Architecture matters

What is your architecture?

Applifting

Lukáš Rychtecký 26.10.2021


Robinson-Crusoe-Haus
Haus Atlantis

Why did we stop design systems?

“We use NestJS with TypeScript, and MySQL as a database and its run on AWS lambda”

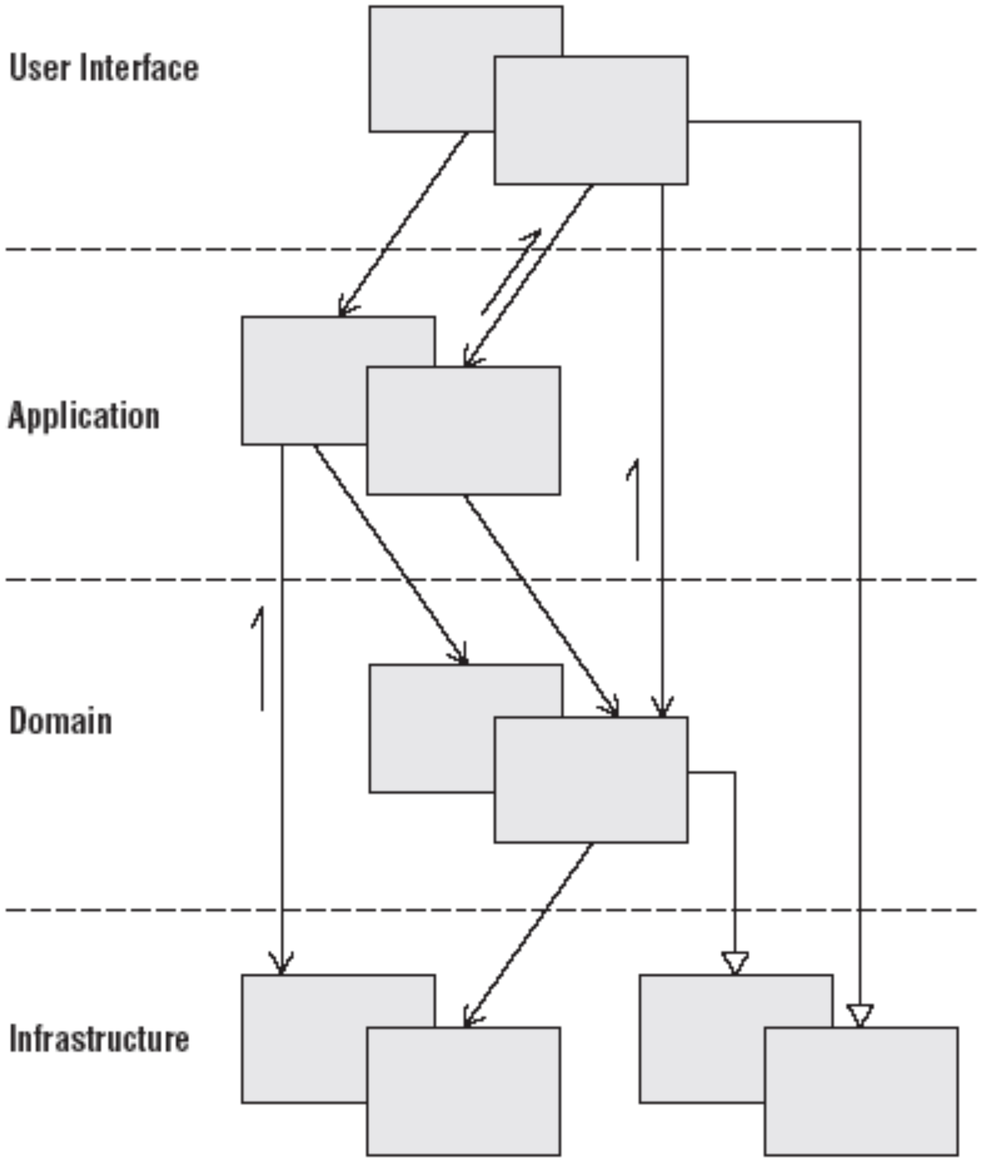
A developer

“We use NestJS with TypeScript, and MySQL as a database and its run on AWS lambda”

vs.

“Our system is designed as decoupled components based on domains interacting by messaging”

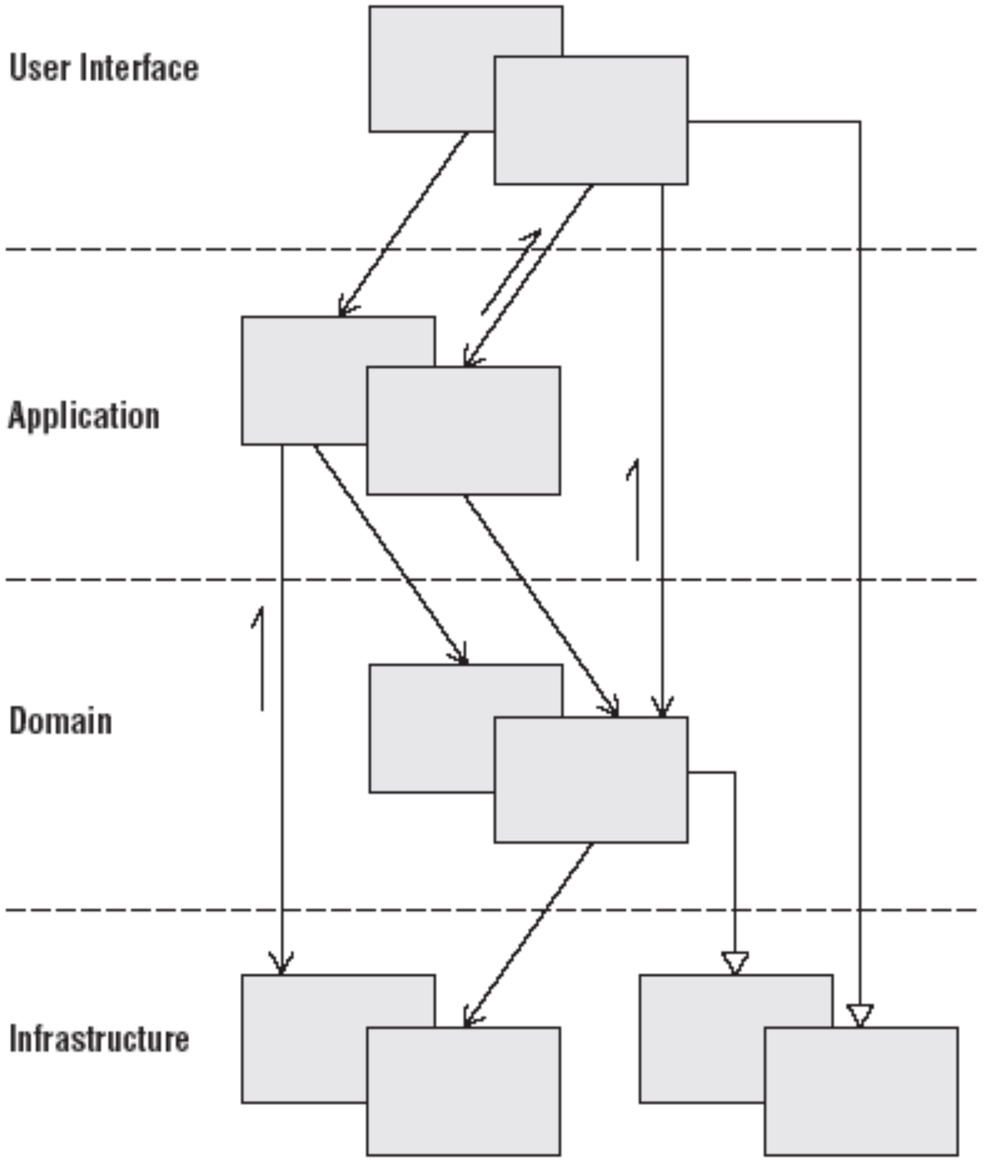
What's wrong with layered architecture?



Is layered architecture an anti-pattern?

TL;DR: NO!

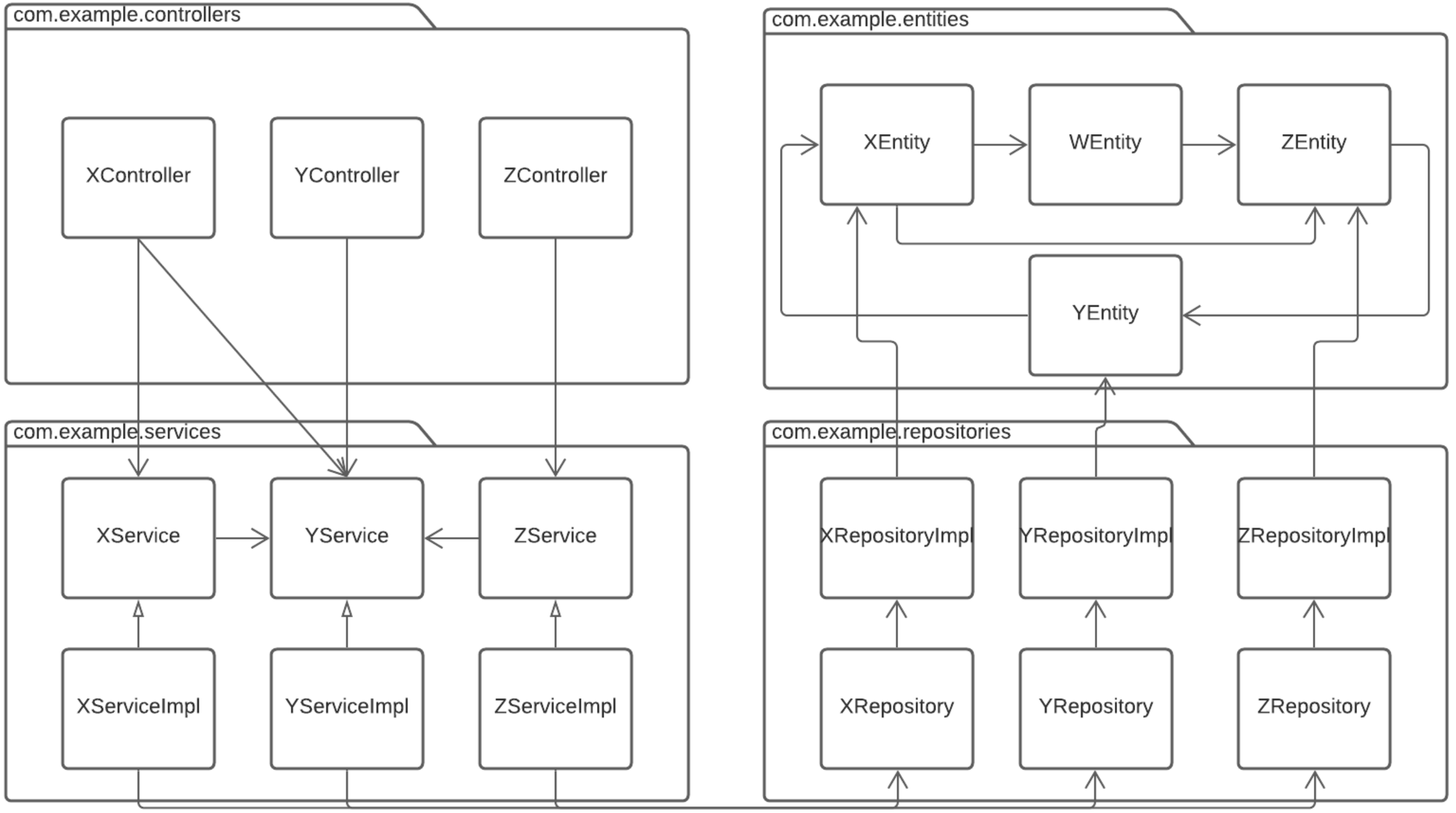
Just organise code in a *different* way.



Decompose the system into decoupled components by domains

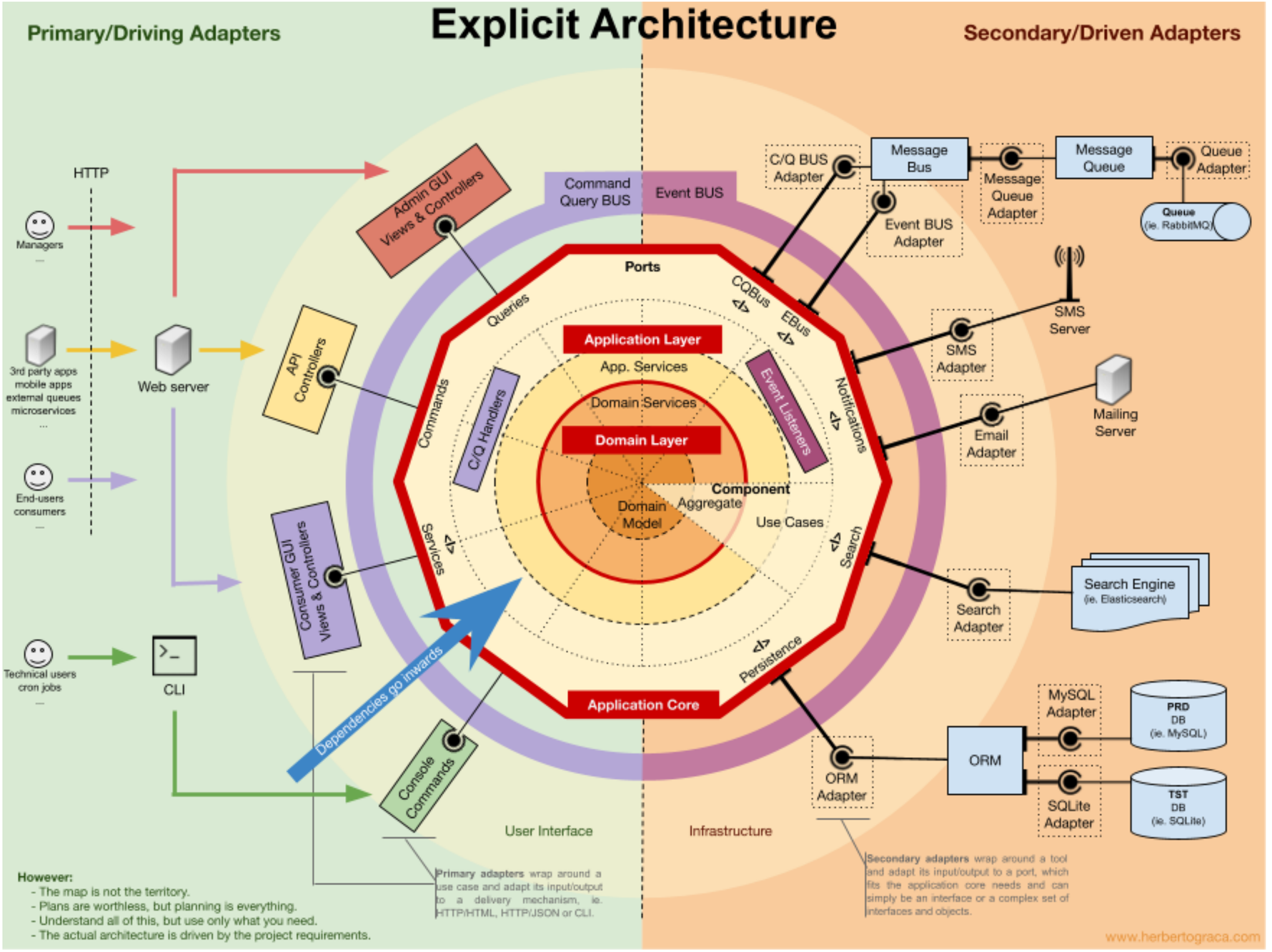
BUT WE ORGANISE THE CODE INTO SERVICES, THAT'S WE HOW DO IT IN THE ENTERPRISE

YOUR CODE IS UNREASONABLE MESS


com.example.controllers
XController
YController
ZController
com.example.entities
XEntity
WEntity
ZEntity
YEntity
com.example.services
XService
YService
ZService
XServiceImpl
YServiceImpl
ZServiceImpl
com.example.repositories
XRepositoryImpl
YRepositoryImpl
ZRepositoryImpl
XRepository
YRepository
ZRepository

“You wanted a banana but what you got was a gorilla holding the banana and the entire jungle”

Joe Armstrong - a creator of Erlang

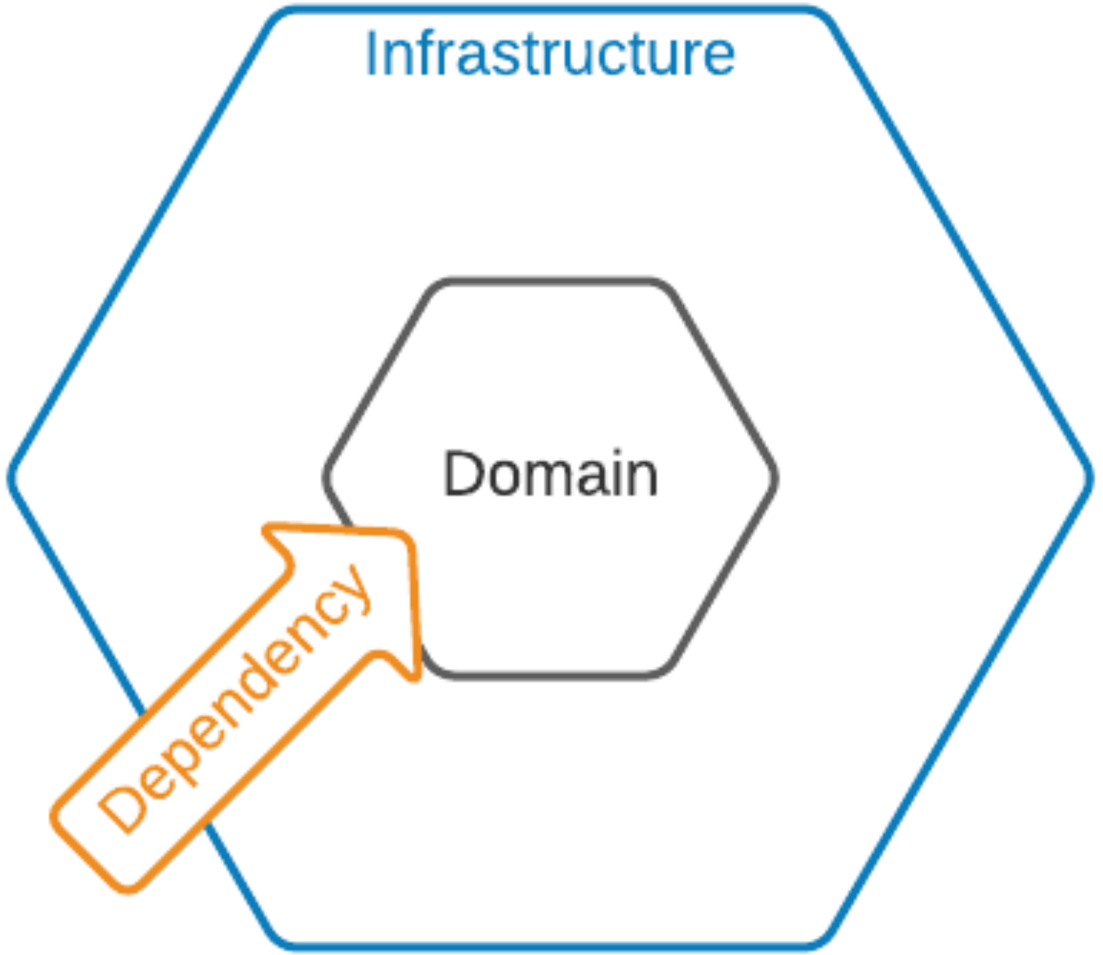

Explicit Architecture
Primary/Driving Adapters
Secondary/Driven Adapters
HTTP
Managers
3rd party apps
mobile apps
external queues
microservices
...
End-users
consumers
...
Technical users
cron jobs
...
Web server
CLI
Admin GUI
Views & Controllers
API
Controllers
Consumer GUI
Views & Controllers
Console
Commands
Command
Query BUS
Event BUS
Ports
Queries
Commands
C/Q Handlers
CQBus
EBus
Application Layer
App. Services
Domain Services
Domain Layer
Component
Domain
Model
Aggregate
Use Cases
Event Listeners
Notifications
Services
Search
Persistence
Application Core
Dependencies go inwards
C/Q BUS
Adapter
Message
Bus
Event BUS
Adapter
Message
Queue
Adapter
Queue
Adapter
Queue
(ie. RabbitMQ)
SMS
Server
SMS
Adapter
Mailing
Server
Email
Adapter
Search Engine
(ie. Elasticsearch)
Search
Adapter
MySQL
Adapter
PRD
DB
(ie. MySQL)
ORM
ORM
Adapter
SQLite
Adapter
TST
DB
(ie. SQLite)
User Interface
Infrastructure
However:
- The map is not the territory.
- Plans are worthless, but planning is everything.
- Understand all of this, but use only what you need.
- The actual architecture is driven by the project requirements.
Primary adapters wrap around a use case and adapt its input/output to a delivery mechanism, ie. HTTP/HTML, HTTP/JSON or CLI.
Secondary adapters wrap around a tool and adapt its input/output to a port, which fits the application core needs and can simply be an interface or a complex set of interfaces and objects.
www.herbertograca.com

Hexagonal architecture

- An architectural pattern
- It aims to ***loosely coupled*** components
- Invented by Alistair Cockburn (2005)
- Separates the business logic from the infrastructure details
- An infrastructure ***should*** depends on the domain, ***not*** vice versa
- Yes, ***Spring*** is an ***infrastructure*** detail, same ***Hibernate***
- Make system dependencies ***explicit***
- Move ***side effects to the boundary*** of the domain
- a.k.a. Ports and Adapters

Domain Driven Design

- It’s a concept that structure the code around the domains
- Define ***a domain dictionary***, make it ***public*** and ***use*** it
- Design the system by ***domains***
- Use language ***first-class citizens*** structures
- Easier for ***testing*** and ***reasoning*** the code


Infrastructure
Domain
Dependency

Keywords

Ubiquitous language

- ***Common language between*** programmers and business
- Define it ***explicitly***
- ***Use*** it in discussions and codes
- Don’t interchange similar works like ***order*** and ***transaction***

Keywords

Entities

- Identity by ***an ID***
- Use ID type, instead of Int/UUID
- Could be plain structures (POJOs, no getter and setters)
- One entity **could** be represented with more classes/data structures
- ***Domain entities aren't Hibernate*** entities (nor other ORM entities)
- Contain ***small amount of logic*** (no, sending an e-mail should not be here)
- Anemic model (FP) vs. Rich model (OOP)

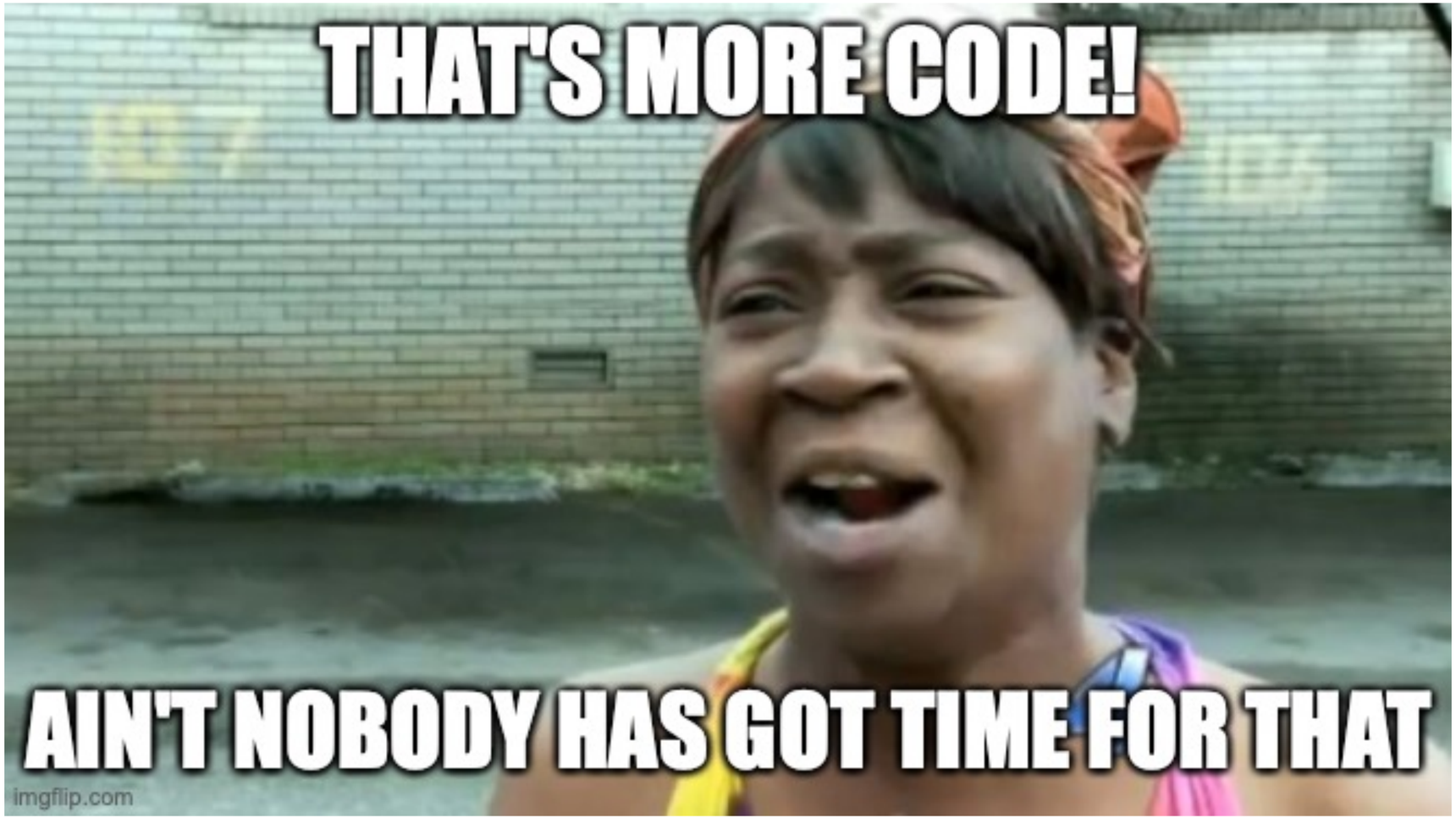
BUT SETTERS
PROVIDE ENCAPSULATION,
THE KEY THING IN OOP

THE API OF THE
DOMAIN OBJECTS SHOULD
SPEAK IN THE DOMAIN LANGUAGE

```
package bad.example.project

import javax.persistence.Entity
import javax.persistence.Id
import javax.persistence.Table
import javax.validation.constraints.Email

@Entity
@Table(name="customers")
class Customer constructor(
    @Id
    private var id: Long,
    @Email
    private var email: String,
    private var firstName: String,
    private var lastName: String,
) {
    fun getId() = id
    fun setId(id: Long) {
        this.id = id
    }
    fun getEmail() = email
    fun setEmail(email: String) {
        this.email = email
    }
    fun getFirstName() = firstName
    fun setFirstName(firstName: String) {
        this.firstName = firstName
    }
    fun getLastName() = lastName
    fun setLastName(lastName: String) {
        this.lastName = lastName
    }
}
```

VS.

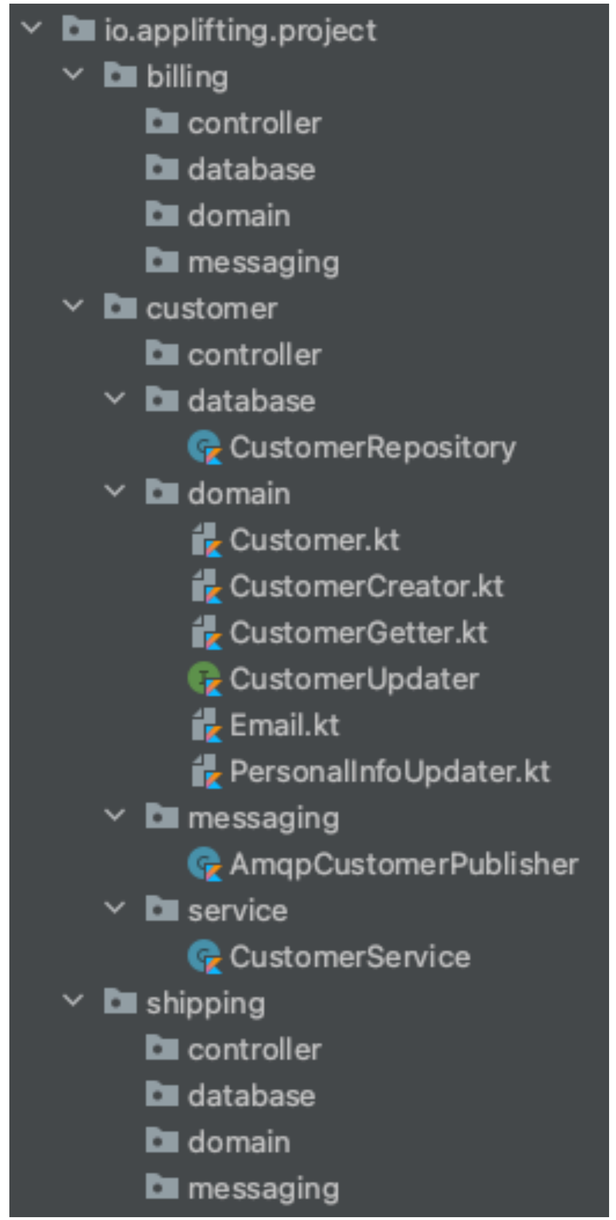
```
package io.applifting.project.customer.domain

class InvalidEmail(message: String?) : Exception(message)

class Email private constructor(val address: String) {

    companion object {
        // TODO: Implement the validation
        private fun isValid(address: String) = false

        fun of(address: String): Result<Email> =
            if (isValid(address)) {
                Result.failure(InvalidEmail("Given value is invalid: $address"))
            } else {
                Result.success(Email(address))
            }
    }
}

data class CustomerId(val id: Long)

data class PersonalInfo(
    val id: CustomerId,
    val firstName: String,
    val lastName: String,
)

class InvalidPersonalInfo(message: String?) : Exception(message)

class Customer constructor(
    private val id: CustomerId,
    private val email: Email,
    private var firstName: String,
    private var lastName: String,
) {
    fun getId(): CustomerId = id
    fun getEmail(): Email = email
    fun getFirstName(): String = firstName
    fun getLastName(): String = lastName
    // TODO: Implement the validation
    private fun isValidPersonalInfo(info: PersonalInfo) = true
    fun updatePersonalInfo(info: PersonalInfo): Result<Customer> =
        if (isValidPersonalInfo(info)) {
            this.firstName = info.firstName
            this.lastName = info.lastName
            Result.success( value: this)
        } else {
            Result.failure(InvalidPersonalInfo("Given an invalid personal info"))
        }
}
```

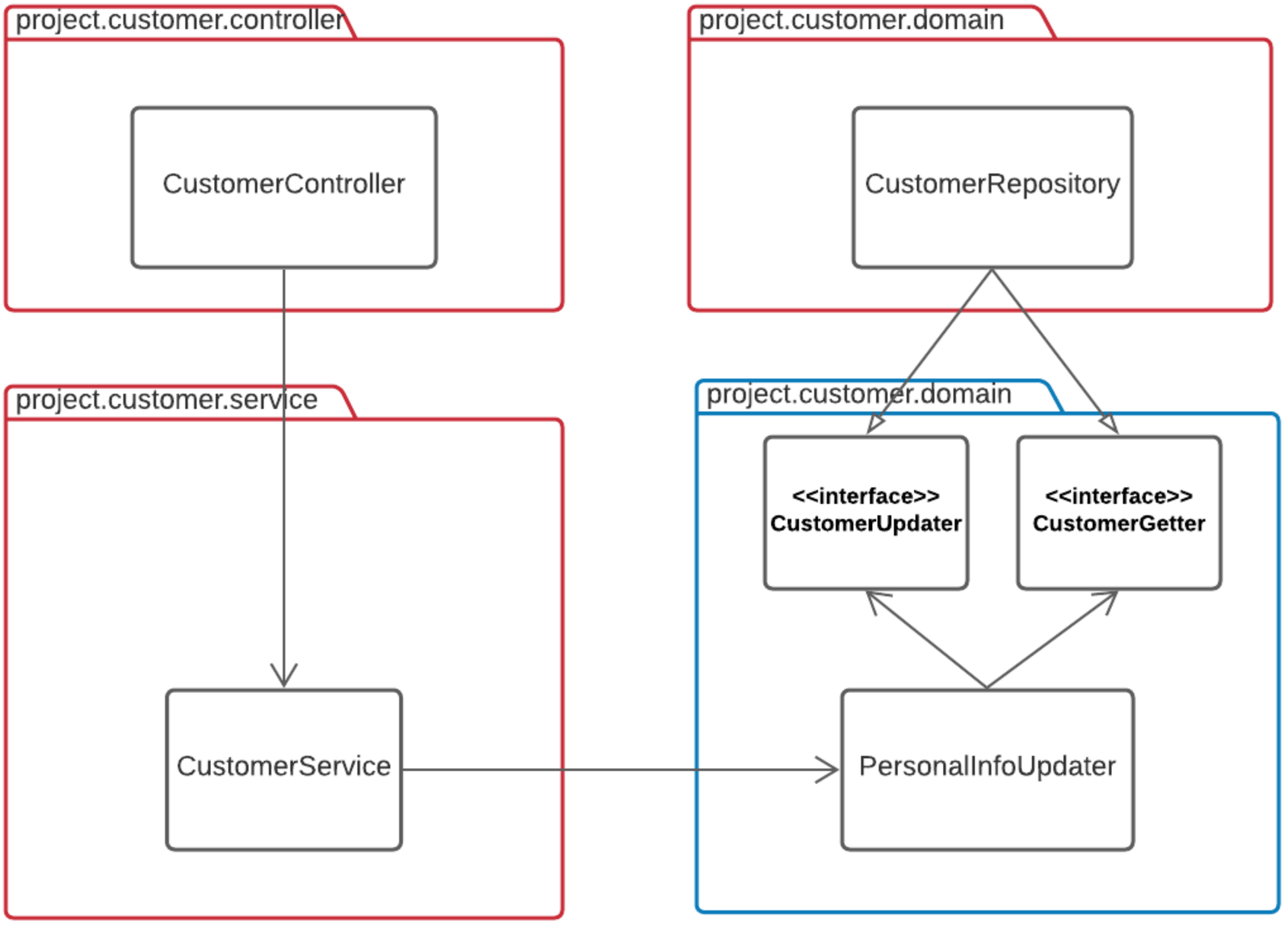

THAT'S MORE CODE!
AIN'T NOBODY HAS GOT TIME FOR THAT
imgflip.com

Keywords

Modules

- A.K.A. components
- Low coupling
- High cohesion
- Couple code by its ***domain*** rather than by its "type"
- Should tell the story "Our company does ***shipping*** for ***customers*** so that we can ***bill*** them."
- Shipping, Customer, Billing could be ***good*** names for modules




project.customer.controller
CustomerController
project.customer.domain
CustomerRepository
project.customer.service
CustomerService
project.customer.domain
<<interface>>
CustomerUpdater
<<interface>>
CustomerGetter
PersonalInfoUpdater

Give the modules names that become part of the ubiquitous language

Keywords

Value objects

- Identity by its value
- Objects that make no sense without an entity
- E.g. Date, Int, Money

Keywords

Repositories

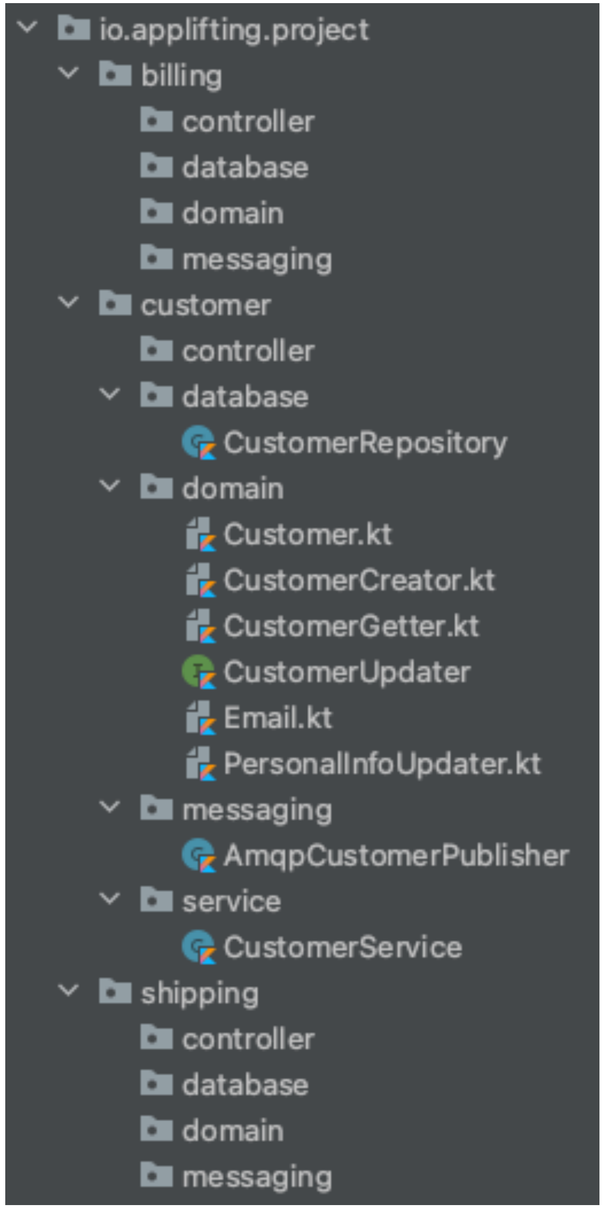
- Persistent containers of *entities* or *aggregates*
- *Avoid* using Active record pattern outside of repositories
- They ***live outside*** of the domain
- They ***implements interfaces*** from the domain

Keywords

Repositories

```kotlin
package io.applifting.project.customer.domain

interface CustomerGetter {
    fun get(id: CustomerId): Customer?
}

interface CustomerCreator {
    fun create(customer: Customer): Customer
}

interface CustomerUpdater {
    fun update(customer: Customer): Customer
}
```

```kotlin
package io.applifting.project.customer.database

import io.applifting.project.customer.domain.Customer
import io.applifting.project.customer.domain.CustomerCreator
import io.applifting.project.customer.domain.CustomerGetter
import io.applifting.project.customer.domain.CustomerId
import io.applifting.project.customer.domain.CustomerUpdater
import org.springframework.stereotype.Repository

@Repository
class CustomerRepository : CustomerGetter, CustomerCreator, CustomerUpdater {
    override fun get(id: CustomerId): Customer? {
        TODO( reason: "Not yet implemented")
    }

    override fun create(customer: Customer): Customer {
        TODO( reason: "Not yet implemented")
    }

    override fun update(customer: Customer): Customer {
        TODO( reason: "Not yet implemented")
    }
}
```



Keywords

Domain Services

- ***Stateless***
- Highly cohesive
- Single Responsibility Principle - ***SRP***
- Contain ***the business logic***, that doesn't naturally fit elsewhere
- Publish domain events
- Use ***composition*** over inheritance
- Language first-class citizens

Keywords

Application Services

- As ***facades*** for your domain services (a nice API with system dependencies)
- Dependencies injected by a framework
- Handle transactions
- Import from a framework
- ***Glue*** of the domain and infrastructure

Keywords

Application Services

- Do you know what it calls internally?

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

```
package bad.example.project

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service

@Service
class BadCustomerService @Autowired constructor(
    val repository: CustomerRepository
) {
    fun updateCustomer(customer: Customer): Customer? {
        val existingCustomer = repository.get(customer.getId())
        if (existingCustomer != null) {
            existingCustomer.setFirstName(customer.getFirstName())
            existingCustomer.setLastName(customer.getLastName())
            repository.update(existingCustomer)
            return customer
        }
        return null
    }
}
```

VS.

```
package io.applifting.project.customer.domain

class CustomerNotFound(message: String?) : Exception(message)

class PersonalInfoUpdater constructor(
    private val getter: CustomerGetter,
    private val updater: CustomerUpdater,
) {
    fun update(info: PersonalInfo): Result<Customer> {
        val existingCustomer = getter.get(info.id)
        if (existingCustomer == null) {
            return Result.failure(CustomerNotFound(info.id.toString()))
        }

        val result = existingCustomer.updatePersonalInfo(info)
        result.onSuccess { updater.update(it) }
        return result
    }
}
```

```
package io.applifting.project.customer.service

import io.applifting.project.customer.database.CustomerRepository
import io.applifting.project.customer.domain.Customer
import io.applifting.project.customer.domain.PersonalInfo
import io.applifting.project.customer.domain.PersonalInfoUpdater
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service

@Service
class CustomerService @Autowired constructor(
    private val repository: CustomerRepository
) {
    fun updatePersonalInfo(info: PersonalInfo): Result<Customer> =
        PersonalInfoUpdater(repository, repository).update(info)
}
```

Define the interface in the terms of the ubiquitous language

Keywords

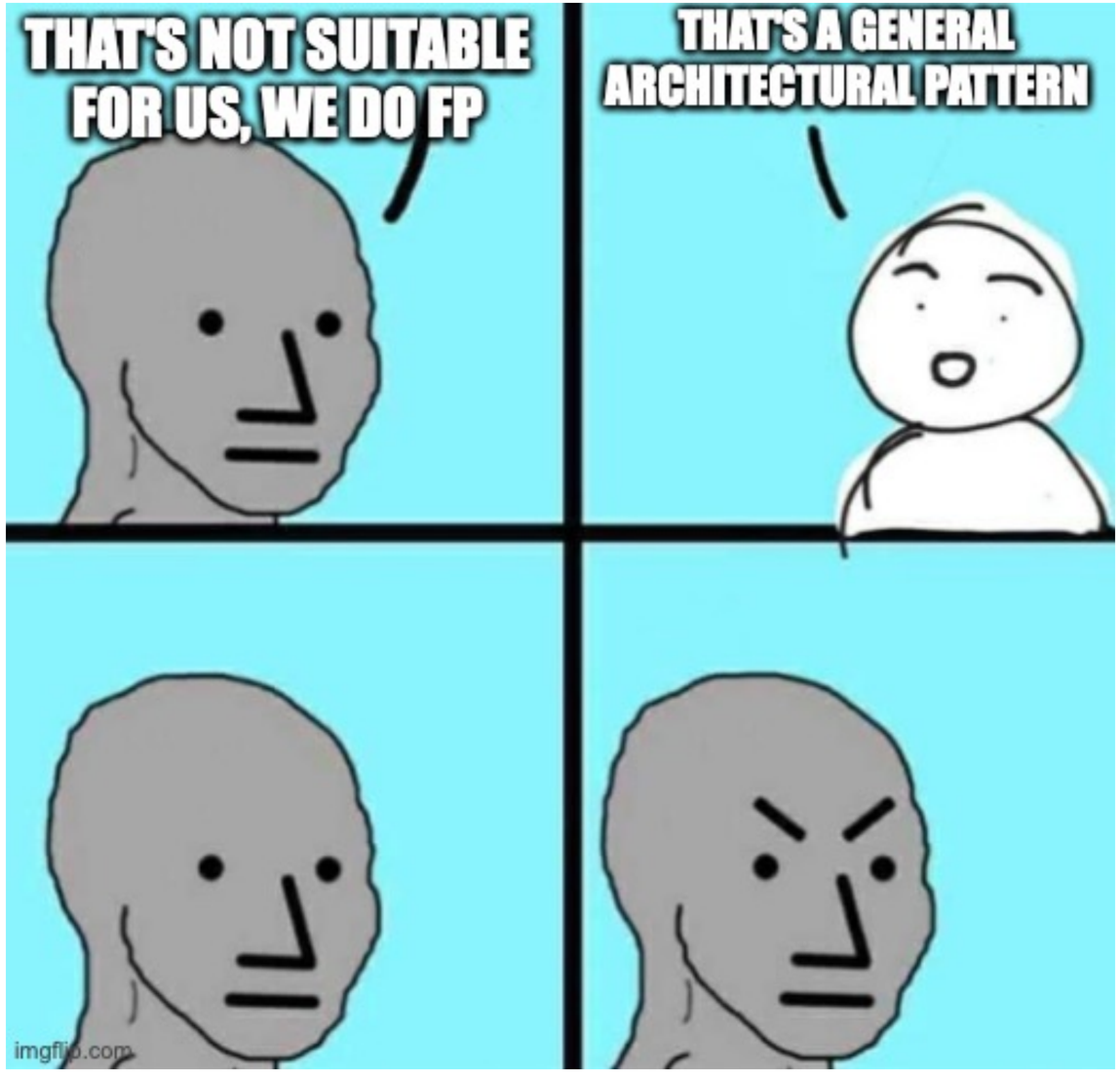
Aggregates

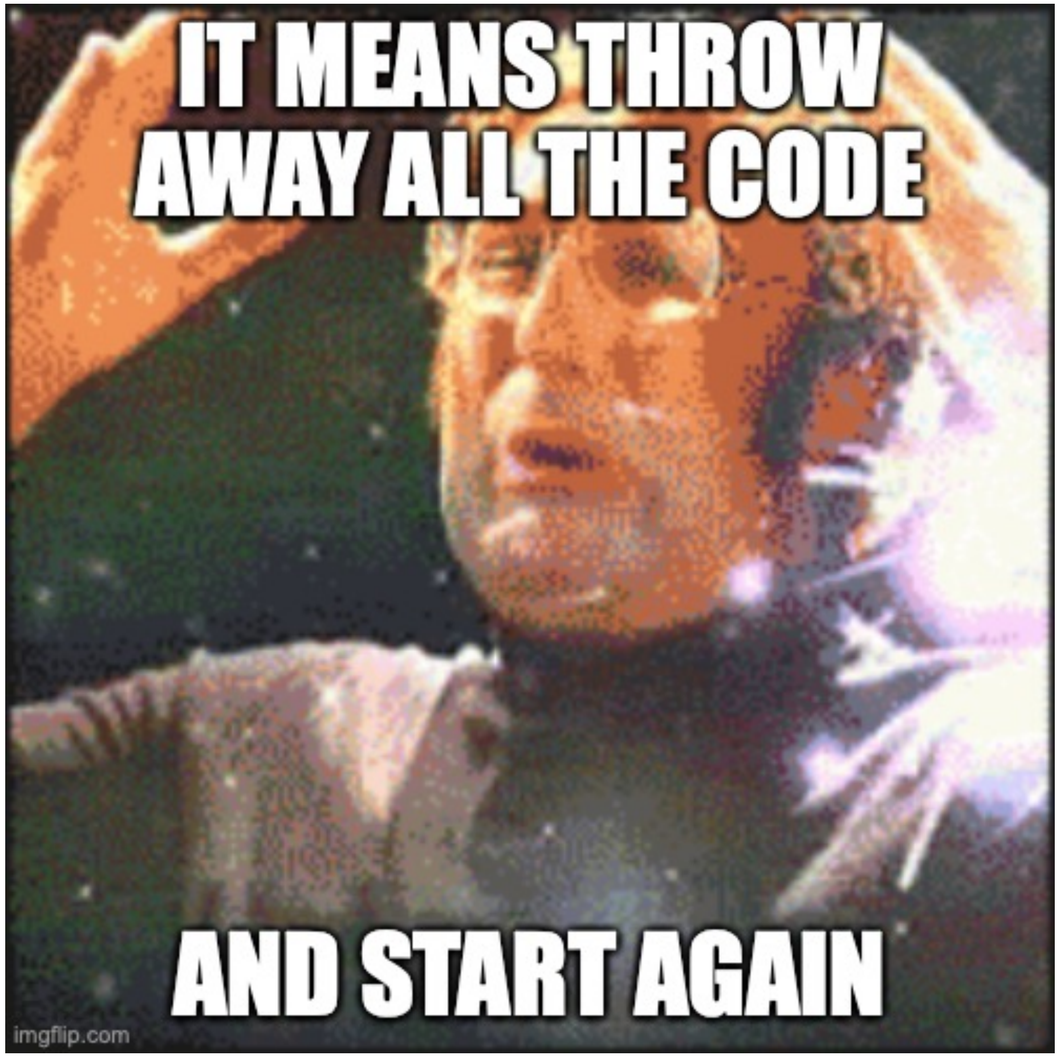
- Cluster ***the entities*** and ***value objects*** into aggregates
- Draw ***an explicit boundary*** around it
- Aggregate root - an entity
- An object cannot reference ***anything*** in the boundary, except its root

Keywords

Factories

- Responsible for ***creation of complex objects*** and aggregates
- Be careful a moment for creating an object ***should*** be triggered by a domain operation and belongs to the domain


THAT'S NOT SUITABLE FOR US, WE DO FP
THAT'S A GENERAL ARCHITECTURAL PATTERN
imgflip.com


IT MEANS THROW
AWAY ALL THE CODE
AND START AGAIN
imgflip.com

Really, where to start?

- Design first
- Reveal domains
- Make a plan and share it with others
- Refactor the code by Incremental refactoring technique
- Create the guidelines for new code and share it
- When starting a green field project, postpone a DB choice as possible

- Domain-Driven Design and the Hexagonal Architecture - https://vaadin.com/learn/tutorials/ddd/ddd_and_hexagonal
- Tactical Domain-Driven Design - https://vaadin.com/learn/tutorials/ddd/tactical_domain_driven_design
- Martin Fowler - https://www.martinfowler.com/tags/domain%20driven%20design.html
- Domain-Driven Design: Tackling Complexity in the Hearth of Software - Eric Evans
- Implementing Domain-Driven Design - Vaughn Vernon
- Domain Modeling Made Functional: Tackle Software Complexity - Scott Wlaschin