OBJECT-ORIENTED DOMAIN DRIVEN DESIGN

Robert Bräutigam

MATHEMA Software GmbH.

GOAL

TO INTRODUCE A NEW** DESIGN PARADIGM WHICH INCREASES MAINTAINABILITY* OF SOFTWARE.

OO *AND* DDD?

ISN'T THAT REDUNTANT?

WHAT WE LEARN...

```java
public interface Animal {
    void speak();
}

public final class Cat implements Animal {
    @Override
    public void speak() {
        System.out.println("Meow...");
    }
}

public final class Dog implements Animal {
    @Override
    public void speak() {
        System.out.println("Woof...");
    }
}
```

"ENTERPRISE" CODE...

```
public interface Animal {
   String getSpeech();
}

public final class Cat implements Animal {
   private final String speech;
   ...
   @Override
   public String getSpeech() {
      return speech;
   }
}

public final class SpeechService {
   public void speak(Animal animal) {
      System.out.println(animal.getSpeech());
   }
}
```

WAIT A SECOND...

WHAT IS ENCAPSULATION THEN?

It means having public and private parts.

It means having secrets!

Having secrets means to have an effective abstraction

Effective abstraction means to solve a problem we don't have to think about ever again.

WHERE IS MY ENCAPSULATION?

```java
public interface Animal {
    String getSpeech();
}

public final class Cat implements Animal {
    private final String speech;
    ...
    @Override
    public String getSpeech() {
        return speech;
    }
}

public final class SpeechService {
    public void speak(Animal animal) {
        System.out.println(animal.getSpeech());
    }
}
```

THESE THINGS TOO?

- Cohesion
- Coupling
- Tell, don't ask
- Law of Demeter
- etc...

GO HOME AND RETHINK YOUR LIFE

OK, SO NOW WHAT?

Objects must have hidden state

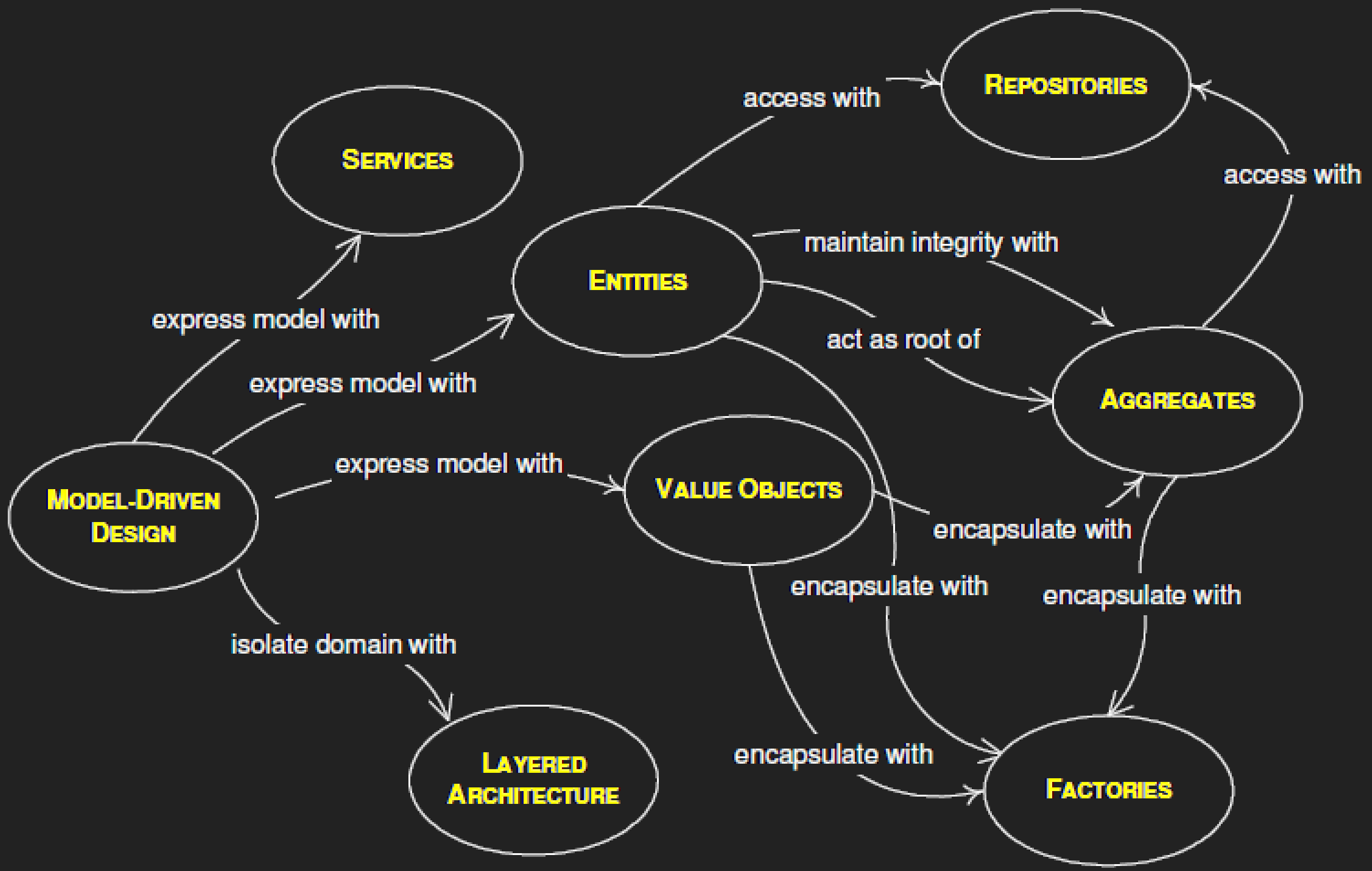
Objects should do stuff, instead of giving out data

Objects' instance variables should not be given out

⇒ SOUNDS LIKE WE SHOULD AVOID GETTERS

HOW DO WE DO DDD WITHOUT GETTERS

BUILDING BLOCKS OF DDD



VALUE OBJECTS

Things that do not have an identity. Objects representing the same value are interchangeable.

COMMONLY IMPLEMENTED AS

```
public final class Amount {
   private final BigDecimal value;
   private final Currency currency;

   public Amount(BigDecimal value, Currency currency) {
       this.value = value;
       this.currency = currency;
   }

   public BigDecimal getValue() {
       return value;
   }

   public Currency getCurrency() {
       return value;
   }

   ...equals(), hashCode(), toString()...
}
```

THAT'S NOT COOL!

There is nothing hidden.

Therefore there is no problem solved here.

This thing has no reason to exist!

HOW CAN WE FIX THIS?

The business people talk a lot about "Amounts", so let's assume it's something we need to have.

What business problem could the "Amount" solve?

OO VALUE OBJECT

```java
public final class Amount {
   private final BigDecimal value;
   private final Currency currency;

   public Amount(BigDecimal value, Currency currency) {
       this.value = value;
       this.currency = currency;
   }

   public Amount add(Amount other) { ... }

   public boolean largerThan(Amount other) { ... }

   ...
}
```

ENTITY OBJECTS

Things that have an identity. Objects are not interchangeable. Objects may represent the same conceptual thing even if some attributes differ.

COMMONLY SEEN AS:

```
public final class Customer {
    private final CustomerId customerId;
    private Name name;
    ...

    public Customer(CustomerId customerId, Name name, ...) {
        this.customerId = customerId;
        this.name = name;
        ...
    }

    ...getters, some setters...
}
```

OH NO, NOT AGAIN!

HOW IT SHOULD LOOK:

```
public final class Customer {
   ...data doesn't matter...

   public void renameTo(Name newName) { ... }

   public void freezeCreditCards() { ... }

   public void unfreezeCreditCards() { ... }

   public CreditStatement createCreditStatement() { ... }

   ...
}
```

SERVICES

"Sometimes, it just isn't a thing." -- Eric Evans

Everything is an object. -- OO

"...any decomposition, however complicated the domain, will result in the identification of a relatively few kinds of objects and only objects. There will be nothing "left over" that is not an object." -- David West

"FIXING" SERVICES

"There are important domain operations that can't find a natural home in an ENTITY or VALUE OBJECT." -- Eric Evans

Aha! It's not OO's fault, the building blocks are incomplete!

PASSWORDSERVICE

(Vaughn Vernon)

```java
public class PasswordService {
    ...no hidden state...

    public String generateStrongPassword();

    public boolean isStrong(String password);

    public boolean isWeak(String password);

    ...
}
```

WHY?

PASSWORD

```java
public final class Password {
   private final String password;

   public Password(String password) {
      this.password = password;
   }

   public boolean isStrong() { ... }

   public boolean isWeak() { ... }

   public static Password generateStrongPassword() { ... }
}
```

It's actually a Value Object.

GroupMemberService.isMemberGroup()

```java
public boolean isMemberGroup(Group aGroup, GroupMember aMember
    boolean isMember = false;
    Iterator<GroupMember> iter = aGroup.groupMembers().iterator
    while (!isMember && iter.hasNext()) {
        GroupMember member = iter.next();
        if (member.isGroup()) {
            if (aMemberGroup.equals(member)) {
                isMember = true;
            } else {
                Group group =
                    this.groupRepository().groupNamed(member.tenant
                if (group != null) {
                    isMember = this.isMemberGroup(group, aMemberGro
                }
            }
        }
    }
    return isMember;
}
```

Why not Group.contains()?

```java
public final class Group implements GroupMember {
    private final Set<GroupMember> members;
    ...
    @Override
    public boolean contains(GroupMember potentialMember) {
        if (equals(potentialMember)) {
            return true;
        }
        return members.stream()
            .filter(member → member.contains(potentialMember))
            .findFirst()
            .isPresent();
    }
}
public final class User implements GroupMember {
    ...
    @Override
    public boolean contains(GroupMember potentialMember) {
        return equals(potentialMember);
    }
}
```

REPOSITORIES

A means to get an initial reference to an object.
"Provide methods to add and remove objects... [and] methods that select objects..." -- Eric Evans

In other words a CRUD service.

PROBLEMS WITH REPOSITORIES

- Reinforces data-based thinking
- Not part of the Domain! Repositories are technical.
- Often implemented by violating encapsulation...

REPOSITORY VS. ENCAPSULATION

```
// Simplified from Vaughn Vernon's example
public class LevelDBTeamRepository {
    ...
    public void save(Team team) {
        String id = team.getTeamId().getId(); // LoD violation
        String name = team.getName(); // Privacy violation
        ...persist team to id + name...
    }
}
```

NOT OK!

PERSISTENCE OPTION #1

```java
public final class SqlCustomer implements Customer {
   private final Connection connection;
   private final String customerId;

   public SqlCustomer(String customerId,
         Connection connection) {
      this.customerId = customerId;
      this.connection = connection;
   }

   @Override
   public void freezeCreditCards() {
      connection.execute("update card set valid = 0 "+
          "where customerId = ?", customerId);
   }
}
```

PERSISTENCE OPTION #2

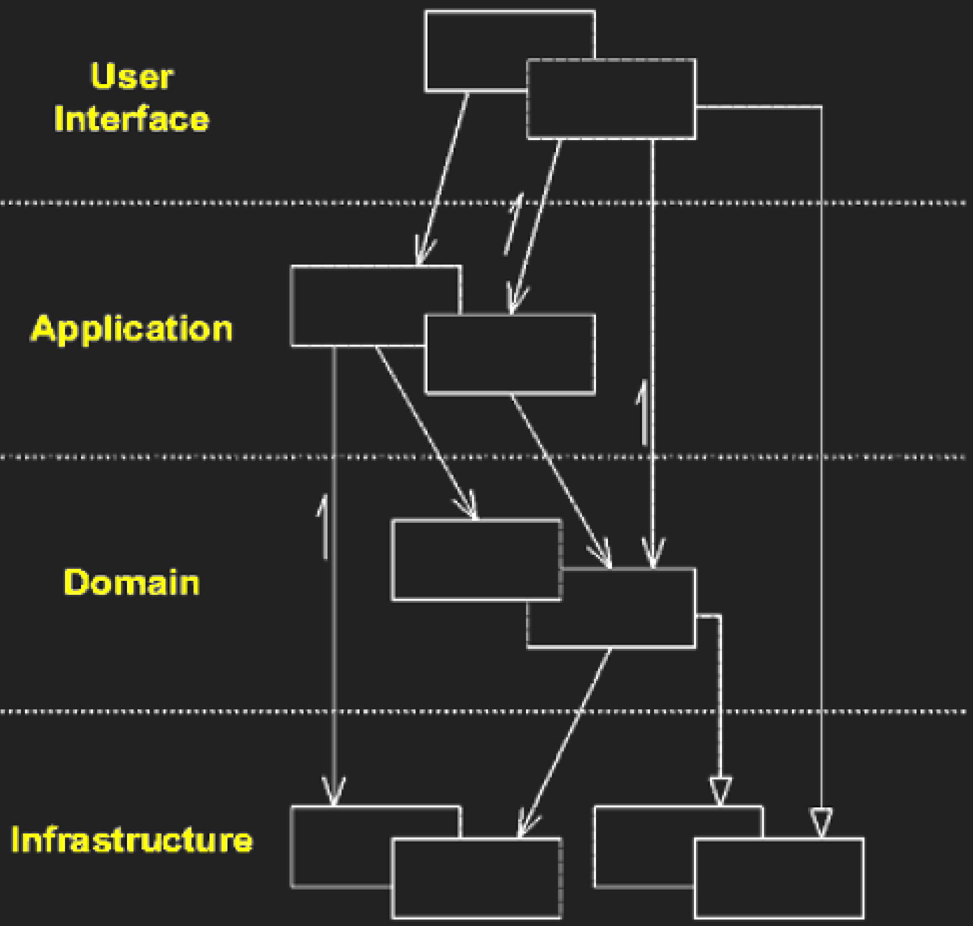
```
public final class Customer {
    ...private parts...

    public Json toJson() {
        ...
    }

    public static Customer fromJson(Json json) {
        ...
    }
}
```

AGGREGATE ROOT

Entities that exclusively control a set of internal entities and value objects. Outside objects are not allowed to hold references to internals, and the aggregate root entity controls access, preserves invariants.

Well, all objects must do this anyway...

LAYERED ARCHITECTURE



PROBLEMS WITH LAYERS

- The "Domain" is only 1/4 of the Application
- Layers usually leak data upwards and create coupling (DTOs)
- UI usually tightly coupled to Domain
- UI (external interfaces) is usually second rate citizen

UI OF OBJECTS

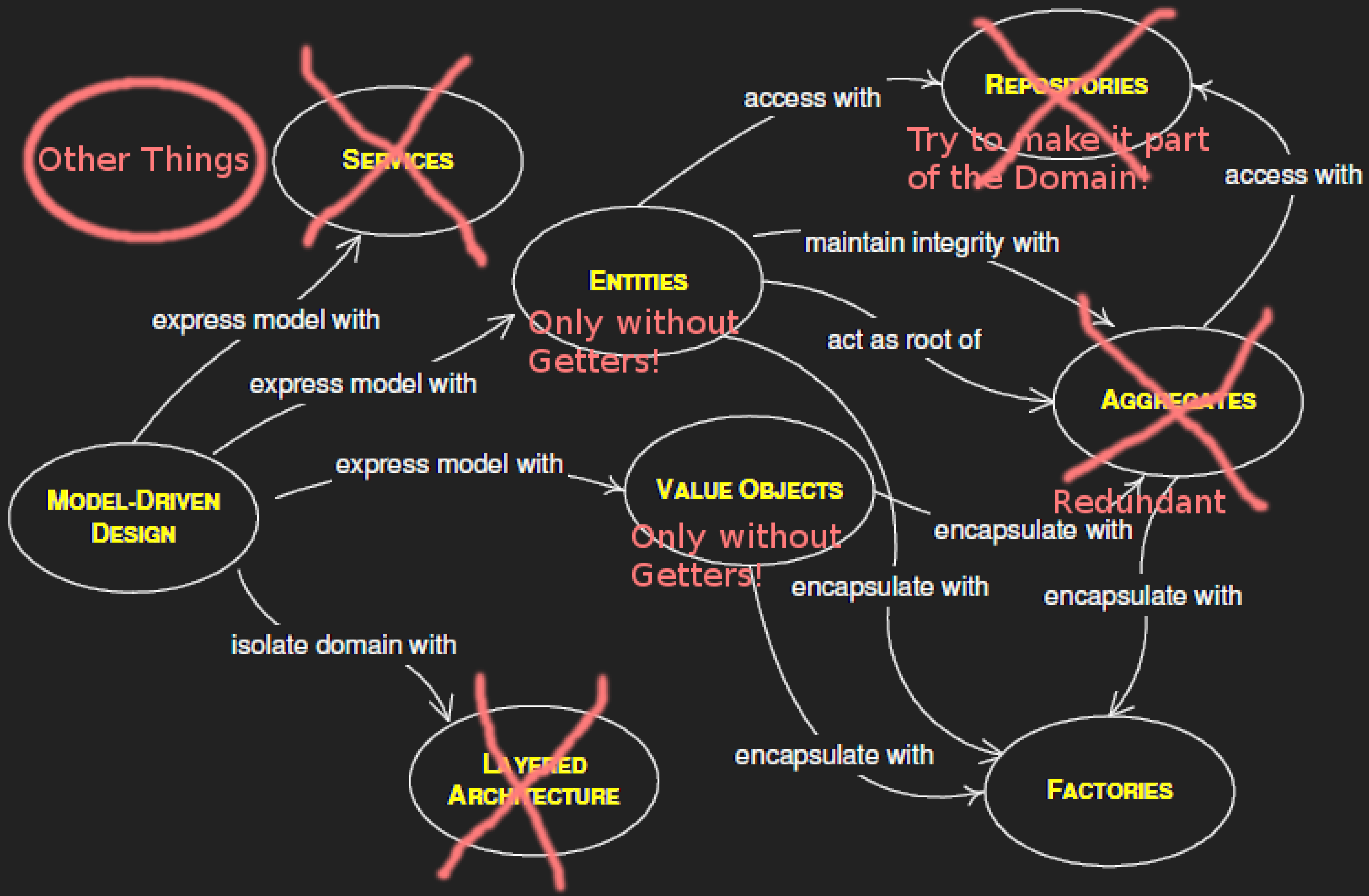
```java
import org.apache.wicket.Component;

public final class AccountNumber {
   private final String accountNumber;
   ...

   public Component display(String componentId) {
      return new Label(componentId, accountNumber);
   }
}
```

UI OF OBJECTS

```java
import org.apache.wicket.Component;

public final class AccountNumber {
    private final String accountNumber;
    ...

    public Component display(String componentId) {
        return new Label(componentId, accountNumber);
    }

    public FormComponent<AccountNumber> displayEditable(String 
        return new TextField<>(componentId, ...);
    }
}
```

SUMMARY

BUILDING BLOCKS OF OO DDD



OO DDD

Ok, maybe we should not concentrate on the orignal building blocks!

But, things that DDD adds to OO:

- Learn and think about the domain. (As opposed to technical stuff including building blocks)
- Exercise and speak the design
- Ubiquitous language
- Bounded Context

THANKS

QUESTIONS?

robert@mathema.de

https://javadevguy.wordpress.com/

@robertbrautigam