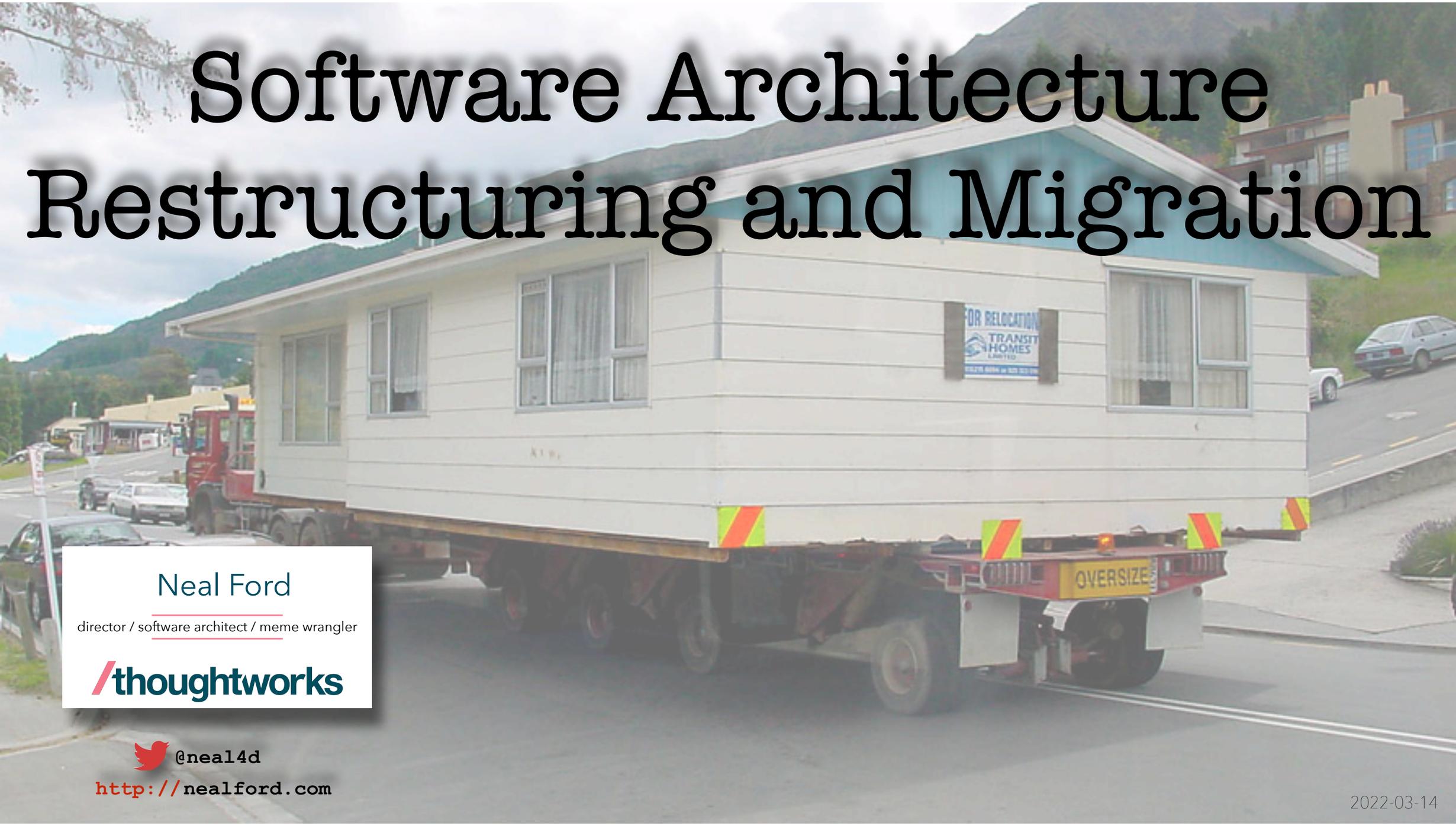
Software Architecture Restructuring and Migration



Neal Ford

director / software architect / meme wrangler

/thoughtworks

@neal4d

http://nealford.com

2022-03-14

what?

how do we assess the current architecture?

where?

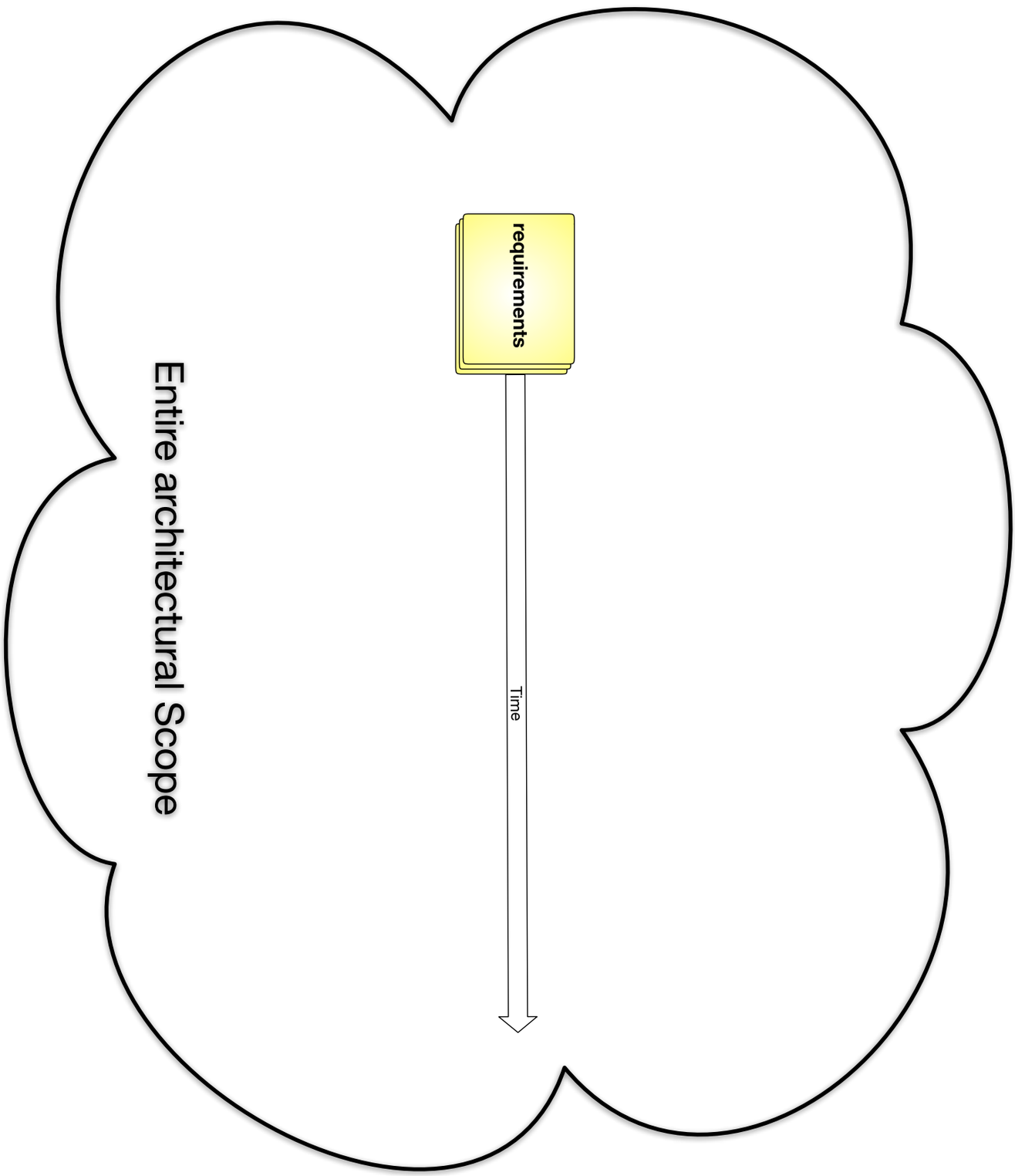
to what should we change it?

how?

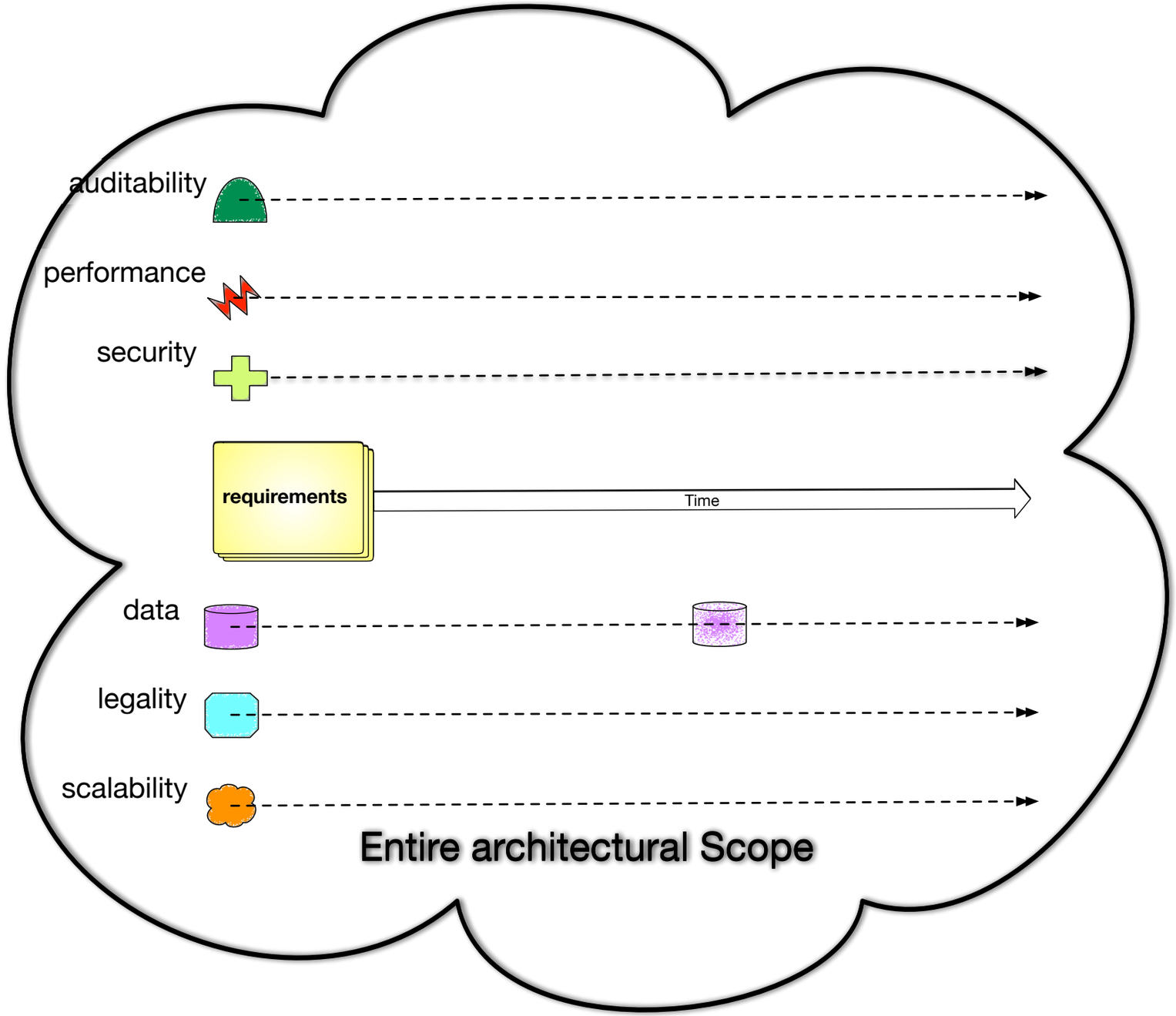
how do we change it?

Scoping architecture characteristics

Architecture Characteristics



Architecture Characteristics



architectural quantum

Your Architectural Kata is...

Going Green

A large electronics store wants to get into the electronics recycling business and needs a new system to support it. Customers can send in their small personal electronic equipment (or use local kiosks at the mall) and possibly get money for their used equipment if it is in working condition.

Requirements:

- Customers can get a quote for used personal electronic equipment (phones, cameras, etc.) either through the web or a kiosk at a mall.
- Customers will receive a box in the mail, send in their electronic, and if it is in good working order receive a check.
- Once the equipment is received, it is assessed (inspected) to determine if it can be either recycled (destroyed safely) or sold (eBay, etc.).
- The company anticipates adding 5-10 new types of electronic that they will accept each month.
- Each type of electronic has its own set of rules for quoting and assessment.
- This is a highly competitive business and is a new line of business for us

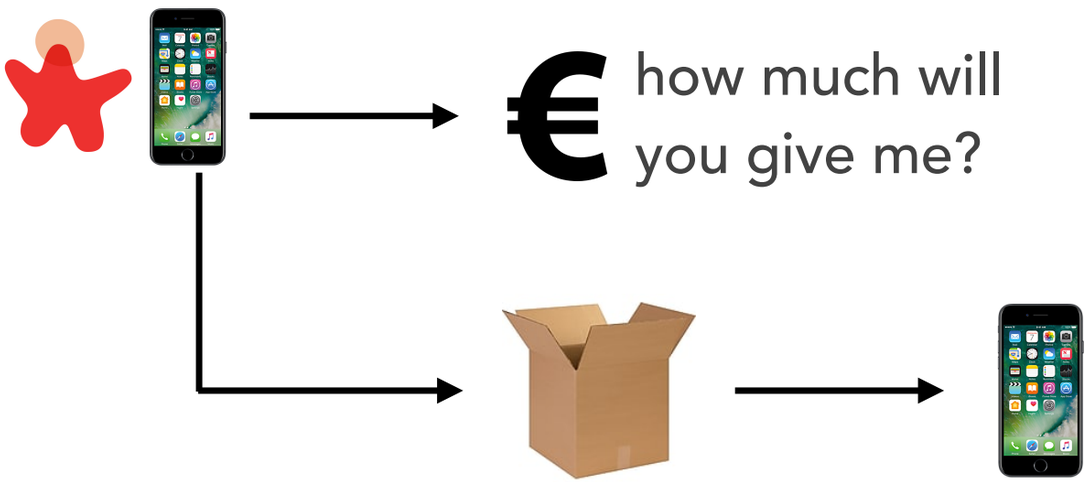
Users: Hundreds, hopefully thousands to millions

architectural quantum

electronics recycling

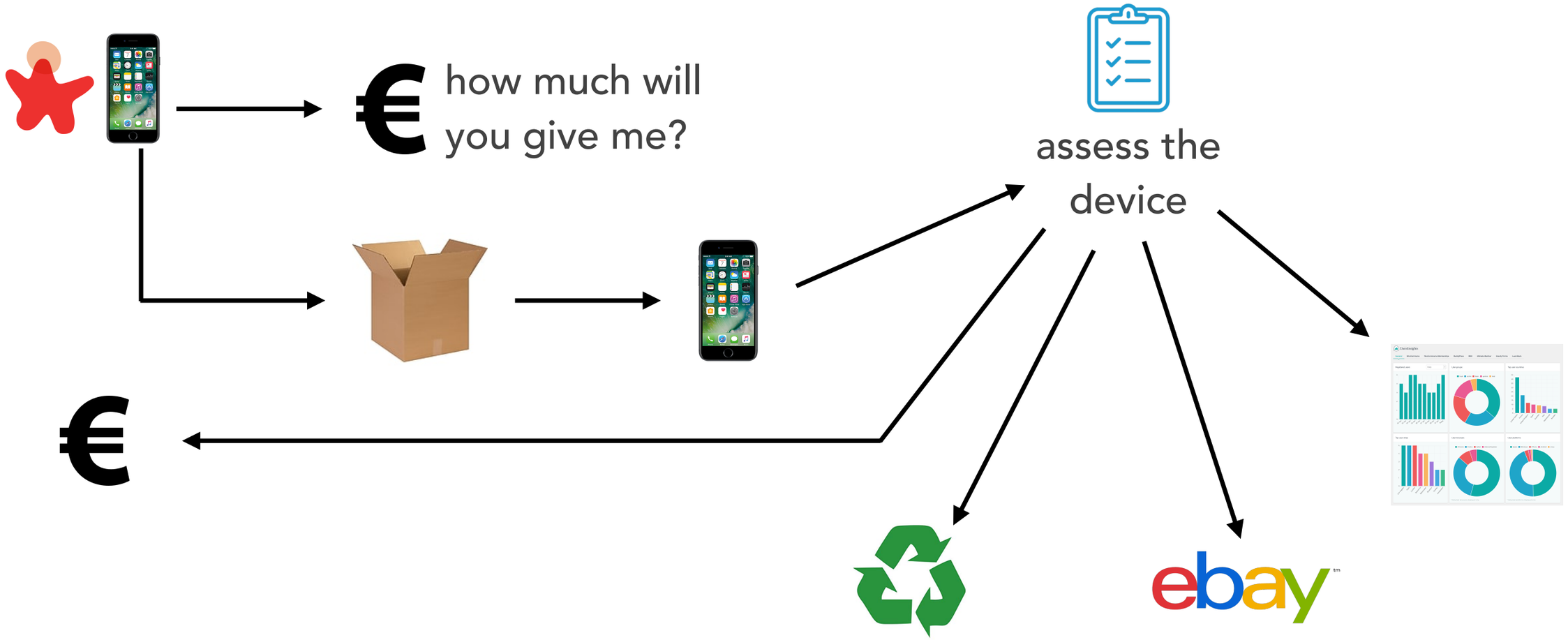
architectural quantum

electronics recycling



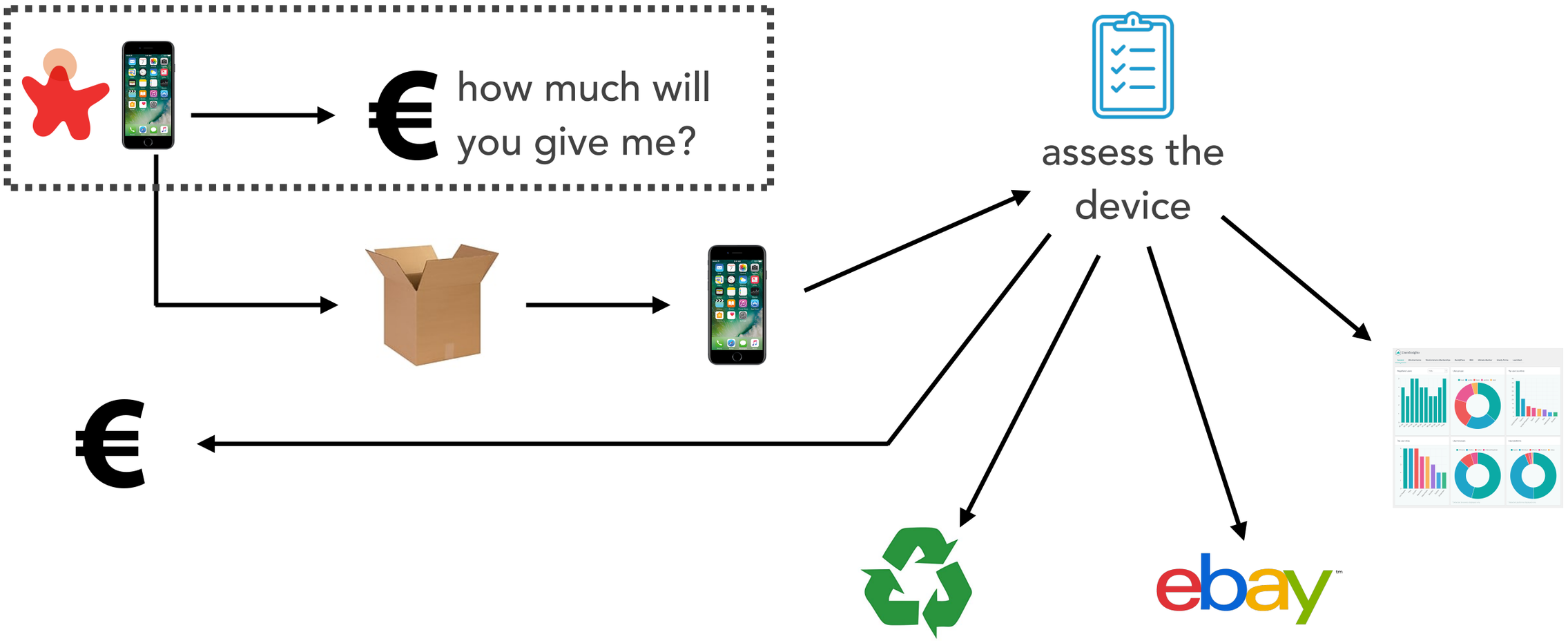
architectural quantum

electronics recycling



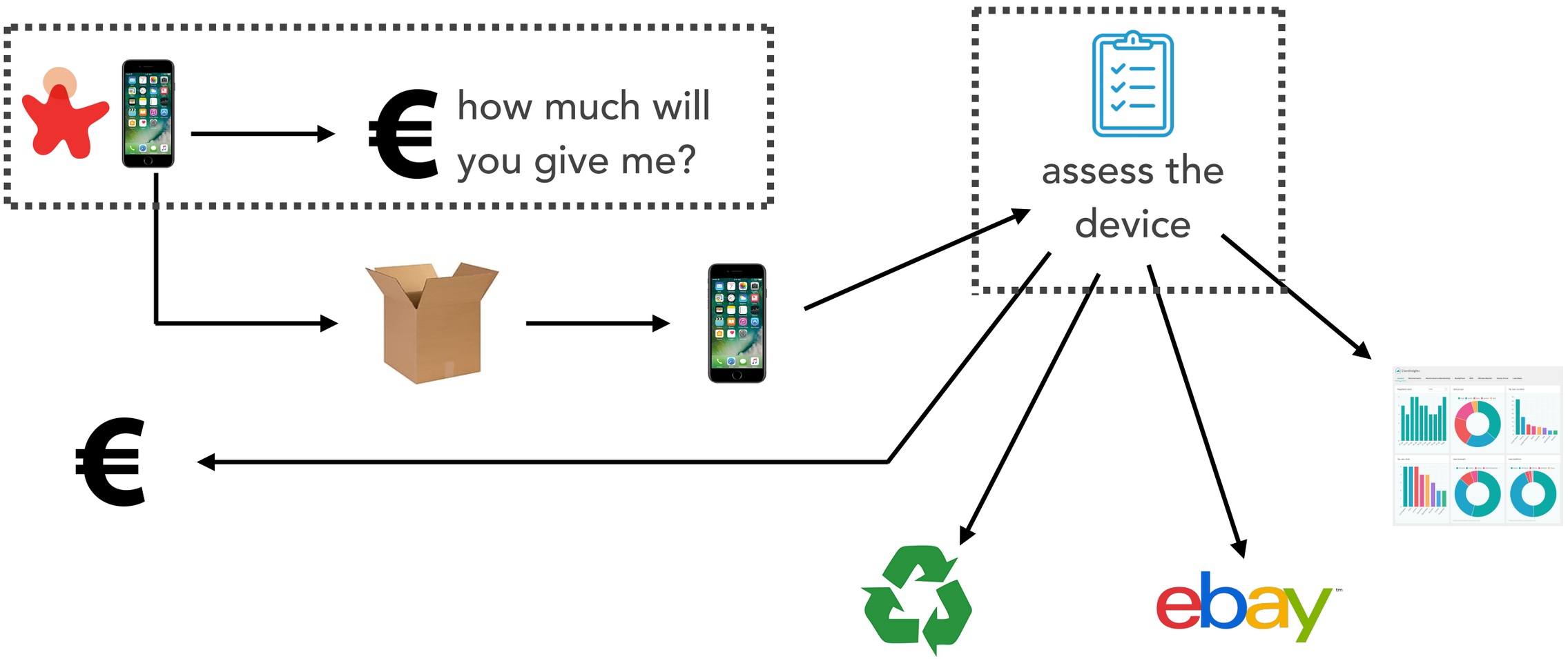
architectural quantum

electronics recycling



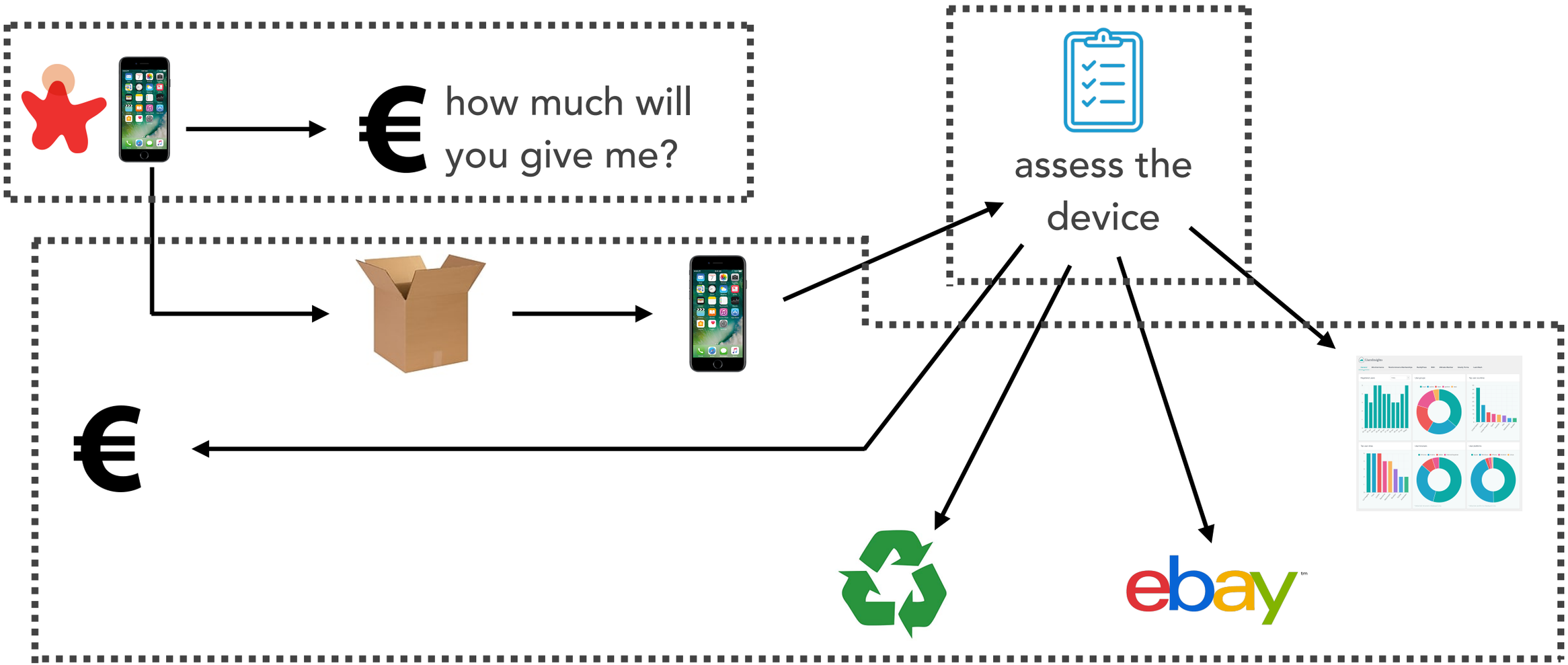
architectural quantum

electronics recycling



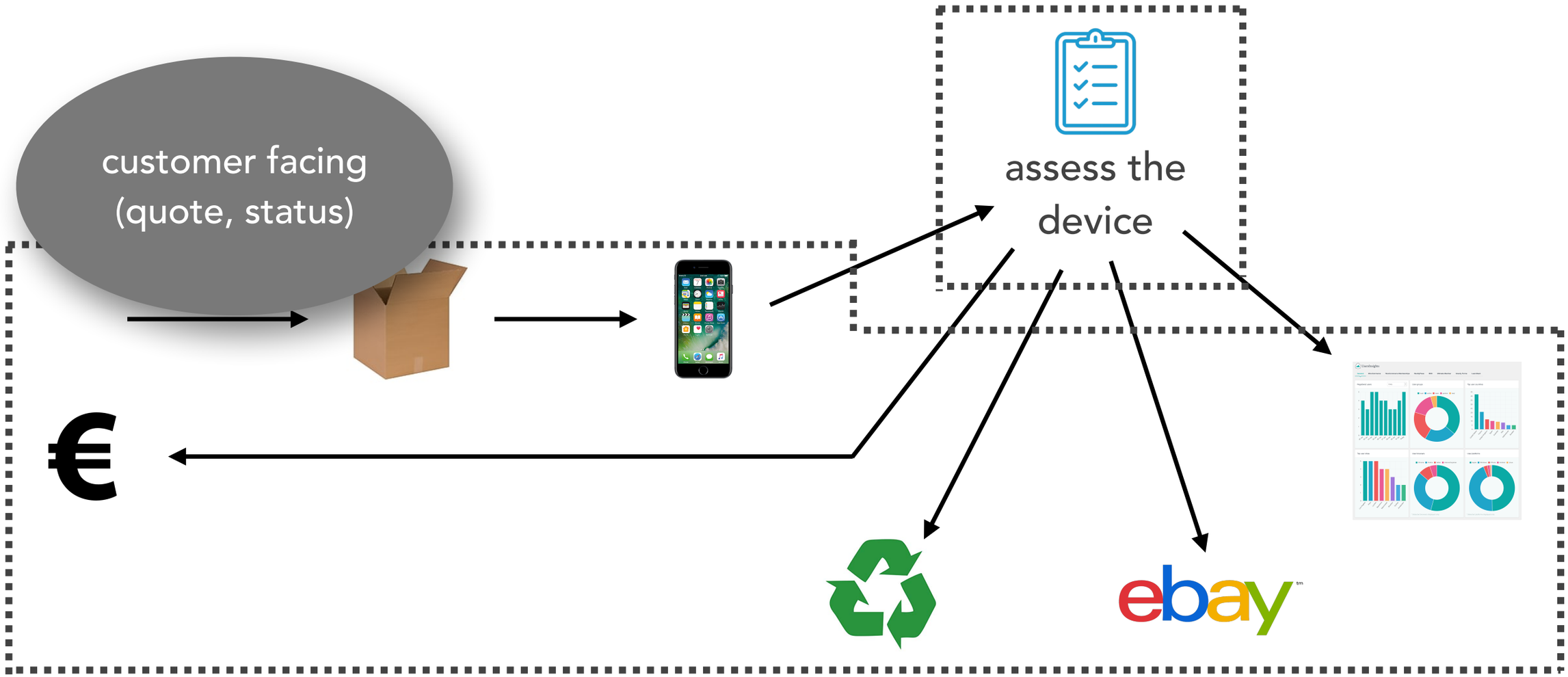
architectural quantum

electronics recycling



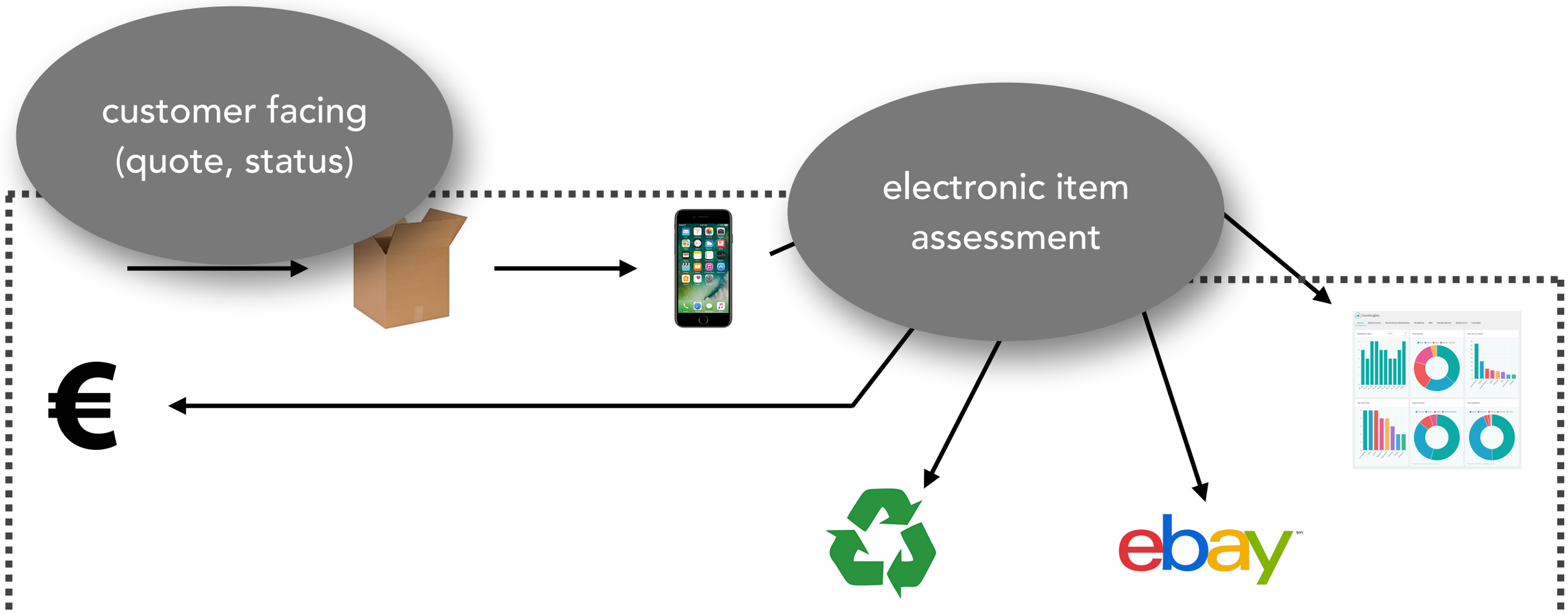
architectural quantum

electronics recycling



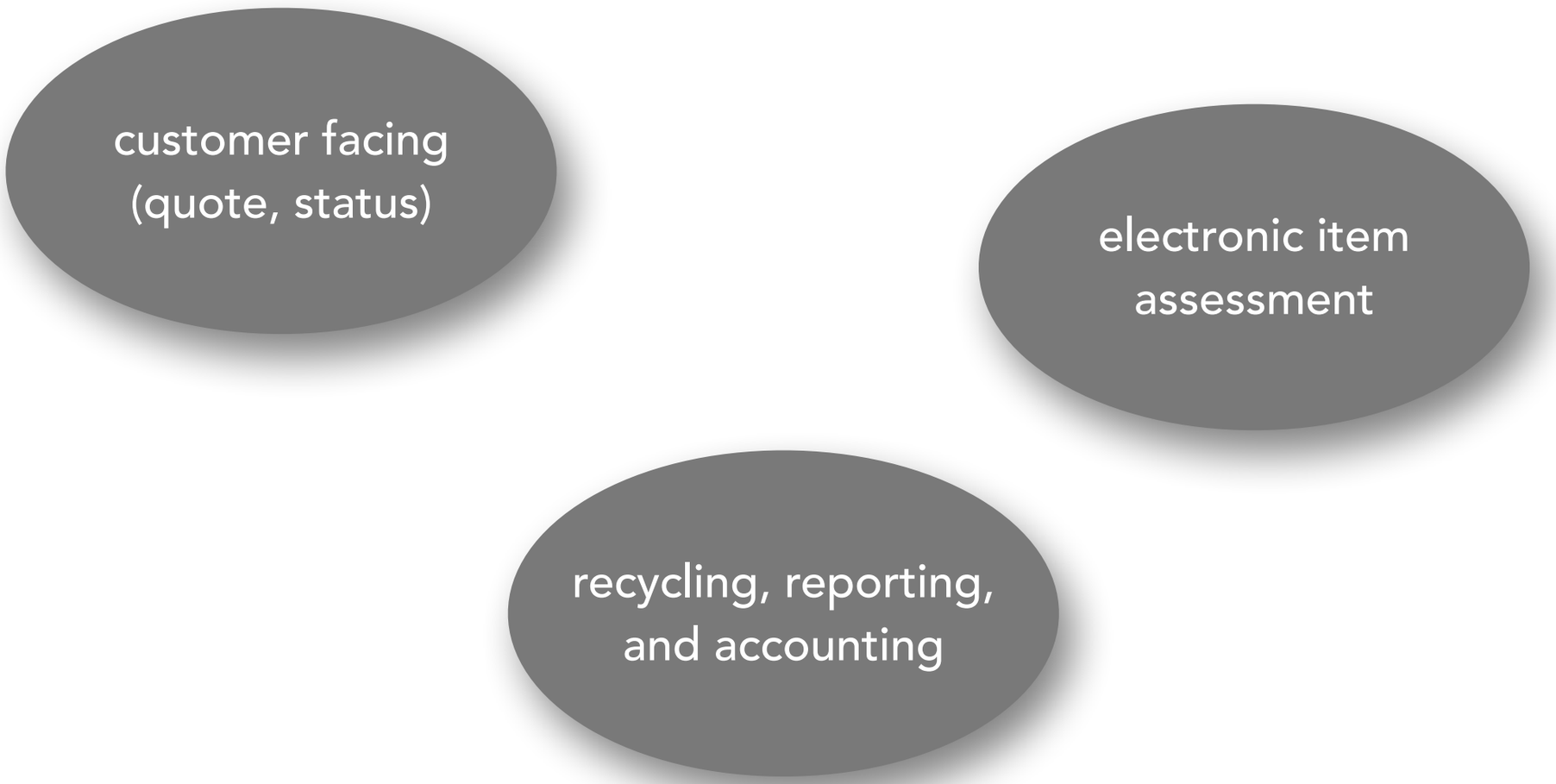
architectural quantum

electronics recycling



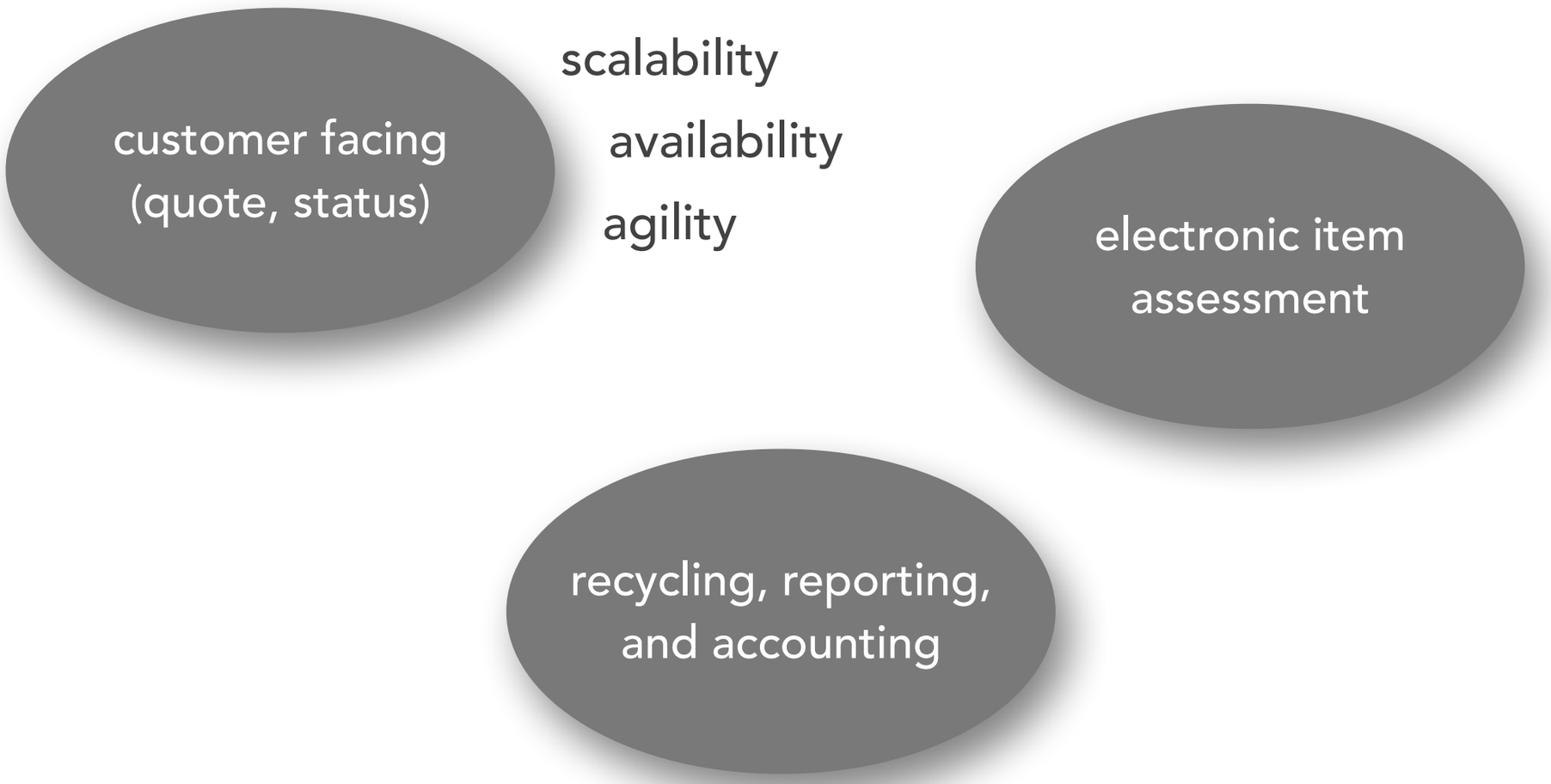
architectural quantum

electronics recycling



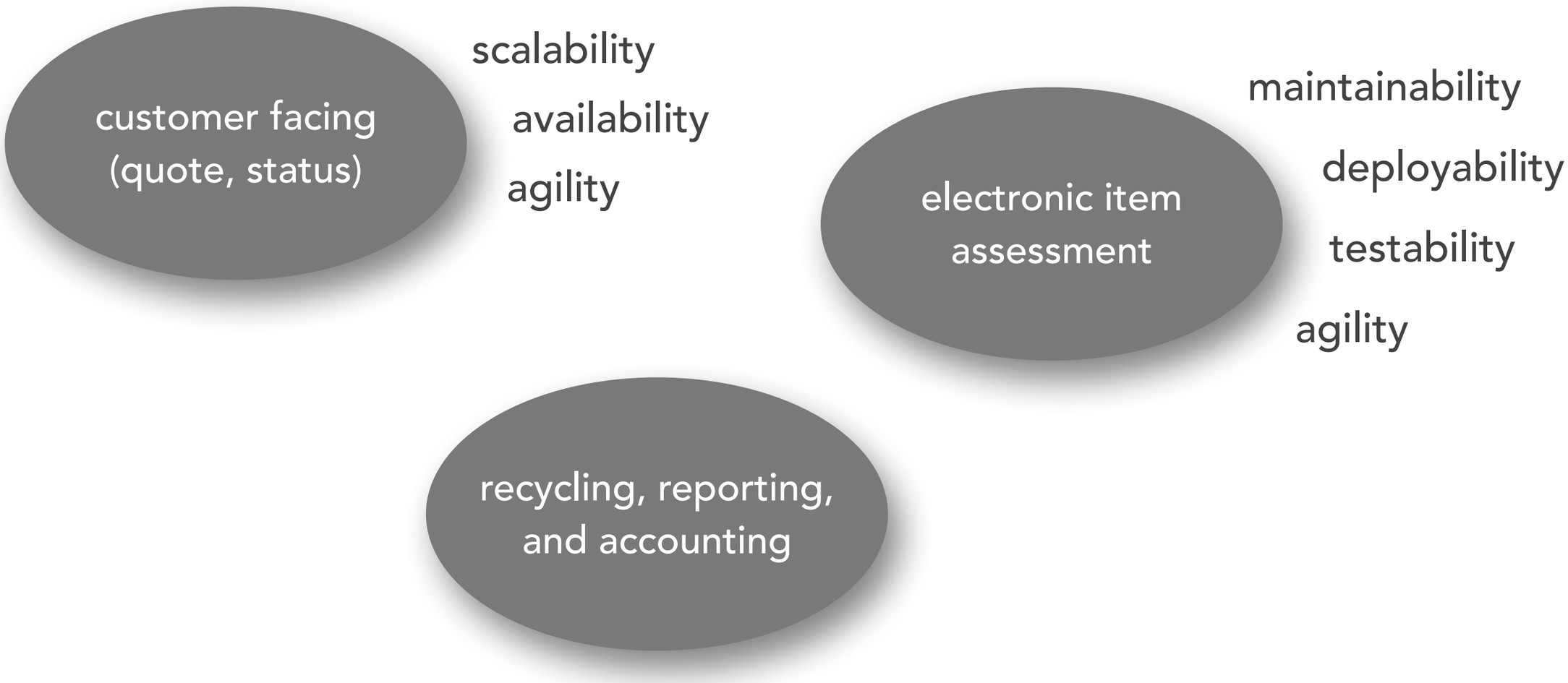
architectural quantum

electronics recycling



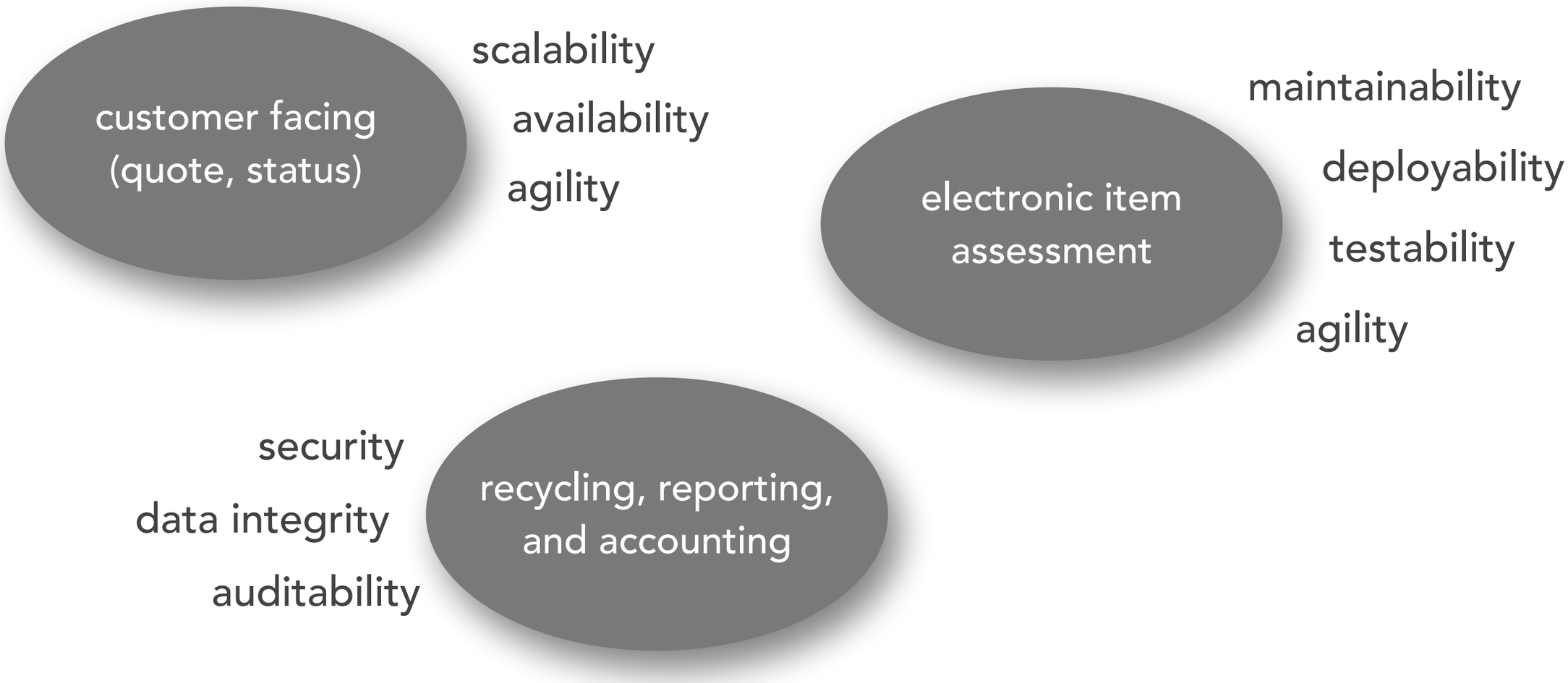
architectural quantum

electronics recycling

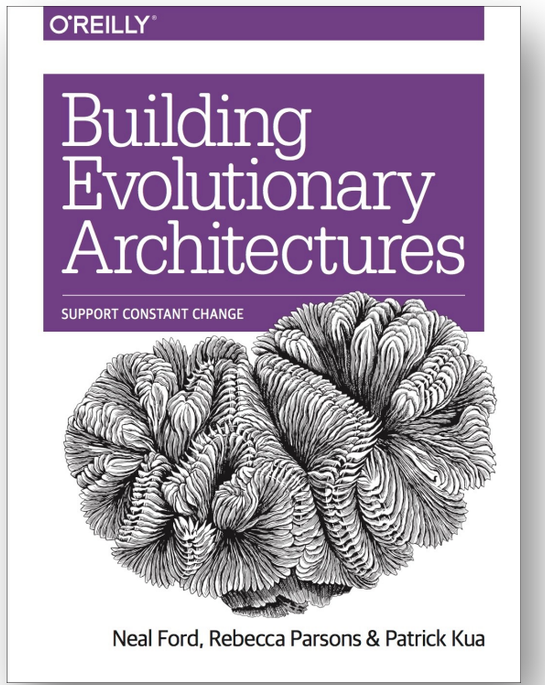


architectural quantum

electronics recycling

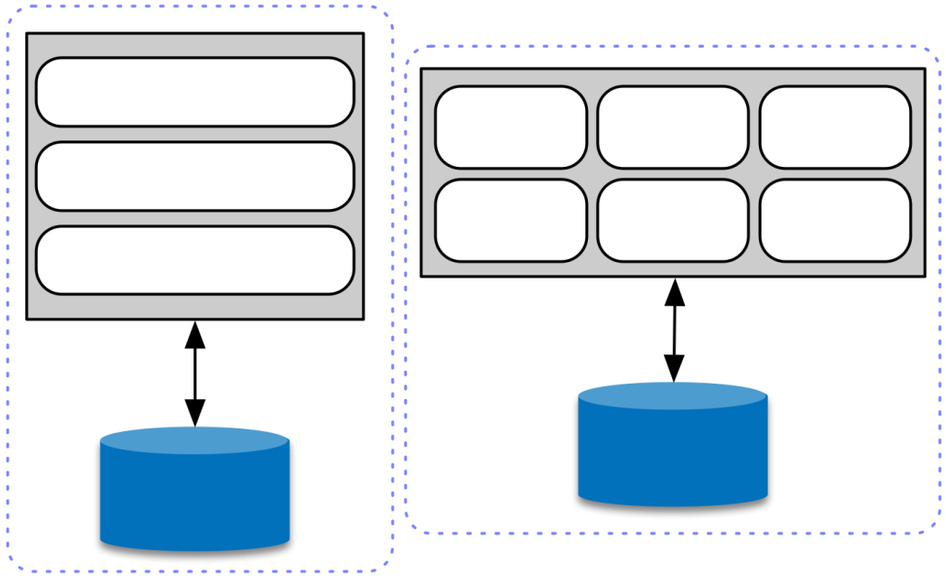


architectural quantum



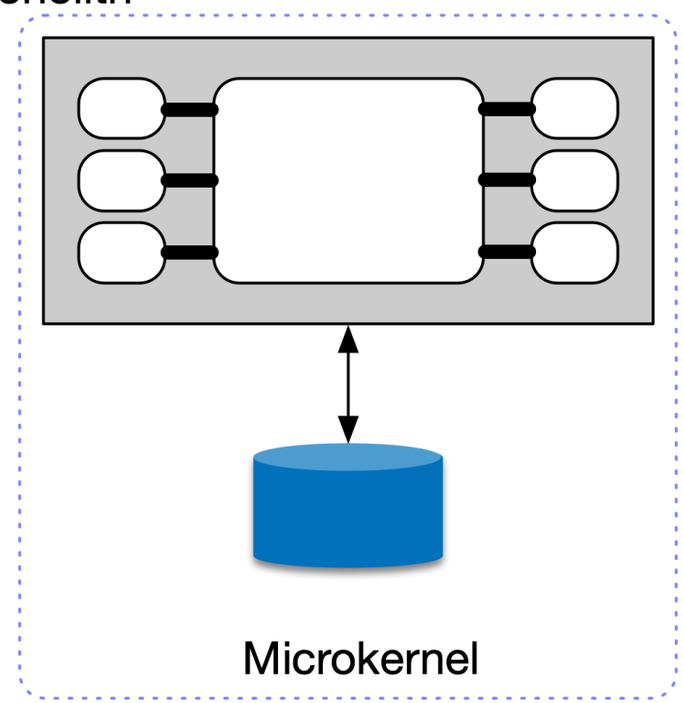
An architectural quantum is an independently deployable component with high functional cohesion .

Single Static Quantum



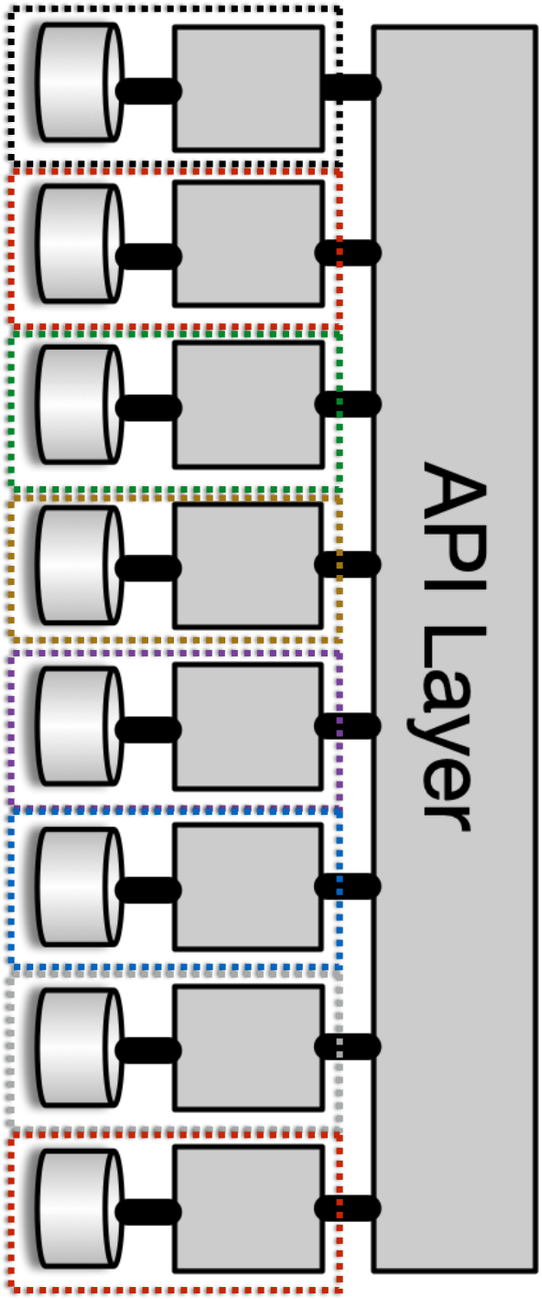
Layered Monolith

Modular Monolith

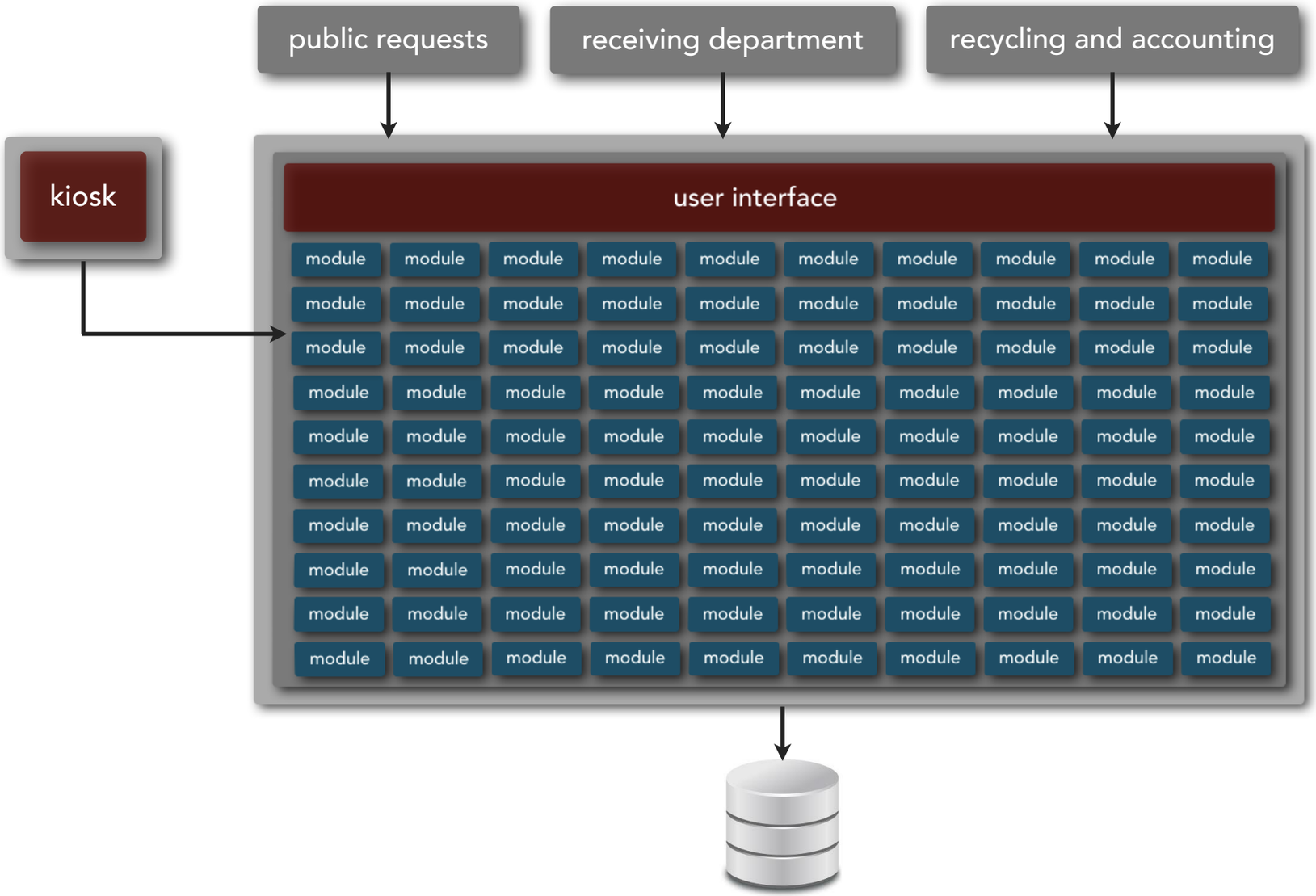


Microkernel

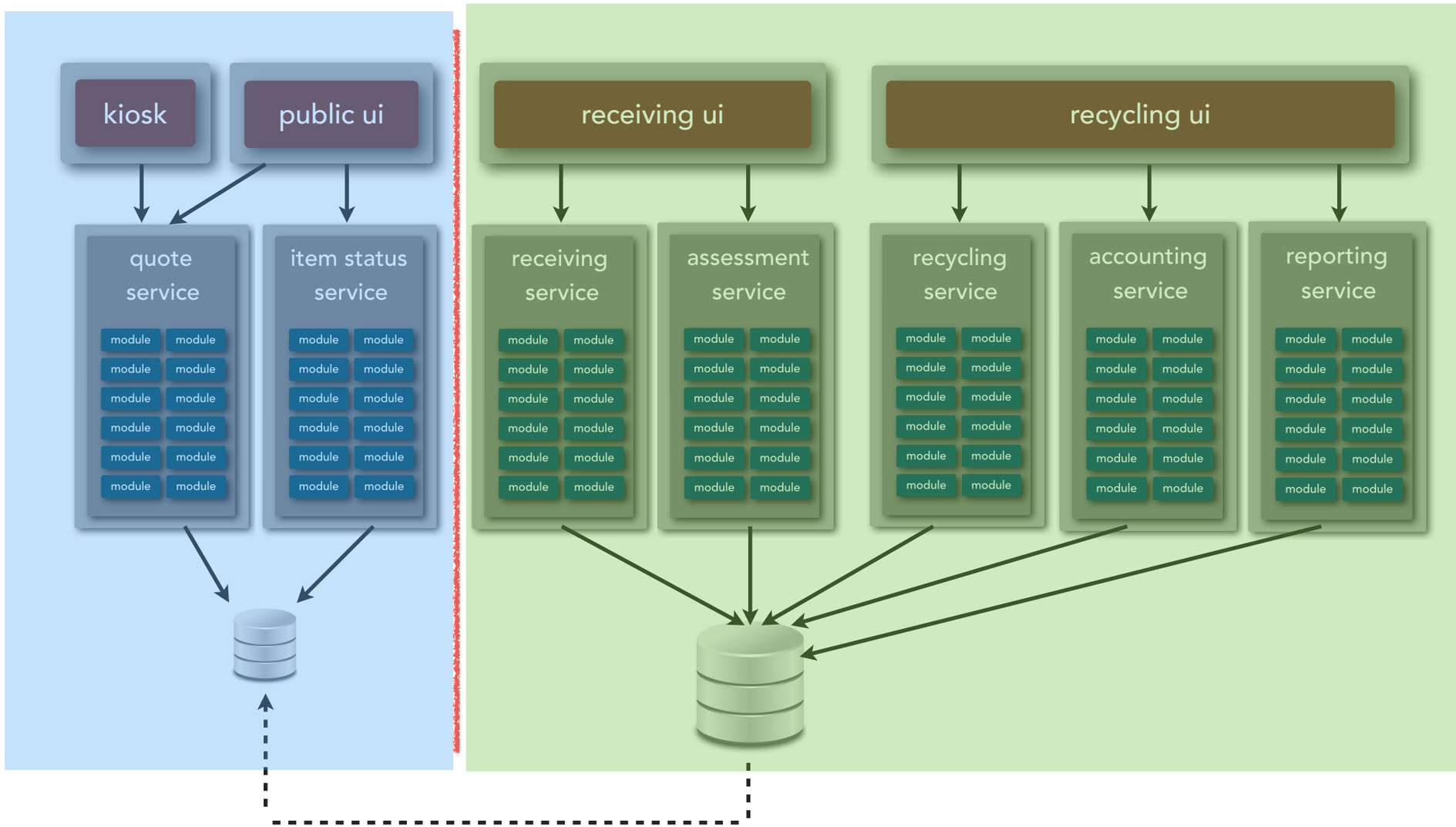
Microservices Quanta (no UI)



Migrating


public requests
receiving department
recycling and accounting
kiosk
user interface
module

Migrating


kiosk
public ui
receiving ui
recycling ui
quote service
item status service
receiving service
assessment service
recycling service
accounting service
reporting service
module

Restructuring architecture?

Moving functionality to different quanta to change architecture characteristics.

Refactoring architecture?

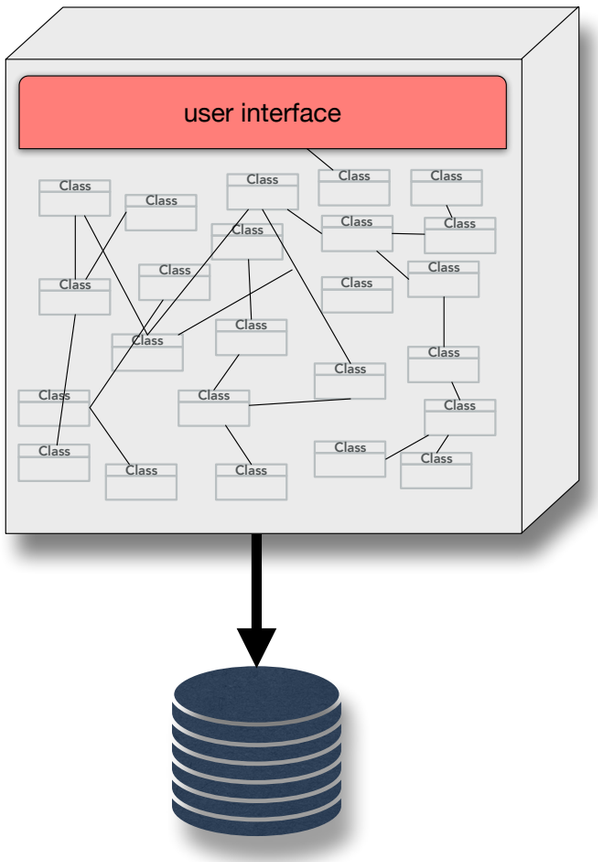
Moving functionality to different quanta to change architecture characteristics without changing domain behavior.

Migrating architecture?

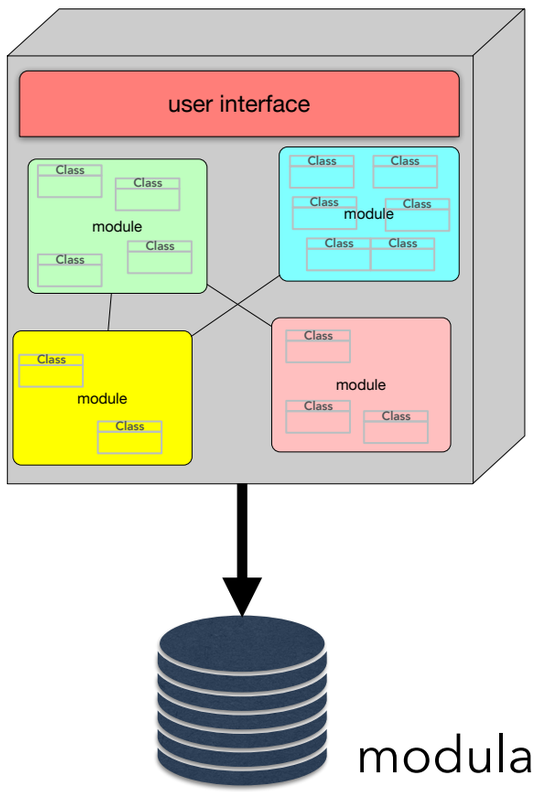
Moving functionality to different quanta to change architecture characteristics and changing/evolving domain behavior.

Architecture Partitioning

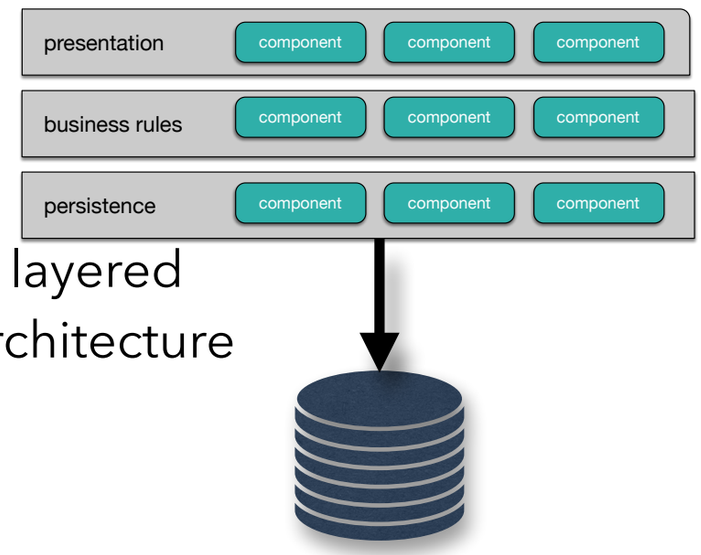
Monoliths



unstructured monolith



modular monolith

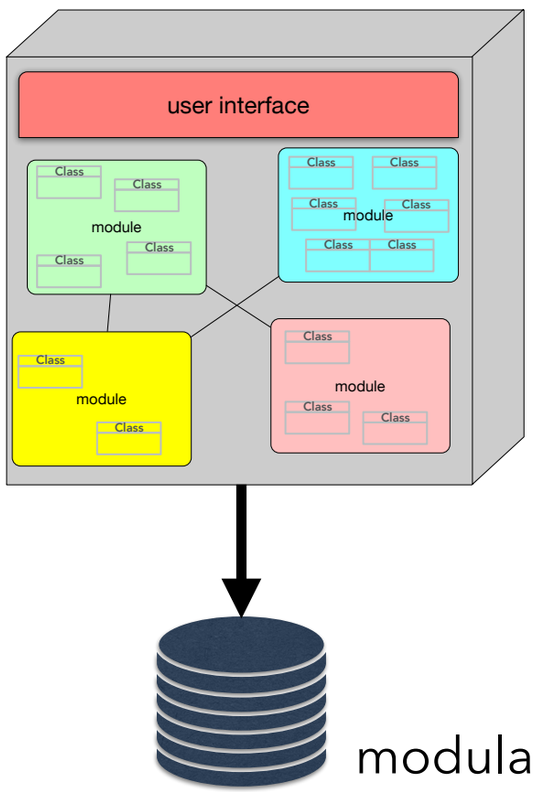


layered architecture

http://www.codingthearchitecture.com/presentations/sa2015-modular-monoliths

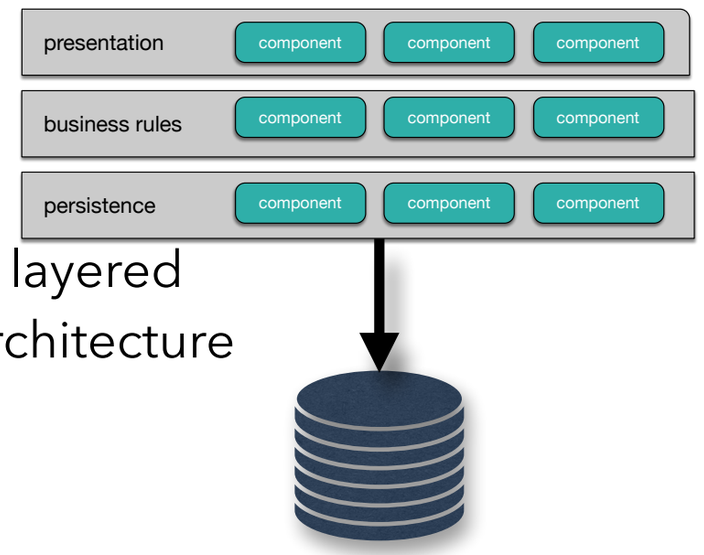
Monoliths

domain



modular monolith

technical



layered architecture

http://www.codingthearchitecture.com/presentations/sa2015-modular-monoliths

what?

how do we assess the current architecture?

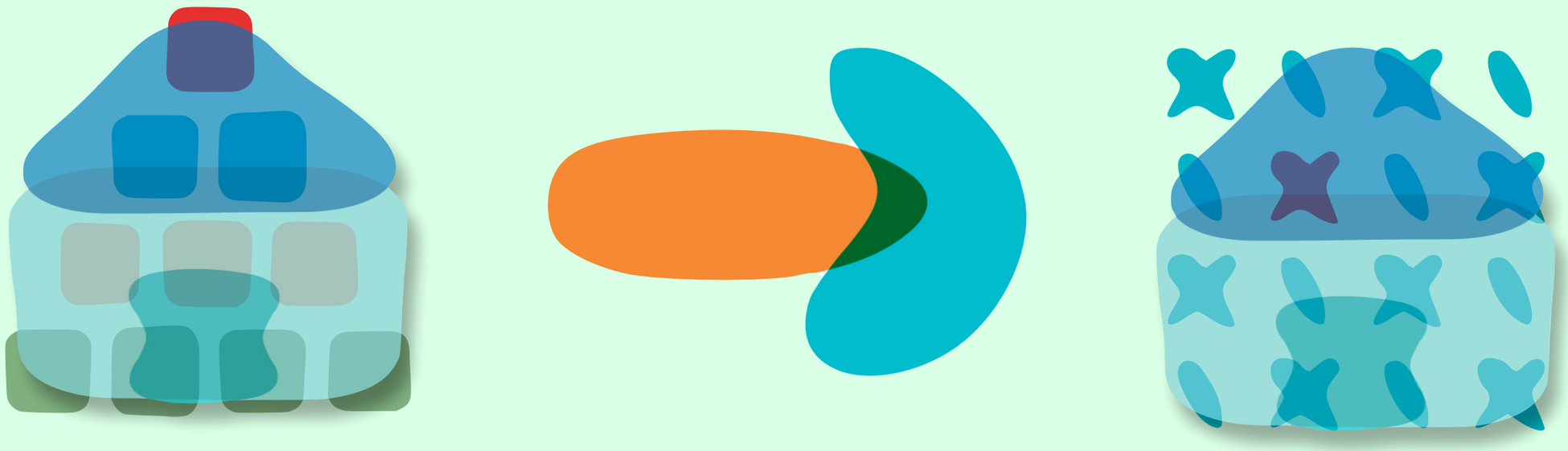
where?

what should we change it to?

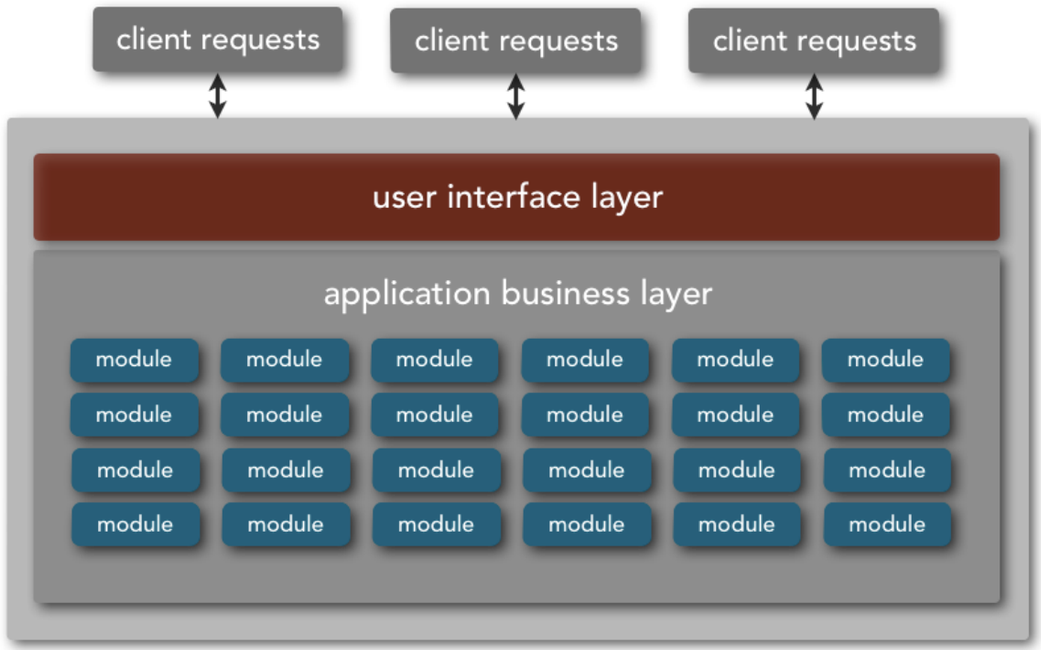
how?

how do we change it?

Migrating Architectures

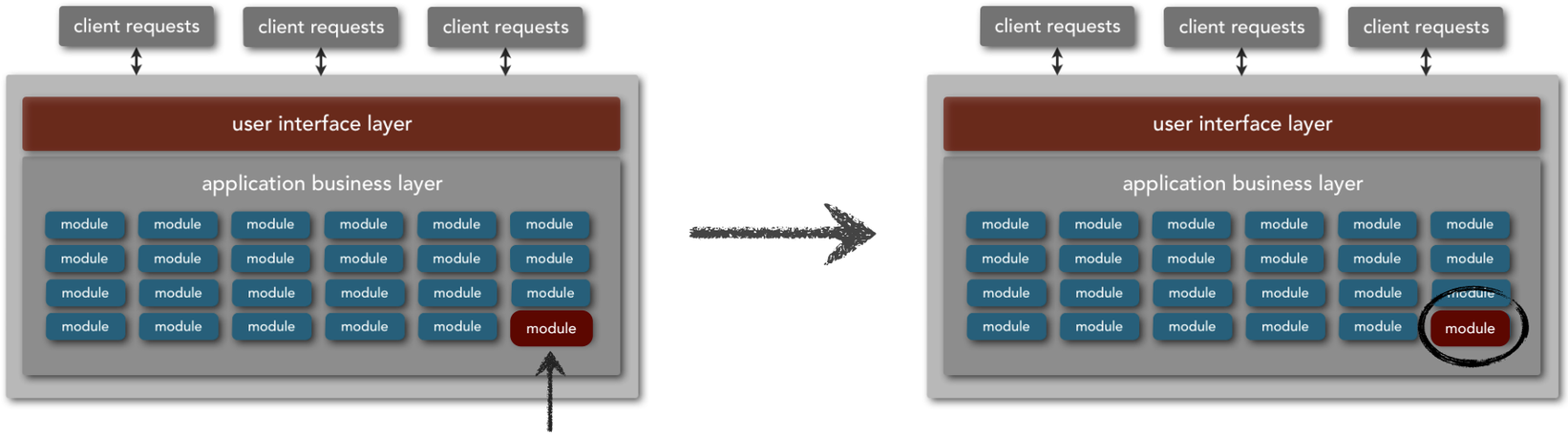


Monolithic Application Issues



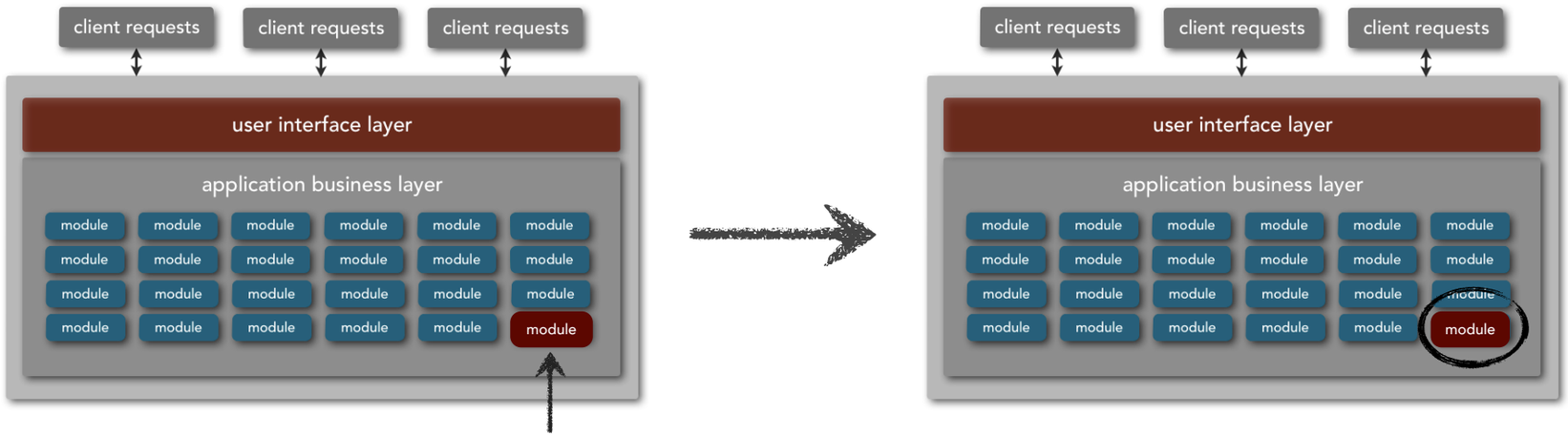
Monolithic Application Issues

deployment



Monolithic Application Issues

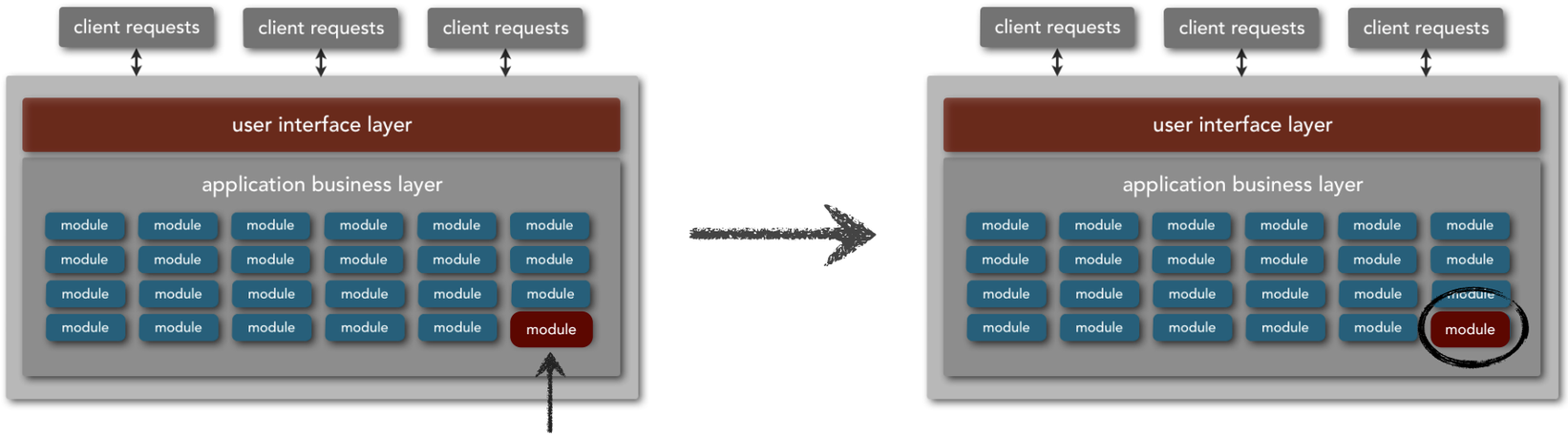
deployment



entire application must be deployed for a small change

Monolithic Application Issues

deployment

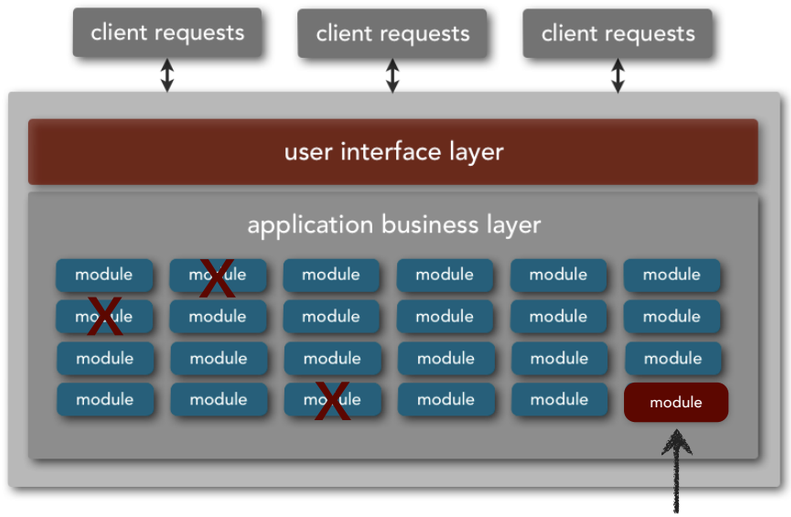


entire application must be deployed for a small change

scheduling and coordination challenges can impact deployment

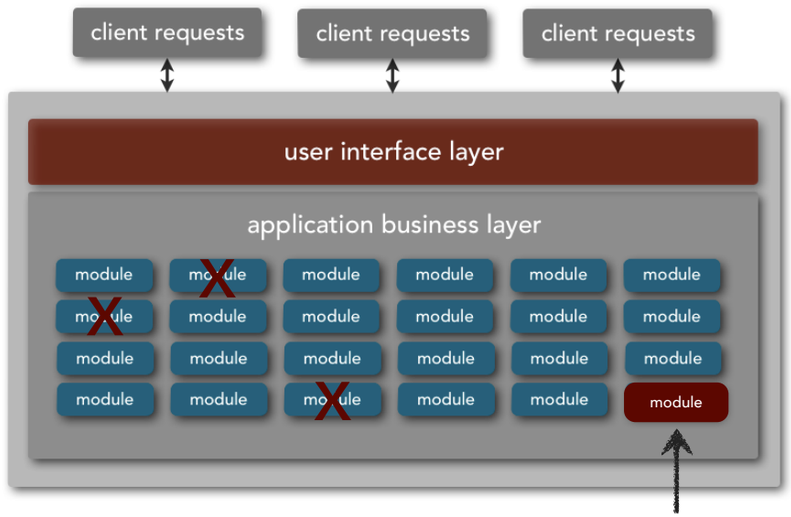
Monolithic Application Issues

reliability and robustness



Monolithic Application Issues

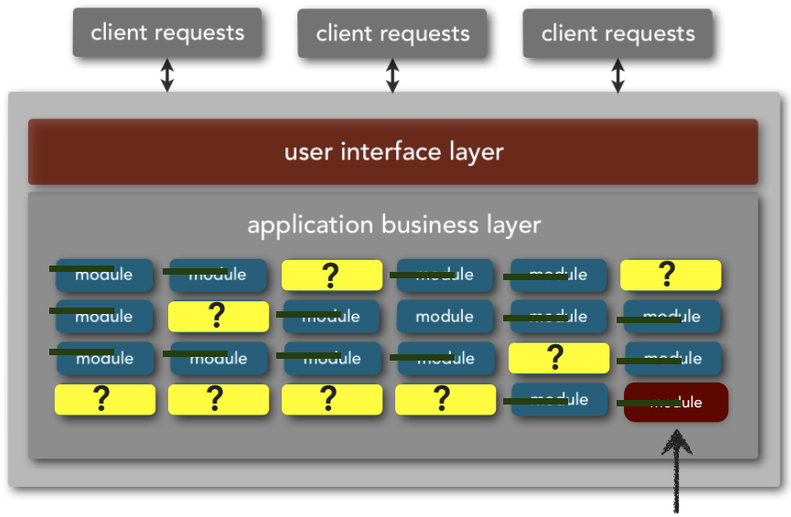
reliability and robustness



due to tight coupling, changes in one area of the application adversely affects other areas

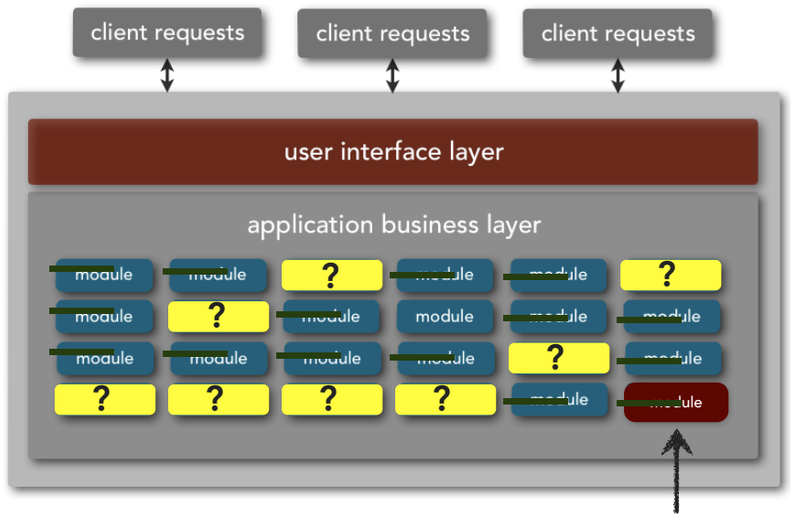
Monolithic Application Issues

testability



Monolithic Application Issues

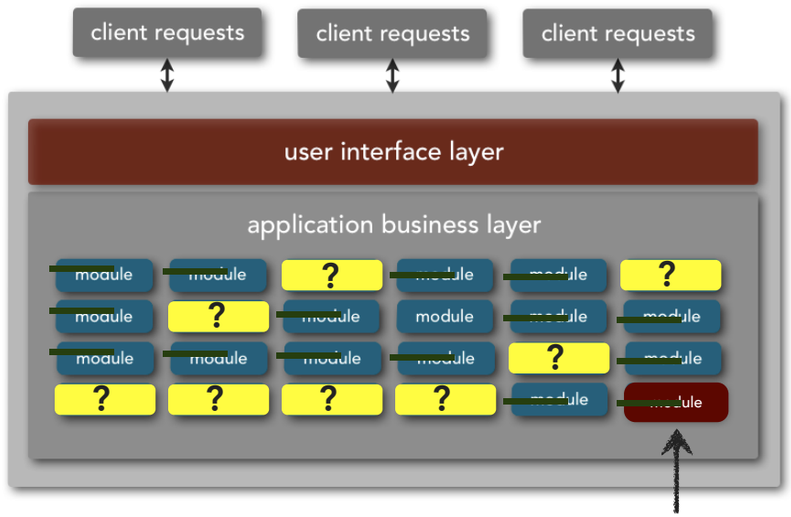
testability



poor test coverage for entire application

Monolithic Application Issues

testability

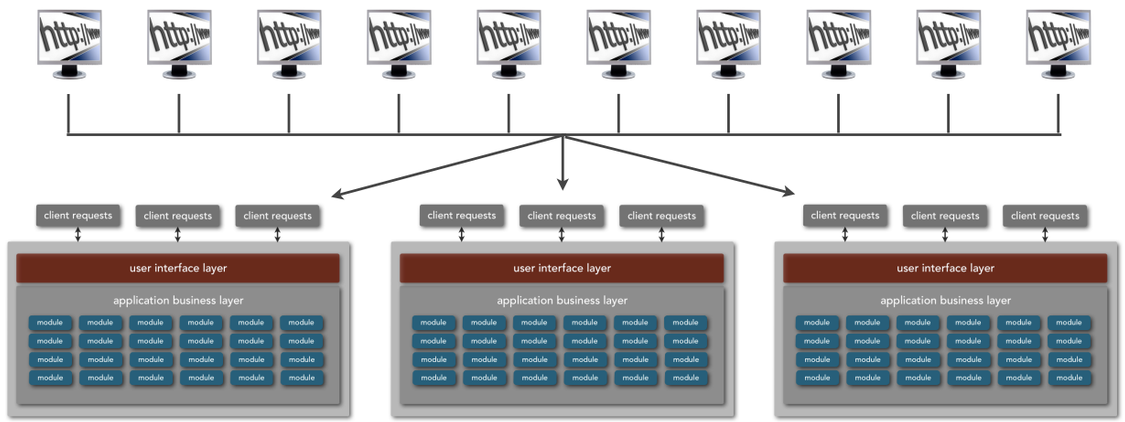


poor test coverage for entire application

low confidence level that nothing else is impacted by the change

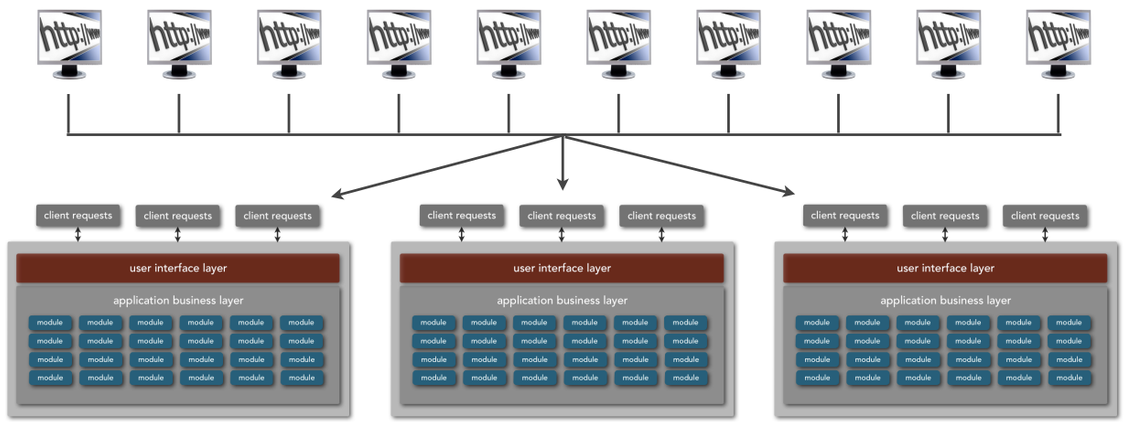
Monolithic Application Issues

scalability



Monolithic Application Issues

scalability



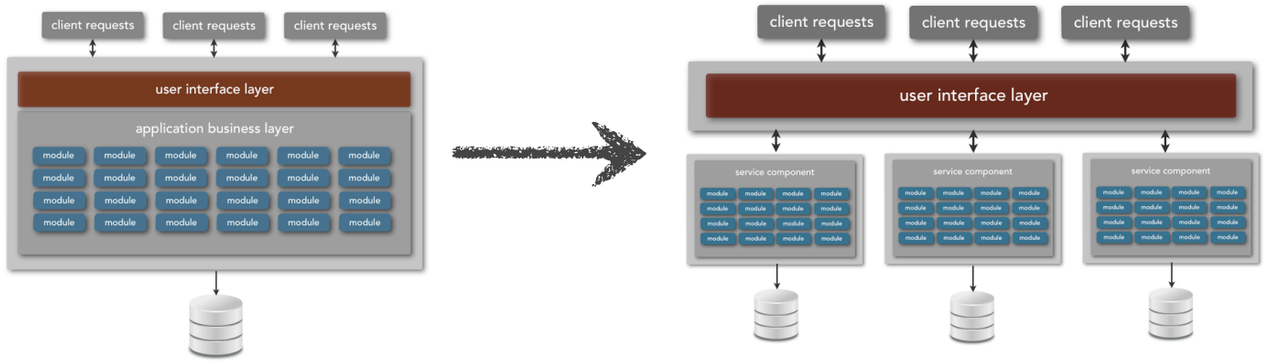
scaling monolithic applications requires you to scale the entire application, which can be both difficult and costly

Monolithic Application Issues

application size

monolithic applications can grow to quickly and consume all available resources

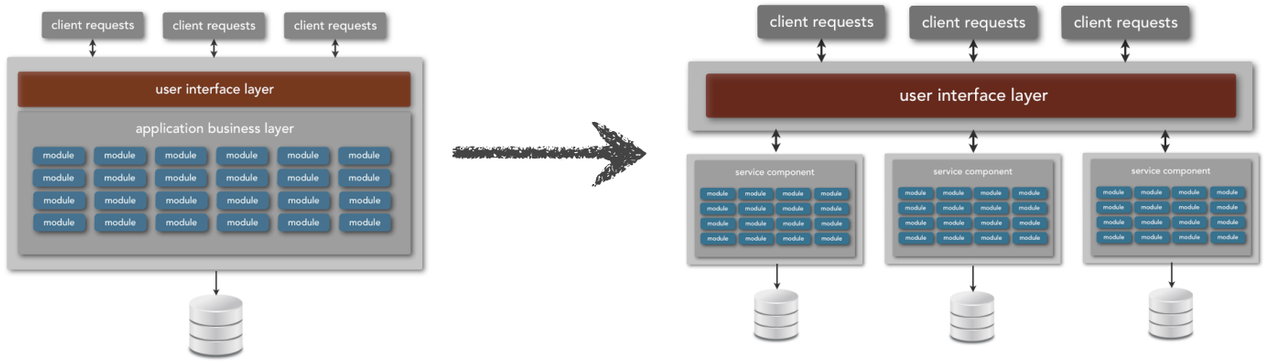
Migration Challenges



share everything architecture

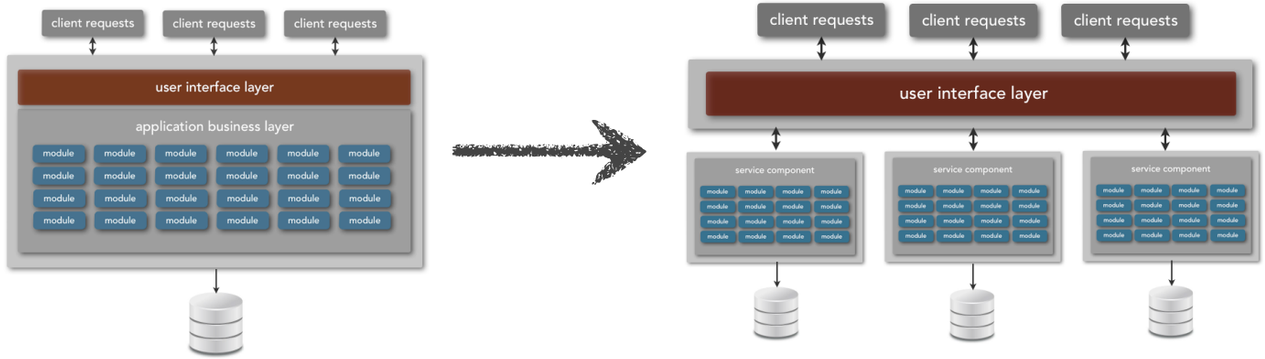
share as little as possible architecture

Migration Challenges

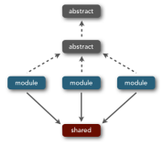


service component granularity and transactional boundaries

Migration Challenges

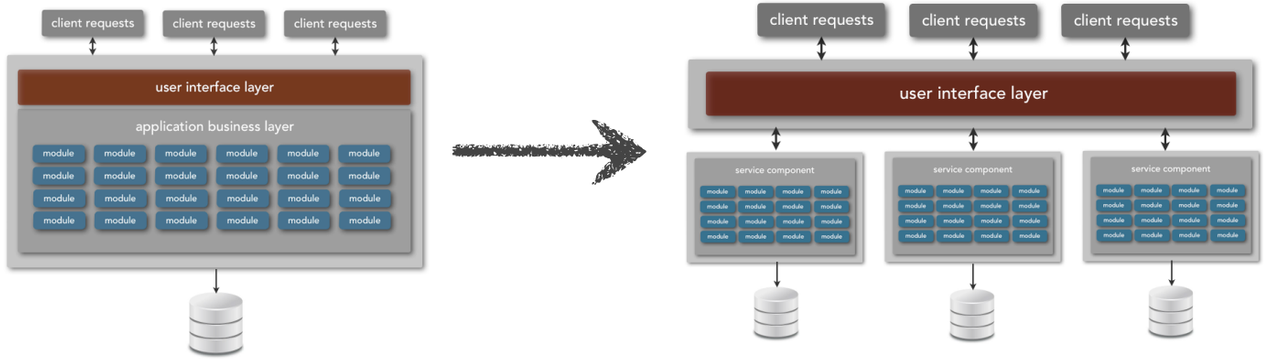


service component granularity and transactional boundaries

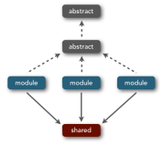


shared services, modules, and object hierarchies

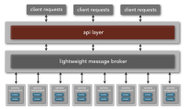
Migration Challenges



service component granularity and transactional boundaries

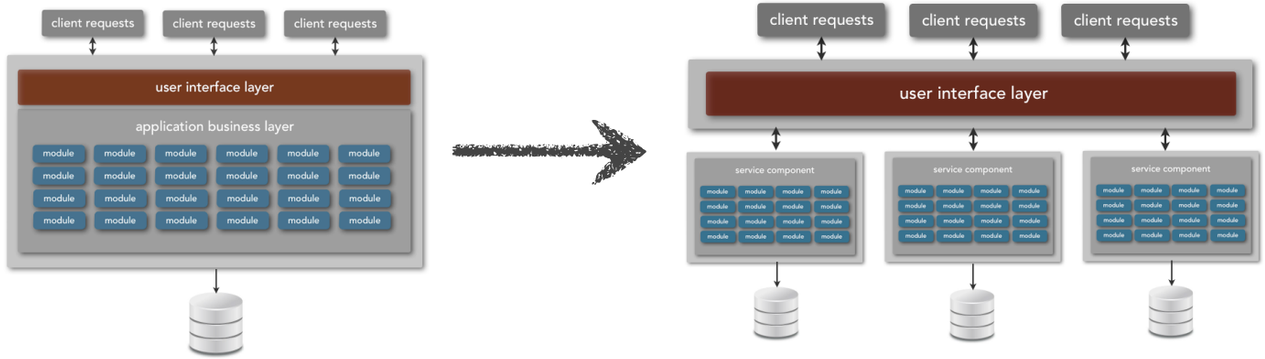


shared services, modules, and object hierarchies

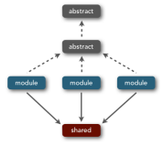


distributed architecture and remote service issues

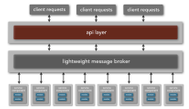
Migration Challenges



service component granularity and transactional boundaries



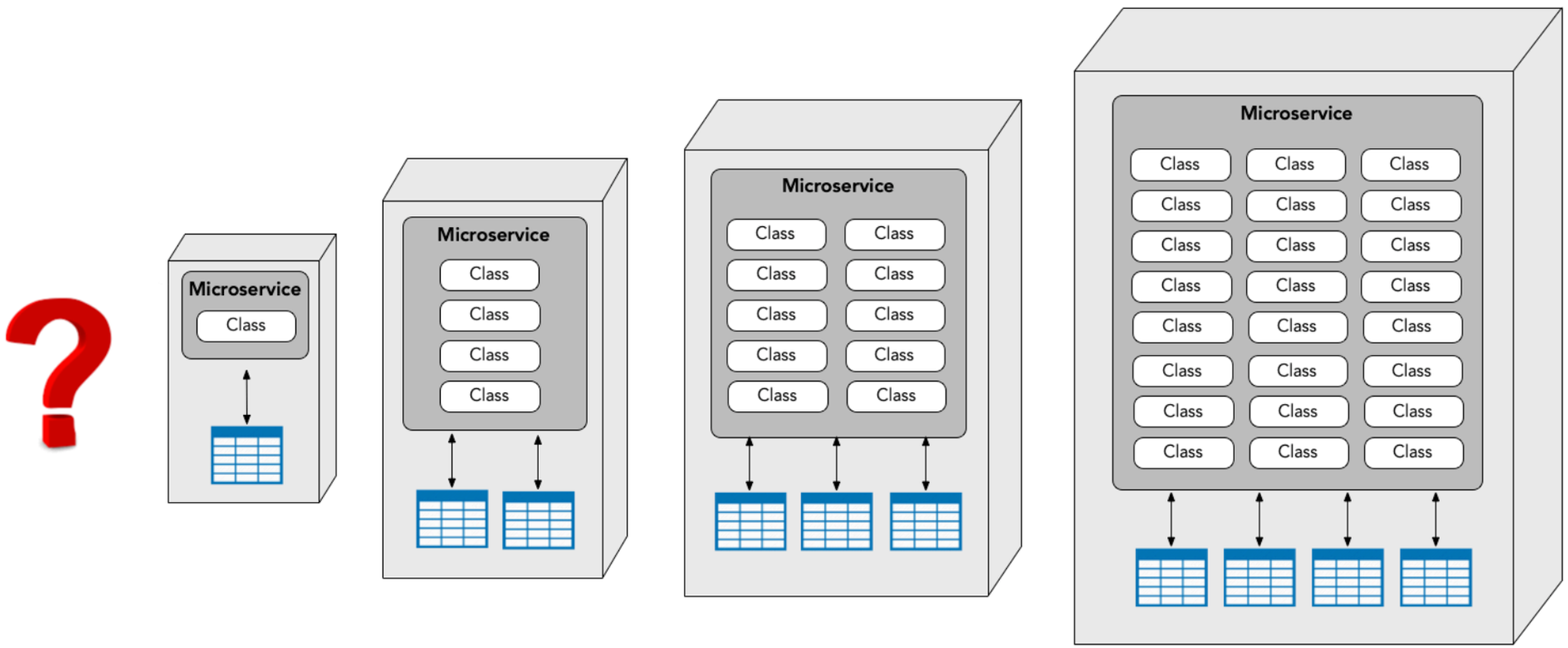
shared services, modules, and object hierarchies



distributed architecture and remote service issues

large shared relational database

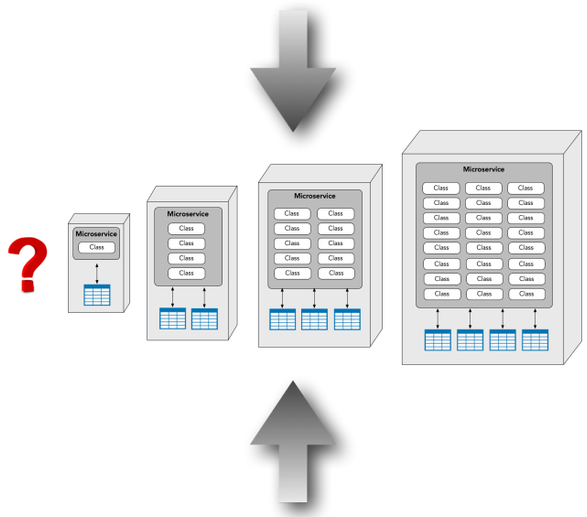
what is the right level of granularity for a service?



service granularity

granularity disintegrators

"when should I consider breaking apart a service?"



granularity integrators

"when should I consider putting services back together?"

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

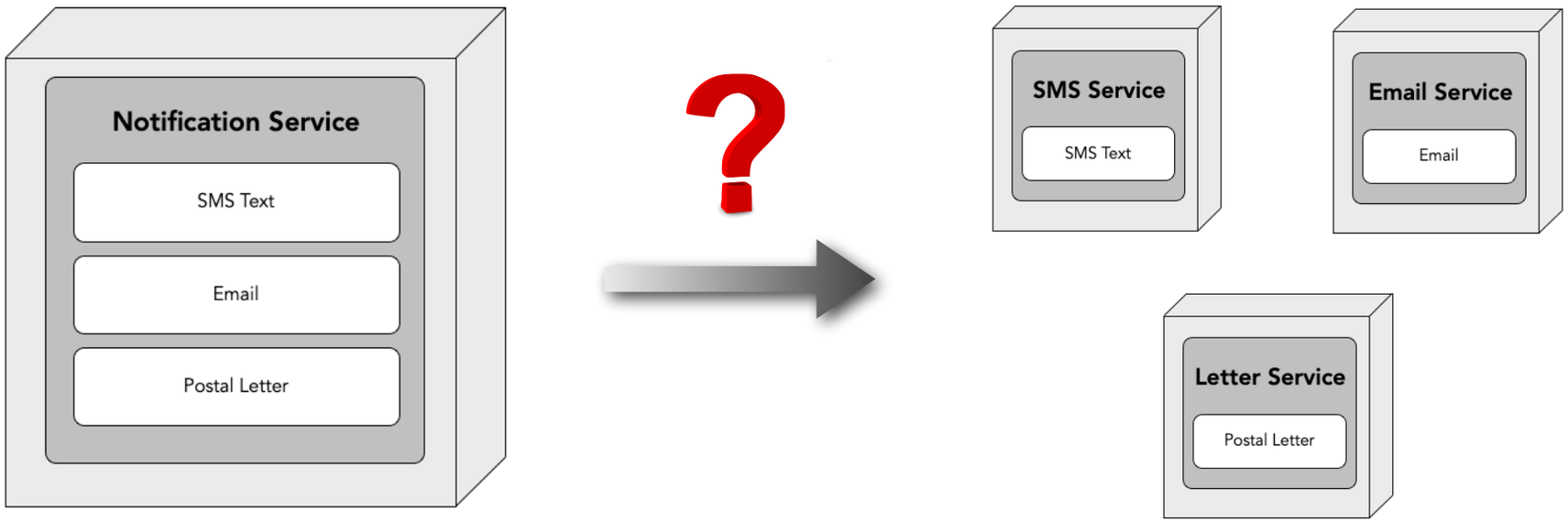
service
functionality

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service functionality

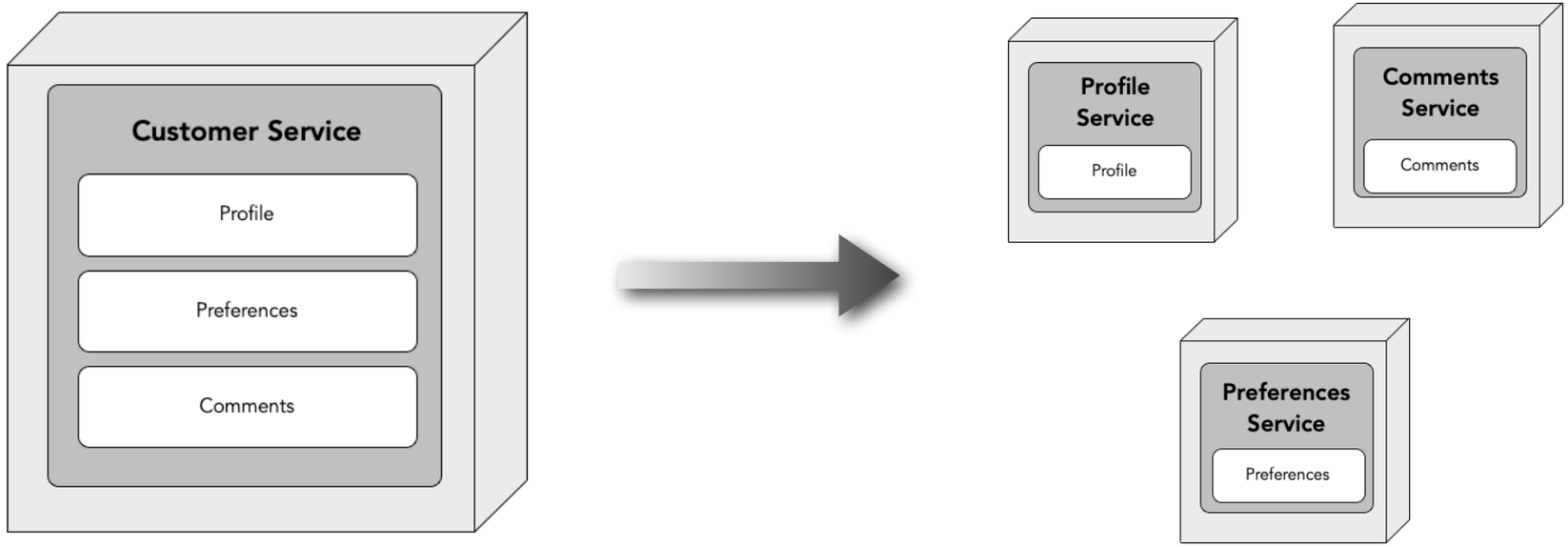


service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service functionality

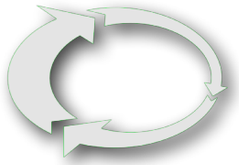


service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service
functionality

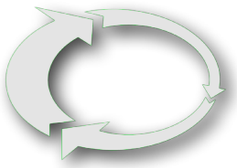


code
volatility

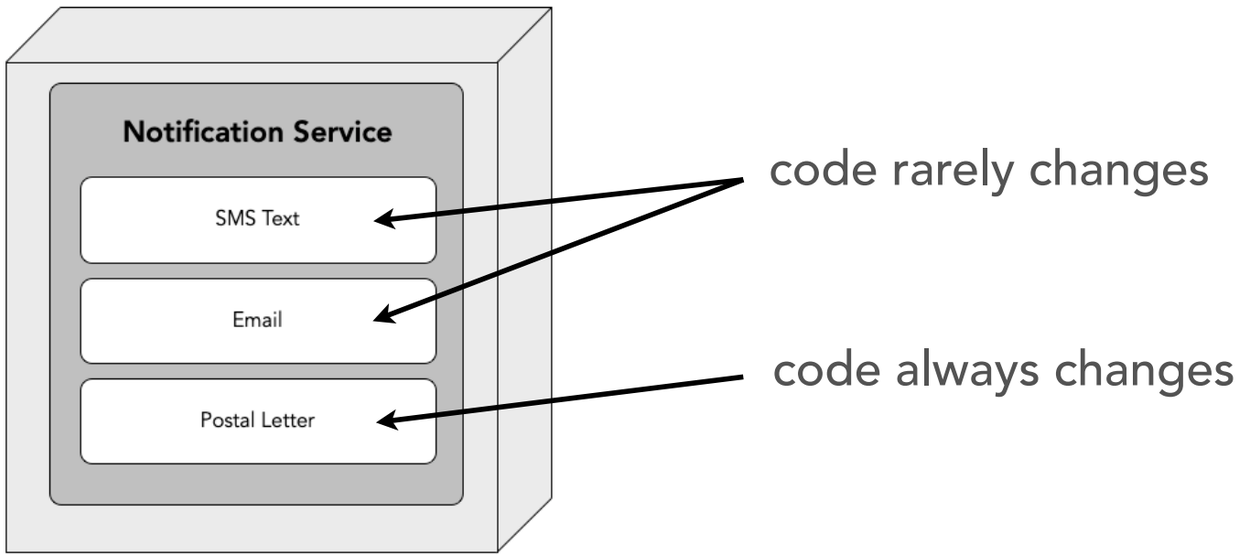
service granularity

granularity disintegrators

"when should I consider breaking apart a service?"



code
volatility

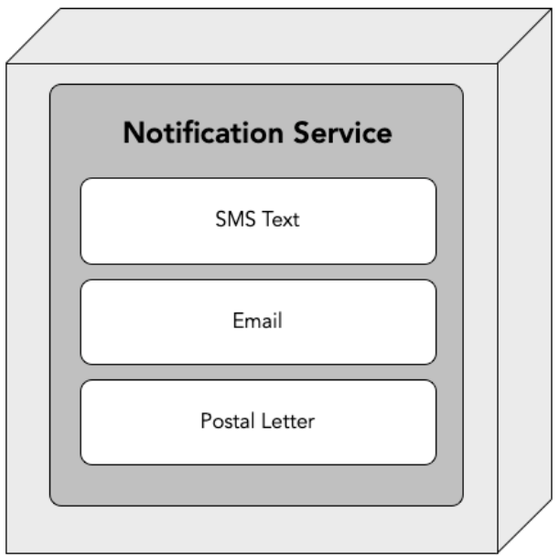


service granularity

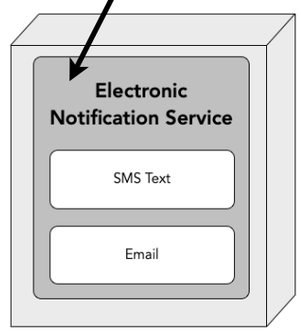
granularity disintegrators

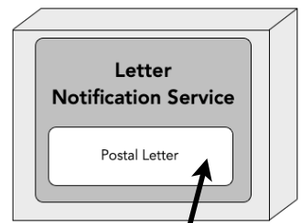
"when should I consider breaking apart a service?"

code volatility



code rarely changes





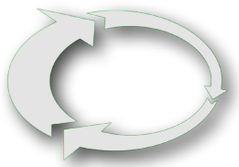
code always changes

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service functionality



code volatility

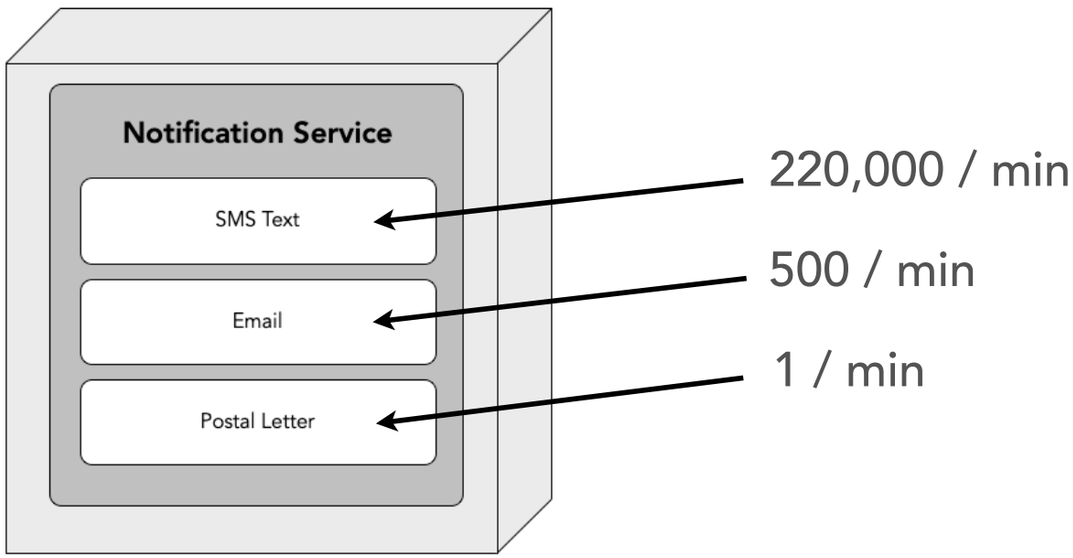
scalability and throughput

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

scalability and throughput

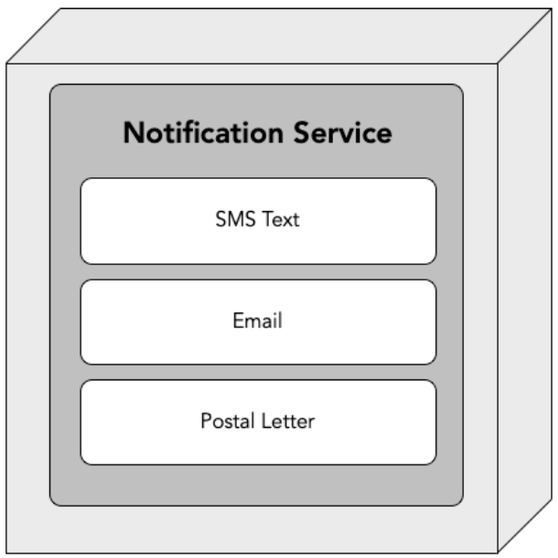


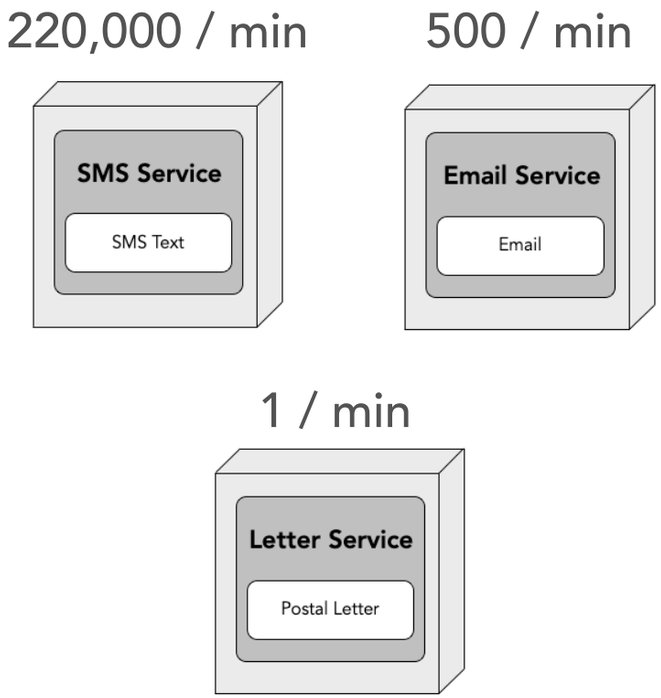
service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

scalability and throughput



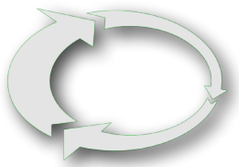


service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service functionality



code volatility

scalability and throughput

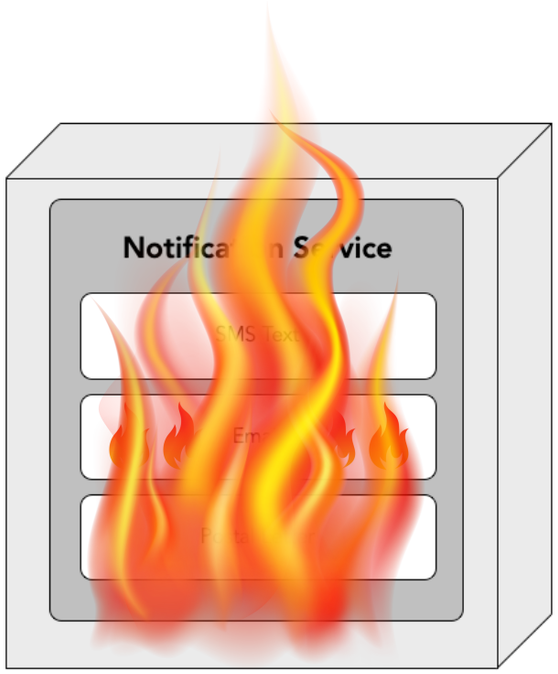
fault tolerance

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

fault
tolerance

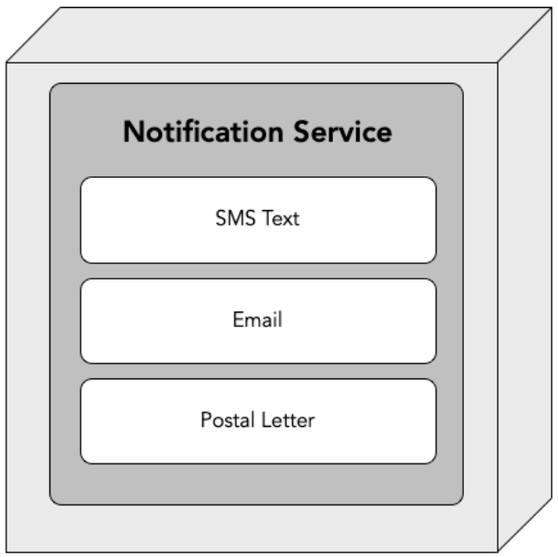


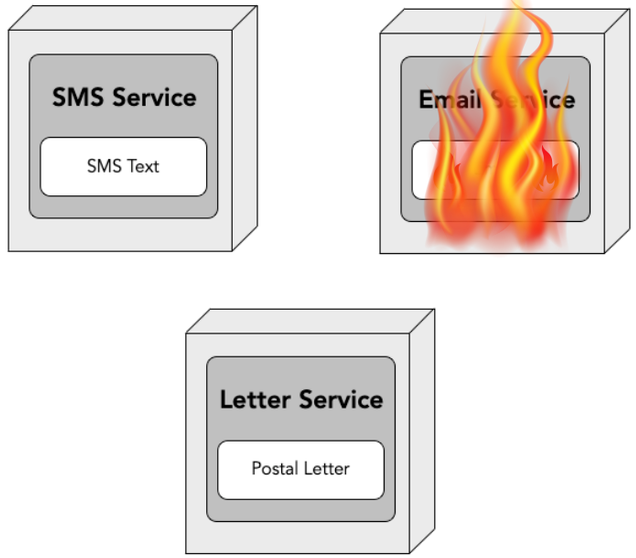
service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

fault tolerance



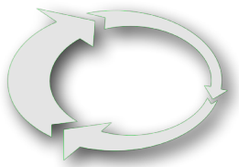


service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service functionality



code volatility

scalability and throughput

fault tolerance

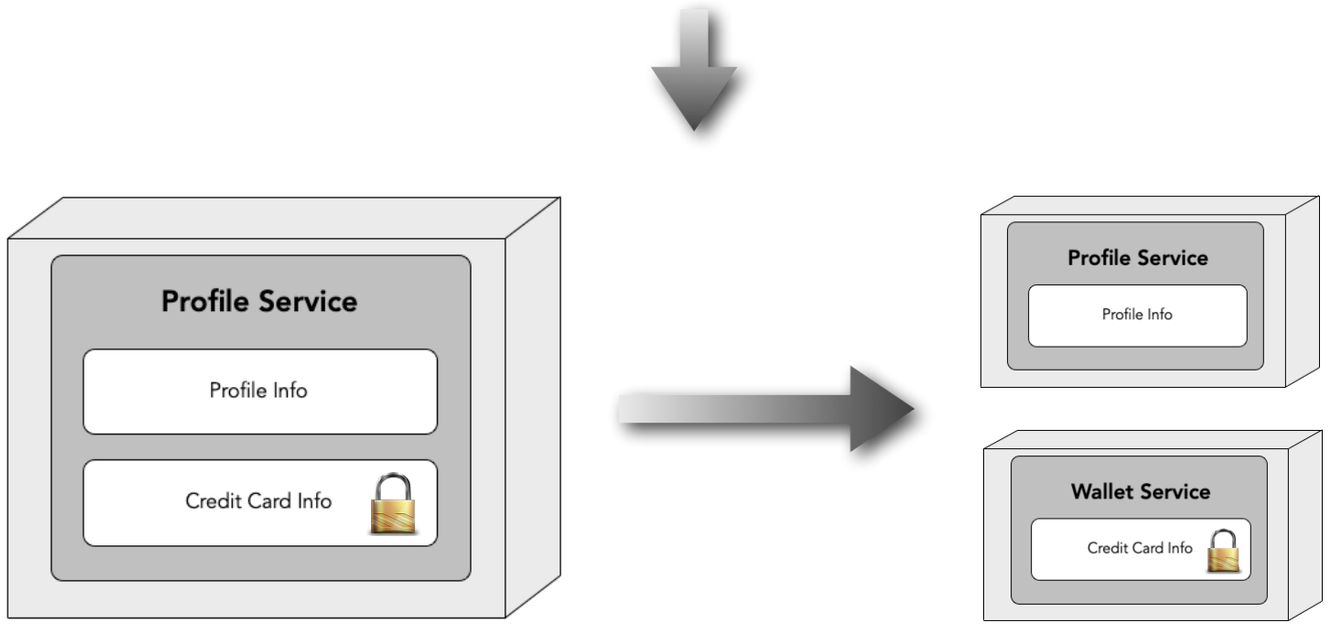
data security

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

data
security

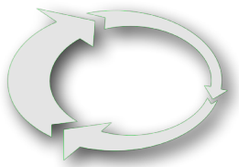


service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

service functionality



code volatility

scalability and throughput

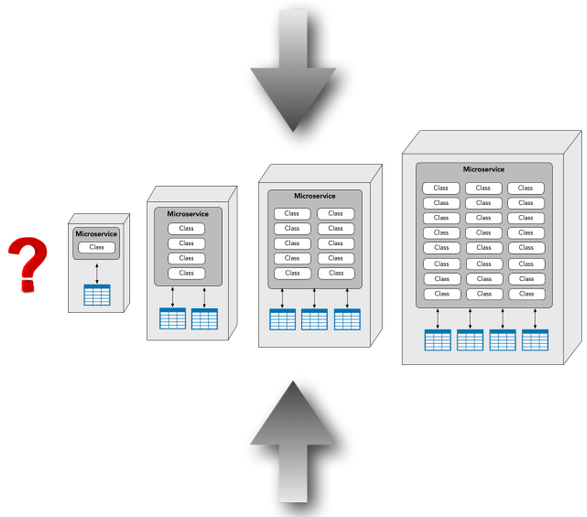
fault tolerance

data security

service granularity

granularity disintegrators

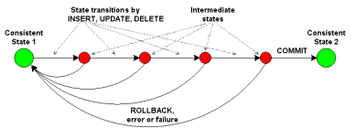
"when should I consider breaking apart a service?"



granularity integrators

"when should I consider putting services back together?"

service granularity

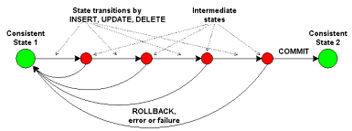


database
transactions

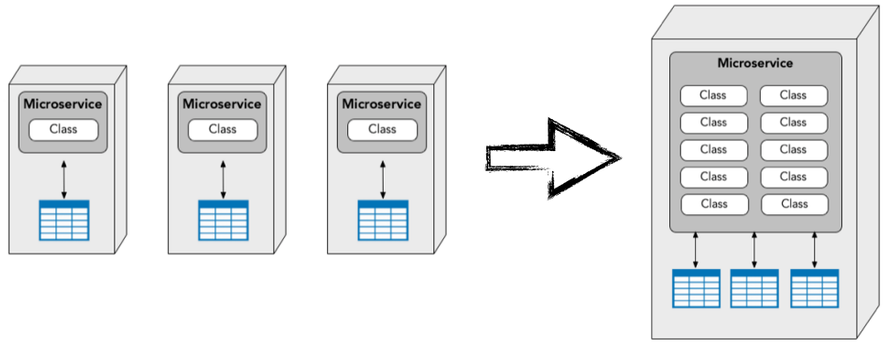
granularity integrators

"when should I consider putting services back together?"

service granularity

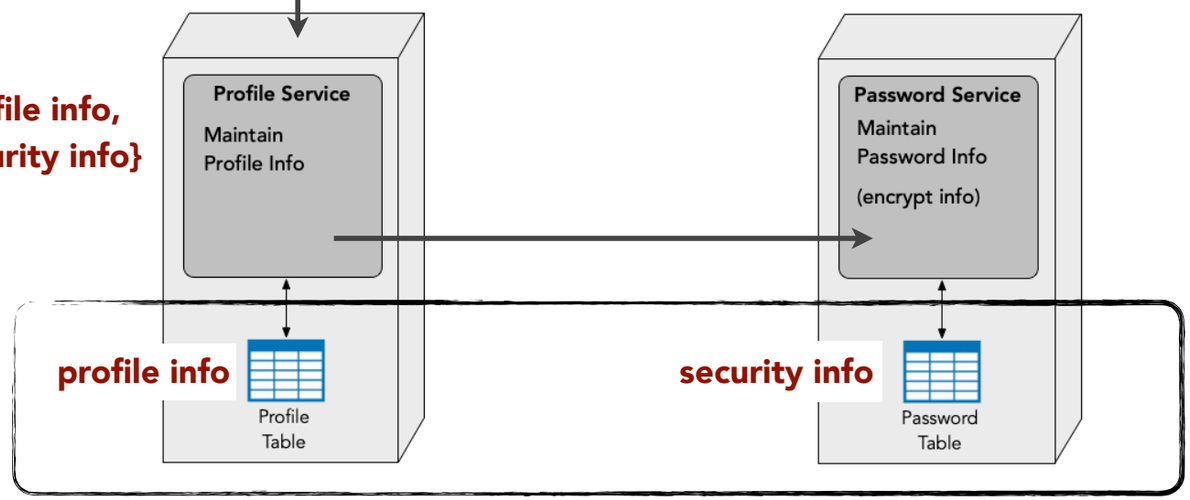


database
transactions



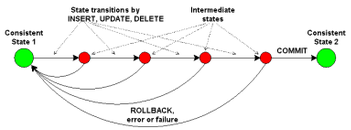
"register a new customer"

{profile info, security info}



no acid (db)
transaction

service granularity

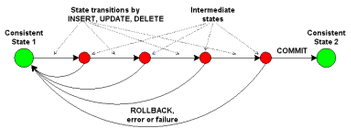


database
transactions

granularity integrators

"when should I consider putting services back together?"

service granularity



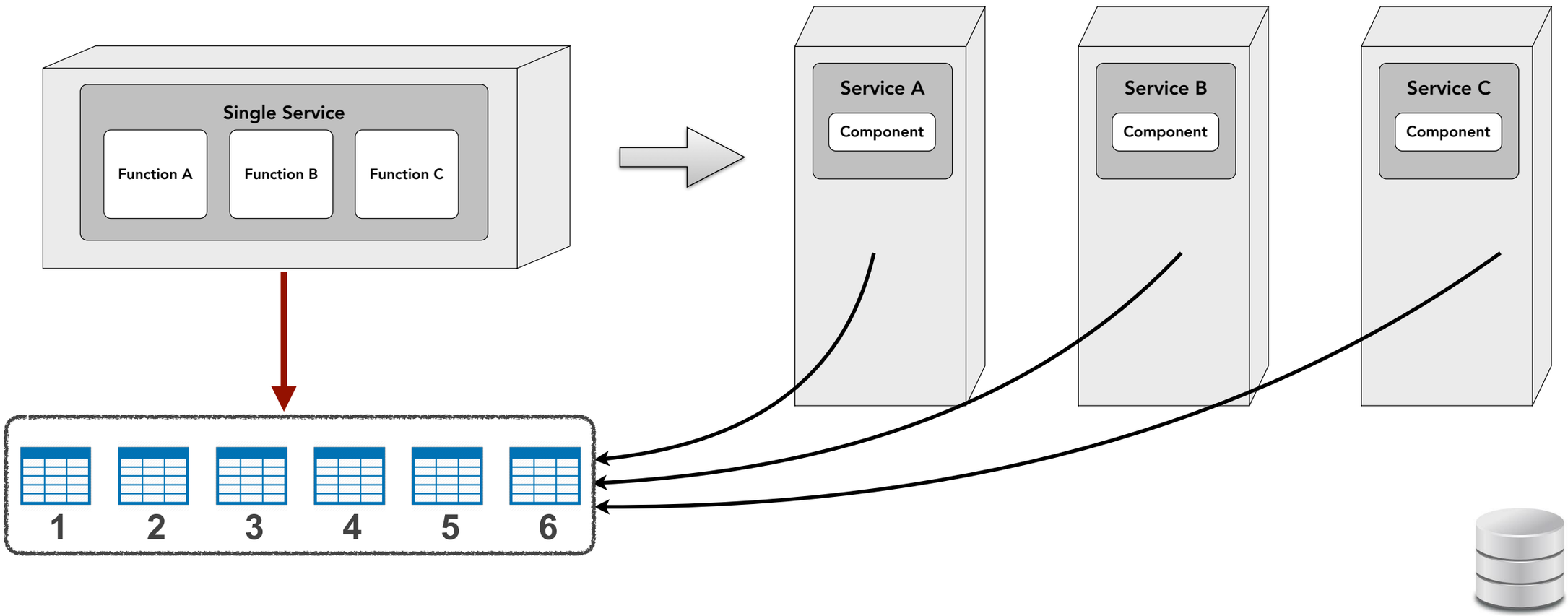
database
transactions

data
dependencies

granularity integrators

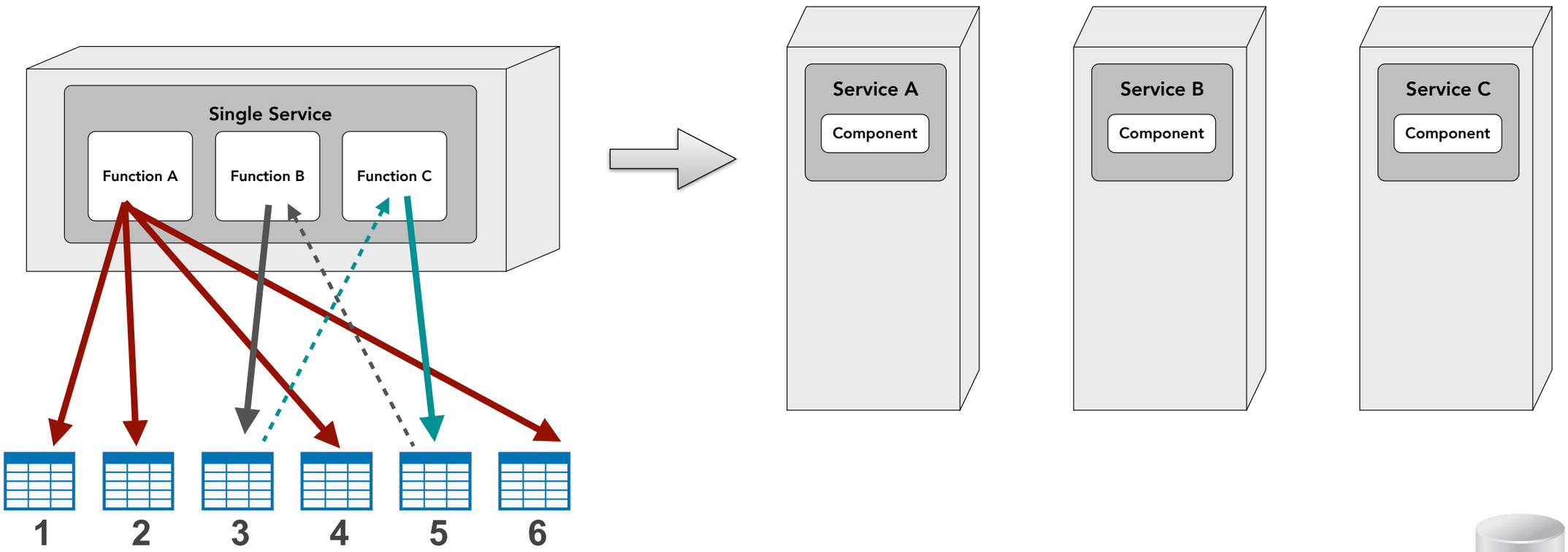
"when should I consider putting services back together?"

service granularity



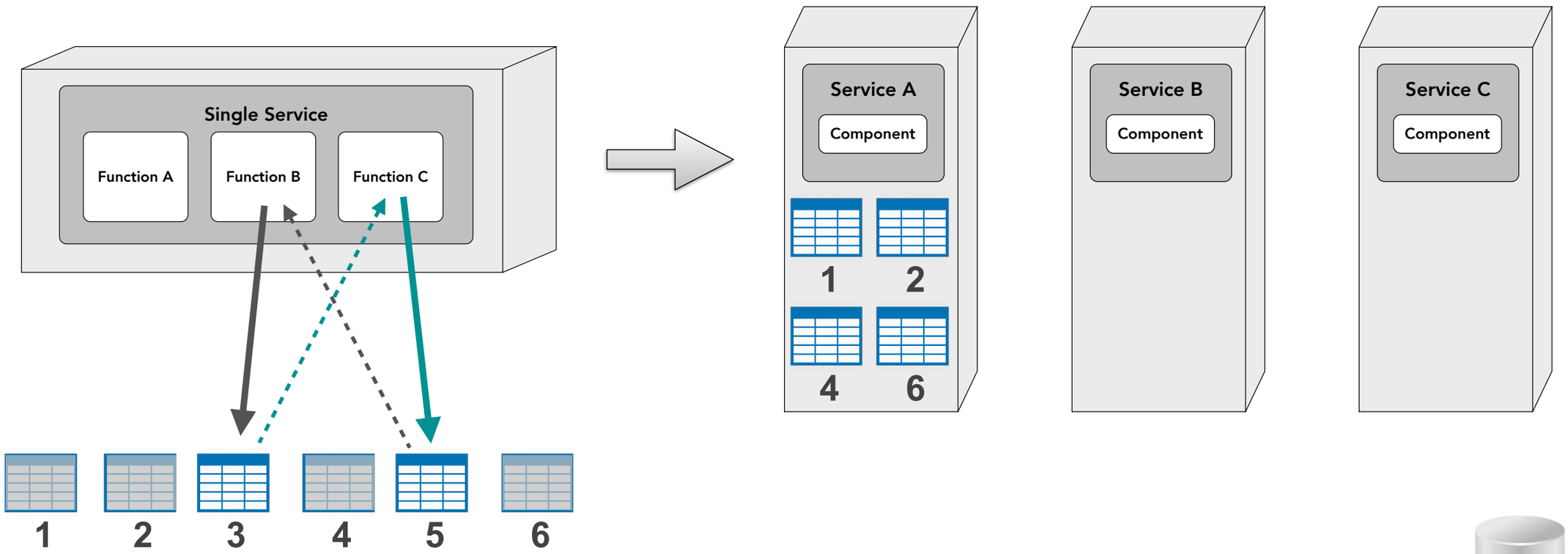
data dependencies

service granularity



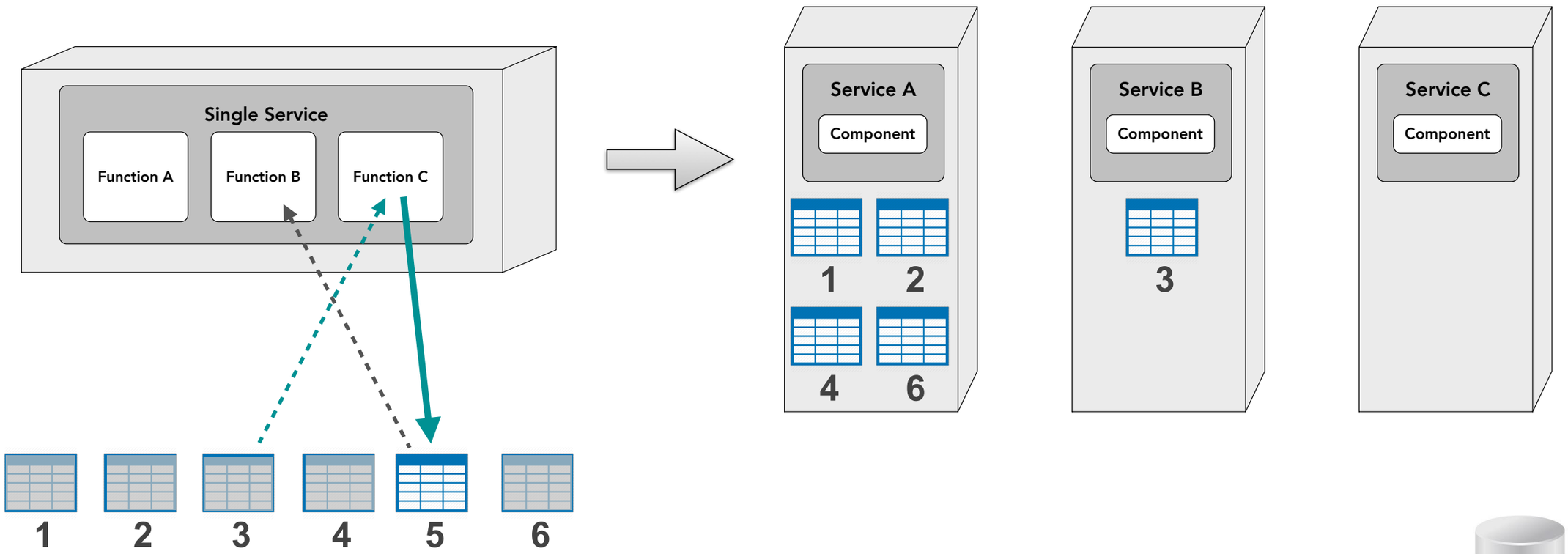
data dependencies

service granularity



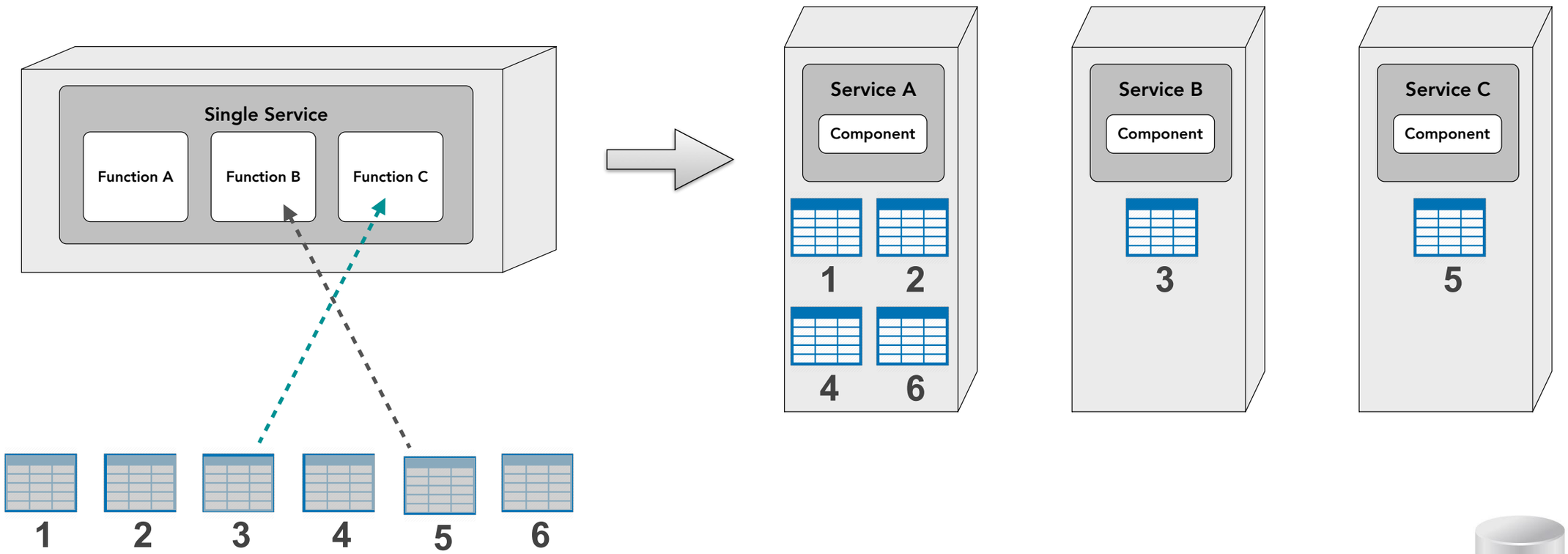
data dependencies

service granularity



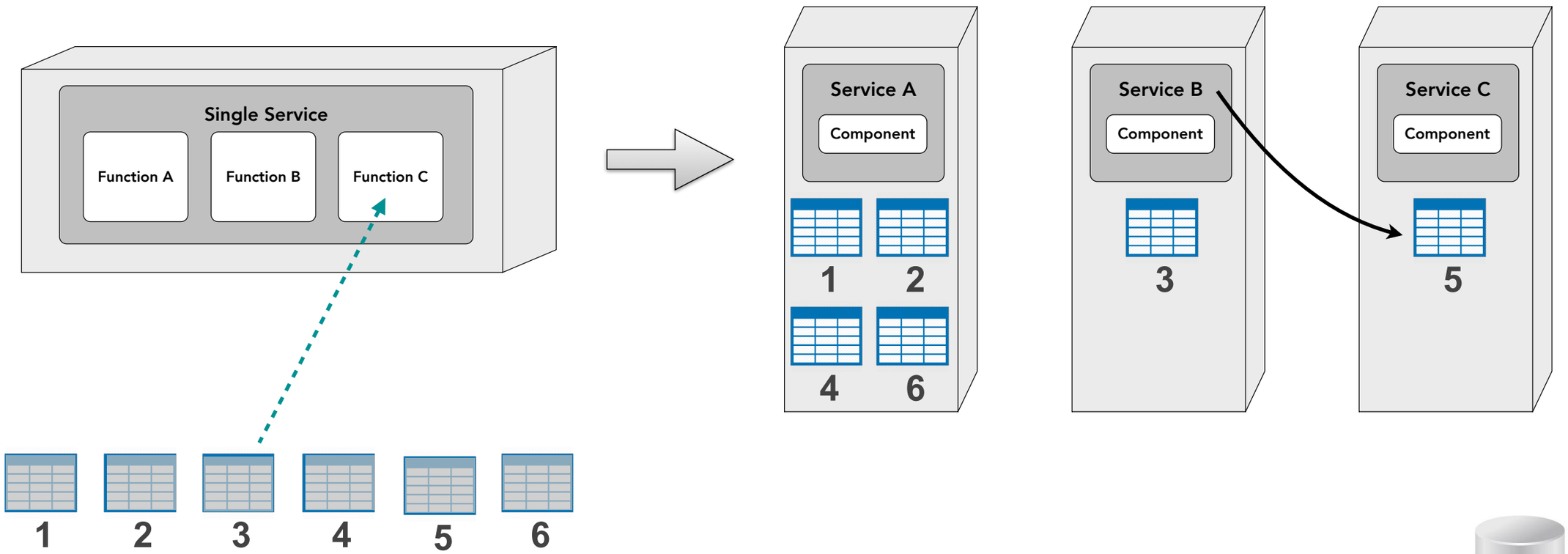
data dependencies

service granularity



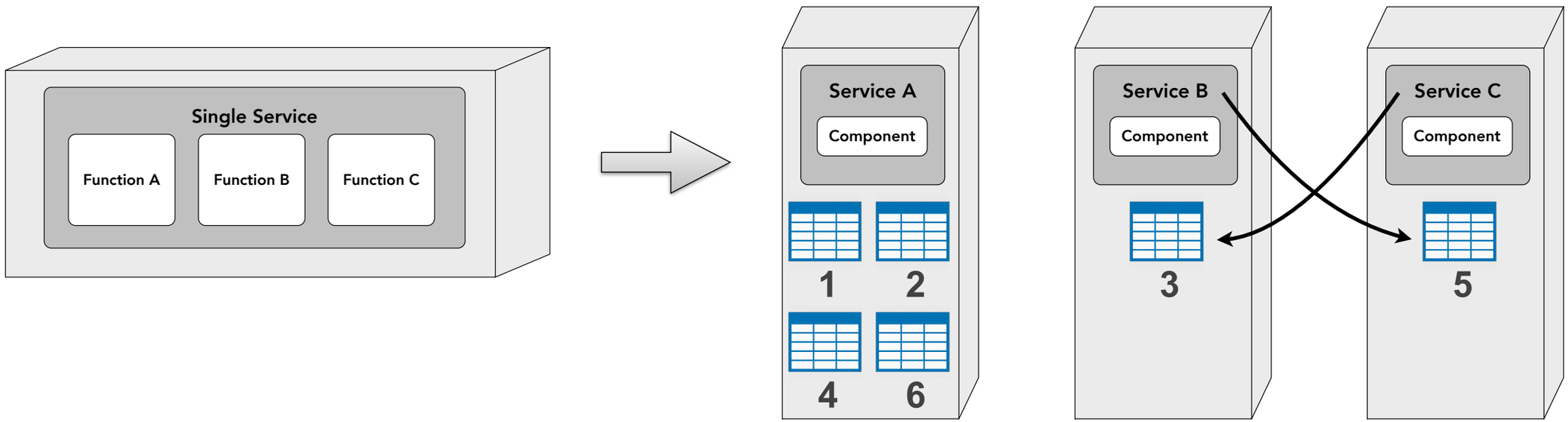
data dependencies

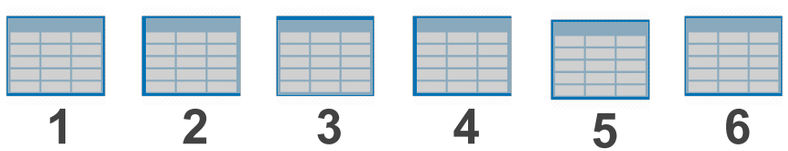
service granularity



data dependencies

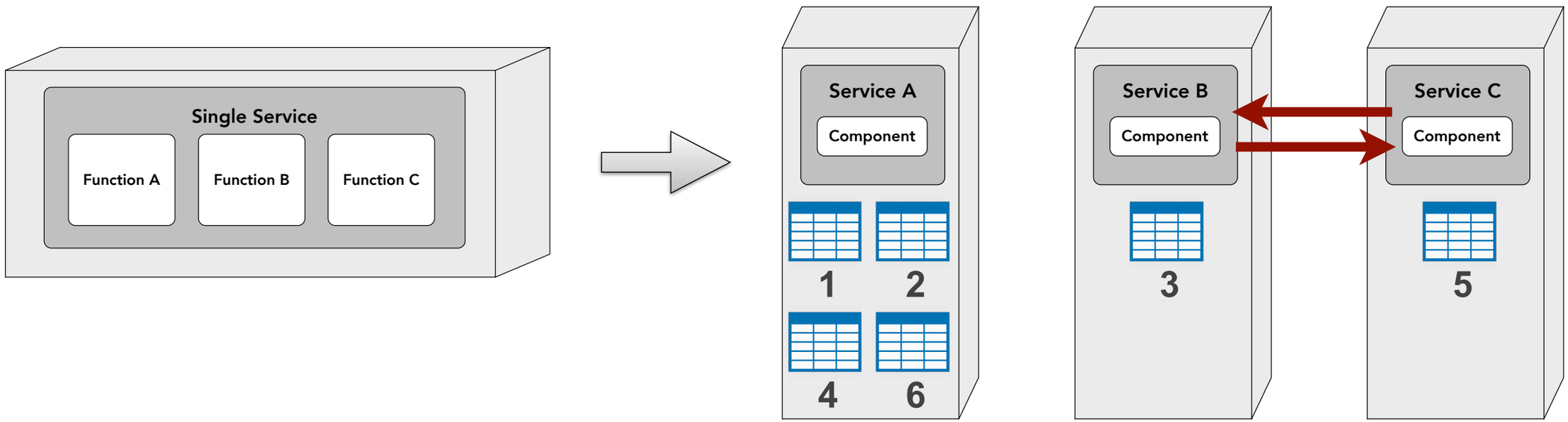
service granularity

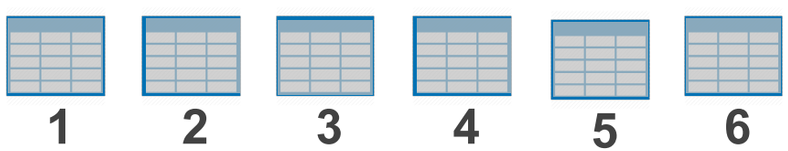




data dependencies

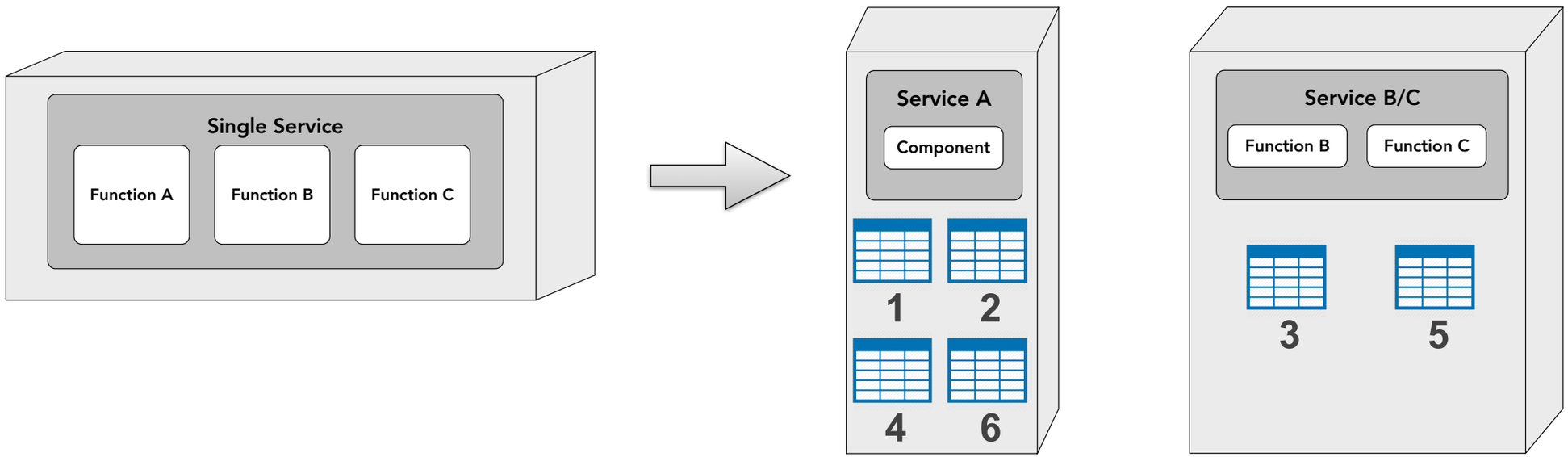
service granularity

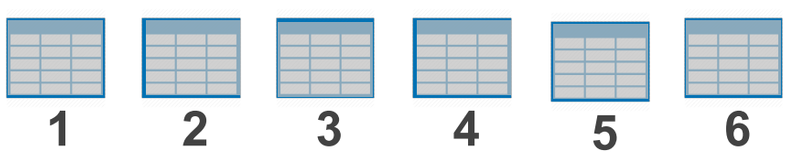




data dependencies

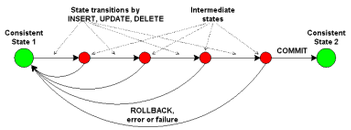
service granularity





data dependencies

service granularity



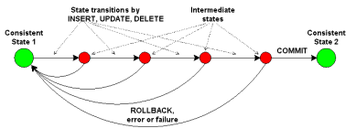
database
transactions

data
dependencies

granularity integrators

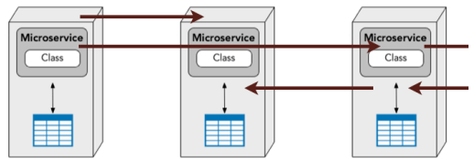
"when should I consider putting services back together?"

service granularity



database
transactions

data
dependencies

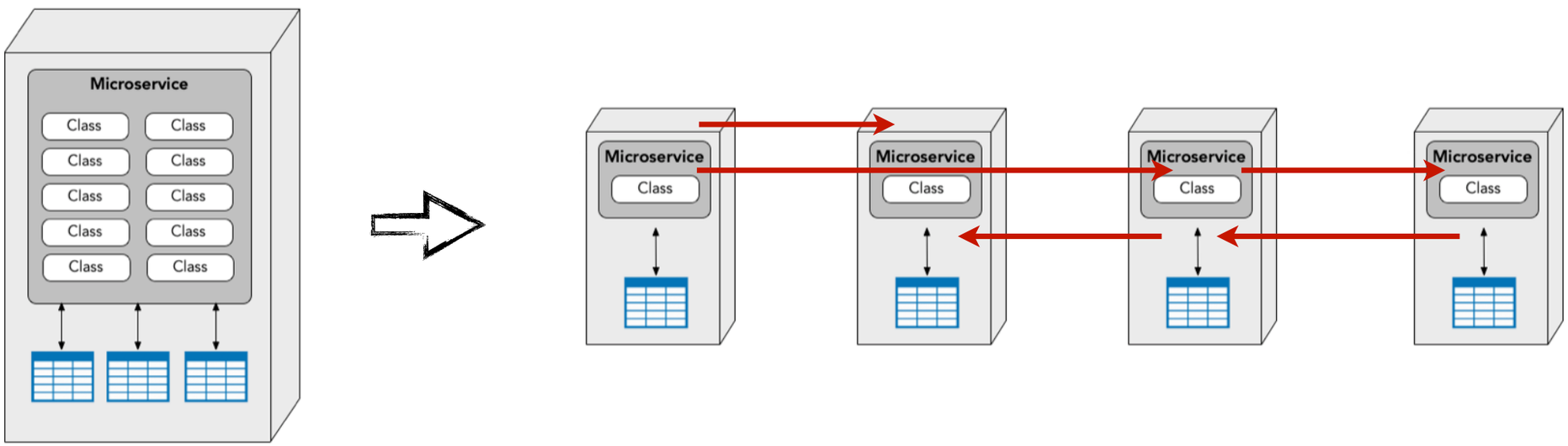


workflow and
choreography

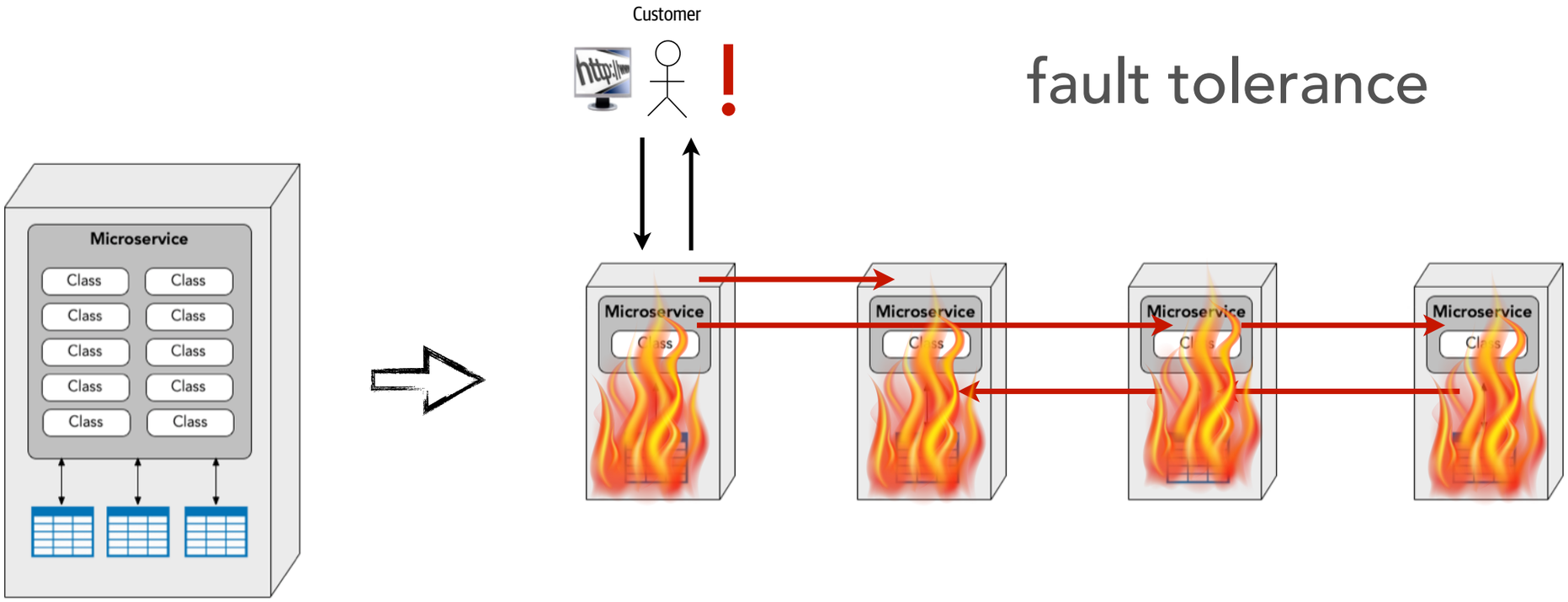
granularity integrators

"when should I consider putting services back together?"

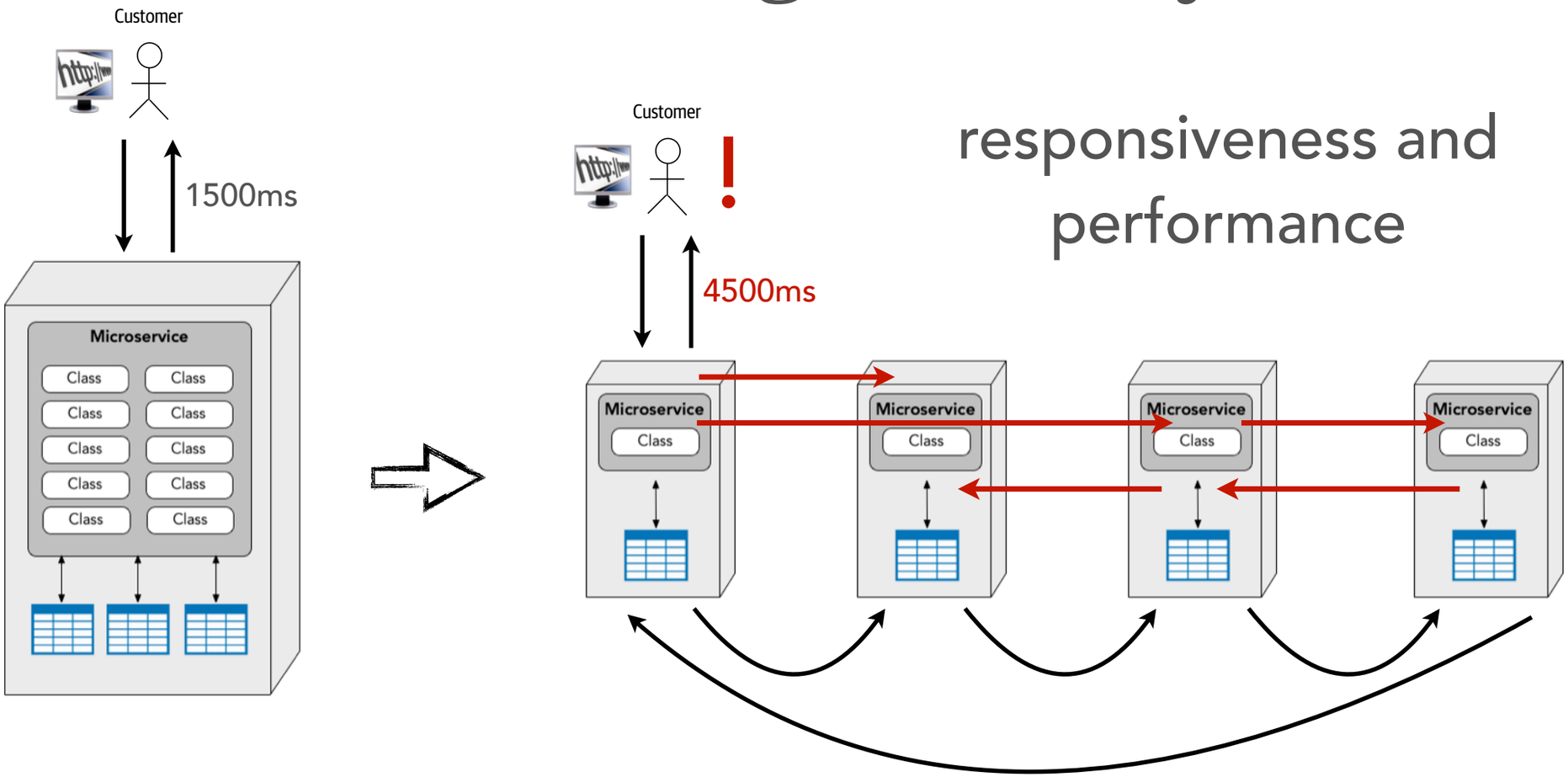
service granularity



service granularity

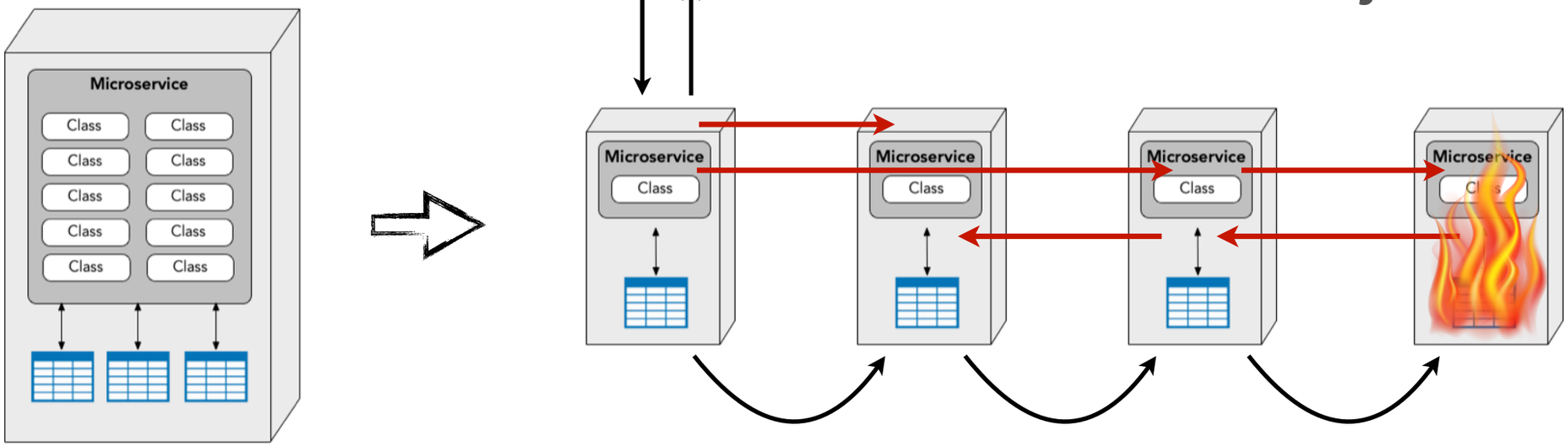


service granularity



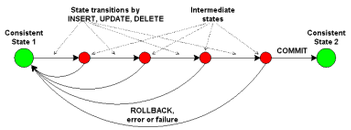
responsiveness and performance

service granularity



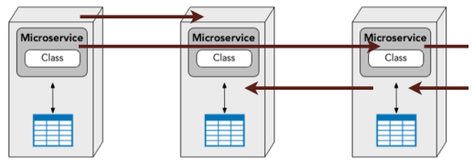
reliability and data consistency

service granularity



database
transactions

data
dependencies



workflow and
choreography

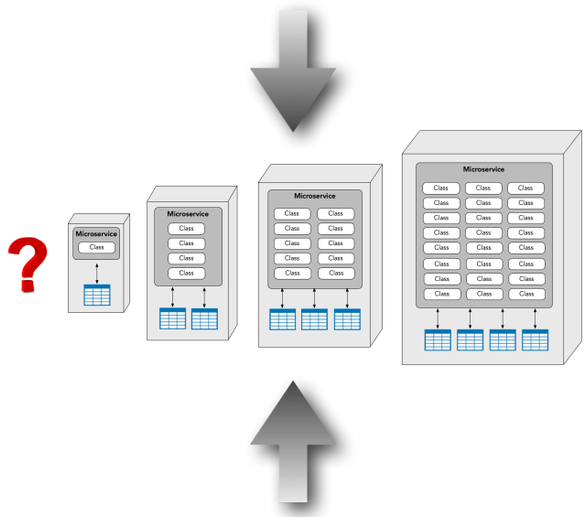
granularity integrators

"when should I consider putting services back together?"

service granularity

granularity disintegrators

"when should I consider breaking apart a service?"

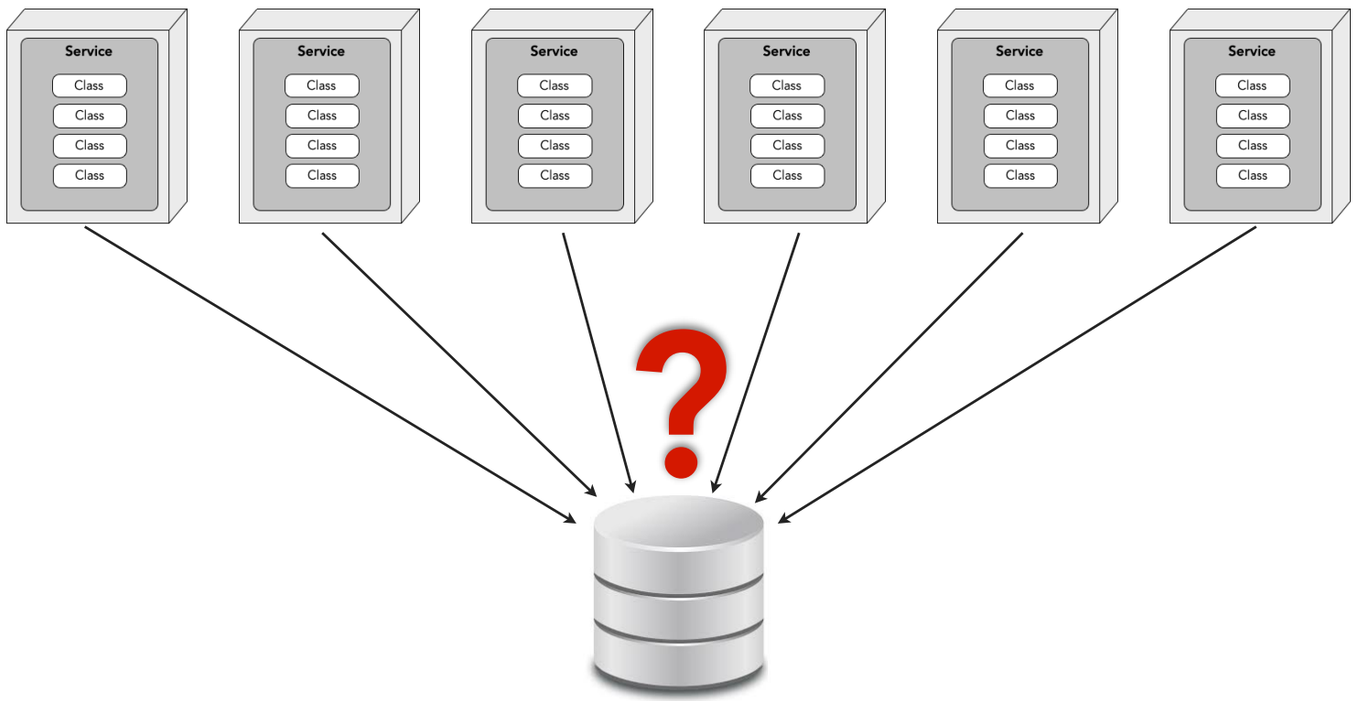


granularity integrators

"when should I consider putting services back together?"

breaking apart data

"when should I consider breaking apart my data?"



breaking apart data

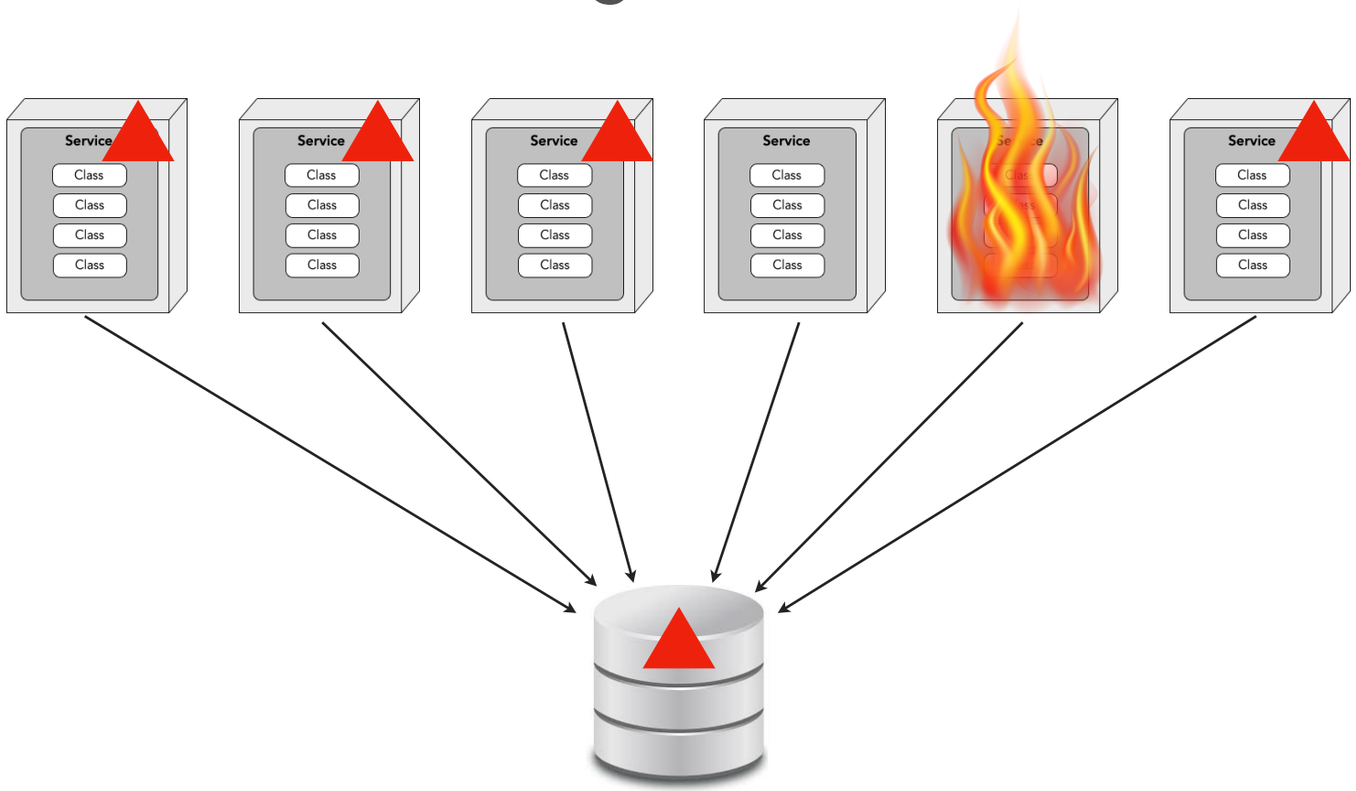
“when should I consider breaking apart my data?”

database granularity drivers

change
control

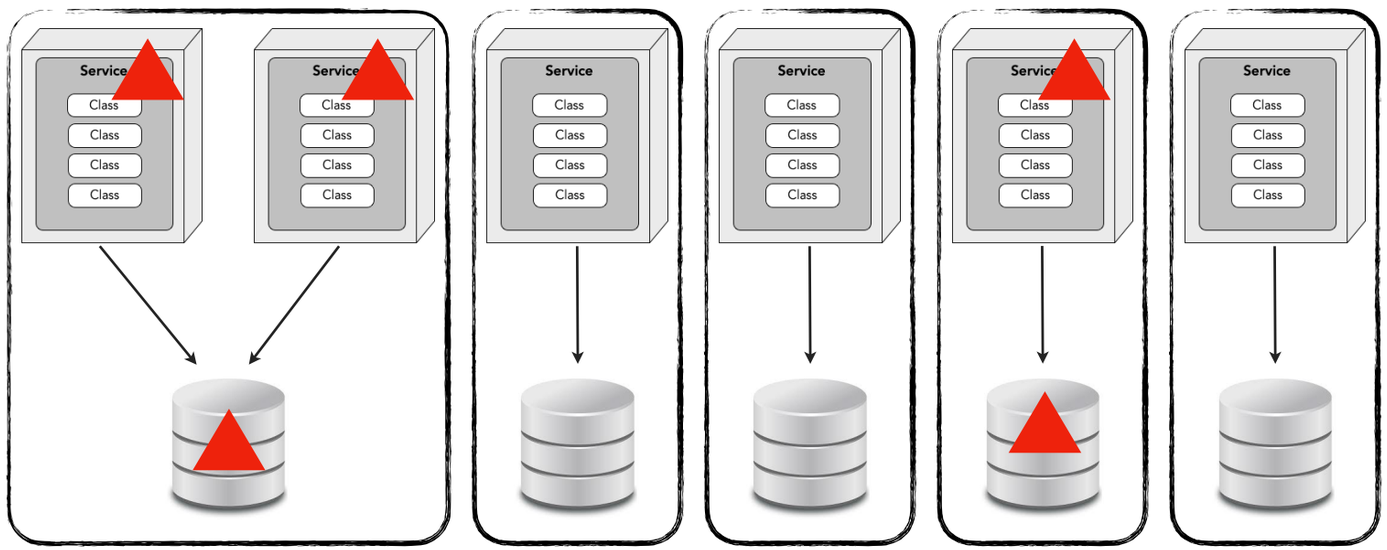
breaking apart data

change control



breaking apart data

change control



breaking apart data

"when should I consider breaking apart my data?"

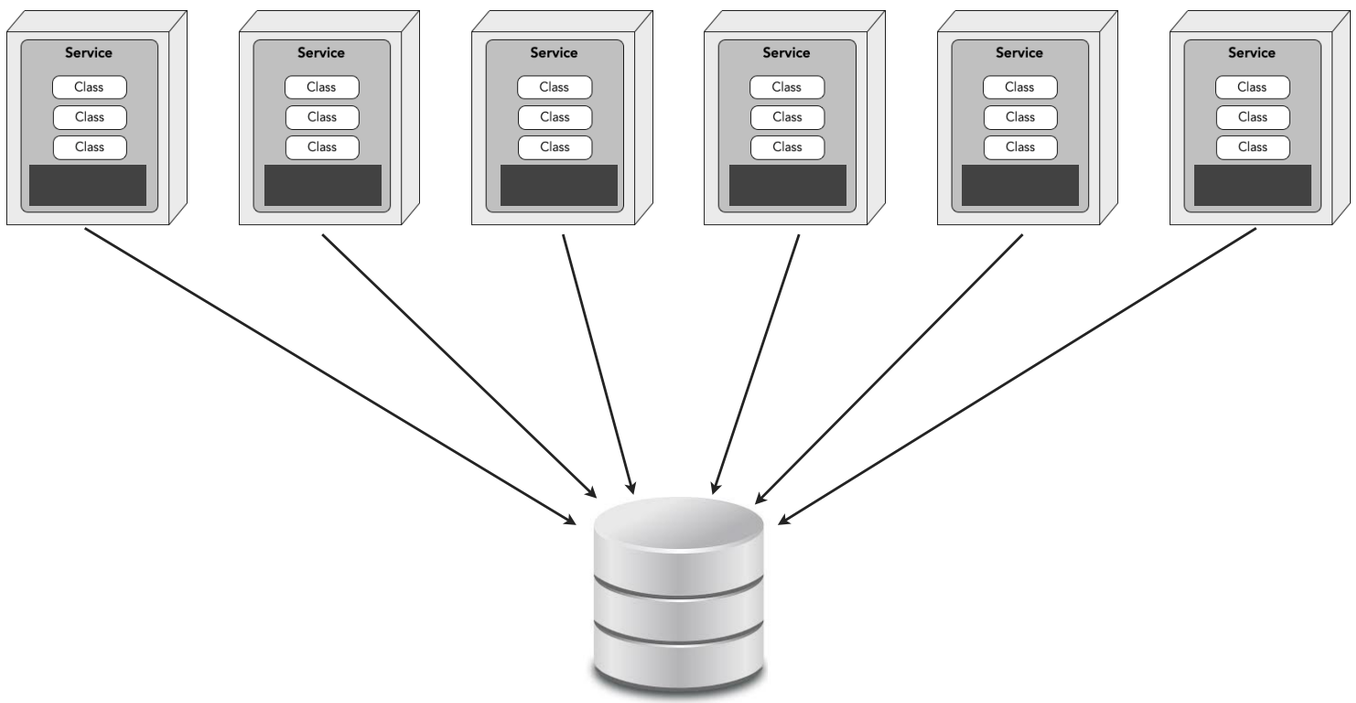
database granularity drivers

change control

database connections

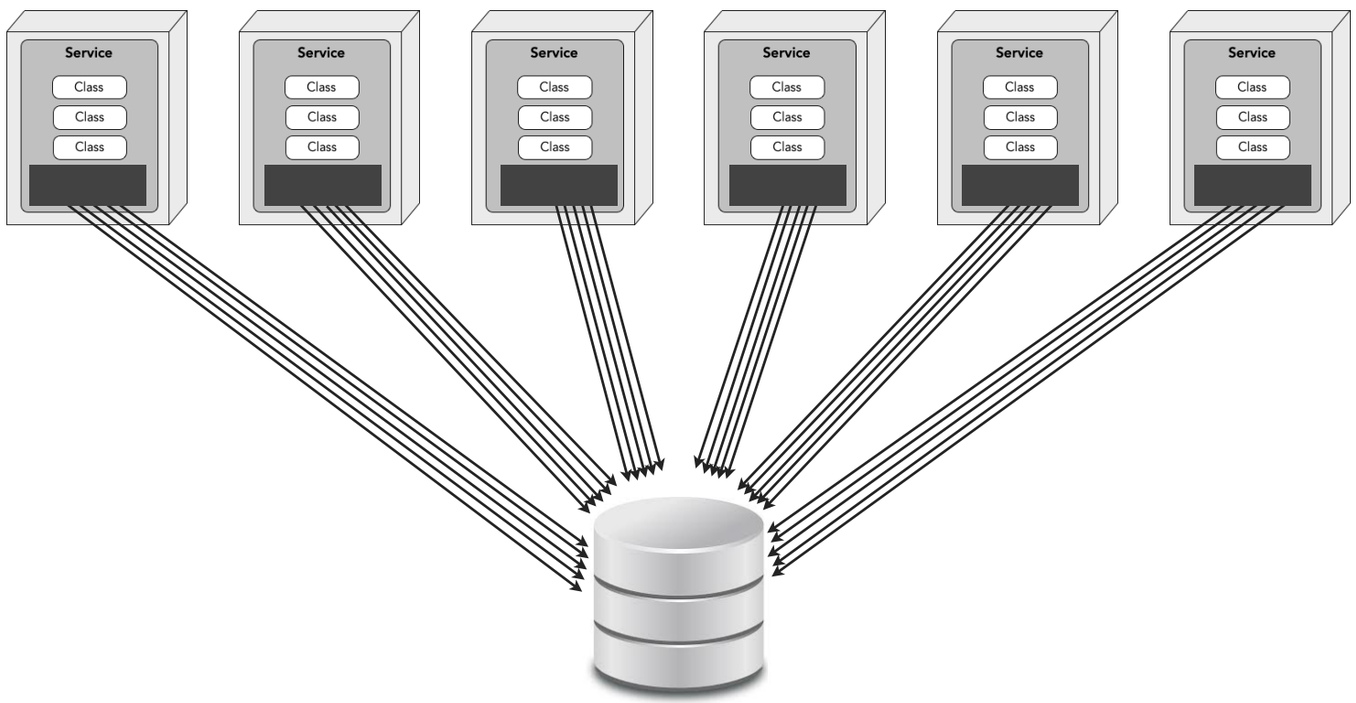
breaking apart data

database connections



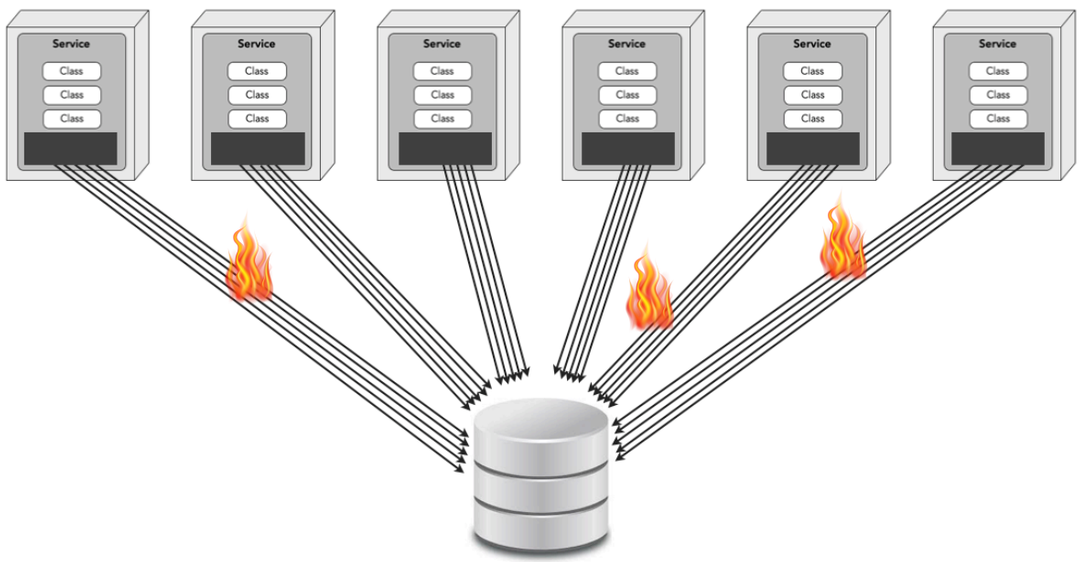
breaking apart data

database connections



breaking apart data

database connections



monolithic application: 200 connections

distributed services: 50

connections per service: 10

service instances (min): 2

distributed connections: 1000 connections

breaking apart data

"when should I consider breaking apart my data?"

database granularity drivers

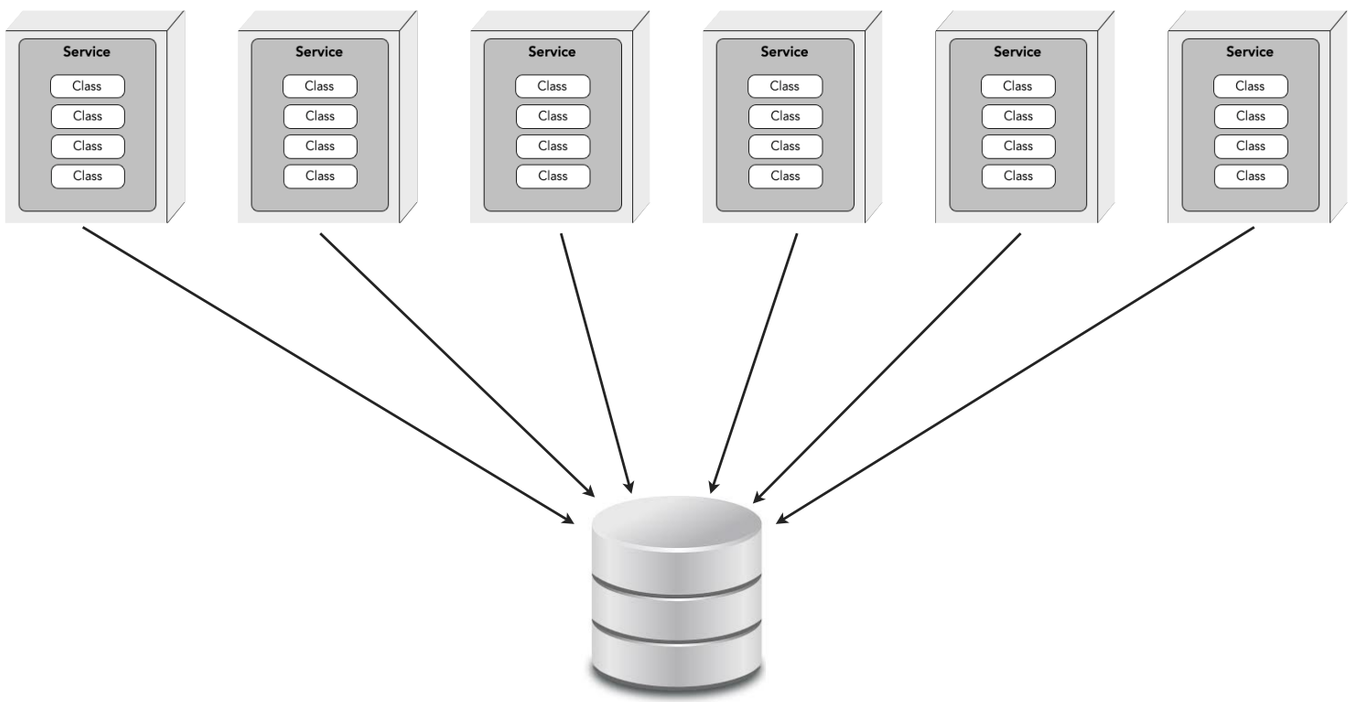
change control

database connections

database scalability

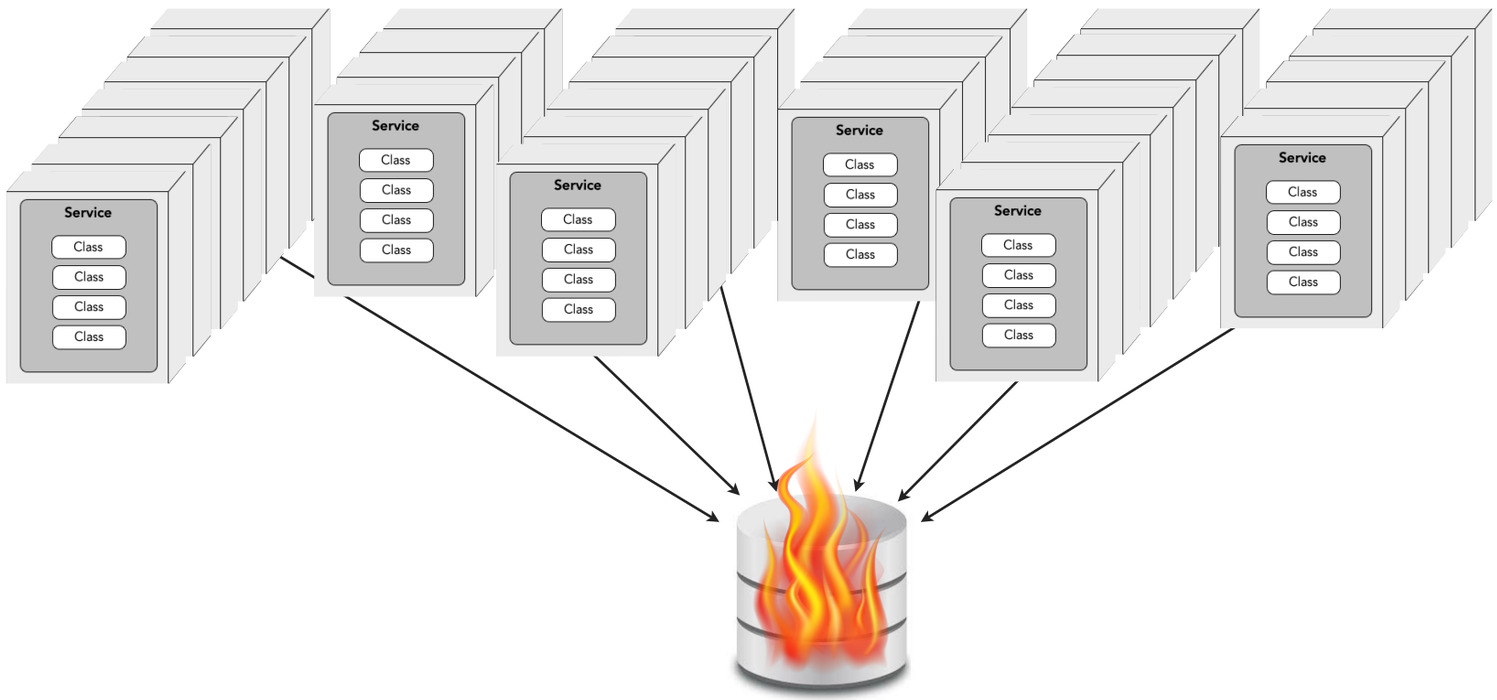
breaking apart data

database scalability



breaking apart data

database scalability



breaking apart data

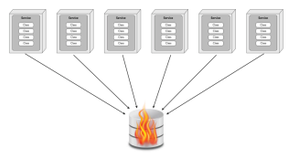
"when should I consider breaking apart my data?"

database granularity drivers

change control

database connections

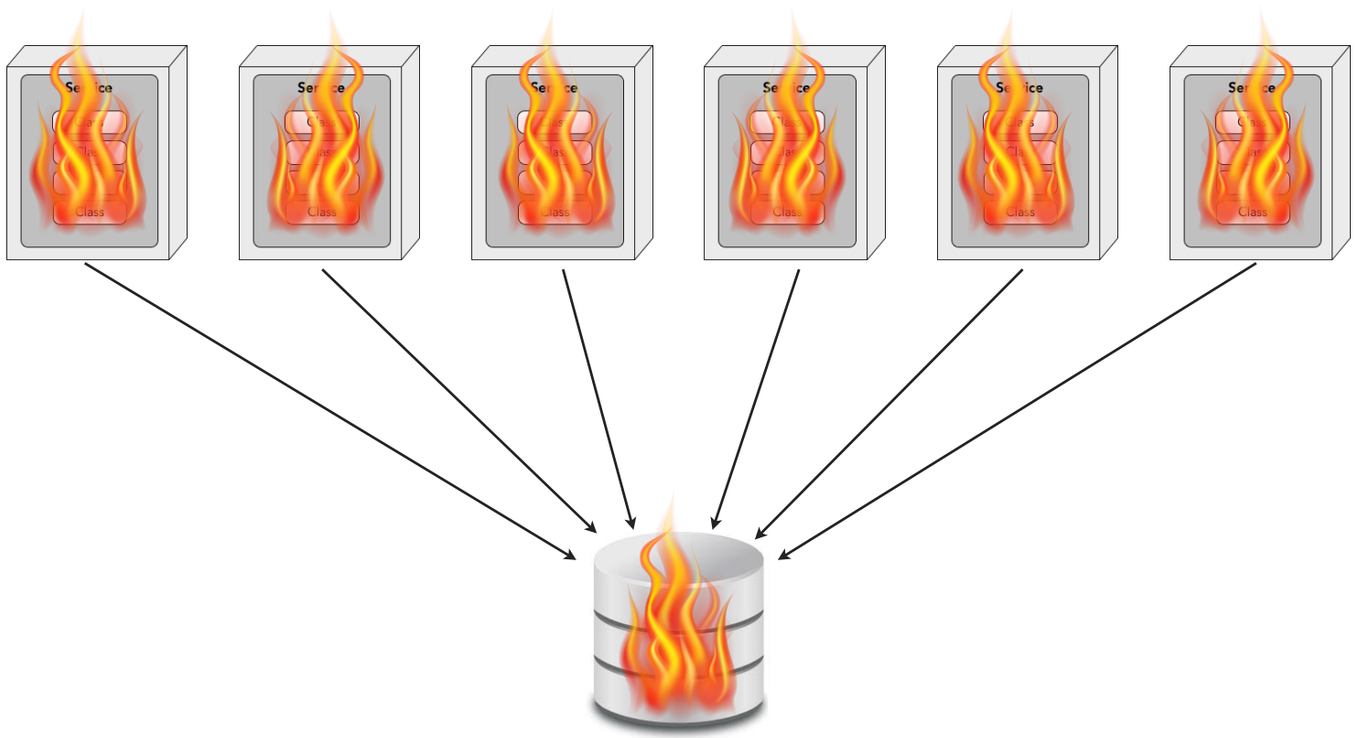
database scalability



fault tolerance

breaking apart data

fault tolerance



breaking apart data

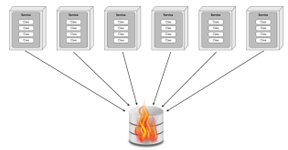
"when should I consider breaking apart my data?"

database granularity drivers

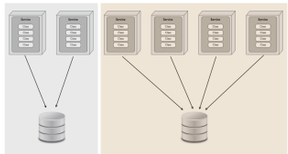
change control

database connections

database scalability



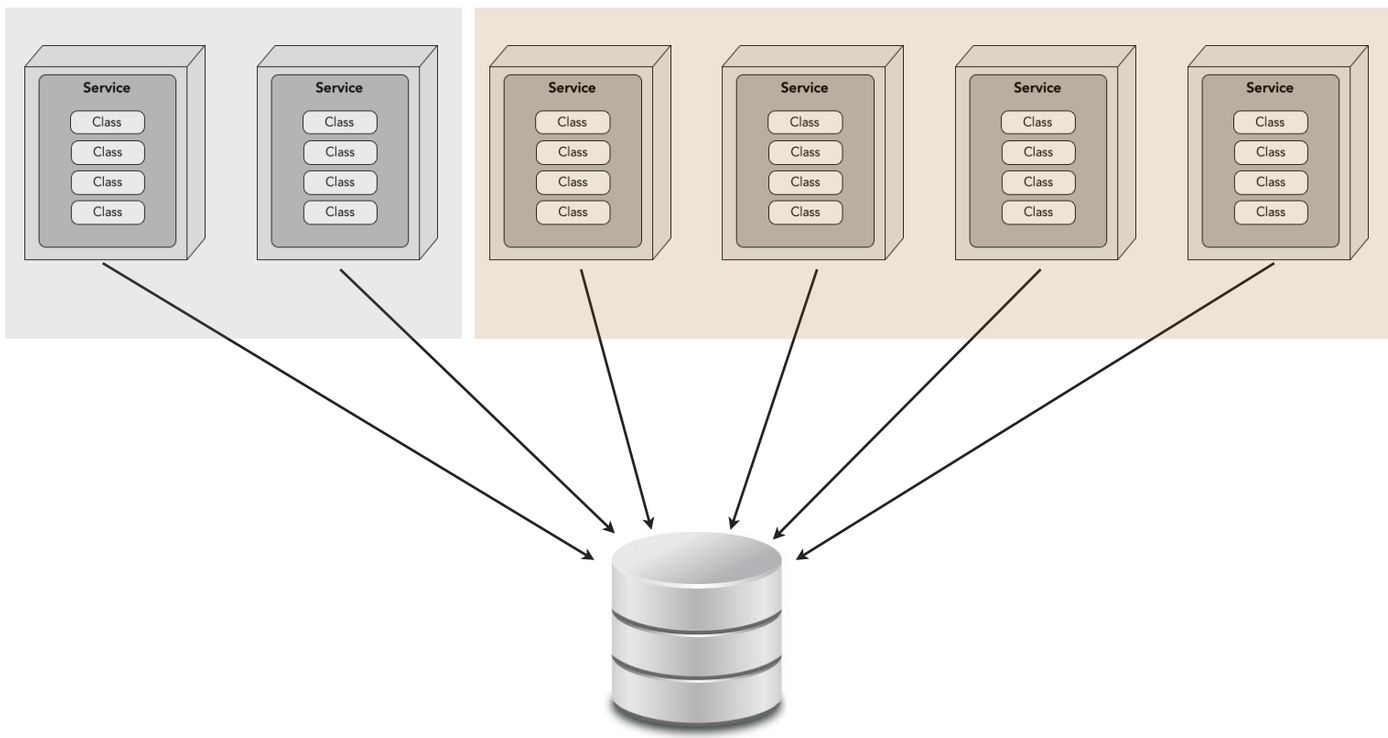
fault tolerance



architectural quantum

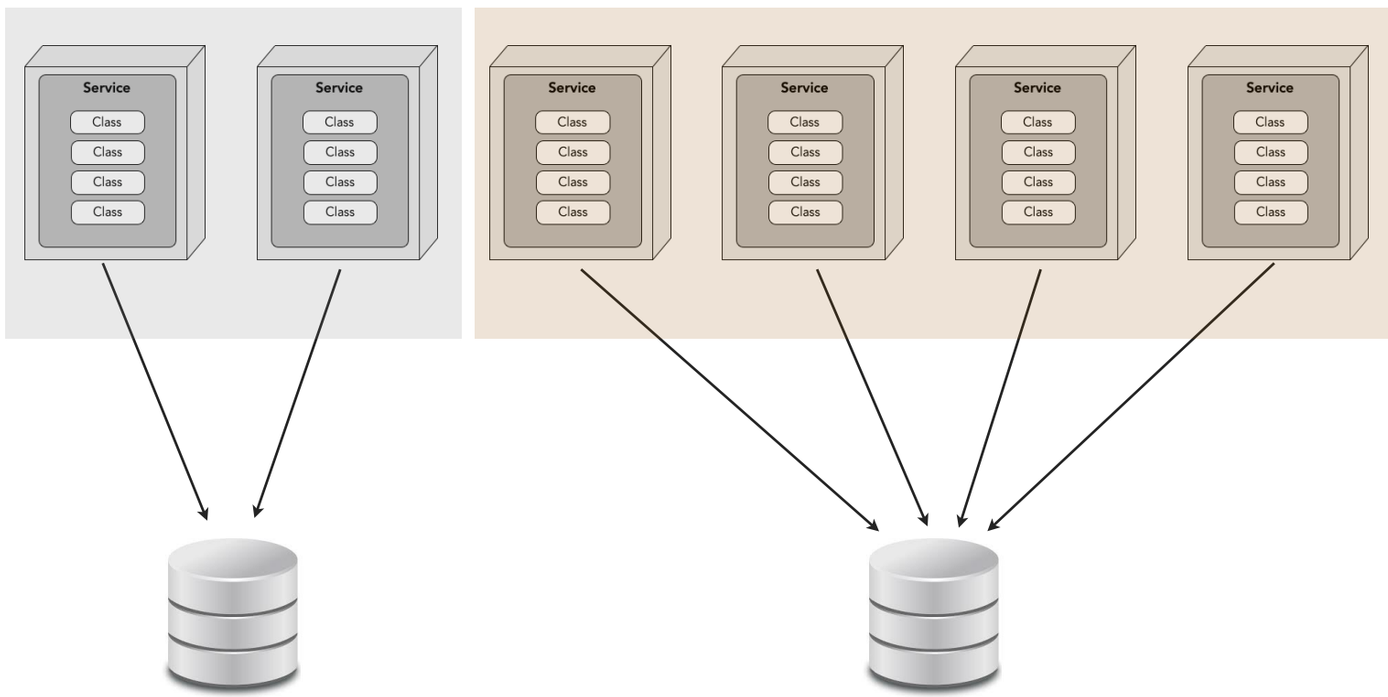
breaking apart data

architectural quantum



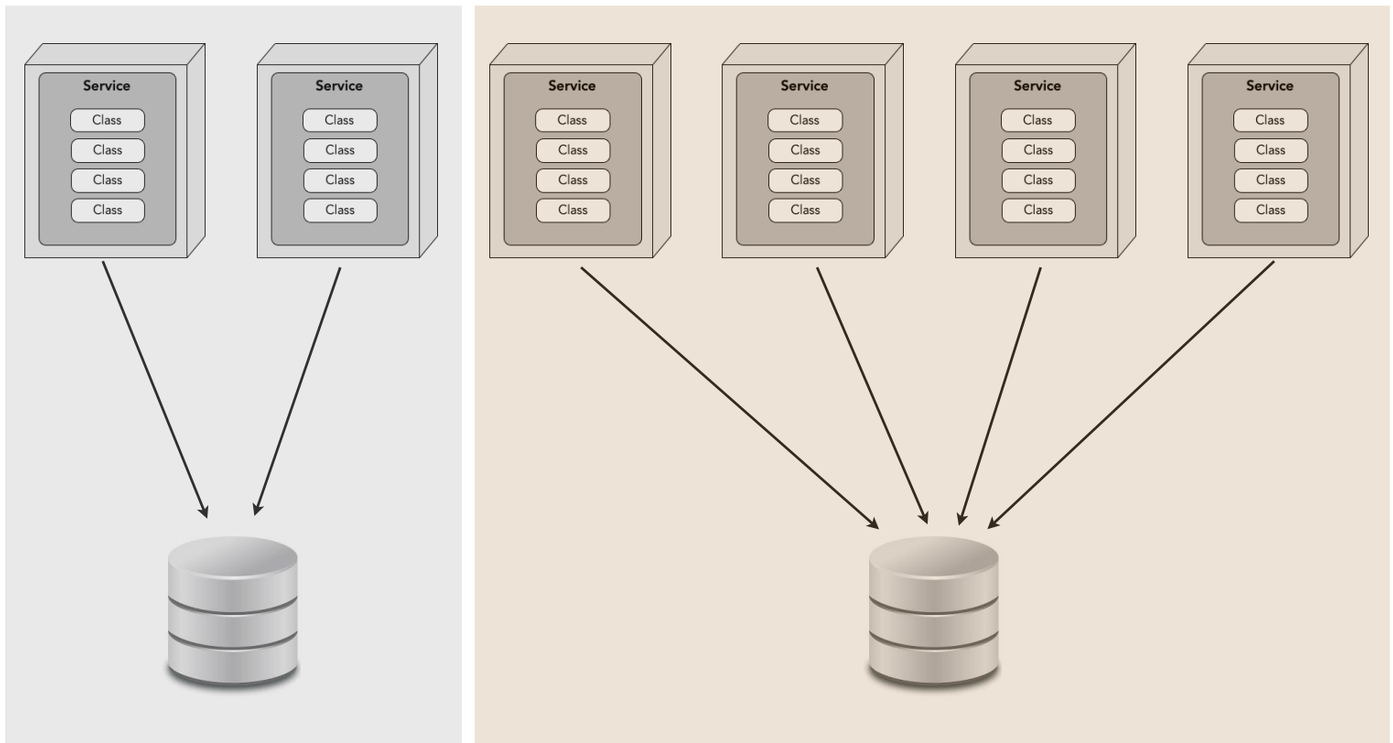
breaking apart data

architectural quantum



breaking apart data

architectural quantum



breaking apart data

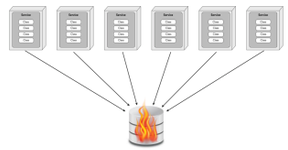
"when should I consider breaking apart my data?"

database granularity drivers

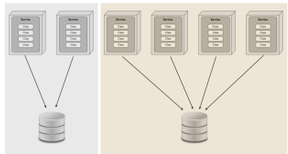
change control

database connections

database scalability

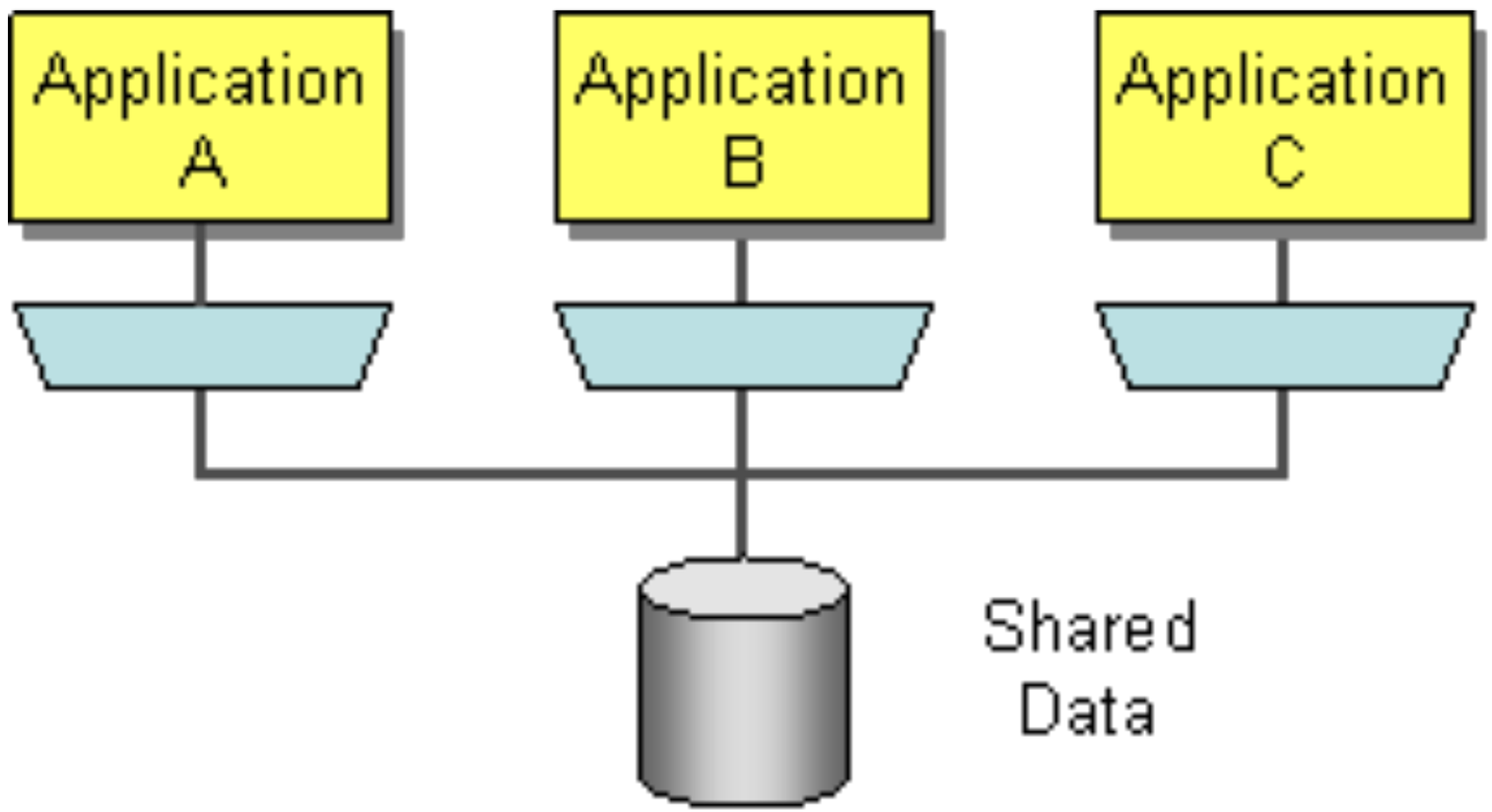


fault tolerance

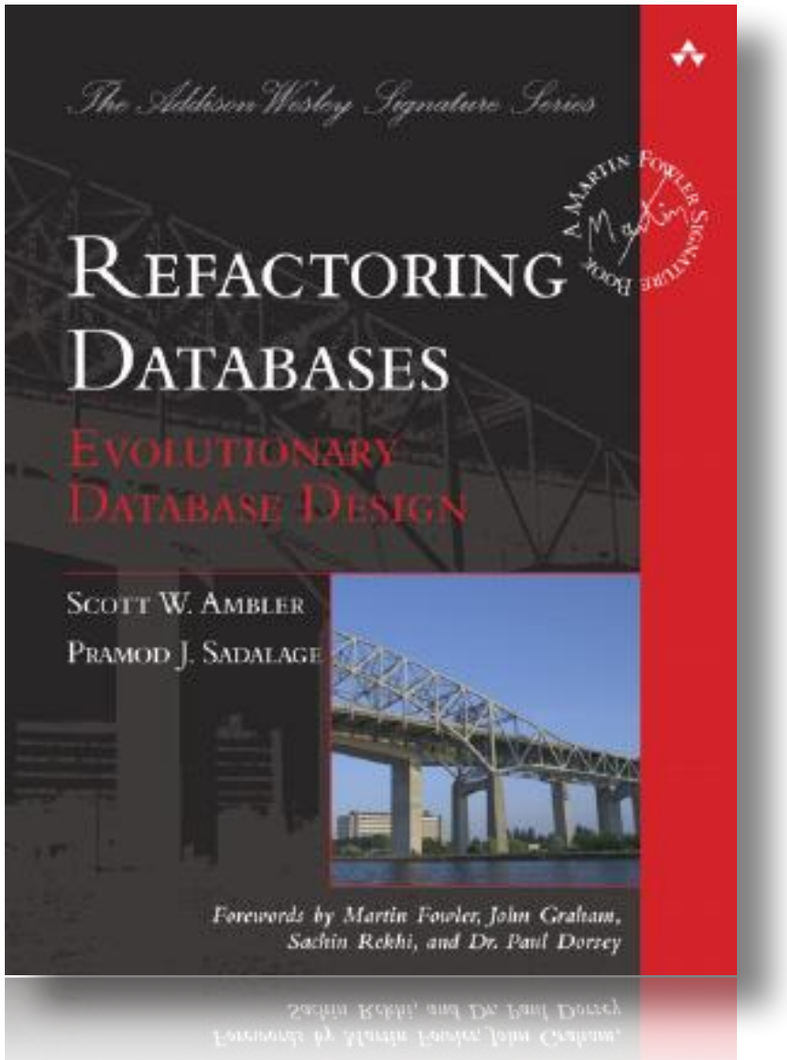


architectural quantum

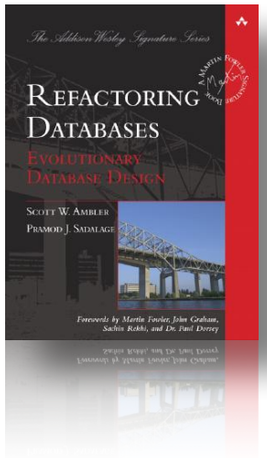
Shared-database Integration

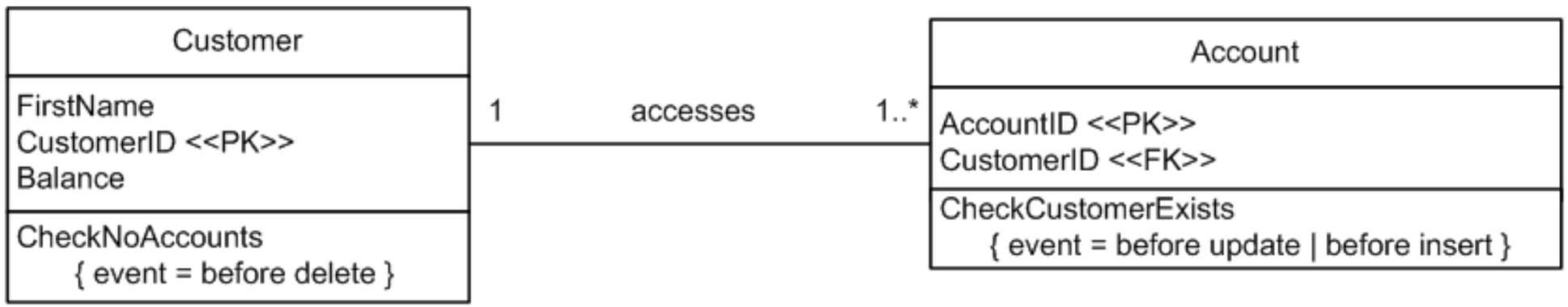


Refactoring Databases



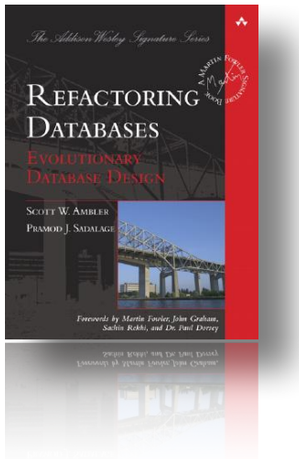
Move Column Refactoring

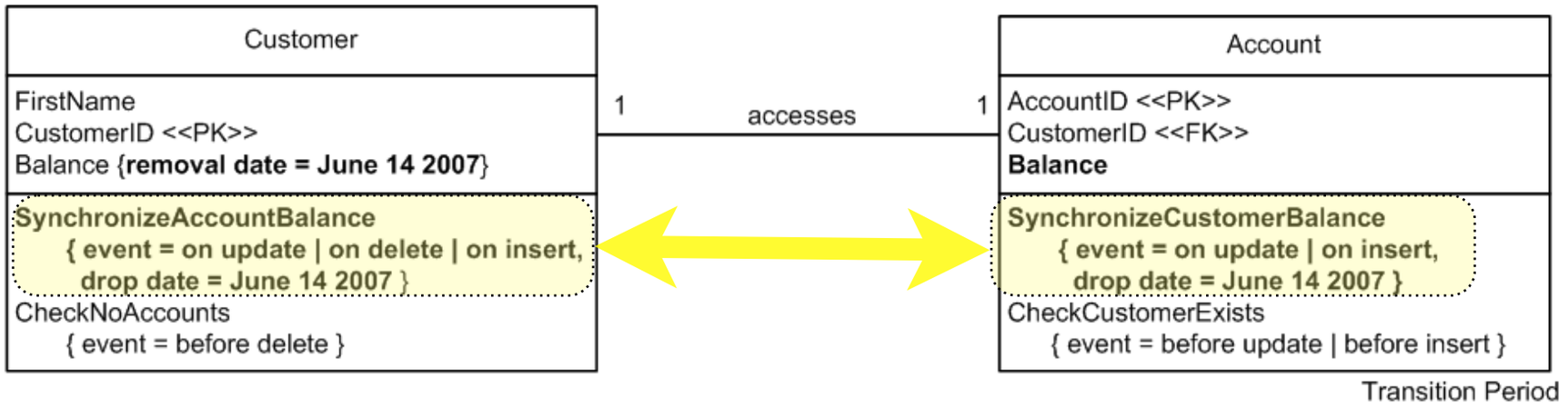


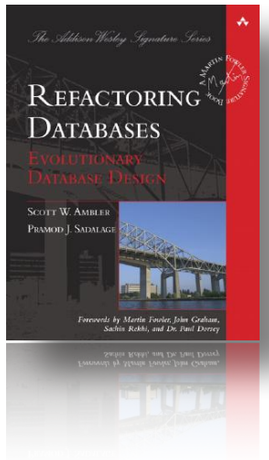


Original Schema

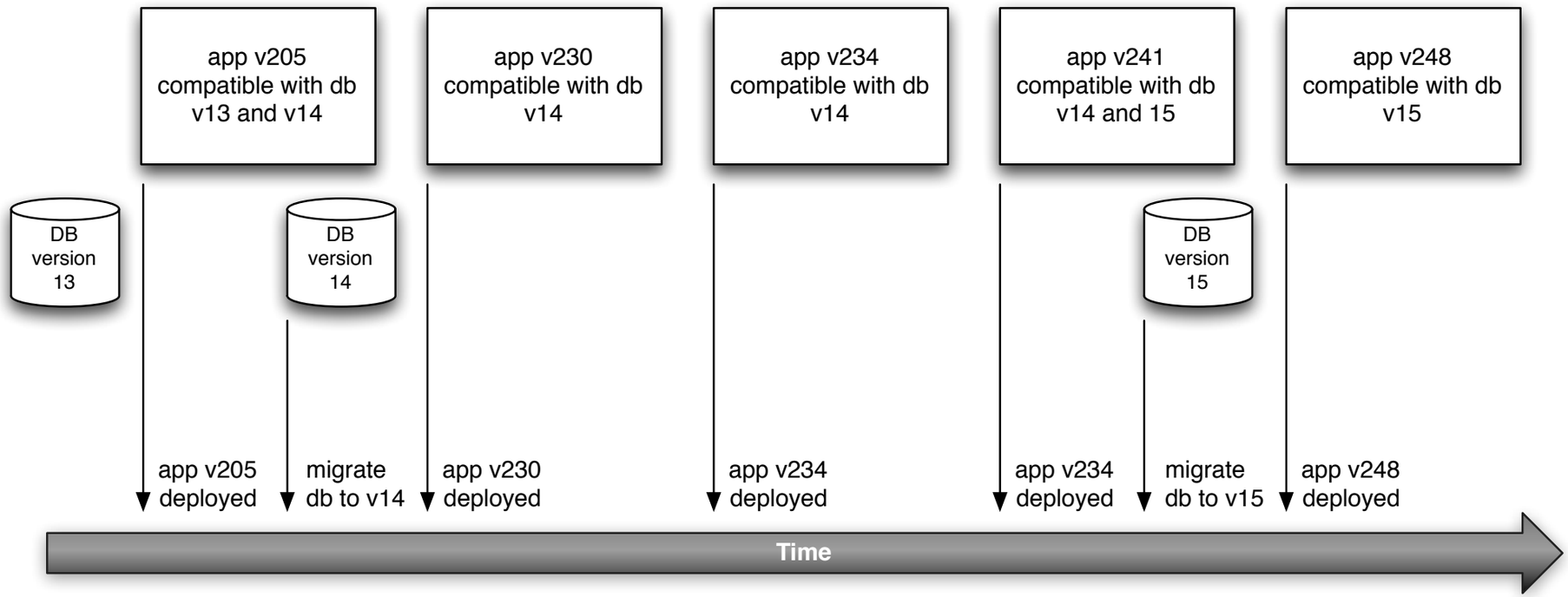
Transition Period



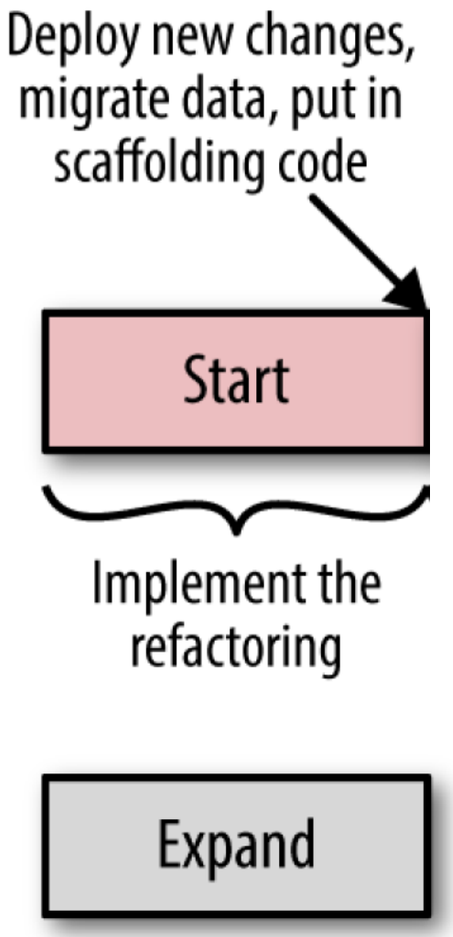




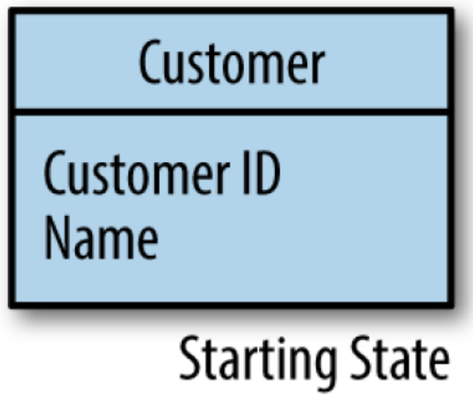
Decouple DB Updates: the Expand/contract Pattern



Expand/Contract Pattern

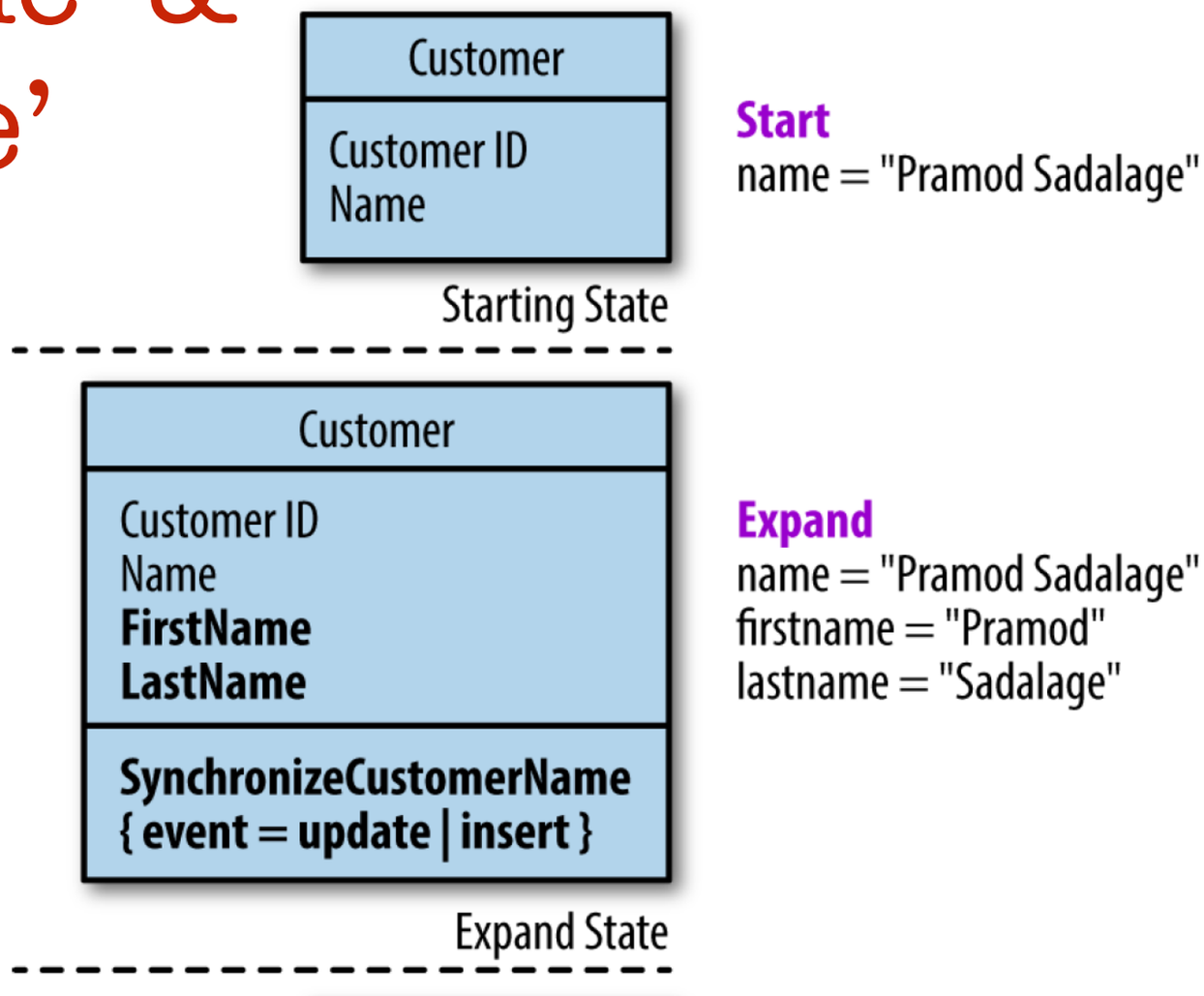


Change ‘name’ to ‘firstname’ & ’lastname’



Start
name = "Pramod Sadalage"

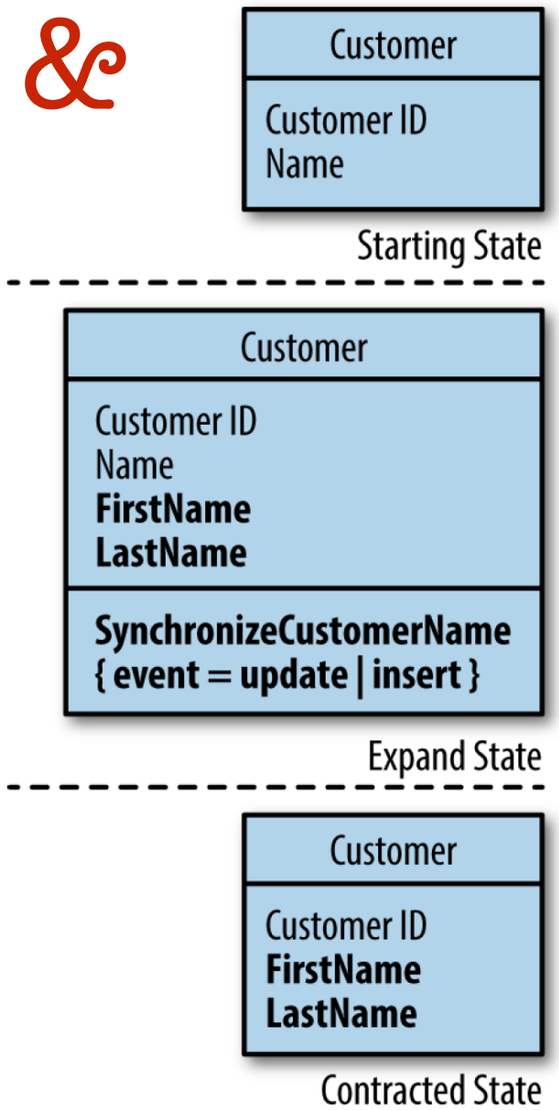
Change ‘name’ to ‘firstname’ & ’lastname’



Start
name = "Pramod Sadalage"

Expand
name = "Pramod Sadalage"
firstname = "Pramod"
lastname = "Sadalage"

Change ‘name’ to ‘firstname’ & ’lastname’



Start
name = "Pramod Sadalage"

Expand
name = "Pramod Sadalage"
firstname = "Pramod"
lastname = "Sadalage"

Contract
firstname = "Pramod"
lastname = "Sadalage"

#1: ~~integration points~~, ~~legacy data~~

```
ALTER TABLE customer ADD firstname VARCHAR2(60);
ALTER TABLE customer ADD lastname VARCHAR2(60);
ALTER TABLE customer DROP COLUMN name;
```

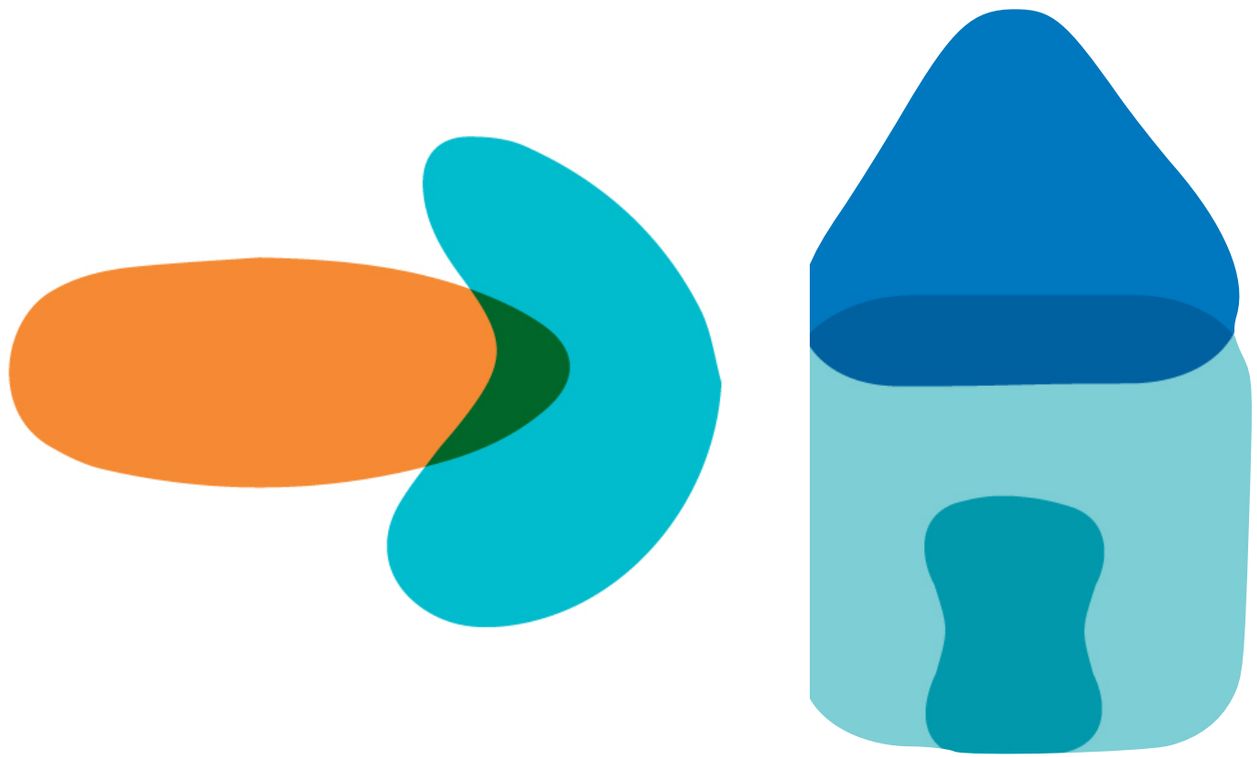
#2: ~~integration points~~, legacy data

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);
UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);
ALTER TABLE customer DROP COLUMN name;
```

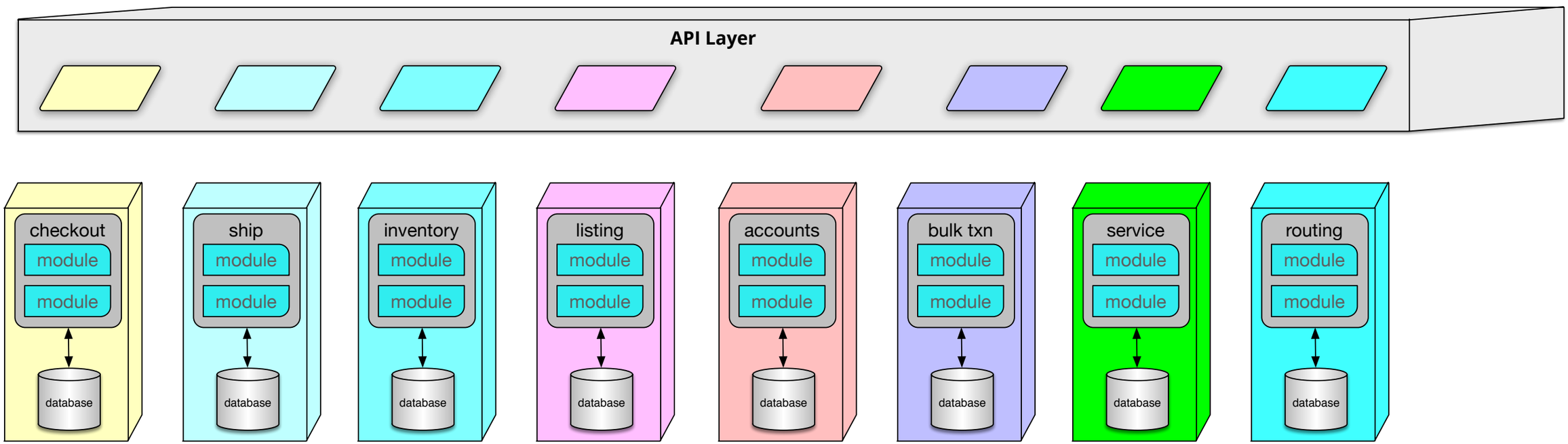
#3: integration points, legacy data

```
ALTER TABLE Customer ADD firstname VARCHAR2(60);
ALTER TABLE Customer ADD lastname VARCHAR2(60);

UPDATE Customer set firstname = extractfirstname (name);
UPDATE Customer set lastname = extractlastname (name);

CREATE OR REPLACE TRIGGER SynchronizeName
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN

IF :NEW.Name IS NULL THEN
  :NEW.Name := :NEW.firstname||' '||:NEW.lastname;
END IF;
IF :NEW.name IS NOT NULL THEN
  :NEW.firstname := extractfirstname(:NEW.name);
  :NEW.lastname := extractlastname(:NEW.name);
END IF;
END;
```

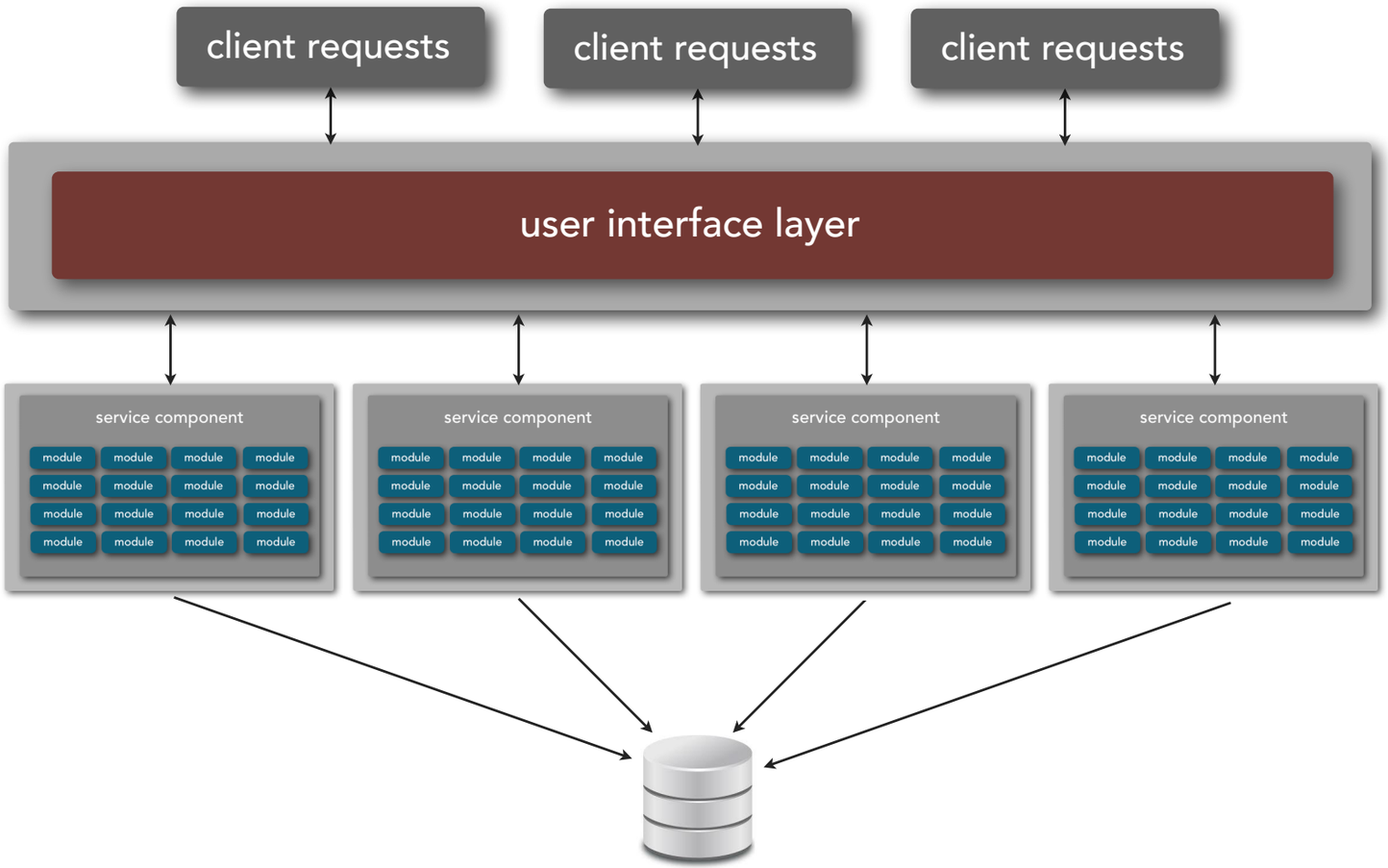
Migration Targets



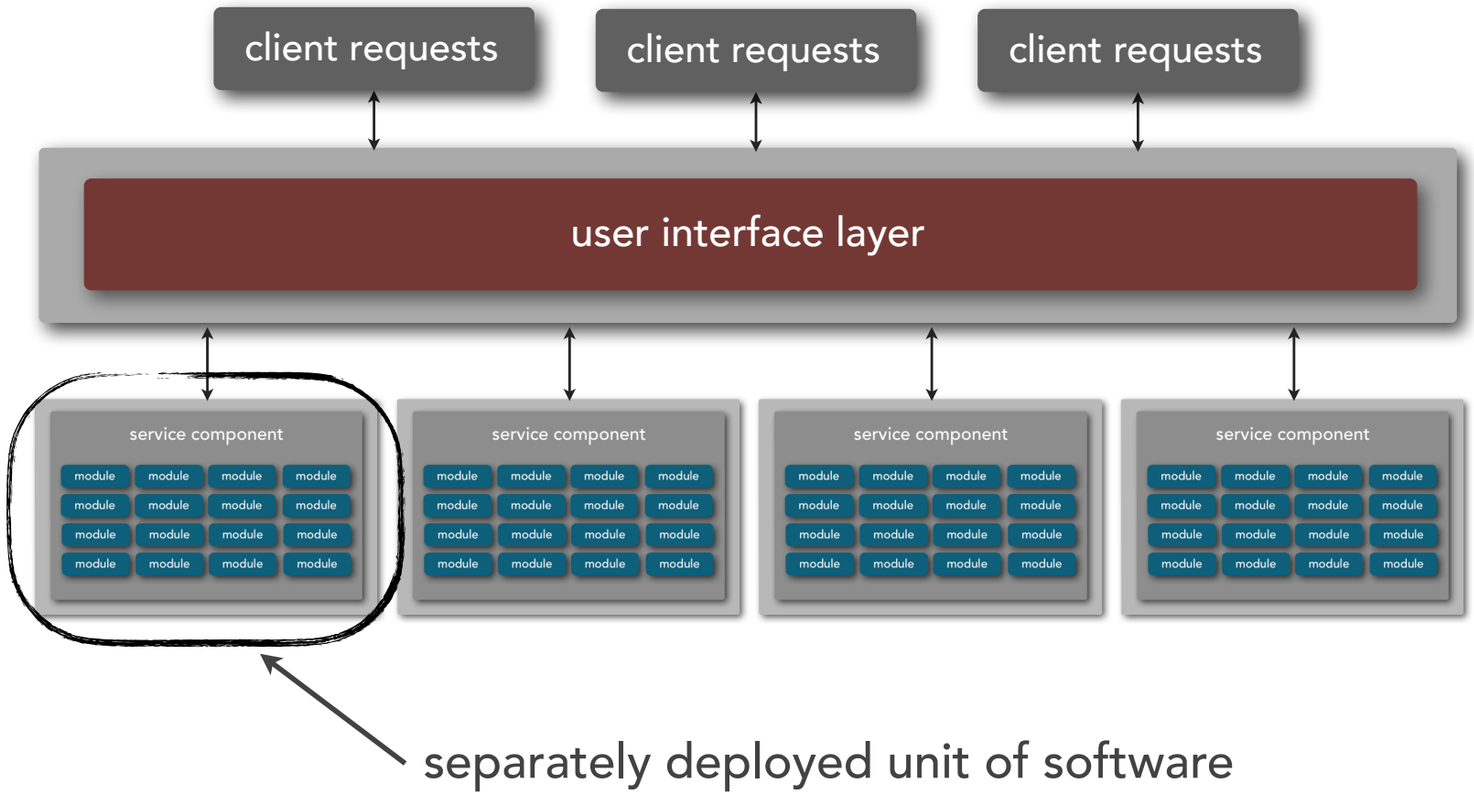
Microservices?



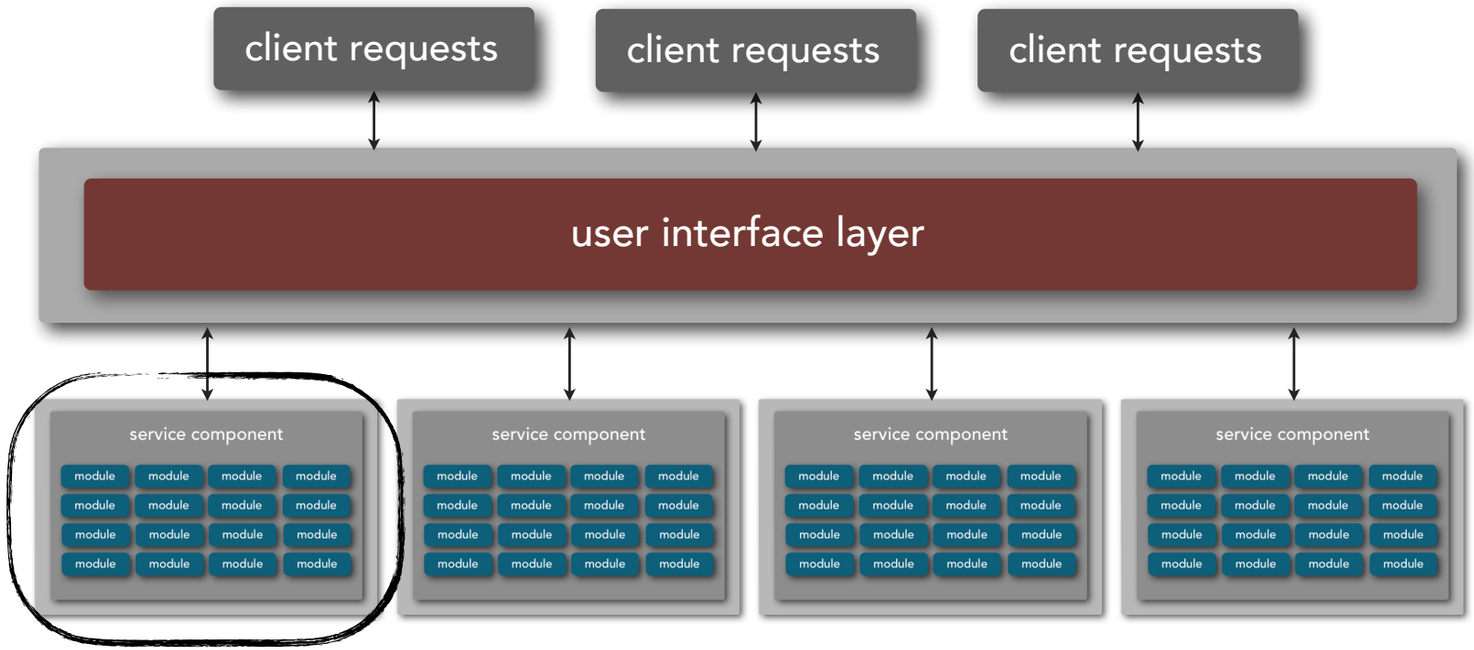
service-based architecture



service-based architecture

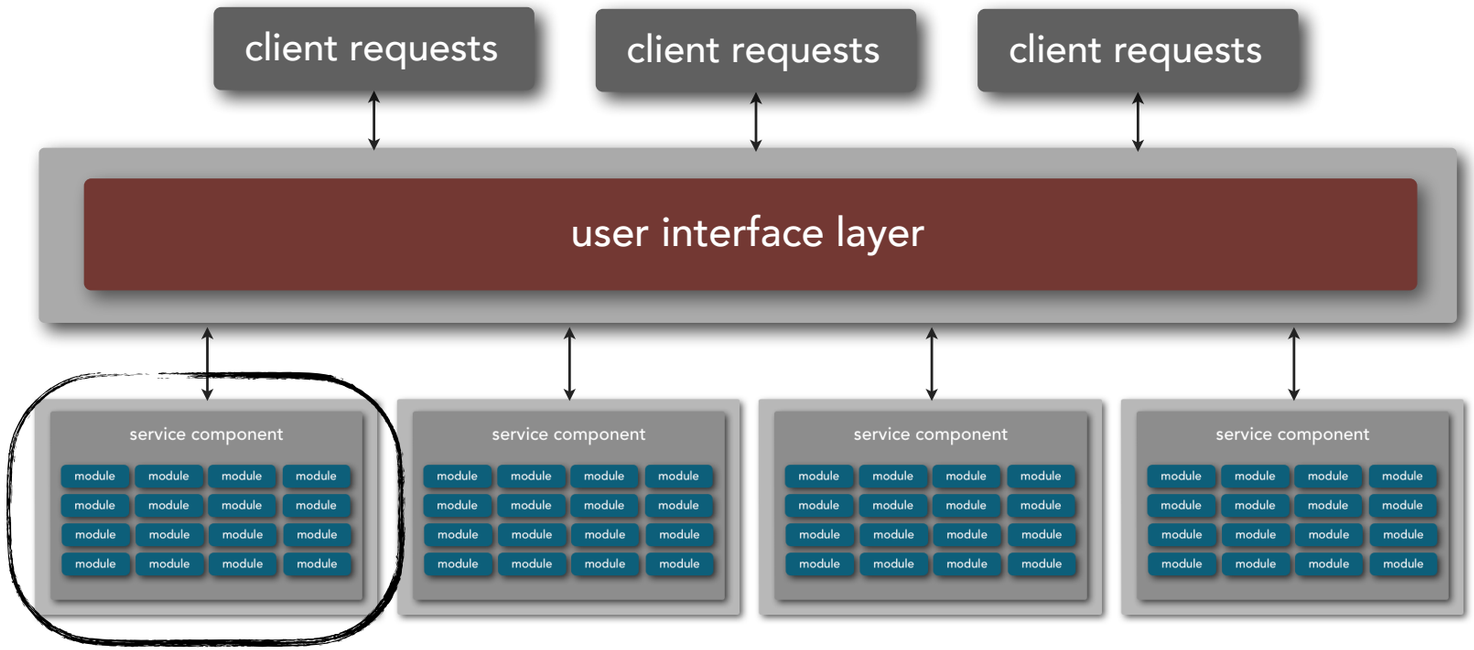


service-based architecture



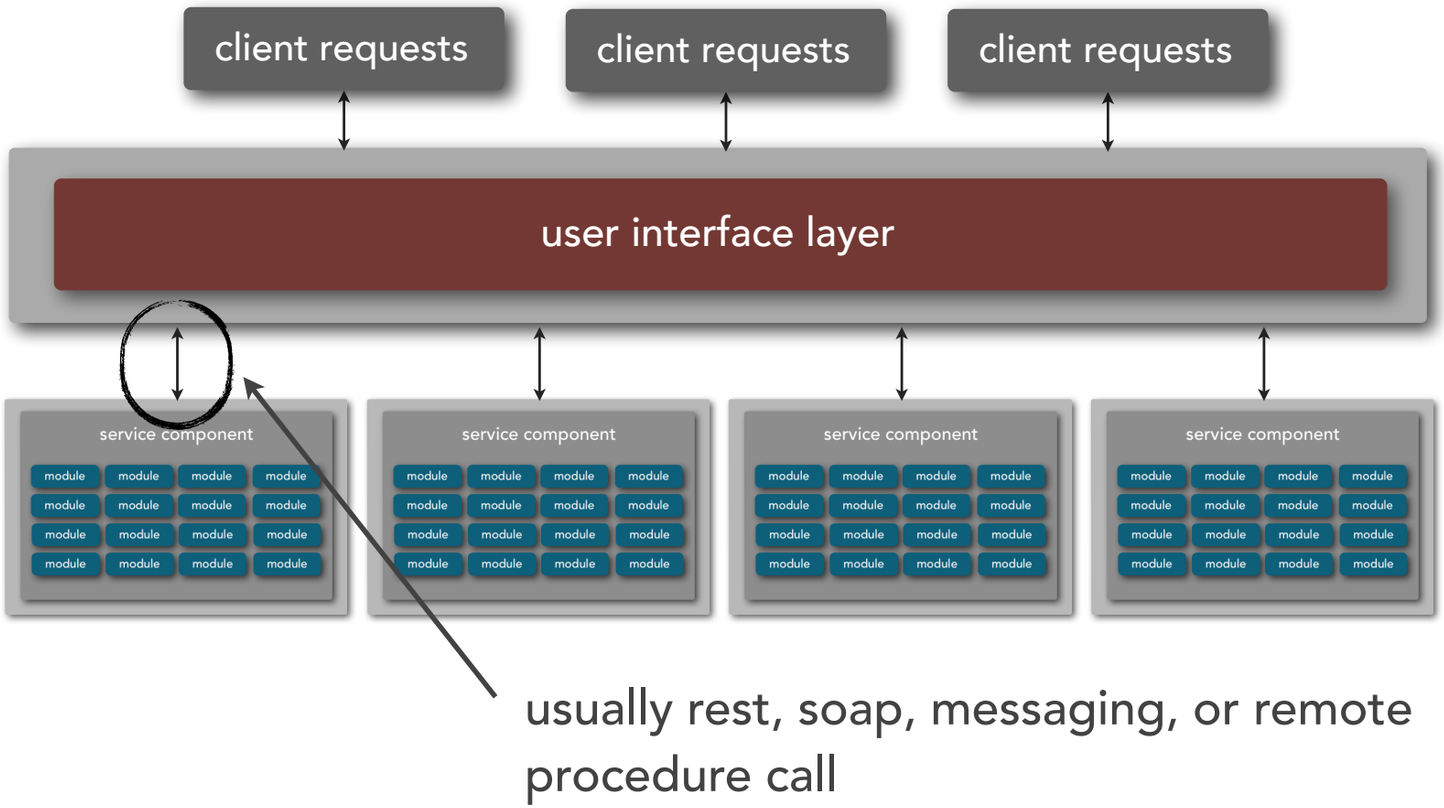
same deployment unit as original application
(ear, assembly, etc.)

service-based architecture

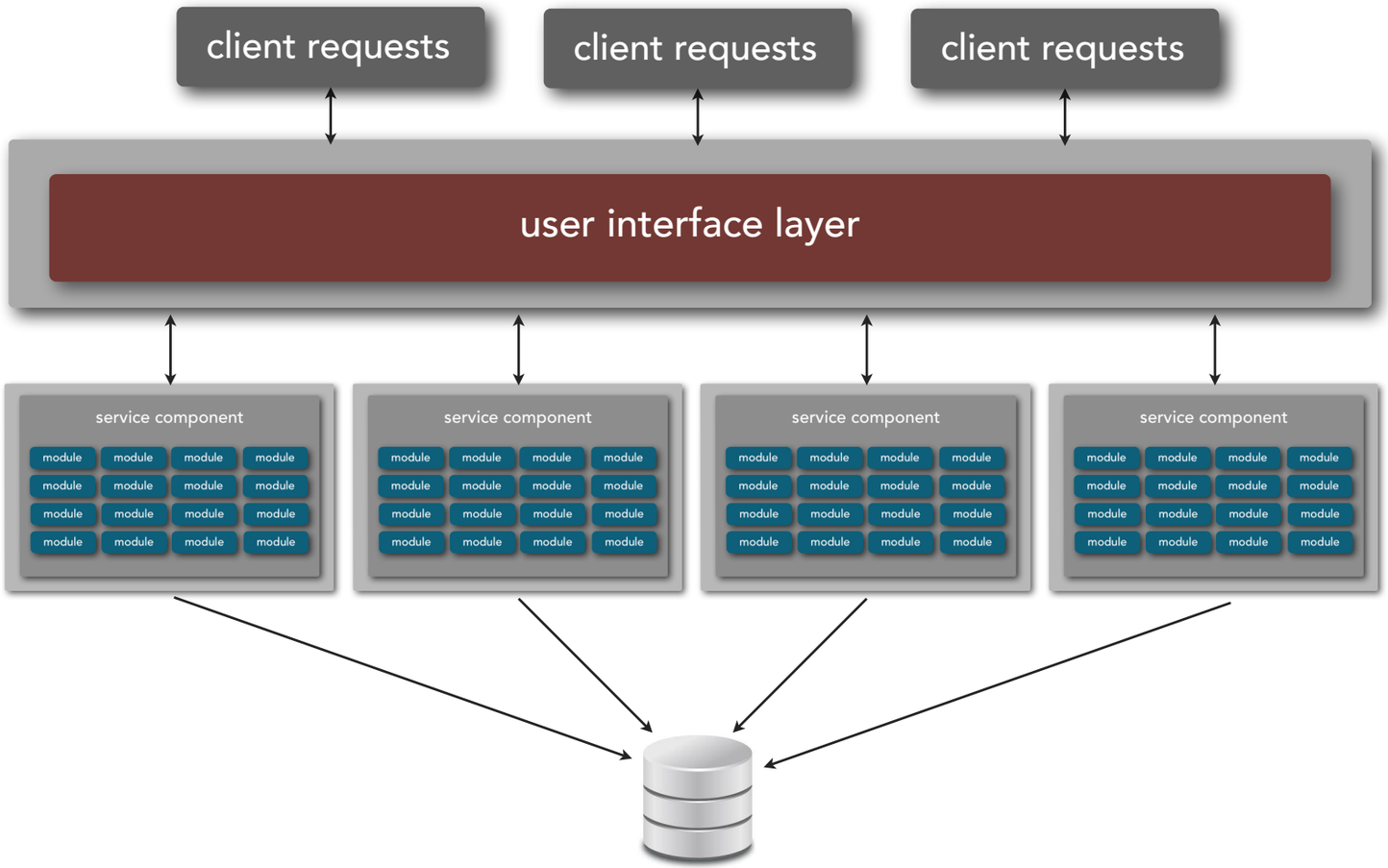


independent self-contained portion of the application (6 - 12 services)

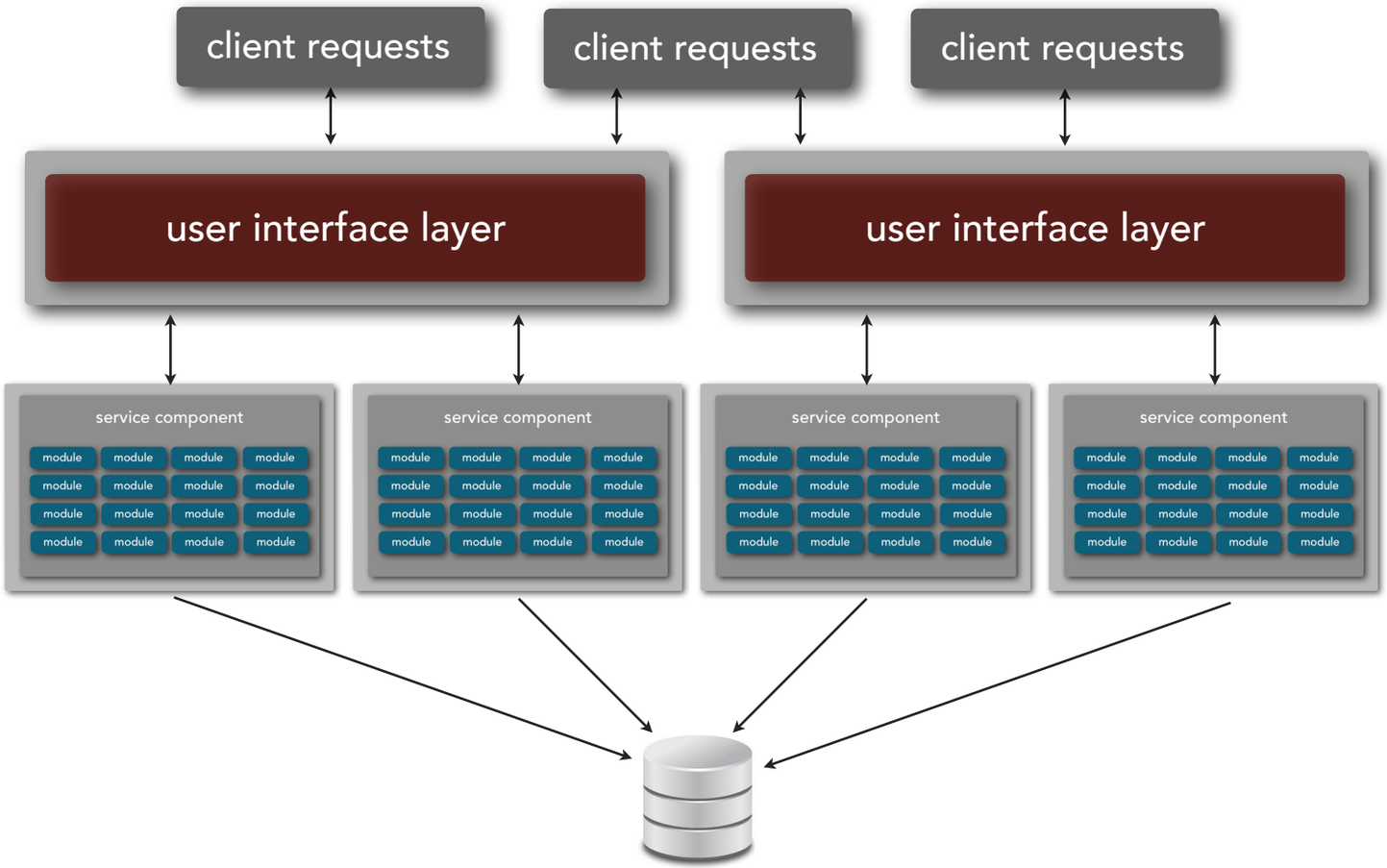
service-based architecture



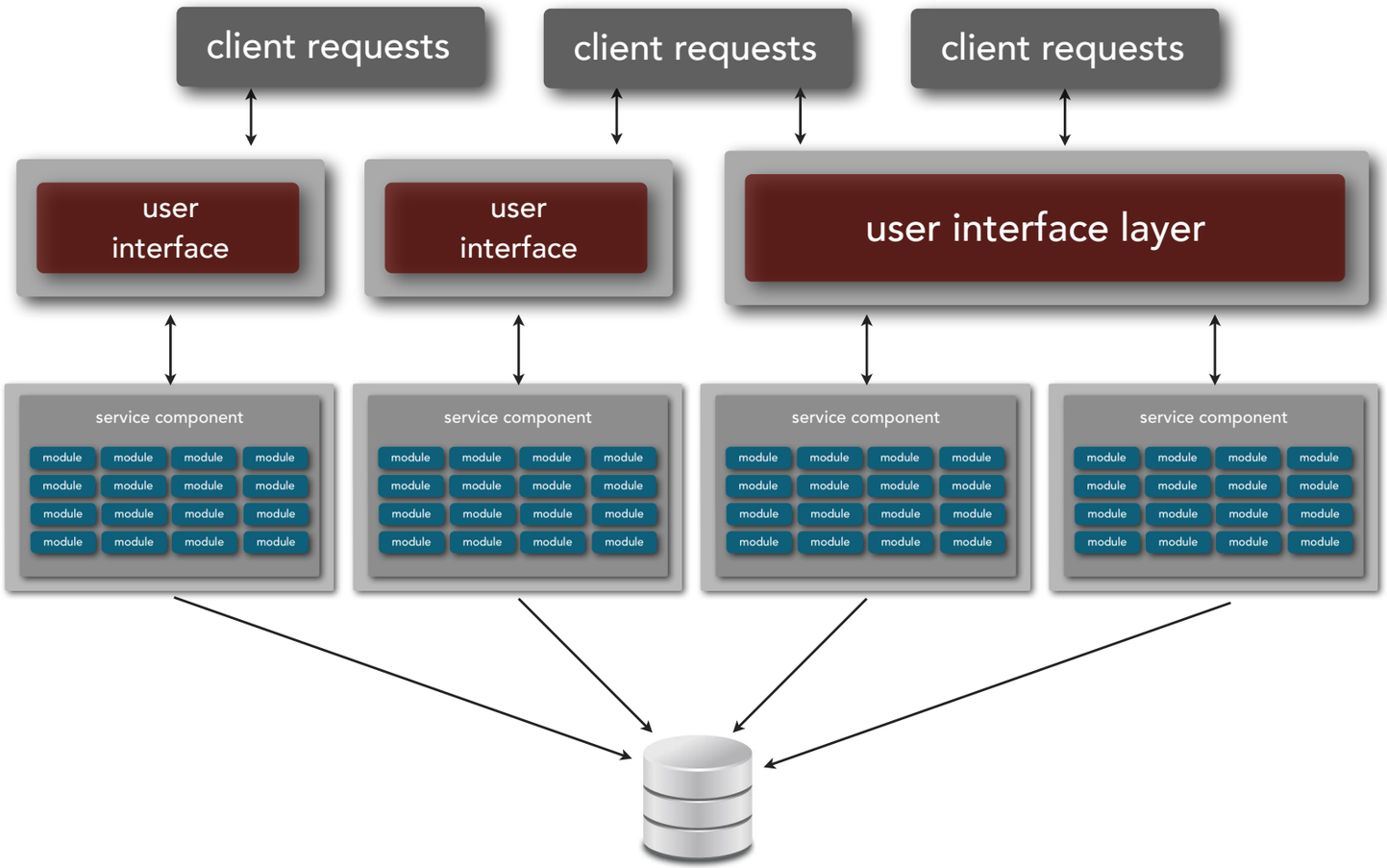
service-based architecture



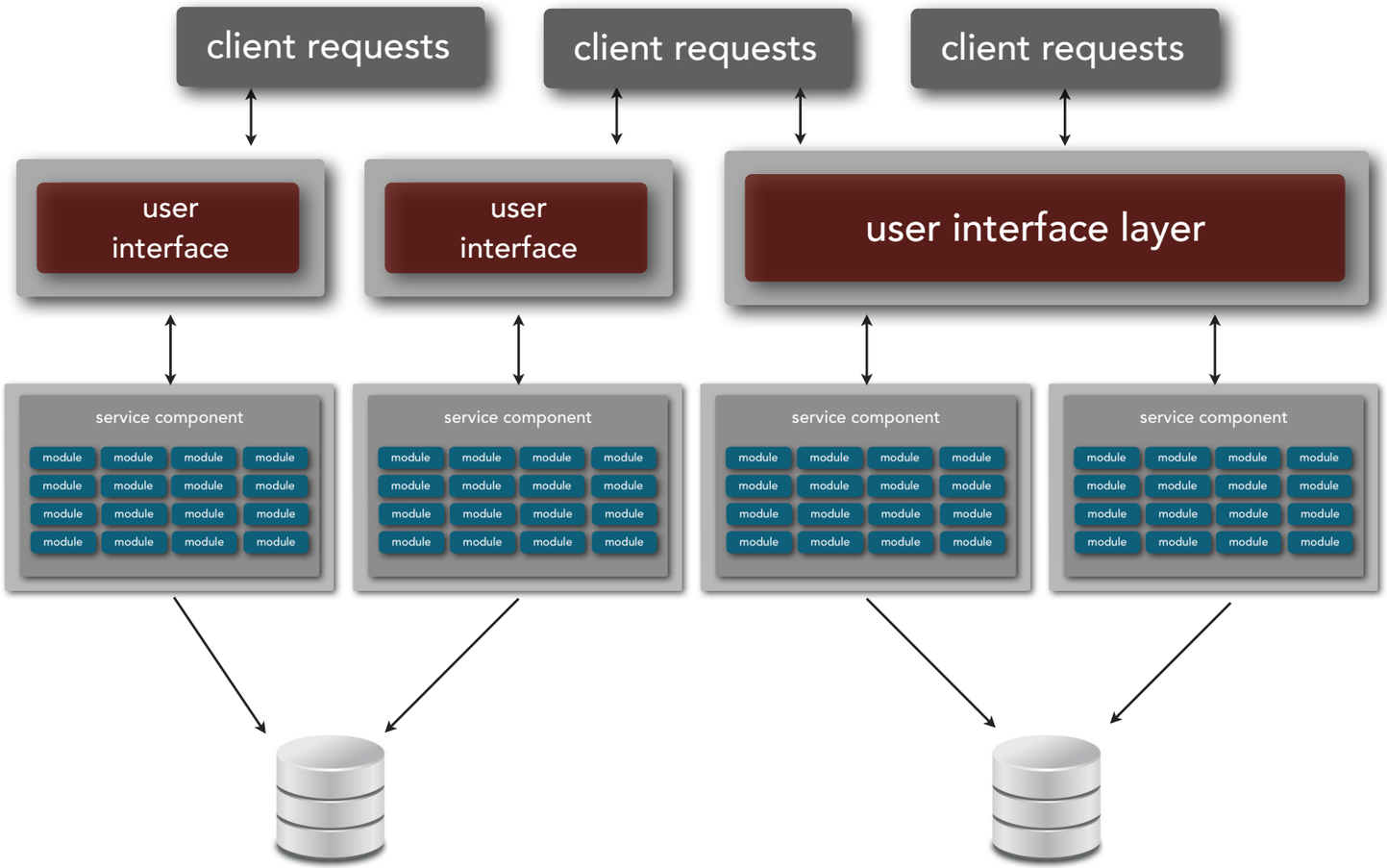
service-based architecture



service-based architecture



service-based architecture



what?

how do we assess the current architecture?

where?

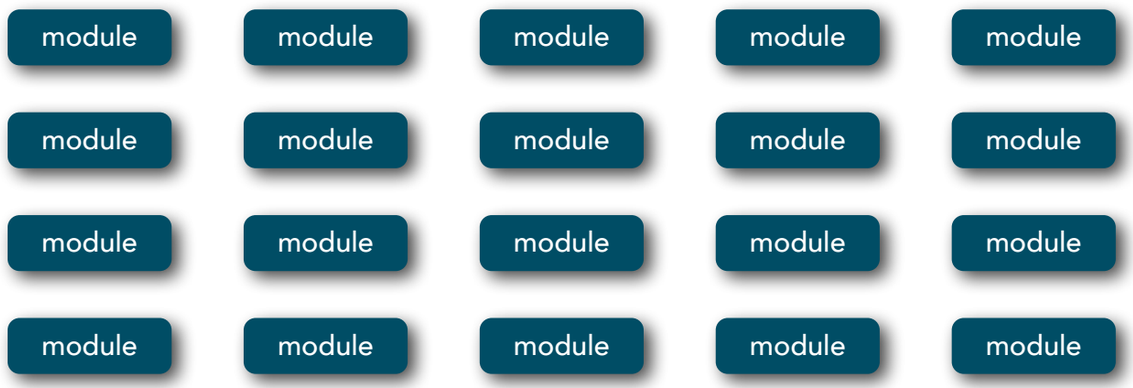
what should we change it to?

how?

how do we change it?

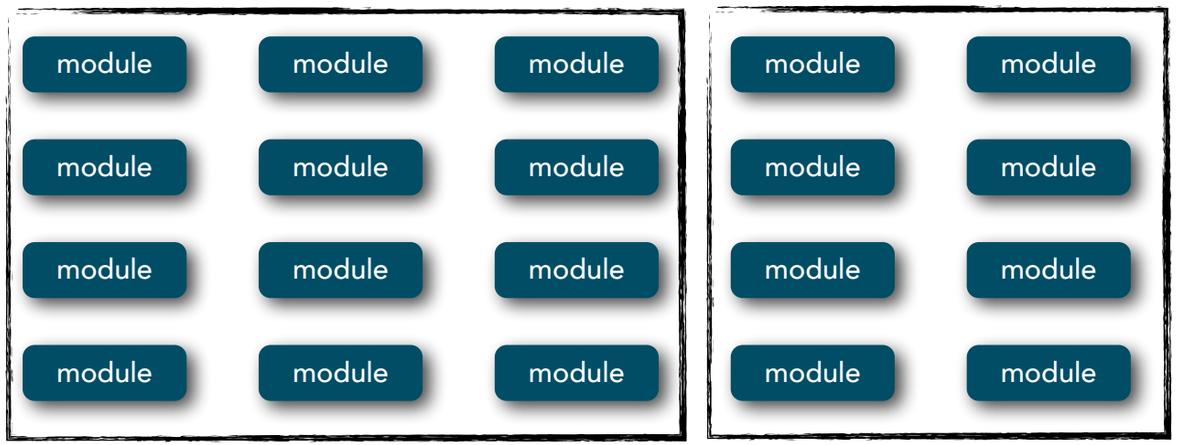
Migration Challenges

service component granularity



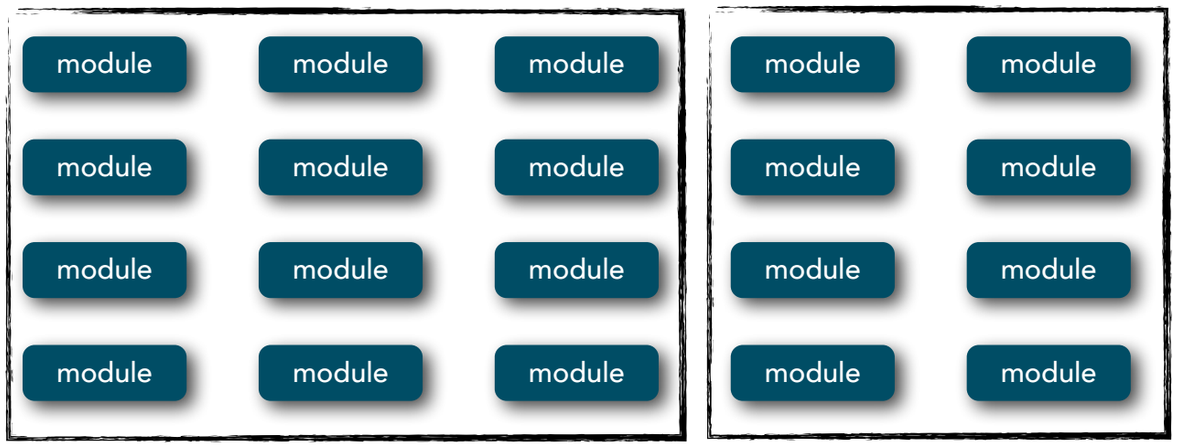
Migration Challenges

service component granularity



Migration Challenges

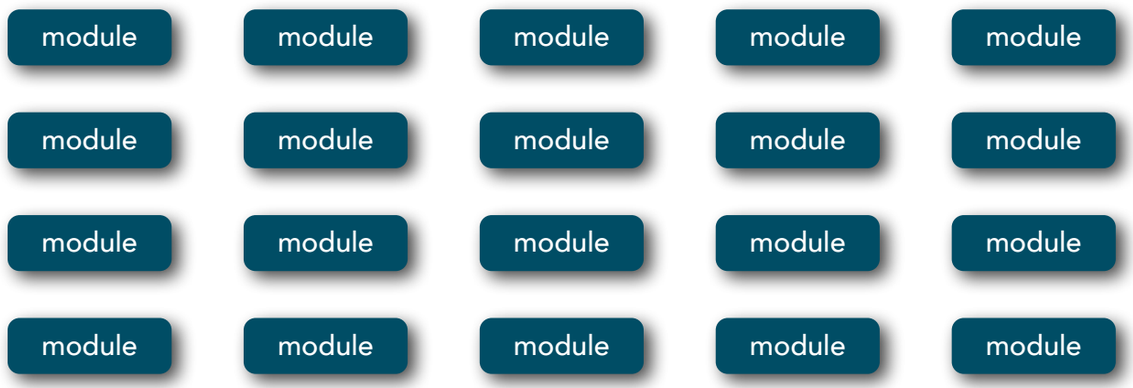
service component granularity



coarse-grained service components address transactional issues but may not achieve your desired goals

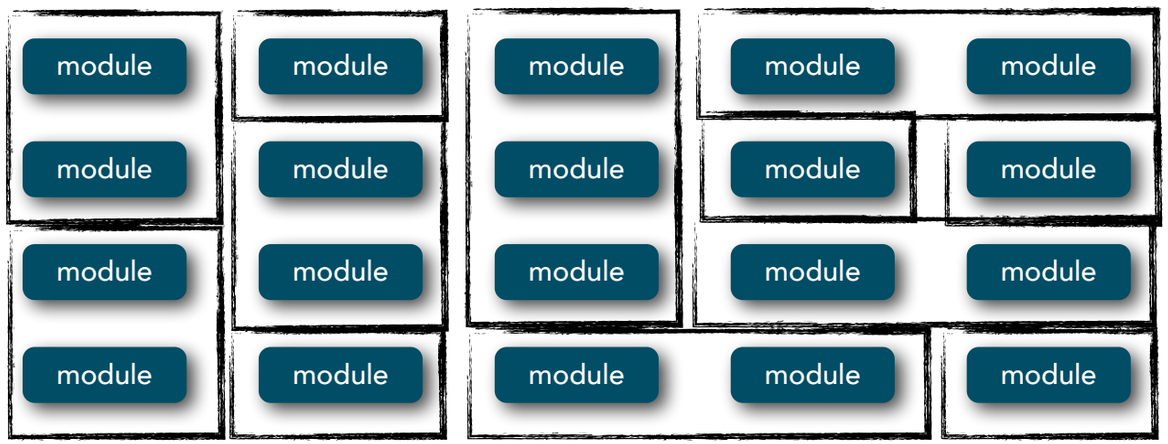
Migration Challenges

service component granularity



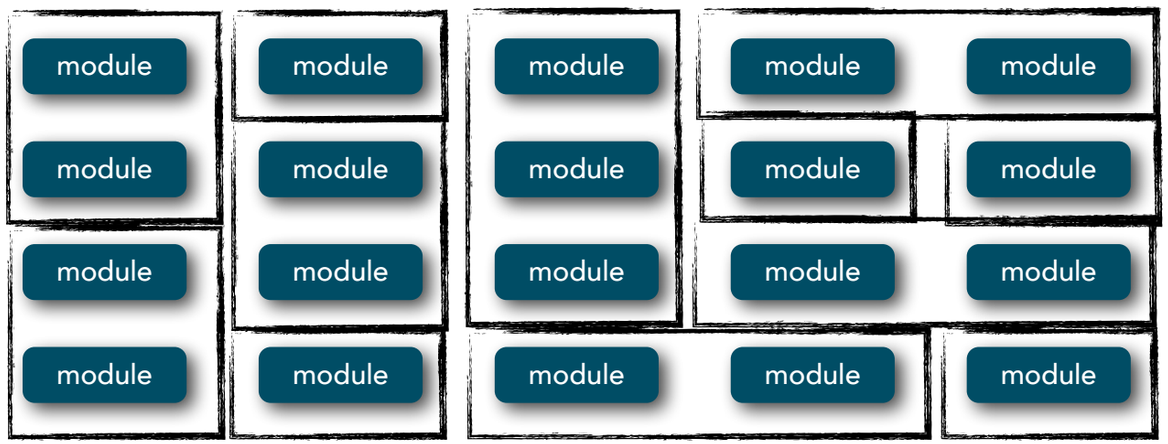
Migration Challenges

service component granularity



Migration Challenges

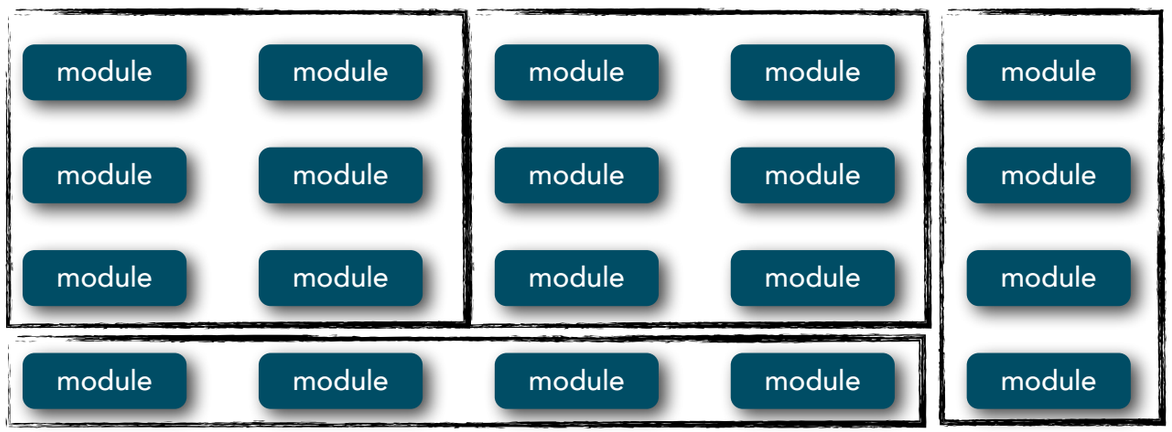
service component granularity



fine-grained service components may lead to too much orchestration and inter-dependency between service components

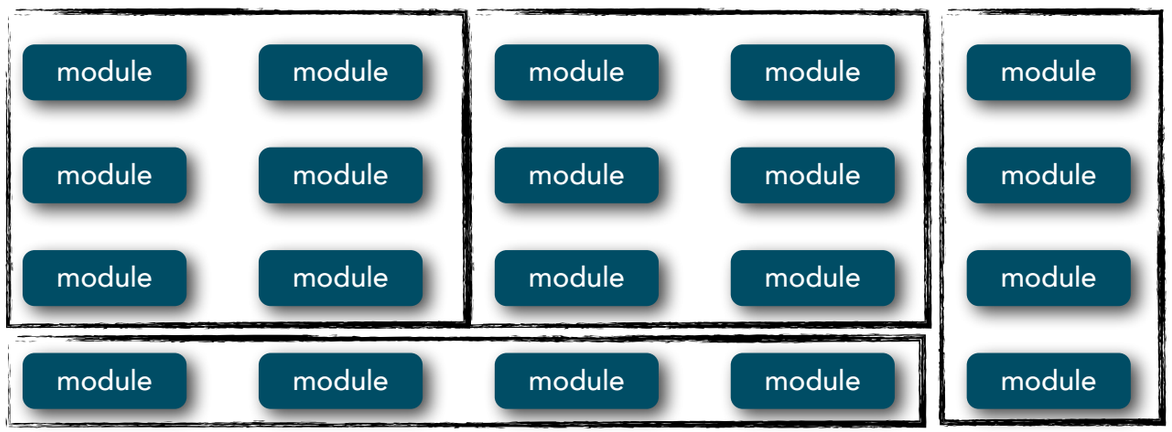
Migration Challenges

service component granularity



Migration Challenges

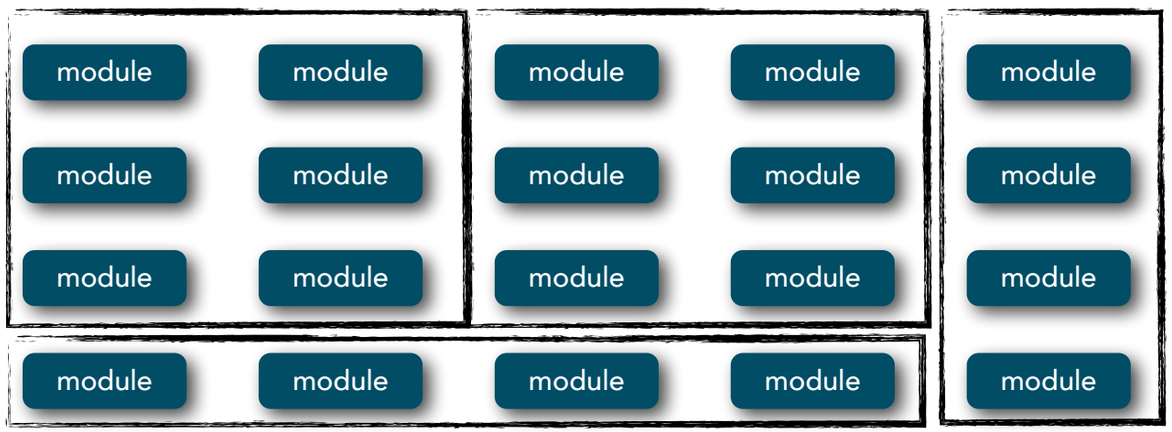
service component granularity



business functionality groupings

Migration Challenges

service component granularity

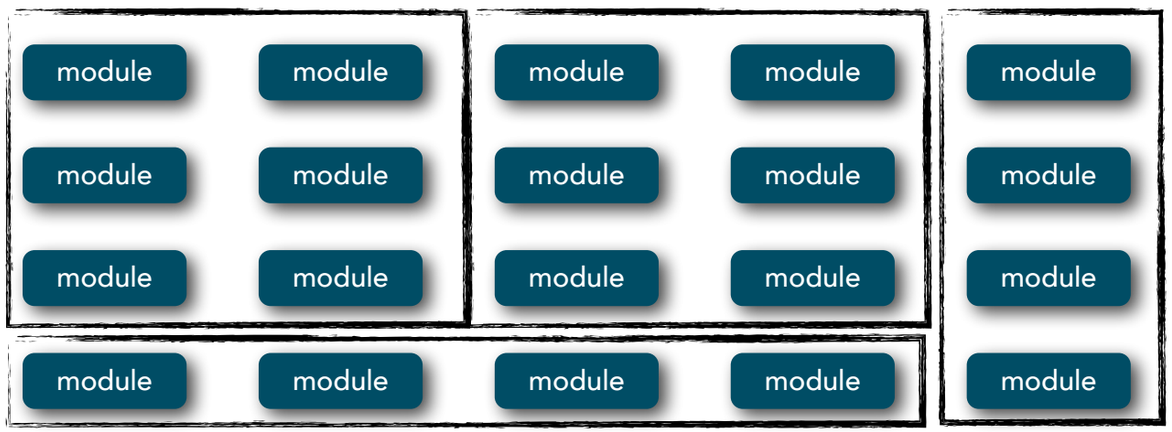


business functionality groupings

transactional boundaries

Migration Challenges

service component granularity



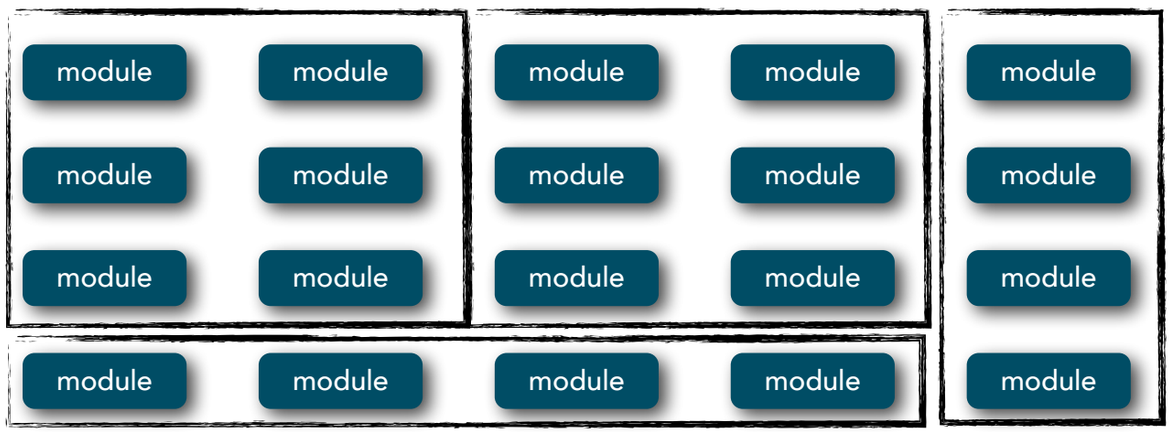
business functionality groupings

transactional boundaries

deployment goals

Migration Challenges

service component granularity



business functionality groupings

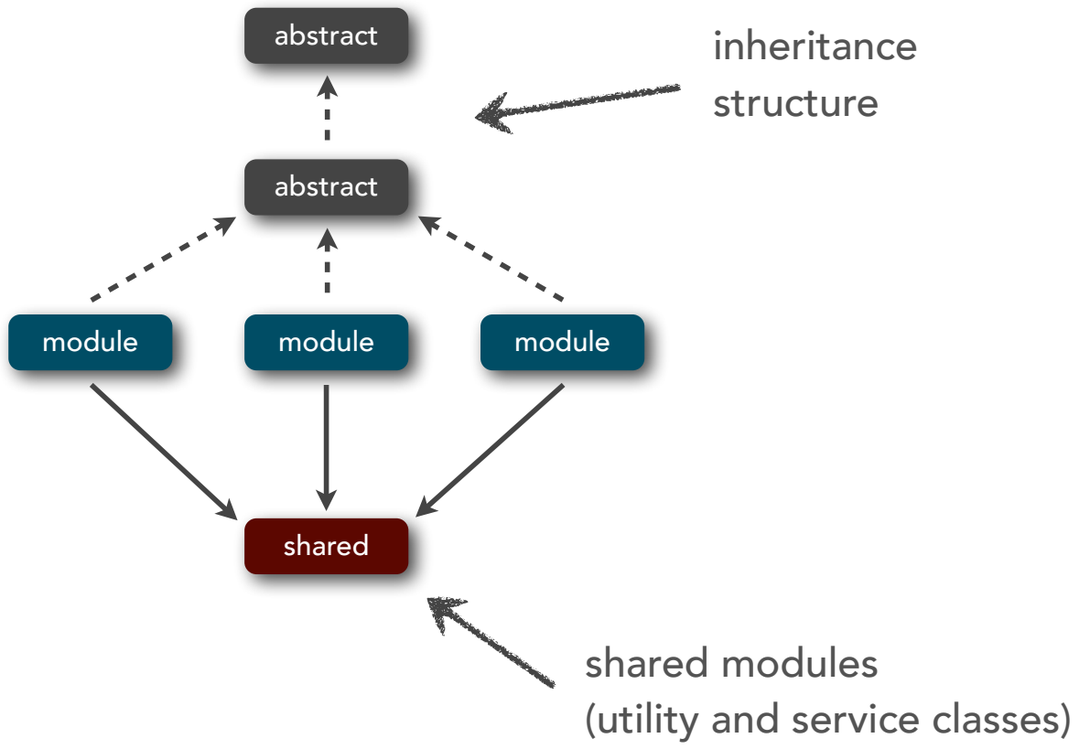
transactional boundaries

deployment goals

scalability needs

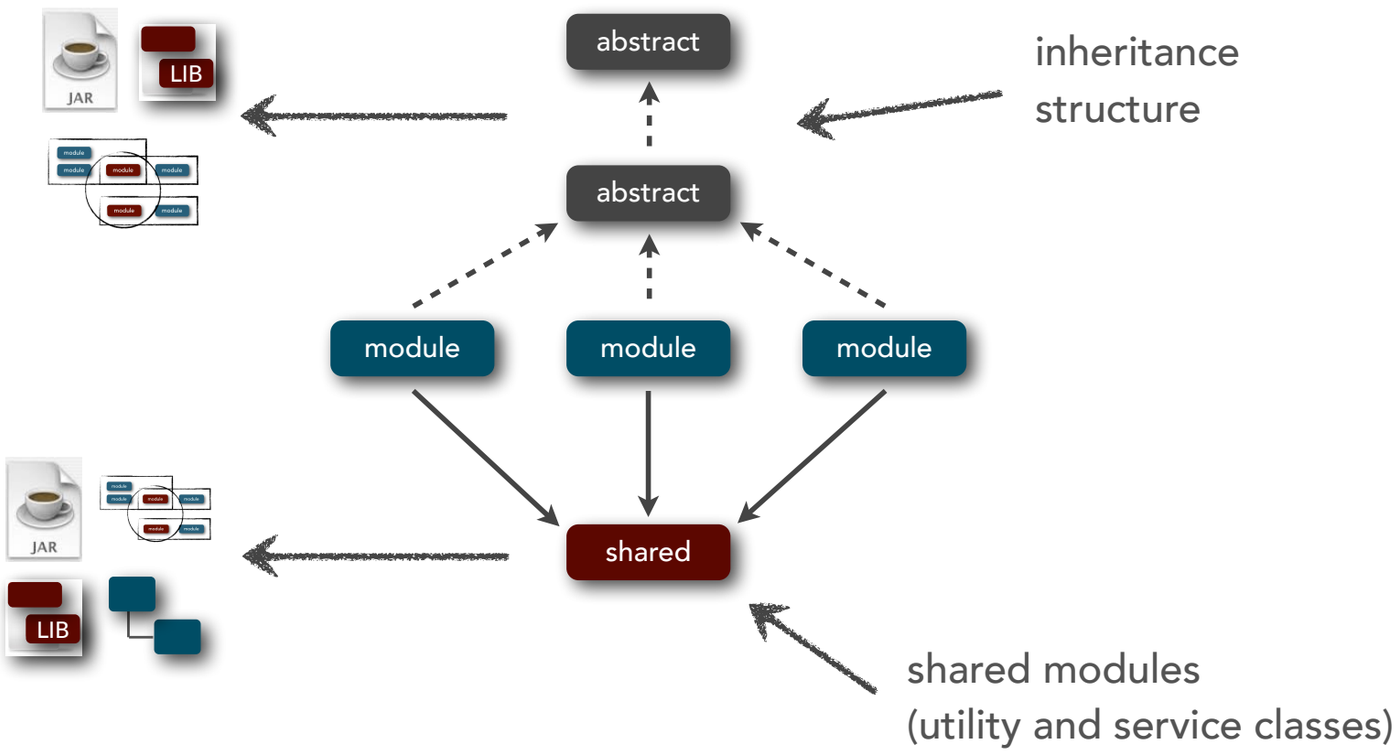
Migration Challenges

shared components and object hierarchies



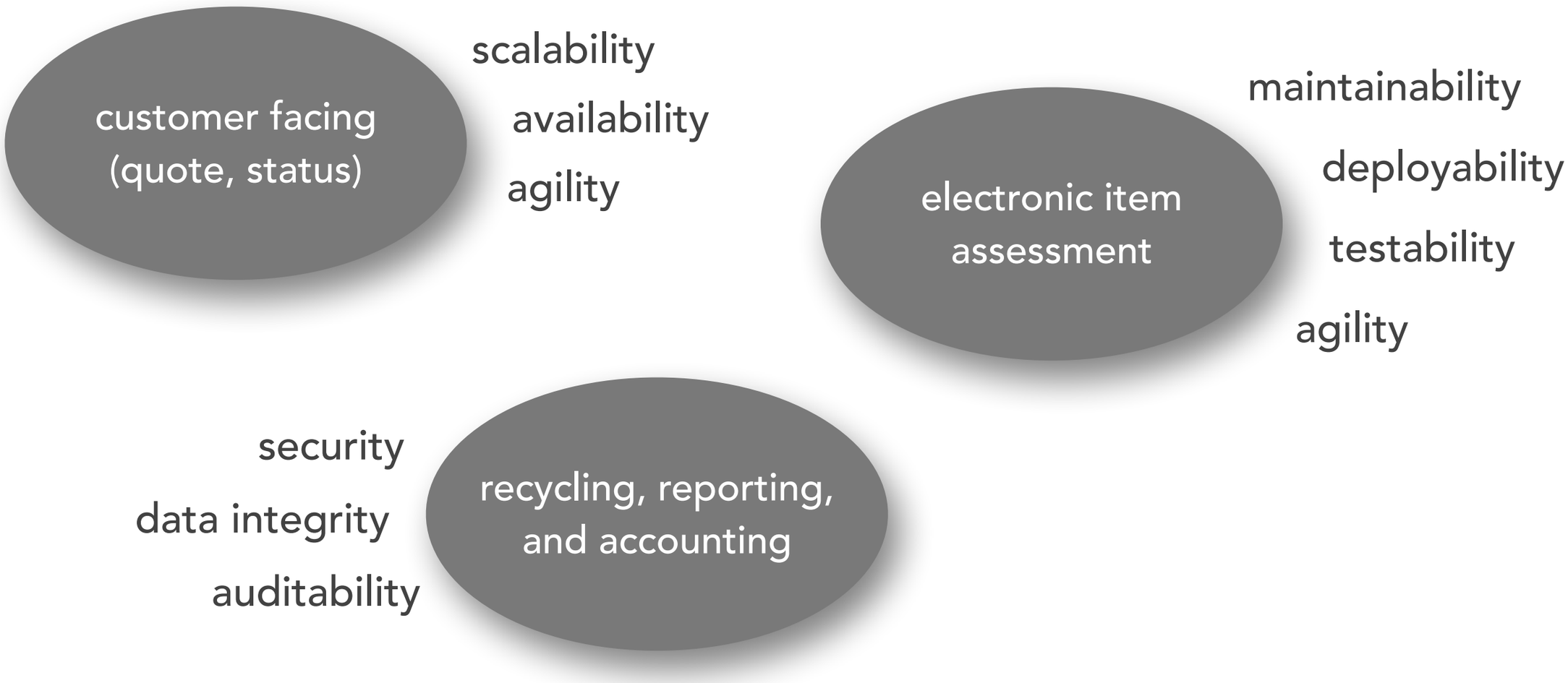
Migration Challenges

shared components and object hierarchies

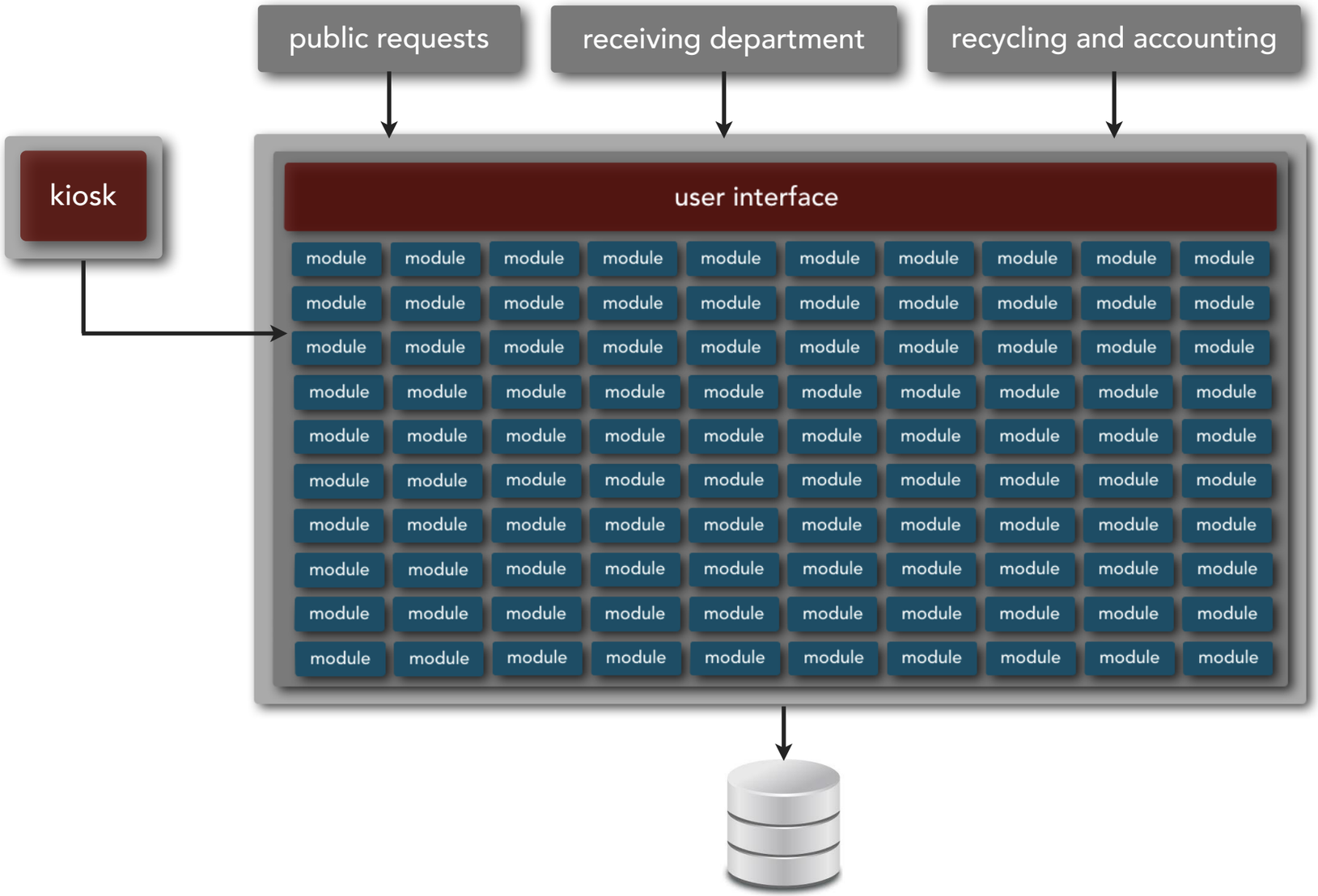


architectural quantum

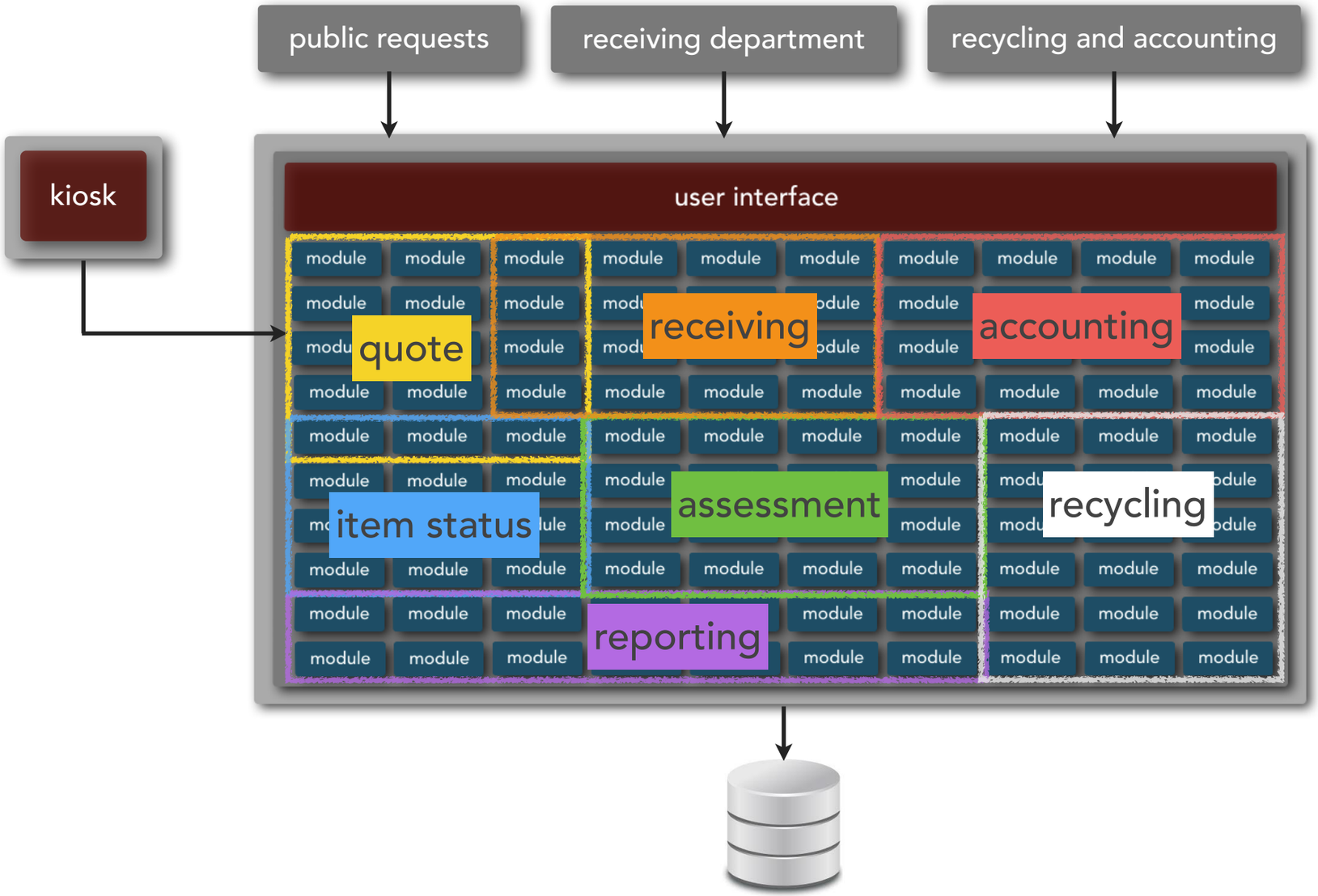
electronics recycling



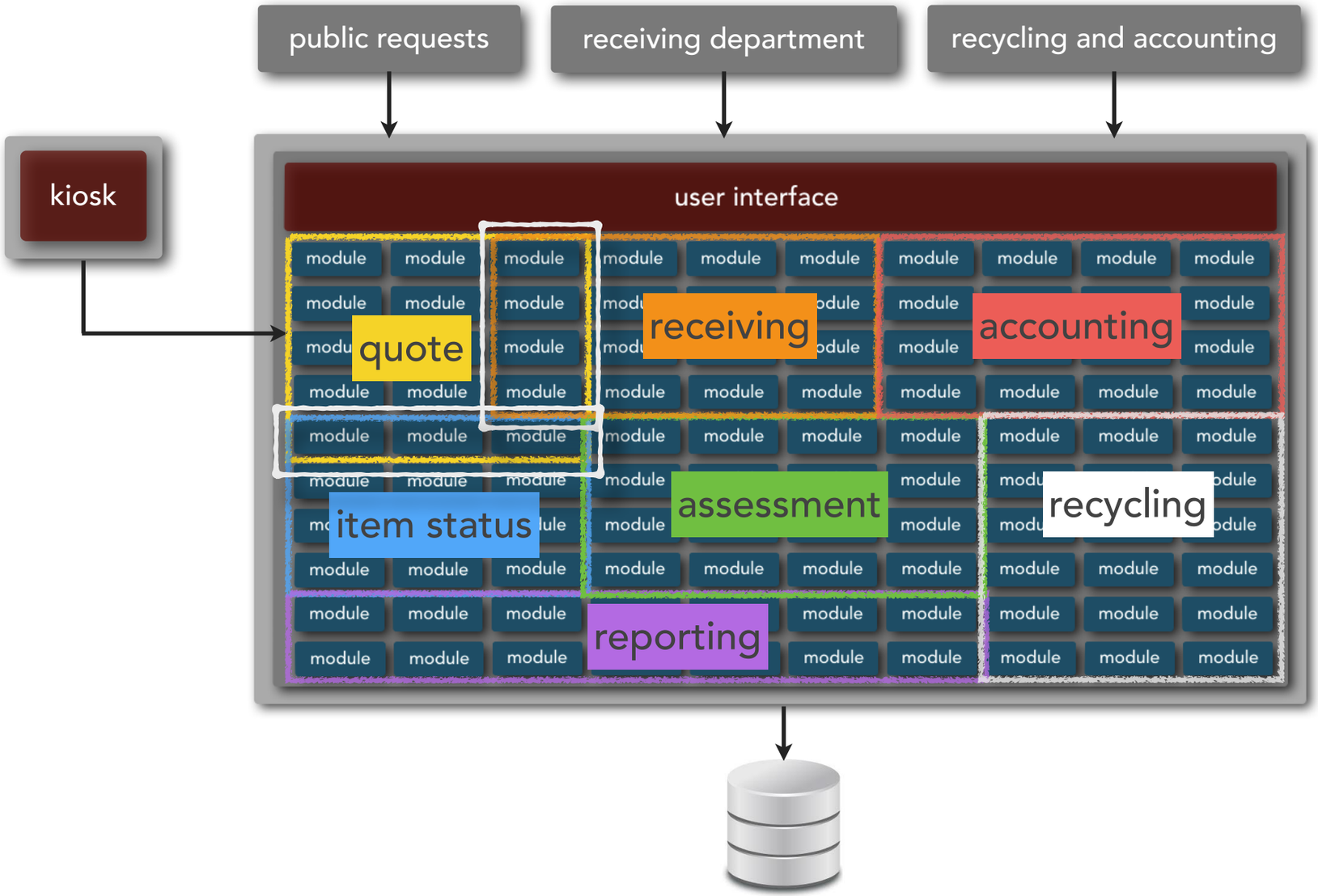
Electronics Recycling Application



Electronics Recycling Application

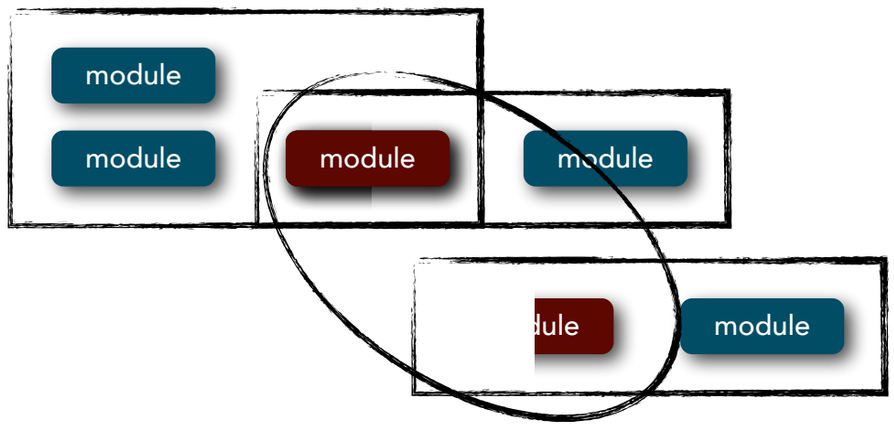


Electronics Recycling Application



Migration Challenges

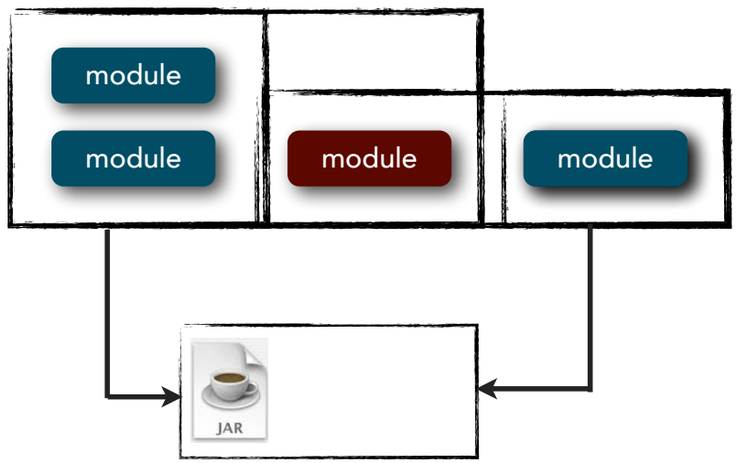
shared components and object hierarchies



developers can *split* modules....

Migration Challenges

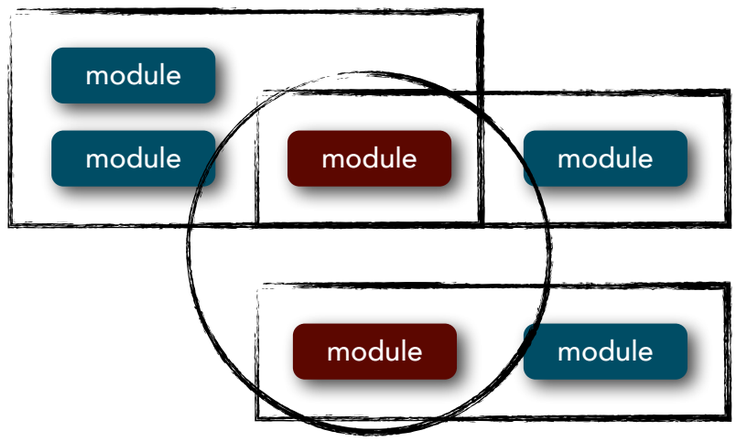
shared components and object hierarchies



…or move into a shared library or jar file...

Migration Challenges

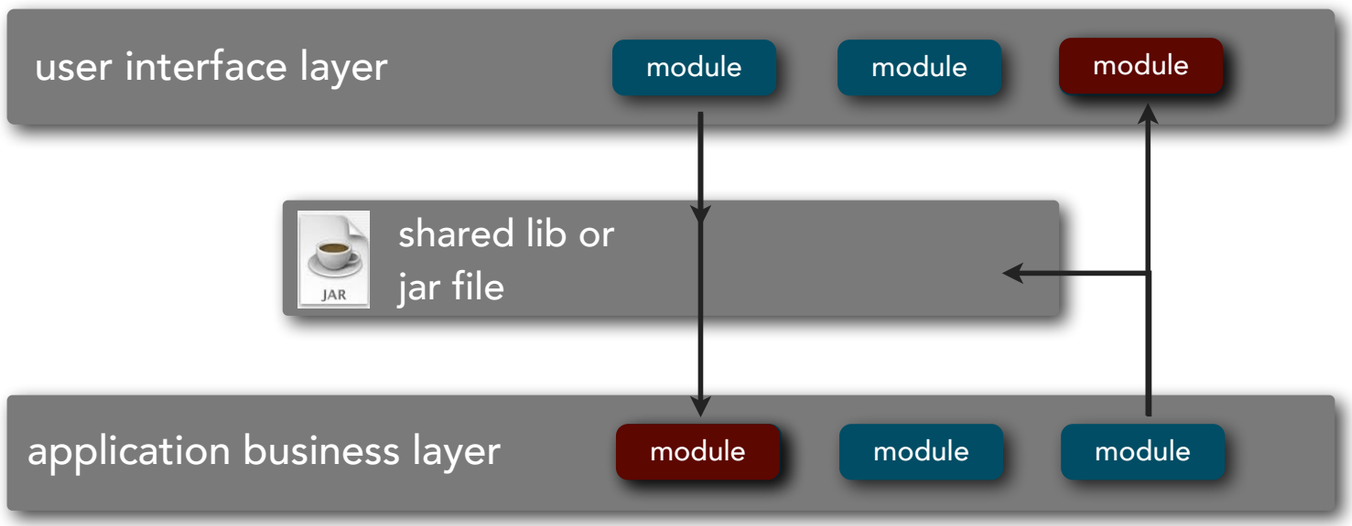
shared components and object hierarchies



…or replicate in each service component

Migration Challenges

shared components and object hierarchies

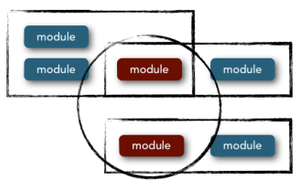


Migration Challenges

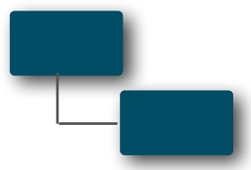
shared component techniques

jar /dll
(compile or runtime)

shared library
(compile time)

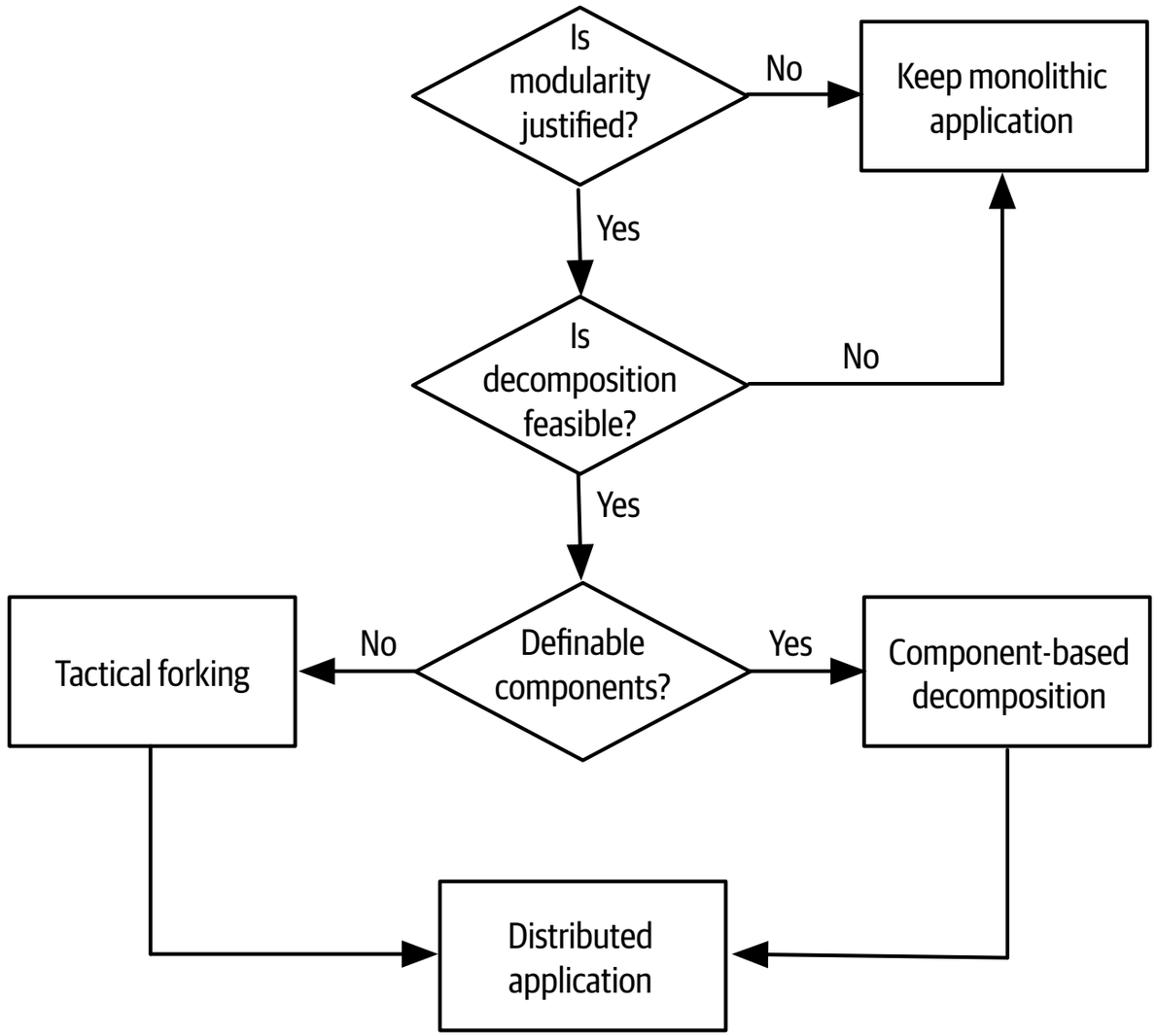


code replication



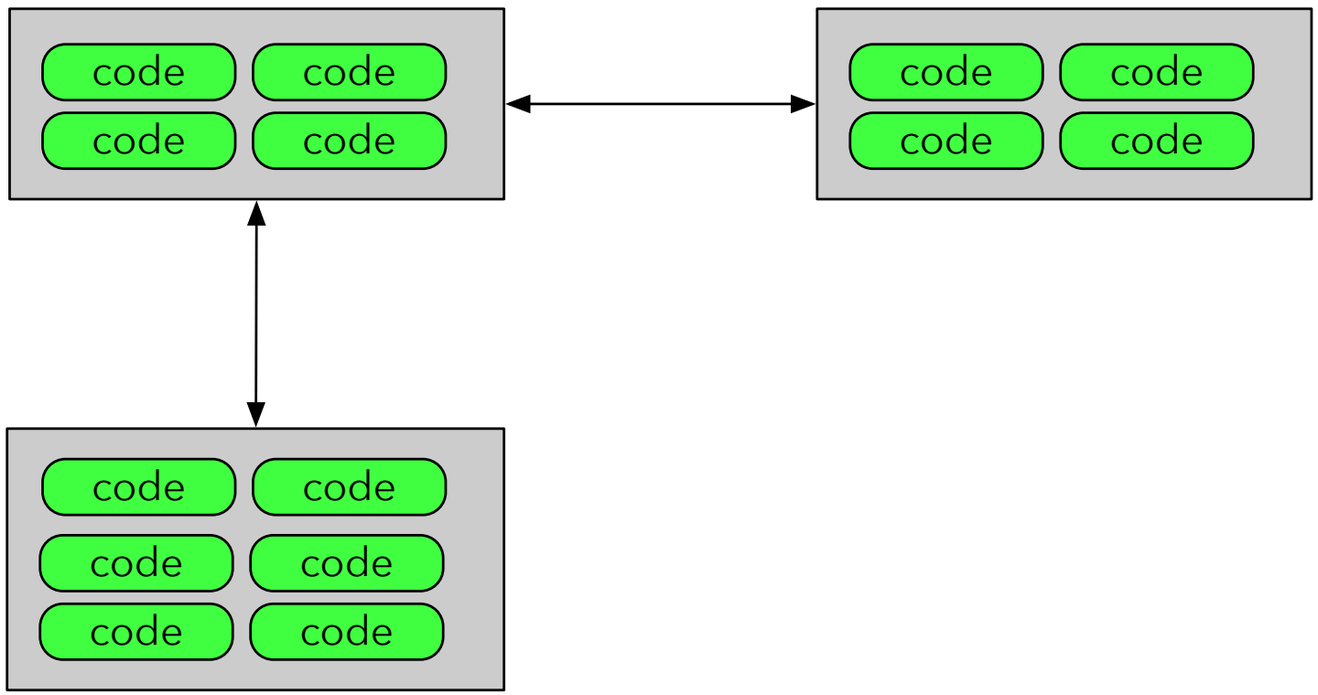
remote services

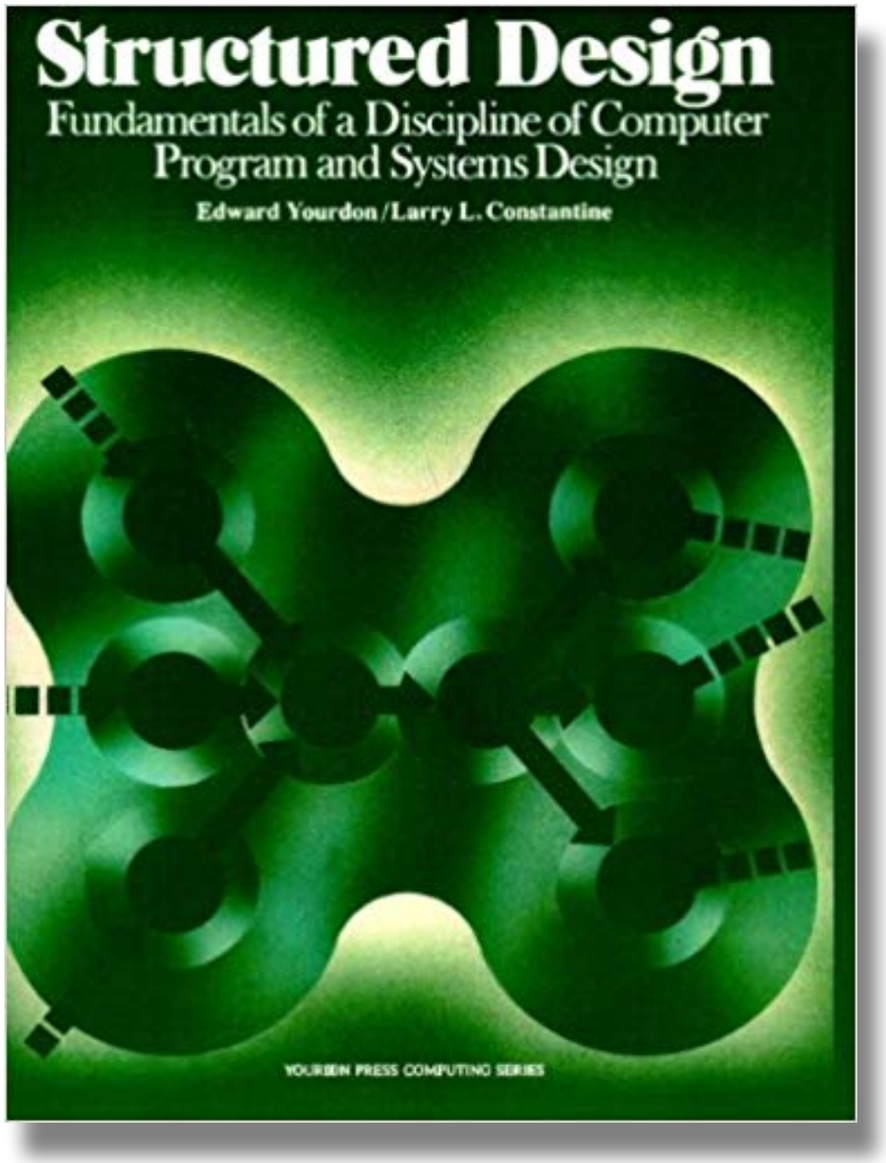
architectural modularity



component coupling

the extent to which components know about each other





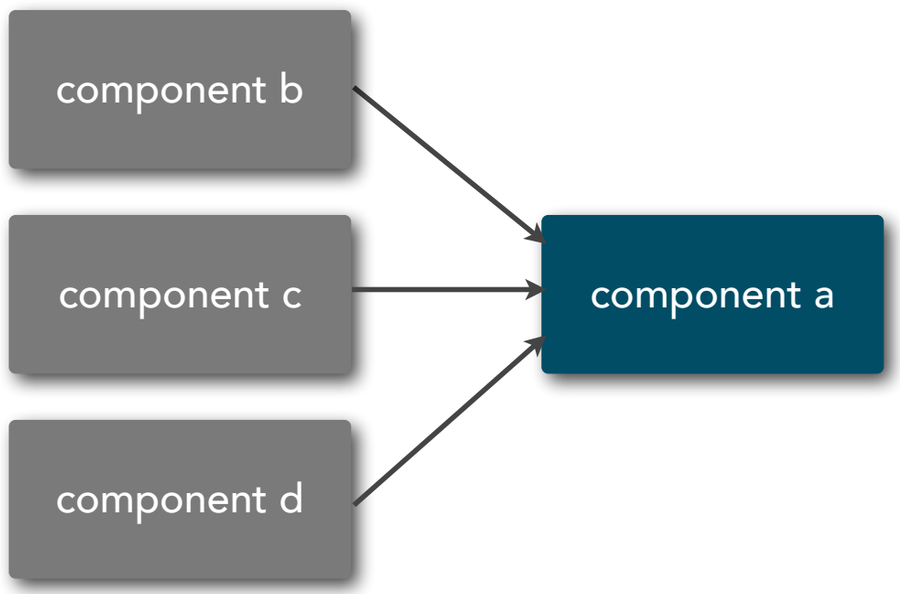
Edward Yourdon & Larry Constantine

1979

component coupling

afferent coupling

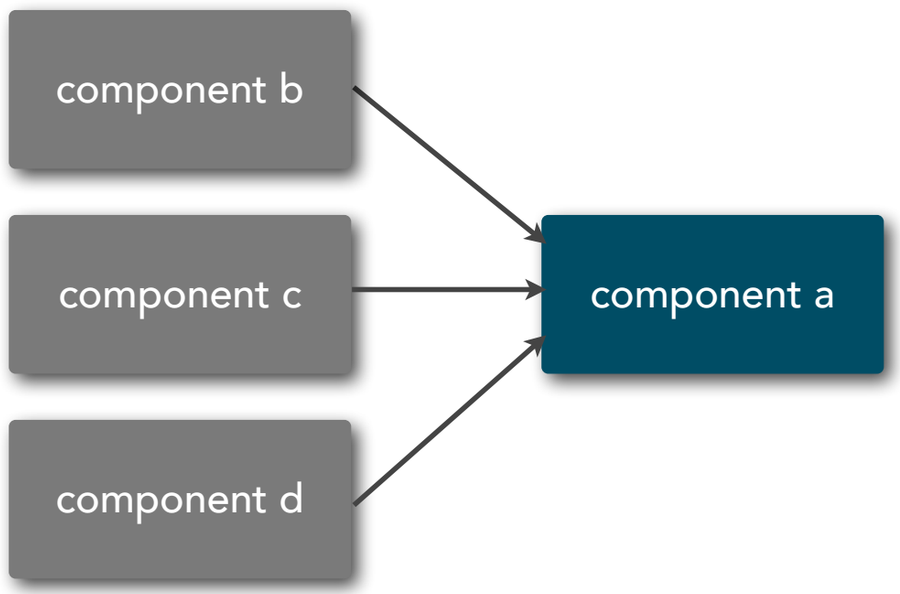
the degree to which other components are dependent on the target component



component coupling

afferent coupling

the degree to which other components are dependent on the target component

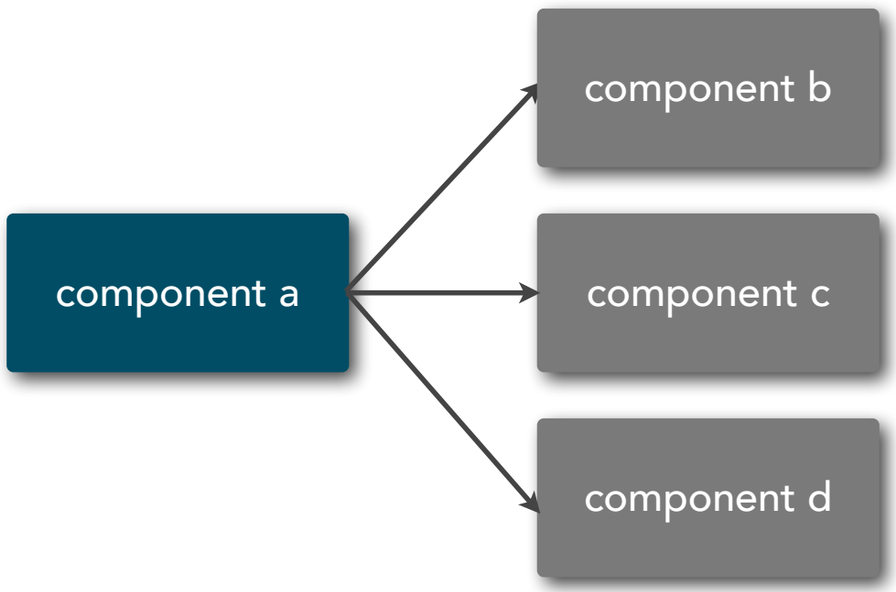


incoming

component coupling

efferent coupling

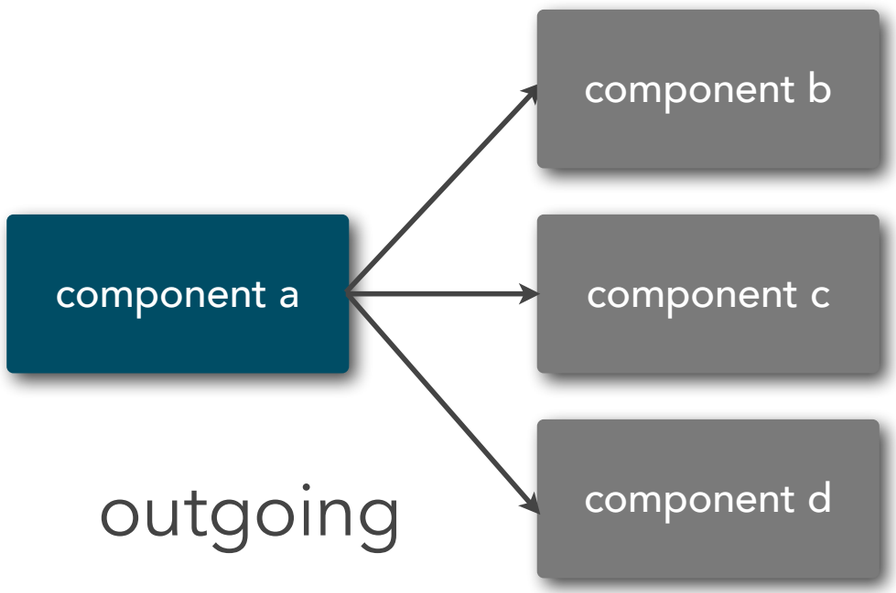
the degree to which the target component
is dependent on other components



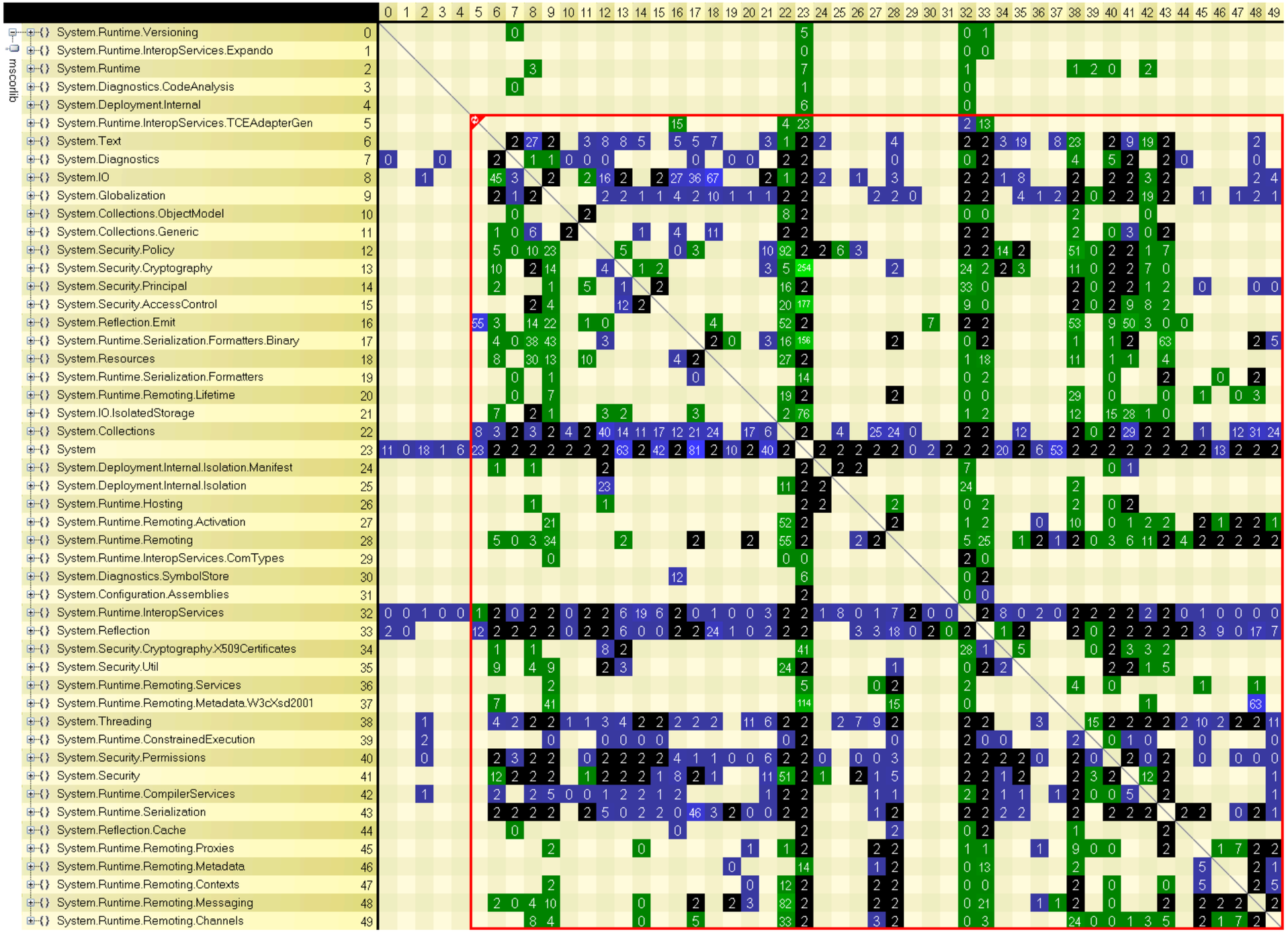
component coupling

efferent coupling

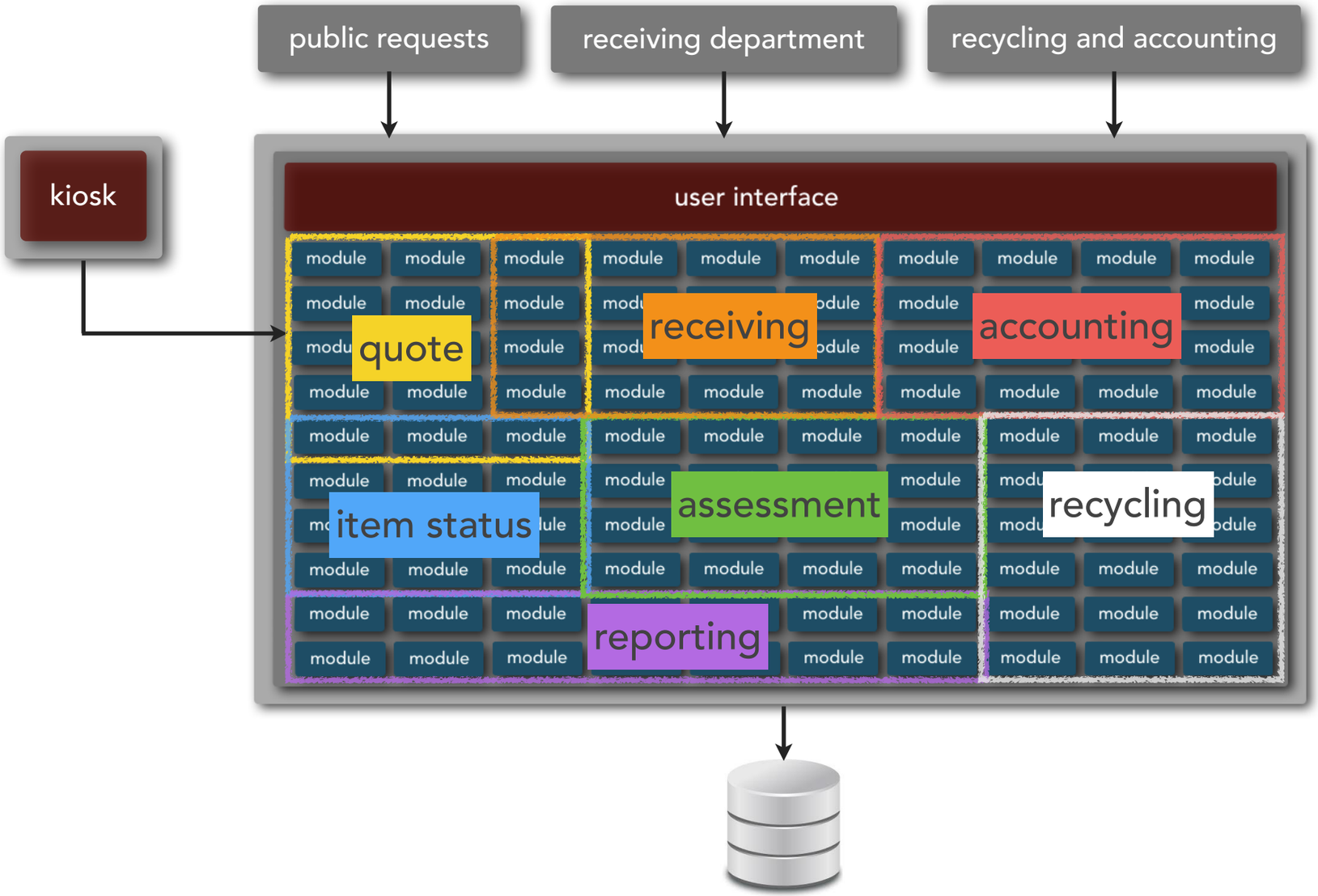
the degree to which the target component
is dependent on other components



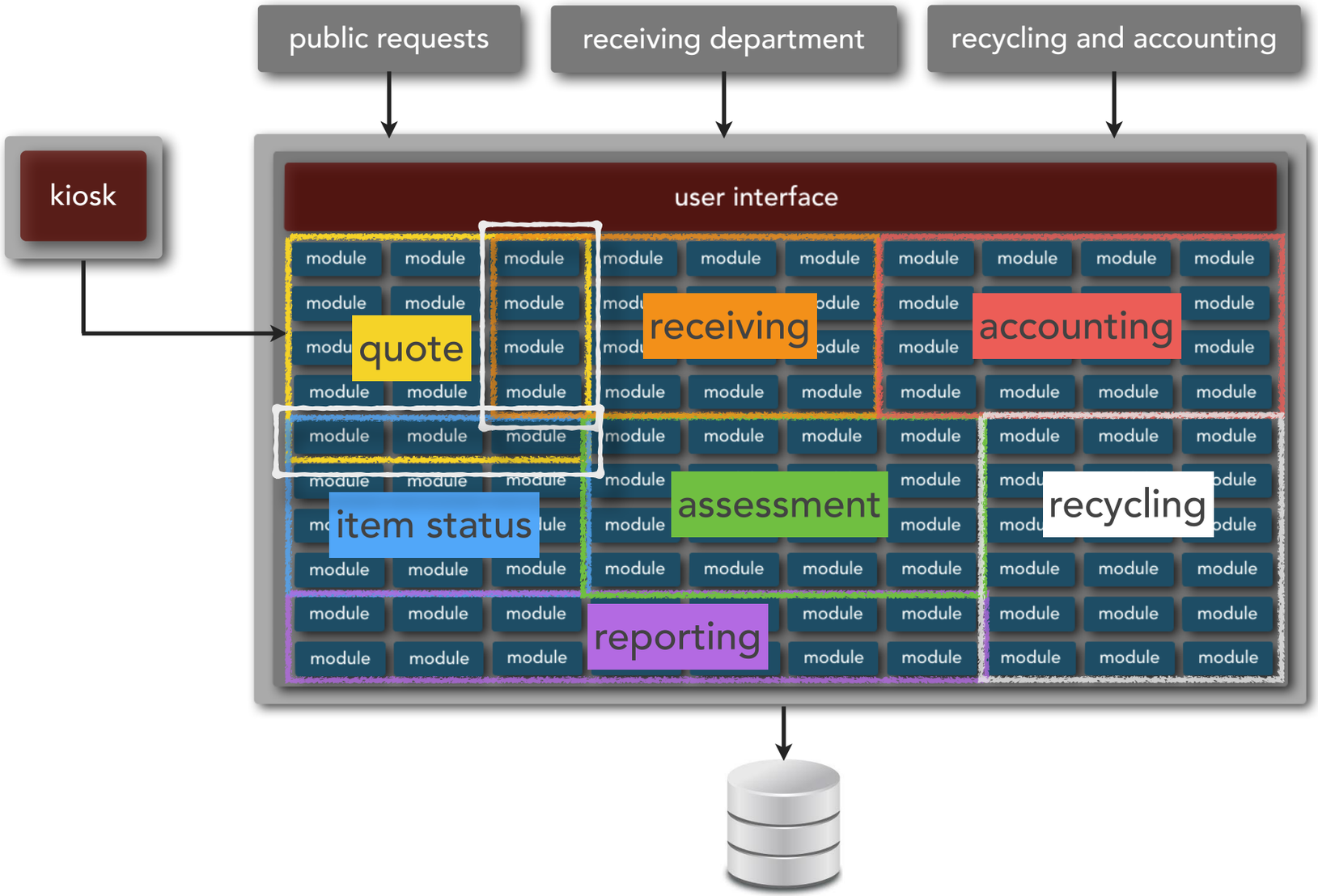
dependency matrix



Electronics Recycling Application



Electronics Recycling Application



derived metrics

https://en.wikipedia.org/wiki/Robert_C._Martin
http://cleancoder.com/

abstractness

$$A = \frac{\sum m^a}{\sum m^c + \sum m^a}$$

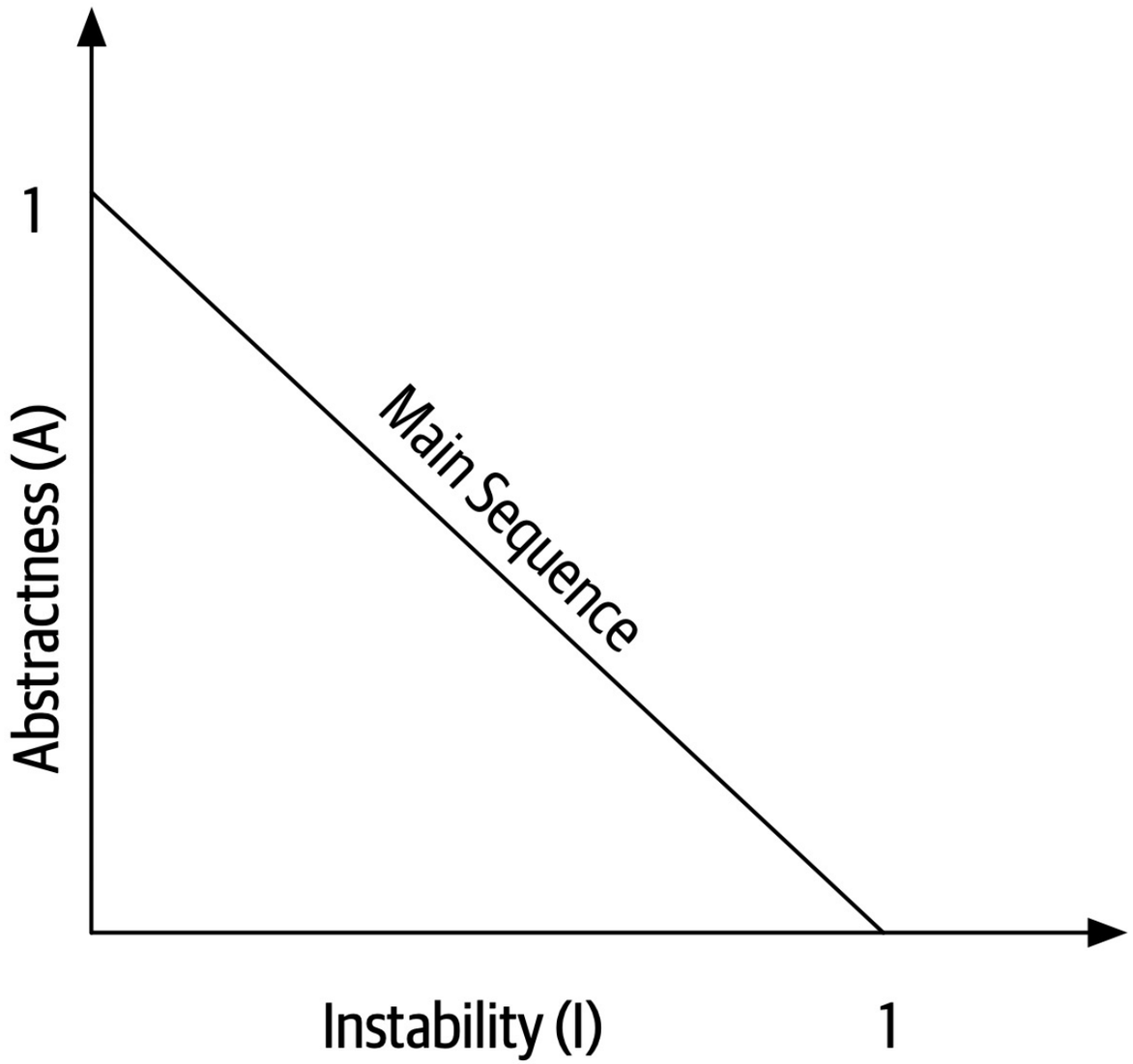
where:
m^a => abstract elements
m^c => concrete elements

instability

$$I = \frac{C^e}{C^e + C^a}$$

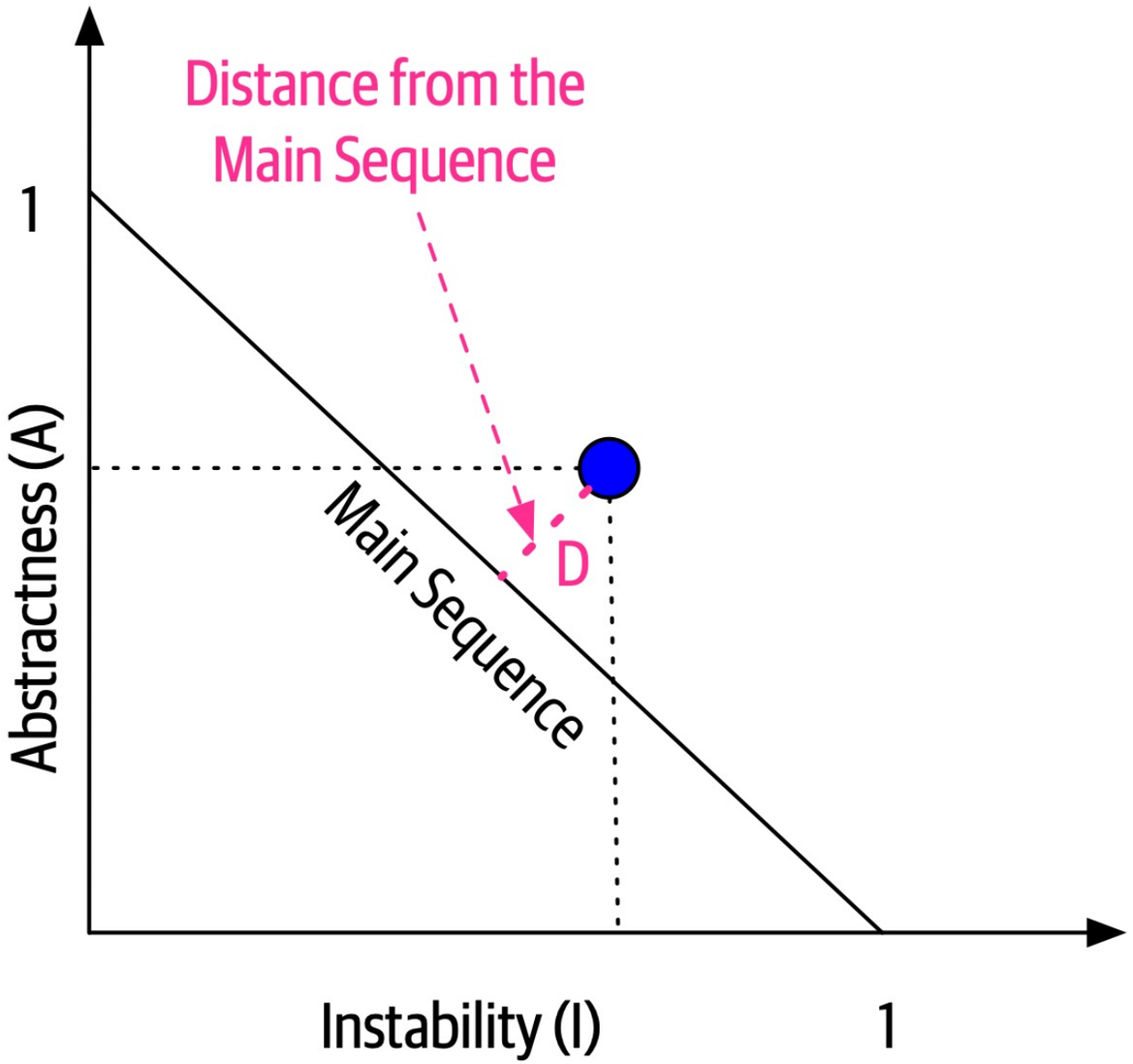
where:
c^e => efferent coupling
c^a => afferent coupling

distance from the main sequence



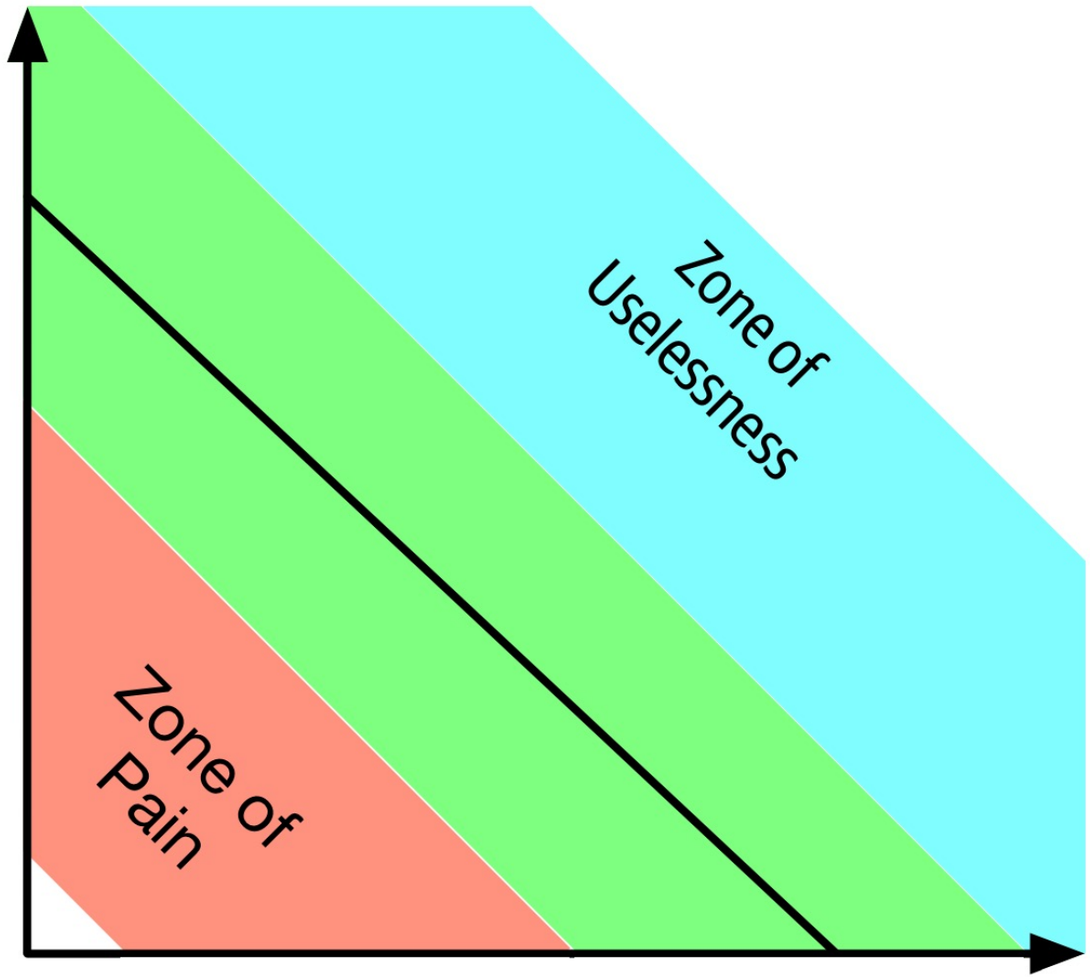
$$D = |A + I - 1|$$

distance from the main sequence



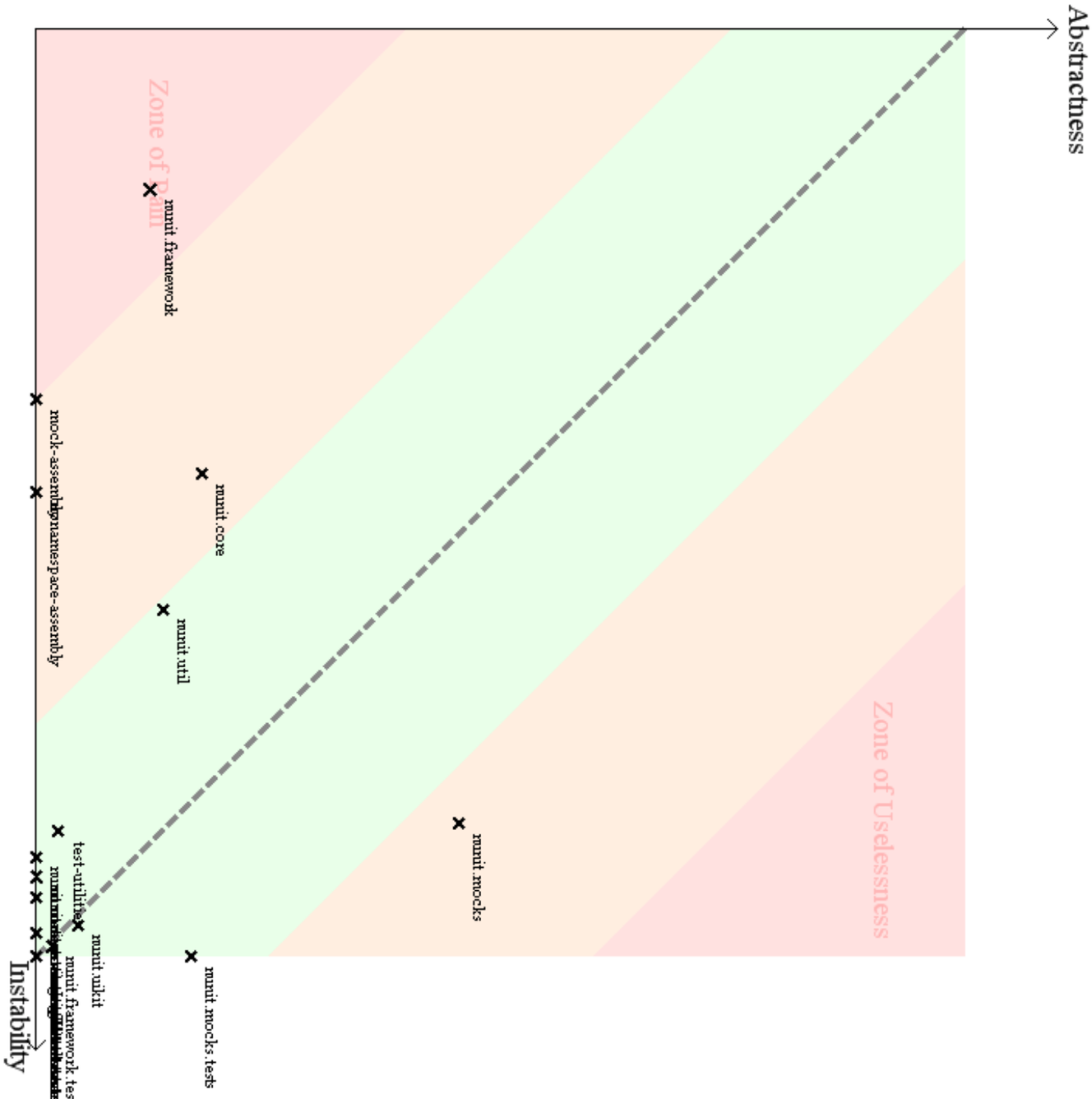
$$D = |A + I - 1|$$

distance from the main sequence

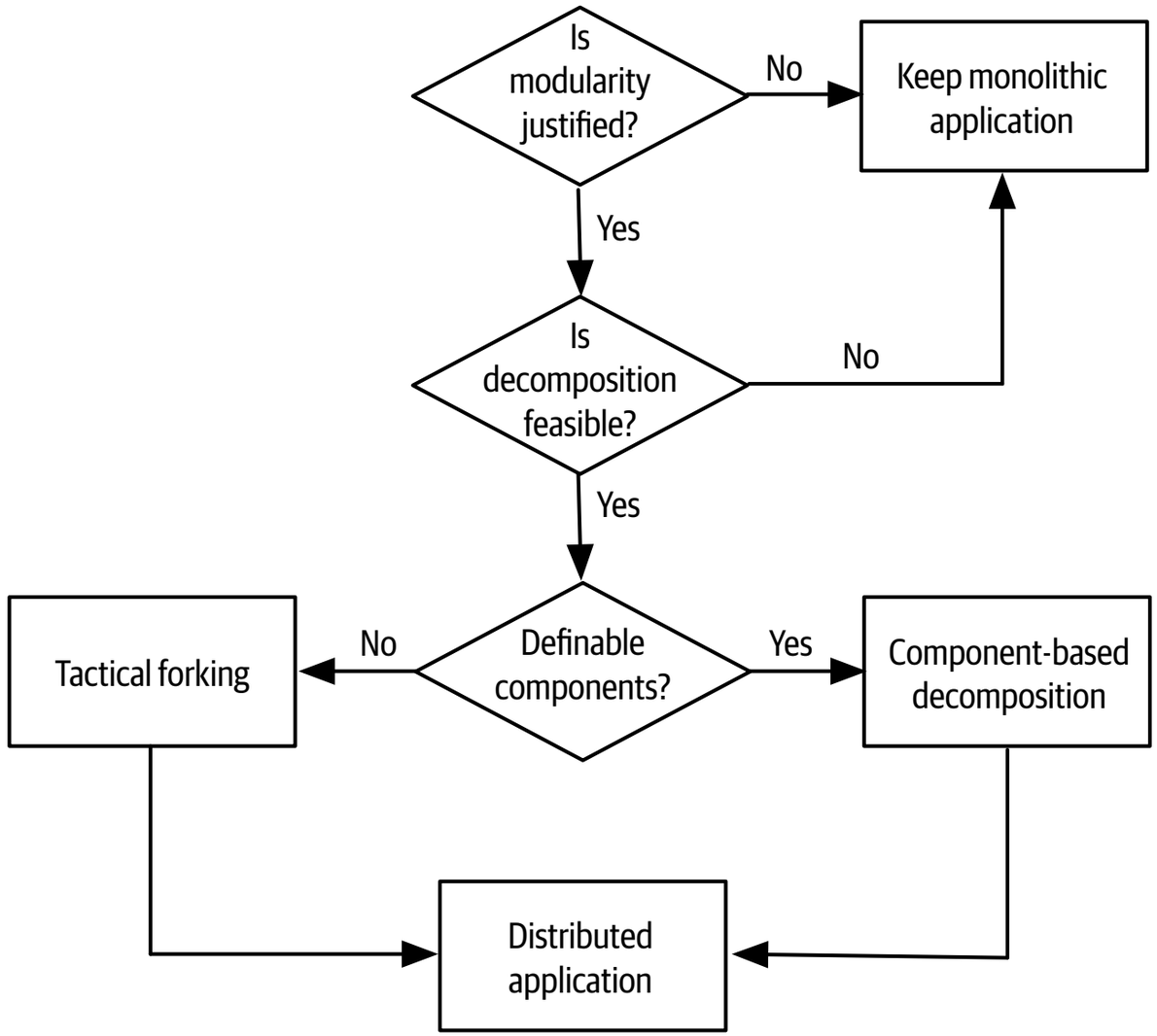


$$D = |A + I - 1|$$

distance from the main sequence

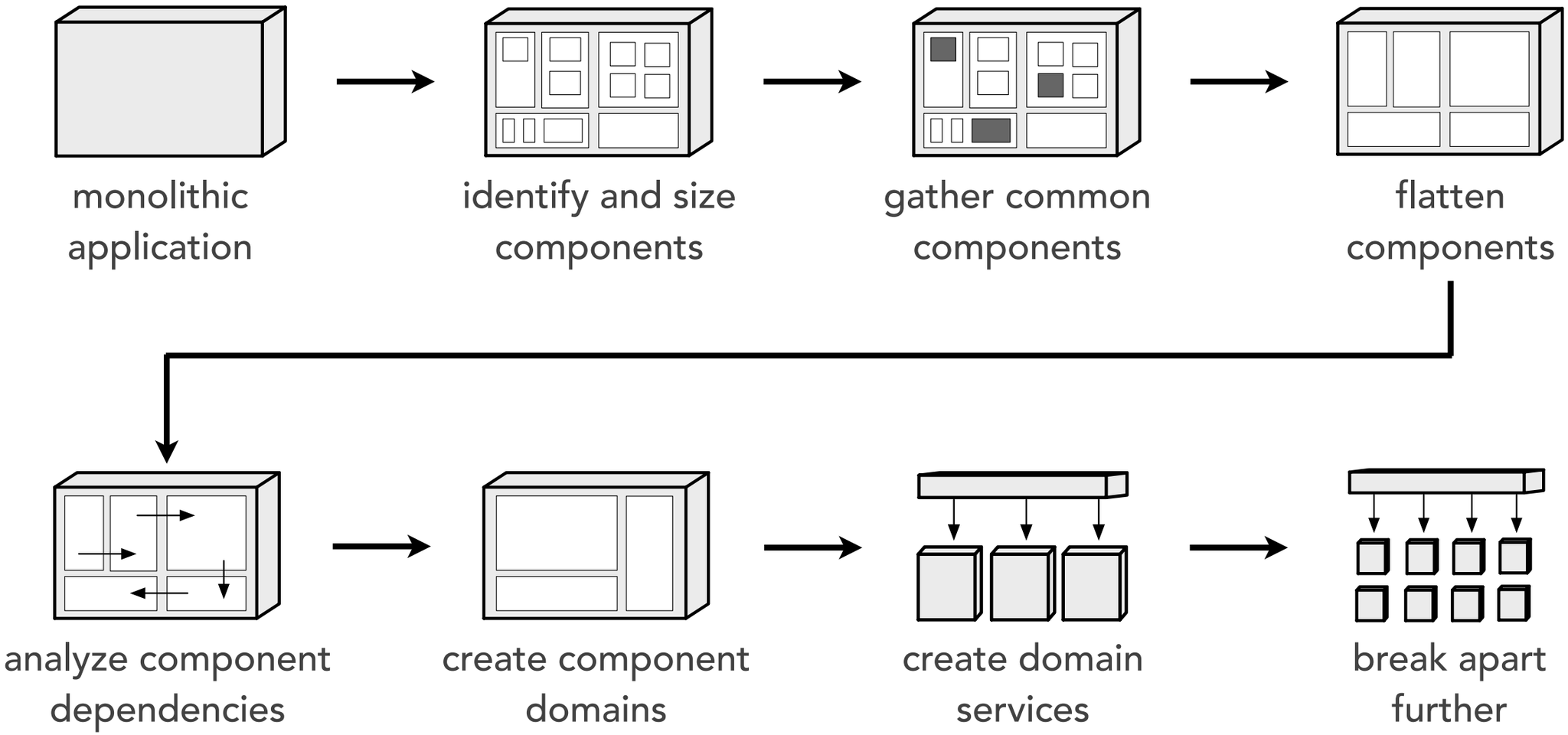


architectural modularity



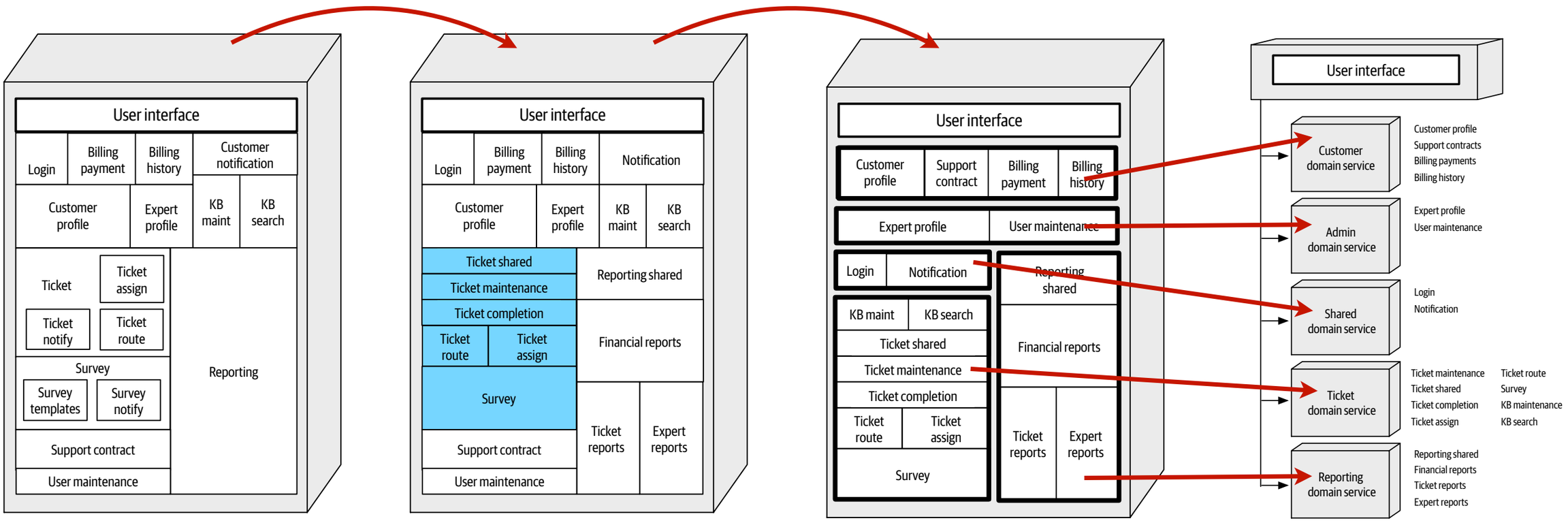
architectural modularity

component-based decomposition



architectural modularity

component-based decomposition



identify & size components pattern

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

standard deviation

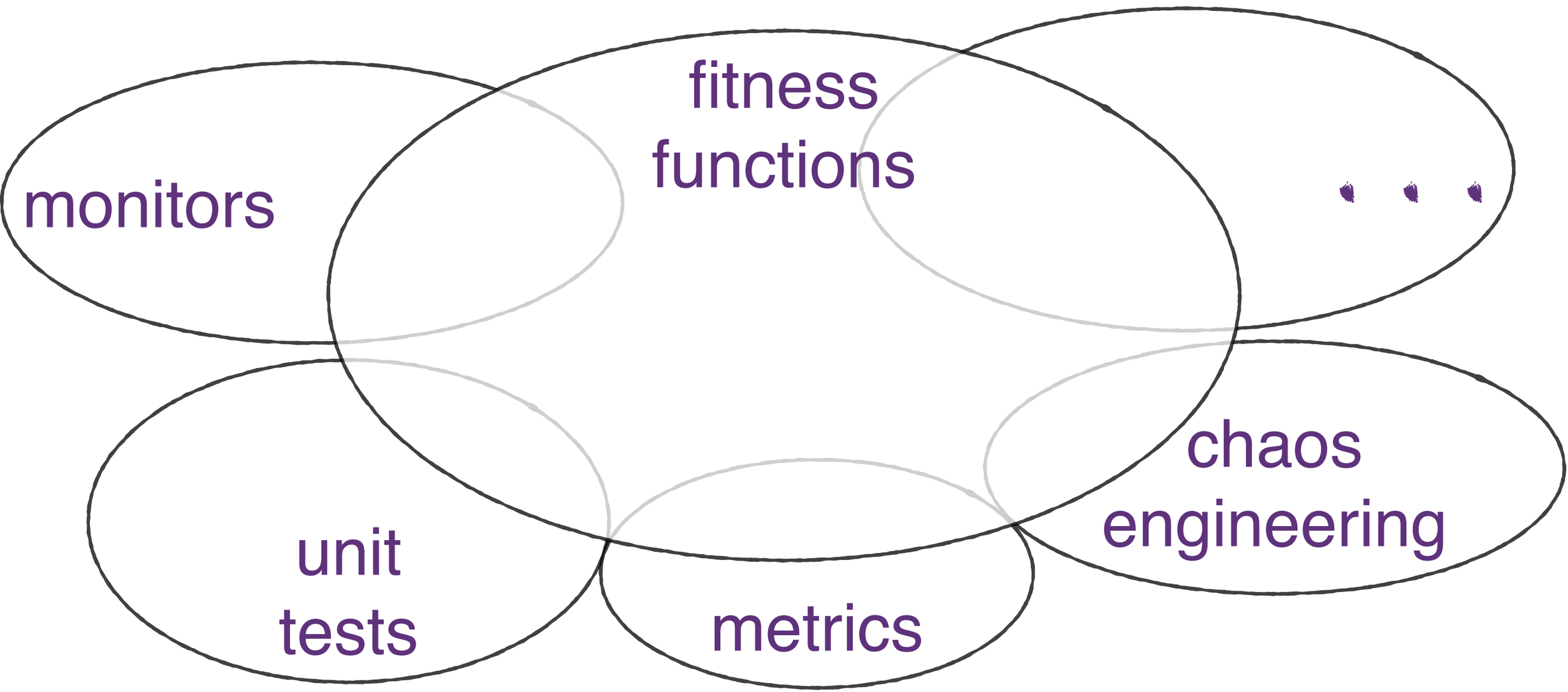
```
# Walk the directory structure, creating namespaces for each complete path
LIST component_list = identify_components(root_directory)

# Walk through all of the source code to accumulate total statements and number
# of statements per component
SET total_statements TO 0
MAP component_size_map
FOREACH component IN component_list {
  num_statements = accumulate_statements(component)
  ADD num_statements TO total_statements
  ADD component,num_statements TO component_size_map
}

# Calculate the standard deviation
SET square_diff_sum TO 0
num_components = get_num_entries(component_list)
mean = total_statements / num_components
FOREACH component,size IN component_size_map {
  diff = size - mean
  ADD square(diff) TO square_diff_sum
}
std_dev = square_root(square_diff_sum / (num_components - 1))

# For each component calculate the number of standard deviations from the
# mean. Send an alert if greater than 3
FOREACH component,size IN component_size_map {
  diff_from_mean = absolute_value(size - mean);
  num_std_devs = diff_from_mean / std_dev
  IF num_std_devs > 3 {
    send_alert(component, num_std_devs)
  }
}
```

guided

***architectural** fitness function*:

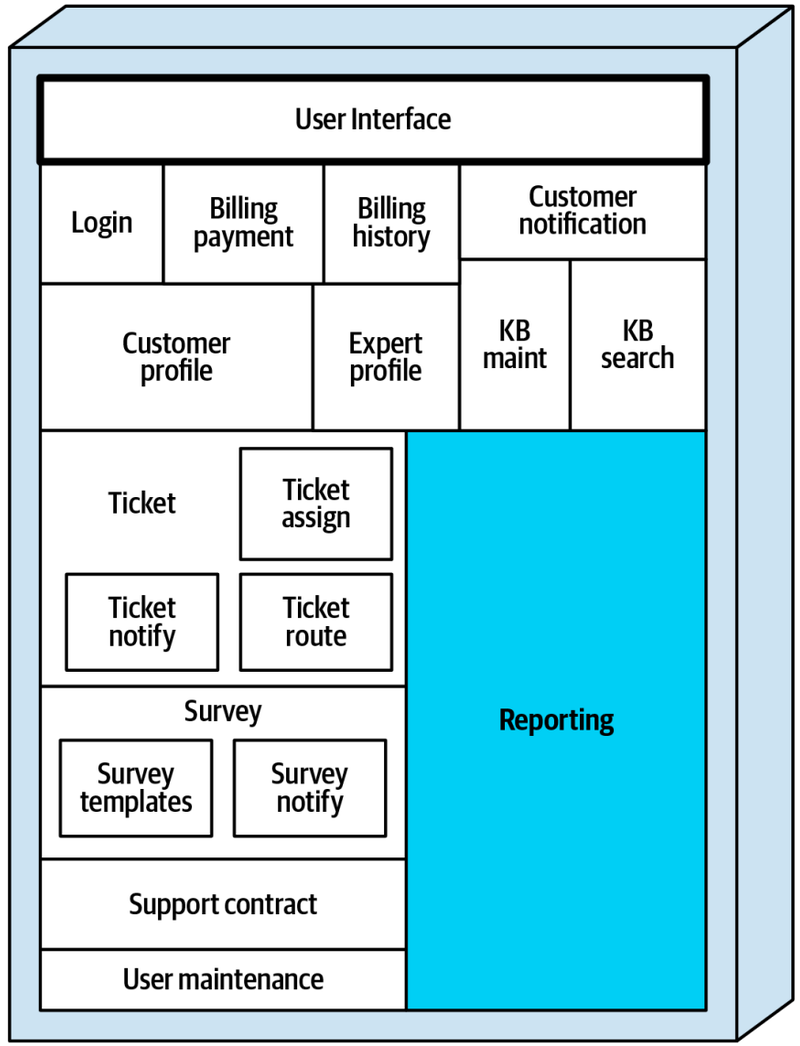
An architectural fitness function provides an objective integrity assessment of some architectural characteristic(s).

Fitness Functions

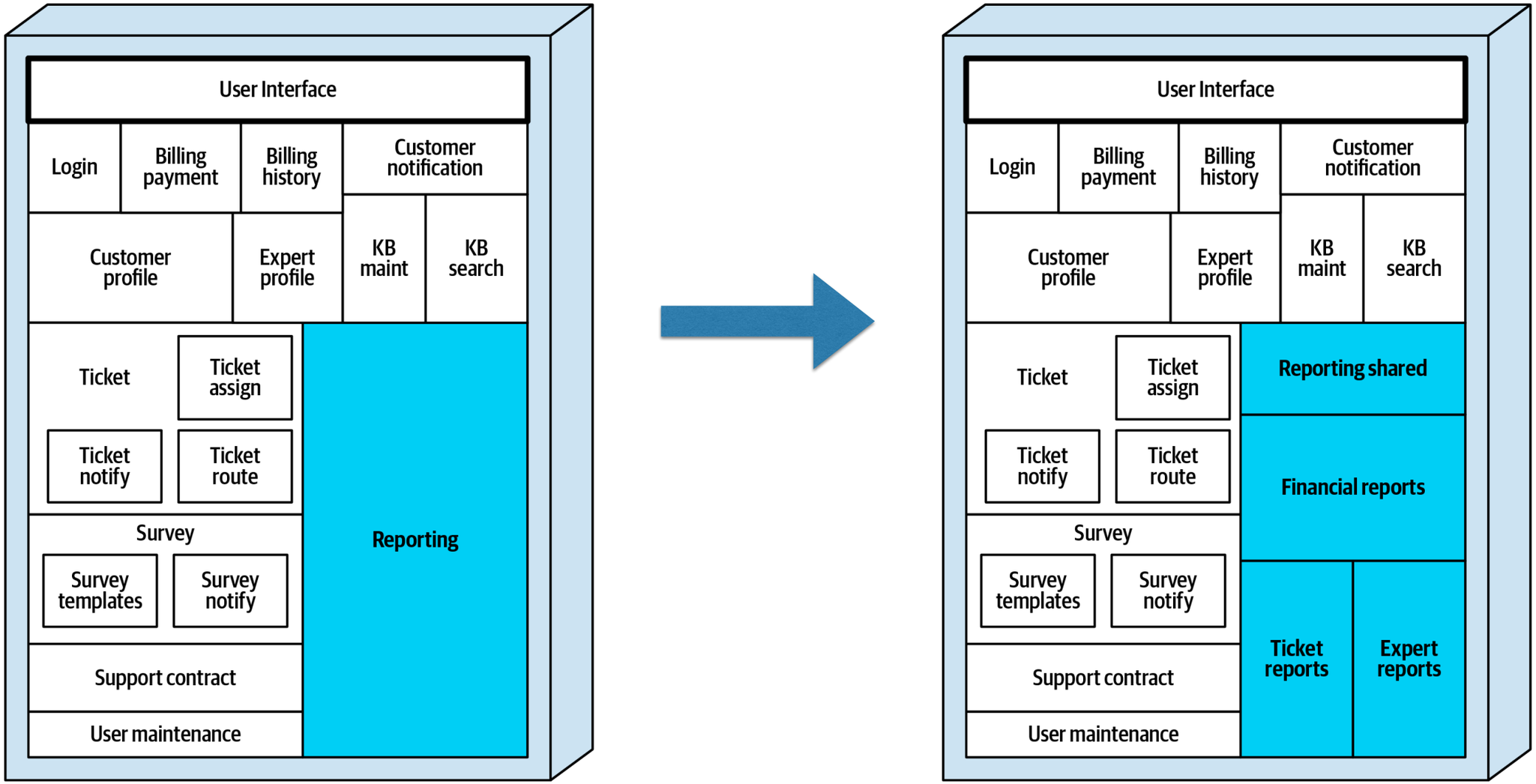


identify & size components pattern

Component name	Component namespace	Percent	Statements	Files
Login	ss.login	2	1865	3
Billing Payment	ss.billing.payment	5	4,312	23
Billing History	ss.billing.history	4	3,209	17
Customer Notification	ss.customer.notification	2	1,433	7
Customer Profile	ss.customer.profile	5	4,012	16
Expert Profile	ss.expert.profile	6	5,099	32
KB Maint	ss.kb.maintenance	2	1,701	14
KB Search	ss.kb.search	3	2,871	4
Reporting	ss.reporting	**33**	**27,765**	**162**
Ticket	ss.ticket	8	7,009	45
Ticket Assign	ss.ticket.assign	9	7,845	14
Ticket Notify	ss.ticket.notify	2	1,765	3
Ticket Route	ss.ticket.route	2	1,468	4
Support Contract	ss.supportcontract	5	4,104	24
Survey	ss.survey	3	2,204	5
Survey Notify	ss.survey.notify	2	1,299	3
Survey Templates	ss.survey.templates	2	1,672	7
User Maintenance	ss.users	4	3,298	12

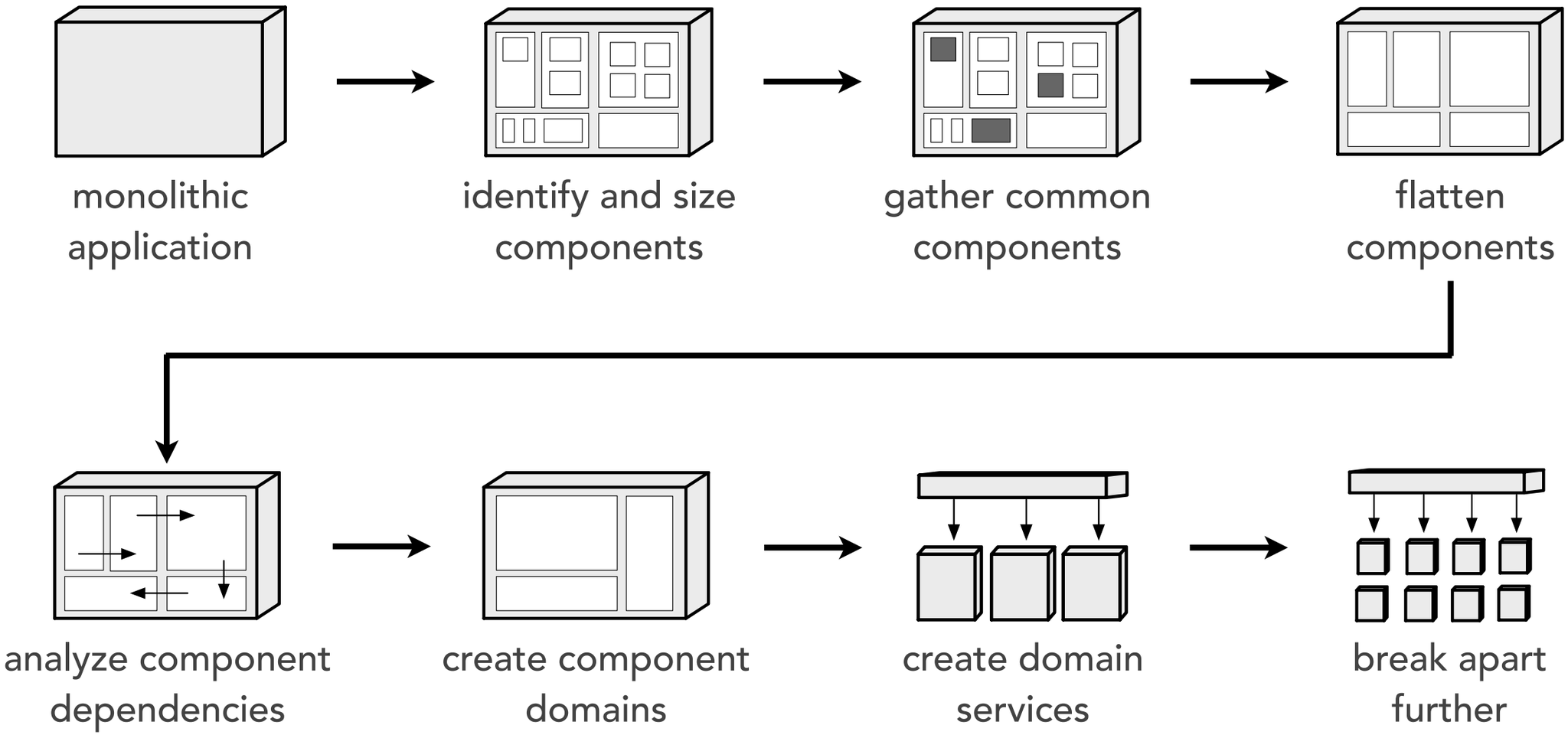


identify & size components pattern

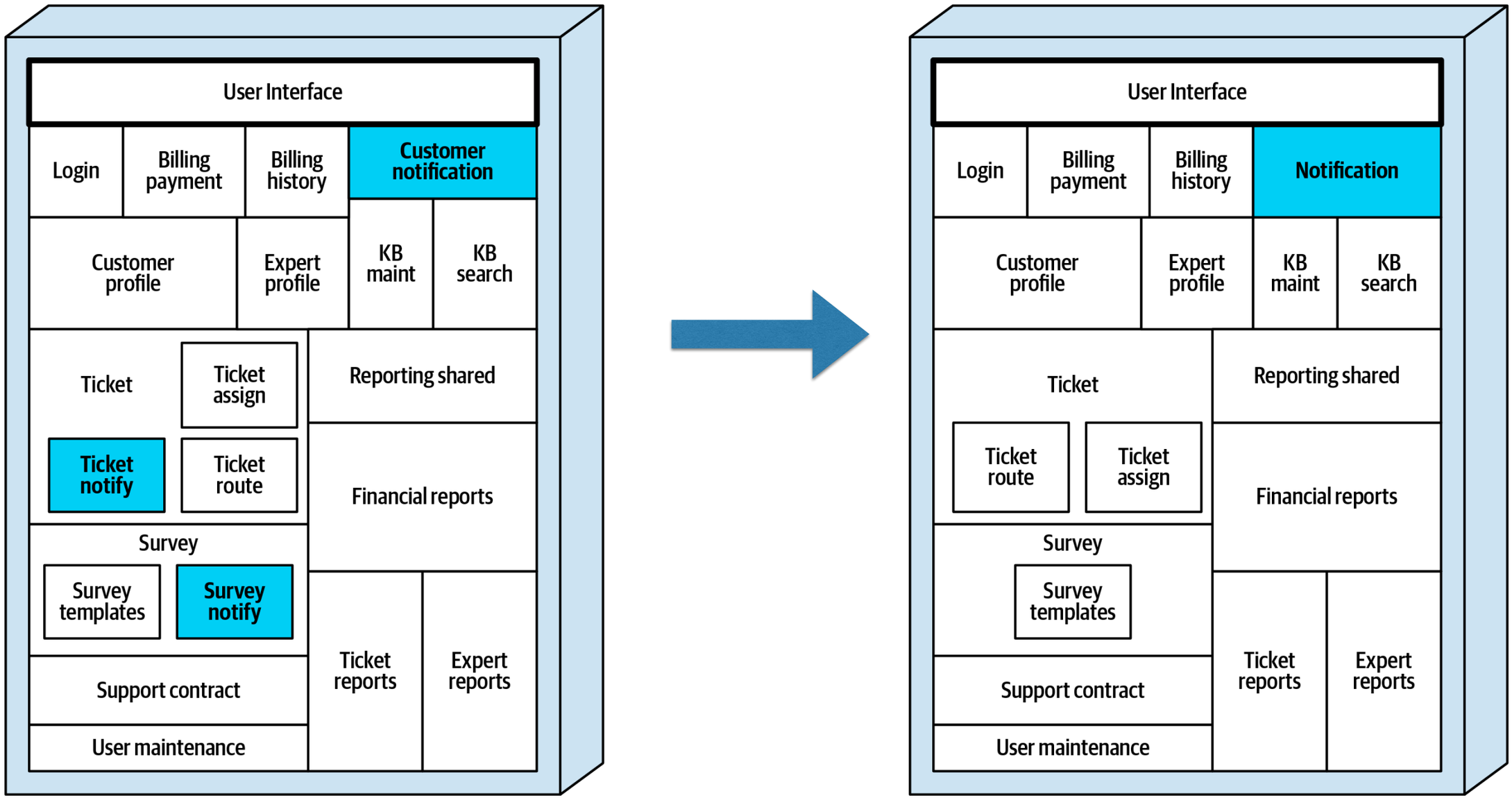


architectural modularity

component-based decomposition

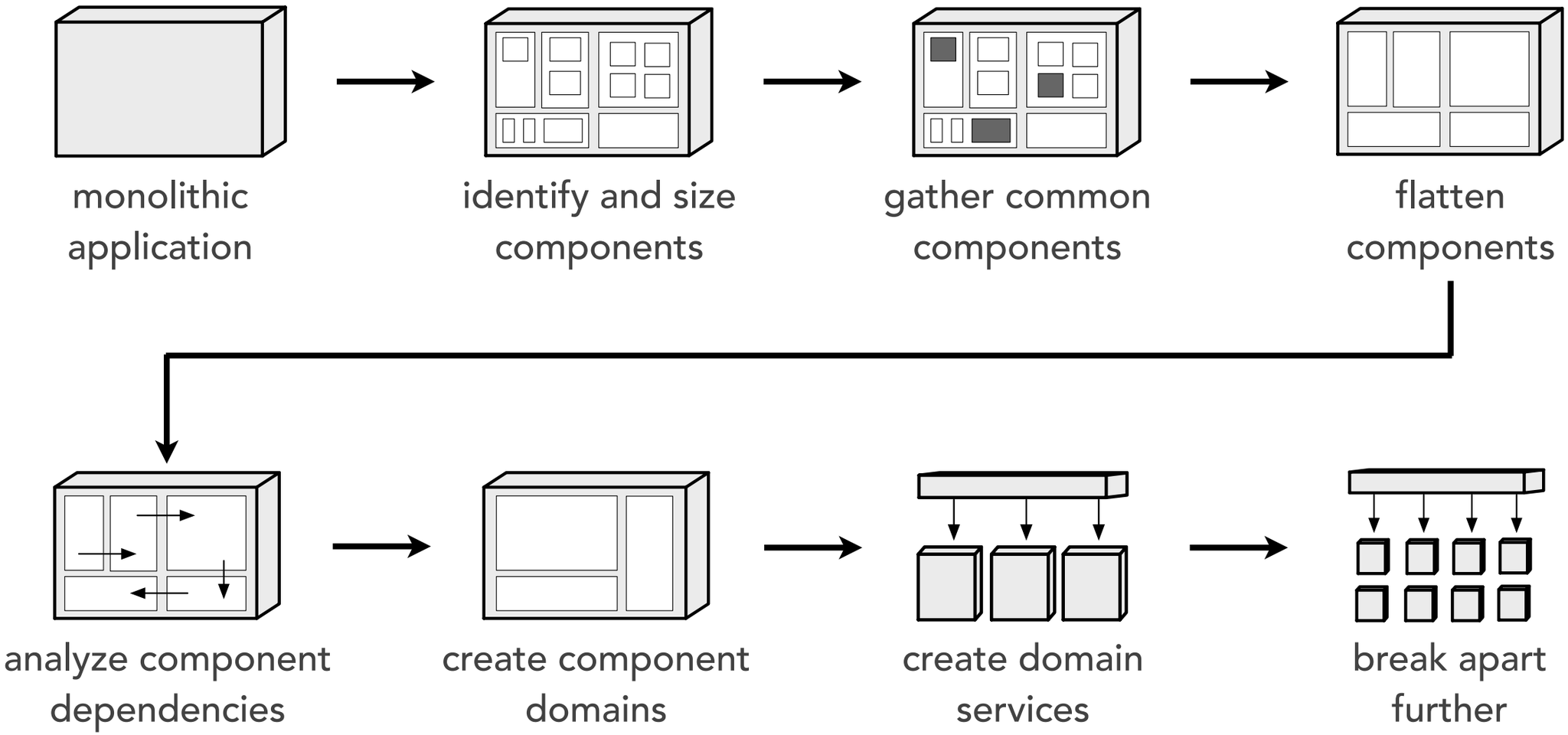


gather common components

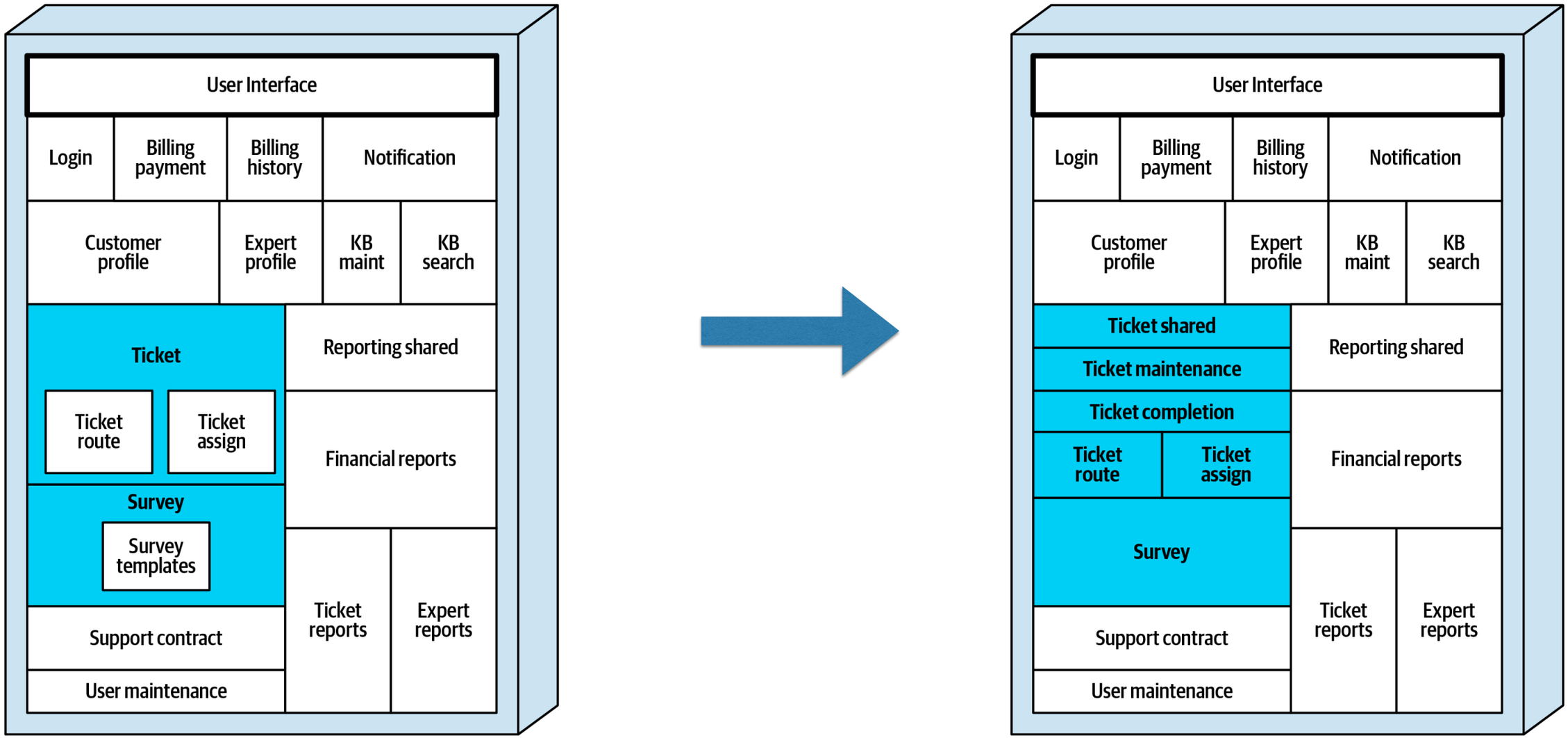


architectural modularity

component-based decomposition

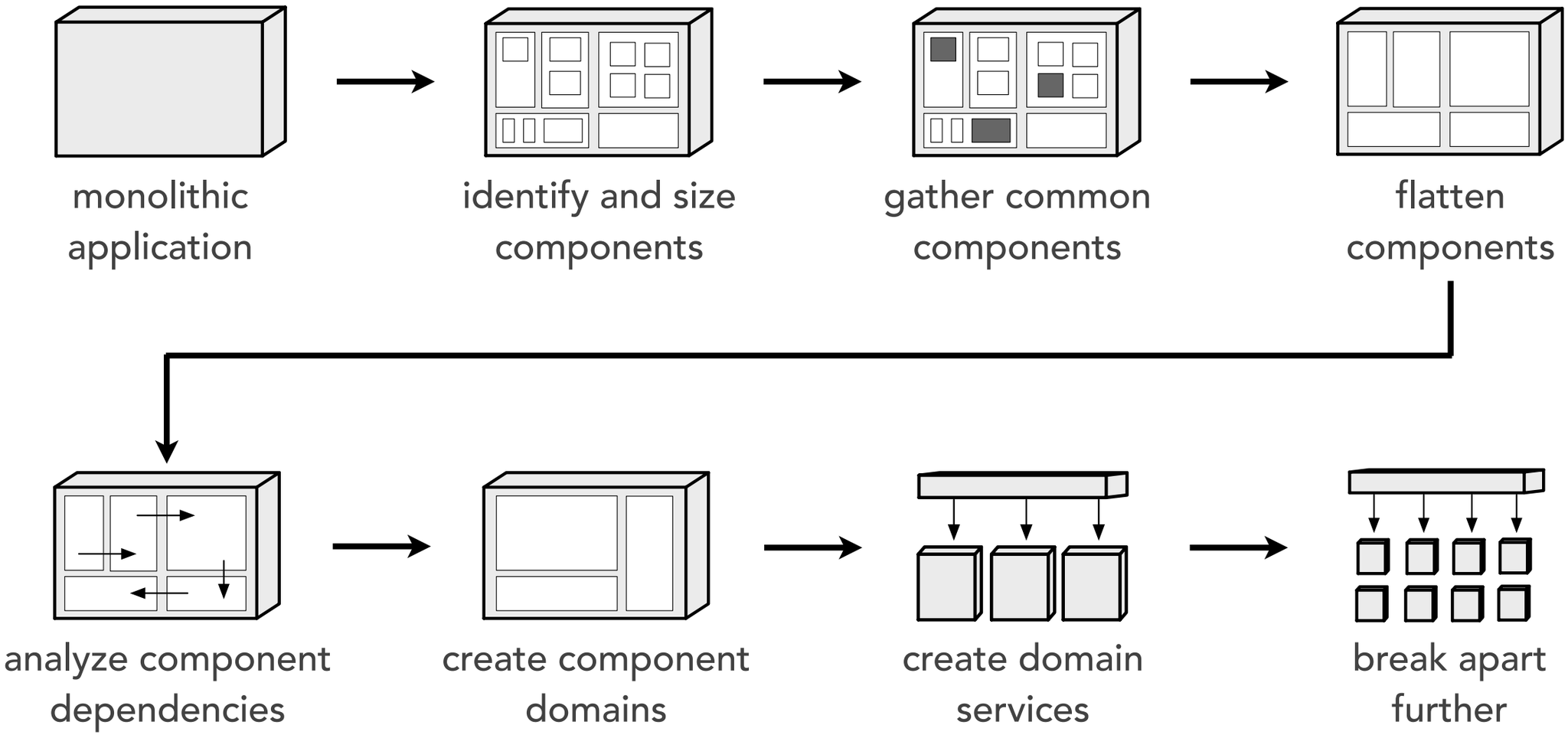


flatten components

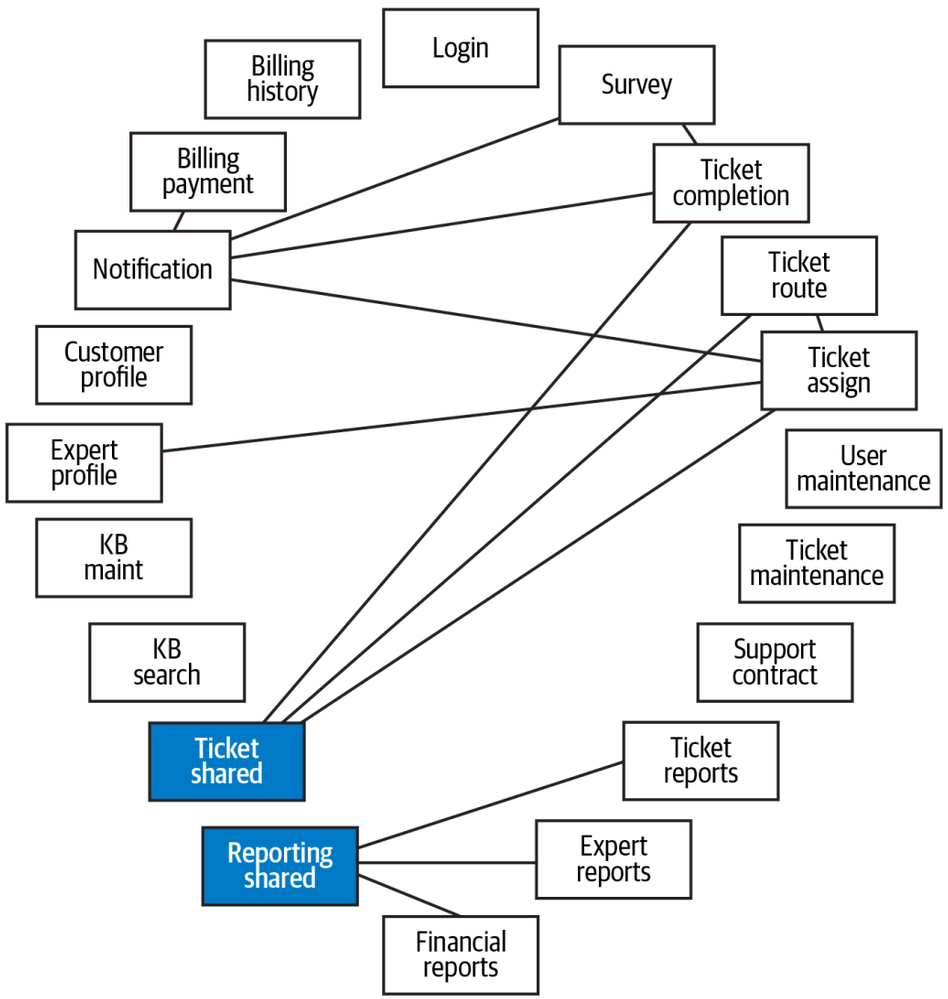


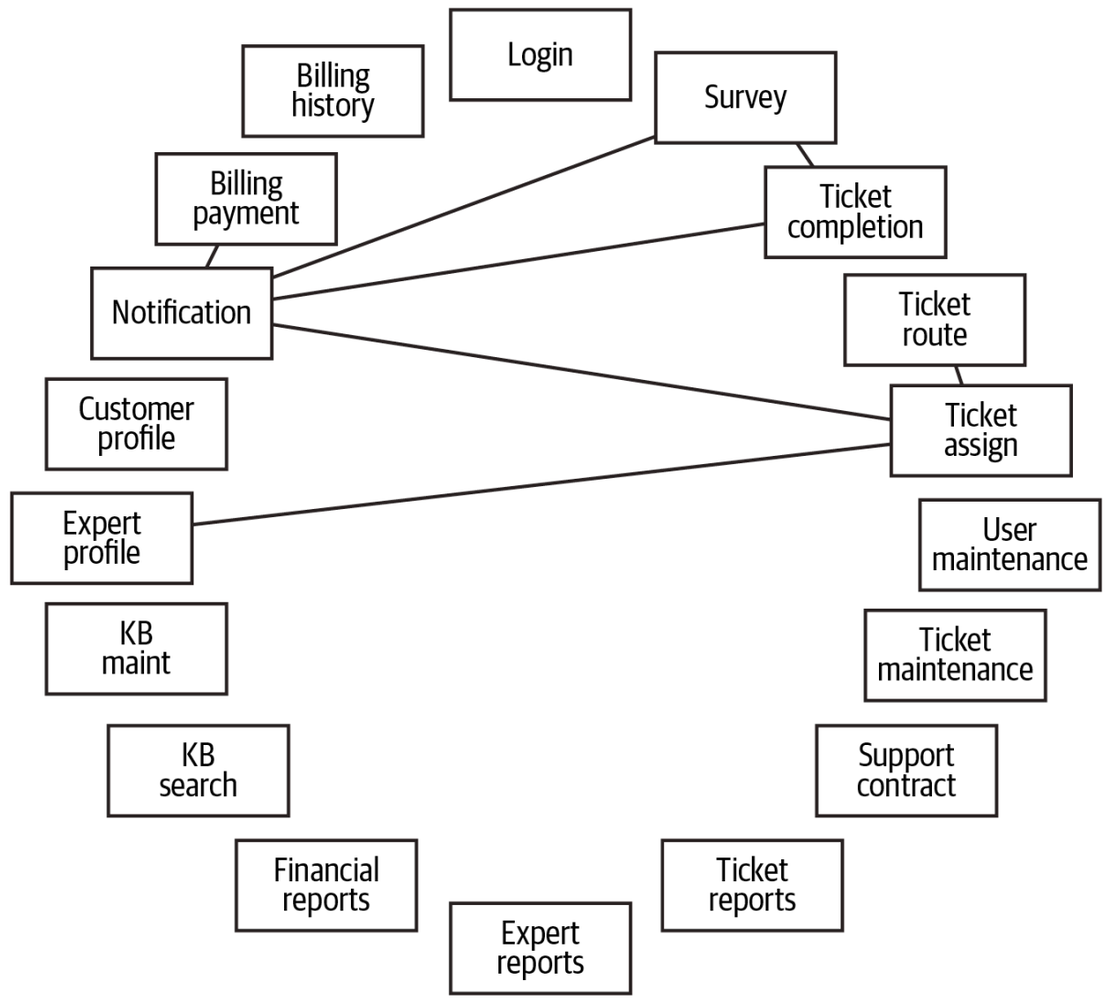
architectural modularity

component-based decomposition

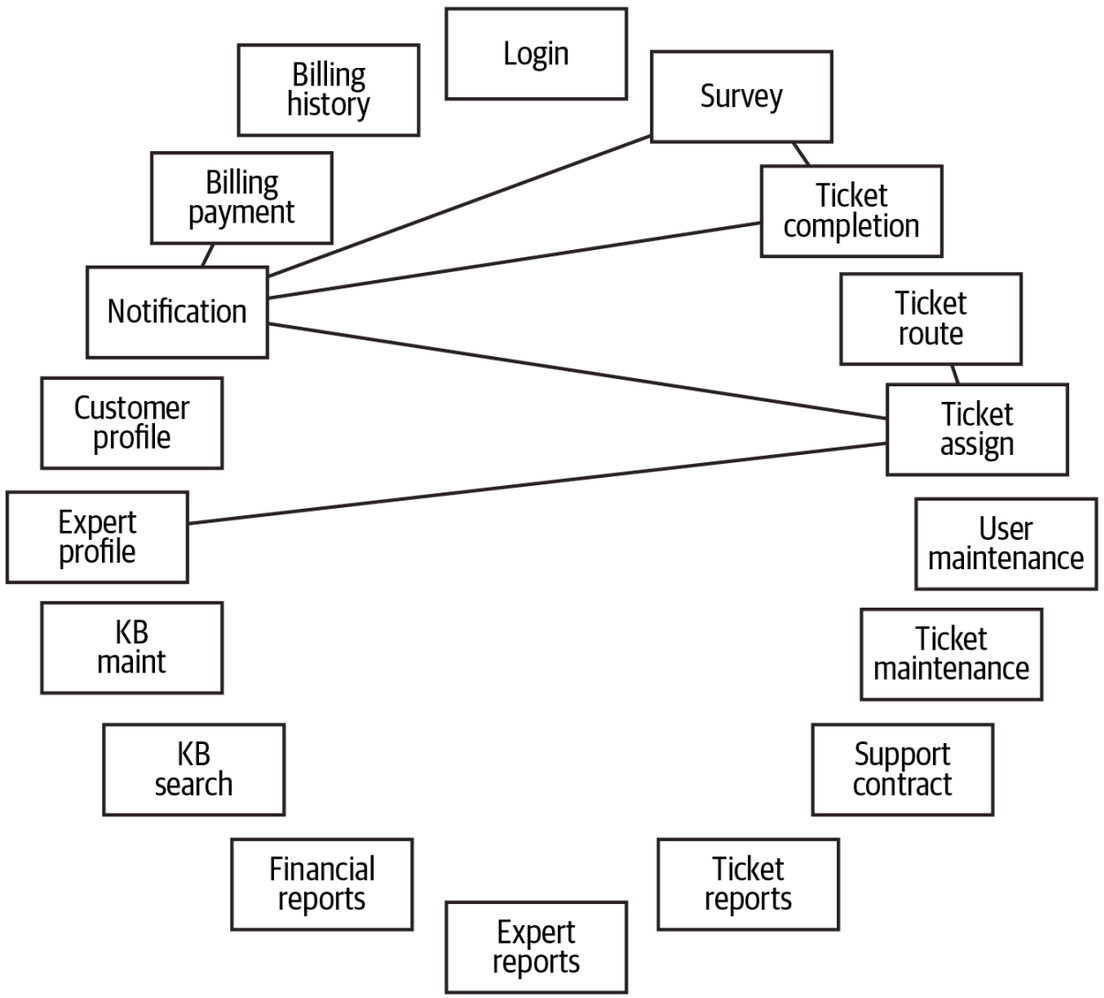


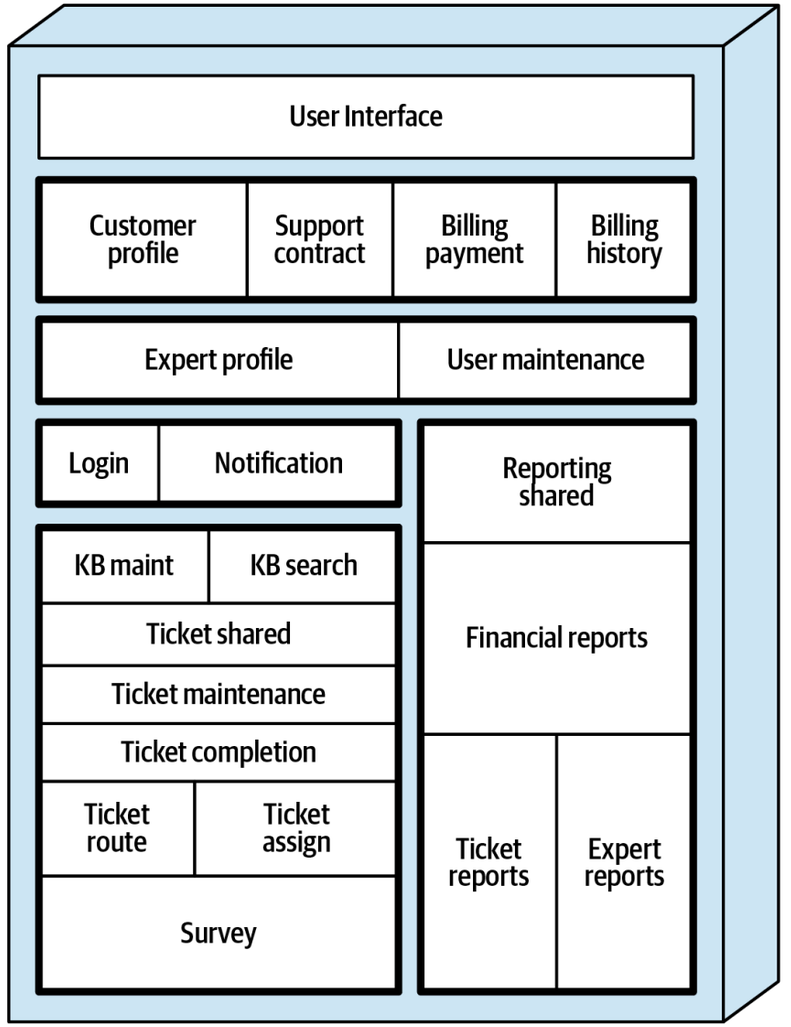
create components domains





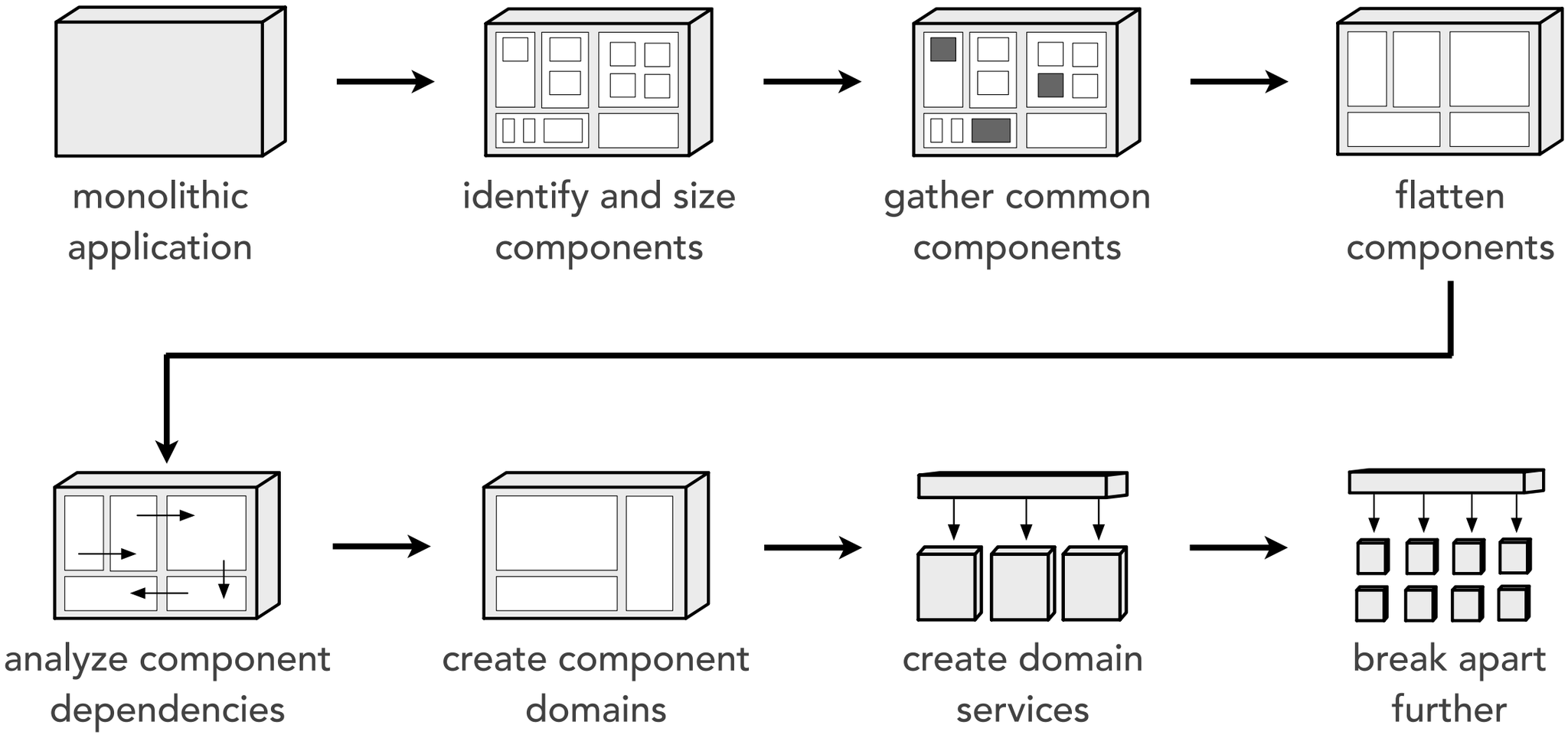
create components domains



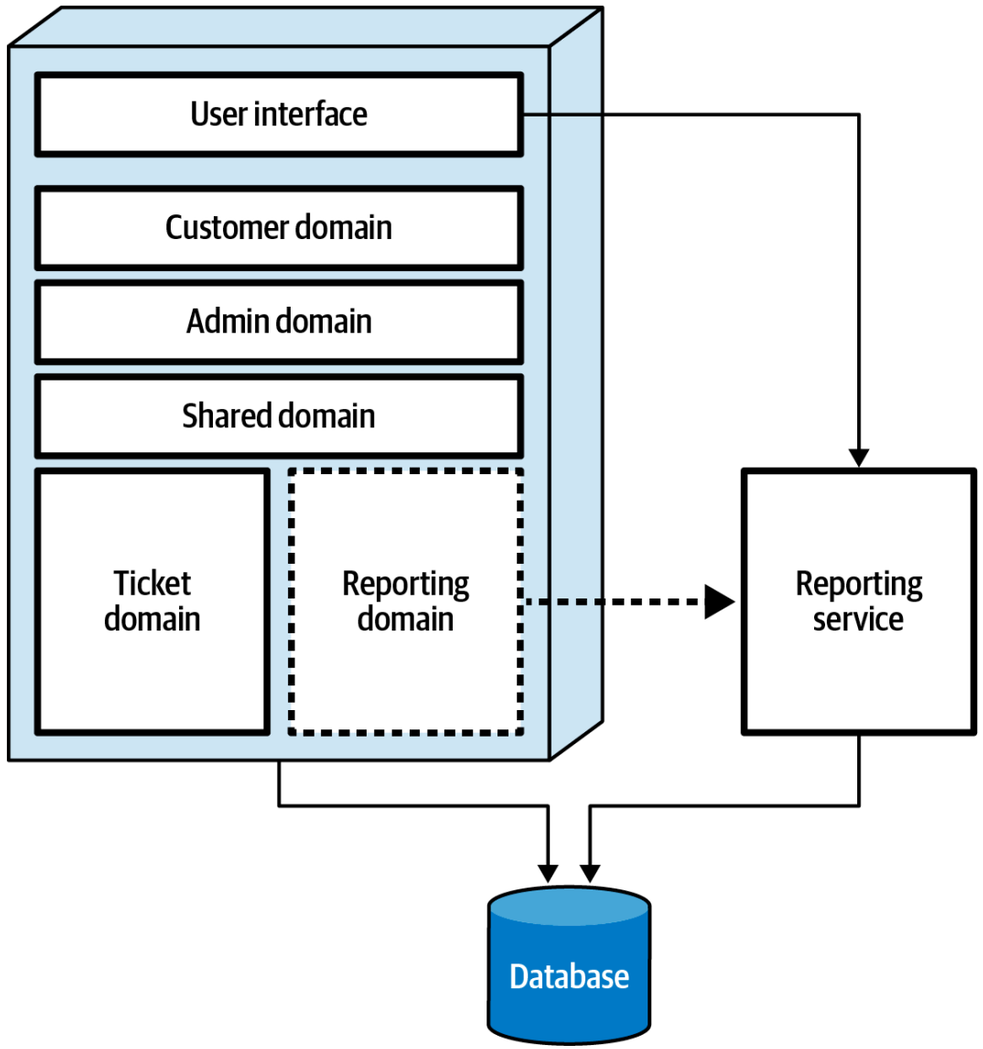


architectural modularity

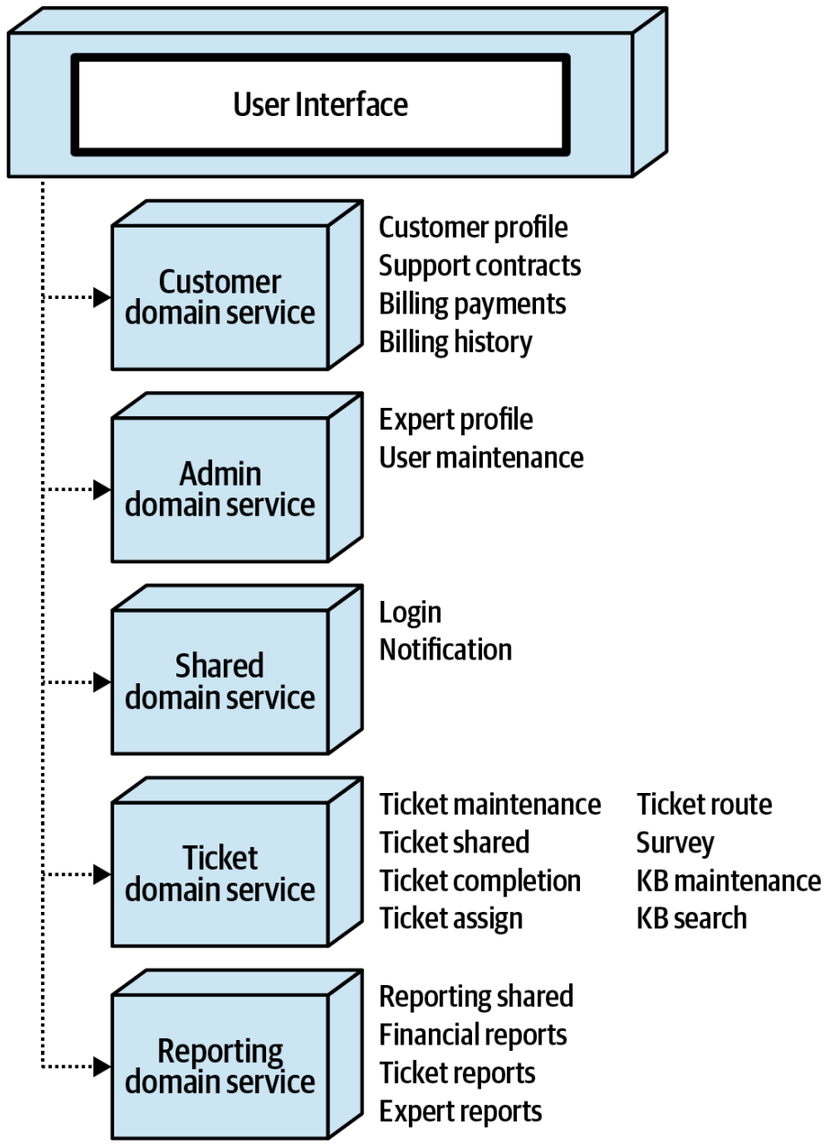
component-based decomposition

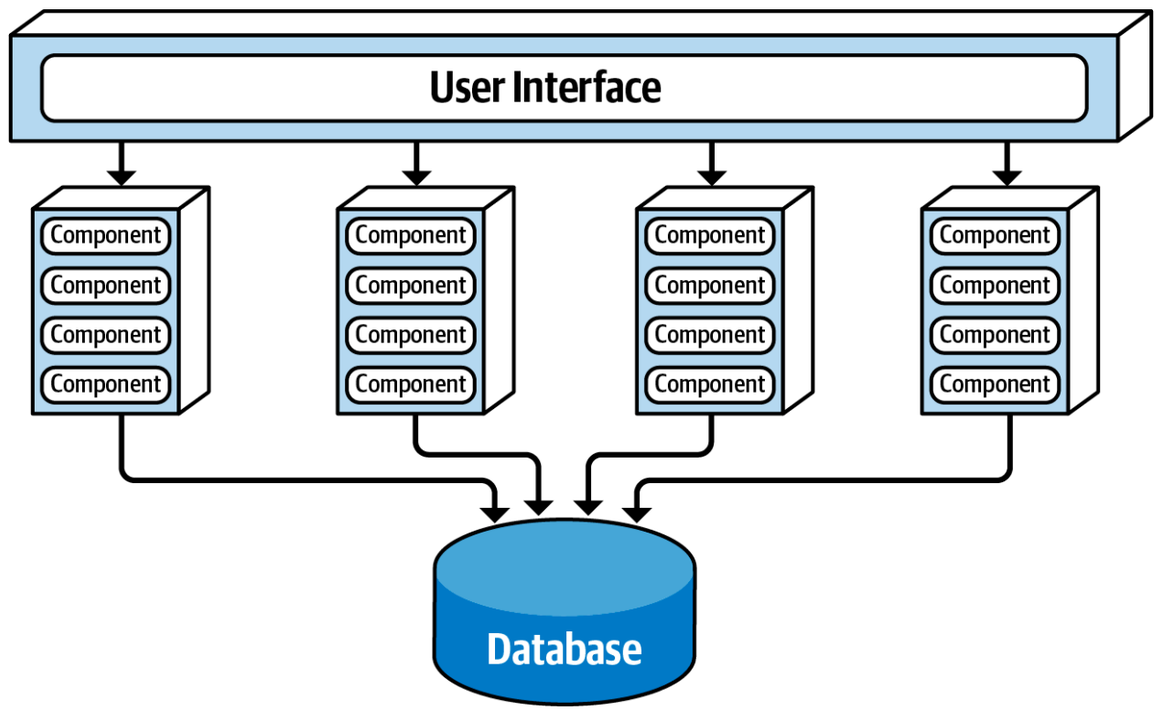


create domain services



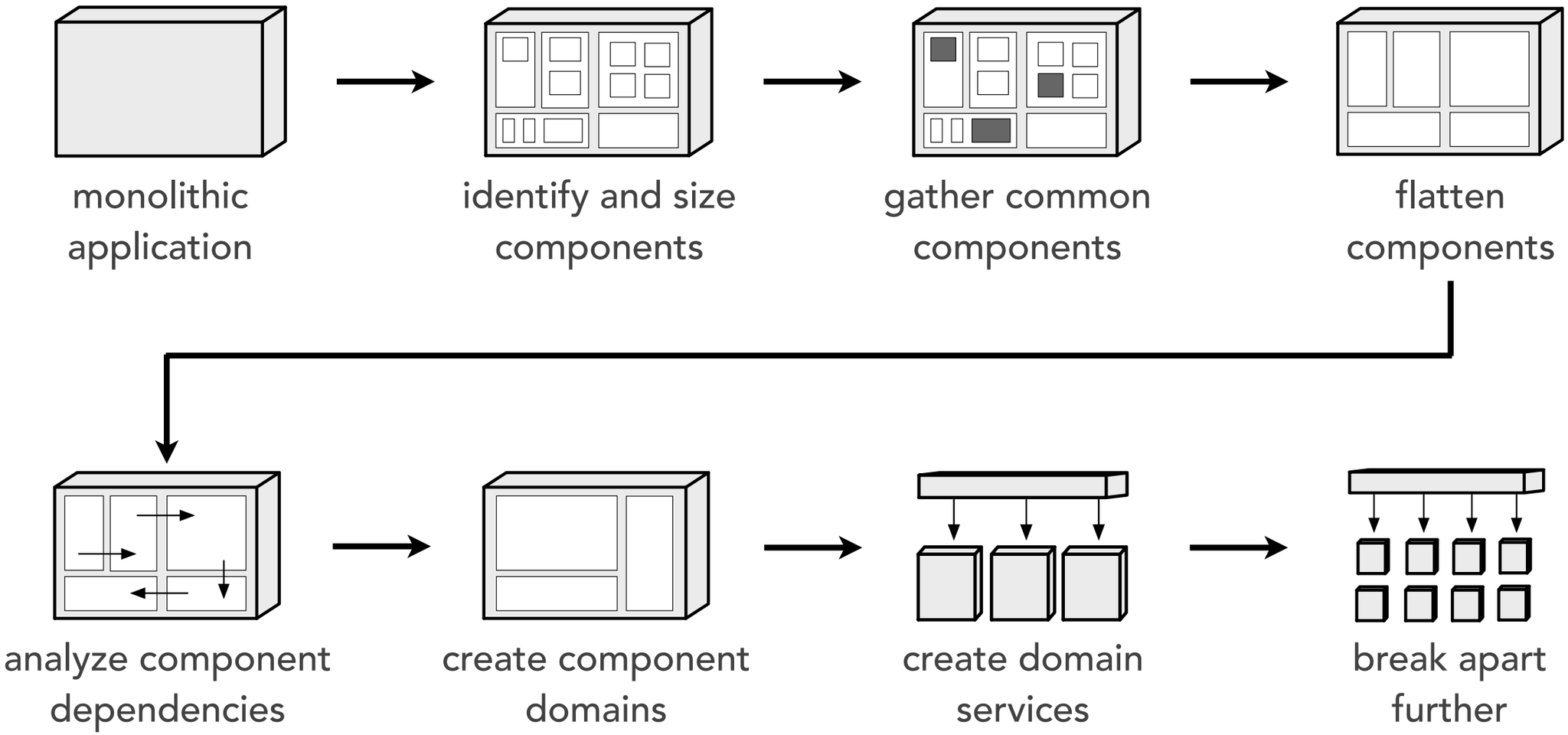
create domain services



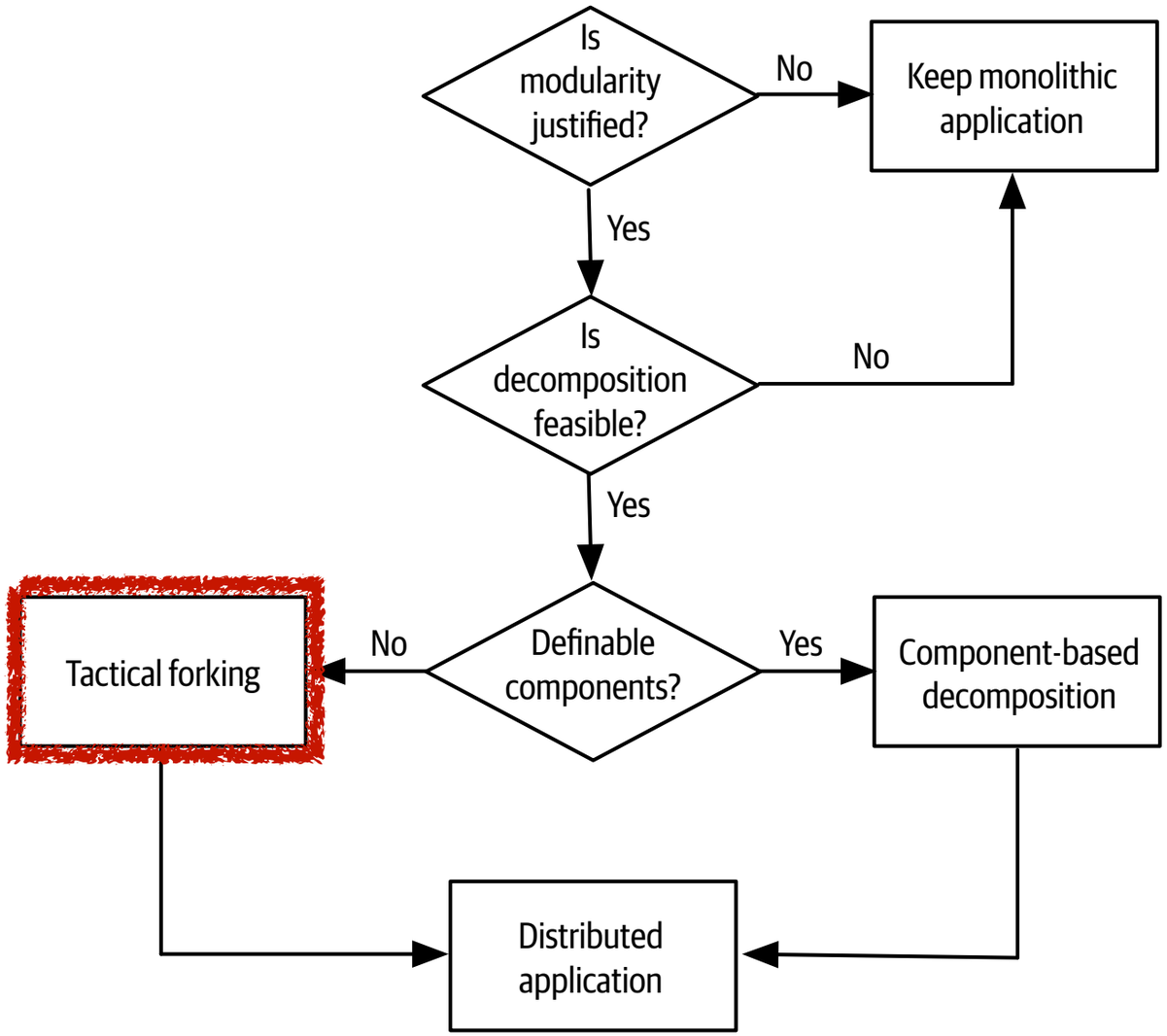


architectural modularity

component-based decomposition

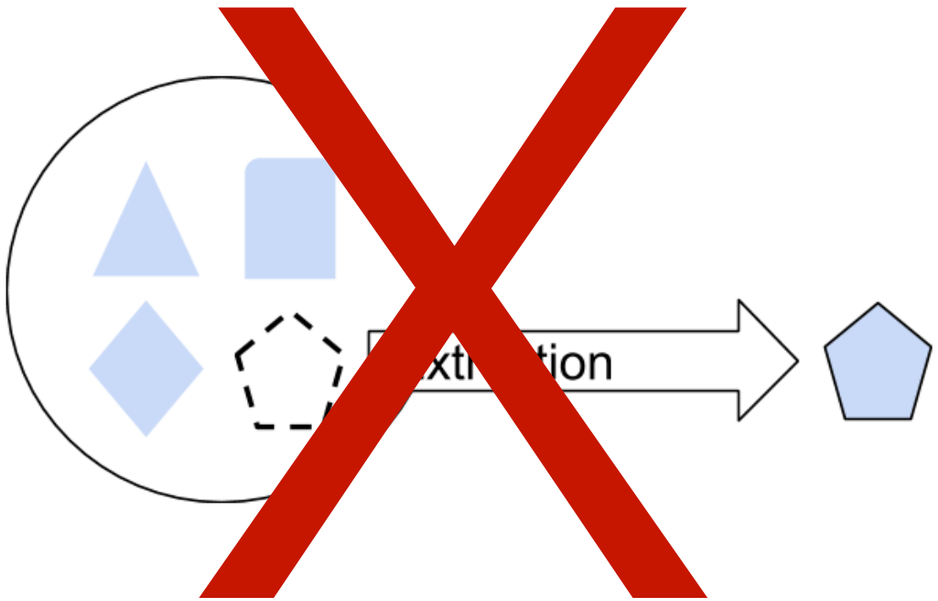


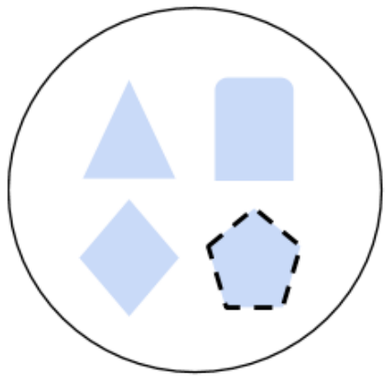
architectural modularity

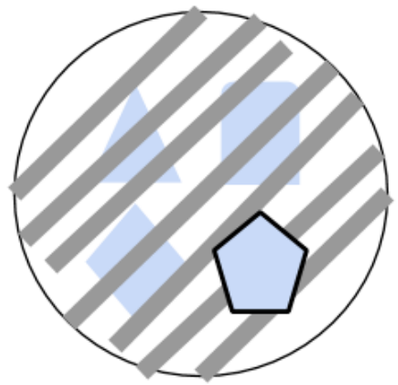


architectural modularity

tactical forking



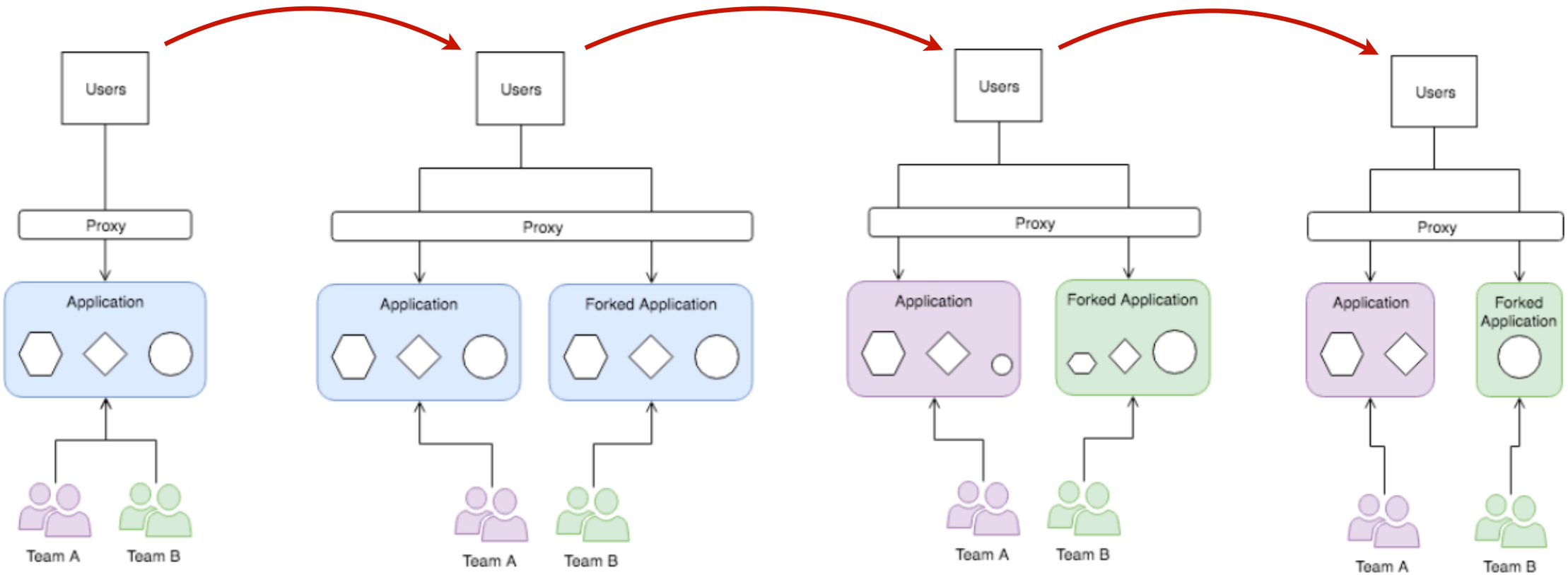




Deletion off
everything else

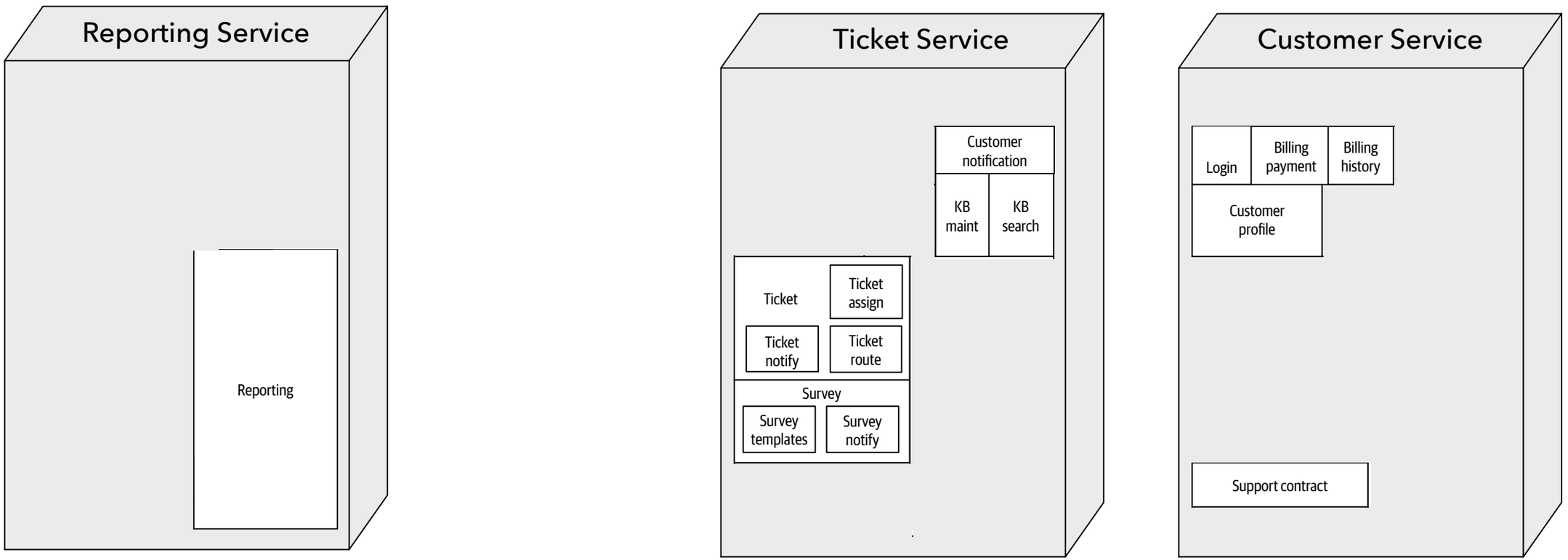
architectural modularity

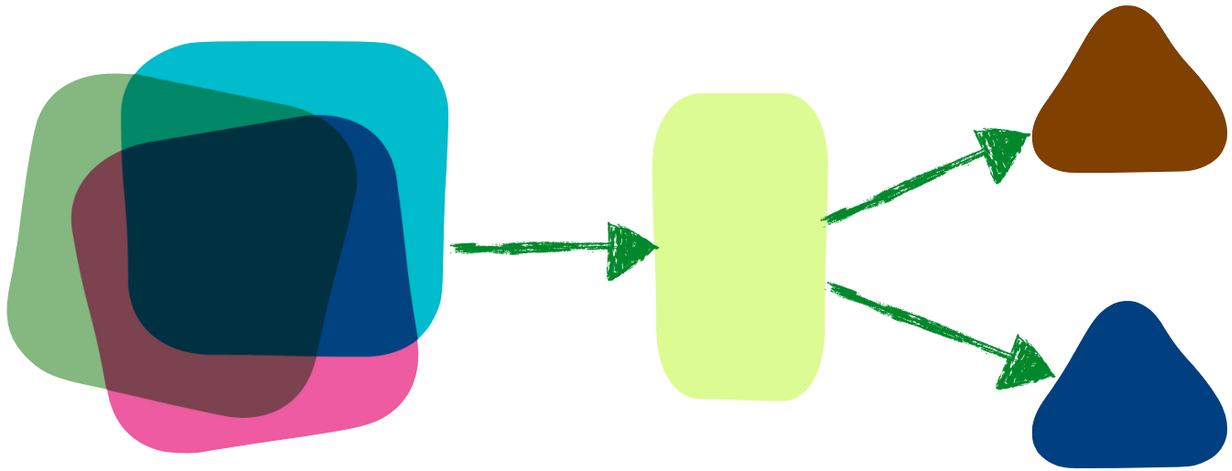
tactical forking



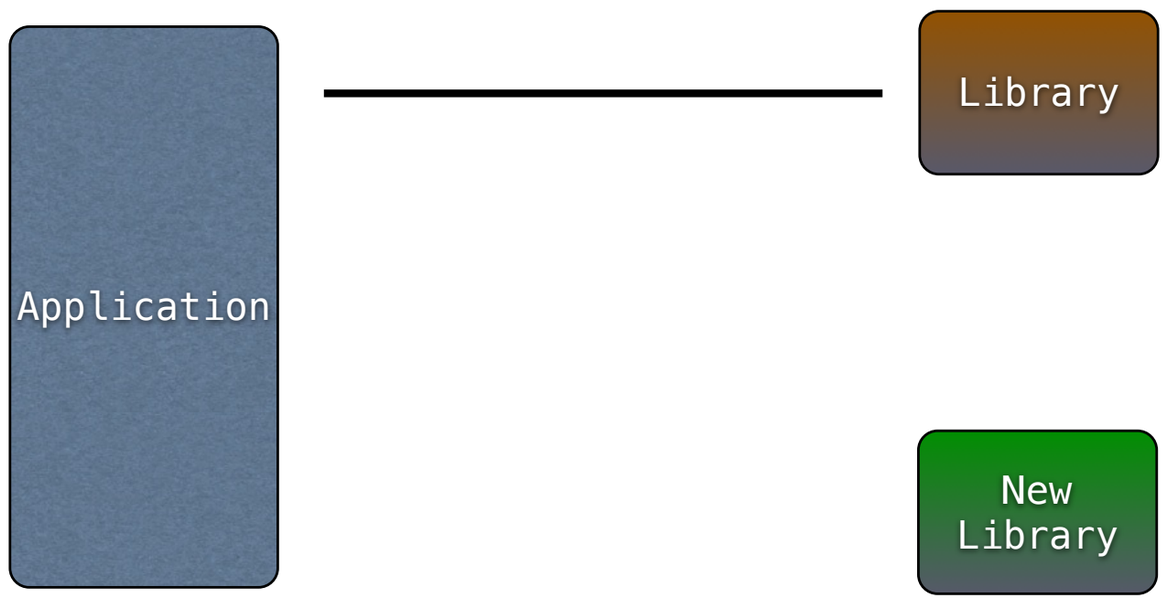
architectural modularity

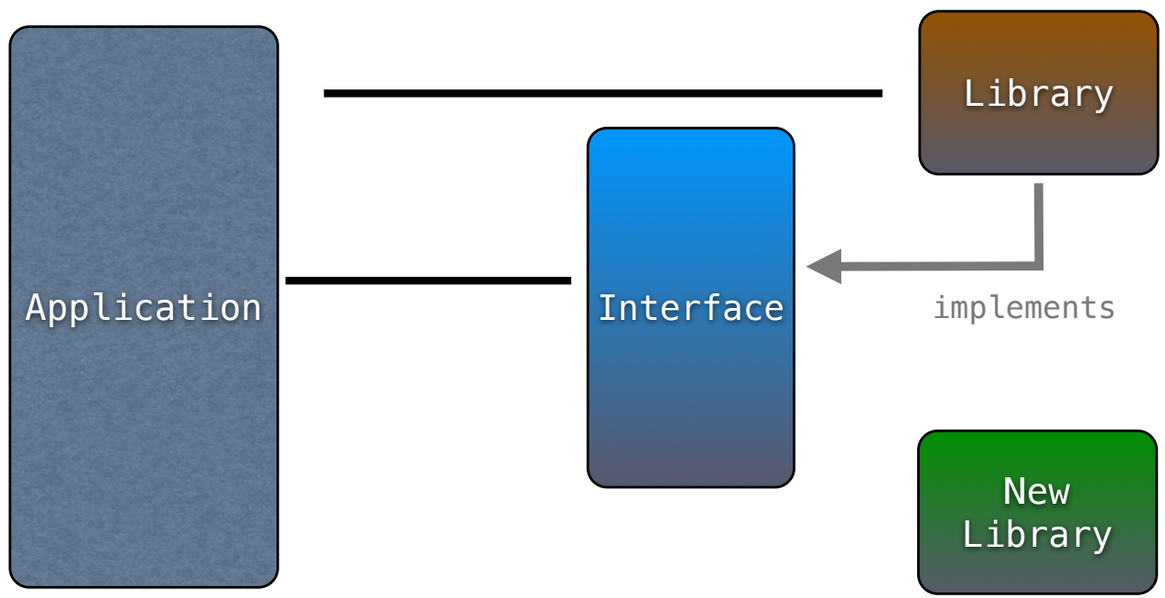
tactical forking

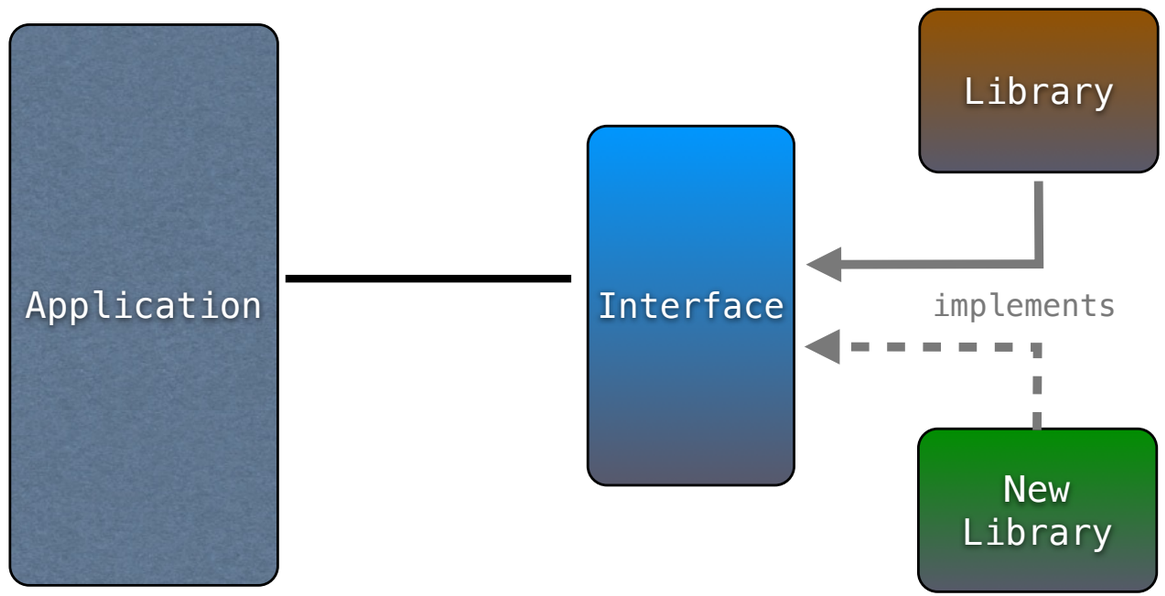


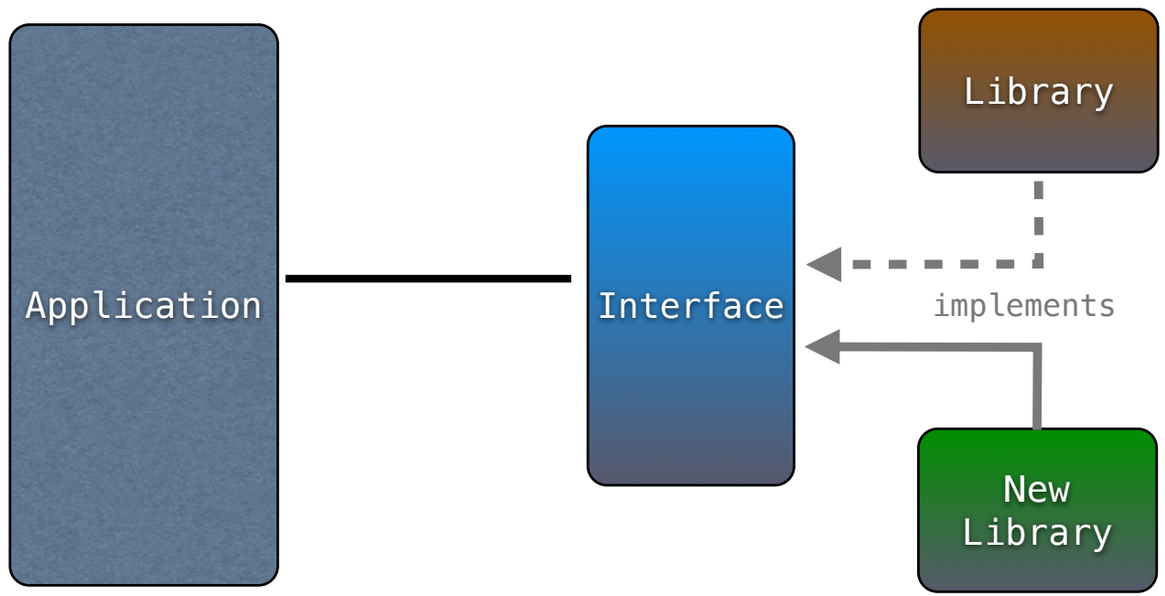


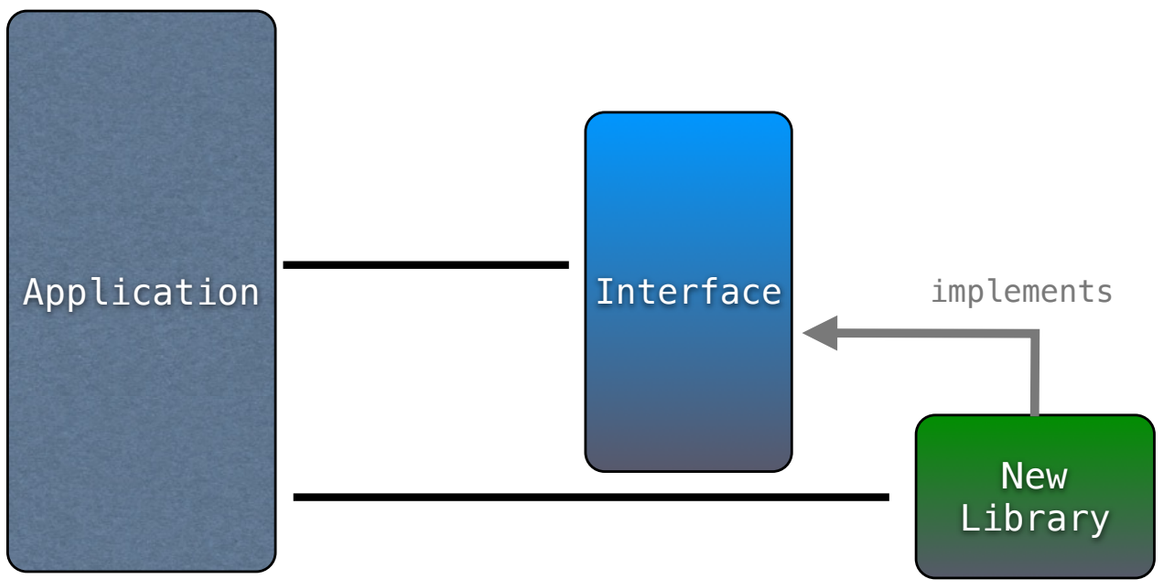
branch by abstraction


Application
Library
New
Library

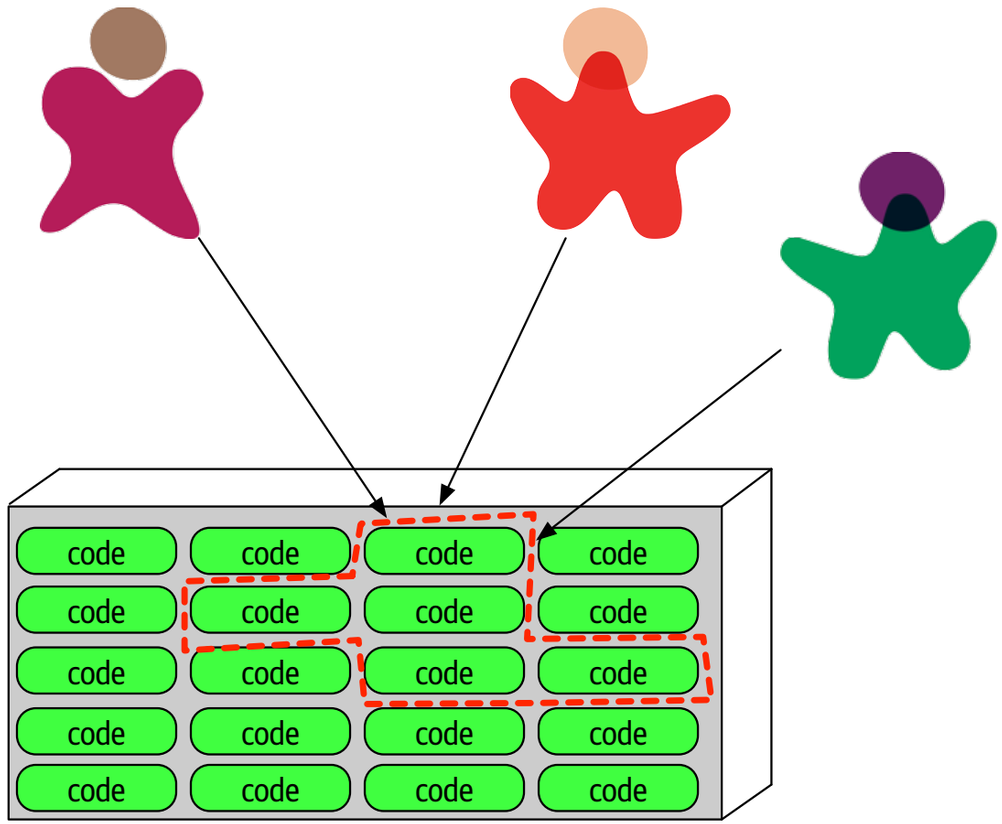

Application
Interface
Library
implements
New
Library

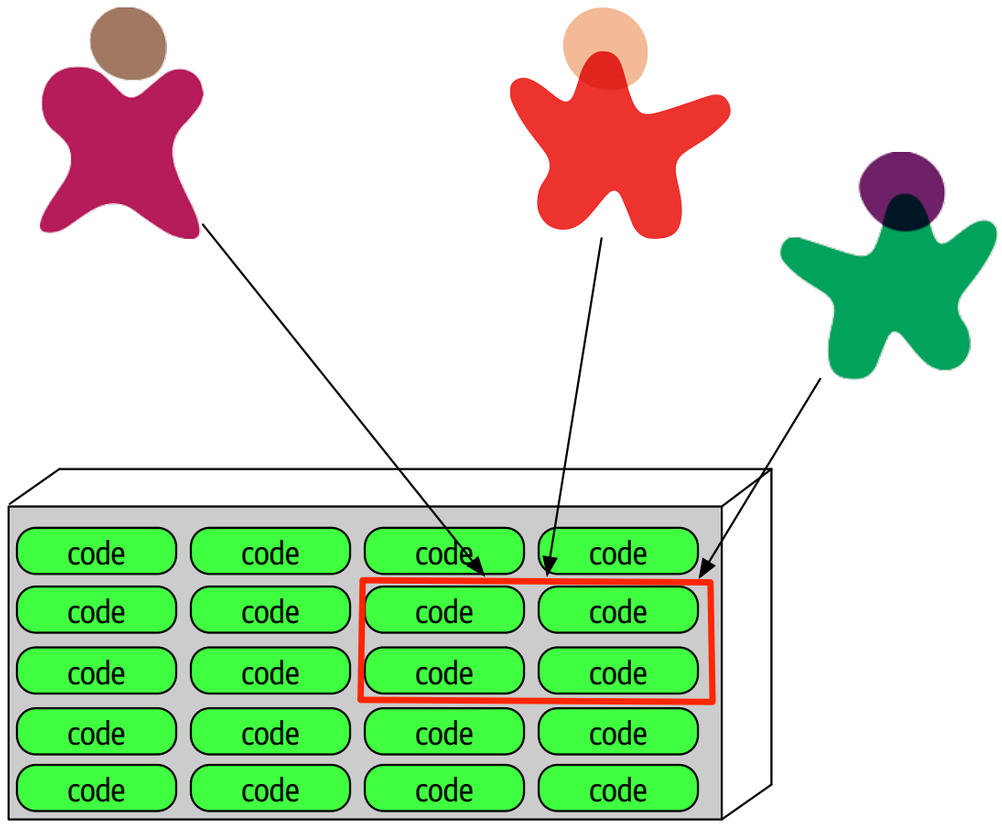

Application
Interface
Library
implements
New
Library

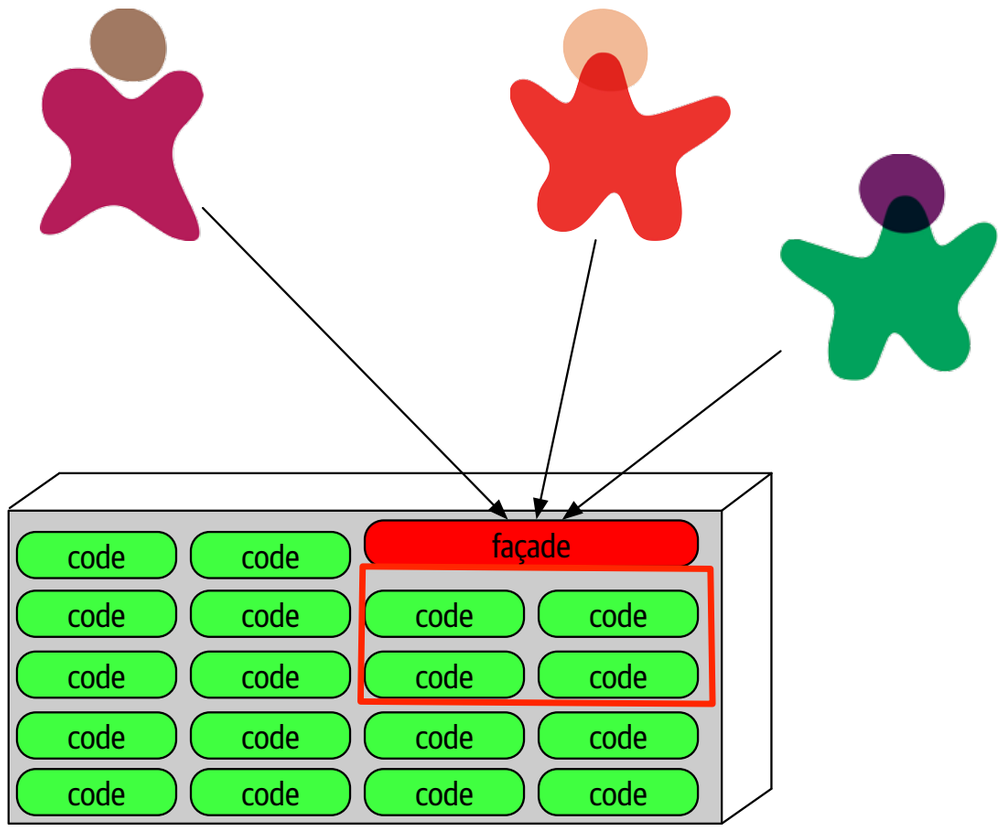

Application
Interface
Library
implements
New
Library

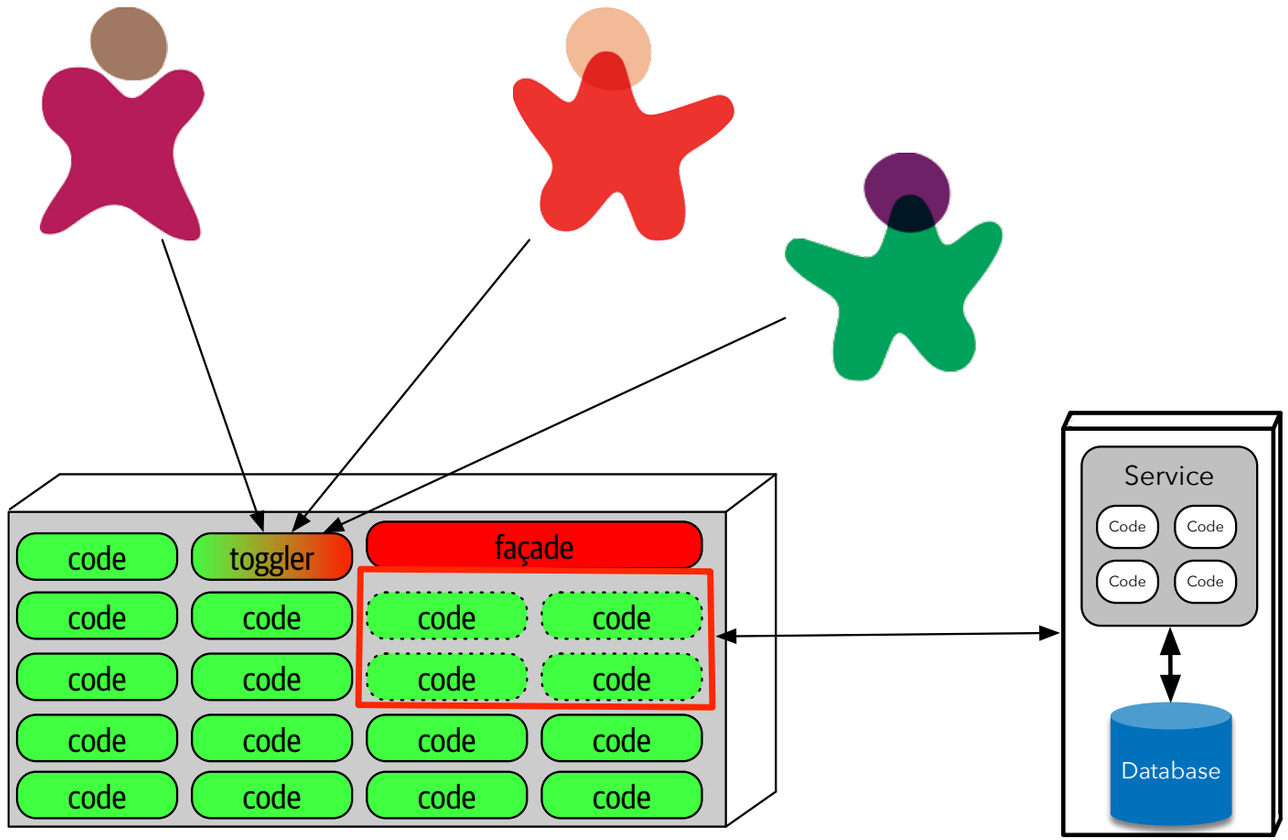

Application
Interface
implements
New
Library

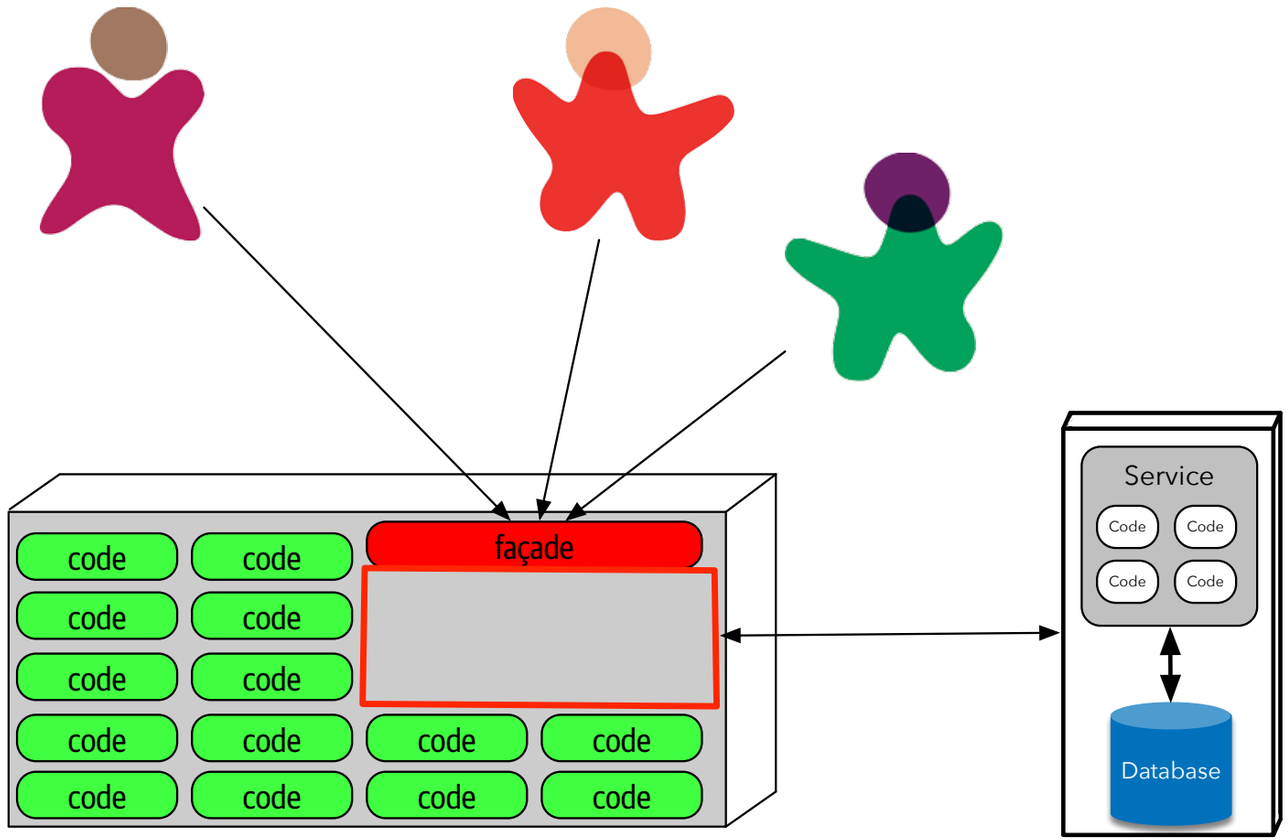
branch by abstraction
with a monolith

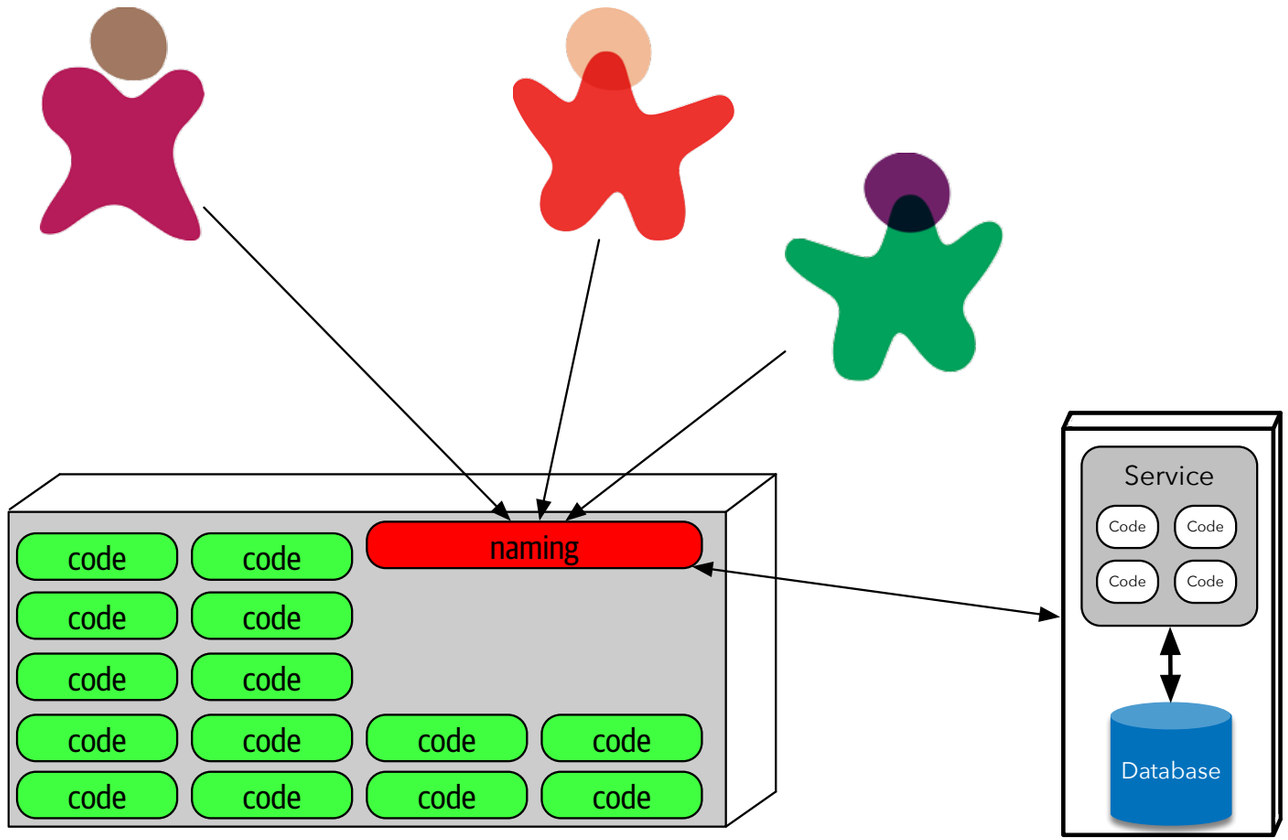

code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code


code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code

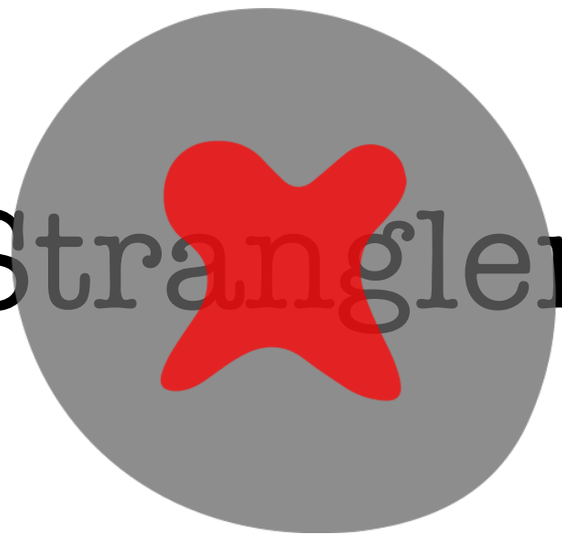

façade
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code


code
toggler
façade
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
Service
Code
Code
Code
Code
Database


code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
code
façade
Service
Code
Code
Code
Code
Database


code
code
naming
code
code
code
code
code
code
code
code
code
code
code
code
code
Service
Code
Code
Code
Code
Database

“Strangler” Pattern

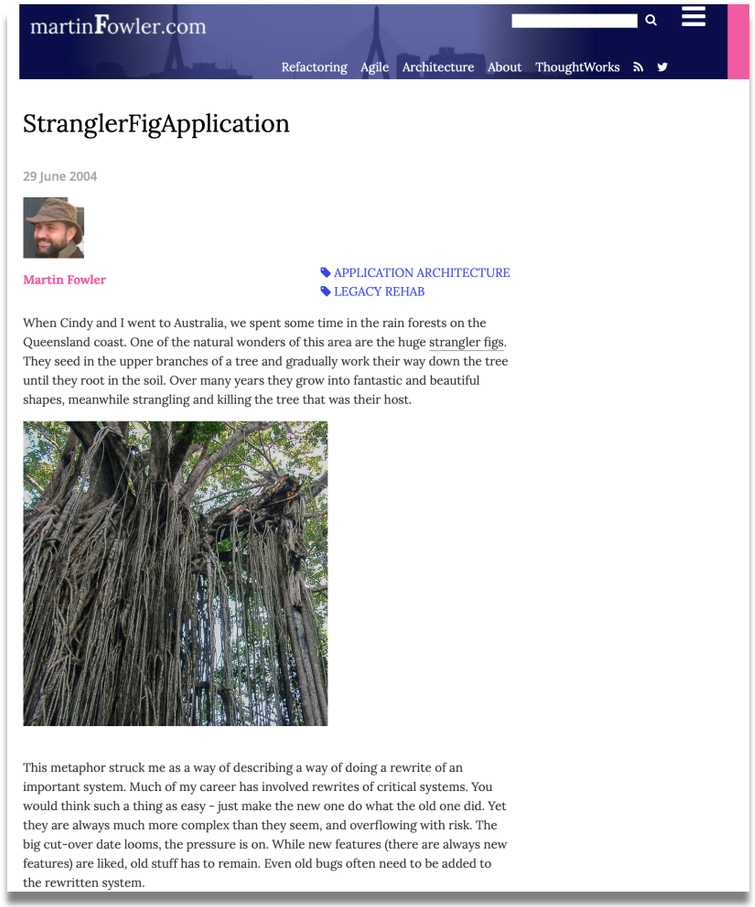


“Strangler” Pattern

StranglerFig Pattern

StranglerFig Pattern

StranglerFig Pattern resources



martinFowler.com

Refactoring Agile Architecture About ThoughtWorks

StranglerFigApplication

29 June 2004

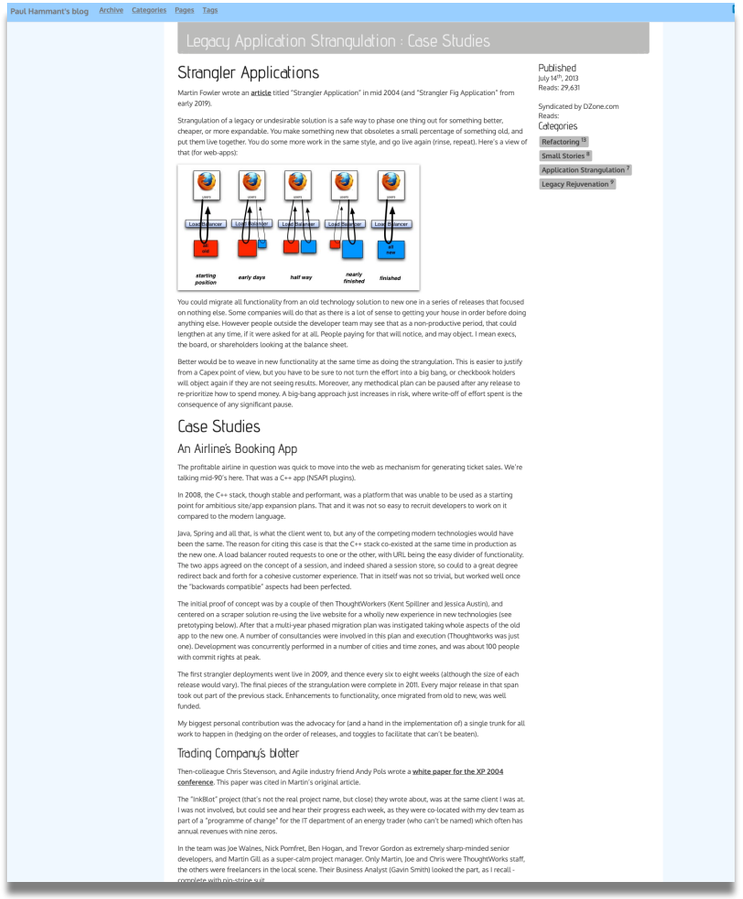
Martin Fowler

APPLICATION ARCHITECTURE
LEGACY REHAB

When Cindy and I went to Australia, we spent some time in the rain forests on the Queensland coast. One of the natural wonders of this area are the huge strangler figs. They seed in the upper branches of a tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host.

This metaphor struck me as a way of describing a way of doing a rewrite of an important system. Much of my career has involved rewrites of critical systems. You would think such a thing as easy - just make the new one do what the old one did. Yet they are always much more complex than they seem, and overflowing with risk. The big cut-over date looms, the pressure is on. While new features (there are always new features) are liked, old stuff has to remain. Even old bugs often need to be added to the rewritten system.

https://martinfowler.com/bliki/StranglerFigApplication.html



Paul Hammant's blog Archive Categories Pages Tags

Legacy Application Strangulation : Case Studies

Strangler Applications

Martin Fowler wrote an article titled "Strangler Application" in mid 2004 (and "Strangler Fig Application" from early 2019).

Strangulation of a legacy or undesirable solution is a safe way to phase one thing out for something better, cheaper, or more expandable. You make something new that obsoletes a small percentage of something old, and put them live together. You do some more work in the same style, and go live again (rinse, repeat). Here's a view of that (for web-apps):

Published
July 14th, 2013
Reads: 29,631

Syndicated by DZone.com
Reads:

Categories
Refactoring 18
Small Stories 8
Application Strangulation 7
Legacy Rejuvenation 6

You could migrate all functionality from an old technology solution to new one in a series of releases that focused on nothing else. Some companies will do that as there is a lot of sense to getting your house in order before doing anything else. However people outside the developer team may see that as a non-productive period, that could lengthen at any time, if it were asked for at all. People paying for that will notice, and may object. I mean execs, the board, or shareholders looking at the balance sheet.

Better would be to weave in new functionality at the same time as doing the strangulation. This is easier to justify from a Capex point of view, but you have to be sure to not turn the effort into a big bang, or checkbook holders will object again if they are not seeing results. Moreover, any methodical plan can be paused after any release to re-prioritize how to spend money. A big-bang approach just increases in risk, where write-off of effort spent is the consequence of any significant pause.

Case Studies

An Airline's Booking App

The profitable airline in question was quick to move into the web as mechanism for generating ticket sales. We're talking mid-90's here. That was a C++ app (NSAPI plugins).

In 2008, the C++ stack, though stable and performant, was a platform that was unable to be used as a starting point for ambitious site/app expansion plans. That and it was not so easy to recruit developers to work on it compared to the modern language.

Java, Spring and all that, is what the client went to, but any of the competing modern technologies would have been the same. The reason for citing this case is that the C++ stack co-existed at the same time in production as the new one. A load balancer routed requests to one or the other, with URL being the easy divider of functionality. The two apps agreed on the concept of a session, and indeed shared a session store, so could to a great degree redirect back and forth for a cohesive customer experience. That in itself was not so trivial, but worked well once the "backwards compatible" aspects had been perfected.

The initial proof of concept was by a couple of then ThoughtWorkers (Kent Spillner and Jessica Austin), and centered on a scraper solution re-using the live website for a wholly new experience in new technologies (see pretotyping below). After that a multi-year phased migration plan was instigated taking whole aspects of the old app to the new one. A number of consultancies were involved in this plan and execution (Thoughtworks was just one). Development was concurrently performed in a number of cities and time zones, and was about 100 people with commit rights at peak.

The first strangler deployments went live in 2009, and thence every six to eight weeks (although the size of each release would vary). The final pieces of the strangulation were complete in 2011. Every major release in that span took out part of the previous stack. Enhancements to functionality, once migrated from old to new, was well funded.

My biggest personal contribution was the advocacy for (and a hand in the implementation of) a single trunk for all work to happen in (hedging on the order of releases, and toggles to facilitate that can't be beaten).

Trading Company's blotter

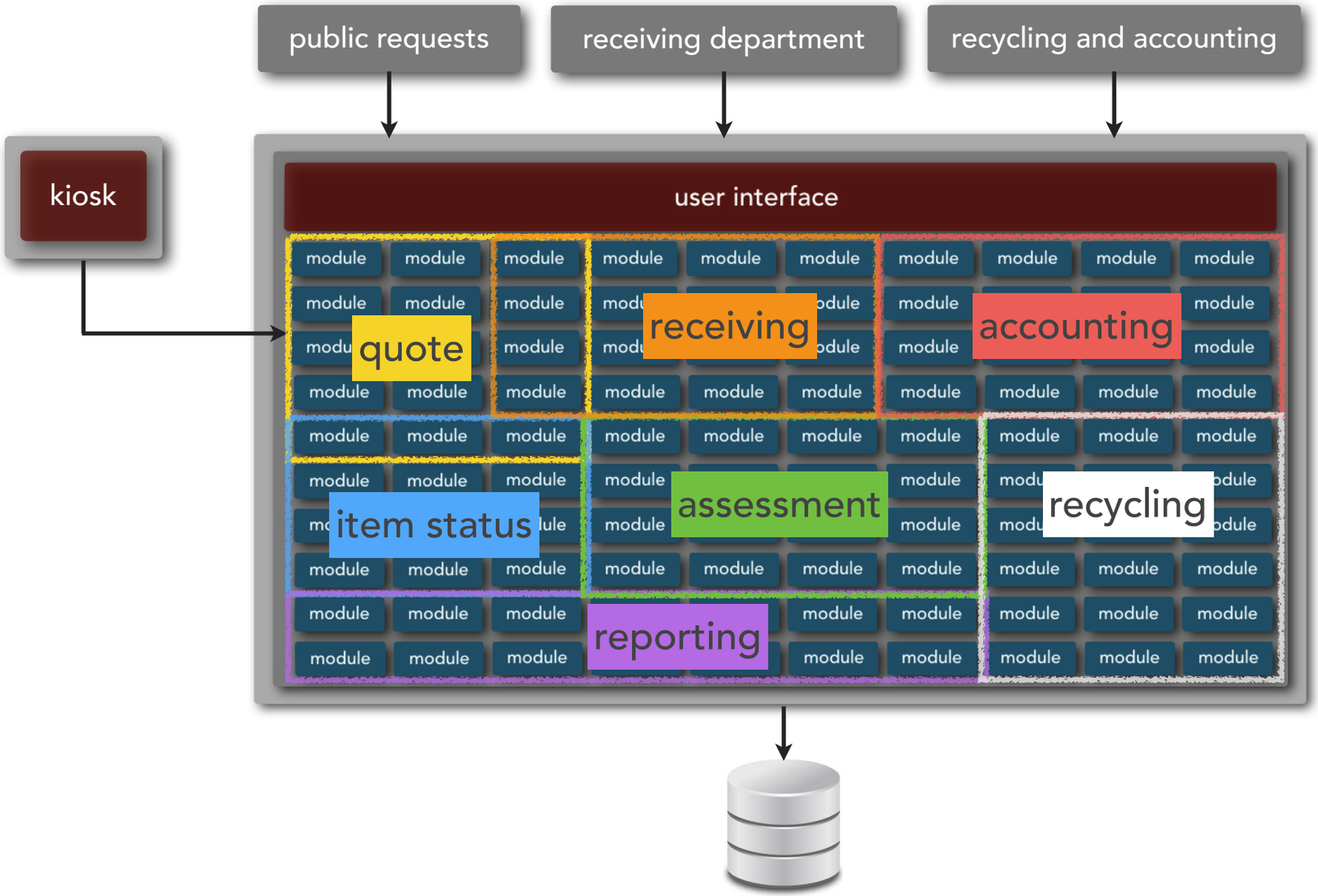
Then-colleague Chris Stevenson, and Agile industry friend Andy Pols wrote a white paper for the XP 2004 conference. This paper was cited in Martin's original article.

The "InkBlot" project (that's not the real project name, but close) they wrote about, was at the same client I was at. I was not involved, but could see and hear their progress each week, as they were co-located with my dev team as part of a "programme of change" for the IT department of an energy trader (who can't be named) which often has annual revenues with nine zeros.

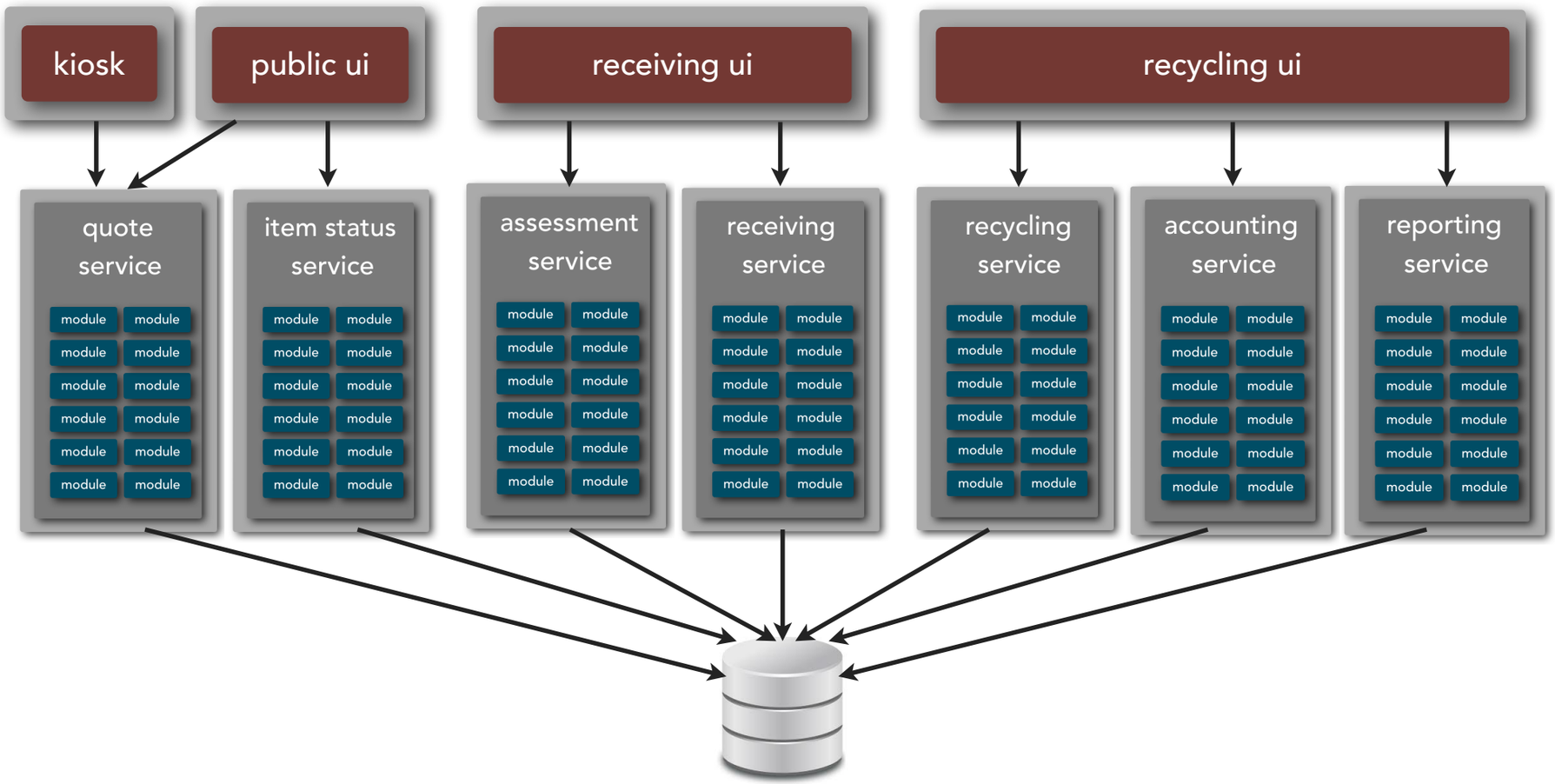
In the team was Joe Walnes, Nick Pomfret, Ben Hogan, and Trevor Gordon as extremely sharp-minded senior developers, and Martin Gill as a super-calm project manager. Only Martin, Joe and Chris were ThoughtWorks staff, the others were freelancers in the local scene. Their Business Analyst (Gavin Smith) looked the part, as I recall - complete with pin-stripe suit

https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/

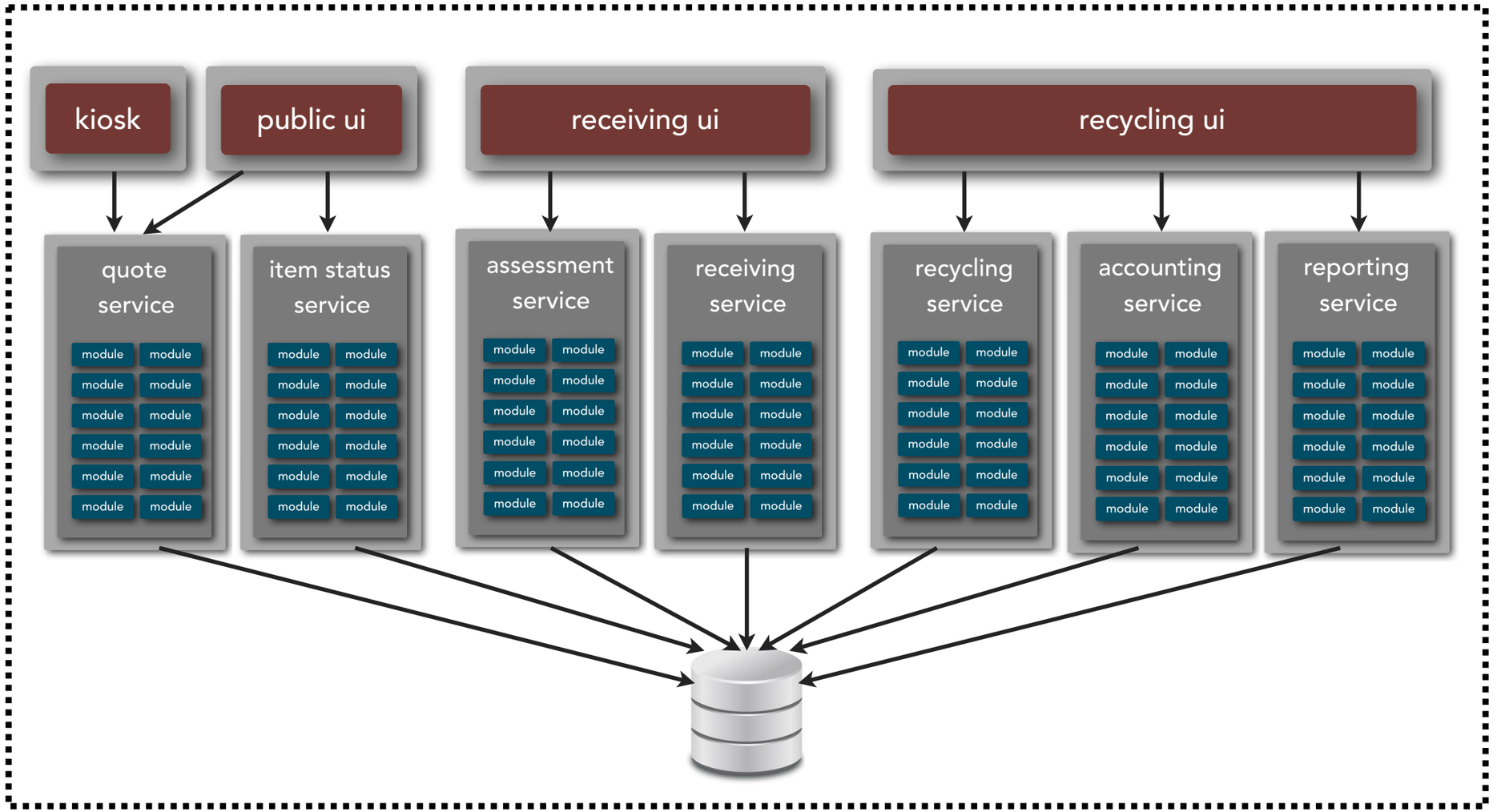
Electronics Recycling Application



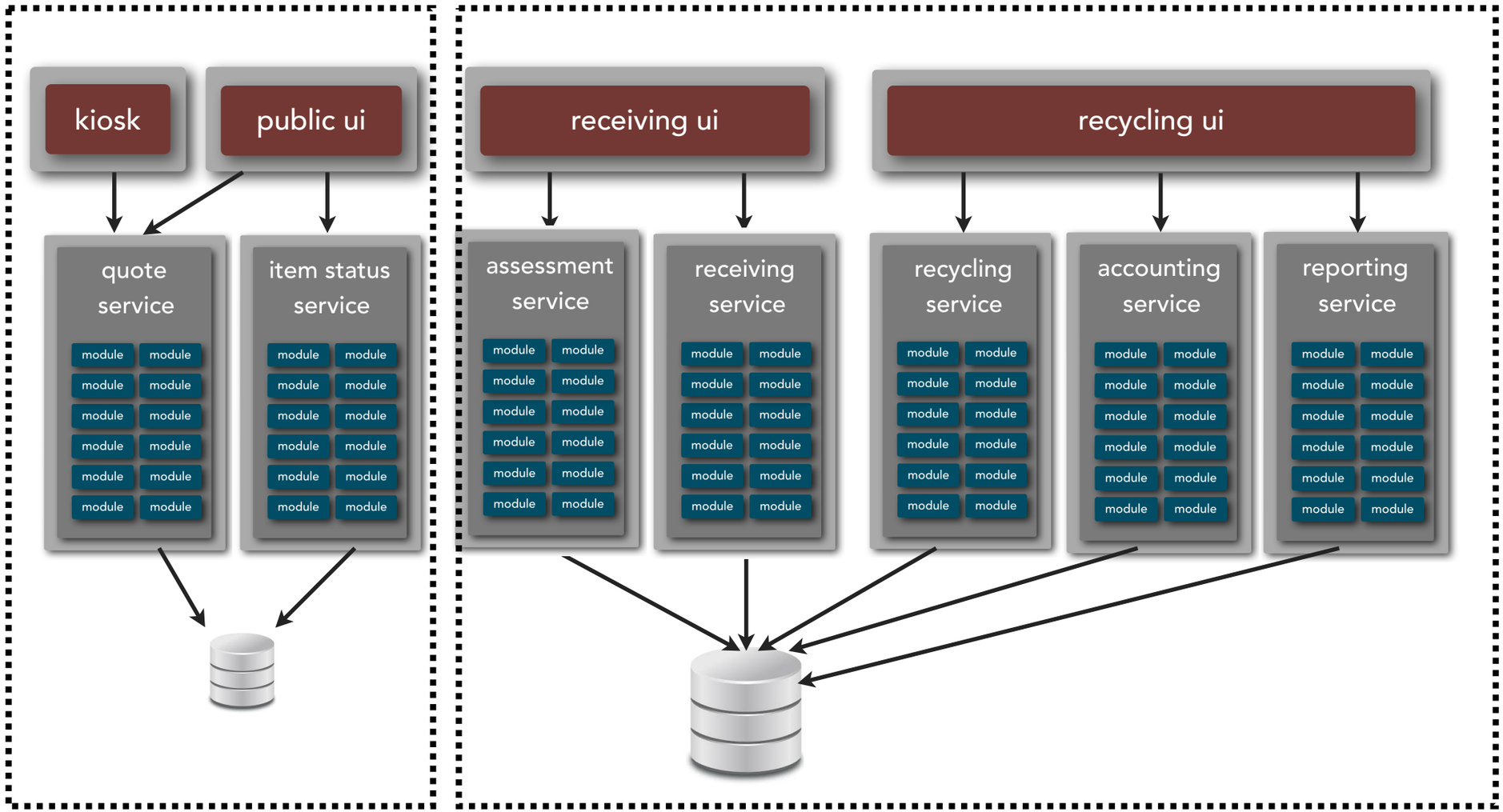
architectural quantum ?



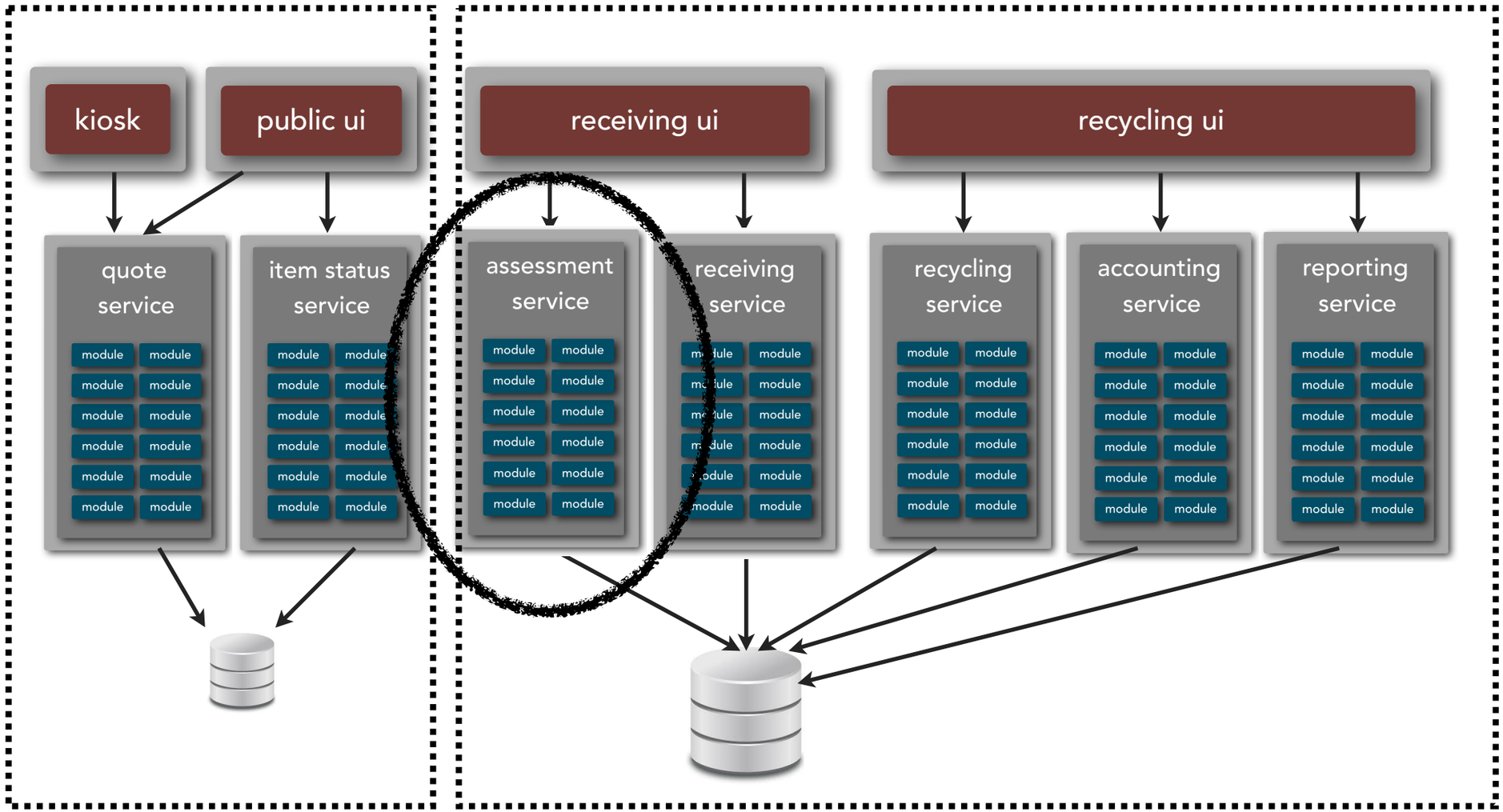
architectural quantum



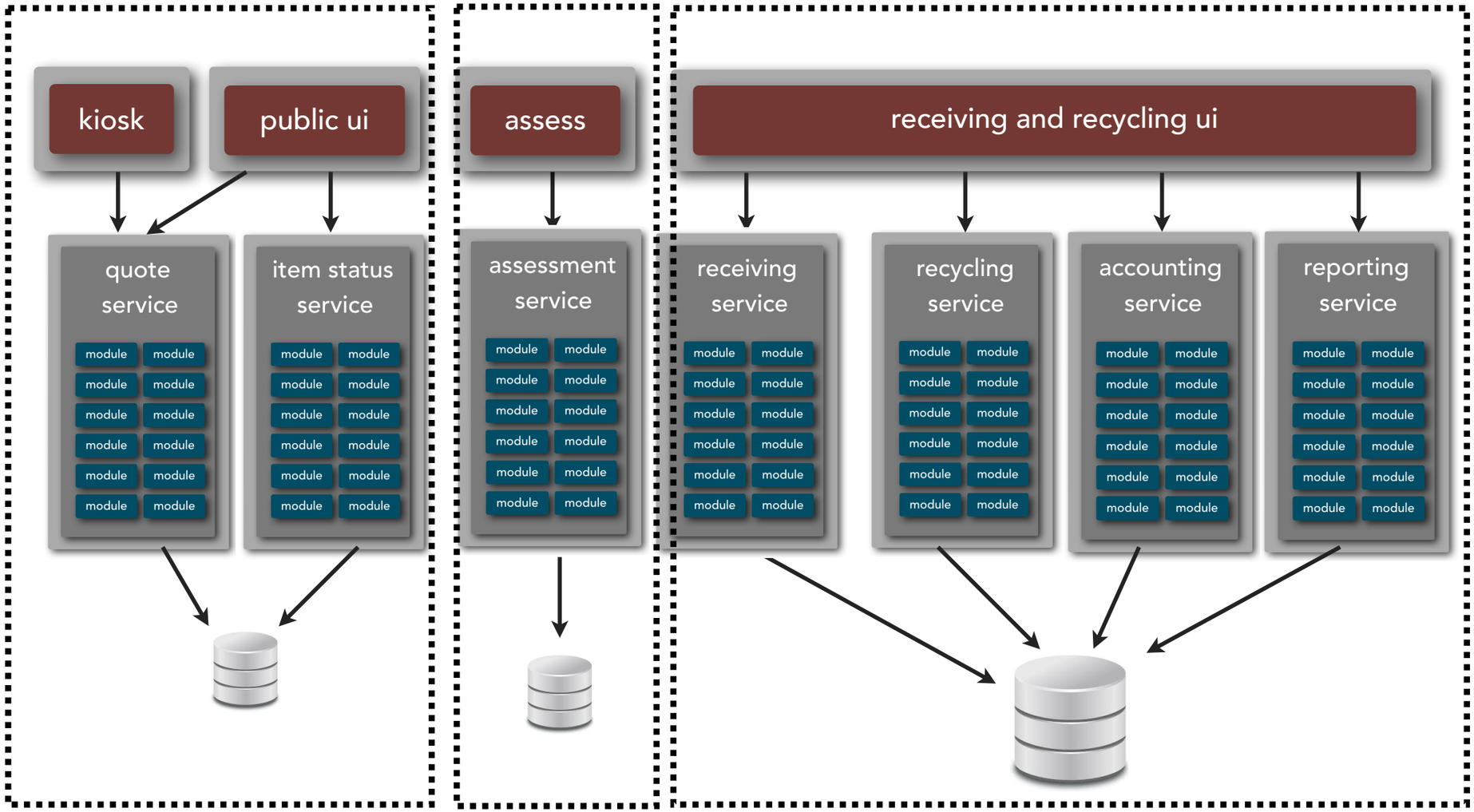
electronics recycling application



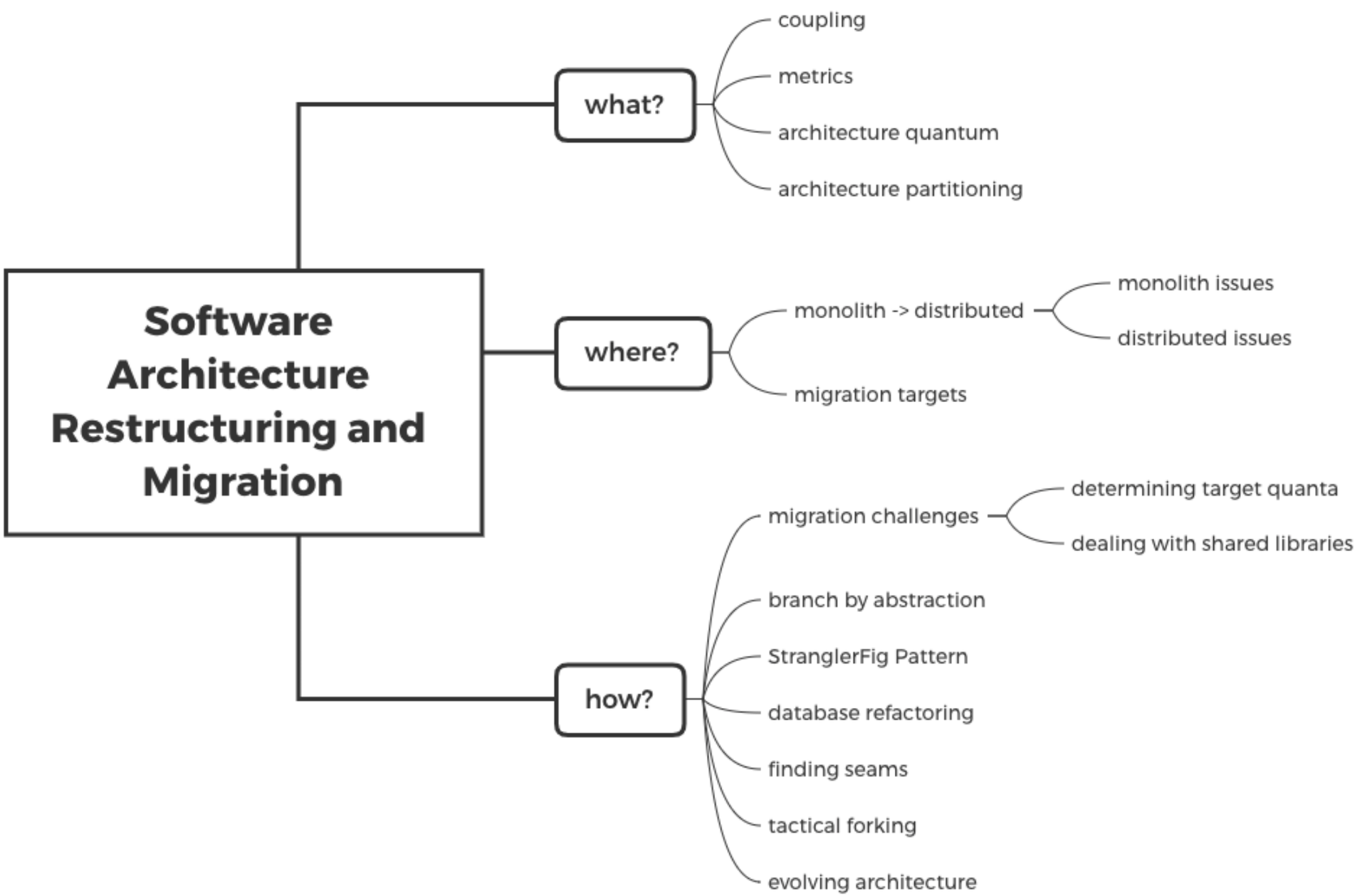
electronics recycling application



electronics recycling application



aGenDA


Software Architecture Restructuring and Migration
what?
coupling
metrics
architecture quantum
architecture partitioning
where?
monolith -> distributed
monolith issues
distributed issues
migration targets
how?
migration challenges
determining target quanta
dealing with shared libraries
branch by abstraction
StranglerFig Pattern
database refactoring
finding seams
tactical forking
evolving architecture

summary

what?

how do we assess the current architecture?

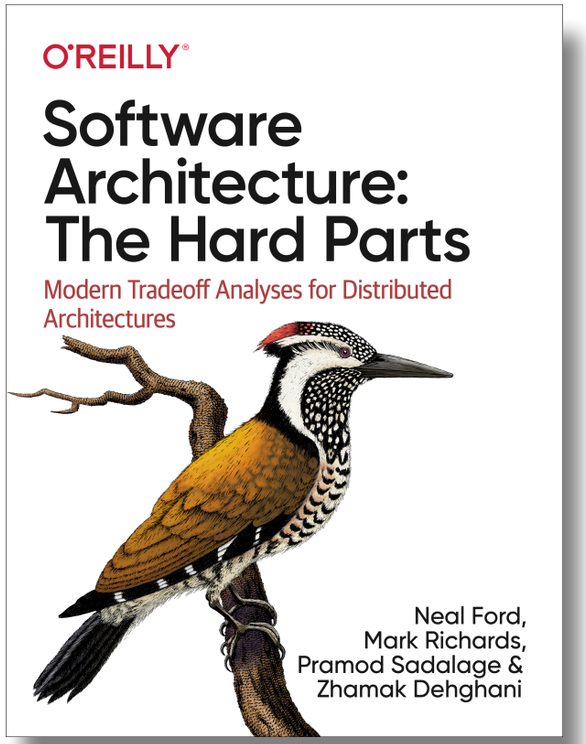
where?

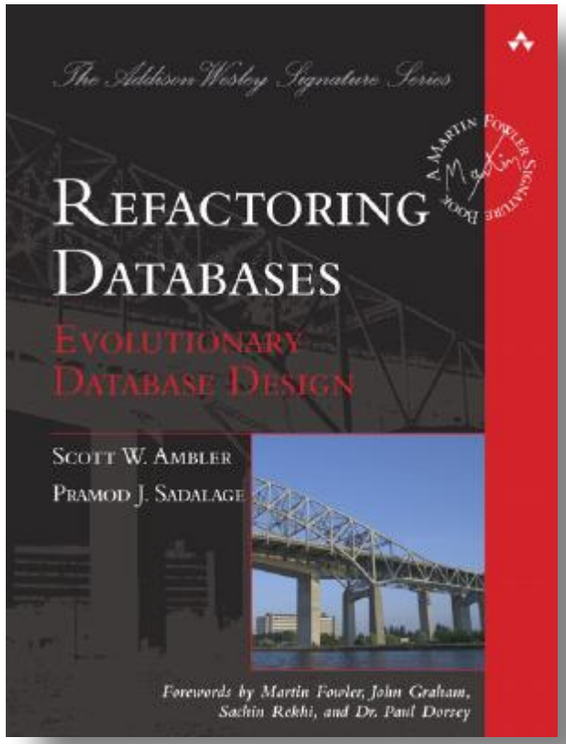
to what should we change it?

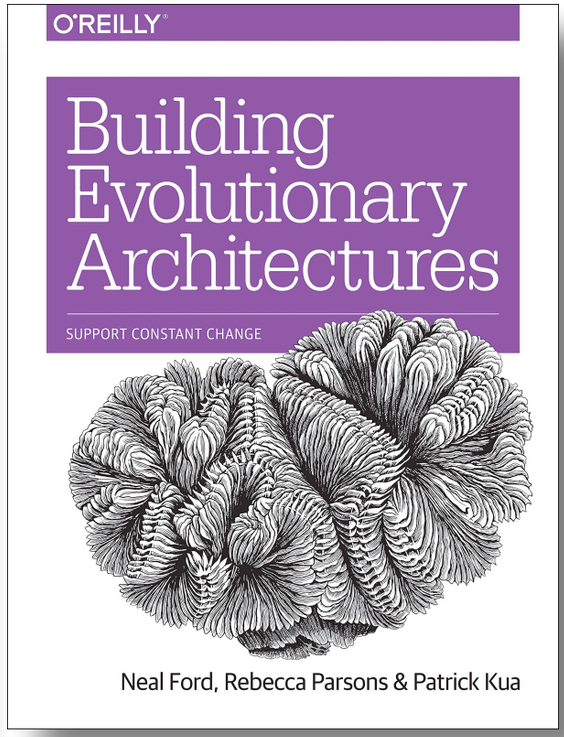
how?

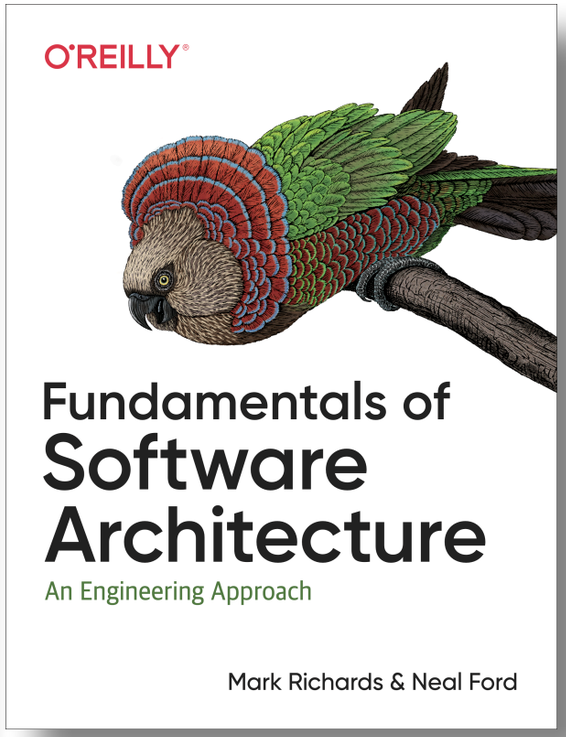
how do we change it?

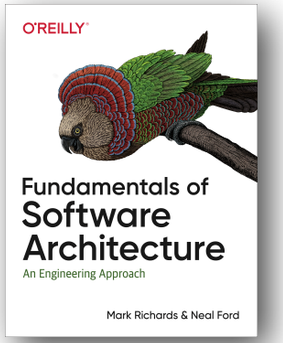
resources

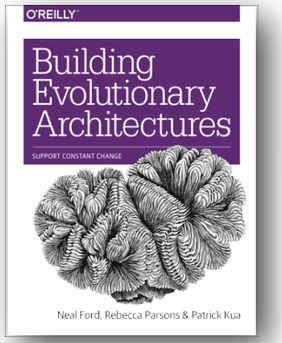

O'REILLY
Software Architecture: The Hard Parts
Modern Tradeoff Analyses for Distributed Architectures
Neal Ford, Mark Richards, Pramod Sadalage & Zhamak Dehghani

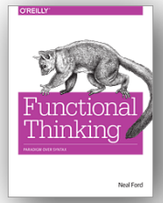

The Addison Wesley Signature Series
A MARTIN FOWLER SIGNATURE BOOK
REFACTORING DATABASES
EVOLUTIONARY DATABASE DESIGN
SCOTT W. AMBLER
PRAMOD J. SADALAGE
Forewords by Martin Fowler, John Graham, Sachin Rekhi, and Dr. Paul Dorsey

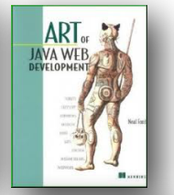

O'REILLY
Building Evolutionary Architectures
SUPPORT CONSTANT CHANGE
Neal Ford, Rebecca Parsons & Patrick Kua

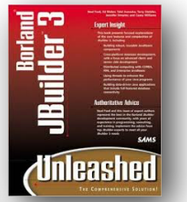

O'REILLY
Fundamentals of Software Architecture
An Engineering Approach
Mark Richards & Neal Ford

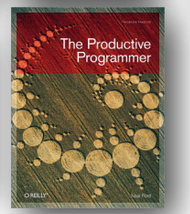


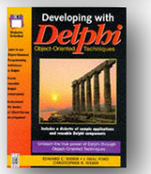


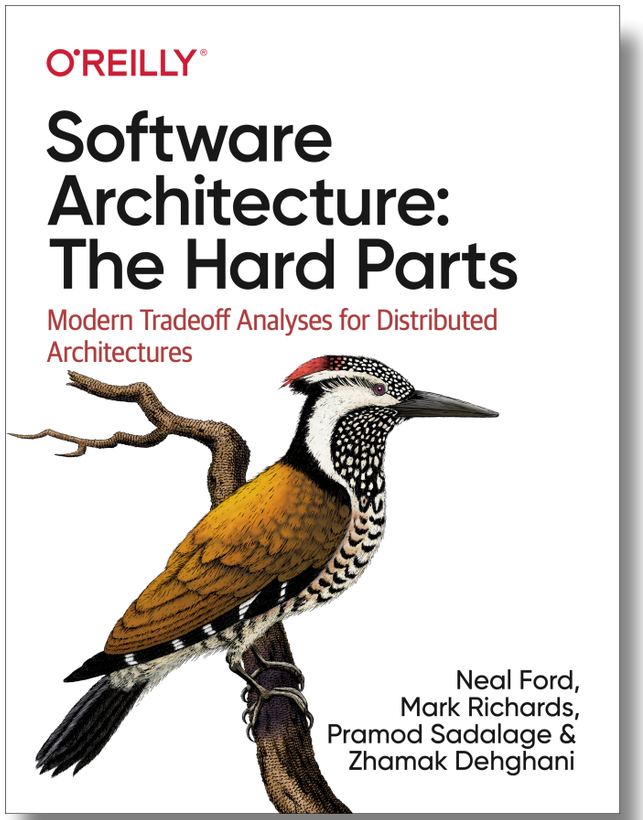


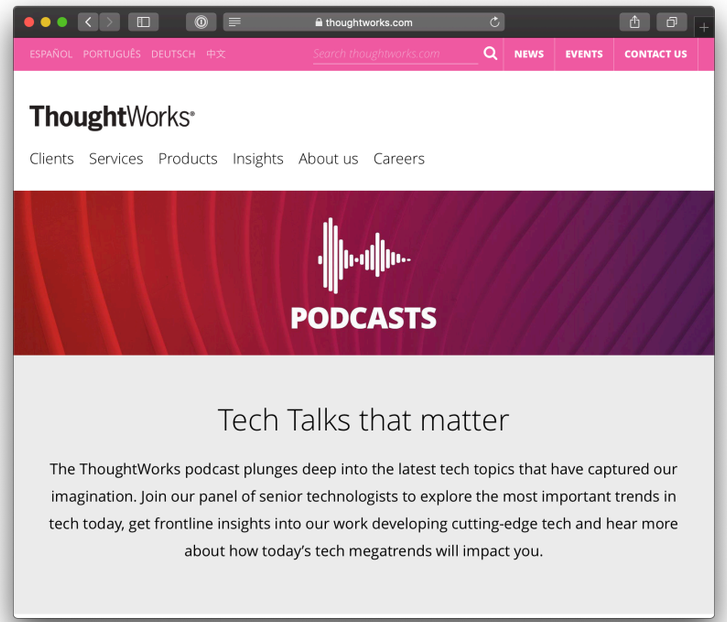












www.thoughtworks.com/podcasts

Neal Ford

director / software architect / meme wrangler

/thoughtworks

@neal4d

http://nealford.com

O'REILLY®

learning.oreilly.com/live-training/