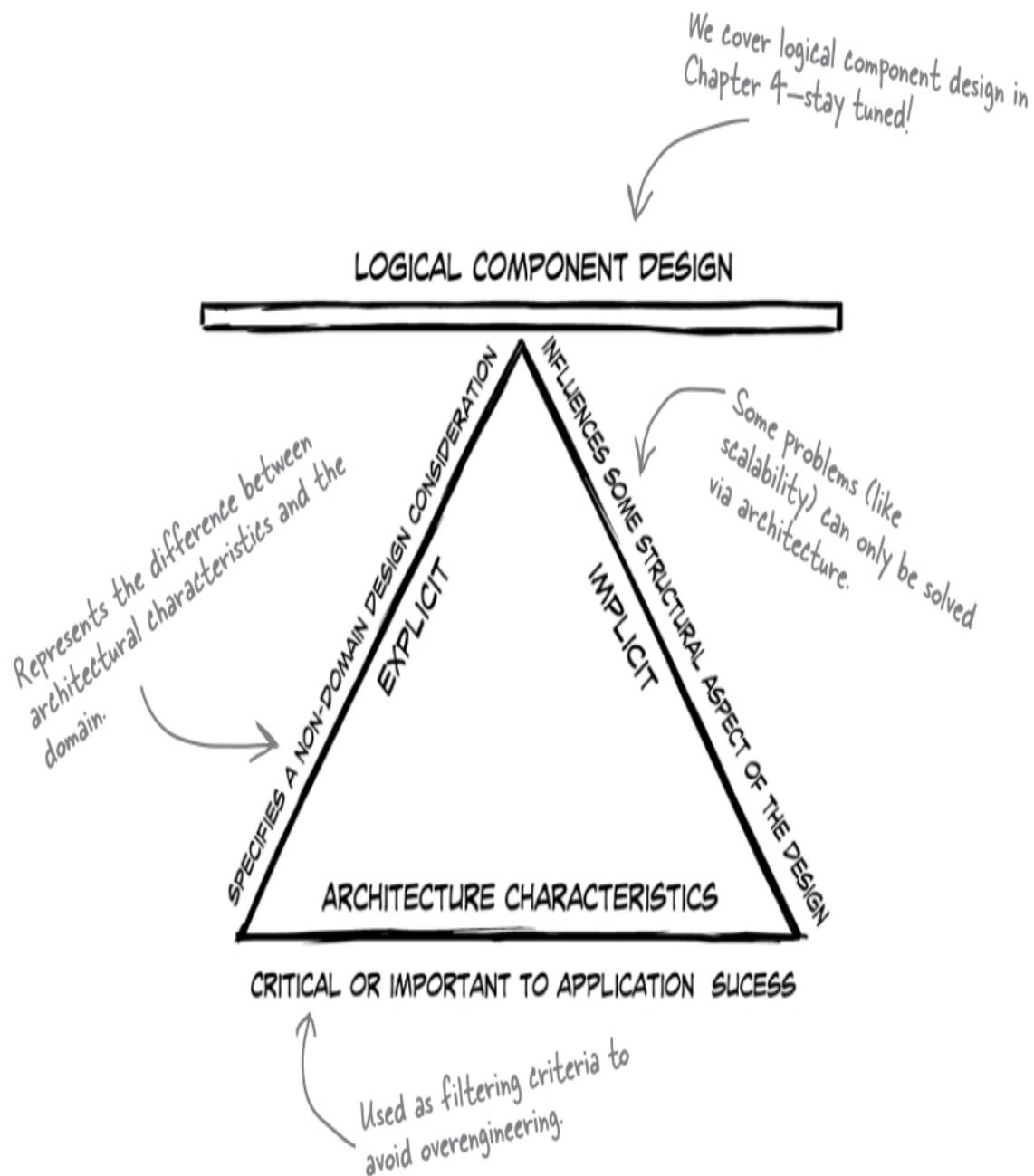
Architects break down structural design for software systems into two types: *logical components* and *architectural characteristics*. Logical components represent the domain of the application—the motivation for writing the software system (we cover these in [Chapter 4](#)). If you combine architectural characteristics with logical components, you have the structural considerations for an architecture.

Architectural characteristics are the important parts of the construction process of a software system or application, independent of the problem

domain. They represent its operational capabilities, internal structure decisions, and other necessary characteristics.

We'll show you lots of examples of architectural characteristics in the upcoming pages, but first we want to cover the concept itself.

We define architectural characteristics in three parts, as shown here.

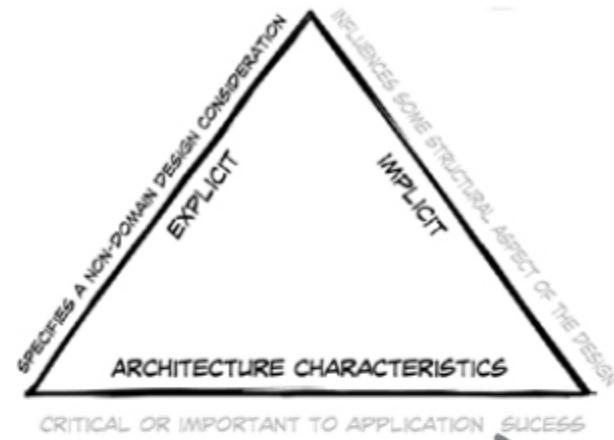


Part 1: A non-domain design consideration

To define architectural characteristics, we first need to look at what they are *not*. The *requirements* specify what the application should do; *architectural characteristics* specify operational and design criteria for success, how to implement the requirements, and why certain choices were made. For example, an application’s level of performance is an important architecture characteristic that often doesn’t appear in requirements documents. Even more pertinent: no requirements document will tell you to “prevent technical debt,” but doing so is still a common design consideration for architects and developers.

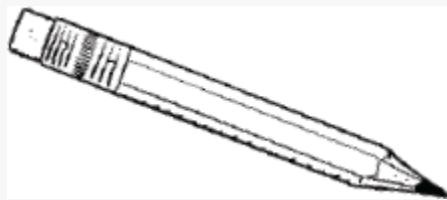
NOTE

“Technical debt” refers to less-than-optimal decisions that accumulate over time and degrade the ability to maintain and/or extend a system.



Architectural characteristics represent important design decisions that don't directly pertain to the domain, yet are required for success.

SHARPEN YOUR PENCIL



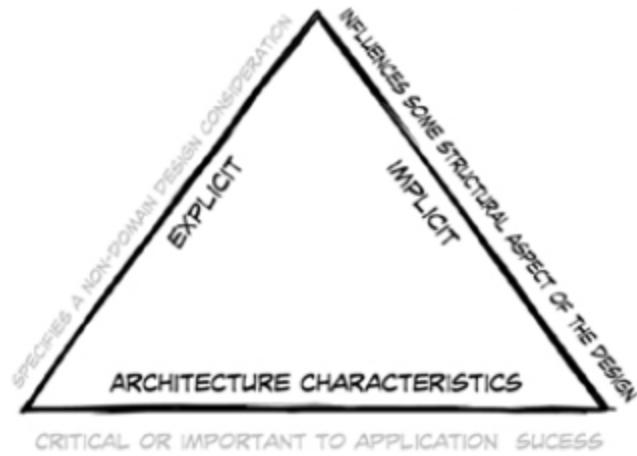
See if you can identify the domain bits within the requirements for Sillycon Symposia's Laffter social media app. For example, Users and Friends are part of the domain. What else belongs to the domain in this example?

1. **Joke/Pun**

2.

Is there anything in the project overview that doesn't fit within the domain? For example, the application needs to be scalable (to support lots of concurrent users) during conferences. What other features are listed in the project summary but are not part of the system's problem domain?

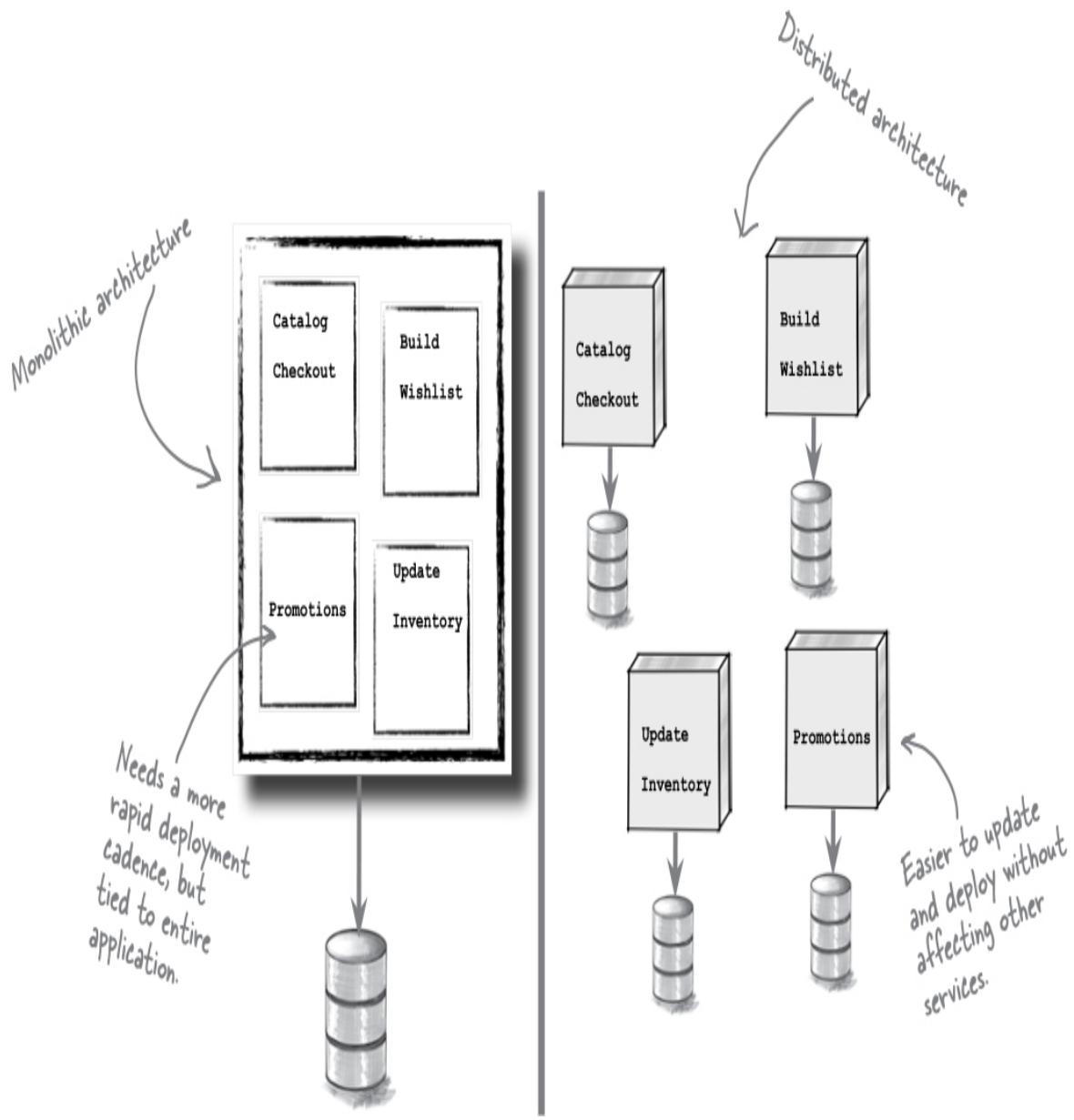
Part 2: An influence on the structure



The primary reason architects try to describe architectural characteristics has to do with design considerations: does this architecture characteristic require special structural support to succeed? For example, security is a concern in virtually every project, and all systems must take baseline precautions during design and coding. However, security becomes an architectural characteristic when the architect needs to design something special to accommodate it.

Consider the architecture diagrams below for a simple order-processing application that includes marketing for upcoming promotions and rules for when each promotion applies. An architect could design this as a monolithic architecture—one with a single deployable unit and matching database—or as a series of independent services.

For the monolithic architecture, the entire application would have to be redeployed when the promotion rules change, because monoliths feature a unified deployment. However, in a distributed architecture, only the promotions service would be affected and could be redeployed independently.

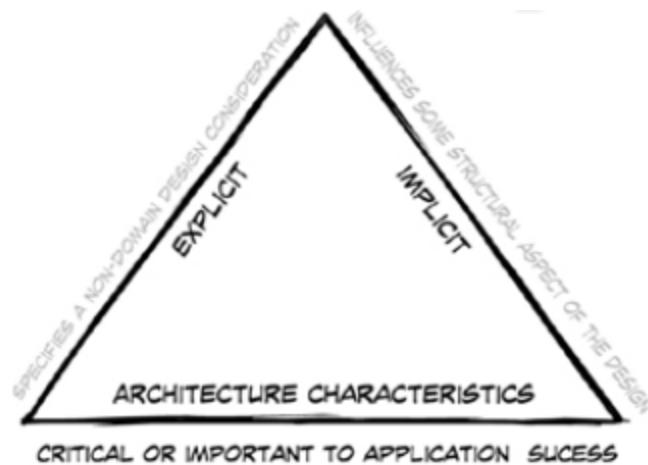


Even these two criteria aren't always sufficient to make this determination.

Part 3: Important

Applications *could* support a huge number of architectural characteristics... but they shouldn't. Every architectural characteristic the system must support adds complexity to its design. Thus, architects try to choose as *few* architectural characteristics as possible, rather than the most.

The sheer number and variety of architectural characteristics means there are so many tempting choices! But architects should limit our choices because architectural characteristics are:



- ① **Impossible to standardize**

Different organizations use different terms for the same architectural characteristics. For example, performance and responsiveness might indicate the same behavior.

NOTE

It's a good idea to create a "ubiquitous language" (shared vocabulary) for architectural characteristics within your organization—this gives you a fighting chance at creating a usable standard list.

- ② **Synergistic**

Architectural characteristics affect other architectural characteristics and domain concerns. For example, if you want to make an application more secure, the required changes will almost certainly affect performance negatively. More on-the-fly encryption and other similar changes will lead to performance overhead.

NOTE

This entanglement implies that you often cannot choose one architectural characteristic without considering how it may affect others.

- **③ Overabundant**

Possible architectural characteristics are extraordinarily abundant, and new ones appear all the time. For example, a few years ago there was no such thing as on-demand elasticity via a cloud provider.

NOTE

Even the number of categories of architectural characteristics has increased over the last few years, with additions such as cloud constraints and capabilities

WATCH IT!



Synergy can be dangerous!

Architects would love to design for architectural characteristics independently from the domain design. Unfortunately, the real world refuses to cooperate. When we say that architectural characteristics are *synergistic*, we mean that changes to one might require changes to other architectural characteristics and/or the domain. In other words, no matter how clever you are, no architect can make every single architecture scalable. Some architectures cannot scale as successfully as others because of physical constraints such as memory and bandwidth.

Be careful if you change one architecture characteristic; consider how that change may affect other parts of your architecture. The same applies to making changes to domain design, such as component boundaries and distribution—changes to the domain may synergistically affect your architectural characteristics. For example, if you change your application to begin storing users' payment information, the security and data integrity architectural characteristics will also change.

BRAIN POWER



Many things in the real world are synergistic—that is, combining them yields something different than the sum of the parts. See if you can think of some real-world examples of synergy. Hint: These might include things that are still identifiable (like peanut butter and chocolate) or things that merge (such as emulsions like oil and vinegar).

Overengineering is Too Easy

NOTE

While it's fun to play with shiny new things, this often leads to overengineering, or trying to add too many toys to the solution.

A common hazard for architects is *overengineering*: supporting too many architectural characteristics and complicating the overall design to little or no benefit. The third criterion for defining architectural characteristics, ***critical or important to success***, acts as a filtering tool. It helps us eliminate features that would be nice to have but add needless complexity to the design.

It's important to make sure that architectural characteristics add value to the project. It's fun to play with new stuff, but doing so doesn't always align with larger priorities. Beware of RDD (resume-driven development)!

Architects must focus on making choices that add value.

Explicit versus implicit

Some things are *explicit*—stated clearly—whereas others are *implicit*—assumed based on context or other knowledge.

Explicit

Packages are stacked outside a door.

The door is locked.

Implicit

No one is home.

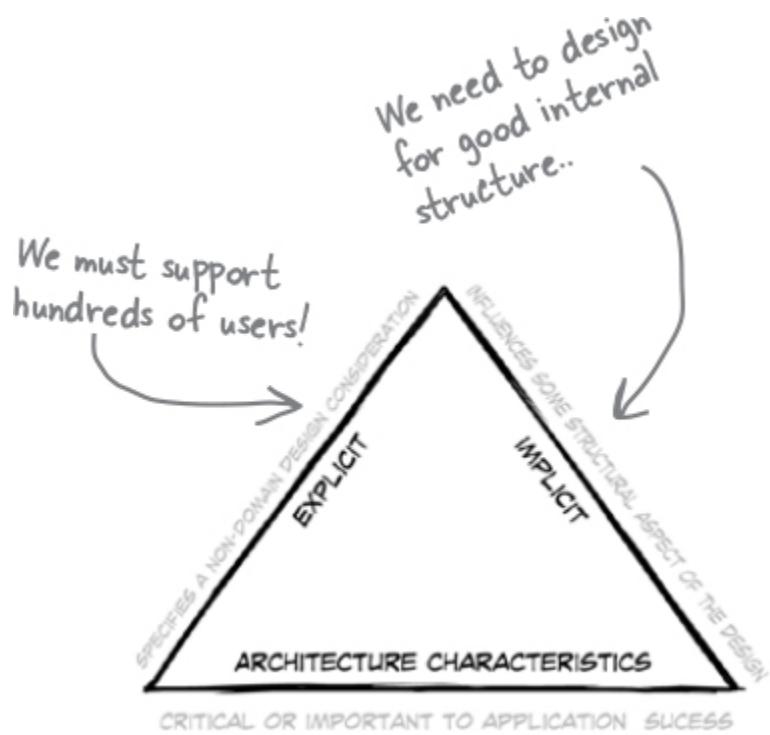
This family orders a bunch of stuff online.

Is this family on vacation?



Explicit architectural characteristics are specified in the requirements for the application.

Implicit architectural characteristics are factors that influence an architect's decision but aren't explicitly called out in the requirements. Security is often an implicit architecture characteristic: even if it isn't mentioned in the requirements, architects know that we shouldn't design an insecure system.



Architects must use our knowledge of the problem domain to uncover these architectural characteristics during the analysis phase. For example, a high-frequency trading firm may not specify how critical it is for transactions to complete within milliseconds, but the architects in that problem domain know how critical it is.

WATCH IT!



Subtle but Important

Some implicit architectural characteristics are more subtle, but just as important. For example, architects should pay attention to the application's internal structure as developers create it, to ensure that sloppy coding and other deficiencies don't degrade the longevity of the application. However, virtually no requirements list will specify "Don't mess up the internal modularity of the system as you build it!"

The International Zoo of “—ilities”

“Ladies and gentlemen, boys and girls, children of all ages— welcome to the International Zoo of —ilities!”

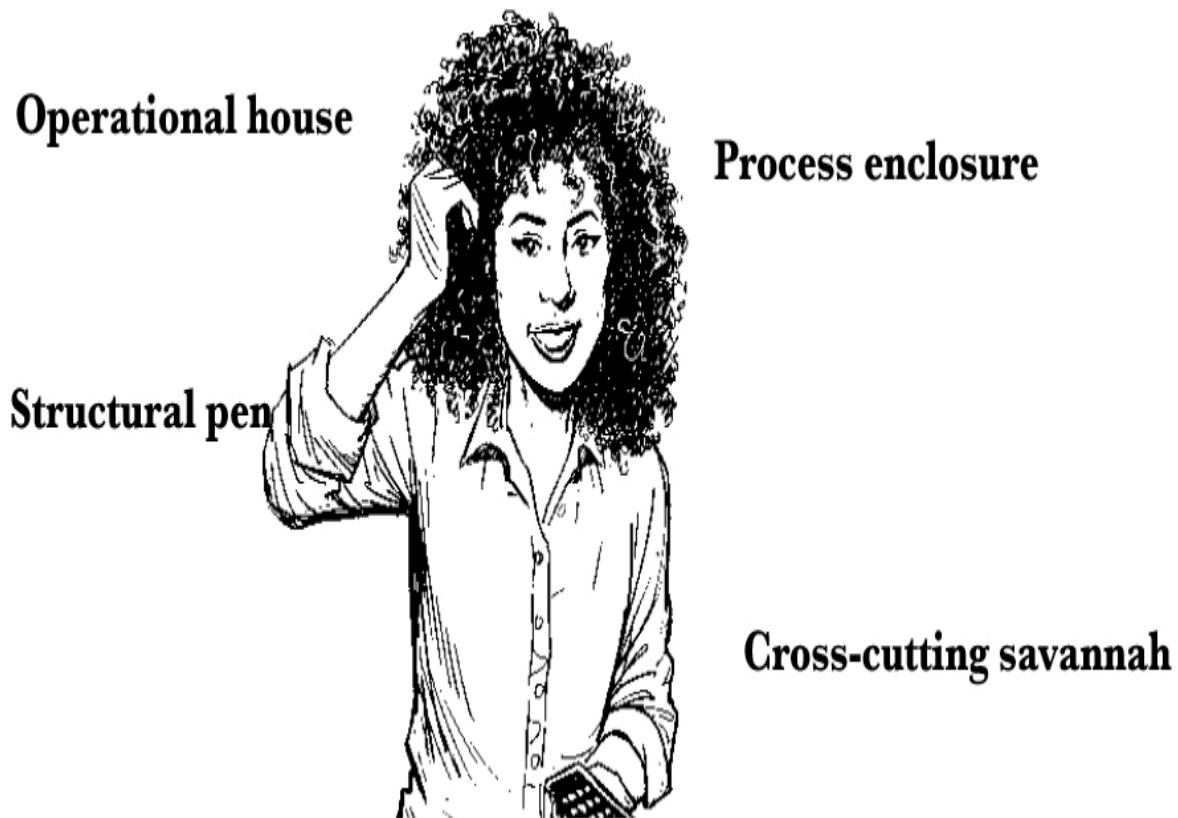
Like the animals in a zoo, architectural characteristics exist along a broad spectrum. They range from low-level code characteristics, such as modularity, to sophisticated operational concerns, such as scalability and elasticity. There is no true universal standard (although people have tried). Instead, each organization interprets these terms for itself. Additionally, the software ecosystem is constantly adding new concepts, terms, measures, and verifications, providing new opportunities to define architectural characteristics.

Not only are there lots of architectural characteristics, but they fall into numerous categories—much like a zoo has different sections for different families of animals. Come visit our architectural characteristics zoo.

Don't worry—we define many
of the more mysterious terms
on the next few pages.

scalability	accessibility	observability
availability	maintainability	testability
interoperability	reusability	portability
security	simplicity	feasability
stability	reliability	usability
agility	integrity	performance
traceability	localizability	auditability

Think this is a big list? Check
out [https://iso25000.com/
index.php/en/iso-25000-
standards/iso-25010](https://iso25000.com/index.php/en/iso-25000-standards/iso-25010)



BRAIN POWER



Why do so many architectural characteristics feature the “-ility” suffix?

Process architectural characteristics

Process architectural characteristics are where the software development process intersects with software architecture. They reflect the decisions

about the mechanics of building software.



modularity

The degree to which the software is composed of discrete components. Modularity affects how architects partition behavior and organize logical building blocks.

agility

A composite architectural characteristic that encompasses testability, deployability, modularity, and a host of other architectural characteristics that facilitate and enable agile software development practices.

Agility is a composite architectural characteristic we'll discuss later in this chapter—stay tuned!

Yes, we know this is a made-up word. That happens a lot in software architecture!

decouple-ability

Coupling describes how parts of the system are joined together; some architectures define how to *decouple* parts in specific ways to achieve certain benefits, which this architecture characteristic measures.

testability

How complete the system's testing is and how easy these tests are to run, including unit, functional, user-acceptance, exploratory, and other forms of testing.

"Testability" refers to testing at development time (such as unit testing), rather than formal quality assurance..

Yep, it shows up twice. Many architectural characteristics cut across categories, as you'll see in the next few pages.

extensibility

How easy it is for developers to extend the system. This may encompass architectural structure, engineering practices, internal design, and governance.

This is one of the many architectural characteristics that make up "agility".

deployability

How easy and efficient it is to deploy the software system.

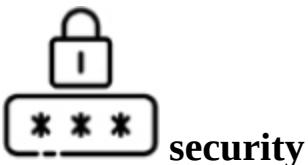
Structural architectural characteristics

Structural architectural characteristics affect the internal structure of the software system, including factors like the degree of coupling between components and the relationships between different integration points.



NOTE

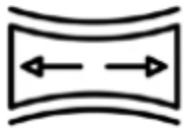
Security appears in every application, as an implicit or explicit architectural characteristic.



How secure the system is, holistically. Does the data need to be encrypted in the database? How about for network communication between internal systems? What type of authentication needs to be in place for remote user access?

NOTE

Yep, it shows up twice. Many architectural characteristics cut across categories, as you'll see in the next few pages..



extensibility

How easy it is for developers to extend the system. This may encompass architectural structure, engineering practices, internal design, and governance.

NOTE

Some architectural characteristics cover development concerns rather than purely domain concerns.



maintainability

How easy it is for architects and developers to apply changes to enhance the system and/or fix bugs.

Portability can apply to any part of the system, including the user interface and implementation platform.



portability

How easy it is to run the system on more than one platform (for example, Windows and OS-X).

NOTE

Another flavor of localization is internationalization (i18n).



localization

Describes how well the system supports multiple languages, units of measurement, currencies, and other factors that allow the system to be used globally.

Operational architectural characteristics

Operational architectural characteristics represent how architecture decisions influence what operational team members can do.

NOTE

This intersection is at least partially why DevOps exists.





availability

How long the system will need to be available and, if 24/7, how easy it is to get the system up and running quickly after a failure.

NOTE

Usually represented as a number of “nines” (99.999% uptime == 5 nines)



robustness

The system’s ability to handle errors and boundary conditions while running, such as if there’s a power, internet connection, or hardware failure.

NOTE

When these are important, they are VERY important.



reliability/safety

Whether the system needs to be fail-safe, or if it is mission-critical in a way that affects lives. If it fails, will it endanger people’s lives or cost the company large sums of money? Common for medical systems, hospital software, and airplane applications.



A good example of the
axiom that you can take any
adjective and add “-ility”
to make a new architecture
characteristic!

recoverability

How quickly the system can get online again and maintain business continuity in case of a disaster. This will affect the backup strategy and requirements for duplicated hardware.



performance

How well the system achieves its timing requirements using the available resources.

NOTE

As you will see shortly, “performance” has many different aspects.

Some -ilities are easier to achieve than others. This one is often difficult.



scalability

How well the system performs and operates as the number of users or requests increases.

NOTE

Our Laffter application will definitely need this.

Cross-cutting architectural characteristics

As much as we'd like a nice, orderly zoo of architectural characteristics, platypuses still show up! Lots of important characteristics defy categorization.





authentication/authorization

How well the system ensures users are who they say they are and makes sure they can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).

NOTE

Authentication and authorization are aspects of security.

NOTE

Another always-present crosscutting concern



security

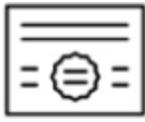
How secure the system is, holistically. Does the data need to be encrypted in the database? How about for network communication between internal systems? What type of authentication needs to be in place for remote user access?

Many government agencies around the world require a baseline level of accessibility.



accessibility

How easy is it for all your users to access the system, including those with disabilities, like colorblindness or hearing loss.



legal

How well the system complies with local laws about data protection and about how the application should be built or deployed.

NOTE

Many countries and regions have strict laws governing privacy, making consistency for international applications tricky



privacy

How well the system hides and encrypts transactions so that internal employees like data operators, architects, and developers cannot see them.



usability

How easy is it for users to achieve their goals. Is training required? Usability requirements need to be treated as seriously as any other architectural issue.

NOTE

A great example of how ambiguous architectural characteristics can be: “usability” can also refer to user experience design.

BRAIN POWER



Security is a concern in every software system. It is sometimes explicitly listed as a requirement, for example in systems that handle payments or other sensitive information. Even when it's not called out, though, it's always implicit, because architects should always avoid building insecure applications. What other architectural characteristics should appear in every application?

THERE ARE NO DUMB QUESTIONS

Q: Where can I find a standard list of architectural characteristics?

A: No standard list really exists (despite several futile efforts) because the software development ecosystem constantly shifts and changes. Anyone trying to create a standard list is trying to hit a moving target.

Q: Isn't security required for every application?

A: It depends! While it's a common concern, if you design a free intra-office lunch-ordering system, the only security concern lies with others finding out that you order an egg salad sandwich every day.

Q: Isn't availability required for every application?

A: It depends! Again, it's a common concern for most applications, but if the mythical sandwich-ordering system mentioned above fails, the only real downside is that everyone has to get their own lunch.

Q: Can I choose any combination of architectural characteristics for my application?

A: Some architectural characteristics oppose one another. For example, architects find it challenging to design for both high performance and scalability. Determining the architectural characteristics for a system is only part of the design process. Combining them with logical component design will point you to an appropriate architecture style.

Q: What does it mean if you don't choose an architecture characteristic like availability in your requirements?

A: The architectural characteristics you choose provide a guideline for the appropriate architecture style. If an architect doesn't choose availability, it doesn't mean she will purposefully design the system to have poor availability. Rather, it's an indication of priority: trading off one architecture characteristic for another.

EXERCISE



Welcome to “Take It or Leave It!” The rules of this game are simple—we’ll give you a business requirement that might come up for the Laffter application, and two architectural characteristics. As you know, everything in architecture is a trade-off. So if you attempt to optimize one, you are probably not going to do as well with the other. Your job is to tell us which characteristic should rate as a higher priority for that requirement.

“We need to get this to market
ASAP!”

fault tolerance agility

“Money’s tight, folks!”

scalability simplicity

“Oh, wow, this conference is
going to be our biggest yet.”

high availability maintainability

“We want to start storing user
credit card information.”

security recoverability

“This site is going to be very
popular upon launch.”

agility elasticity



o o

I see that there are lots of architectural
characteristics . . . but how do I know
which ones are critical or important to my
project?

Architectural characteristics don't just appear out of thin air! They derive from the problems we're trying to solve by writing software, of course, but that's not the only place we must keep an eye out for them.

Part of software architects' skillset is analyzing a problem to determine what architectural characteristics the system requires. For example, the requirements may specify "lots of users"; as an architect, you must dig deeper to determine how many users are expected (scalability), how many of them will be there at the same time (concurrency), and how rapidly they'll show up (elasticity). Many structural design decisions come directly from the problem definition.

However, complications abound! Architectural characteristics also come from a couple of other sources: environmental awareness and holistic domain knowledge.

Let's look at each of these sources in turn.

The problem domain

Architects derive many of the necessary architectural characteristics from the problem domain—it is, after all, the motivation for writing the software in the first place. That means we must translate items stated in requirements documents into their corresponding architectural characteristics.

EXERCISE



Domain requirements are often a rich source of architectural characteristics. For example, our Laffter application needs to support hundreds of users, so scalability will be one necessary architecture characteristic. Can you uncover more?

Sillycon Symposia is hosting a social media network of like-minded technologists.

Users: hundreds of speakers, thousands of users

Requirements:

- Users can register for user names and approve privacy policy
- Users can add new content on Laffter as a “Joke” (long-form post) or “Pun” (short-form post)
- Followers can “HaHa” (indicating strong approval) or “Giggle” (a milder approval message) content they like
- Speakers at Sillycon Symposia events have a special icon
- Speakers can host forums related to their content
- Users can post messages of up to 281 characters
- Users can also post links to external content

Architectural characteristics

Scalability

Additional Context:

- International support
- Very small support staff
- ‘Bursty’ traffic: extremely busy during live conferences

The problem domain is a rich source of architectural characteristics. In fact, we have a whole chapter coming up that walks through a couple of

extended examples. However, first, let's cover the other two common sources for architectural characteristics.

Environmental awareness

Many implicit architectural characteristics come from the architect's basic understanding of the overall organization. For example, an architect working for a fast-moving startup will prioritize agility whether it is specified or not.

NOTE

Sorry, but this means you're going to have to start paying attention in those business prioritization meetings!

It is important to understand organizational priorities so we can make more durable decisions. For example, let's say we must make a decision about how to integrate two subsystems. The choices are a customized but highly suited protocol and an industry-standard protocol that will require a little more effort. In a vacuum, we might choose the first. However, if we know that the organization's goal is to engage heavily in mergers with other companies, that fact could tip the decision toward the more open solution.

NOTE

Architects can't make decisions in a vacuum—context is always important.

NOTE

Some architects stay within particular domains exactly because they have the advantage of domain knowledge.

Holistic domain knowledge

Some architectural characteristics come from our own inherent *domain knowledge*: information that isn't explicitly spelled out in the requirements but that we implicitly understand about important aspects of the domain.

For example, suppose we design an application that handles class registration for university students. To make the math easy, assume that the school has 1,000 students and 10 hours for registration. Should we design the system using a consistent scale, implicitly assuming that the students will distribute themselves evenly during the registration process?

Based on real-world experience, we can say that this won't work. Think about what you know about the target demographic. Some students are hyper-diligent and some tend to procrastinate. Thus, the actual design must handle an elastic burst of students in the first hour (as everyone signs up early for hard-to-get classes), stay mostly idle for the bulk of the day, and then handle another elastic burst just before the registration window closes to accommodate all the stragglers.

NOTE

One of the most dangerous discoveries in life is how much you can procrastinate and still (mostly) get the job done.

NOTE

Never underestimate some university students' capacity to procrastinate..

Architects must apply all information sources available to understand the full range of trade-offs inherent in our architecture decisions.

Solution versus problem

Users often come to architects with solutions rather than requirements. For example, back in the 1970s, the US Air Force commissioned a fighter jet and included a requirement that it be capable of achieving speeds up to Mach 2.5. The designers tried, but the technology of the time just wasn't sufficient to meet the requirement. They came back to the Air Force and asked: "Why does it need to go Mach 2.5?" The answer was, "Well, these things are expensive, so we want it to be able to flee a fight if necessary." With that knowledge in mind, they went back and designed the F-16 fighter jet. It had a maximum speed of Mach 2.1, but it was the most maneuverable jet and fastest-accelerating jet ever created.

When users bring us solutions rather than requirements, it's the architect's job to imitate an annoying toddler and keep asking "But why?!?" enough times to uncover the actual requirements hidden within that solution.

WHO DOES WHAT?

It's sometimes difficult to untangle requirements from solutions. Here are some phrases you might have heard in your own company that indicate something might be one or the other. Can you identify the requirements and the solutions?

“We need a system to track user preferences and customizations, and save them between sessions.”

requirement

solution

“Do we really need to build our own survey service? Surely we can find one that does what we need.”

requirement

solution

“An enterprise service bus solves some of our current problems (albeit with some changes and work-arounds) and offers extreme extensibility.”

requirement

solution

“According to the friendly sales rep, this software package does all the things Accounting needs now and in the near future.”

requirement

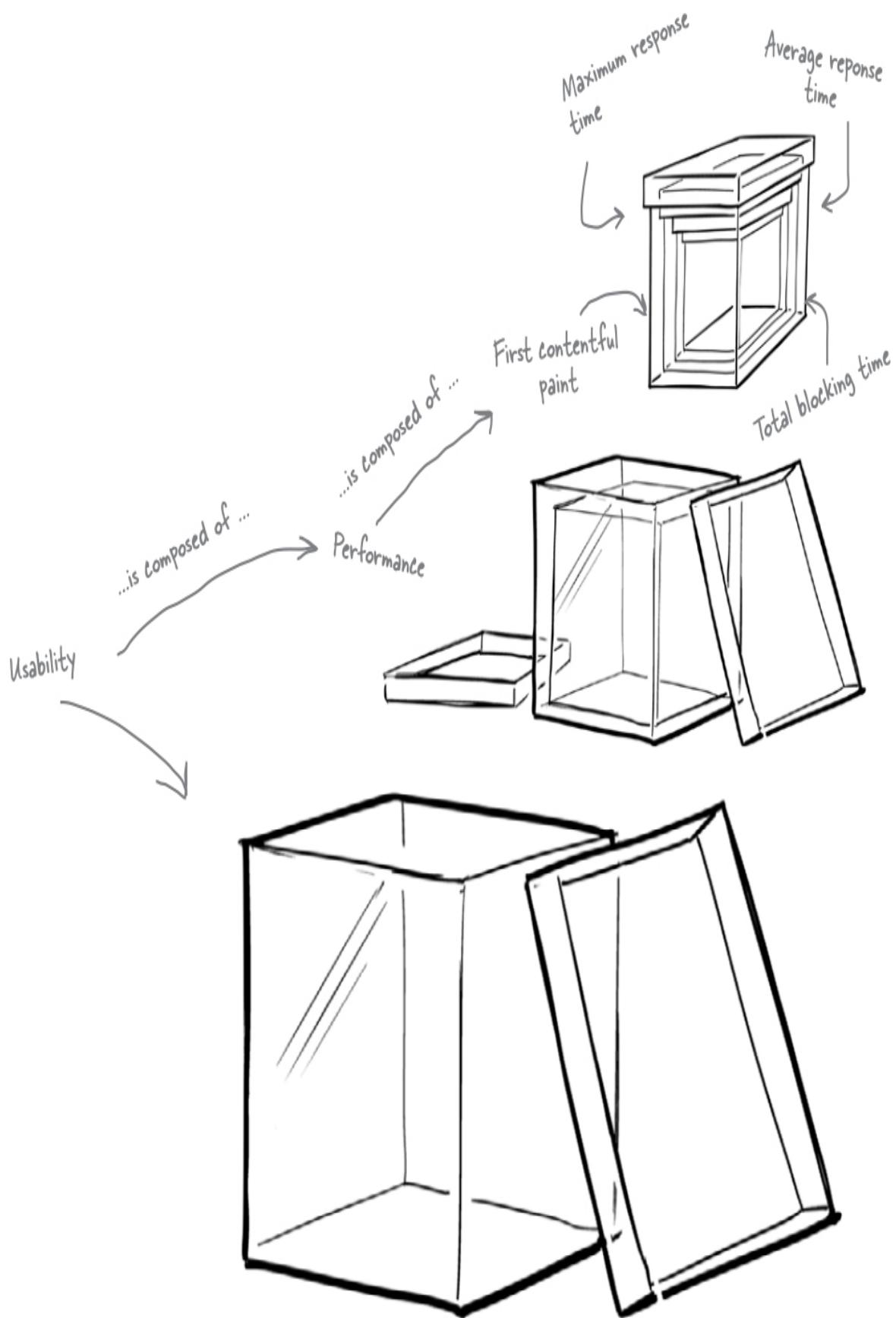
solution

Composite architectural characteristics

A composite is a combination of two or more things, and often architectural characteristics combine with each other to create (seemingly) new ones. We call these *composite architectural characteristics*.

For example, an architect may be asked to create a “reliable” architecture. What does that mean? We can measure many different aspects of reliability, like how available the system is, how consistent the user interface workflows are, and how well it handles data integrity.

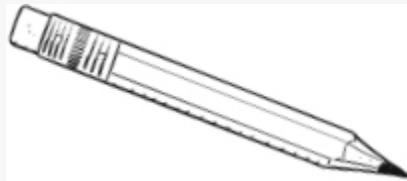
To identify composites, ask: “Can I objectively measure this architecture characteristic?” While we often discuss performance as a single value, it is actually a composite—we have to be more specific to get to something measurable. An example of a measurable architecture characteristic is *first contentful paint*, which measures the time it takes for a webpage to load on a mobile device.



Priorities are Contextual

It's impossible to choose the same set of architectural characteristics for every project. Remember the First Law of Software Architecture: everything is a trade-off. The set of architectural characteristics you choose for a particular application, and how you prioritize each one, will differ based on context.

SHARPEN YOUR PENCIL



Context matters. At the top, we've listed several architectural characteristics. Below that are three application scenarios. For each scenario, rank each architecture characteristic based on how important it is for that type of application. (Hint: Some applications won't need all of them.)

B. Scalability

A. Performance

D. Extensibility

C. Security

E. Data integrity

Scenario #1	Scenario #2	Scenario #3
Building an e-commerce site in a competitive market	A system for an enterprise whose goal is to grow via mergers	An application to automate standardized testings and grading for university admissions
1 _____	1 _____	1 _____
2 _____	2 _____	2 _____
3 _____	3 _____	3 _____
4 _____	4 _____	4 _____
5 _____	5 _____	5 _____

Wait a minute. None of my business analysts or subject-matter experts knows what "scalability" or "elasticity" is. How are they going to know to ask for this stuff?

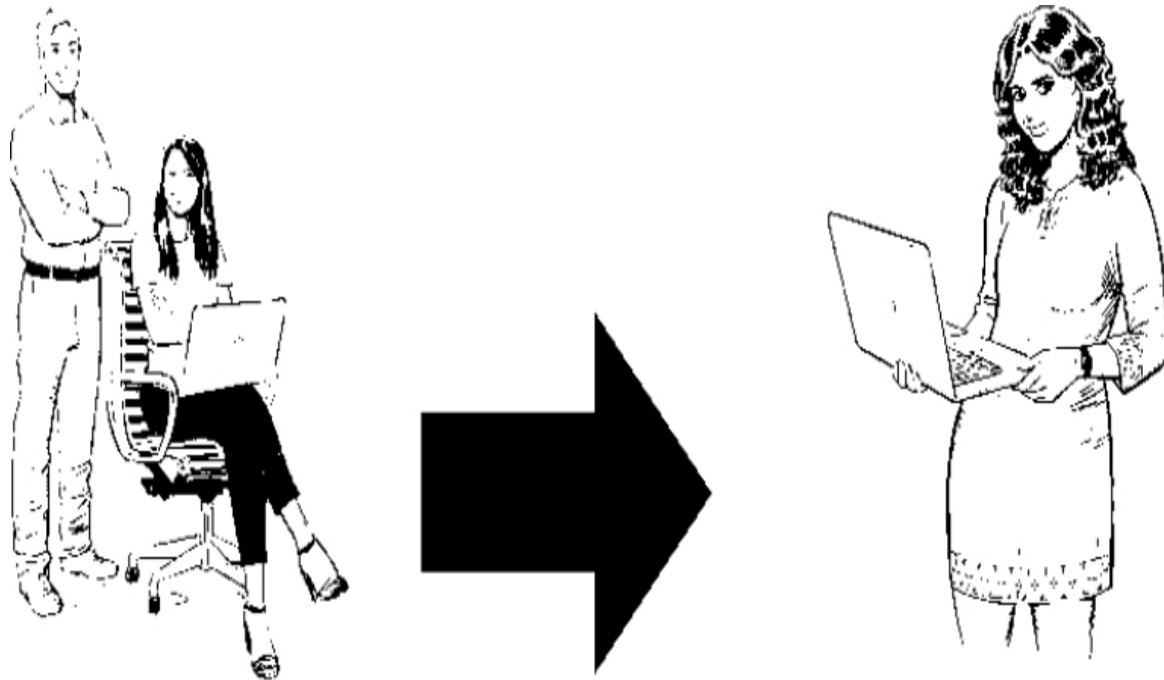


Congratulations, you have yet another job.

You're right to be skeptical about how sophisticated an understanding your co-workers have of architecture concepts. That means you have one more job as an architect...translation!

As much as it would be nice for our colleagues to learn our language, architects are generally the ones who have to translate the business's goals into identifiable and measurable architectural characteristics.

Lost in Translation



When business analysts and subject-matter experts say:

"Our business is constantly changing to meet new marketplace demands."

"Due to new regulatory requirements, it is imperative that we complete daily processing by the end of each day."

"Our plan is to engage heavily in mergers and acquisitions in the next three years."

Of course, no one would *ever* ask "We have a very tight for this impossible timeframe and a fixed scope combination...ahem and budget for this project."

Software architects

translate:

- Agility
- Modularity

- Extensibility

We must perform well but also recover quickly in case of error.

- Performance

- Recoverability

- Scalability

- Resum-ability

- Integratability

Architects often have
a unique perspective
as to what's possible
within a given
timeframe.

- Feasibility

- Simplicity

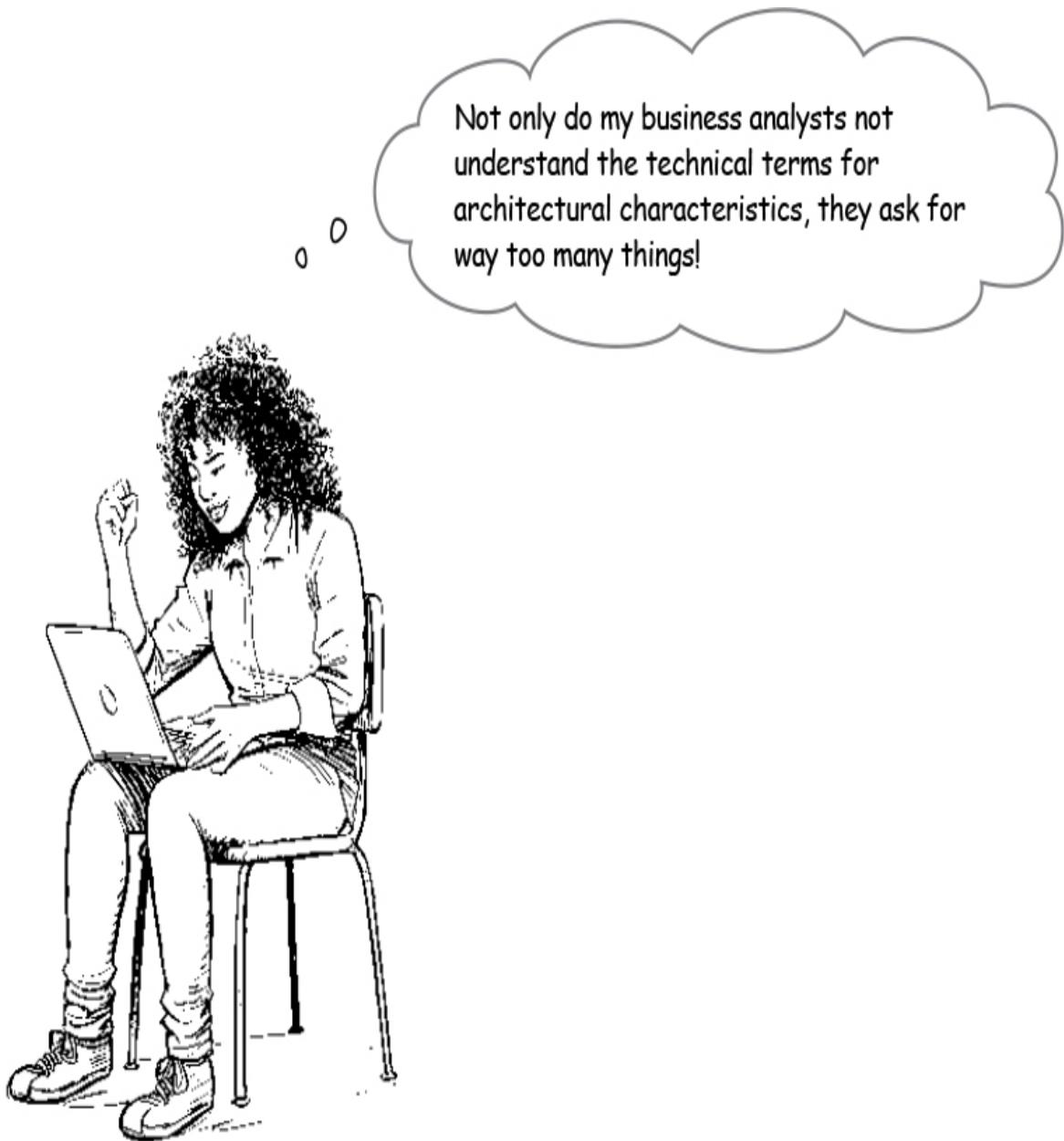
Good modularity allows for faster change without rippling side effects.

More of an architect-the-person characteristic

"The ability to update your resume..." Many people would rather not work in a place undergoing constant mergers.

Interoperability

Feasibility - evaluating whether something is possible - is an underutilized architecture "ility".



More requirements are NOT better.

What happens when an architect takes a list of possible architectural characteristics to a group of business users and asks them, “Which of these do you want for the X system?”

They invariably answer: “All of them!”

As nice as it would be to be able to accommodate that request, it's not a good idea to try.

Remember, architectural characteristics are synergistic with each other and with the problem domain. That means that the more architectural characteristics the system must support, the more complex its design must be.

When undertaking structural design for a system, architects must find a balance between domain priorities and the architectural characteristics necessary for success.

Balancing domain and architectural characteristics

Architects use architectural characteristics and logical component analysis to determine the appropriate architecture style. We need to strike a balance between the two.

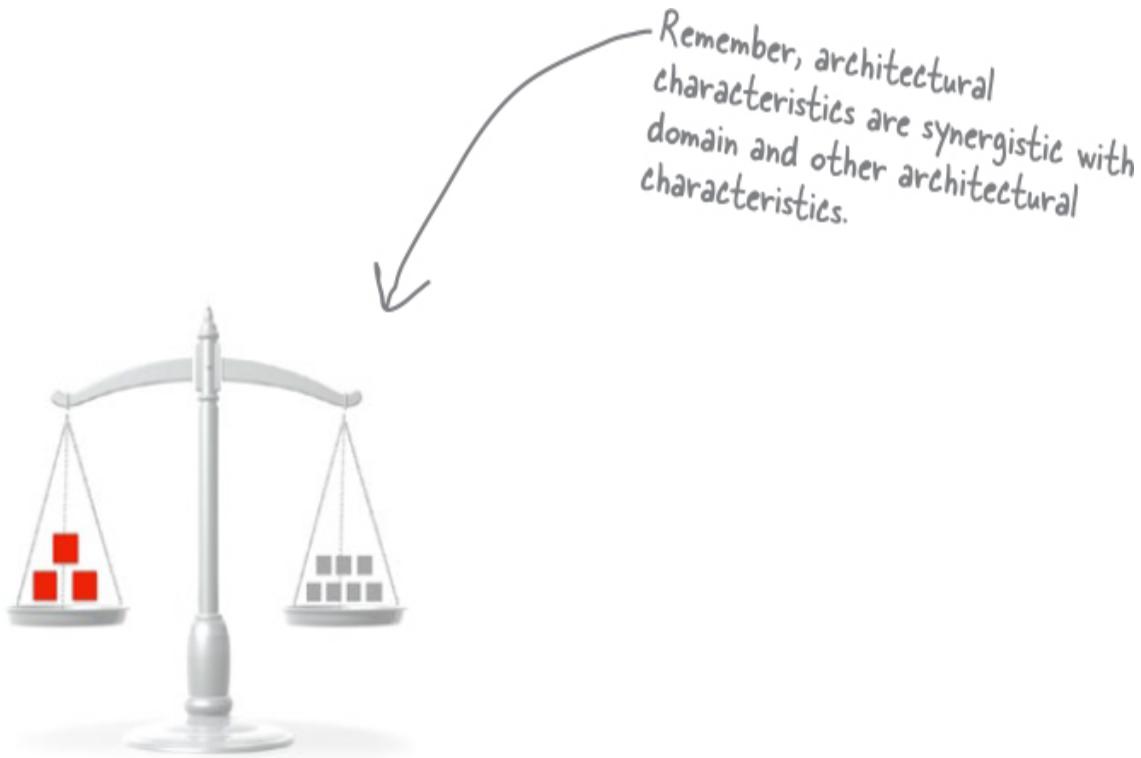
No architectural characteristics

Sometimes we don't take the time to analyze architectural characteristics before designing the system, leading to expensive and time-consuming re-work as we discover that we cannot achieve important goals.



Good balance between...

In this scenario, architects have achieved a balance in our design decisions between architectural characteristics and domain considerations.



This allows us to achieve operational and structural goals without overengineering.

...architectural characteristics & domain

Too many architectural characteristics

Unfortunately, architects sometimes retreat to an ivory tower and spend too much time analyzing architectural characteristics, or identify too many of them to be useful. This leads to overengineering, wasting time and effort for both implementation and ongoing maintenance.

Many systems that try to support too many architectural characteristics end up with too little space left to support the domain.



Limiting architectural characteristics

When the business stakeholders want *all* of the possible architectural characteristics, how can you limit their enthusiasm?

The Magic Number 7

NOTE

“The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information” is a famous paper from 1956 by psychologist George Miller.

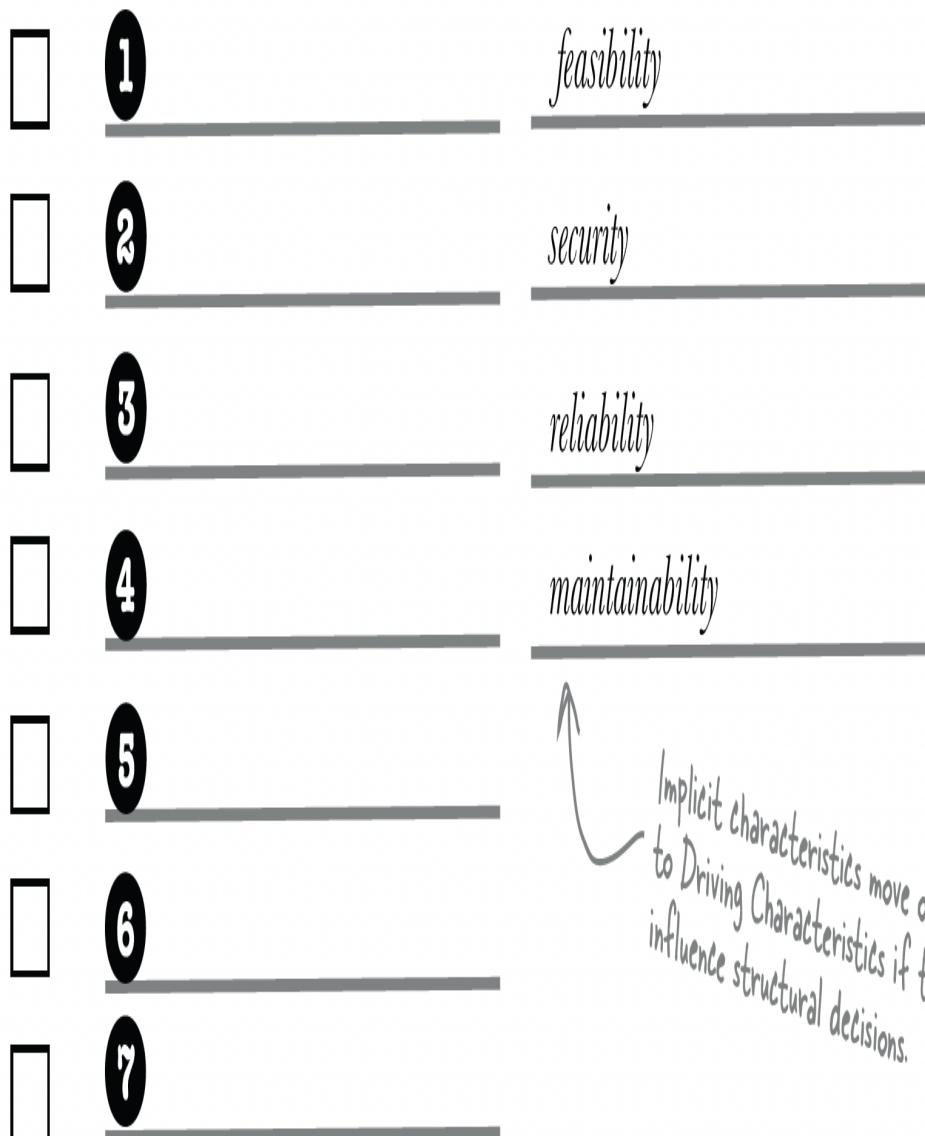
One useful guideline for the conversation between architects and business analysts is to limit the number of architectural characteristics they can choose to seven. Why seven? Psychological research indicates that people remember items in chunks of seven (one of the reasons that early phone numbers were seven digits). It’s also large enough to provide some variety without creating a paradox of choice by offering too many.

We created a worksheet to help architects work with other stakeholders to arrive at a reasonable number. This is a demo—you'll get to use it soon.

Top three most
important ones,
in any order

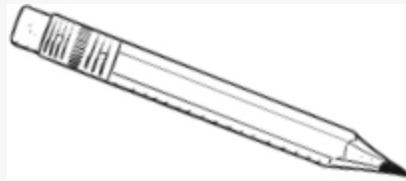
Top 3 Driving Characteristics Implicit Characteristics

Driving characteristics are
architectural characteristics that
drive important design decisions.



Implicit characteristics move over
to Driving Characteristics if they
influence structural decisions.

SHARPEN YOUR PENCIL



In “[The problem domain](#)”, you identified architectural characteristics for Sillycon Symposia’s social media application, Laffter. To make sure you have achieved a good balance, limit the number to seven. Then, check the boxes next to the top three most important.

Possible Candidate Architectural Characteristics

performance

responsiveness

availability

fault tolerance

scalability

elasticity

data integrity

data consistency

adaptability

extensibility

interoperability

concurrency

deployability

testability

configurability

customizability

recoverability

auditability

Top 3	Driving Characteristics	Implicit Characteristics
<input type="checkbox"/>	1	<u>feasibility</u>
<input type="checkbox"/>	2	<u>security</u>
<input type="checkbox"/>	3	<u>reliability</u>
<input type="checkbox"/>	4	<u>Maintainability</u>
<input type="checkbox"/>	5	
<input type="checkbox"/>	6	
<input type="checkbox"/>	7	

BULLET POINTS

- *Architectural characteristics* represent one part of the structural analysis that architects use to design software systems. (The other, logical components, appears in Chapter 3.)
- Architectural characteristics describe the capabilities of a system.
- Some architectural characteristics overlap with operational concerns (such as availability, scalability, and so on).
- There are many categories of architectural characteristics . No one can make a comprehensive list because the software development ecosystem constantly changes.
- When identifying architectural characteristics, architects look for factors that influence structural design.
- Architects should be careful not to specify too many architectural characteristics, because they are synergistic— changing one requires other parts of the system to change.
- Some architectural characteristics are *implicit* : not explicitly stated in requirements, yet part of an architect's design consideration.
- Some architectural characteristics may appear in multiple categories.
- Many architectural characteristics are cross-cutting: they interact with other parts of (and decisions in) the organization.
- Architects must derive many architectural characteristics from requirements and other domain design considerations.
- Some architectural characteristics come from domain and/or environmental knowledge, outside of the requirements of a specific application.

- Some architectural characteristics are *composites* : they consist of a combination of other architectural characteristics.
- Architects must learn to translate “business speak” into architectural characteristics.
- Architects should limit the number of architectural characteristics they consider to some small number, such as seven.

EXERCISE SOLUTION



From “Sharpen your pencil”

See if you can identify the domain bits within the requirements for Sillycon Symposia’s Laffter social media app. For example, Users and Friends are part of the domain. What else belongs to the domain in this example?

1. Joke/Pun

- 2. HaHa/Giggle**
3. Speakers
4. Conferences/events
5. Forums
6. Messages
7. External links

Is there anything in the project overview that doesn’t fit within the domain? For example, the application needs to be scalable (support lots of concurrent users) during the conferences. What other features are listed in the project summary but are not part of the system problem domain?

EXERCISE SOLUTION



From “Exercise”

Welcome to “Take It or Leave It!” The rules of this game are simple—we’ll give you a business requirement that might come up for the Laffter application, and two architectural characteristics. As you know, everything in architecture is a trade-off. So if you attempt to optimize one, you are probably not going to do as well with the other. Your job is to tell us which characteristic rate as a higher priority for that requirement.

“We need to get this to market ASAP!”

fault tolerance

agility

“Money’s tight, folks!”

scalability

simplicity

“Oh, wow, this conference is going to be our biggest yet.”

high availability

maintainability

“We want to start storing user credit card information.”

security

recoverability

“This site is going to be very popular upon launch.”

agility

elasticity

EXERCISE SOLUTION



From “Exercise”

Domain requirements are often a rich source for architectural characteristics. For example, our Laffter social media application needs to support hundreds of users, so scalability will be one necessary architecture characteristic. Can you uncover more?

Sillycon Symposia is hosting a social media network of like-minded technologists.

Architectural characteristics

Users: hundreds of speakers, thousands of users

Scalability

Requirements:

- Users can register for user names and approve privacy policy
- Users can add new content on Laffter as a “Joke” (long-form post) or “Pun” (short-form post)
- Followers can “HaHa” (indicating strong approval) or “Giggle” (a milder approval message) content they like
- Speakers at Sillycon Symposia events have a special icon
- Speakers can host forums related to their content
- Users can post messages of up to 281 characters
- Users can also post links to external content

Elasticity

Authorization

Authentication

Internationalization

Customizability

Additional Context:

- International support
- Very small support staff
- ‘Bursty’ traffic: extremely busy during live conferences

EXERCISE SOLUTION



From “Solution versus problem”

It's sometimes difficult to untangle requirements from solutions. Here are some phrases you might have heard in your own company that indicate something might be one or the other. Can you identify the requirements and the solutions?

“We need a system to track user preferences and customizations, and save them between sessions.”



requirement



solution

“Do we really need to build our own survey service? Surely we can find one that does what we need.”



requirement



solution

“An enterprise service bus solves some of our current problems (albeit with some changes and workarounds) and offers extreme extensibility.”



requirement



solution

“According to the friendly sales rep, this package software does all the things Accounting needs, now and in the near future.”



requirement



solution

EXERCISE SOLUTION



From “Sharpen your pencil”

Context matters. At the top, we've listed several architectural characteristics. Below that are three application scenarios. For each scenario, rank each architecture characteristic based on how important it is for that type of application. (Hint: Some applications won't need all of them.)

B. Scalability

A. Performance

D. Extensibility

C. Security

E. Data integrity

Scenario #1

Building an e-commerce site in a competitive market

Scenario #2

A system for an enterprise whose goal is to grow via mergers

An application to automate standardized testings and grading for university admissions

1 C. Security

1 D. Extensibility

1 E. Data integrity

2 A. Performance

2 B. Scalability

2 C. Security

3 B. Scalability

3 _____

3 A. Performance

4

4 _____

4 _____

5

5 _____

5 _____

EXERCISE SOLUTION



From “Sharpen your pencil Solution”

In “Exercise”, you identified architectural characteristics for the Sillycon Symposia Laffter social media application. However, to make sure you have achieved a good balance, limit the number to seven. Then, check the boxes next to the top three most important.

Possible Candidate Architectural Characteristics

performance

responsiveness

availability

fault tolerance

scalability

elasticity

data integrity

data consistency

adaptability

extensibility

interoperability

accessibility

testability

configurability

customizability

recoverability

auditability

portability

Top 3 Driving Characteristics Implicit Characteristics

<input type="checkbox"/>	1 Scalability	<i>feasibility</i>
<input type="checkbox"/>	2 Security	<i>security</i>
<input type="checkbox"/>	3 Elasticity	<i>reliability</i>
<input type="checkbox"/>	4 Responsiveness	<i>maintainability</i>
<input type="checkbox"/>	5 Performance	
<input type="checkbox"/>	6 Portability	
<input type="checkbox"/>	7 Accessibility	

Remember, no single correct answer exists. The question is: can you justify your choices?

Chapter 3. 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 3-1.

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 the 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 3-2.

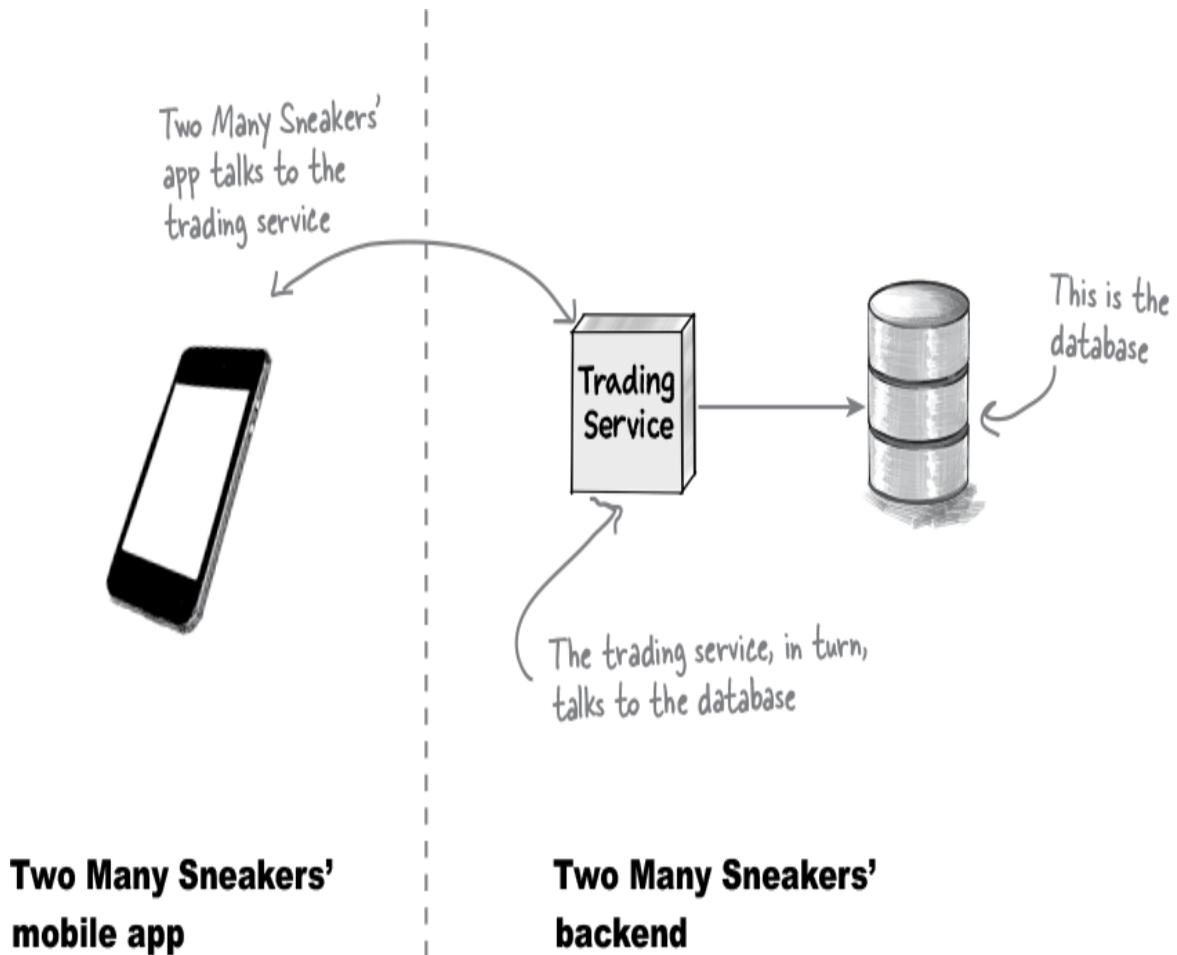


Figure 3-3.

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 knockoffs, 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 3-4.

The database is still part of the architecture. We just left it out in this diagram.

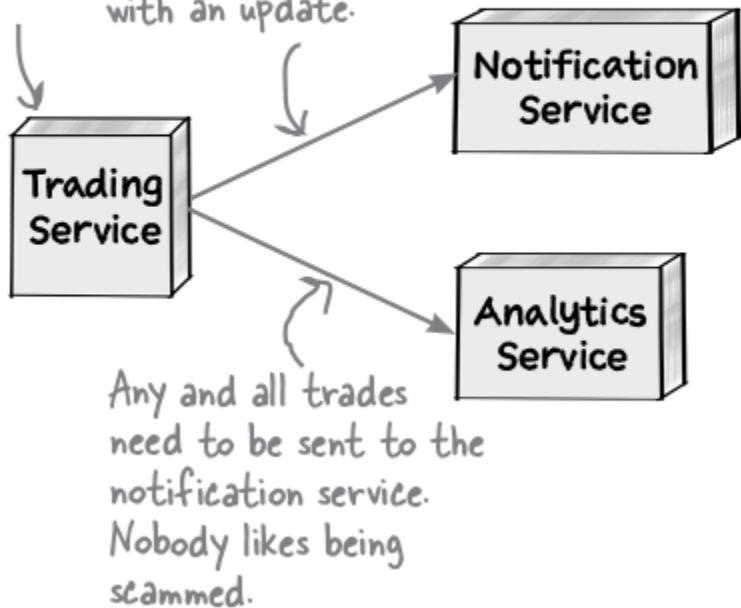


Figure 3-5.

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.*

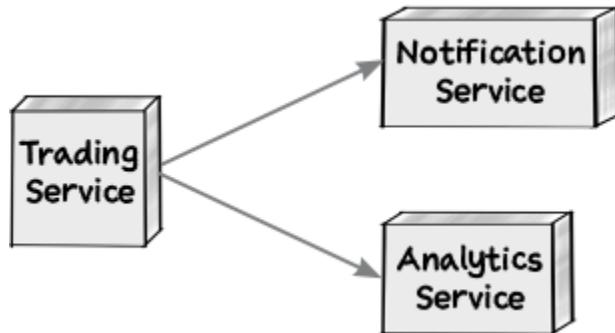


Figure 3-6.

- ☒ *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**.*

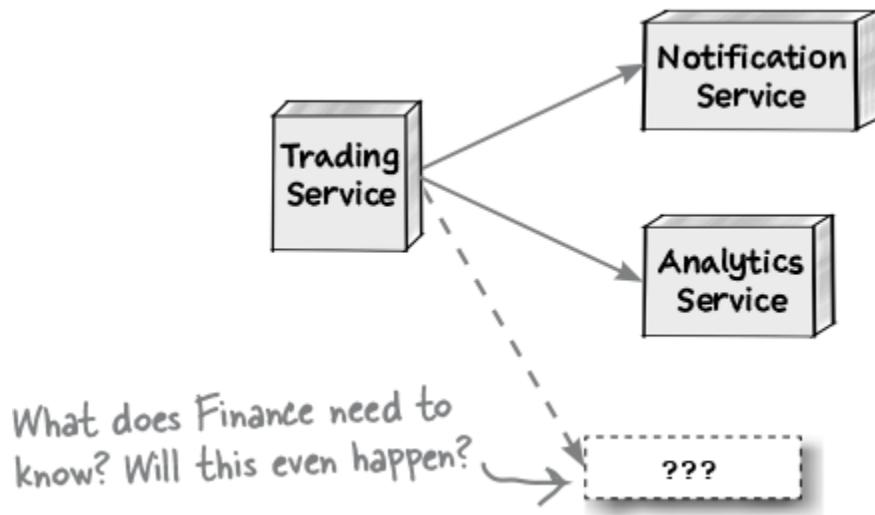


Figure 3-7.

- ☒ *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 3-8.

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:

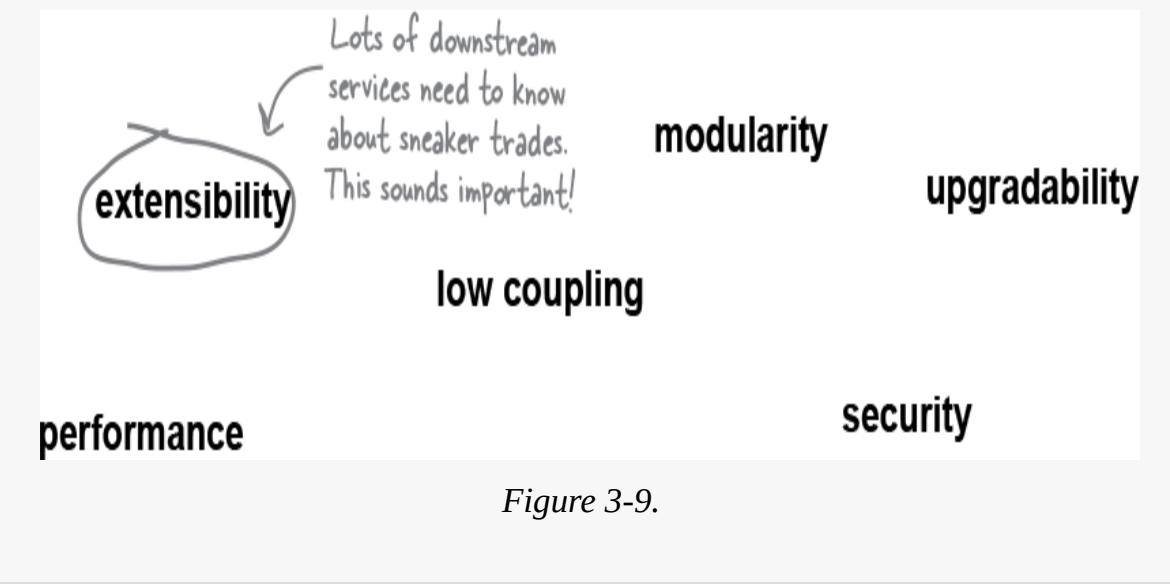


Figure 3-9.

BRAIN POWER



Figure 3-10.

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.”

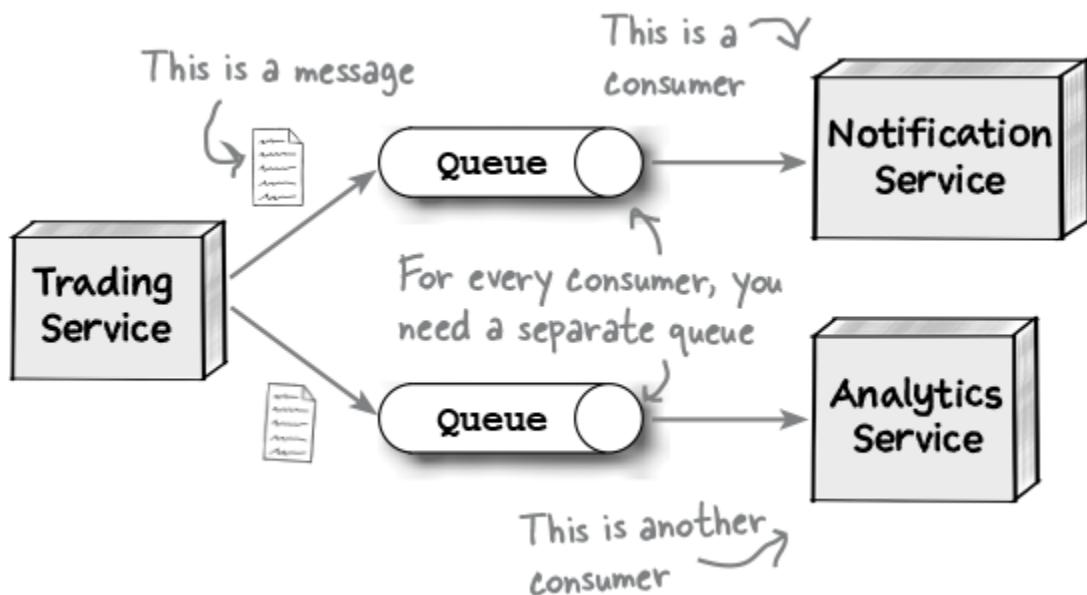


Figure 3-11.

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.

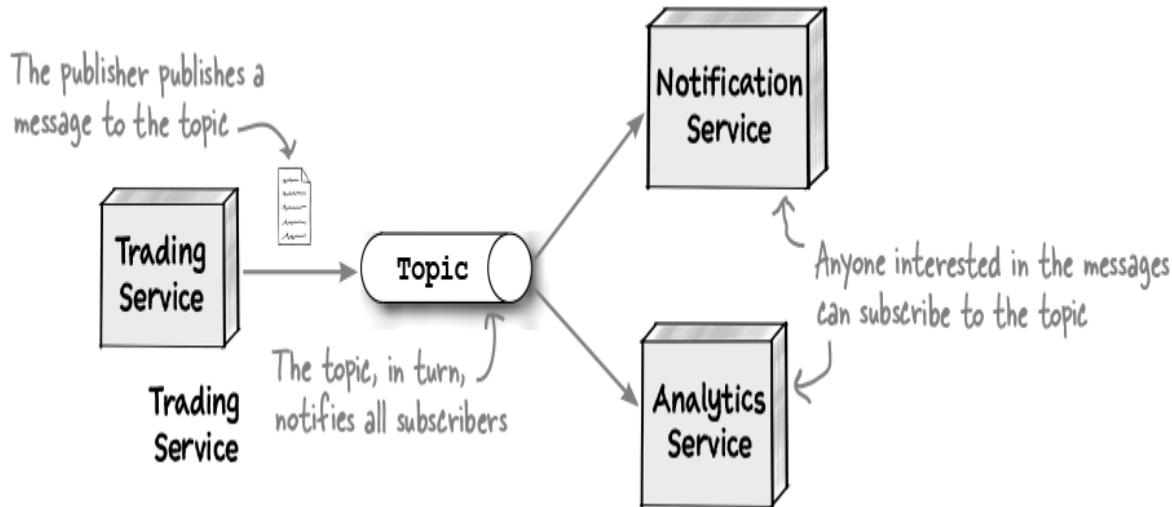


Figure 3-12.

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 3-13.

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 another (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.

Just a friendly reminder of what using a queue would look like, so you don't have to flit back and forth.

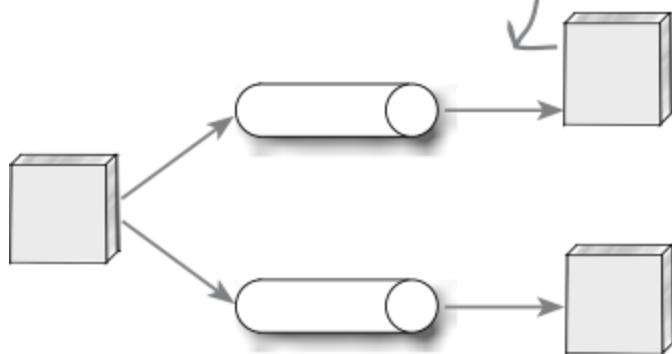


Figure 3-14.

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.

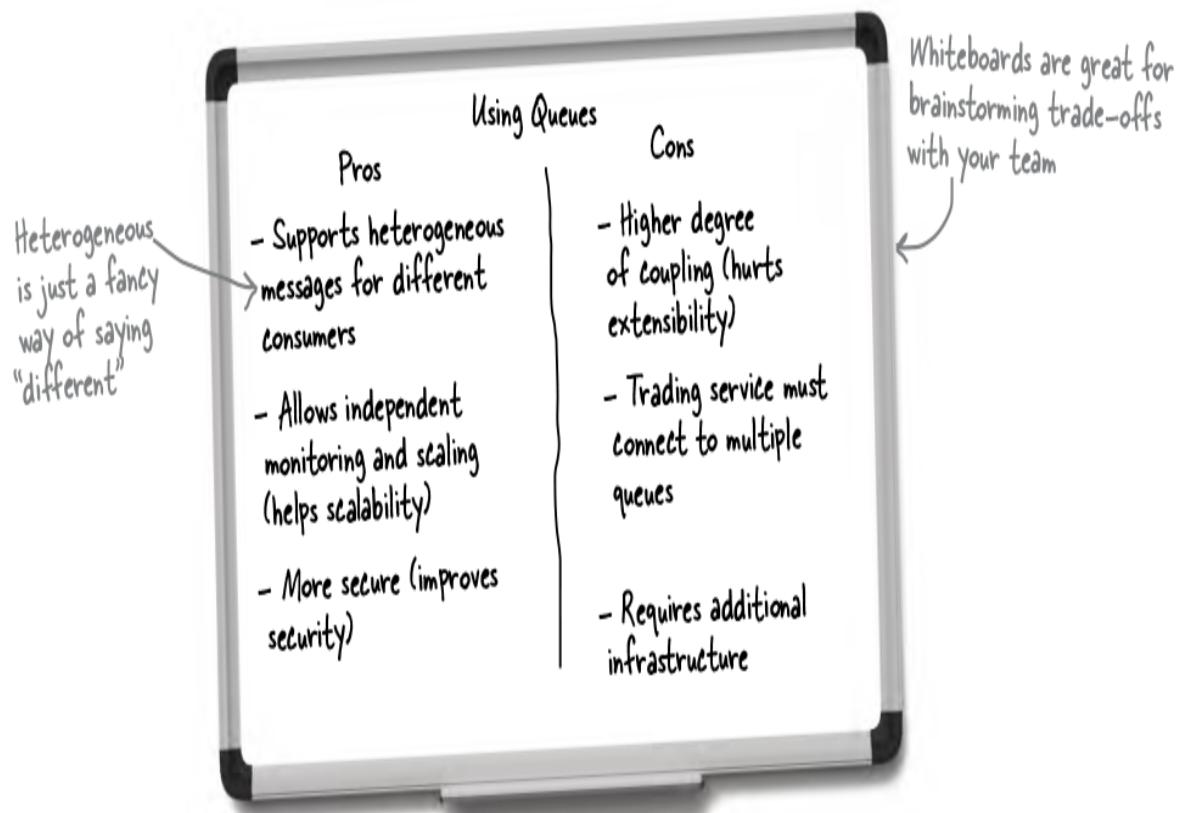


Figure 3-15.

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.

This is what using a topic looks like

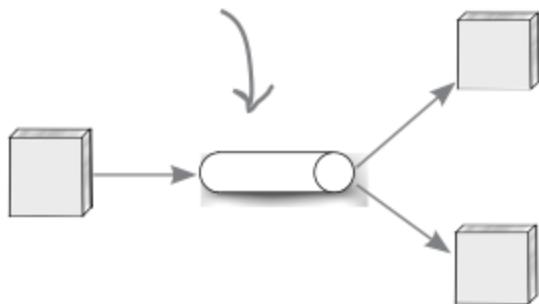


Figure 3-16.

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 3-17.

SHARPEN YOUR PENCIL

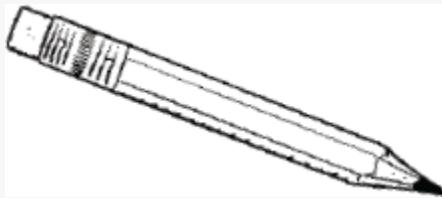


Figure 3-18.

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

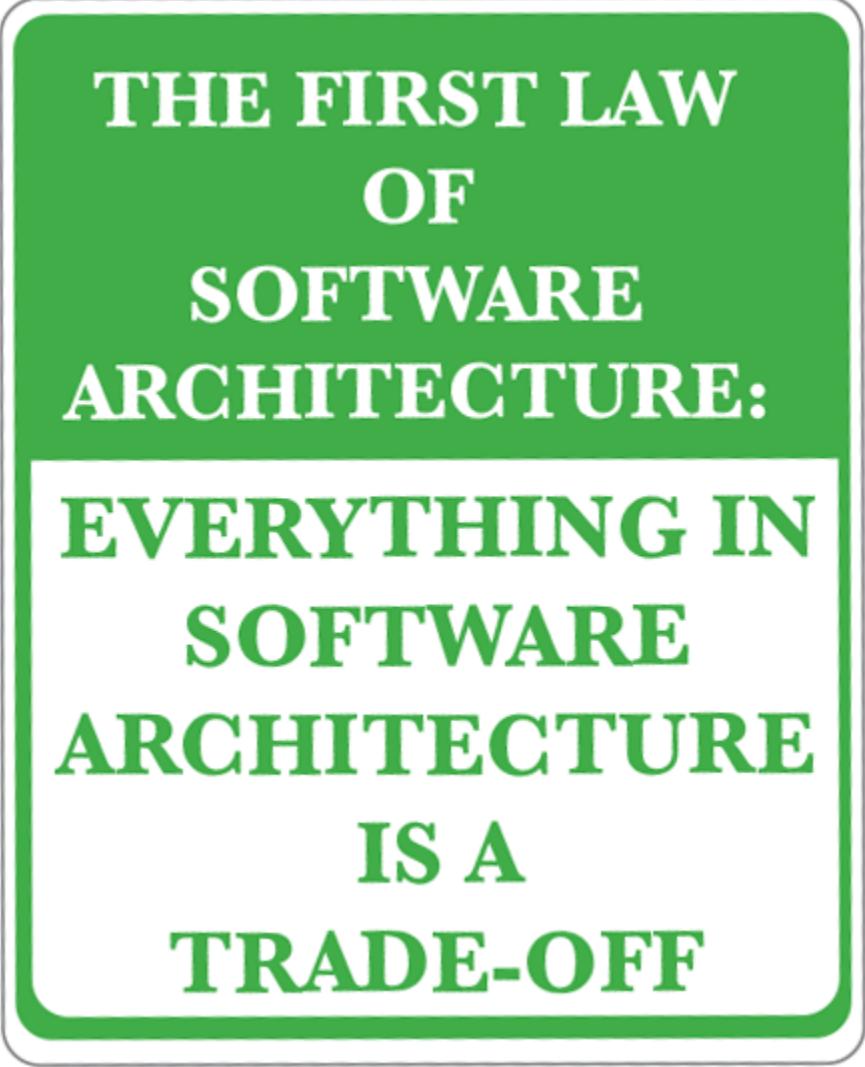
extensibility is its biggest priority. That means we should pick the topic option.

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**.



Figure 3-19.



**THE FIRST LAW
OF
SOFTWARE
ARCHITECTURE:**

**EVERYTHING IN
SOFTWARE
ARCHITECTURE
IS A
TRADE-OFF**

Figure 3-20.

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

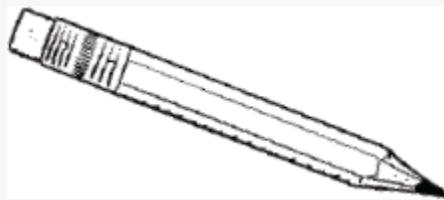


Figure 3-21.

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.

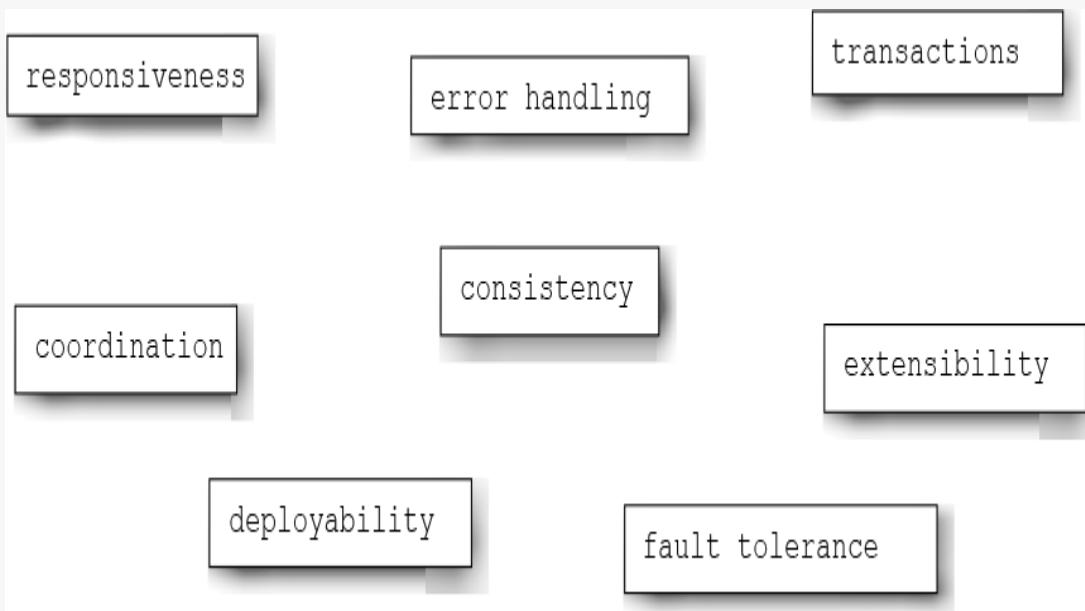


Figure 3-22.

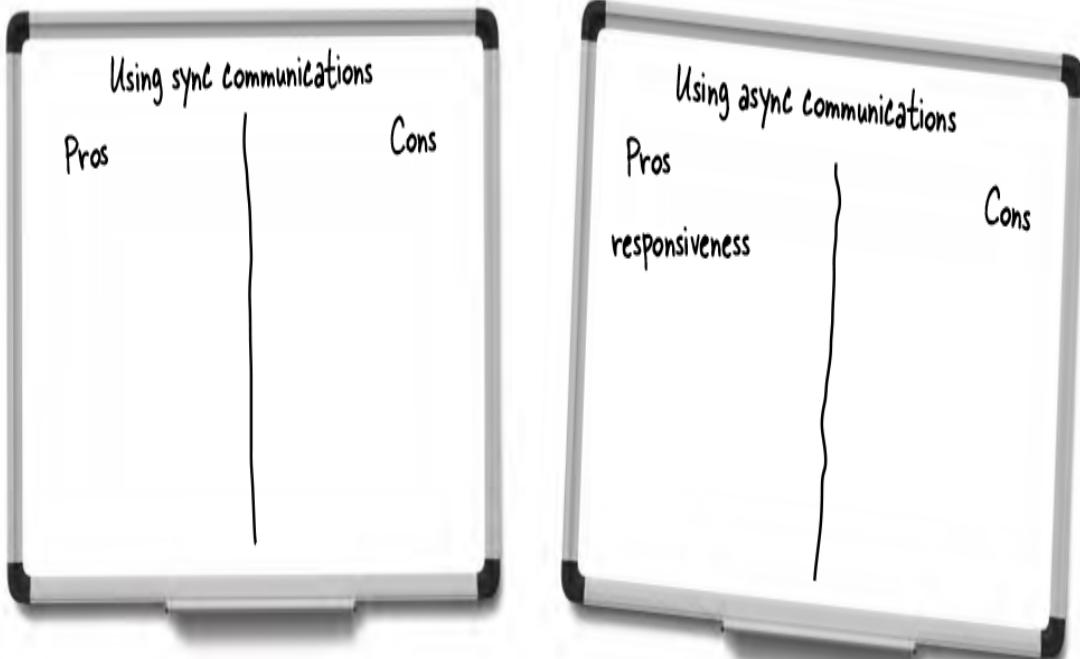


Figure 3-23.

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 limitations—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.

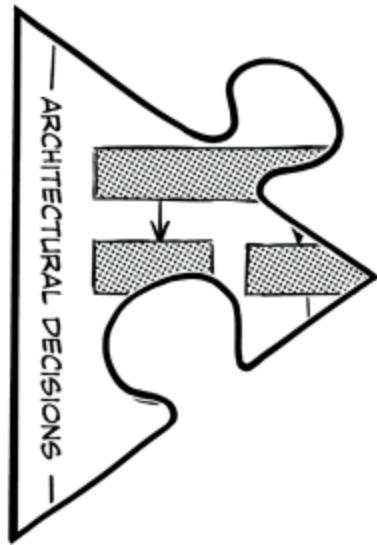


Figure 3-24.



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

Figure 3-25.

We will use a cache to reduce the load on the database and improve performance.

Notice how this decision introduces an additional piece of infrastructure. It's also something the implementing team must keep in the back of their minds when accessing or writing data.

Figure 3-26.

We will build the reporting service as a modular monolith.

This one is pretty obvious—
it literally describes the
structure of a service.

Figure 3-27.

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:

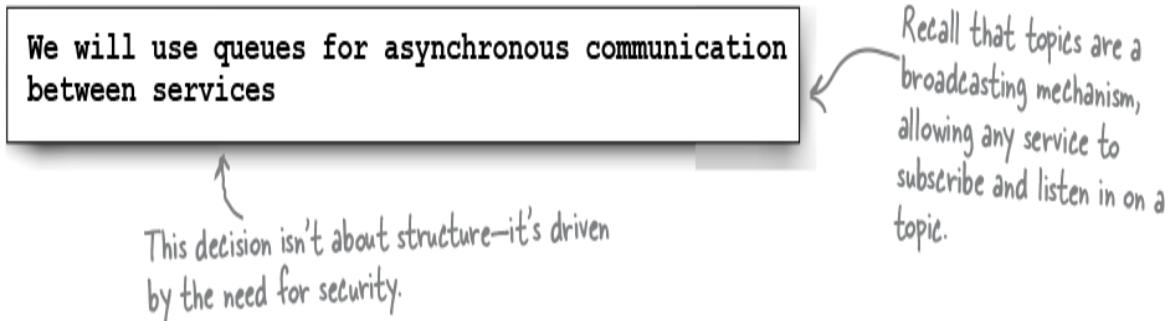


Figure 3-28.

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:

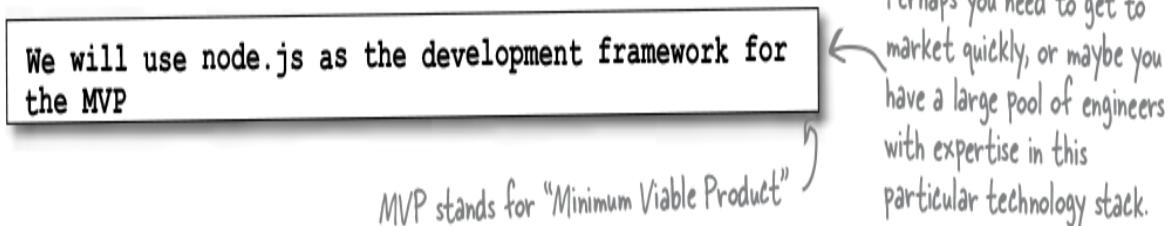


Figure 3-29.

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

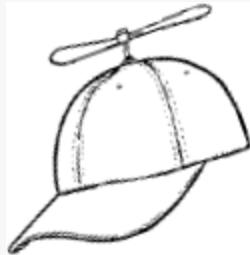


Figure 3-30.

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>.



Figure 3-31.

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**.

**THE SECOND LAW
OF
SOFTWARE
ARCHITECTURE:**

**WHY
IS MORE
IMPORTANT
THAN
HOW**

Figure 3-32.

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.

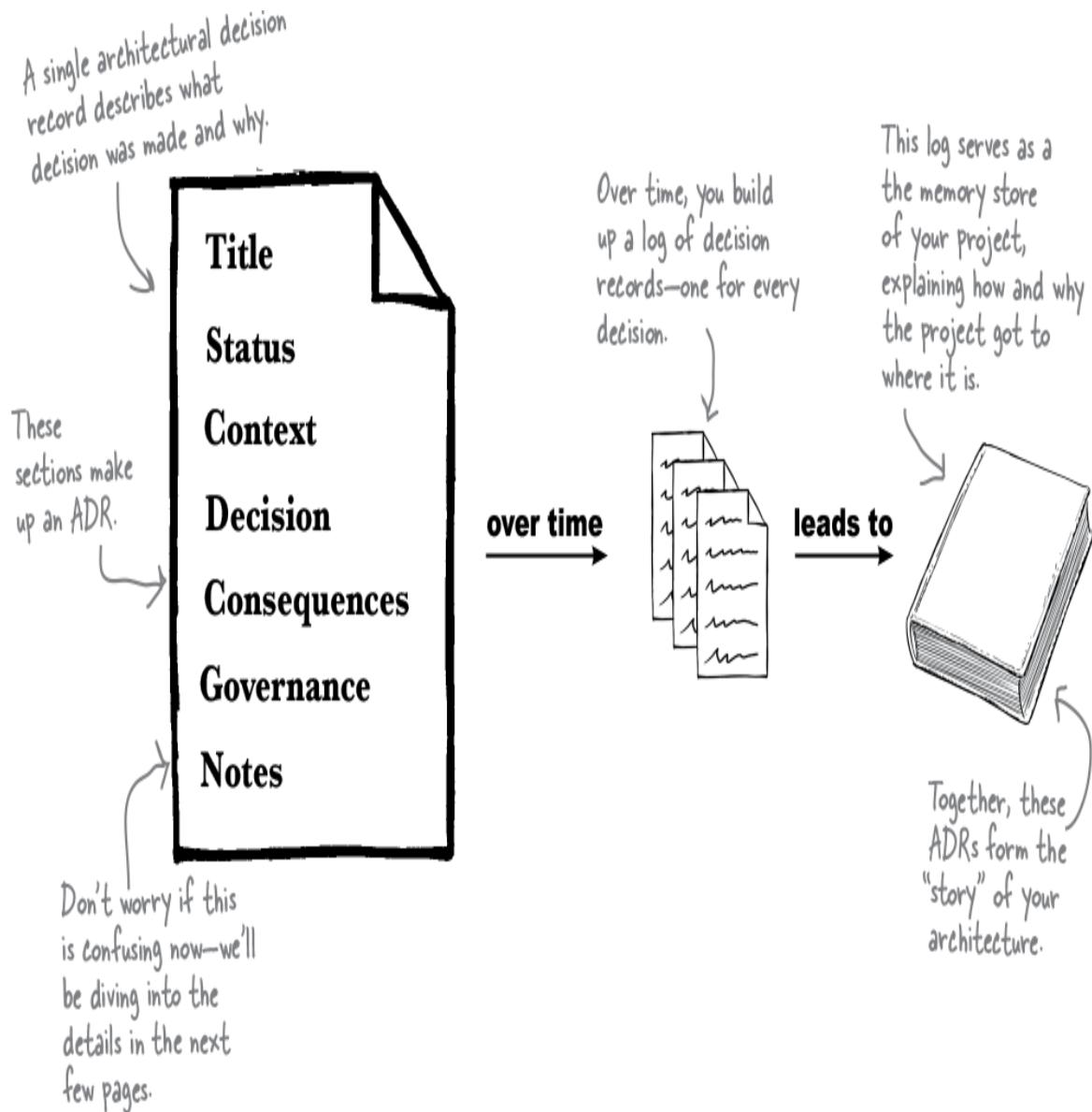


Figure 3-33.

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

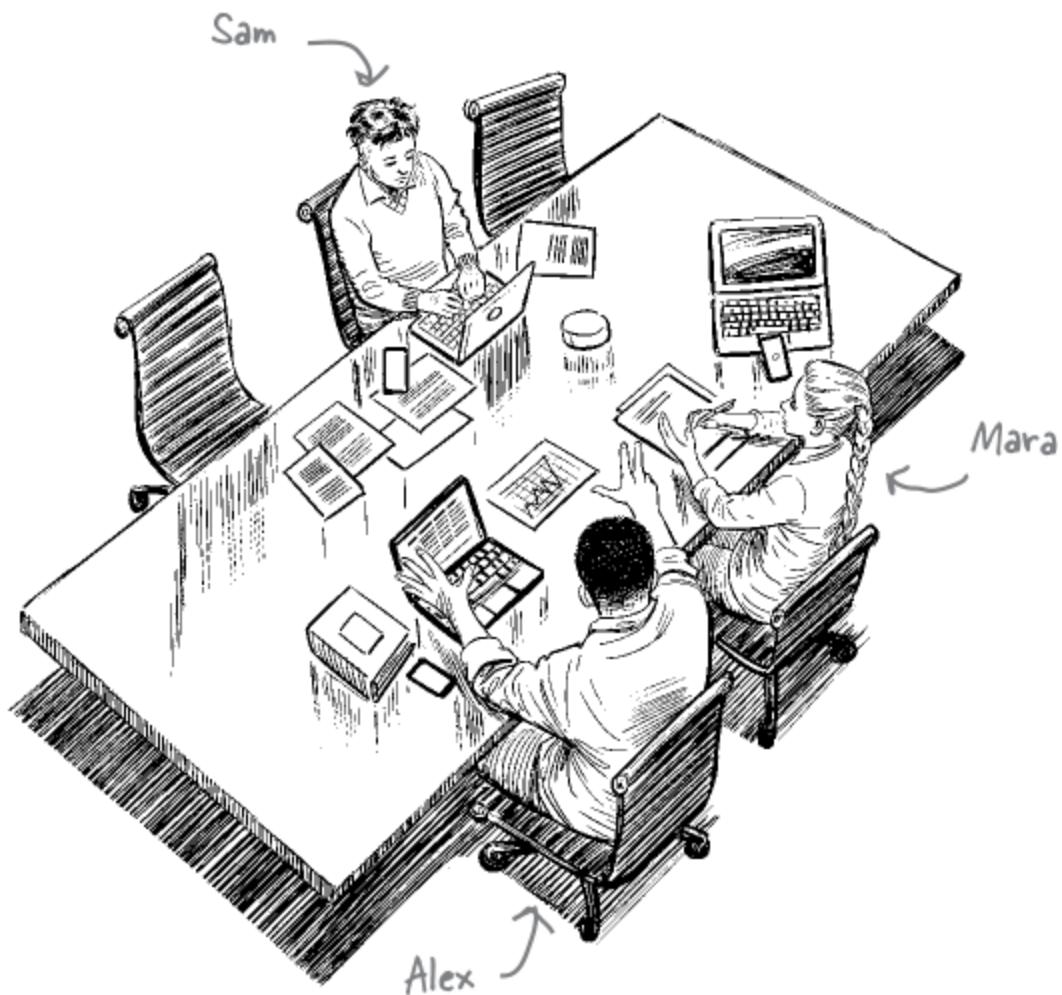


Figure 3-34.

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:

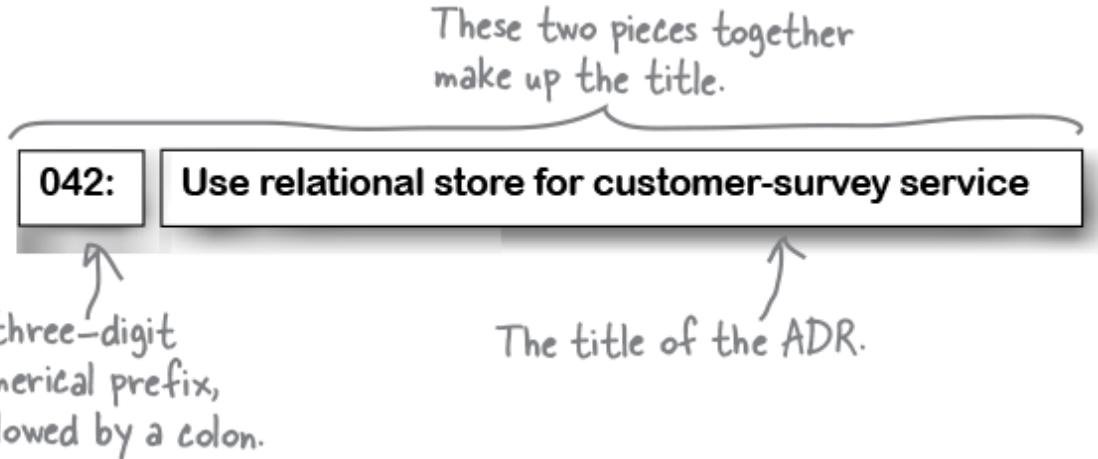


Figure 3-35.

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.

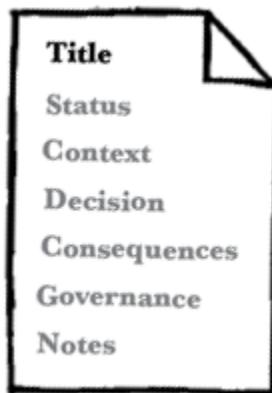


Figure 3-36.

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 3-37.

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.

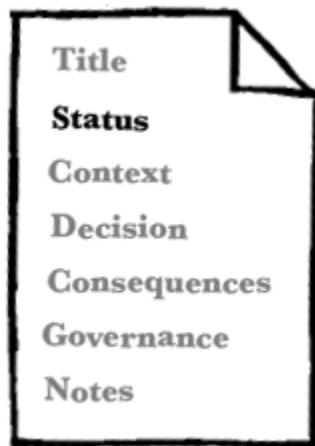


Figure 3-38.



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:

The previous ADR

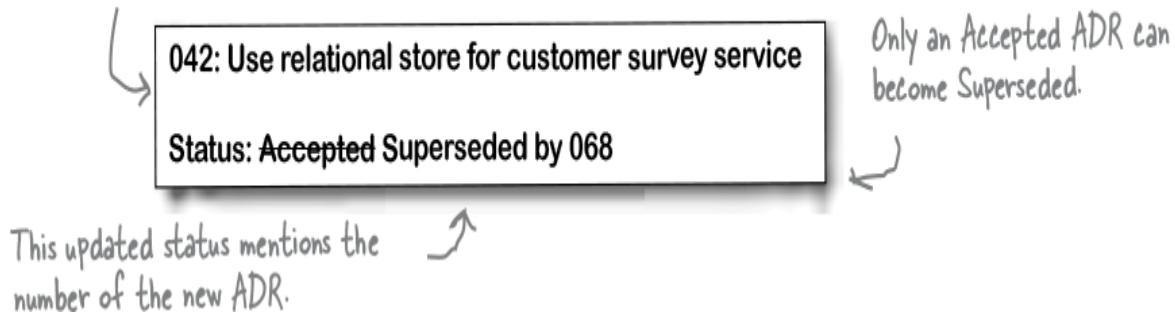


Figure 3-39.

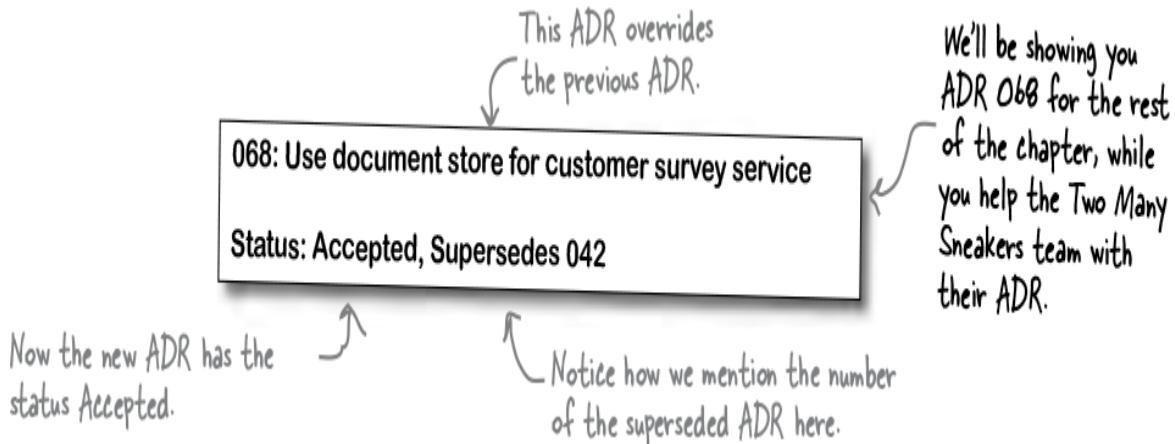


Figure 3-40.

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 3-41.

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.

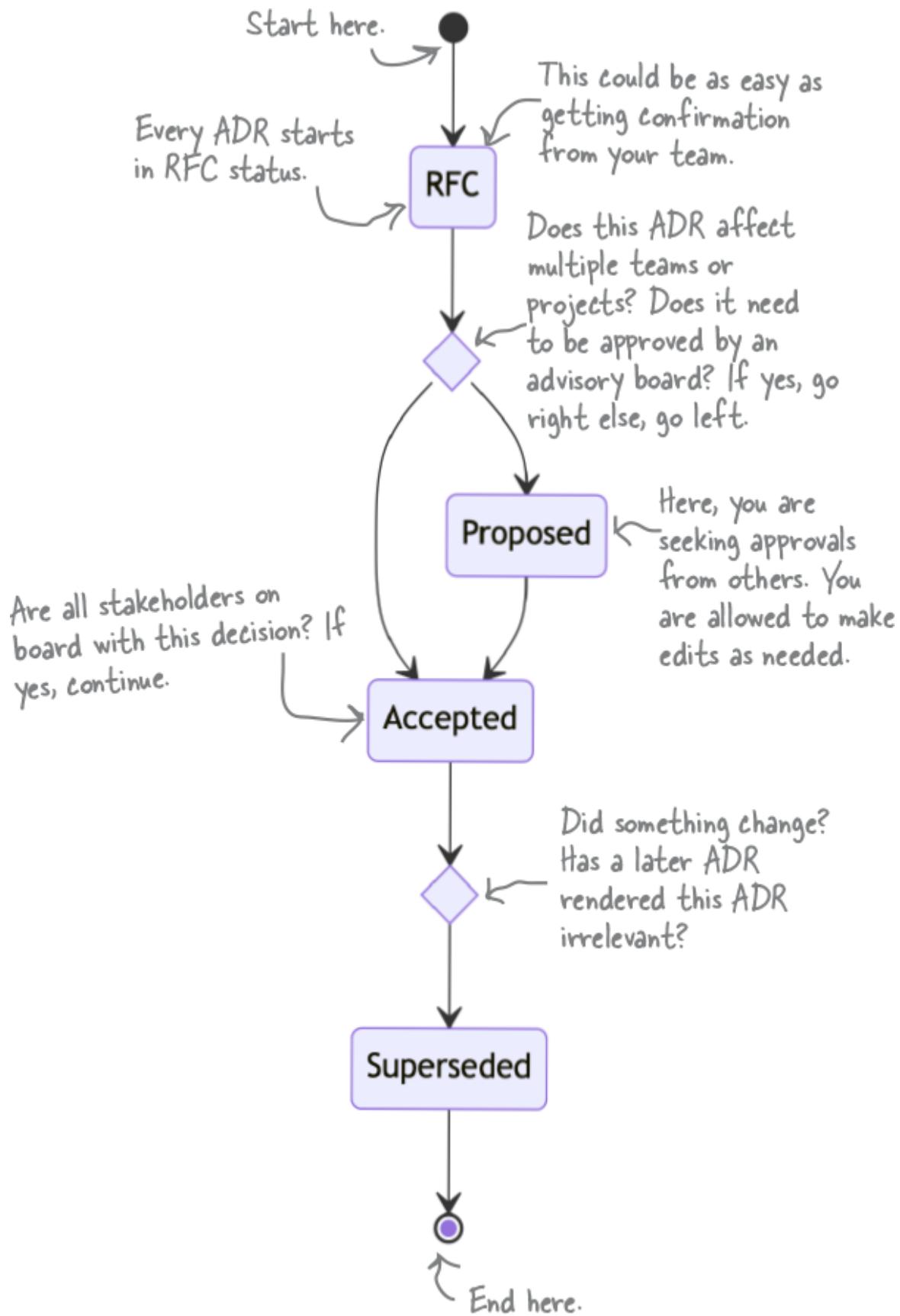


Figure 3-42.

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.

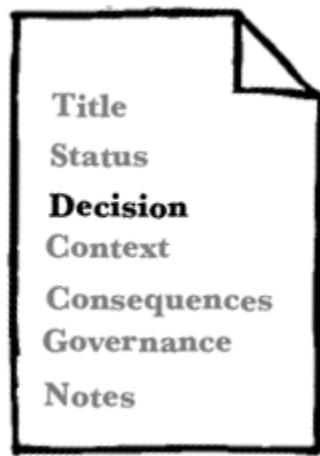


Figure 3-43.

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

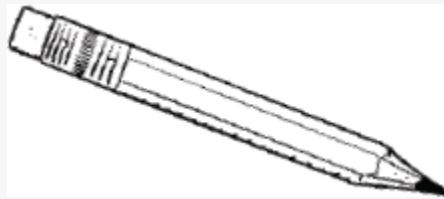


Figure 3-44.

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:

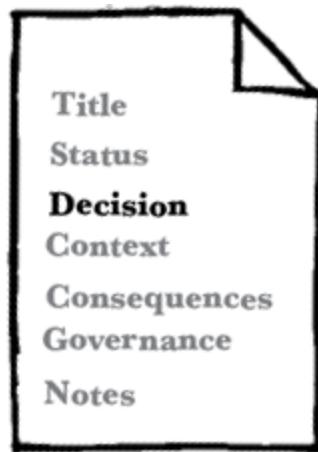


Figure 3-45.

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 3-46.

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 3-47.

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!)

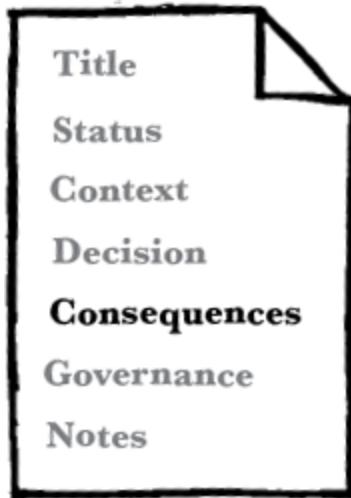


Figure 3-48.

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

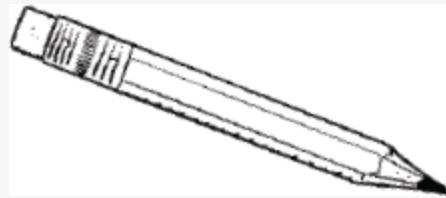


Figure 3-49.

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 3-50.

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

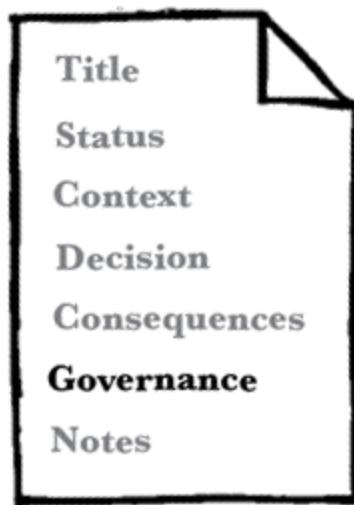


Figure 3-51.

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

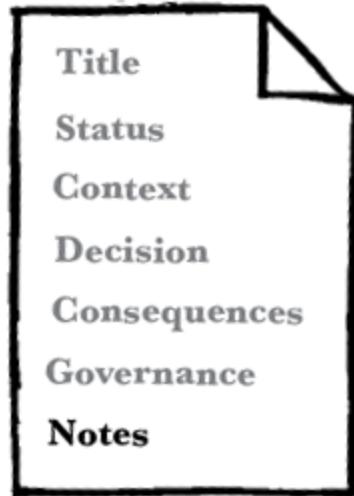


Figure 3-52.

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 3-53.

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.")

Title

Status

Context

Decision

Consequences

Figure 3-54.

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 recommend 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:

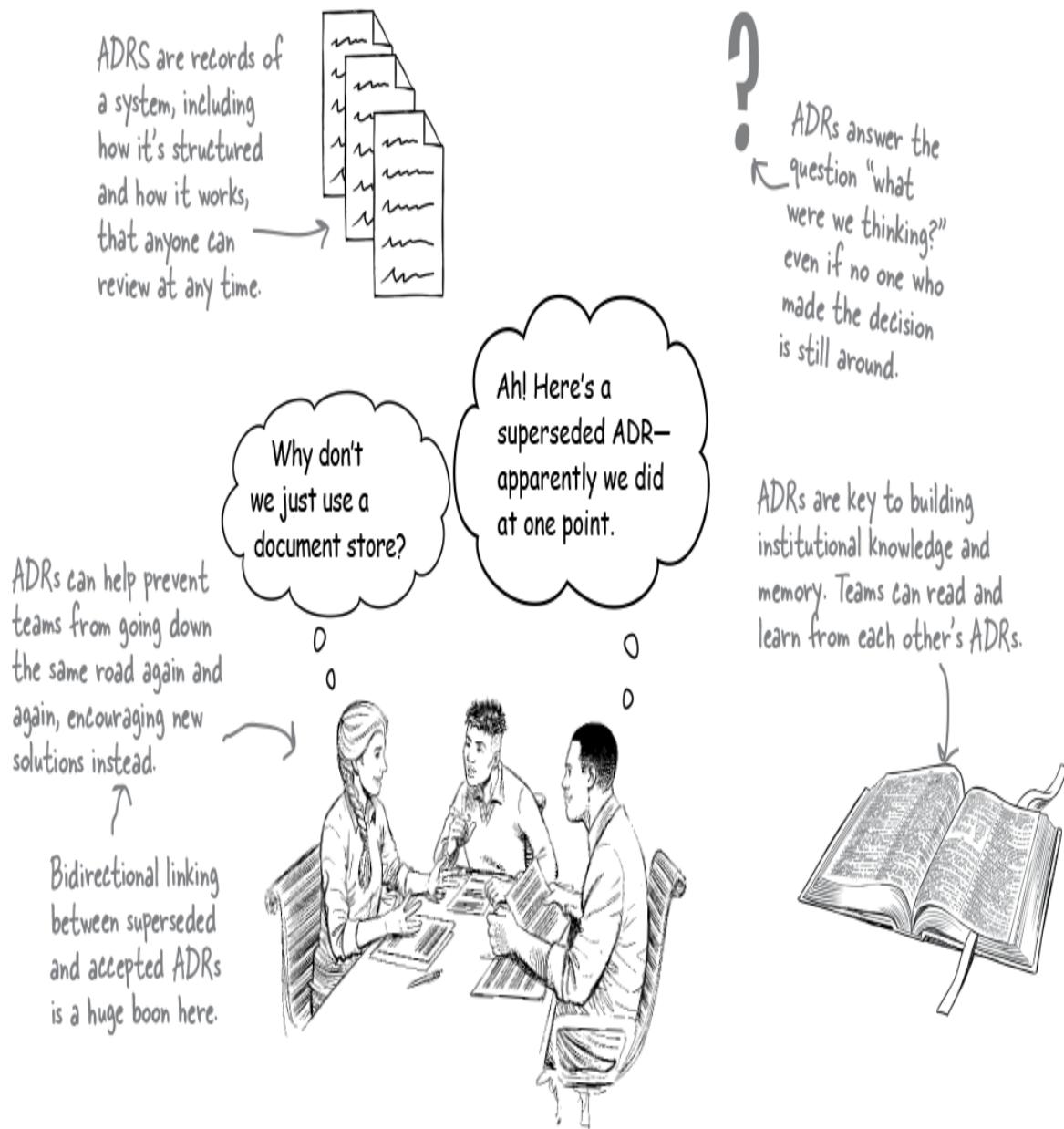


Figure 3-55.

WATCH IT!



Figure 3-56.

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.

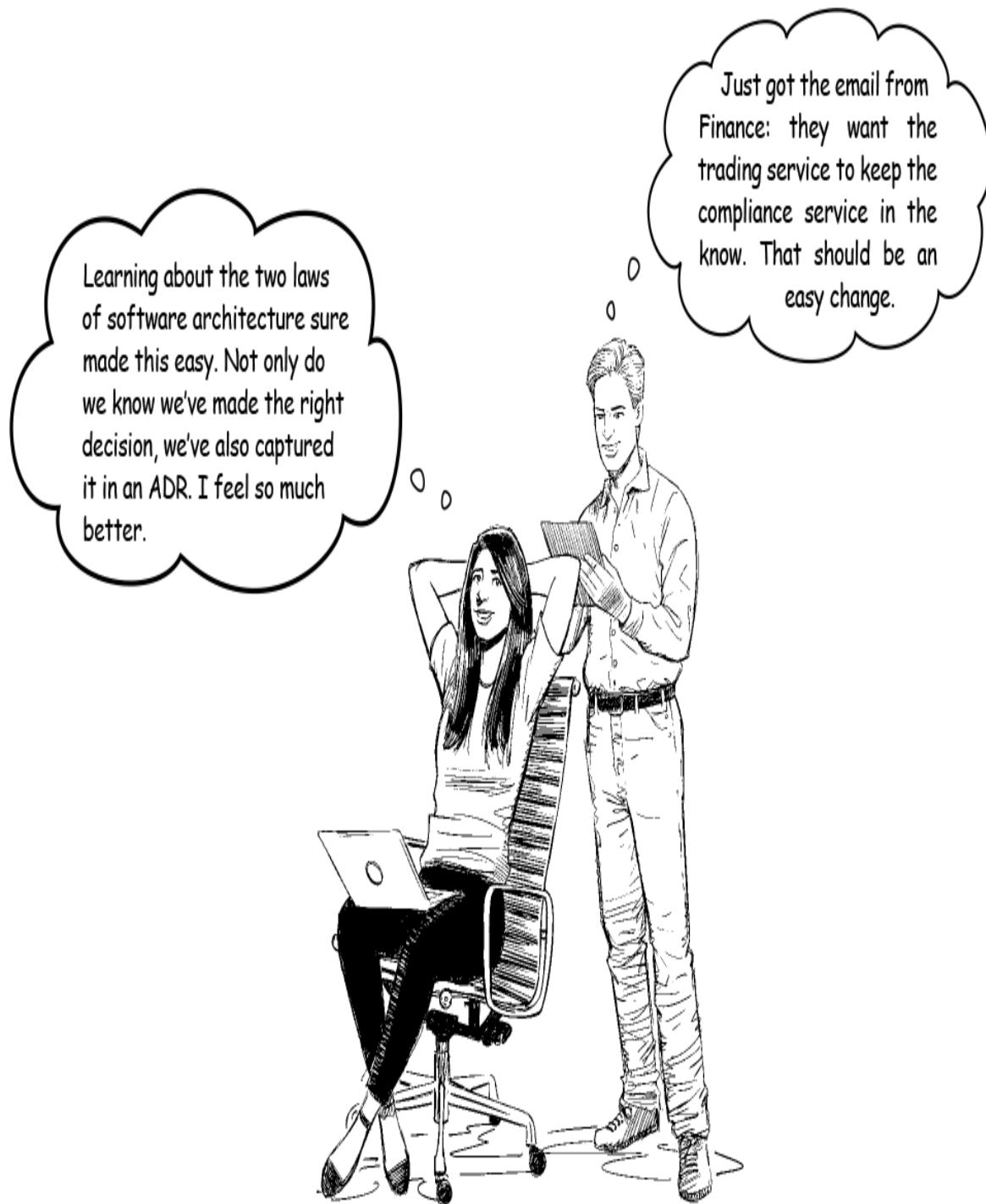


Figure 3-57.

Grokked 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 3-58.

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

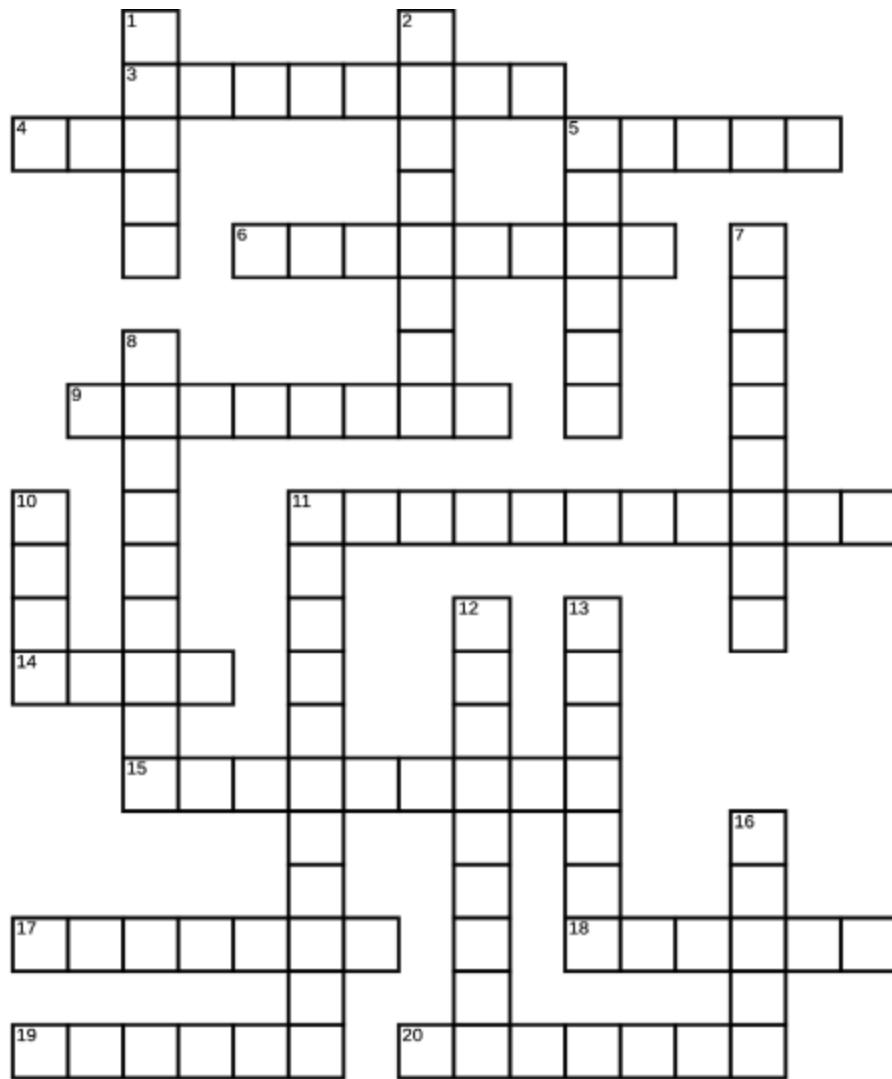


Figure 3-59.

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 3-60.

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:

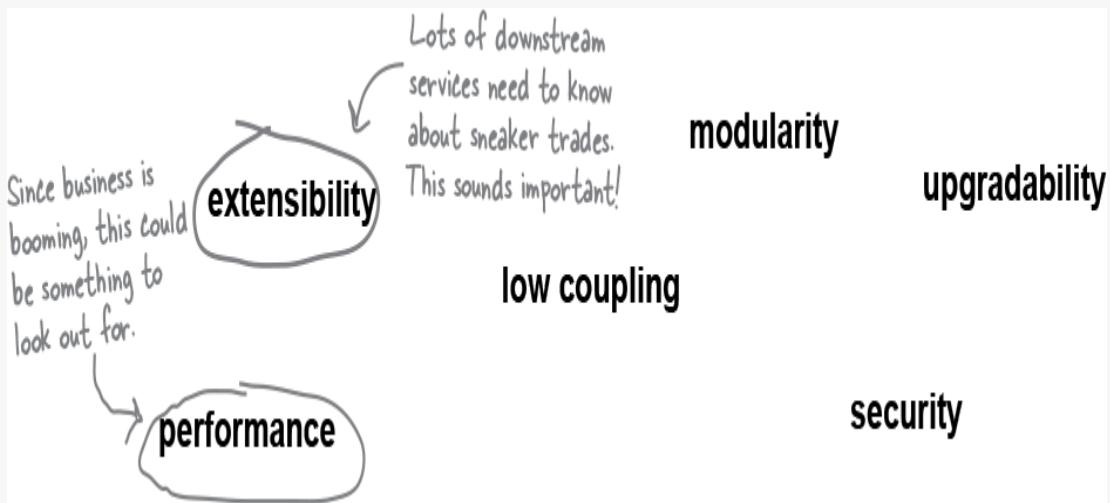


Figure 3-61.

SHARPEN YOUR PENCIL

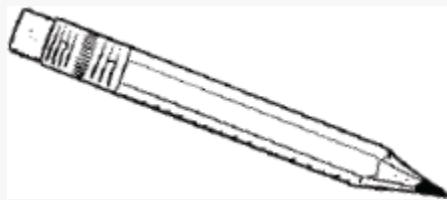


Figure 3-62.

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 3-63.

SHARPEN YOUR PENCIL SOLUTION

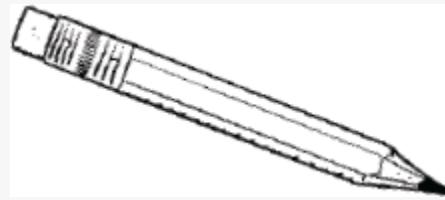


Figure 3-64.

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.

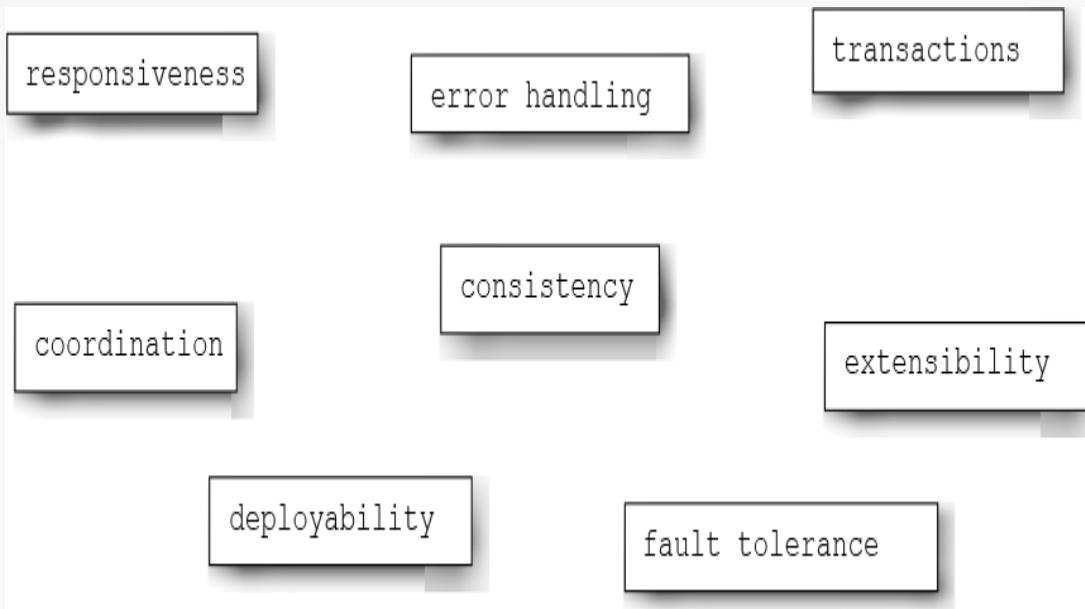


Figure 3-65.

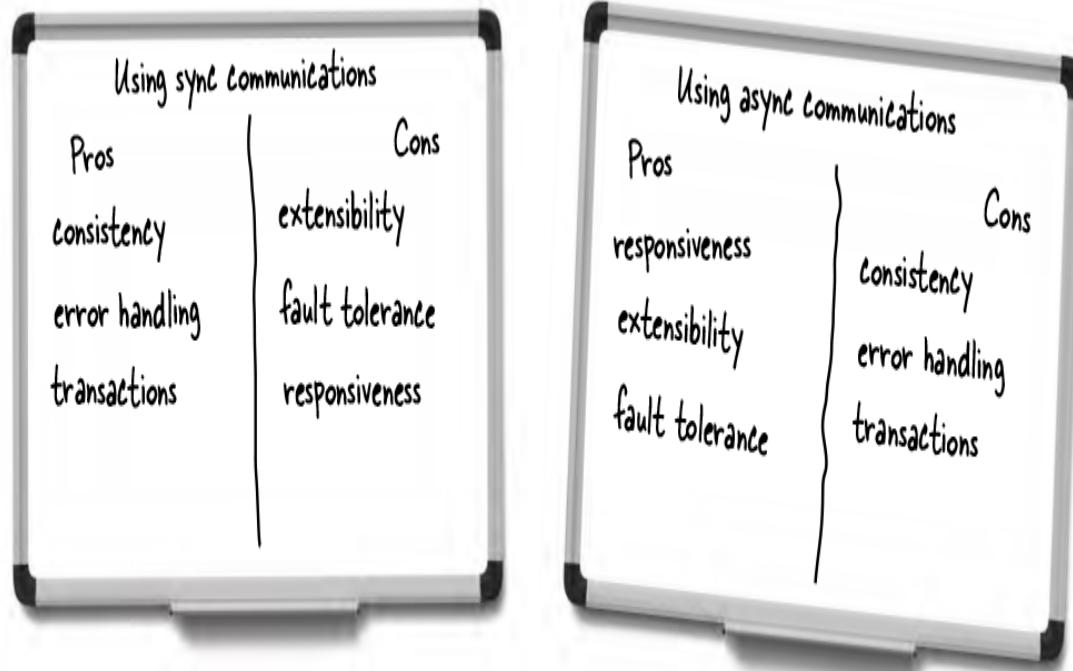


Figure 3-66.

EXERCISE SOLUTION



Figure 3-67.

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 3-68.

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 3-69.

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 3-70.

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 3-71.

SHARPEN YOUR PENCIL SOLUTION

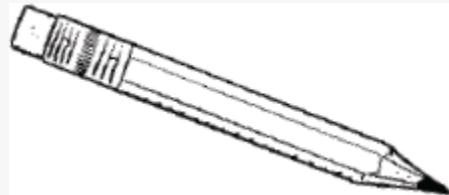


Figure 3-72.

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 3-73.

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

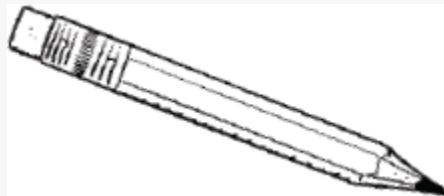


Figure 3-74.

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 infrastructure. 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 3-75.

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

O12: 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 infrastructure. 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 3-76.

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

The “two laws” crossword solution



Figure 3-77.

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

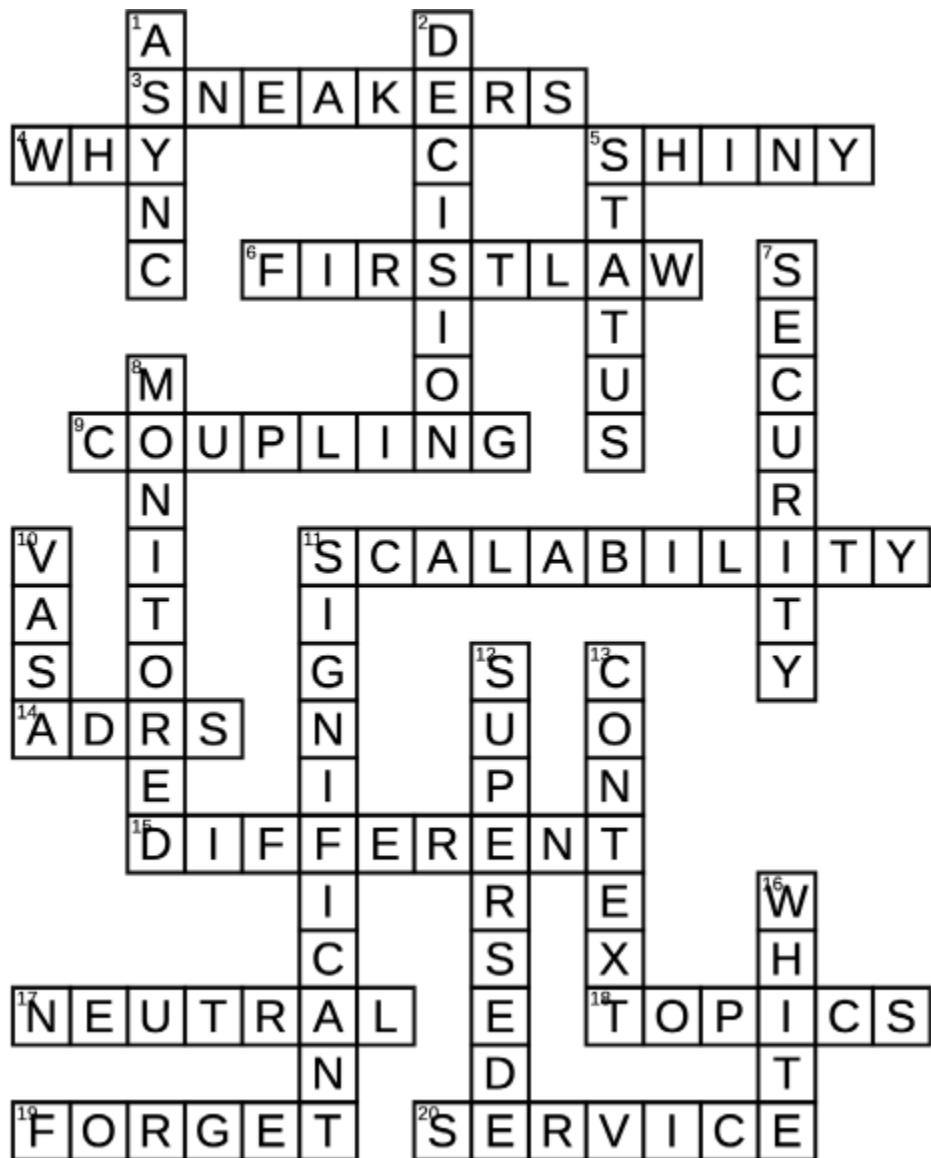


Figure 3-78.

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

Chapter 4. Logical Components: *The building blocks*

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 4th 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)



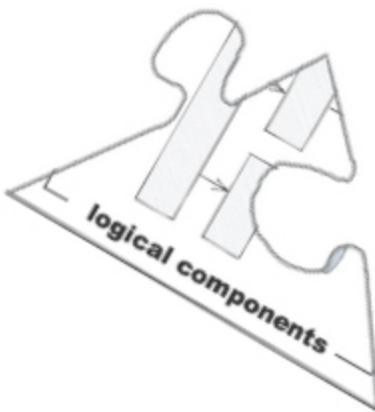
Ready to start creating an architecture? It's not as easy as it sounds, and if you don't do it correctly your software system could come crumbling to the ground, just like a poorly designed skyscraper or bridge.

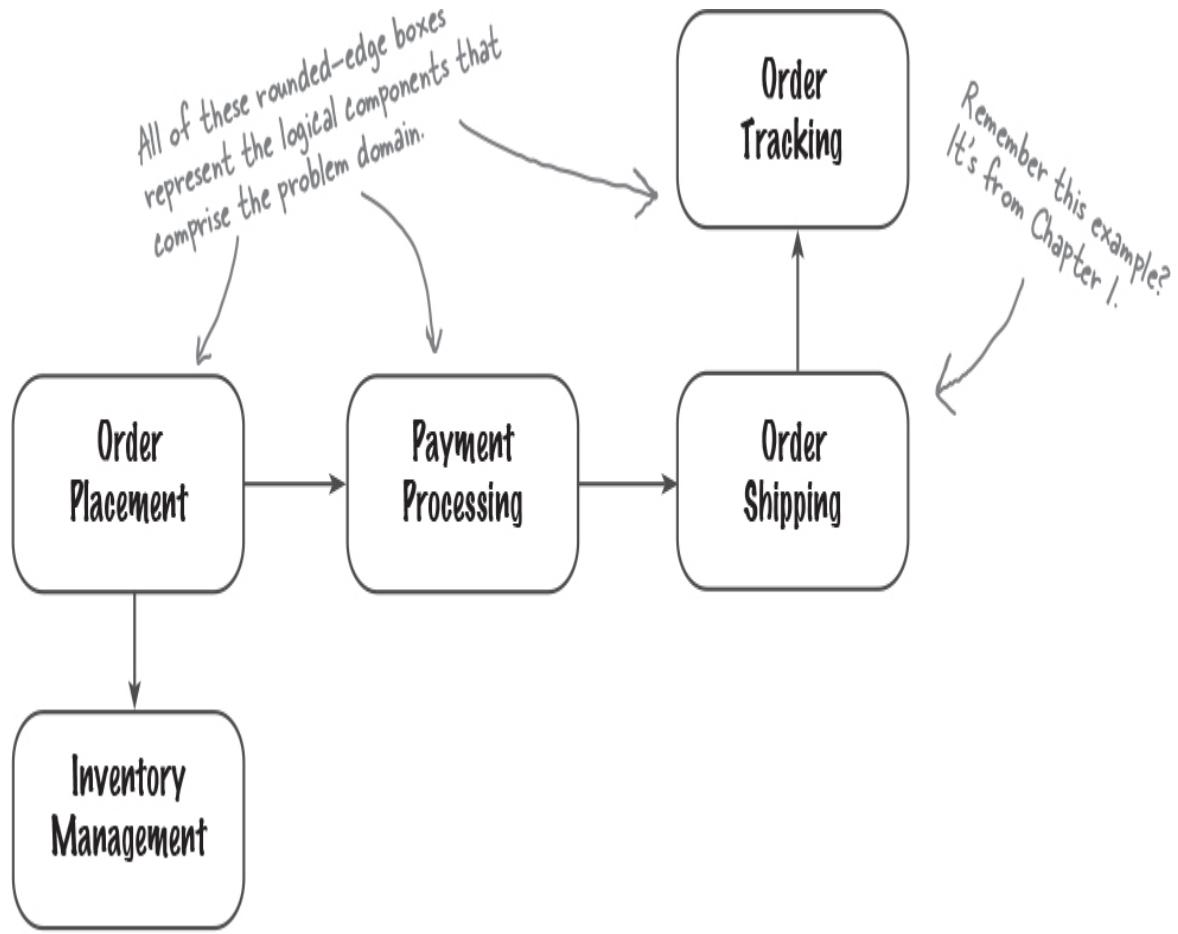
In this chapter we'll show you several approaches for identifying and creating *logical components*, the functional building blocks of a system that describe what the system does and how it all fits together. Using the techniques described in this chapter will help you to create a solid architecture—a foundation upon which you can build a successful software system.

Put on your hard hat and gloves, get your tools ready, and let's get started.

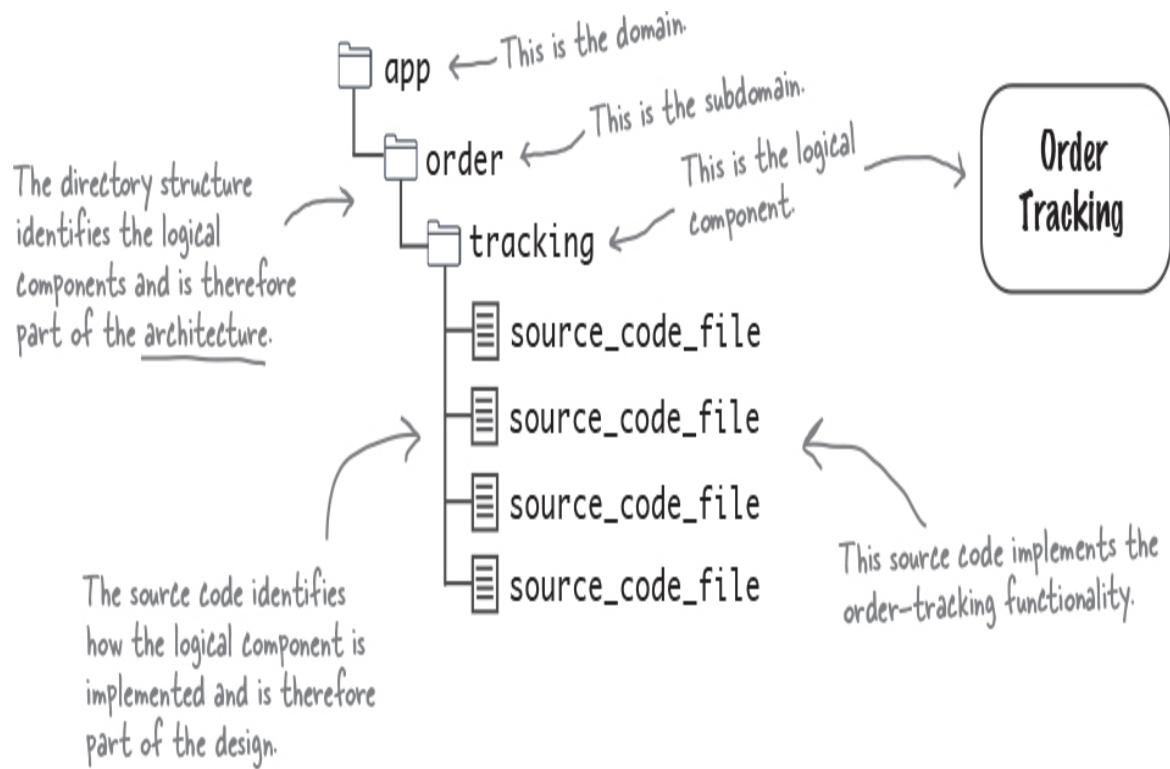
Logical components revisited

Logical components are one of the dimensions of software architecture. They are the functional building blocks of a system that make up what is known as the *problem domain*. In Chapter 1 you learned a bit about them, and in this chapter we'll dive deep into what logical components are and how to create them.





Remember that logical components are represented through the directory structure of your source code repository. For example, source code located in the `app/order/tracking` directory would be contained within a logical component named Order Tracking.

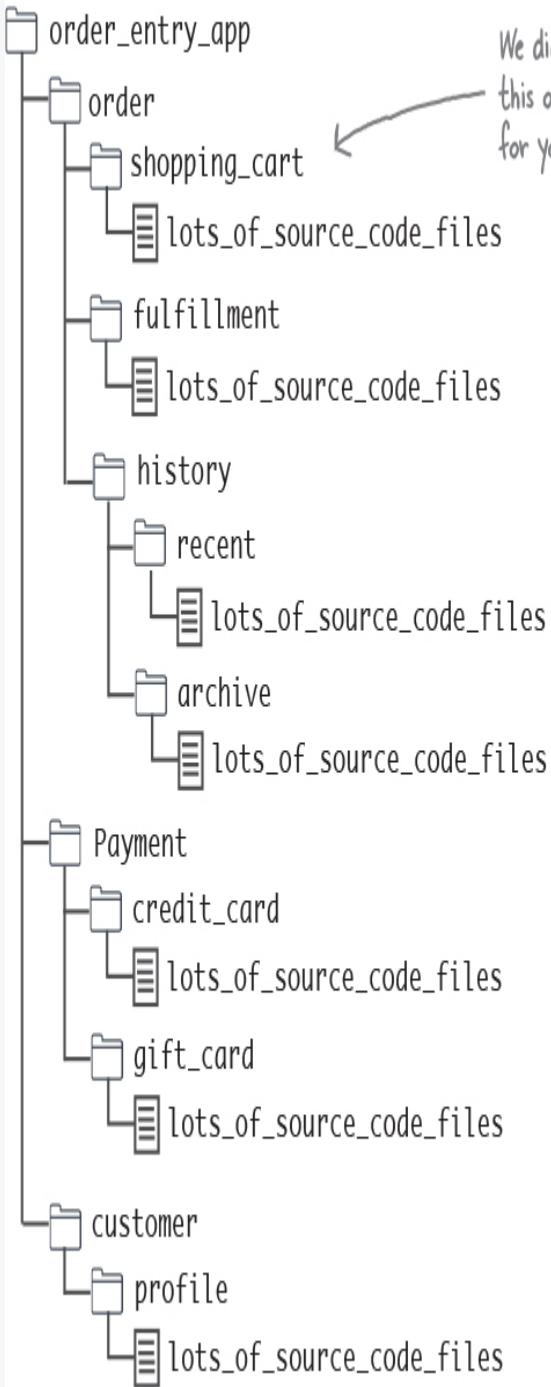


EXERCISE



Example 4-1. Name that component

It's your first week on the job as the new architect, and you've been assigned to an existing project to build a trouble-ticket system. You want to understand the logical components of the architecture, but your team doesn't know anything about logical components—they just started coding. To identify the logical architecture, you have to look at the existing directory structure. How many individual logical components can you identify from the codebase below?



Shopping Cart

Draw your components in this space over here.

Adventurous Auctions goes online



Want to go on a safari in Tanzania? Observe wildlife in the Galapagos Islands? Hike to the base camp of Mount Everest? Adventurous Auctions is here to help!

You've probably seen our ads or attended some of our live auctions around the country. These kinds of adventures are hard to come by and can take years to reserve, so our company auctions them off at a significant cost savings.

We want more people around the world to be able to access these great trips, so we're taking our adventurous auctions online (in addition to our in-person auctions).

That's where you come in: we need a new system to support the online auctions of our adventurous trips.

Here's what the new system needs to do:



- Scale up to meet demand, so hundreds or even thousands of people can participate in each auction.
- Include both in-person and online bids in every auction.
- Allow Kate to register with Adventurous Auctions and provide us with her credit card so she can pay if she wins a trip.
- Allow Kate to view a live video stream of the in-person auction, as well as all bids placed so far, both in person and online.
- Allow Kate to bid on a trip she likes, just like the people in the room.

- Determine which online bidder bids the asking price first (this is called “winning the bid”). If an online bidder bids at the same time a live bidder bids, the auctioneer then determines who bid first.
- When the auctioneer announces the winner, the system charges the winner’s credit card, notifies the winner, then moves on to the next trip in the auction.

NOTE

Pay attention, because we’re going to show you how to create a logical architecture for this system.

Logical versus physical architecture

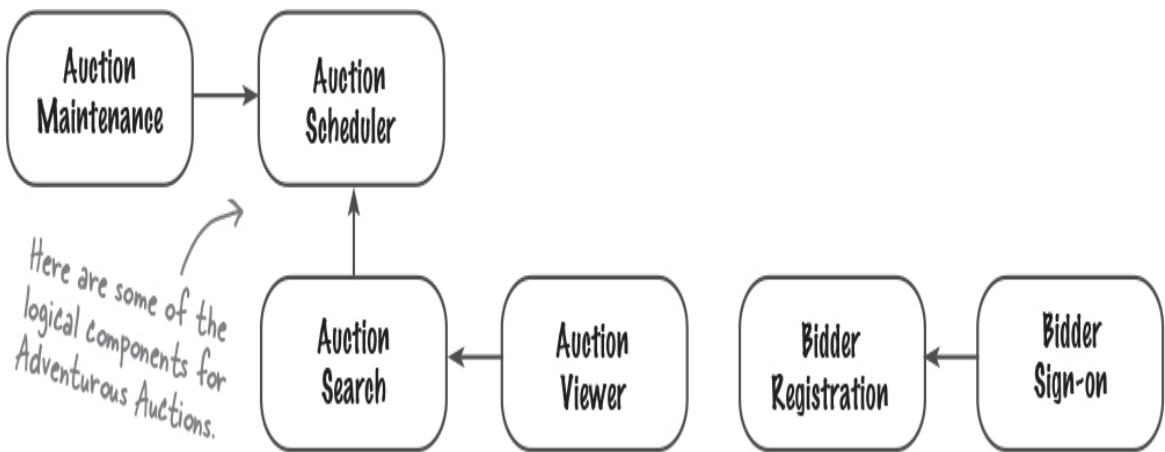
A **logical architecture** shows all of the logical building blocks of a system and how they interact with each other (known as **coupling**). A **physical architecture**, on the other hand, shows things like the architecture style, services, protocols, databases, user interfaces, API gateways and so on.

NOTE

We’re going to be talking a lot about component coupling later in this chapter.

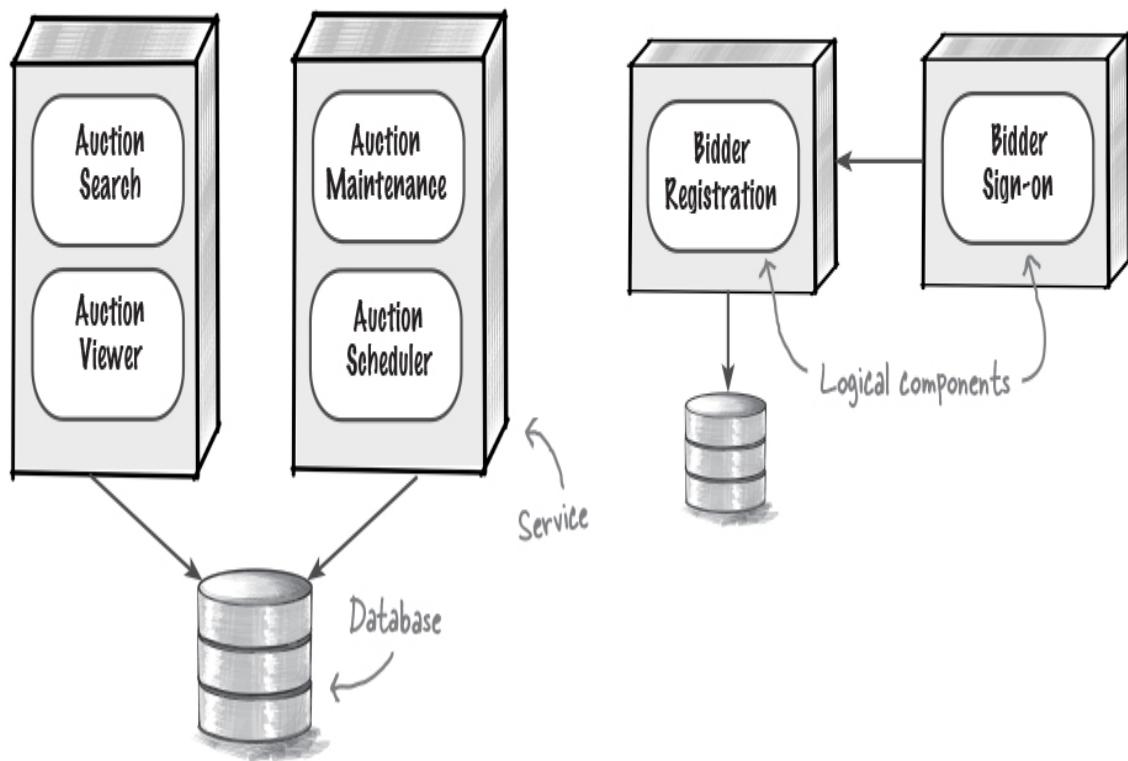
The logical architecture of a system is independent of the physical architecture—meaning the logical architecture doesn’t care about databases, services, protocols, and so on. Let’s look at an example of what we mean by a logical architecture.

As you remember, Adventurous Auctions needs to create and schedule online auctions and allow bidders to register, search for auctions, and view the trips that are up for bid. Here are some of the components—the functional building blocks—that will help make that happen.



See how the logical architecture doesn't include things like services, databases, or user interfaces? The logical architecture is a different view of the system. To see what we mean, compare the physical architecture below to the logical architecture above. Notice how the physical architecture associates services with components from the logical architecture, and also shows the services and databases for the system.

This physical architecture shows some of the services and databases.



WHO DOES WHAT?

We had our logical and physical architecture responsibilities all figured out, but somehow they got all mixed up. Can you help us figure out who does what? Be careful—some responsibilities may not have a match (they aren't part of a logical or physical architecture).

Shows which programming language is used for each component

Logical architecture Maps components to services

Shows the logical components within the system and how they communicate with each other

Shows how many databases there are in the system and which services access them

Shows communication between services and the protocol they use (like REST)

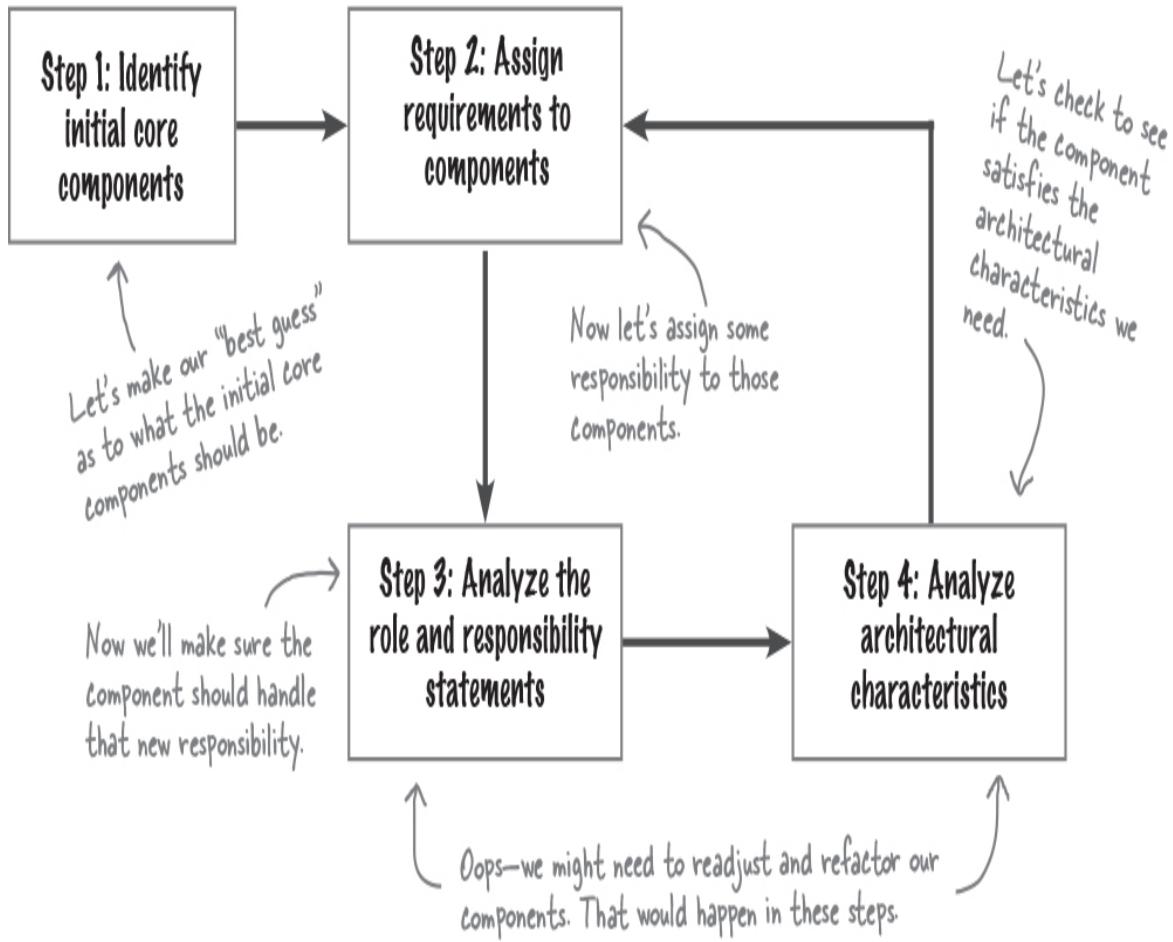
Physical architecture Shows the source code files used to implement a component

Shows the components and their interactions within the user interface

Shows the API gateways and load balancers used in the system

Creating a logical architecture

Identifying logical components isn't as easy as it sounds. To help, we've created this flowchart. Don't worry—we'll be covering all of these steps in detail in the following pages.

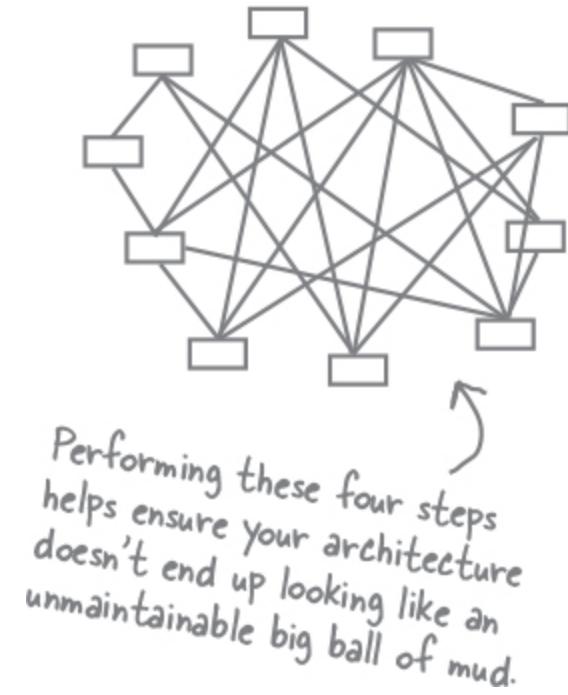


This flow continues as long as the system is alive.



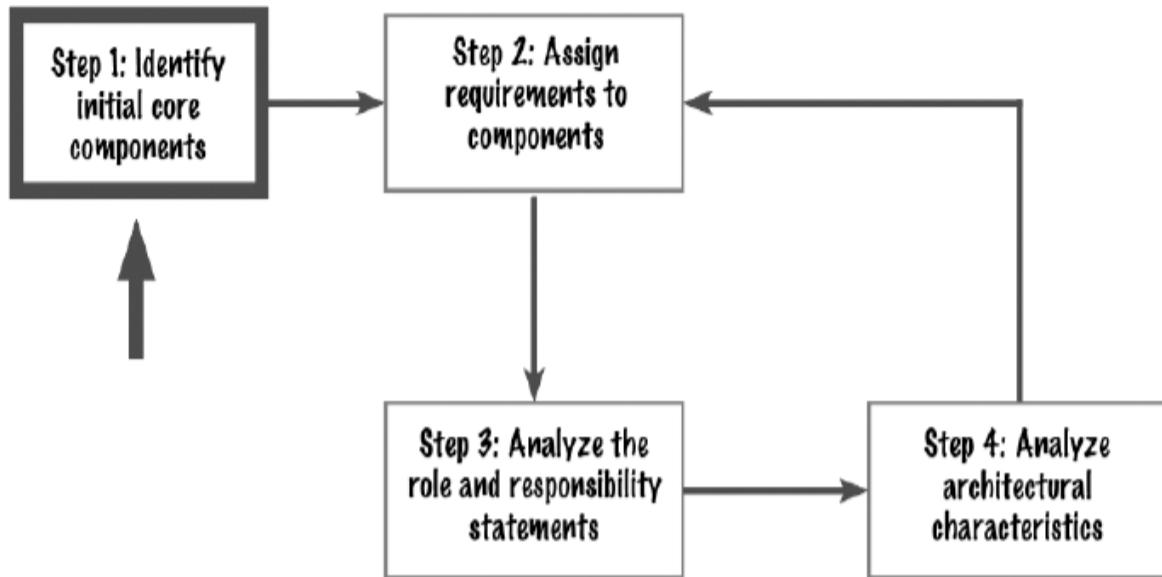
This flowchart shows a series of steps to begin greenfield applications (new systems created from scratch) and perform ongoing maintenance on existing ones. Ever wonder why it's so common for a well-designed system to end up as an unmaintainable mess in no time? It's because teams don't pay enough attention to the logical architecture of their systems.

Anytime you make a change or add a new feature to the system, you should always go through each of these steps to ensure that the logical components are the right size and are doing what they are meant to do.

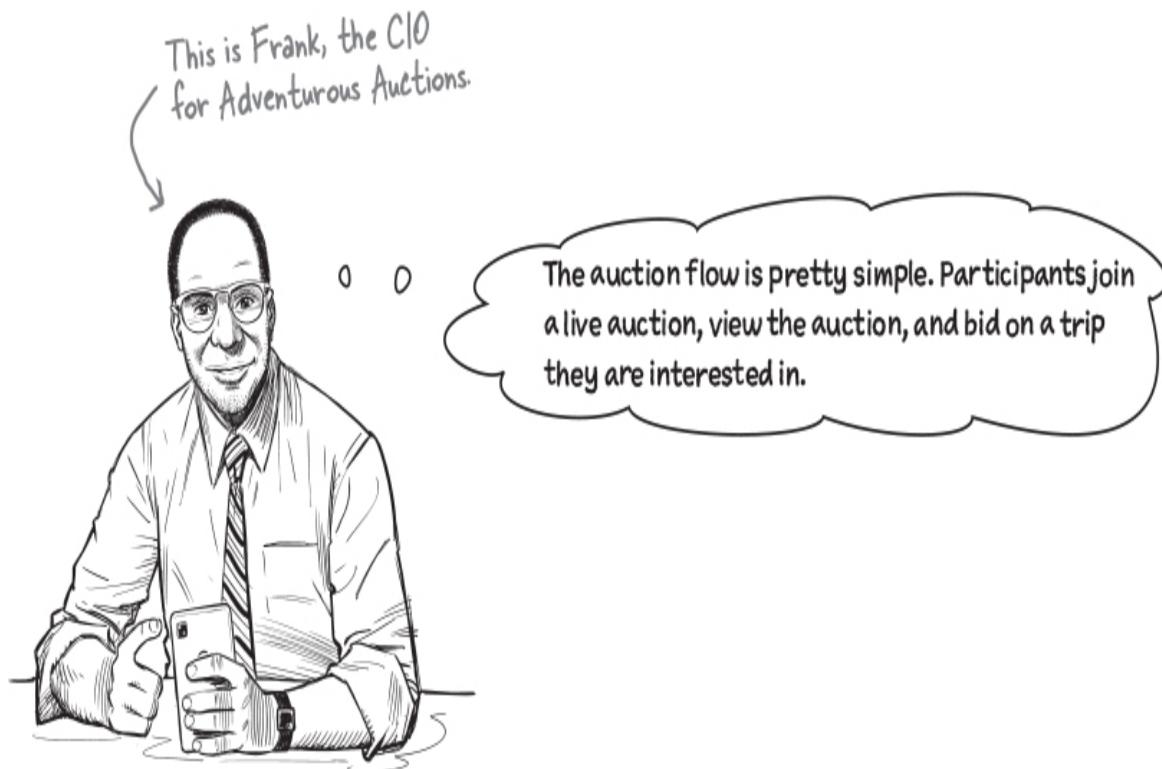


Step 1: Identifying initial core components

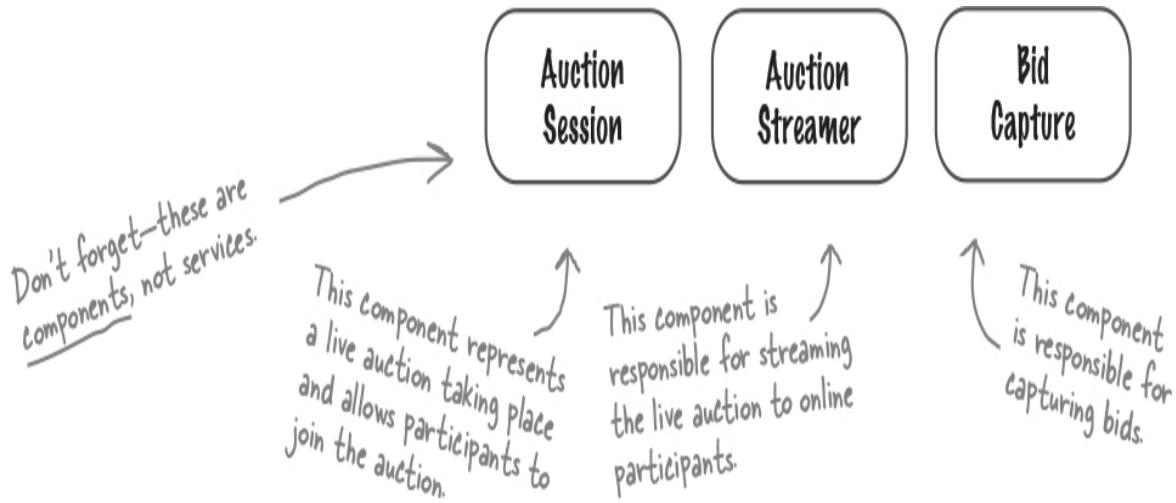
The first step in creating a logical architecture is identifying the initial core components. Many times this is purely a guessing game, and you'll likely refactor the components you initially identify into others. So don't spend a lot of time worrying about how big or small your components are—we'll get to that. First, let's show you what we mean by a "guessing game."



This roadmap shows you
which step you're on.



Given this simple description, you can start out by creating three logical components, one for each of the three major things the system does.



These components aren't really doing anything yet. You see, we've identified the initial components, but we haven't assigned them any responsibility yet. You could think of them as *empty jars*. They represent our initial *best guess* based on a major action that takes place in the system. That's why we call them *initial* core components.





Creating a logical architecture is all a “guessing game”? That’s the most ridiculous thing I’ve ever heard. Isn’t there some way to take the guesswork out of identifying logical components?

O
O

Yes, there is! In fact, you can use several approaches to remove some of the guesswork.

You don't know a lot of details about the system and requirements yet, so the components you initially identify are likely to change as you learn more. That's why identifying components is still a "guessing game"—and that's perfectly okay!

We'll show you two common approaches for identifying initial core components: the ***workflow approach*** and the ***actor/action approach***.

There are other approaches that seem like good ideas initially, but can lead you down a very bad path. We'll discuss those after we show you all the good stuff.

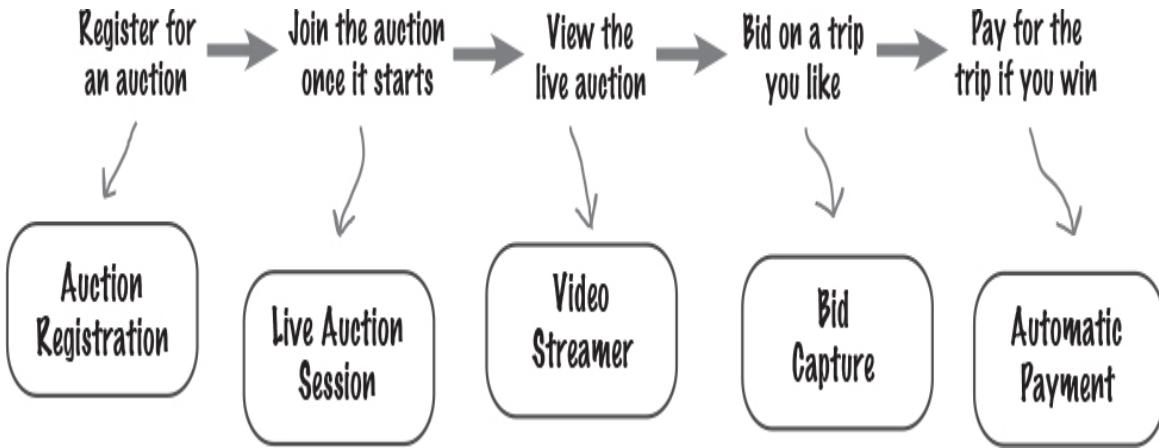
Workflow approach

The workflow approach is an effective way to identify an initial set of core components by thinking about the major workflows of the system—in other words, the journey a user might take through the system. Don't worry about capturing every step; start out with the major processing steps, and then work your way down to more details.

NOTE

You can model different workflows to create even more initial components.

Let's use the workflow approach to identify some initial core components for the Adventurous Auctions architecture.



THERE ARE NO DUMB QUESTIONS

Q: You identified “Video Streamer” as a logical component, but what if our team decides to use a third-party library or service to stream the auction?

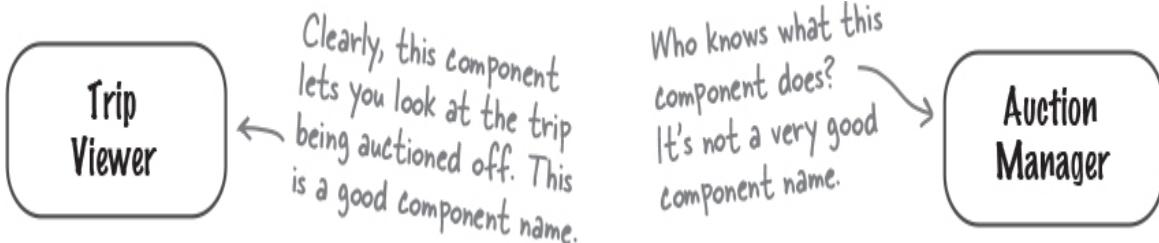
A: Great question! Even though you might not develop the functionality yourself, it’s still part of the logical architecture.

Q: Is each step in a workflow always mapped to a single logical component?

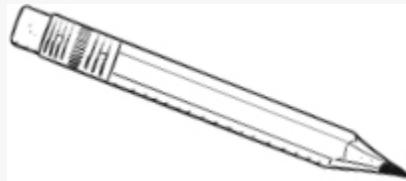
A: Not always! You might have several steps in a workflow that point to the same logical component, particularly if their functionalities are closely related.

Names matter

Pay close attention to how you name your initial core components. A good name should succinctly describe what that component does.



SHARPEN YOUR PENCIL



Your company wants a new system to assign workers to construction sites, and it's your job as the software architect to identify its initial core components. Using the workflow approach, identify as many core components as you can, matching each to its associated workflow step. Remember, a workflow step can have multiple components, and not every workflow step has to have a unique component.



Step 1. Maintain a list of all construction workers, their skills, and their locations

Step 2. Create a new construction project and specify the work site

Step 3. Create a schedule for when various construction projects start and end

Step 4. When a new project starts, assign workers based on their skills and locations

Step 5. When the project completes, free up workers so they can be reassigned

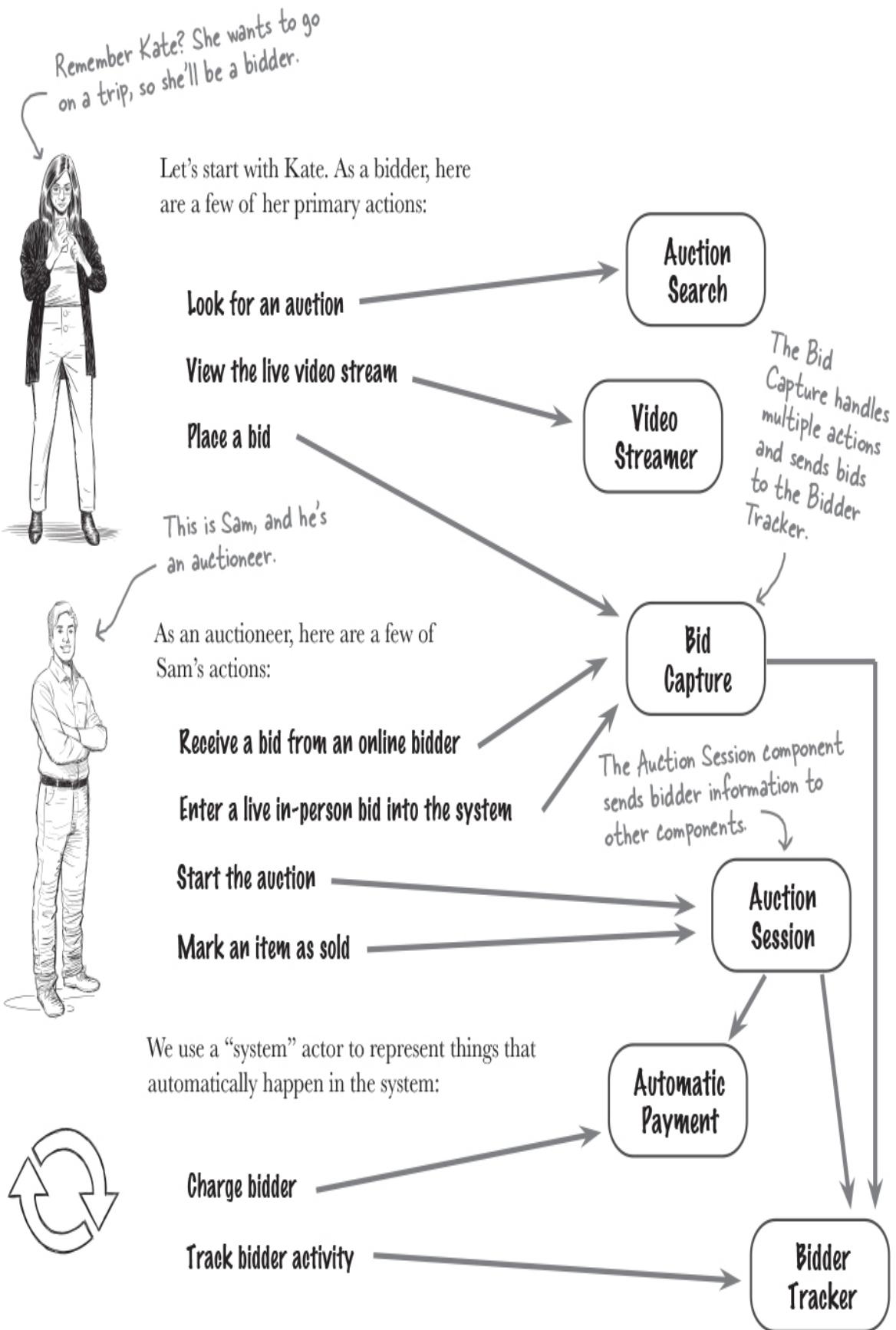
NOTE

Draw your logical components in this space here. Remember to give them good descriptive names.

Actor/Action approach

The actor/action approach is particularly helpful if you have multiple actors (users) in the system. You start by identifying the various actors. Then, you identify some of the primary actions they might take and assign each action to a new or existing component.

Returning to our Adventurous Auctions example, we'll use the actor/action approach to identify some initial core components.



EXERCISE



Peggy's bakery is ready to expand operations, and she would like a new system that lets customers view, order, and pay for bakery items online for pickup. Orders are sent to Jen (the bakery coordinator), who purchases ingredients and schedules orders. Peggy receives the schedule of items to bake each morning and tells the system when she is done with an order. The system then emails the customer to let them know their items are ready for pickup.

Using the actor/action approach, identify what actions each actor might take. Then draw as many logical components as you can for the new bakery system, matching the actions you identified to the components.

The entity trap

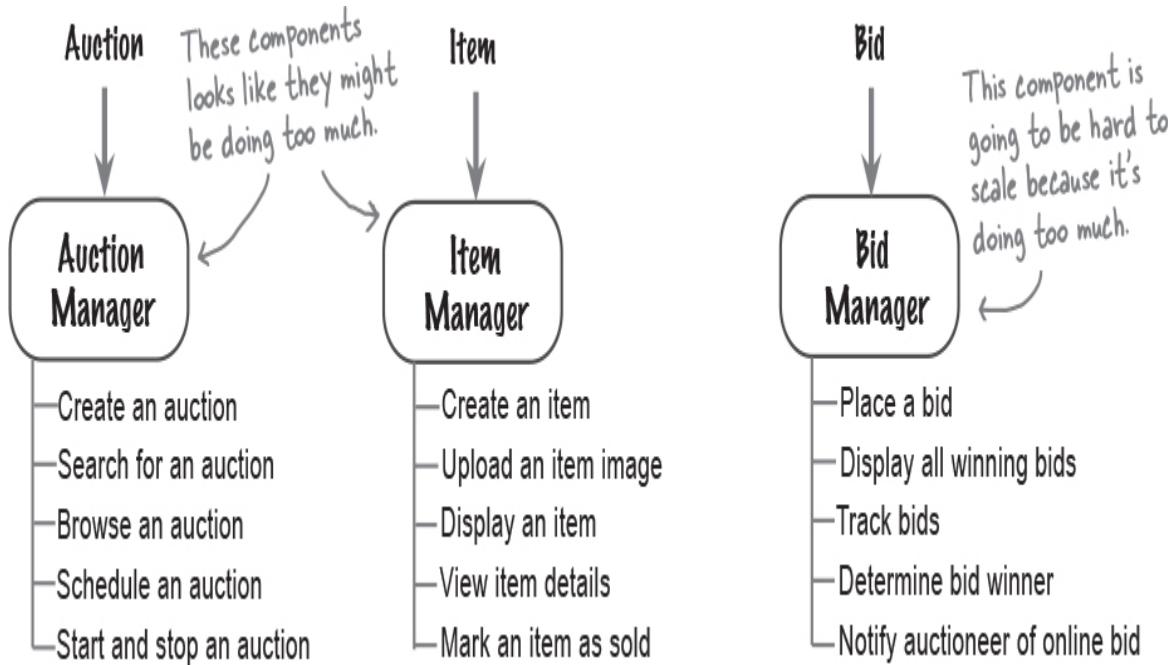
It's time to switch gears and show what *not* to do. A technique to watch out for is what we call the ***entity trap***. This technique focuses on the major

entities in the system—singular, identifiable objects—and assigns a single component to represent each entity.

NOTE

Although we're showing you an example of this technique, we don't recommend using it. Keep reading to find out why.

Some of the major entities for Adventurous Auctions include *auction*, *items*, and *bid*. Here's what components you would identify if you were using the entity trap:



Why is this approach called a *trap*?

We call this approach the ***entity trap*** because it's very easy to fall into it when identifying the initial core logical components. Here are a few of the issues that it causes:

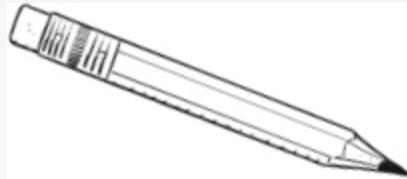
- **Component names are too vague**

What exactly does the *Auction Manager* component do? You might say it manages auctions, but that's what the entire system does. A name like this doesn't tell us enough about the component's role and responsibility.

- **Components have too many responsibilities**

All too often, components in the entity trap become convenient dumping grounds for all functionality related to that entity. As a result, they take on too much responsibility and become too big and difficult to maintain.

SHARPEN YOUR PENCIL



What other words besides **manager** can you list that, if they appeared in a component name, might indicate that you've fallen into the entity trap?

supervisor

NOTE

We did this one for you.

EXERCISE



Can you select the most appropriate approach to identifying initial core components in the following scenarios? In some cases, more than one approach may be appropriate.

The system has only one type of user **Workflow** **Actor/Action** **Entity Trap**

The system has well-defined entities **Workflow** **Actor/Action** **Entity Trap**

You have minimal functional requirements **Workflow** **Actor/Action** **Entity Trap**

The system has many complex user journeys **Workflow** **Actor/Action** **Entity Trap**

The system has many types of users **Workflow** **Actor/Action** **Entity Trap**

THERE ARE NO DUMB QUESTIONS

Q: The actor/action approach reminds me a lot of event storming. Are these the same thing?

A: Great observation, and we're glad you saw the similarities. Event storming is a workshop-based approach that is part of domain-driven design (DDD). With this approach, you analyze the business domain to identify domain events. While both approaches have the final goal of identifying actions performed within the system, event storming takes identifying components much deeper further than the actor/action approach does. The closest association between the two approaches is that the actor/ action approach identifies the *domain event* and *actor* elements of *r* event storming, but doesn't continue with other elements such as *command*, *aggregate*, and *view*. You can learn more about event storming at https://en.wikipedia.org/wiki/Event_storming.

Q: Can you combine the workflow approach and actor/ action approach, or do you have to choose between them?

A: You can combine them, and in most cases this is a good idea. If you start with the actor/action approach to identify actions, you can then use the workflow approach to arrange them in the order in which they are likely to occur.

Q: Are you telling me I should never use the words *manager*, *r supervisor*, and so on as part of my component names?

A: Not necessarily—there is no hard and fast rule to the entity trap. Sometimes it's hard to come up with a name for something that does a general task. Take, for example, a component that manages all of the reference data in your application—name-value pairs like country codes, store codes, color codes, and so on. This component manages the reference data within the application, so a good name would be “Reference Data Manager.” However, “Order Manager” or “Response Handler” would be too broad and don't describe what those components actually do.

If you find you are using such words a lot, stop and make sure you aren't just making components out of them.

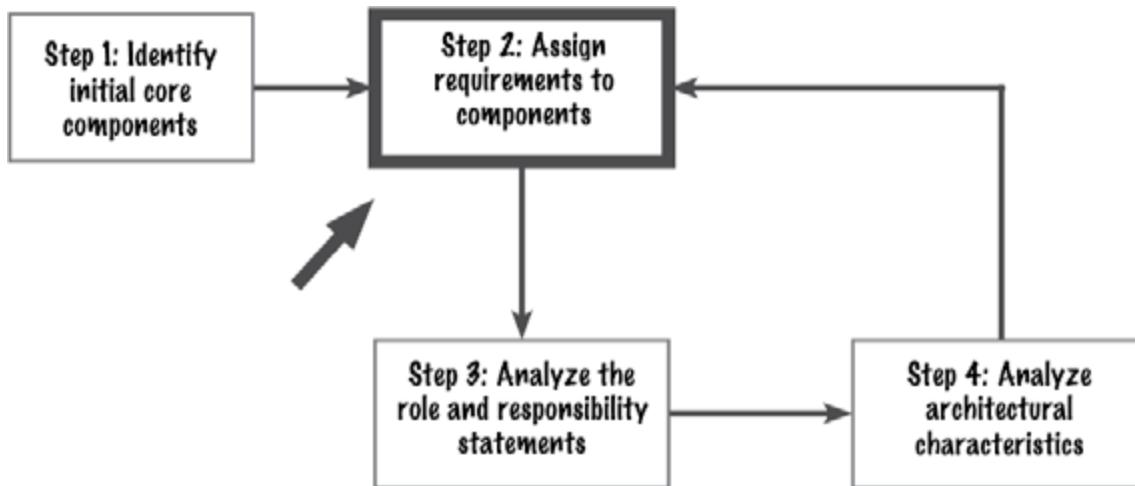
Q: When using the actor/action approach, how many actions should you identify for each actor?

A: That's a tough question. The purpose of identifying actions is to tease out likely logical components and what they might be responsible for. We usually look at the *primary* actions an actor y might take, rather than diving into too many details.

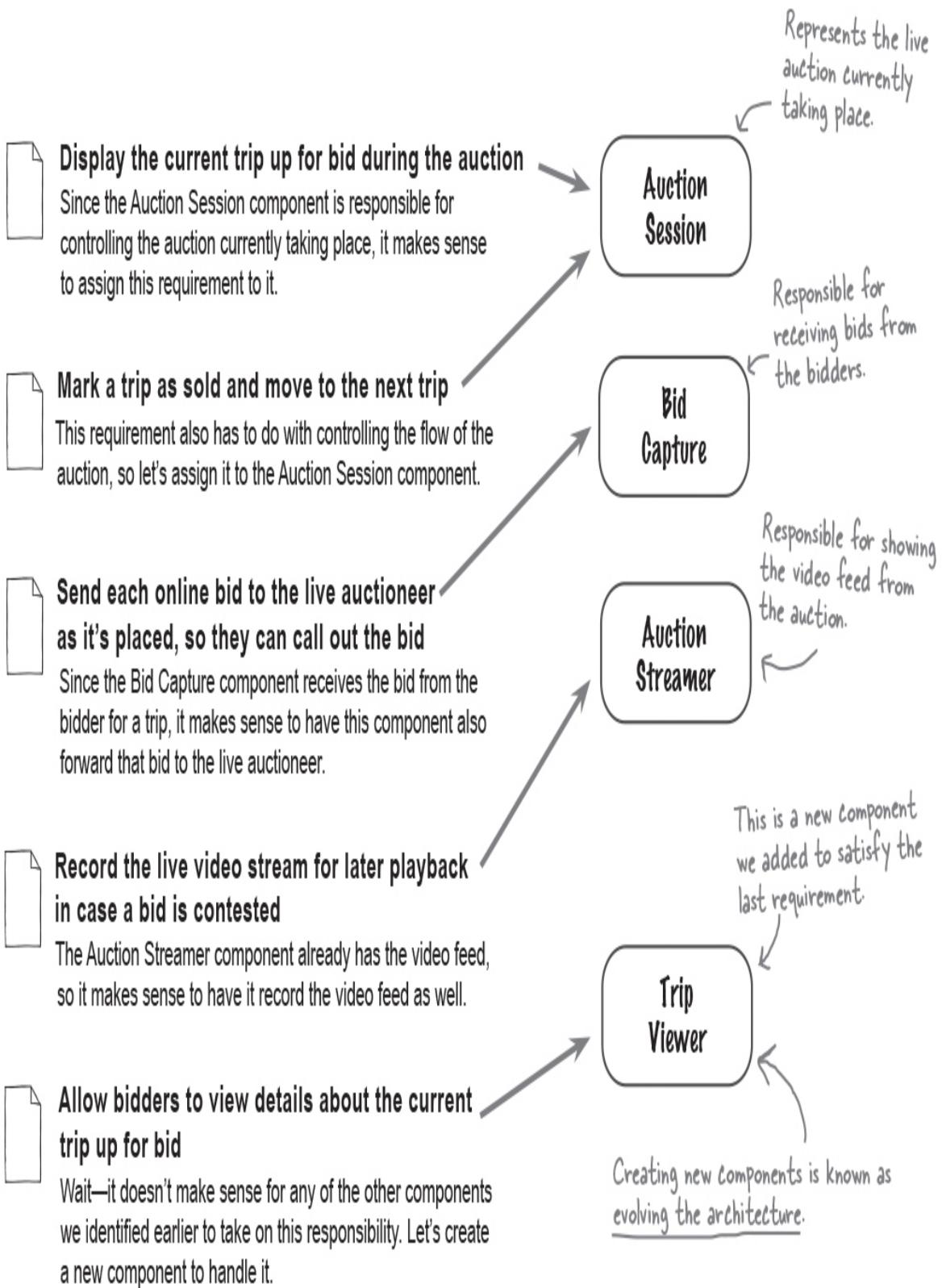
Step 2: Assigning requirements

Once you've identified some initial core components, it's time to move on to the next step: assigning requirements to those logical components.

In this step, you'll take functional user stories or requirements and figure out which component should be responsible for each one. Remember, each component is represented by a directory structure. Your source code resides in that directory, so it contains that requirement.

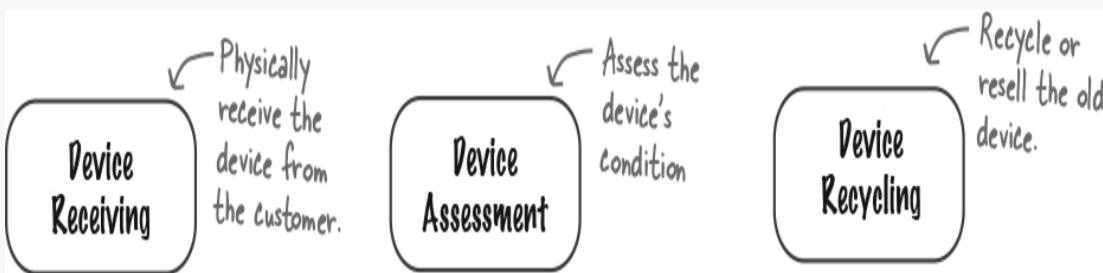


We've already defined three components based on what Frank (the CIO) said about the basic workflow of Adventurous Auctions. Now it's time to assign some responsibility to these components.



WHO DOES WHAT?

Your company, Going Green Corporation, wants a new system to support a new electronics recycling program where customers can send in their old electronic devices (like cellphones) and get money. We've already identified some of the initial core components. Your job is to figure out which component should be responsible for the functionalities listed below or if a new component is needed, and name any new components.



Locate the nearest safe disposal facility for destroying the device

- Device Receiving Device Assessment Device Recycling Other: _____
-

Capture and store customer information (name, address, etc.)

- Device Receiving Device Assessment Device Recycling Other: _____
-

Post the device on a third-party site to resell it on the secondary market

- Device Receiving Device Assessment Device Recycling Other: _____
-

Pay the customer for their recycled device

Device Receiving Device Assessment Device Recycling Other: _____

Determine if the device can be resold or if it should be destroyed

Device Receiving Device Assessment Device Recycling Other: _____

Record that the device has been received and is ready for assessment and valuation

Device Receiving Device Assessment Device Recycling Other: _____

Determine the value (if any) of the recycled device

Device Receiving Device Assessment Device Recycling Other: _____

NOTE

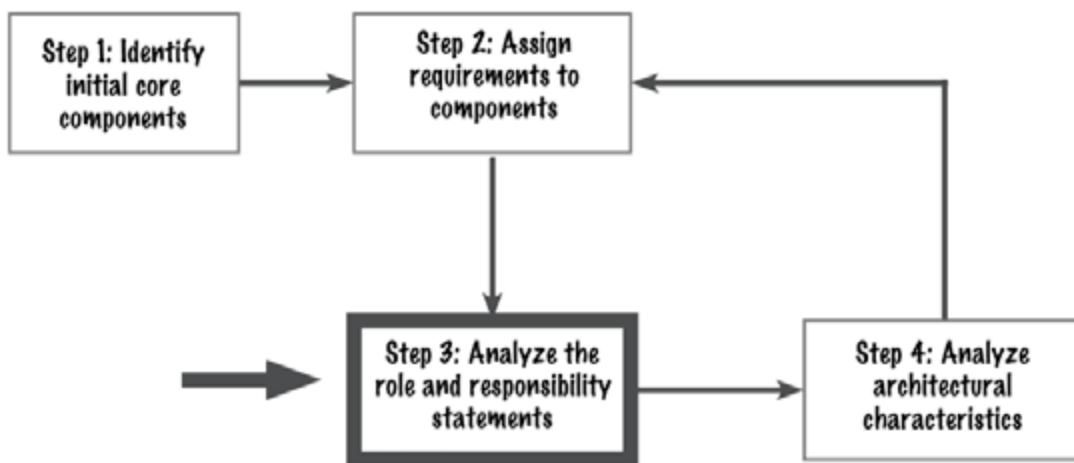
Going Green Corp. needs to make a profit, after all.

Determine the monthly profit and loss for recycled and resold devices

Device Receiving Device Assessment Device Recycling Other: _____

Step 3: Analyze Roles and Responsibilities

As you start assigning functionality (in other words, user stories or requirements) to logical components, the role and responsibility of each component will start to naturally grow. The purpose of this step is to make sure that the component to which you are assigning functionality should actually be responsible for that functionality and that it doesn't end up doing too much.



It's normal for this to happen. That's part of how an architecture evolves through iteration. Let's revisit the **Auction Session** component from our Adventurous Auctions example. Here's the list of requirements assigned to this component and its corresponding role and responsibility statement:



This component is responsible for starting and stopping the auction and keeping track of the current bidders and what their bids are. It's also responsible for managing the trips in the auction, showing which trip is currently up for bid, and moving to the next trip once the current trip has been auctioned off. It also keeps track of which bidders won which trips and notifying the bidders via e-mail that they won the trip.

NOTE

Hey, component, can you wash my car too?

It's clear that this component is taking on way too much responsibility. It's not clear exactly what this component **should** be doing. Bad component names can do this: a component without a good, self-describing name becomes a dumping ground for all sorts of unrelated functionalities. In the example above, the name **Auction Session** implies that this component manages the live auction session taking place. However, "manages" could mean anything.

GEEK NOTE



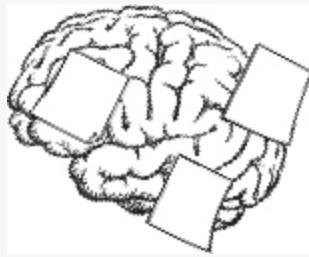
Have you ever created a class file called **Utility**? What did it do? Chances are it contained a bunch of unrelated functions that you had a hard time placing. The same is true with logical components within software architecture. Try to avoid components that contain lots of unrelated functions.

Sticking to cohesion

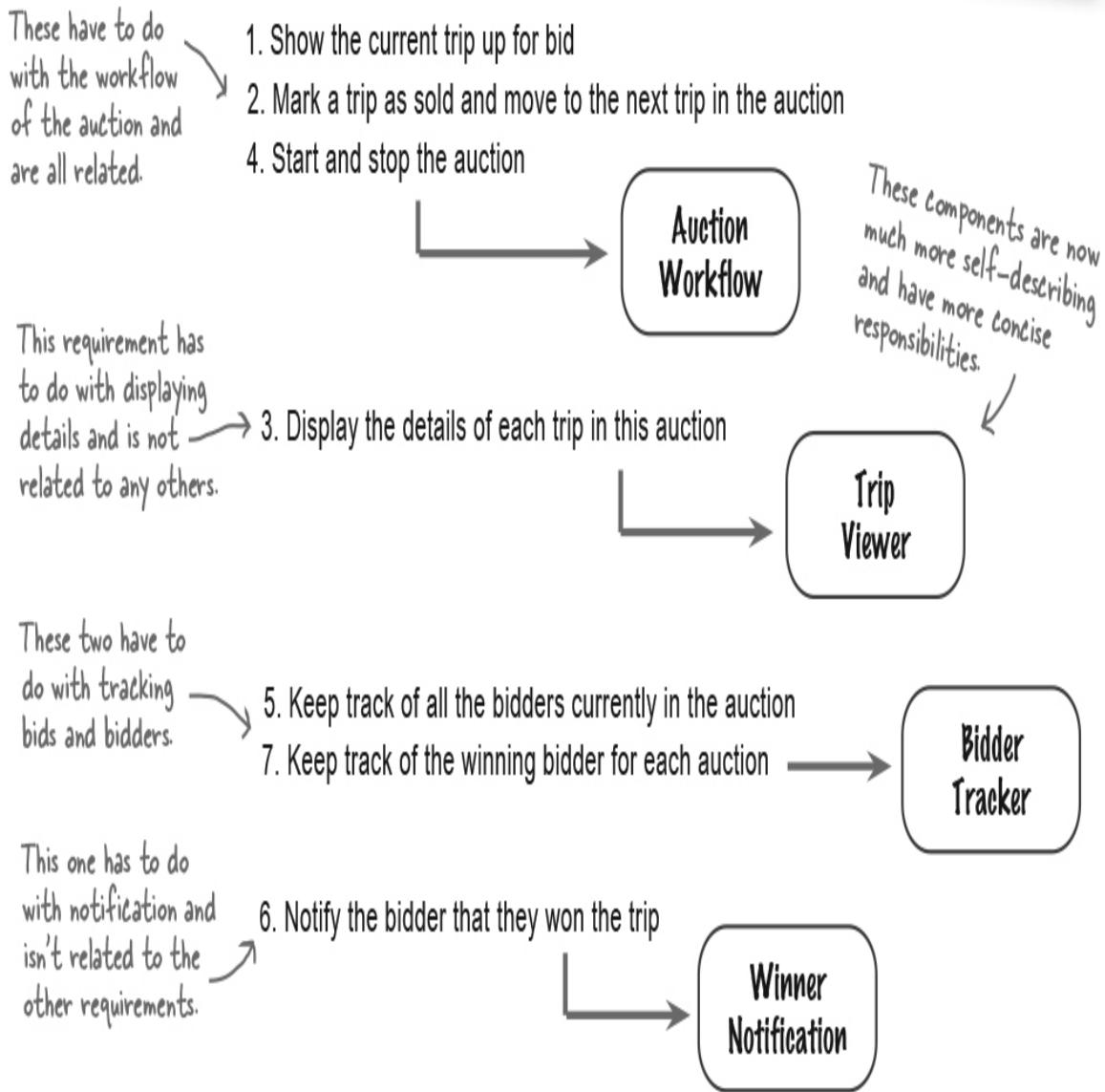
When you analyze a component's role and responsibility statement or set of operations, check to see if the functionality is closely related. This is known as *cohesion*: the degree and manner to which the operations of a component are interrelated. Cohesion is a useful tool for making sure a component has the right responsibility.

Let's see if we can apply cohesion to fix the Auction Session component problem. First, we'll group the requirements that we assigned on the prior page so that they are more closely related.

MAKE IT STICK



Component functions should all be related, But if they're not, don't get frustrated Just start to break the component apart, And you'll be considered very smart.



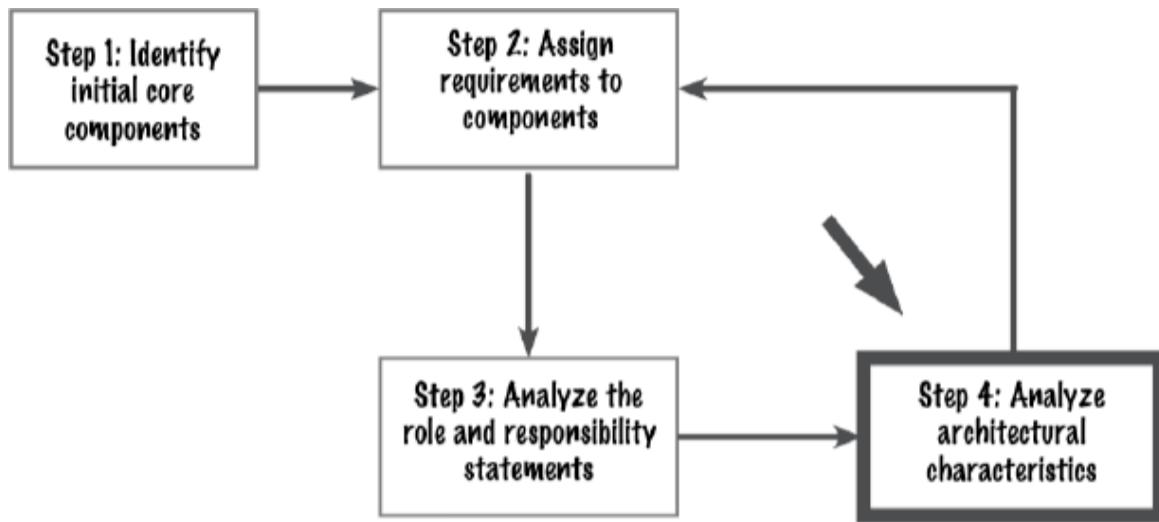
After analyzing the role and responsibility model and applying cohesion, we break up the Auction Session component into four new components, each with its own succinct role and responsibility. Each component name now provides a clear definition of what that component is responsible for.

NOTE

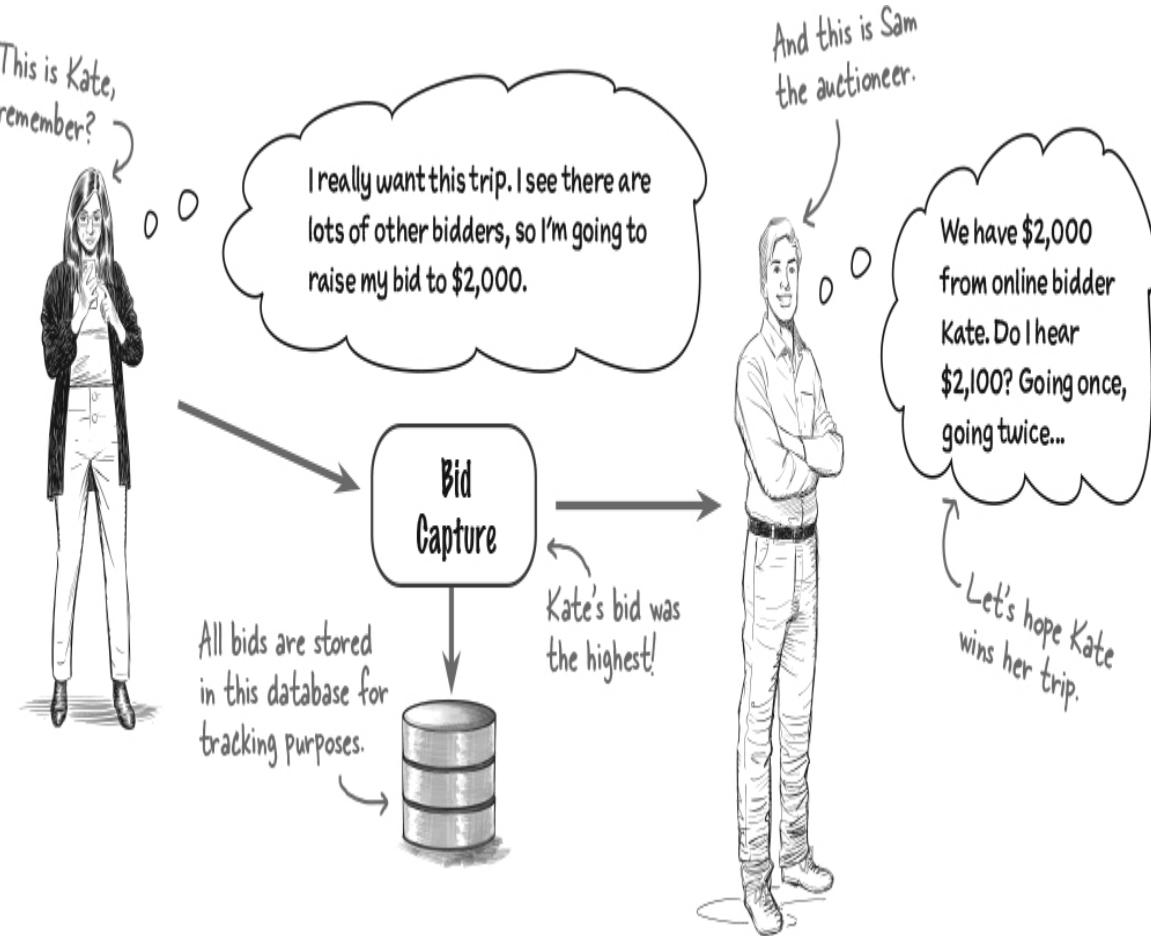
This is also known as the “Single Responsibility Principle,” or simply SRP.

Step 4: Analyze characteristics

The final step in identifying initial core components is to verify that each component aligns with the driving architectural characteristics that are critical for success. In most cases this involves breaking apart a component for better scalability, elasticity, or availability, but it could also involve putting components together if their functionalities are tightly coupled.



Let's look once again at our Adventurous Auctions example. We previously identified a Bid Capture component that is responsible for *accepting bids*, *determining which bid is the highest*, and *storing all bids in a Bid Tracker database*. Here is the overall flow for the Bid Capture component:



This architecture looks good, but just to be sure, we should analyze the critical architecture characteristics (those that are important for success) against the Bid Capture component.

We know the system has to support thousands of bidders per second—that's **scalability**. We also know the system must be up and running while auctions are taking place—that's **availability**. Finally, the system must accept a bid and get it to the auctioneer as fast as possible—that's **performance**.

NOTE

These are all important to the success of Adventurous Auctions.

Now it's your turn to analyze the Bid Capture component against these critical architecture characteristics.

BE the architect

Your job is to play architect and analyze the Bid Capture component on the prior page to see if it matches with the critical architectural characteristics we identified. Our solution is on the next page.



NOTE

Use this area to draw how you might change the Bid Capture component based on the critical architecture characteristics above.

NOTE

Here's a hint to help you out: should the bid capture functionality be contained in one component or spread across multiple components?

These are the critical architectural characteristics for Adventurous Auctions

Scalability: The system has to support thousands of bidders per second

Availability: The system must be up and running while the auctions are taking place

Performance: The system must accept a bid and get it to the auctioneer as fast as possible.

Our solution...

Let's review the current responsibilities of the Bid Capture component:

1. ***Accept bids from online bidders and from the auctioneer for live bidders***
2. ***Determine which online bid is the highest***
3. ***Write all bids to a Bid Tracking database for tracking purposes***

It makes sense for the Bid Capture component to write the bids to the database, since it has them. But database connections and throughput are limited, so having Bid Capture do this significantly impacts *scalability*. It also impacts *performance* by adding wait time for writing the data to the database, as well as *availability* if the database were ever to go down.

NOTE

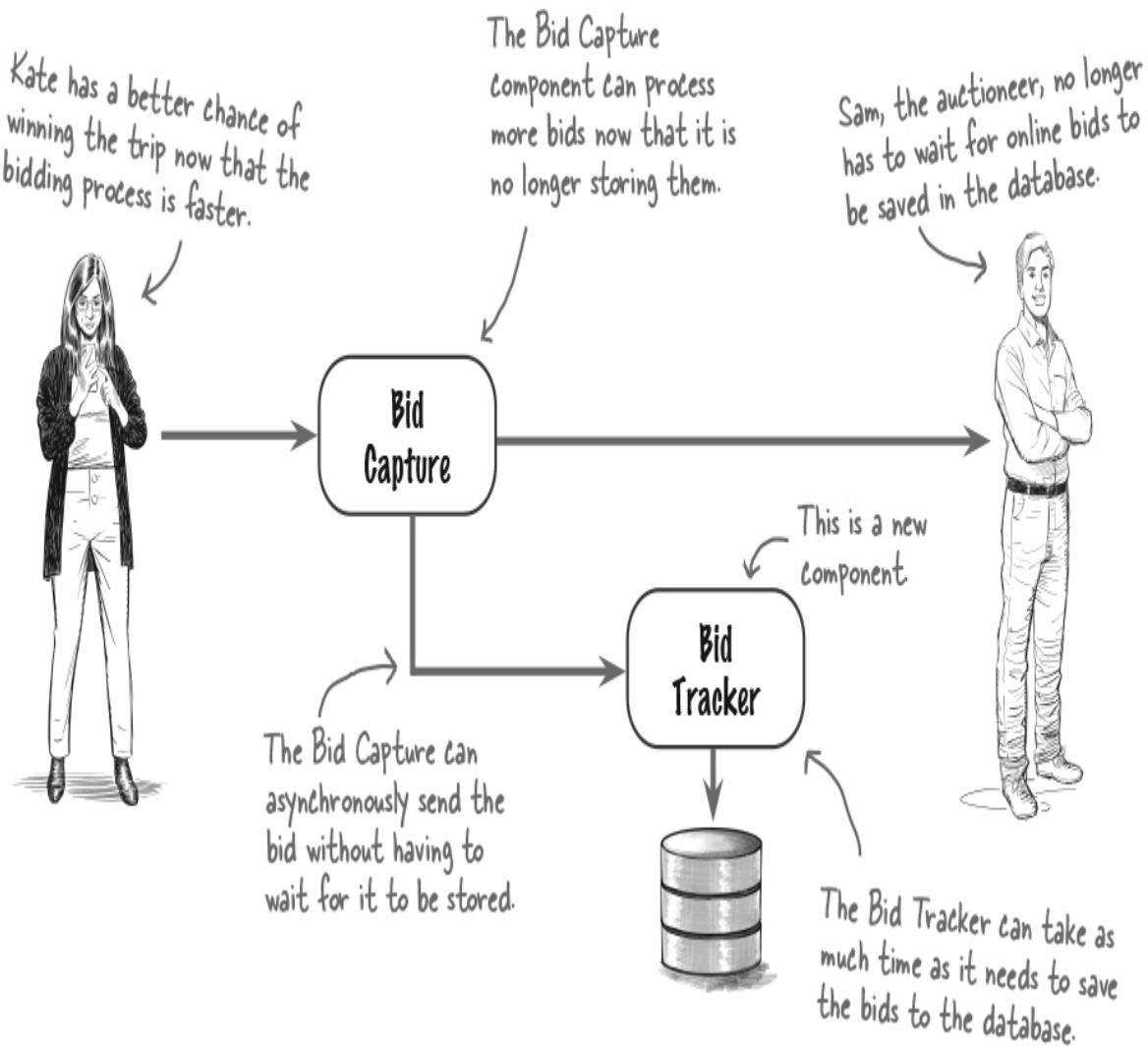
This is what we mean by analyzing characteristics.

If we assign the last requirement to a new component called ***Bid Tracker***, we can significantly increase the scalability, performance, and availability of the Bid Capture component. That lets the system process more bids faster and get the highest bid to the auctioneer as quickly as possible. The Bid

Capture component can send the bids to the Bid Tracker and won't have to wait for the bid to be written to the database.

NOTE

There are times when you will break apart or combine components in this step, based on the architectural characteristics needed.



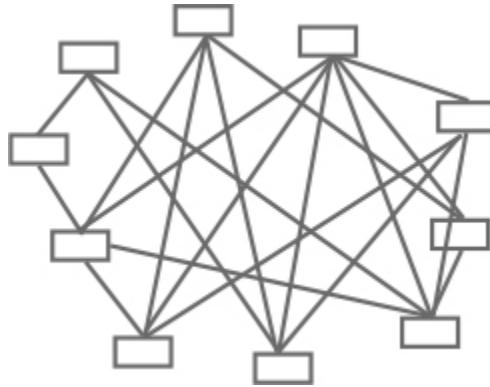
Component coupling



Yes, and this is the right time to do it.

As you identify the initial core components, it's important to determine how they interact. This is known as ***component coupling***: the degree and manner to which components know about and rely on each other. The more

the components interact, the more tightly coupled the system is and the harder it will be to maintain.



Remember this diagram from several pages ago? It's called a "big ball of mud" because there are so many component interactions and dependencies that the diagram starts to look like a ball of mud (or maybe like a bowl of spaghetti).

That's why it's so important to pay attention to how components interact and what dependencies exist between them.

You need to be concerned about two types of coupling when creating logical components: ***afferent coupling*** and ***efferent coupling***. Don't be concerned if you've never heard these formal terms before—we're going to explain them in the following pages.

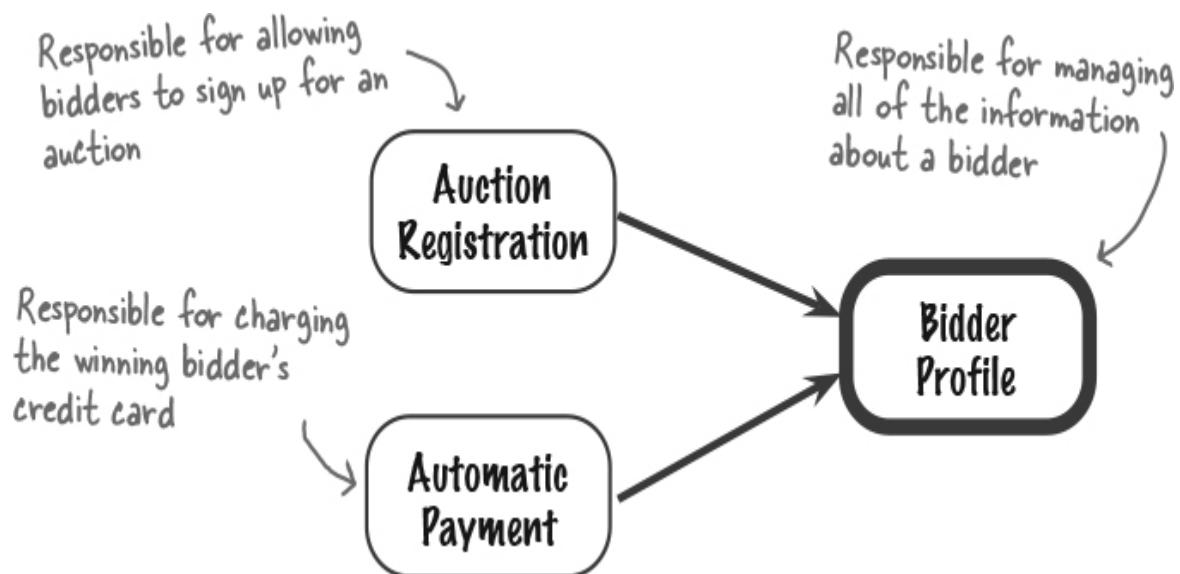
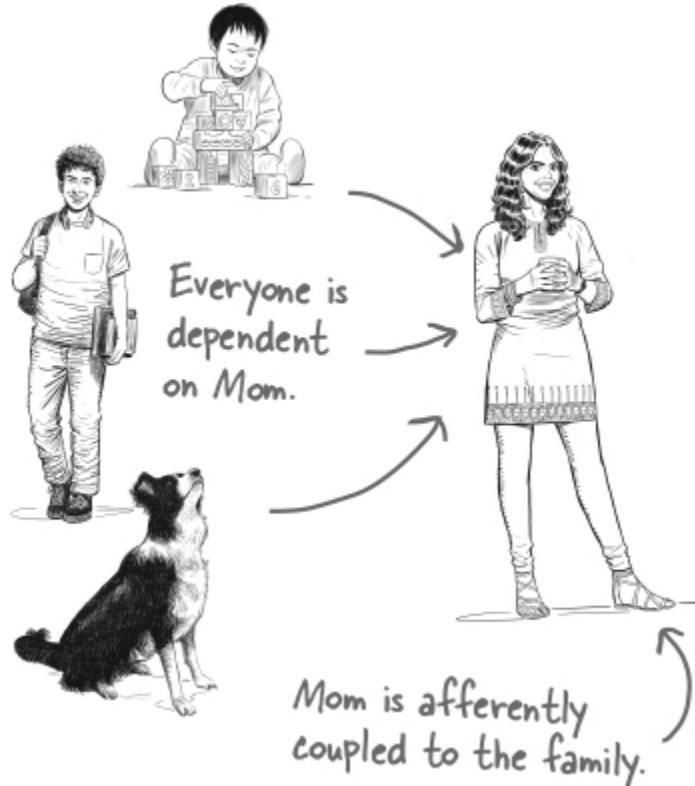
Afferent coupling

Children depend on their parents for a lot of things—like making sure they have plenty of food to eat and a safe place to live, driving them to soccer practice, or even giving them an allowance so they can buy candy or a really cool comic book. As it turns out, parents are afferently coupled to children, their siblings, and even the family dog, because all of them are *dependent on parents* for something.

Afferent coupling is the degree and manner to which other g components are dependent on some target component (in this case, Mom). It's

sometimes referred to as *fan-in*, or *incoming*, coupling. In most code analysis tools, it's simply denoted as CA.

To see how afferent coupling works, let's look at the interaction between three of the logical components within the Adventurous Auctions logical architecture:



Both the Auction Registration component and the Auto Payment component depend on the Bidder Profile component to return bidder profile information. In this scenario, the Bidder Profile component has an afferent coupling level of 2, because two components depend on it to complete their work.

If you have too much afferent coupling between your logical components, it's likely that you made them too fine-grained. Combining them can reduce the system's overall coupling.

EXERCISE



Using the Adventurous Auctions logical components above, think about the architecture characteristics associated **only** with the **Bidder Profile** component and circle all of the architecture characteristics listed below that might be affected by its afferent coupling to the Auction Registration and Automatic Payment components.

Scalability	Availability	Elasticity	Responsiveness	Fault Tolerance
		Scalability	Testability	Maintainability Deployability

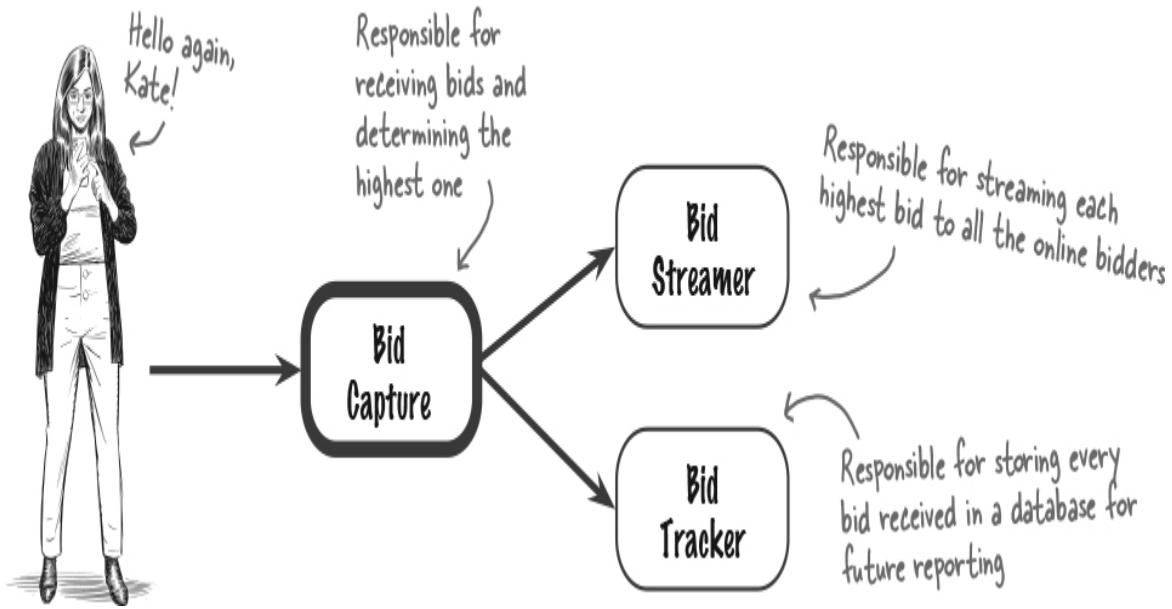
Efferent coupling



Now let's look at things from a young child's point of view. As a child, you might have been dependent not only on your parents, but also your teachers, friends, classmates, and so on. Being dependent on other people is known as **efferent coupling**.

Efferent coupling is exactly the opposite of afferent coupling, and it's measured by the number of components on which a target component depends. It's also known as *fan-out coupling* or *outgoing coupling*. In static source-code analysis tools it's usually denoted as *CE*.

So, what does efferent coupling look like with logical components? Let's take a look at Adventurous Auctions again, this time considering the process of accepting a bid from Kate for a trip.



Because the Bid Capture component depends on the Bid Streamer and Bid Tracker components to process a bid, it is *efferently coupled* to these components and has an efferent coupling level of 2 (in other words, it's dependent on two other components).

EXERCISE



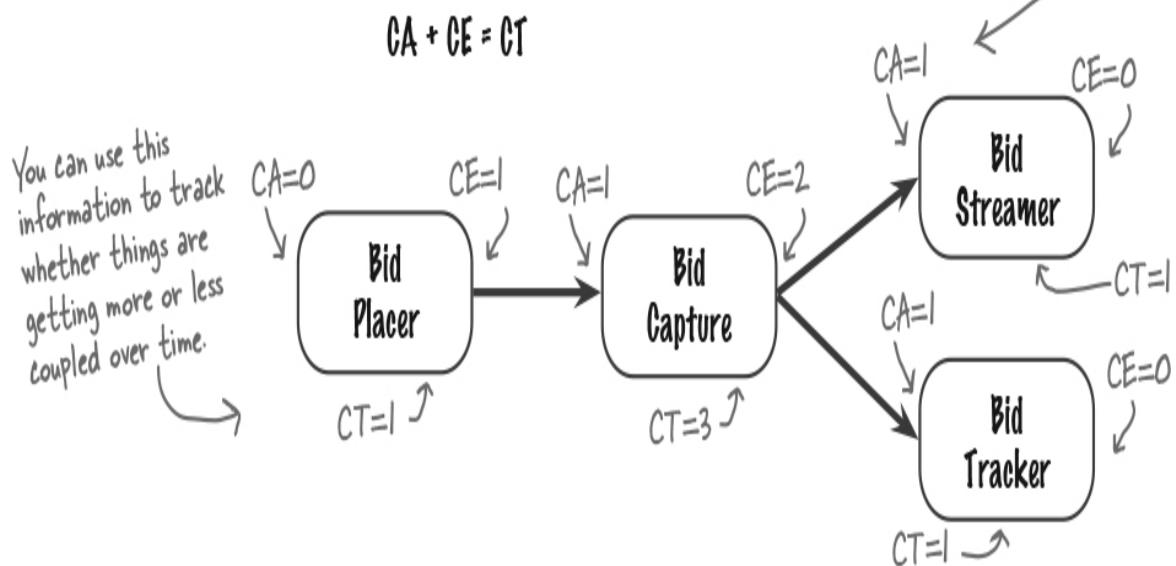
Now's your chance to see the difference between afferent and efferent coupling. Just like in the previous exercise, think about the architecture characteristics associated **only** with the **Bid Capture** component above. Now circle all of the architecture characteristics listed below that might be affected by Bid Capture's efferent coupling to the Bid Streamer and Bid Tracker components.

Scalability	Elasticity	Responsiveness	Fault Tolerance
Availability	Scalability	Maintainability	Deployability
	Testability		

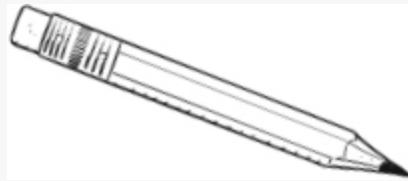
Measuring Static Coupling

You can measure a particular component's amount of coupling using three things: its total afferent coupling (CA), its total efferent coupling (CE), and its total coupling (CT), or the sum of the total afferent and efferent coupling. These measurements tell you which components have the highest and lowest coupling, as well as the entire system's overall coupling level.

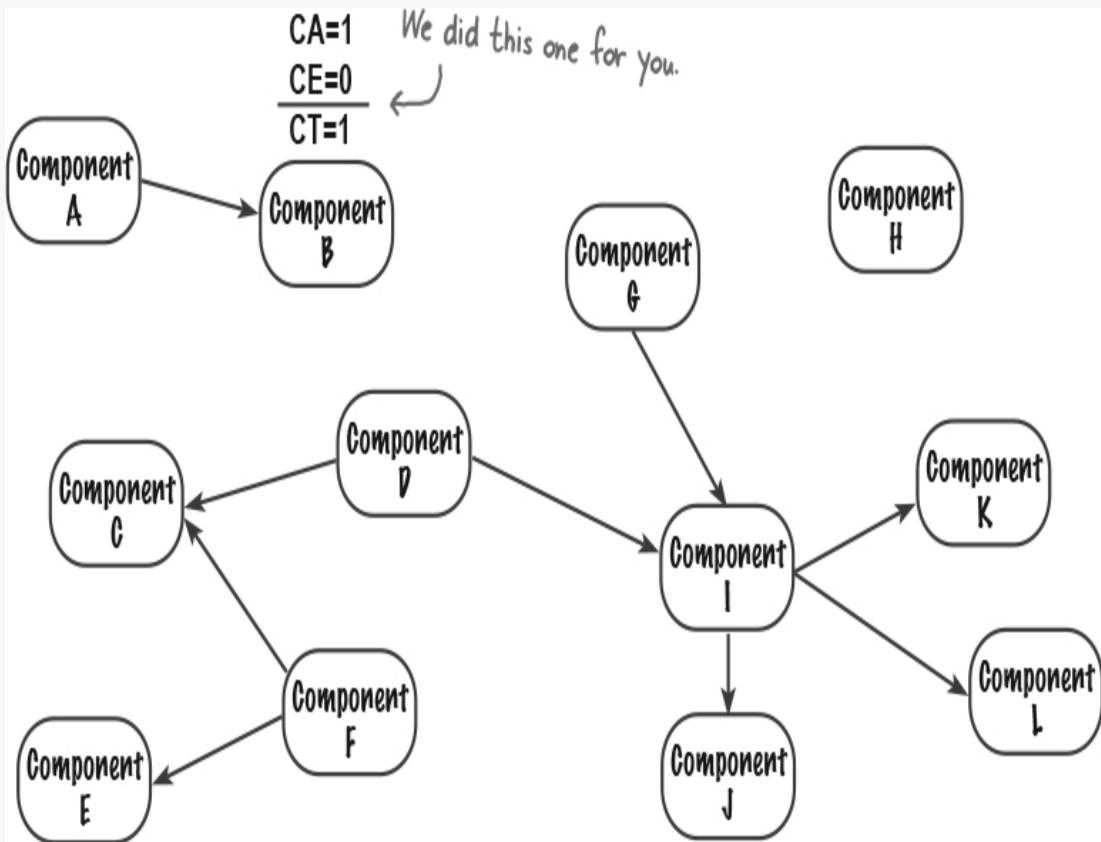
This logical architecture has a total coupling (CT) level of 6.



SHARPEN YOUR PENCIL



Given the following components below, can you identify the total afferent coupling (CA), total efferent coupling (CE), and total coupling for each component (CT)? Also, what is the total coupling level for this logical architecture? Does the total coupling for this architecture seem high or low to you?



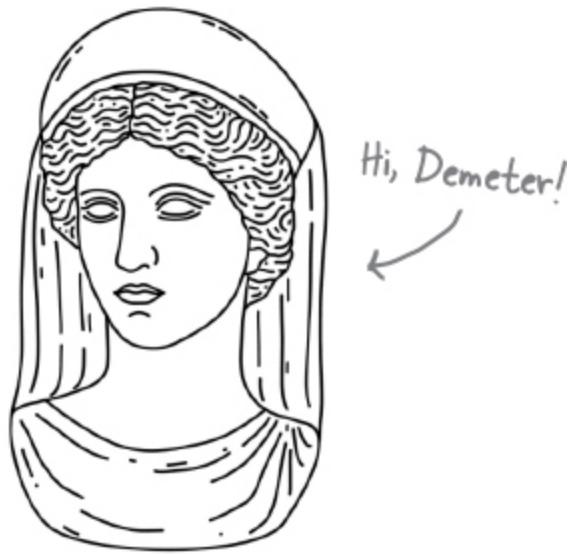
Total system coupling level: _____



Okay, now I get static coupling and how to measure it. But how do I reduce component coupling to create loosely coupled systems?

Great question. Developers are taught to strive for loosely coupled systems, but not *how* to do it. We'll show you how on the next page, by using a technique called the ***Law of Demeter***.

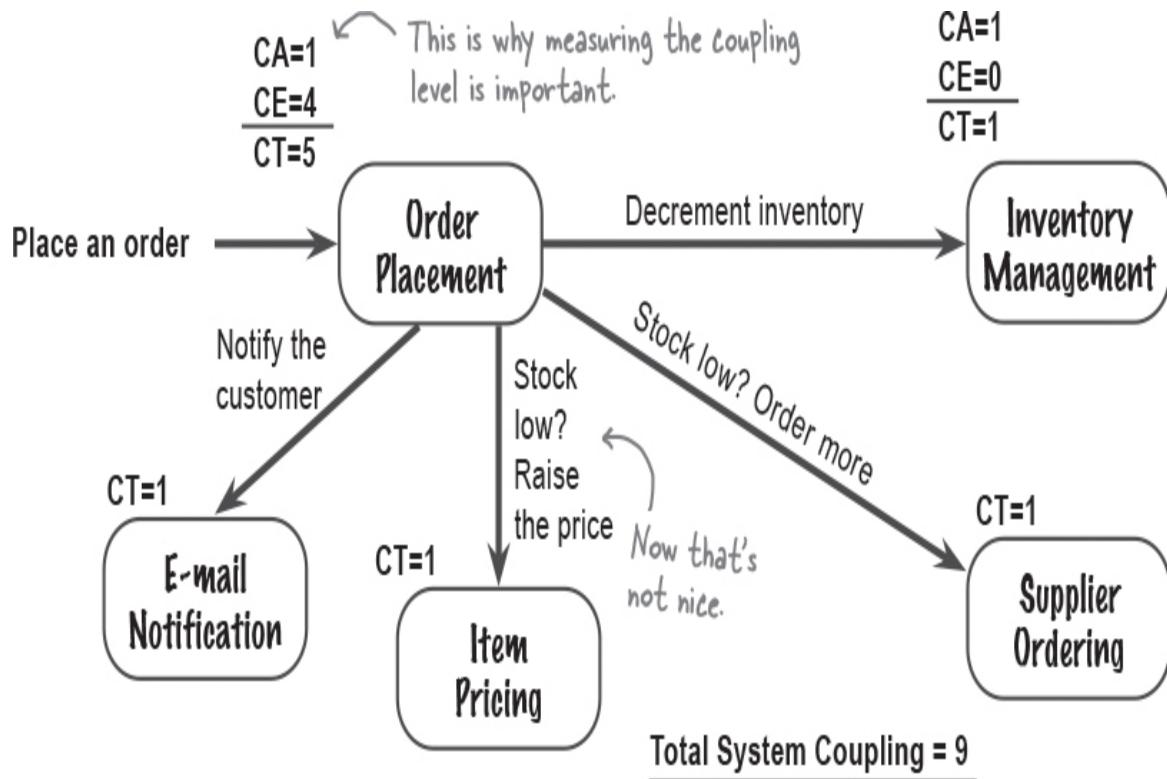
Law of Demeter



The ***Law of Demeter*** is also known as the ***Principle of Least Knowledge***. This law is named after Demeter, the Greek goddess of agriculture. She produced all grain for mortals to use, but she had no knowledge of what they did with the grain. Because of this, Demeter was loosely coupled to the mortal world.

Logical components work in the same way. The more knowledge a component has about other components and what needs to happen in the system, the more coupled it is to those components. By reducing its knowledge of other components, we reduce that component's level of coupling.

Let's see how this law works by taking a look at the logical architecture for a typical order-entry system:



Notice that each of these components performs a specific role within this system, which initially seemed like a good idea. However, this might not be the best architecture to move forward with.

BRAIN POWER



What possible issues do you see with the logical architecture above?

Law of Demeter Applied

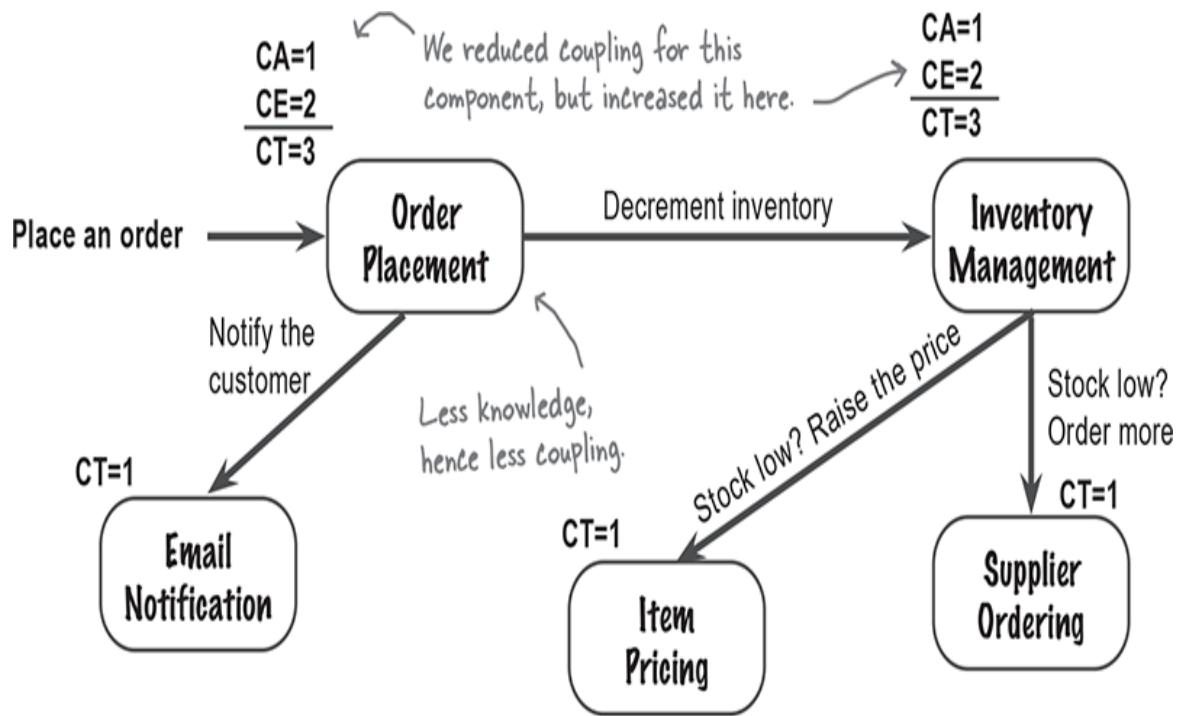
The two issues we found with the architecture on the prior page are as follows:

- 1. The Order Placement component is too tightly coupled to the rest of the system.**
- 2. The Order Placement component knows too much about what needs to happen when a customer places an order.**

Let's apply the Law of Demeter to fix these problems. Notice that while Order Placement is not *responsible* for the rest of the functionality in the other components, it is responsible for *knowing that those operations need to happen*. That's too much knowledge. Let's decouple the Order Placement by delegating that knowledge to the Inventory Management component.

NOTE

The Order Placement component knows too much.



Total System Coupling = 9

By moving the knowledge of actions to take for a “low stock” condition to Inventory Management, the Order Placement component has *less knowledge, hence less coupling*. However, did you also notice that we *increased* the knowledge for the Inventory Management component, and thus increased its coupling? This is what the Law of Demeter is all about—less knowledge, less coupling; more knowledge, more coupling.

NOTE

This is the same principle that makes assembly lines work.

GEEK NOTE



Did you notice that while we reduced the coupling of the Order Placement component, the total system coupling level remained the same at 9? That's because we didn't **remove** the knowledge from the system, we just **moved** it to another component—Inventory Management.

Too much knowledge?

The Law of Demeter teaches us that having too much knowledge of what needs to happen in the system creates tighter coupling between components. By delegating that knowledge to other components, we can create a more loosely coupled system. Now it's your turn to see if you understand the Law of Demeter.

NOTE

Knowledge is a good thing, except in loosely coupled systems. Thank you, Law of Demeter.

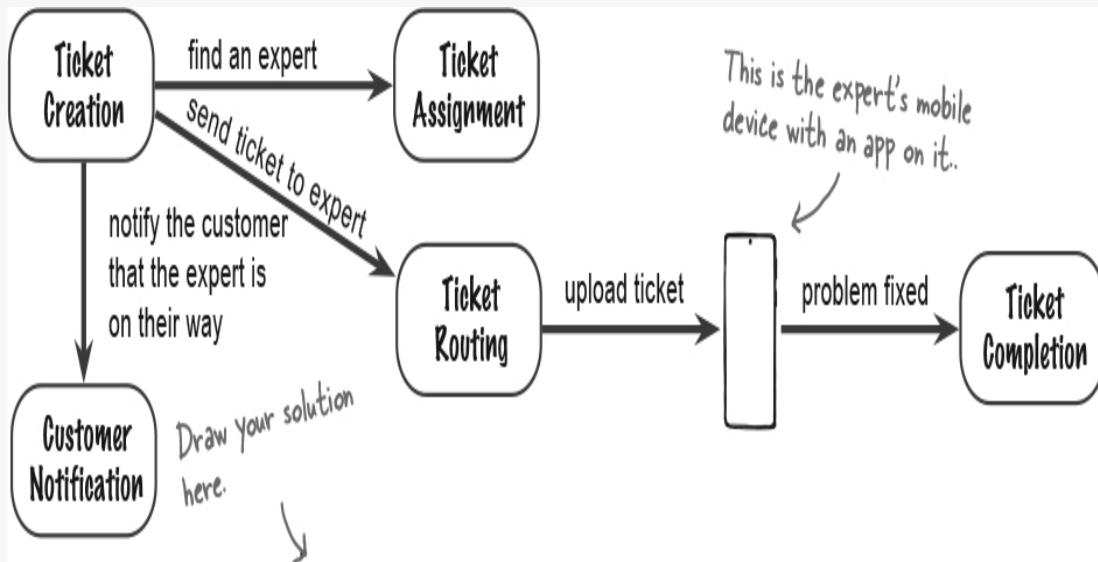
TEST DRIVE



Now it's time to take the Law of Demeter for a test drive to see if you can decouple a logical architecture for a trouble ticket system where customers who have purchased a support plan can submit a trouble-ticket for an electronic item they have purchased and have an expert come out to fix it.

Here's how it currently works: A customer creates a ticket. The ticket then gets assigned to an expert in the field. Once assigned, the ticket is then sent to that expert's mobile device (that's Ticket Routing) and the customer is notified that the expert is on their way. Once the expert marks the ticket as completed, the system sends out a survey to the customer

Assuming you keep the components the same, do you see a way to make the components in this architecture more loosely coupled?



A balancing act

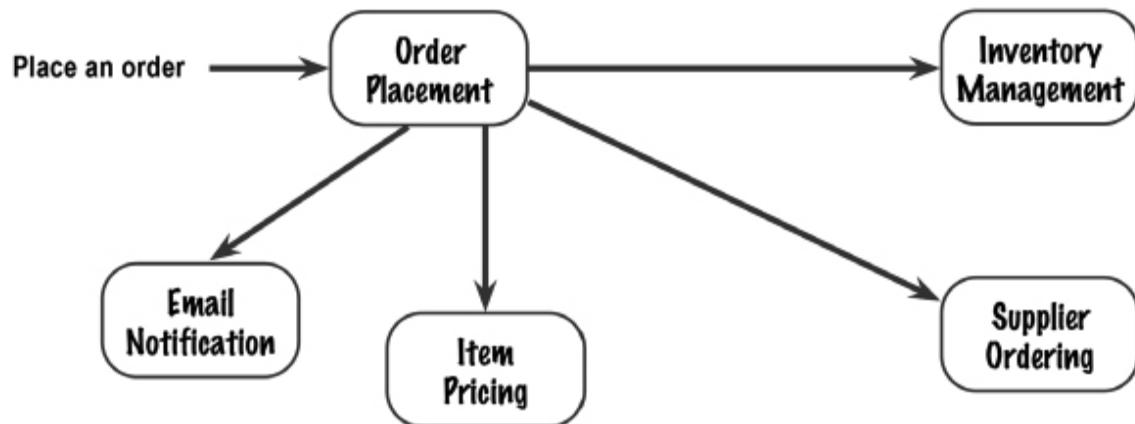
Do you remember the first law of software architecture? Here it is again (because it's so important):

Everything in software architecture is a trade-off.

Loose coupling is no exception. Let's compare the two architectures we've just seen and analyze their trade-offs.



Tightly coupled



Centralized workflow, but more risk with each change

NOTE

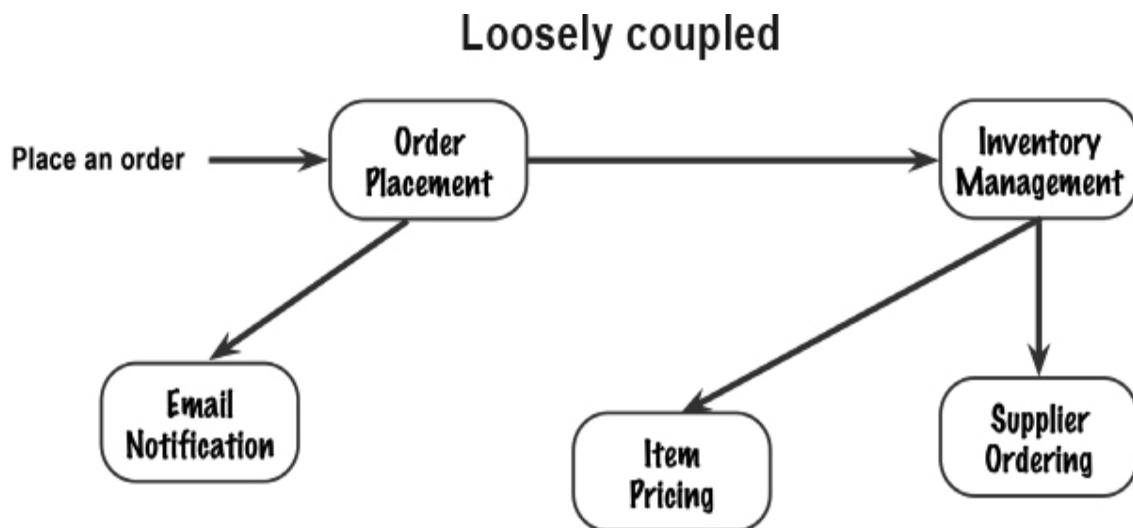
These are the trade-offs for tight coupling.

With the tightly coupled architecture, if you wanted to know what happens when a customer places an order, you would only have to look at the Order Placement component to understand the workflow.

However, in this case, the Order Placement component is dependent on 4 other components. If any one of those components changes, it could break the Order Placement component.

With loose coupling, you distribute the knowledge about what needs to happen, so that no one component knows all the steps. If you wanted to understand the workflow of placing an order, you would have to go to multiple components to get the full picture.

However, changing the Item Pricing and Supplier Ordering components will no longer affect the Order Placement component.



Distributed workflow, but less risk with each change

NOTE

These are the trade-offs for tight coupling.

Two components are coupled if a change in one component might cause a change in the other component.

NOTE

This is a good rule to remember.

Some final words about components

Congratulations! By identifying logical components and the dependencies between them, you’re on your way to creating a software architecture. We know this was a long chapter, but it’s also an important one. Thinking about a system as a collection of logical components helps you, as an architect, better understand its overall structure and how it works.

In the next part of your software architecture journey, you’ll be focusing on the technical details of the system—things like architecture styles, services, databases, and communication protocols. But before you go, review the following bullet points to make sure you fully understand everything about logical components.

BULLET POINTS

- ◊ *Logical components* are the functional building blocks of a system.
- ◊ A logical component is represented by a *directory structure*—the folder where you put your source code.
- ◊ When naming a component, be sure to provide a descriptive name to clearly identify what the component does.
- ◊ Creating a logical architecture involves four continuous steps: identify components, assign requirements, analyze component responsibilities, and analyze architectural characteristics.
- ◊ You can use the *workflow approach* to identify initial core logical components by assigning the steps in a primary customer journey to components.
- ◊ You can use the *actor/action approach* to identify initial core logical components by identifying the actors in the system and assigning their actions to components.
- ◊ The *entity trap* is an approach that associates each major entity in the system to a component. Avoid using this approach, because it creates ambiguous components that are too large and have too much responsibility.
- ◊ When assigning requirements to components, review the component's role and responsibility to make sure it should be performing that function.
- ◊ *Coupling* happens when components depend on one other to perform a business function.
- ◊ *Afferent coupling*, also known as incoming coupling, occurs when other components are dependent on a target component.
- ◊ *Efferent coupling*, also known as outgoing coupling, occurs when a target component is dependent on other components.

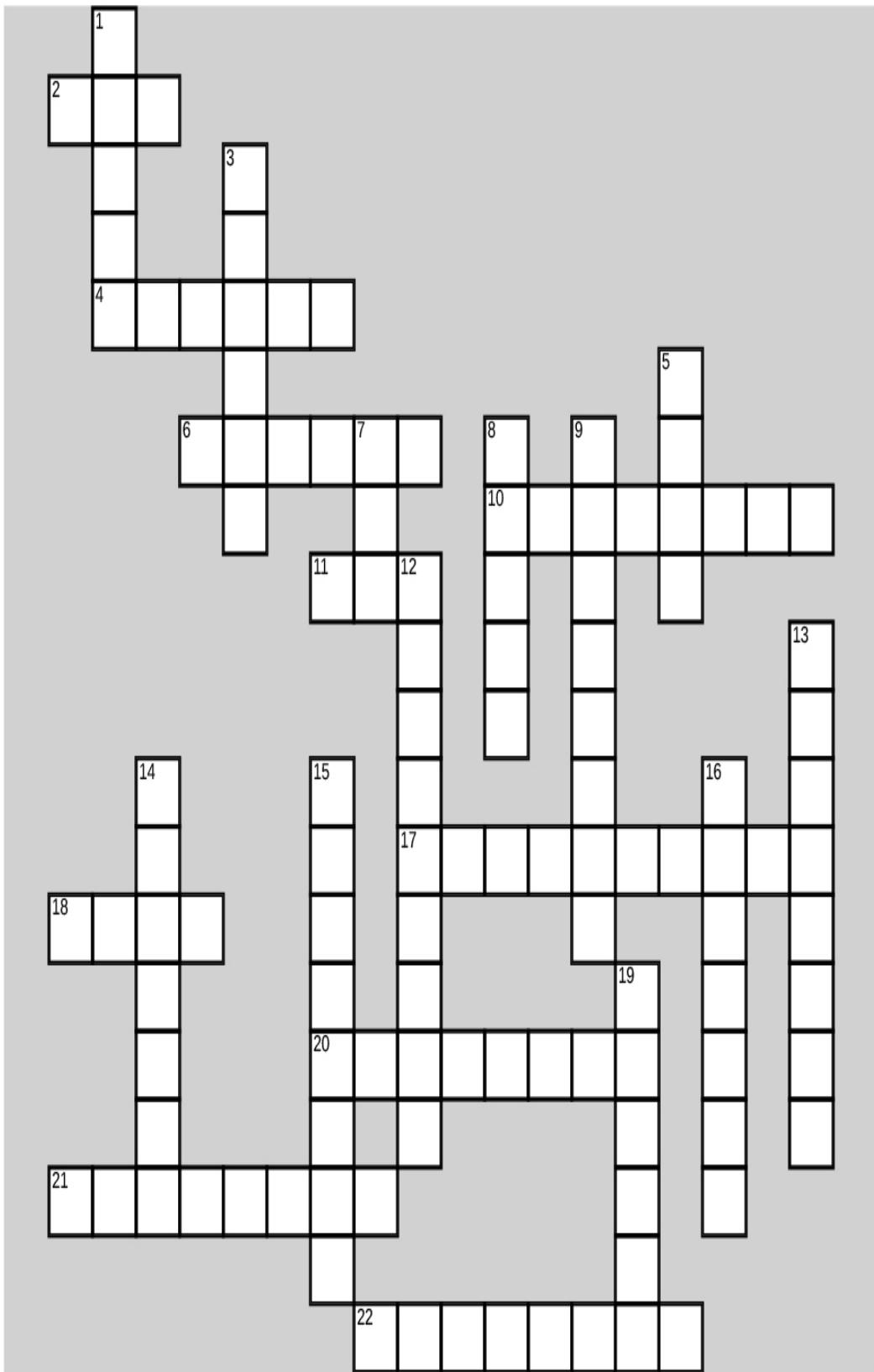
- ◊ Afferent and efferent coupling are forms of *static coupling*.
- ◊ Components having too much knowledge about what needs to happen in the system increases component coupling.
- ◊ The Law of Demeter is useful for creating loosely coupled systems.
- ◊ While loose coupling reduces component dependencies, it also distributes workflow knowledge, making it harder to manage and control that knowledge.
- ◊ Determining the total coupling of a logical architecture involves adding the afferent and efferent coupling levels for each component.

Software Architecture Crossword

Now's your chance to have a little fun and see how much knowledge you've gained. See if you can fill in this crossword puzzle with clues about logical components.



Head First Software Architecture, Chapter 4



Across

- 2** Avoid building a big ball of _____
- 4** Afferent and efferent coupling are both forms of _____ coupling
- 6** One system component might be a live video _____
- 10** It can be tight or loose
- 11** Adventurous Auctions lets users _____ on trips
- 17** Logical _____ are the functional building blocks of a system
- 18** Give each component a descriptive _____
- 20** A physical architecture associates _____ with components
- 21** A user's journey through the system is called their _____
- 22** Coupling might be _____ or efferent

Down

- 1** Identifying logical components may involve taking your best _____
- 3** Be sure to avoid the _____ trap
- 5** Each logical component has a _____ and a responsibility
- 7** _____ gateways appear in a physical architecture but not a logical one
- 8** You can identify components with an _____/action approach
- 9** Each component performs a _____
- 12** A good place to look for components is the codebase's _____ structure
- 13** An architecture diagram can show the logical or _____ architecture
- 14** The Principle of Least Knowledge is also called the Law of _____
- 15** The degree and manner to which the operations of a component are interrelated
- 16** Early on, you'll identify _____ core components
- 19** Step 2 is to _____ requirements to logical components

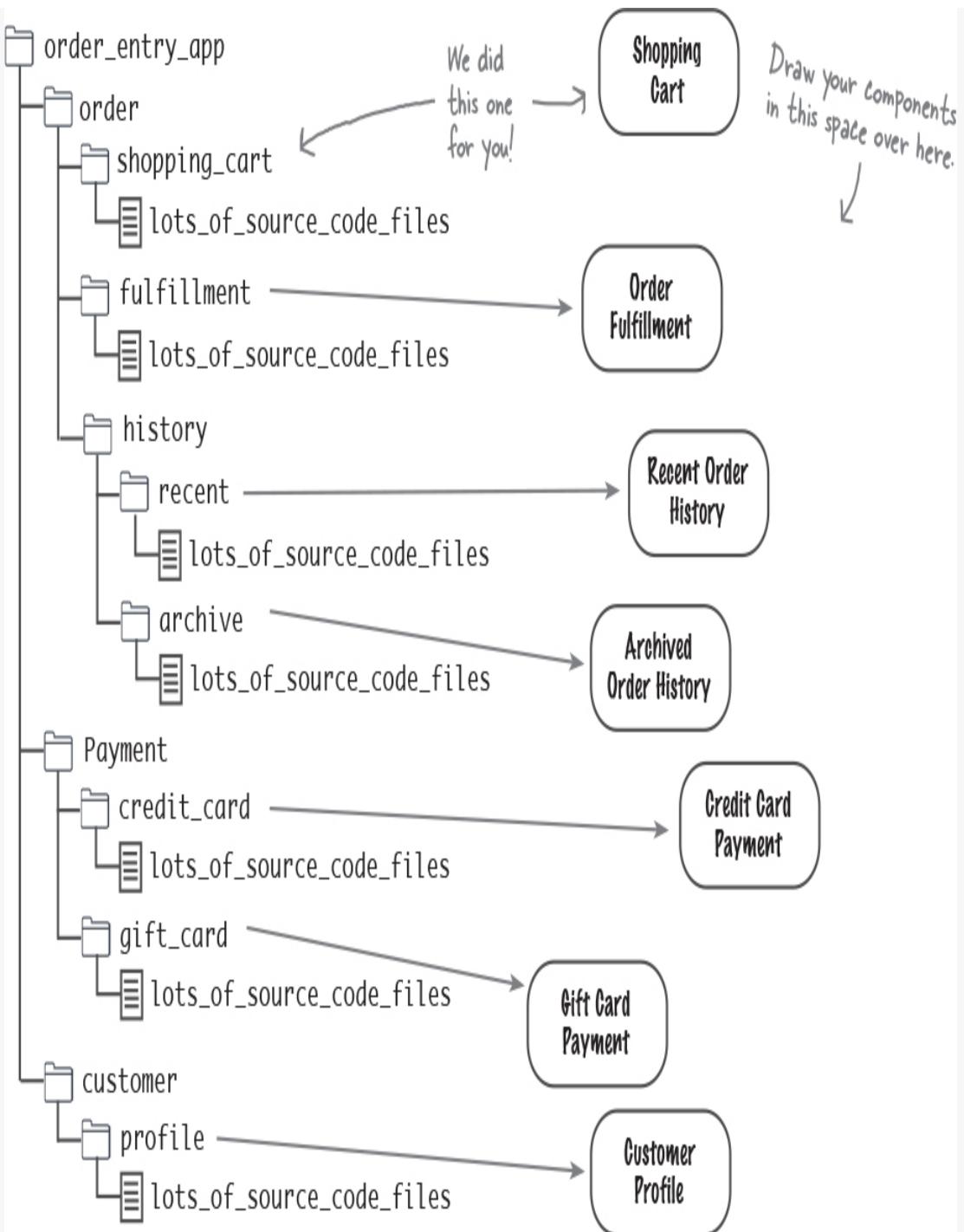
EXERCISE SOLUTION



Name that component

From Example 4-1

It's your first week on the job as the new architect, and you've been assigned to an existing project to build a trouble-ticket system. You want to understand the logical components of the architecture, but your team doesn't know anything about logical components—they just started coding. To identify the logical architecture you have to look at the existing directory structure. How many individual logical components can you identify from the codebase below?



We did
this one
for you!

Shopping
Cart

Draw your components
in this space over here.

Order
Fulfillment

Recent Order
History

Archived
Order History

Credit Card
Payment

Gift Card
Payment

Customer
Profile

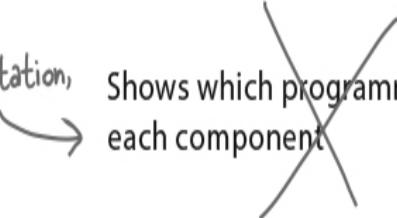
WHO DOES WHAT? SOLUTION

From “Who Does What?”

We had our logical and physical architecture responsibilities all figured out, and somehow they got all mixed up. Can you help us figure out who does what? Be careful—some responsibilities may not have a match (they aren’t part of a logical or physical architecture).

This is part of implementation,
not architecture.

Shows which programming language is used for
each component



Logical architecture

Maps components to services

Shows the logical components within the system and how they communicate with each other

Physical architecture

Shows communication between services and the protocol they use (like REST)

Shows the source code files used to implement a component

Shows the components and their interactions within the user interface

Shows the API gateways and load balancers used in the system

SHARPEN YOUR PENCIL SOLUTION



From “Sharpen your pencil”

Your company wants a new system to assign workers to construction sites, and it's your job as the software architect to identify its initial core components. Using the workflow approach, identify as many core components as you can, matching each to its associated workflow step. Remember, a workflow step can have multiple components, and not every workflow step has to have a unique component.



Step 1. Maintain a list of all construction workers, their skills, and their locations

Step 2. Create a new construction project and specify the work site

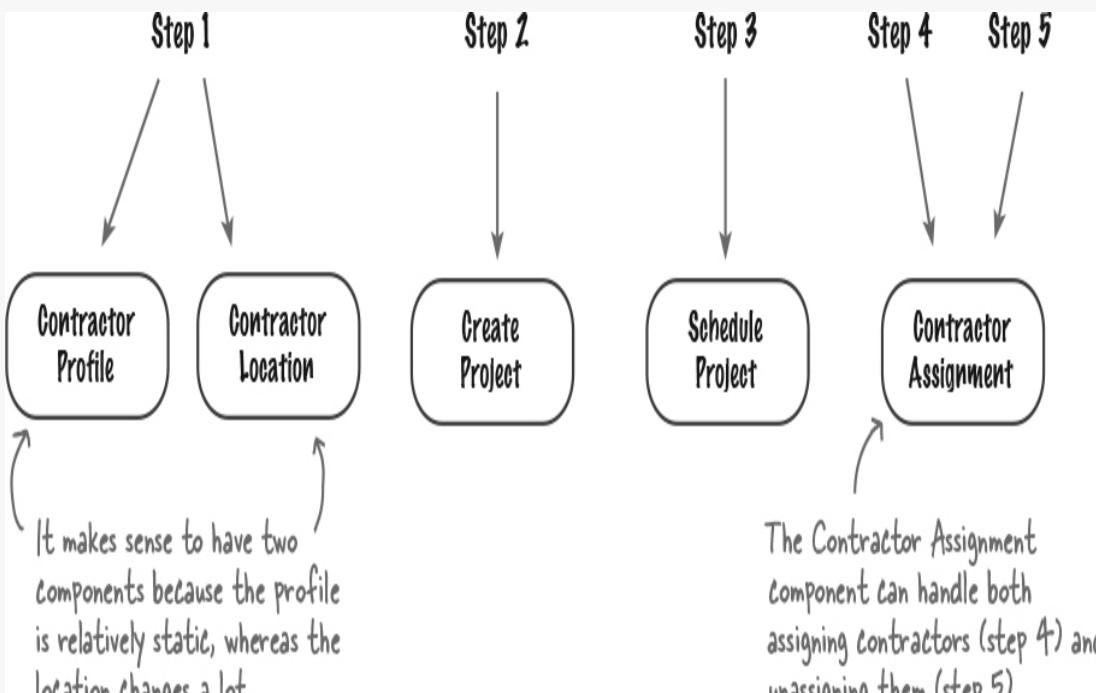
Step 3. Create a schedule for when various construction projects start and end

Step 4. When a new project starts, assign workers based on their skills and locations

Step 5. When the project completes, free up workers so they can be reassigned

NOTE

Draw your logical components in this space here. Remember to give them good descriptive names.



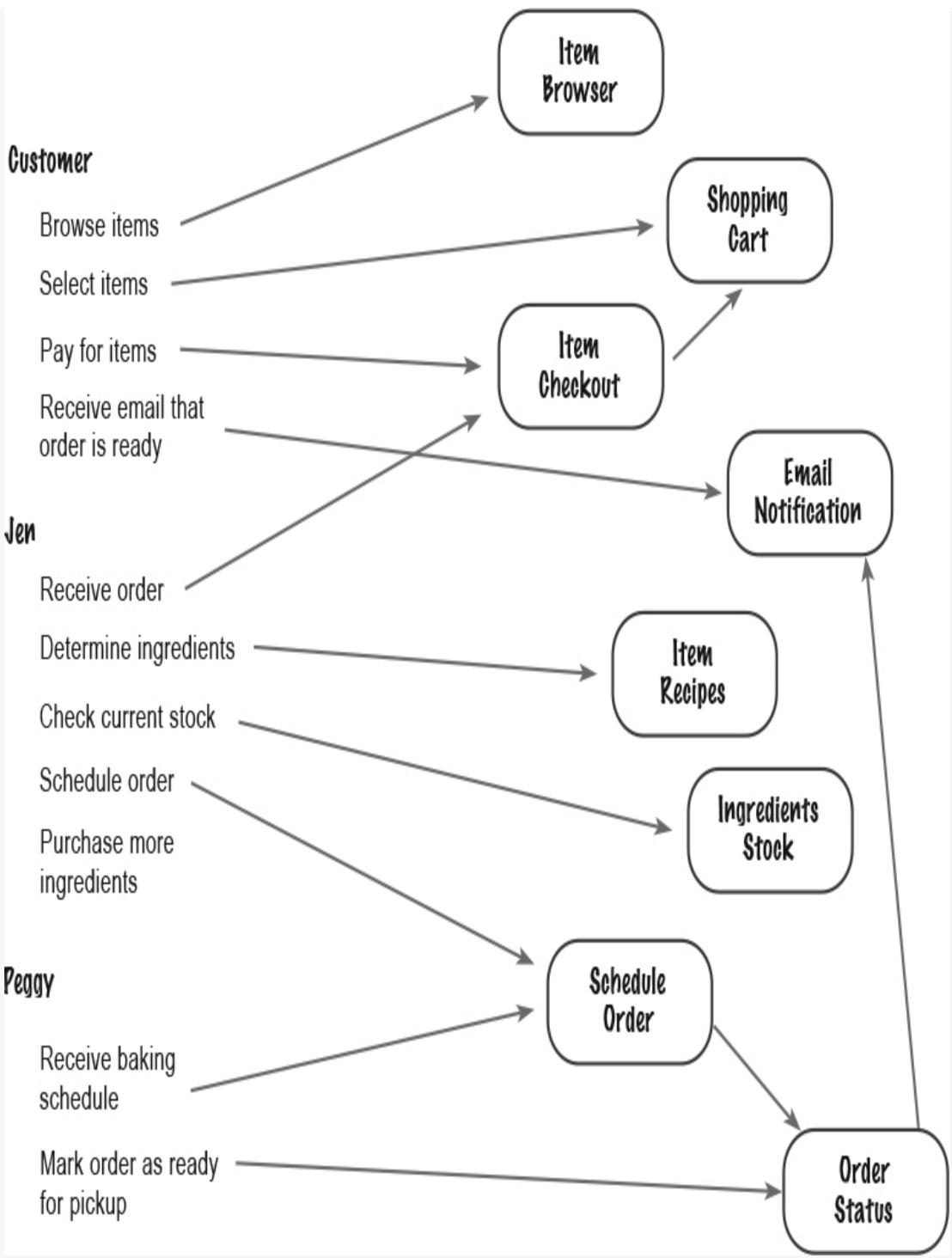
EXERCISE SOLUTION



Peggy's bakery is ready to expand operations, and would like a new system that lets customers view, order, and pay for bakery items online for pickup. Orders are sent to the Jen (the bakery coordinator), who purchases ingredients and schedules orders. Peggy receives the schedule of items to bake each morning and tells the system when she is done with an order. The system then sends an email to the customer letting them know their items are ready for pickup.

Using the actor/action approach, identify what actions each actor might take, then draw as many logical components as you can for the new bakery system, matching the actions you identified to the components.





SHARPEN YOUR PENCIL SOLUTION



From “Sharpen your pencil”

What other words besides **manager** can you list that, if they appeared in a component name, might be a good indication you are in the entity trap?

We did this one for you.	supervisor	agent	Add the word "Order" to the front of these words and you'll see what we mean.
	handler		
	controller	service	
utility			
	engine		
worker	mediator	orchestrator	
	coordinator		processor

EXERCISE SOLUTION



From “Exercise”

Can you select the most appropriate approach for identifying initial core components in the following scenarios? In some cases, more than one approach may be appropriate.

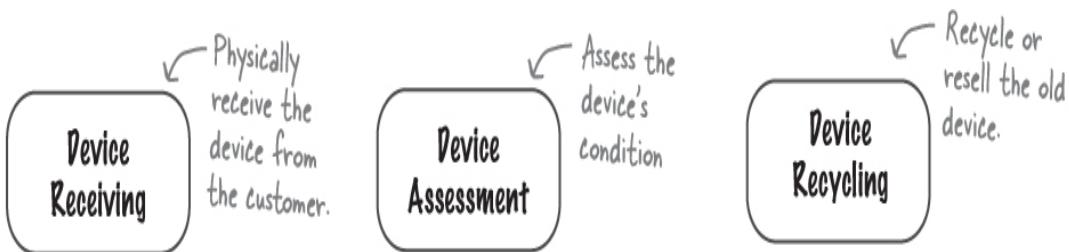
The system has only one type of user	<input checked="" type="checkbox"/> Workflow	<input type="checkbox"/> Actor/Action	<input type="checkbox"/> Entity Trap
The system has well-defined entities	<input checked="" type="checkbox"/> Workflow	<input checked="" type="checkbox"/> Actor/Action	<input type="checkbox"/> Entity Trap
You have minimal functional requirements	<input checked="" type="checkbox"/> Workflow	<input checked="" type="checkbox"/> Actor/Action	<input type="checkbox"/> Entity Trap
The system has many complex user journeys	<input checked="" type="checkbox"/> Workflow	<input type="checkbox"/> Actor/Action	<input type="checkbox"/> Entity Trap
The system has many types of users	<input type="checkbox"/> Workflow	<input checked="" type="checkbox"/> Actor/Action	<input type="checkbox"/> Entity Trap

Remember to avoid the entity trap, even if the system has well-defined entities.

WHO DOES WHAT? SOLUTION

From “Who Does What?”

Your company, Going Green Corporation, wants a new system to support a new electronics recycling system where customers can send in their old electronic devices (like a cellphone), and get money. We've already identified some of the initial core components. Your job is to now figure out which component should be responsible for the user stories listed below or if a new component is needed, and if so, what the name should be.



Locate the nearest safe disposal facility for destroying the device

Device Receiving Device Assessment Device Recycling Other: _____

Capture and store the customer information (name, address, etc.)

Device Receiving Device Assessment Device Recycling Other: Customer Profile

Post the device on a third-party site to resell it on the secondary market

Device Receiving Device Assessment Device Recycling Other: _____

Make a payment to the customer for their recycled device

Device Receiving Device Assessment Device Recycling Other: Accounts Payable

Determine if the device can be resold or if it should be destroyed

Device Receiving Device Assessment Device Recycling Other: _____

Record that the device has been received and is ready for assessment and valuation

Device Receiving Device Assessment Device Recycling Other: _____

Determine the value (if any) of the recycled device

Device Receiving Device Assessment Device Recycling Other: _____

Determine the monthly profit and loss for recycled and resold devices

Device Receiving Device Assessment Device Recycling Other: Financial Reporting

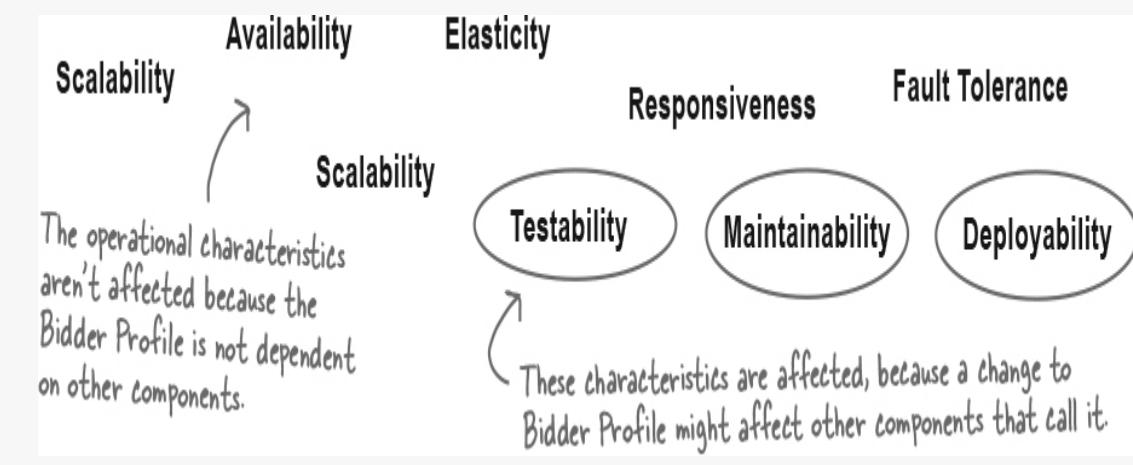
Going Green Corp. needs to make a profit, after all.

EXERCISE SOLUTION



From “Exercise”

Using the Adventurous Auctions logical components above, think about the architecture characteristics associated **only** with the **Bidder Profile** component and circle all of the architecture characteristics listed below that might be impacted because it is afferently coupled to the Auction Registration and Automatic Payment components.

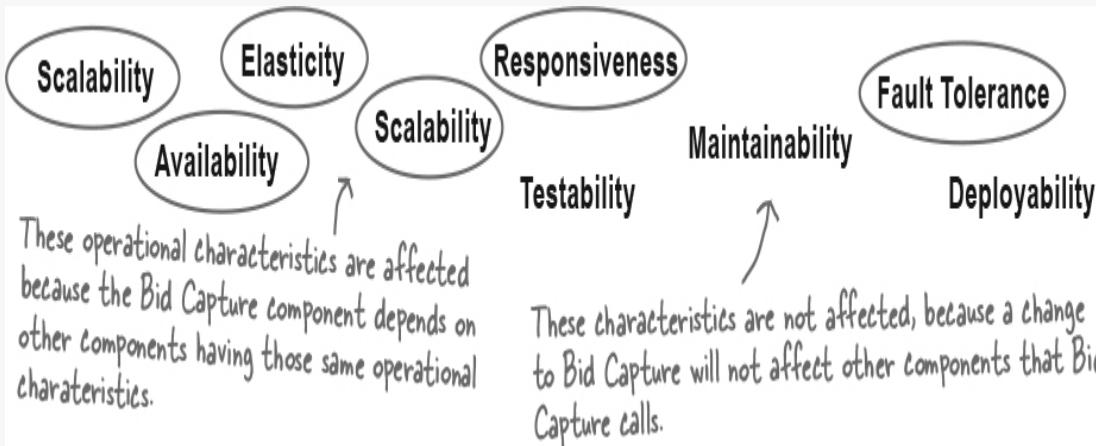


EXERCISE SOLUTION



From “Exercise”

Now's your chance to see the difference between afferent and efferent coupling. Just like the previous exercise, think about the architecture characteristics associated *only* with the **Bid Capture** component above. Now circle all of the architecture characteristics listed below that might be affected by Bid Capture's efferent coupling to the Bid Streamer and Bid Tracker components.



TEST DRIVE SOLUTION

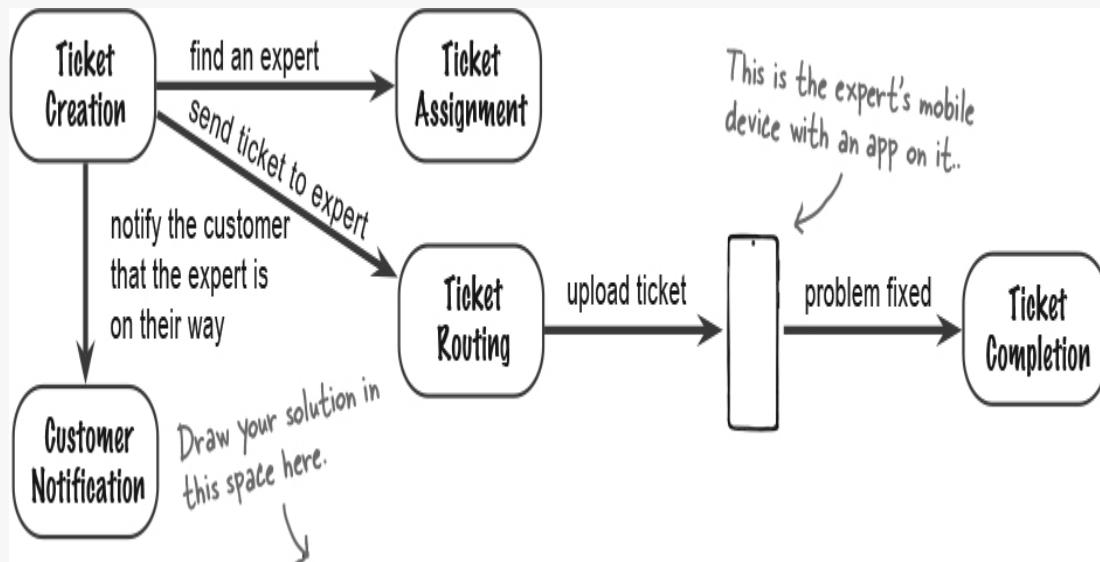


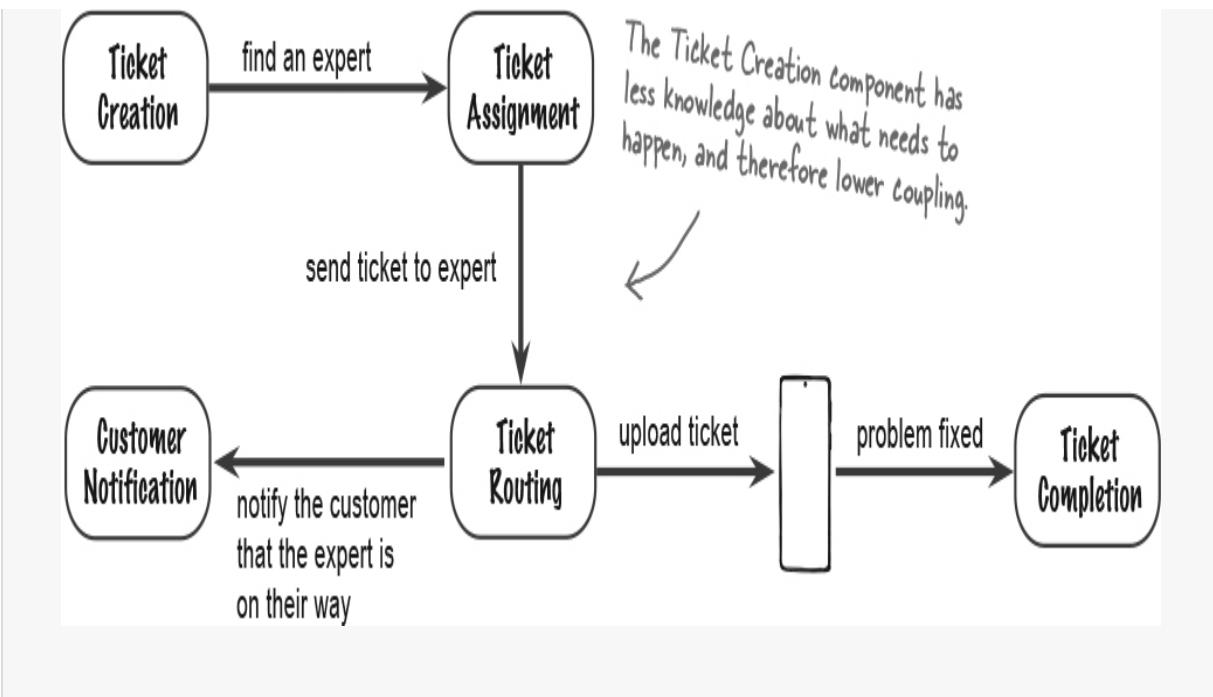
From “Sharpen your pencil”

Now it's time to take the Law of Demeter for a test drive to see if you can decouple a logical architecture. Below is a logical architecture for a trouble ticket system where customers who have purchased a support plan can submit a trouble-ticket for an electronic item they have purchased and have an expert come out to fix it.

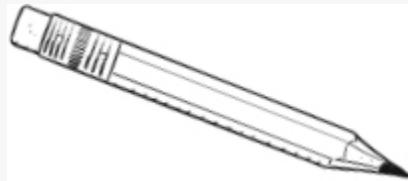
Here's how it currently works: A customer creates a ticket. The ticket then gets assigned to an expert in the field. Once assigned, the ticket is then sent to that expert's mobile device (that's Ticket Routing) and the customer is notified that the expert is on their way. Once the expert marks the ticket as completed, the system sends out a survey to the customer

Assuming you keep the components the same, do you see a way to make the components in this architecture more loosely coupled?



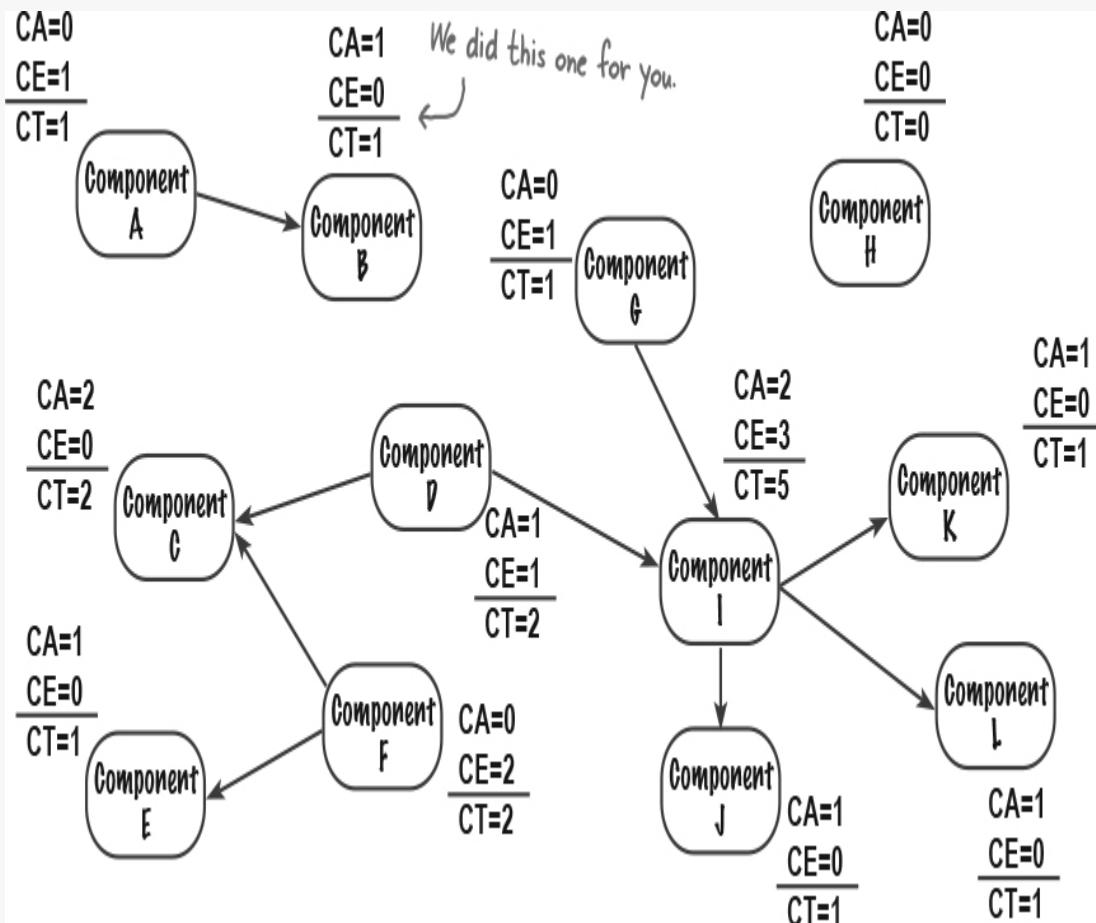


SHARPEN YOUR PENCIL SOLUTION



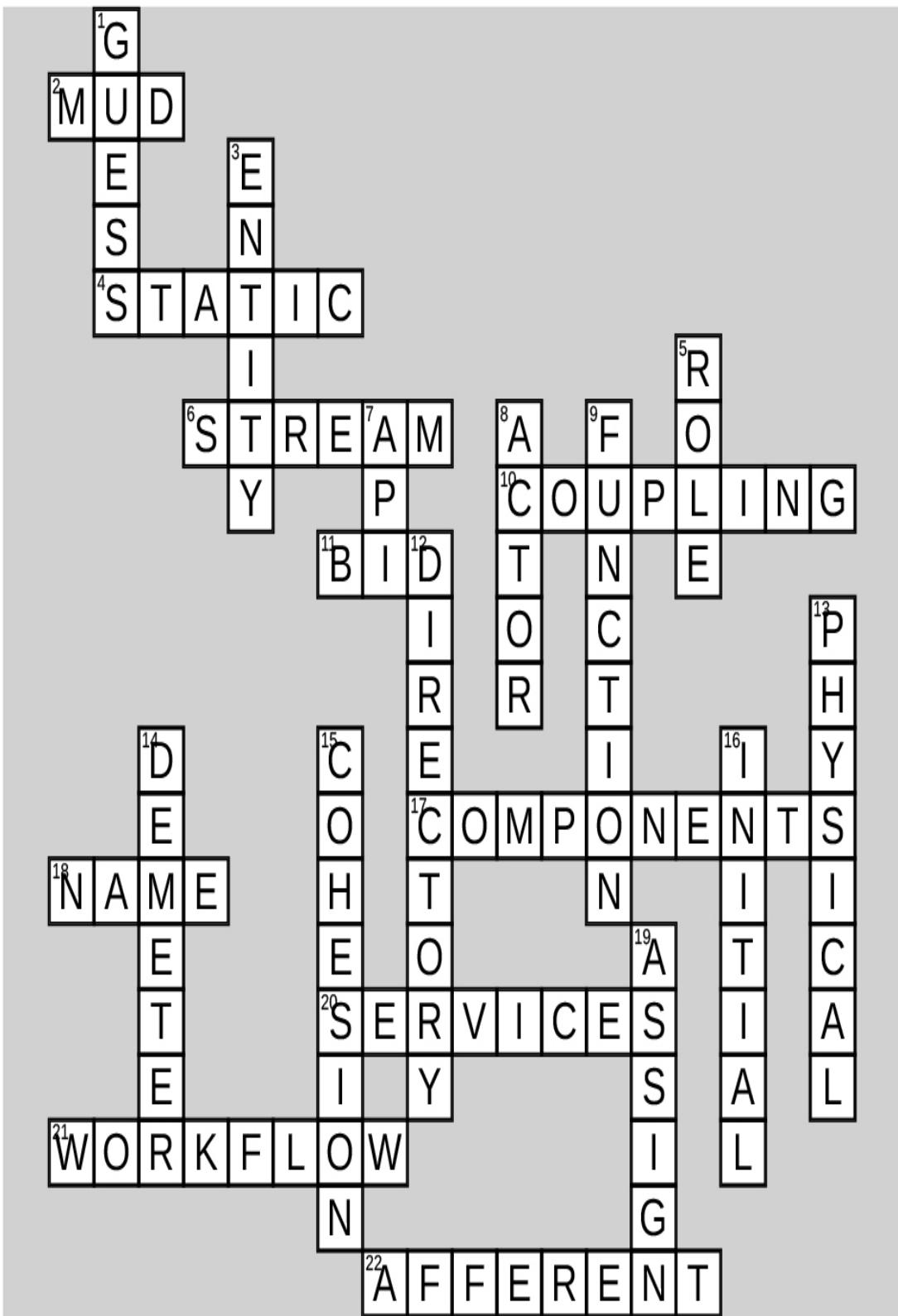
From “Test Drive”

Given the following components below, can you identify the total afferent coupling (CA), total efferent coupling (CE), and total coupling for each component (CT)? Also, what is the total coupling level for this logical architecture? Does the total coupling for this architecture seem high or low to you?



Total system coupling level: 18. This seems high to us.

Head First Software Architecture, Chapter 4



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.