Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Single **R**esponsibility **P**rinciple
High Cohesion

A method should do only ONE thing.
A class should have only 1 reason to change..?

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Don't Repeat Yourself

Never duplicate logic (Capital Sin).
Extract-and-Invoke.
Antonym: **W**rite **E**verything **T**wice.

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Keep It Short & Simple

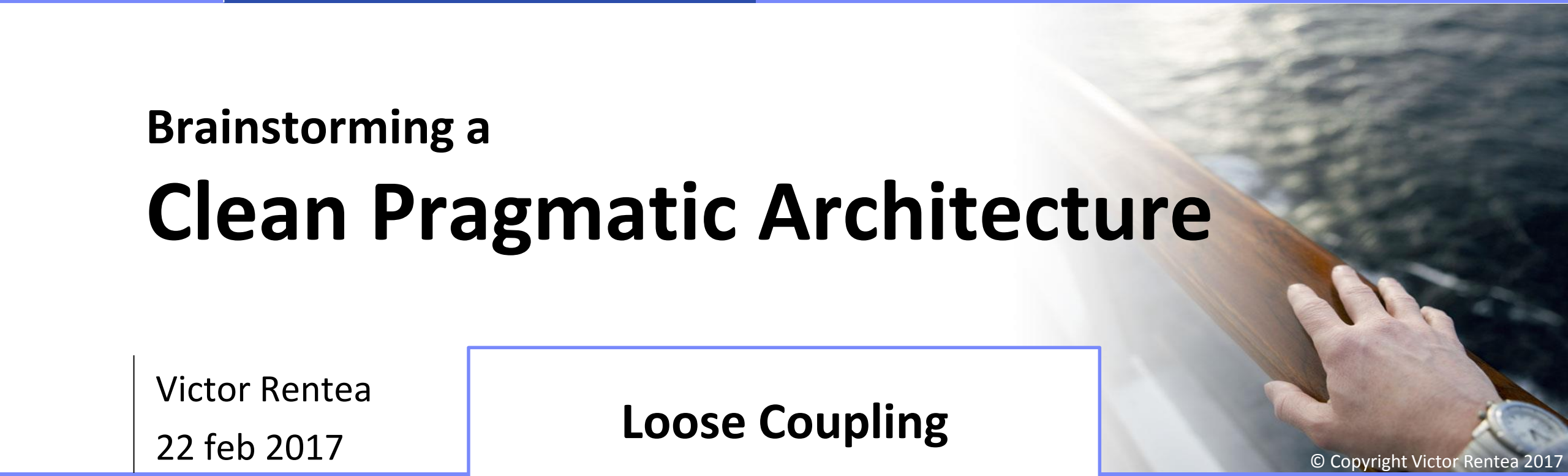
Keep It Simple, Stupid - *US Navy*

Systems work best if kept simple.
Avoid overengineering at all costs.

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training



Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Loose Coupling

Classes with few dependencies.

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

You Ain't Gonna Need It

Extreme Programming

Don't add functionality until deemed necessary.

Implement things when you actually need them,
NOT when you just foresee that you need them.

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Yoda Conditions

```
if ("a".equals(param)) {
```

Avoids NPE. Equals FEAR?

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

A TRAINING BY VICTOR RENTEA

Enterprise Java Training

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Favor Composition Over Inheritance

GoF

A `extends` B is bad.

Use composition: `a.setB(b);`

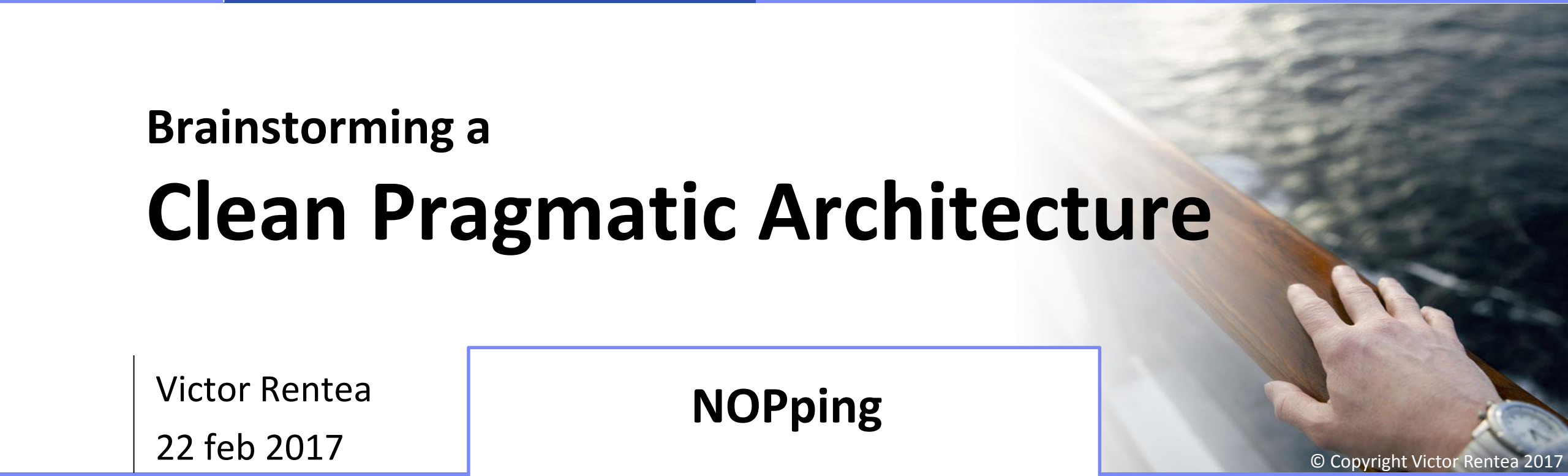
JPoint

VictorRentea.ro

victor.rentea@gmail.com

@victorrentea

Enterprise Java Training



Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

NOPping

Stanislav sat watching the screensaver and nopped for a while.

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Pair Programming

Two programmers working together on one PC.
The ***driver*** writes code, the ***navigator*** reviews it.
They switch often.
Is cheaper on long-run.

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

Dependency **I**nversion **P**rinciple

Abstractions should not depend on details.
Details should depend on abstractions.

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brainstorming a

Clean Pragmatic Architecture

Victor Rentea

22 feb 2017

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Victor Rentea

Consultant, Technical Lead, PhD(CS)

Lead Architect at IBM Romania

Clean Code Evangelist

International Speaker

DEVOXX MOROCCO · JEE Conf · INTERNET & MOBILE WORLD 2016

VOXXED DAYS VIENNA · VOXXED DAYS BUCHAREST · VOXXED DAYS BELGRADE

A TRAINING BY VICTOR RENTEA — Rentea

Night Job ☺:

Trainer & Coach

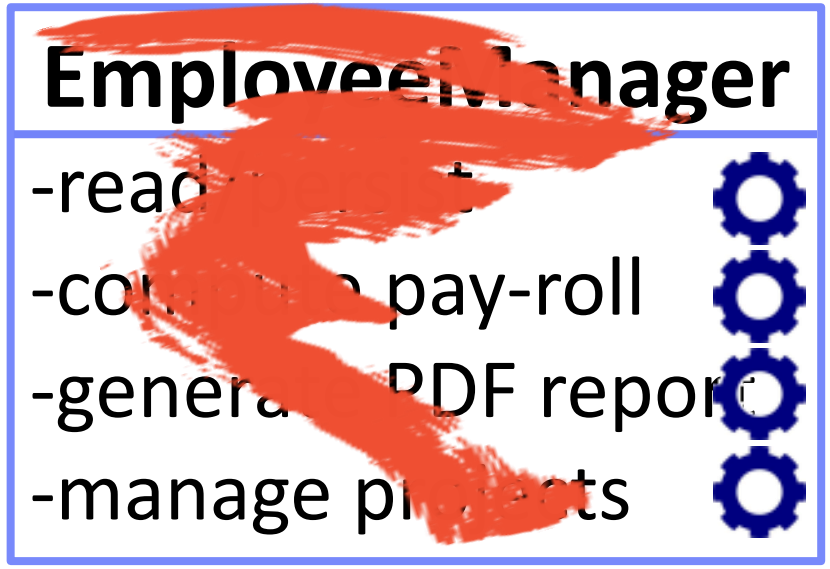
- Spring and Java EE™
- Clean Code, Design Patterns
- TDD, Coding Dojos
- Java Performance, etc

victorrentea@gmail.com @victorrentea VictorRentea.ro

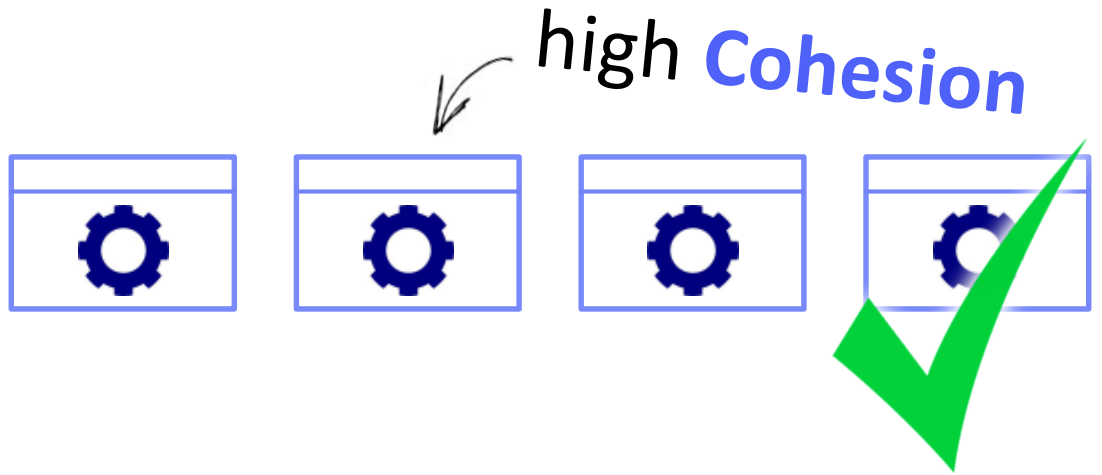
Agenda

- [ ] Driving Principles – KISS
- [ ] Modeling Data – Enemy data
- [ ] Organizing Logic – Extract when it Grows
- [ ] Clean Architecture – The Onion
- [ ] Tests. Fear.

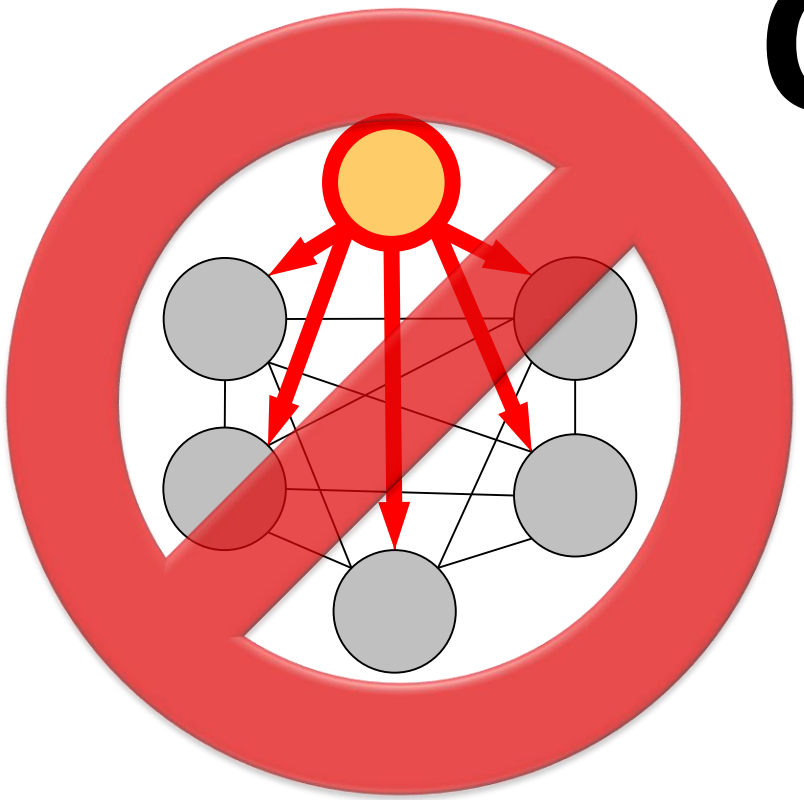
Single Responsibility Principle



vs

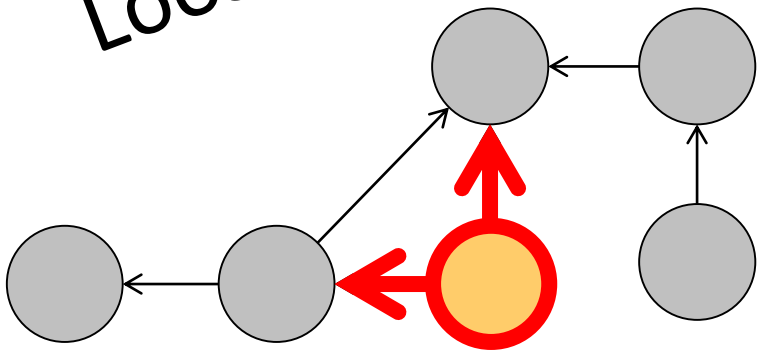


Coupling



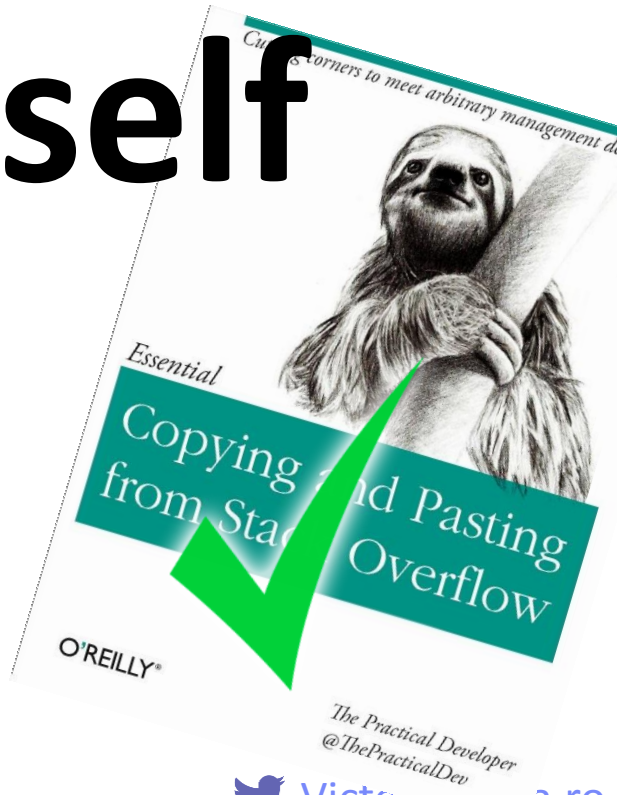
vs

Loose Coupling



Read more: https://dzone.com/articles/probably-the-best-package-structure-in-the-world

Don’t Repeat Yourself



Keep It Short & Simple

Overengineering *is the root of all evil*

– Adam Bien

Less, simple code ⇨ Developer Happiness™ ☺

You Ain't Gonna Need It

Overengineering is the root

Less, simple code ⇨ Developer Happiness™ ☺

YAGNI

Keep It Short & Simple

Overengineering *is the root of all evil*

Less, simple code ⇨ Developer Happiness™ ☺

Protect the Developers

Invisible Magic to reduce effort and risk

Global Exception Handlers

Aspects

Code generators

...

Spring/CDI extensions

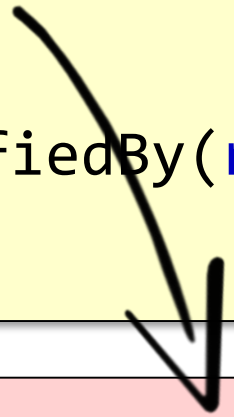
Request/Thread Scope

Bytecode enhancement

Avoid (yet another)
Custom Framework
(to learn) ☹

Request/Thread Scope

```
@Autowired
private MyRequestContext requestContext;
... {
  entity.setModifiedBy(requestContext.getCurrentUser());
}
```



```
@Component
@Scope(value = "request", proxyMode = TARGET_CLASS)
class MyRequestContext { ... }
```

Protect the Developers

Fear Kills **Creativity** ⇨ Simplify Unit Testing

- Strong regression defense

Always Think

Regular Brainstorming

Where to implement the domain logic ?

Transaction Script

Procedures working with **Anemic Entities**

- Map easily to real-world business procedures

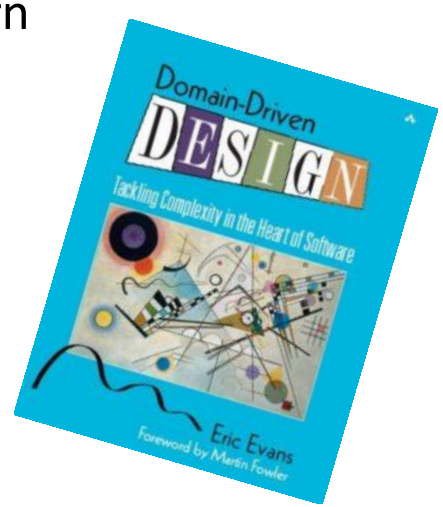
➔So **I** separate logic from data

or

Domain Driven Design

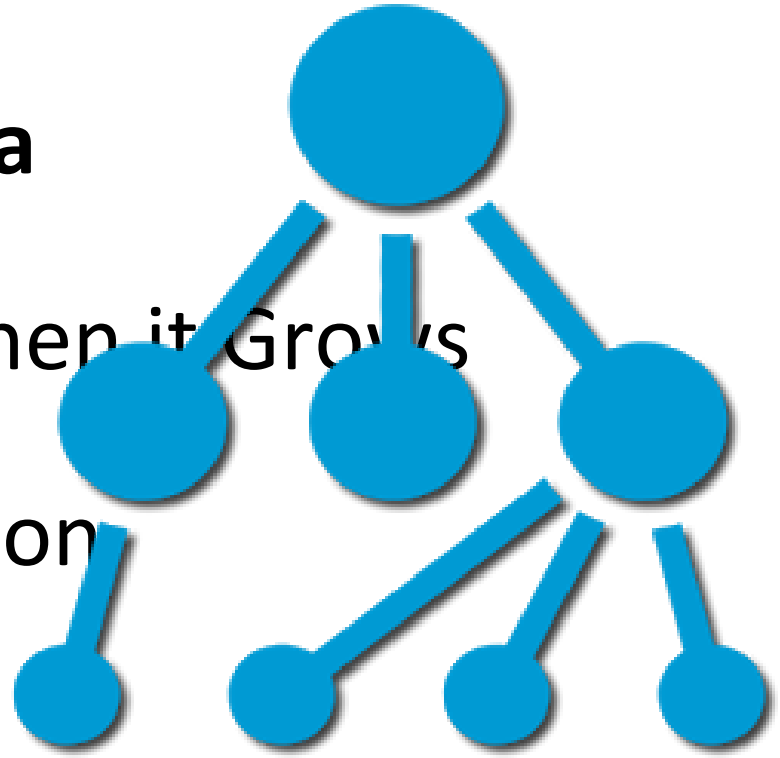
Rich Entities (OOP) + many XyzServices

- Requires deep business involvement
- Promises easier maintenance on the long-run
- Harder to learn



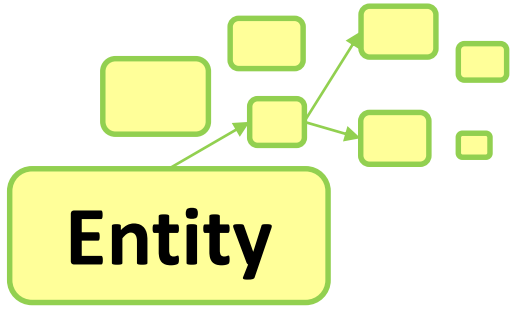
Agenda

- [x] Driving Principles – KISS
- [ ] **Modeling Data – Enemy data**
- [ ] Organizing Logic – Extract when it Grows
- [ ] Clean Architecture – The Onion
- [ ] Tests. Fear.



Entities hold **your** persistent data

You control them!



Use them to **simplify** the implementation of your domain logic

Put small bits of highly reusable domain logic in your

Domain Entities

no setters

```
public class Customer {
  // fields, getters and
  public String getFullName() {
    return firstName + " " + lastName;
  }

  public void activate(User user) {
    if (status != Status.DRAFT) {
      throw new IllegalStateException();
    }
    status = Status.ACTIVE;
    activatedBy = user;
    activatedDate =
  }

  public boolean isActive() {
    return status == Status.ACTIVE;
  }

  public boolean canPlaceOrders() {
    return status == Status.ACTIVE && !isBan
  }
```

Synthetic getters

Some validations

Set audit fields

Lambda helpers (Java8)

`filter(Customer::isActive)`

Put small bits of highly reusable domain logic in your

Fit Domain Entities


```
public String toExportString() {
  return String.format("%s;%s;%d",
    firstName, lastName, isActive()?1:0);
}
```

```
    activatedBy = user;
    activatedDate = new
  }

  public boolean isActi
    return status ==
  }

  public boolean can
    return status ==
  }

  public void addAddress(Address
    address.setCustomer(this);
    addresses.add(address);
  }

  public List<Address> getAddresses() {
    return Collections.unmodifiableList(
                                addresses);
  }
}
```

Synthetic getters

Some validations

Set audit fields

Lambda helpers (Java8)

filter(Customer::isActive)

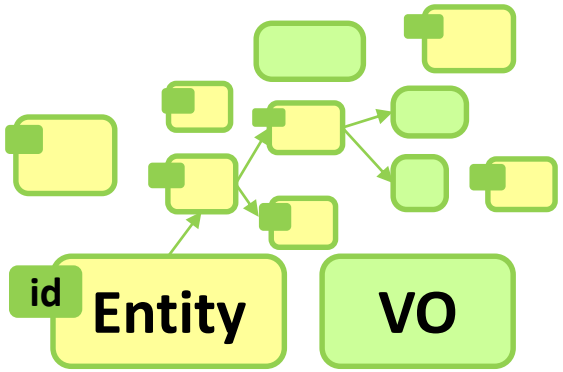
ORM links

(always set the owner side)

~~parent.getChildren().add(child);~~

Value Object: grouping of **domain data** that move together

- **Small**: Developer Comfort™
- **Immutable**: Developer Safety™
- **Transient**: no persistent ID/PK (vs Entity)




```
public class Money {
  private final Currency currency;
  private final BigDecimal amount;

  public Money(Currency currency, BigDecimal amount) {
    this.currency = currency;
    this.amount = amount;
    validate();
  }

  public Currency getCurrency() { return currency; }

  public BigDecimal getAmount() { return amount; }
  public boolean equals(Object other) { ... }
}
```

BASED ON ALL FIELDS

Then, you start building a User Interface (REST, JSF, …)

But you quickly realize UI is your enemy.

They want data structures to match the screens.

Their goal is different.

Customer Name	Last Order Date	Region	Total Orders Price	Responsible
John Doe	2016-01-01	EMEA	1,654 RON	Angajat 1235
Jane Doe	2017-02-02	US	510 RON	Alt Angajat22

Order.isDeletable:Boolean

to decide to show/hide the Delete button

Never expose your Entities to ***them***.

Instead, give your clients

Data Transfer Objects

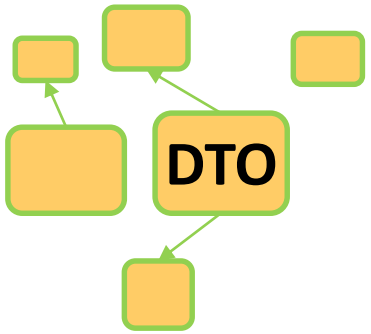
Bare data structures

e.g. Form/Request →

e.g. View/Response ←

e.g. DTO ⇆

e.g. SearchCriteria/SearchResult




```
public class CustomerDto {
  private String fullName;
  private String phoneNumber;
  private Date birthDate;

  public final String getFullNam
    return fullName;
  }
  public final void setFullName(
    this.fullName = fullName;
  }
  public final String getPhoneNu
    return phoneNumber;
  }
  public final void setPhoneNumb
{
    this.phoneNumber = phoneNumb
```

```
public class CustomerDto {
  public String fullName;
  public String phoneNumber;
  public Date birthDate;
}
```

I personally like them to be dumb

- **public** fields ?! 

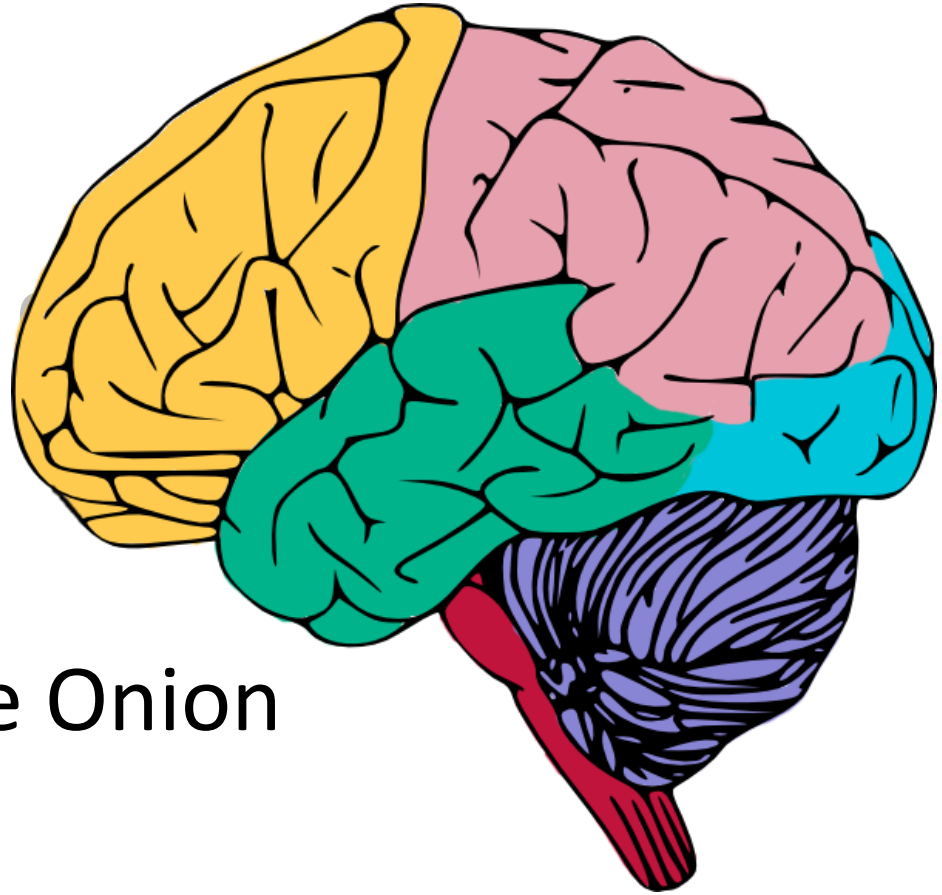
!!.. But why not? It's a **struc**

Why? You'll see soon…

```
dto.fullName = customer.getFullName();
```

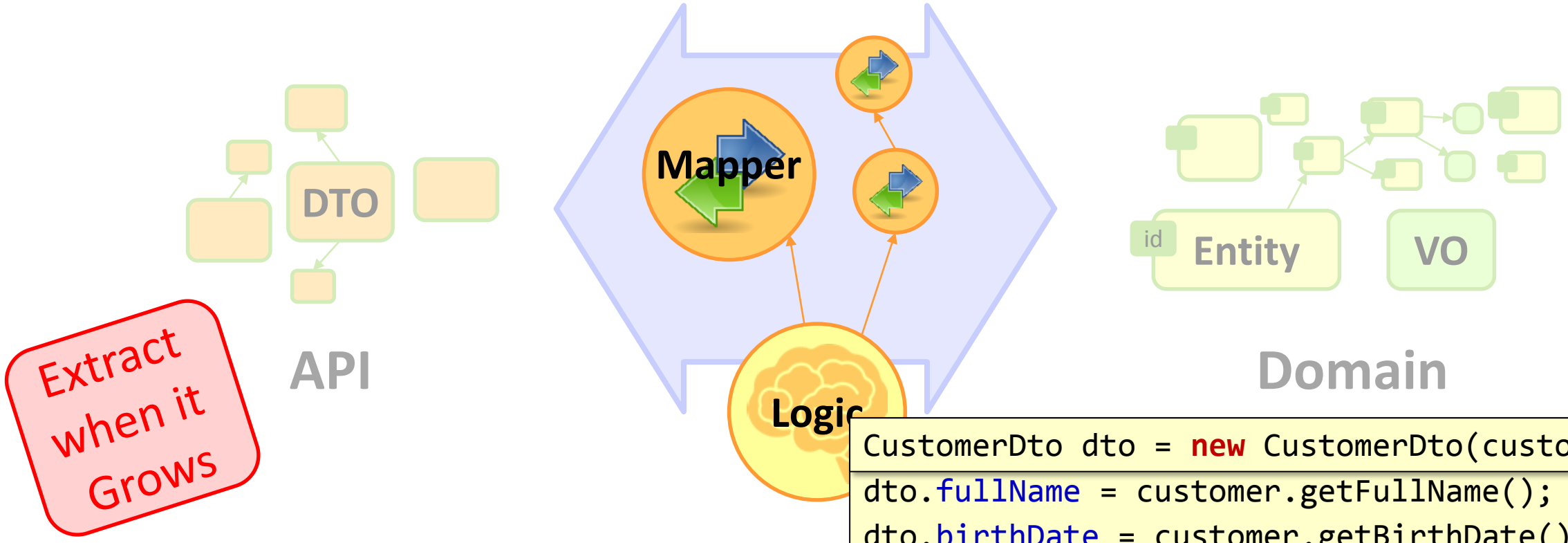
Agenda

- [x] Driving Principles – KISS
- [x] Modeling Data – Enemy
- [ ] **Organizing Logic**
- [ ] Clean Architecture – The Onion
- [ ] Tests. Fear.



When DTO ⇔ Entity conversion is complex or too long

Extract Mappers to clean up the domain logic




```
CustomerDto dto = new CustomerDto(customer);
dto.fullName = customer.getFullName();
dto.birthDate = customer.getBirthDate();
dto.phoneNumber = customer.getPhoneNumber();
```

Mappers go to DB ?

KISS: Yes !

```
public Customer toEntityForCreate(CustomerDto
  Customer customer = new Customer();
  customer.setBirthDate(dto.birthDate);
  customer.setGroup(groupRepo.getReference(dto.groupId));...
  return customer;
}

public CustomerDto
  CustomerDto dto = r                    {
  dto.birthDate = custome getBirthDate();
  for (Address a:customer.getAddresses()) {
  for (Address a:addressRepo.getByCustomer(customer.getId())){
  ...
  }
  return dto;
}
}
```

Actually, no DB call

Link to DB entities

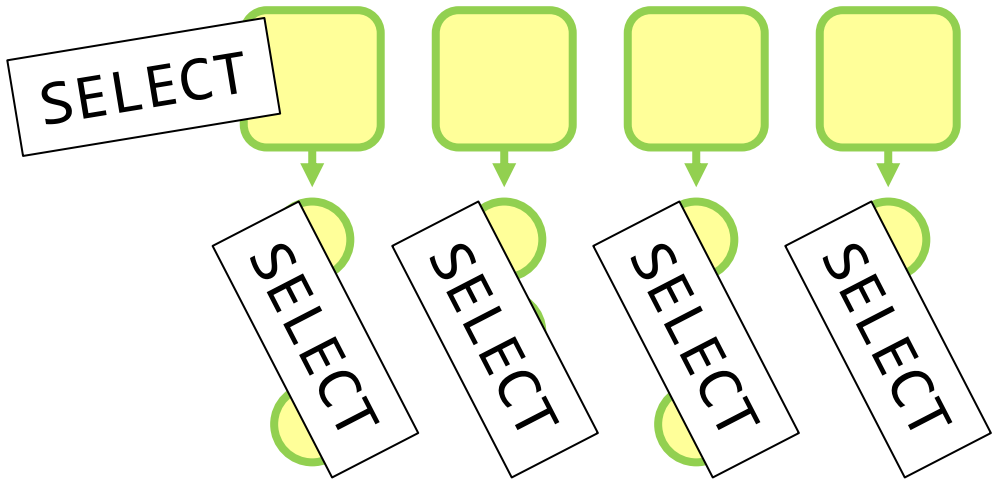
N+1 Queries

Lazy-load with ORM

Load more data from DB

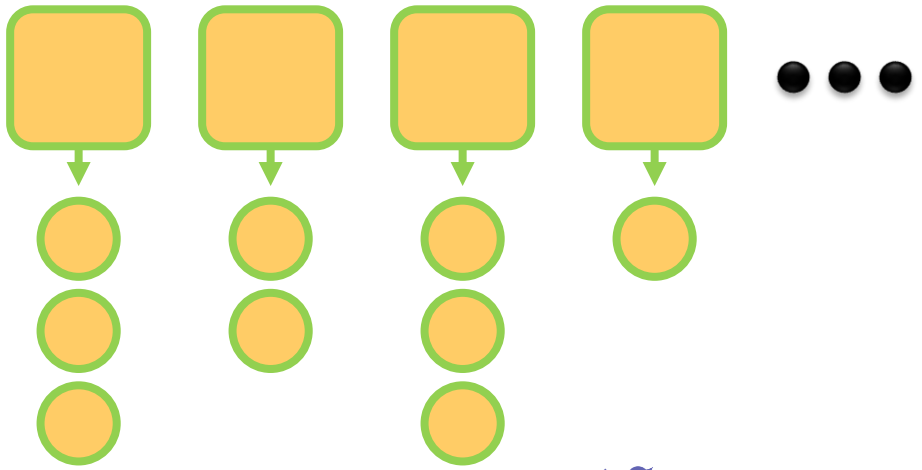
Explicit load w/o ORM

```
… WHERE ADDR.CUSTOMER_ID = ?
```



1001 queries,
coming up !

N+1 Queries



```
public Customer toEntityForCreate(CustomerDto
  Customer customer = new Customer();
  customer.setBirthDate(dto.birthDate);
  customer.setGroup(groupRepo.getReference(dto.groupId));...
  return customer;
}

public CustomerDto toDto(Customer customer) {
  CustomerDto dto = new CustomerDto();
  dto.birthDate = customer.getBirthDate();
  for (Address a:customer.getAddresses()) {
  for (Address a:addressRepo.getByCustomer(customer.getId())){
  ...
  }
  return dto;
}
}
```

Link to DB entities

Actually, no DB call

Load more data from DB

Lazy-load with ORM

Explicit load w/o ORM

```
… WHERE ADDR.CUSTOMER_ID = ?
```

```
public List<CustomerDto> search(CustomerCriteria criteria) {
  List<Customer> customers = customerRepo.search(criteria);
  return customers.stream()
        .map(customerMapper::toDto)
        .collect(toList());
}
```

N+1 Queries

(you know the solution)

Don’t anticipate.

Keep things simple.

➔CLEAN CODE ➔

```
public CustomerDto toDto(Customer customer) {
  CustomerDto dto = new CustomerDto();
  for (Address a:customer.getAddresses()) {
  for (Address a:addressRepo.getByCustomer(customer.getId())){
  return dto;
}
```

JPQL: LEFT JOIN FETCH customer.addresses

SQL: WHERE ADDR.CUSTOMER_ID IN (?,?,…)

Performance ?

Measure, Don’t Guess®

Premature ***optimization*** *is the root of all evil*

– Donald Knuth

Abused DTOs are reused in different use cases

- But some fields are not used in some other cases... Which ones ?!..

Ignored for create use-case

```
public class CustomerDto {
  public String fullName;
  public Date birthDate;
  public Long groupId;

  public UserTime creation;
  public UserTime modification;
}
```

Strict DTOs don't have useless fields

- How to reuse them ?
- Favor Composition over Inheritance[GoF] (**extends** is bad)

DRY

```
public CustomerDto customer;
```

```
public class CustomerDto {
  public String fullName;
  public Date birthDate;
  public Long groupId;
}
```

```
public class CustomerView extends CustomerDto{
  public UserTime creation;
  public UserTime modification;
}
```

You are cheating

when’re DRYing code based on a fragile coincidence

(on the client side, the data might be interpreted independently)

You are cheating

when're DRYing code based on a fragile coincidence

```
double price = 2;
if (daysRented > 2)
    price += (daysRented - 2) * 1.5;
return price;
```

Pure coincidence.
Different business meaning.

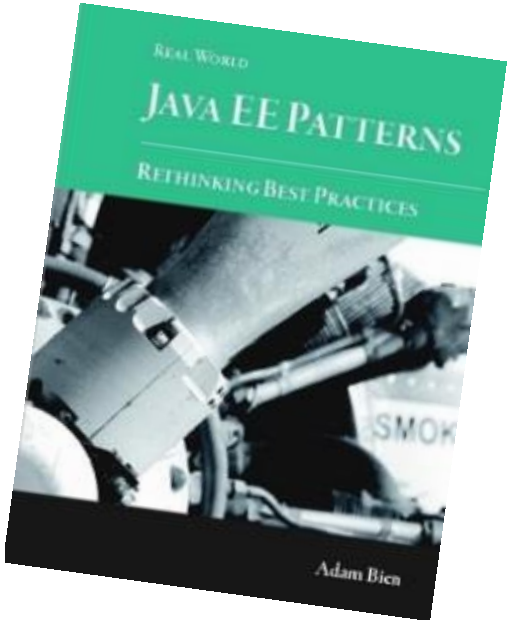
Don't make
a constant here

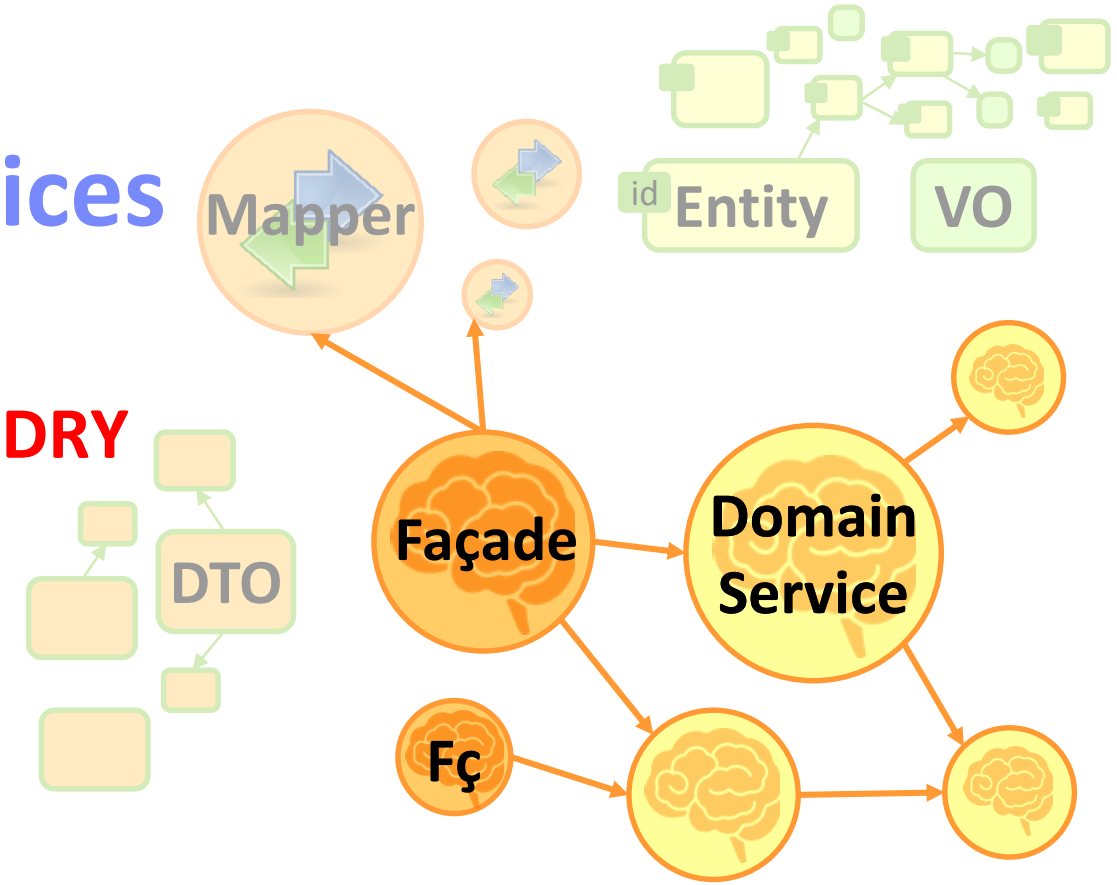
Extract constants ONLY to explain the logic

Start implementing domain logic in a Facade

Extract logic into Domain Services

- To hide complexity – **SRP / KISS**
- For Reuse (across Facades or Services) - **DRY**





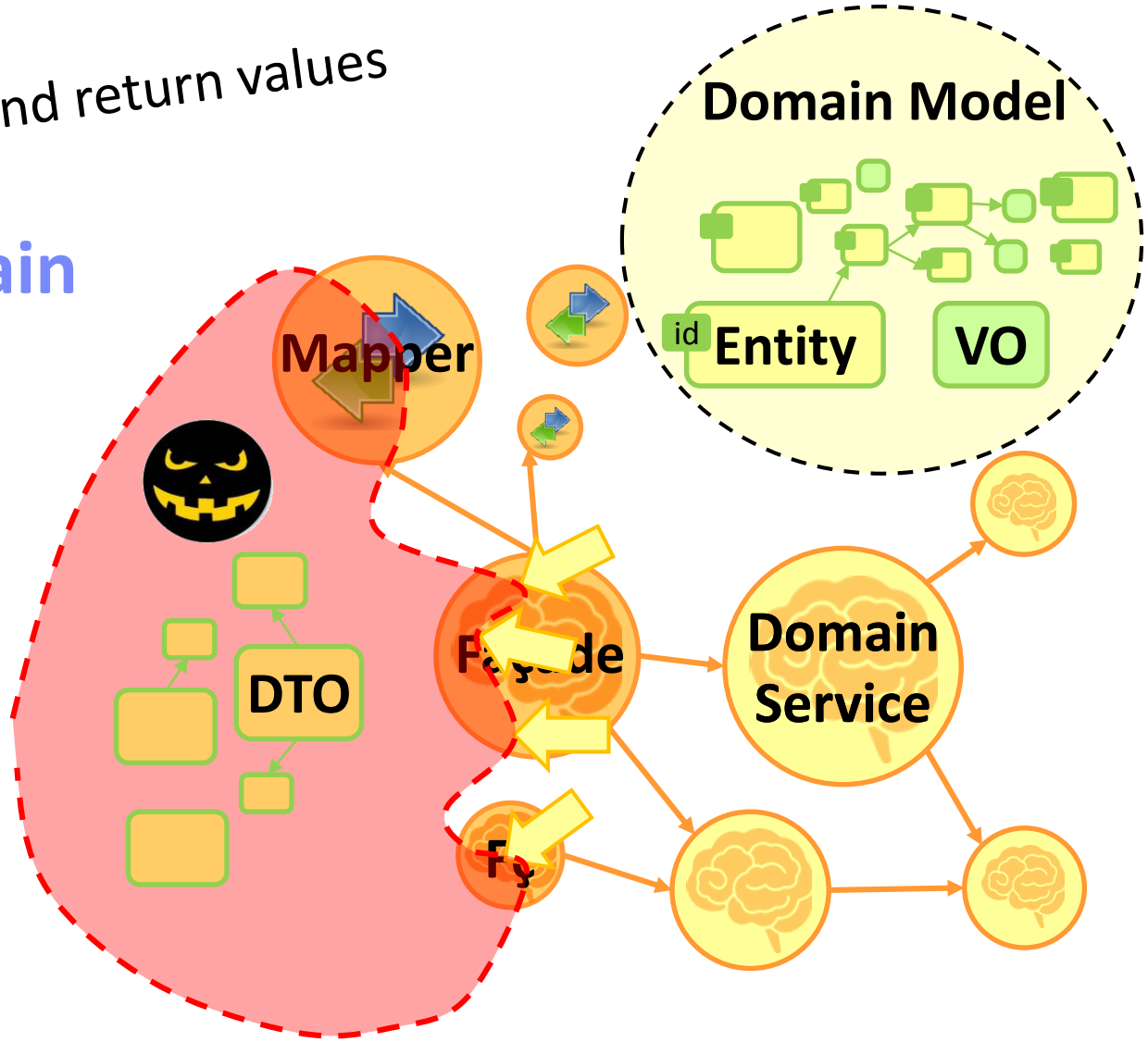
parameters and return values

Domain Services speak Domain Model

Fragile, under enemy control

Keep DTOs out of your logic !

➔Convert them to ***your*** Domain



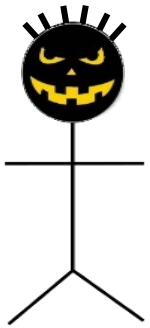
* That's why I like dumb DTOs

Facade Roles

Aspects

- Transaction
- Logging
- Global Exception Handling*

Even earlier, in Controller?



DTO

Extract when it Grows

Convert Data

Mapper

Façade

id Entity

VO

Domain Service

Fc

Validate Data

Validator

Implement Logic

What do you mean ?

How?

Huh? If I find a good name, I extract? That's it?

Piece a cake!

Extract when it Grows

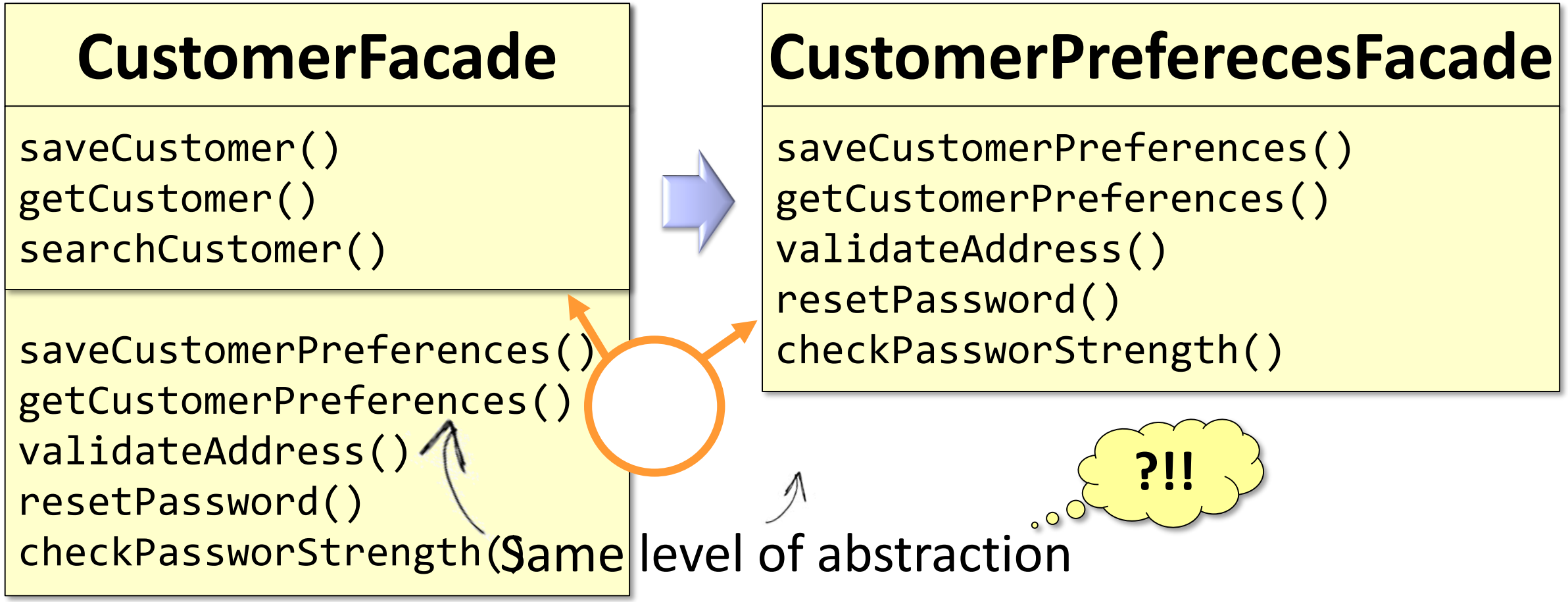
When a class grows too big (>~200 lines?)
➔ break it

Look for a good class name to group some of its methods

Exactly!

He-he!☺
*"There are only two things **hard** in programming: Cache Invalidation and **Naming Things**"*

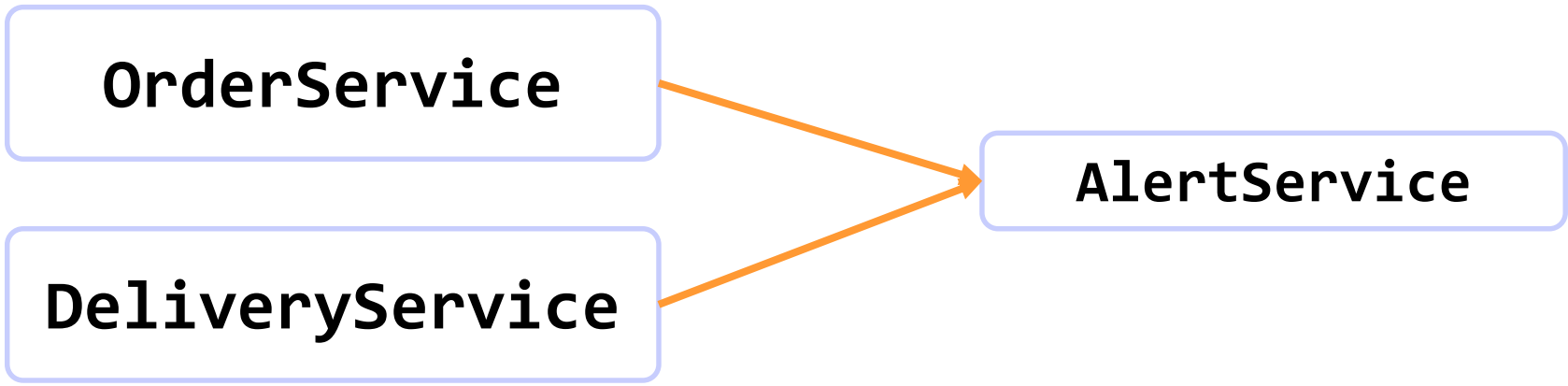
Extract when it Grows



Separation by Layers of Abstraction

Extract for Reuse

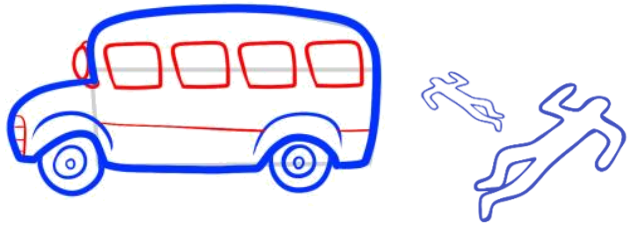
DRY



In L/XL applications: hard to feel opportunities

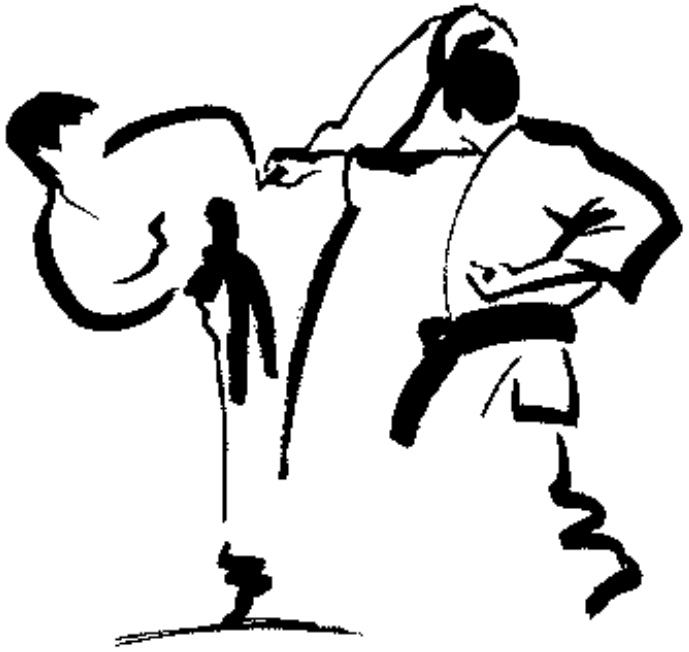
➔ **Pair programming**, design brainstorming

➔ Increase Bus Factor =2



Continuous Practice

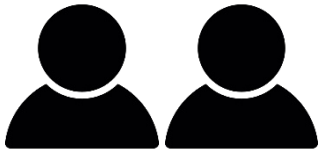
Coding Katas



Refactoring Hackaton (throw-it-away)

Offensive Refactoring

Coding Dojo



Pair Programming

DEVELOPER COMFORT™
IS ESSENTIAL FOR
EMERGING ARCHITECTURES

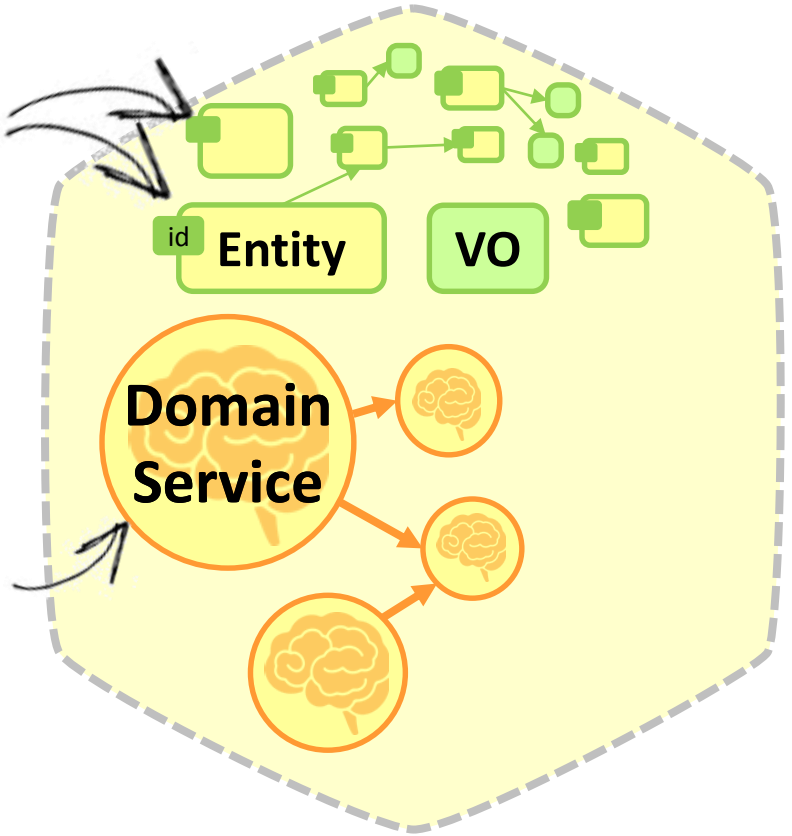

I'M A
TIME, LET'S
SUME I'M

Agenda

- [x] Driving Principles – KISS
- [x] Modeling Data – Enemy data
- [x] Organizing Logic – Extract when i
- [ ] **Clean Architecture – The Onion**
- [ ] Tests. Fear.

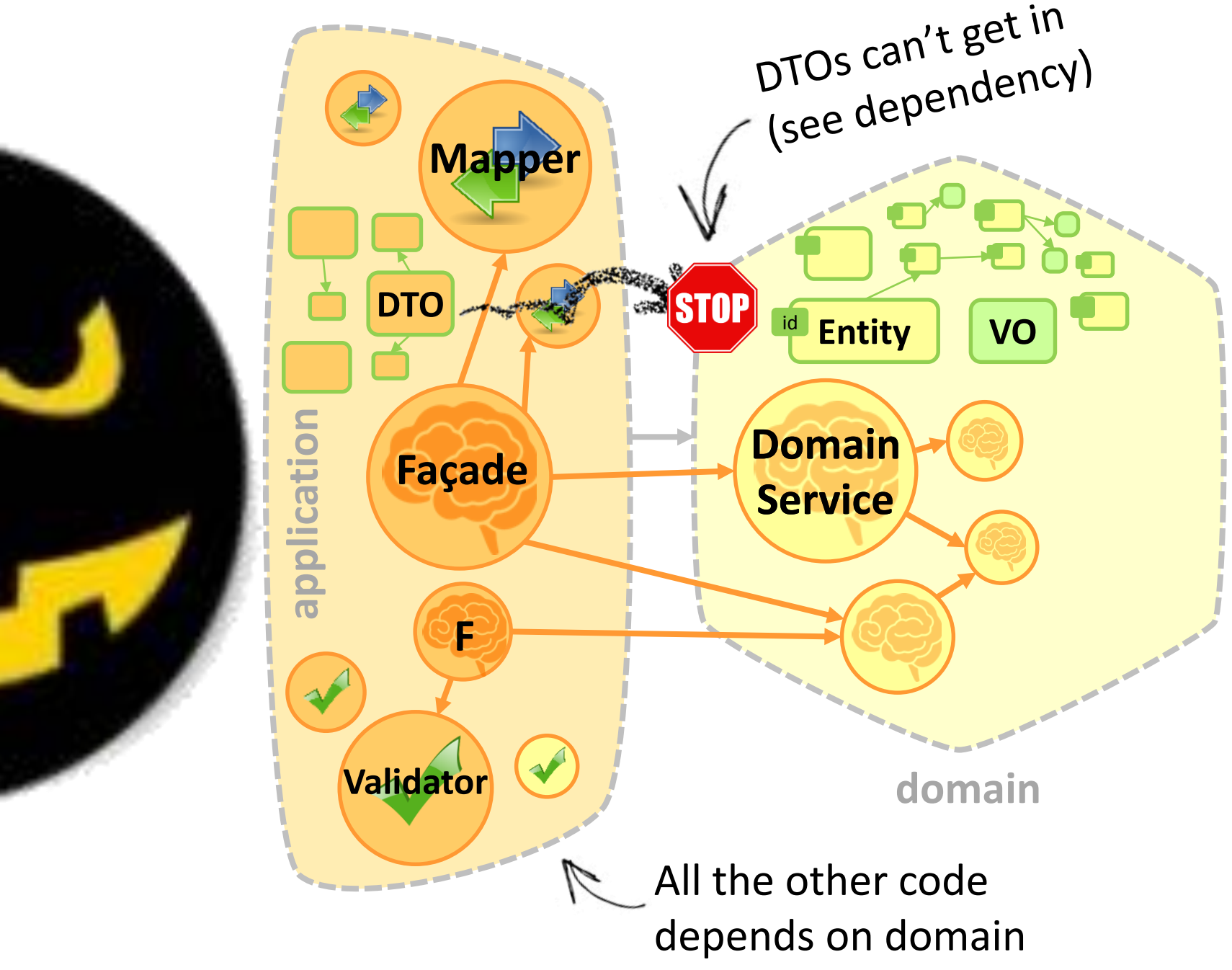
What code would you protect?

Domain Objects



Priceless
Domain Logic

Put it in the **domain** module


DTOs can't get in
(see dependency)
Mapper
DTO
STOP
id
Entity
VO
application
Façade
Domain
Service
F
Validator
domain
All the other code
depends on domain

domain

Domain Service

External Service

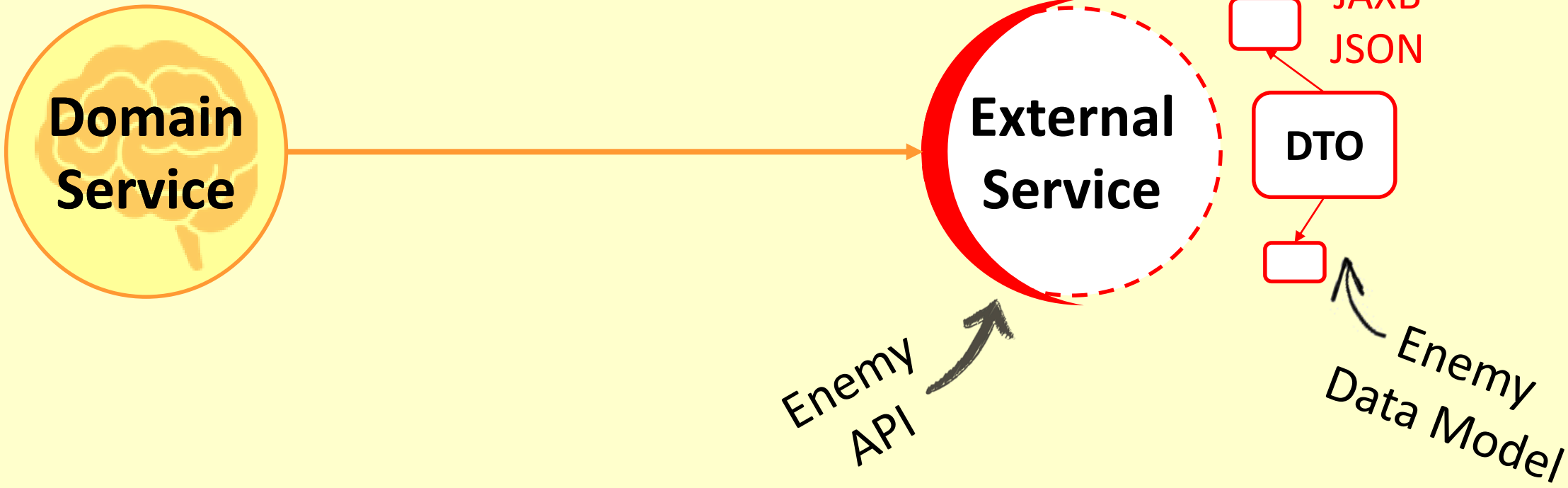
JAXB
JSON

DTO

Enemy API

Enemy Data Model

domain


Domain Service
External Service
JAXB
JSON
DTO
Enemy API
Enemy Data Model

domain

Domain Service

External Service

JAXB
JSON

DTO

ExtServ Adapter

Is what ***your*** Domain Logic ***needs*** from the external service

Anti-Corruption Layer
(A huge Adapter Pattern[GoF])

External Service Adapter

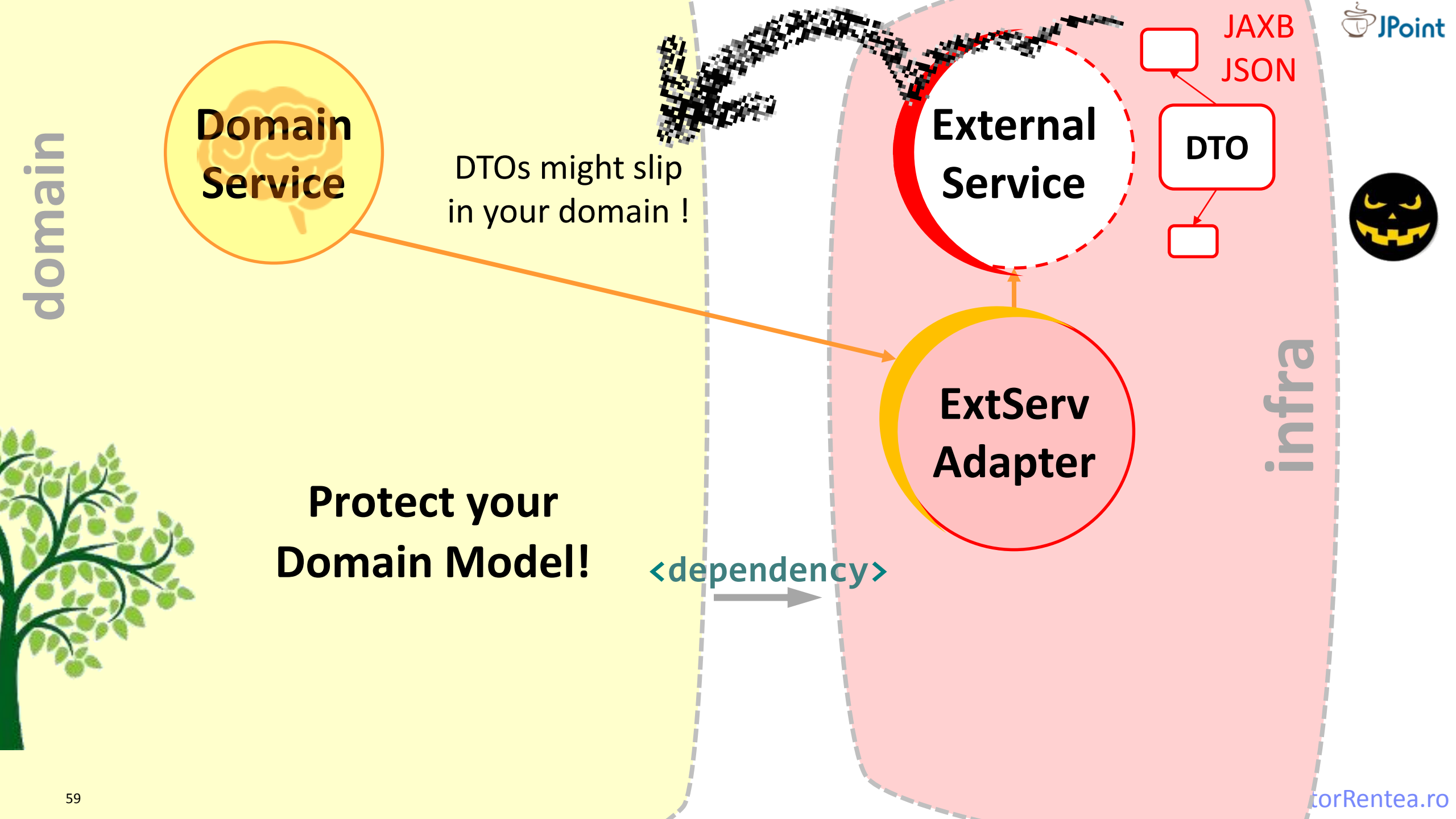
```
public class LDAPUserServiceAdapter {
  private LdapContext ctx;
  ...
  public LdapUserSearchResult findAccountByAccountName(String ldapSearchBase, String accountName) {
    try {
      String searchFilter                    r)(sAMAccountName={1}))";
      SearchControls searchControls = new SearchControls();
      searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);

      NamingEnumeration<SearchResult> results = ctx.search(
          ldapSearchBase, searchFilter, new Object[]{accountName} searchControls);

      if (!results.hasMoreElements())
        return null;
      }
      SearchResult searchResult = (SearchResult) result
      if (results.hasMoreElements()) {
        throw new MyAppException(ErrorCode.DUPLICATE_USE
      }
      return new LdapUserSearchResult(searchResult);
    } catch (NamingException e) {
      throw new MyAppException(ErrorCode.LDAP_LOOKUP_FAIL
    }
  }
}
```

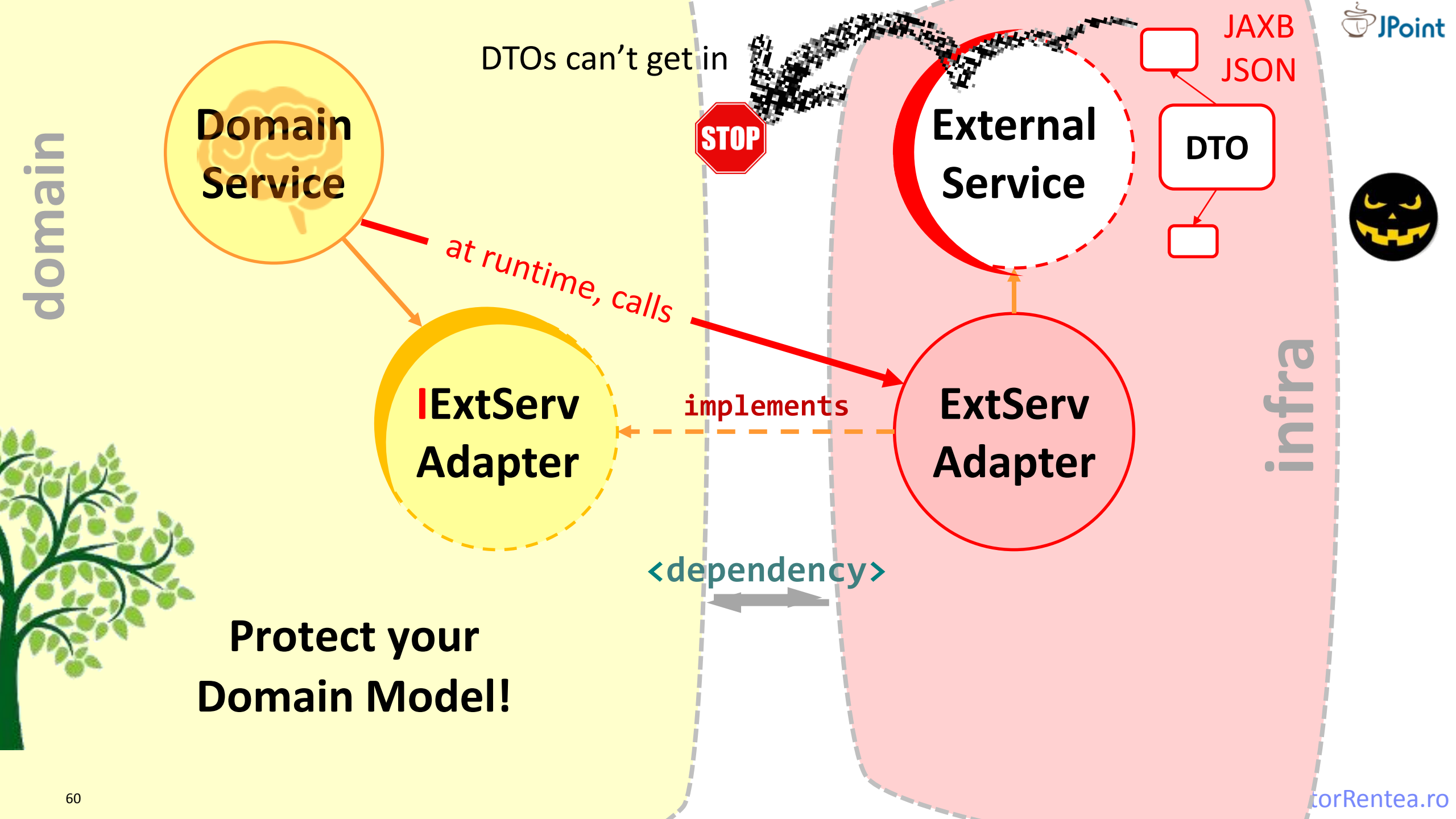
my VO vs *their* DTO

Anti-Corruption Layer

(A huge Adapter Pattern[GoF])

- Hide ***their*** ugly API
- Handle/wrap ***their*** Exceptions
- Decouple Entities ⇔ ***their*** API data model
- Validate any evil data ***they*** send to you



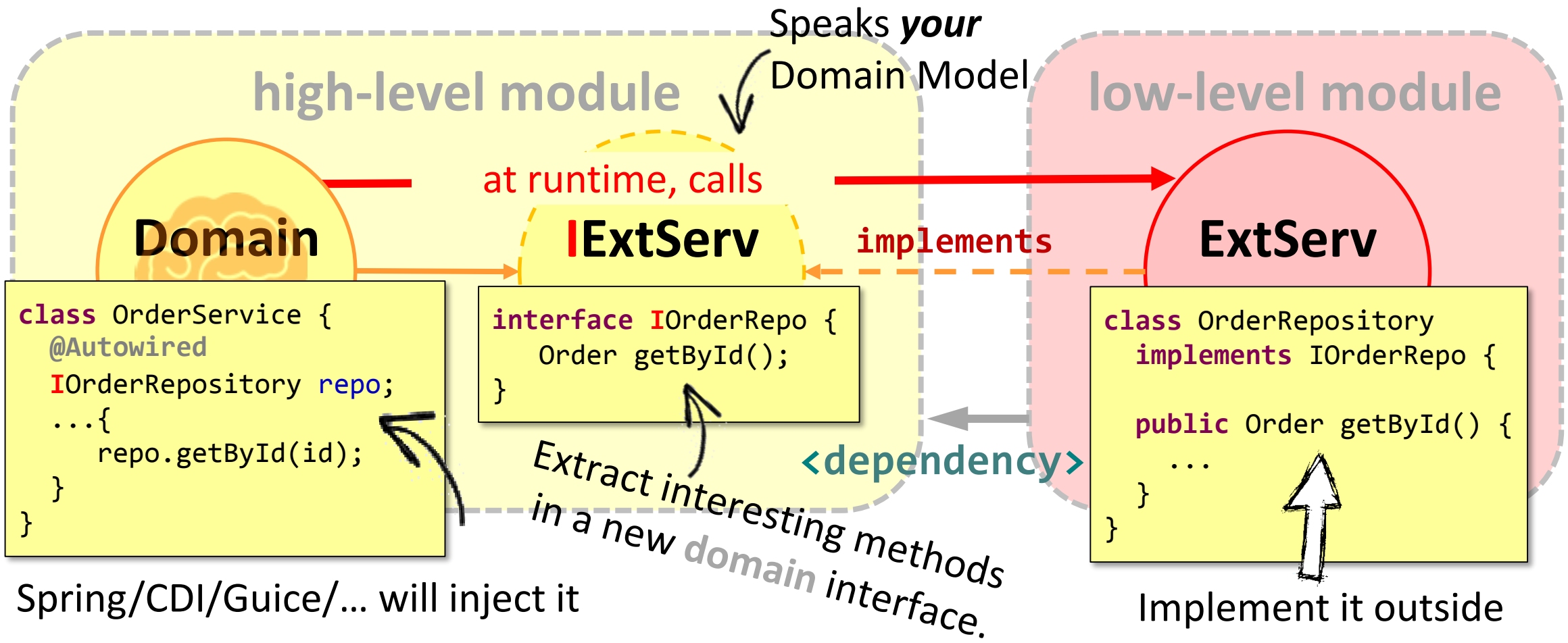


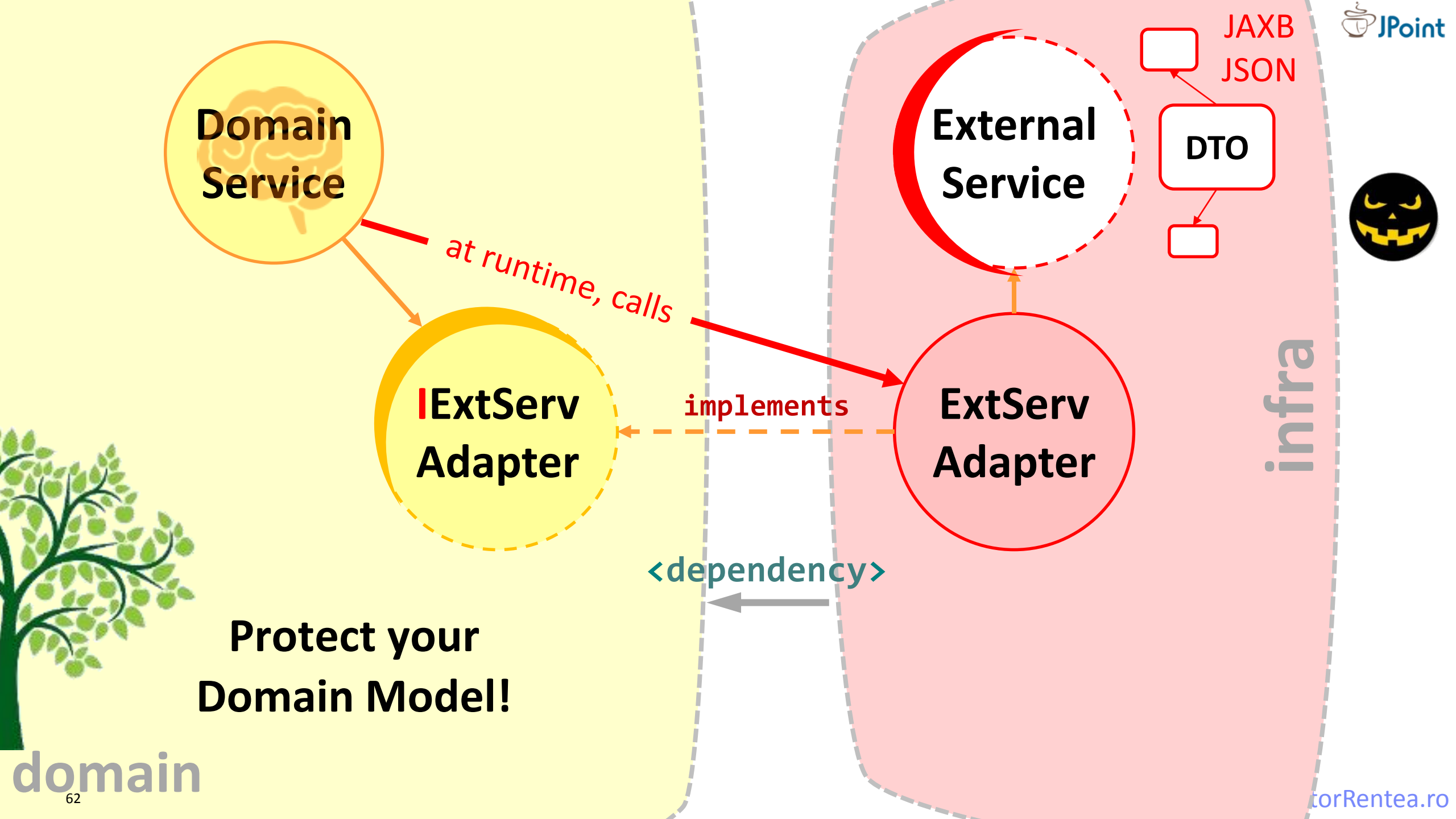
Dependency Inversion Principle

Best of OOP - Uncle Bob

JPoint

Higher level modules should not depend on lower-level modules




Domain Service
at runtime, calls
IExtServ Adapter
implements
ExtServ Adapter
External Service
JAXB
JSON
DTO
infra
<dependency>
Protect your Domain Model!
domain

DTOs can't get in

STOP

External Service

JAXB
JSON

DTO

IExtServ Adapter

implements

ExtServ Adapter

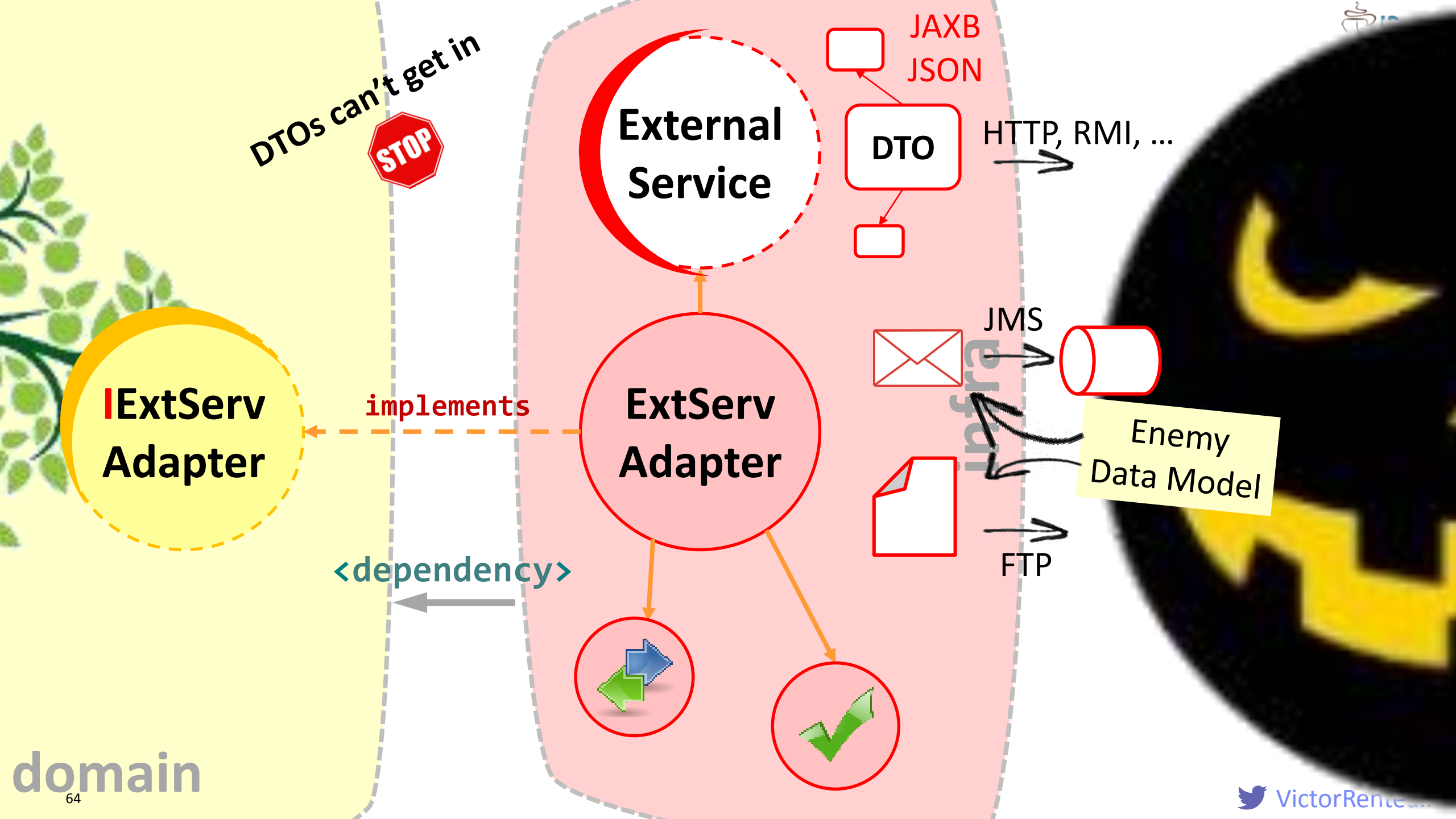
<dependency>

infra

domain

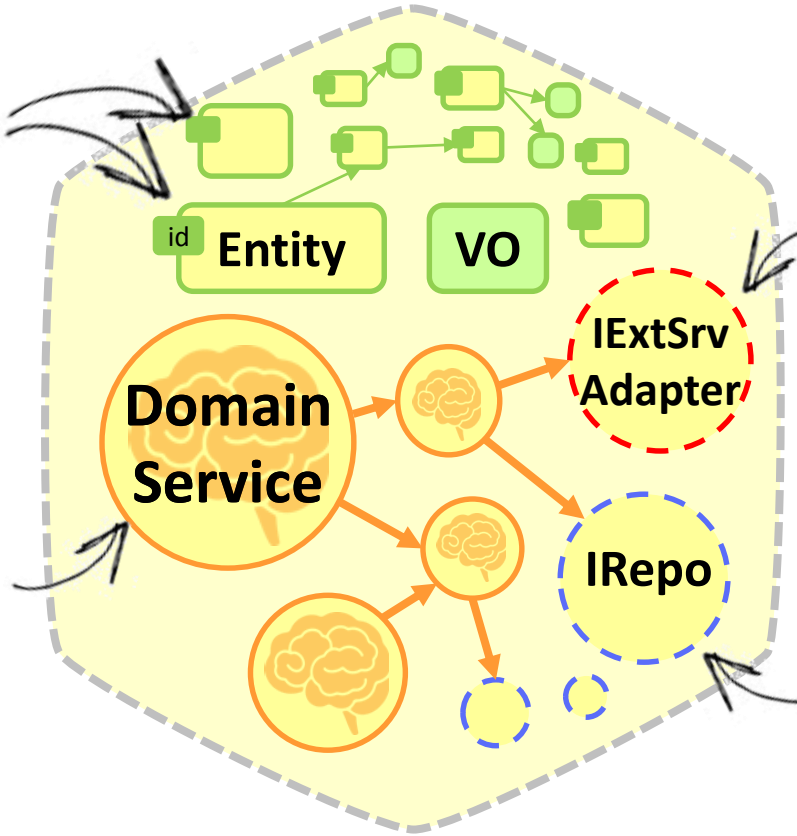
A huge Adapter Pattern GoF

- Hide ***their*** ugly API
- Handle/re-throw ***their*** Exceptions
- Decouple Entities ⇔ ***their*** API data model
- Validate any evil data ***they*** send to you



What code would you protect?

Domain Objects



Interfaces for
External Services
you consume

Priceless Domain Logic

Interfaces for
Repositories
(DB access)

Put it in the **domain** module

Behold,

The Architecture

a.k.a. Clean, Hexagonal, Ports-and-Adapters

Behold,

The Architecture

a.k.a. Clean, Hexagonal, Ports-and-Adapters

Hide the persistence!
Clean your Logic!

```java
public Employee getById(String employeeId) {
  String sql = "SELECT id, name, phone FROM employee WHERE id = ?";
  Connection conn = null;
  try {
    conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, employeeId);
    ResultSet rs = ps.executeQuery();
    if (rs.next()) {
      Employee employee = new Employee();
      employee.setId(rs.getString(1));
      employee.setName(rs.getString(2));
      employee.setPhone(rs.getString(3));
      return employee;
    }
  } catch (SQLException e) {
    // TODO: something, don't know what..
  } finally {
    if (conn != null) {
      try {
        conn.close();
      } catch (SQLException e) {
        // TODO: something, don't know what..
      }
    }
  }
  return null;
}
```

Plain Old JDBC (’90-style)

Hide the persistence!
Clean your Logic!

Plain Old JDBC ('90-style)

```
public Employee getById(String employeeId) {
  String sql = "SELECT id, name, phone FROM employee WHERE id = ?";
  Connection conn = null;
  try {
  }
  return null;
}
```

```
public Employee getById(String employeeId) {
  return jdbcTemplate.queryForObject(
      "SELECT id, name, phone FROM employee WHERE id = ?",
      new RowMapper<Employee>() {
        public Employee mapRow(ResultSet rs, int rowNum)
                      throws SQLException {
          Employee employee = new Employee();
          employee.setId(rs.getString(1));
          employee.setName(rs.getString(2));
          employee.setPhone(rs.getString(3));
          return employee;
        }
      },
      employeeId);
}
```

Spring' JdbcTemplate (or similar)

Hide the persistence!

Clean your Logic!

Plain Old JDBC ('90-style)

```
public Employee getById(String employeeId) {
  String sql = "SELECT id, name, phone FROM employee WHERE id = ?";
  Connection conn = null;
  try {
```

Spring' JdbcTemplate (or similar)

```
  public Employee getById(String employeeId) {
    return jdbcTemplate.queryForObject(
        "SELECT id, name, phone FROM employee WHERE id = ?",
                                                    int rowNum)
        employeeId);
  }
```

```
class IOrderRepository {

  Employee getEmployeeBasicById(int id);
}
```

MyBatis DataMapper

```
<select id="getEmployeeBasicById" parameterType="int" resultType="Employee">
  SELECT id, name, phone_number AS phoneNumber
  FROM EMPLOYEES
  WHERE ID = #{id}
</select>
```

```
  }
  return null;
}
```

Hide the persistence!

Clean your Logic!

Plain Old JDBC ('90-style)

Spring' JdbcTemplate (or similar)

MyBatis DataMapper

```
public Employee getById(String id) {
  List<Employee> list = em.createQuery(
    "SELECT e from Employee e where e.id=:id", Employee.class)
      .setParameter("id", id)
      .getResultList();
  if (list.isEmpty()) {
    return null;
  } else {
    return list.get(0);
  }
}
```

JPA/Hibernate

Hide the persistence!

Clean your Logic!

Plain Old JDBC ('90-style)

Spring' JdbcTemplate (or similar)

MyBatis DataMapper

JPA/Hibernate

Spring Data JPA ... (fine) selection

```
@Repository
public interface EmployeeRepository
    extends JpaRepository<Employee, Integer>

  public Employee getById(Integer employeeId);
  public Optional<Employee> getByName(String name);
  public Long countByName(String name);

  @Query("SELECT e FROM Employee e LEFT JOIN FETCH e.projects")
  public List<Employee> getAllFetchProjects();
}
```

Hmm..
Are these
realy so dirty ?

After all, they walk
the Domain Model.
(JPQL, HQL)

Hide the persistence!
Clean your Logic!

Hide the persistence!
Clean your Logic!

Hide the persistence!
Clean your Logic!

Speaks **your** Domain Model

Hide the persistence! Clean your Logic!

DIP

IRepository

`implements`

calls

Repository implementation

1 repo/Entity

Order Repo

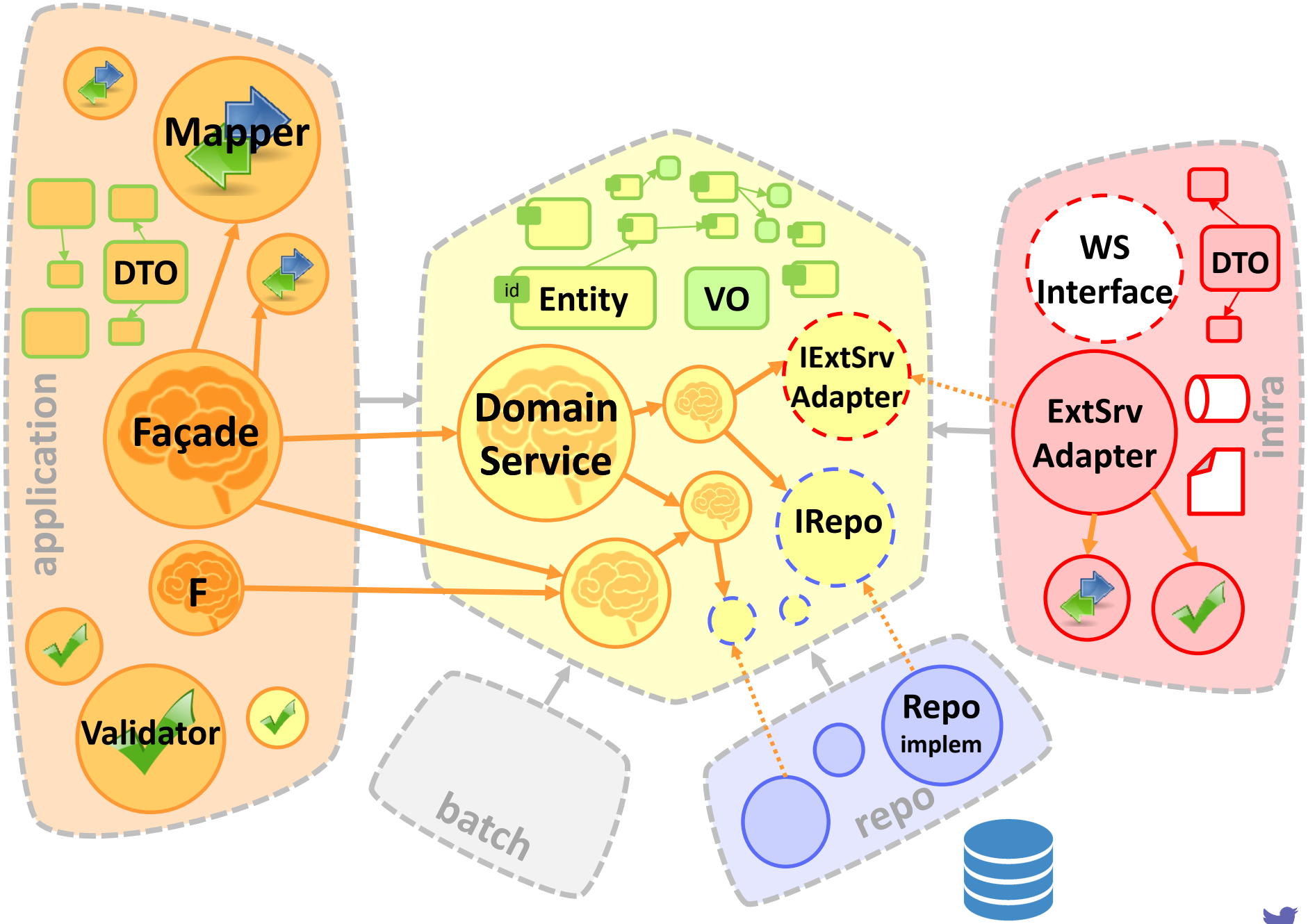
I ♥ Brainless, Boilerplate Repositories

Split them when they grow

Pull the logic into the domain

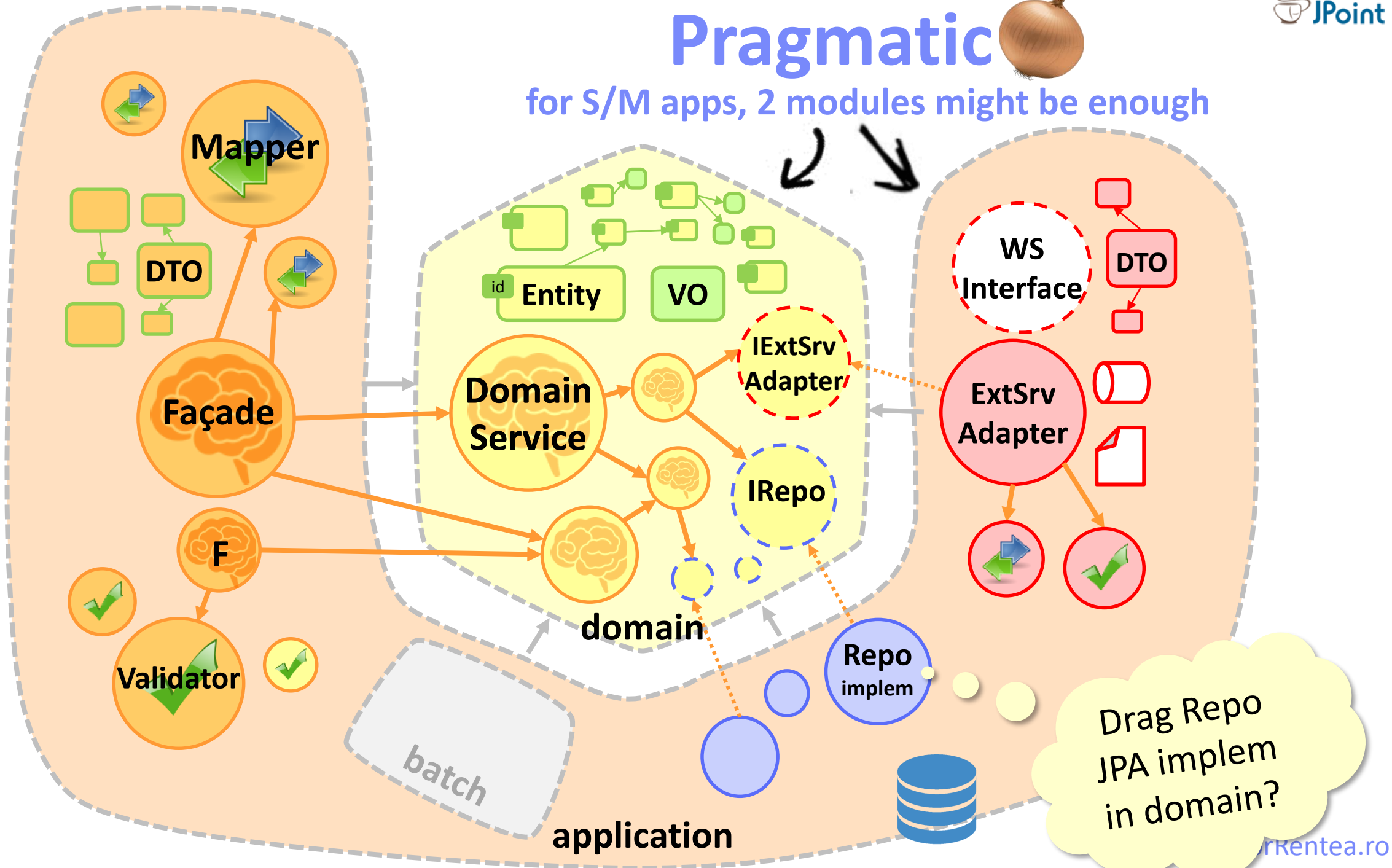
it's a better, Cleaner, Testable world

domain


Mapper
DTO
Façade
F
Validator
application
id
Entity
VO
Domain
Service
IExtSrv
Adapter
IRepo
WS
Interface
DTO
ExtSrv
Adapter
infra
batch
Repo
implem
repo

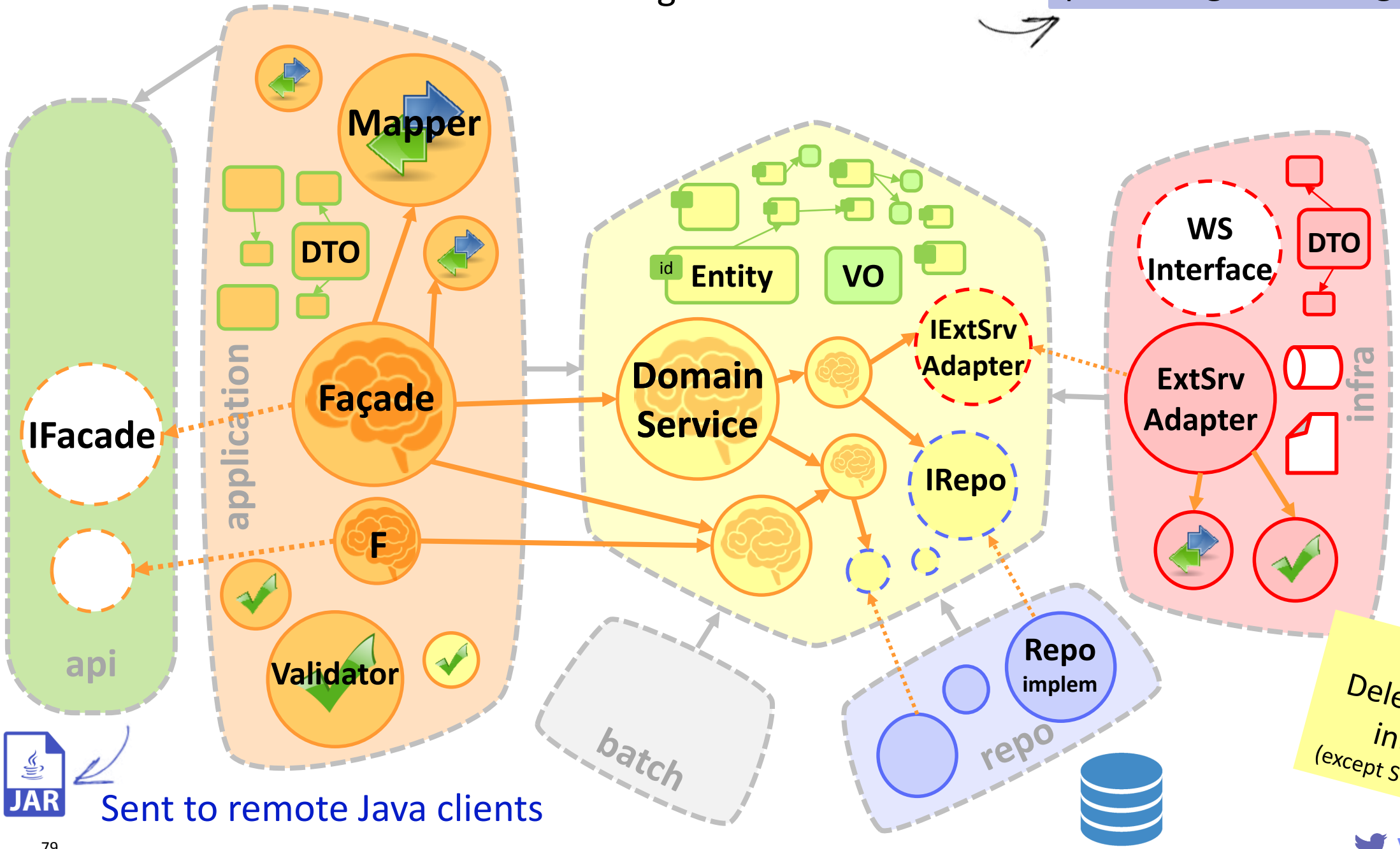
Pragmatic

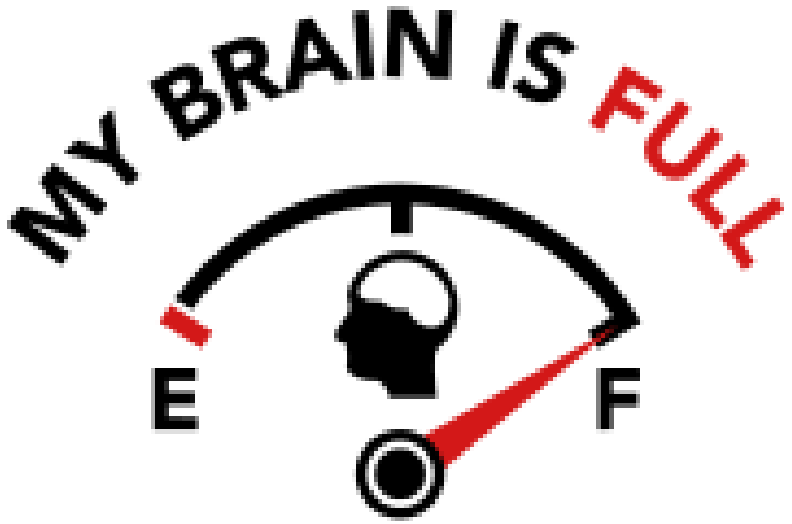
for S/M apps, 2 modules might be enough



That's all I got on architectures

persisting knowledge...



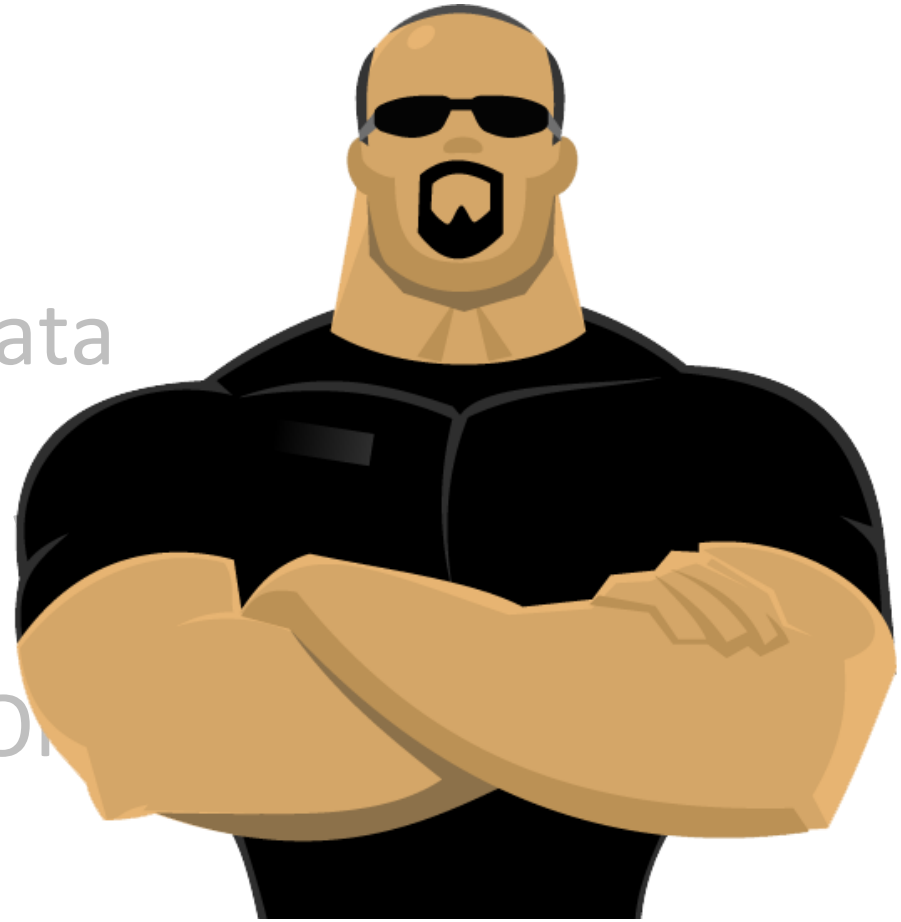

MY BRAIN IS FULL
E
F

Uncle Bob on TDD

https://youtu.be/GvAzrC6-spQ

Agenda

- [x] Driving Principles – KISS
- [x] Modeling Data – Enemy data
- [x] Organizing Logic – Extract
- [x] Clean Architecture – The O
- [ ] **Tests. Fear.**



Lots of Unit Tests are Good !

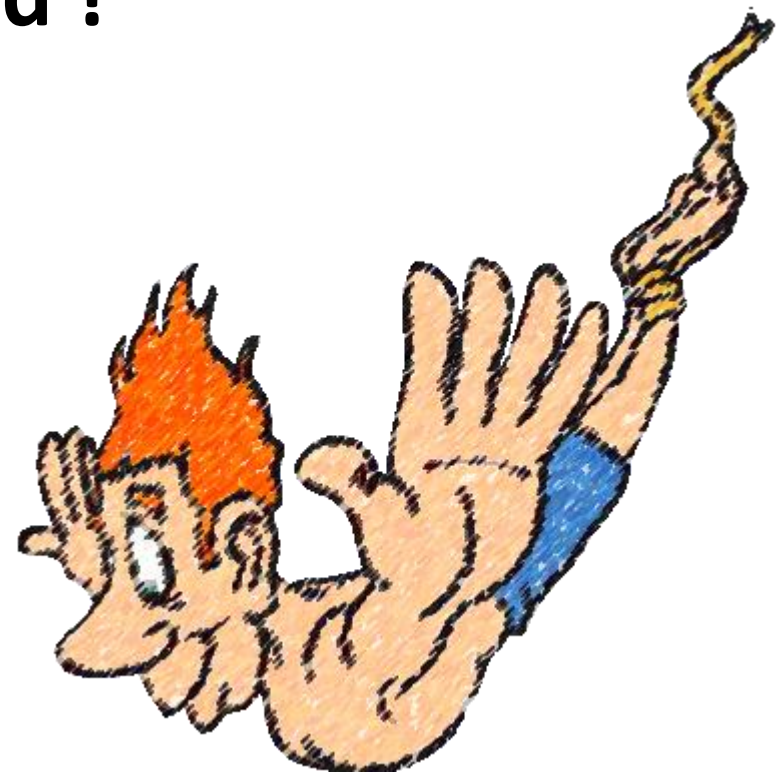
(It's still a single )

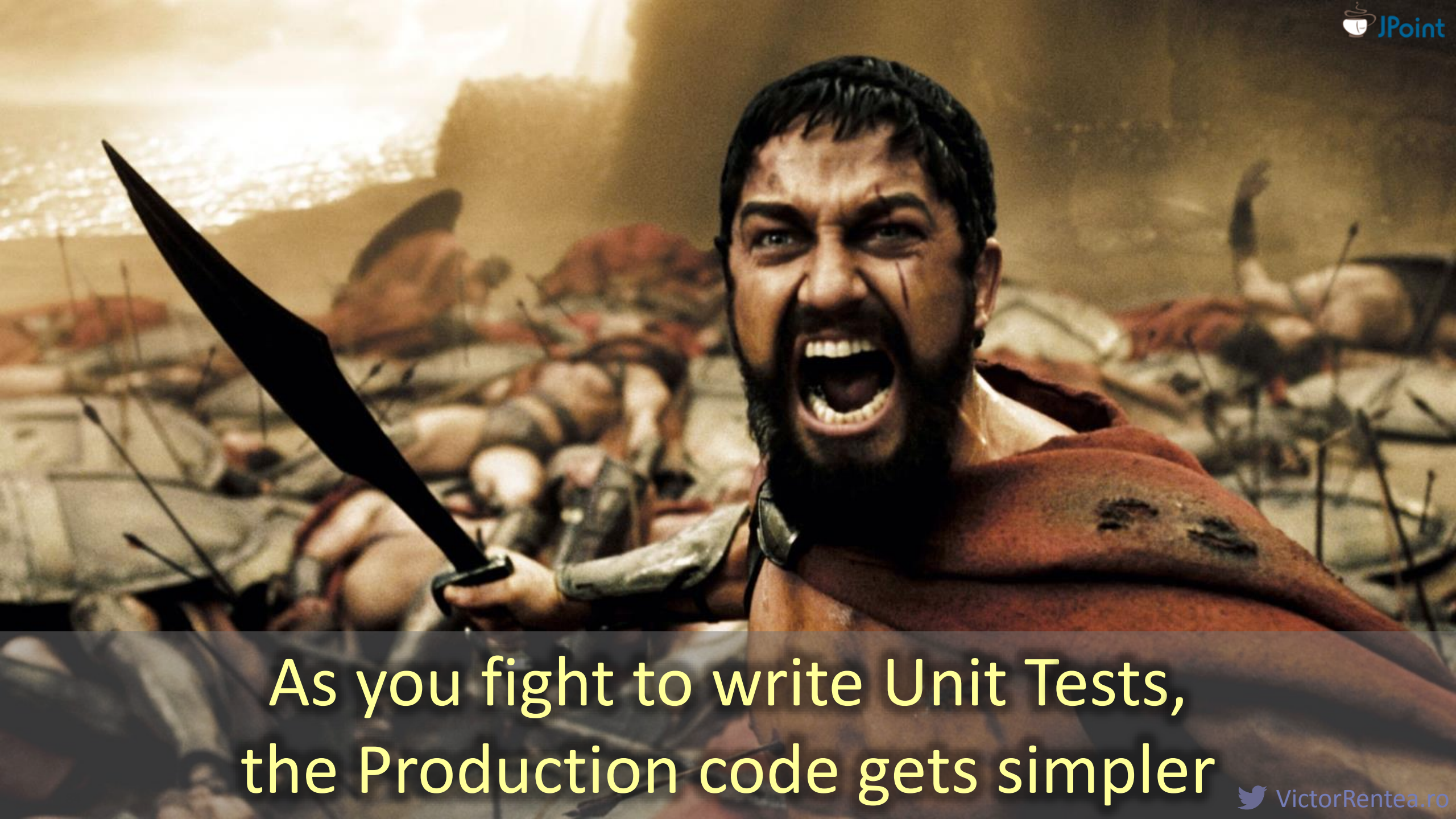
false

A sense of confidence

Developer Courage™

Continuous Refactoring ✓




JPoint
As you fight to write Unit Tests,
the Production code gets simpler
VictorRentea.ro

Testable code == Good code

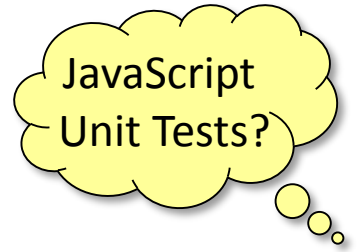
Simpler

Decoupled

Live-documented

➔ **First couple of unit tests – The most valuable**

Fast tests, Fast feedback



A. **Input-output**

Pure functions, no side effects. Ideally, no dependencies !

$$f(x, y) = x^2 + y^2$$

B. Domain logic by **mocking** dependencies

Less readable tests

C. **Queries** on a fresh database for each test (in-memory)

Reuse complex test data -> Fragile tests ?

Short, Readable, Clean tests

```
@Test
public void deactivate_ok() {
    InternalReference rau = new InternalReferenceBuilder().activate(user).build();
    service.deactivate(rau);

    assertEquals(Status.INACTIVE, rau.getStatus());
    verify(alertService).raiseDeactivationAlert(rau, Arrays.asList());
}
```

```
lidationException.class)

NameFails() {

dCustomer().withName(null).build());
```

```
);

() {
Builder().withShortName("xAh").build());

teria).size());

teria).size());

hine1));
```

```
@Test
public void createThrowsIfUsernameInDB() {
    expectedException.expectMessage(ErrorCode.UNIQUE_USERNAME.toString());
    when(repository.countByUsername("uid", -1L)).thenReturn(1);

    UserDto dto = new UserDto ();
    dto.username = "uid";
    facade.create(dto);
}
```

```
@Tes
publ
    List<ObjectToMapView> objectsToMap = facade.getHomeView();

    ObjectToMapView rauView = objectsToMap.stream().filter(o -> o.id.equals(RAU.name())).findFirst().get();
    assertEquals(2, rauView.mappingTypes.size());
    assertTrue(rauView.mappingTypes.stream().anyMatch(mt -> mt.label.contains(rauSun1.getNiceName()) && mt.id == 13L));
    assertTrue(rauView.mappingTypes.stream().anyMatch(mt -> mt.label.contains(rauShine1.getNiceName()) && mt.id == 14L));
}
```

size

Disclaimer

Mostly Backend Guy

Experience with
Banking, Flavors, ERP and Posting
applications

Projects of size
S-L

Open-minded Client
(Freedom to architecture)

Agenda

- [x] Driving Principles – KISS
- [x] Modeling Data – Enemy data
- [x] Organizing Logic – Extract when it Grows
- [x] Clean Architecture – The Onion
- [x] Tests. Fear.

Takeaways

- ☑ Driving Principles – KISS
- ☑ Modeling Data – Enemy data
- ☑ Organizing Logic – Extract when it Grows
- ☑ Clean Architecture – The Onion
- ☑ Tests. Fear.

Takeaways

KISS: Avoid premature Encapsulation *or Optimization*

MAGIC to protect your Developers

- ☑ Modeling Data – Enemy data
- ☑ Organizing Logic – Extract when it Grows
- ☑ Clean Architecture – The Onion
- ☑ Tests. Fear.

Takeaways

or Optimization

KISS: Avoid premature Encapsulation

MAGIC to protect your Developers

ENEMY DATA in your DTOs: keep them out

Organizing Logic – Extract when it Grows

Clean Architecture – The Onion

Tests. Fear.

Takeaways

KISS: Avoid premature Encapsulation *or Optimization*

MAGIC to protect your Developers

ENEMY DATA in your DTOs: keep them out

EXTRACT WHEN IT GROWS: for **SRP** or **DRY**

Clean Architecture – The Onion

Tests. Fear.

Takeaways

KISS: Avoid premature Encapsulation

or Optimization

MAGIC to protect your Developers

ENEMY DATA in your DTOs: keep them out

EXTRACT WHEN IT GROWS: for **SRP** or **DRY**

THE ONION, DIP: domain agnostic to externals

(Adapt® them)

Tests. Fear.

Takeaways

KISS: Avoid premature Encapsulation *or Optimization*

MAGIC to protect your Developers

ENEMY DATA in your DTOs: keep them out

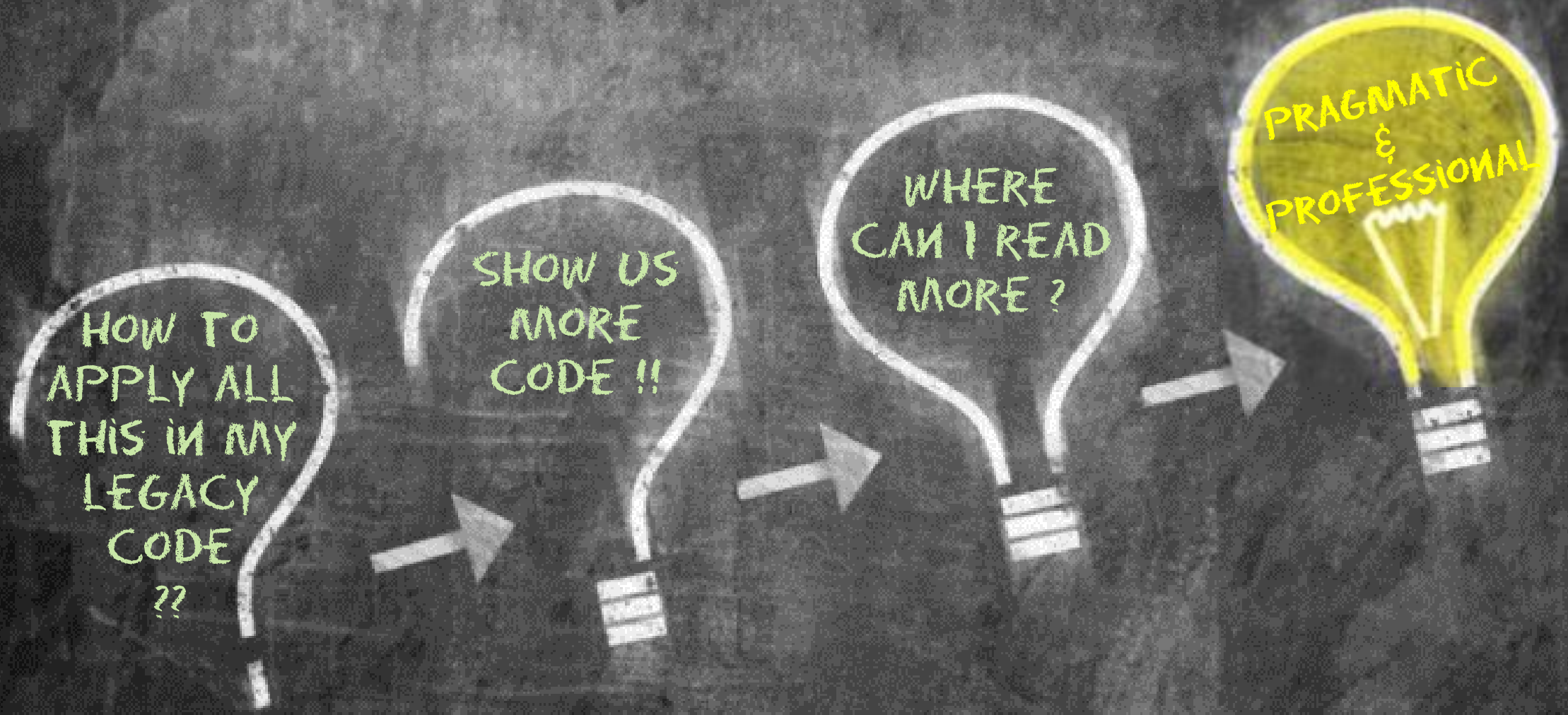
EXTRACT WHEN IT GROWS: for **SRP** or **DRY**

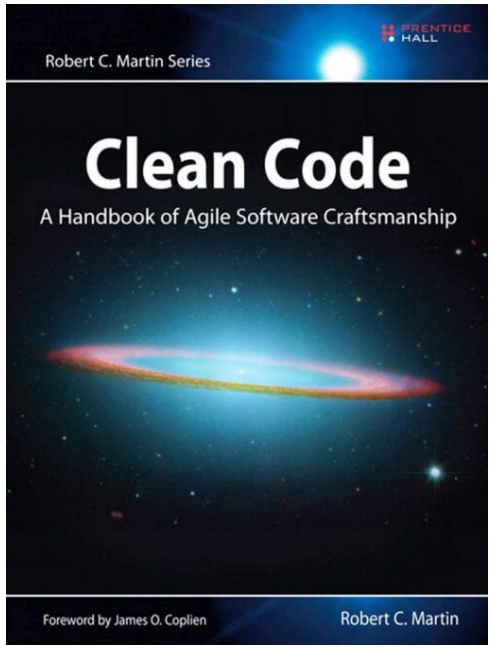
THE ONION, DIP: domain agnostic to externals *(Adapt® them)*

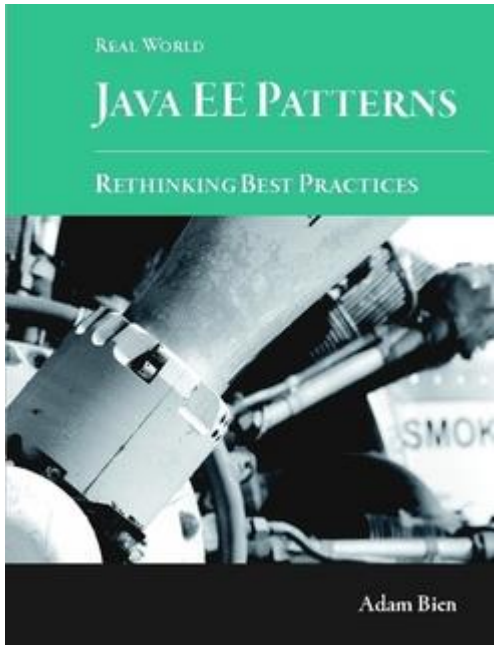
TESTS: let them smash your design

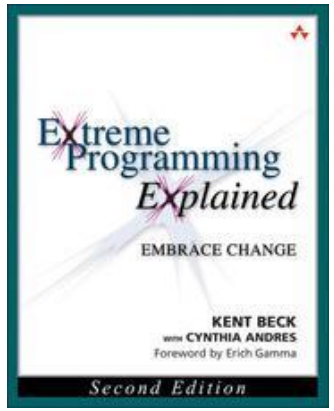

KISS

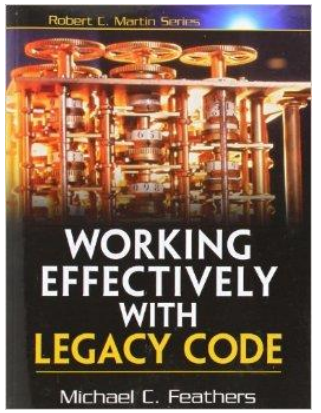
KEEP IT SIMPLE

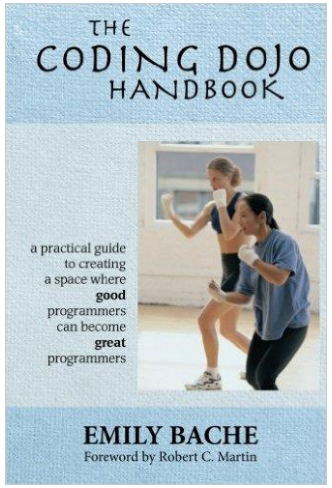


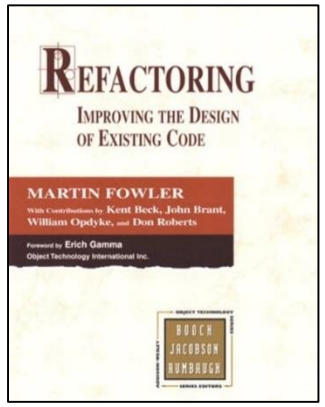












- *7 Virtues of a Good Object*
- NULL – the worst mistake in IT
  - *https://dzone.com/articles/the-worst-mistake-of-computer-science-1*
- The Clean Architecture:
  - http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Some ☺ Programming Jargon
  - http://blog.codinghorror.com/new-programming-jargon/
- Code quality: WTFs/minute
  - http://commadot.com/wtf-per-minute/
- SOLID is WRONG
  - https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong
- Good software is written 3 times
  - http://www.javaworld.com/article/2072651/becoming-a-great-programmer--use-your-trash-can.html
- Prezi-like effect in PowerPoint 2016: “Morph”

HUGE Thanks
to the JPoint Reviewers !!

Vladimir Sitnikov

Anton Arhipov

Enterprise Java Training

A TRAINING BY VICTOR RENTEA

Brai ning

C Pra ctitectur

Thank you!

Victor Rentea

22 feb 2017

JPoint

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea