

Software Architecture Fundamentals Workshop

Part 2: A Deeper Dive



Mark Richards

Independent Consultant

Hands-on Enterprise / Integration Architect

Published Author / Conference Speaker

<http://www.wmrichards.com>

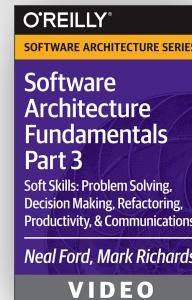
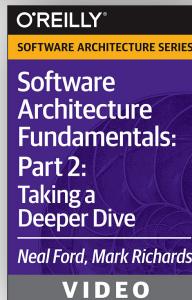
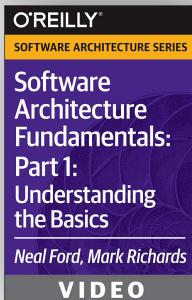
<http://www.linkedin.com/pub/mark-richards/0/121/5b9>



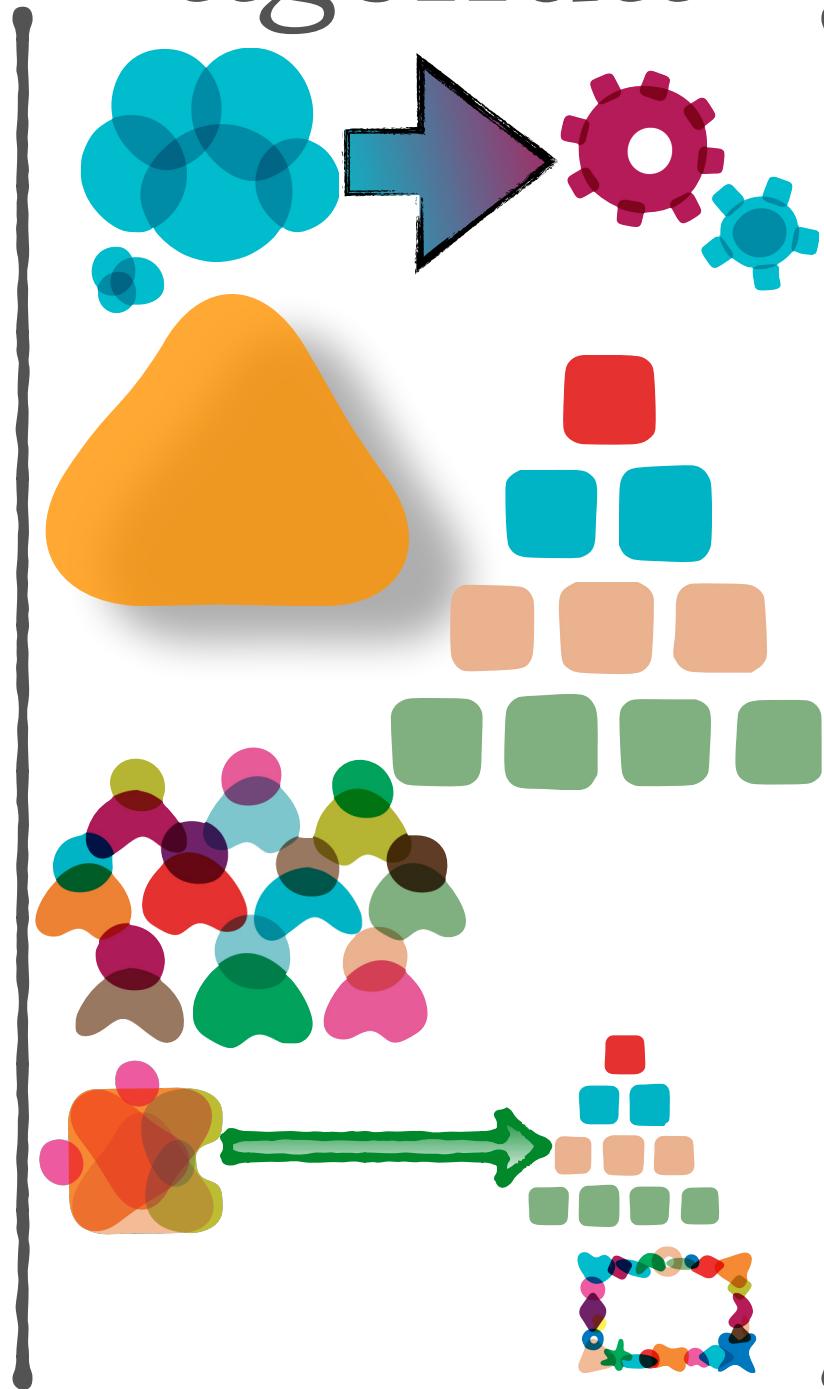
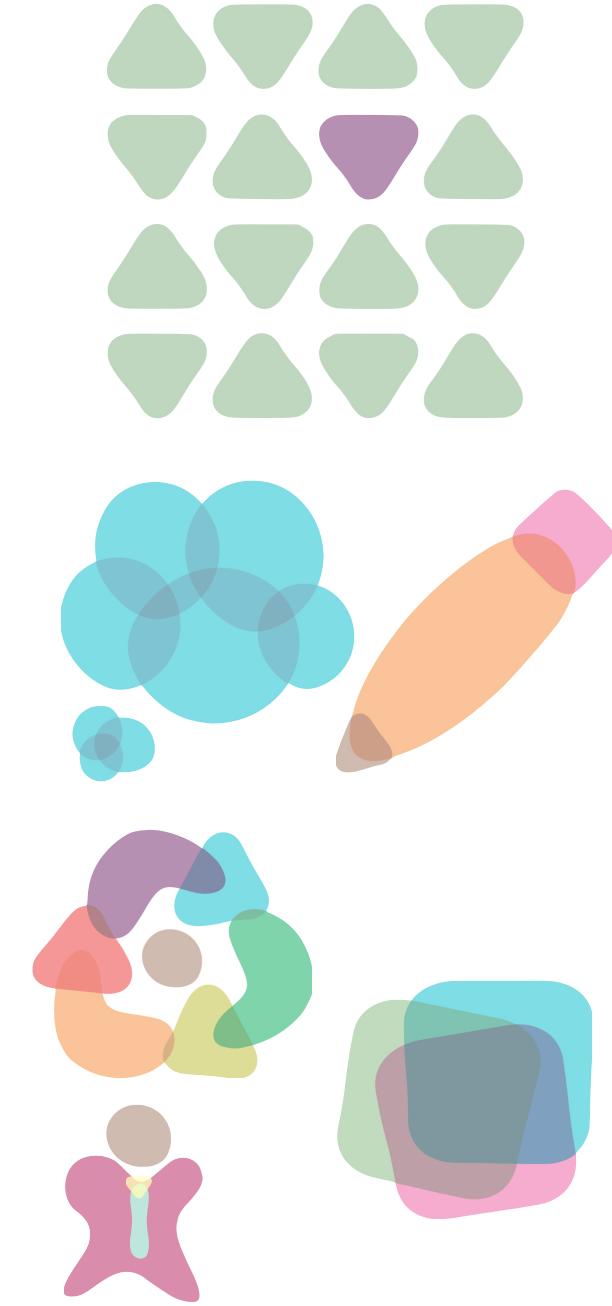
ThoughtWorks®

NEAL FORD

Director / Software Architect / Meme Wrangler



agenda



nealford.com

Neal Ford | Author, Thoughtworker, & Meme Wrangler...

Architectural Katas

inspired by Ted Neward's original [Architectural Katas](#)

"How do we get great designers?
Great designers design,
of course."
Fred Brooks

"So how are we supposed
to get great architects, if
they only get the chance
to architect fewer than
a half-dozen times in
their career?"
Ted Neward

About
Architectural Katas are intended as a small-group (3-5 people) exercise, usually as part of a larger group (4-10 groups are ideal), each of whom is doing a different kata. A Moderator keeps track of time, assigns Katas (or allows this website to choose one randomly), and acts as the facilitator for the exercise.
The Architectural Katas started as a presentation workshop by Ted Neward. They've taken on a life of their own. [Learn more](#) »

Rules
Doing an Architectural Kata requires you to obey a few rules in order to get the maximum out of the activity. [Read Rules](#) »

Lead
Want to run the Architectural Katas yourself? There's only a few things you need to know before you do. Ted Neward, the originator of Architectural Katas, has information on his site about leading Kata exercises. [Read on the original site](#) »

List Katas »

Random Kata »

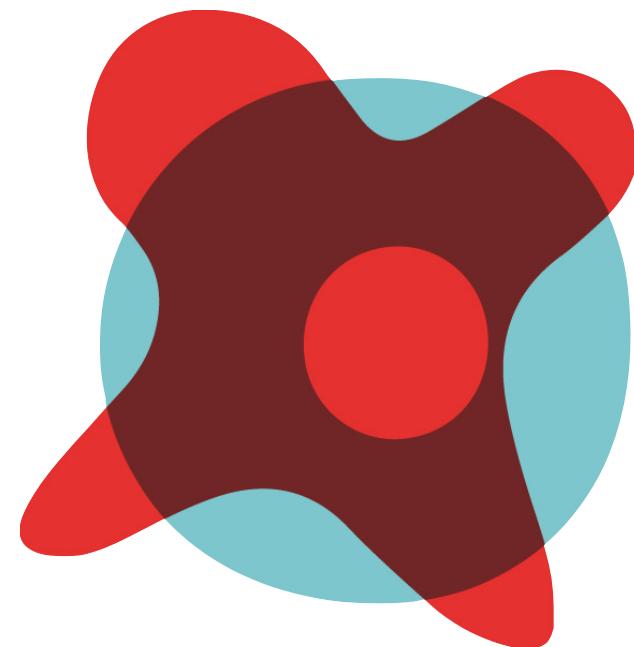
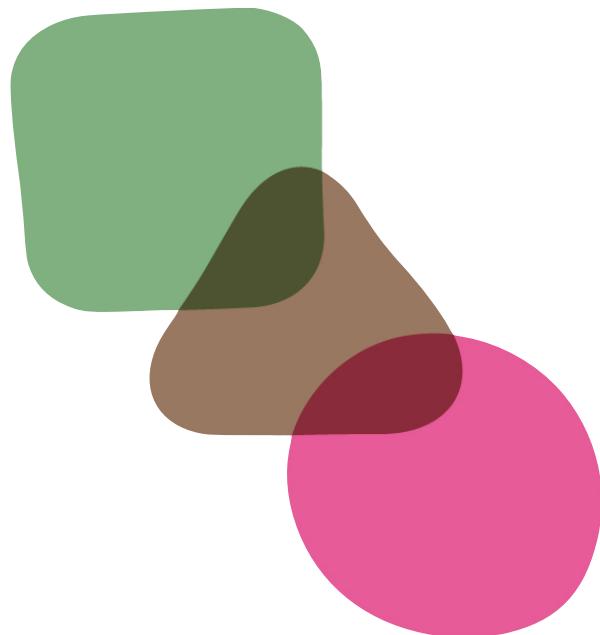
Follow Neal on Twitter at @neal4d

- Neal works at [ThoughtWorks](#), a very interesting place.
- Neal speaks frequently on the [No Fluff, Just Stuff](#) conference circuit.
- [Meme Agora](#) RSS feed.

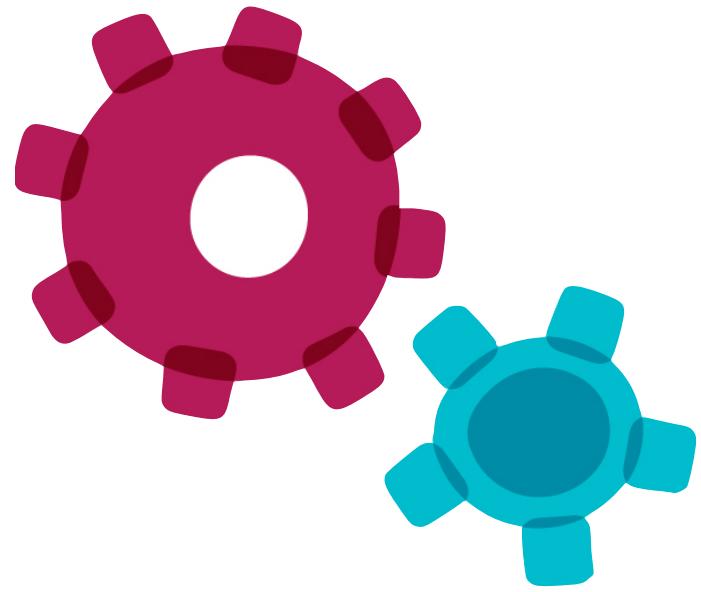
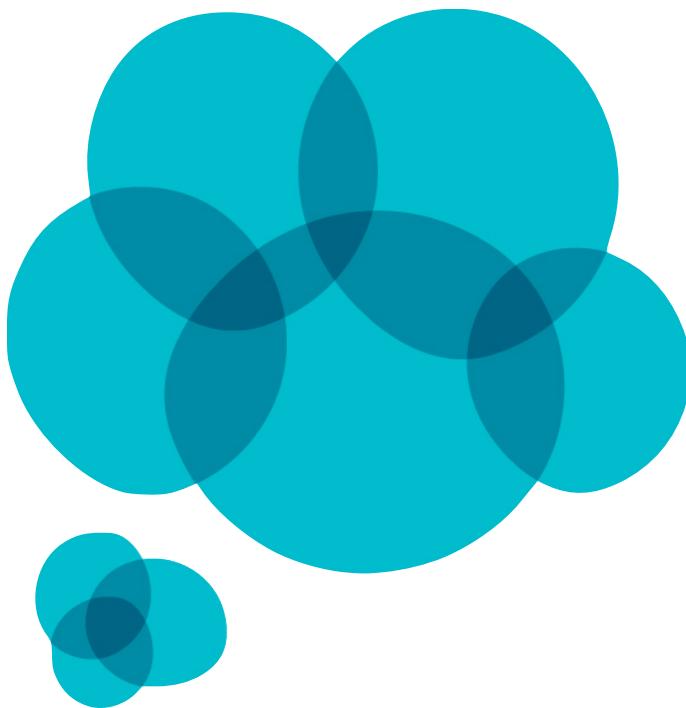
nealford.com/katas/



Software architecture
reflects the mapping
between capabilities
and constraints.



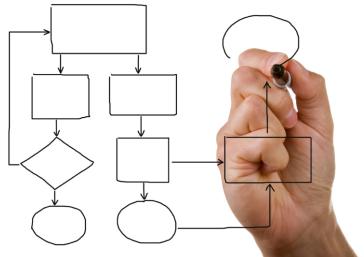
translating requirements



translating requirements



“our business is constantly changing
to meet new demands of the
marketplace”

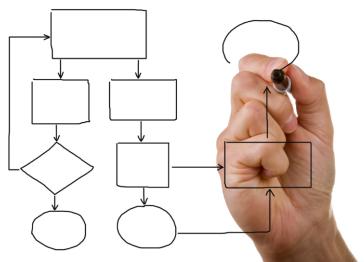


???

translating requirements



“due to new regulatory requirements,
it is imperative that we complete end-
of-day processing in time”

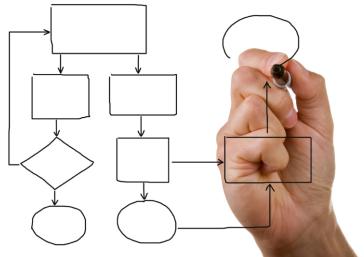


???

translating requirements



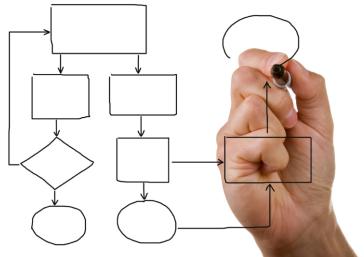
“we need faster time to market to remain competitive”



translating requirements



“our plan is to engage heavily in mergers and acquisitions in the next three years”

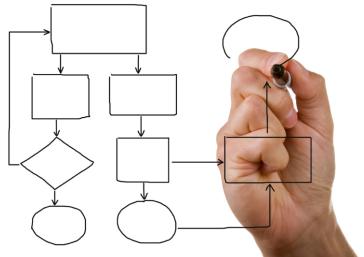


???

translating requirements

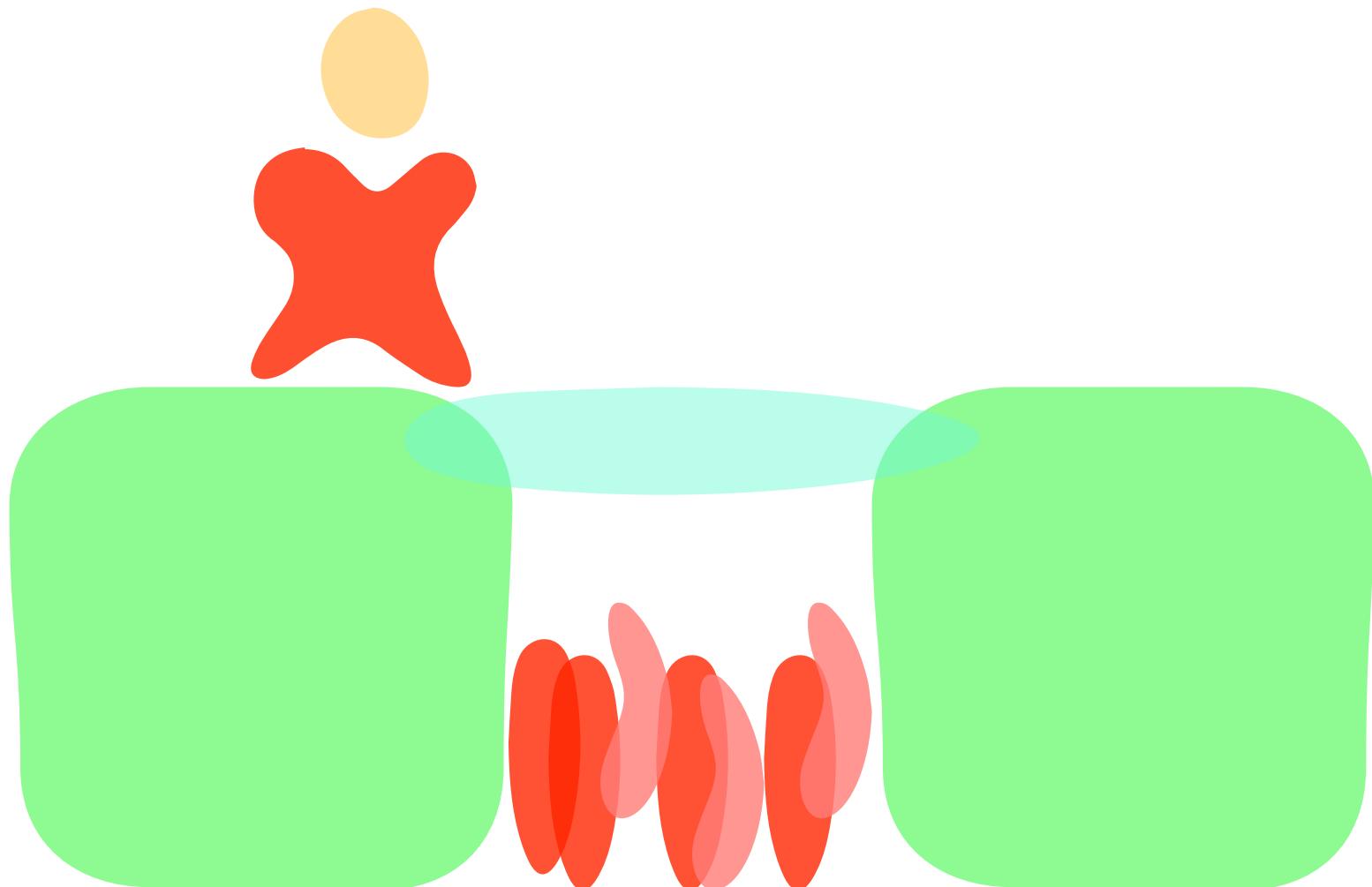


“we have a very tight timeframe and budget for this project”





architecture pitfall



armchair architecture

whiteboard sketches are handed off as final architecture standards without proving out the design



armchair architecture

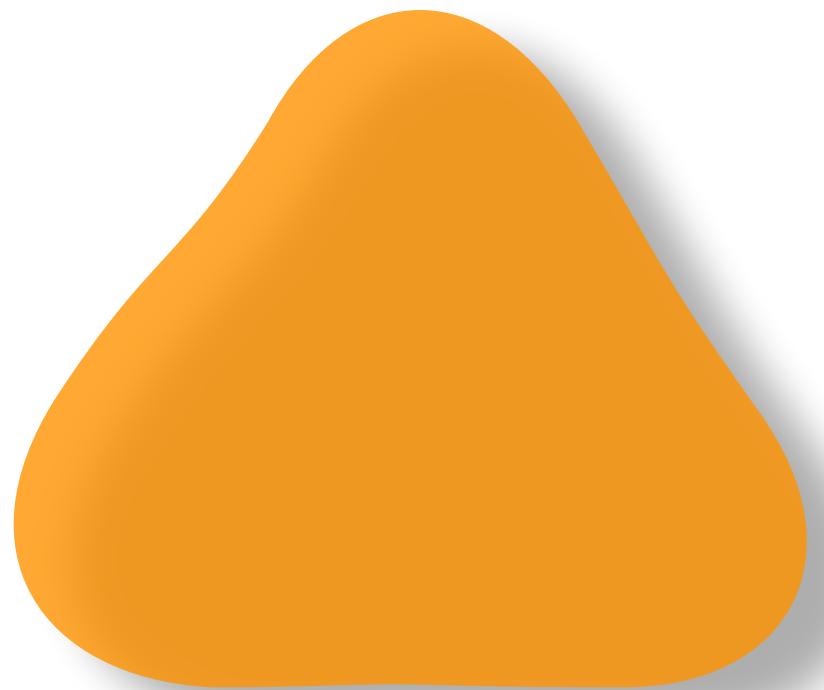
occurs when you have non-coding architects

occurs when architects are not involved in the full project lifecycle

occurs when architects don't know what they are doing



architecting for change



architecting for change

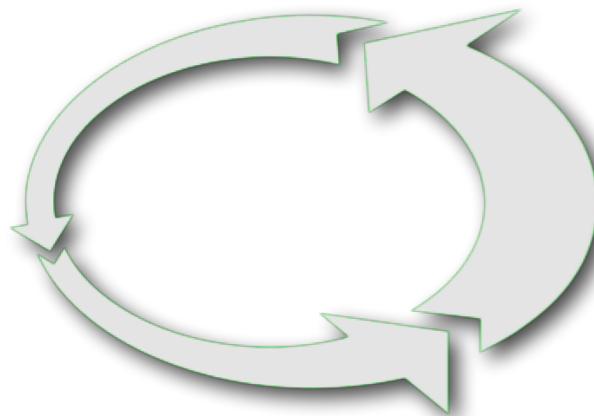
business is in a constant state of change

increased
competition

mergers

regulatory
changes

growth



acquisitions

architecting for change

technology is in a constant state of change

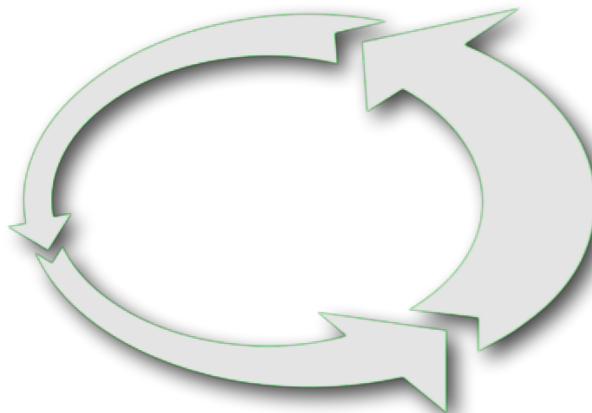
platforms

products

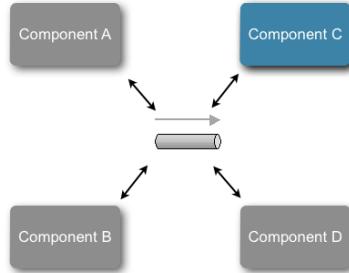
languages

patterns

frameworks



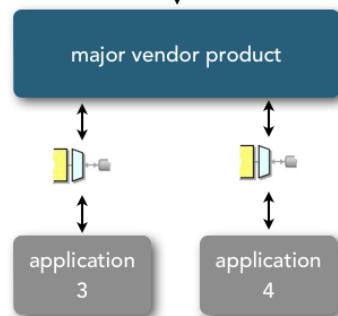
techniques for change



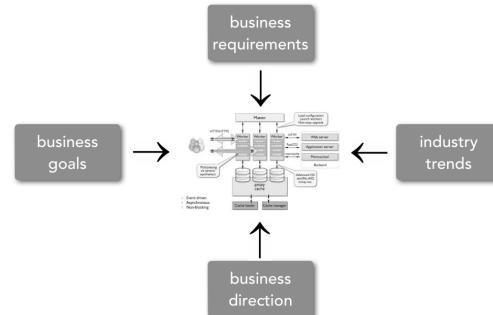
reduce dependencies



leverage standards

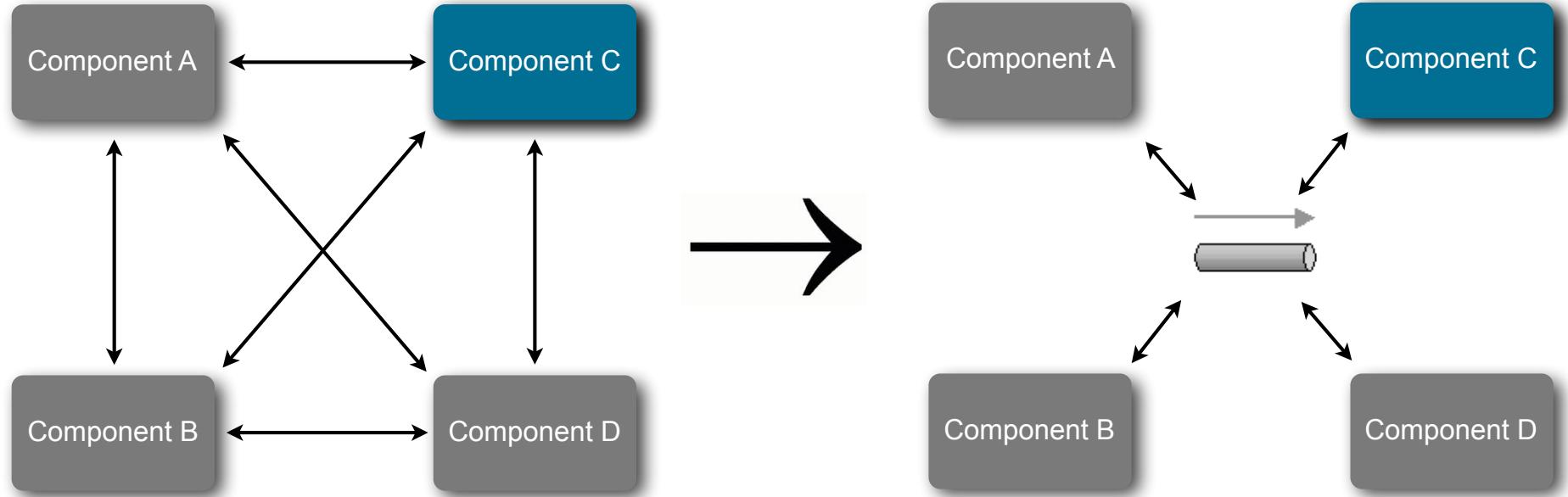


create product-agnostic
architectures



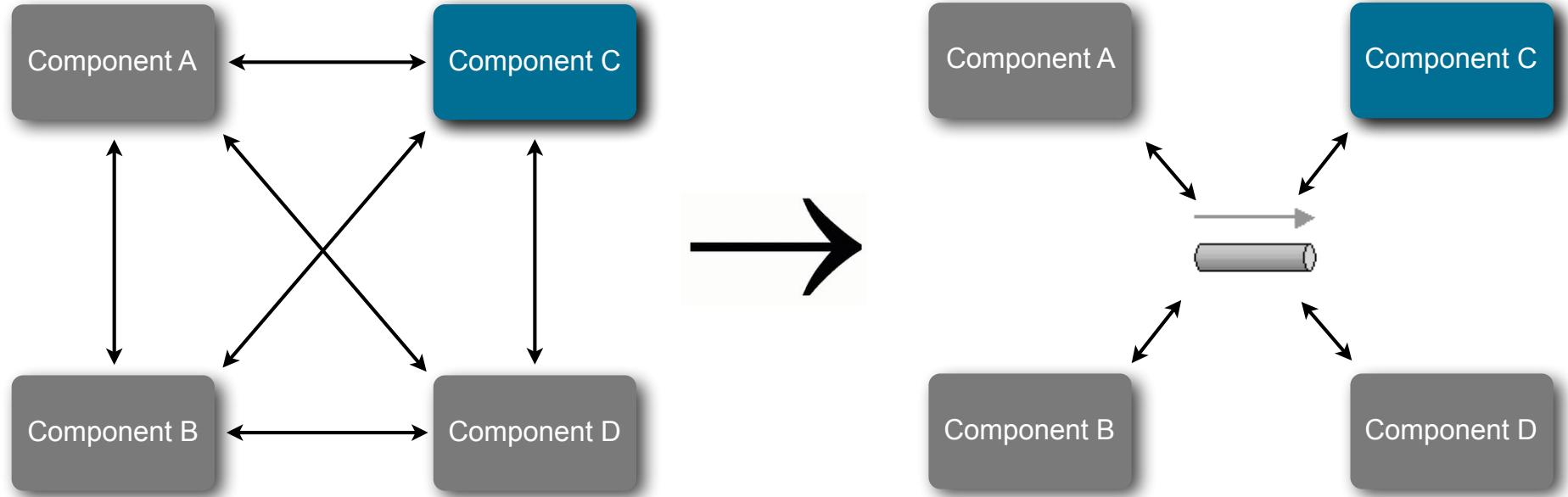
create domain-specific
architectures

reduce dependencies



less dependencies → easier to update (independent modules)

reduce dependencies



messaging service bus adapters
architecture patterns

leverage standards

industry standards

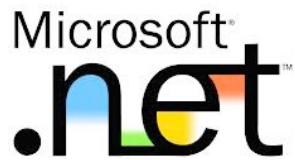


<?xml?>



leverage standards

corporate standards



ORACLE



leverage standards

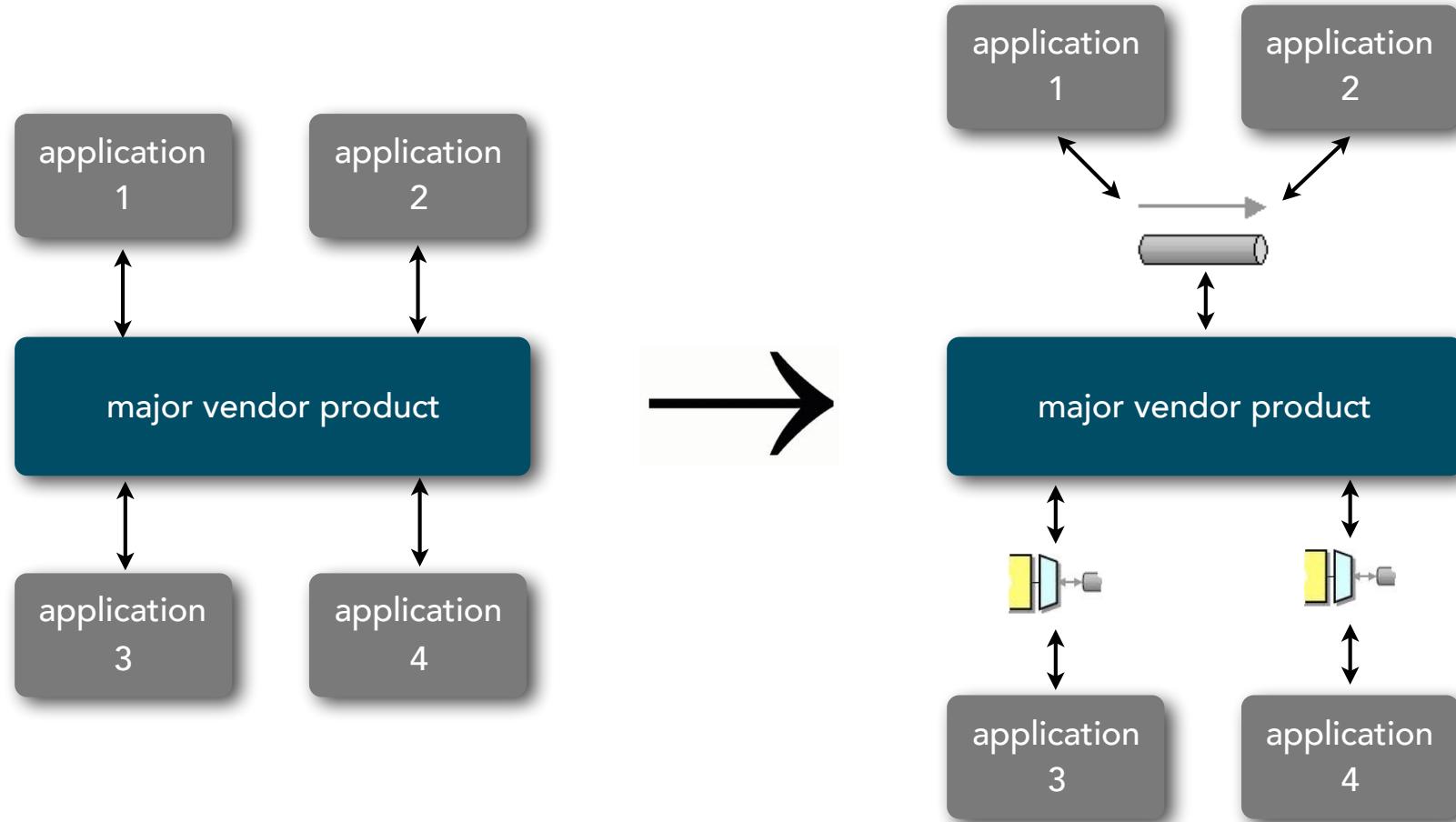
standards may not always be your first choice, but they significantly help in reducing the effort for change

larger resource pool

better integration with other systems

product-agnostic architecture

isolate products to avoid vendor lock-in

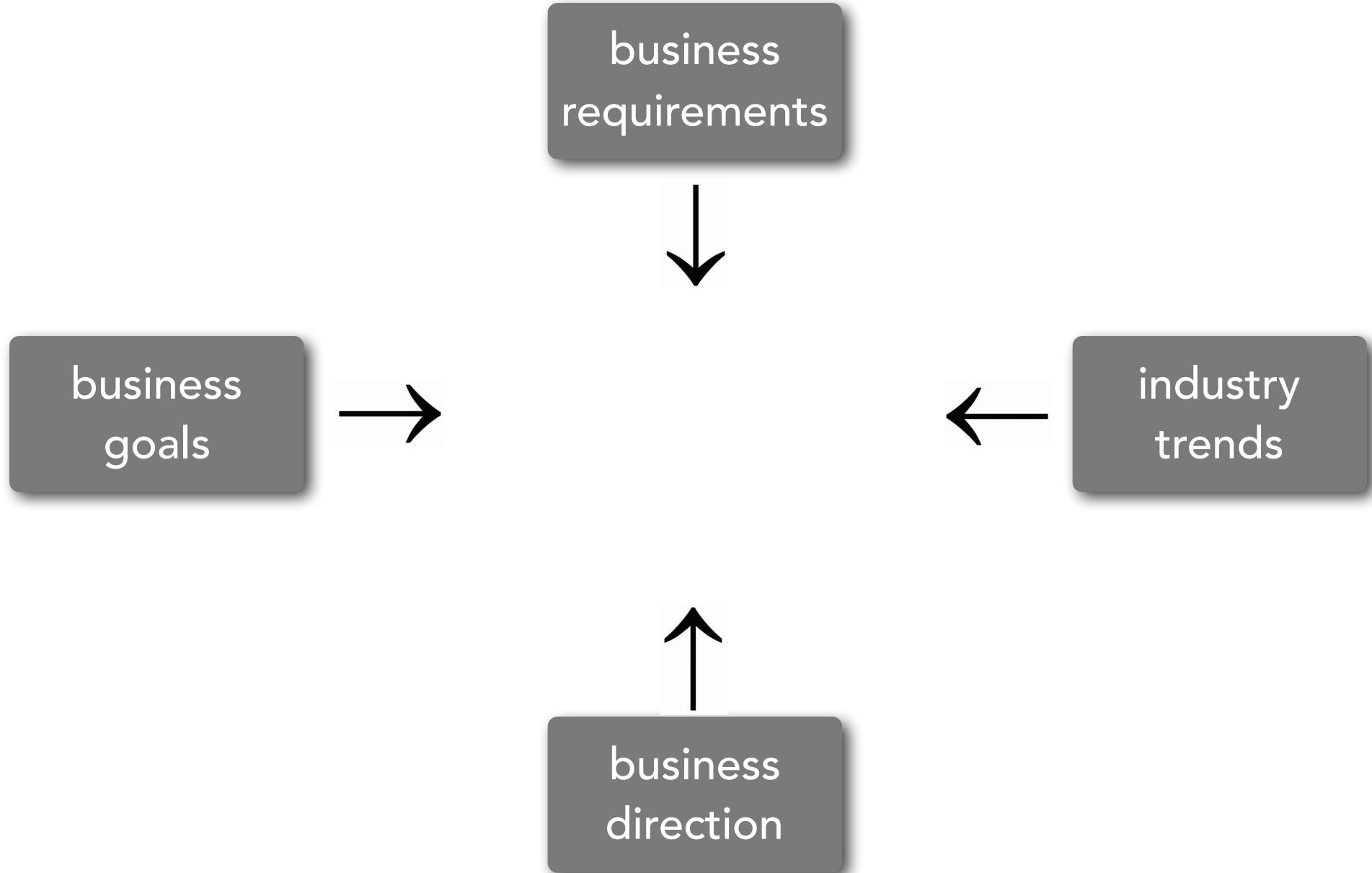


domain-specific architecture

generic architectures are difficult to change
because they are too broad and take into
account scenarios that aren't used

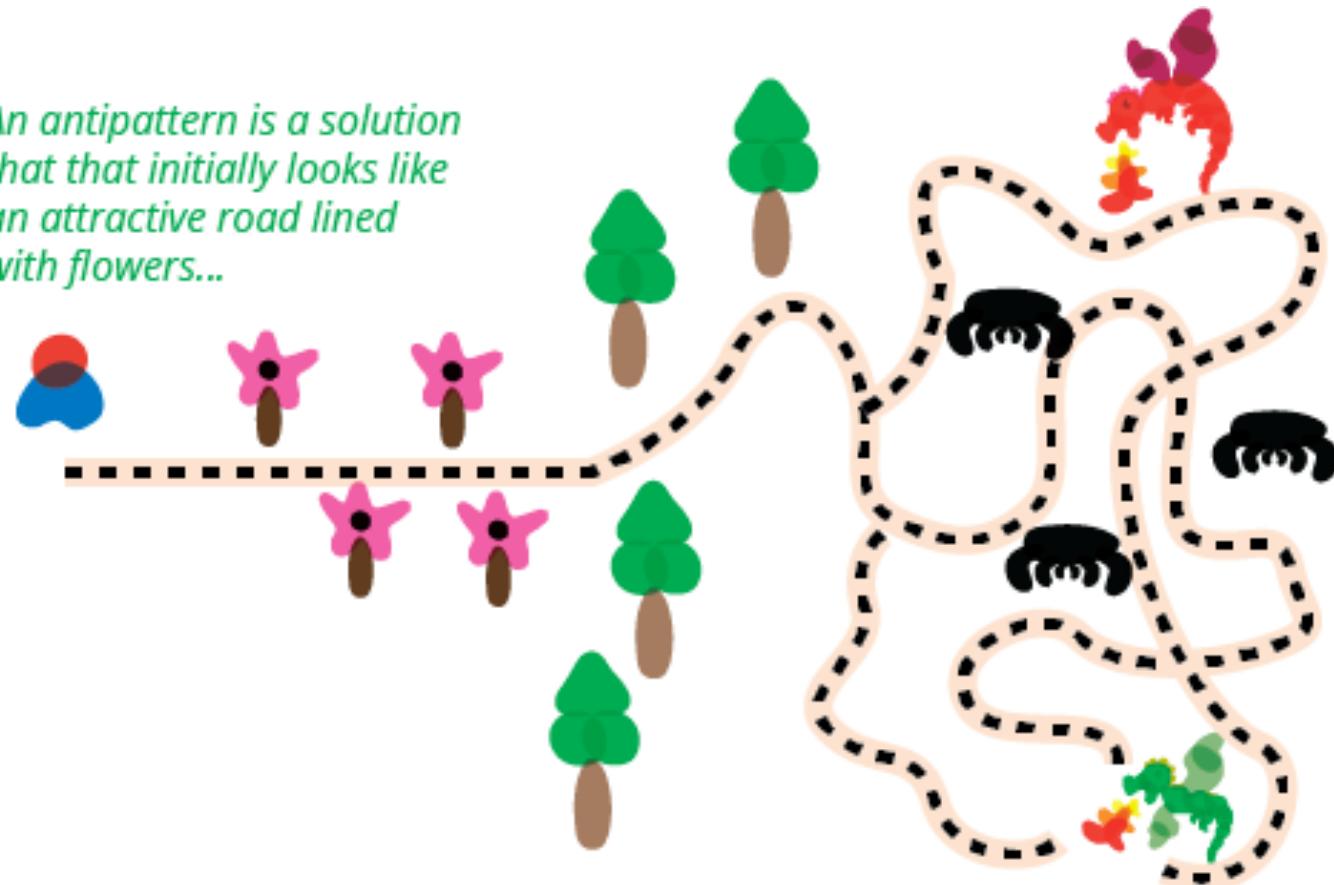
limit the scope of the architecture by taking
into account drivers, requirements,
business direction, and industry trends

domain-specific architecture



Architecture Anti-pattern

*An antipattern is a solution
that initially looks like
an attractive road lined
with flowers...*



*...but further on leads you into
a maze filled with monsters*

vendor king

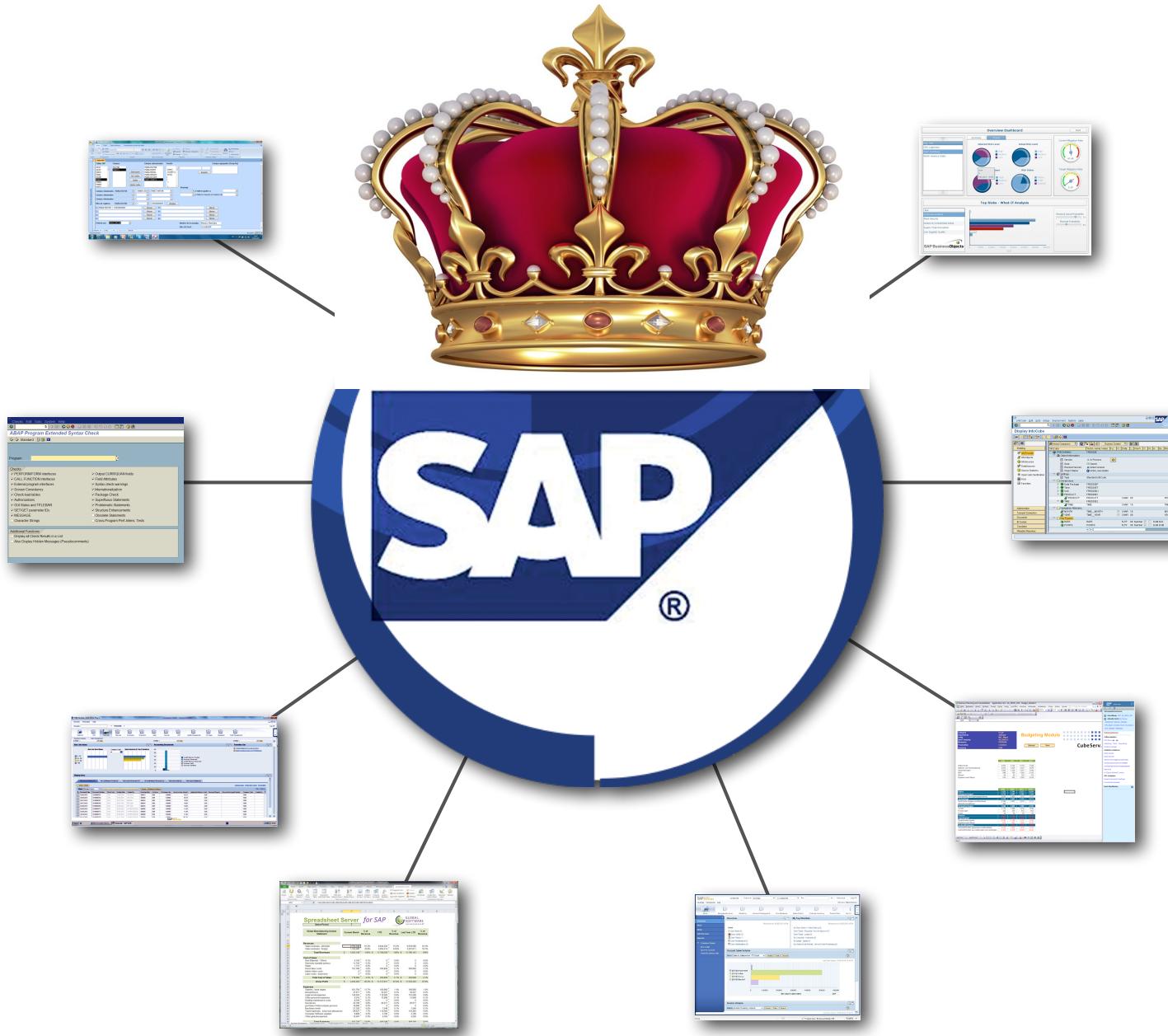
product-dependent architectures leading to a loss
of control of architecture and development costs



vendor king

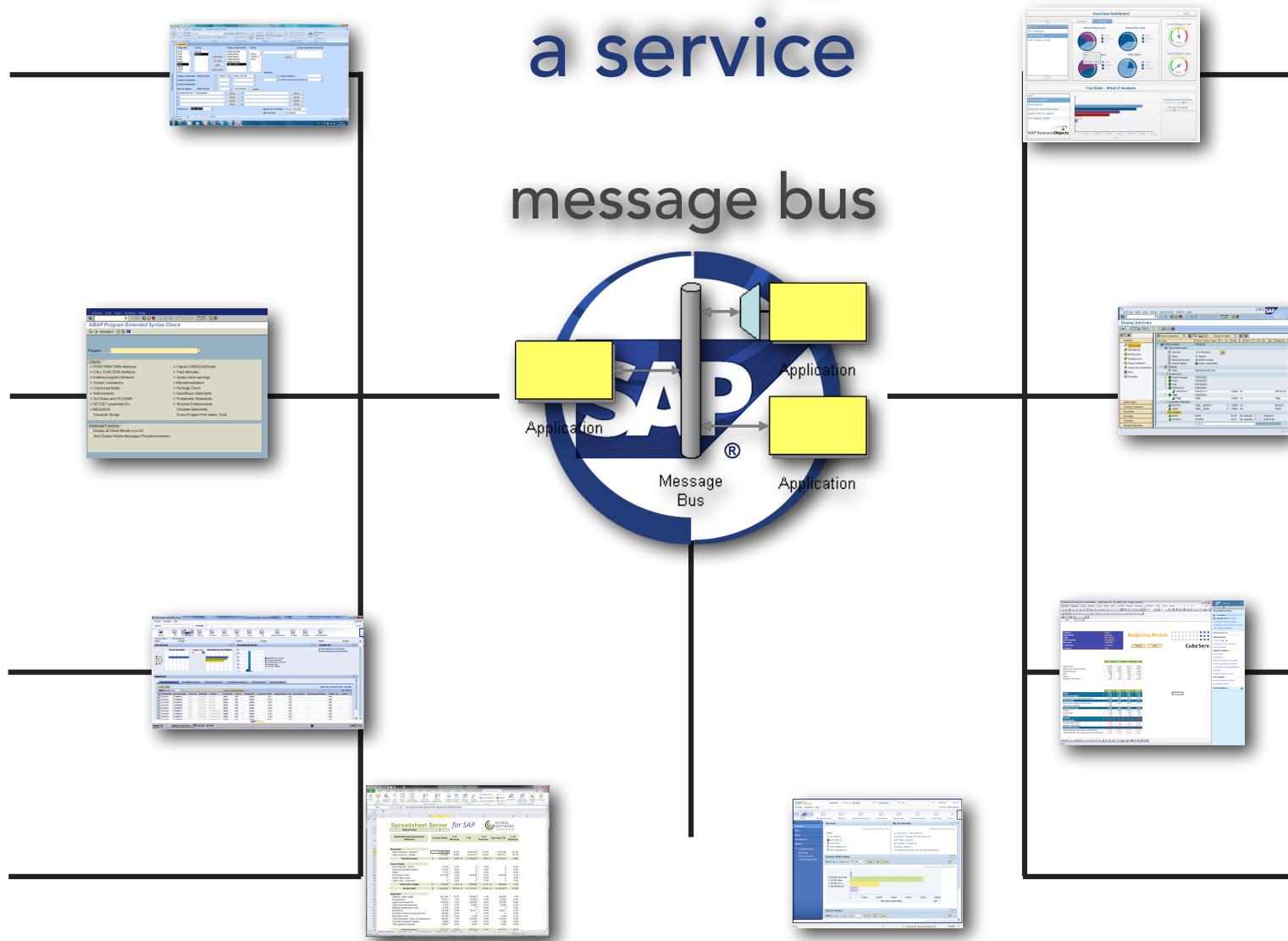


vendor king

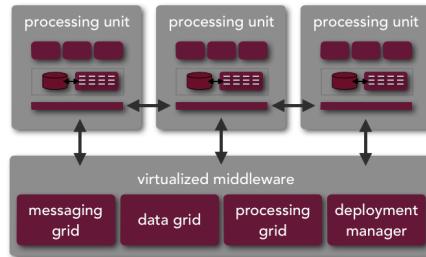


vendor king

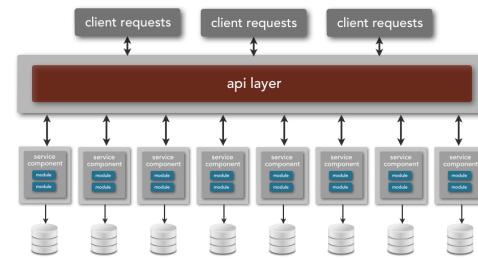
vendor app as a service



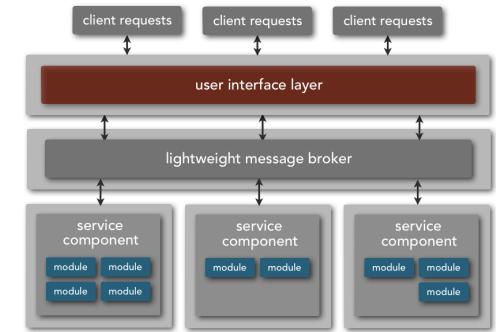
(More) Architecture Patterns



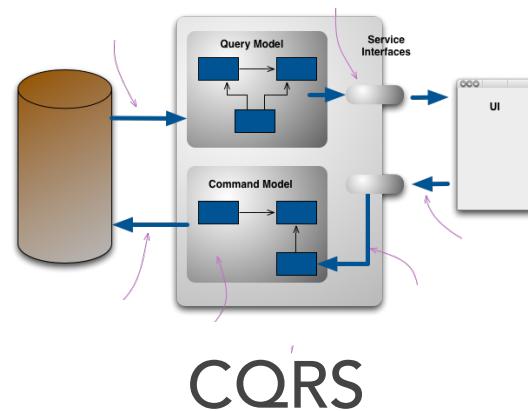
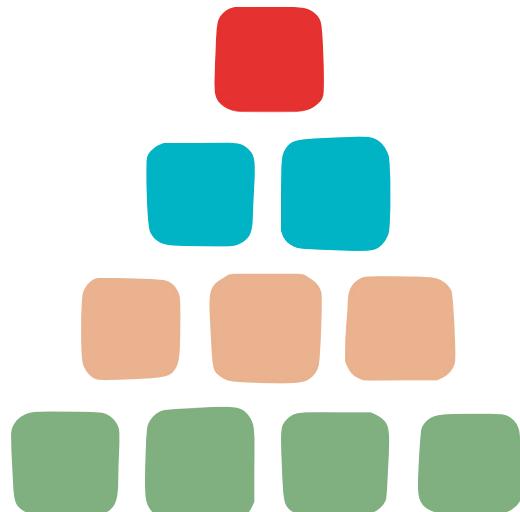
space-based
architecture



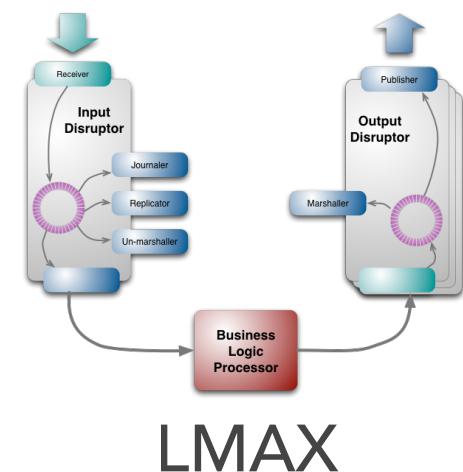
microservices
architecture



service-based
architectures



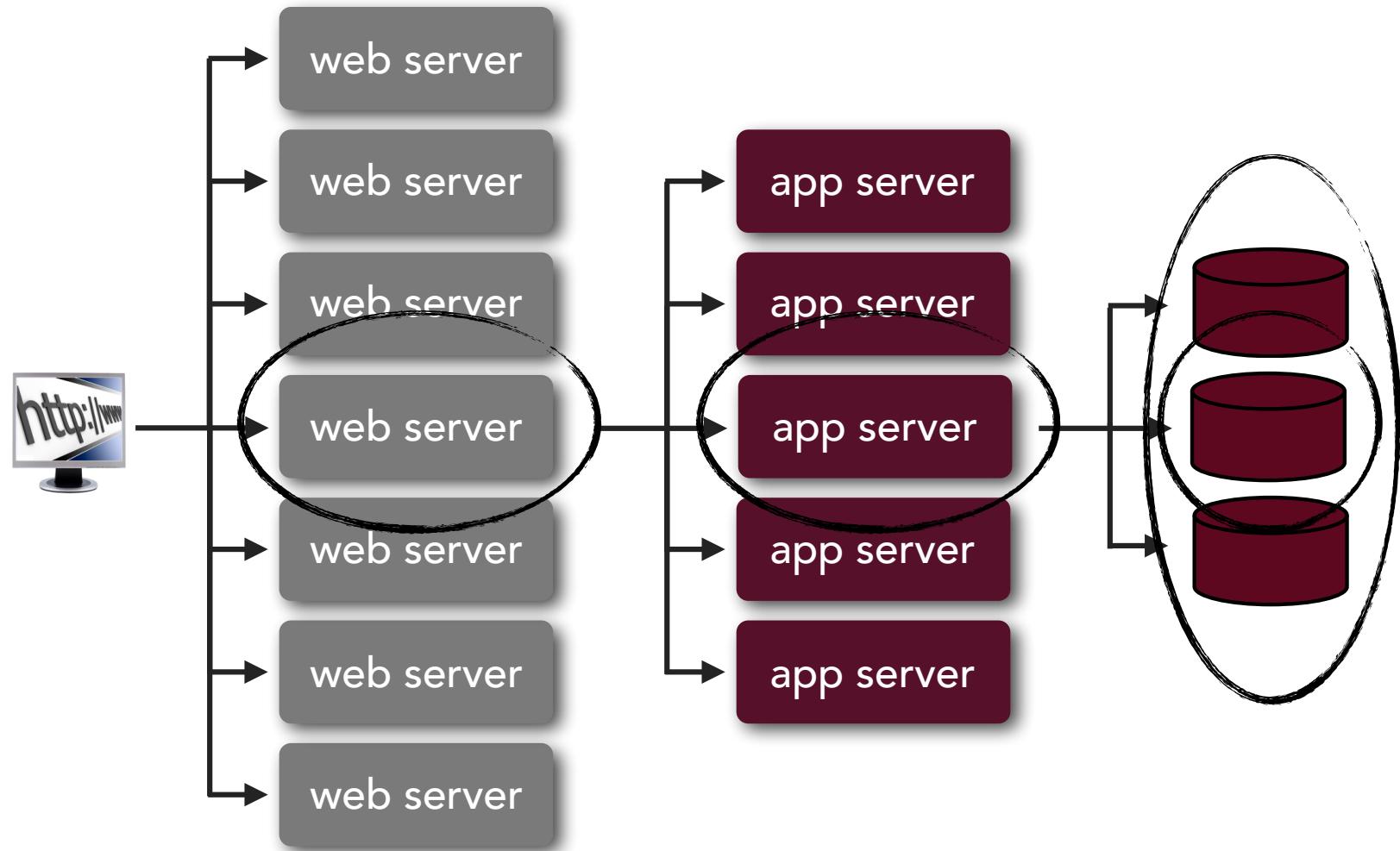
CQRS



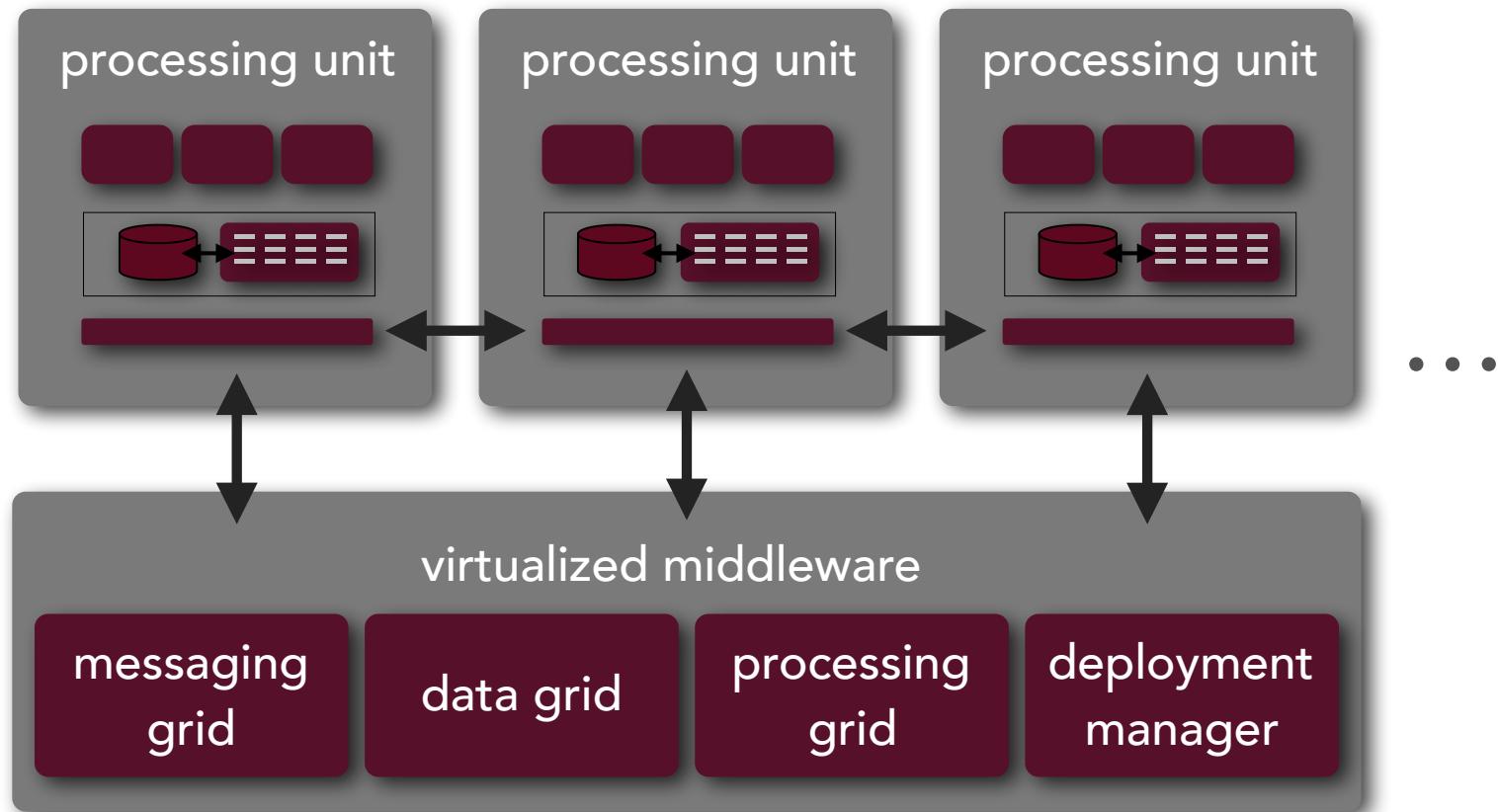
LMAX

space-based architecture

let's talk about scalability for a moment...

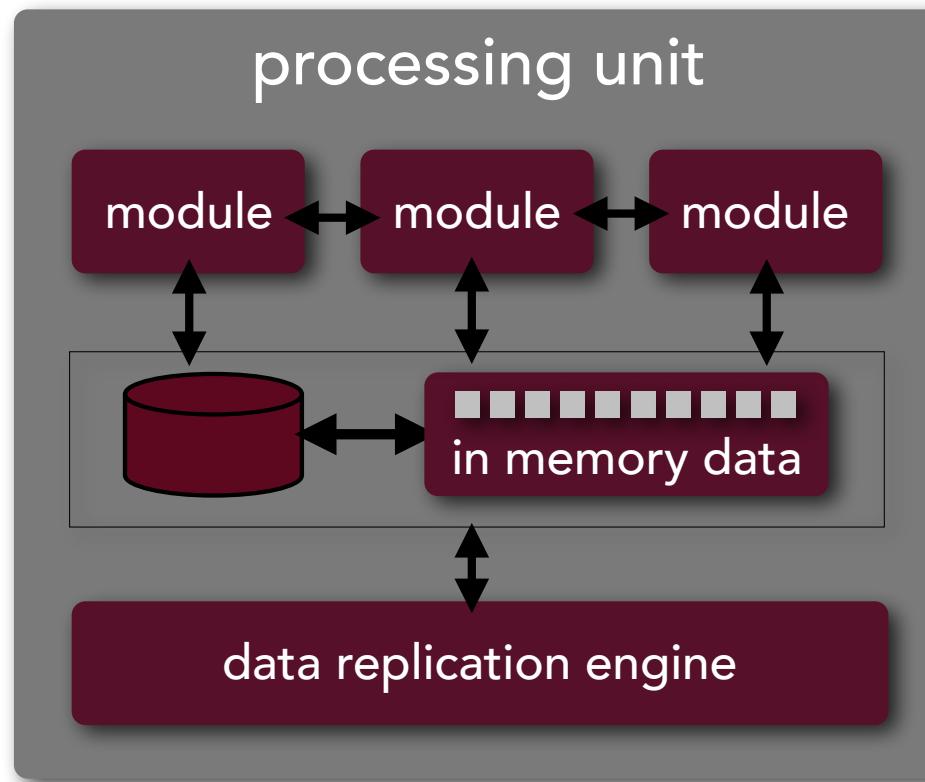


space-based architecture



space-based architecture

processing unit



space-based architecture

middleware

messaging
grid

data grid

processing
grid

deployment
manager

space-based architecture

middleware

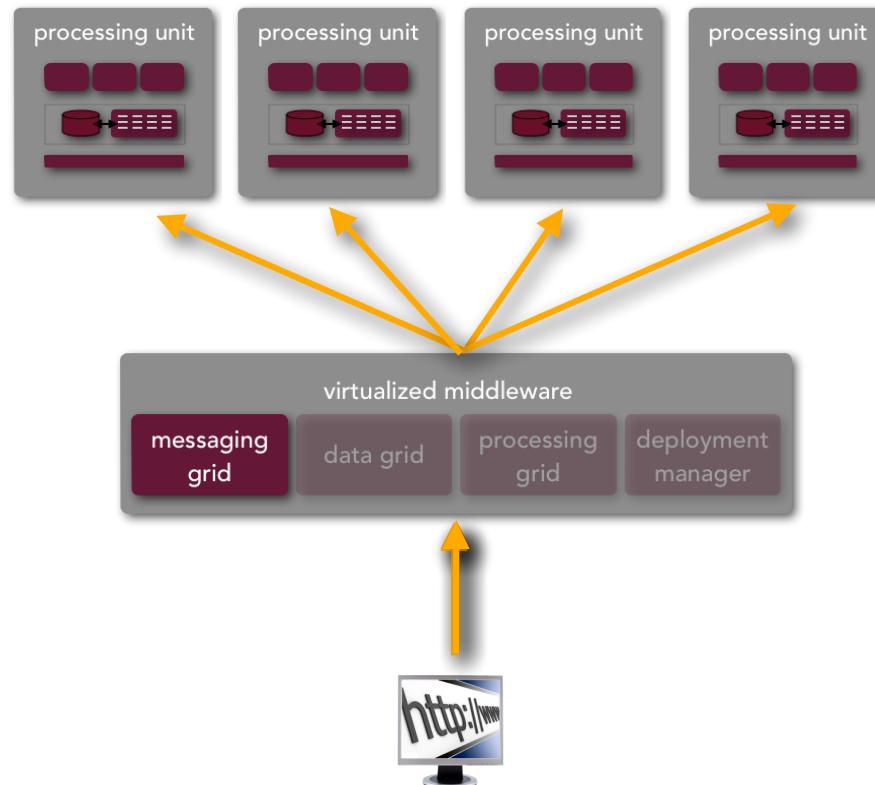
messaging
grid

data grid

processing
grid

deployment
manager

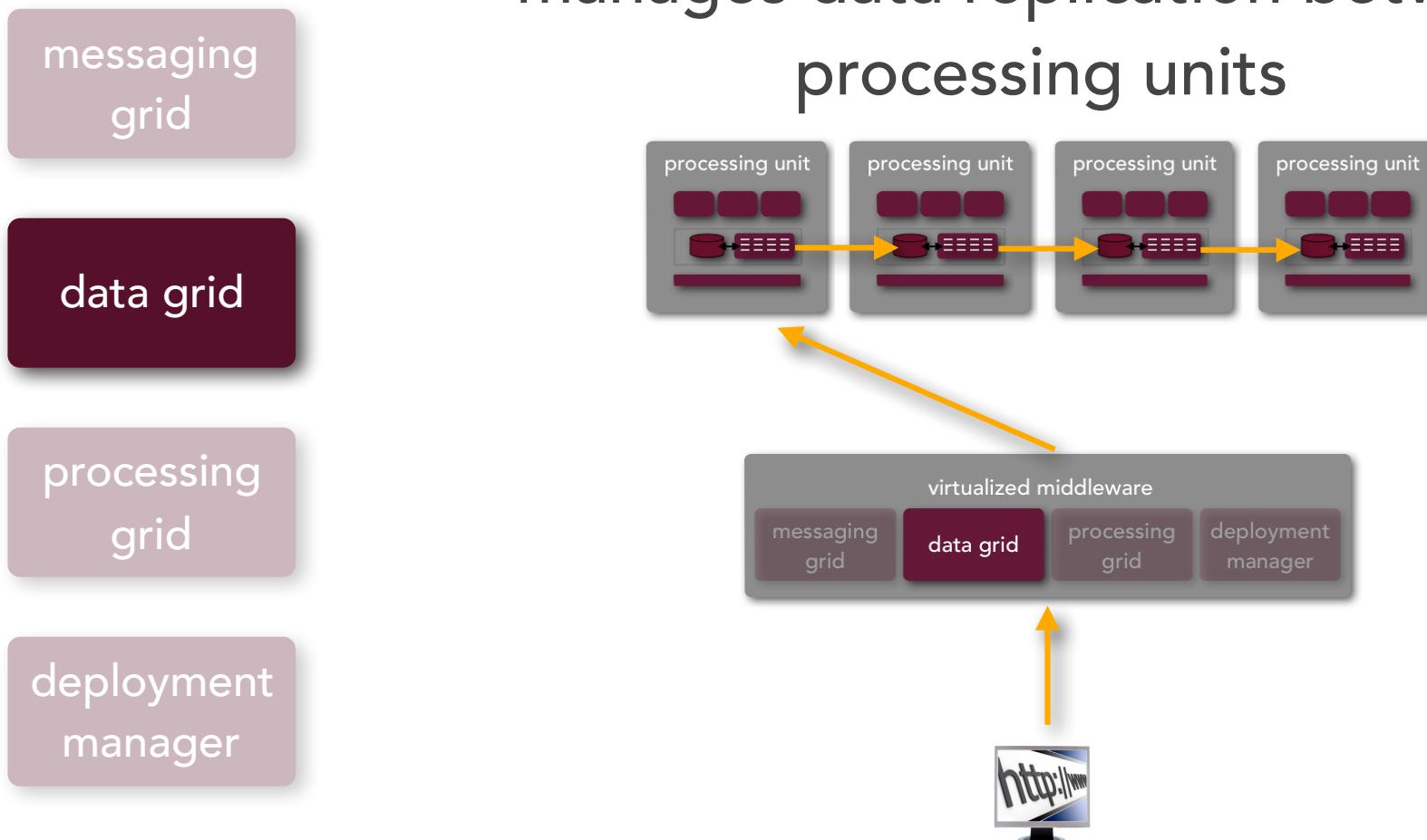
manages input request and session



space-based architecture

middleware

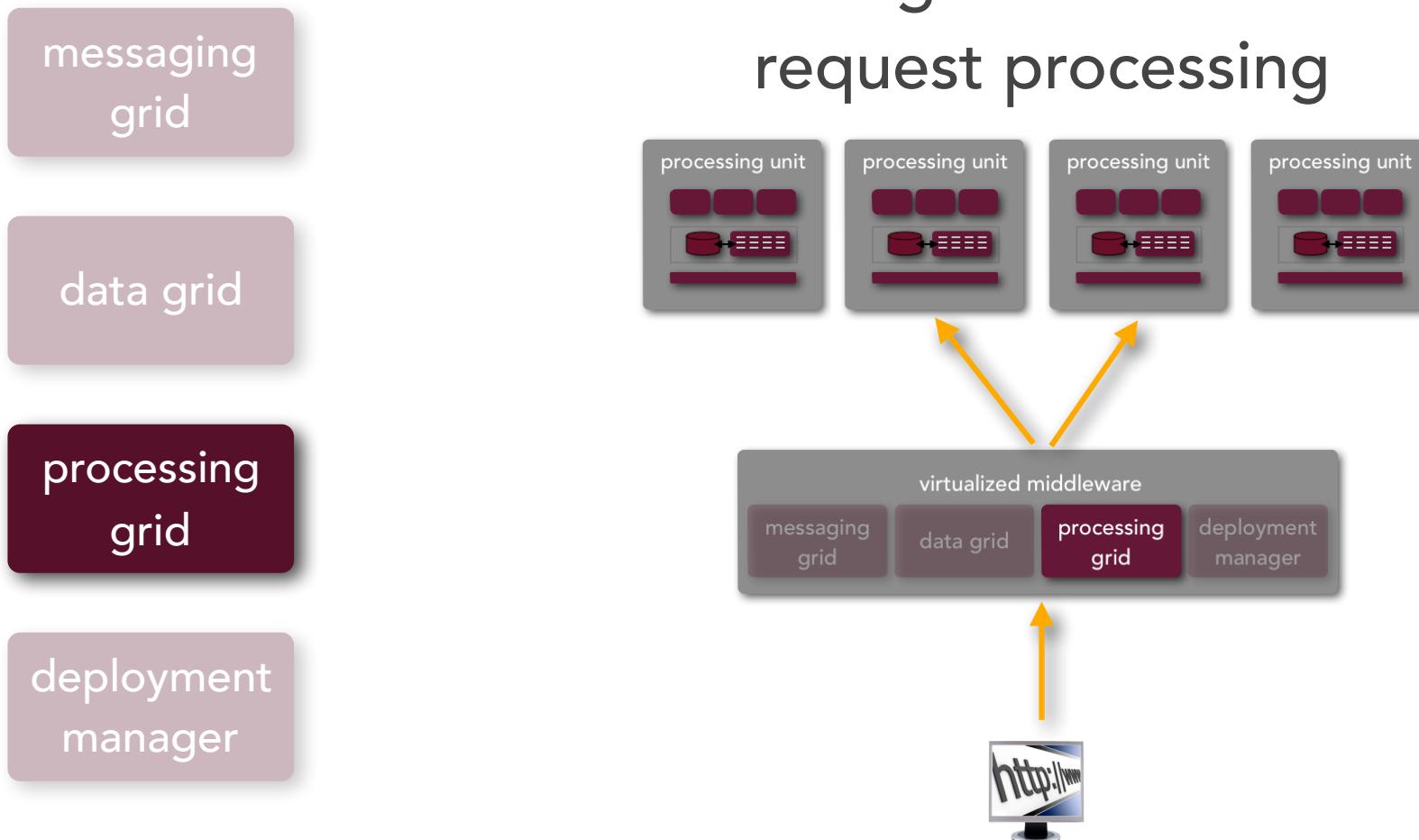
manages data replication between
processing units



space-based architecture

middleware

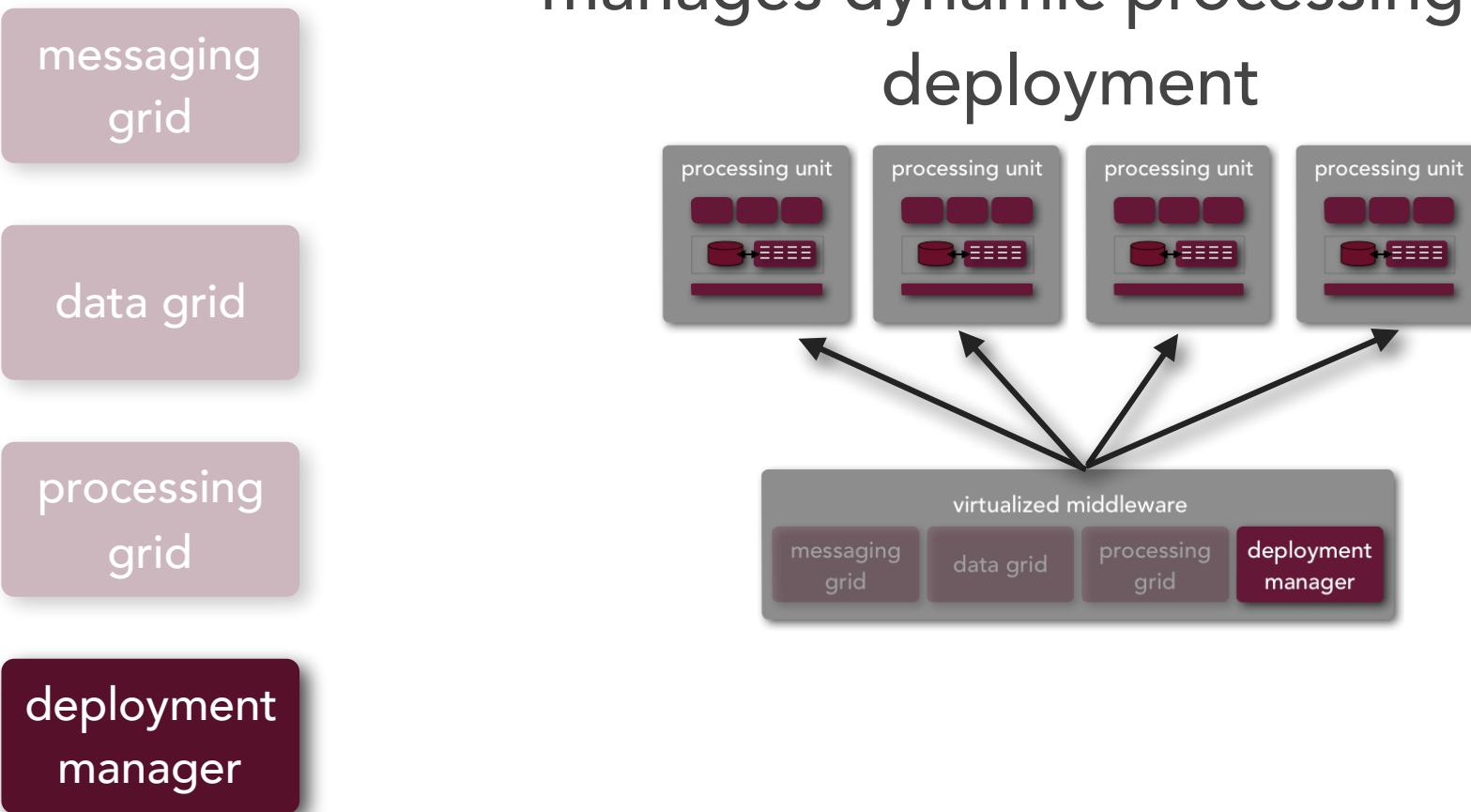
manages distributed
request processing



space-based architecture

middleware

manages dynamic processing unit deployment



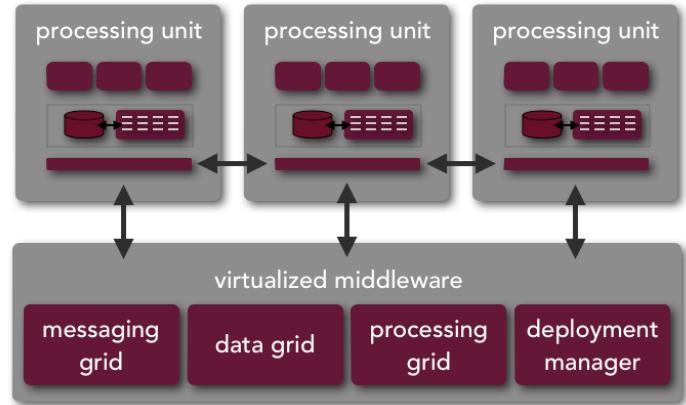
space-based architecture

it's all about variable scalability...

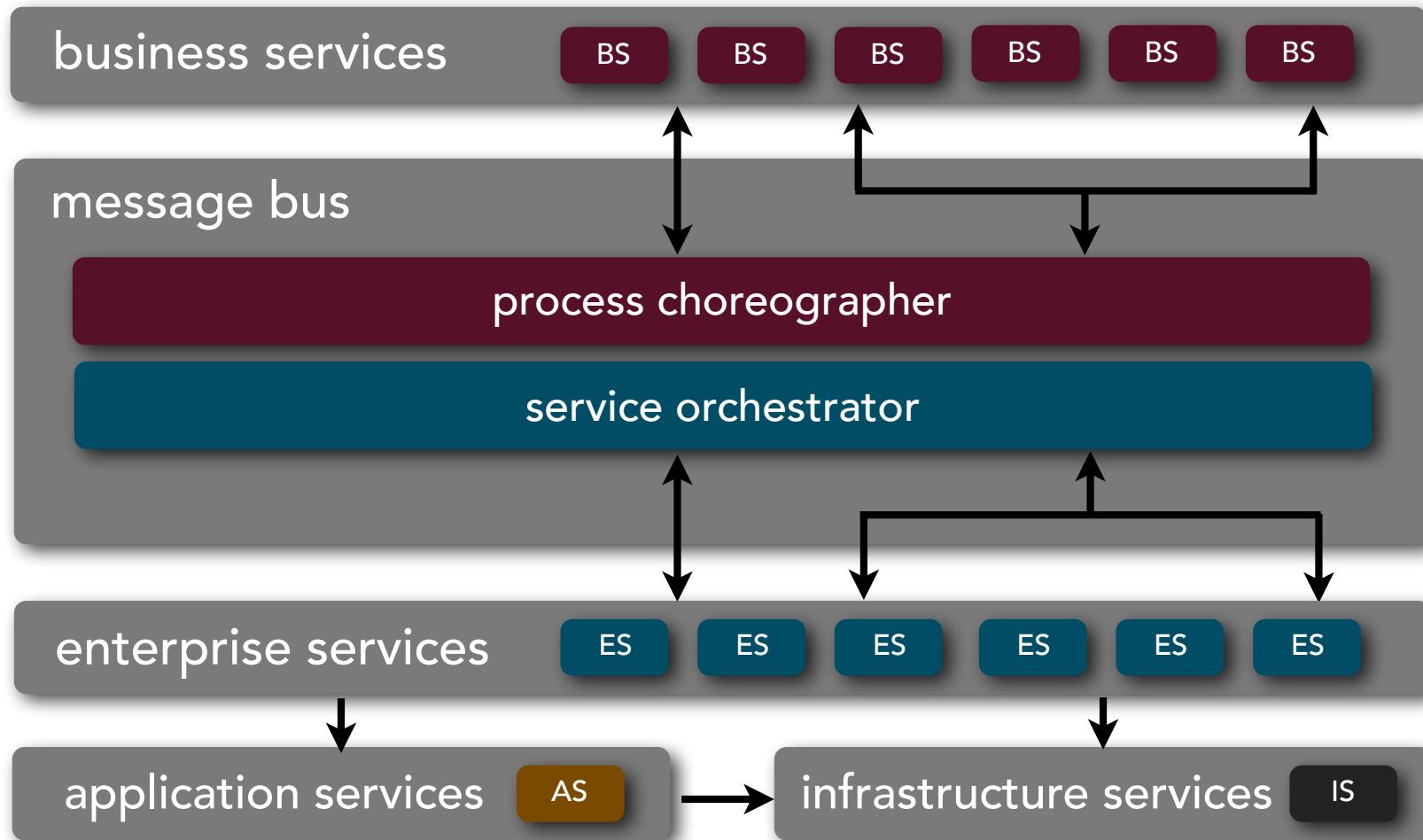
good for applications that have
variable load or inconsistent peak
times

not a good fit for traditional large-scale relational
database systems

relatively complex and expensive pattern to implement



service-oriented architecture



service-oriented architecture

business services

BS

BS

BS

BS

BS

BS

abstract enterprise-level coarse-grained services
owned and defined by business users

no implementation - only name, input, and output
data represented as wsdl, bpel, xml, etc.

ExecuteTrade

PlaceOrder

ProcessClaim

service-oriented architecture

concrete enterprise-level coarse-grained services
owned by shared services teams

custom or vendor implementations that are one-to-one
or one-to-many relationship with business services

enterprise services

ES

ES

ES

ES

ES

ES

CreateCustomer

CalcQuote

ValidateTrade

service-oriented architecture

concrete application-level fine-grained services
owned by application teams

bound to a specific application context

AddDriver

UpdateAddress

CalcSalesTax

application services

AS

service-oriented architecture

concrete enterprise-level fine-grained services owned by infrastructure or shared services teams

implements non-business functionality to support both enterprise and business services

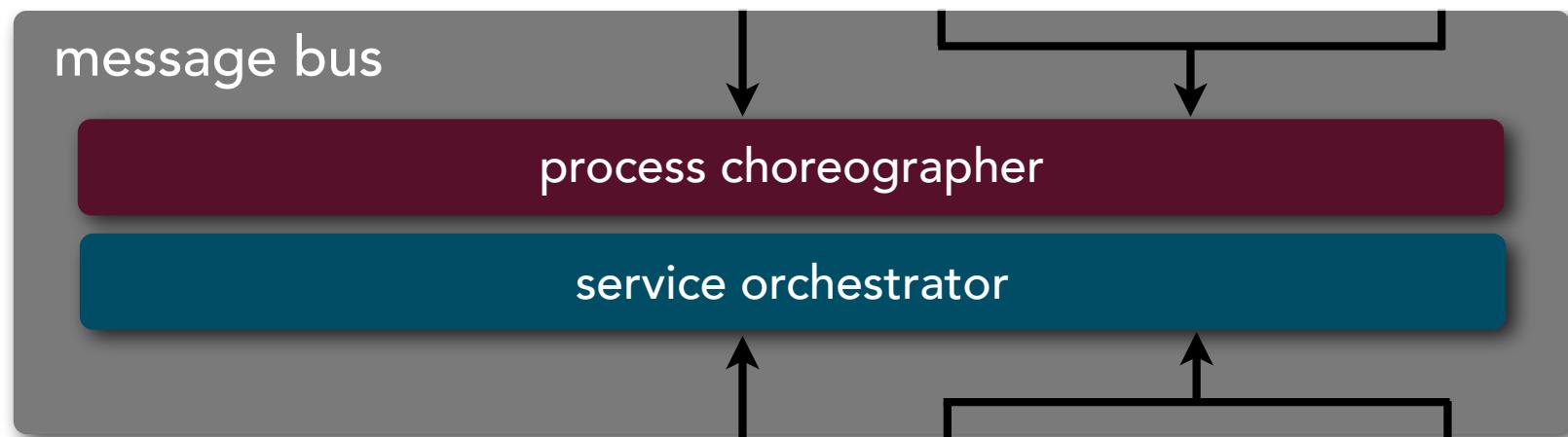
WriteAudit

CheckUserAccess

.LogError

infrastructure services IS

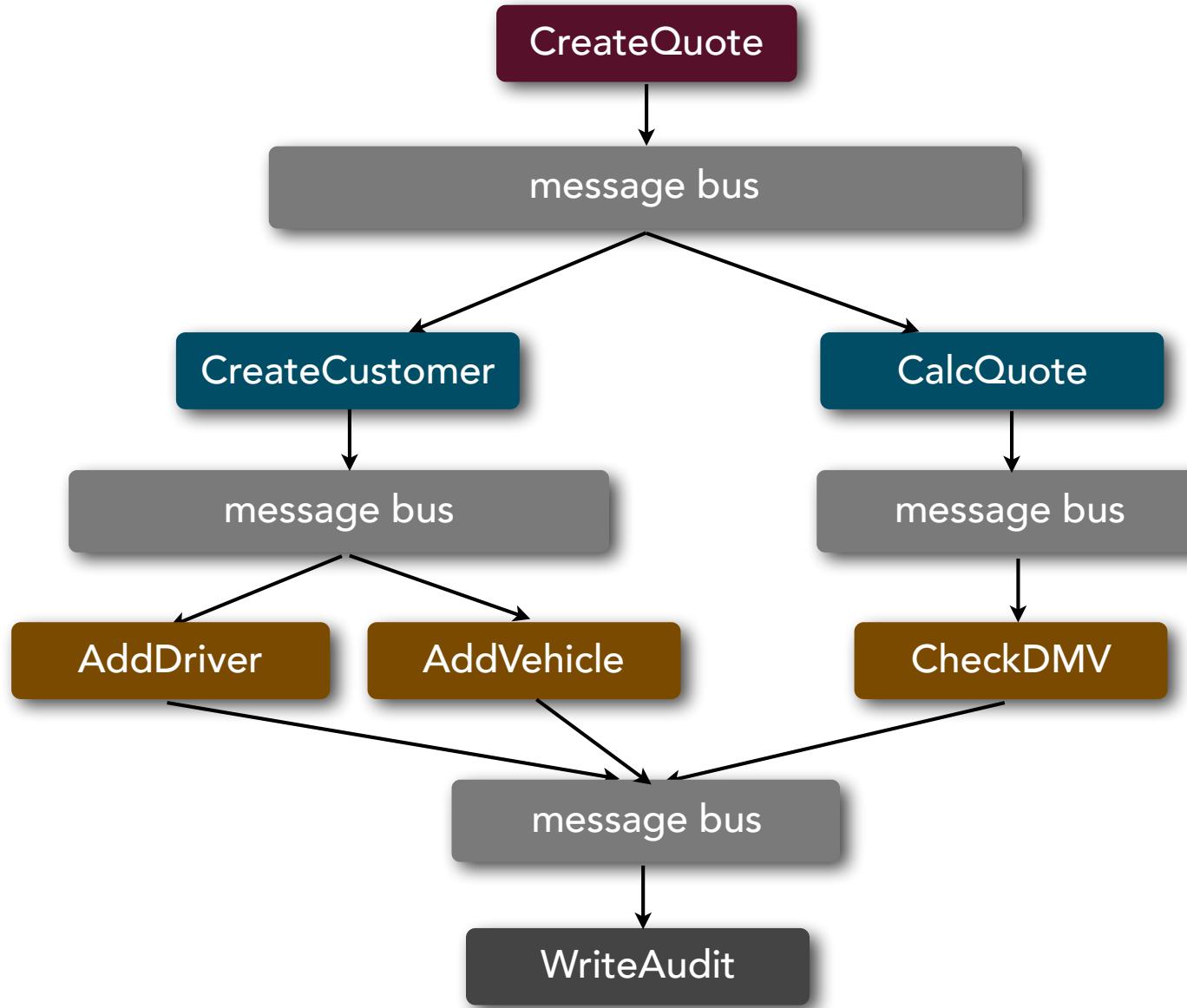
service-oriented architecture



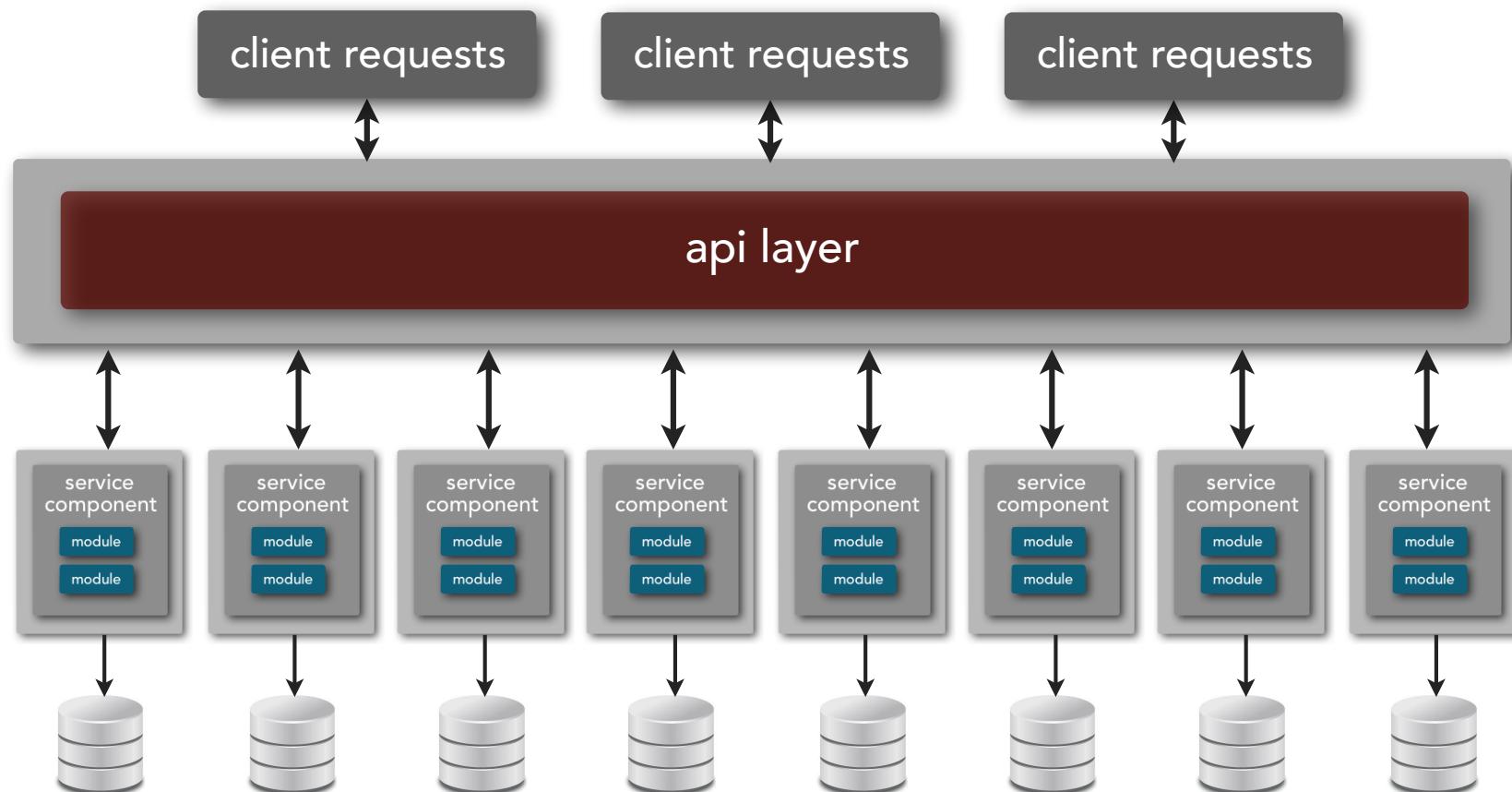
mediation and routing
process choreography
service orchestration

message enhancement
message transformation
protocol transformation

service-oriented architecture

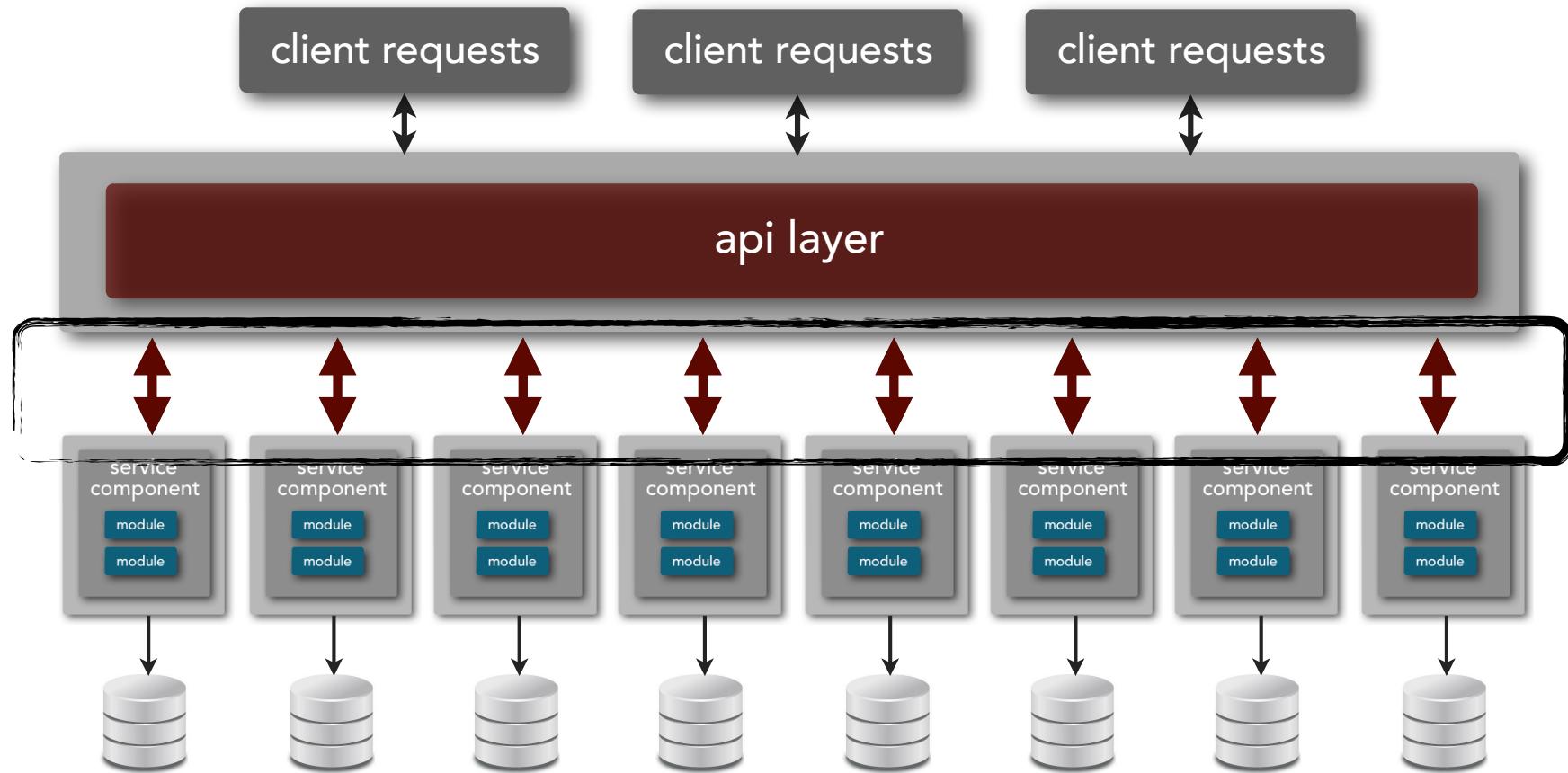


microservices architecture



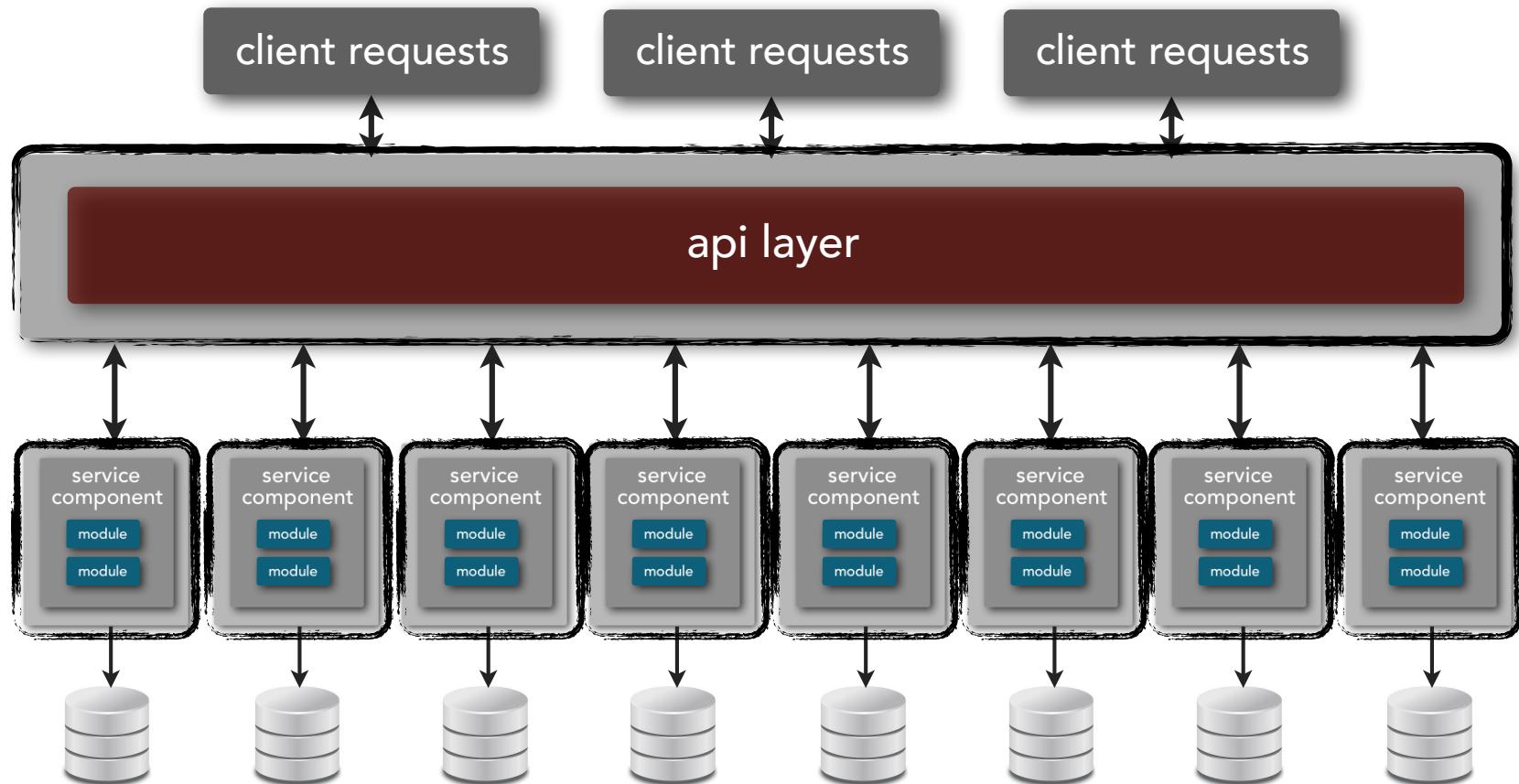
microservices architecture

distributed architecture



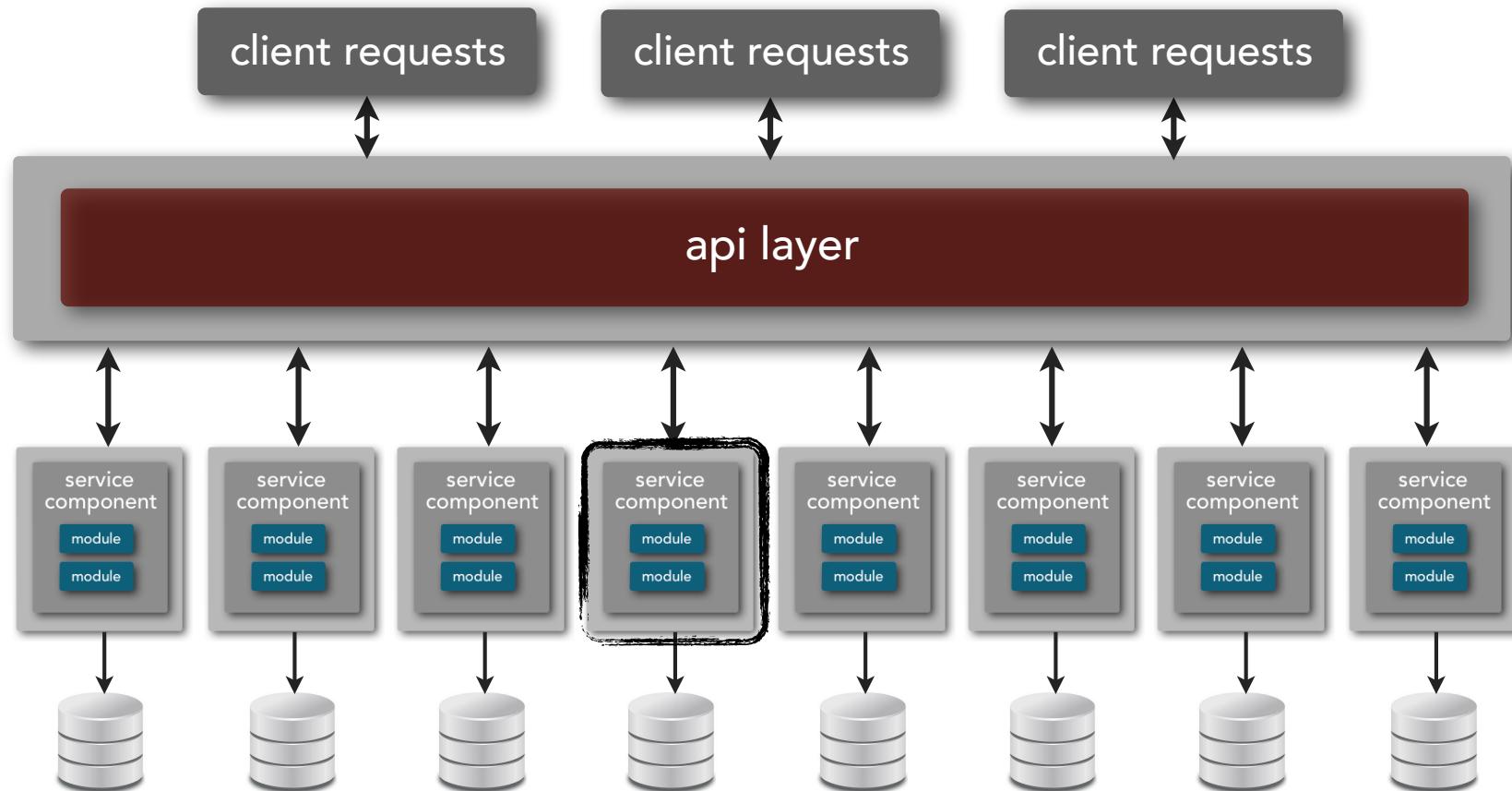
microservices architecture

separately deployed components



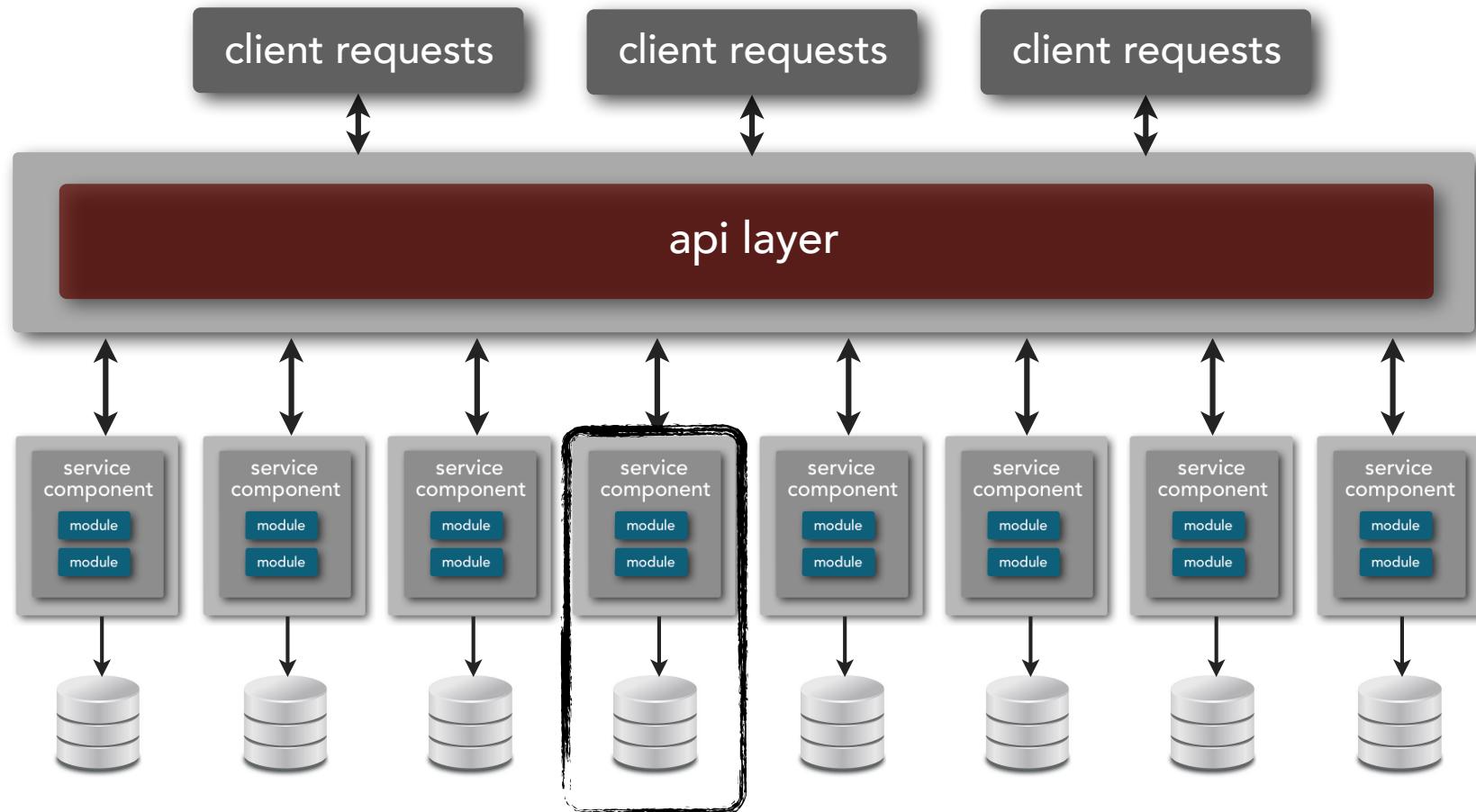
microservices architecture

service component



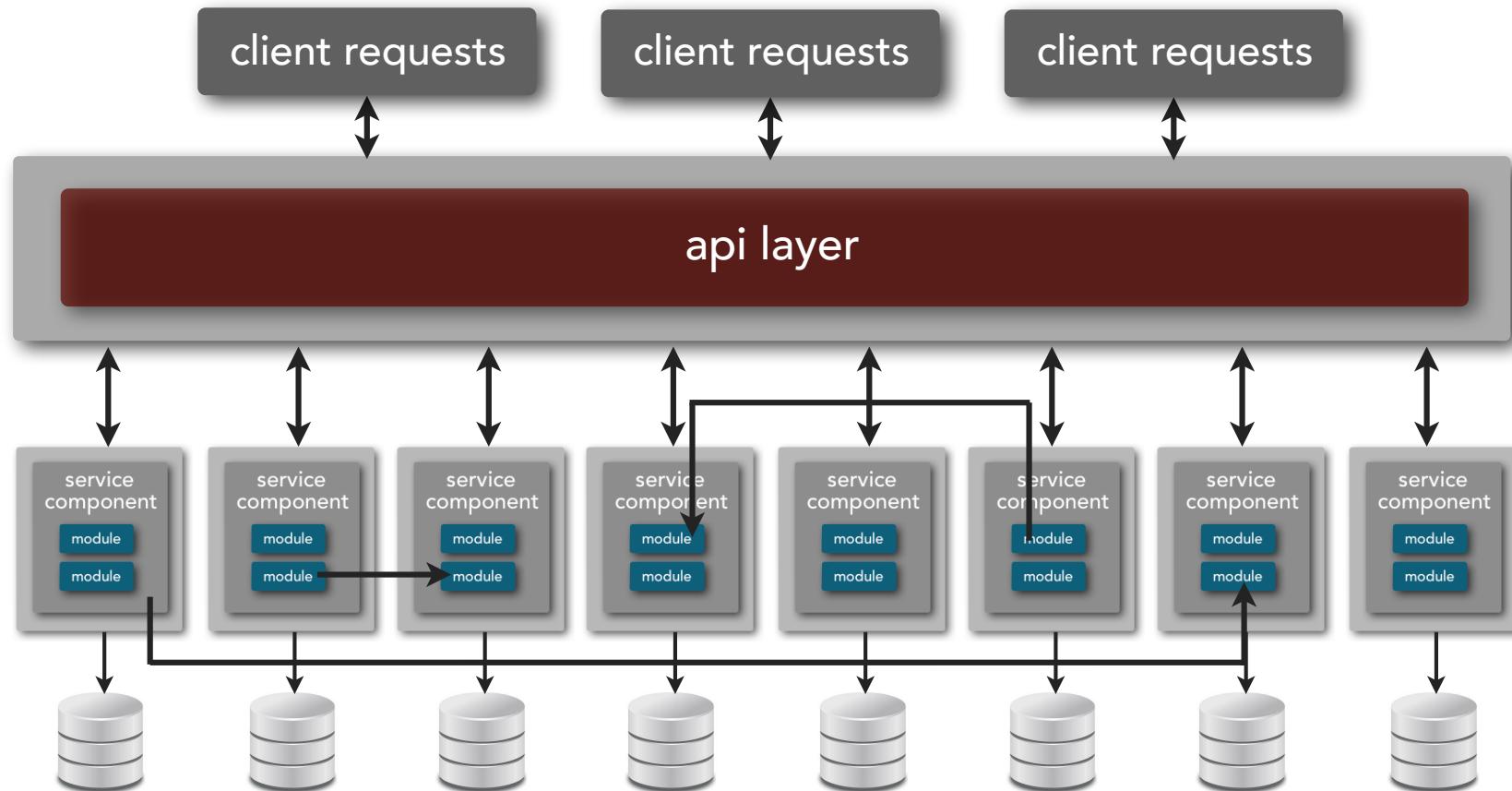
microservices architecture

bounded context



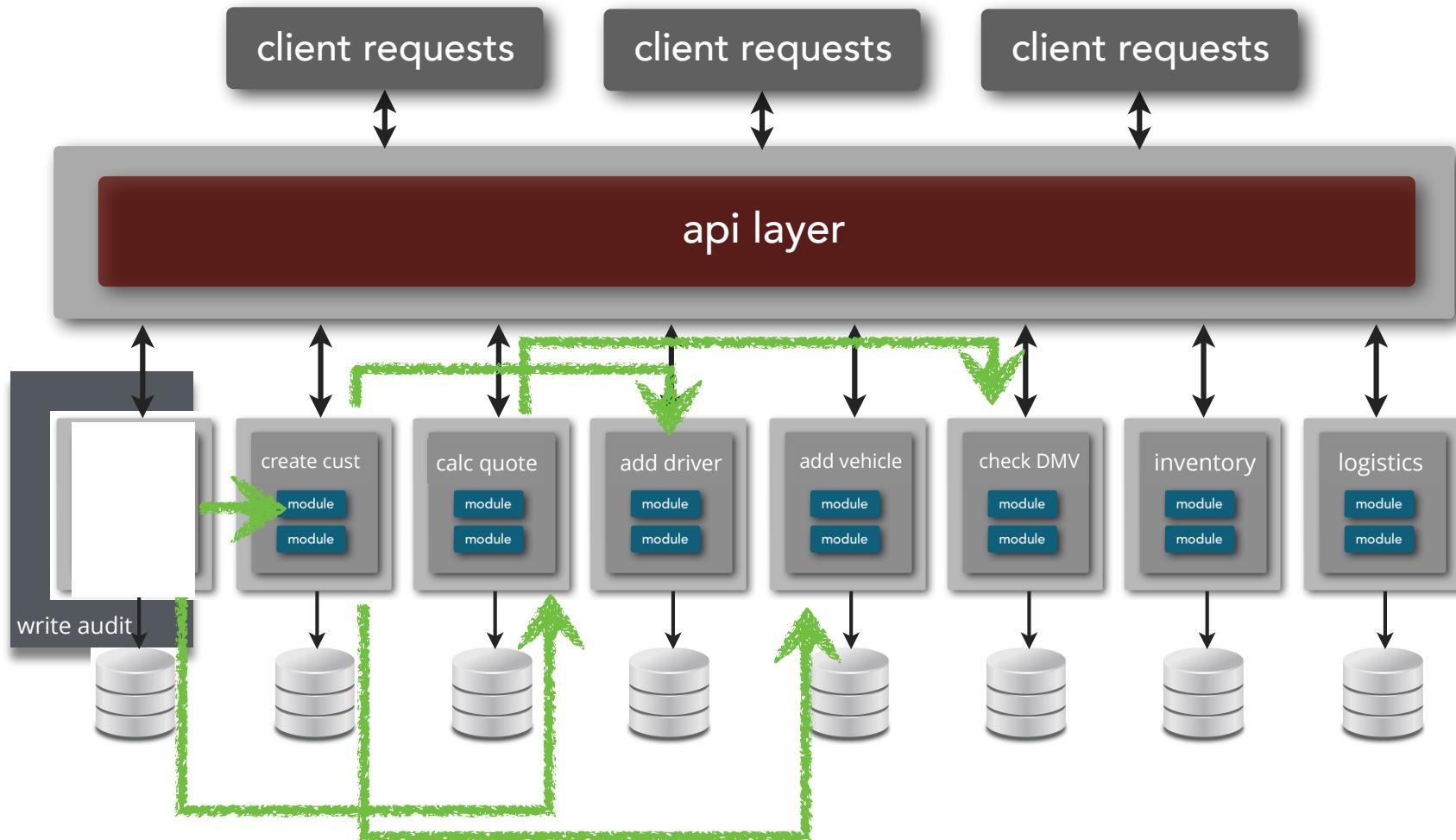
microservices architecture

service orchestration

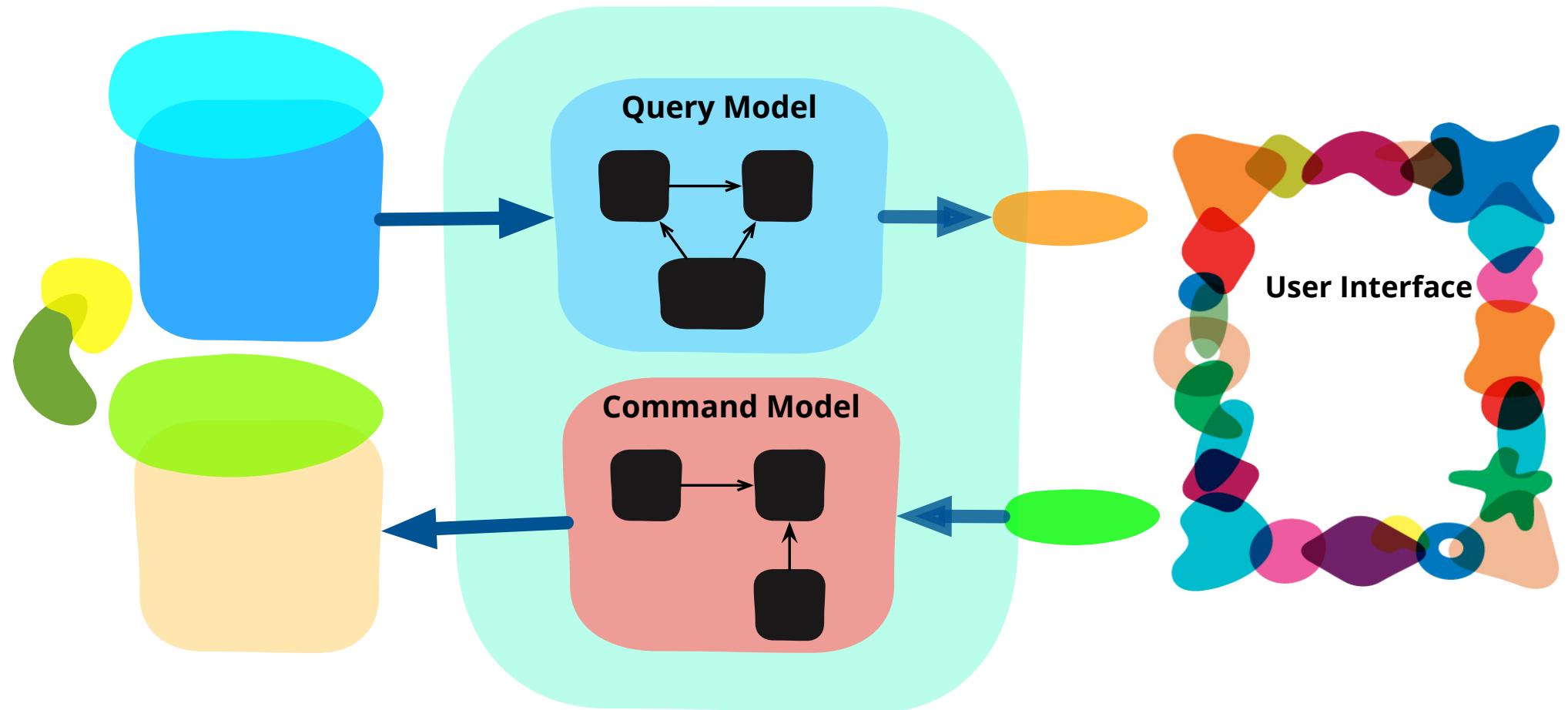


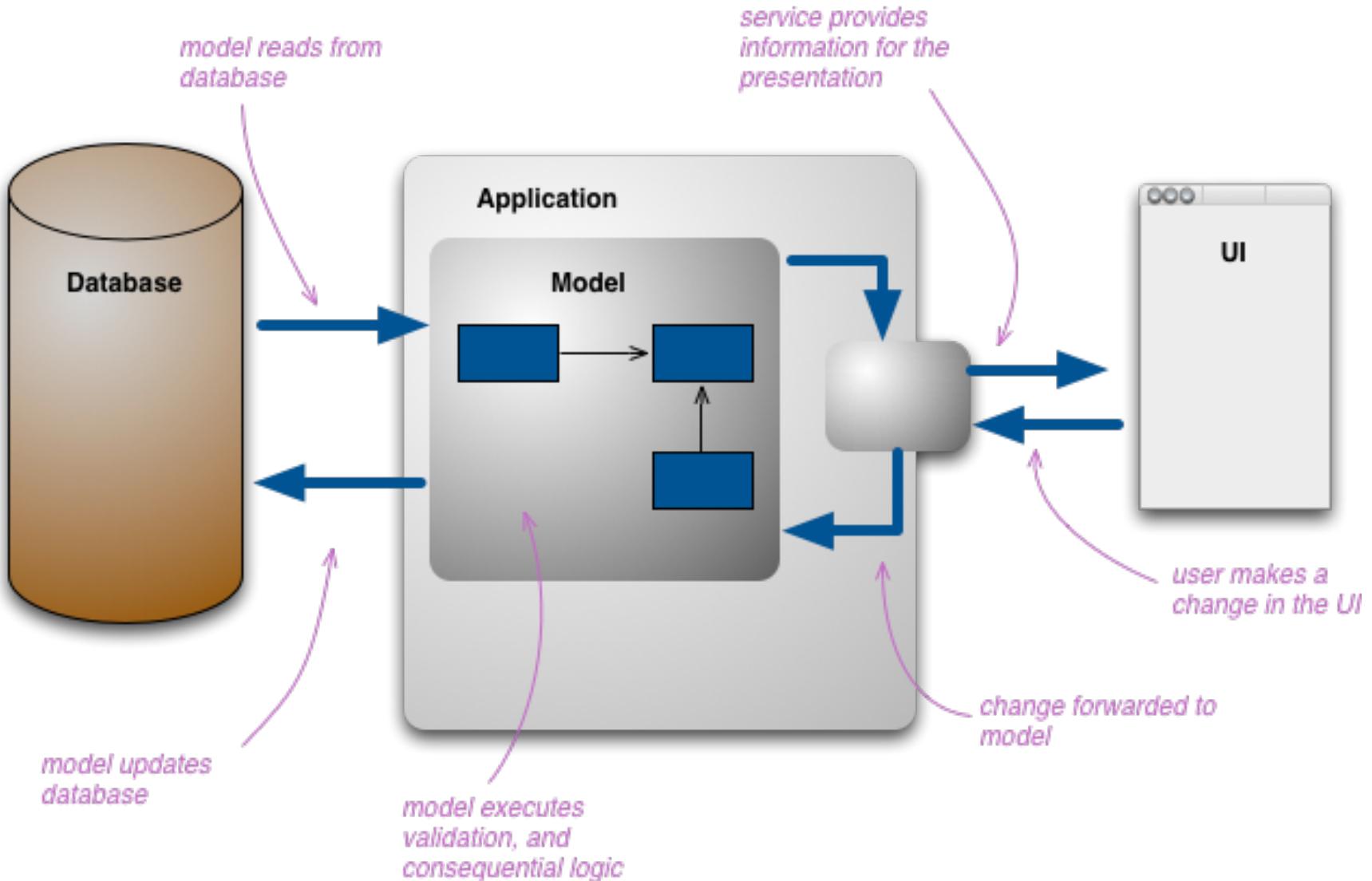
microservices architecture

service orchestration

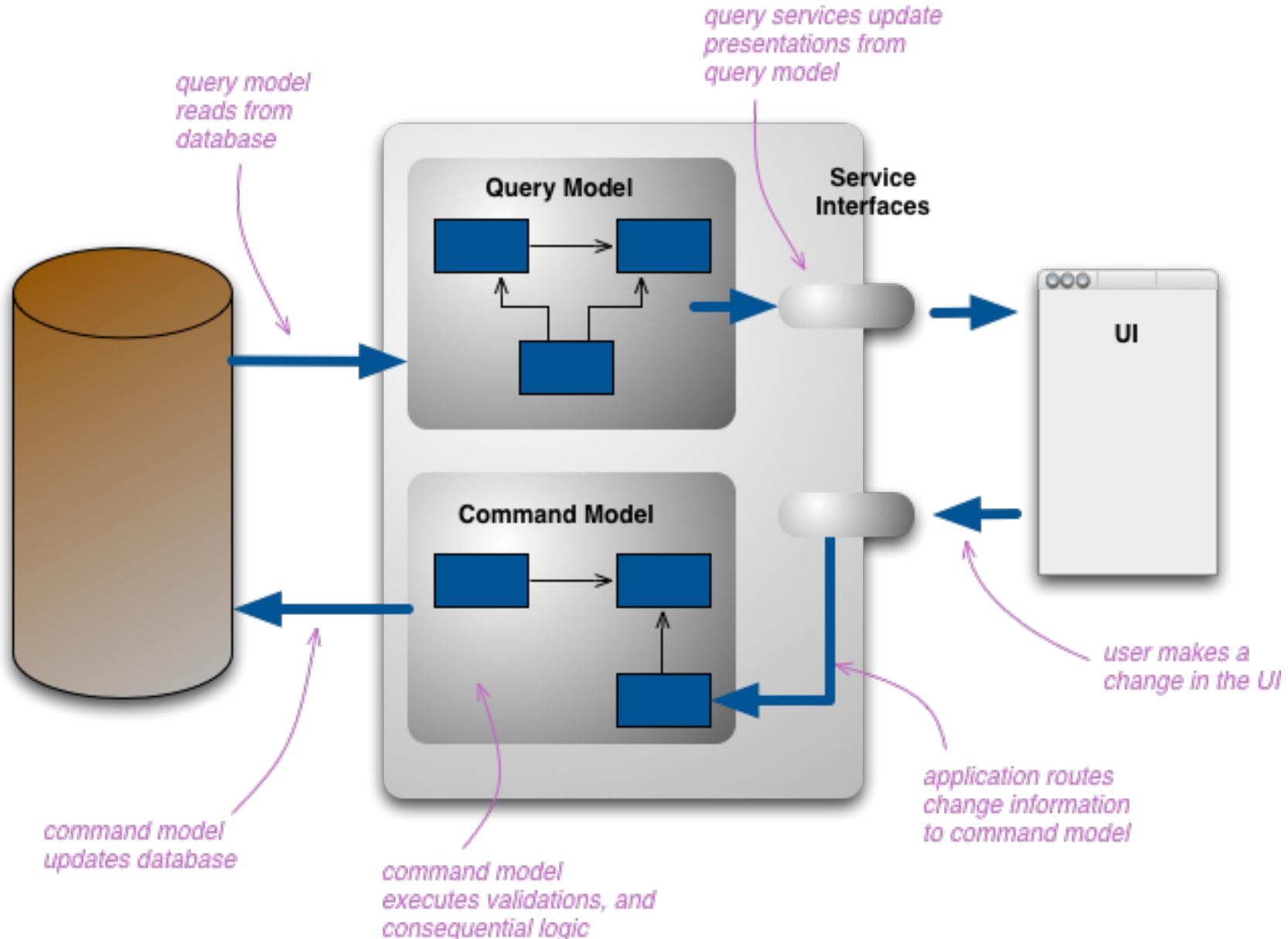


CQRS





traditional



CQRS Command Query Responsibility Separation

CQRS natural fits

task-based user interface

meshes well with event sourcing

eventual consistency

eventual consistency

Eventually Consistent – Revisited – All Things Distributed
www.allthingsdistributed.com/2008/12/eventually_consistent.html

All Things Distributed

Werner Vogels' weblog on building scalable and robust distributed systems.

Eventually Consistent - Revisited

By Werner Vogels on 22 December 2008 04:15 PM | [Permalink](#) | [Comments \(14\)](#)

I wrote a [first version of this posting](#) on consistency models about a year ago, but I was never happy with it as it was written in haste and the topic is important enough to receive a more thorough treatment. [ACM Queue](#) asked me to revise it for use in their magazine and I took the opportunity to improve the article. This is that new version.

Eventually Consistent - Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

At the foundation of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and need to be accounted for up front in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when the underlying distributed system provides an eventual consistency model for data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. In this article I present some of the relevant background that has informed our



Contact Info

Werner Vogels
CTO - Amazon.com
werner@allthingsdistributed.com

Other places

Follow werner on [twitter](#) if you want to know what he is current reading or thinking about.
At [wernerly](#) he posts material that doesn't belong on this blog or on twitter.

Syndication

Subscribe to this weblog's [atom feed](#) or [rss feed](#)

“Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.”

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

CQRS natural fits

task-based user interface

meshes well with event sourcing

eventual consistency

consistency or availability
(but never both)

complex or granular domains



LMAX

LMAX

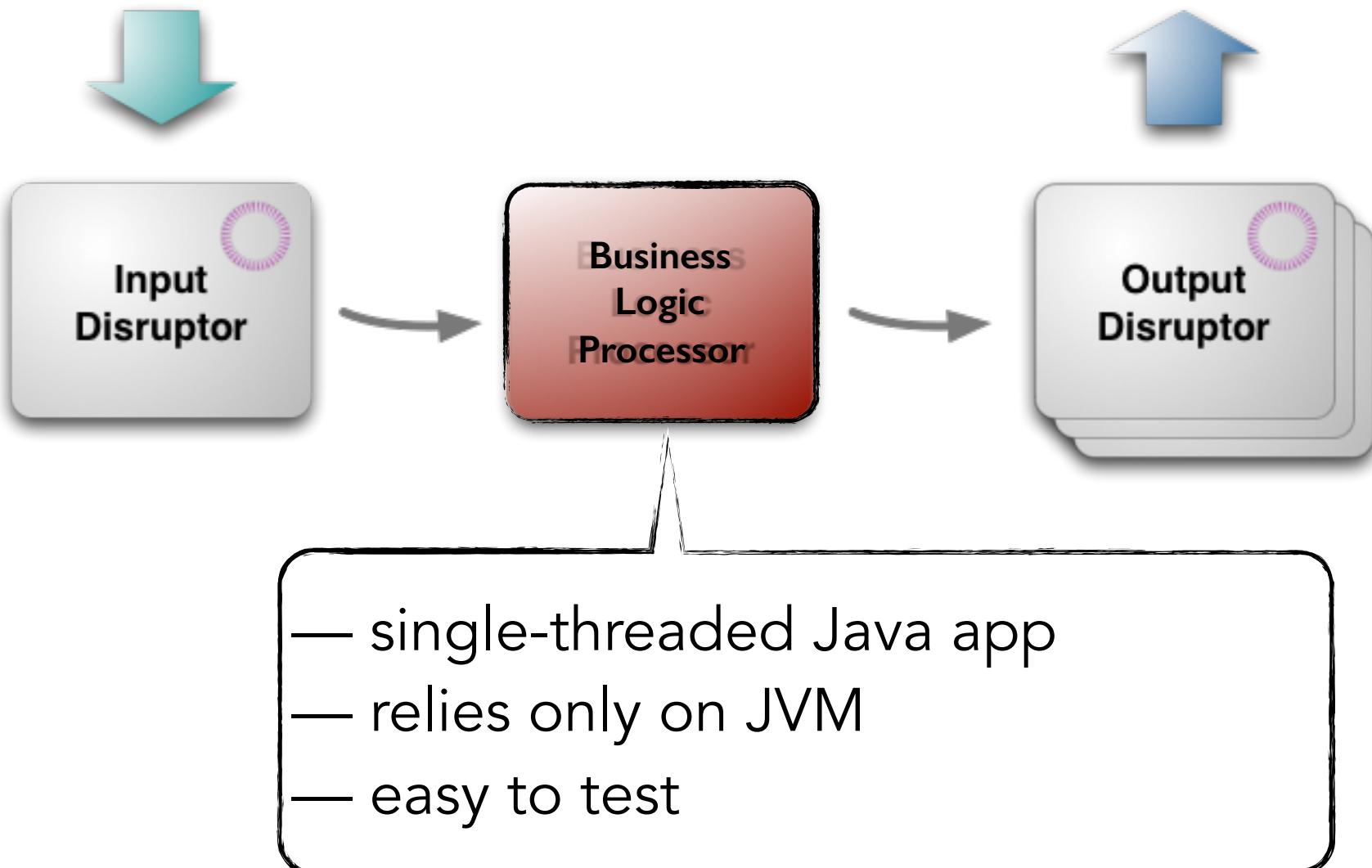
<http://martinfowler.com/articles/lmax.html>

JVM-based retail financial trading
platform

centers on Business Logic Processor
handling 6,000,000 orders/sec on 1
thread

surrounded by Disruptors, network of
lock-less queues

overall structure



Business
Logic
Processor

business logic processor

in-memory

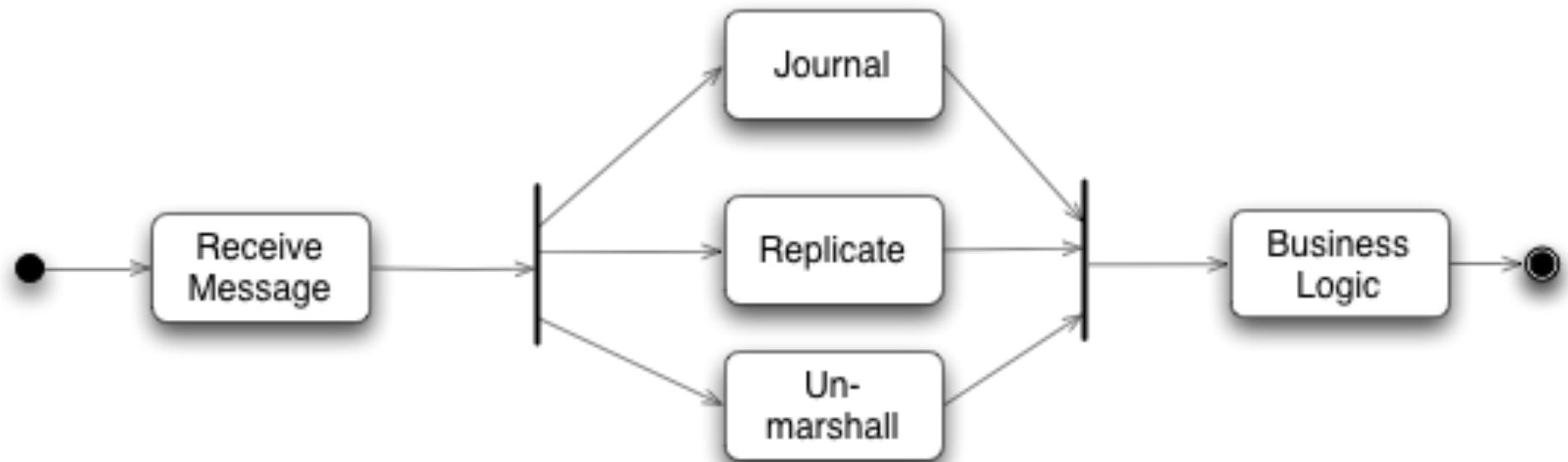
event sourcing via input disruptor

snapshots (full restart—JVM + snapshots
— less than 1 min)

multiple instances running

each event processed by multiple
processors but only one result used

input/output disruptors



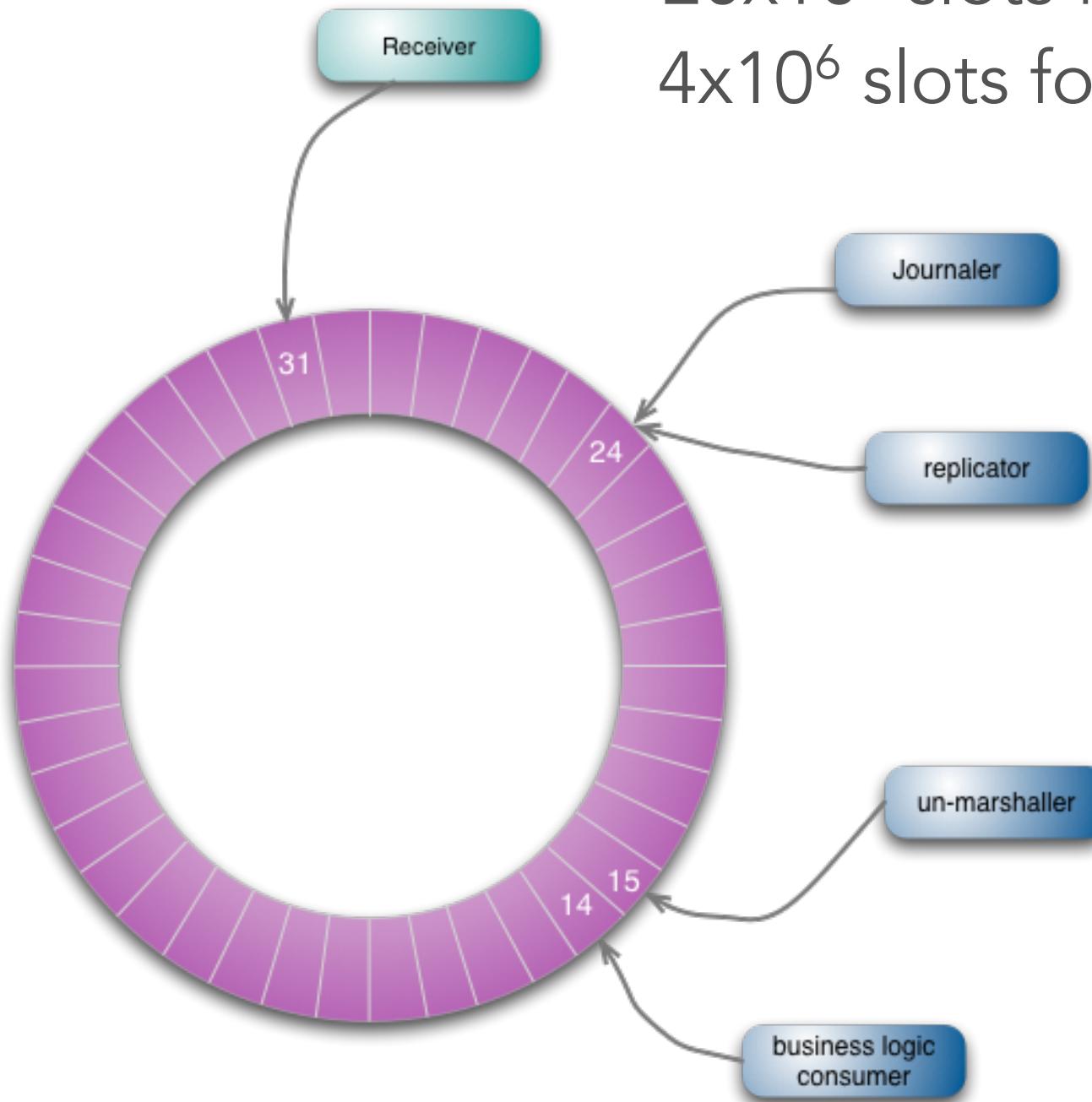
disruptors

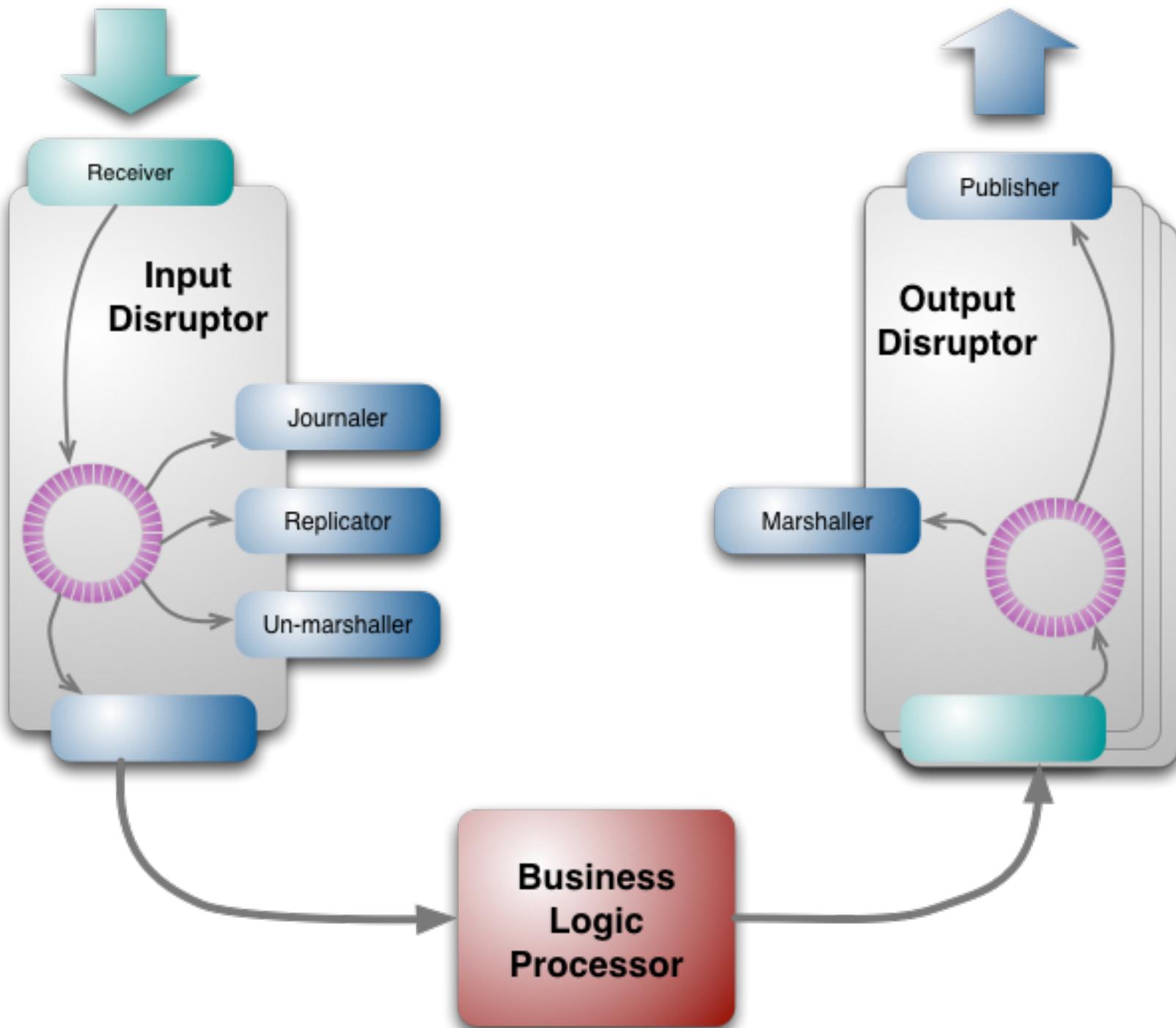
custom concurrency component

multi-cast graph of queues where
producers enqueue objects and
consumers dequeue in parallel

ring buffer with sequence counters

20×10^6 slots for input buffer
 4×10^6 slots for output buffer





“mechanical sympathy”

started with transactions

switched to Actor-based concurrency

hypothesized & measured results

CPU caching is key \rightarrow single writer principle

IT
DEPENDS !

“One ring to rule them all...”

no architecture fits every circumstance

evaluate good, bad, and ugly

watch for primrose paths that turn ugly

embrace pragmatism

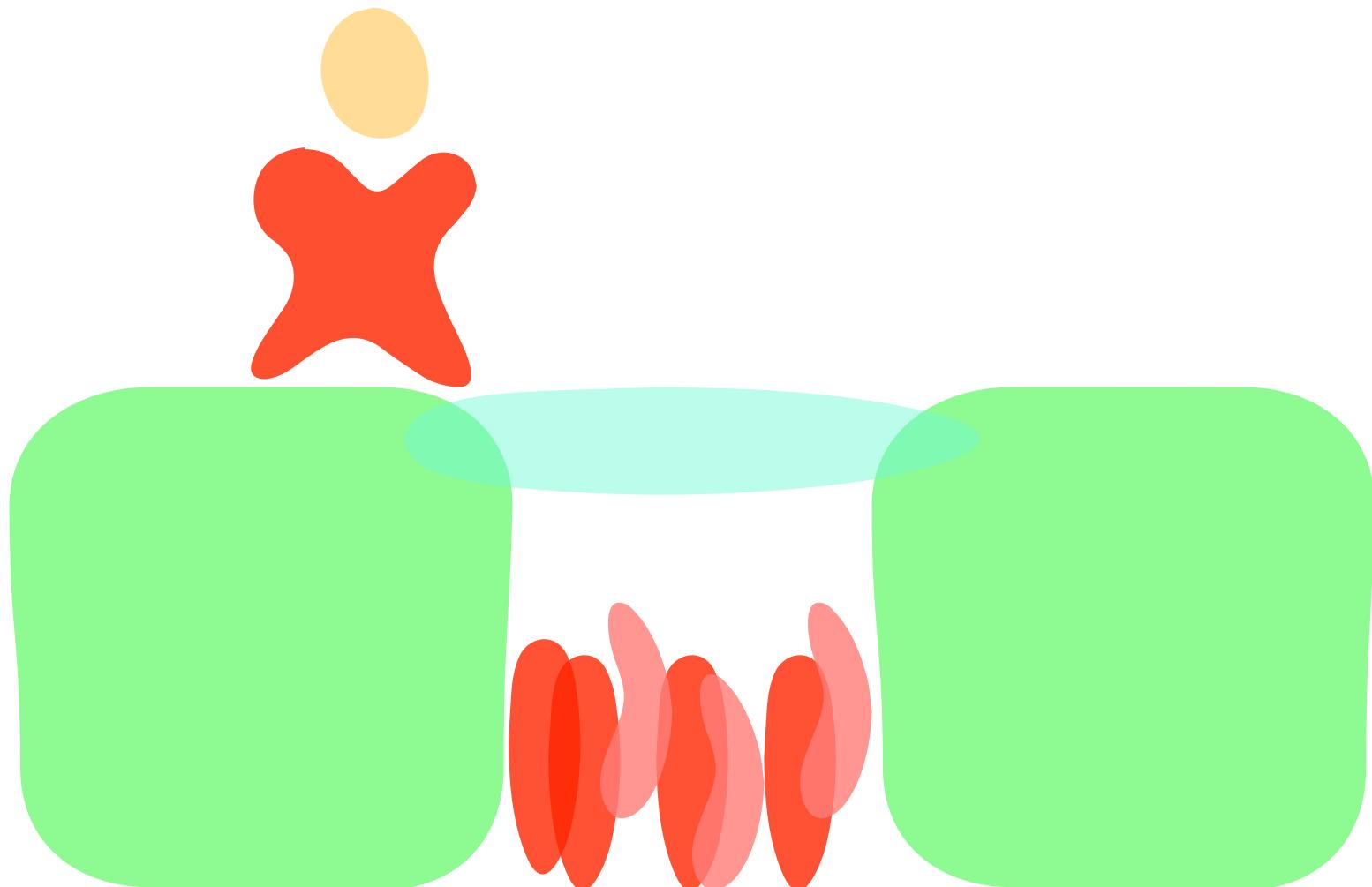
what's your day job?

building wicked cool architectures is an
stimulating, rewarding intellectual
challenge...

...building CRUD applications isn't

coolness sometimes equals accidental
complexity

architecture pitfall

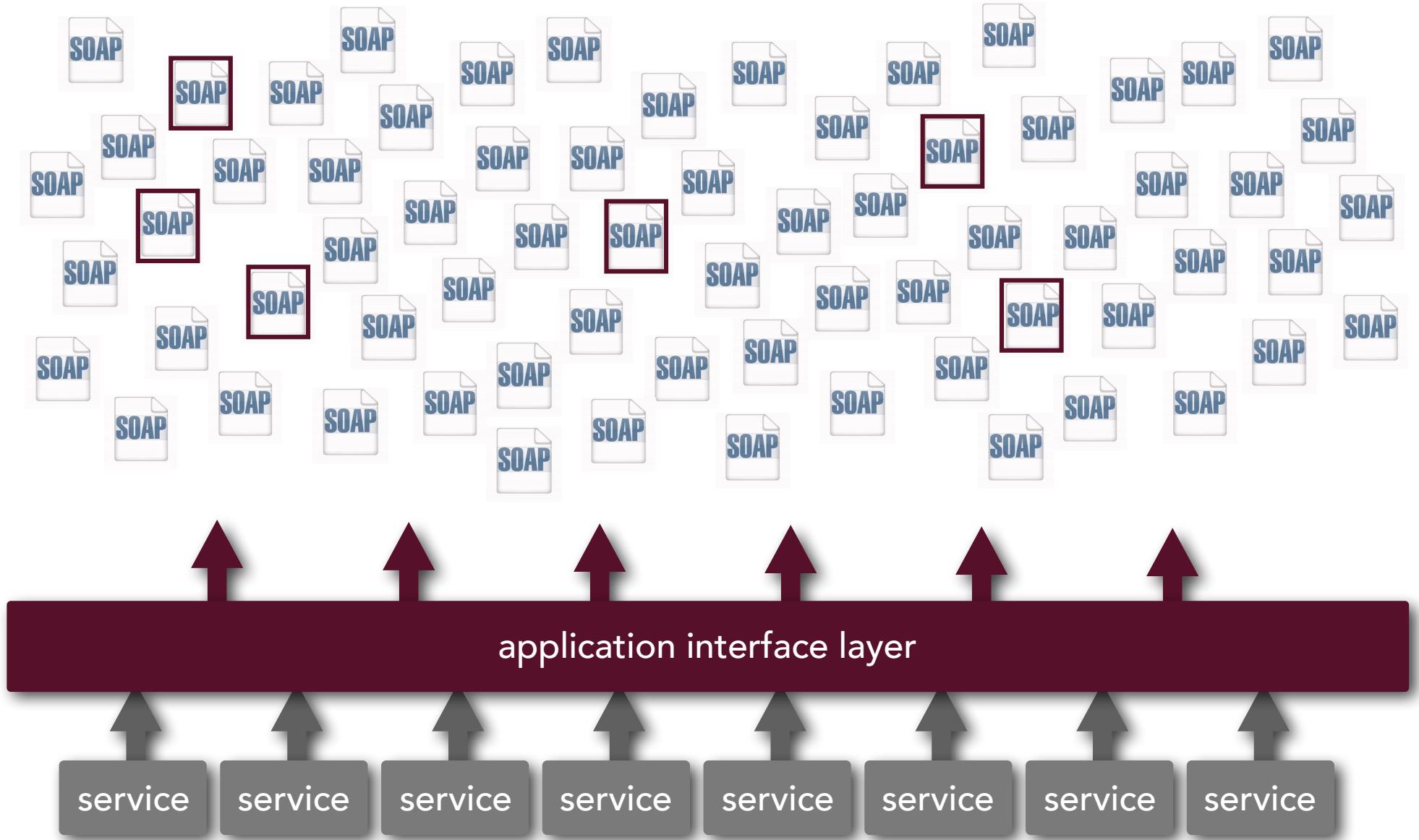


spider web architecture

creating large numbers of web services that
are never used just because you can



spider web architecture

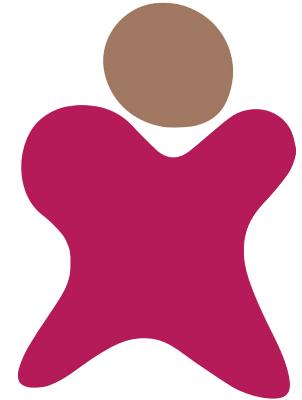


spider web architecture

just because you can create a web service at the click of a button doesn't mean you should!

let the requirements and business needs drive what services should be exposed





Meeting Hacks



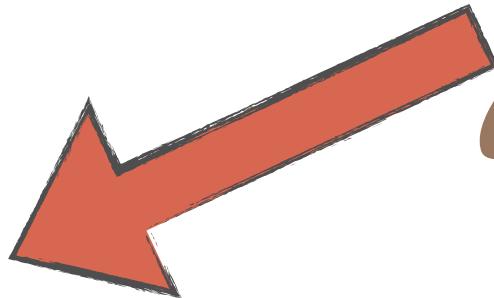
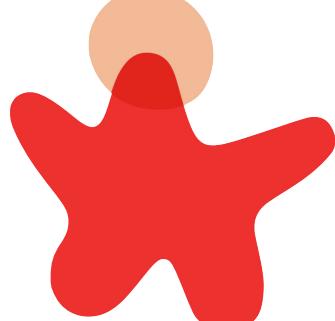
makers vs. managers

The screenshot shows a web browser window with the URL paulgraham.com in the address bar. The page title is "PAUL GRAHAM". On the left, there's a sidebar menu with links like Home, Essays, H&P, Books, YC, School, Arc, Lisp, Spam, Responses, FAQs, RAQs, Quotes, RSS, Bio, Twitter, Search, and Index. The main content area has a header "MAKER'S SCHEDULE, MANAGER'S SCHEDULE" and a call-to-action button "Want to start a startup? Get funded by [Y Combinator](#)". Below that, it says "July 2009". The text discusses the difference between the manager's schedule (a strict hourly appointment book) and the maker's schedule (preferring longer, less frequent work periods). It notes that powerful people often fit into the manager's schedule, which can be problematic for makers.

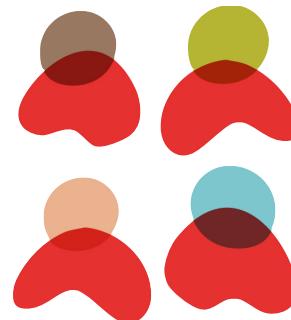
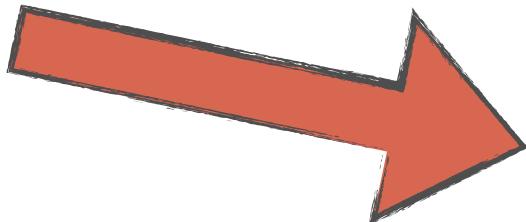
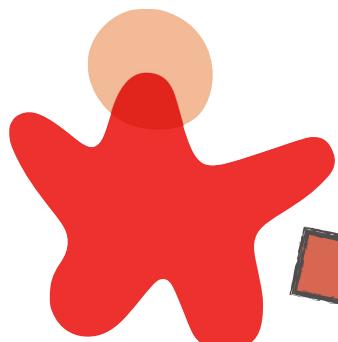
Since most powerful people operate on the manager's schedule, they're in a position to make everyone resonate at their frequency if they want to. But the smarter ones restrain themselves, if they know that some of the people working for them need long chunks of time to work in.

www.paulgraham.com/makersschedule.html

imposed upon



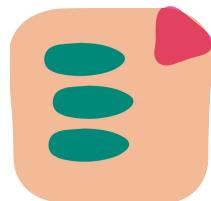
meetings...



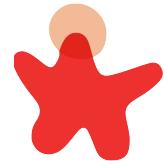
imposed by



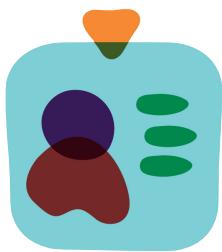
imposed meetings



stick to the agenda



take one (or many) for the team

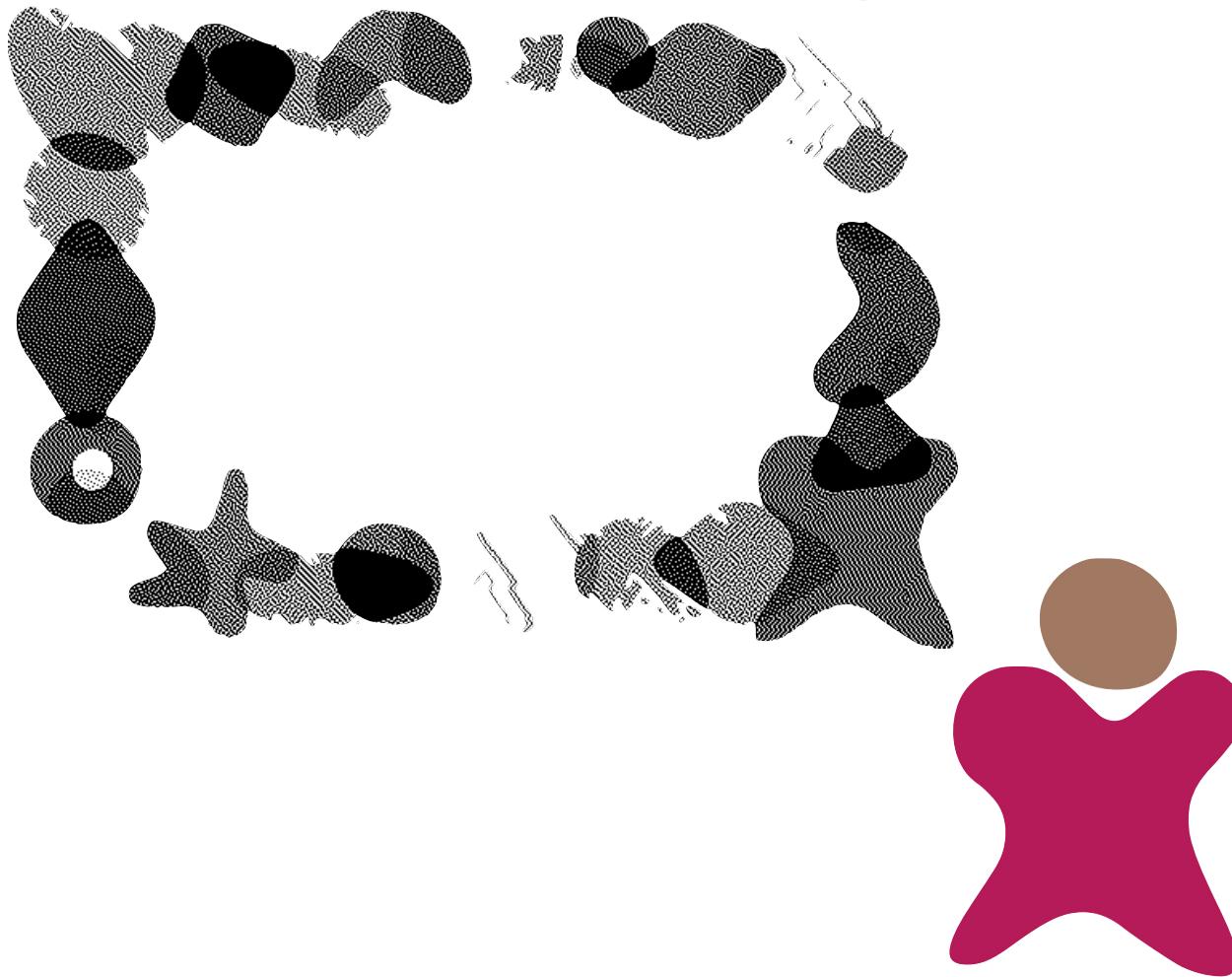


know the dramatis personæ



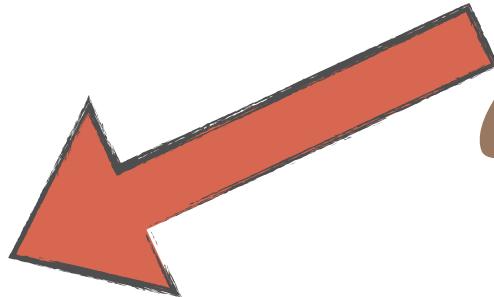
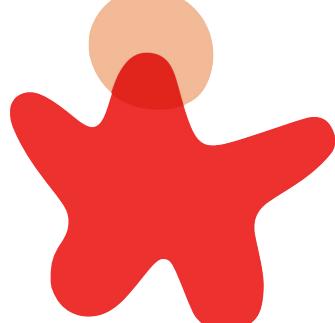
don't ask important questions
you don't already know the answer to

meeting imagery

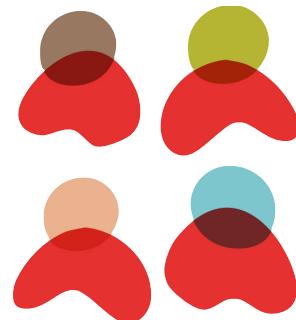
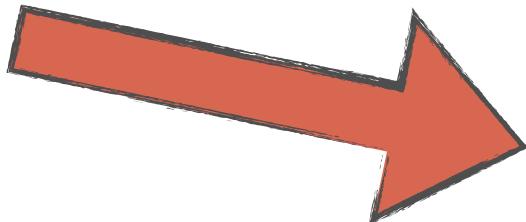
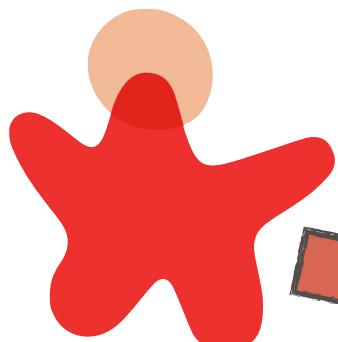


don't improvise white board
drawings in important meetings

imposed upon

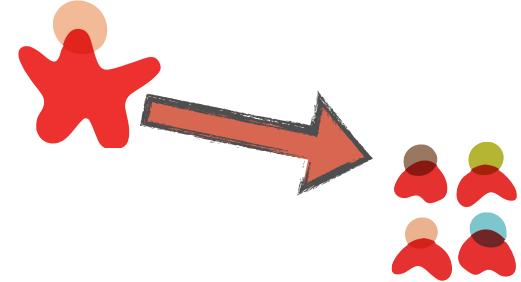


meetings...

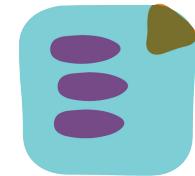


imposed by

imposed by



is this more important than the work
you are pulling people away from?



set an agenda



have someone take lightweight notes



specialized meetings



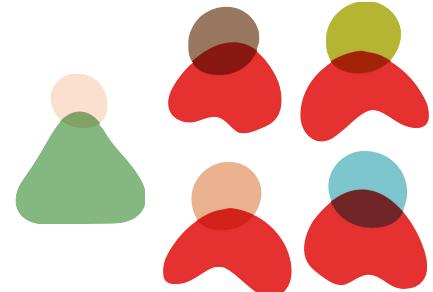
stand-up

<role>-huddle

BA-huddle

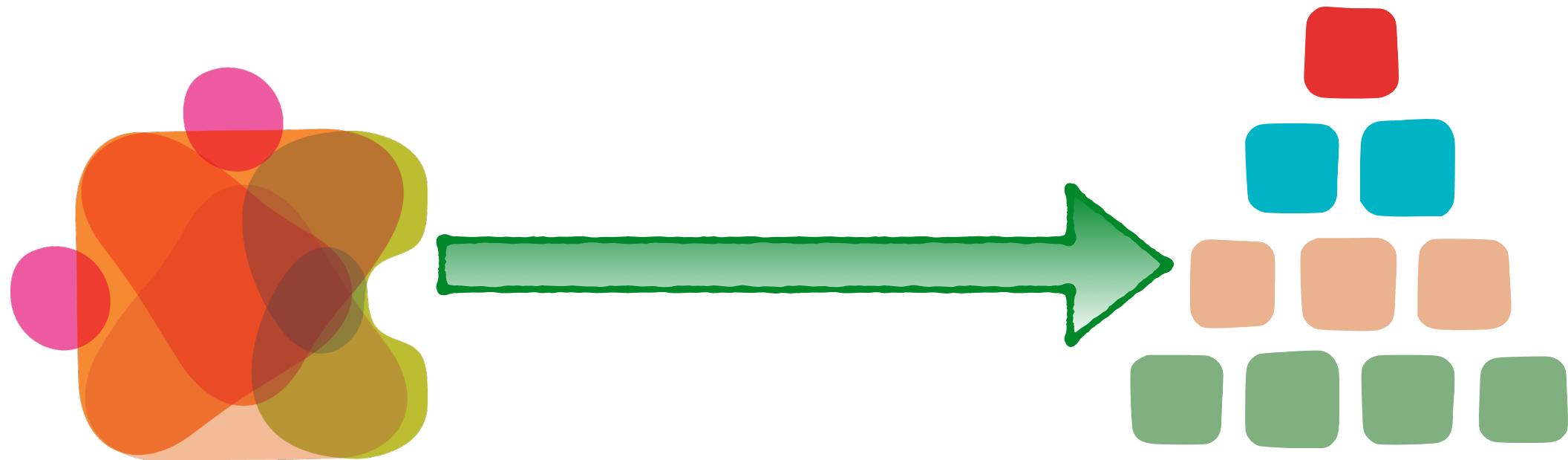


dev-huddle

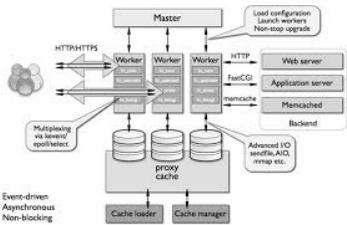


- keep it short
- keep it relevant
- everyone
- too many?
- geographically dispersed?
- police inappropriate levels of interaction

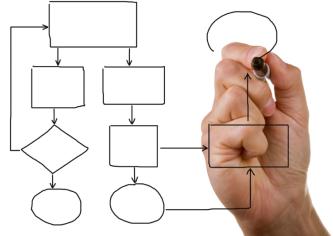
Architecture Refactoring Techniques



architecture refactoring



determine the architectural root cause of each issue



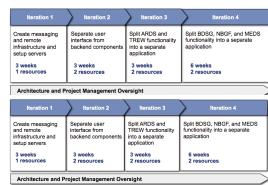
determine what architecture changes are needed



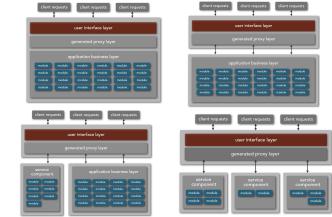
create a business case justifying the changes



present your case and plan to the business for approval and funding

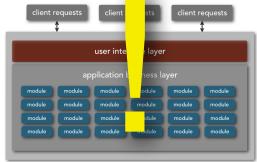


create a timeline containing estimates and resources

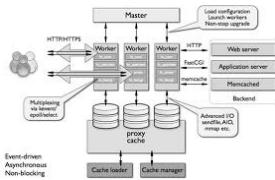


develop a high-level architecture refactoring plan

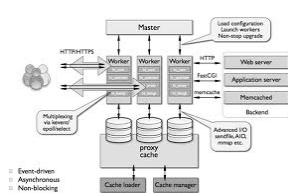
determine root cause



application builds fail almost every time something is committed to the repository

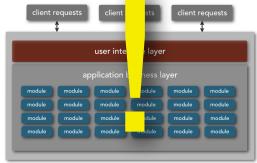


application components are too tightly coupled and dependent on one another

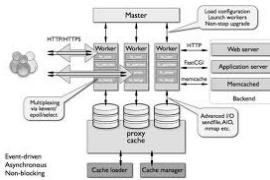


application components do not have the right level of granularity and isolation from a roles and responsibility standpoint

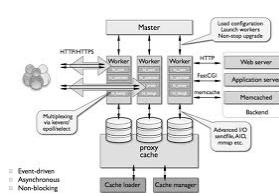
determine root cause



every time the application is deployed with new functionality, something else usually breaks

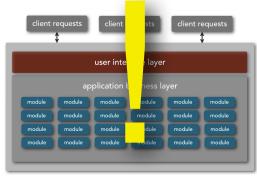


application components are too tightly coupled and dependent on one another

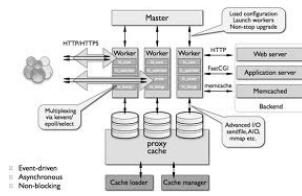


application components do not have the right level of granularity and isolation from a roles and responsibility standpoint

determine root cause

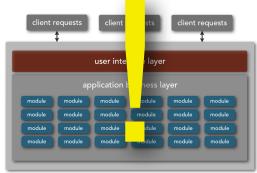


application deployments take a long time,
sometimes lasting up to 25 minutes

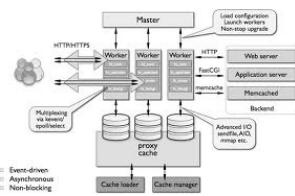


the application is a monolithic and growing quickly
based on new features and functionality; it is getting
too large for a single application.

determine root cause



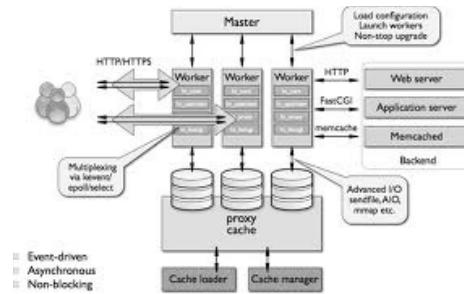
users are increasingly reporting performance issues with the application



the application is becoming too large; it is consuming about 95% of the available jvm resources during normal load (memory, cpu, threads)

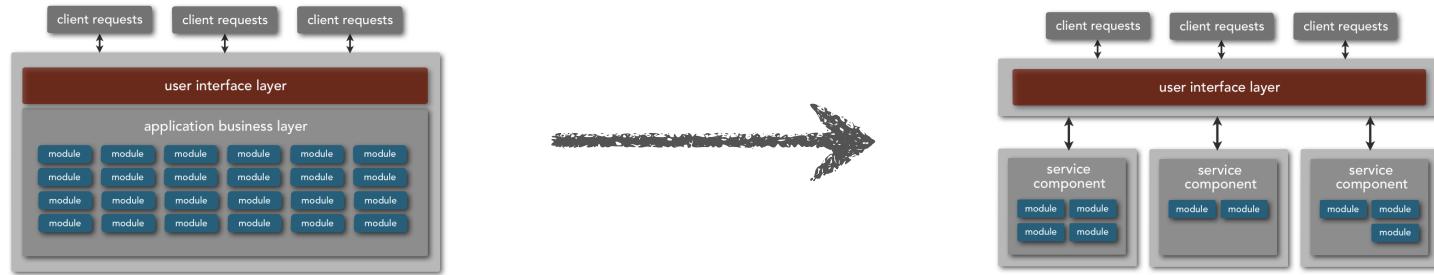
determine root cause

root cause summary



the application is too tightly coupled and
is growing beyond what the current
architecture can support

determine architecture changes

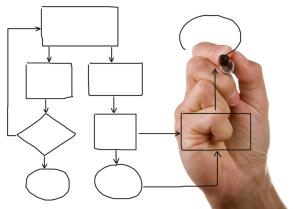


the monolithic application is too tightly coupled and is growing beyond what the current architecture can support

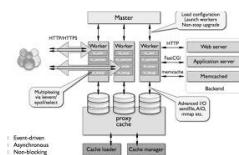
split the application into multiple deployable units, thereby decoupling components and allowing for more growth potential

justifying your case

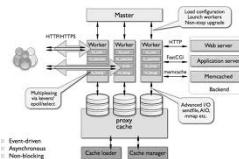
technical justification



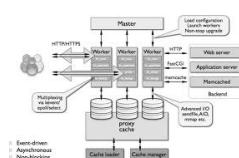
split the monolithic application into multiple deployable units, thereby decoupling components and allowing for more growth potential



components will be more decoupled, thereby eliminating frequent build issues



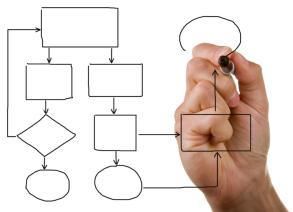
each part will use fewer jvm resources, thereby increasing performance and allow for more growth



deployment is limited to a separate application unit, thereby reducing deployment time and increasing robustness

justifying your case

business justification



split the monolithic application into multiple deployable units, thereby decoupling components and allowing for more growth potential



new functionality can be delivered faster, thereby improving overall time to market

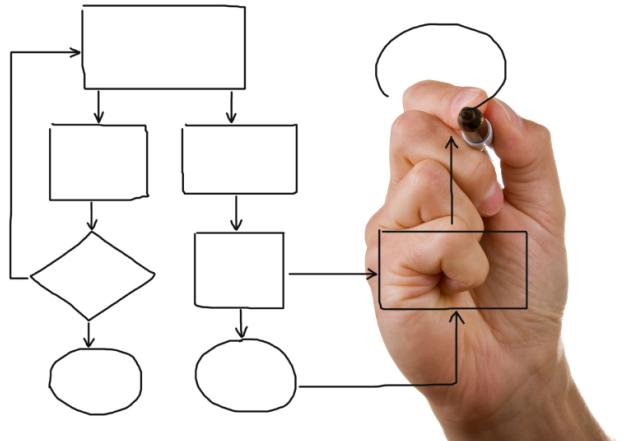


overall application quality will be improved, thereby reducing bugs and the associated costs of fixing them



development and deployment costs associated with developing new functionality will be significantly reduced

refactoring techniques



work in small iterations

identify the technical and business value expected at each iteration

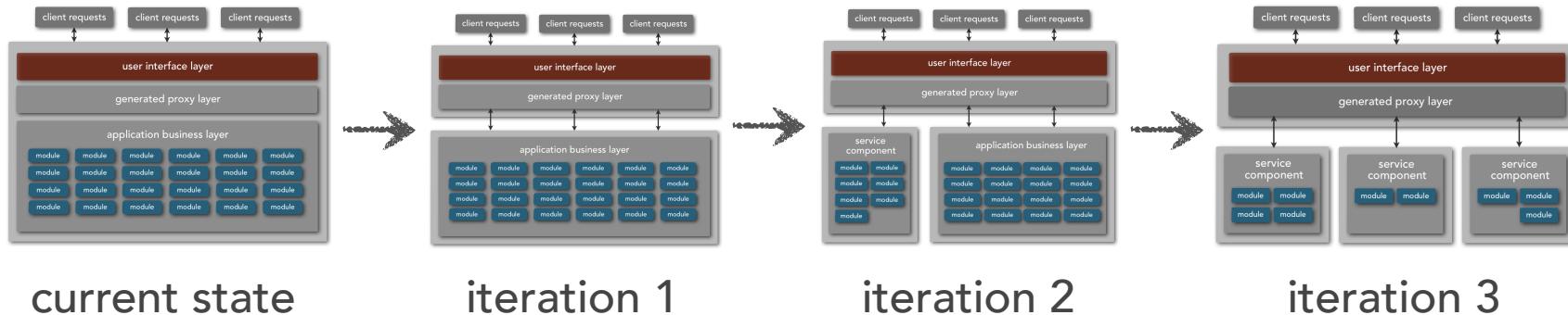
use a playbook approach to outline the architecture transformations

decide on migration vs. adaptation (or a combination of both)

refactoring techniques

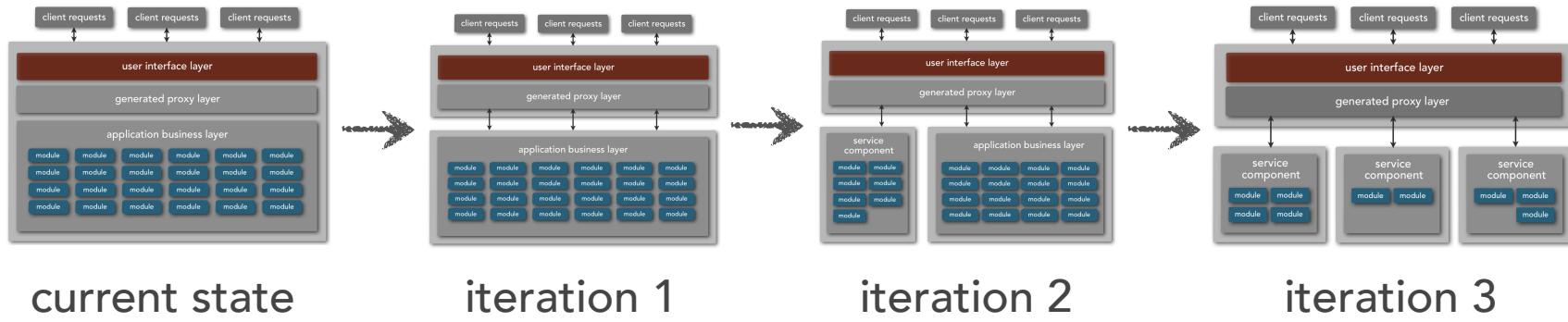
playbook approach

each iteration should clearly illustrate the changes to the architecture each step along the way



refactoring techniques

playbook approach



identify the purpose behind each iteration

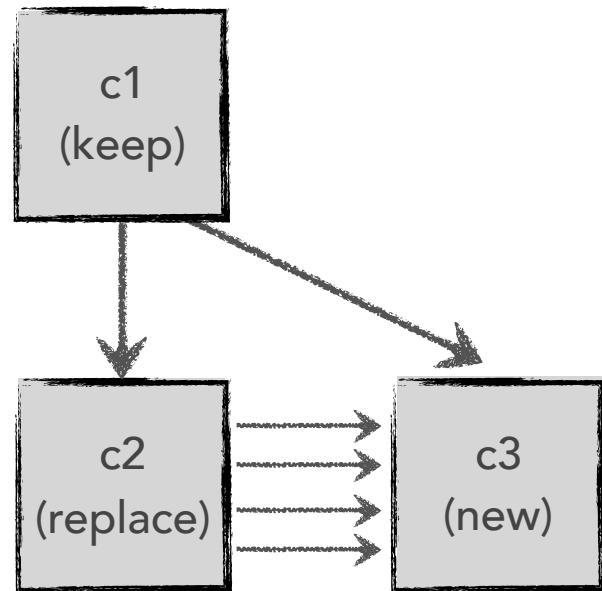
identify the technical and business value for each iteration

try to minimize "staging iterations"

keep iterations as small as possible while still providing enough technical and business value

refactoring techniques

migration vs. adaptation

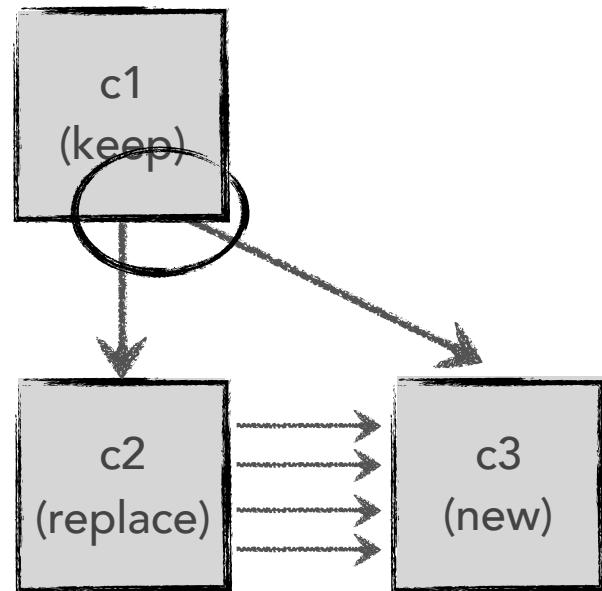


migration

*the replacement of old components
with new ones through migration
over time*

refactoring techniques

migration vs. adaptation



migration

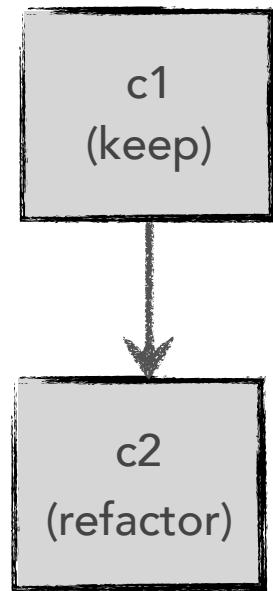
easier to rollback changes

less overall risk

requires switching logic in calling components

refactoring techniques

migration vs. adaptation

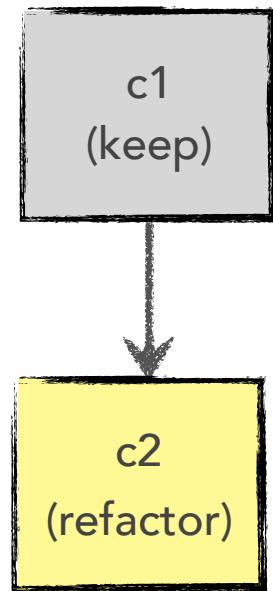


adaptation

*refactoring of existing components
into new functionality*

refactoring techniques

migration vs. adaptation



adaptation

refactor vs. replacement

harder to rollback changes

no changes to calling components

presenting your case

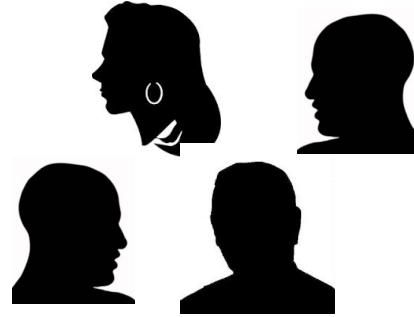


presenting your case



"how did things get this bad in the
first place?"

presenting your case



always present your plan to your immediate manager before going to the business

presenting your case



don't scare people -
"present with urgency, not with panic"

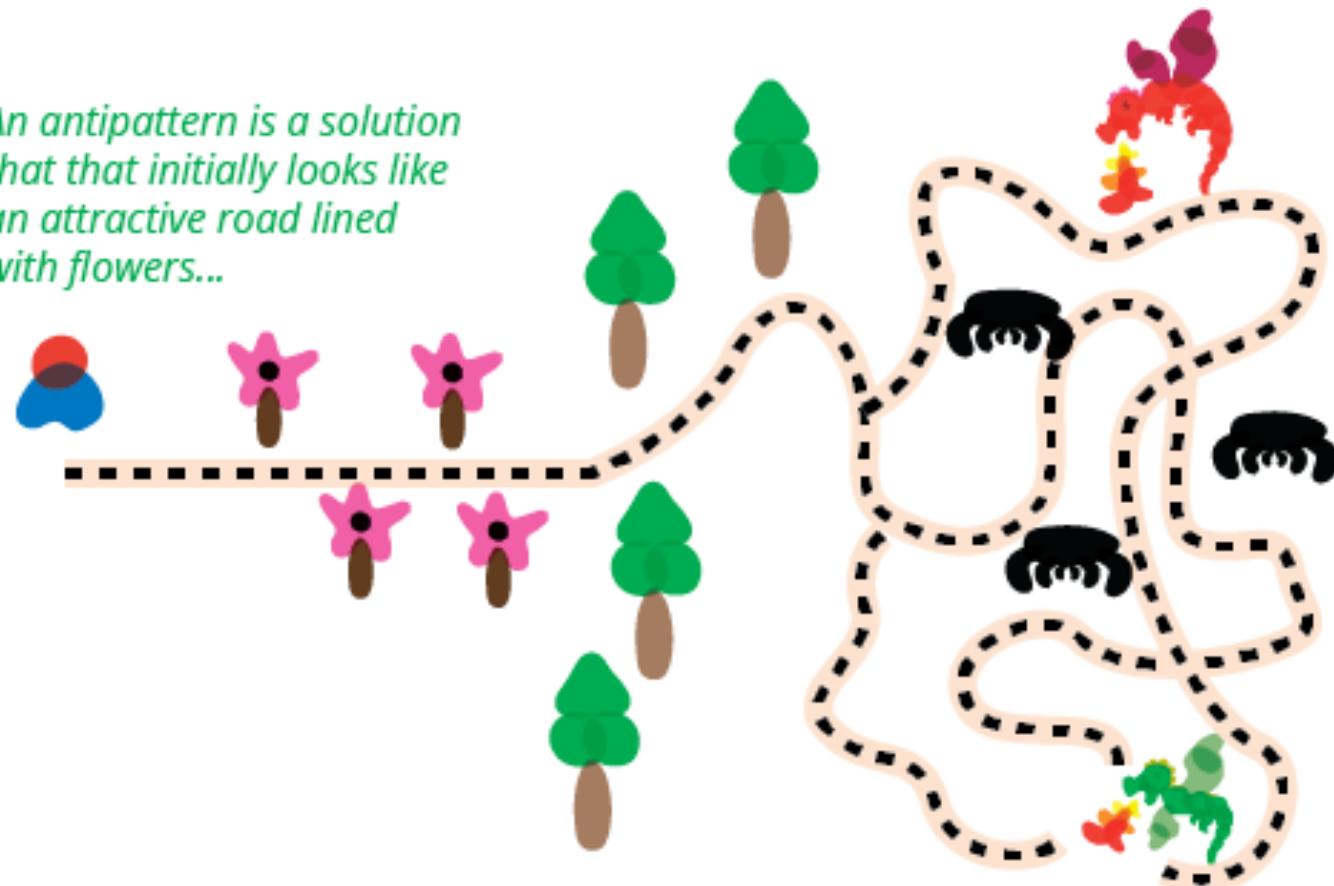
presenting your case



although you may know all the answers, don't be argumentative - take this as an opportunity to educate the business instead

Architecture Anti-pattern

*An antipattern is a solution
that initially looks like
an attractive road lined
with flowers...*



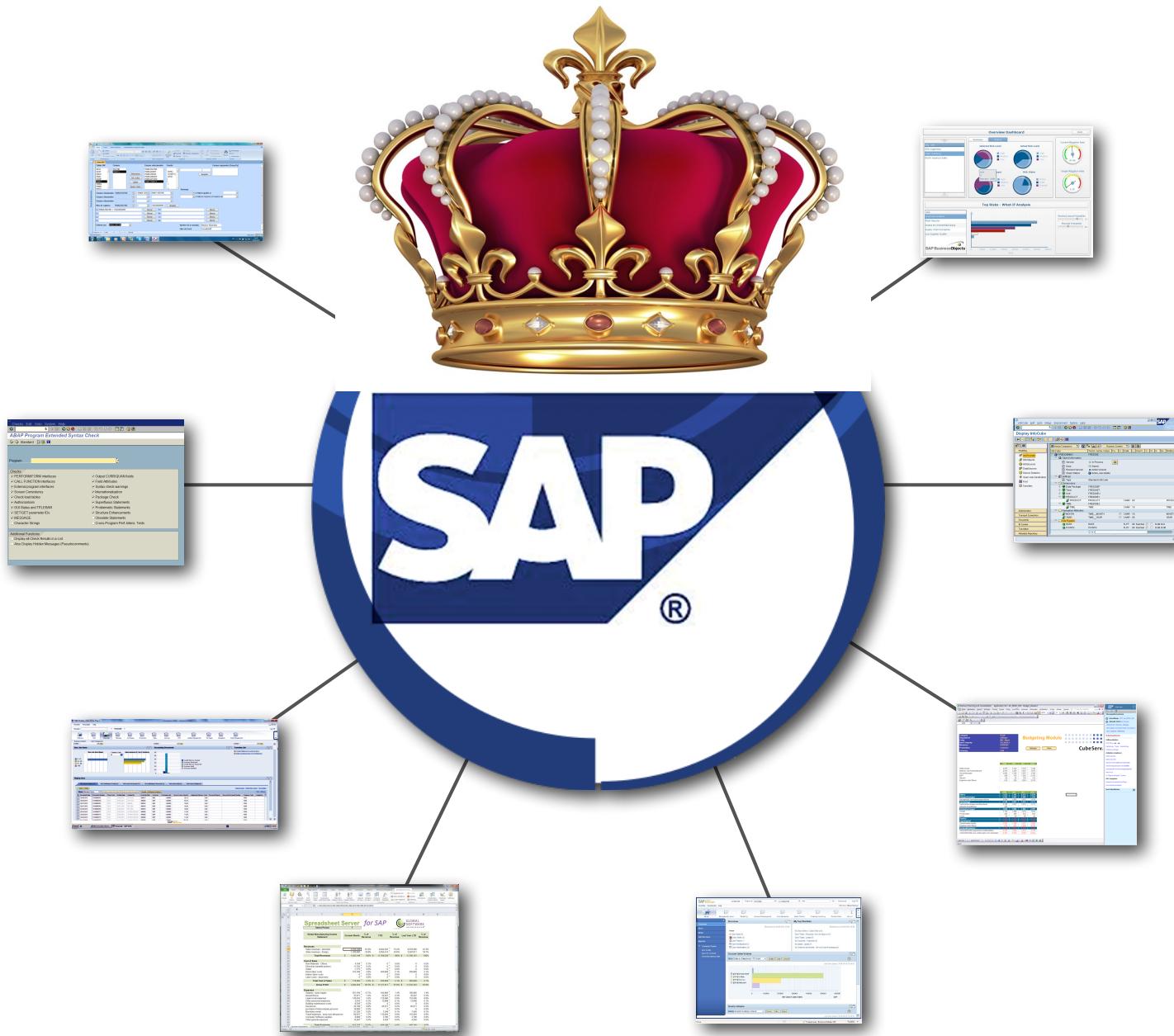
*...but further on leads you into
a maze filled with monsters*

vendor king

product-dependent architectures leading to a loss
of control of architecture and development costs



vendor king

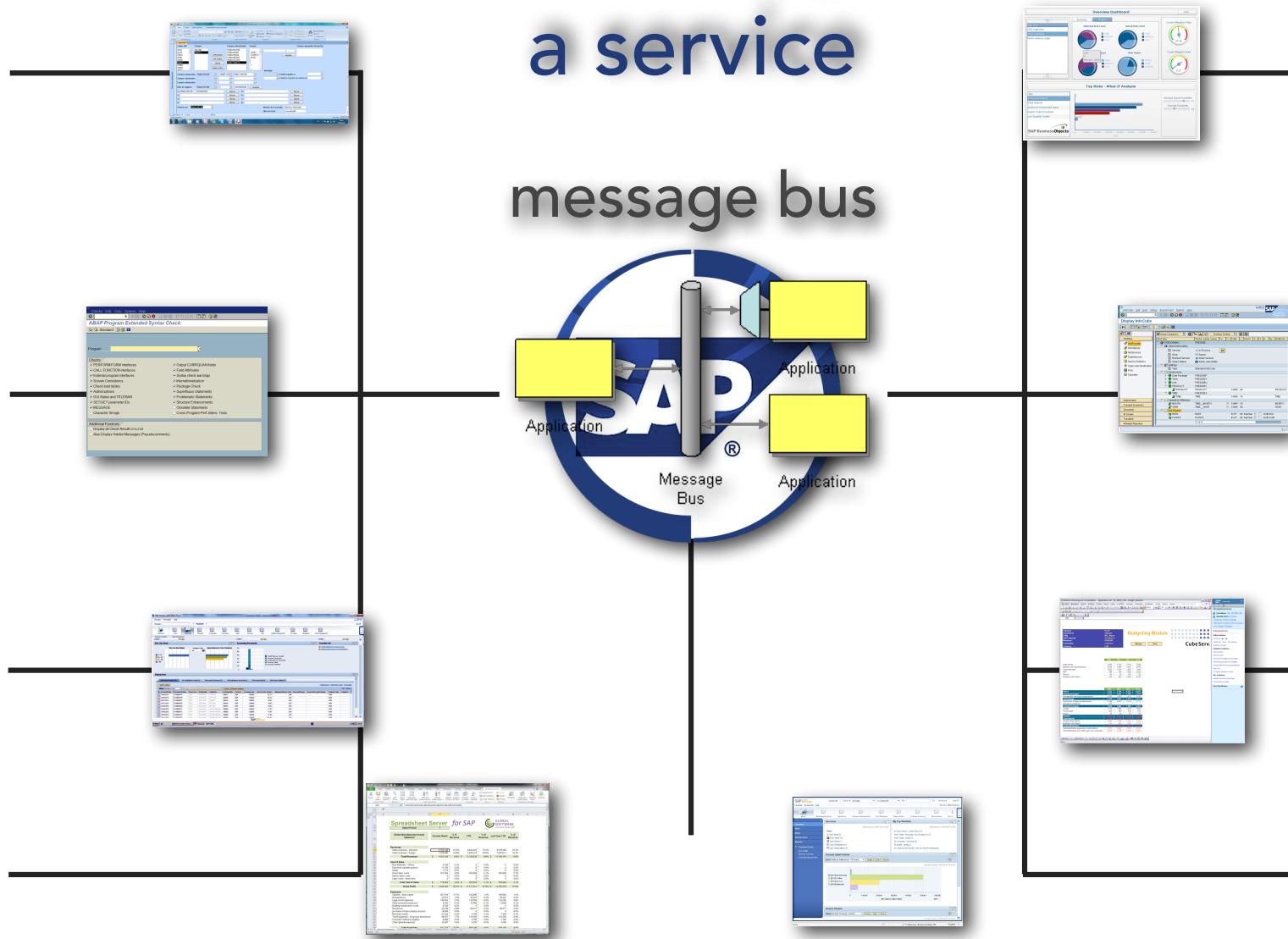


vendor king

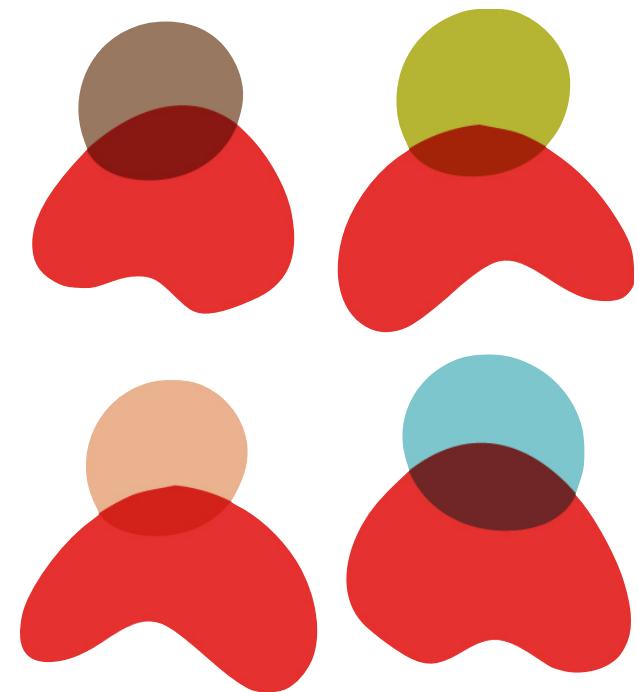
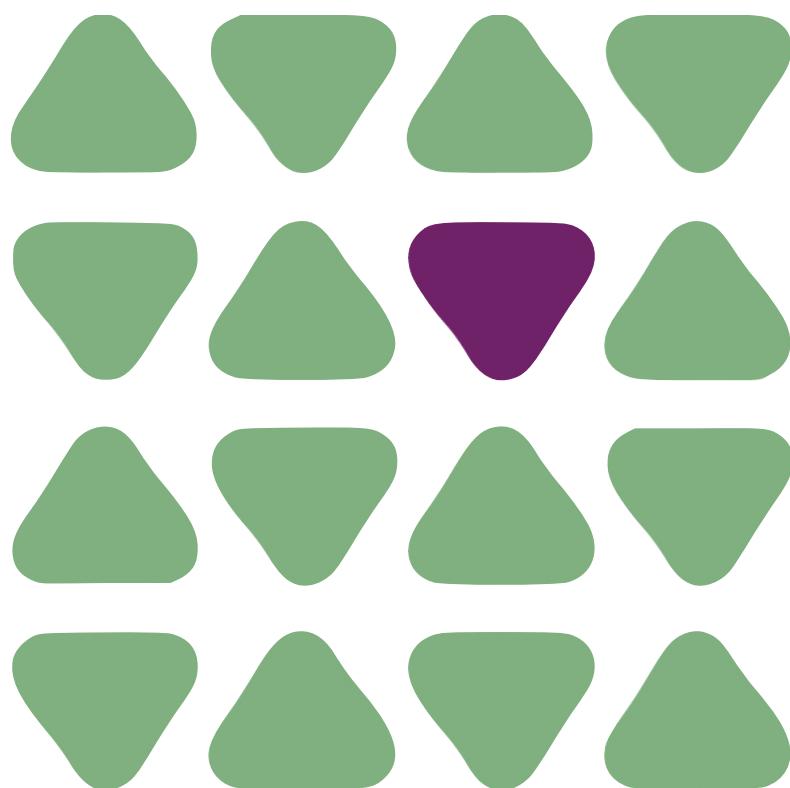


vendor king

vendor app as a service



understanding large codebases





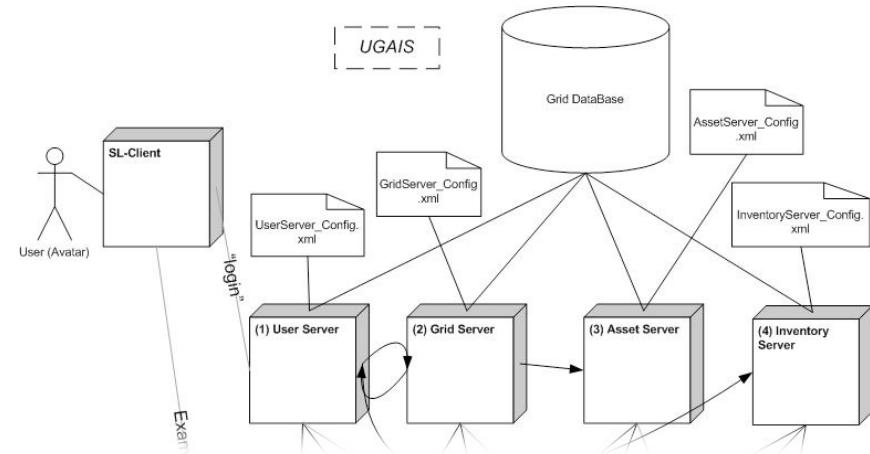
```

public void assembleBuildDetail(BuildDetail build, Map parameters) {
    Iterator<String> iterator = parameters.keySet().iterator();
    while (iterator.hasNext()) {
        String key = iterator.next();
        try {
            String value = parameters.get(key);
            if (value.startsWith("#")) {
                parameters.put(key, value.substring(1));
            } else {
                parameters.put(key, value);
            }
        } catch (Exception e) {
            log.error("Error assembling build detail for key: " + key + ", value: " + value);
        }
    }
}

void assemblePlugin(BuildDetail build, Map parameters, String className) {
    String classNameTrimmed = className.trim();
    if (classNameTrimmed.startsWith("#") || StringUtils.isEmpty(classNameTrimmed)) {
        return;
    }
    Class clazz = Class.forName(className);
    Widget digesterService = (Widget) clazz.newInstance();
    mergeParameters(build, parameters);
    build.addPluginOutput(digesterService.getDisplayName(), c
        .getOutput(parameters));
}

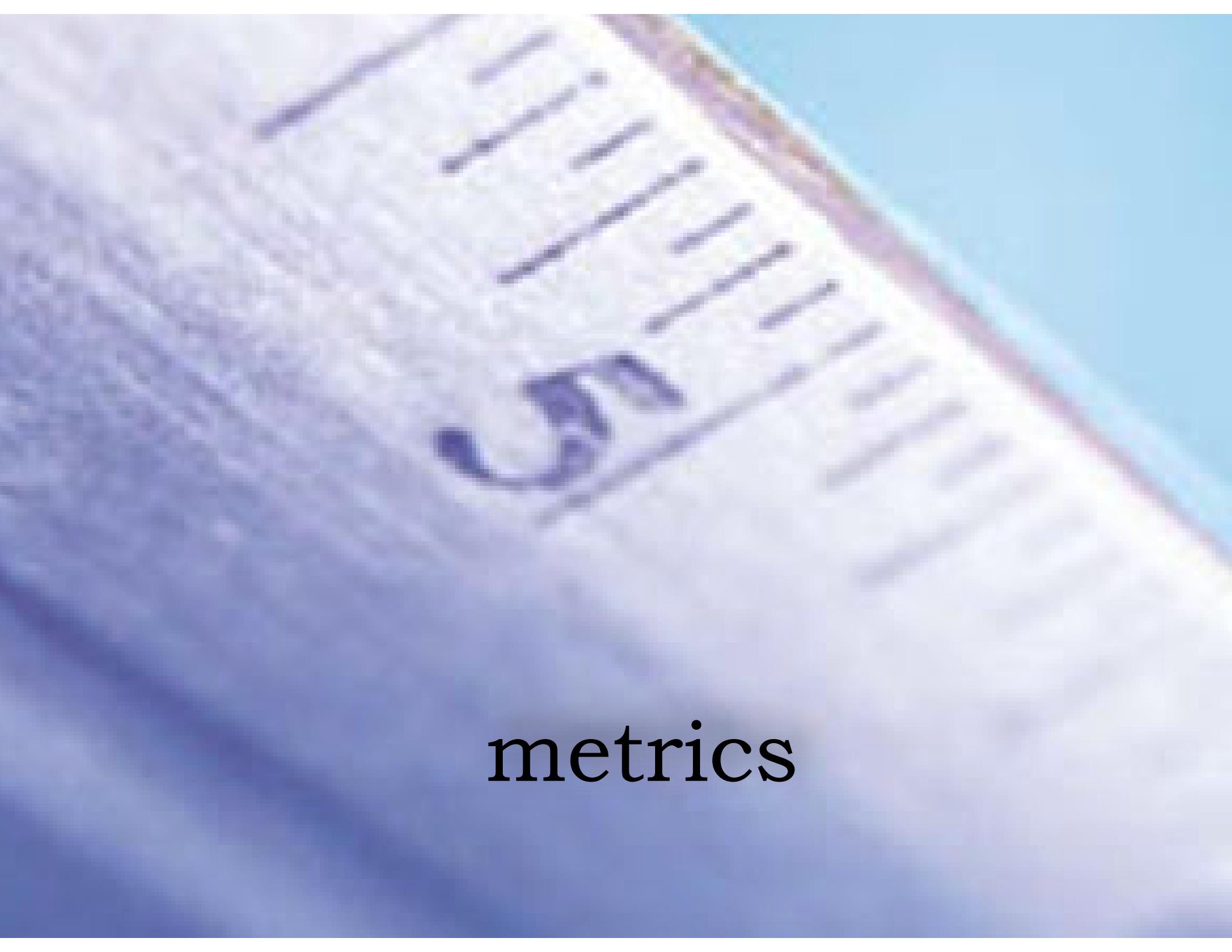
private void mergeParameters(BuildDetail build, Map parameter
    parameters.put(Widget.PARAM_CC_ROOT, configuration.getCC
    parameters.put(CC_PARAM_NAME, configuration.getCC
    parameters.put(CC_PARAM_NAME, configuration.getCC
}

```



micro <=> macro



A photograph of a stack of papers with a calculator resting on top. The calculator is a standard four-function model with a digital display showing '123'.

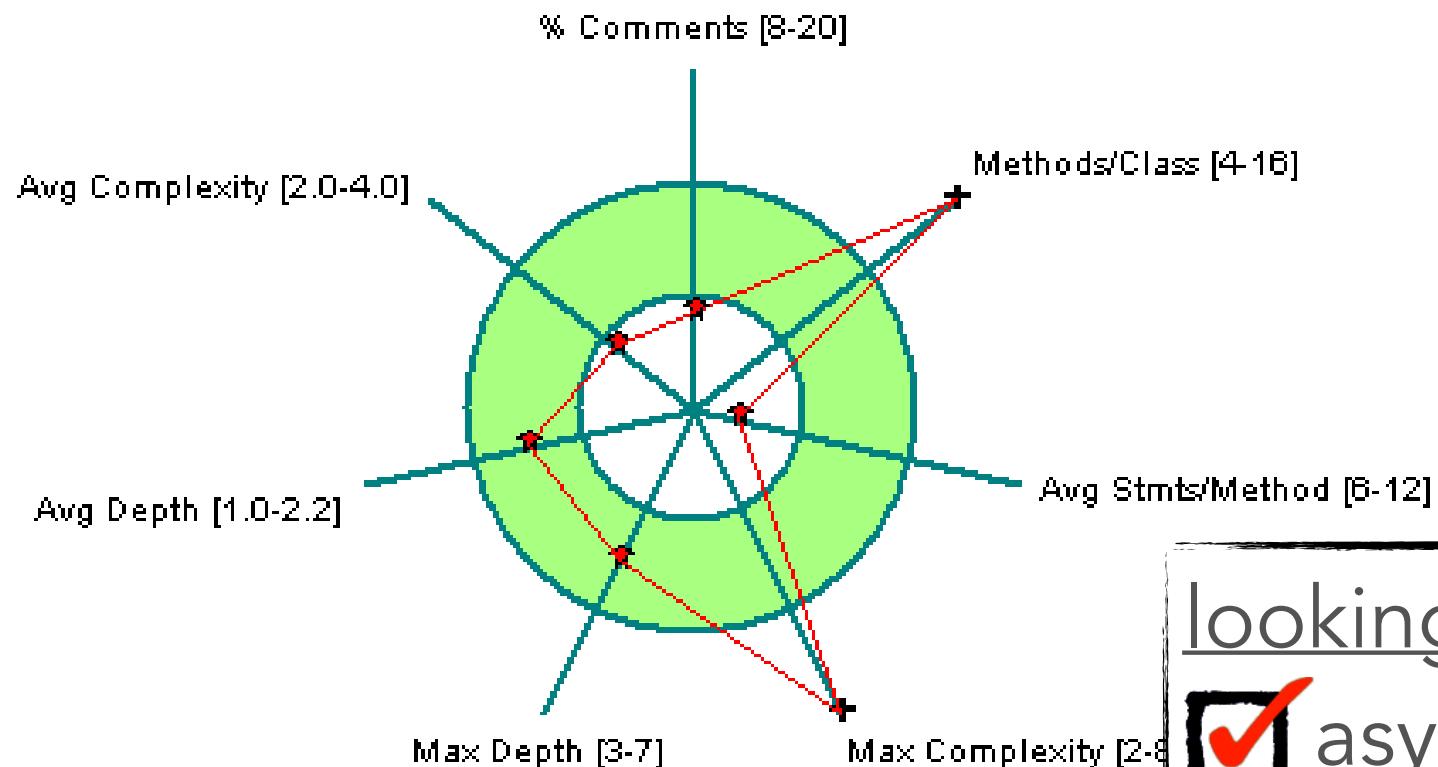
metrics

Source Monitor

<http://www.campwoodsw.com/sourcemonitor.html>



Kiviat Metrics Graph: Project 'core'
Checkpoint 'Baseline'
File 'src\main\java\org\apache\struts2\components\DoubleListUIBean.java'

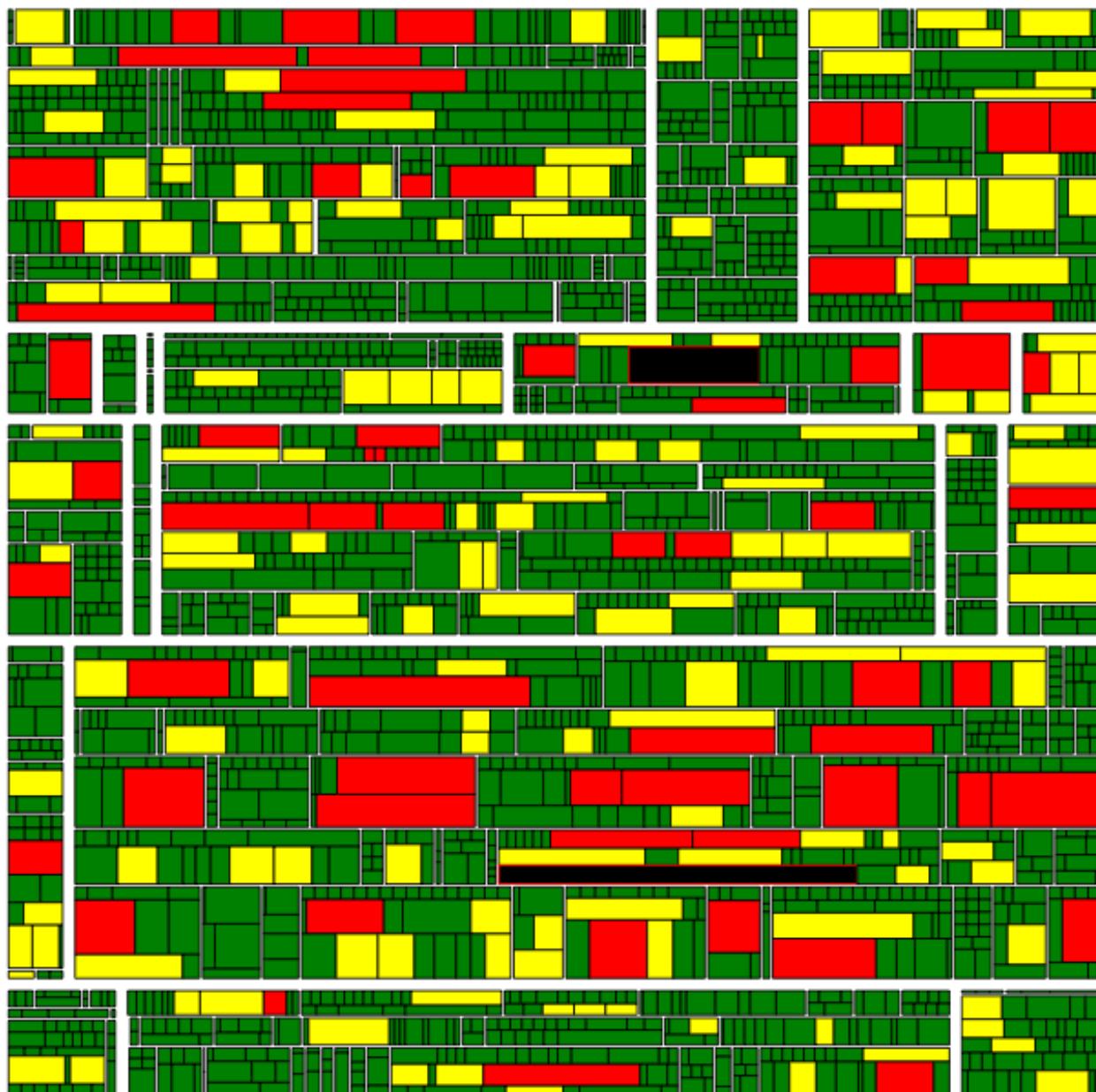


looking for:

- asymmetries
- outliers

interactive-complexity-treemap.svg

CruiseControl Complexity



Details

(Click on a rectangle to view details)

Project:

Package:

File:

Class:

NCSS:

Method Coverage:

Block Coverage:

Line Coverage:

Method:

NCSS:

CCN:

Block Coverage:

Line Coverage:

CCN 1-5

CCN 6-9

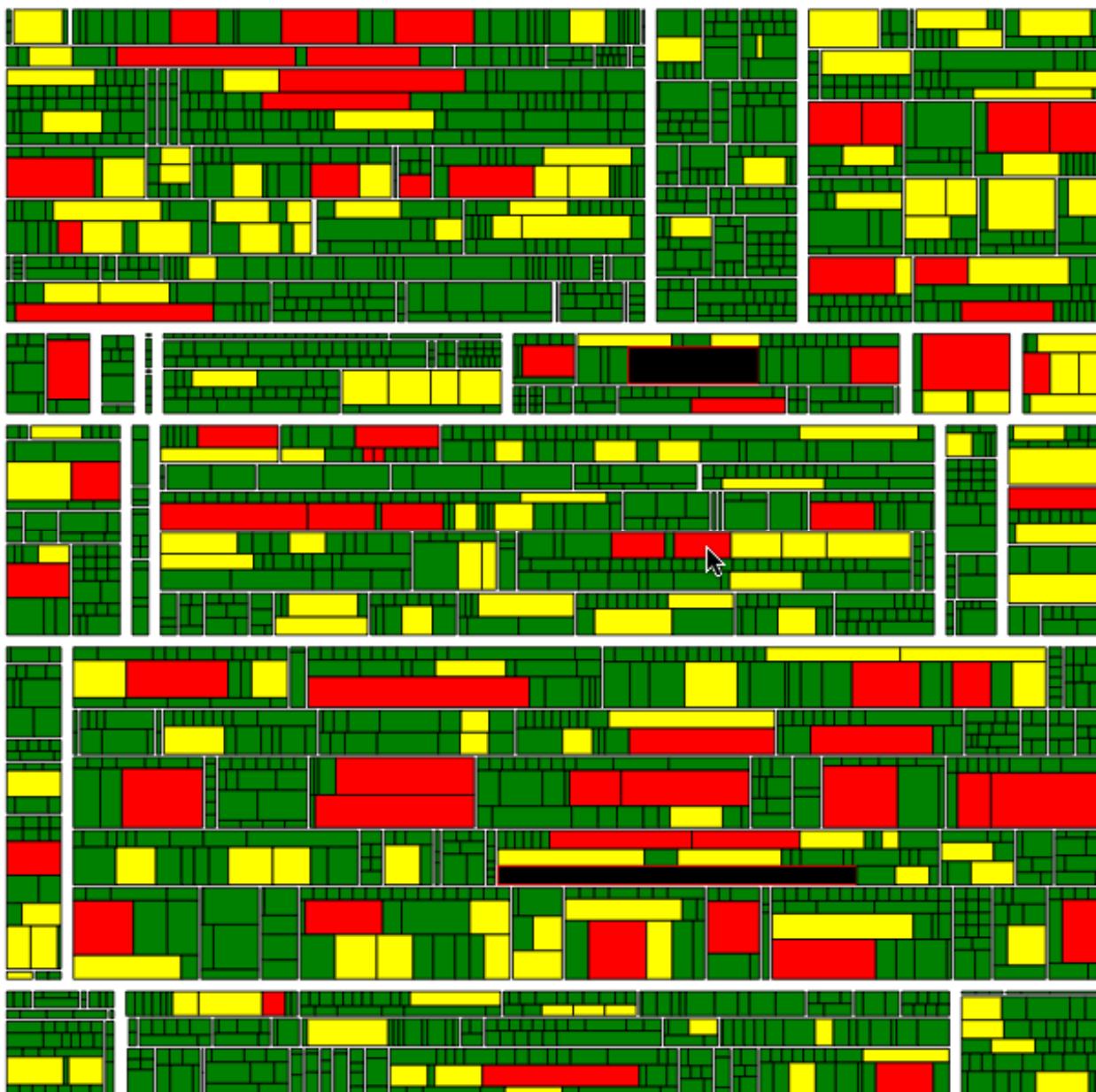
CCN 10-24

CCN 25+

N/A

interactive-complexity-treemap.svg

CruiseControl Complexity



Details

(Click on a rectangle to view details)

Project: CruiseControl

Package: net.sourceforge.cruisecontrol.publishers

File: EmailPublisher.java

Class: EmailPublisher

NCSS: 345

Method Coverage: 72.0% (36.0/50.0)

Block Coverage: 67.4% (819.0/1215.0)

Line Coverage: 68.2% (187.6/275.0)

Method: createUserSet(XMLLogHelper)

NCSS: 19

CCN: 11

Block Coverage: 100.0% (122.0/122.0)

Line Coverage: 100.0% (18.0/18.0)

CCN 1-5

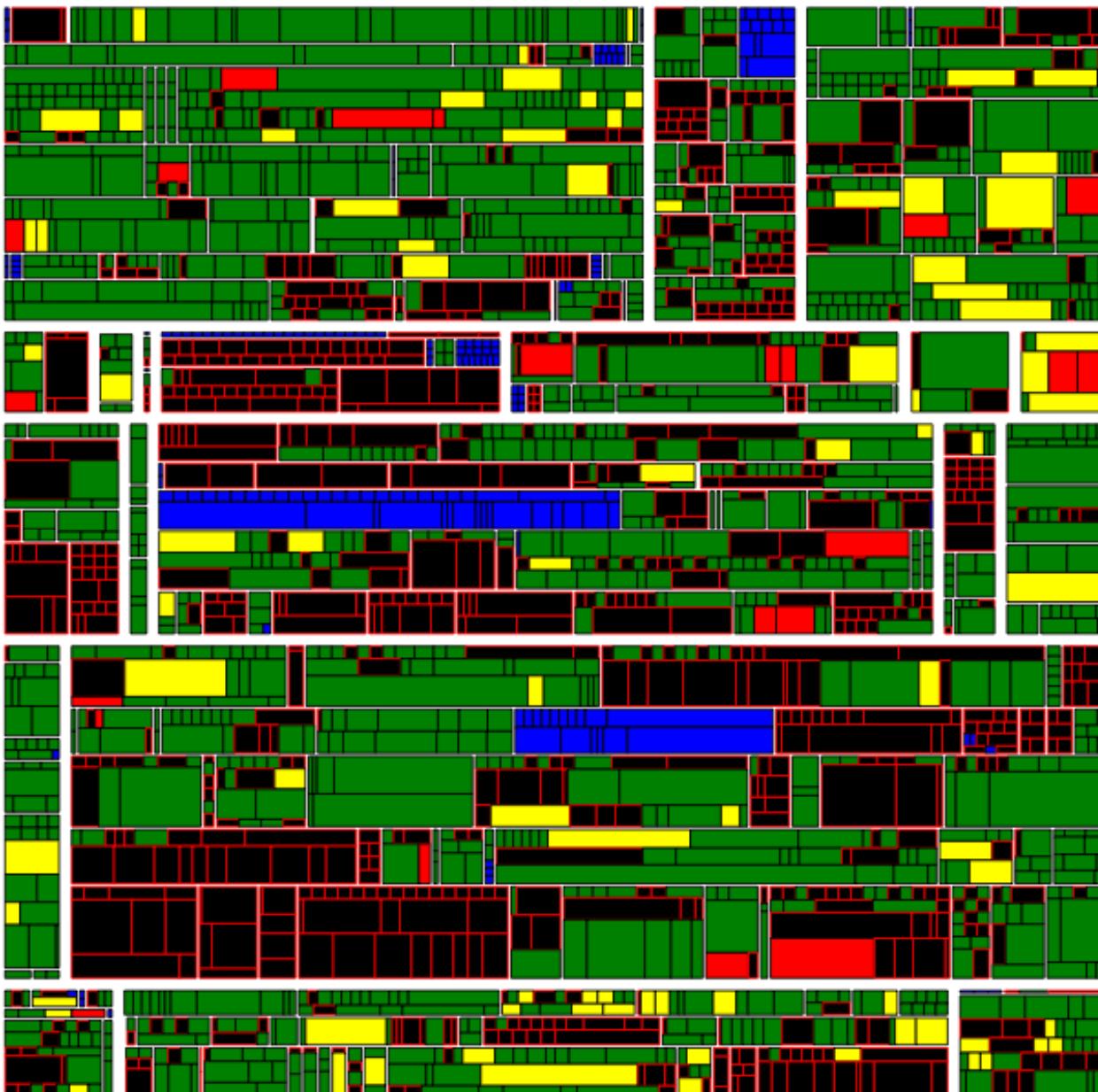
CCN 6-9

CCN 10-24

CCN 25+

N/A

CruiseControl Code Coverage

Details

(Click on a rectangle to view details)

Project:

Package:

File:

Class:

NCSS:

Method Coverage:

Block Coverage:

Line Coverage:

Method:

NCSS:

CCN:

Block Coverage:

Line Coverage:

looking for:

hot-spots



clusters

size & complexity pyramid

		5.75	NOP	224			
		6.13	NOC	1289			
	5.69	NOM		7905	7905	NOM	2.79
0.26	LOC			44988	22039	CALLS	0.40
CYCLO				11602	8798	FANOUT	

	Low	Medium	High
CYCLO / Line	0.16	0.20	0.24
LOC / method	7	10	13
NOM / class	4	7	10
NOC / package	6	17	26
CALLS / method	2.01	2.62	3.20
FANOUT / call	0.56	0.62	0.68

Design Flaws Overview

	NDD	0.6
HIT	1.6	
14.13	NOP	30
6.72	NOC	42
8.17	NOM	285
0.21	LOC	2332
CYCLO		506

Interpretation

Class hierarchies tend to be **tall** and **wide**, classes with many directly derived sub-classes.

Classes tend to be:

- contain an **average** number of methods
- be organized in **average-sized** packages

Methods tend to:

- be rather **short** and having an **average** number of parameters
- call **several methods** from **few components**

URITag
AnchorTag
DoubleSelectTag
Component

Loading the model took: 1m 1s

inCode Helium

Package Map X

Coupling perspective on system main

DF CX EN CP

looking for:
 holistic compliance
 outliers

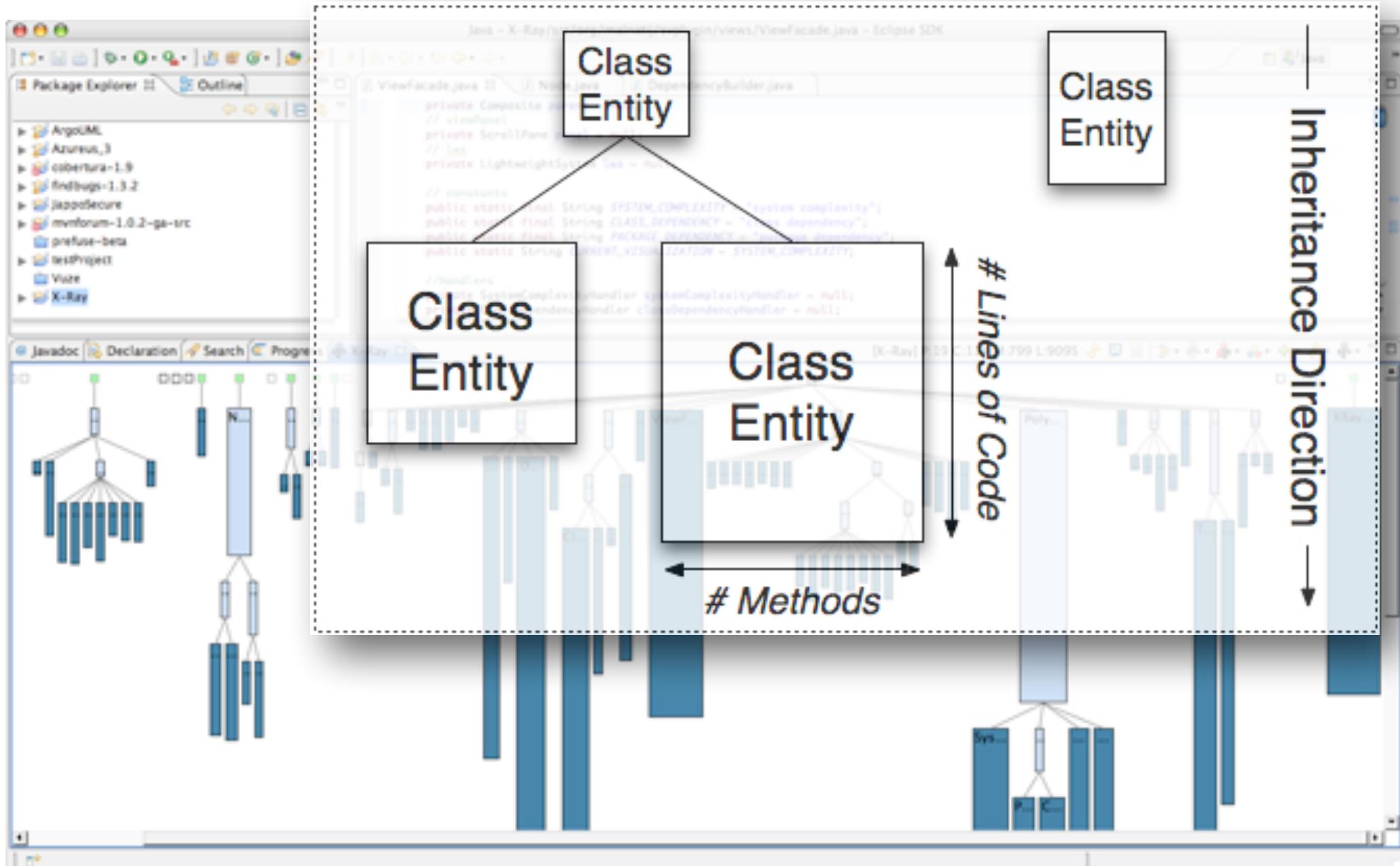


<http://www.intooitus.com>

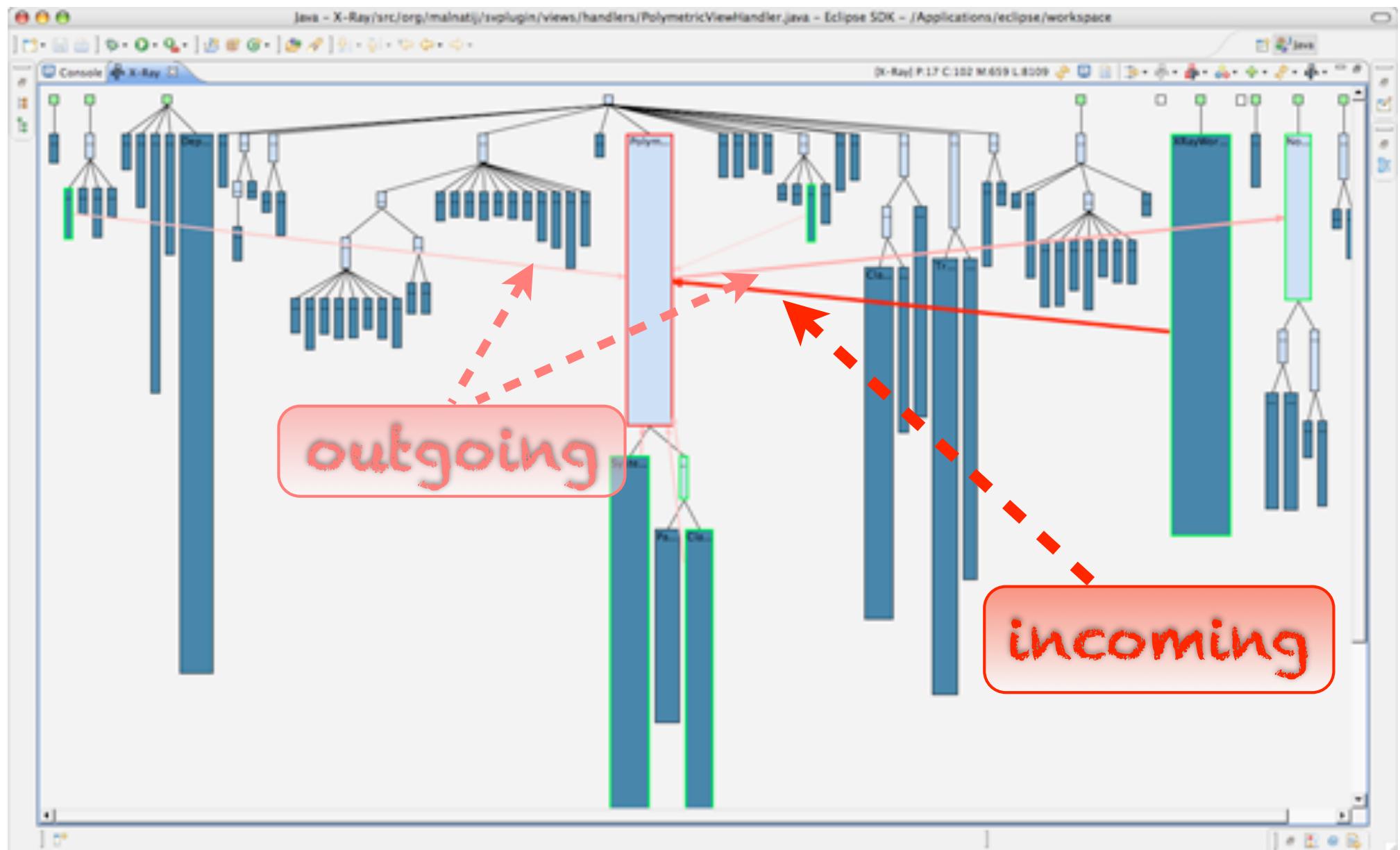


X-Ray

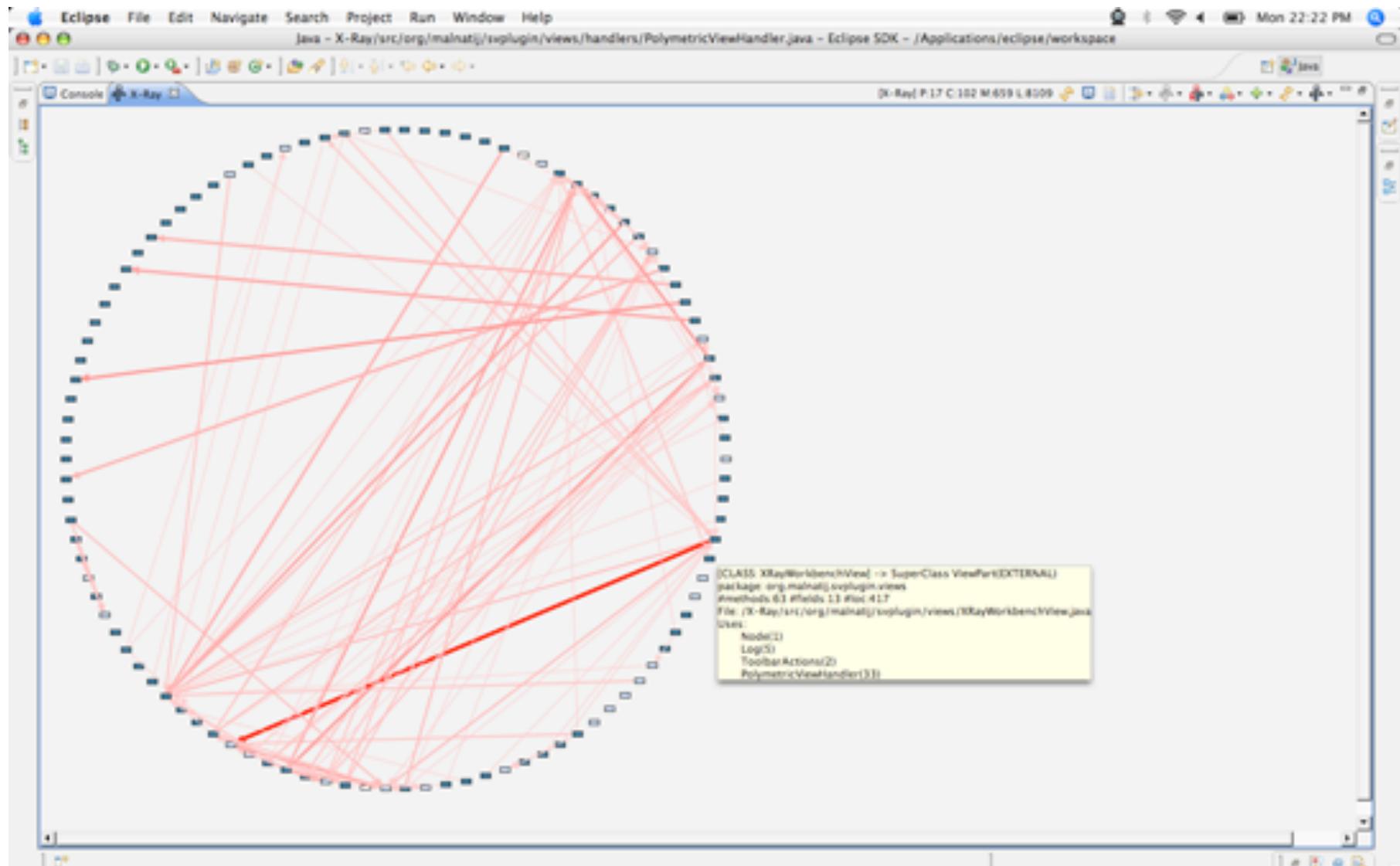
<http://xray.inf.usi.ch/xray.php>



XRay dependencies

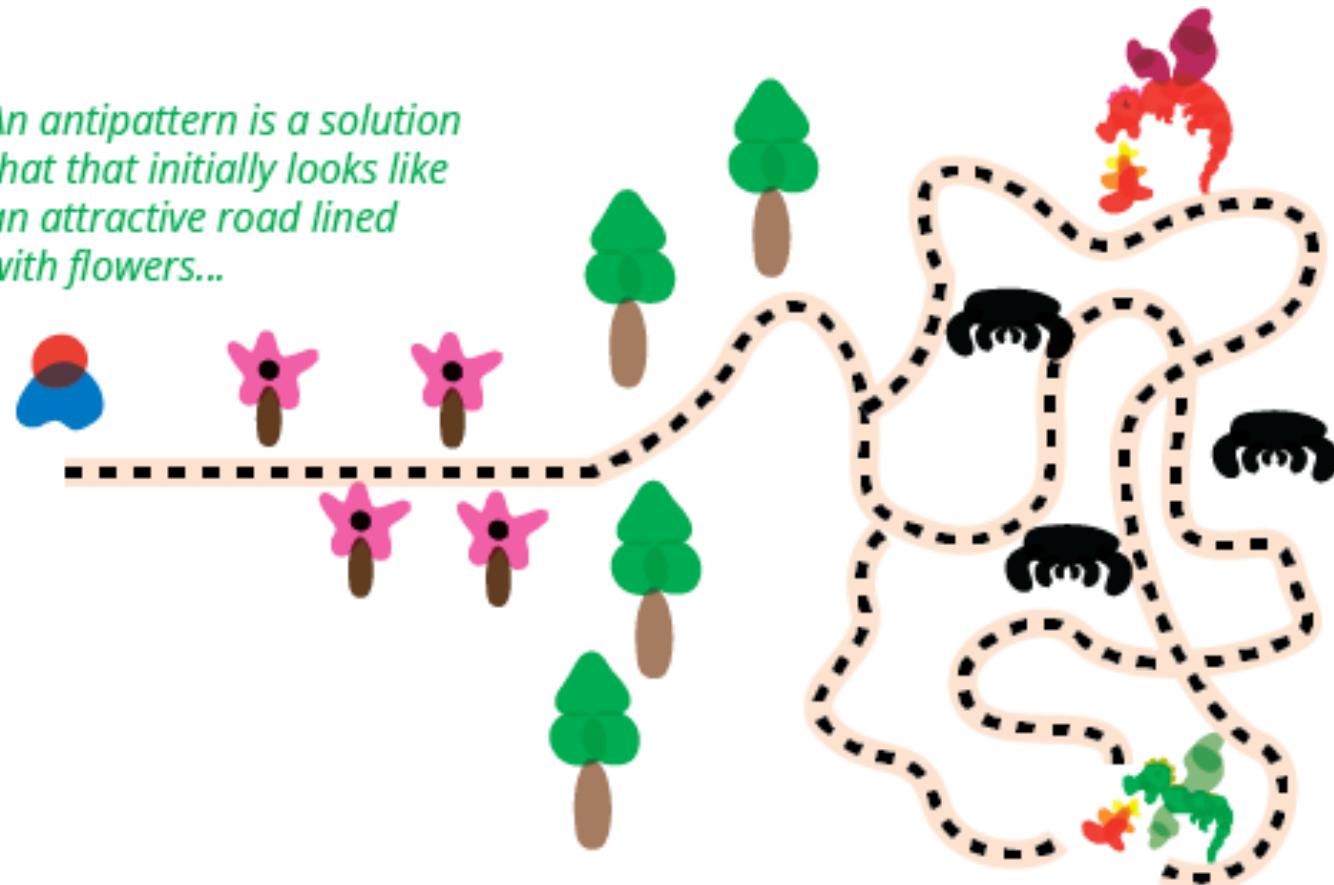


class & package dependency views



Architecture Anti-pattern

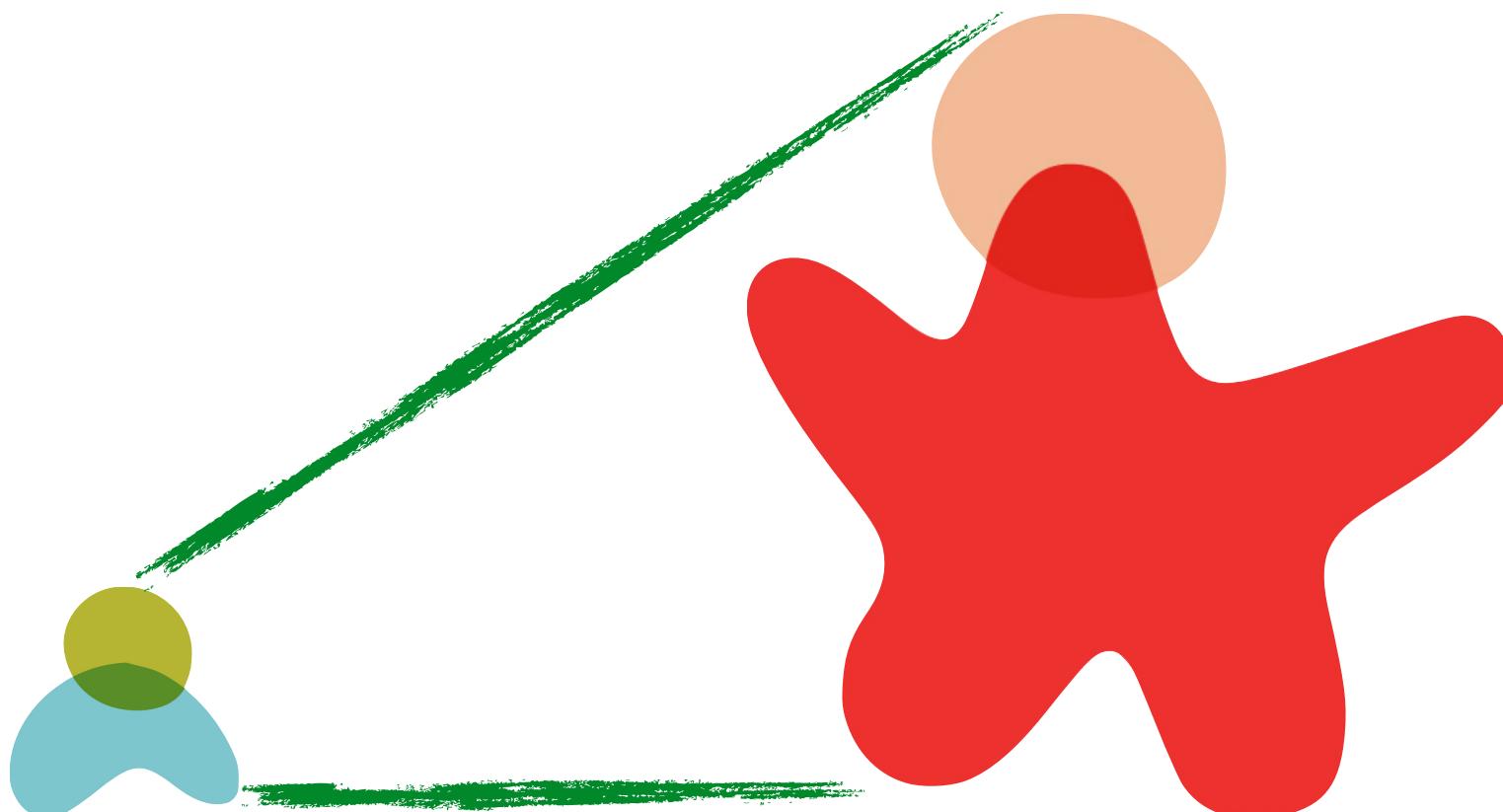
*An antipattern is a solution
that initially looks like
an attractive road lined
with flowers...*

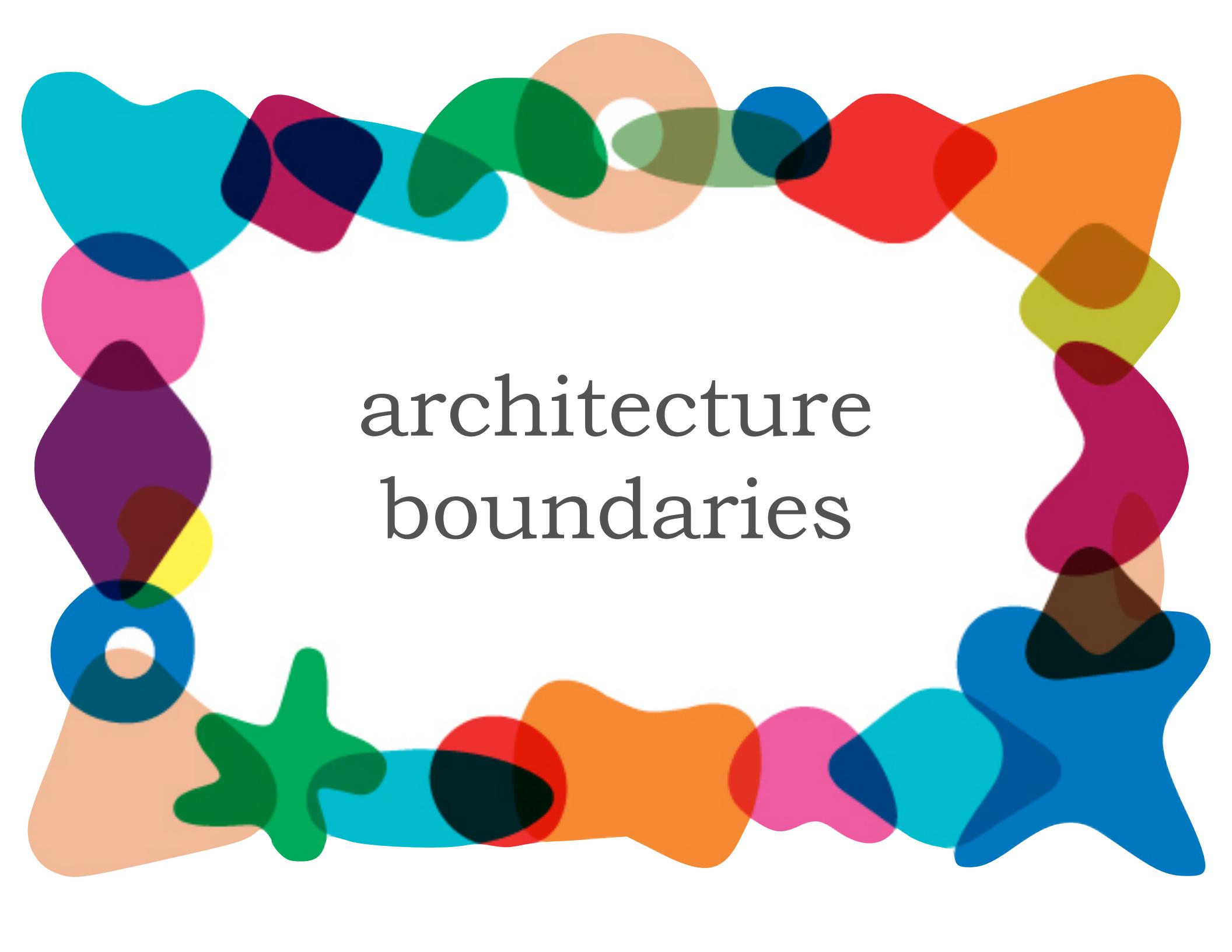


*...but further on leads you into
a maze filled with monsters*

Imposter Syndrome

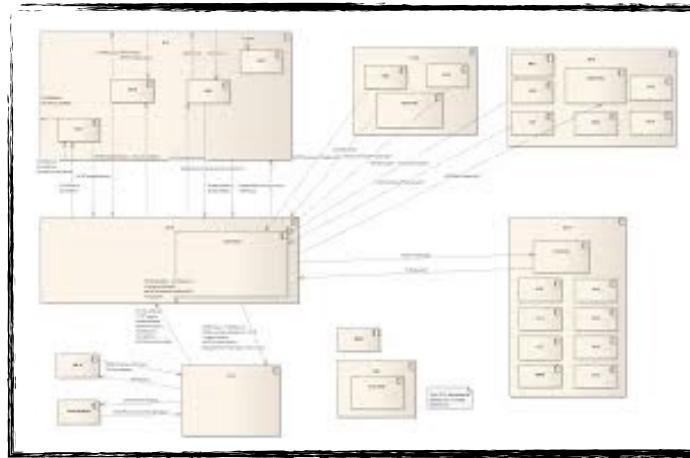
a term coined in the 1970s by psychologists and researchers to informally describe people who are unable to internalize their accomplishments



The background of the image consists of a dense arrangement of overlapping, organic-shaped, rounded rectangles in various colors. These colors include teal, magenta, orange, green, blue, yellow, purple, pink, and brown. The shapes are layered and intertwined, creating a complex and textured visual field.

architecture
boundaries

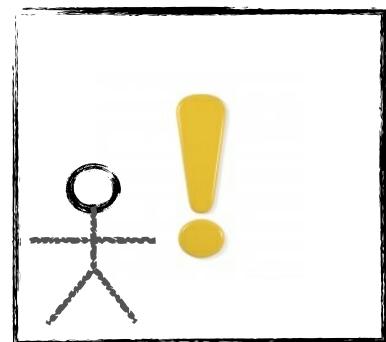
architectural boundaries



there is an art to defining the box
that development teams can work
in to implement the architecture

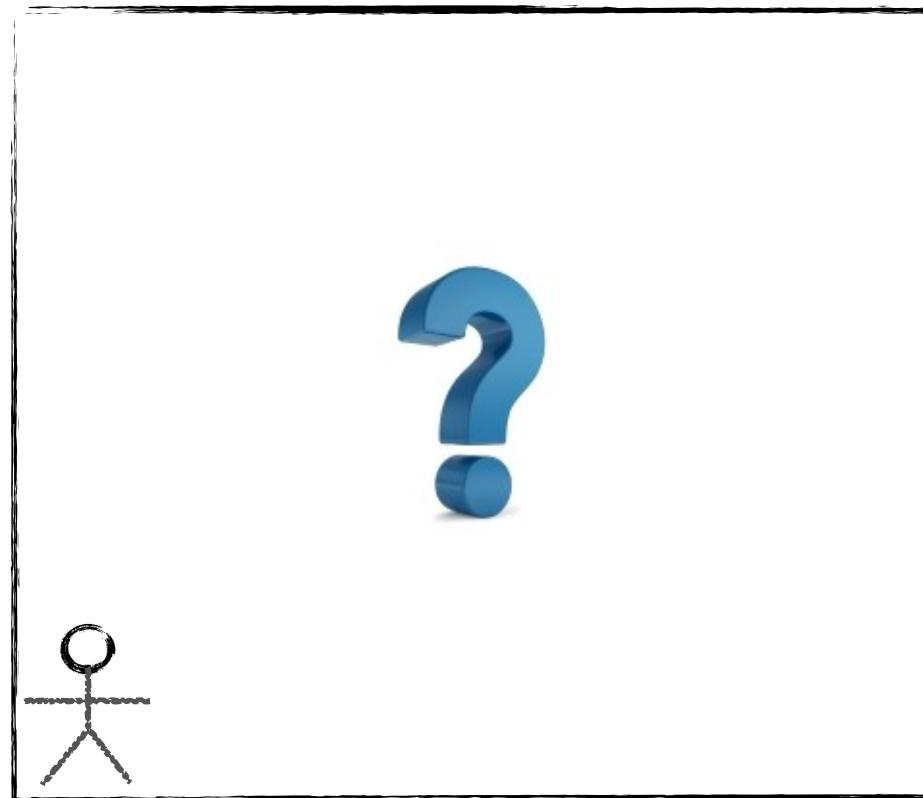
architectural boundaries

tight boundaries



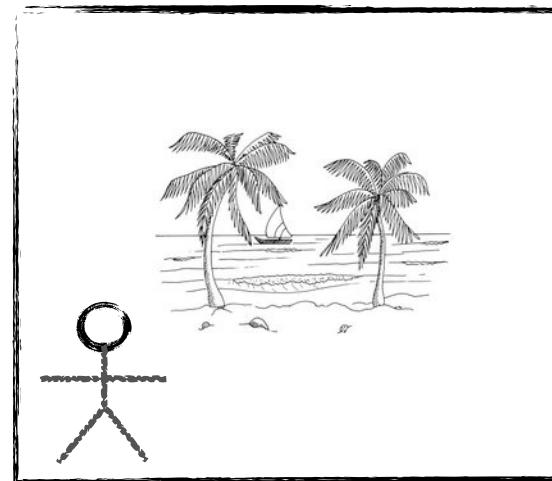
architectural boundaries

loose boundaries



architectural boundaries

appropriate boundaries



architect personalities



control freak architect

too involved in the implementation details
rejects every suggestion for new libraries or tools
decisions and constraints are too fine-grained
controls every aspect of the development effort

architect personalities



armchair architect

the architect hasn't coded in a very, very, very long time and doesn't take implementation into account

the architect is overly vague and doesn't supply enough details to provide the right direction

the architect has no idea what they are doing and is way over their head with respect to the technology or business

architect personalities



effective architect

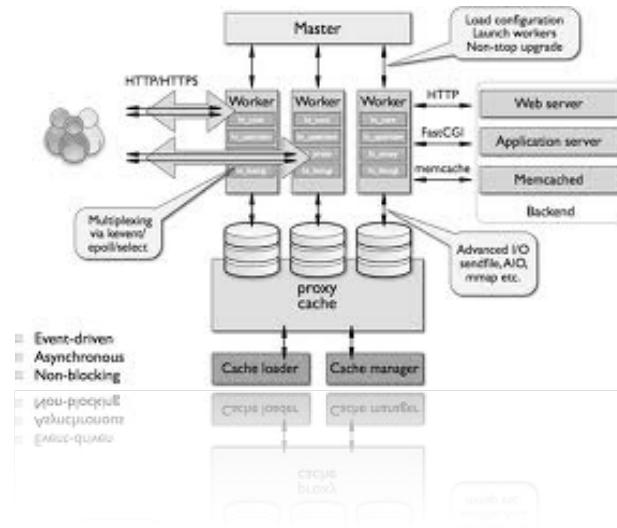
the right level of decisions and guidance are communicated so that the development teams can make their own choices

the development teams are largely able to work independently without constant architect involvement

makes sure the development teams have the right tools for the right job

controlling the boundaries

the architect defines the architecture
and design principles used to **guide**
technology decisions



controlling the boundaries controlling the layered stack

development team: we decided to incorporate the guava library for the camel case conversion requirement.



"take it out. I only want you to use the core
java api for this application. period."

controlling the boundaries controlling the layered stack

development team: we decided to incorporate the guava library for the camel case conversion requirement.



"guava - that's a cool library name. carry on..."

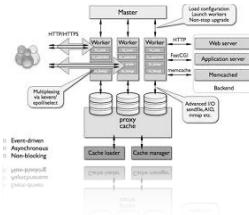
controlling the boundaries controlling the layered stack

development team: we decided to incorporate the guava library for the camel case conversion requirement.



if that's the only feature you are leveraging, you should just use the java api. if there are other features you can justify, then we can talk about it.

controlling the boundaries controlling the layered stack



what design principle would you create to manage this type of boundary?

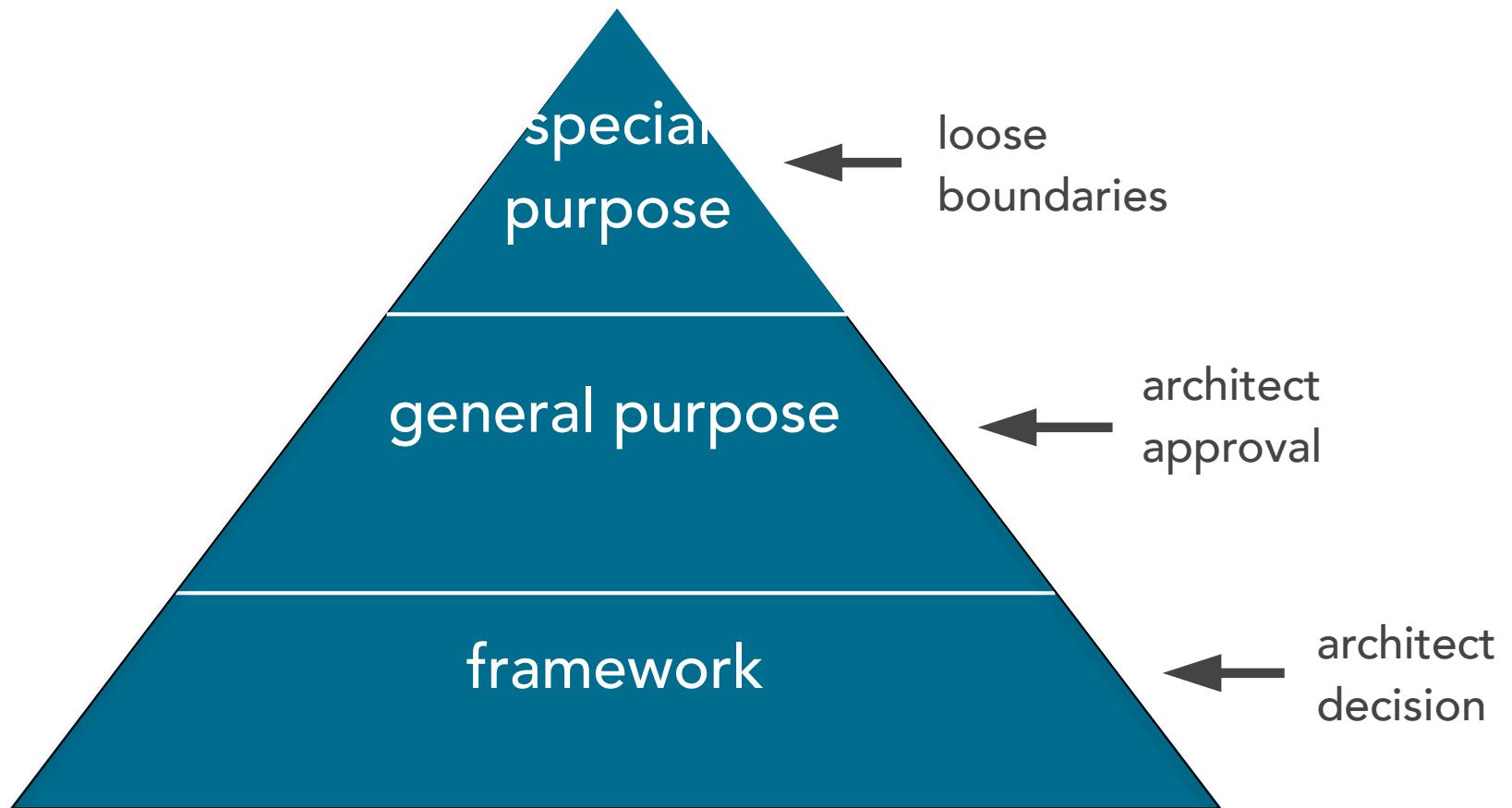
look for overlaps in existing functionality

always seek justification for adding a new library

provide guidance by making it clear what type of libraries need discussion and approval and which ones don't

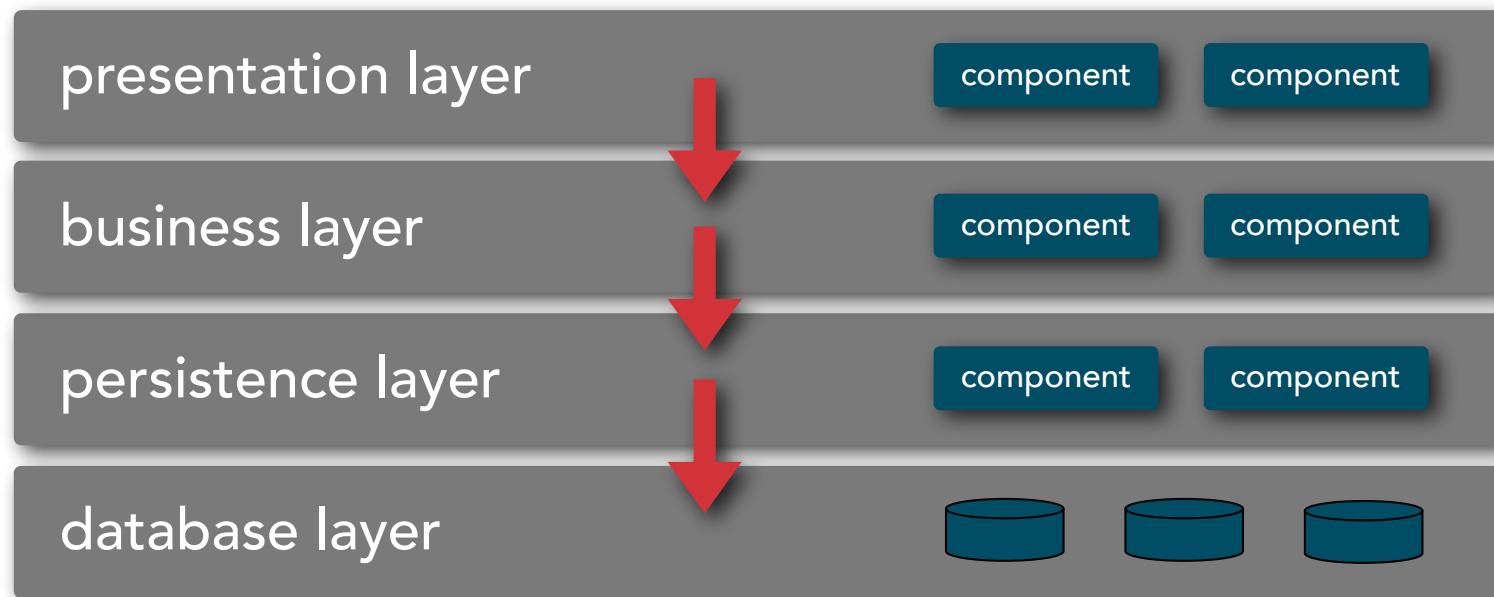
controlling the boundaries

controlling the layered stack



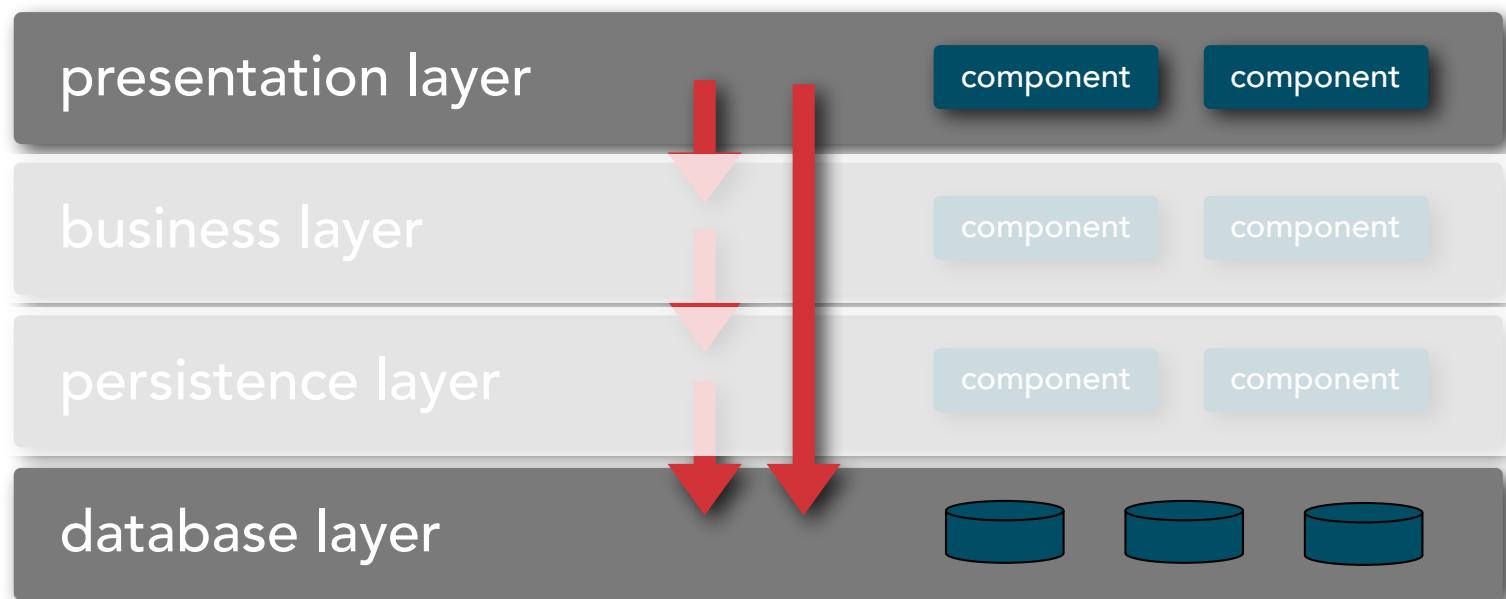
controlling the boundaries pattern governance

development team: we need better performance - can we access the database directly from the presentation layer?



controlling the boundaries pattern governance

development team: we need better performance - can we access the database directly from the presentation layer?



controlling the boundaries

pattern governance

development team: we need better performance - can we access the database directly from the presentation layer?



"no."

controlling the boundaries

pattern governance

development team: we need better performance - can we access the database directly from the presentation layer?



"it doesn't matter to me. if you think it would
help performance, then go for it."

controlling the boundaries pattern governance

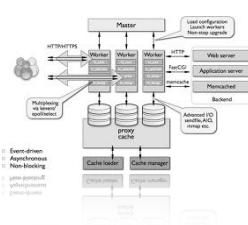
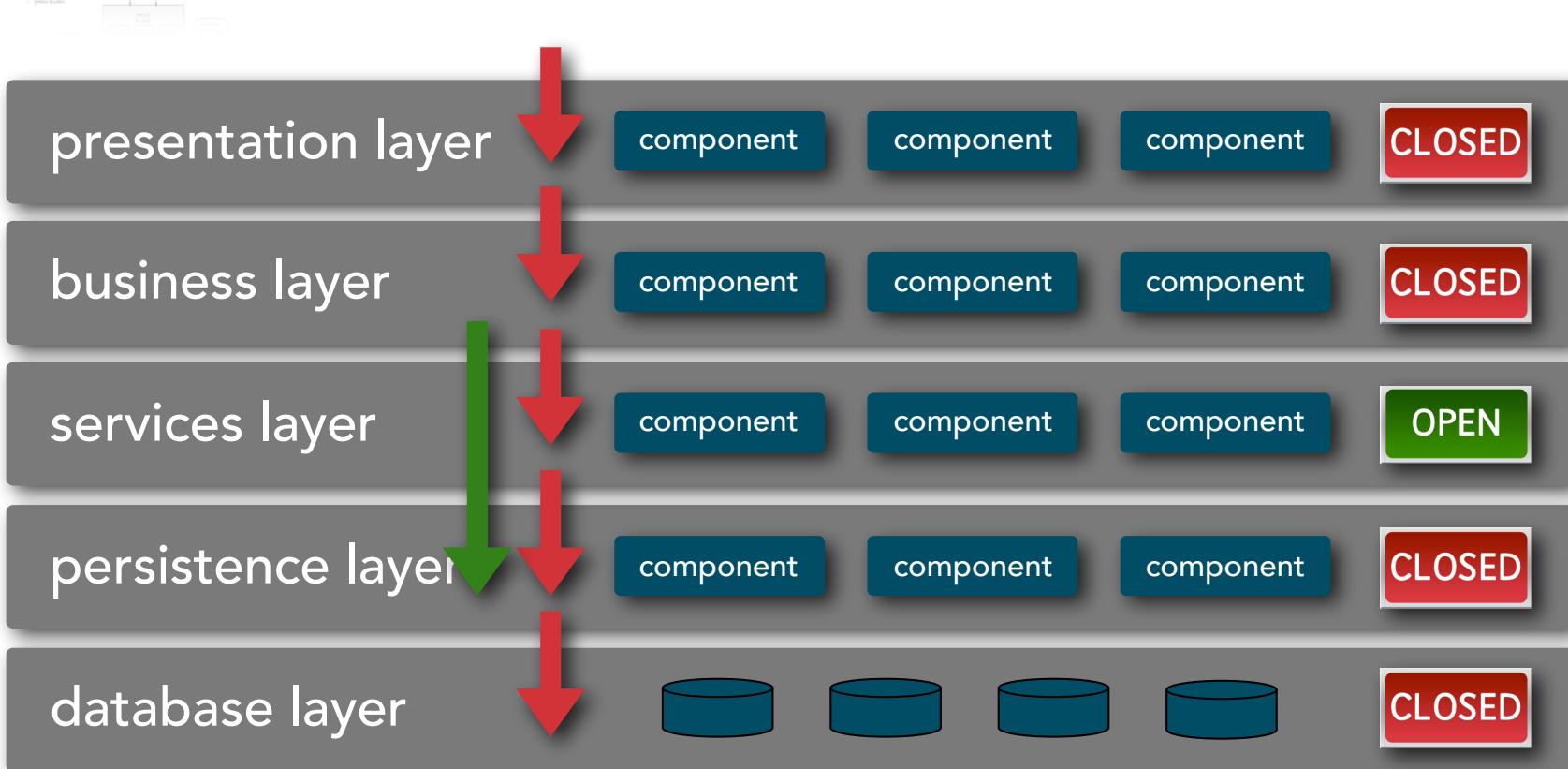
development team: we need better performance - can we access the database directly from the presentation layer?



"those layers are closed so that we can better control change through layer isolation, so no. have you been able to identify what might be causing the performance issues?"

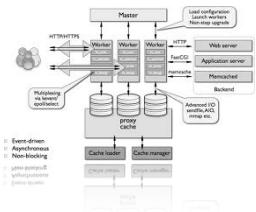
controlling the boundaries pattern governance

what design principle would you create to manage this type of boundary?



controlling the boundaries pattern governance

what design principle would you create to manage this type of boundary?



clearly document and diagram the architecture

justify your reasons for the architecture decisions

make sure you effectively communicate your decisions

controlling the boundaries

architecture scope

development team: we added some really cool capabilities
that might be needed sometime in the future...



"did i tell you to add those capabilities? didn't
think so. take them out."

controlling the boundaries

architecture scope

development team: we added some really cool capabilities
that might be needed sometime in the future...



"great forward thinking guys! someday the users might
need that capability, and now we have it ready..."

controlling the boundaries

architecture scope

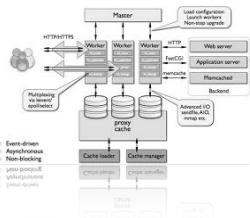
development team: we added some really cool capabilities that might be needed sometime in the future...



"let's verify those features with the analysts to see if we need them. if not then we'll take them out. we just need to make sure we don't do anything to prevent that capability from being added in the future."

controlling the boundaries architecture scope

what design principle would you create to manage this type of boundary?



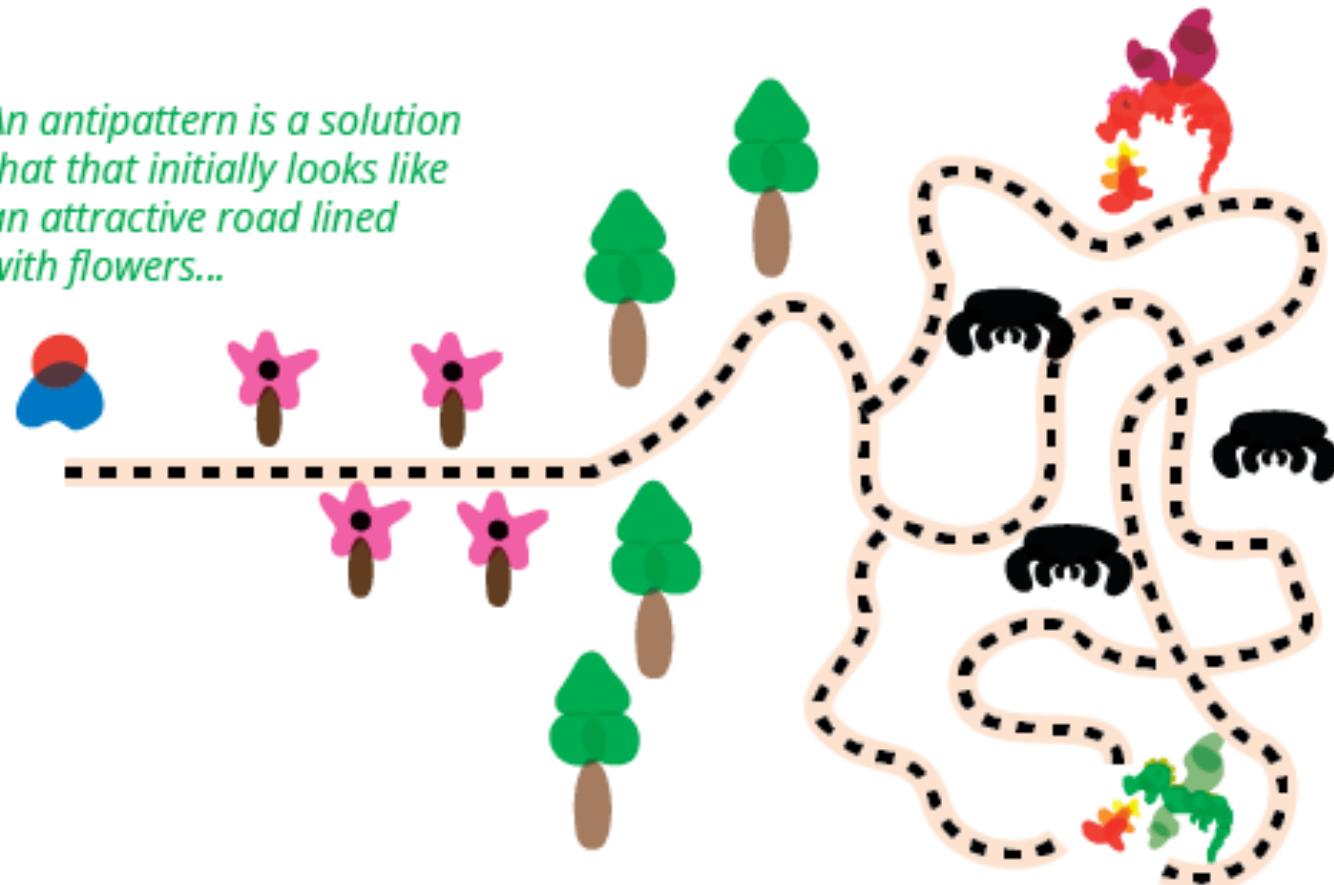
adding additional features over and above the requirements adds additional development, testing, and maintenance time and costs

this practice can lead to the *infinity architecture anti-pattern*

document non-required features, verify it with the user community, and **make sure you don't do anything to restrict that functionality in the future**

Architecture Anti-pattern

*An antipattern is a solution
that initially looks like
an attractive road lined
with flowers...*



*...but further on leads you into
a maze filled with monsters*



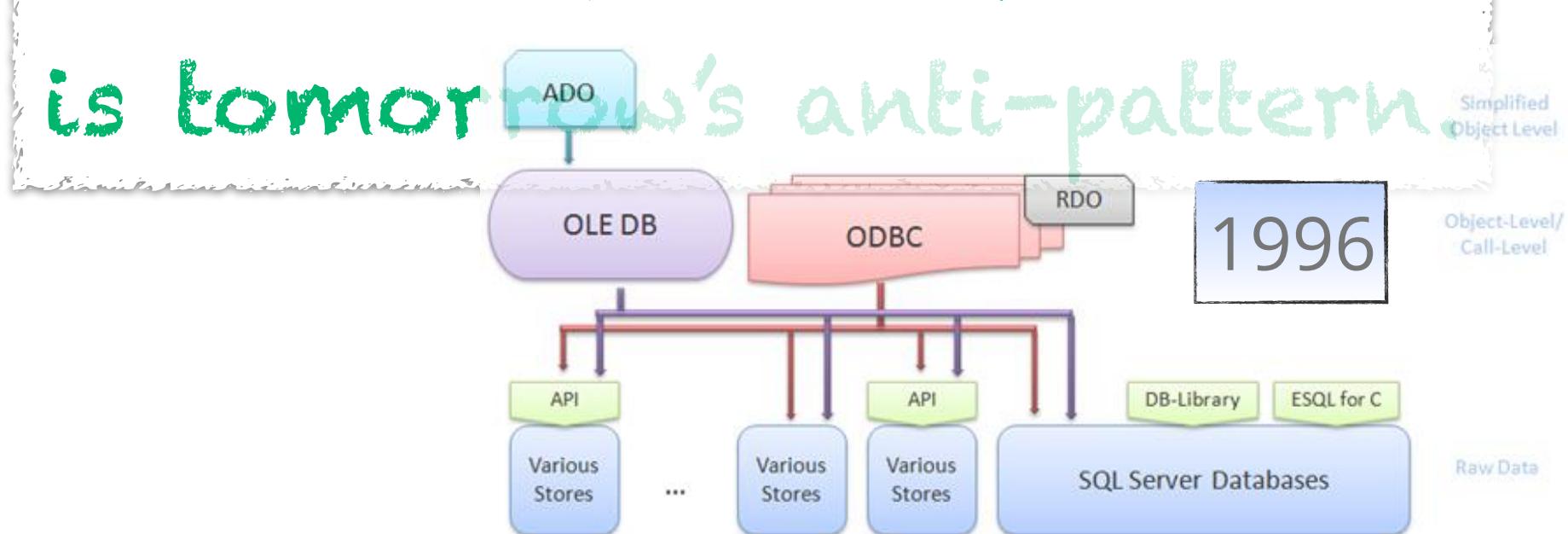
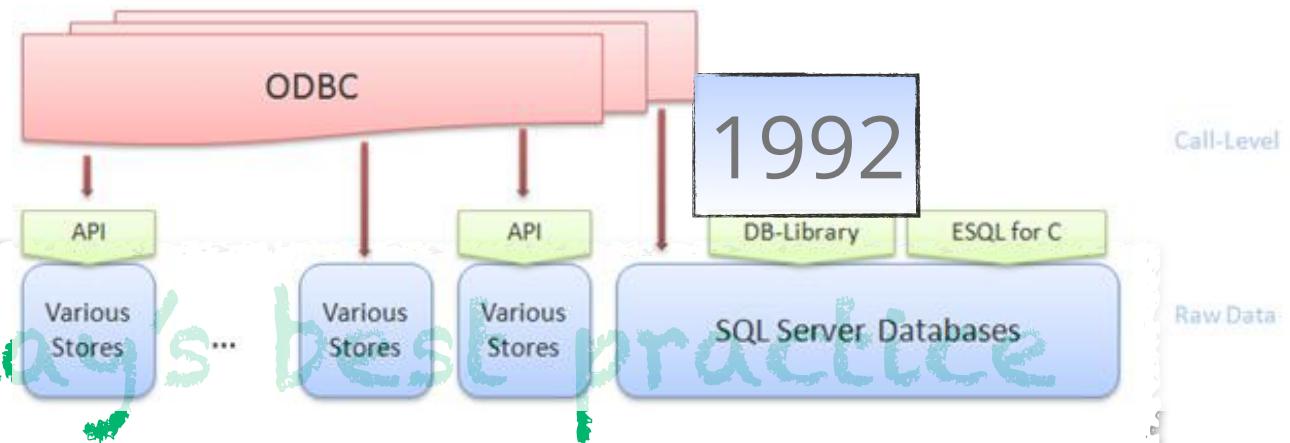
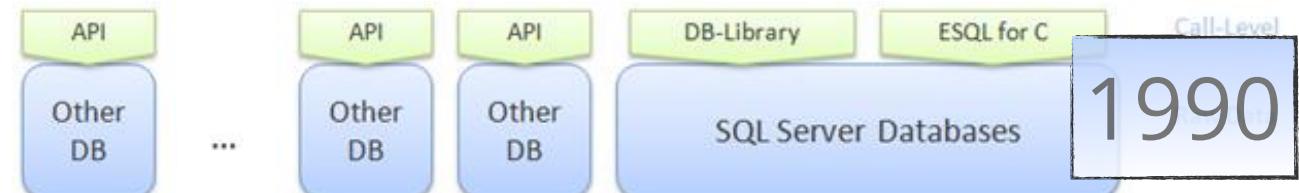
Yesterday's best practice
is tomorrow's anti-pattern.

A Case against the GO TO Statement.

by Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

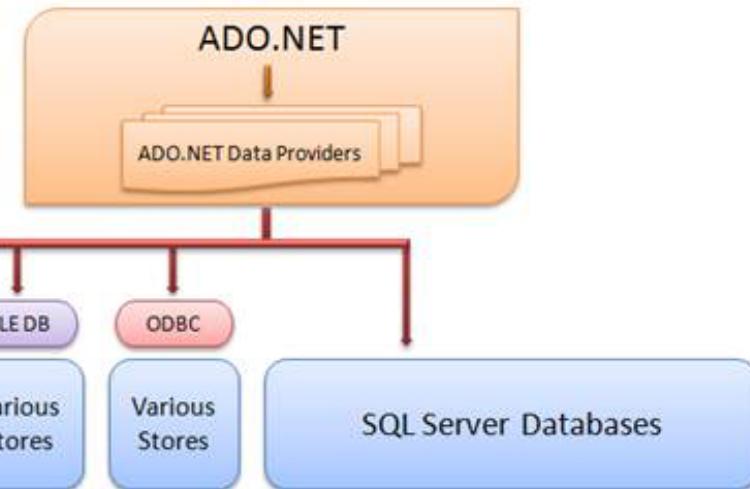
Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to effectuate the desired effect; it is this process that in its dynamic behaviour has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.





2002

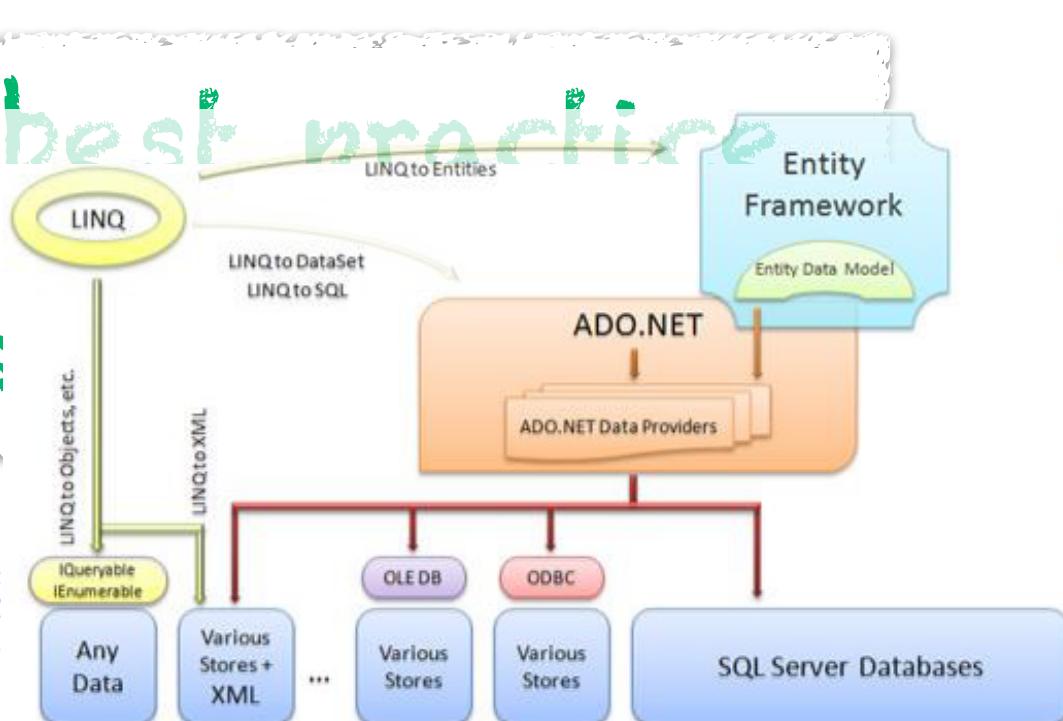
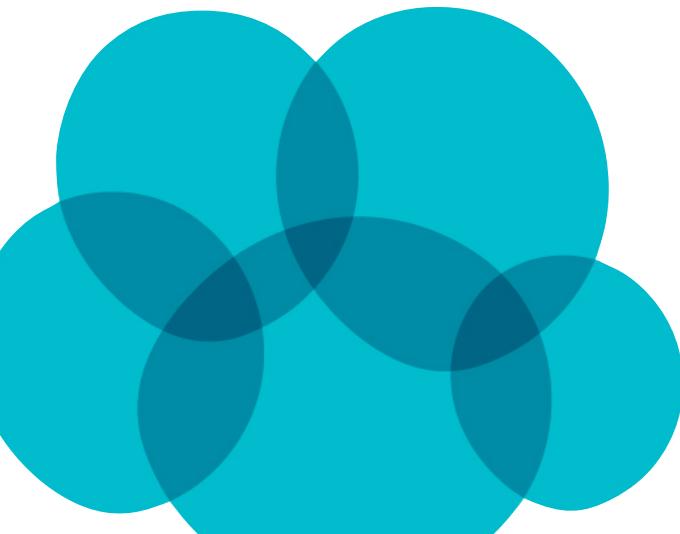


Yesterday's best practice

is tomorrow's

2007

2008





Yesterday's best practice
is tomorrow's anti-pattern.

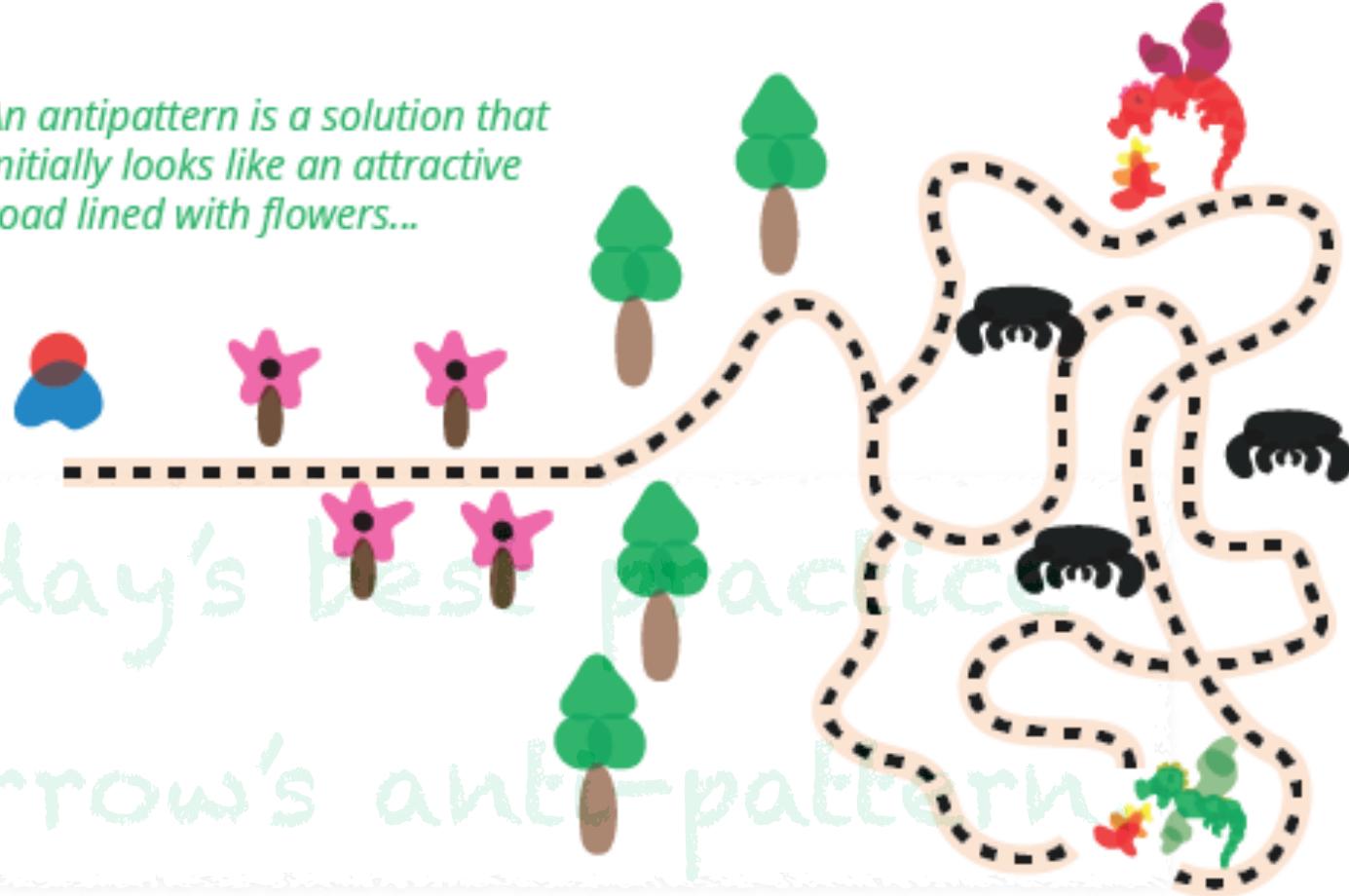


ENTERPRISE
Java Beans



An antipattern is a solution that initially looks like an attractive road lined with flowers...

Yesterday's best practices
is tomorrow's anti-pattern



*...but further on leads you into
a maze filled with monsters*

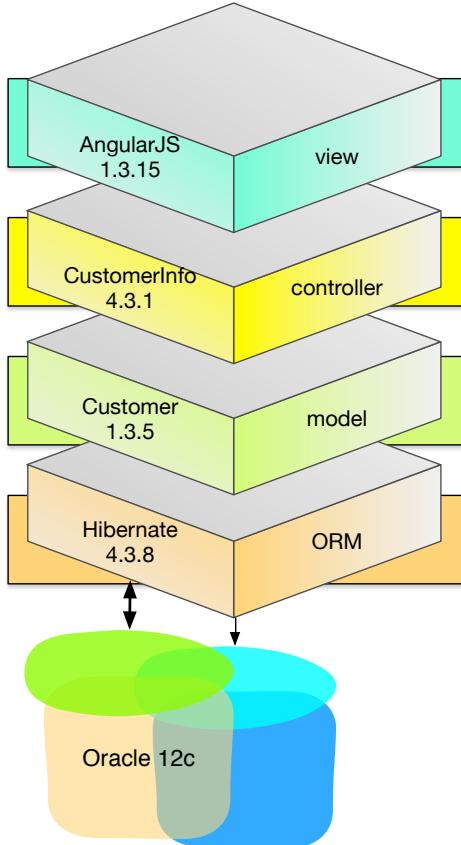


Architecture is abstract
until operationalized.

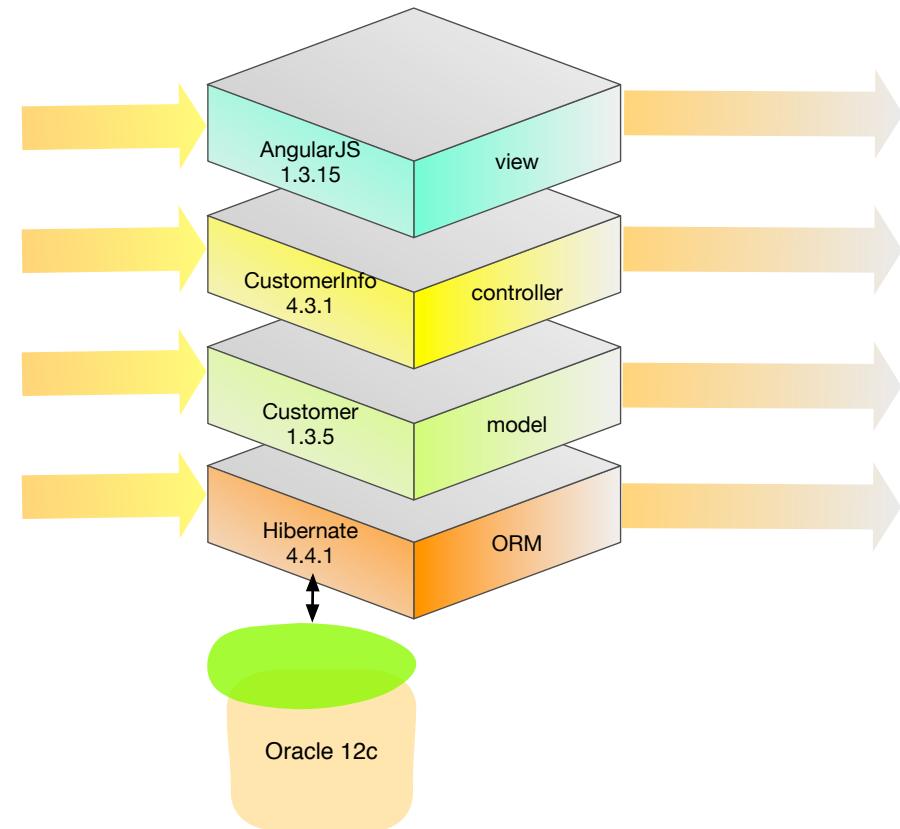
nealford.com/memeagora/2015/03/30/architecture_is_abstract_until_operationalized.html



Architecture is abstract until operationalized.

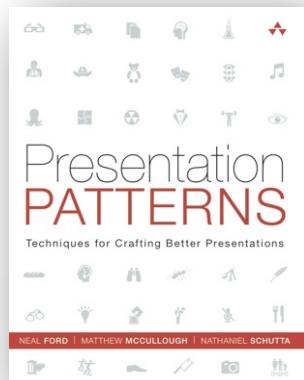


2D



3D

4D



nealford.com



@neal4d

ThoughtWorks®

NEAL FORD

Director / Software Architect / Meme Wrangler

