HOCHSCHULE HANNOVER
UNIVERSITY OF APPLIED SCIENCES AND ARTS
Fakultät IV Wirtschaft und Informatik

Service Computation 2021
April 18 – 22, 2021 – Porto, Portugal

Keynote Speech on

Model-driven Microservice Development

— Applying Domain Driven Design (DDD) to Microservices Architecture —

Andreas Hausotter
Faculty of Business and Computer Science
University of Applied Sciences and Arts Hannover
Ricklinger Stadtweg 120 30459 Hannover
andreas.hausotter@hs-hannover.de

CC_ITM Competence Center Information Technology and Management Institute at the University of Applied Sciences and Arts Hannover | cc_itm@hs-hannover.de

CC_ITM Competence Center Information Technology and Management Institute at the University of Applied Sciences and Arts Hannover | cc_itm@hs-hannover.de

HOCHSCHULE HANNOVER
UNIVERSITY OF APPLIED SCIENCES AND ARTS
Fakultät IV Wirtschaft und Informatik

Keynote Speaker

Dr. ANDREAS HAUSOTTER is a professor emeritus for distributed information systems and database systems at the University for Applied Sciences and Arts, Hanover, Germany, Faculty of Business and Computer Science. His area of specialization comprises service computing – including service-oriented Architectures (SOA) and microservices – Java EE, webservices, distributed information systems, business process management, business rules management, and information modeling.

In 1979 he received his PhD in mathematics at Kiel University, Faculty of Mathematics and Natural Sciences. After graduation he started his career with KRUPP ATLAS ELEKTRONIK, Bremen, as a systems analyst and systems programmer in the area of real time processing. In 1984 he was hired as systems engineer and group manager SNA Communications for NIXDORF COMPUTER, Paderborn. After that, he worked for HAAS CONSULT, Hanover, as a systems engineer and product manager for traffic guidance systems.

In 1996 he was appointed professor of operating systems, networking and database systems at the University of Applied Sciences and Arts, Hanover. He has been retired since March 2018.

From the beginning he was involved in several research projects in cooperation with industry partners. During his research semester he developed a Java EE / EJB application framework. Based on this framework a web-based simulation software for securities trading was implemented by his research group to train the apprentices of the industry partner.

In 2005, the Competence Center IT & Management (CC_ITM) was founded in cooperation with industry partners. Different ambitious research projects have since then been carried out in the context of service-computing, microservices, cloud computing, business process management, and business rules management.

Competence Center Information Technology and Management Institute at the University of Applied Sciences and Arts Hannover | cc_itm@hs-hannover.de

HOCHSCHULE HANNOVER UNIVERSITY OF APPLIED SCIENCES AND ARTS

Fakultät IV Wirtschaft und Informatik

CC_ITM @ HsH

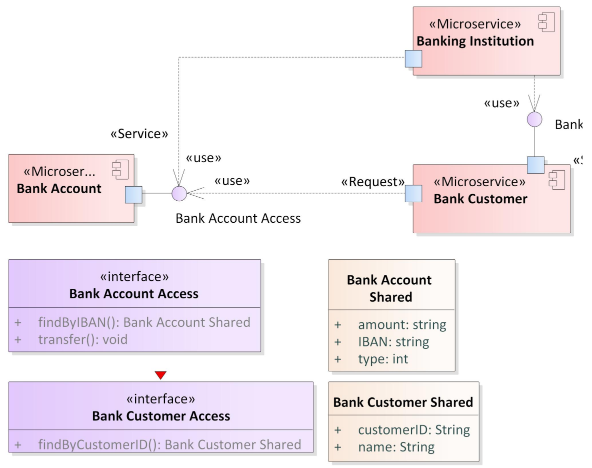
- **Competence Center Information Technology & Management (CC_ITM)**
  - Institute at the University of Applied Sciences and Arts, Hannover
  - Founded in 2005 by colleagues from the departments of **Business Information Systems and Computer Science**
  - Members: **Faculty staff, industry partners** (practitioners) of different areas of businesses
- Main objective
  - **Knowledge transfer** between university and industry
- Research topics
  - Management of information processing
  - Service computing, including Microservices, Service-oriented Architectures (SOA), Business Process Management (BPM), Business Rules Management (BRM)
  - Cloud Computing

Outline

Introduction

What is Model-Driven Development? (1/2)

- **Models** [in general]
  - Abstractions of ‘real’ world problems
  - Ignore irrelevant details, focus on the relevant ones
  - Help us to understand a complex problem and its potential solutions
  - Different concepts / notations are used to highlight various perspectives or views of a system
  - Examples of modeling notations: UML (Use Case, Class, Component, Activ-ity), Code (Java, ...)



- (Model) **Transformation**
  - Conversion of one model in another model with distinct perspectives and levels of abstraction
  - Examples: Analysis Model → Design Model, Design Model → Code

Introduction

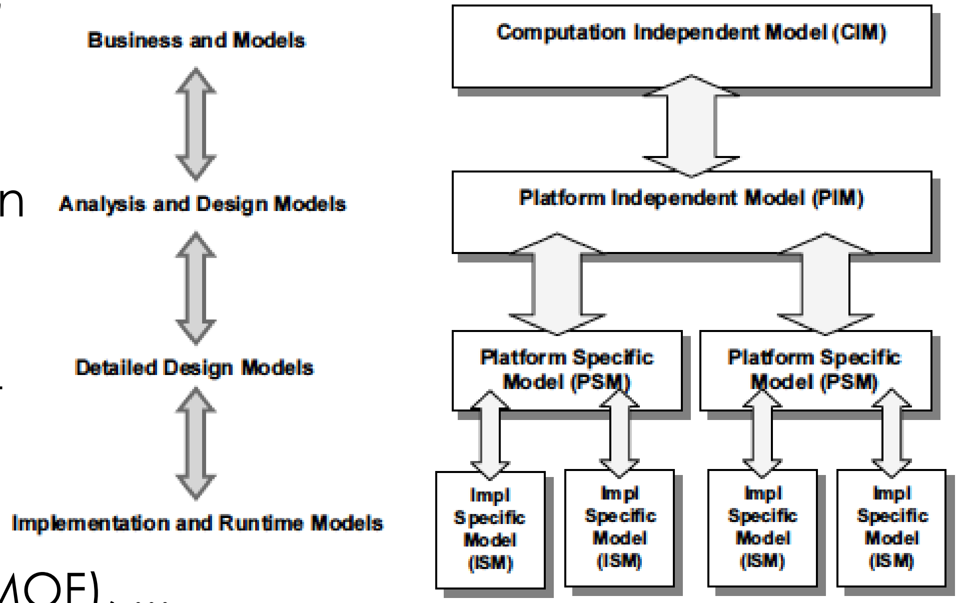
What is Model-Driven Development? (2/2)

- **Model Driven Development (MDD)**
  - Category of software development approaches, based on models, model-ing and model transformation
  - Models are successively transformed into more specific and detailed ones by applying transformation rules.
- Benefits of the MDD approach
  - Better understanding of the problem space
  - Models provide a unique lingua franca between software engineer.
  - Transformation process may be supported by software tools.
- Challenges of the MDD approach
  - Software quality and implementation effort strongly depend on the model quality which is influenced by characteristics like abstraction, understand-ability, accuracy.

Introduction

Examples of Model Driven Development

- **Model Driven Architecture (MDA) - OMG**
  - MDD approach, based on a set of standards *how* to define a set of models, notations, and transformation rules
  - Objective: Automatically generate platform-specific code from abstract models
  - Standards: Unified Modeling Language (UML), Meta Object Facility (MOF), ...
- **Domain Driven Design – Evans, 2004**
  - Model-driven design approach, focuses on the application domain as primary drivers for architecture design
  - Collaborative modeling of domain experts and software engineers.

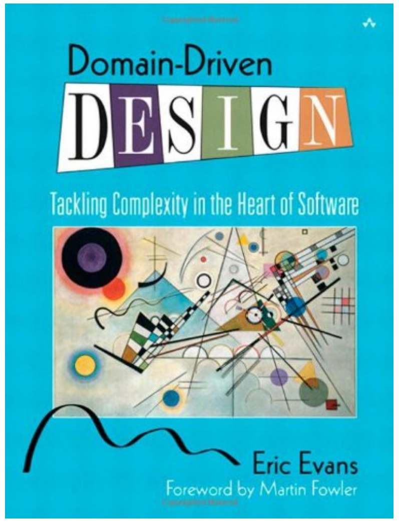

Business and Models
Analysis and Design Models
Detailed Design Models
Implementation and Runtime Models
Computation Independent Model (CIM)
Platform Independent Model (PIM)
Platform Specific Model (PSM)
Platform Specific Model (PSM)
Impl Specific Model (ISM)
Impl Specific Model (ISM)
Impl Specific Model (ISM)
Impl Specific Model (ISM)

Layers and Transformations of MDA. Source: [BeBoEd98]

Introduction

What is Domain Driven Design?

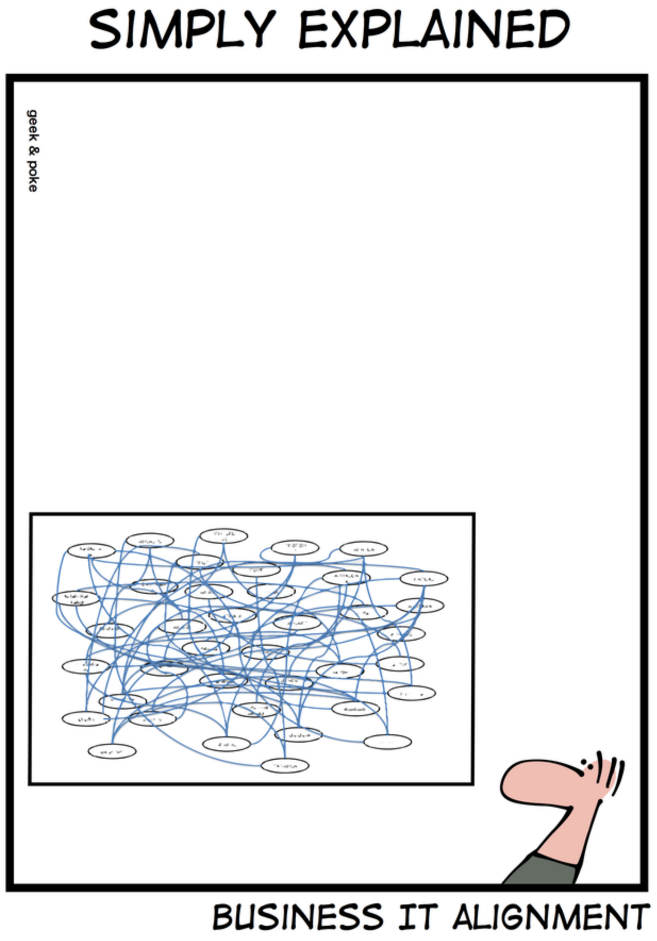
- **Domain Driven Design (DDD)**
  - Approach to software development (Eric Evans, 2004 **))
  - Focuses on the application domain, its concepts and their relationships as primary drivers for architecture design.
- Core principles of DDD
  - Capturing relevant domain knowledge in →Domain Models
  - Collaborative modeling of domain experts and software engineers
  - Aligning model and implementation and continuous model refactoring
  - Promoting communication between domain experts and software engineering by jointly defining a shared →Ubiquitous Language.


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Introduction

DDD Goals

- Mastering complexity
  - Make large software products with complex business logic manageable
  - Build correct, understandable and maintain-able software systems within time and bud-get.
- Bridging the gap between user and software engineers
  - Try to understand what the software is ex-pected to do
  - Meet the customer's needs and expecta-tions.



Source: Geek&Poke, http://geek-and-po-ke.com/geekandpoke

Basic Conceps

The DDD approach

- Domain Driven Design is an approach to develop complex software systems, applying the following principles:
  - Focus on the →Core Domain.
  - Explore models in a creative collaboration of domain experts and software engineers.
  - Speak a →Ubiquitous Language within an explicitly →Bounded Context.
- Main effort in DDD focuses on ...
  - … trying to understand what the user wants to do
  - … modeling his (domain) knowledge

Basic Conceps

Context and Bounded Context

- **Context**
  - The area in which a word or statement appears, thereby determining its meaning.
  - Statements about a model can only be understood in a context.
- **Bounded Context**
  - Description of a boundary within which a particular model is defined and applicable
  - Typically a software subsystem or the work of a particular team.
- In DDD for Microservice Architectures, every Bounded Context is mapped to one microservice.

Source: Geek&Poke, http://geek-and-poke.com/geekandpoke

Basic Conceps

What does ‘Domain’ mean?

- **Domain**
  - Sphere of knowledge, influence, or activity
  - Subject area, to which a user applies is the domain of the software
  - The **Core Domain** comprises the most valuable concepts of the domain.
- Example ‘Online Banking’
  - Core Domain comprises the use cases: Login/Logout, Check Account Bal-ance, Make a Funds Transfer, Display Turnovers
  - Use cases, not subject of the Core Domain: Send a Message to the Banking Institution, Make an Appointment with a Bank Consultant, ...

Basic Conceps

Using a Common Language (1/2)

- **Ubiquitious Language**
  - Language, shared between all participants, i.e. Users / Domain Experts, Software Engineers
  - Understood by all participants
  - Uses business-oriented terms, not technical-oriented terms
- Strictly adhere to the Ubiquitous Language in ...
  - … discussions and communications
  - … documents
  - … the models, code (e.g. class names, attribute names, …).
- A glossary should comprise all terms used.

Basic Conceps

Using a Common Language (2/2)

- The Domain Expert must double-check each term defined
  - Do I understand the term?
  - Does the definition express what I think?
  - Can I unambiguously express my problem with it?
- The Developer must double-check each term defined
  - Is the term unambiguous, consistent, well-defined, …?
  - Is it possible to write code for it?

Basic Conceps

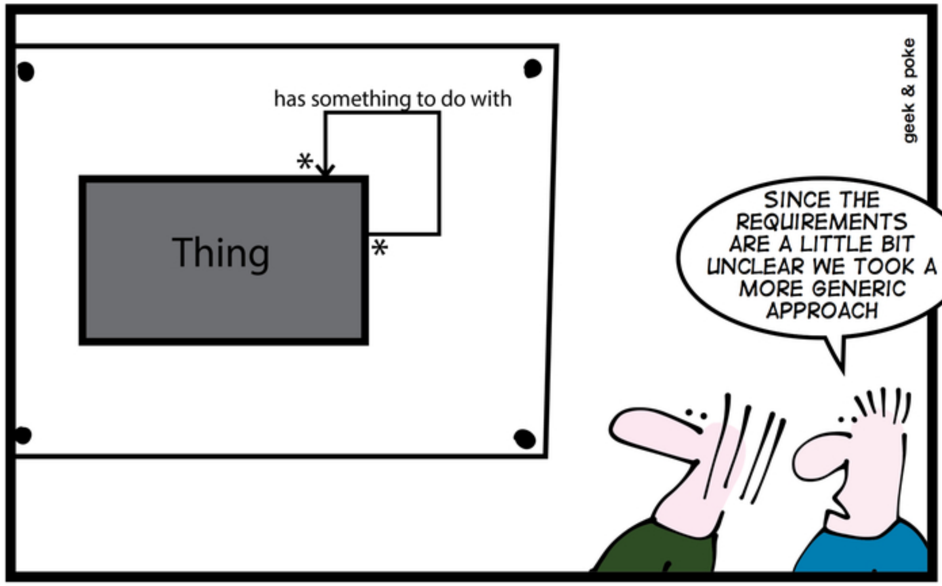
Creating a Domain Model (1/2)

- **Domain Model**
  - System of abstractions
  - Describes selected and relevant aspects of a domain
  - Can be used to solve problems related to that domain.
- Requirements for Domain Models
  - Created in close collaboration between Domain Experts and Developers
  - Not internal to the developers
  - Avoid technical terms and concepts
  - Must be readable and understandable for Developers as well as Domain Experts.

Basic Conceps

Creating a Domain Model (2/2)

- Domain Models are preferably represented by UML Class diagrams (Evans)
  - Named classes with attributes and methods
  - Associations, usually non-navi-gable
  - Multiplicities
  - Constraints, e.g. in the form of notes
- Use any other suitable notation
  - Plain text
  - Free-hand drawings
  - Documented code, e.g. JavaDoc


has something to do with
*
*
Thing
geek & poke
SINCE THE REQUIREMENTS ARE A LITTLE BIT UNCLEAR WE TOOK A MORE GENERIC APPROACH

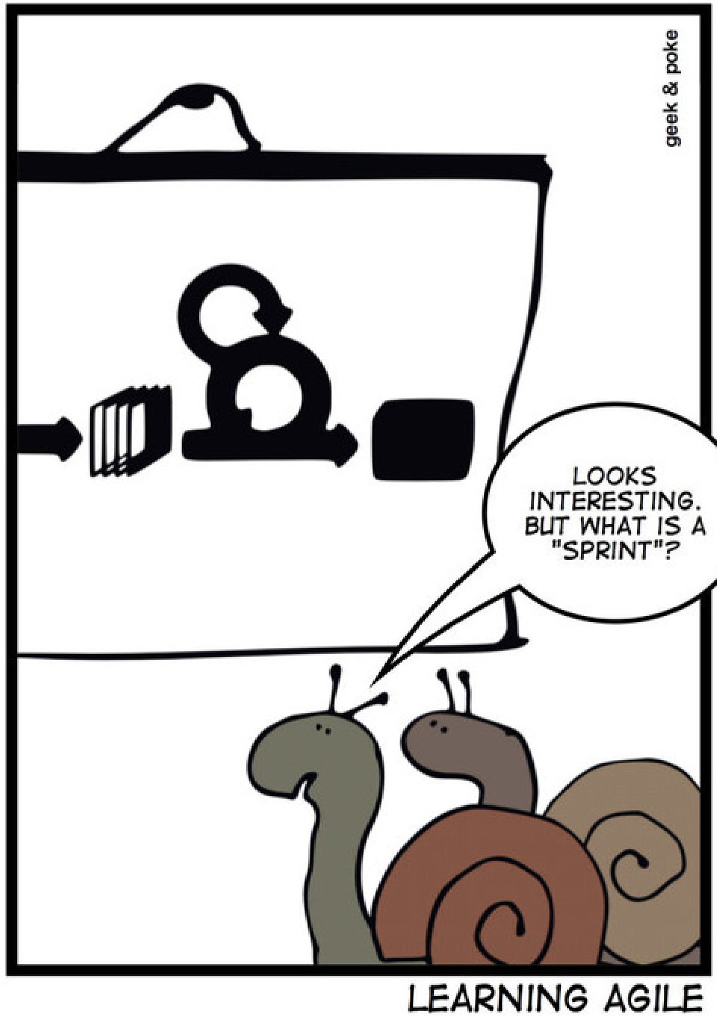
HOW TO CREATE A STABLE DATA MODEL

Source: Geek&Poke, http://geek-and-poke.com/geekandpoke

Basic Conceps

DDD and Agile Software Project Management

- Domain Driven Design
  - is NOT a software project management methodology
  - But requires some agile software development principles
  - Goes well with Scrum, XP, ...
  - Won't work with 'classical' project management methodologies like Waterfall, Spiral, ...



Source: Geek&Poke, http://geek-and-poke.com/geekandpoke

Basic Conceps

DDD and Agile Principles

- Interaction
  - Direct and frequent discussion ...
  - … between Domain Experts and Software Engineers
  - … during the project’s lifecycle
- Iteration
  - Language and domain model evolve during design and implementation

    … Unclear semantics, missing concepts?
    … Implementation or performance problems?
    … Technical or functional refactoring required?
  - Discuss all problems above with the Domain Experts
  - Requires →Continuous Integration

Basic Conceps

DDD and Agile Principles

- **Continuous Integration**
  - Merge, build, and test (unit, integration, ...)
  - Test automation is highly recommended.
- Overall Goal
  - At any time, Ubiquitous Language, Domain Model, Model-driven Design and Code must match each other.

Building Blocks of a Model Driven Design

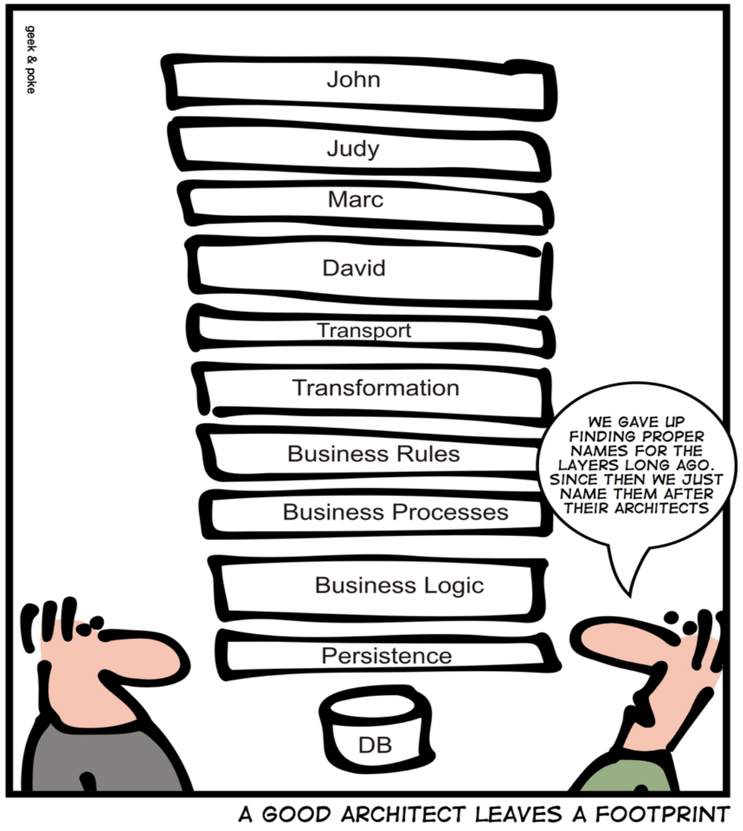
DDD Patterns

- Domain Driven Design specifies a variety of **DDD Patterns** to refine the structural Domain Model for Model-driven Design
  - Layered Architecture: Software structure
  - Entity: Object with identity
  - Value Object: Values without an identity
  - Aggregate: Combines an Entity with other Entities and / or Value Objects
  - Factory: Generate Aggregates
  - Repository: Stores Entities and Aggregates persistently
  - Service: Functionality which is not assignable to a single Entity
  - Module: Structures the model

Building Blocks of a Model Driven Design

Layered Architecture – Software architecture

- **Layered Architecture**
  - Apply the Design Pattern 'Layers' to isolate the Domain Model from the infrastructure, the user interface and even the application logic.
- DDD Layers
  - **Presentation Layer** – User Interface
  - **Application Layer** – Coordination and client session management
  - **Domain Layer** – Business data and business logic
  - **Infrastructure Layer** – Persistence, Communi-cation ...


geek & poke
John
Judy
Marc
David
Transport
Transformation
Business Rules
Business Processes
Business Logic
Persistence
DB
WE GAVE UP FINDING PROPER NAMES FOR THE LAYERS LONG AGO. SINCE THEN WE JUST NAME THEM AFTER THEIR ARCHITECTS
A GOOD ARCHITECT LEAVES A FOOTPRINT

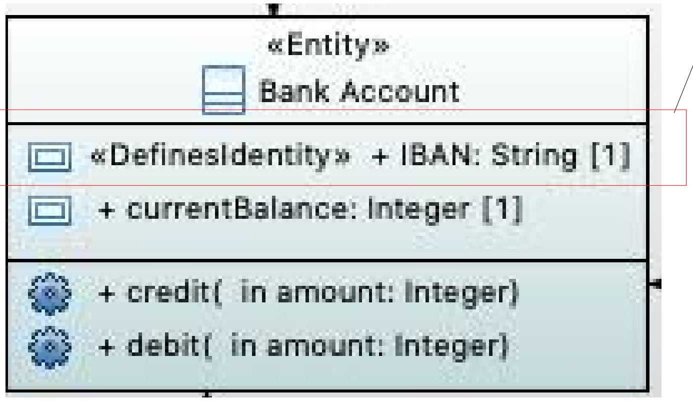
Source: Geek&Poke, http://geek-and-po-ke.com/geekandpoke

Building Blocks of a Model Driven Design

Entity – Object with identity

- **Entity**
  - Instances are distinguishable from others by a domain-specific identity
  - The instance is NOT defined by its attribute values
  - The instance has a state and a life cycle
  - Has a behavior, specified by its methods
- Examples
  - Employee
  - Contract
  - Bank Account

Using the **UML stereotypes** `<<Entity>>` and `<<DefinesIdentity>>` to express that `Bank Account` conforms to the Entity pattern.

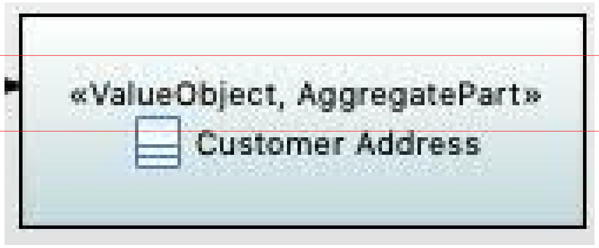

«Entity»
Bank Account
«DefinesIdentity» + IBAN: String [1]
+ currentBalance: Integer [1]
+ credit(in amount: Integer)
+ debit(in amount: Integer)

Building Blocks of a Model Driven Design

Value Object – Values without an identity

- **Value Object**
  - An instance represent values or properties.
  - An instance is defined by its attribute values.
  - An instance is immutable.
  - Has no domain-specific identity.
- Examples
  - Color
  - Temperature
  - Customer Address

Using the **UML stereotype** `<<ValueObject>>` to express that `Customer Address` conforms to the Value Object pattern.

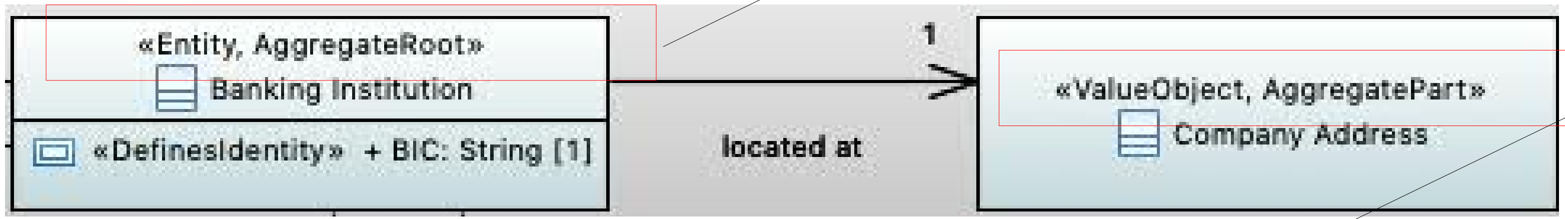


Building Blocks of a Model Driven Design

Aggregate – Combines Entities or Value Objects (1/2)

- **Aggregate**
  - Cluster of associated Entities and Value Objects
  - Members can only be accessed by referencing its →Aggregate Root
- Examples
  - Car: clusters engine + powertrain + tires + ...
  - Banking Institution + Company Address

Using the **UML stereotype** `<<AggregateRoute>>` to express that `Banking Institution` conforms to the Aggregate Root pattern.

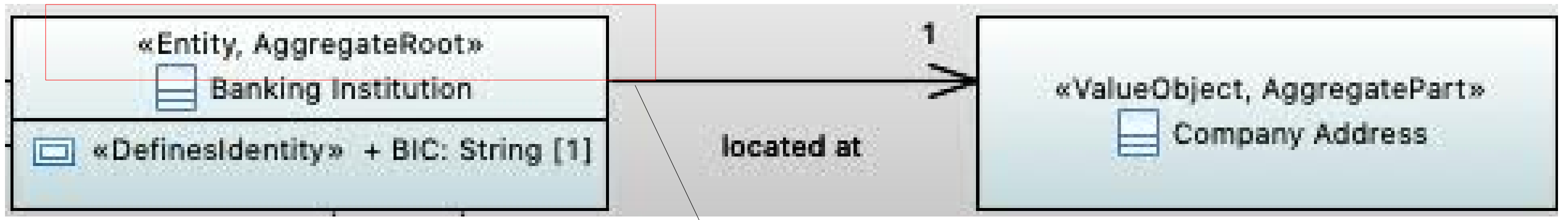


Using the **UML stereotype** `<<AggregatePart>>` to express that `Company Address` conforms to the Aggregate Part pattern.

Building Blocks of a Model Driven Design

Aggregate – Combines Entities or Value Objects (2/2)

- **Aggregate Root**
  - Topmost Entity, representing the Aggregate
  - Is the owner of all objects in the Aggregate
  - Is the object that gives the Aggregate its identity
  - Is the only object whose identity is visible from outside the Aggregate
  - Is the only object whose methods can be invoked from outside the aggre-gate



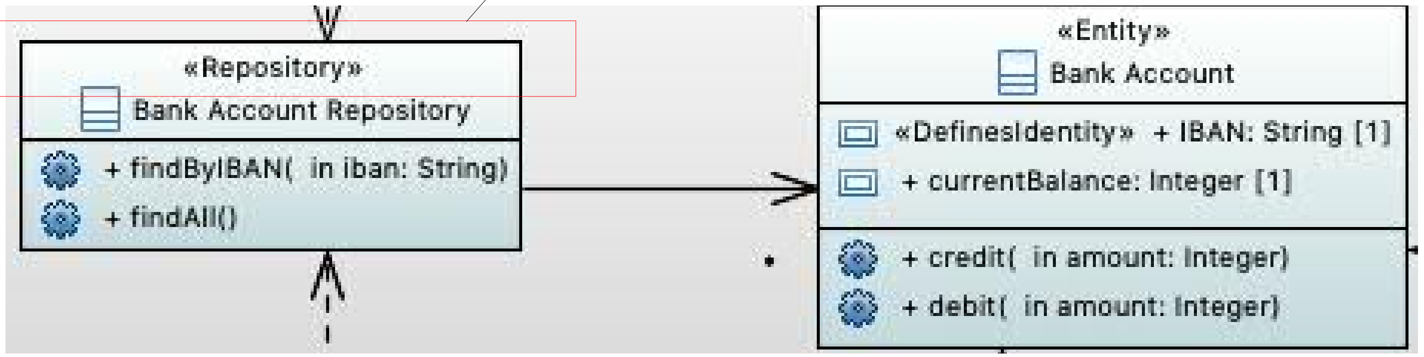
Using the **UML stereotype** `<<AggregateRoute>>` to express that `Banking Institution` conforms to the Aggregate Root pattern.

Building Blocks of a Model Driven Design

Repository – Stores Entities and Aggregates persistently

- **Repository**
  - Supports access to persistent objects
  - Provides operations that perform instance selection based on search criteria
  - Typical operations: `add, remove, find, list`
- Examples
  - Car Repository
  - Contract Repository
  - Bank Account Repository

Using the **UML stereotype** `<<Repository>>` to express that `Bank Account Repository` conforms to the Repository pattern.

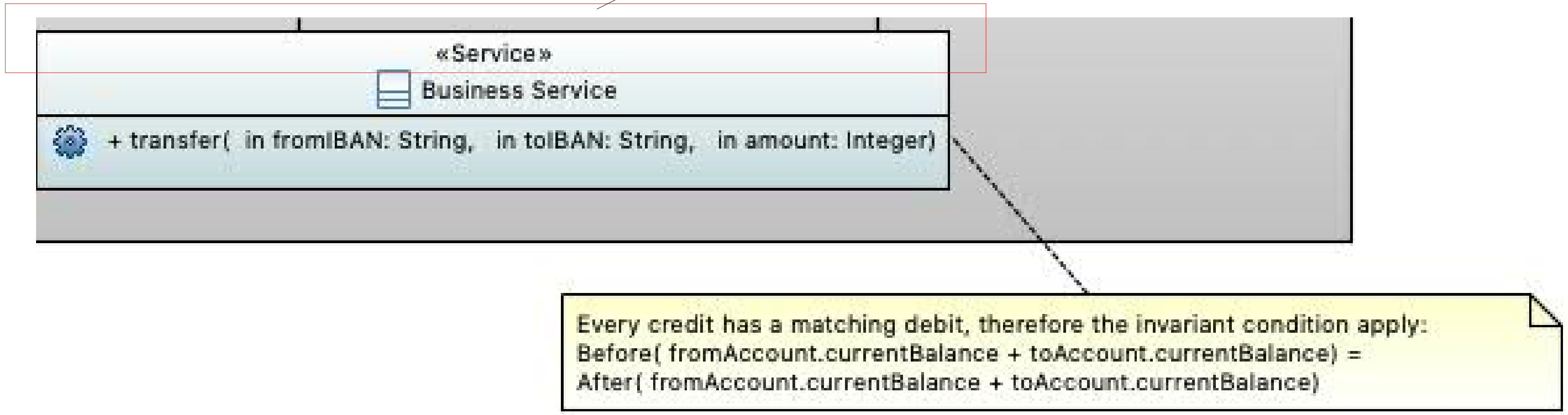

«Repository»
Bank Account Repository
+ findByIBAN(in iban: String)
+ findAll()
«Entity»
Bank Account
«DefinesIdentity» + IBAN: String [1]
+ currentBalance: Integer [1]
+ credit(in amount: Integer)
+ debit(in amount: Integer)

Building Blocks of a Model Driven Design

Service – Functionality which is not assignable to a single Entity

- **Service**
  - Provides functionality that is not in the responsibility of one Entity or Value Object resp.
  - Business processes or business rules are typically provided as Services
- Example
  - 'Funds transfer'

Using the **UML stereotype** `<<Service>>` to express that `Business Service` conforms to the Service pattern.

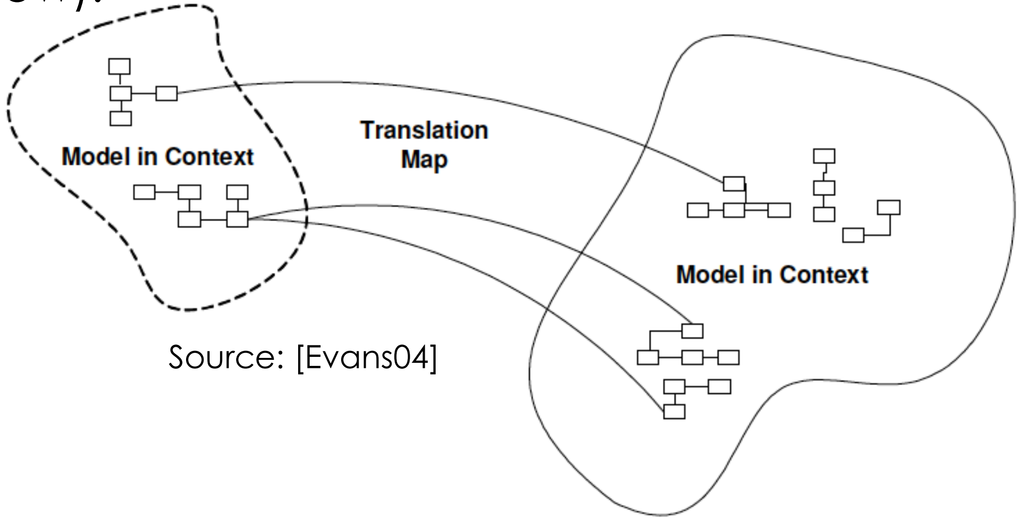

«Service»
Business Service
+ transfer(in fromIBAN: String, in toIBAN: String, in amount: Integer)

Every credit has a matching debit, therefore the invariant condition apply:
Before(fromAccount.currentBalance + toAccount.currentBalance) =
After(fromAccount.currentBalance + toAccount.currentBalance)

Strategic Design

Interaction between Bounded Contexts

- **Context Map**
  - Models the different Bounded Contexts, the ‘contact points’ and the inter-action based on patterns (see below).
- DDD defines six **Patterns of Interaction**
  - Shared Kernel, Customer / Supplier, Open Host Service, Anticorruption Layer, Conformist, and Separate Ways
- The goal of the strategic patterns is to manage the tradeoff between
  - Level of integration of functionality
  - Communication needs between the teams responsible for their Bounded Context.

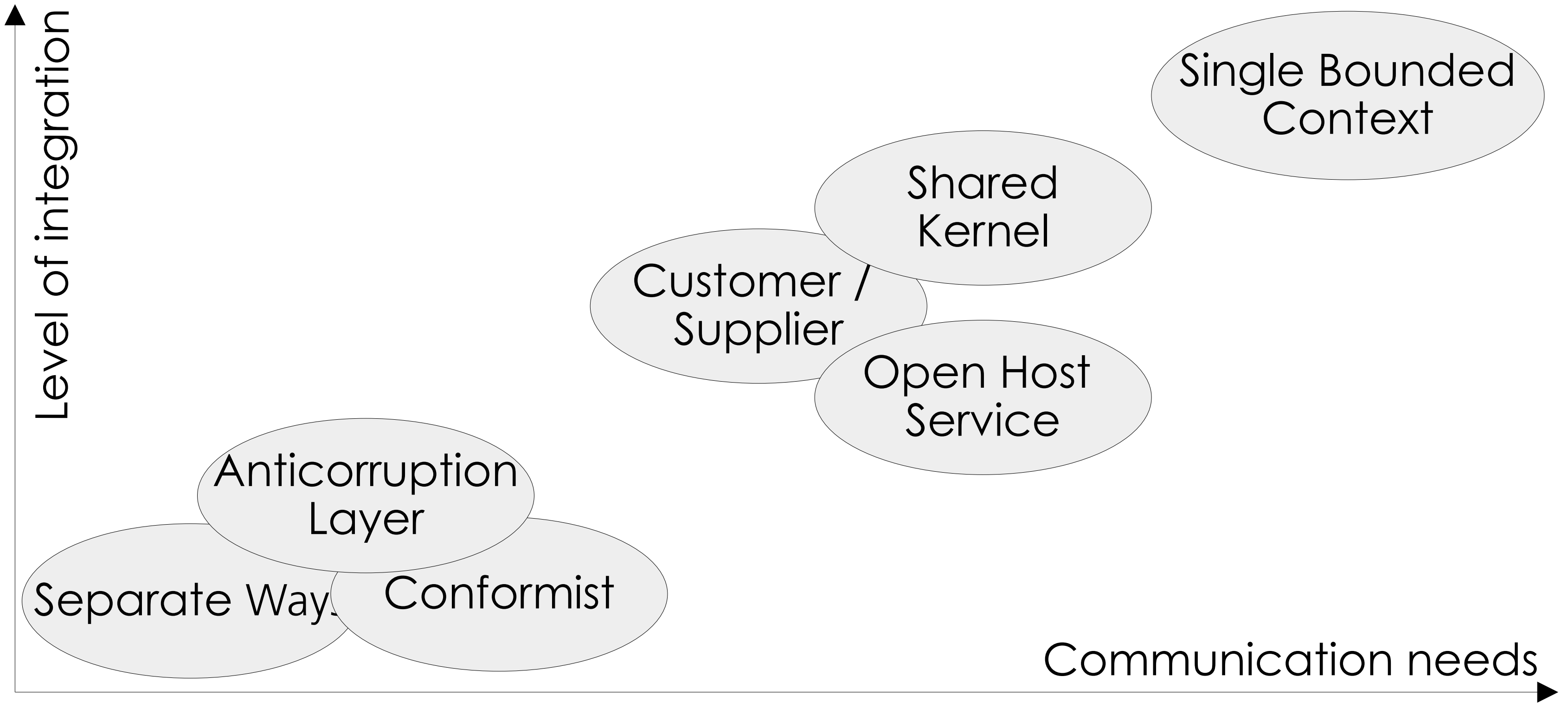


Source: [Evans04]

Strategic Design

Patterns of Interaction (1/8)

- Tradeoff between communication needs and level of integration

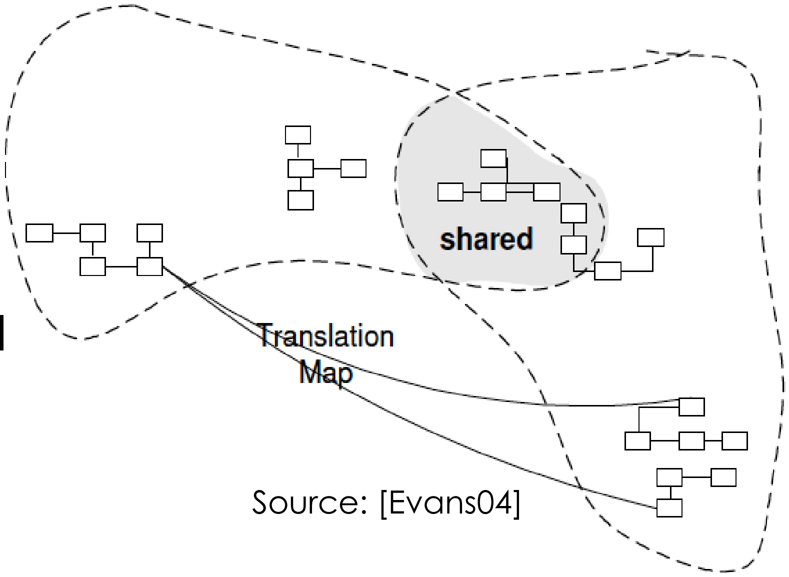


Source: Own representation, based on [Evans04]

Strategic Design

Patterns of Interaction (2/8)

- **Shared Kernel**
  - Subset of the domain model and code is shared between bounded contexts
- Motivation
  - Teams do not have the skill or organizational backing to perform continuous integration.
- Benefits
  - Reduction of duplicated code
- Challenges
  - Both teams may modify the shared kernel, so ...
  - … Teams must agree on any changes
  - … the code has to be merged and tested as soon as possible.



Source: [Evans04]

Strategic Design

Patterns of Interaction (3/8)

- **Customer / Supplier**
  - One subsystem (customer) depends on the other (supplier).
  - The supplier offers a domain model to the customer.
- Motivation
  - Customer-Supplier relationship between subsystems.
- Benefits
  - Reduction of development effort.
- Challenges
  - The customer team should present their requirements to the supplier team – the supplier team should schedule the tasks accordingly.
  - The interface between the subsystems must be carefully specified.
  - Development of acceptance tests for interface validation.

Strategic Design

Patterns of Interaction (4/8)

- **Open Host Service**
  - Many systems depend on one external system.
  - The external system acts as a provider of services – an open protocol gives access as a set of services.
- Motivation
  - An external subsystem is used by several client systems.
- Benefits
  - Prevents from providing a translation layer per client system (see pattern Customer – Supplier).
- Challenges
  - A subsystem may have special requirement to access the external system.
  - To handle new integration requirements may be complex and costly.

Strategic Design

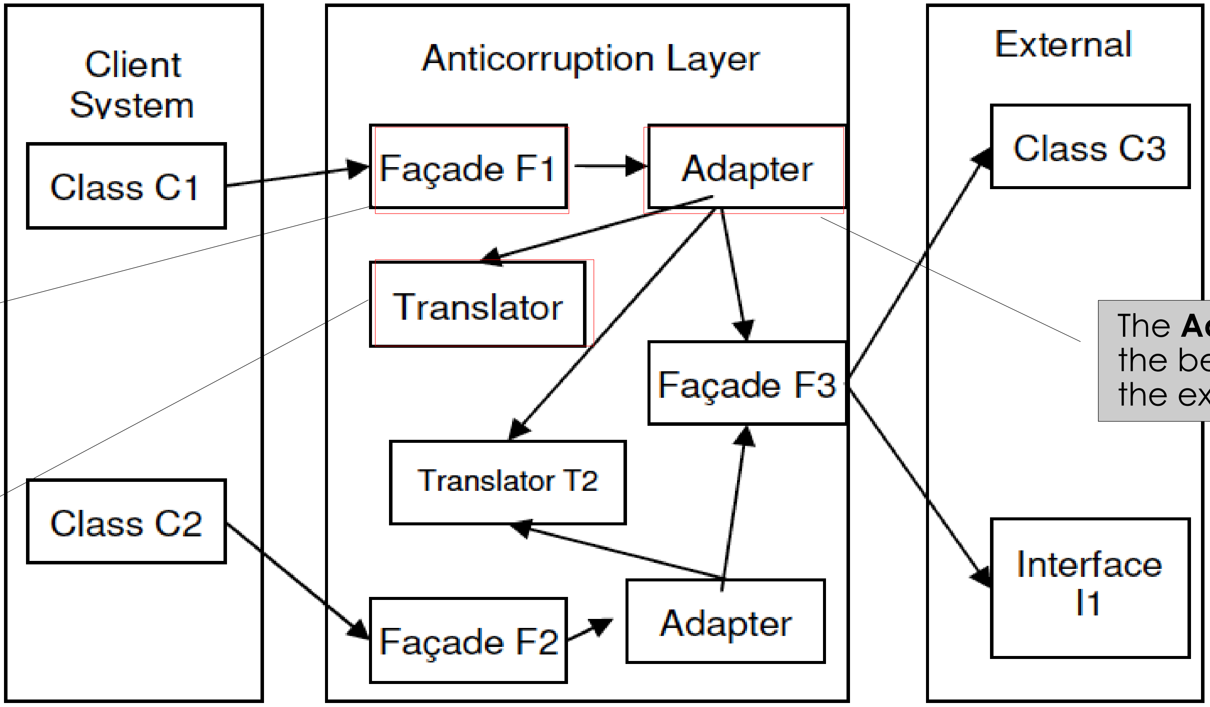
Patterns of Interaction (5/8)

- **Anticorruption Layer**
  - Create a layer between the client model and the external one (see below).
  - The layer is a ‘natural part’ of the client – it uses concepts and actions that are familiar to the client model.
  - The layer interacts with the external model using its concepts and actions.
- Motivation
  - The system has to interact with an external application, i.e. a legacy system.
  - The external application’s model is confused and hard to work with.
- Benefits
  - Prevents the client model to be altered by the external one.
- Challenges
  - Implementation of an anticorruption layer may be complex and costly.

Strategic Design

Patterns of Interaction (6/8)

- Building blocks of the Anticorruption Layer



Facade `F1` acts as a **Service** provide to the client system..

The **Translator** per-forms data and object conversion.

The **Adapter** wraps the behaviour of the external system.

Source: [MarAvr07]

Strategic Design

Patterns of Interaction (7/8)

- **Conformist**
  - One subsystem (customer) depends on the other (supplier).
  - The supplier offers a domain model to the customer - the customer accepts the supplier's model, conforming entirely to it.
- Motivation
  - Customer-Supplier relationship between subsystems, but the supplier team has no motivation to consider the customer's requirements.
- Benefits
  - Reduction of development effort.
- Challenges
  - The customer must accept the supplier's model in any case.
  - Development of acceptance tests for interface validation.

Strategic Design

Patterns of Interaction (8/8)

- **Separate Ways**
  - The bounded contexts and their models are created independently from each other.
- Motivation
  - An application is to be built of smaller subsystems which have only little in common from a modeling perspective.
  - Integration of the bounded contexts would introduce too much effort.
- Benefits
  - Only little integration effort – the subsystems share just a common thin GUI which acts as a portal.
- Drawbacks
  - To integrate the independently developed subsystems is a challenge.

Domain Driven Design & Microservices

Why is DDD Relevant to the Design of Microservice Architectures?

- **Microservices**
  - Architectural style for distributed, service-based systems
  - Applications composed by a great number of loosely coupled small services
  - Services implement a set of coherent business features.
  - Services are designed and developed by autonomous teams, largely independent from each other.
- DDD
  - Provides various modeling patterns and techniques for the identification of domain concepts and their encapsulation
  - The concepts are characterized by high coherence within conceptual boundaries.
- DDD might serve as a solid foundation for service decomposition.

Domain Driven Design & Microservices

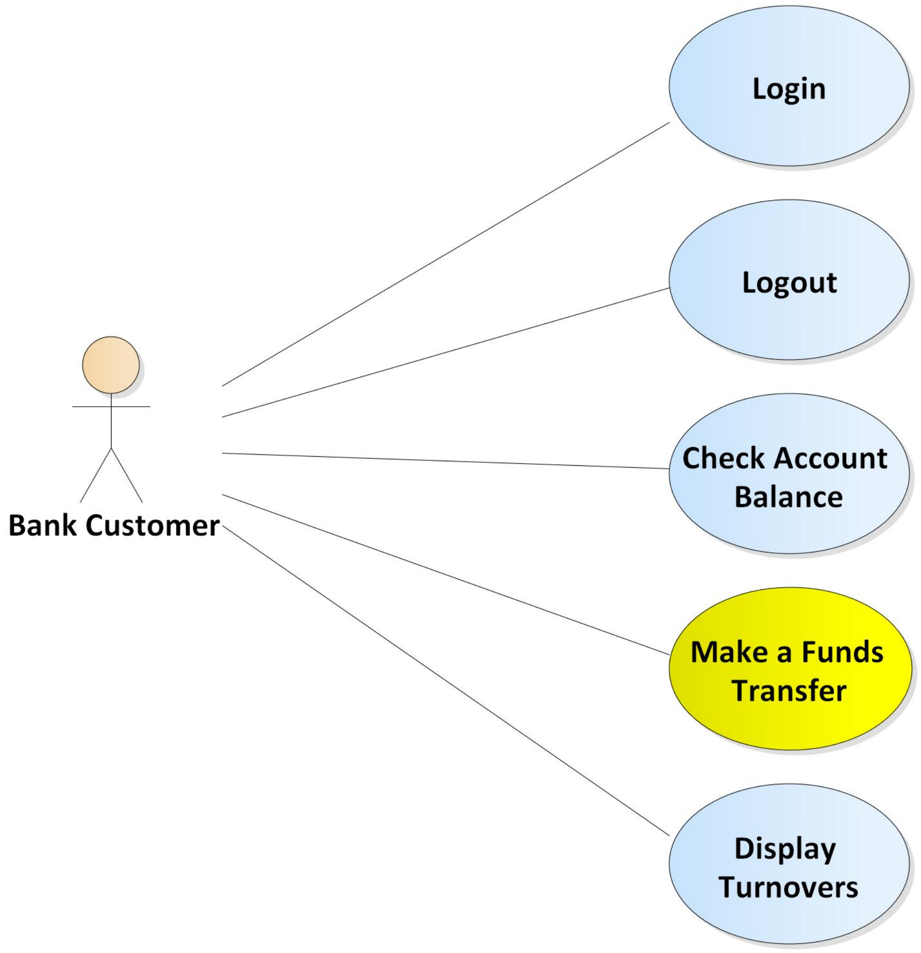
Approach

- Deduction of Microservices
  - Each Bounded Context is mapped to one single microservice.
- Interaction between microservices
  - Based on one of the DDD patterns of interaction, e.g. ...
  - Anticorruption Layer: **Associations** between different Bounded Contexts are mapped to service interfaces, through which shared objects are ex-changed.
- Challenges*)
  - Missing information, necessary for the deduction of microservices, e.g. inter-faces, operations, endpoints, protocols, message formats.
  - Domain modeling across distributed autonomous microservice teams.

Example

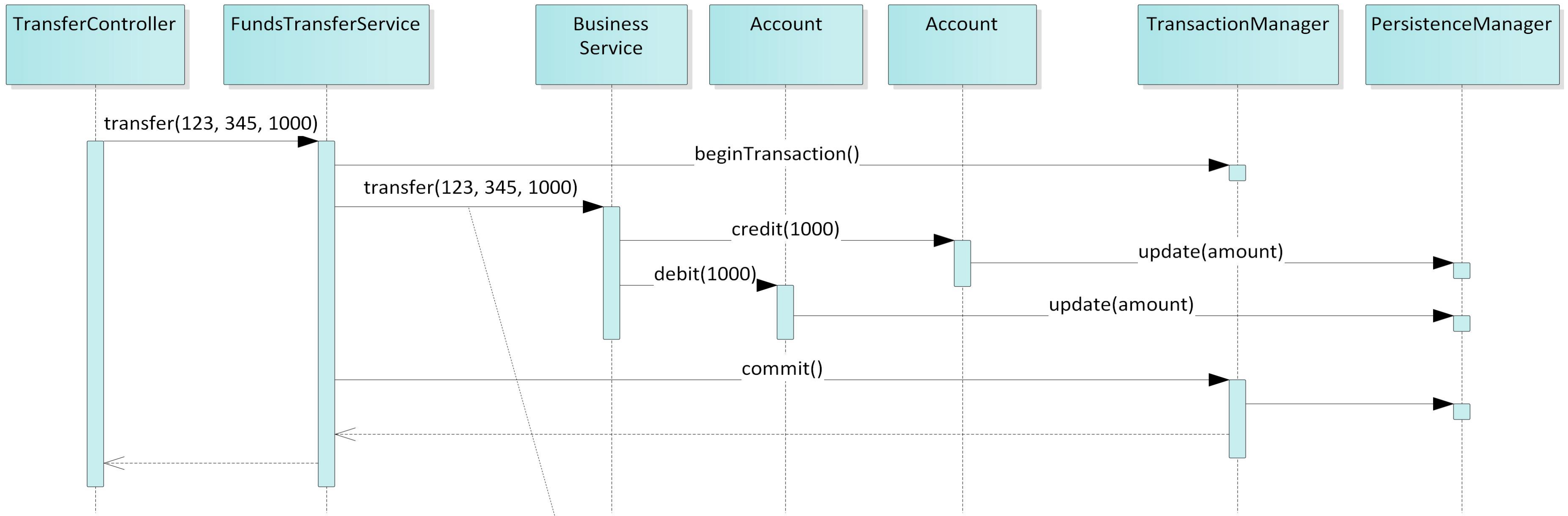
Online Banking – Use Cases

- The application provides several features for maintaining a bank account
- 'Funds transfer'
  - User enters 2 account numbers and an amount of money and initiates the transfer.
- To make the example manageable …
  - … the design is kept as simple as possible (is NOT realistic!)
  - … major technical features are omitted.



Example

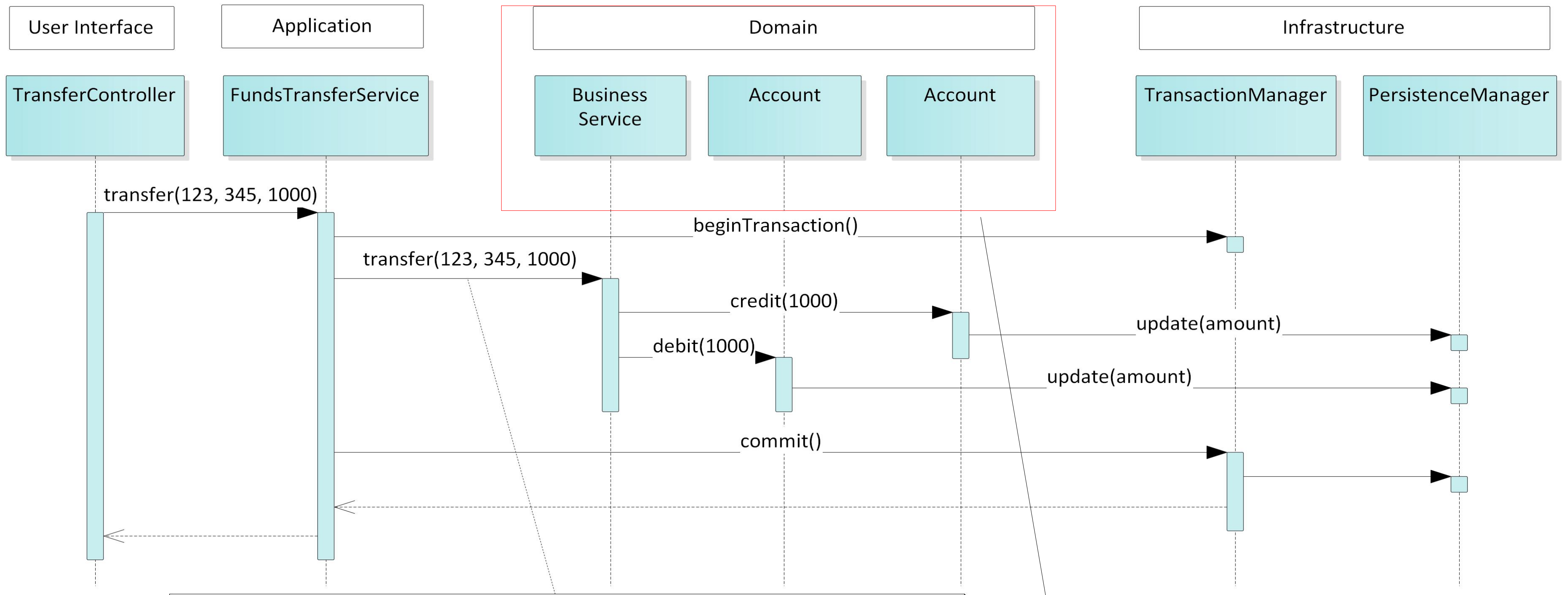
Online Banking – Sequence Diagram



Every credit has a matching debit. Therefore the following assertion (invariance) apply:
- PRECONDITION: a:= fromAccount.amount + toAccount.amount;
- POST CONDITION: a == fromAccount.amount + toAccount.amount.
If the assertion fails, transfer will throw a business exception and the transaction will be rolled-back.

Example

Online Banking – Layered Architecture



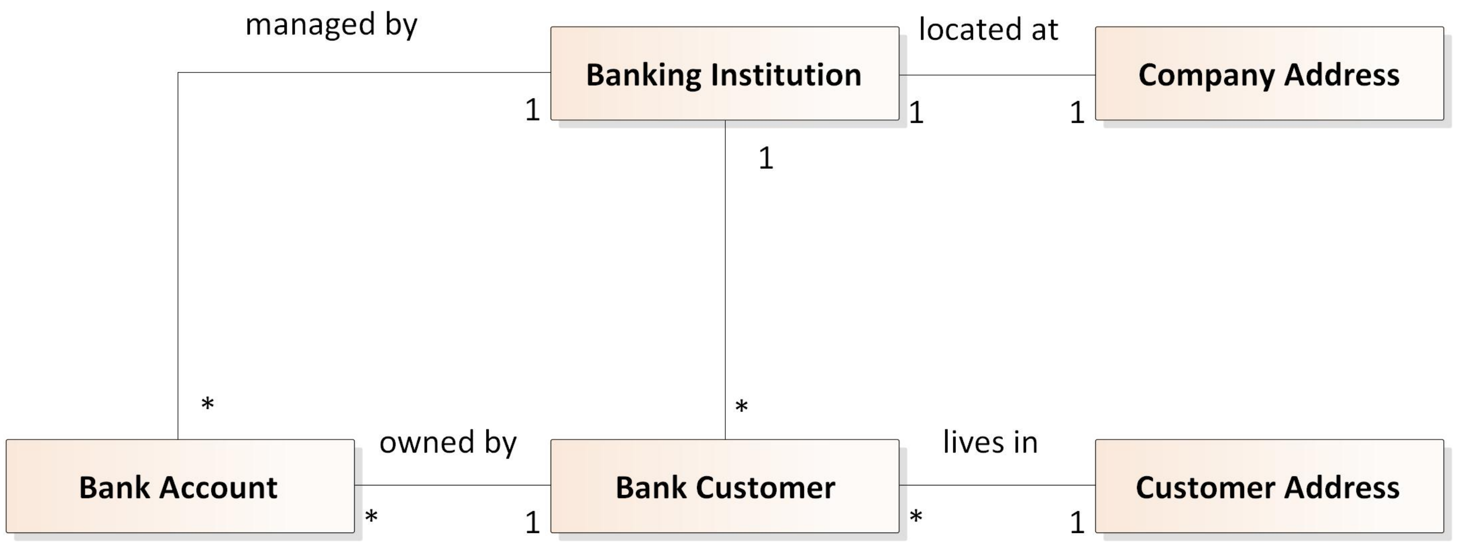
Every credit has a matching debit. Therefore the following assertion (invariance) apply:
- PRECONDITION: a:= fromAccount.amount + toAccount.amount;
- POST CONDITION: a == fromAccount.amount + toAccount.amount.
If the assertion fails, transfer will throw a business exception and the transaction will be rolled-back.

The **domain layer**, not the application layer is responsible for ensuring business rules, such as the invariance of amount sums.

Example

Online Banking – Structural Domain Model

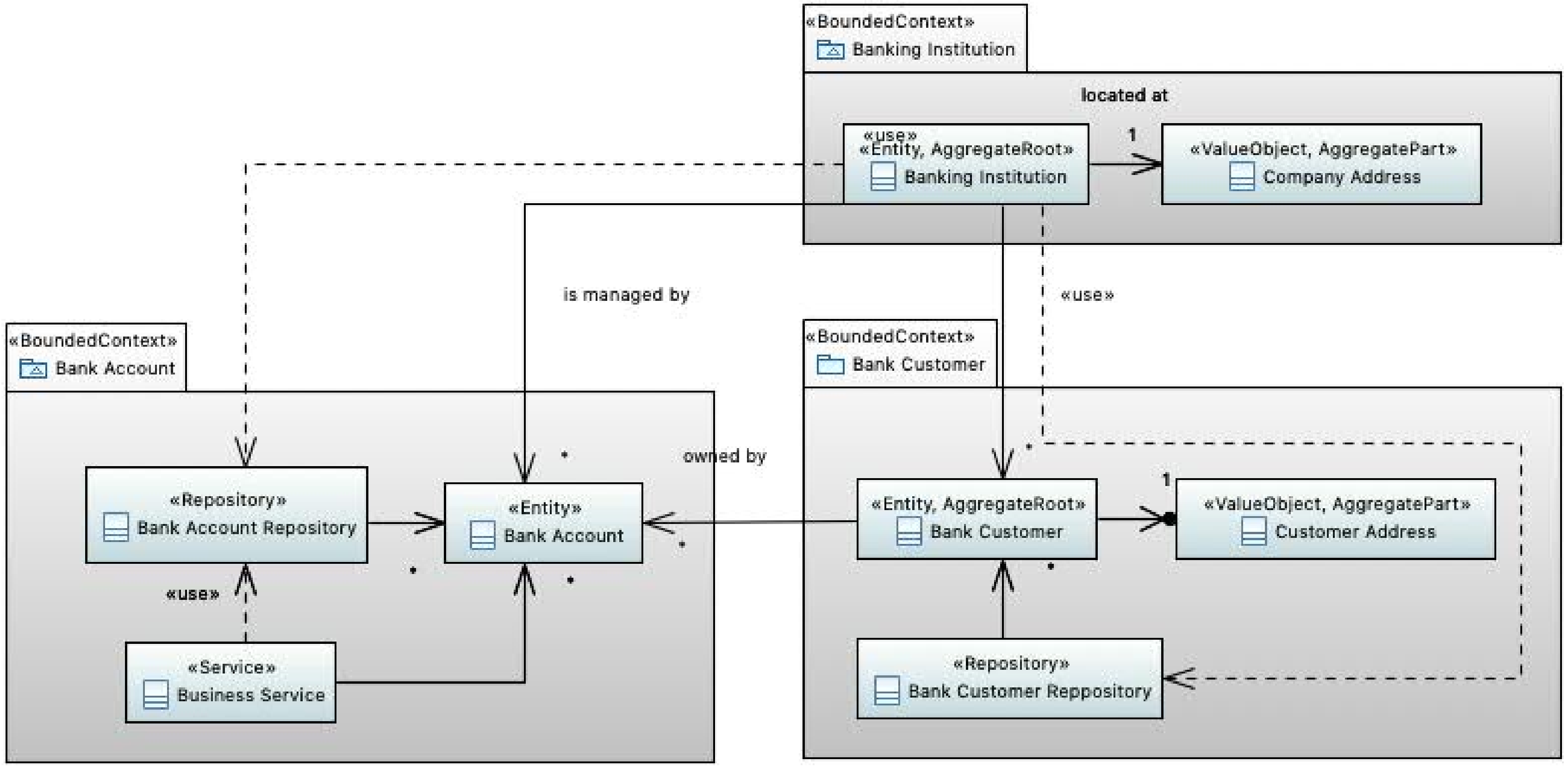
- Structural Domain Models are preferably represented by UML Class diagrams (Evans)
  - Named classes
  - Attributes and methods
  - Associations (normally non-navigable)
  - Multiplicities
  - Constraints (e.g. in the form of notes)



Example

Online Banking – Model-driven Design

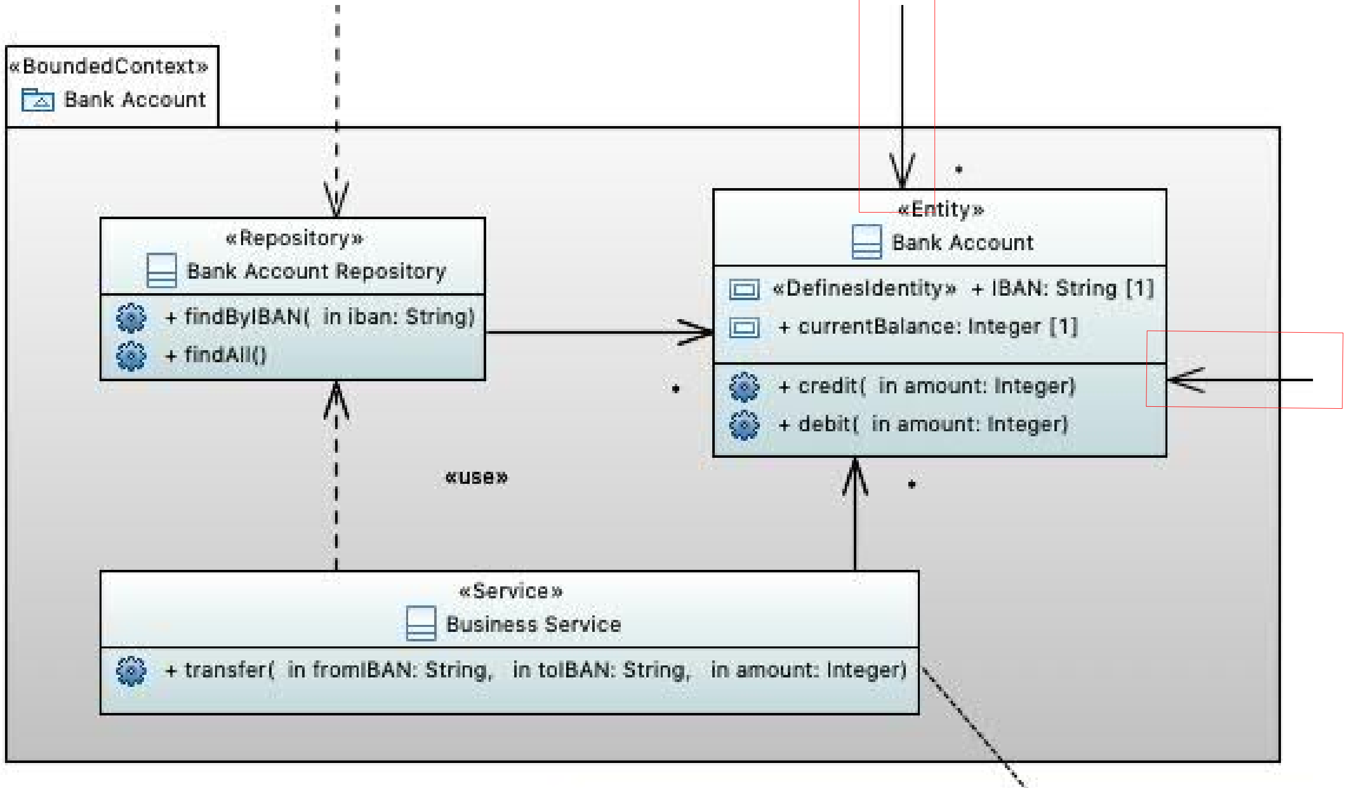
- The Structural Domain Model is refined by applying additional DDD patterns.



Example

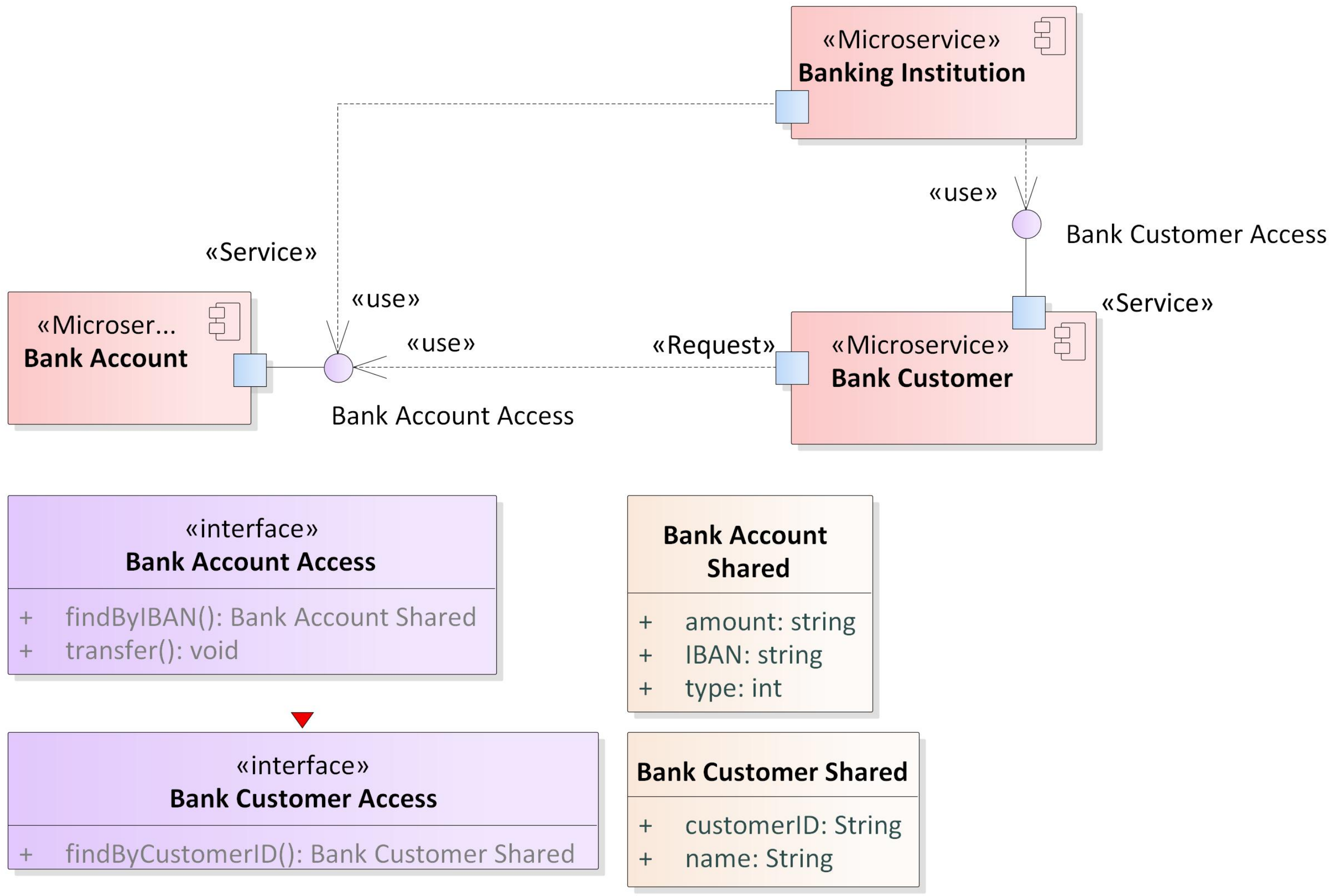
Online Banking – Domain Driven Design for Microservices (1/2)

- Deduction of Microservices
  - Each Bounded Context is mapped to one single micro-service.
  - Associations bet-ween different Bounded Contexts are mapped to service interfaces, through which shared objects are exchanged.


«BoundedContext»
Bank Account
«Repository»
Bank Account Repository
+ findByIBAN(in iban: String)
+ findAll()
«Entity»
Bank Account
«DefinesIdentity» + IBAN: String [1]
+ currentBalance: Integer [1]
+ credit(in amount: Integer)
+ debit(in amount: Integer)
«use»
«Service»
Business Service
+ transfer(in fromIBAN: String, in toIBAN: String, in amount: Integer)
*
*
*

Example

Online Banking – Domain Driven Design for Microservices (2/2)



References

BeRaSV18	Betts, T., and Rayner, Paul (2018), Domain-Driven Design in Practice, InfoQ eMag, no 65.
BeBoEd98	S. Beydeda, M. Book, and V. G. (Eds.) (1998), Model-driven Software Development. Springer, Berlin and Heidelberg.
Evans04	Evans, E. (2004), Domain-driven design: tackling complexity in the heart of software, Addison-Wesley Professional, Upper Saddle River et al.
Evans15	Evans, E. (2015), Domain–Driven Design Reference - Definitions and Pattern Summaries, Domain Language, Inc.
KaetPat08	S. Kätker and S. Patig (2008), Model-Driven Development for Service-Oriented Architectures in Practice, in: PIK Praxis der Informationsverarbeitung und Kommunikation, vol. 31 (2008) 4, pp. 210–217.
Kusche13	Kusche, K. (2013), Domain Driven Design, WWW: https://www.computerix.info/archiv.html, accessed 18.08.2019.
MarAvr07	Marinescu, F., and Avram, A. (2007), Domain-driven design Quickly. InfoQ.-com.
RaSoSa18	Rademacher, F., Sorgalla, J., and Sachweh, S. (2018), Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective, in: IEEE Software, vol 35, no 3, p. 36 - 43.

References

RaSaZu17	Rademacher, F., Sachweh, S., and Zündorf, A. (2017), Towards a UML profile for domain-driven design of microservice architectures, in: IEEE International Conference on Software Engeneering and Formal Methods, p. 230–245.
Selic03	B. Selic (2003), The pragmatics of model-driven development, in: IEEE software, vol. 20, no. 5, Art. no. 5