Karlsruhe Institute of Technology

Software Engineering II

Prof. Dr. Raffaela Mirandola

Topic 09

Domain-Driven Design

SASIS – SELF-ADAPTIVE SOFTWARE-INTENSIVE SYSTEMS
KASTEL – INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY

sasis.kastel.kit.edu

Overview on Domain-Driven Design Lecture

Karlsruhe Institute of Technology

Content

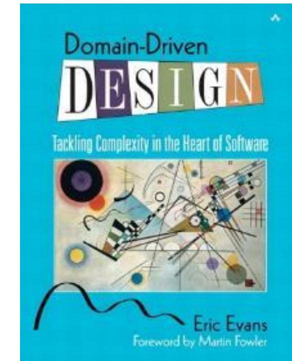
- Motivation
- Foundations
  - Bounded Context, Ubiquitous Language and Model-Driven Design
- Building Blocks
  - Entities, Value Objects, Services, and Modules
  - Factories, Aggregates, and Repositories
- Strategic Design
  - Context Map and Interactions
  - Layered Architecture

Learning Goals

- Get acquainted with ideas and principles underlying Domain-Driven Design
- Learn to distinguish and apply different building blocks
- Understand how to apply Domain-Driven Design in an arbitrary domain

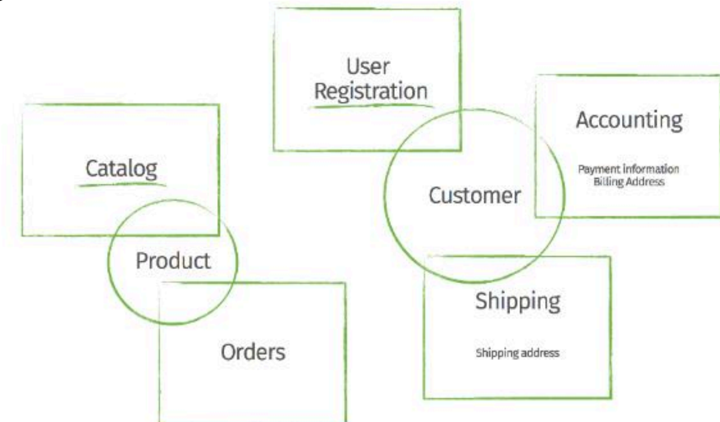
Foundations for This Lecture

- Domain-driven Design:
  Tackling Complexity in the Heart of Software
  - Core idea, principles and concepts of Domain-driven Design
  - Includes valuable insights and examples



[Evans 2004]

- Domain-Driven Design and Spring
  - Highlights concepts using a simple example
  - Additionally, discusses relation to Spring Framework
    (not part of this lecture)



[Gierke 2019]

Who’s that Guy?

“The heart of software is its ability to solve ***domain-related problems*** *for its user. All other features, vital though they may be, support this basic purpose.”*

– [Evans 2004, p. 5]

Demystifying the Magic

- How can we understand the Problem?
- How can a solution be encoded?
- How can this solution be iteratively developed/improved?

Problem / opportunity in the real world

Common Language

???

Code

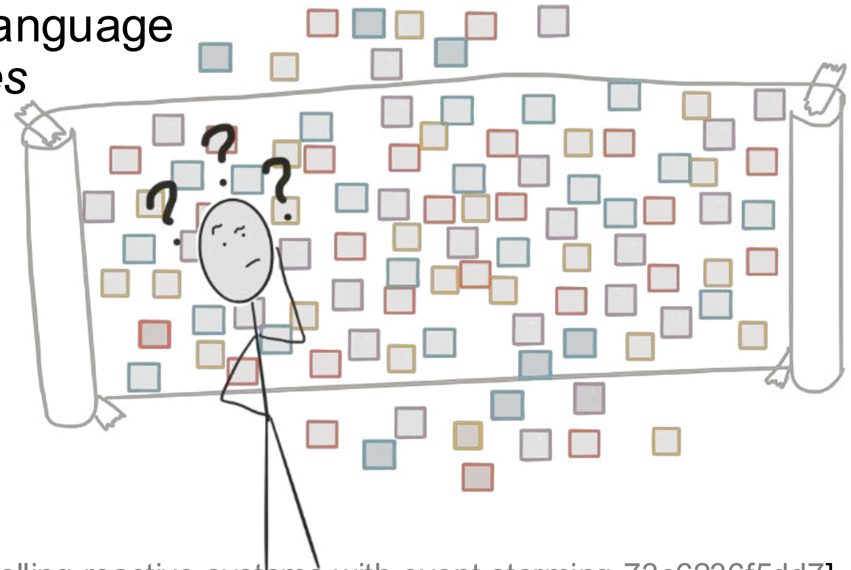
Motivating Example

E-Commerce System

- *Create a Webshop for a big/medium shop*
- Stakeholders:
  - consumers, merchants, order manager, finance, …
  - card issuer, payment processors, payment gateways, …

However,

- each stakeholder with different view / language to describe *requirements* and *use cases*
- Almost impossible to define suitable
  - *common language* or
  - *domain model*
- Business rules and logic are easily *lost in translation*



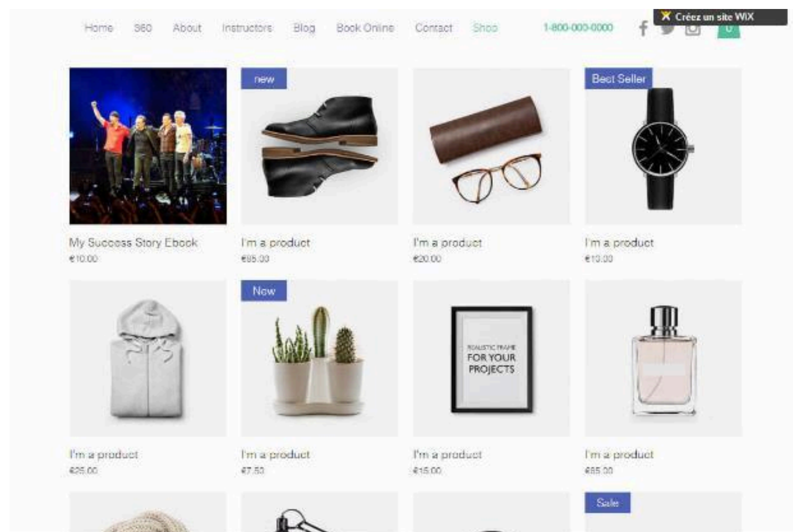
[https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7]

Motivation: First Design

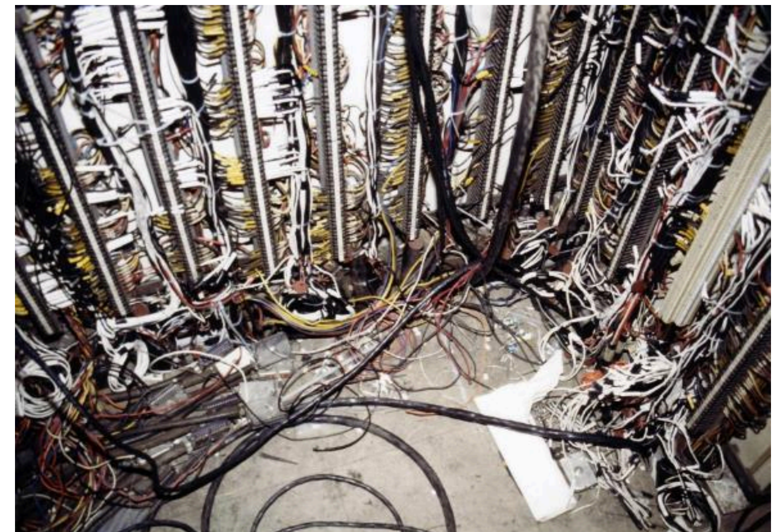
Smart UI (Anti-pattern)

- All functionality concentrated in *webshop's website*
  - Frontend delivering html, database on same machine
- Everything integrated into frontend

How it looks on the outside.



How does it look on the inside?



[*Unrelated example website* https://www.websitetooltester.com/testberichte/wix/ecommerce/]
[*Cable salad* by Achim Hering (CC BY 3.0) https://de.wikipedia.org/wiki/Datei:Cable_salad.jpg]

Motivation: Second Design

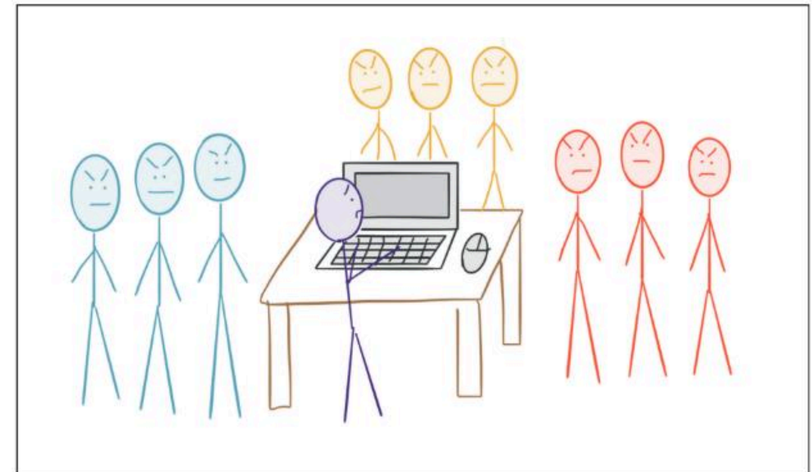
Enterprise Architecture

Employ Layered Architecture

- Presentation Layer:
  encompasses GUI / Views for all stakeholders
- Domain Layer:
  contains *common Domain Model* and use cases
- Data Source:
  handles data persistency and collaboration with payment providers

Better design, but

- Domain Model becomes unwieldy due to
  - number of concepts from different views
  - number of use cases
- Developers like to work on
  Data Source layer (backend)
- UI Designer like to work on GUIs
- Actual domain model and *business logic*
  implemented by interns



[https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7]

Domain-Driven Design

Karlsruhe Institute of Technology

Application Domain

- Domain a software system deals with
- Software systems might have many application domains
- Example webshop: *user registration, orders, accounting, shipping, …*

Domain-Driven Design in a Nutshell

1) Identify bounded context of application domains
2) For each application domain
   1) Extract the application domain's ubiquitous language
   2) Distill domain model from that language
   3) Reflect this model in working software using building blocks
   4) Employ layered architecture to develop domain application (RR: well, that book is from 2002, today you may consider better a clean architecture)
3) Deploy and Link domain application as indicated by context map

RR: today: consider realising each "domain application" as microservice.

FOUNDATIONS

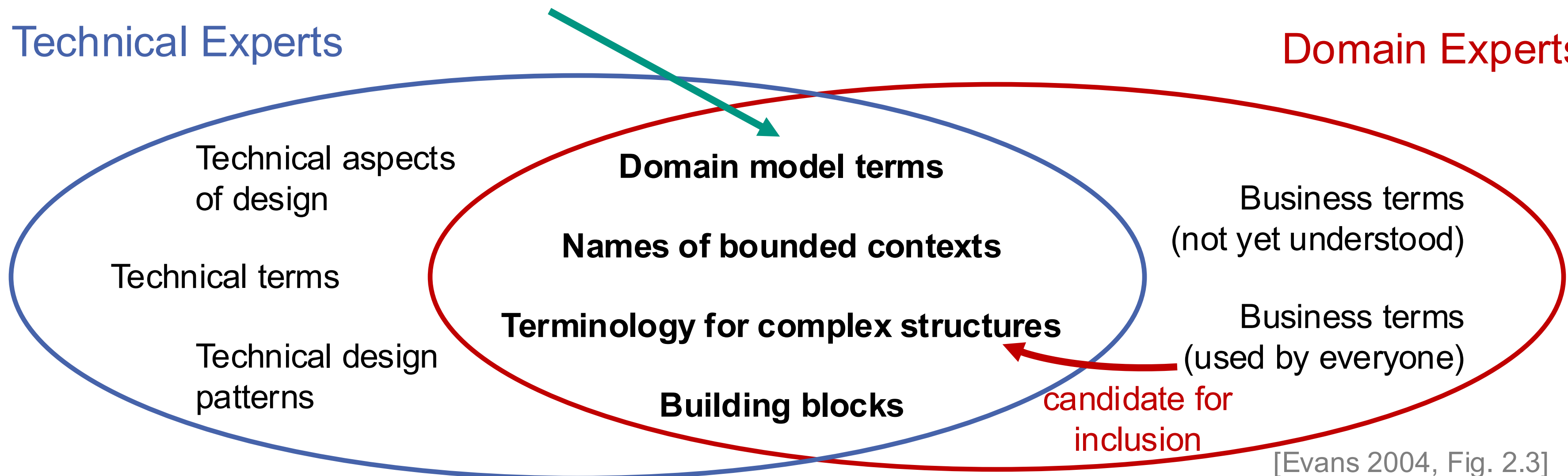
Towards Ubiquitous Language

[https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7]

Ubiquitous Language

- Software development involves translating domain concepts into program constructs
- According to Evans, ”translation blunts communication and makes knowledge crunching anemic” [Evans 2004, p.5]
  - Domain experts use jargon
  - Technical experts think in terms of technical terms
  - Domain terminology used is disconnected from terminology used in code
- Domain knowledge is “Lost in Translation”

Use common Ubiquitous Language for both design and implementation



[Evans 2004, Fig. 2.3]

Ubiquitous Language in Context

KIT Karlsruhe Institute of Technology

- Classical approach: include all application domains in one large model
- However, any attempt to model it as a whole is doomed to fail as
  - number of stakeholders increase complexity and
  - their views on domain might be very different
- *Impossible* or *infeasible* to craft single, unique model satisfying all of their needs

Ubiquitous language only meaningful in one domain (context)

1) Separate application domains into bounded contexts
2) Establish ubiquitous language for each bounded context
3) Map concepts shared between contexts in context map

Bounded Context

KIT Karlsruhe Institute of Technology

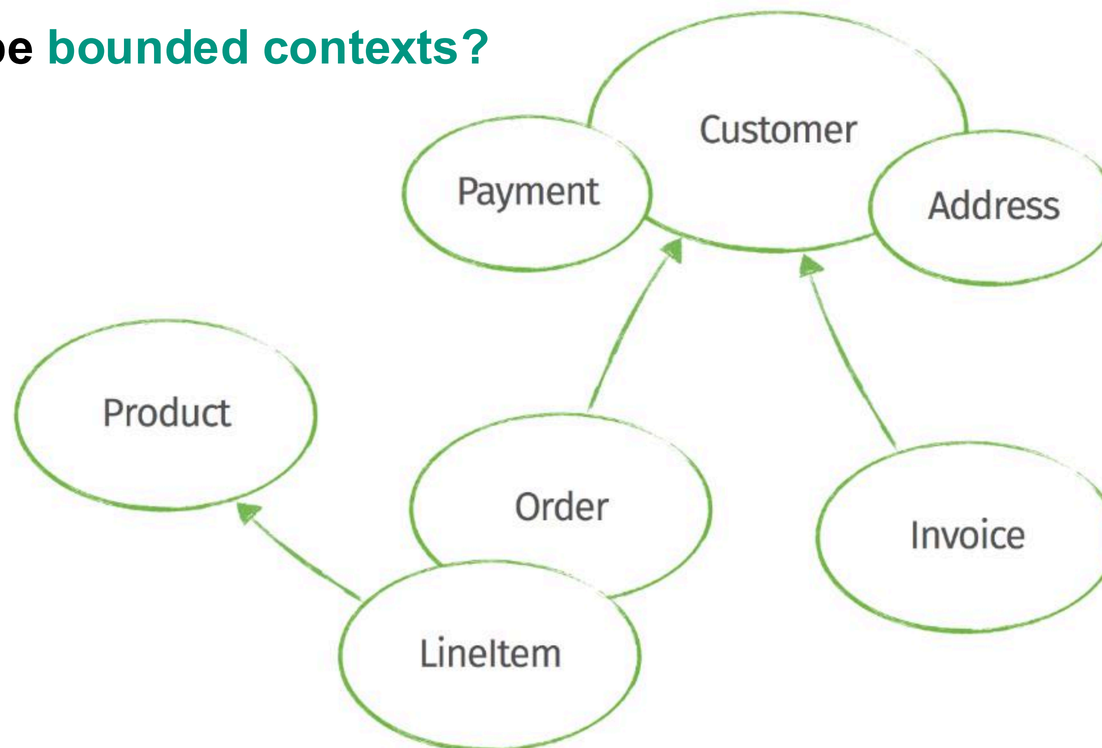
- Declares definitional boundary (scope) for ubiquitous language
- Depends on
  - Application domain *(e.g., actors supported by application)*
  - Team organization *(e.g., number of teams working on project)*
  - Physical manifestation *(e.g., code base and database schemas)*

- Strictly enforce consistency of ubiquitous language and domain model within bounds
- Focus on issues inside your boundary
  - *"don't be distracted or confused by issues outside."* [Evans 2004, p.336]
- Define relations between bounded context in context map
  - Which entities are shared?
  - Which domain events are send between them?

Example: Ubiquitous Language

- Sketched concepts in webshop domain
- Terms for ubiquitous language of webshop design
- Some concepts *closer* related then others

What could be bounded contexts?



[Oliver Gierke, CC BY-NC-SA 4.0 http://static.olivergierke.de/lectures/ddd-and-spring/]

Example: Bounded Context

- Sketched bounded concepts in webshop design

Which concepts are shared between contexts?

User Registration

Catalog

Accounting

Orders

Shipping

[Oliver Gierke, CC BY-NC-SA 4.0 http://static.olivergierke.de/lectures/ddd-and-spring/]

Example: Context Map

- Customer is shared between three contexts
  - Customer plays different role in each
  - Different information associated to customer
- Products shared between Catalog and Orders
  - *LineItems* refer to *Products*

User Registration

Catalog

Product

Orders

Customer

Accounting

Payment information
Billing Address

Shipping

Shipping address

[Oliver Gierke, CC BY-NC-SA 4.0 http://static.olivergierke.de/lectures/ddd-and-spring/]

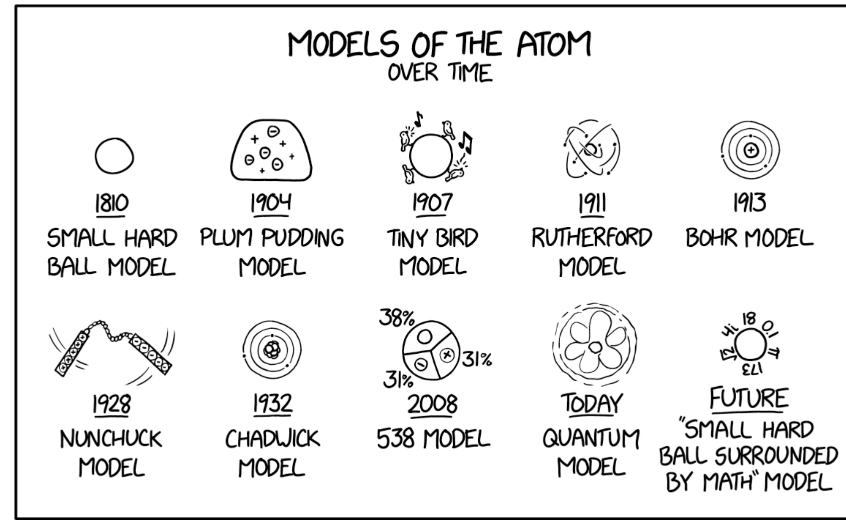
Model-Driven Design (MDD)

"Tightly relating the code to an underlying model gives the code meaning and makes the model relevant."

– [Evans 2004, p. 47]

- Distill ubiquitous language in **domain model**
  - Represents a selective abstraction of domain knowledge
  - Rigorous organization of domain concepts
  - Encodes ubiquitous language used by all team members
- Code reflects domain model in literal way
  - Employ domain terminology
  - Derive responsibilities from domain knowledge
- Tie implementation to domain model
  - Code expresses domain model
  - Employ suitable modeling paradigm *(e.g., Object-orientation)*
  - Changes in code must be reflected in domain model and vice versa
- *Model-Driven Engineering* provides tool support for MDD
  (discussed in another Lecture/Course)

Ingredients of Effective Modeling



“All models are wrong but some are useful.” – [Box 1976]

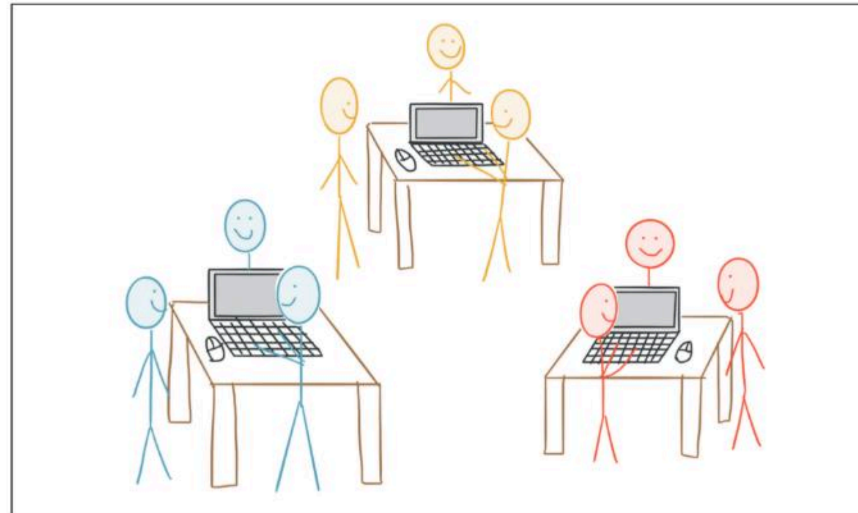
Effective Domain Models

- encodes deeper understanding of particular domain
- includes all relevant, but not more, concepts to solve *domain-related problem*
- capture and cultivate language used within a team
- permits detecting ambiguities or awkwardness when discussing design choices
- **do not** expose technical implementation details

[Models of the Atom by Randall Munroe https://xkcd.com/2100/]

Third Design: Domain-Driven

- Design splits webshop application into parts wrt. bounded contexts and number of teams
- Each team establishes own ubiquitous language and domain model
- Teams rely on context map to organize interactions between bounded contexts
- Teams build individually testable and deployable parts of application

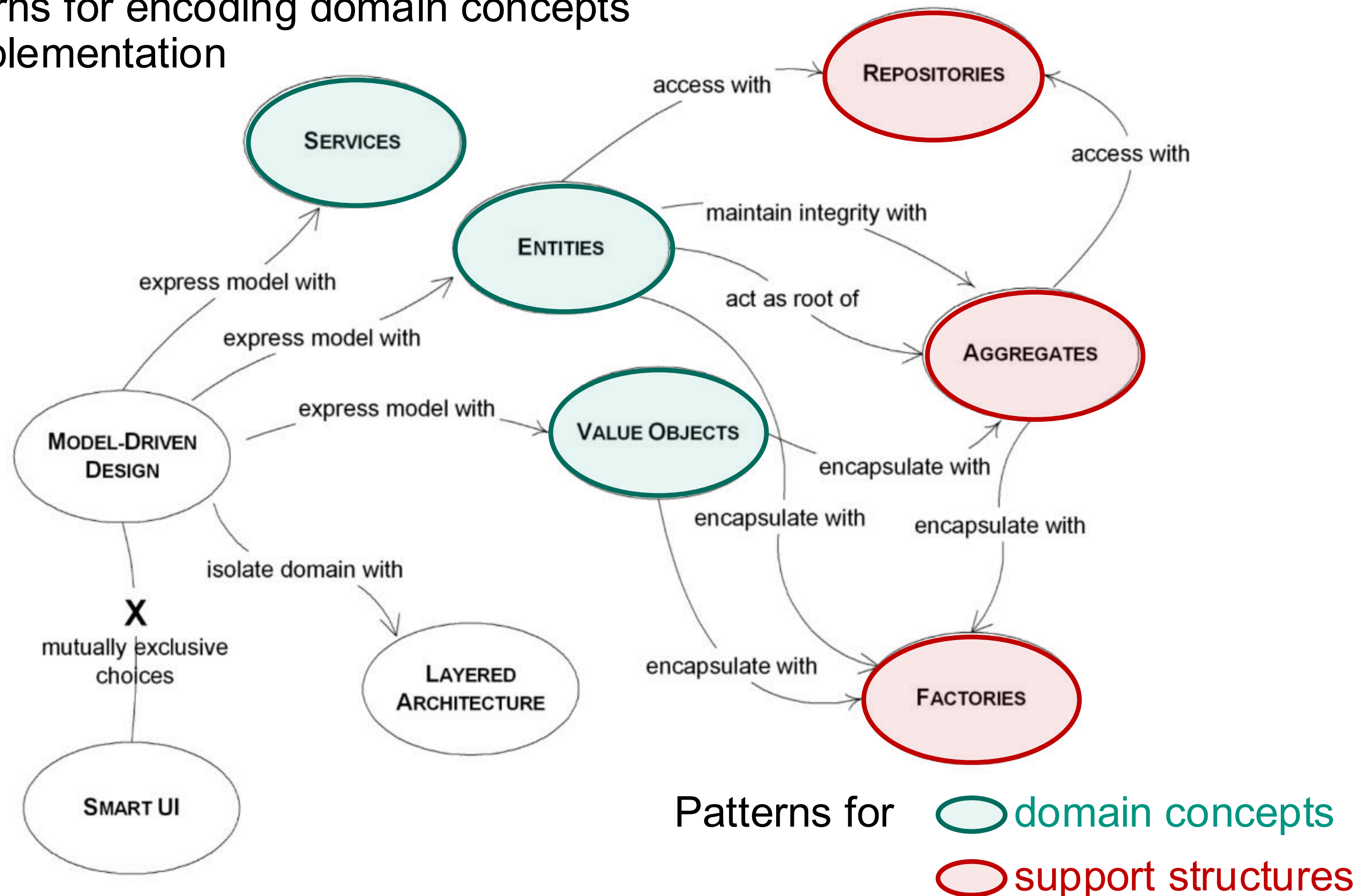


[https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7]

BUILDING BLOCKS

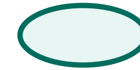
Overview on Building Blocks

- Patterns for encoding domain concepts in implementation



[Oliver Gierke, CC BY-NC-SA 4.0 http://static.olivergierke.de/lectures/ddd-and-spring/]

Entities



"Many objects are not fundamentally defined by their attributes, but rather by a thread of continuity and identity."

– [Evans 2004, pp. 89]

Entity denotes domain concept

- with a unique identity
  - established upon its creation; preserved during store, load, and transmission
  - (usually) not defined by its attributes
- with continuity throughout its life cycle
  - traceable / accountable throughout application
  - distinction relevant for domain users

Example: Entities

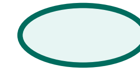
Entities in webshop application

- *Customer, Product, Order, …*
- In Java, employ `@Entity` JPA annotation
- Add *auto-generated* identity field

```
@Entity
public class Customer {
  private @Id @GeneratedValue long id; //auto-generated identity
  private String address;
  @OneToOne //Association to UserAccount
  private UserAccount userAccount;

  public Customer(UserAccount userAccount, String address) {
    this.userAccount = userAccount;
    this.address = address;
  }
  public long getId() { return id; }
  public String getAddress() { return address;}
  public void setAddress(String address) {this.address = address;}
  public UserAccount getUserAccount() {return userAccount;}
}
```

Value Objects



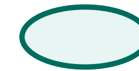
"Many objects have no conceptual identity. These objects describe some characteristic of a thing."

– [Evans 2004, pp. 97]

Value Object denotes domain concept

- *"that describe other things"*
- without identity
  - identity derived from attributes
  - distinguishable only by attribute values
- that are immutable
  - permits safe sharing, duplication, and distribution
  - changeably by full replacement

Example: Value Objects



Karlsruhe Institute of Technology

Value Objects in webshop application

  - *Quantity*, *Monetary Amount, …*
- In Java employ `@Embeddable` annotation
- All fields must be not null and immutable

```
@Embeddable        //directly embed Quantity in referencing table
@EqualsAndHashCode //generate equals and hashCode methods
public class Quantity {
  private @NonNull BigDecimal amount;
  private @NonNull Metric metric; //Enum for units of measurement
  //prevent arbitrary construction
  private Quantity(BigDecimal amount; Metric metric){ /*…*/ }
  public static Quantity valueOf(double amount, Metric metric){
    return new Quantity(BigDecimal.valueOf(amount), metric);
  }
  /* other Factory Methods */
  public Quantity add(Quantity other) {
    assertCompatibility(other); //throws excpetion if incompatible
    return new Quantity(this.amount.add(other.amount), this.metric);
  }
  /* other Arithmetic operations */
}
```

Services

“Sometimes, it just isn't a thing. Some concepts from the domain aren't natural to model as objects.“

– [Evans 2004, pp. 104]

Service denotes domain concept

- that encode functionality
  - not uniquely assignable to entity of value object
  - carries meaning in domain model
  - Interface defined in terms of entities and value objects
- that is stateless
  - regardless of complexity of operation / transformation
  - permits easy access and distribution
- if not stateless, consider instead modelling as entity with its methods

Example: Services

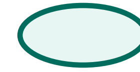
Services in webshop application

- Customer Managemer, *Order Manager*, *Accountancy*, …

- JavaEE permits design as `@Stateless` Session beans
- Otherwise implementation with `Singleton` design pattern

```
@Stateless //Declares stateless session bean
public OrderManager implements OrderManagerRemote{
  @Override
  public void completeOrder(Order order) throws OrderCompletionFailure{
    if (order!=null)
      throw new IllegalArgumentException("Order must not be null!");
    if (!order.isPaid())
      throw new OrderCompletionFailure(order, "Order is not paid yet!");
    order.complete();
  }
  @Override
  public boolean payOrder(Order order, Payment p){ /*…*/ }
  @Override
  public boolean cancelOrder(Order order) { /*…*/ }
}
```

Entity of Value Object: It depends…



Distinction is domain-dependent

- Managing many entities might induce performance penalty
- Only promote concepts to entities if necessary
- Immutable value objects can be freely shared and distributed
- Decision depends on domain

Example Concept: `Address` [Evans 2004, pp. 98]

The webshop domain

- `Address` is used as invoice and shipping destination
- Yet, customers can have the same address

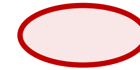
→ model `Address` as ___________________________

The postal service domain

- Manages hierarchy of regions, citys, postel zones, blocks, and finally addresses
- Postal service might reassign postal zones affecting all addresses

→ model `Address` as ___________________________

Aggregates



Karlsruhe Institute of Technology

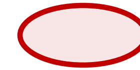
“Aggregates mark off the scope within which invariants have to be maintained at every stage of the life cycle.”

– [Evans 2004, pp. 135]

Aggregate clusters related entities and value objects

- **defines boundaries around specific root entities**
  - root entity serves as Façade to access enclosed elements
  - identity of aggregate derived from root entity
  - removal of root removes complete aggregate
- ensures invariants and concurrent access
  - prohibit references to elements other than its root
  - changes to aggregate only through root ensuring invariants
  - constructed as a whole satisfying invariants
  - locking strategy decided for whole aggregate

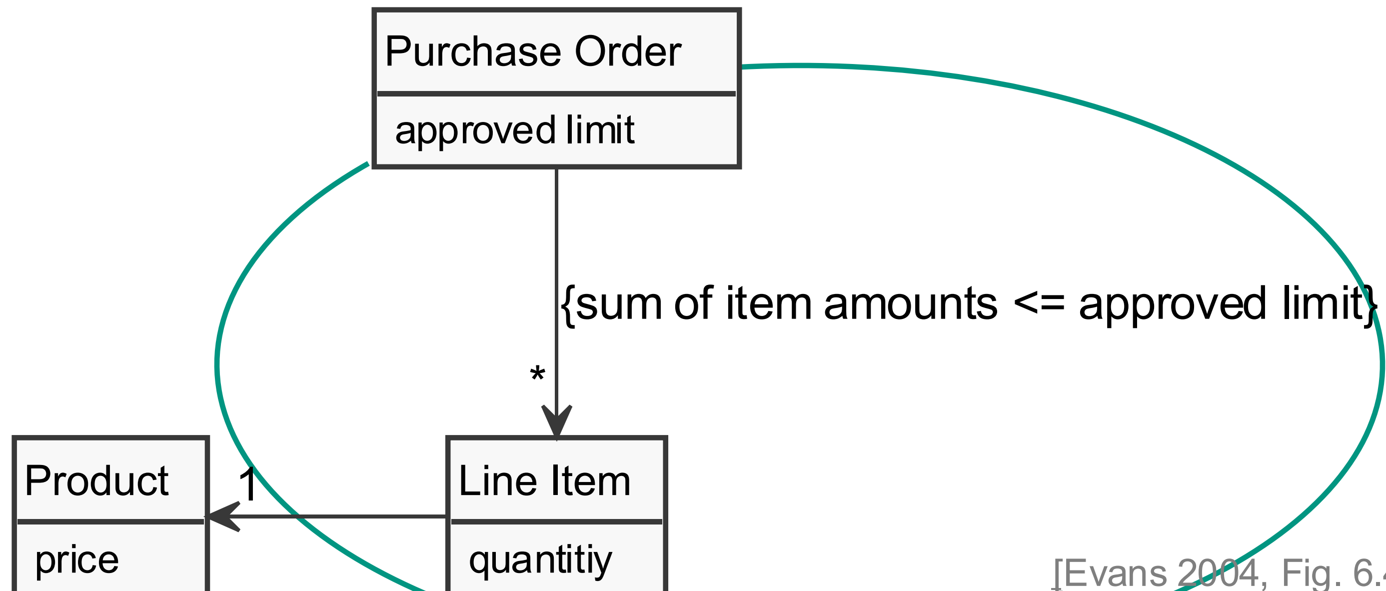
Example: Aggregates



Aggregates in webshop application

- *<u>Purchase Order</u>, Shopping Cart, Accountency…*
- Boundary delineates elements belonging to aggregate
- Access to elements only via root entity
- Sometimes requires introducing data redundancies

Where to draw the boundary?



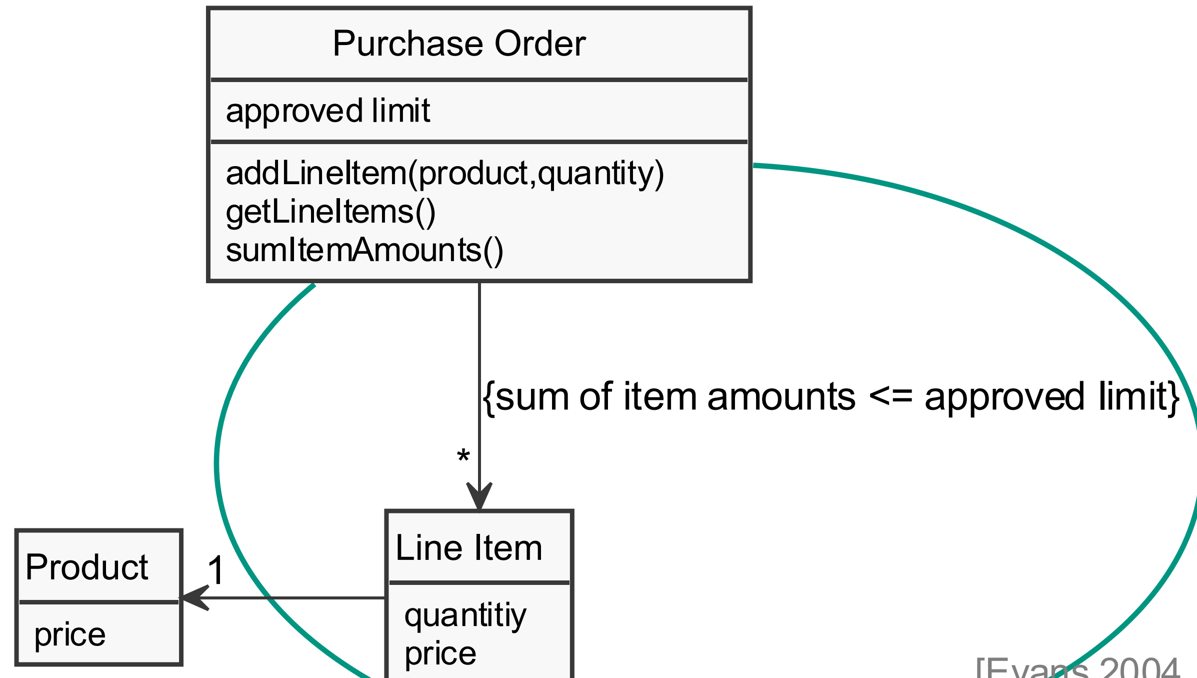
[Evans 2004, Fig. 6.4]

Example: Aggregates

Aggregates in Webshop application

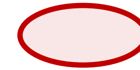
- *Purchase Order, Shopping Cart, Accountency…*
- Boundary delineates elements belonging to aggregate
- Access to elements only via root entity
- Sometimes requires introducing data redundancies

Resulting purchase order aggregate



[Evans 2004, Fig. 6.4]

Factories



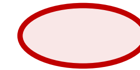
"When creation of an object, or an entire aggregates becomes complicated or reveals too much of the internal structure, factories provide encapsulation."

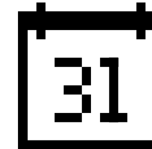
– [Evans 2004, pp. 136]

Factory provides means to create domain concept

- that hides complexity of object construction
  - creation of aggregates / related objects in atomic operation
  - ensures that products' invariant holds
- that separates construction concern
  - extracted into separate domain concept
  - only factory depends on aggregate's internal elements

Factories for Entities or Values





Factories for Entities

- creates minimal valid entities or aggregates
- ensure invariants upon creation

Factories for Value Objects

- product constructed as a whole (due to immutability)
- requires complete Specification of value object

Factory not applicable if:

- class is standalone and not used polymorphically
- client cares about implementation (e.g., picking a Strategy)
- construction is simple *(easy to ensure all invariants)*

Example: Factories

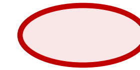
Factories in webshop application

- *Customer Factory, Shipment Factory…*
- Implemented as `@Stateless @Service` in Java Spring
- `RegistrationForm` serves as specification for construction

```
@Service  //makes CustomerFactory via service discovery
@Stateless
public class CustomerFactory {
  public Customer createCustomer(RegistrationForm form) {
    var password = UnencryptedPassword.of(form.getPassword());
    //create User Account
    var userAccount = createUserAccount(form.getName(),
                                          password,
                                          CUSTOMER_ROLE);
     //create customer linked to userAccount ensuring one-to-one relation
     return new Customer(userAccount, form.getAddress()));
  }
  private UserAccount createUserAccount(String name,
          UnencryptedPassword password, Role role){ /*…*/ }

}
```

Repositories



“[…] we must have a starting point for a traversal to an Entity or Value in the middle of its life cycle.“

– [Evans 2004, pp. 147]

Repository permits finding domain concept

- simulate collection of entities or aggregates
  - allows for accessing all, subsets or individual items
  - provides implementation for complex queries
- usually backed by persistence mechanism
  - database permits efficient querying
  - handles reconstitution of entities and aggregates
  - exposes simple method to store entities/aggregates, If any

Example: Repositories

Repositories in Webshop application

- *Catalog, Order Repository, Accountancy Entry Repository …*
- Implemented via @Repository annotation in Java Spring
- Provides operations to find and select `Products`

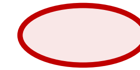
```
@Repository //Declares Catalog as repository injecting database layer
public interface Catalog {
  @Query(
    "select p from catalog p where :category member of p.categories")
  Streamable<Product> findByCategory(String category);

  @Query(
    "select p from catalog p join p.categories c where c in :categories")
  Streamable<Product> findByAnyCategory(Collection<String> categories);

  @Query("…")
  Streamable<Product> findByAllCategories(Collection<String> categories);

  //automatically generated Query
  Streamable<Product> findByName(String name);
}
```

Modules



"The Modules in the domain layer should emerge as a meaningful part of the model, telling the story of the domain on a larger scale."

– [Evans 2004, pp. 109]

Modules divides concepts in a domain model

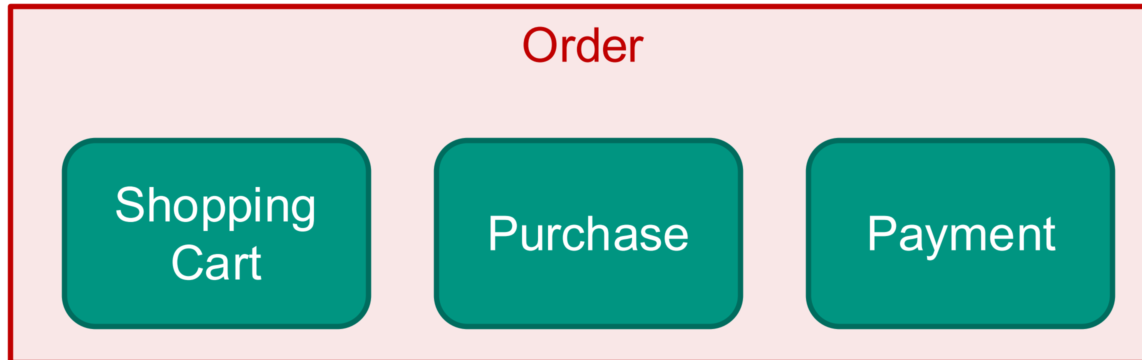
- low coupling to other modules
  - limit concepts to those required for understanding
  - Separate concepts independent of model
- high cohesion within them
  - combine closely related/dependent concepts
  - contain cohesive set of concepts
- denotes a high-level domain concept

Example: Modules

Karlsruhe Institute of Technology

Modules in webshop application

- Bounded Contexts: *Catalog*, *<u>Orders</u>*, *User Registration*, …



- Most object-oriented languages only support *packages*
- Sometimes frameworks impose predefined package structure
  - Added technical concepts unnecessarily complicate domain model
- Modules and packages must evolve with domain model
  - Effort for refactoring of package names often neglected
  - Modules become cluttered with concepts and coupled

Example: Conference Management

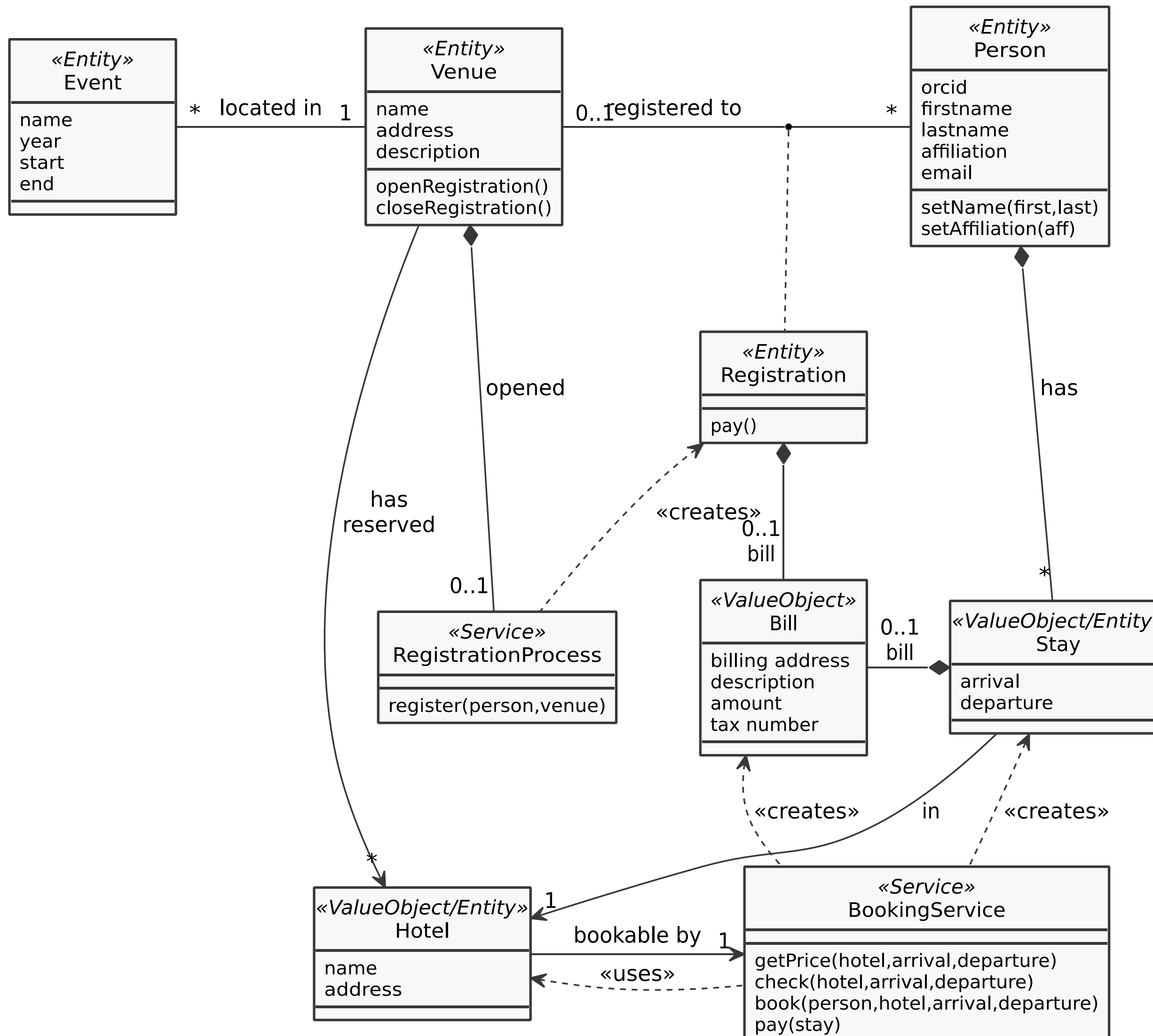
Event: name, year, start, end
Event * located in 1 Venue
Venue: name, address, description; openRegistration(), closeRegistration()
Venue 0..1 registered to * Person
Person: orcid, firstname, lastname, affiliation, email; setName(first,last), setAffiliation(aff)
Registration: pay()
Venue opened 0..1 RegistrationProcess
RegistrationProcess: register(person,venue)
RegistrationProcess «creates» Registration
Registration 0..1 bill Bill
Bill: billing address, description, amount, tax number
Person has * Stay
Stay: arrival, departure
Stay 0..1 bill Bill
Venue has reserved * Hotel
Stay in 1 Hotel
Hotel: name, address
Hotel bookable by 1 BookingService
BookingService «uses» Hotel
BookingService «creates» Bill
BookingService «creates» Stay
BookingService: check(hotel,arrival,departure), book(person,hotel,arrival,departure), pay(stay)

Bounded Context *Conference Registration*

- Persons register to event at venue
- Book hotel
- Pay bills

Classify each class as either *Entity*, *Value Object* or *Service*?

Example: Conference Management



Bounded Context
Conference Registration

- Persons register to event at venue
- Book hotel
- Pay bills

Summary: Building Blocks

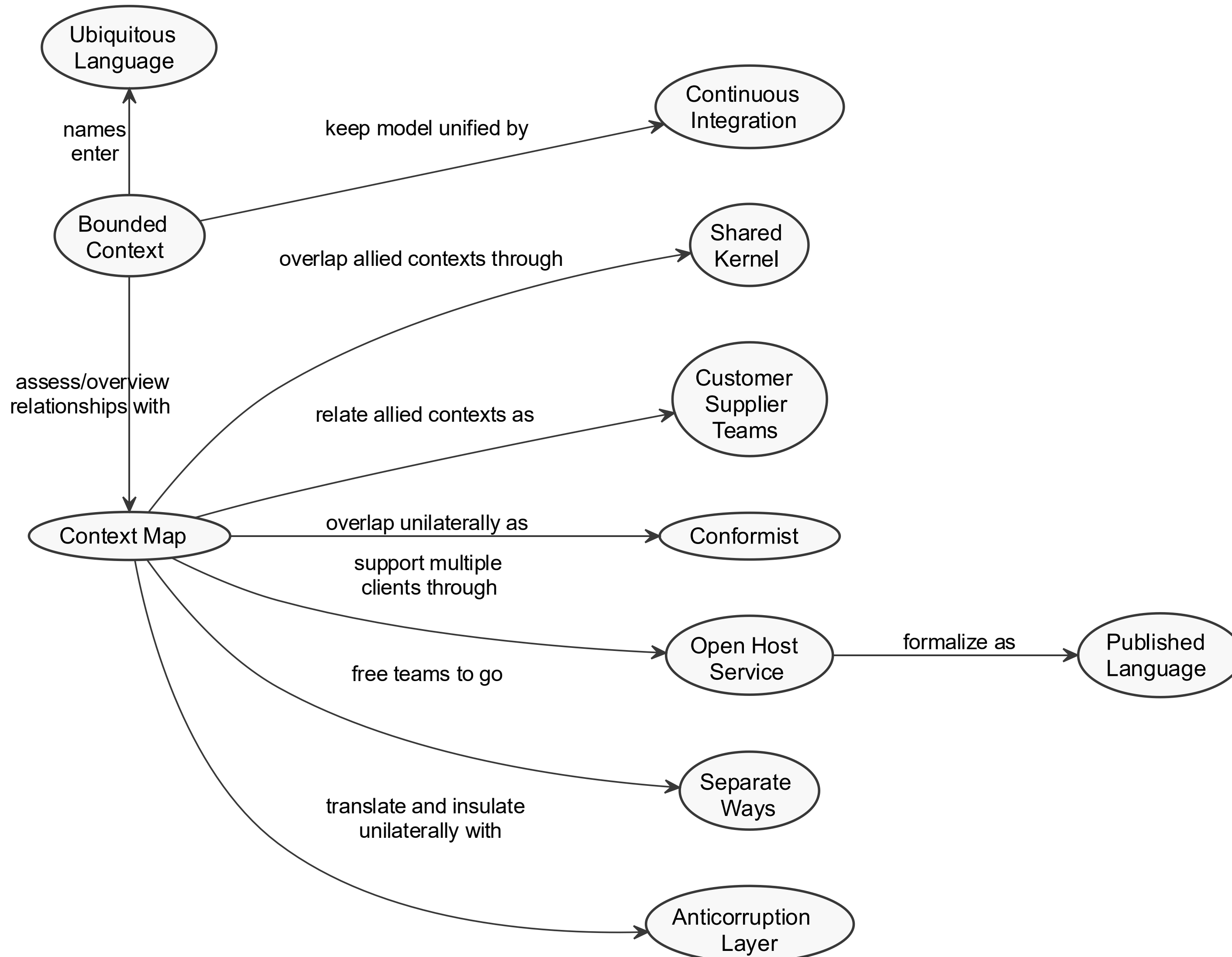
Karlsruhe Institute of Technology

- **Entities** traceable concepts with unique identity
- **Value Objects** attributes or properties of concepts
- **Services** complex domain operation/activities
- **Aggregates** wholistic representation of object cluster
- **Factories** produce complex entities, aggregates or value objects
- **Repositories** collects & queries for entities/aggregates (stores & reconstitutes objects)
- **Modules** organizes highly coupled concepts separates unrelated concepts

Karlsruhe Institute of Technology

STRATEGIC DESIGN

Overview on Strategic Design



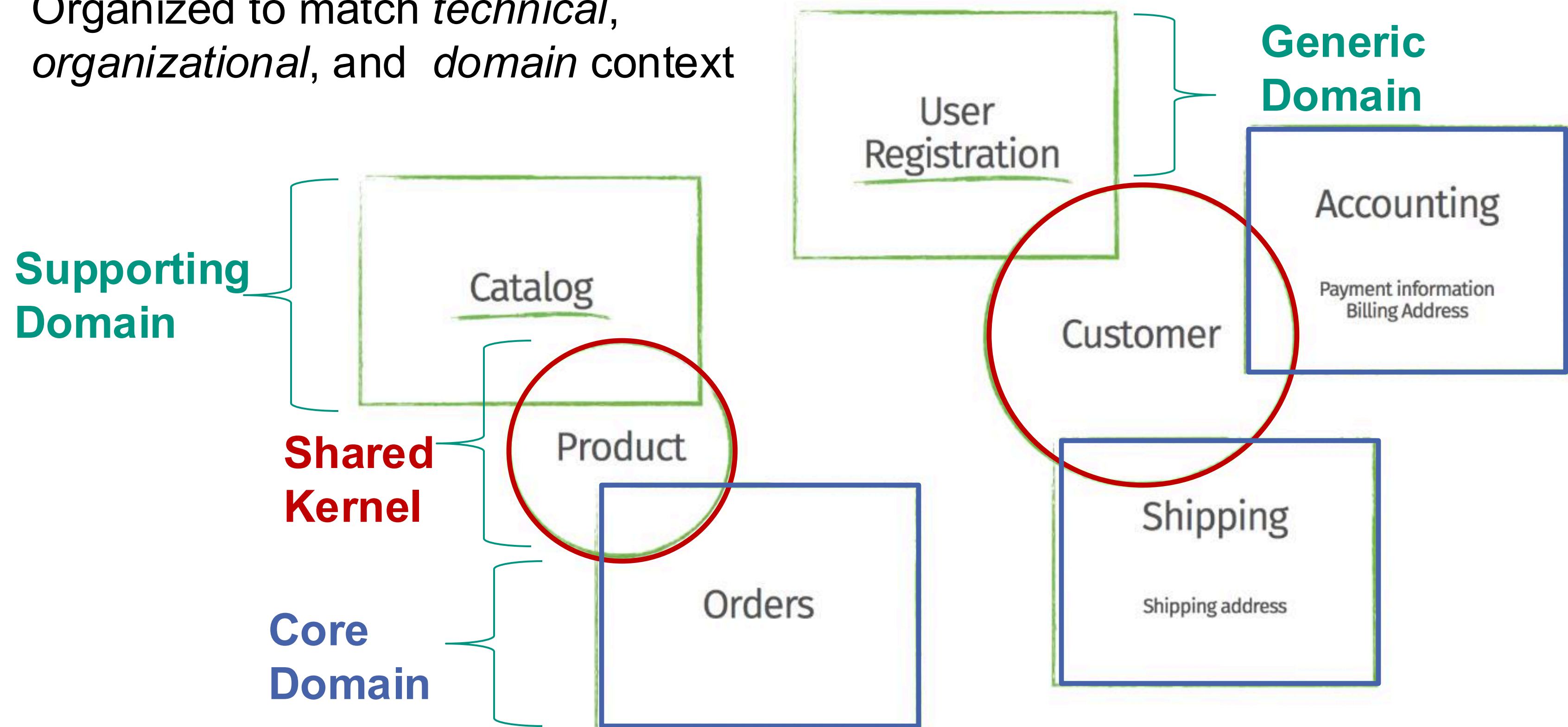
Domains and Bounded Context

- Separate application domains regarding their contribution to solving domain-related problem
- Typically, one bounded context per application domain

[Oliver Gierke, CC BY-NC-SA 4.0 http://static.olivergierke.de/lectures/ddd-and-spring/]

Context Map

- Specifies top-level perspective on all bounded contexts
- Declares communication and shared concepts between bounded contexts
  - through published interface, translation layers or anti-corruption layers
- Organized to match *technical*, *organizational*, and *domain* context



[Oliver Gierke, CC BY-NC-SA 4.0 http://static.olivergierke.de/lectures/ddd-and-spring/]

Different kinds of Domains

Karlsruhe Institute of Technology

Core Domain [Evans 2004, pp. 400]

- crucial aspect of domain model to solve domain-related problem
- all other domains support core domain

Shared Kernel [Evans 2004, pp. 354]

- domain concepts shared between different domains
- maintained and evolved by multiple teams
- requires effort for coordination and integration

Supporting Domain

- encompasses concepts extracted from core domain
- domain concepts playing a support role for core domain

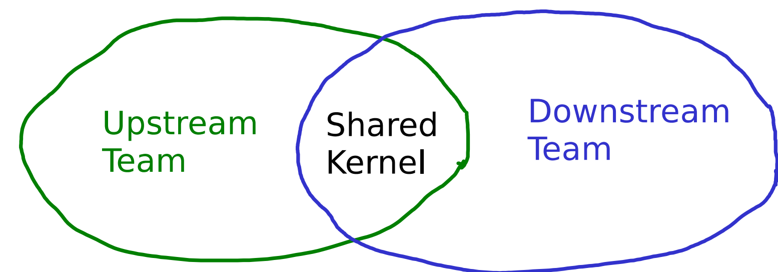
Generic Subdomain [Evans 2004, pp. 406]

- encapsulates most generic concepts not belonging to core domain
- could be outsourced or replaced with off-the-shelf components

Interactions Between Domains

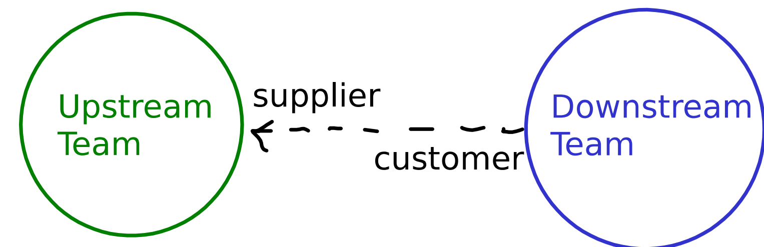
Shared Kernel
[Evans 2004, pp. 354]

- establish common domain concept as shared language



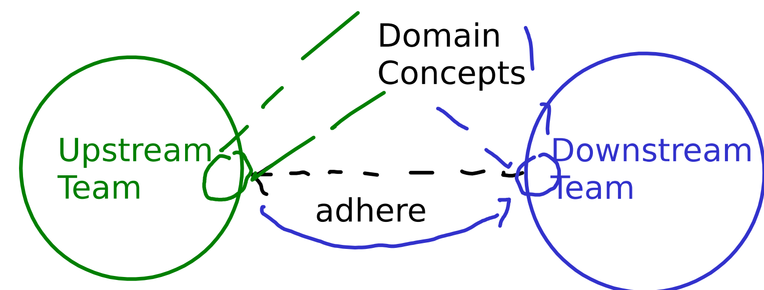
Consumer/Supplier Development Team [Evans 2004, pp. 356]

- upstream and downstream team as supplier and consumer, respectively
- downstream requests functionality from upstream team
- usually only works, if both teams have common management above, motivating the upstream team



Conformist [Evans 2004, pp. 361]

- downstream "slavishly" adheres to domain model of upstream team
- favor conformity over development freedom of downstream team

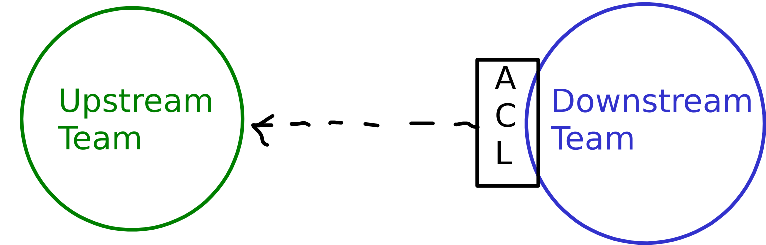


Interactions Between Domains

Karlsruhe Institute of Technology

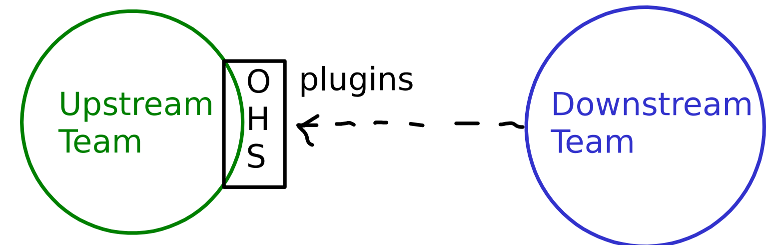
Anticorruption Layer
[Evans 2004, pp. 364]

- downstream sets up layer translating upstream's domain model
- implements interface by upstream teams required interface



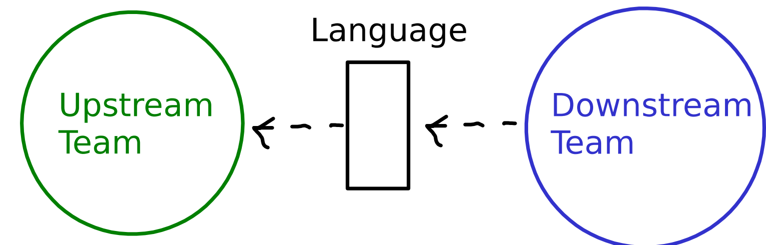
Open Host Service
[Evans 2004, pp. 374]

- define protocol to allow (thirdparty) downstream services to use services of upstream team

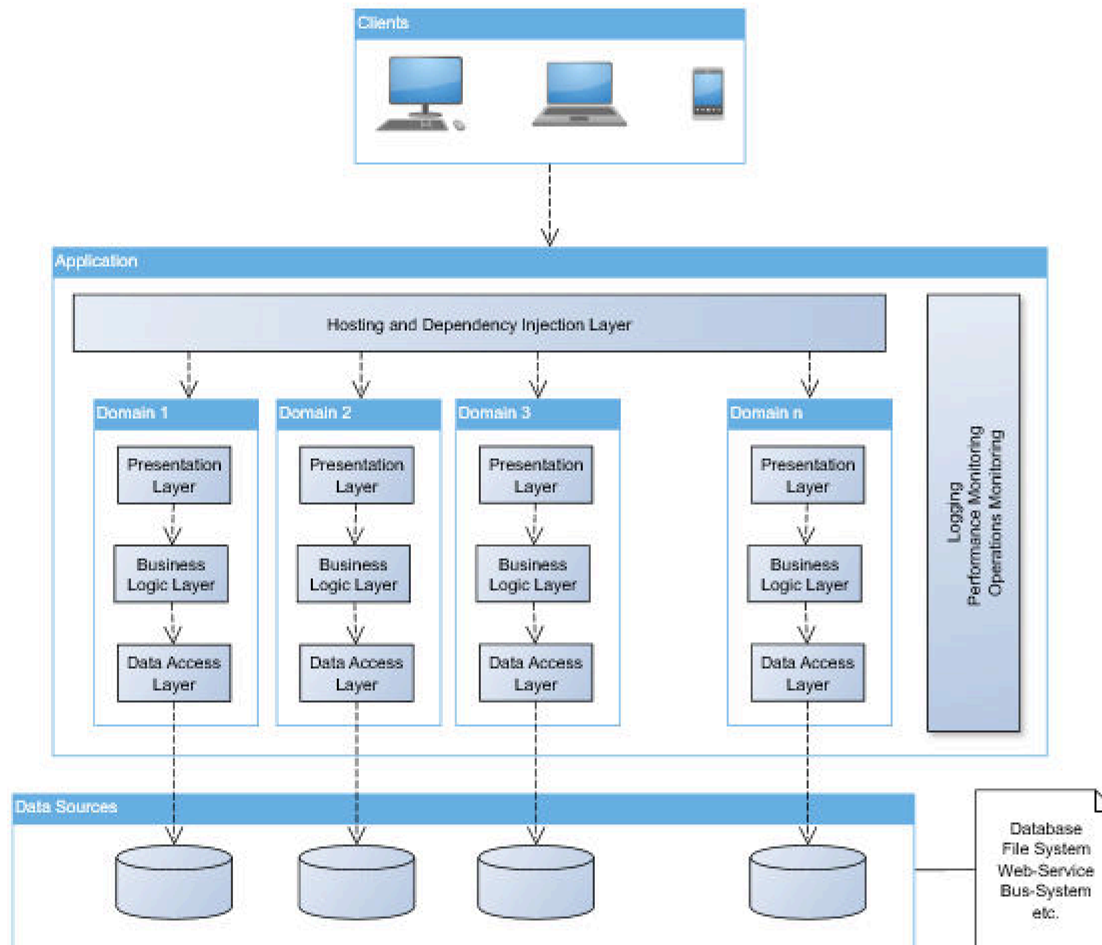


Published Language
[Evans 2004, pp. 375]

- formalize protocol to domain-specific language



Layered Architecture and Microservices



Domain application

- implemented with layered architecture
- are separately deployed
- linked by appropriate framework via
  - Dependency Injection
  - Service Registry
  - Connectors

[Schichtenarchitektur by MovGP0 (Copyrighted free use) https://commons.wikimedia.org/w/index.php?curid=58203255]

Summary: Strategic Design

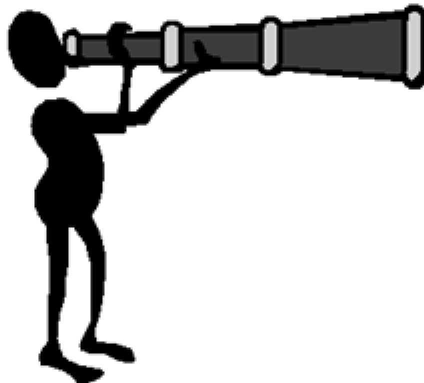
- Identify bounded contexts
- Declare context map and interactions
- Separately implement domain as domain application/microservice
- Deploy and link domain applications with interaction patterns

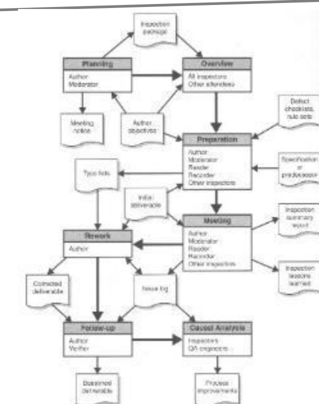
[https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7]

Conclusion

What have we learned?

- Understood ubiquitous language and domain model underlying Domain-Driven Design
- Learned building blocks and their application
- Discussed design process and strategic decisions



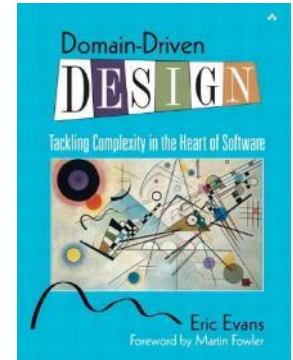


Reviews

Picture from Karl Wiegers

Literature

- [Evans 2004] Evans, Eric. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
- [Marinescu 2007] Marinescu, Floyd, and Abel Avram. *Domain-driven design Quickly*. Lulu.com, 2007.
- [Gierke 2019] Gierke, Oliver. *Domain-Driven Design and Spring. Olivergierke.de,* 2019. (CC BY-NC-SA 4.0) http://static.olivergierke.de/lectures/ddd-and-spring/



Online Resources

- Federated wiki with extracts of [Evans 2004] http://ddd.fed.wiki.org/view/welcome-visitors
- Evans, Eric. *Tackling Complexity in the Heart of Software* Domain-Driven Design Europe 2016. https://www.youtube.com/watch?v=dnUFEg68ESM
- Webber, Kevin. *Modelling Reactive Systems with Event Storming and Domain-Driven Design.* RedElastic, 2017. https://blog.redelastic.com/corporate-arts-crafts-modelling-with-event-storming-73c6236f5dd7