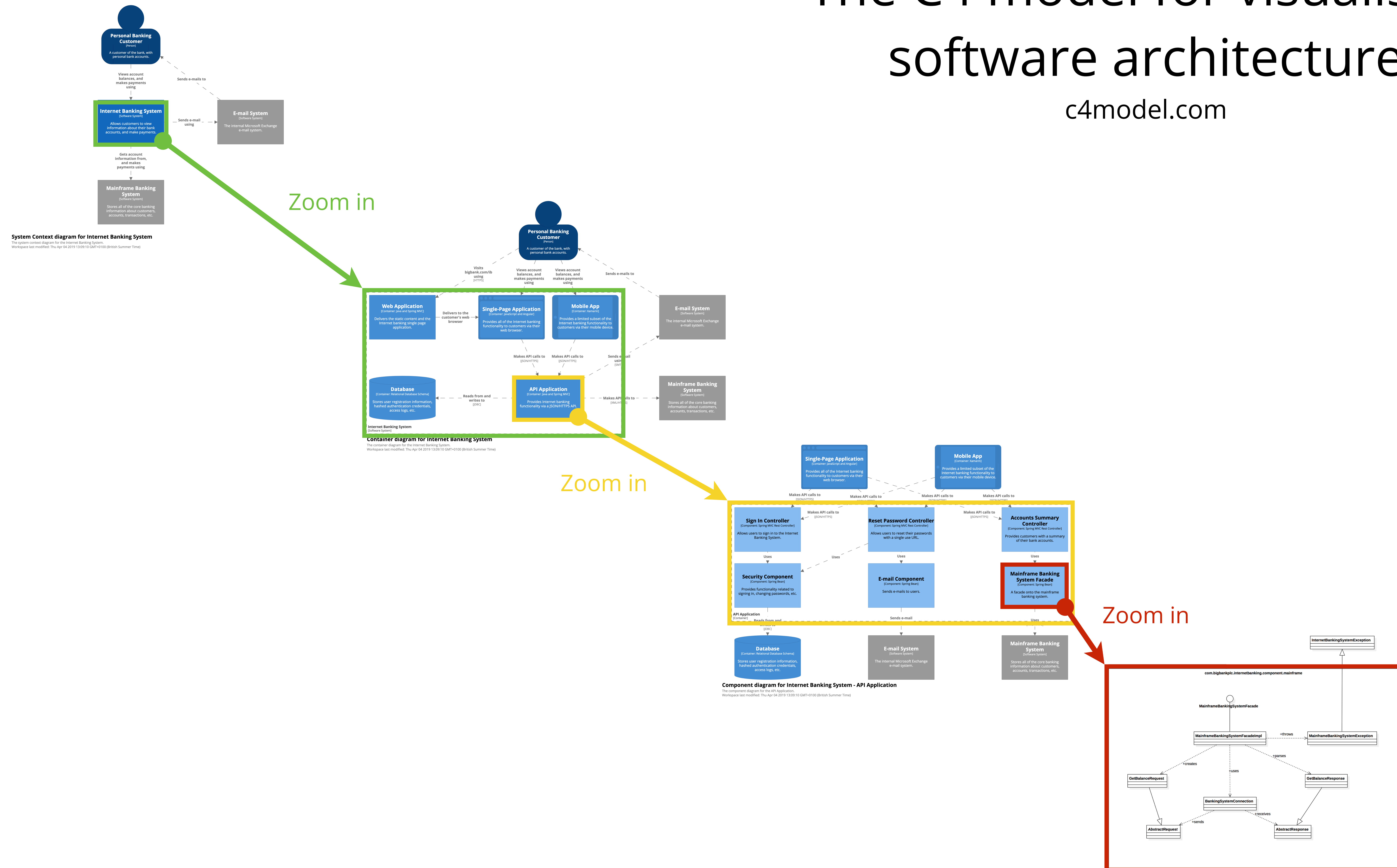
Modular monoliths

Simon Brown

The C4 model for visualising software architecture

c4model.com



Level 1	Level 2	Level 3	Level 4
Context	Containers	Components	Code

A well structured codebase
is easy to **visualise**

Context diagram

(level 1)

Container diagram

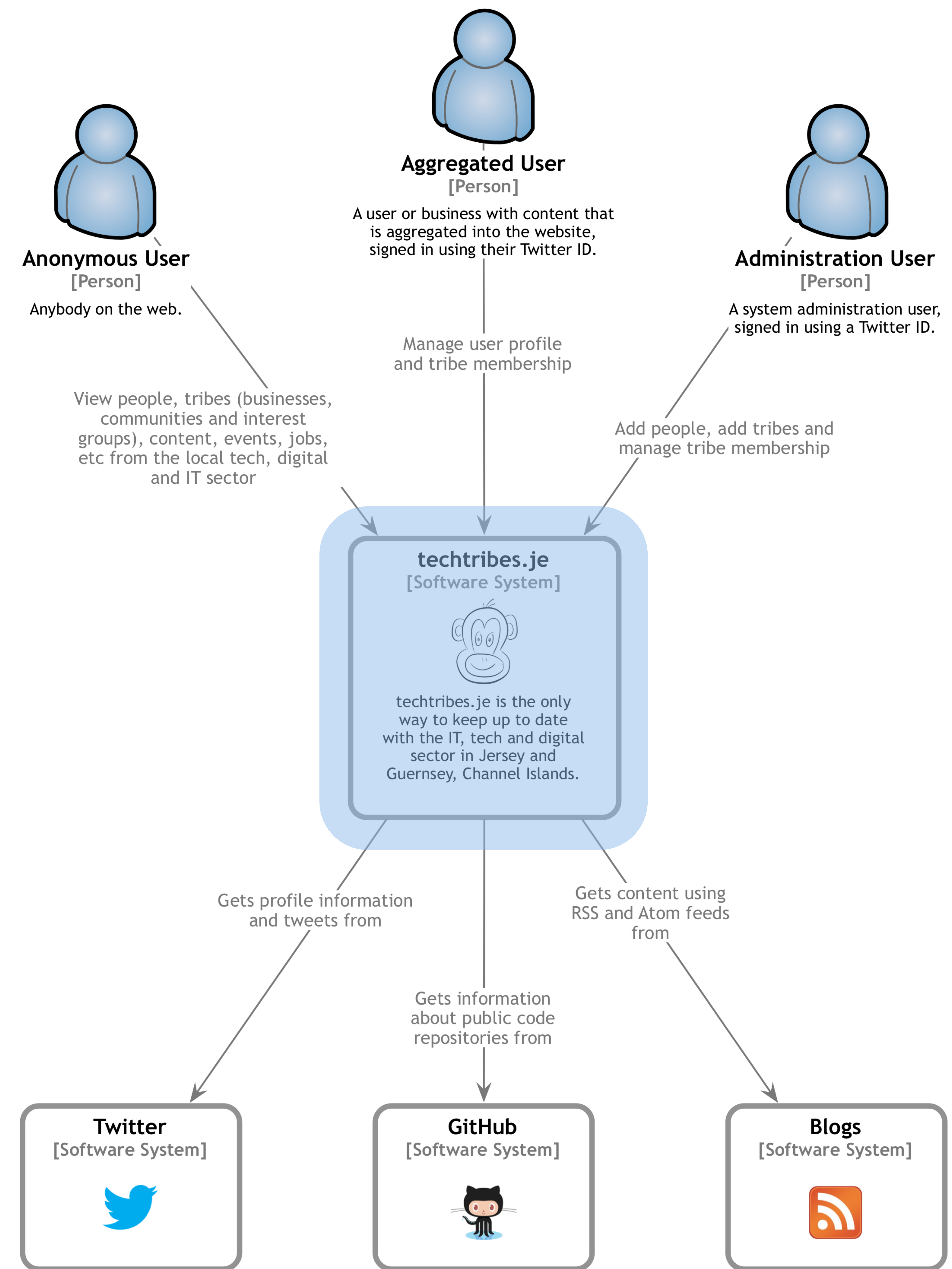
(level 2)

Component diagram

(level 3)

Class diagram

(level 4)



Context diagram

(level 1)

Container diagram

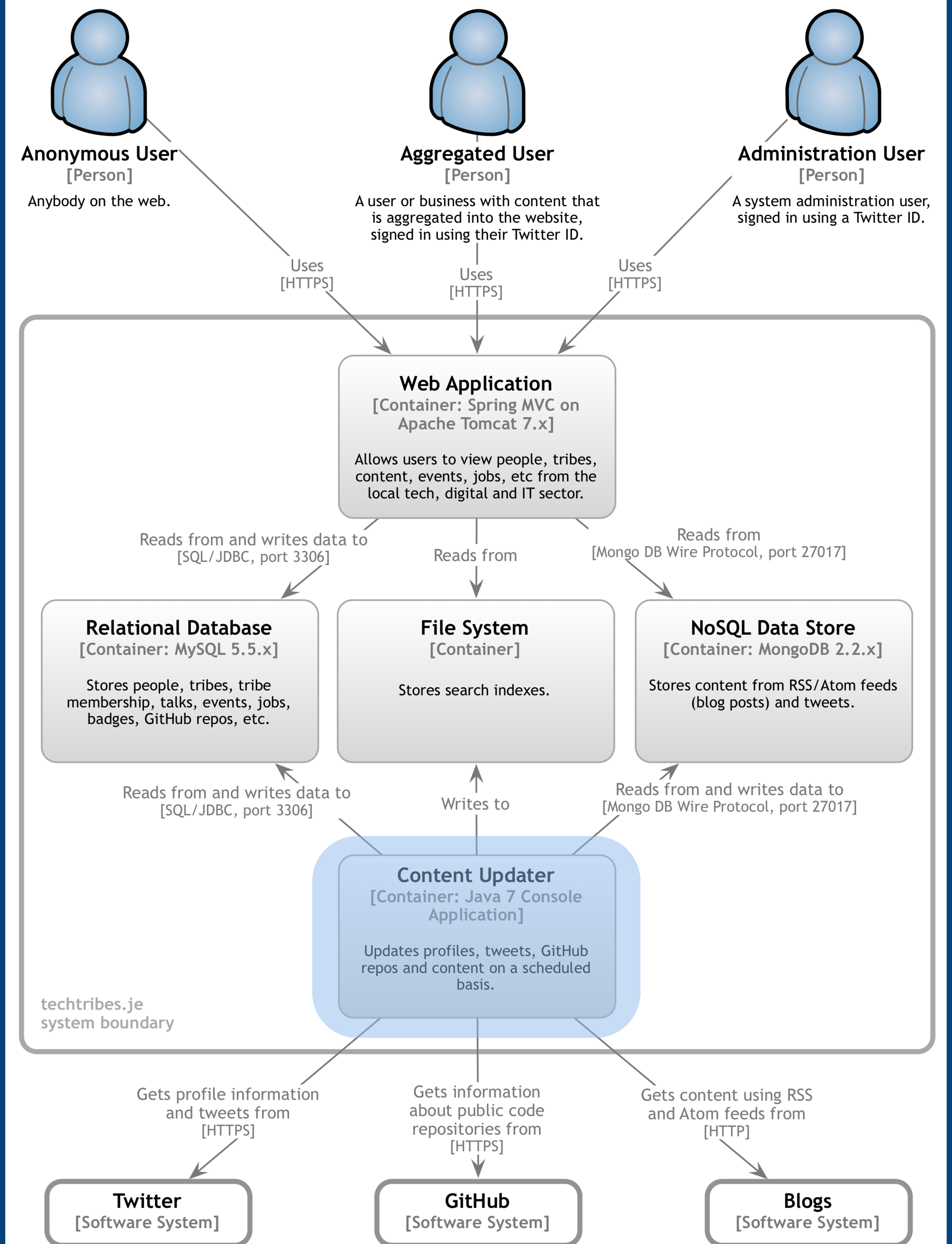
(level 2)

Component diagram

(level 3)

Class diagram

(level 4)



Context diagram

(level 1)

Container diagram

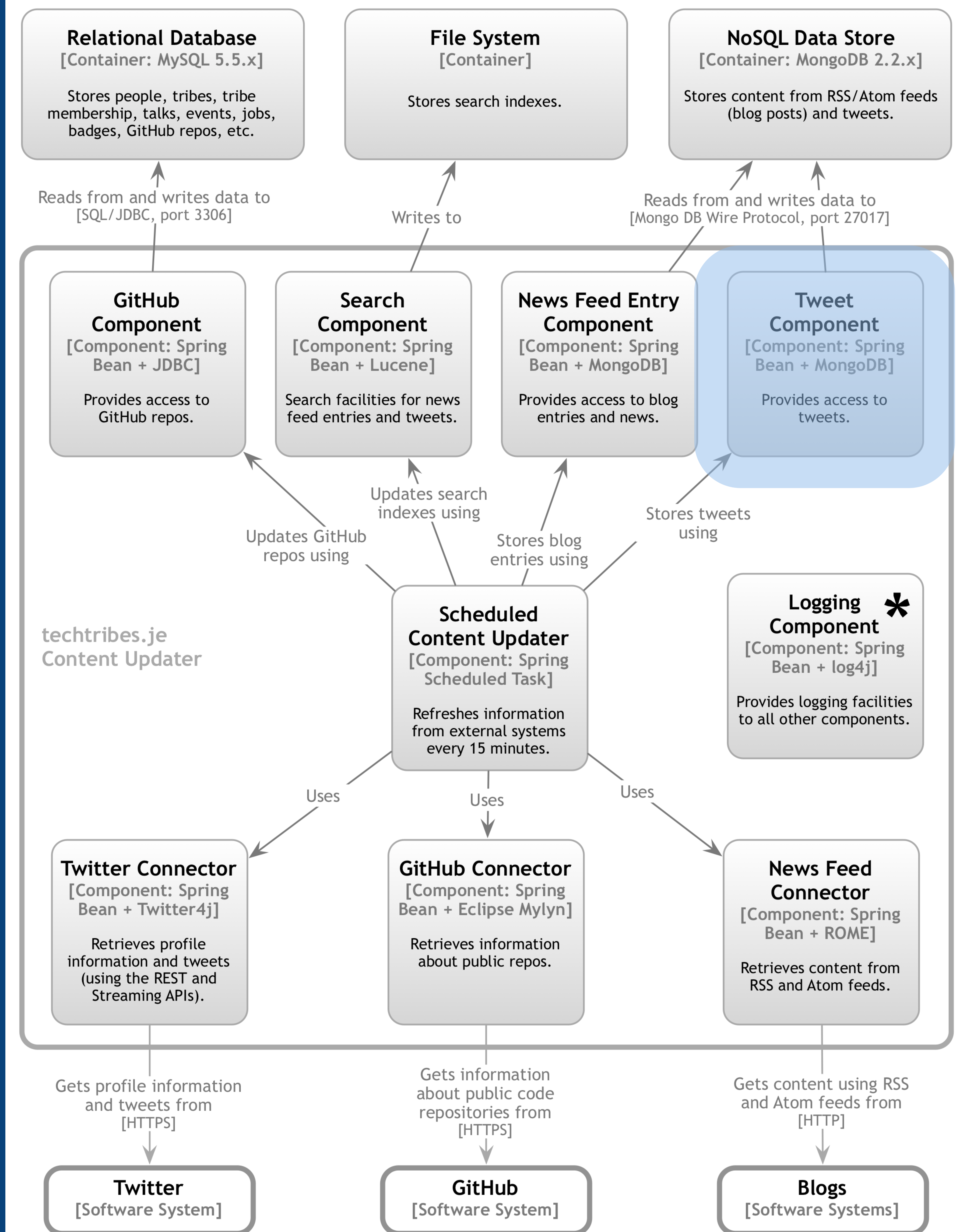
(level 2)

Component diagram

(level 3)

Class diagram

(level 4)



Context diagram

(level 1)

Container diagram

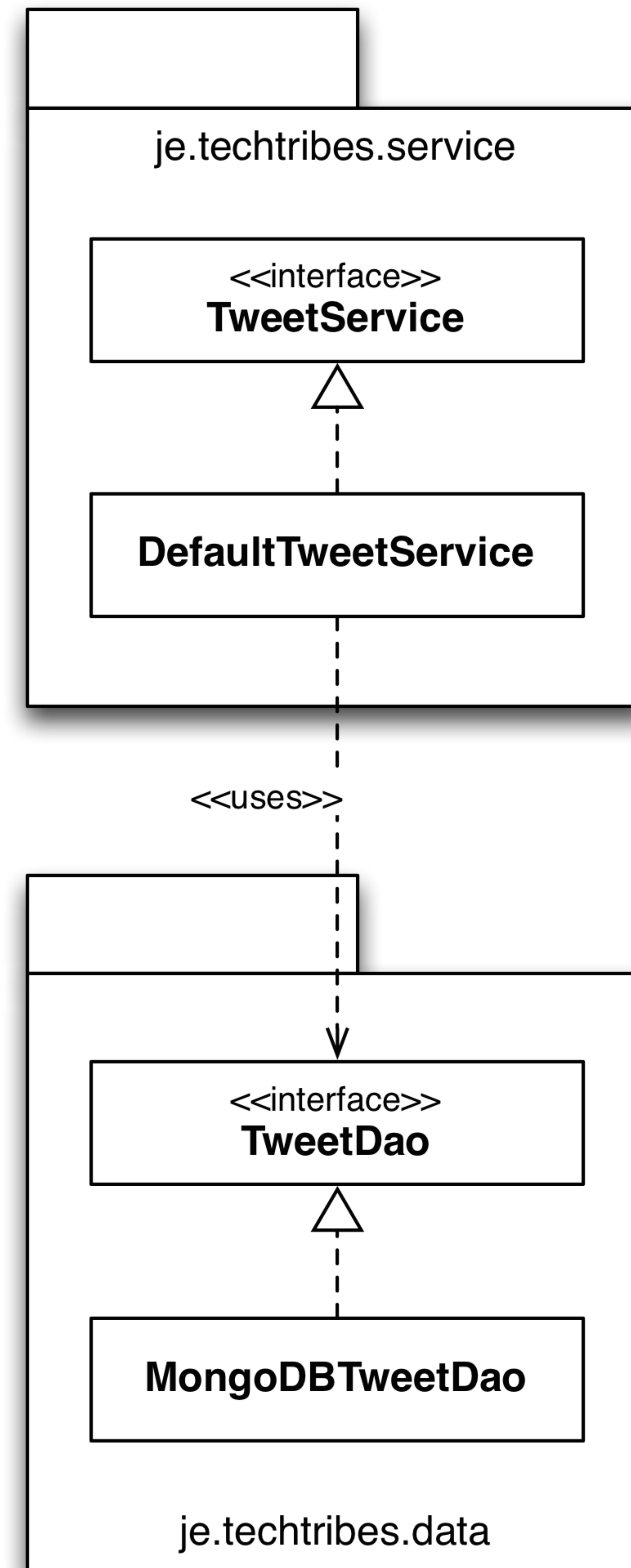
(level 2)

Component diagram

(level 3)

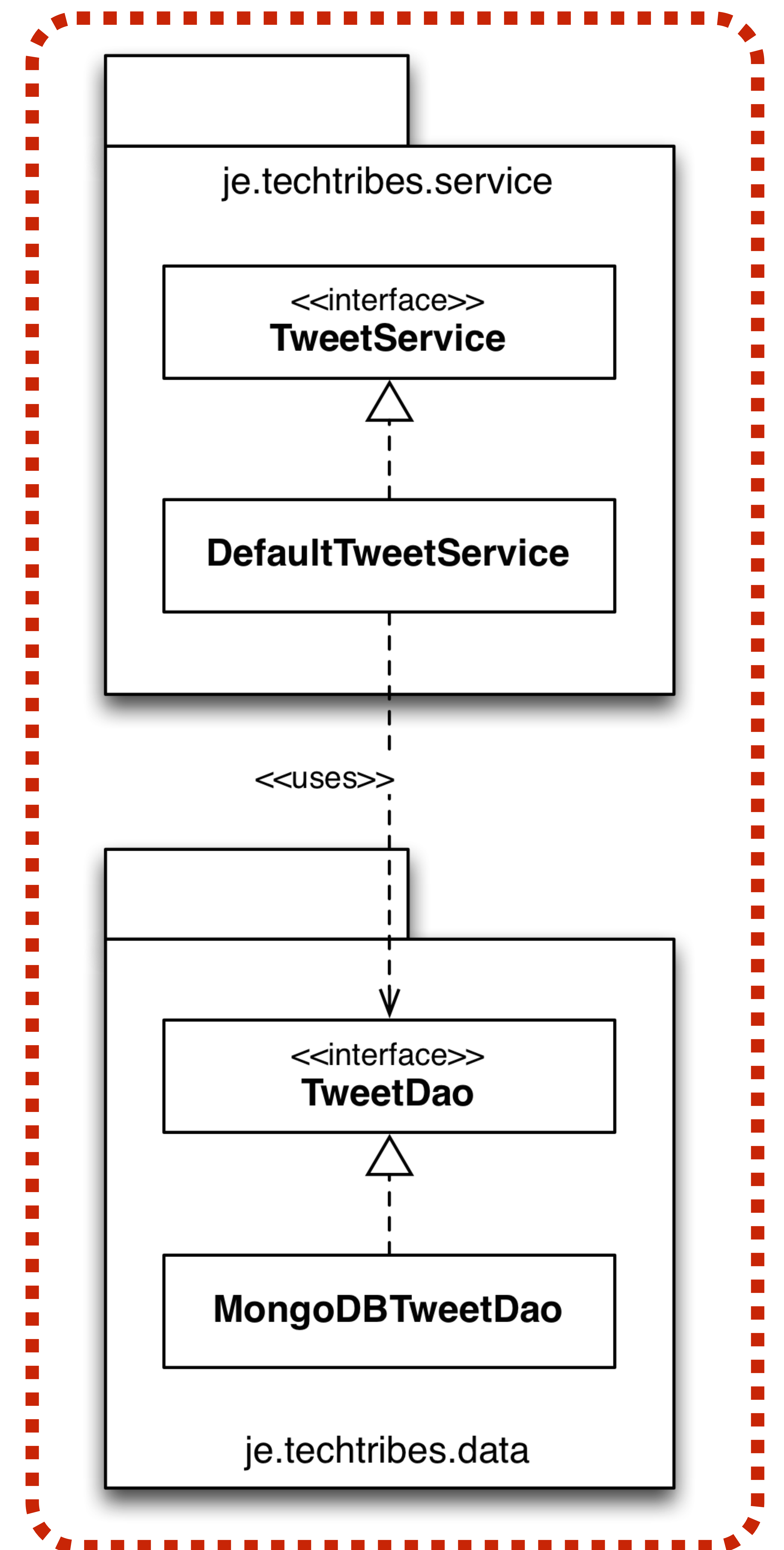
Code diagram

(level 4)



Where’s my “component”?

(the “Tweet Component” doesn’t exist as a single thing; it’s a combination of interfaces and classes across a layered architecture)



“the component exists **conceptually**”

Abstractions should reflect the code

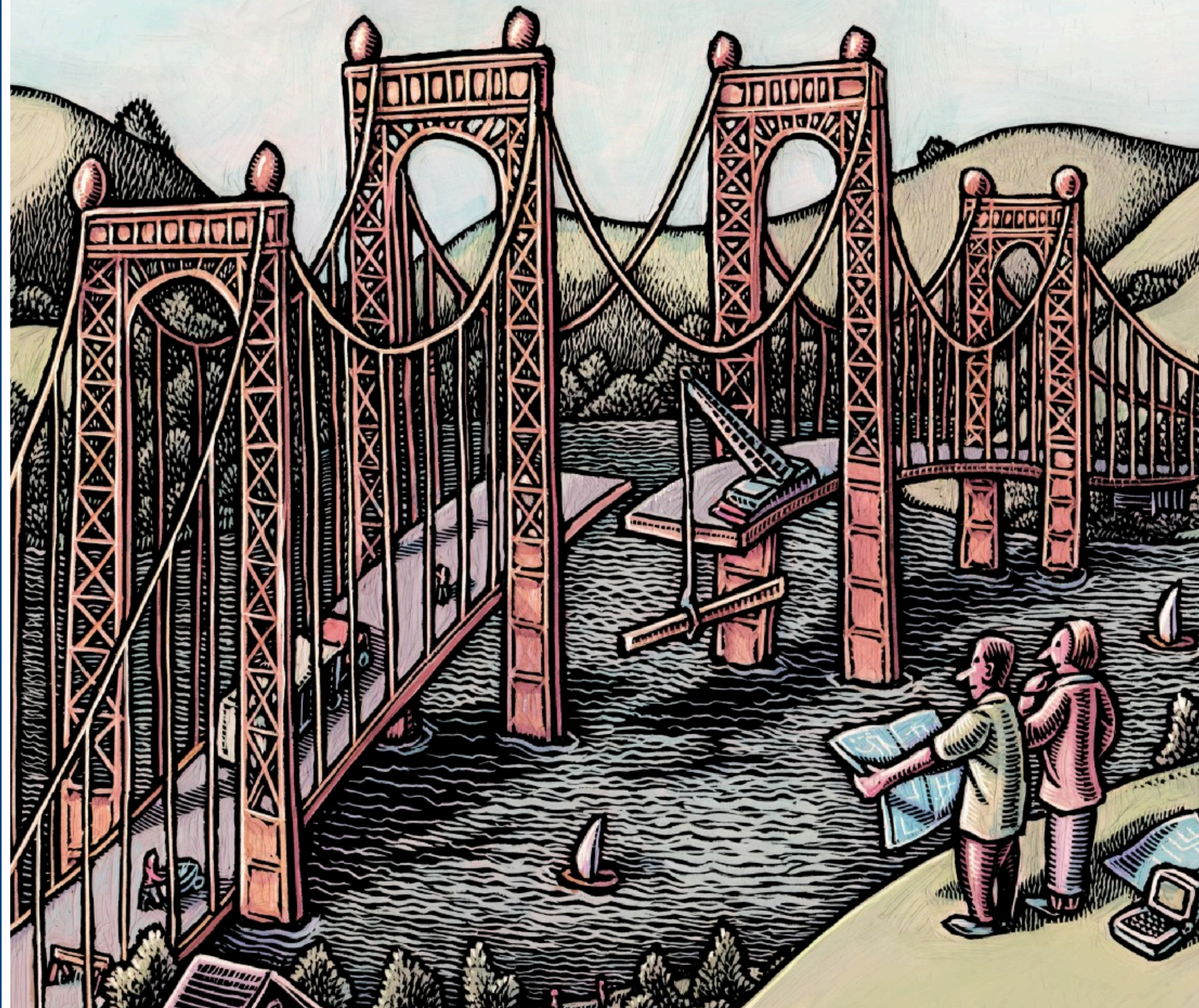
Model-code gap. Your architecture models and your source code will not show the same things. The difference between them is the *model-code gap*. Your architecture models include some abstract concepts, like components, that your programming language does not, but could. Beyond that, architecture models include intensional elements, like design decisions and constraints, that cannot be expressed in procedural source code at all.

Consequently, the relationship between the architecture model and source code is complicated. It is mostly a refinement relationship, where the extensional elements in the architecture model are refined into extensional elements in source code. This is shown in Figure 10.3. However, intensional elements are not refined into corresponding elements in source code.

Upon learning about the model-code gap, your first instinct may be to avoid it. But reflecting on the origins of the gap gives little hope of a general solution in the short term: architecture models help you reason about complexity and scale because they are abstract and intensional; source code executes on machines because it is concrete and extensional.

“model-code gap”

“Our architecture diagrams don’t match the code.”



Model-code gap. Your architecture models and your source code will not show the same things. The difference between them is the *model-code gap*. Your architecture models include some abstract concepts, like components, that your programming language does not, but could. Beyond that, architecture models include intensional elements, like design decisions and constraints, that cannot be expressed in procedural source code at all.

Consequently, the relationship between the architecture model and source code is complicated. It is mostly a refinement relationship, where the extensional elements in the architecture model are refined into extensional elements in source code. This is shown in Figure 10.3. However, intensional elements are not refined into corresponding elements in source code.

Upon learning about the model-code gap, your first instinct may be to avoid it. But reflecting on the origins of the gap gives little hope of a general solution in the short term: architecture models help you reason about complexity and scale because they are abstract and intensional; source code executes on machines because it is concrete and extensional.

“architecturally-evident coding style”

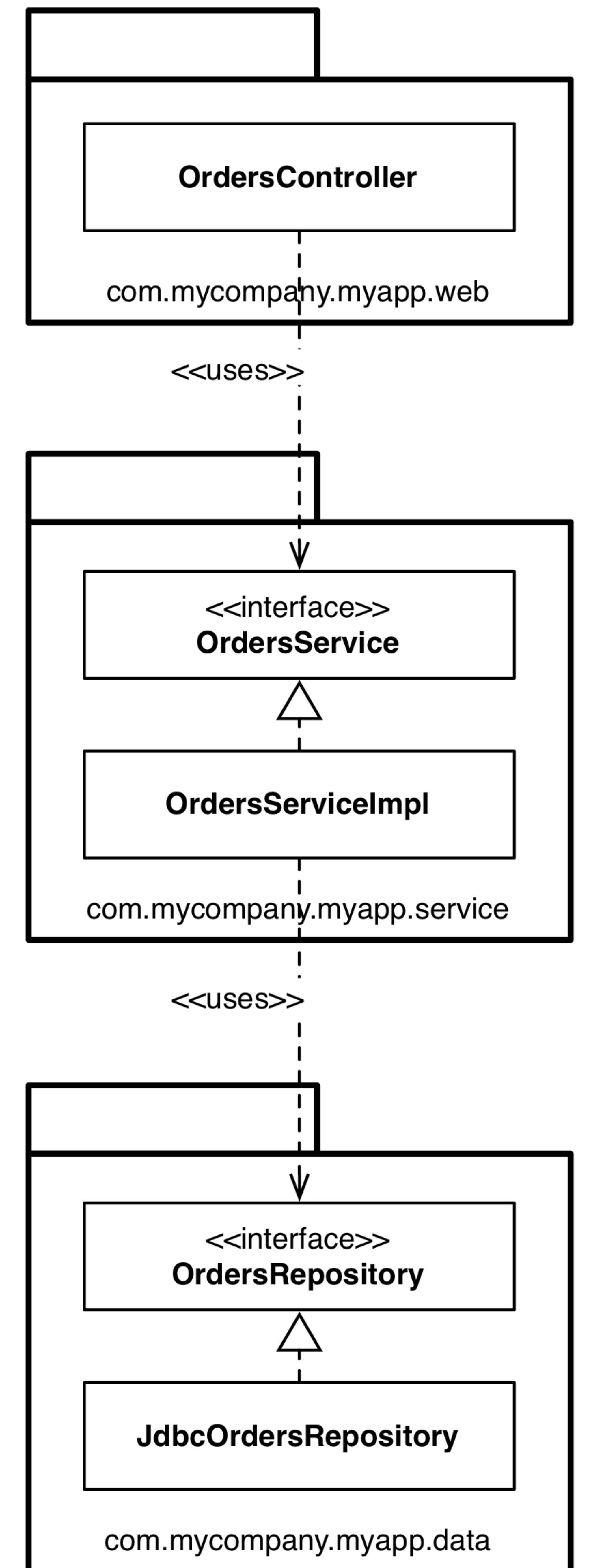
The code structure should reflect the architectural intent

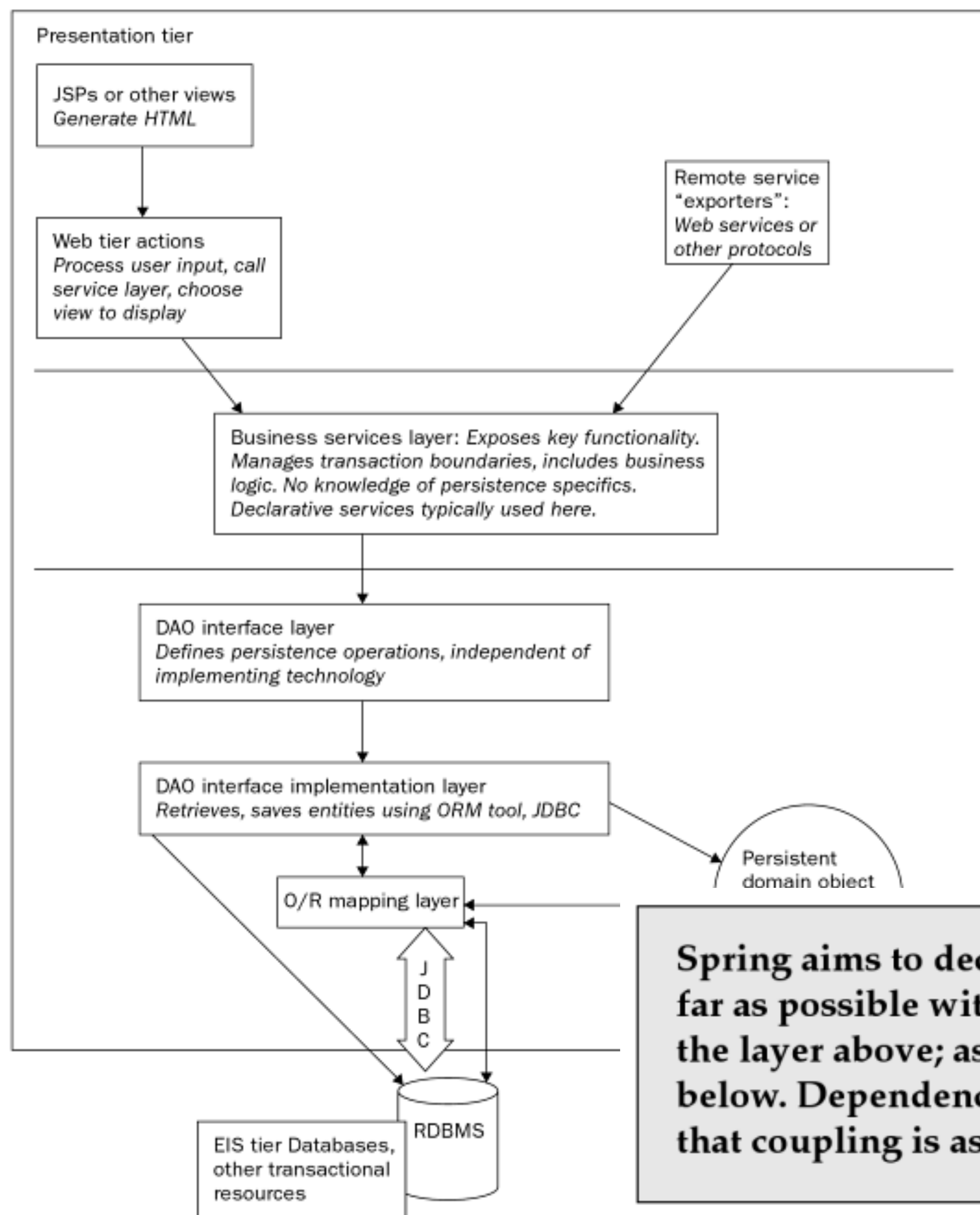
Package by layer

Organise code based upon what the code does from a technical perspective

Package by layer

is a “**horizontal**” slicing


OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
com.mycompany.myapp.service
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.data



Let's summarize each layer and its responsibilities, beginning closest to the database or other enterprise resources:

- **Presentation layer:** This is most likely to be a web tier. This layer should be as thin as possible. It should be possible to have alternative presentation layers — such as a web tier or remote web services facade — on a single, well-designed middle tier.
- **Business services layer:** This is responsible for transactional boundaries and providing an entry point for operations on the system as a whole. This layer should have no knowledge of presentation concerns, and should be reusable.
- **DAO interface layer:** This is a layer of interfaces *independent of any data access technology* that is used to find and persist persistent objects. This layer effectively consists of *Strategy* interfaces for the Business services layer. This layer should not contain business logic. Implementations of these interfaces will normally use an O/R mapping technology or Spring's JDBC abstraction.
- **Persistent domain objects:** These model real objects or concepts such as a bank account.
- **Databases and legacy systems:** By far the most common case is a single RDBMS. However, there may be multiple databases, or a mix of databases and other transactional or non-transactional legacy systems or other enterprise resources. The same fundamental architecture is applicable in either case. This is often referred to as the *EIS* (*Enterprise Information System*) tier.

In a J2EE application, all layers except the EIS tier will run in the application server or web container. Domain objects will typically be passed up to the presentation layer, which will display data they contain, *but not modify them*, which will occur only within the transactional boundaries defined by the business services layer. Thus there is no need for distinct Transfer Objects, as used in traditional J2EE architecture.

In the following sections we'll discuss each of these layers in turn, beginning closest to the database.

Spring aims to decouple architectural layers, so that each layer can be modified as far as possible without impacting other layers. No layer is aware of the concerns of the layer above; as far as possible, dependency is purely on the layer immediately below. Dependency between layers is normally in the form of interfaces, ensuring that coupling is as loose as possible.

Spring aims to decouple architectural layers, so that each layer can be modified as far as possible without impacting other layers. No layer is aware of the concerns of the layer above; as far as possible, dependency is purely on the layer immediately below. Dependency between layers is normally in the form of interfaces, ensuring that coupling is as loose as possible.

Also sample codebases, starter projects, demos at conferences, etc...

Cargo cult programming can also refer to the results of applying a design pattern or coding style blindly without understanding the reasons behind that design principle.

https://en.wikipedia.org/wiki/Cargo_cult_programming

Screaming Architecture

Uncle Bob / 30 Sep 2011 Architecture

Imagine that you are looking at the blueprints of a building. This document, prepared by an architect, tells you the plans for the building. What do these plans tell you?

If the plans you are looking at are for a single family residence, then you'll likely see a front entrance, a foyer leading to a living room and perhaps a dining room. There'll likely be a kitchen a short distance away, close to the dining room. Perhaps a dinette area next to the kitchen, and probably a family room close to that. As you looked at those plans, there'd be no question that you were looking at a *house*. The architecture would *scream*: **house**.

Or if you were looking at the architecture of a library, you'd likely see a grand entrance, an area for check-in-out clerks, reading areas, small conference rooms, and gallery after gallery capable of holding bookshelves for all the books in the library. That architecture would *scream*: **Library**.

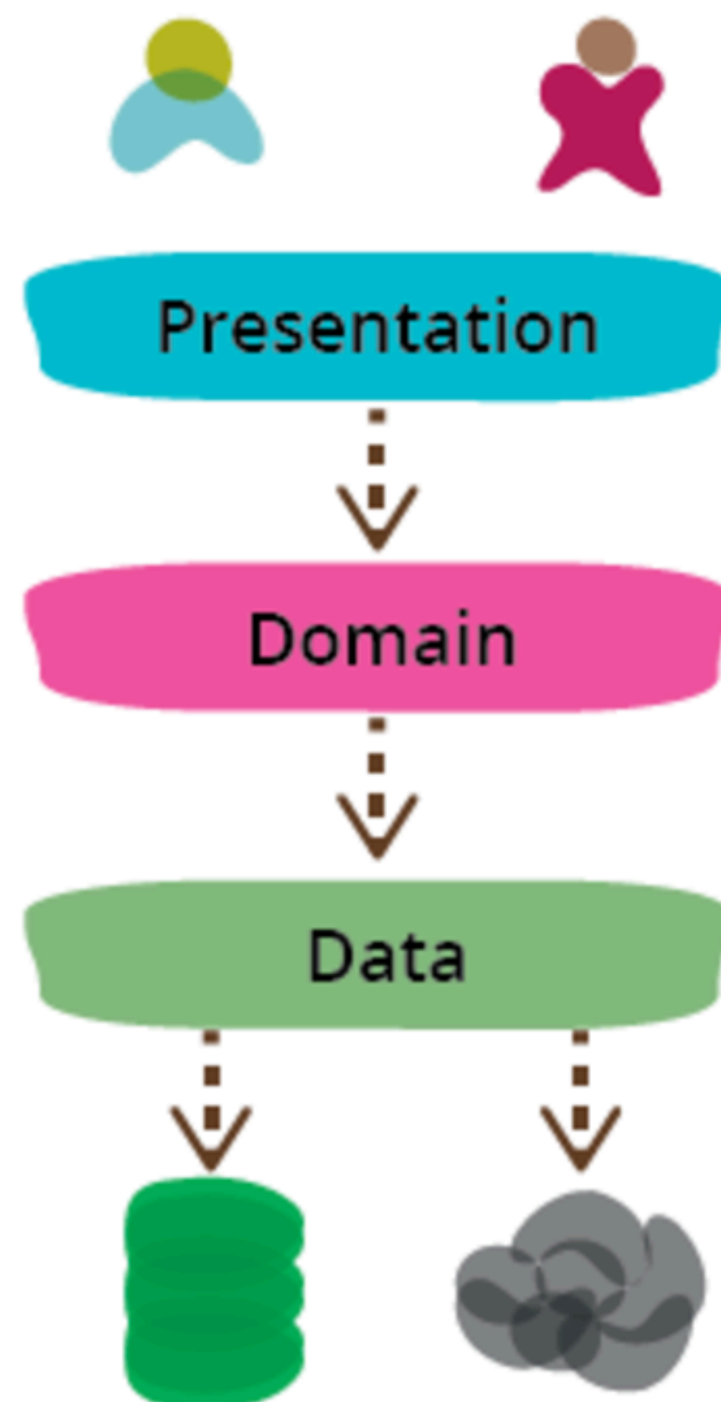
So what does the architecture of your application scream? When you look at the top level directory structure, and the source files in the highest level package; do they scream: **Health Care System**, or **Accounting System**, or **Inventory Management System**? Or do they scream: **Rails**, or **Spring/Hibernate**, or **ASP**?

PresentationDomainDataLayering

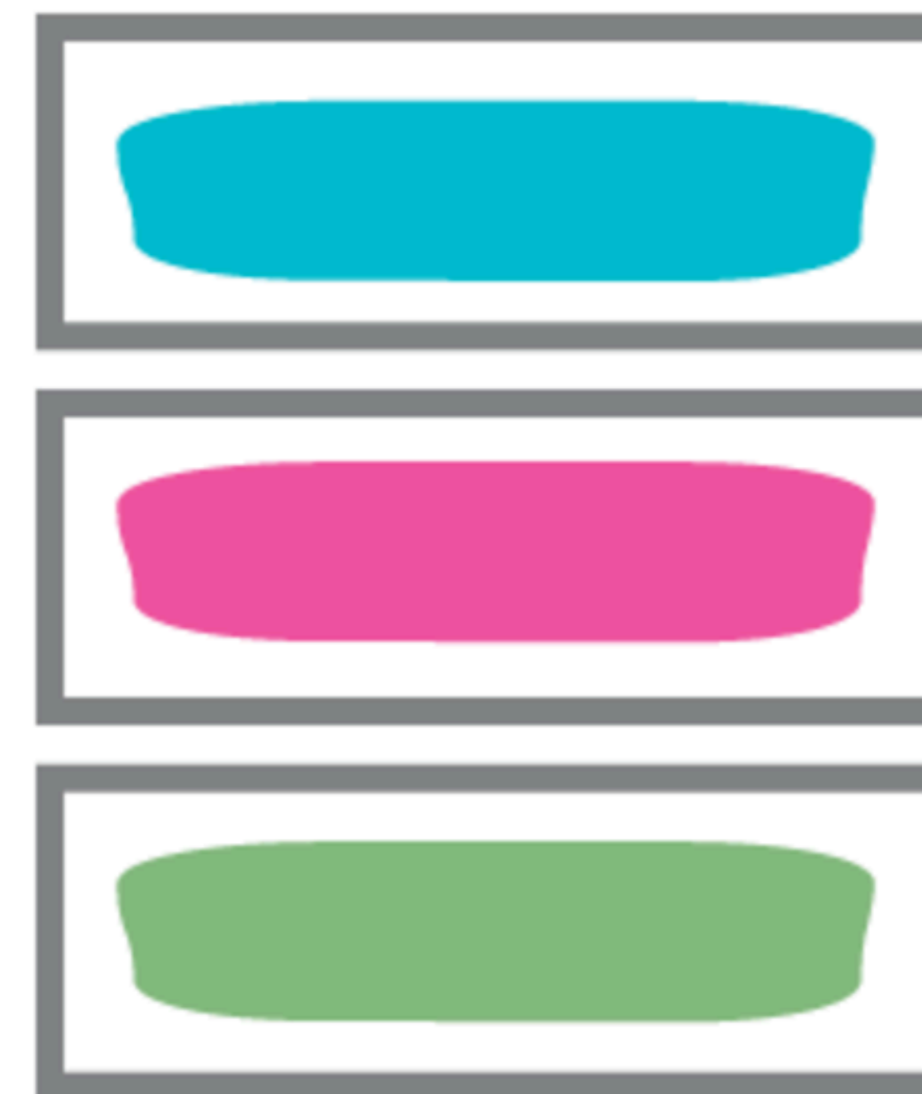
Martin Fowler

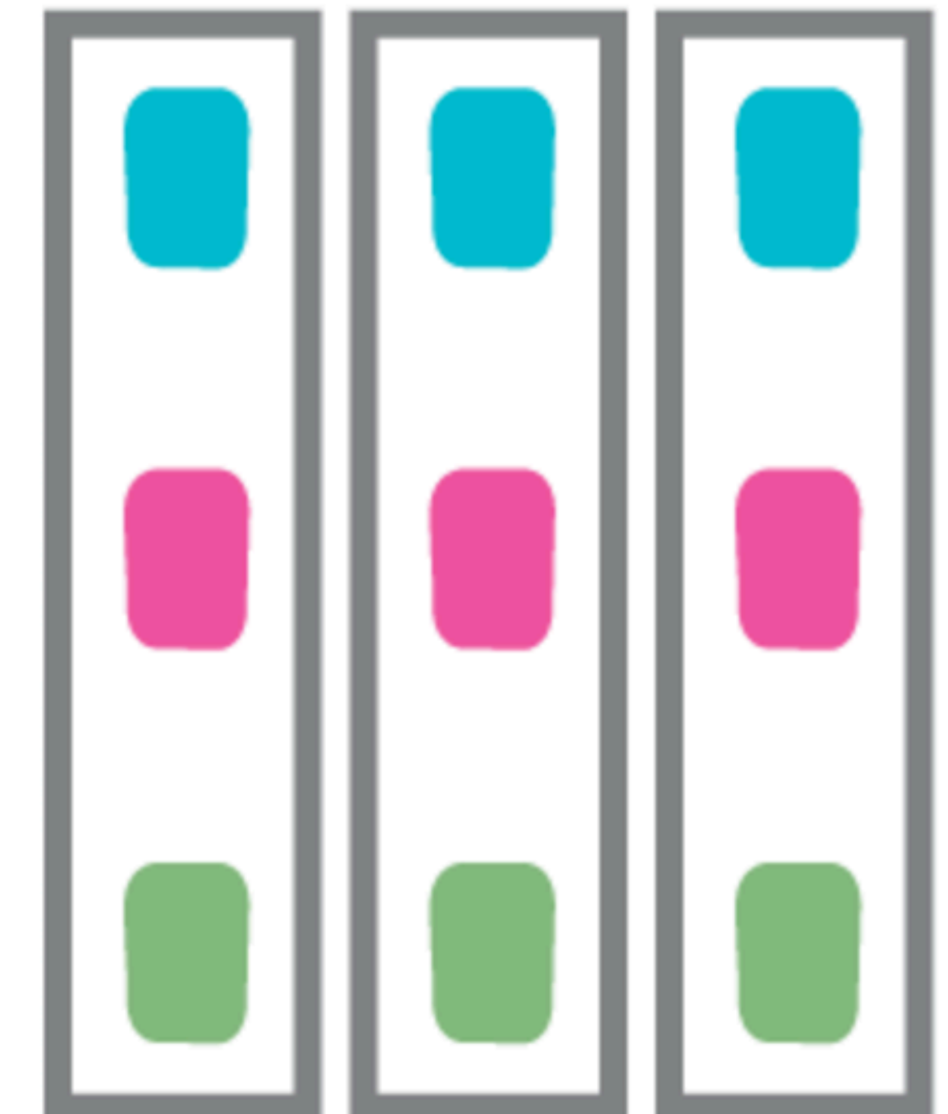
26 August 2015

One of the most common ways to modularize an information-rich program is to separate it into three broad layers: presentation (UI), domain logic (aka business logic), and data access. So you often see web applications divided into a web layer that knows about handling http requests and rendering HTML, a business logic layer that contains validations and calculations, and a data access layer that sorts out how to manage persistant data in a database or remote services.



Although presentation-domain-data separation is a common approach, it should only be applied at a relatively small granularity. As an application grows, each layer can get sufficiently complex on its own that you need to modularize further. When this happens it's usually not best to use presentation-domain-data as the higher level of modules. Often frameworks encourage you to have something like view-model-data as the top level namespaces; that's ok for smaller systems, but once any of these layers gets too big you should split your top level into domain oriented modules which are internally layered.





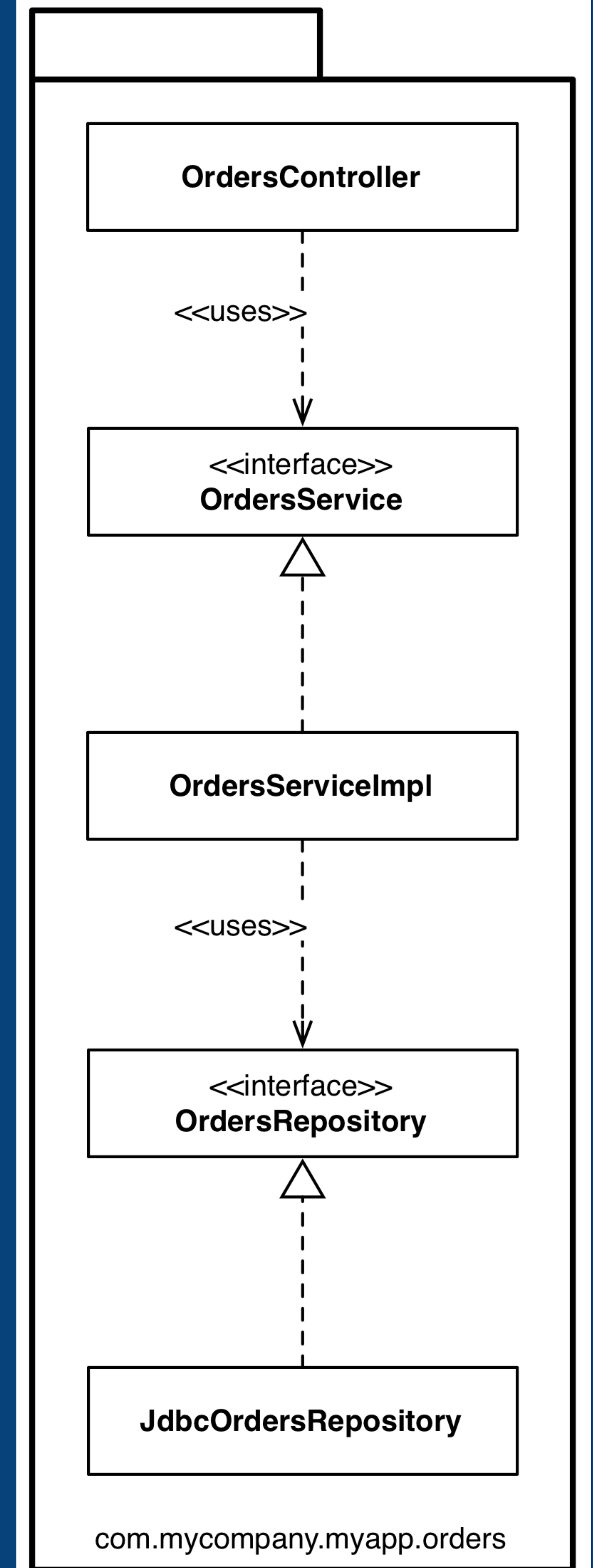
Changes to a layered architecture
usually result in changes
across all layers

Package by feature

Organise code based upon
what the code does from
a functional perspective

Features, domain concepts, aggregate roots, etc

Package by feature
is a “**vertical**” slicing


OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders

Cited benefits include higher cohesion, lower coupling, and related code is easier to find

Web Application

Business/domain

Relational Database

Web Application

Business/domain

Abstraction (e.g. ORM)

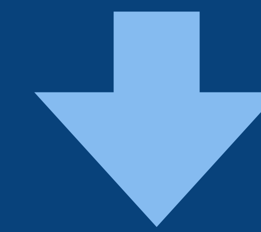
Relational Database

Web Application

Business/domain

Abstraction abstraction

Abstraction (e.g. ORM)



Relational Database

Ports and adapters, hexagonal, clean, onion, etc

Keep domain related code separate from technical details

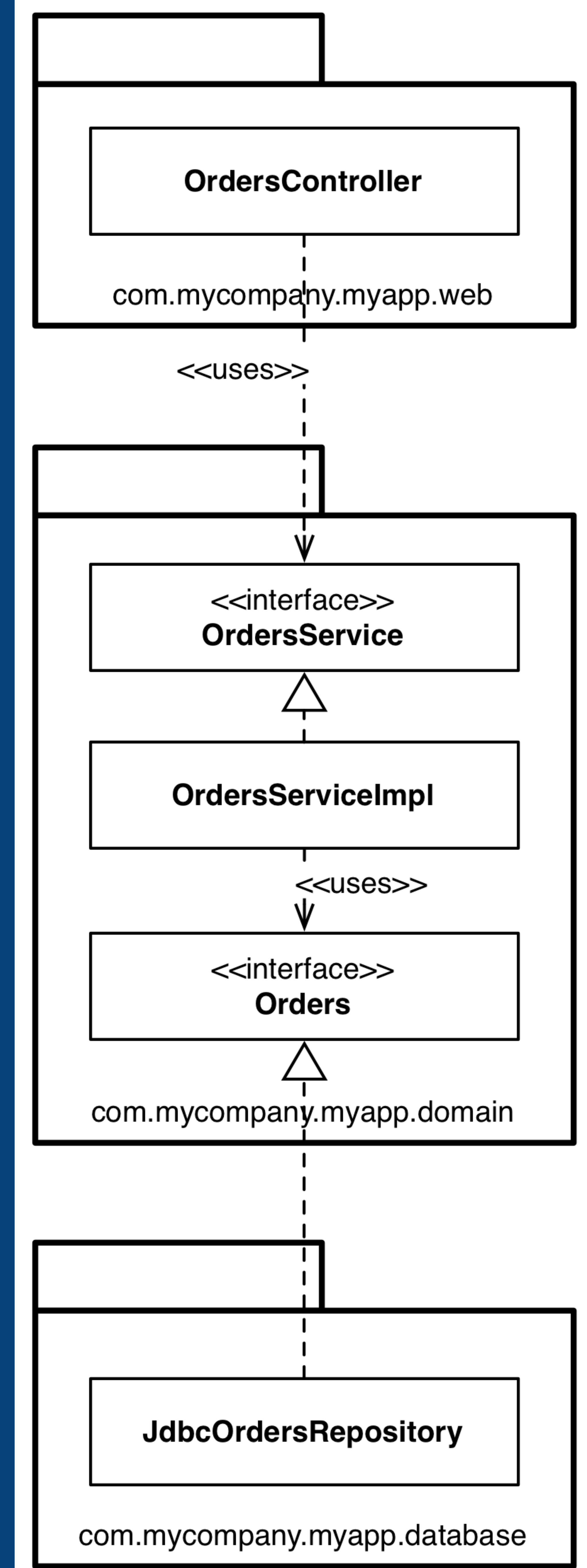
The “inside” is technology agnostic, and is often described in terms of a **ubiquitous language**

The “outside” is technology specific

The “outside” depends upon the “inside”

Infrastructure
(outside)

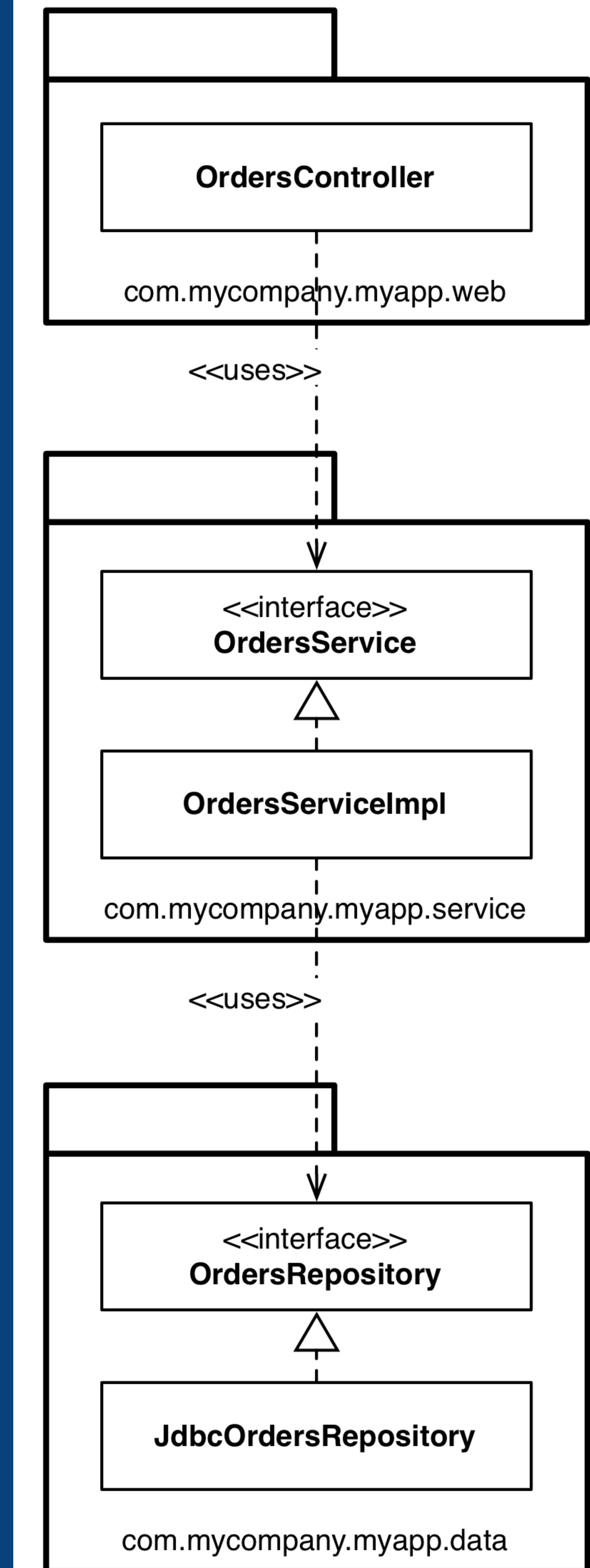
Domain
(inside)



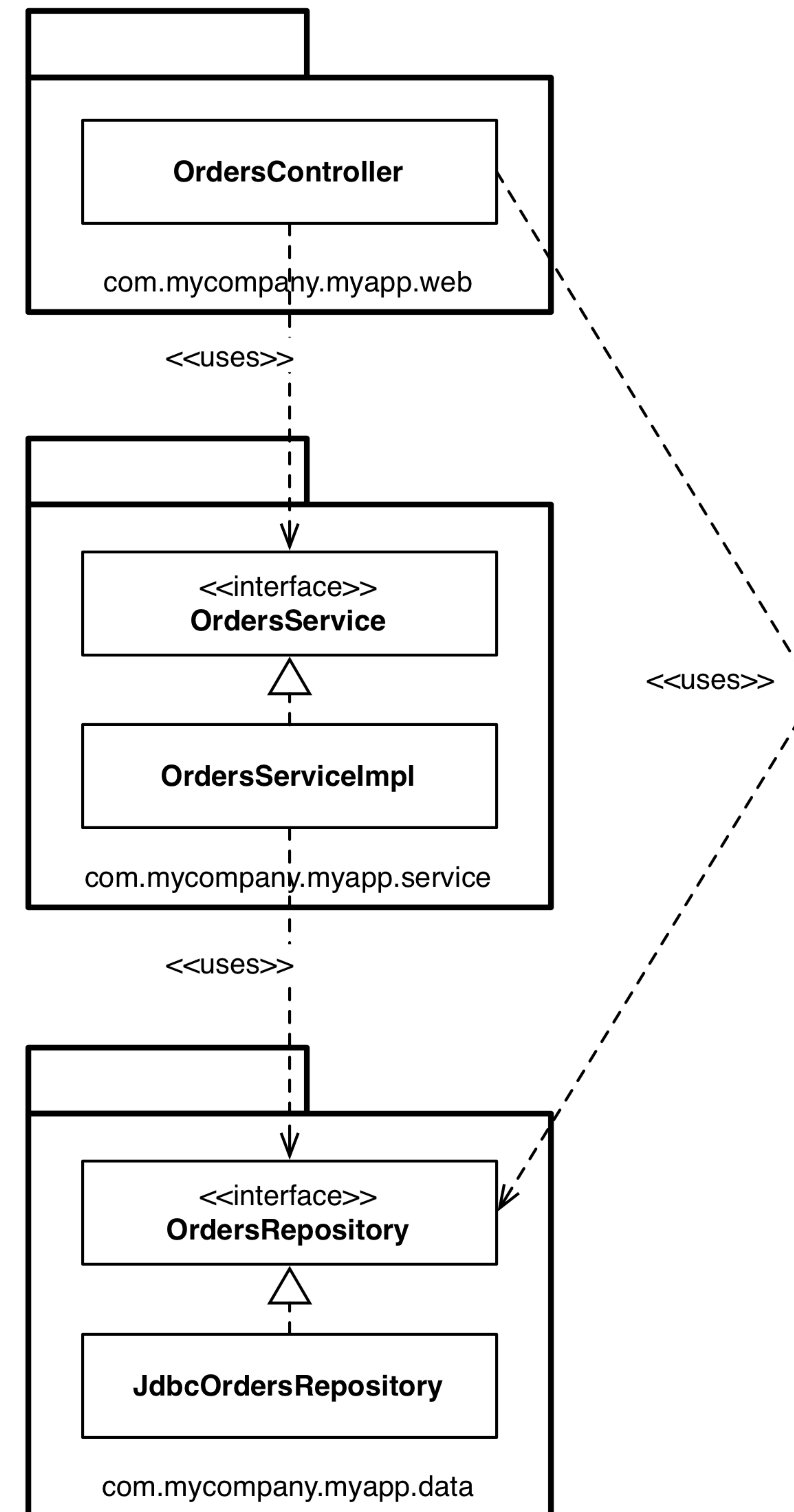
This approach
is also
“cargo culted”,
yet not all
frameworks
are equal

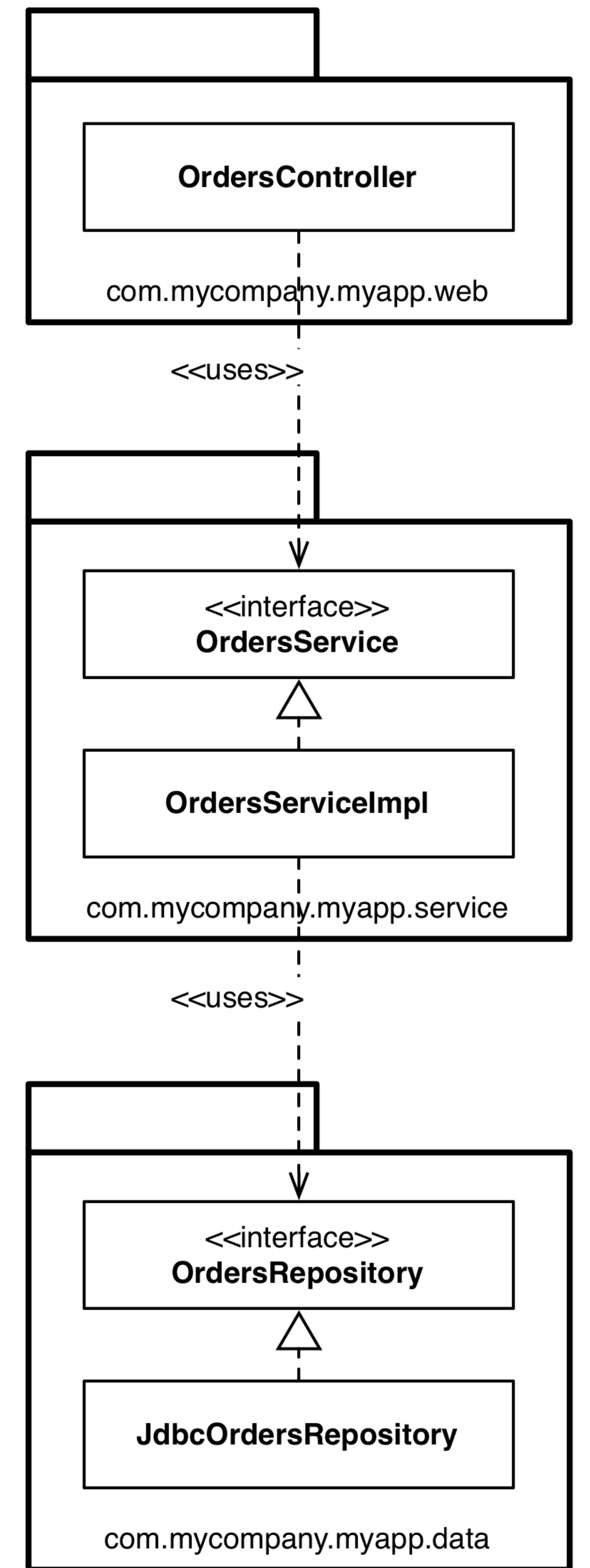
But...

Hi, can you add feature X to the orders functionality?



Sure!





OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
com.mycompany.myapp.service
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.data

> A big ball of mud is a casually, even haphazardly, structured system. Its organization, if one can call it that, is dictated more by expediency than design.

Big Ball of Mud

Brian Foote and Joseph Yoder

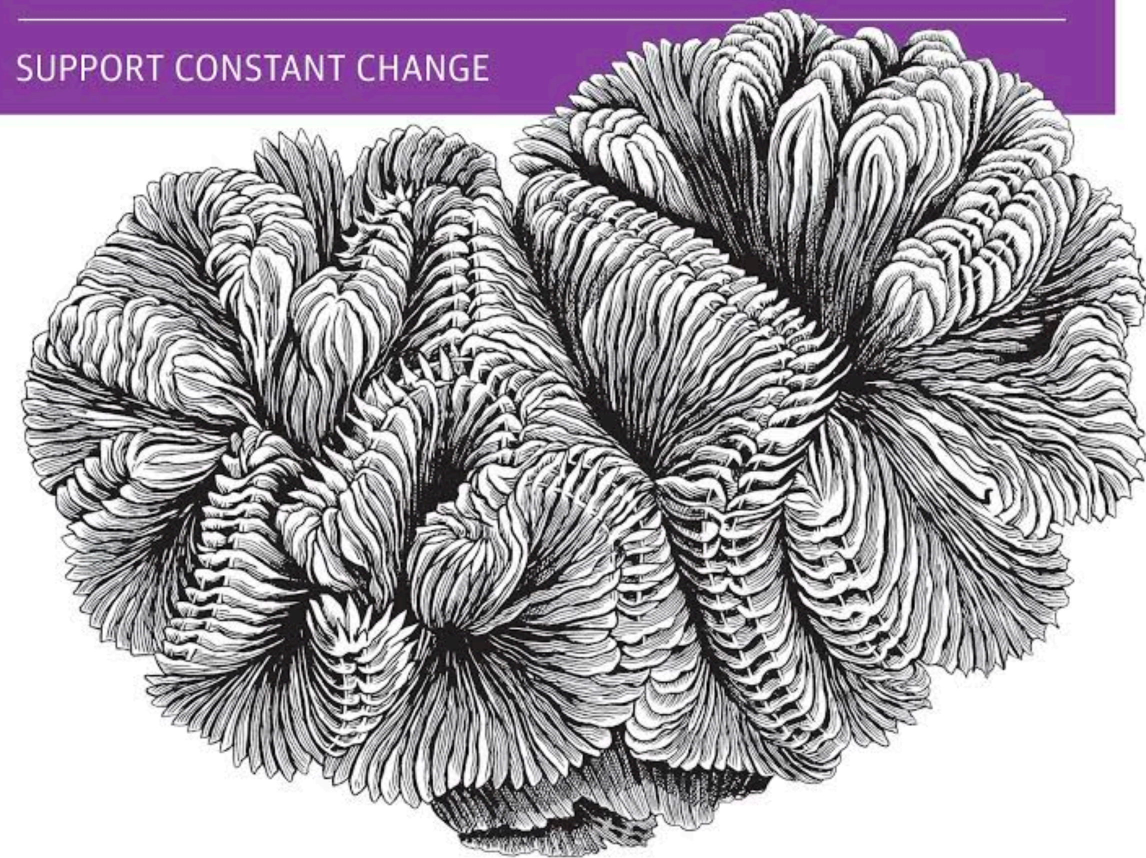
Architectural principles introduce consistency via constraints and guidelines

> web controllers should never access repositories directly

"we enforce this principle through good discipline and code reviews, because we trust our developers"

Responsible, professional software developers are still human :-)

It’s 2024! In a world of artificial intelligence and machine learning, why don’t we use **tools** to help us build “good” software?



“Fitness functions”

(e.g. cyclic complexity, coupling, etc)

Tooling?

Static analysis tools, architecture violation checking, etc

types in package `**/web` should not access types in `**/data`

Using tools to assert good code structure seems like a hack

But Java’s access modifiers are flawed…

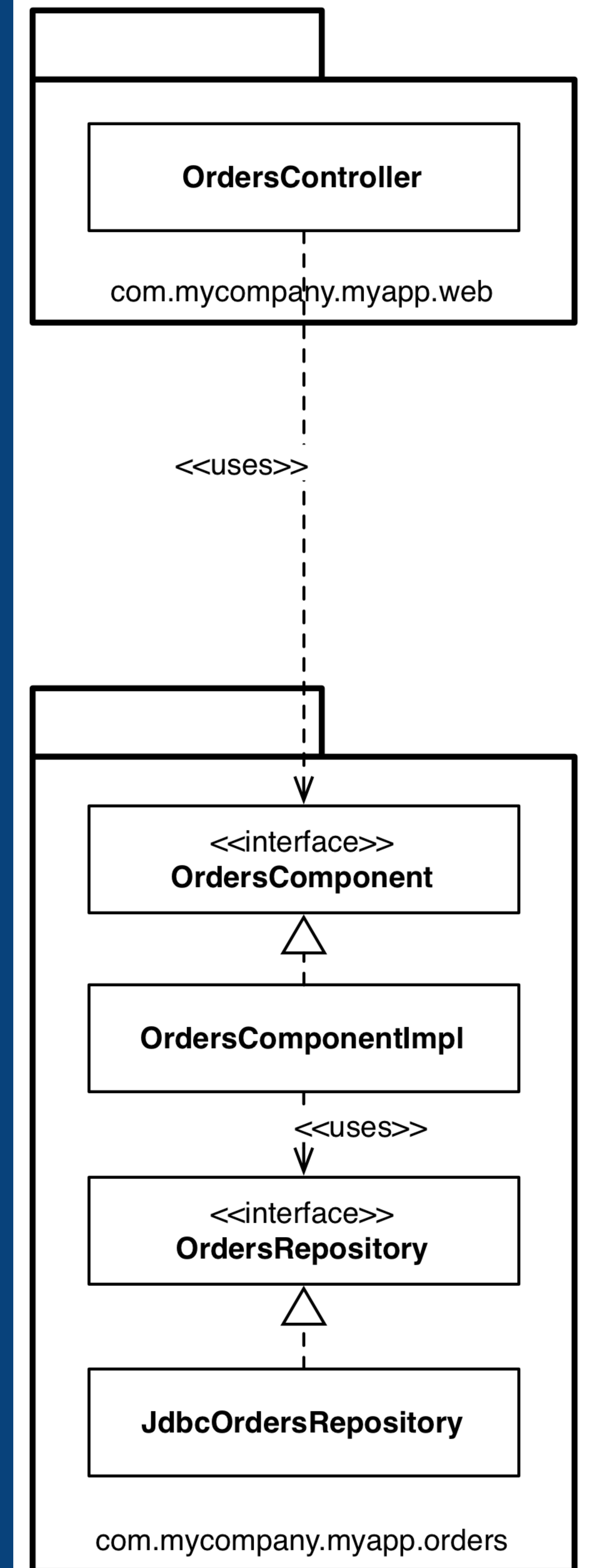
Package by component

Organise code by bundling together everything related to a “component”

Component?

a grouping of related functionality,
accessed via a well-defined interface,
residing inside an application (i.e. a C4 container)

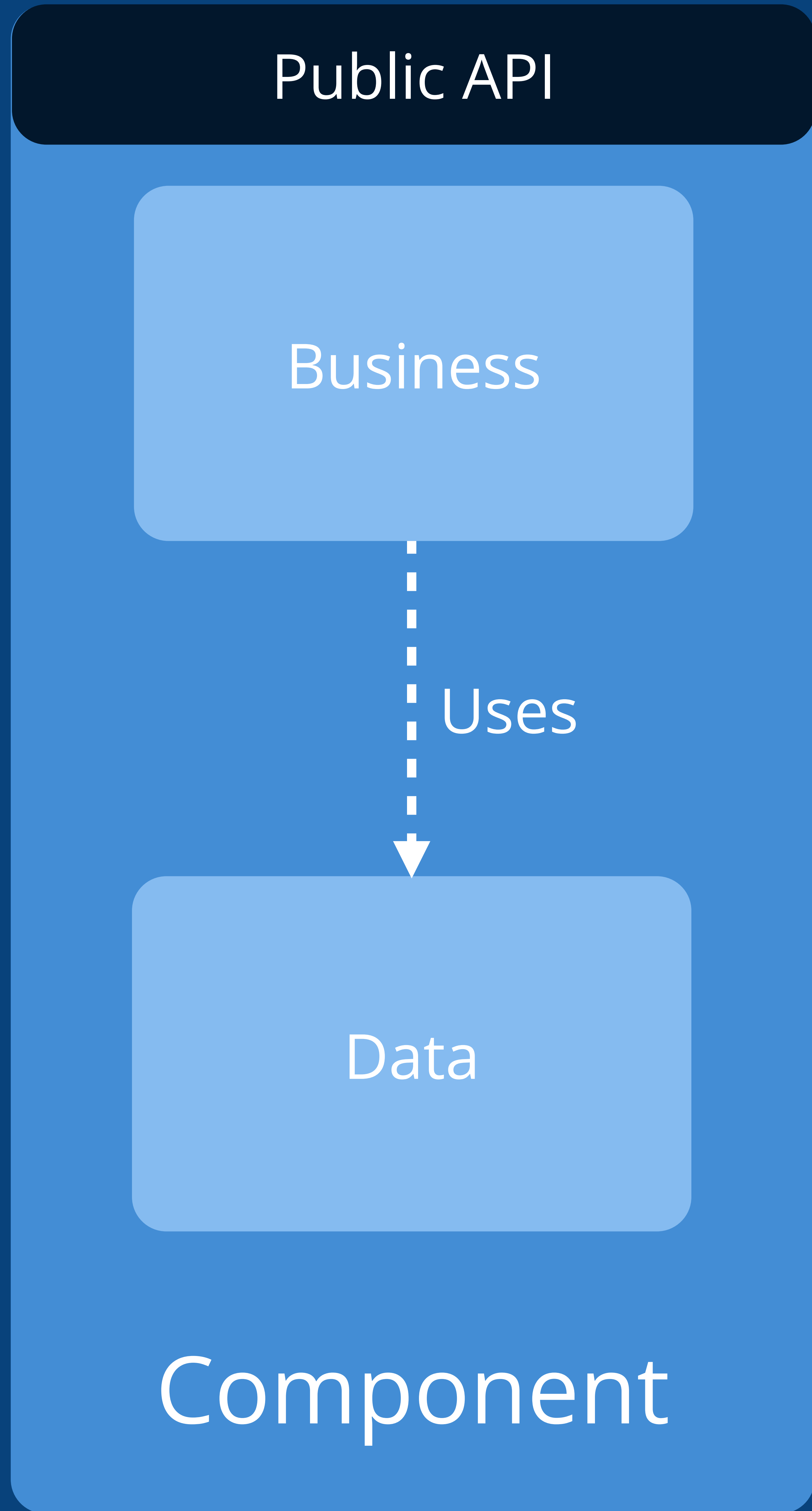
A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).


OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersComponent
OrdersComponentImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders

Package by component is about applying **component-based** or **service-oriented** design thinking to a monolithic codebase

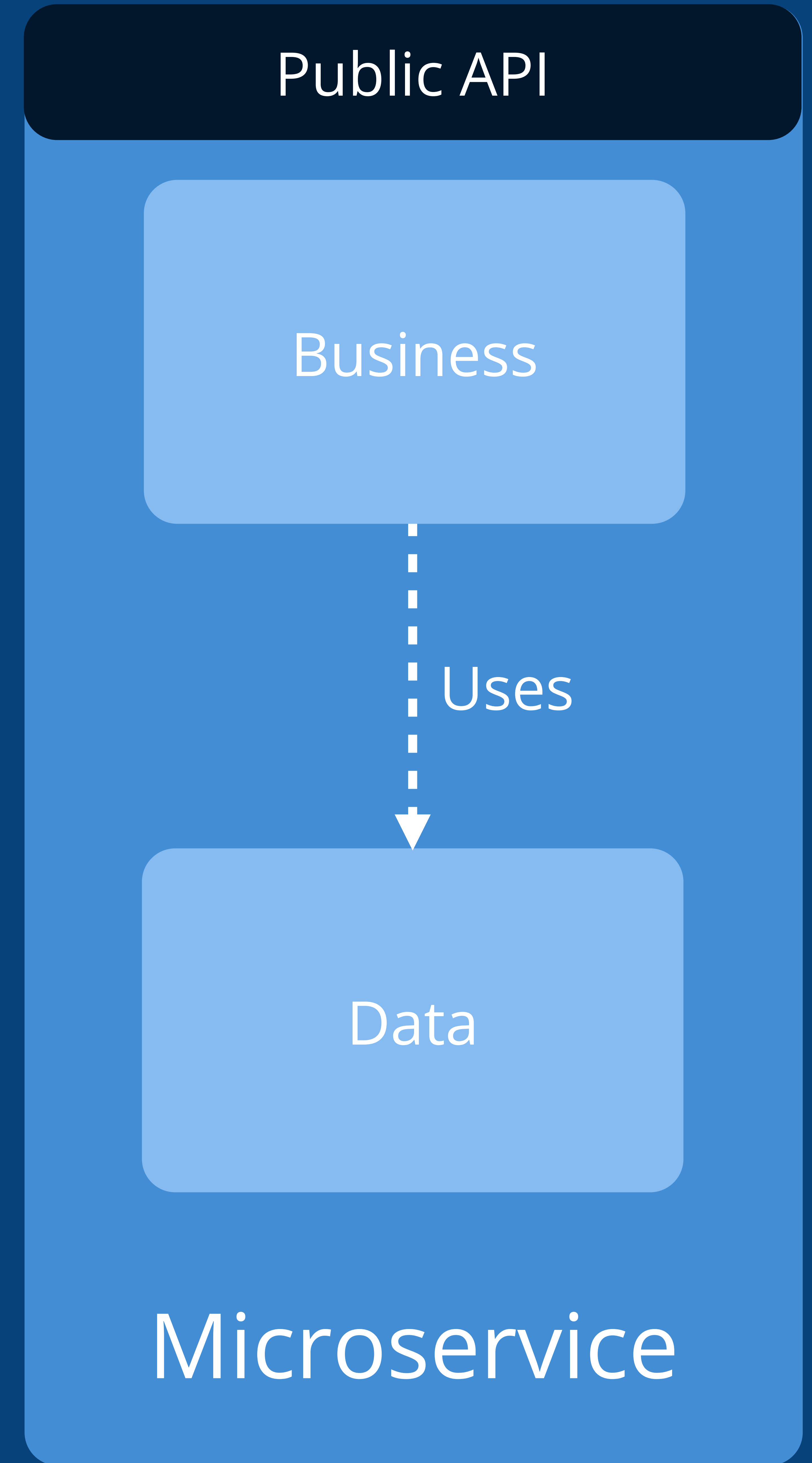
Modularity as a principle

Separating interface from implementation



Impermeable boundaries

Access modifiers vs network boundaries

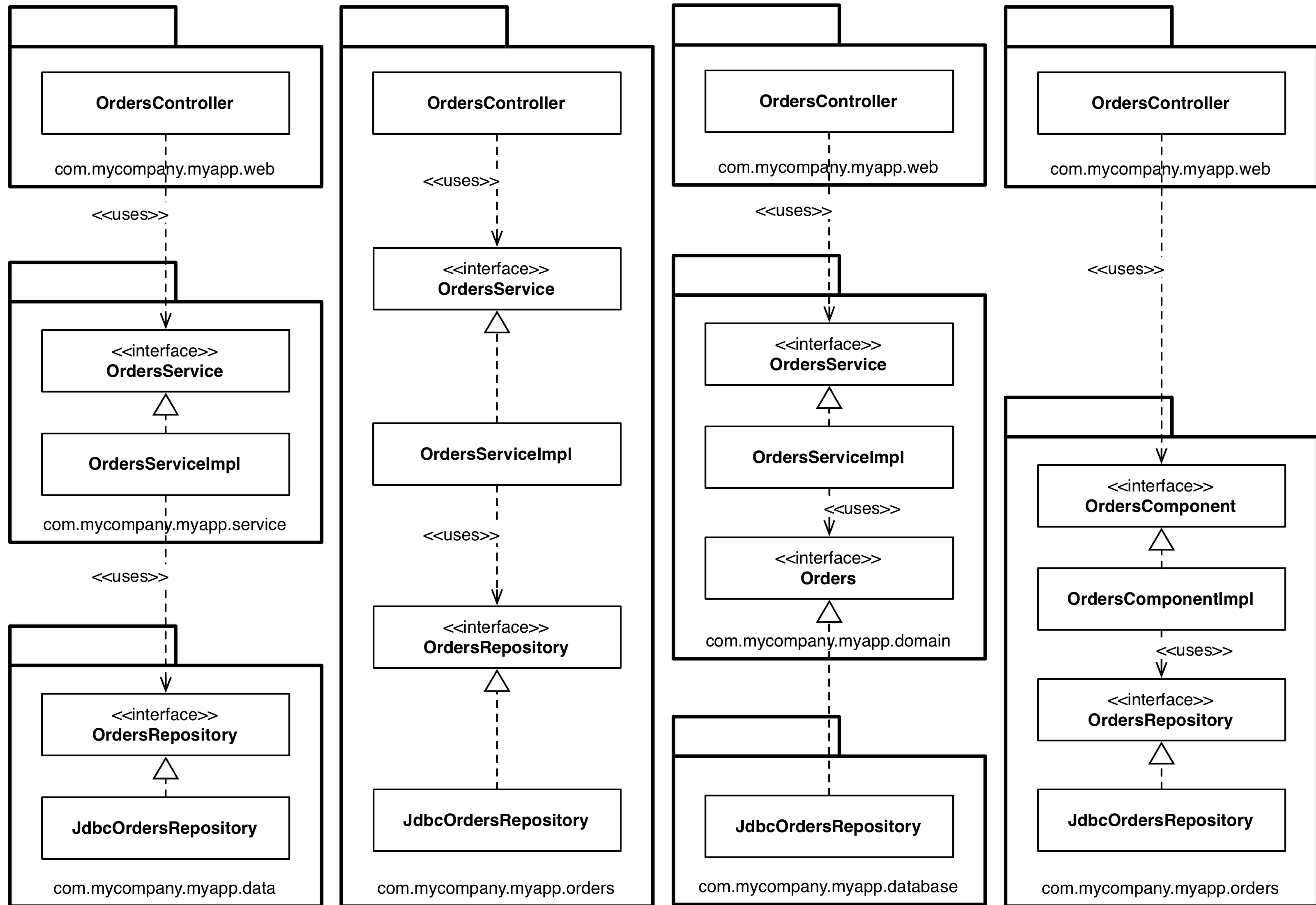


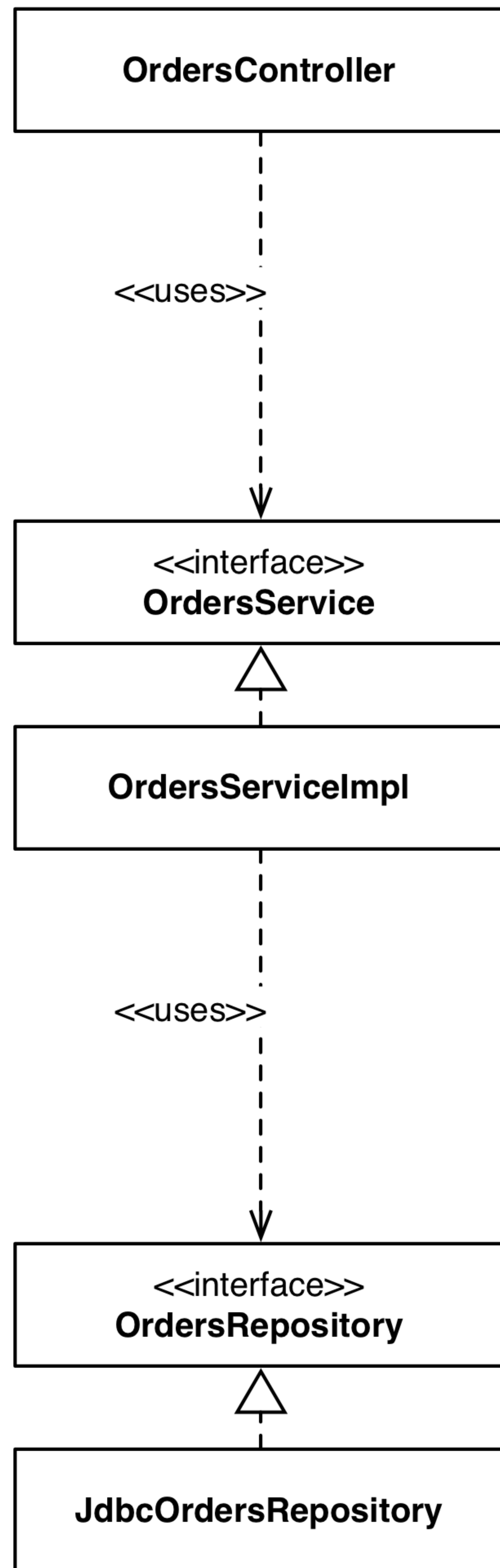
The devil is in the implementation details

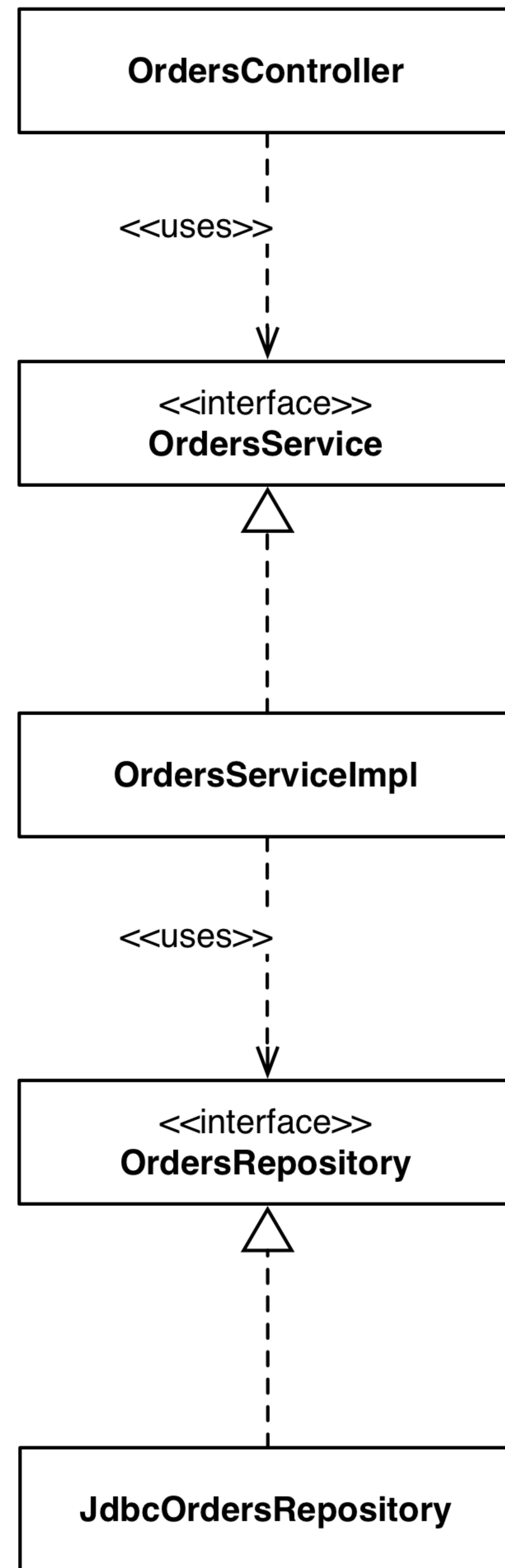
public

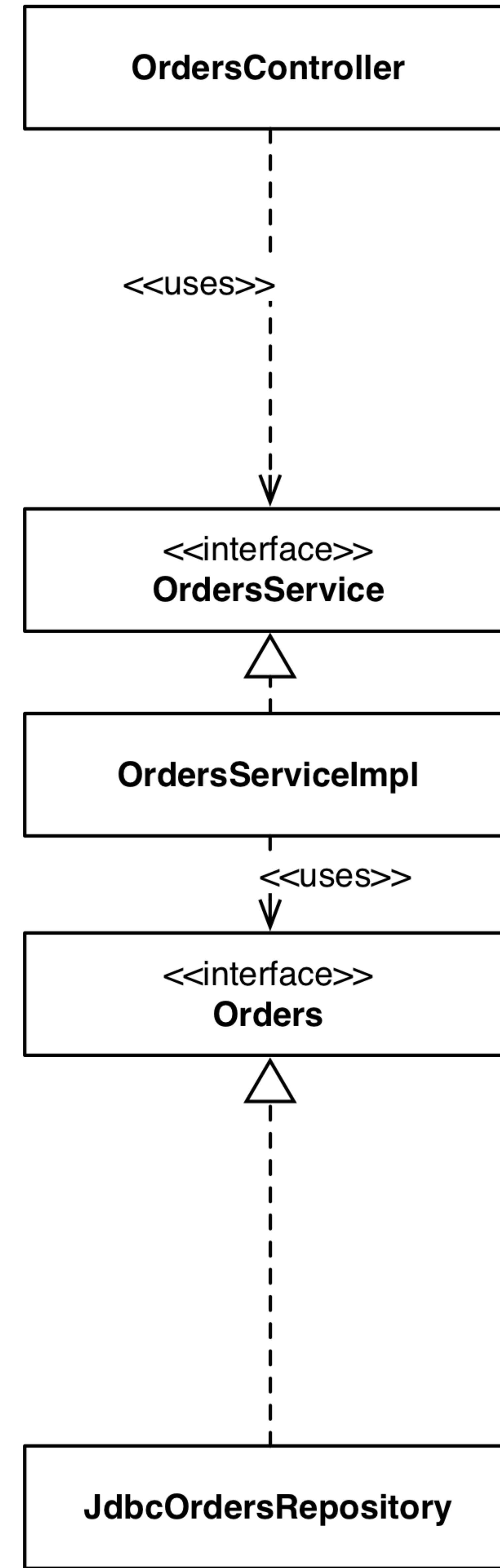
Organisation vs encapsulation

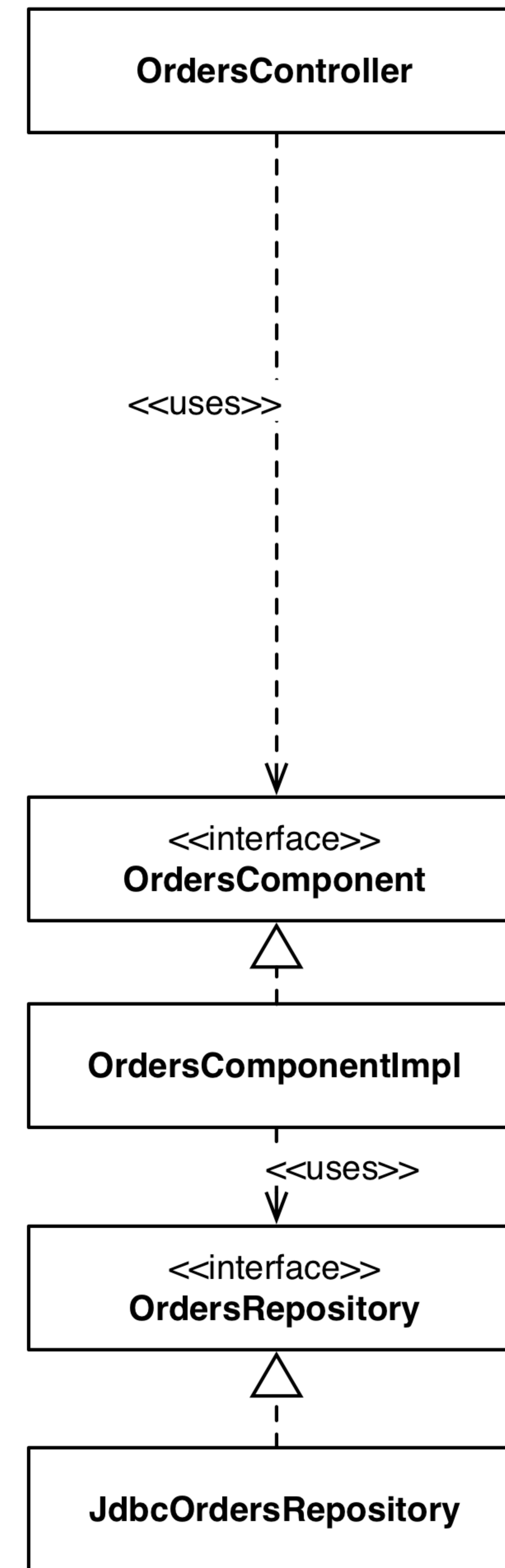
If you make all types `public`,
architectural styles
can be **conceptually different**,
but **syntactically identical**

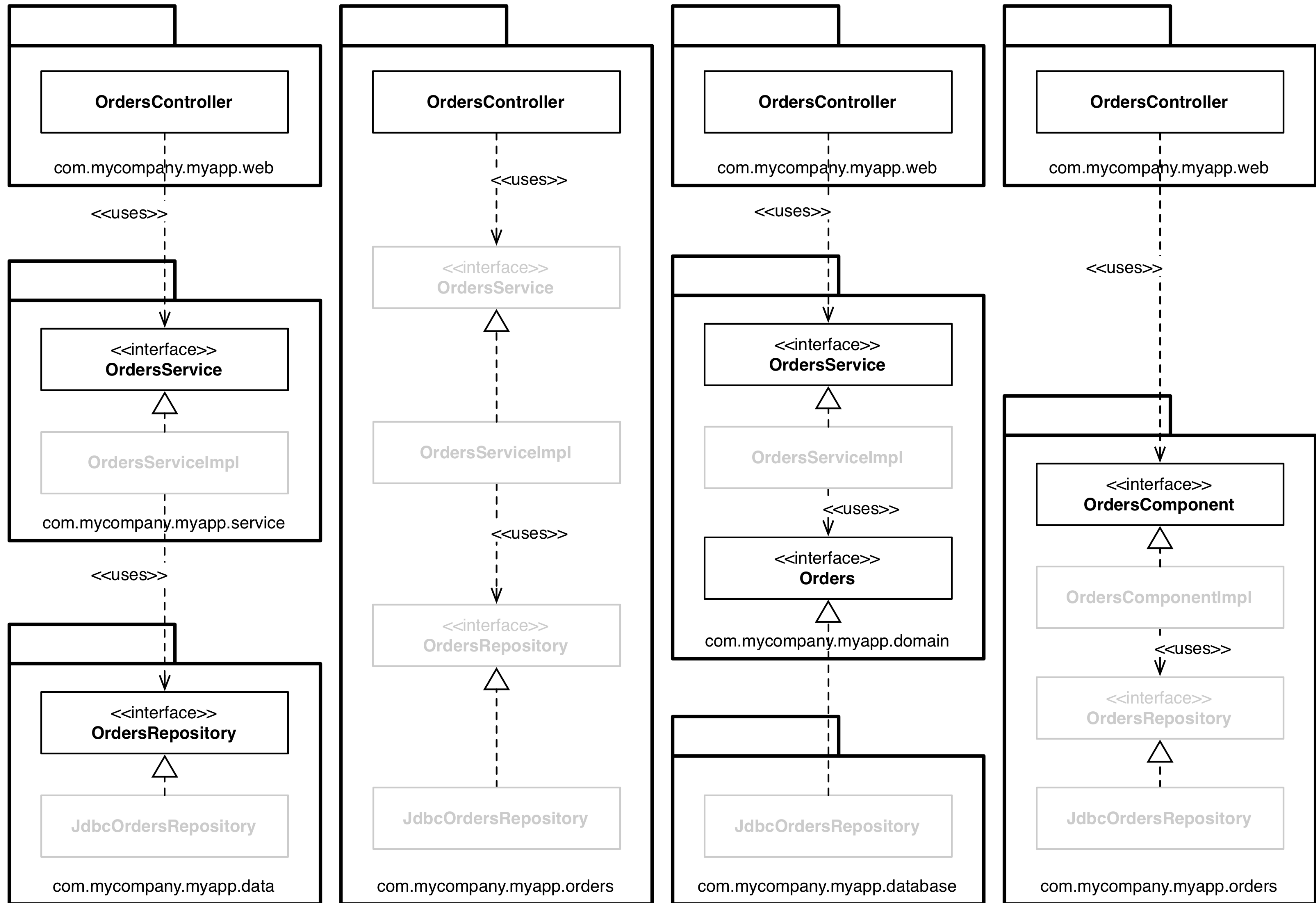

OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
com.mycompany.myapp.service
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.data
OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders
OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
Orders
com.mycompany.myapp.domain
JdbcOrdersRepository
com.mycompany.myapp.database
OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersComponent
OrdersComponentImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders


OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository


OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository


OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
Orders
JdbcOrdersRepository


OrdersController
<<uses>>
<<interface>>
OrdersComponent
OrdersComponentImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository



OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
com.mycompany.myapp.service
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.data
OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders
OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
Orders
com.mycompany.myapp.domain
JdbcOrdersRepository
com.mycompany.myapp.database
OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersComponent
OrdersComponentImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders

Use encapsulation to **minimise** the number of potential **dependencies**

The surface area of your internal public APIs should match your **architectural intent**

If you're building a monolithic application with a single codebase, **try to use the compiler to enforce boundaries**

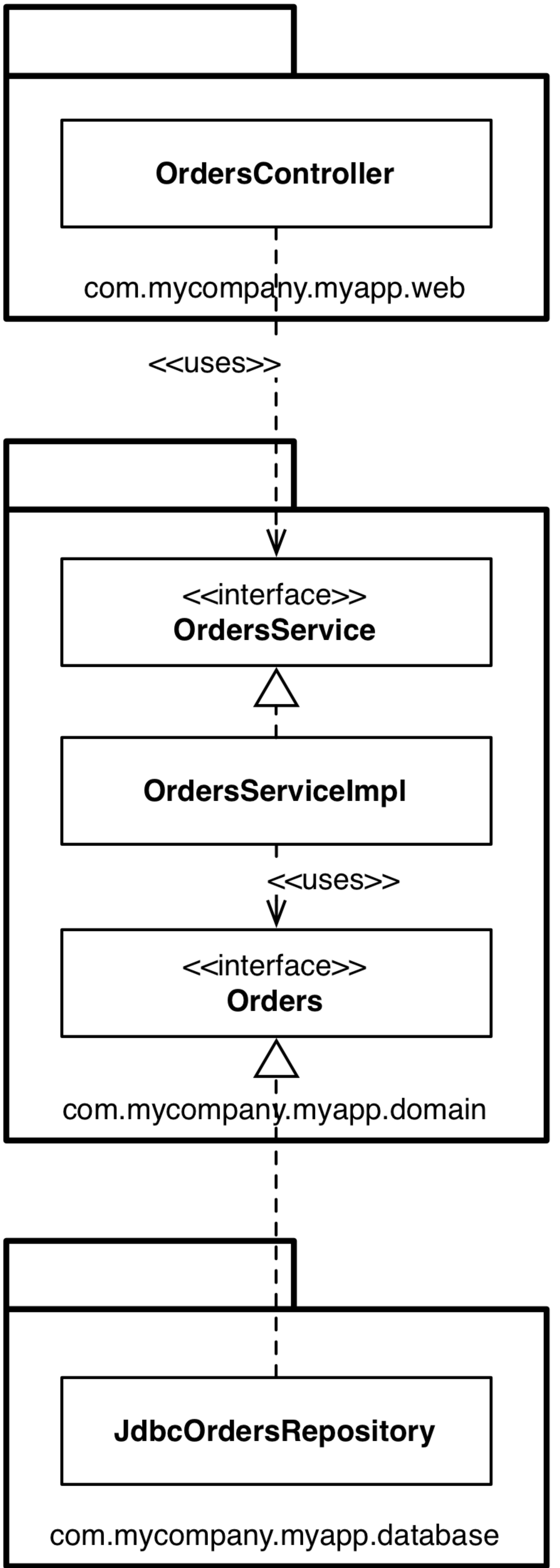
Or other decoupling modes such as a module framework that differentiates **public** from **published** types

(e.g. Java module system, Spring Modulith)

Or **split the source code tree** into multiple parts

Infrastructure

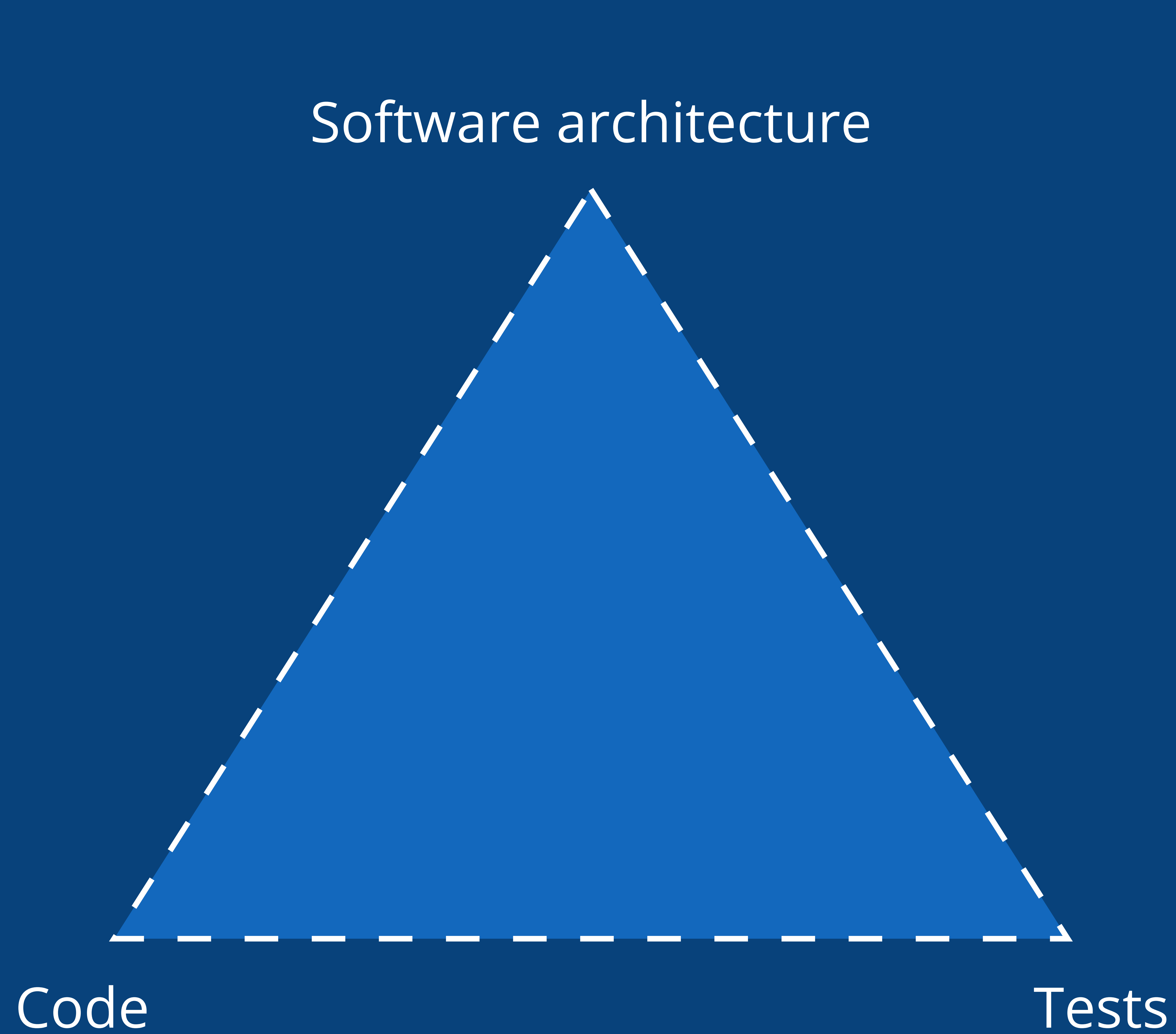
Domain

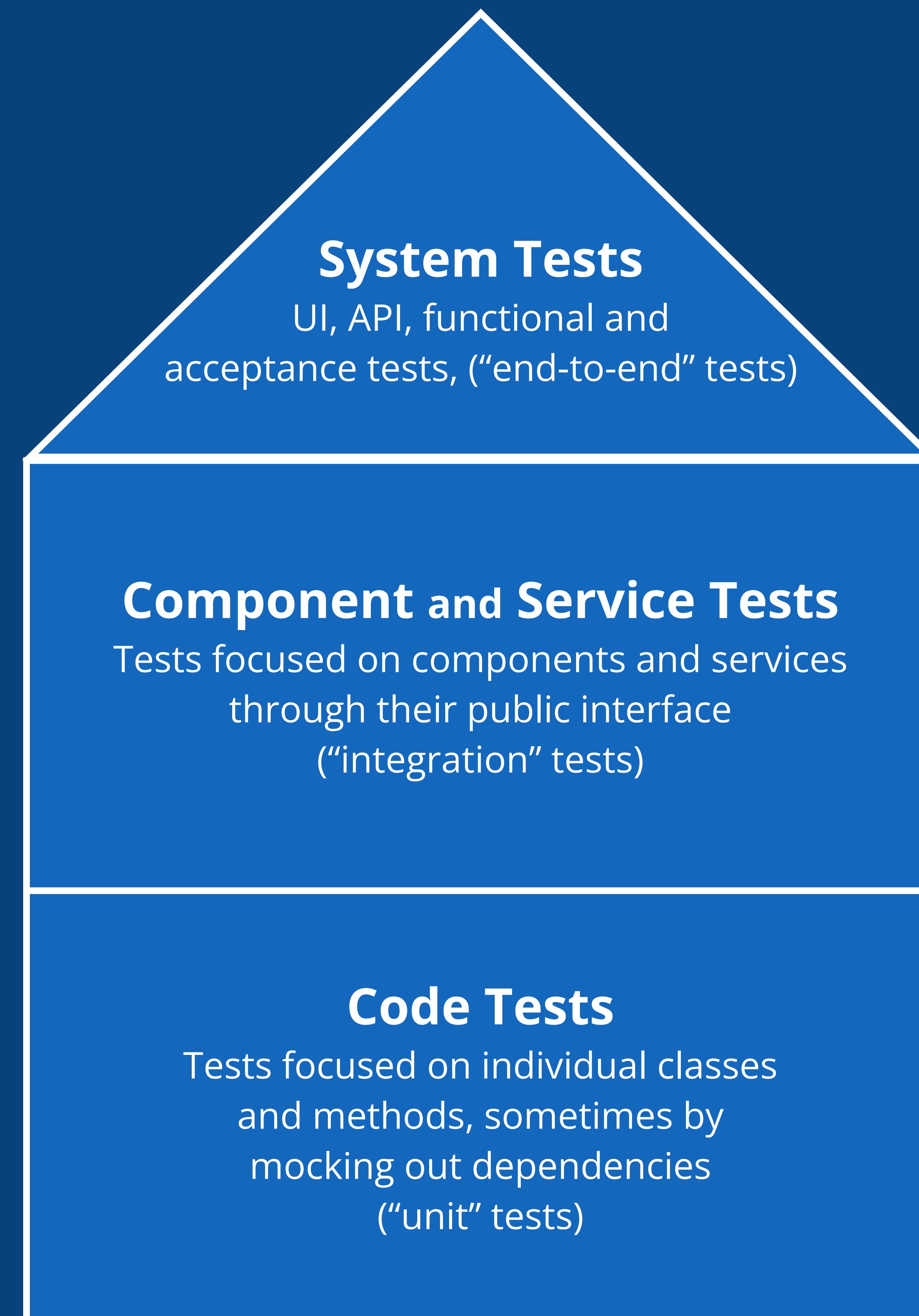


There are real-world trade-offs with many source code trees

And, more generally, each decoupling mode has different trade-offs

(modular monoliths vs microservices)



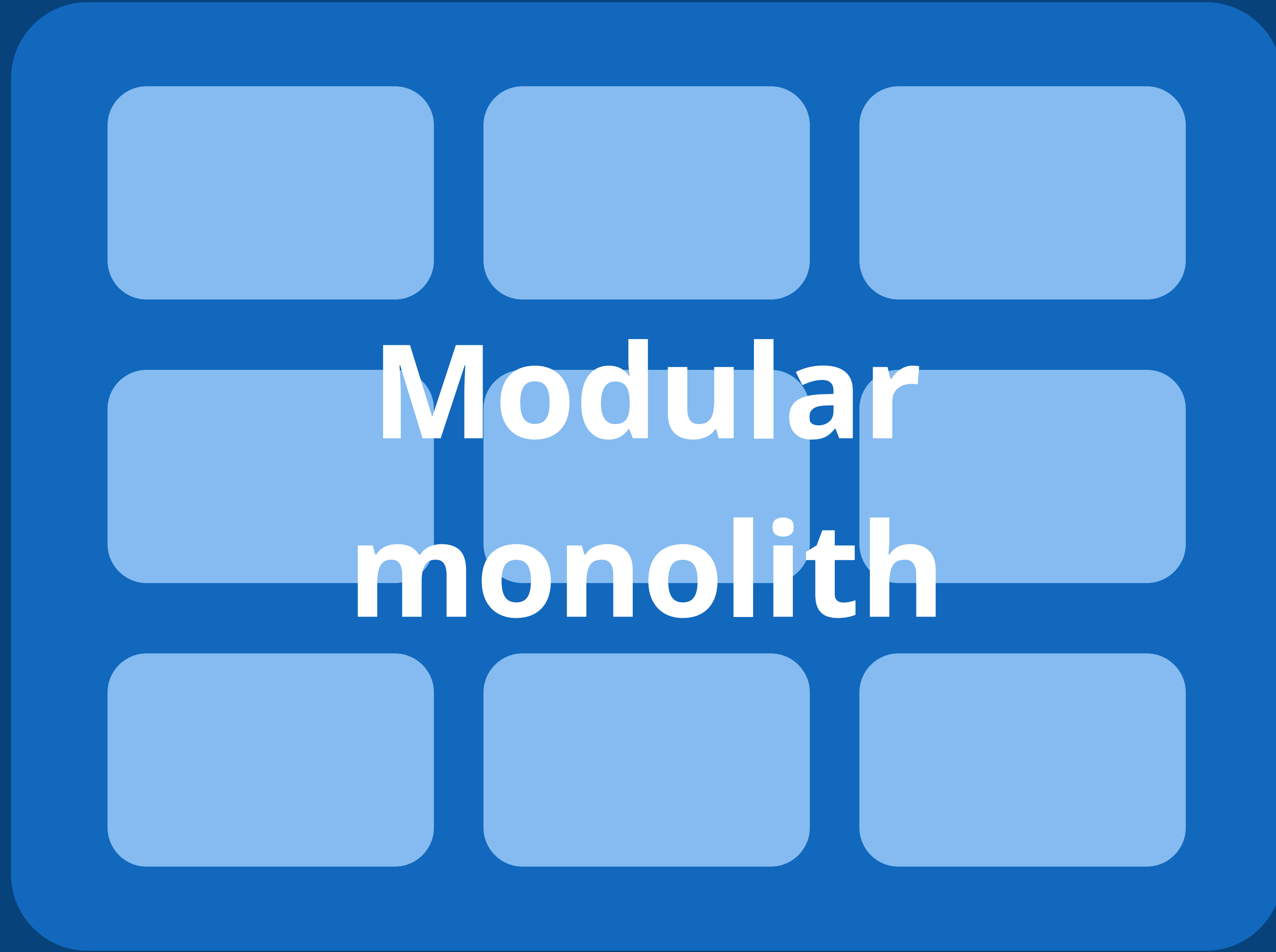


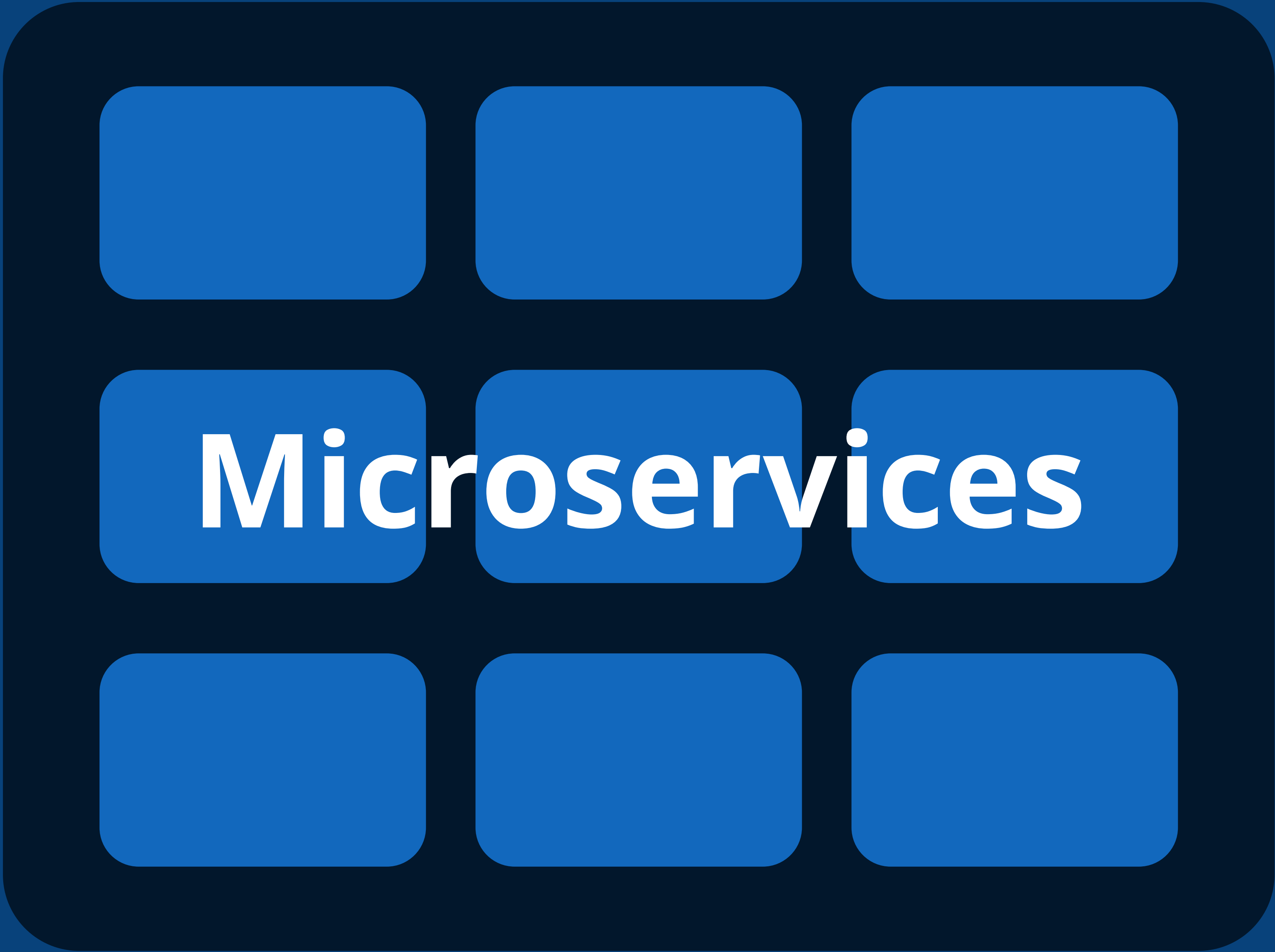
Should the relationship between
software architecture, code, and tests be more explicit?

Granularity vs testability

(some architectural styles, when combined with
dependency injection and “unit testing” promote high testability
… perhaps at the expense of coarse-grained modularity?)

A good architecture rarely happens through **architecture-indifferent design**



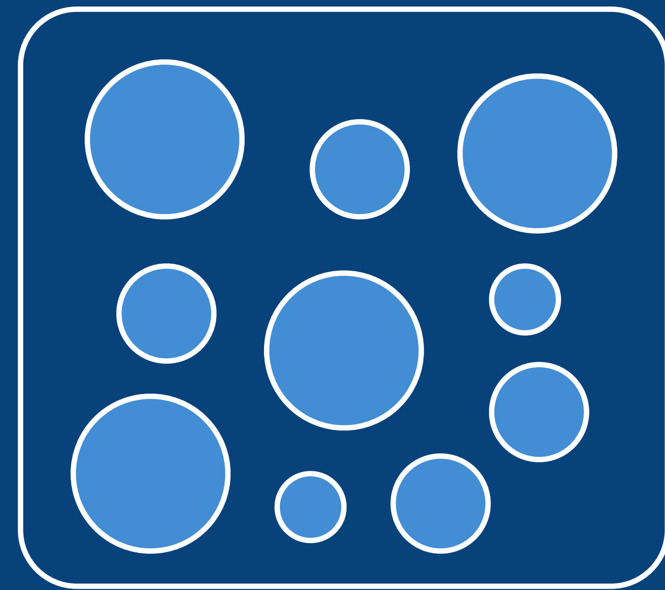


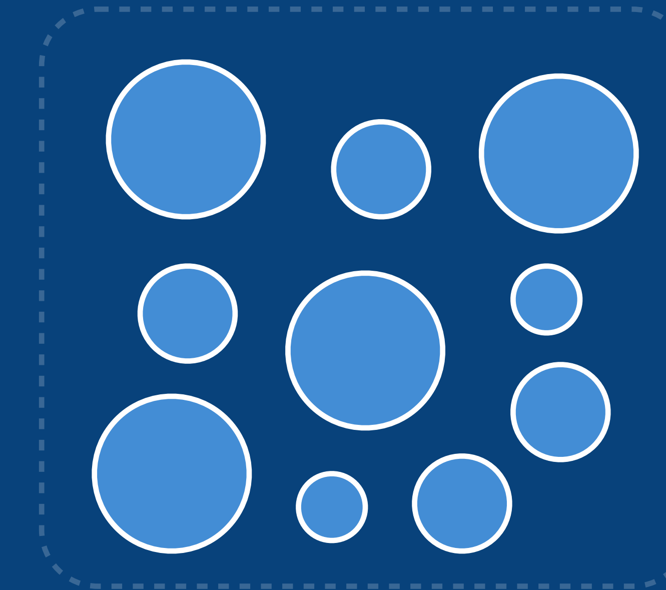
Modularity

Number of deployment units

Well-defined, in-process components is a stepping stone to out-of-process components

(i.e. microservices)





High cohesion

Low coupling

Focussed on a business capability

Bounded context or aggregate

Encapsulated data

Substitutable

Composable

< All of that plus

Individually deployable

Individually upgradeable

Individually replaceable

Individually scalable

Heterogeneous technology stacks

Choose microservices for the benefits, not because your monolithic codebase is a mess

Whatever architectural approach you choose, don’t forget about the **implementation details**

Beware of the

model-code gap

Thank you!

Simon Brown