arnaud.nauwynck@gmail.com

Architecture Design

Part 2 : Model & Service

Entity - Model vs Service

SOLID principles for business logic code

This document:
http://github.com/arnaud-nauwynck/Presentations/java/
Architecture-Design-part2-Model.pdf

Outline

- Entities problems : FLAT , Anemmic classes
- Model to the rescue ?
- Organize code with SOLID principle
- Services

Reminder Part 1 : Entity classes

Entity Restrictions

1/ Entities FIT the database Tables design

2/ Restrictions from JPA / Relational model
2bis/ NO complex (table join) classes hierarchy

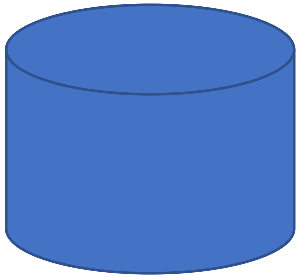
3/ Lifecycle managed by Transaction

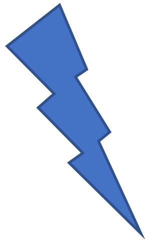
4/ NOT suited to put business logic code (SOLID principle)
4bis/ Only POJO getters/setters

… DTO + Service (+ Model?) to the rescue

Focus on (Persistence) Data vs Behavior

An Entity is unique via its « ID »
And completely defined via all its state data





Not focusing (yet) on behavior methods on Entity
Assume only Getter/Setters
+ Add/Remove relations to other Entities

Behavior will be handle by Service classes
or corresponding Model class

POJO Entity class : weakness for Smart Code

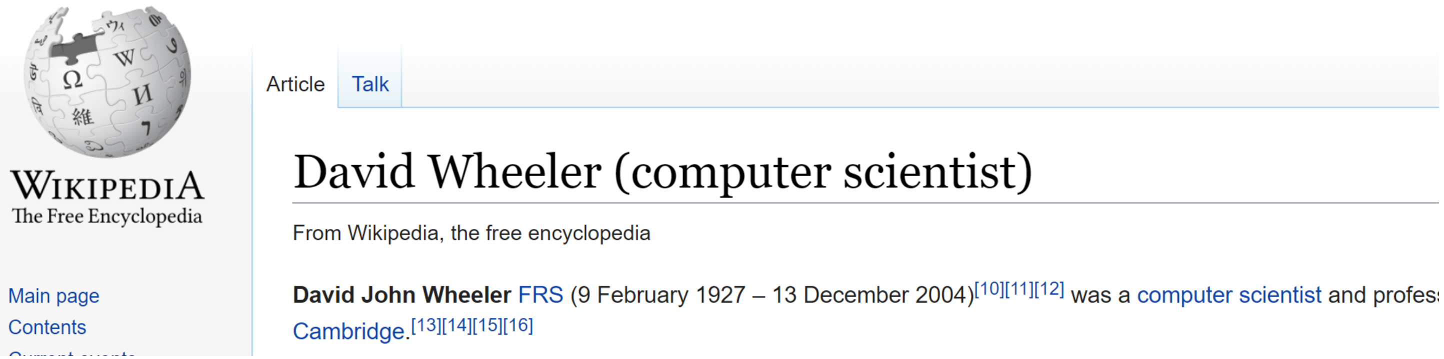
« Anemic » classes

```
@Entity
@Getter @Setter // POJO with lombok
Class XEntity {

@Id @GeneratedValue
private long id;

private int field1, field2, field3, field4;
private String field5, field6, field7;
private Date field8, field9, field10;

}
```

🙂 Class looks full of fields/data
May save lot of « things »

☹ Only getter/setters – No methods
No business logic / behavior

« The » Computer Science Theorem



David Wheeler (computer scientist)

From Wikipedia, the free encyclopedia

David John Wheeler FRS (9 February 1927 – 13 December 2004)[10][11][12] was a computer scientist and profes

Cambridge.[13][14][15][16]

David Wheeler FRS

Born	David John Wheeler 9 February 1927[1]

“ All problems in computer science
can be solved
by another level of indirection,
except for the problem of too many layers of indirection.”

Model always needed / usefull for Entity ?

If you ask question
then you know the answer

Model … indirection to Entity

Model is an indirection to Entity
& (Entity) “problems can be solved **by another level of indirection”**

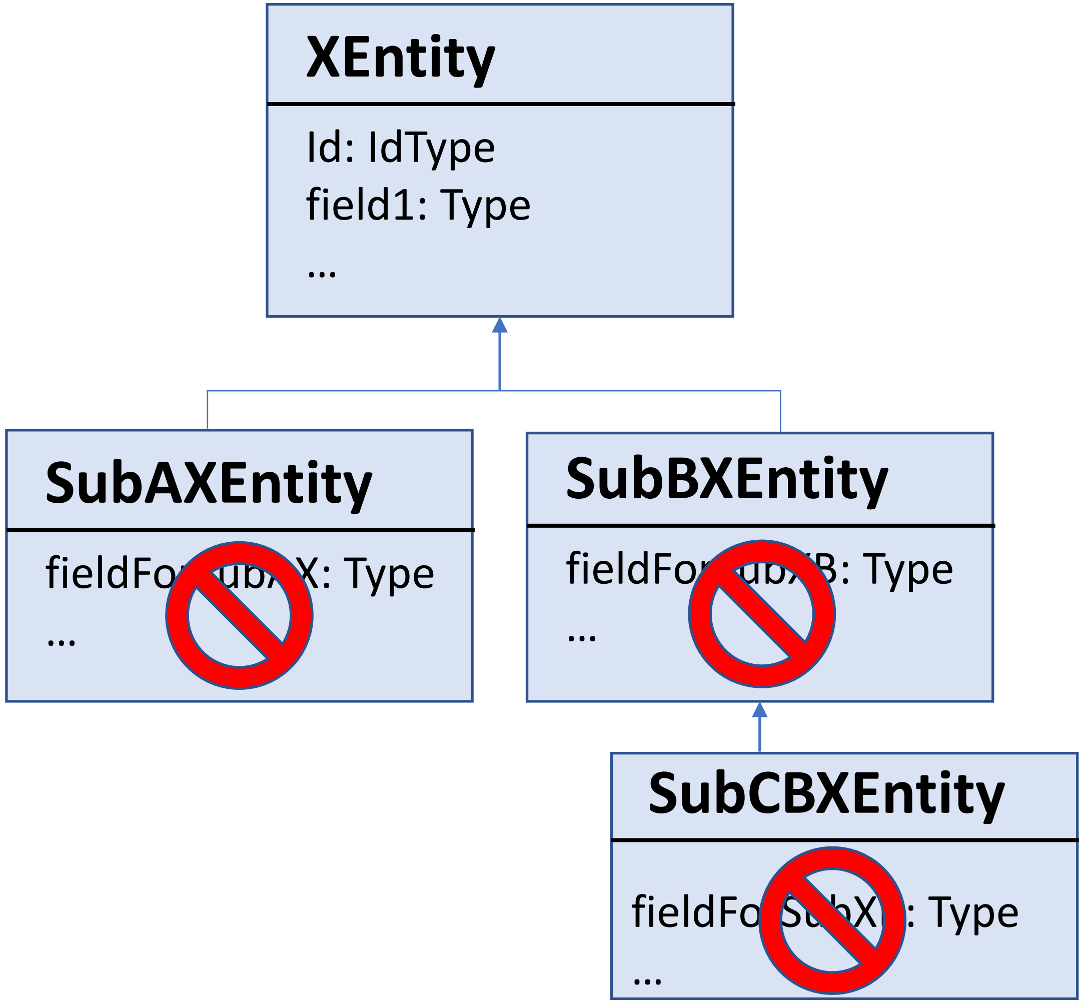
=> Possibility that Model indirection solve some Entity problem

For simple « CRUD » … There is absolutly NO problem

… except that « Model » defines too many additional level of indirection

Adding Model indirection level falls in the rule of “**too many layers of indirection**”

Flat / Object-Oriented classes hierarchy

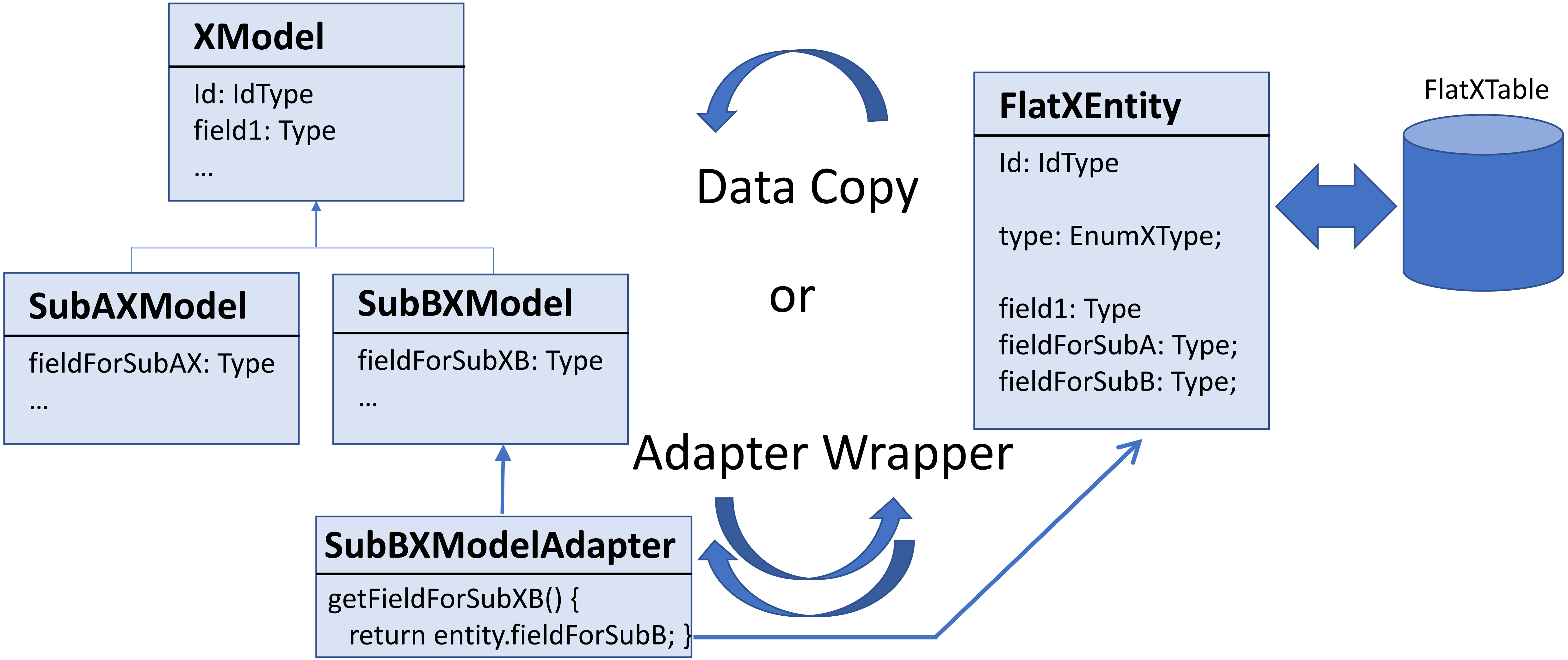


Start simple …

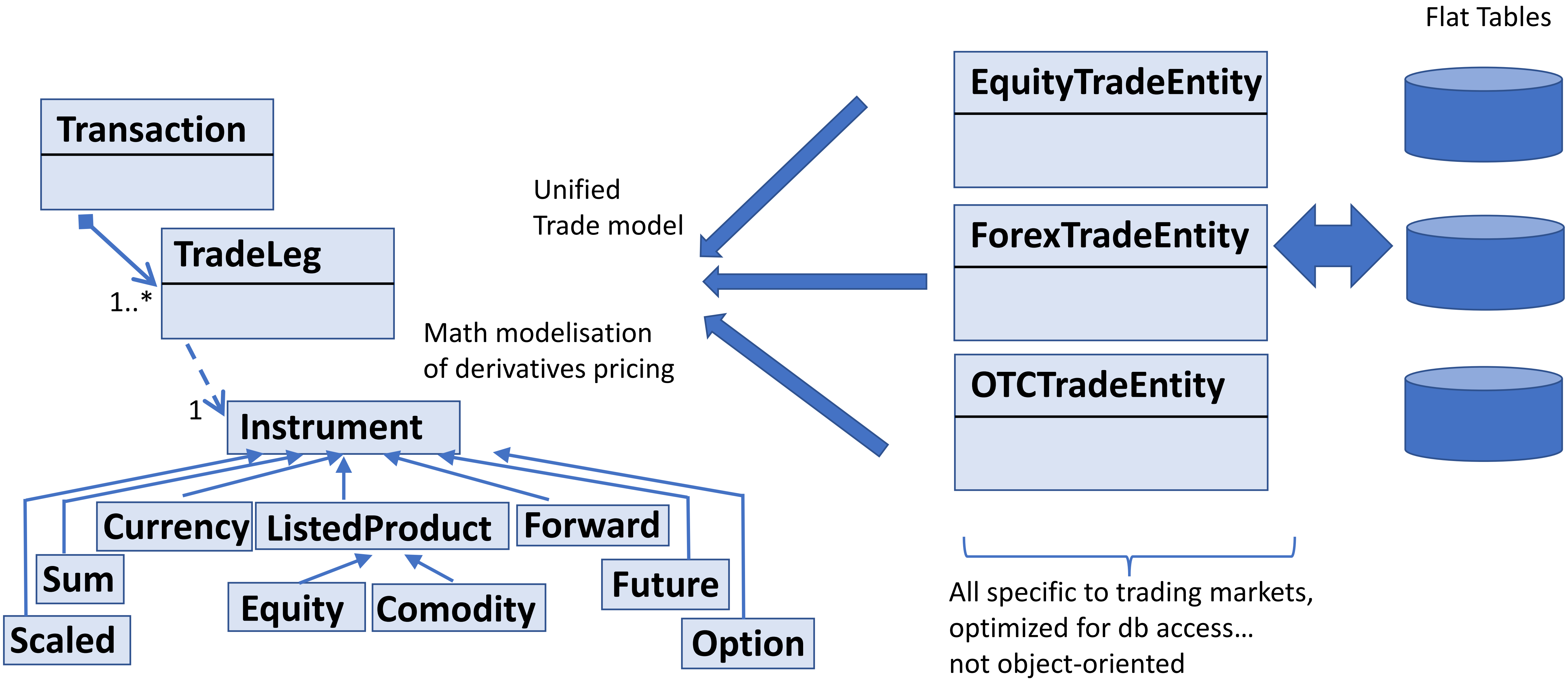
NO interface
NO « diamond » inheritance
NO sub-class without new fields
NOT focusing on « behaviour »
… focusing only on « data »

Favor delegation: « has »
rather than inheritance: « is »

Object-Oriented for Code <-> Flat for DB



Typical mismatch between Finantial Instrument Models <-> Optimized Database

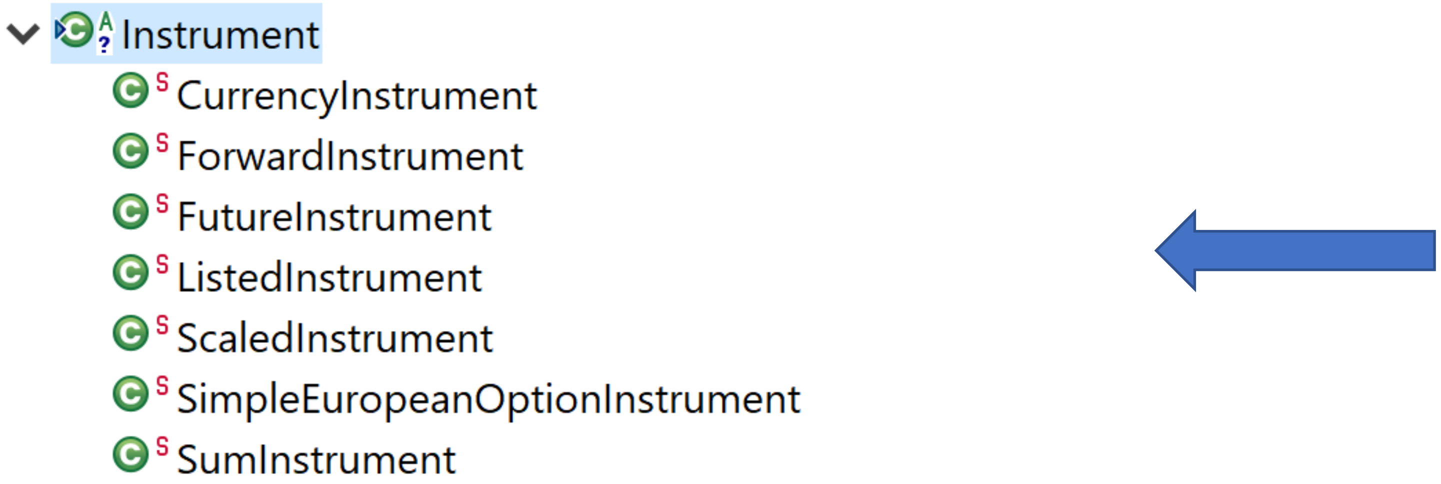


Unified
Trade model

Math modelisation
of derivatives pricing

All specific to trading markets,
optimized for db access…
not object-oriented

Typical Flat to Tree Converter (Pricing Finance)

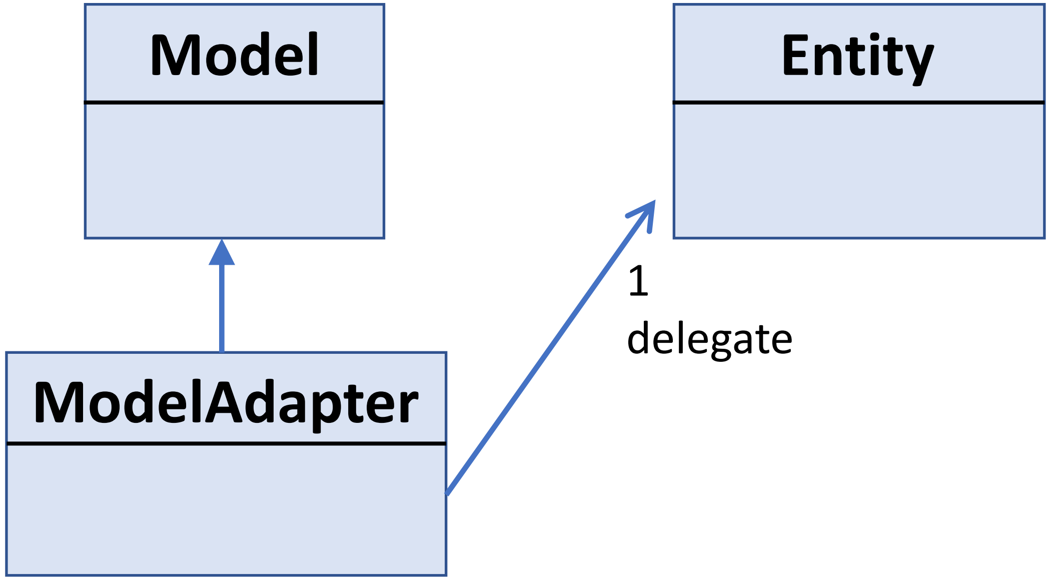


- Instrument
  - CurrencyInstrument
  - ForwardInstrument
  - FutureInstrument
  - ListedInstrument
  - ScaledInstrument
  - SimpleEuropeanOptionInstrument
  - SumInstrument

```
@Entity
@Getter @Setter
public class FlatTradeEntity {

    @Id
    @GeneratedValue
    private long id;

    String type;

    String currency;
    String underlying;
    double quantity;
    double strike;
    private Date expiryDate;
```

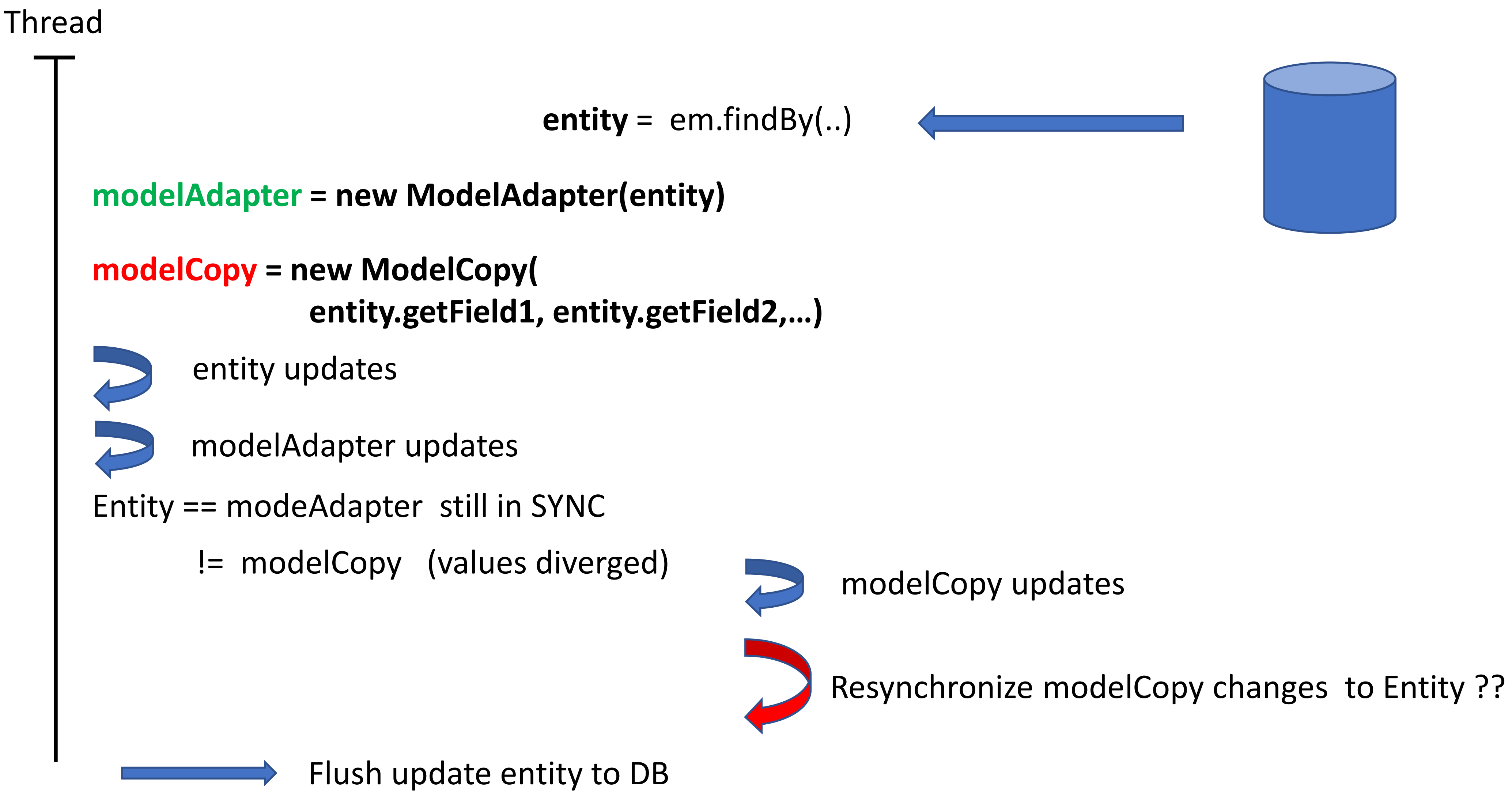
```
public Instrument toInstrument(FlatTradeEntity src) {
    String type = src.getType();
    if ("FxCallEuro".equals(type)) {
        val currency = CurrencyInstrument.of(src.getCurrency()
        return SimpleEuropeanOptionInstrument.builder()
                .currency(currency)
                .payoffType("CallEuro")
                .strike(src.getStrike())
                .expiryDate(src.getExpiryDate())
                .underlying(new ScaledInstrument(src.getQuantity(),
                        CurrencyInstrument.of(src.getUnderlying()))
                        )
                .build();
    } else {
```

Model Adapter design-pattern



```
interface Model {
    int getField();
    int setField(int field);
}

class ModelAdapter implements Model {

   private Entity entityDelegate;

   @Override
   public int getField() {
       return entityDelegate.getField();
   }

   @Override
   public void setField(int p) {
       entityDelegate.setField(p);
   }
}
```

Copy / Adapter
out-of-sync / updates to Entity ?



Thread

entity = em.findBy(..)

modelAdapter = new ModelAdapter(entity)

modelCopy = new ModelCopy(
entity.getField1, entity.getField2,…)

entity updates

modelAdapter updates

Entity == modeAdapter still in SYNC

!= modelCopy (values diverged)

modelCopy updates

Resynchronize modelCopy changes to Entity ??

Flush update entity to DB

Read-Only Model : simpler

```
interface ReadOnlyModel {
    int getField();
    // NO setter
}

class ReadOnlyModelAdapter implements Model {

    private Entity entityDelegate;

    @Override
    public int getField() {
        return entityDelegate.getField();
    }

}
```

Model using Immutable Copy + Converter (Builder pattern)

```
interface Model {
    int getField();
    // NO setter
}
@Builder // lombok
@Getter
class ImmutableModel implements Model {
    public final int field1;
    public final int field2;
    public final int field3;
}
```

```
// usage
public ImmutableModel copyEntityToModel(Entity src) {
    return ImmutableModel.builder()
        .field1(src.getField1())
        .field2(src.getField2())
        .field3(src.getField3())
        .build();
}
```

```
// Builder class generated from Lombok
public static class Builder {
    public int field1;
    public Builder field1(int p) {
        this.field = p;
        return this;
    }
    … field2, field3, …
    ImmutableModel build() {
        return new ImmutableModel(field1, field2, field3);
    }
}
```

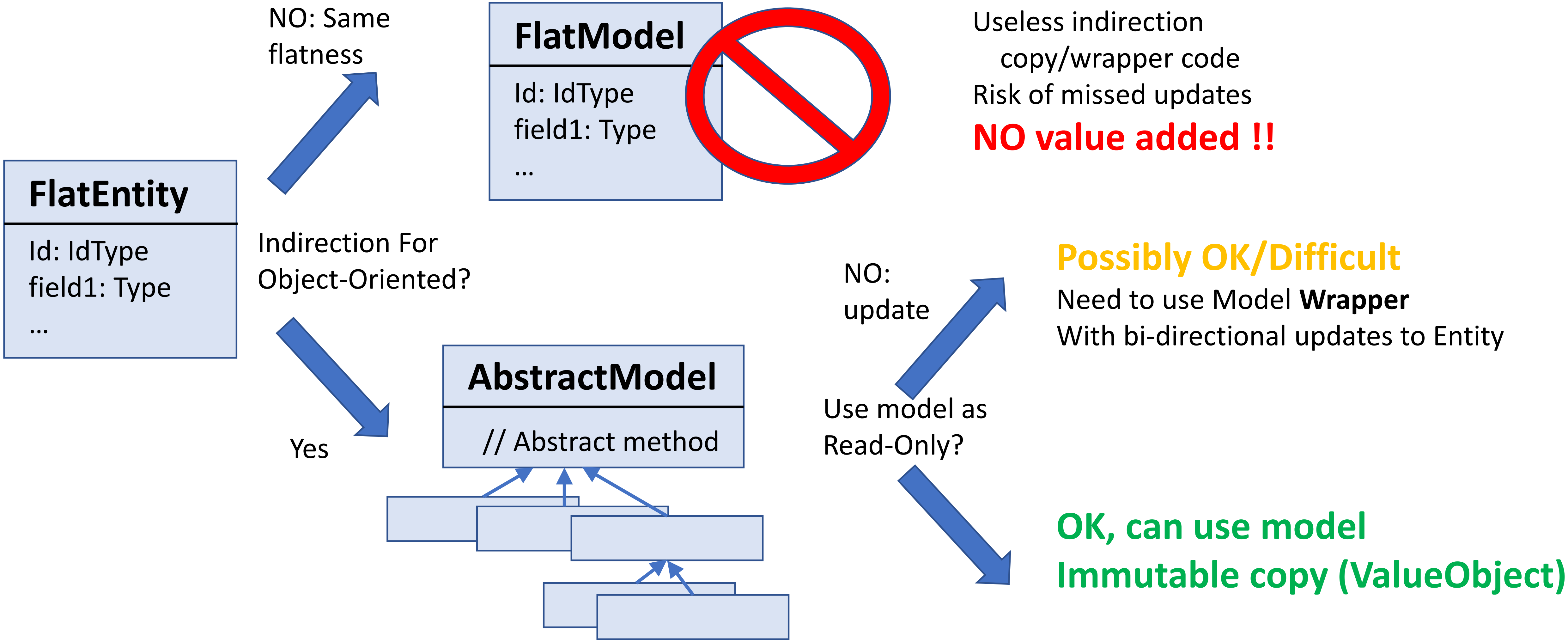
Unmodifiable (wrapper) vs Immutable (copy)

see java.util.UnmodifiableList / guava ImmutableList

- Underlying object can be changed outside
- Unsafe code can get content, then update
- UnmodifiableList extends List …. Type are compatible
- Confusing: All methods set/add/clear are declared
  … but throw RuntimeException
- Same method for List & UnmodifiableList

- Safe code. Final fields copy
- Optimized layout for list 1,2,3…
- Type are NOT compatible
- Only read-only methods declared
- different methods for Immutable
  (incompatible type signatures)

Decision Diagram to use Model



FlatEntity

Id: IdType
field1: Type
…

Indirection For Object-Oriented?

NO: Same flatness

FlatModel

Id: IdType
field1: Type
…

Useless indirection
copy/wrapper code
Risk of missed updates
NO value added !!

Yes

AbstractModel

// Abstract method

Use model as Read-Only?

NO: update

Possibly OK/Difficult

Need to use Model **Wrapper**
With bi-directional updates to Entity

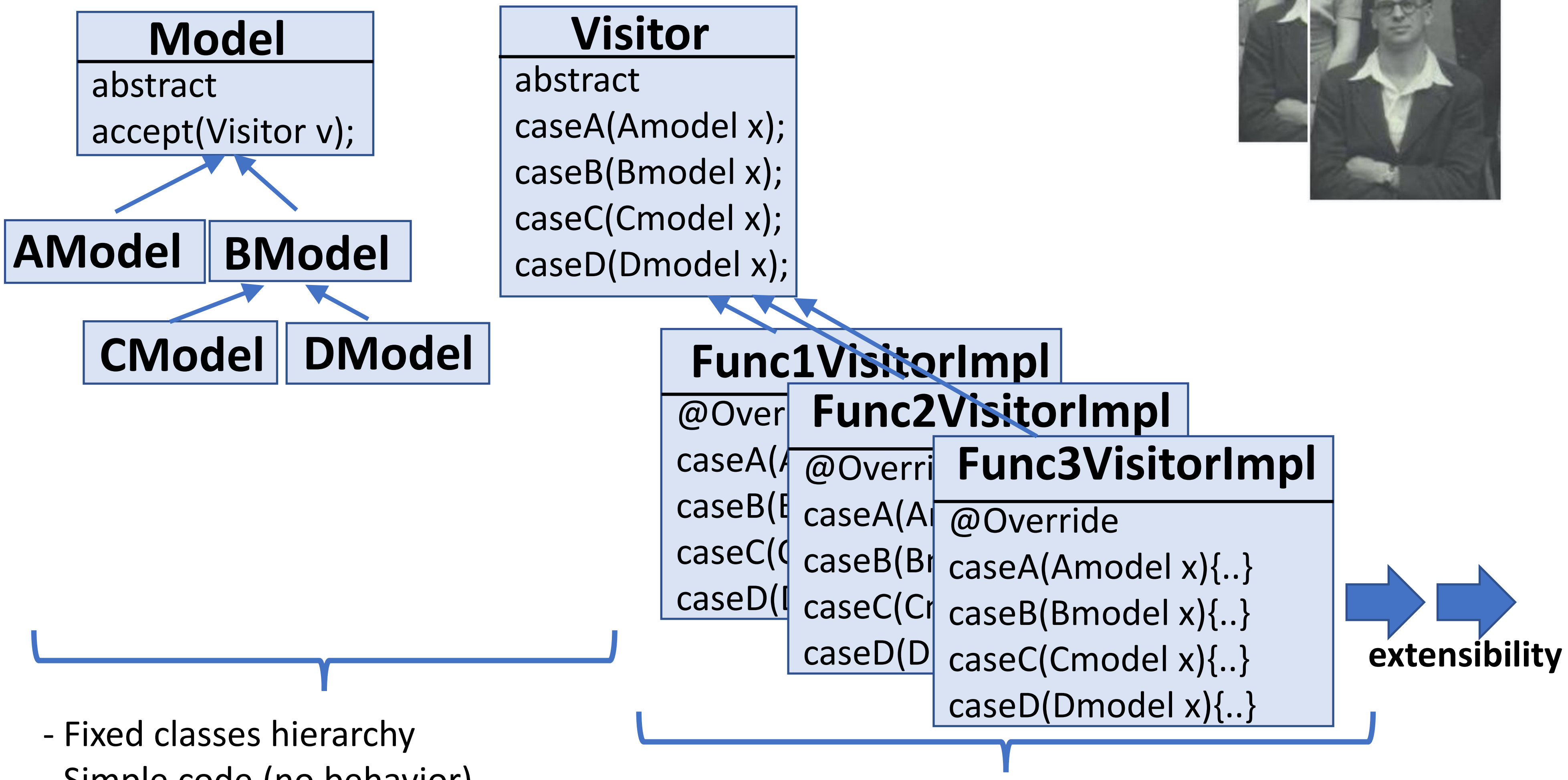
OK, can use model
Immutable copy (ValueObject)

Annexe

Pb => 1 More Indirection to Model
« Visitor » Design-Pattern

« Magic » dispatches

```
class X extends Model {
  @Override
  accept(Visitor v) {
    v.caseX(this);
  }
}
```



Model
abstract
accept(Visitor v);

AModel **BModel**

CModel **DModel**

Visitor
abstract
caseA(Amodel x);
caseB(Bmodel x);
caseC(Cmodel x);
caseD(Dmodel x);

Func1VisitorImpl
@Over
caseA(
caseB(
caseC(
caseD(

Func2VisitorImpl
@Overri
caseA(A
caseB(B
caseC(C
caseD(D

Func3VisitorImpl
@Override
caseA(Amodel x){..}
caseB(Bmodel x){..}
caseC(Cmodel x){..}
caseD(Dmodel x){..}

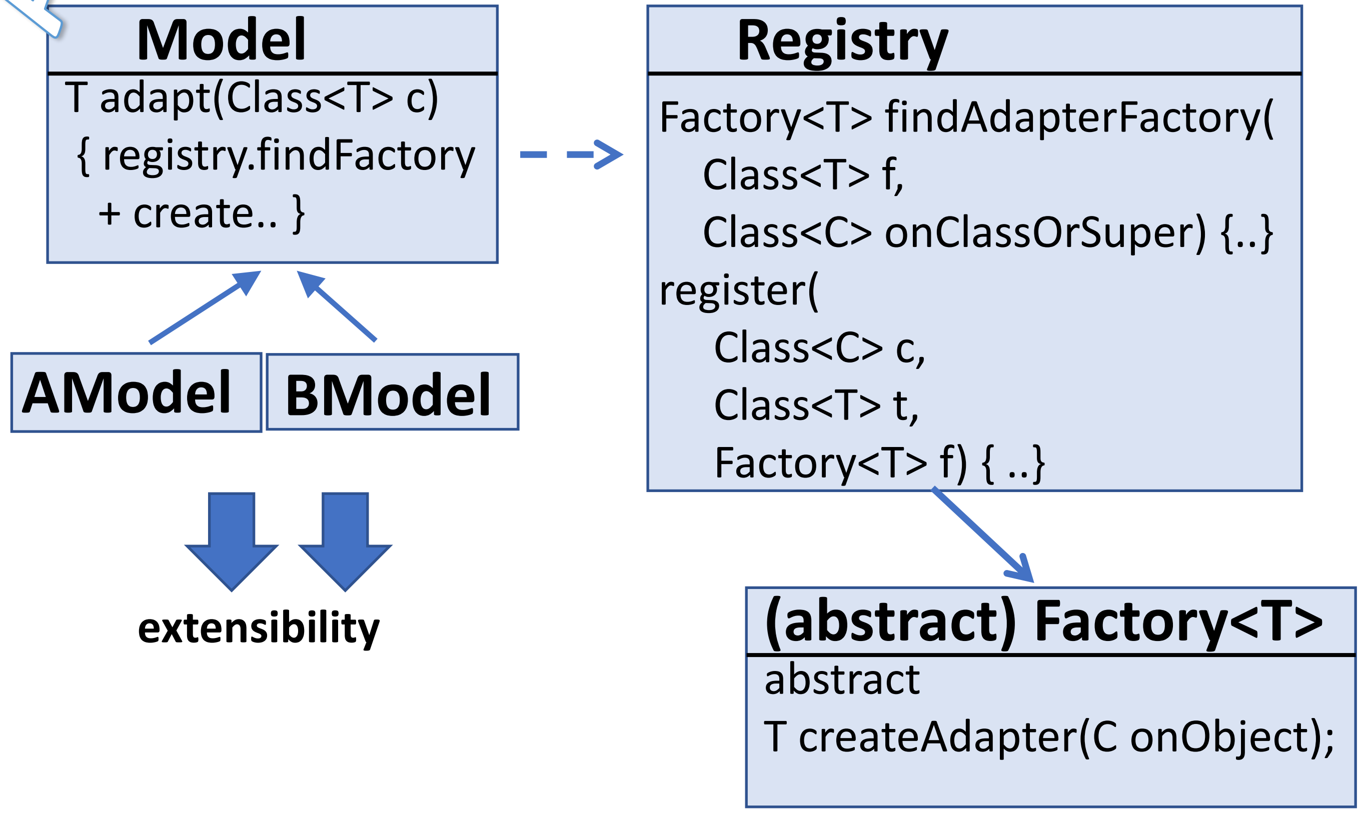
extensibility

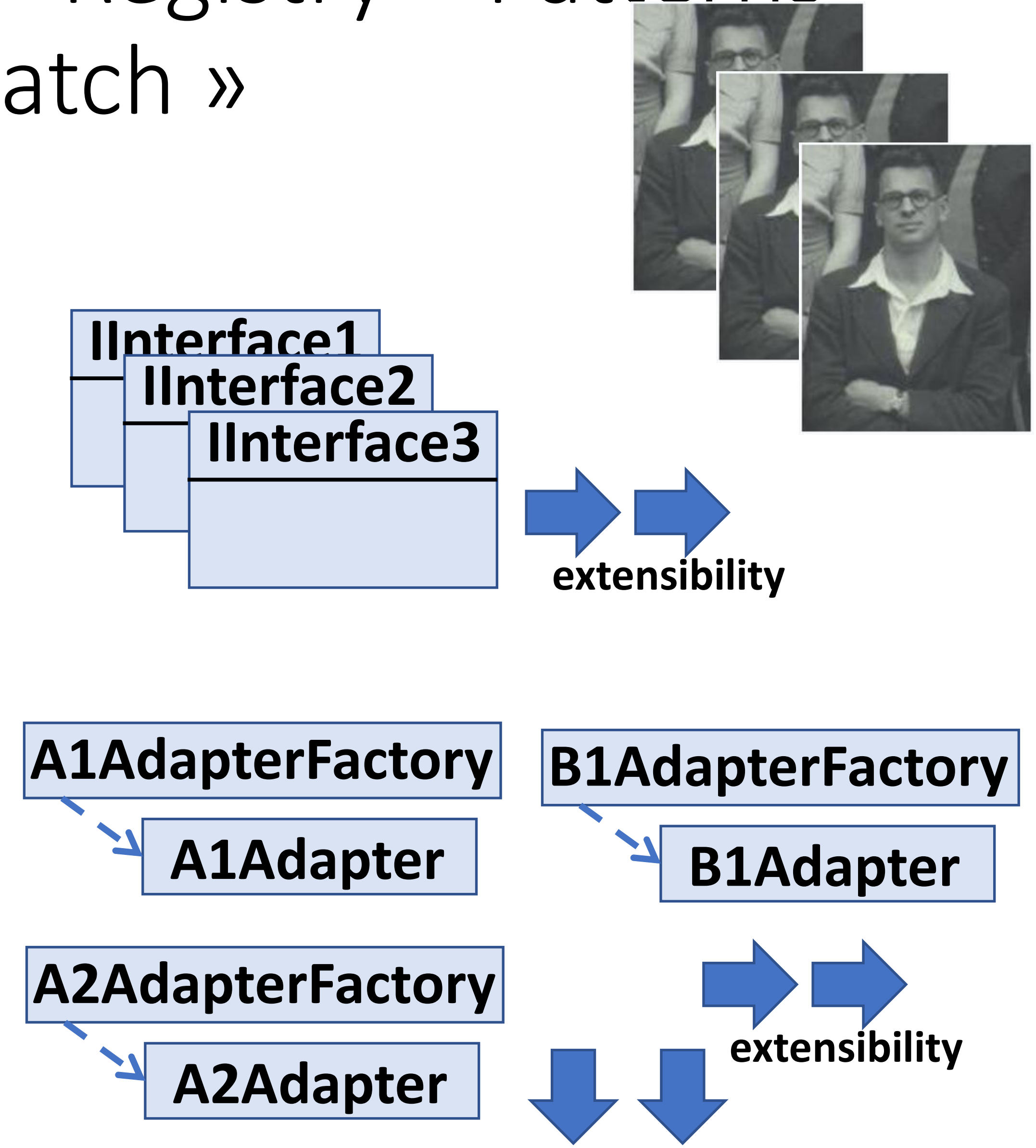
- Fixed classes hierarchy
- Simple code (no behavior)
- Extensible behaviors via visitor

- Independant behavior classes
- SOLID principle

Annexe

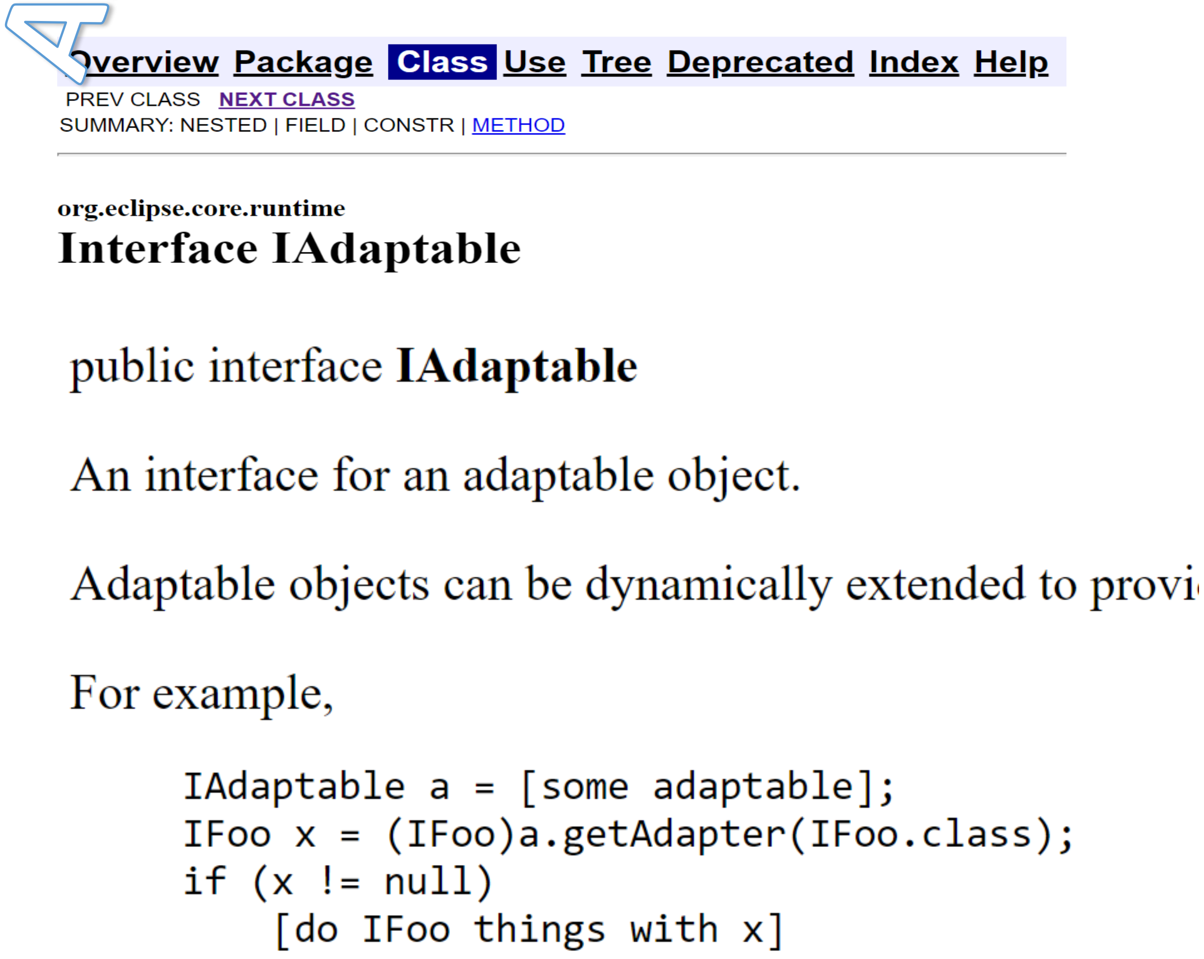
Still Pb => 1 + 1 More Indirections to Model
« Adapter » + « Factory » + « Registry » Patterns
« Double dispatch »





Annexe

See org/eclipse/core/runtime/**IAdaptable.java**
Eclipse Platform « core » Framework
+ Plugins Extensions



Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

org.eclipse.core.runtime

Interface IAdaptable

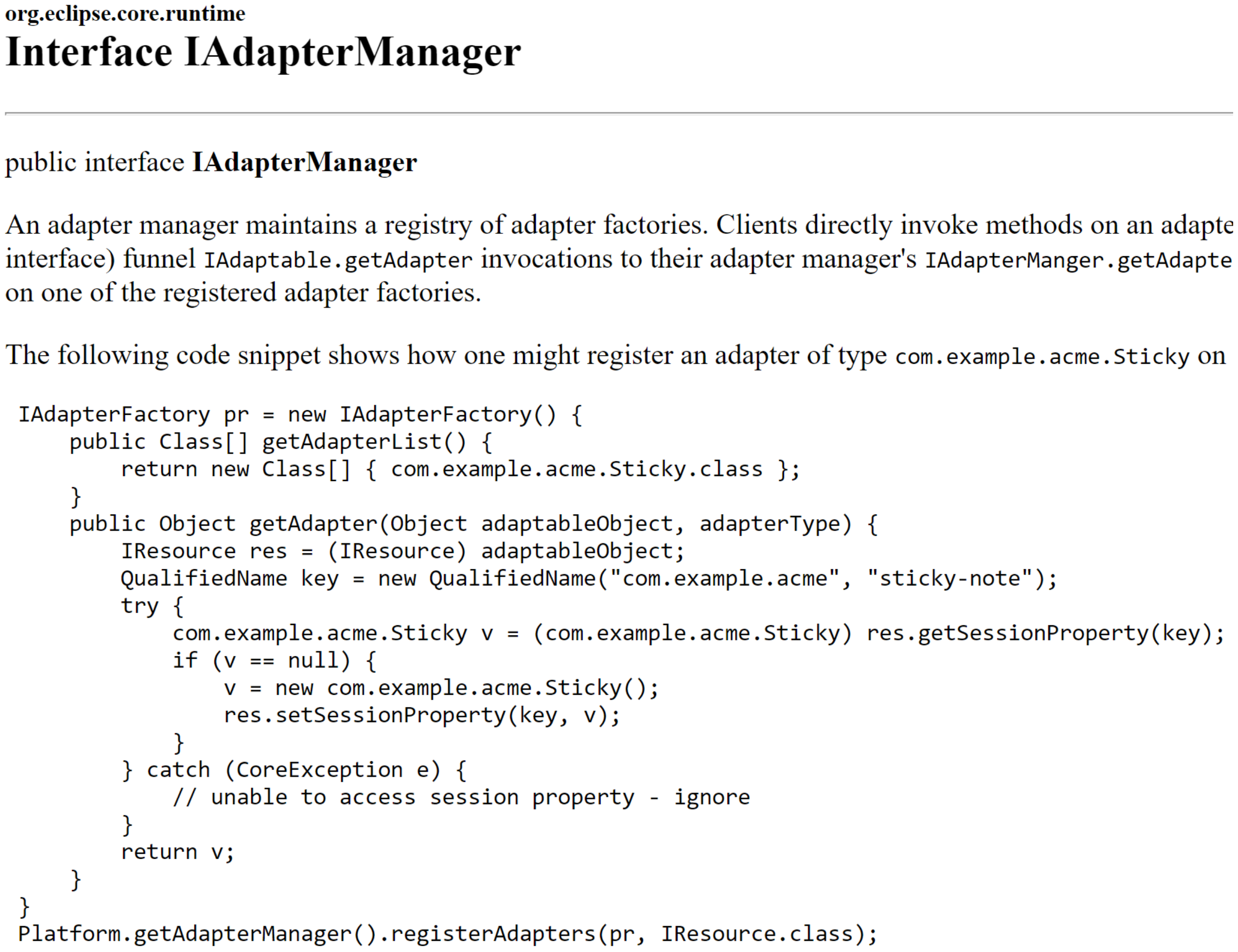
public interface **IAdaptable**

An interface for an adaptable object.

Adaptable objects can be dynamically extended to provi

For example,

```
    IAdaptable a = [some adaptable];
    IFoo x = (IFoo)a.getAdapter(IFoo.class);
    if (x != null)
        [do IFoo things with x]
```



org.eclipse.core.runtime

Interface IAdapterManager

public interface **IAdapterManager**

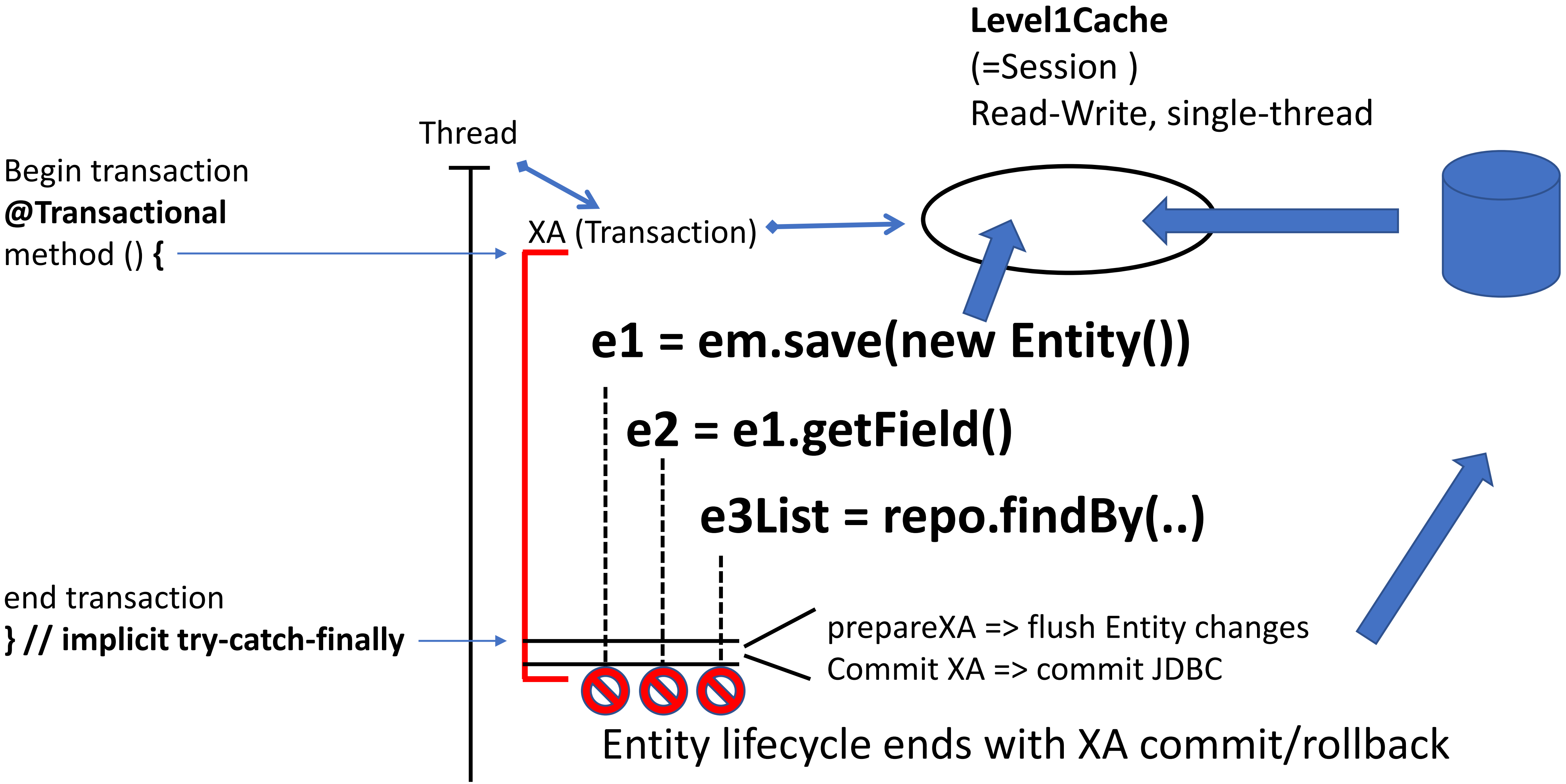
An adapter manager maintains a registry of adapter factories. Clients directly invoke methods on an adapte interface) funnel `IAdaptable.getAdapter` invocations to their adapter manager's `IAdapterManger.getAdapte` on one of the registered adapter factories.

The following code snippet shows how one might register an adapter of type `com.example.acme.Sticky` on

```
 IAdapterFactory pr = new IAdapterFactory() {
     public Class[] getAdapterList() {
         return new Class[] { com.example.acme.Sticky.class };
     }
     public Object getAdapter(Object adaptableObject, adapterType) {
         IResource res = (IResource) adaptableObject;
         QualifiedName key = new QualifiedName("com.example.acme", "sticky-note");
         try {
             com.example.acme.Sticky v = (com.example.acme.Sticky) res.getSessionProperty(key);
             if (v == null) {
                 v = new com.example.acme.Sticky();
                 res.setSessionProperty(key, v);
             }
         } catch (CoreException e) {
             // unable to access session property - ignore
         }
         return v;
     }
 }
 Platform.getAdapterManager().registerAdapters(pr, IResource.class);
```

Restriction on Model Adapter class
… same lifecycle as Entity anyway

Entity : Lifecycle managed by Session (Transaction)



Level1Cache
(=Session)
Read-Write, single-thread

Thread

Begin transaction
@Transactional
method () **{**

XA (Transaction)

e1 = em.save(new Entity())

e2 = e1.getField()

e3List = repo.findBy(..)

end transaction
} // implicit try-catch-finally

prepareXA => flush Entity changes
Commit XA => commit JDBC

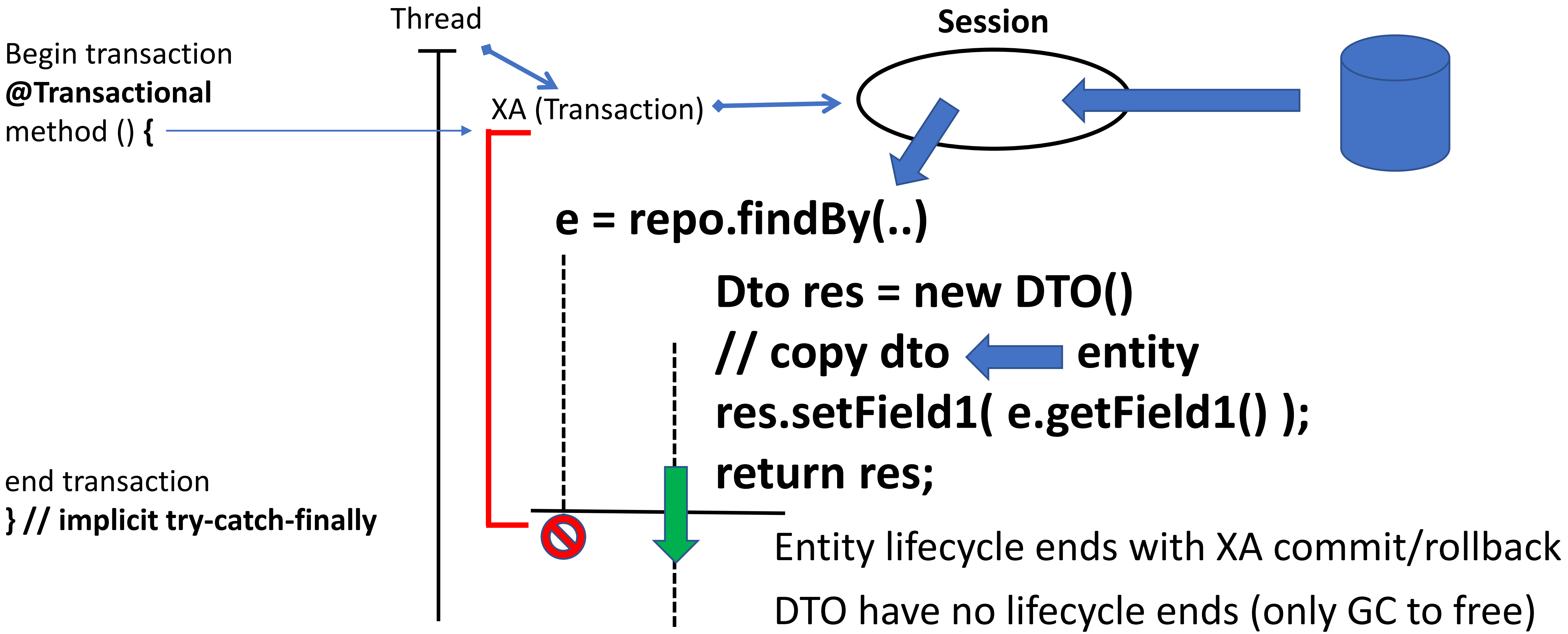
Entity lifecycle ends with XA commit/rollback

Use Entity outside of Transaction ?
RuntimeException

```
// not transactional
// no open-session-in-view
class Controller {

@Autowired FooXAService service;

  public void foo() {
    FooEntity e = service.foo();
    e.getField();  // => Exception !!!
  }
}
```

```
@Transactional
class FooXAService {

  public void foo() {
    FooEntity e = findBy …();
    return e;
  }  // <= INVALID e outside XA
}
```

Copy Entity data to Transfer before Commit



Thread

Session

Begin transaction
@Transactional
method () **{**

XA (Transaction)

e = repo.findBy(..)

Dto res = new DTO()
// copy dto ⬅ entity
res.setField1(e.getField1());
return res;

end transaction
} // implicit try-catch-finally

Entity lifecycle ends with XA commit/rollback

DTO have no lifecycle ends (only GC to free)

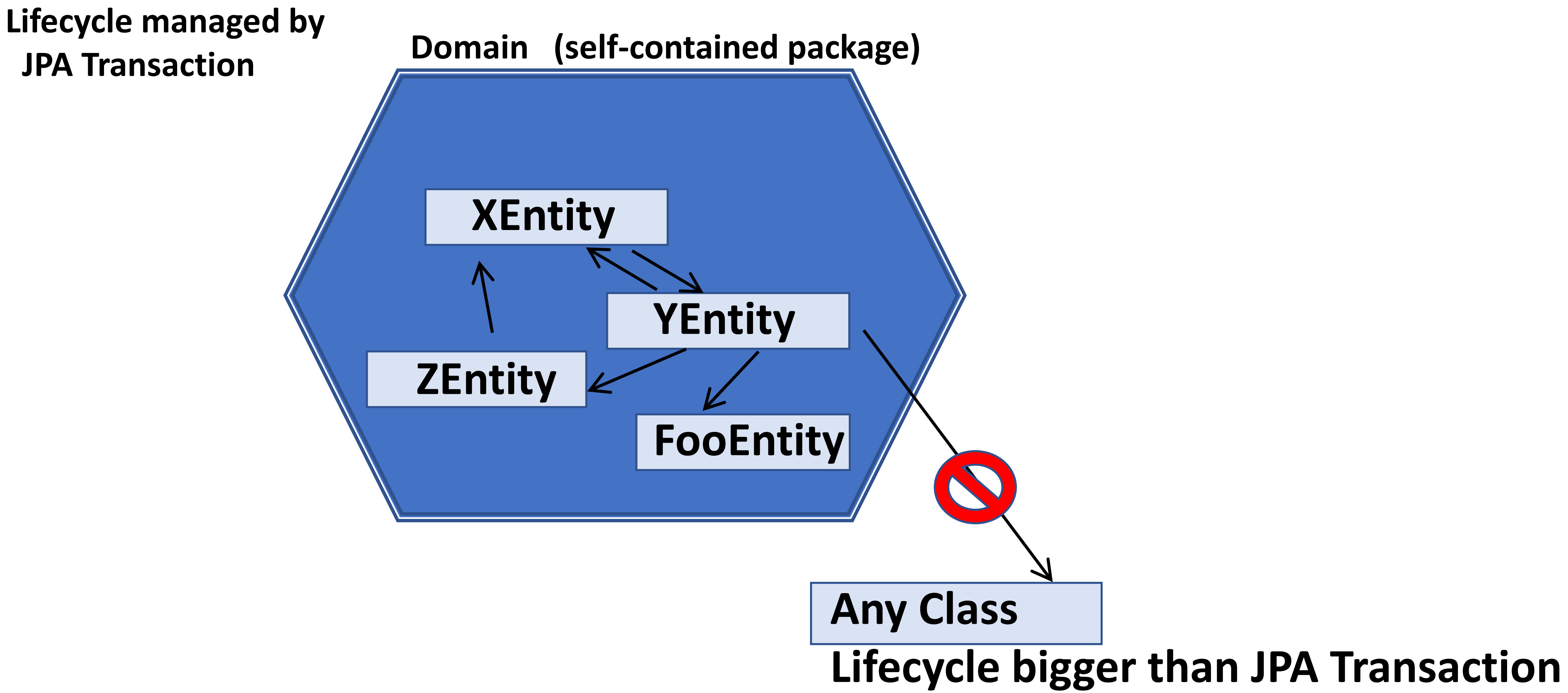
Do not confuse Model with DTO
.. or DTO with Model

Model = better Object-Oriented modelisation

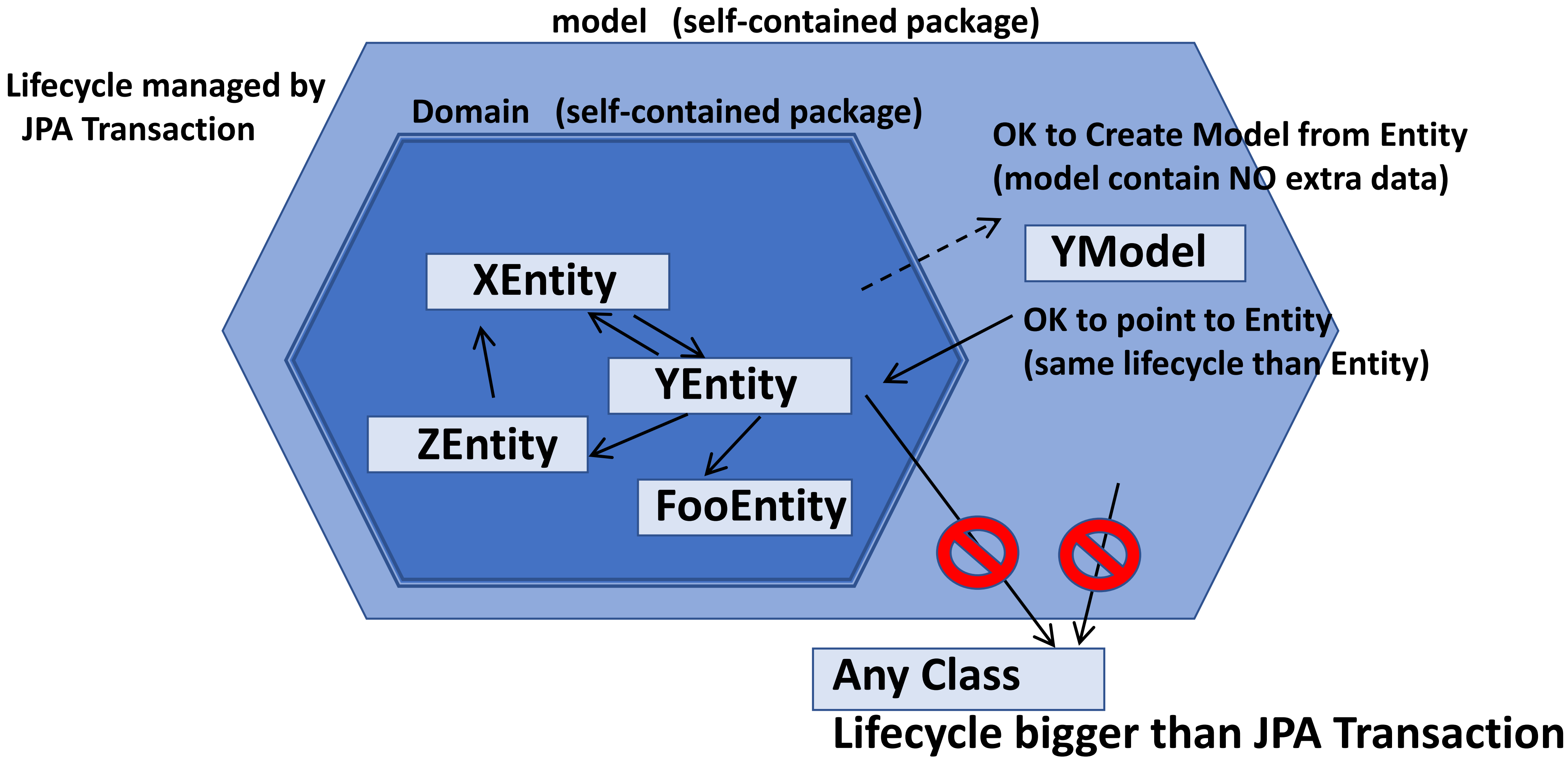
DTO = Data Transfer Object

= an outgoing serializable POJO class to output DATA

Relations (Graph) between Entity restrict to kernel « domain »



Entity – Model ... same managed Lyfecycle



model (self-contained package)

Lifecycle managed by
JPA Transaction

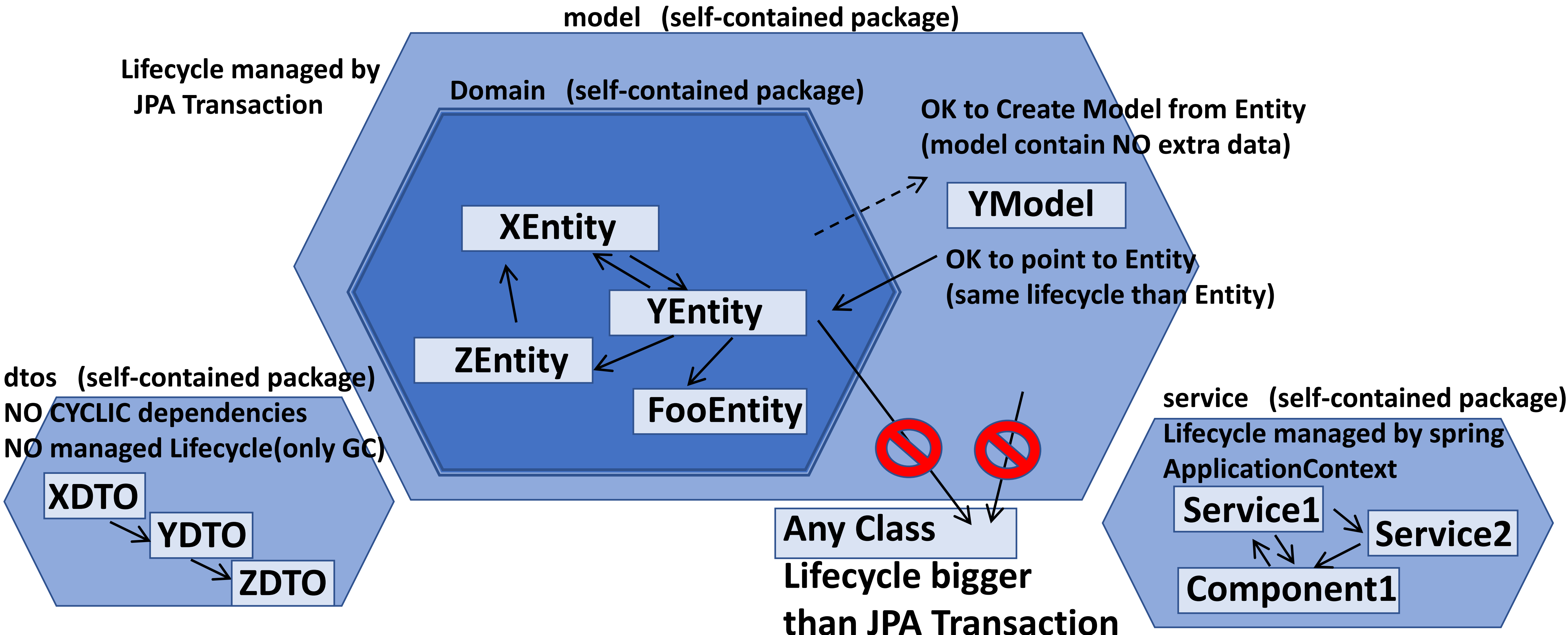
Domain (self-contained package)

OK to Create Model from Entity
(model contain NO extra data)

OK to point to Entity
(same lifecycle than Entity)

Lifecycle bigger than JPA Transaction

DTO - Entity – Model – Service
… different managed Lyfecycle



model (self-contained package)

Lifecycle managed by
JPA Transaction

Domain (self-contained package)

XEntity

YEntity

ZEntity

FooEntity

OK to Create Model from Entity
(model contain NO extra data)

YModel

OK to point to Entity
(same lifecycle than Entity)

dtos (self-contained package)
NO CYCLIC dependencies
NO managed Lifecycle(only GC)

XDTO

YDTO

ZDTO

Any Class

Lifecycle bigger
than JPA Transaction

service (self-contained package)
Lifecycle managed by spring
ApplicationContext

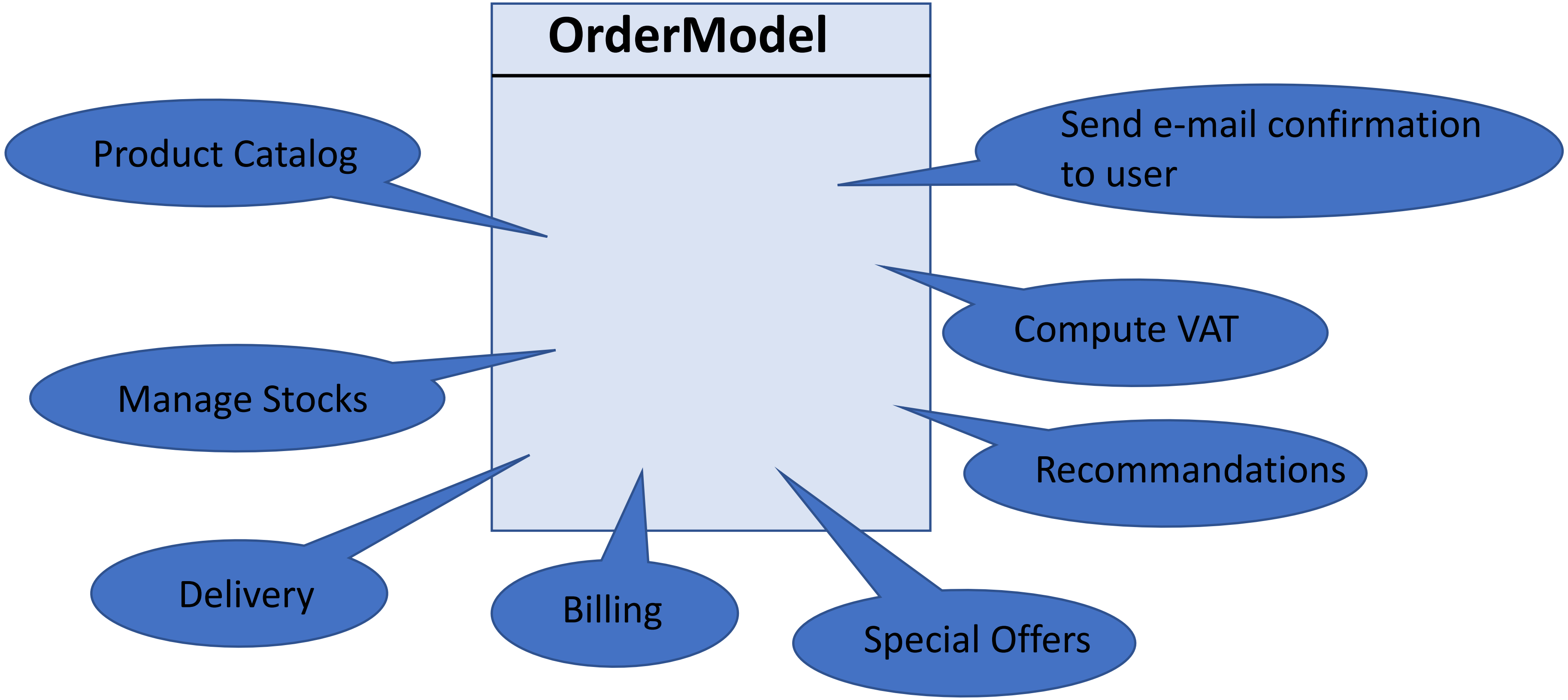
Service1

Service2

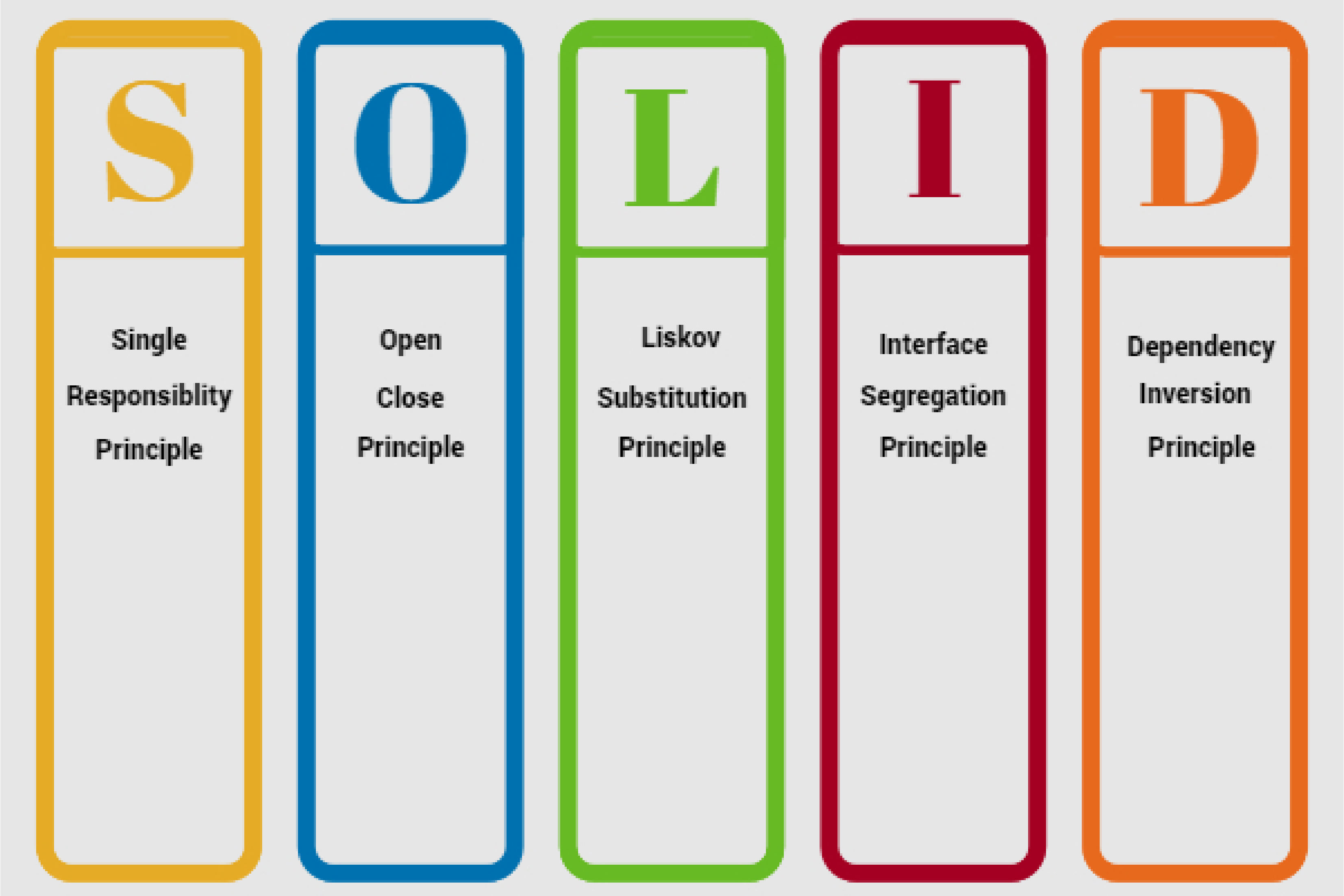
Component1

Adding business logics in Model class

Example: save « Order » on e-commerce site



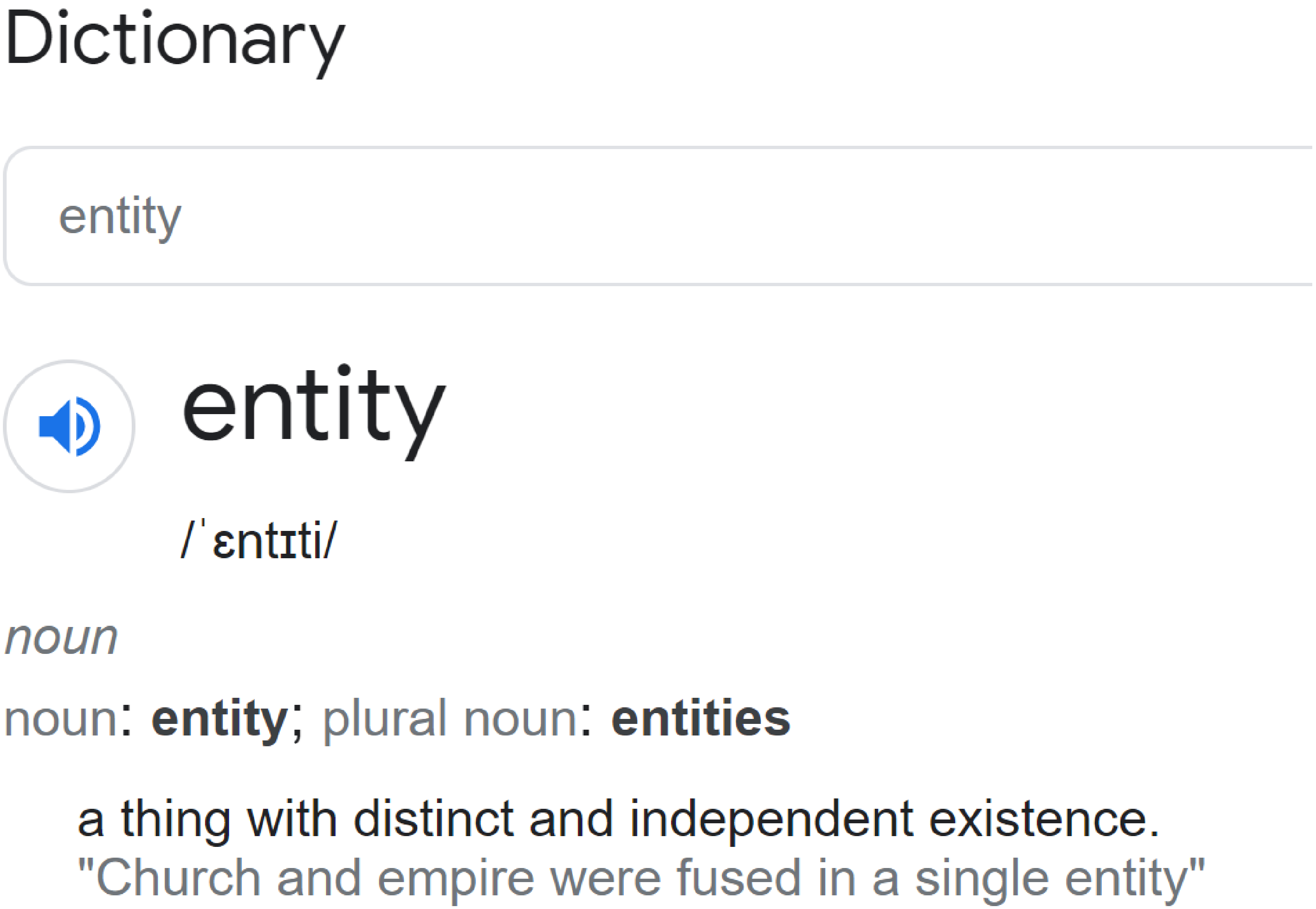
« solid » principles


S
O
L
I
D
Single Responsiblity Principle
Open Close Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

In Solid … S = **SINGLE**

A class should do only 1 thing,
given by its naming convention.
Delegate all other things to other classes

English Definition « Entity »



Dictionary

entity

entity

/ˈɛntɪti/

noun

noun: **entity**; plural noun: **entities**

a thing with distinct and independent existence.
"Church and empire were fused in a single entity"

Single » principle of SOLID for Entity:

Store some data for a given @ID

Entity should be simple POJO with NO code

Annexe

Remark Note: zoom « Entity » definition

An Entity is a « thing »

Could be whatever make sense

with distinct

They have a UNIQUE « ID » (primary key)
To distinct between objects

and independent

May contains group of dependent data/field/values/parts
but entities are independent of each others

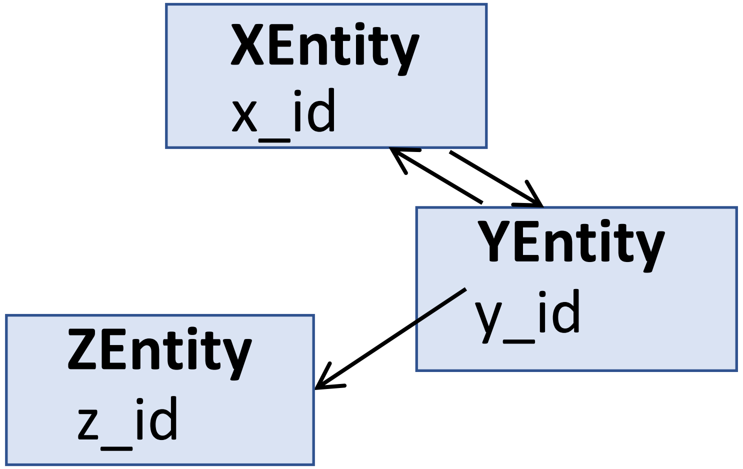
existence

Entity have a lifecycle.
They exist after you create and name them with ID.

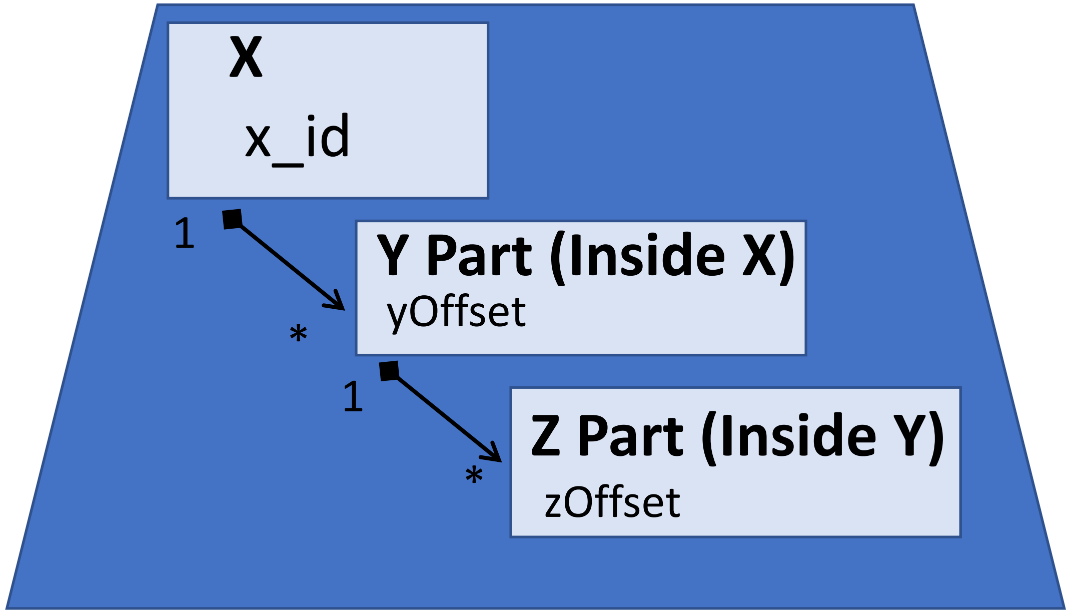
Annexe

Remark Note:
Entity versus « Aggregate » (Trees of data)

Technically, these are « entities »
There can have references to each-other
But exist independently.



Technically, these are « NOT entities »
Then all are dependent of the « Aggregate starting at X»



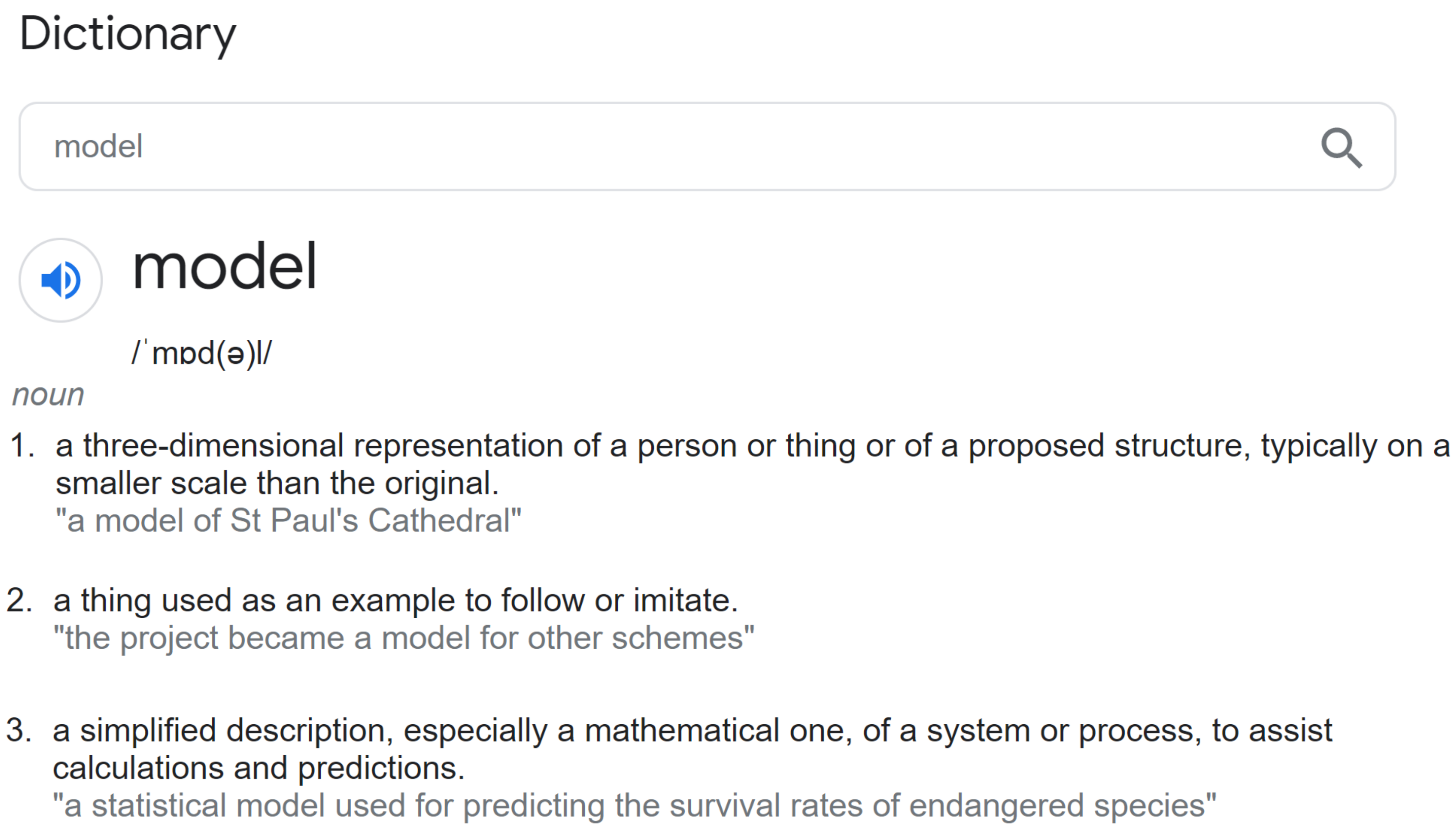
Entities with « cascade delete » rule:
- When deleting « X » => all child are deleted
- Y and Z can not be created-without /detached-from X

Put « methods » in SOLID classes

… In « Models » or « Services » SOLID classes ?

Is it necessary to have a « model(isation) » ?
Why not a simple service

Model … put more abstraction on perfection representation



Dictionary

model

model

/ˈmɒd(ə)l/

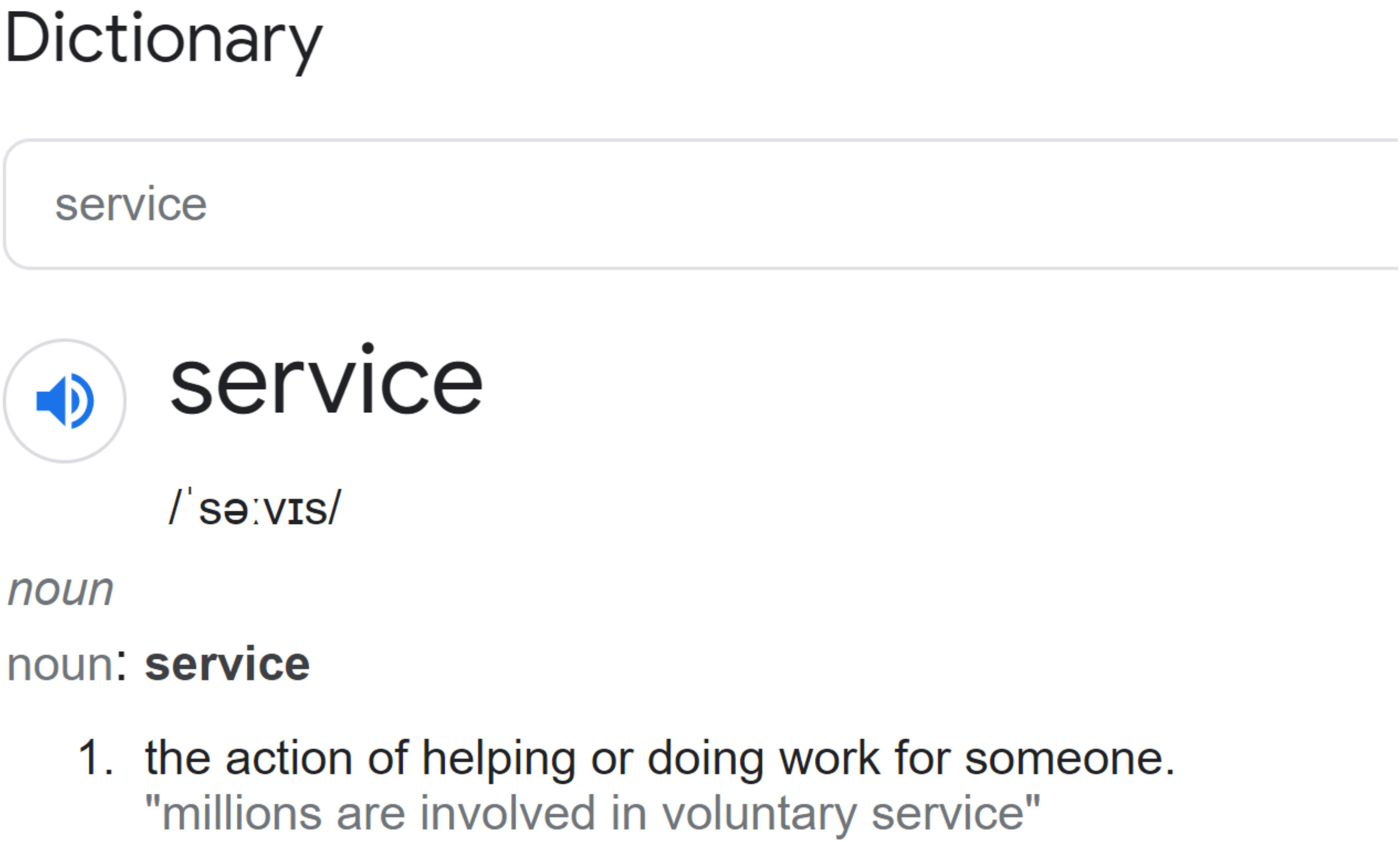
noun

1. a three-dimensional representation of a person or thing or of a proposed structure, typically on a smaller scale than the original.
   "a model of St Paul's Cathedral"

2. a thing used as an example to follow or imitate.
   "the project became a model for other schemes"

3. a simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions.
   "a statistical model used for predicting the survival rates of endangered species"

Service … where action are done



Dictionary

service

service

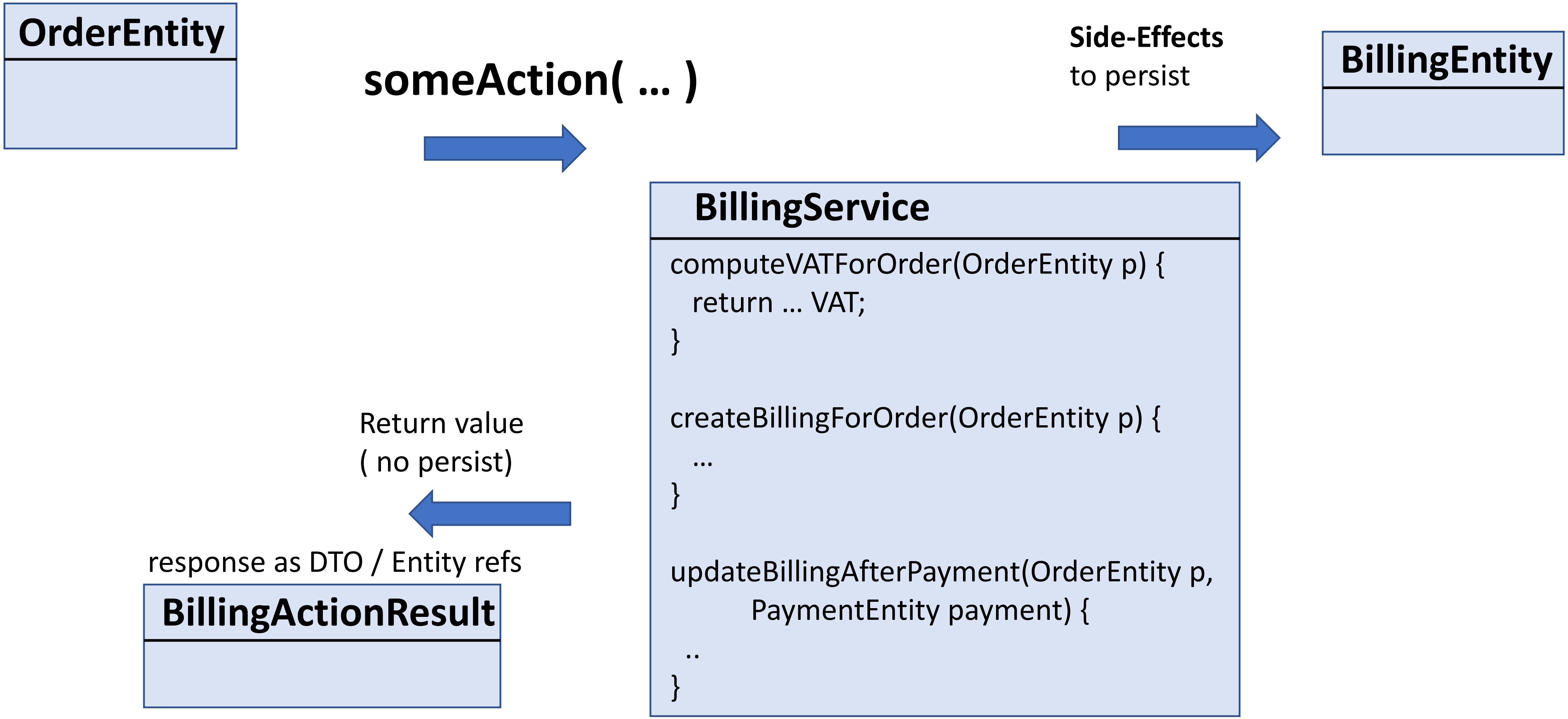
/ˈsəːvɪs/

noun

noun: **service**

1. the action of helping or doing work for someone.
   "millions are involved in voluntary service"

Move behaviors methods in N x « Services »



OrderEntity

someAction(…)

Side-Effects
to persist

BillingEntity

BillingService

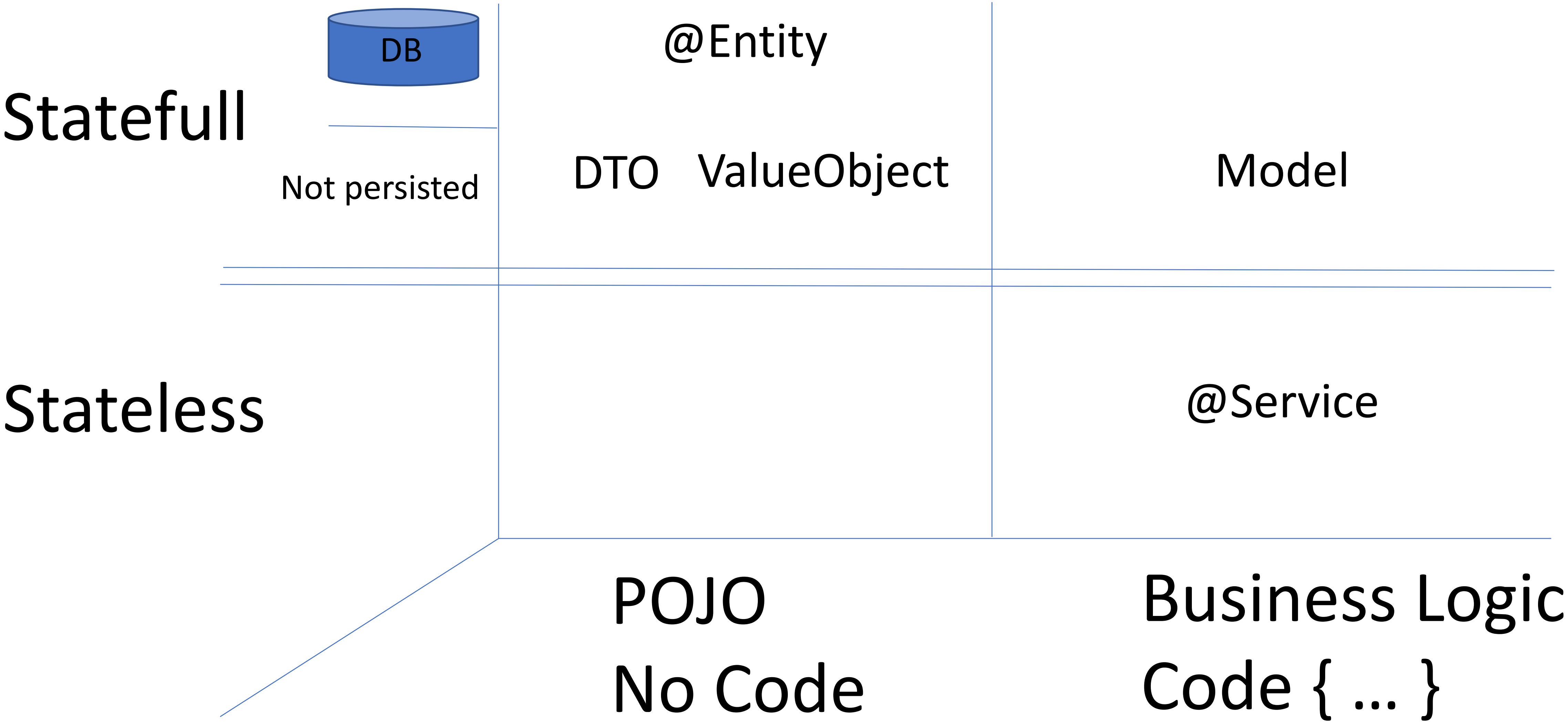
```
computeVATForOrder(OrderEntity p) {
  return … VAT;
}

createBillingForOrder(OrderEntity p) {
   …
}

updateBillingAfterPayment(OrderEntity p,
          PaymentEntity payment) {
  ..
}
```

Return value
(no persist)

response as DTO / Entity refs

BillingActionResult

Entity/Model (Statefull) vs Service (Stateless)

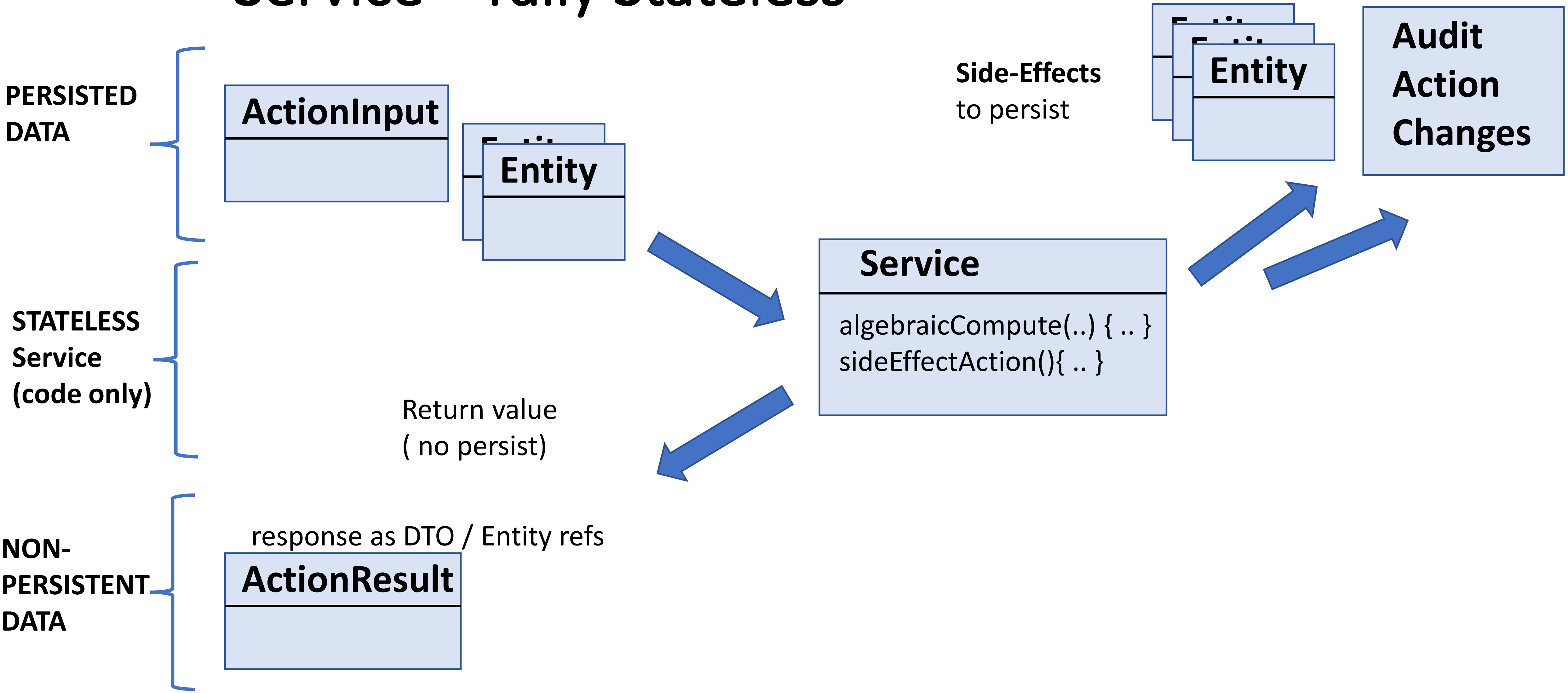


		POJO No Code	Business Logic Code { ... }
Statefull	DB	@Entity	
	Not persisted	DTO ValueObject	Model
Stateless			@Service

Entity = persist DATA + STATE between actions

Actions = may have side-effects, code in Service

Service = fully Stateless



PERSISTED DATA

ActionInput

Entity

STATELESS Service (code only)

Service

algebraicCompute(..) { .. }
sideEffectAction(){ .. }

Side-Effects to persist

Entity

Audit Action Changes

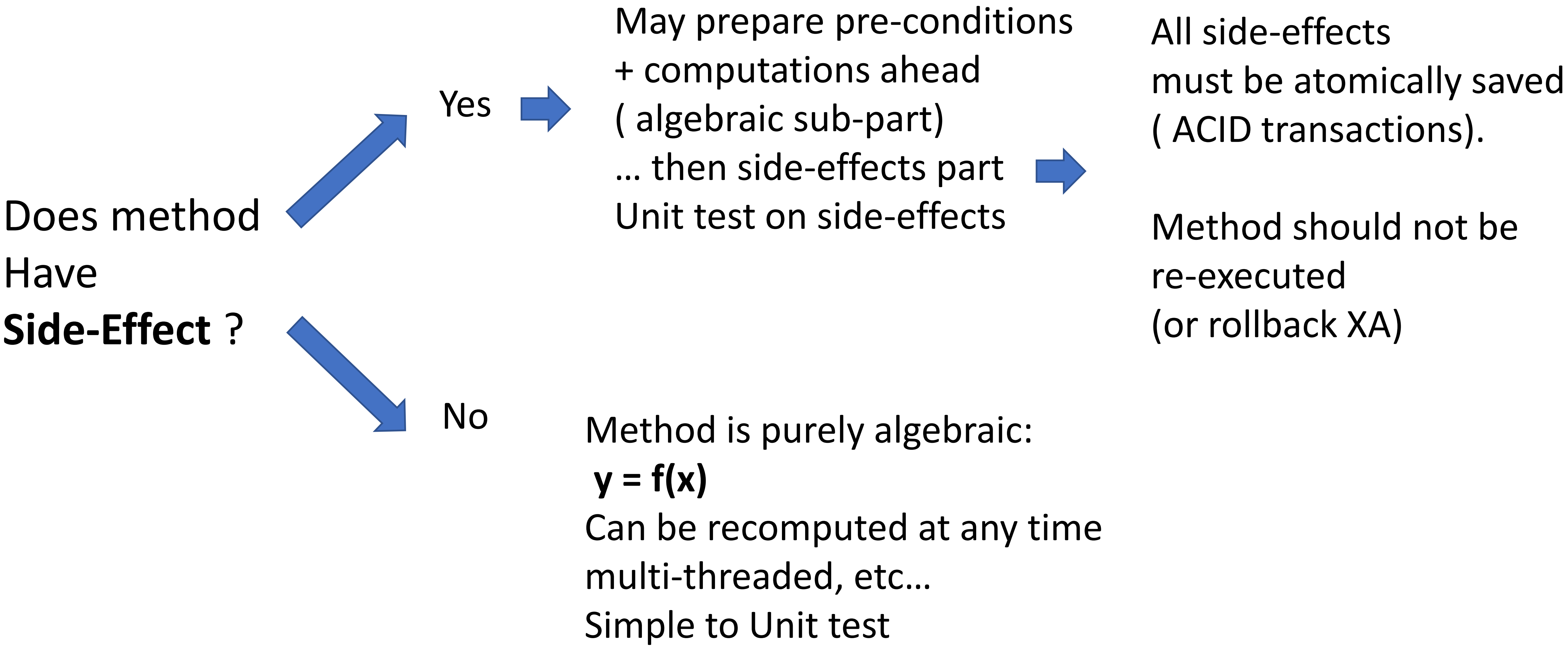
Return value (no persist)

NON-PERSISTENT DATA

response as DTO / Entity refs

ActionResult

Service Methods Dichotomy
Side-Effect ..or.. NO-Side-Effect



Does method
Have
Side-Effect ?

Yes

May prepare pre-conditions
+ computations ahead
(algebraic sub-part)
… then side-effects part
Unit test on side-effects

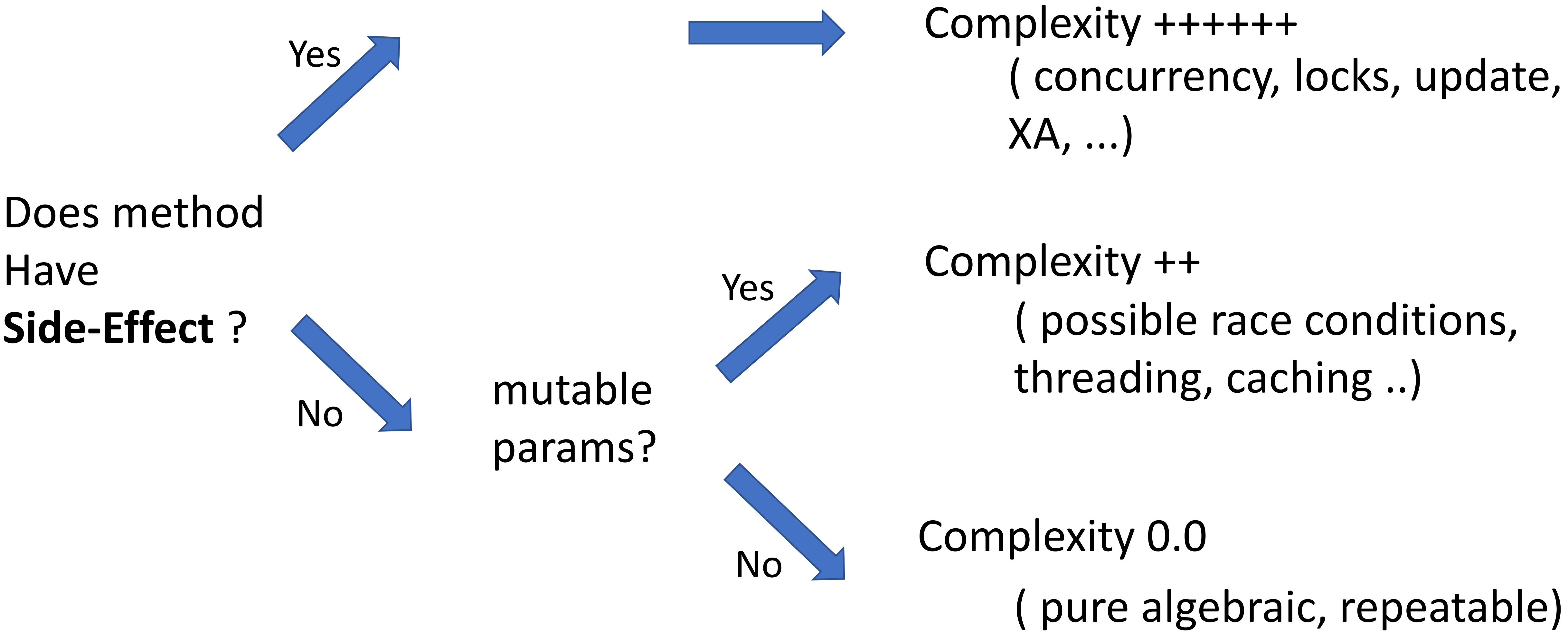
All side-effects
must be atomically saved
(ACID transactions).

Method should not be
re-executed
(or rollback XA)

No

Method is purely algebraic:
y = f(x)
Can be recomputed at any time
multi-threaded, etc…
Simple to Unit test

Where Complexity is



Does method
Have
Side-Effect ?

Yes

Complexity ++++++
(concurrency, locks, update, XA, ...)

No

mutable params?

Yes

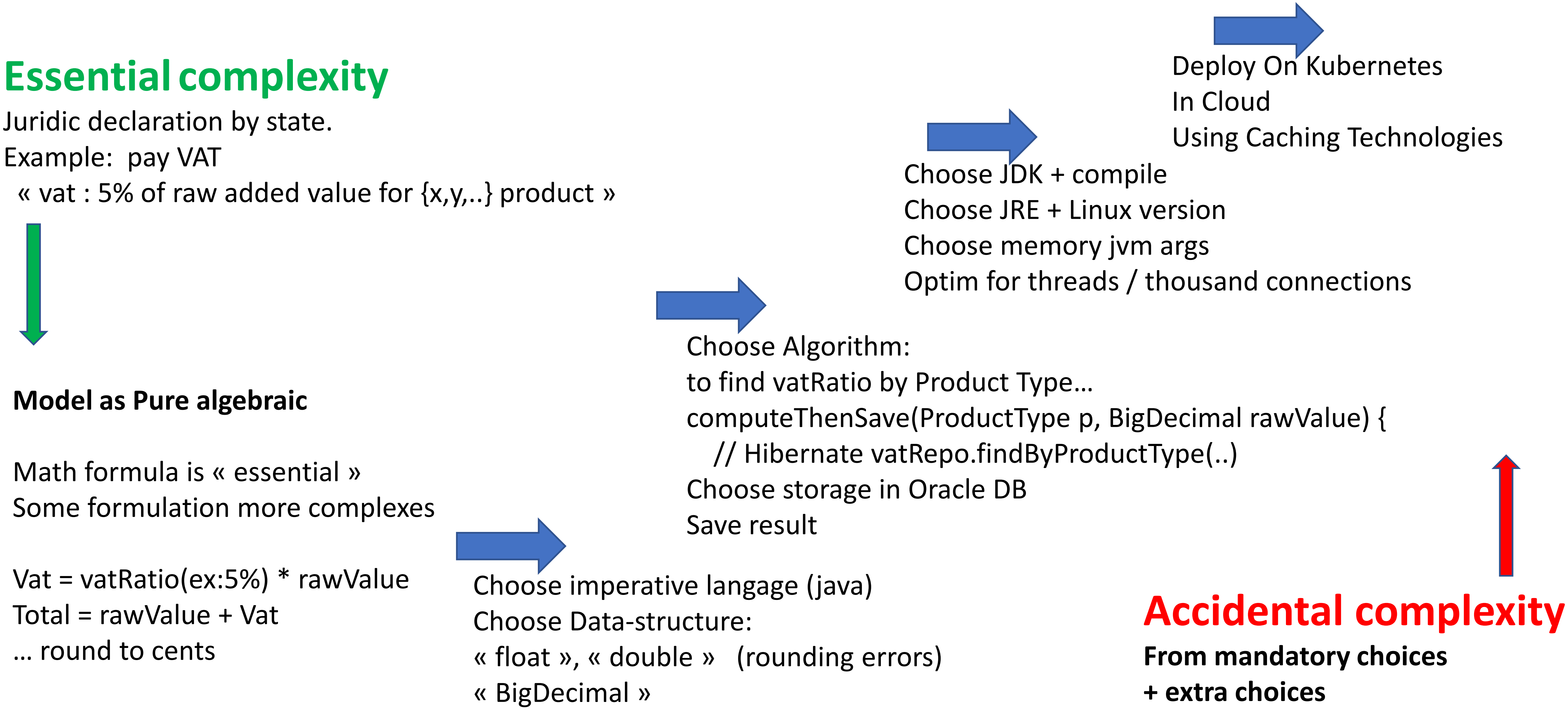
Complexity ++
(possible race conditions, threading, caching ..)

No

Complexity 0.0

(pure algebraic, repeatable)

algebraic Part / side-effect part
on « Essential vs Accidental Complexity »



Essential complexity

Juridic declaration by state.
Example: pay VAT
« vat : 5% of raw added value for {x,y,..} product »

Model as Pure algebraic

Math formula is « essential »
Some formulation more complexes

Vat = vatRatio(ex:5%) * rawValue
Total = rawValue + Vat
… round to cents

Choose imperative langage (java)
Choose Data-structure:
« float », « double » (rounding errors)
« BigDecimal »

Choose Algorithm:
to find vatRatio by Product Type…
computeThenSave(ProductType p, BigDecimal rawValue) {
 // Hibernate vatRepo.findByProductType(..)
Choose storage in Oracle DB
Save result

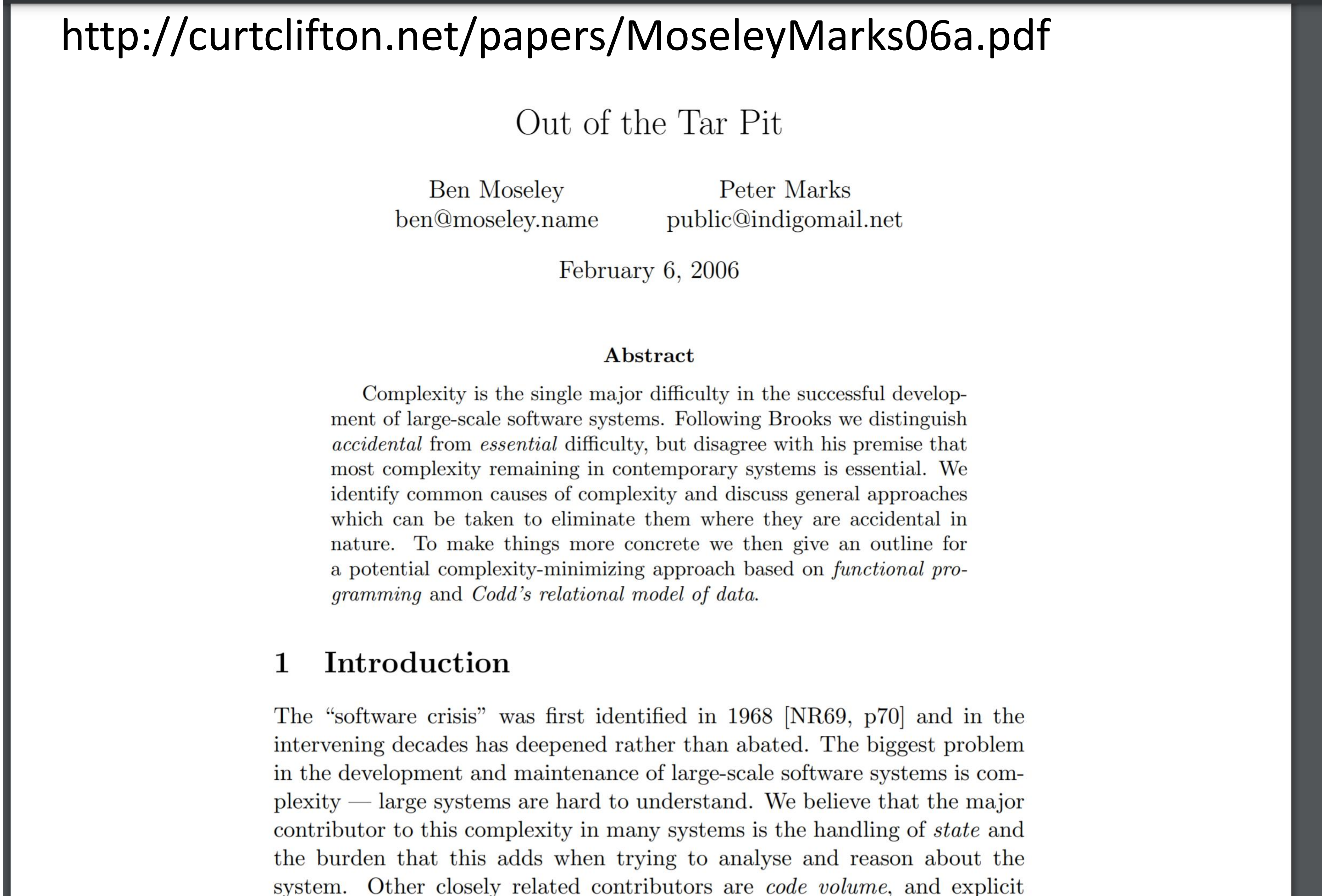
Choose JDK + compile
Choose JRE + Linux version
Choose memory jvm args
Optim for threads / thousand connections

Deploy On Kubernetes
In Cloud
Using Caching Technologies

Accidental complexity

From mandatory choices
+ extra choices

http://curtclifton.net/papers/MoseleyMarks06a.pdf



Out of the Tar Pit

Ben Moseley
ben@moseley.name

Peter Marks
public@indigomail.net

February 6, 2006

Abstract

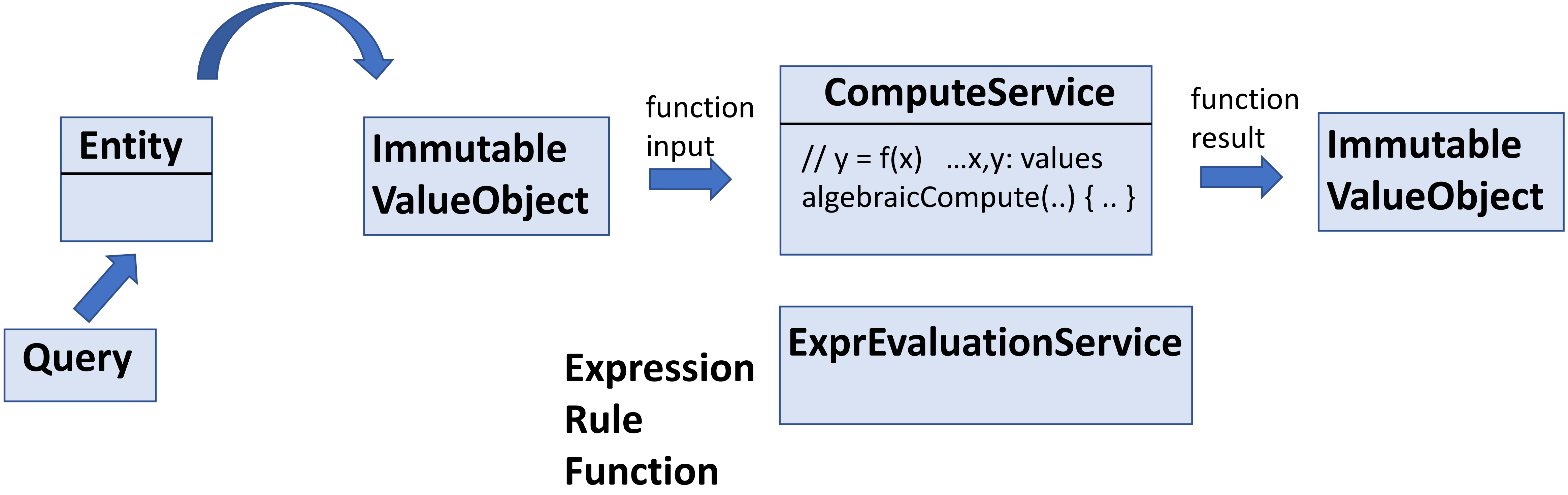
Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional programming* and *Codd's relational model of data.*

1 Introduction

The "software crisis" was first identified in 1968 [NR69, p70] and in the intervening decades has deepened rather than abated. The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the handling of *state* and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are *code volume*, and explicit

Algebraic Service Methods :
may use declarative / functional / math code on Immutable Value Objects

Convert entity to immutable model
Using getters only + constructors/builder only



Entity

Query

Immutable ValueObject

function input

ComputeService

```
// y = f(x)   ...x,y: values
algebraicCompute(..) { .. }
```

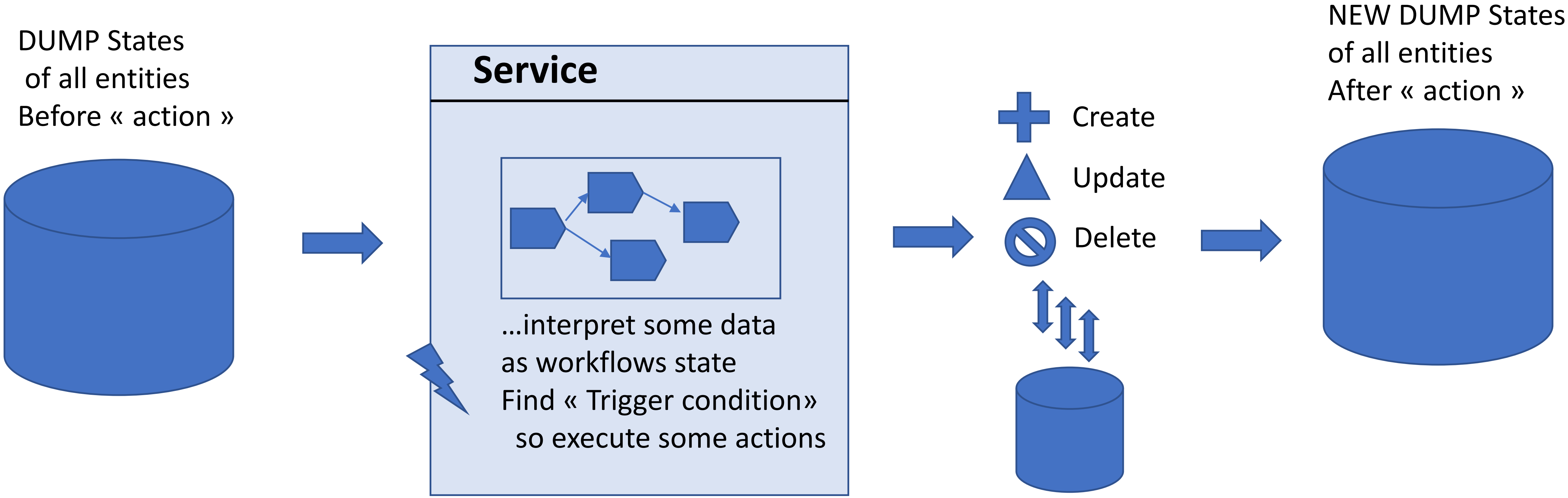
function result

Immutable ValueObject

Expression
Rule
Function

ExprEvaluationService

Side-Effect Service Methods :
may use imperative code
or Workflow engine or Rule Inference engine



DUMP States
of all entities
Before « action »

Service

…interpret some data
as workflows state
Find « Trigger condition»
so execute some actions

Create

Update

Delete

NEW DUMP States
of all entities
After « action »

Model / Service / Workflow Code / Rule Engine Code

Still need service…

Often: NO object-oriented
 specific needs (no dynamic methods)
… simpler to use « static methods»

How to split SOLID principles ?

- services?
- Visitor design pattern ?
- Adapter pattern ?
- Dynamic dispatch Adapters+Factory
 (eclipse-like fwk)

Still need Service
To prepare « WorflowContext »
by extracting Entity

Once preparation is done…
Worflow is often TOO simple
So better done in Service itself

Only very complex business workflow
Need? accidental complexity
 of JBPM / Camunda / Activiti…
In practice: < 1% of business cases?

IDEM …
Still need Service
To prepare « RuleContext »
by extracting Entity

Need? accidental complexity
 of Drules / JRules…
In practice: < 1/100000 of cases?

Choose the best Paradigm / Solution

Declarative >> Imperative

Example 1 : maven, bazel, makefile >> better than>> ant, shell scripts, graddle custom codes

Example 2: deploy on Kubernetes >>better than>> deploy using SSH custom restarts shell, or Docker Swarm

Example 3 : provisionning using Puppet,Ansible,Terraform >>simpler than>> shell, code or custom workflow engines

Choose Langage & Tool

Example: use algebraic tool for math, generate automated code from declaration

NO Silver Bullet

Do not use JBPM, or Drules if it is NOT absolutely adapted/necessary

Conclusion

Try to reduce software complexity

- Respect naming conventions
- Respect proven correct architectures
- Use SOLID principles on classes
- Read books

- Do no create too many additionnal Layer of indirections in your architecture (Interface + Command classes + Copies + Model + DAO/Repositories …)