

2.– 5. September 2013
in Nürnberg



Herbstcampus

Wissenstransfer
par excellence

Geschichtet, nicht gerührt?

Einführung in die N-Layer Domain-Oriented Architecture

Tobias Krügel

complement AG

Agenda



Einführung

N-Layered Domain-Oriented Architecture

Architecture Patterns

(Visual Studio – Architecture Validation)

Fazit



Einführung

N-Layer Domain-Oriented Architecture

Architecture Patterns

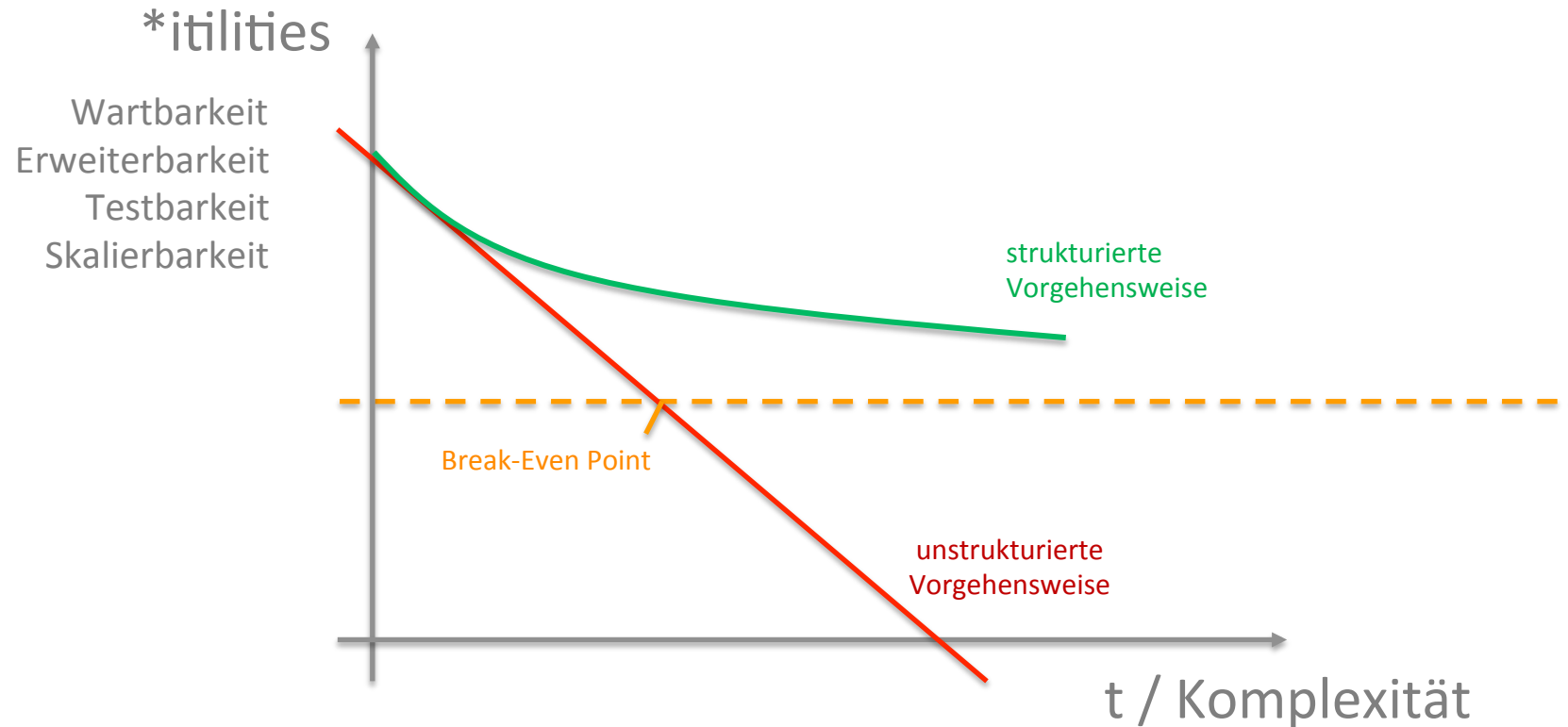
Visual Studio – Architecture Validation

Fazit

Einführung



Nutzen der N-Layered Domain-Oriented Architecture?



N-Layered Domain-Oriented Architecture Guide

Leitfaden zur Unterstützung der Realisierung komplexer, flexibler und skalierbarer Anwendungen



Microsoft Spain - Domain Oriented N-Layered .NET
4.0 Sample App

<http://microsoftnlayerapp.codeplex.com/>



Layer vs. Tier

Layer

Beschreibt logische Gliederung von Komponenten

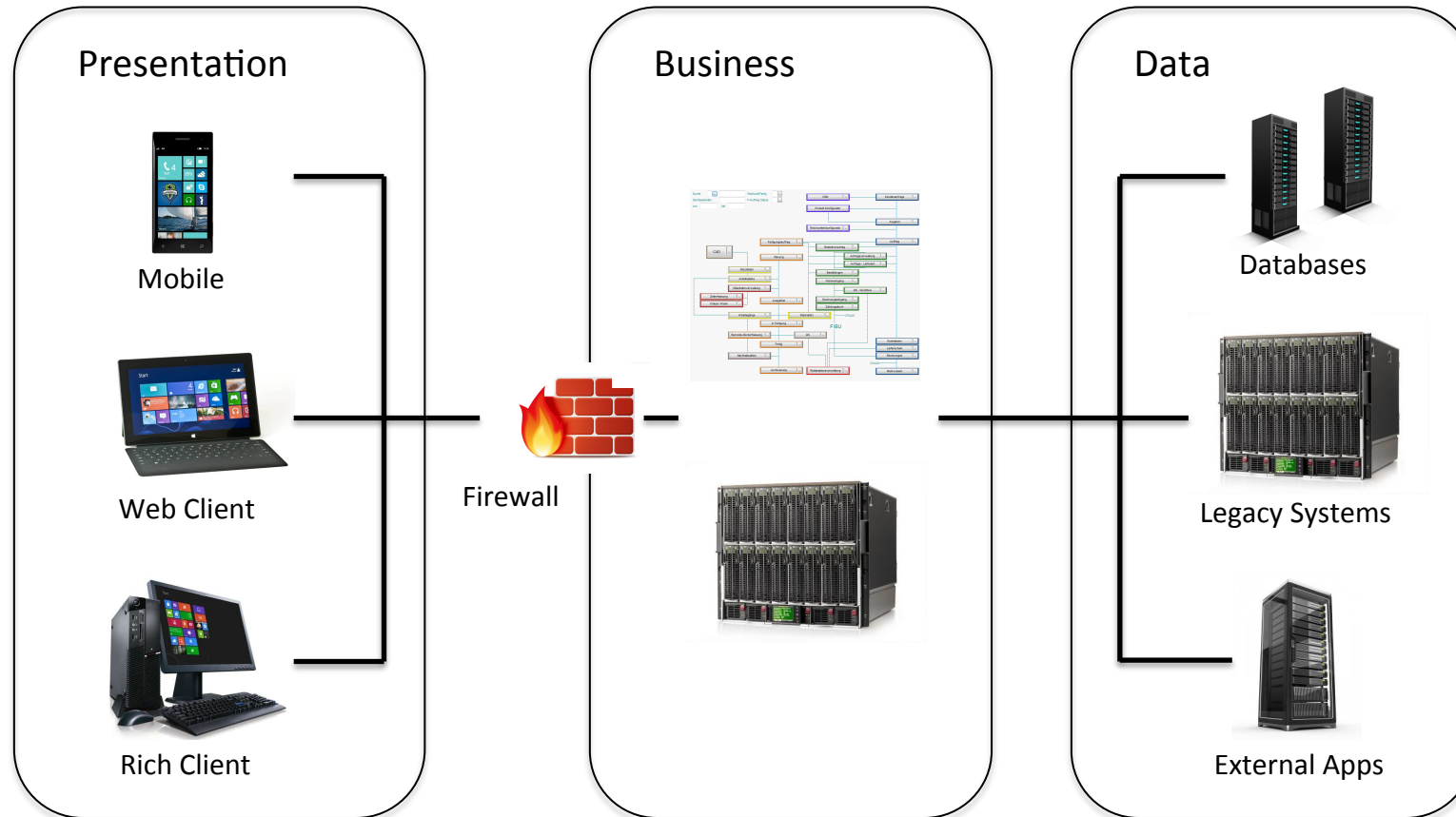
Tier

Beschreibt physikalischen Verteilung von Komponenten (inkl. Netzwerktopologie)

Einführung

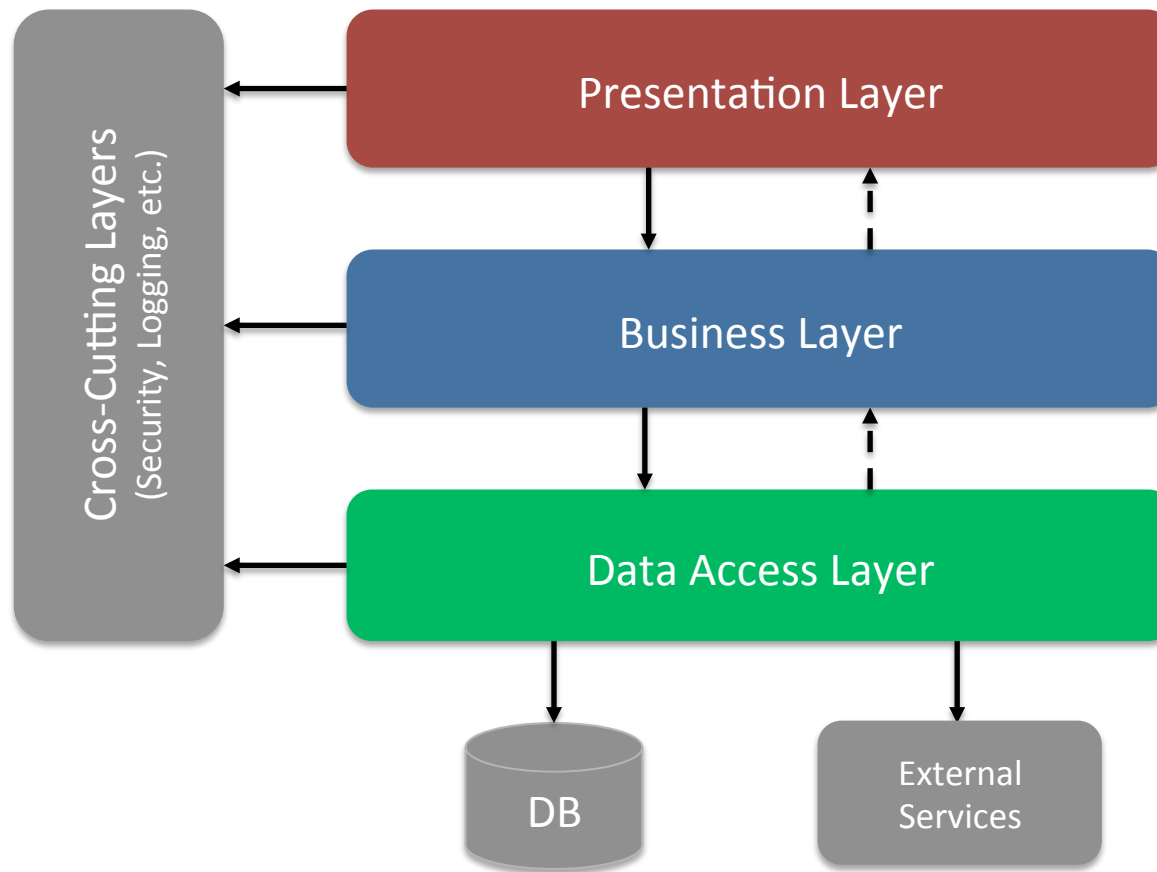


3-Tier Architecture (physical)





N-Layer Architecture (logical)





Domain-Driven Design (DDD)...

ist eine Herangehensweise an die Modellierung komplexer objektorientierter Software

Annahmen:

- Der Schwerpunkt des Softwaredesigns liegt auf der Fachlichkeit und der Fachlogik

Voraussetzung: Genügend komplexe fachliche Logik !!!

- Der Entwurf komplexer fachlicher Zusammenhänge sollte auf einem Modell der Anwendungsdomäne, dem Fachmodell basieren.

http://de.wikipedia.org/wiki/Domain-Driven_Design



Single Responsibility Principle

Open Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle



Design Prinzipien

S

Single Responsibility Principle

Jede Klasse erfüllt genau eine fest definierte Aufgabe, wobei die Abhängigkeiten zu anderen Klassen minimal sind.

O

Open Close Principle

Das Verhalten einer Klasse sollte erweiterbar sein, ohne den bestehenden Code ändern zu müssen

L

Liskov Substitution Principle

Subsysteme müssen auf Basis abstrakter Typen ersetzt werden können (Implementierung gegen Interfaces anstatt gegen konkrete Typen).

I

Interface Segregation Principle

Interfaces sollten nur die Methoden und Properties veröffentlichen, die vom jeweiligen Konsumenten benötigt werden (Aufteilung in logische Einheiten statt eines großen komplexen Interfaces).

D

Dependency Inversion Principle

Direkte Abhängigkeiten von Klassen zueinander sollten über abstrakte Typen aufgelöst werden.

Agenda



Einführung

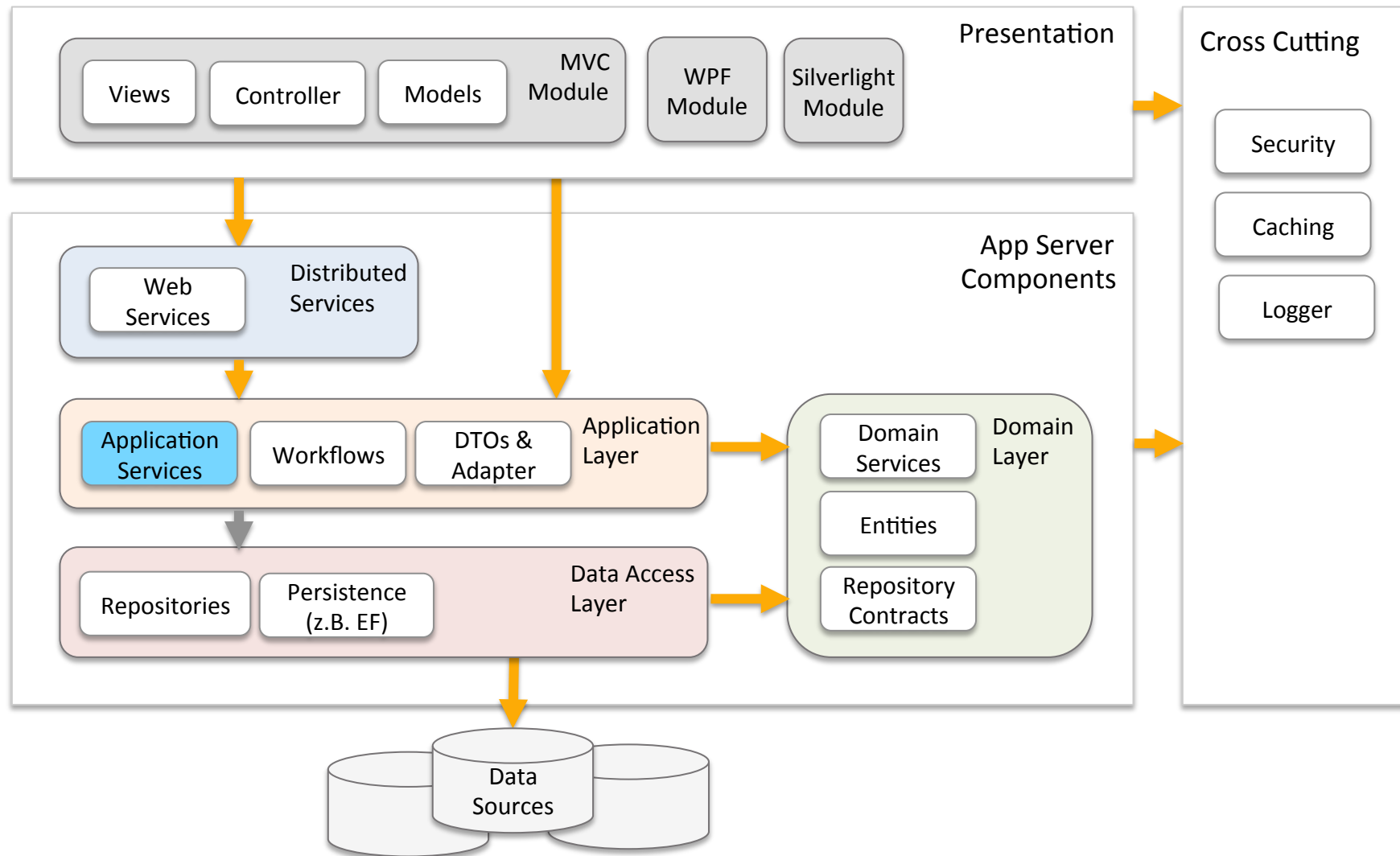
N-Layered Domain-Oriented Architecture

Architecture Patterns

Visual Studio – Architecture Validation

Fazit

N-Layered Domain-Oriented Architecture



Agenda



Einführung

N-Layered Domain-Oriented Architecture

Architecture Patterns

Visual Studio – Architecture Validation

Fazit



Architecture Patterns

- Dependency Injection / Inversion Of Control
- Data Transfer Objects / Type Adapter
- Repository
- Unit Of Work
- Specification



Architecture Patterns

- **Dependency Injection / Inversion of Control**
- Data Transfer Objects / Type Adapter
- Repository
- Unit Of Work
- Specification

Dependency Injection / Inversion of Control

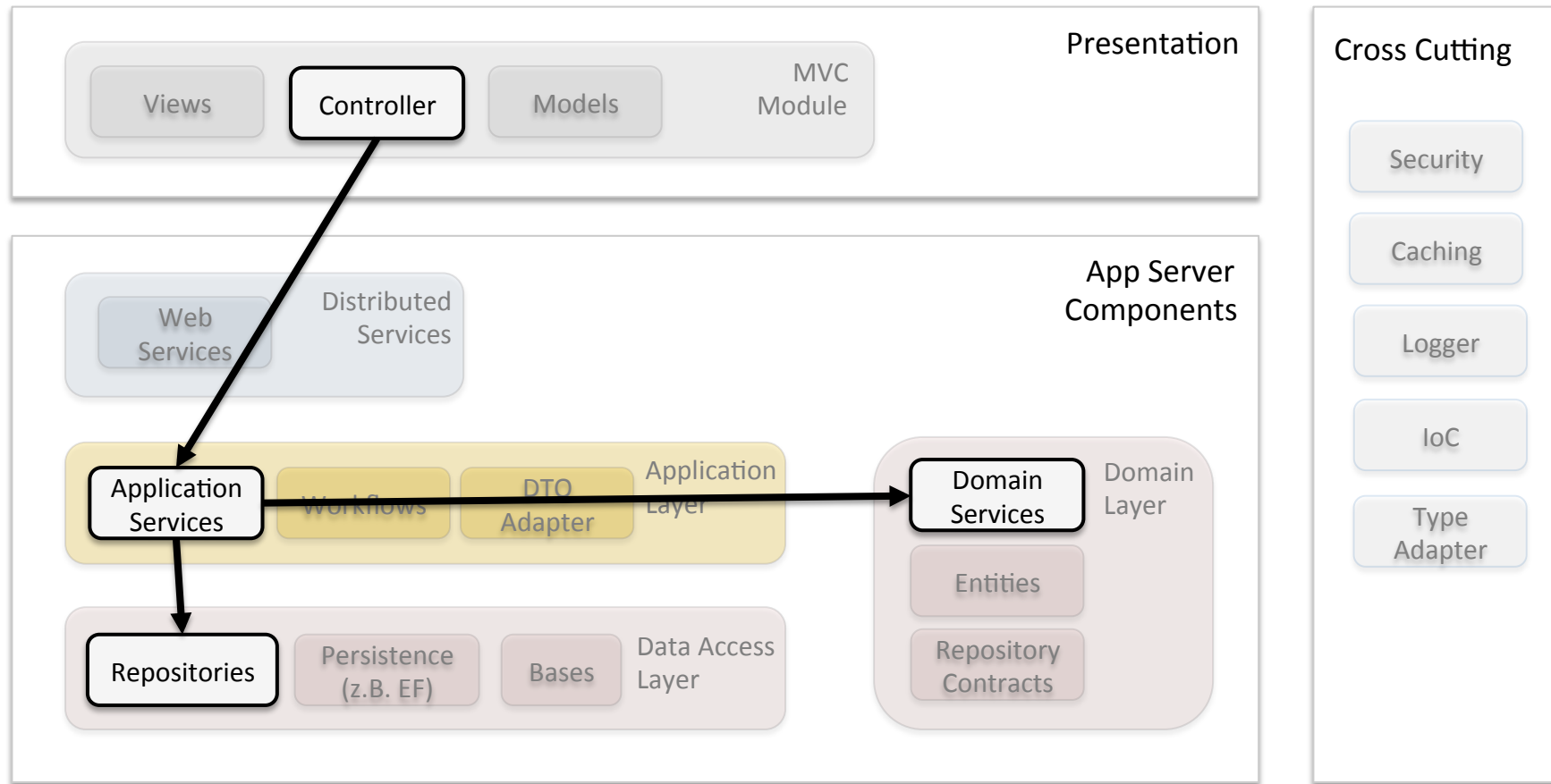


- Starke Abhängigkeiten (engen Kopplung)
- Aufwand bei Änderungen
- Automatisierte Tests schwer bis gar nicht realisierbar



Dependency Injection / Inversion of Control

Abhängigkeiten



Dependency Injection / Inversion of Control

Antipattern



```
public class ApplicationService
{
    private readonly DomainService _service;
    private readonly BankRepository _repository;

    public ApplicationService()
    {
        _service = new DomainService();
        _repository = new BankRepository();
    }

    public void Transfer(int source, int target, decimal amount)
    {
        BankAccount sourceAccount = _repository.GetById(source);
        BankAccount targetAccount = _repository.GetById(target);

        _service.Transfer(sourceAccount, targetAccount, amount);
    }
}

public class DomainService
{
    public void Transfer(BankAccount source, BankAccount target, decimal amount)
    {
        // Do transfer
    }
}
```

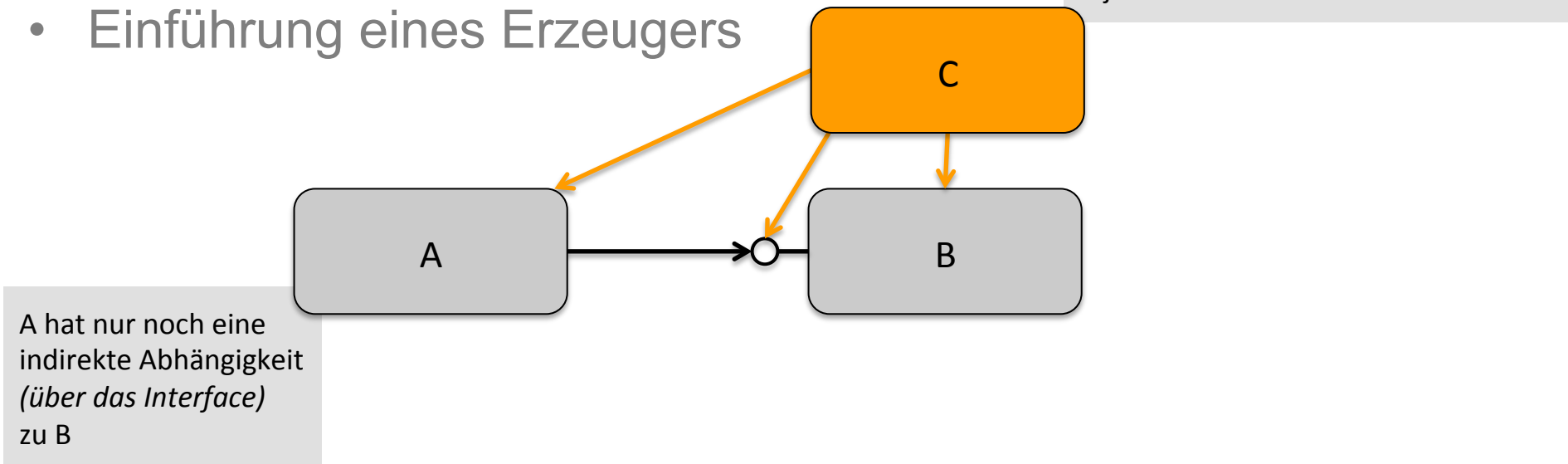
Starke Abhängigkeit durch konkrete Typen

Erzeugung von Abhängigkeiten über Konstruktor

Dependency Injection / Inversion of Control



- Abhängigkeit injizieren (anstatt zu erzeugen)
- Entkopplung durch Interfaces
- Einführung eines Erzeugers



Dependency Injection / Inversion of Control



```
public class ApplicationService
{
    private readonly IDomainService _service;
    private readonly IBankRepository _repository;

    public ApplicationService(IDomainService service, IBankRepository repository)
    {
        _service = service;
        _repository = repository;
    }

    public void Transfer(int source, int target, decimal amount)
    {
        BankAccount sourceAccount = _repository.GetById(source);
        BankAccount targetAccount = _repository.GetById(target);

        _service.Transfer(sourceAccount, targetAccount, amount);
    }
}

public class IoCContainer
{
    public ApplicationService CreateApplicationService()
    {
        return new ApplicationService(new DomainService(), new BankRepository());
    }
}
```



Entkopplung starker
Abhängigkeiten durch
Interfaces

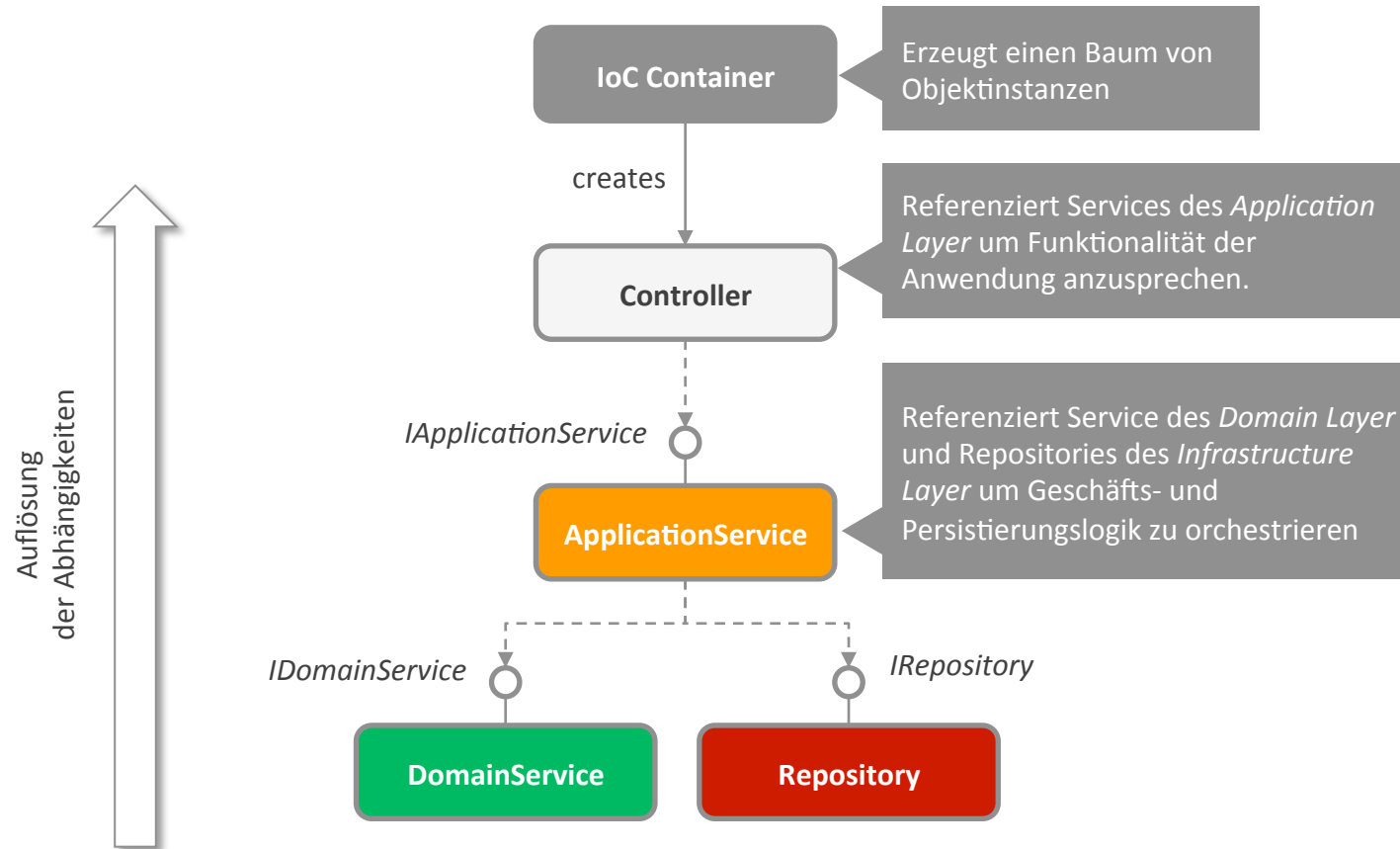


Erzeugung von
Abhängigkeiten durch
Injizierung



Container kümmert
sich um Auflösung der
Abhängigkeiten

Dependency Injection / Inversion of Control



Dependency Injection / Inversion of Control



patterns & practices - Unity



<http://unity.codeplex.com/>



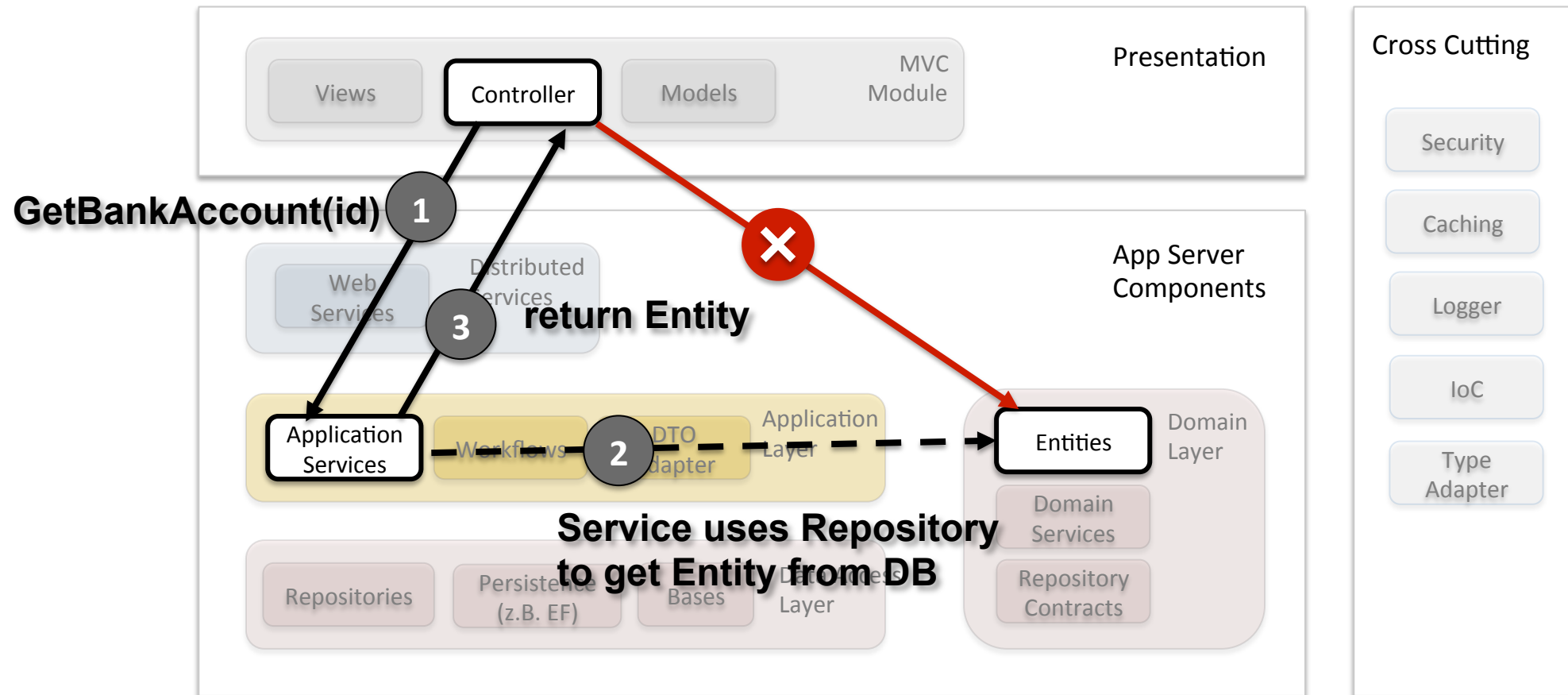
Architecture Patterns

- Dependency Injection / Inversion Of Control
- **Data Transfer Objects / Type Adapter**
- Repository
- Unit Of Work
- Specification

Data Transfer Object / Type Adapter



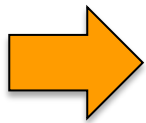
Data Flow





Probleme

- Presentation Layer ist vom Domain Layer abhängig
- Implizite Veröffentlichung von Domain Logik im Presentation Layer

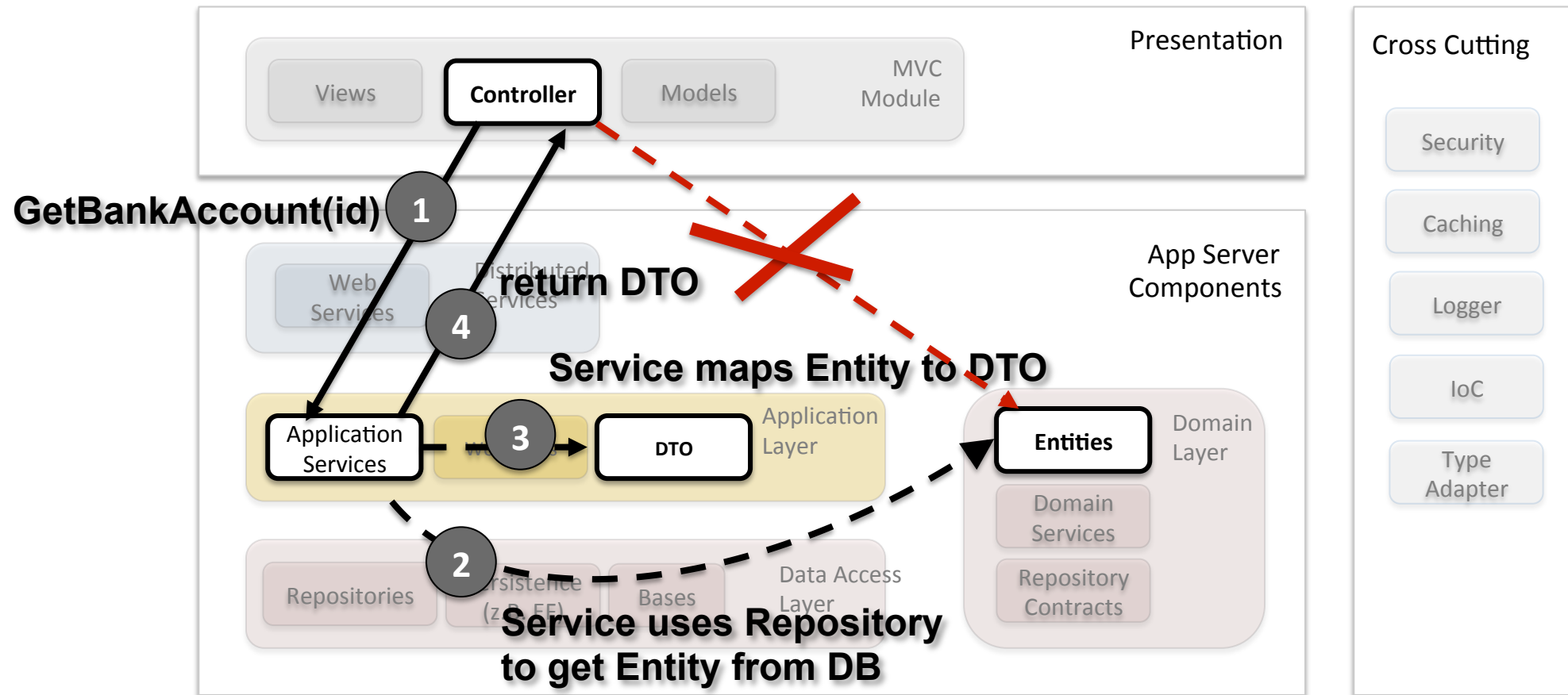


Entitäten dürfen nicht an die Presentation Layer weitergegeben werden

Data Transfer Object / Type Adapter



Data Flow



Data Transfer Object / Type Adapter



Beispiel

```
public class ApplicationService
{
    private readonly IBankRepository _repository;

    public BankAccountService(IBankRepository repository)
    {
        _repository = repository;
    }

    public BankAccountDTO GetBankAccount(int id)
    {
        BankAccount bankAccount = _repository.Get(id);

        BankAccountDTO bankAccountDto =
            Mapper.Map<BankAccountDTO>(bankAccount);

        return bankAccountDto;
    }
}
```

Wird von Presentation Layer aufgerufen

Ermittlung einer Entität aus dem Domain Layer

Mapping auf DTO unter Verwendung einer 3rd Party Komponente (automapper)

DTO wird anstelle der Entität zurück gegeben

Data Transfer Object / Type Adapter



Automapper

auto~~x~~mapper



<http://automapper.codeplex.com/>



Architecture Patterns

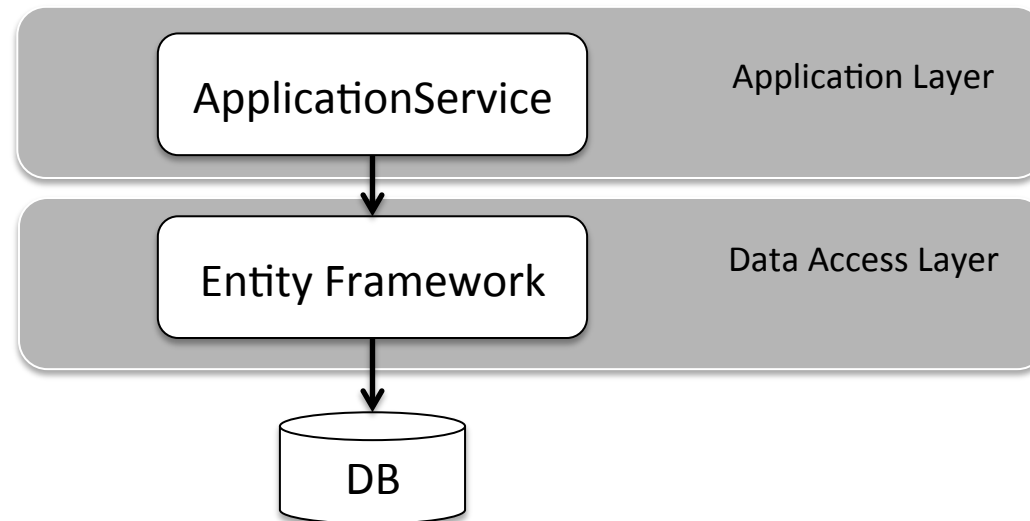
- Dependency Injection / Inversion Of Control
- Data Transfer Objects / Type Adapter
- **Repository**
- Unit Of Work
- Specification

Repository



Probleme

- Abhängigkeiten von Datenzugriffstechnologie (z.B. EF)
- Erhöhter Aufwand beim Austausch
- Automatisierte Tests schwer bis gar nicht realisierbar



Repository

Antipattern



```
public class ApplicationService
{
    private readonly DbContext _context;

    public BankAccountService(DbContext context)
    {
        _context = context;
    }

    public BankAccountDTO GetBankAccount(int id)
    {
        BankAccount bankAccount =
            _context.BankAccounts.Where(a => a.Id == id);

        BankAccountDTO bankAccountDto =
            Mapper.Map<BankAccountDTO>(bankAccount);

        return bankAccountDto;
    }
}
```



Starke Abhängigkeit von konkreter Technologie (hier: Entity Framework)



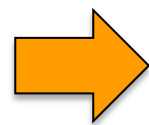
Starke Abhängigkeit von konkreter Technologie (hier: Entity Framework)

Repository

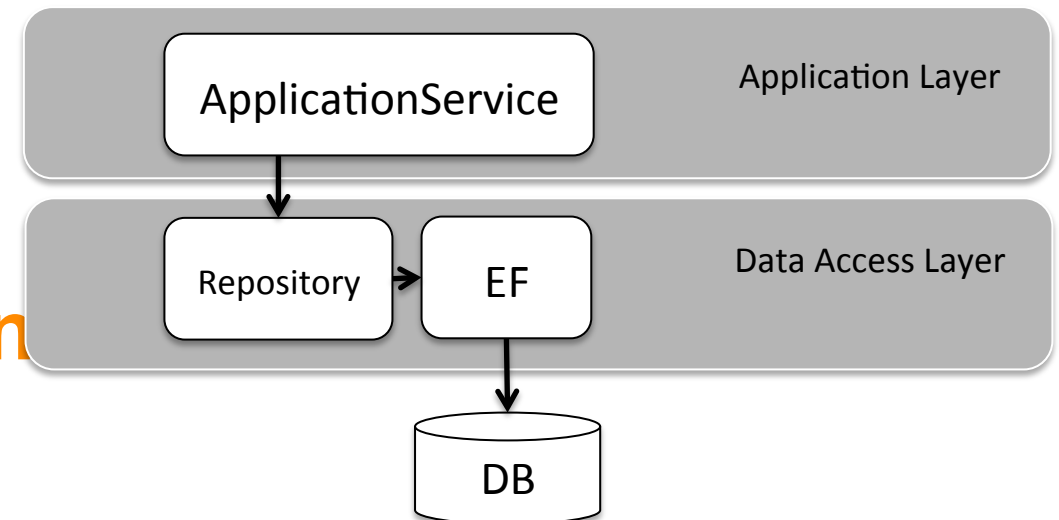


Lösung

- Abstraktion der Datenzugriffstechnologie
- Minimale Auswirkung bei Änderungen
- Vorteil: „Mocking“ der Datenbank

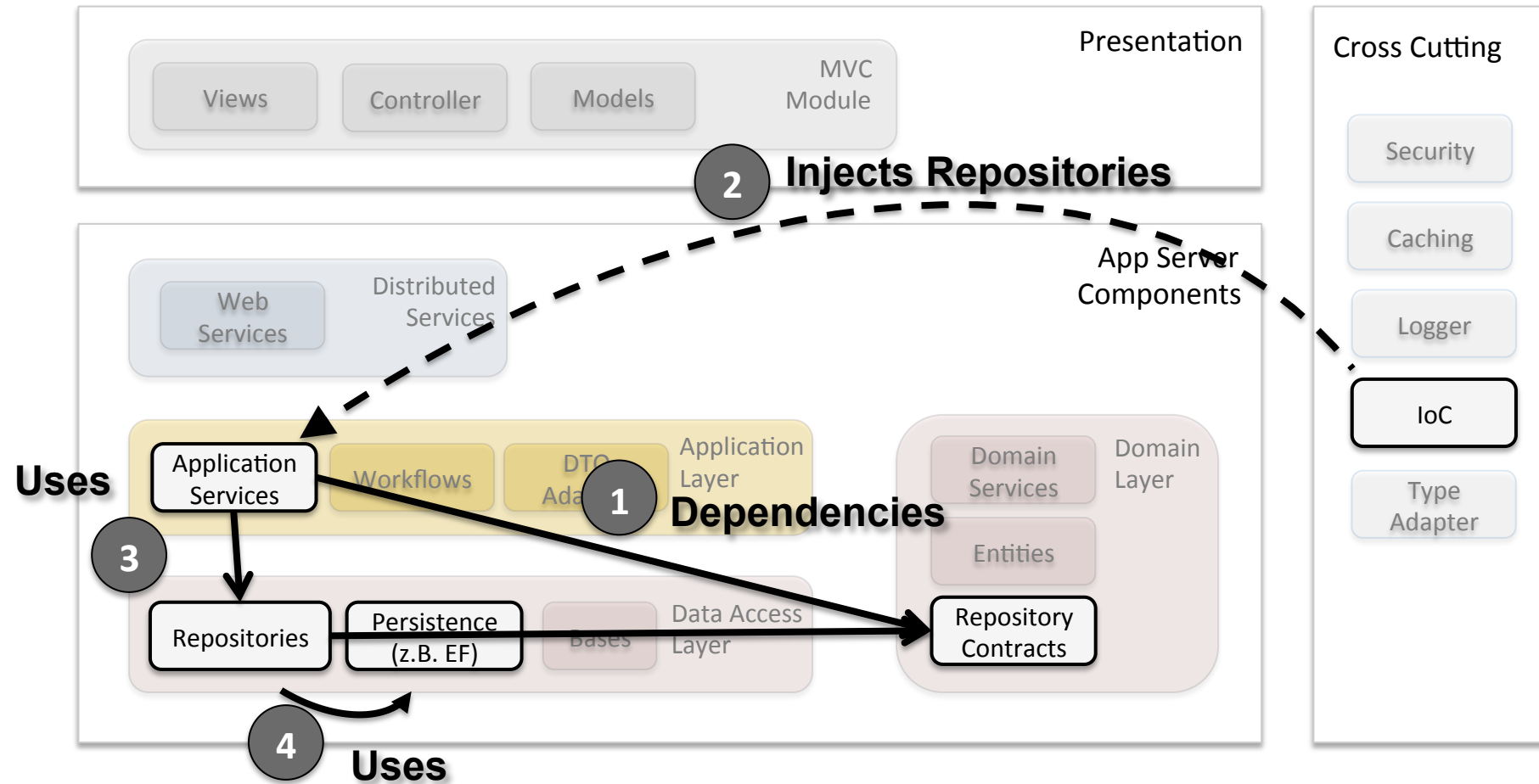


Repository Pattern



Repository

Data Flow



Repository



Beispiel

```
public class ApplicationService
{
    private readonly IBankRepository _repository;

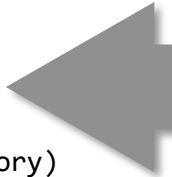
    public ApplicationService(IBankRepository repository)
    {
        _repository = repository;
    }

    public void Transfer(int source, int target, decimal amount)
    {
        BankAccount sourceAccount = _repository.GetById(source);
        BankAccount targetAccount = _repository.GetById(target);

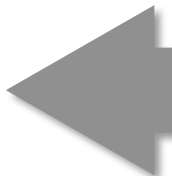
        // Call to DomainService
    }
}

public class BankRepository : IBankRepository
{
    private readonly DbContext _context;

    public BankAccount GetBankAccountById(int id)
    {
        return _context.BankAccounts.Where(a => a.Id == id);
    }
}
```



Entkopplung des
Service von konkreter
Technologie



Abstraktion konkreter
Technologie mit Hilfe
eines Repositories



Architecture Patterns

- Dependency Injection / Inversion Of Control
- Data Transfer Objects / Type Adapter
- Repository
- **Unit Of Work**
- Specification

Unit of Work

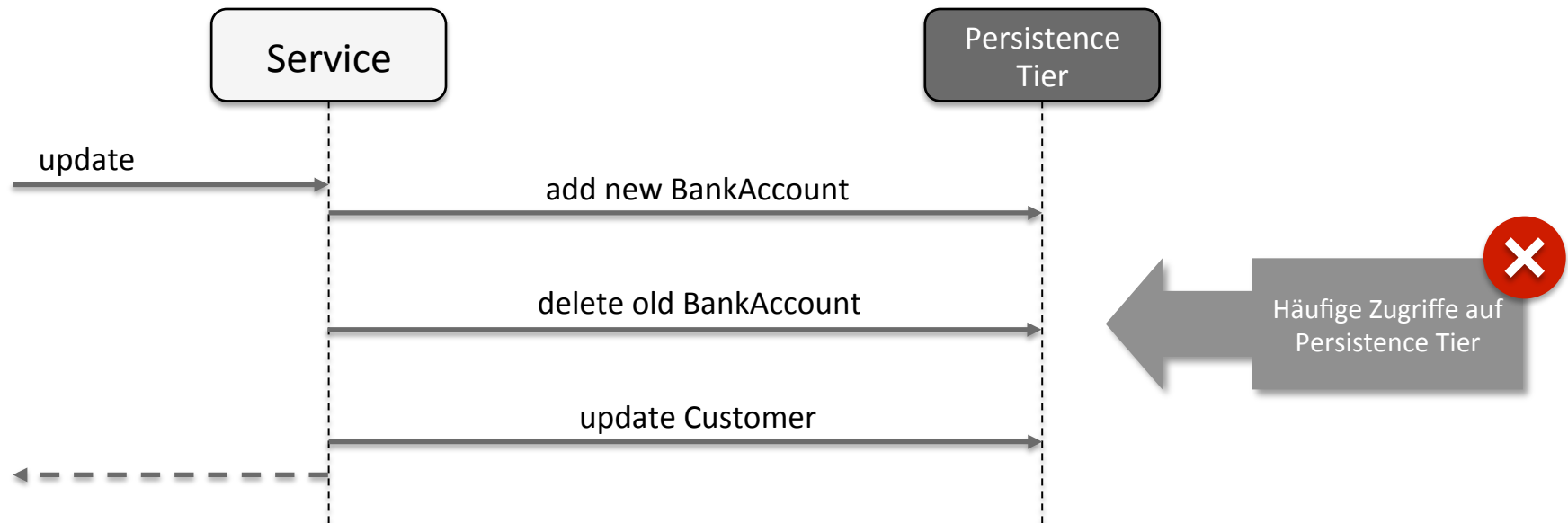


Problem

- Häufige Zugriffe auf die Datenquelle
- Performance
- Gefährdung der Konsistenz der Daten

Unit of Work

Antipattern





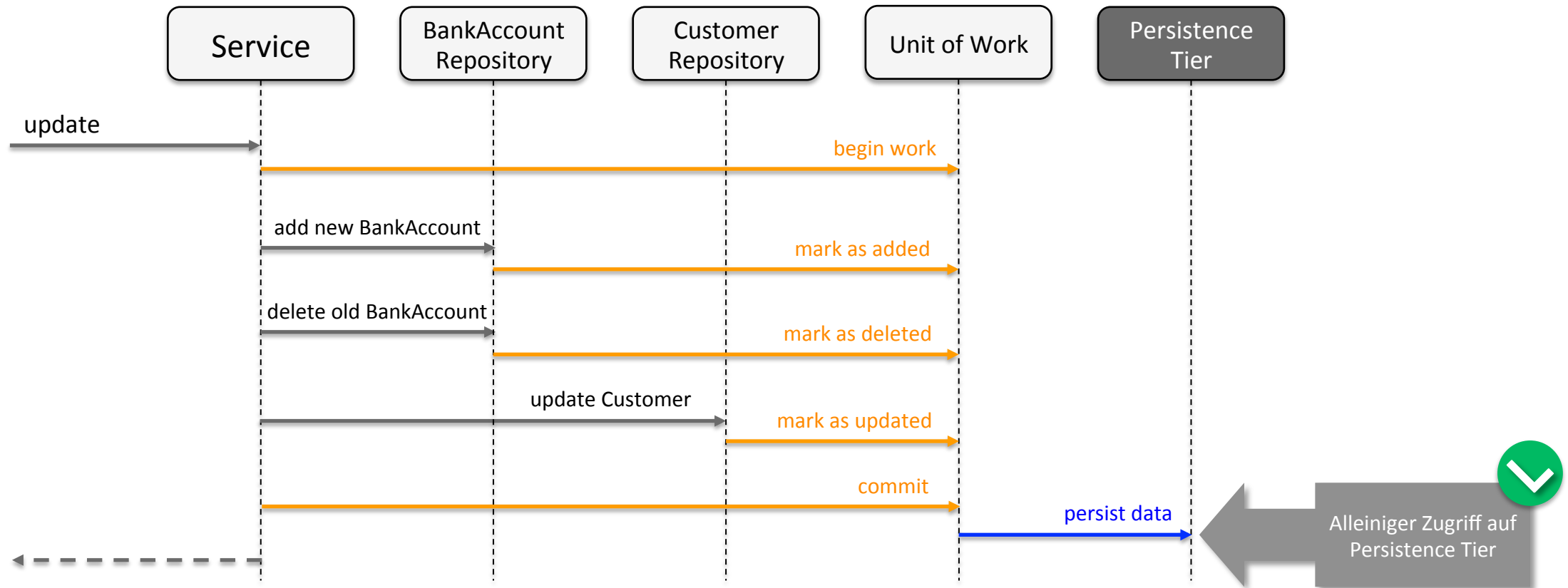
Lösung

- Vermeidung häufiger Datenbankzugriffe
- Zusammenfassung mehrerer Operationen
- Bewahrung der Datenkonsistenz (Rollback)

Unit of Work



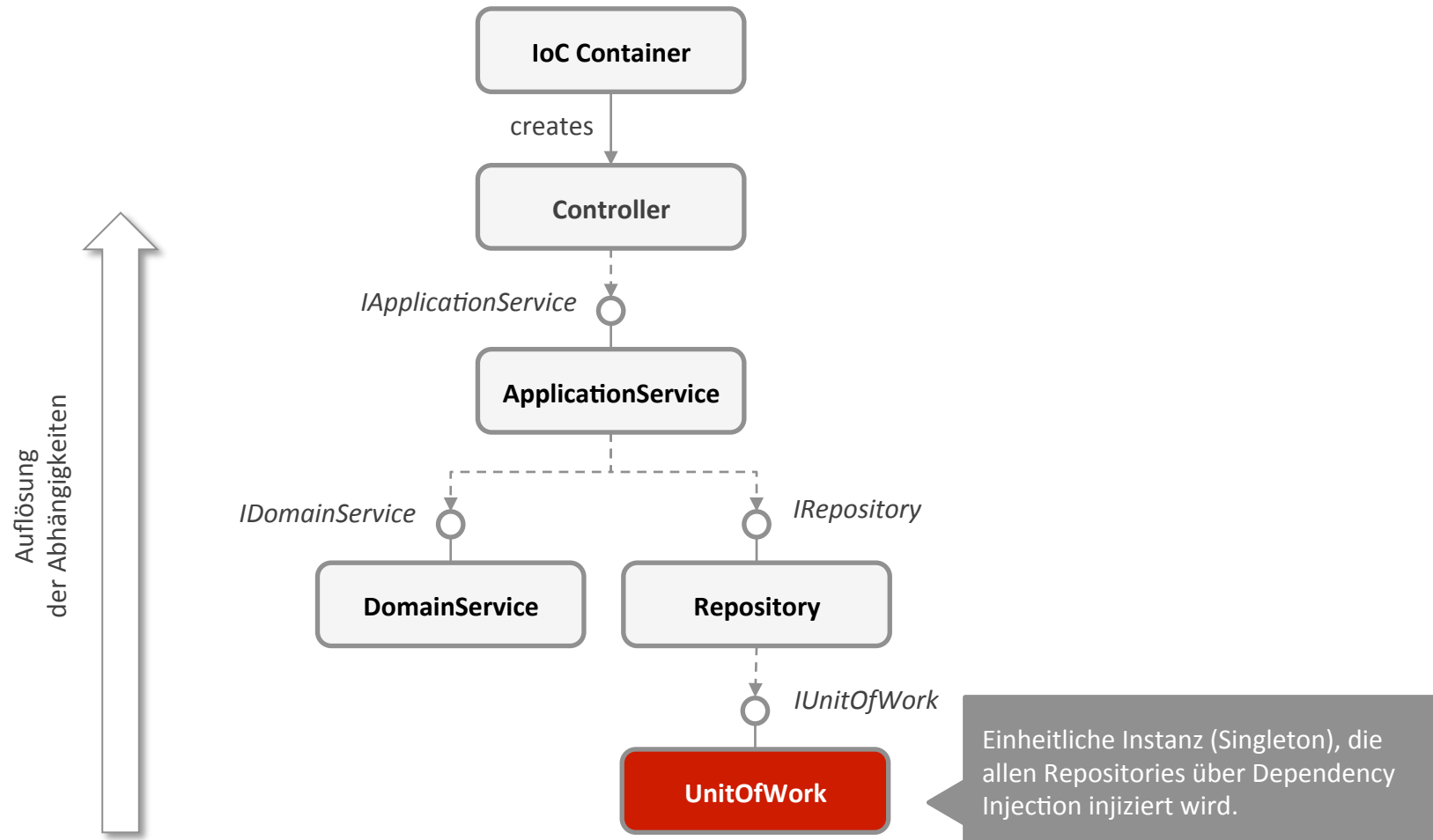
Beispiel



Unit of Work



Auflösung der Abhängigkeiten (IoC)



Unit of Work



Beispiel (1/3)

```
public class MainBCUnitOfWork : DbContext, IQueryableUnitOfWork
{
    IDbSet<Customer> _customers;
    public IDbSet<Customer> Customers
    {
        get { return _customers == null) ?? base.Set<Customer>(); }
    }

    // Weitere Properties

    public DbSet<TEntity> CreateSet<TEntity>() where TEntity : class
    {
        return base.Set<TEntity>();
    }

    public void Commit()
    {
        base.SaveChanges();
    }

    public void RollbackChanges()
    {
        base.ChangeTracker.Entries().ToList().ForEach(
            entry => entry.State = System.Data.EntityState.Unchanged);
    }
}
```

Konkretisierung des
DbContext des EF

Entitäten über
Properties

Transaktional
wirkende Methoden

Unit of Work



Beispiel (2/3)

```
public class Repository<TEntity> : IRepository<TEntity> where TEntity:Entity
{
    IQueryableUnitOfWork _UnitOfWork;

    public Repository(IQueryableUnitOfWork unitOfWork)
    {
        _UnitOfWork = unitOfWork;
    }

    public IUnitOfWork UnitOfWork
    {
        get { return _UnitOfWork; }
    }

    public virtual void Add(TEntity item)
    {
        if (item != (TEntity)null)
            GetSet().Add(item); // add new item in this set
        else
        {
            LoggerFactory.CreateLog()
                .LogInfo(Messages.info_CannotAddNullEntity, typeof(TEntity).ToString());
        }
    }
}
```

← Injizierung der Unit of Work über IoC Container

← Abstraktion der Unit of Work über Methoden des Repository

Unit of Work



Beispiel (3/3)

```
public class CustomerAppService : ICustomerAppService
{
    readonly ICountryRepository _countryRepository;
    readonly ICustomerRepository _customerRepository;

    public CustomerAppService(ICountryRepository countryRepository, ICustomerRepository customerRepository)
    {
        _countryRepository = countryRepository;
        _customerRepository = customerRepository;
    }

    public void UpdateCustomer(CustomerDTO customerDTO)
    {
        var persisted = _customerRepository.Get(customerDTO.Id);

        if (persisted != null)
        {
            var current = MaterializeCustomerFromDto(customerDTO);
            customerRepository.Merge(persisted, current);
            customerRepository.UnitOfWork.Commit();
        }
        else
            LoggerFactory.CreateLog().LogWarning(Messages.warning_CannotUpdateNonExistingCustomer);
    }
}
```

← Injizierung der Repositories über IoC Container

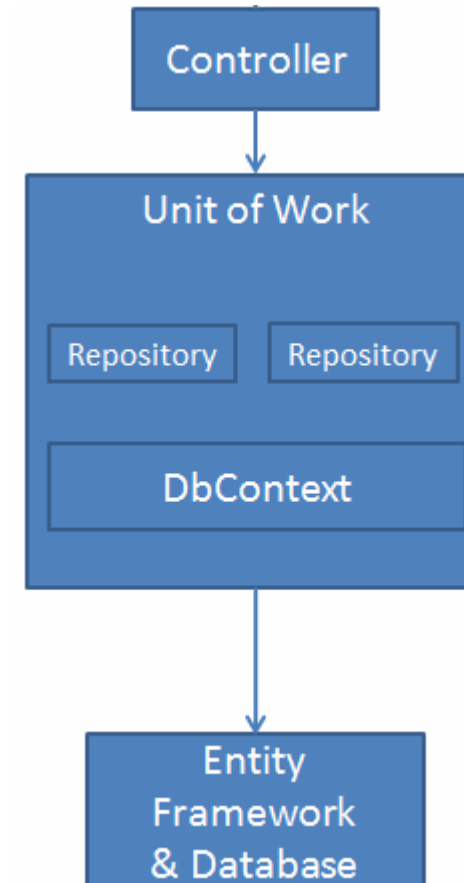
← Implizite und explizite Nutzung der *Unit of Work*

Unit of Work



Microsoft Empfehlung

Implementing the
Repository and Unit of Work Patterns
in an ASP.NET MVC Application (9 of 10)

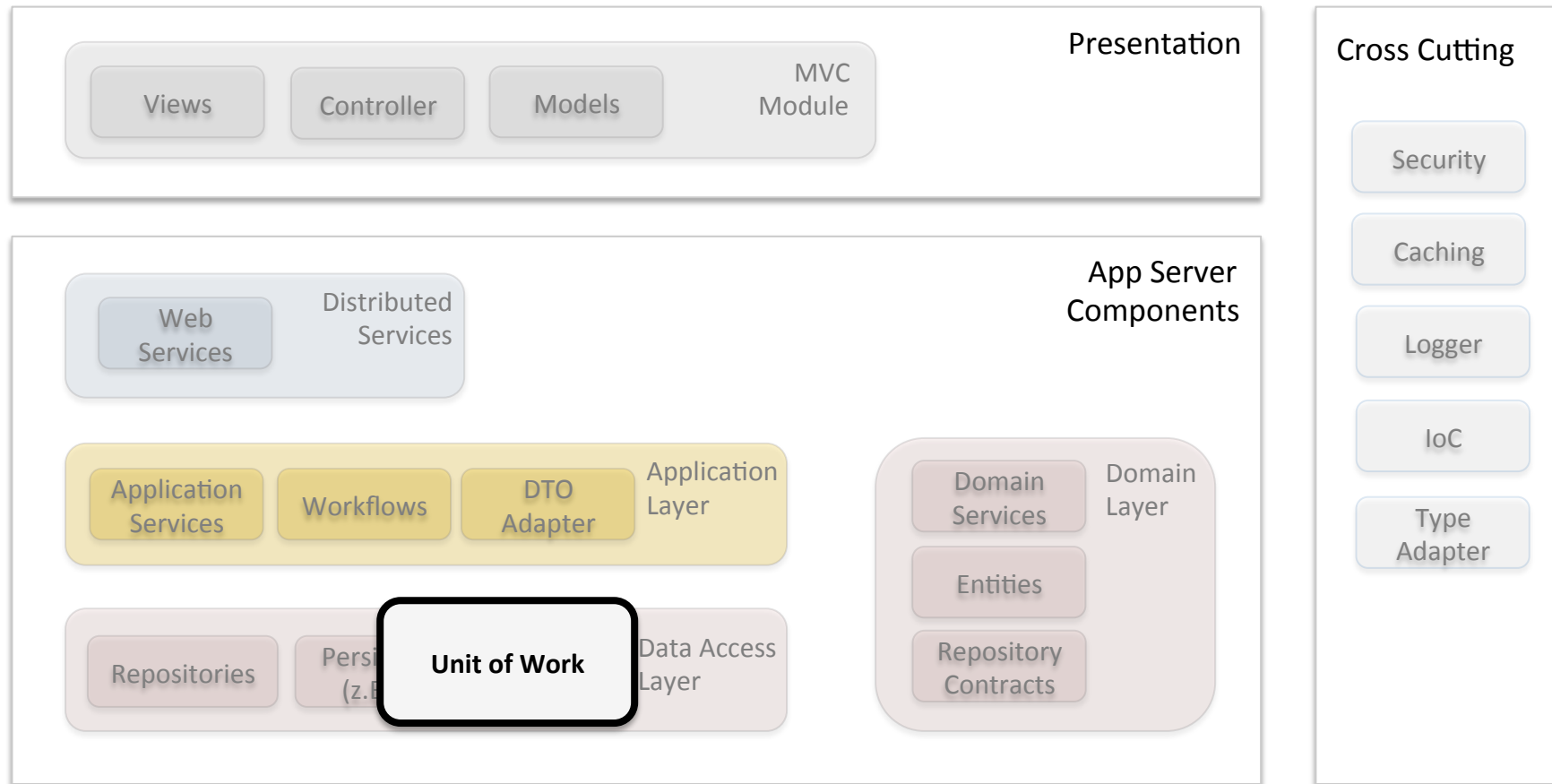


<http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>

Unit of Work



Anordnung im Architektur Model





Architecture Patterns

- Dependency Injection / Inversion Of Control
- Data Transfer Objects / Type Adapter
- Repository
- Unit Of Work
- **Specification**

Specification



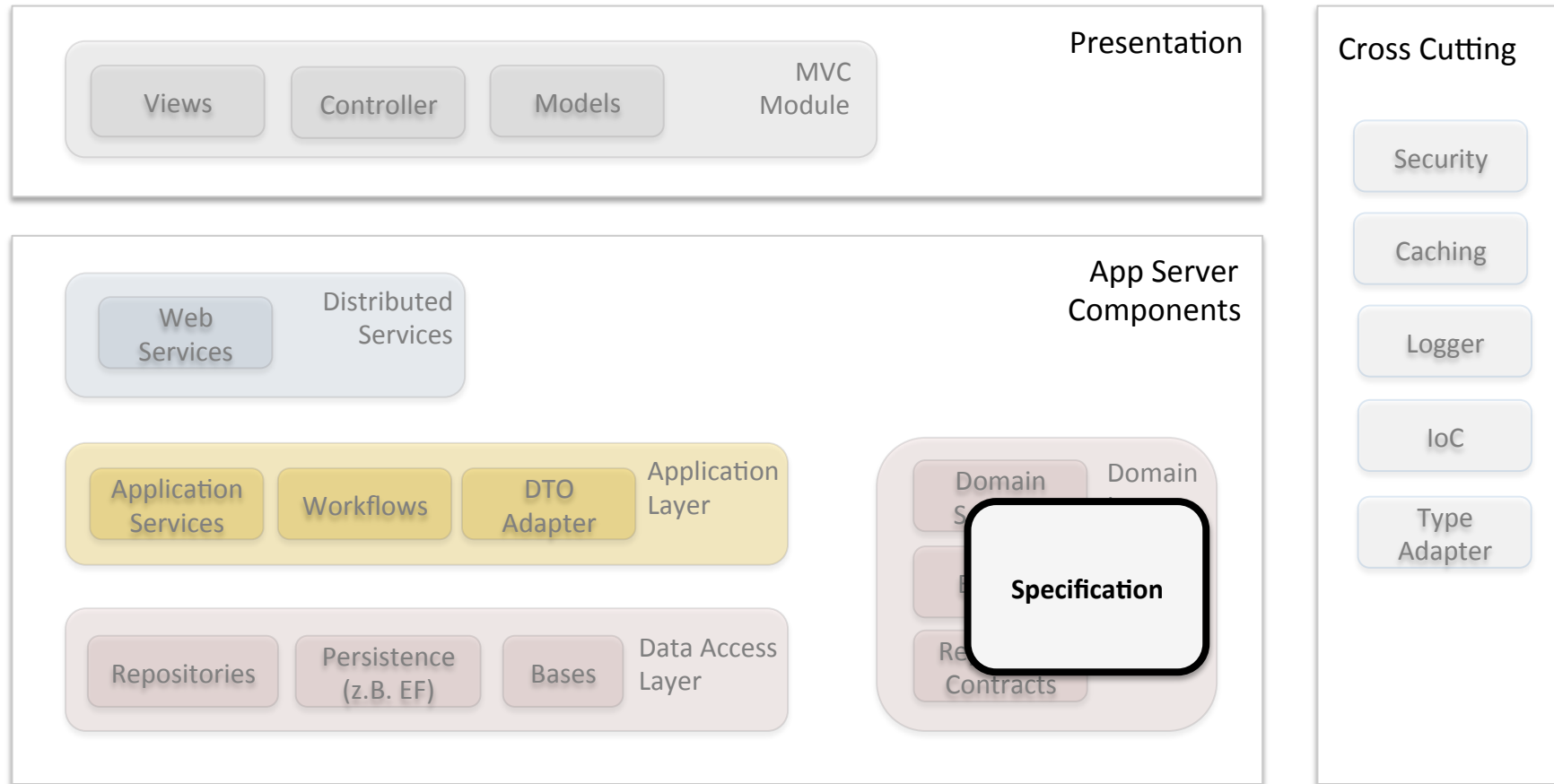
Ziele:

- Kapselung von Geschäftsregeln
- Entkopplung fachlicher Regeln von der Datenzugriffslogik
- Wiederverwendbarkeit
- Verknüpfung von Regeln über logischer Operatoren

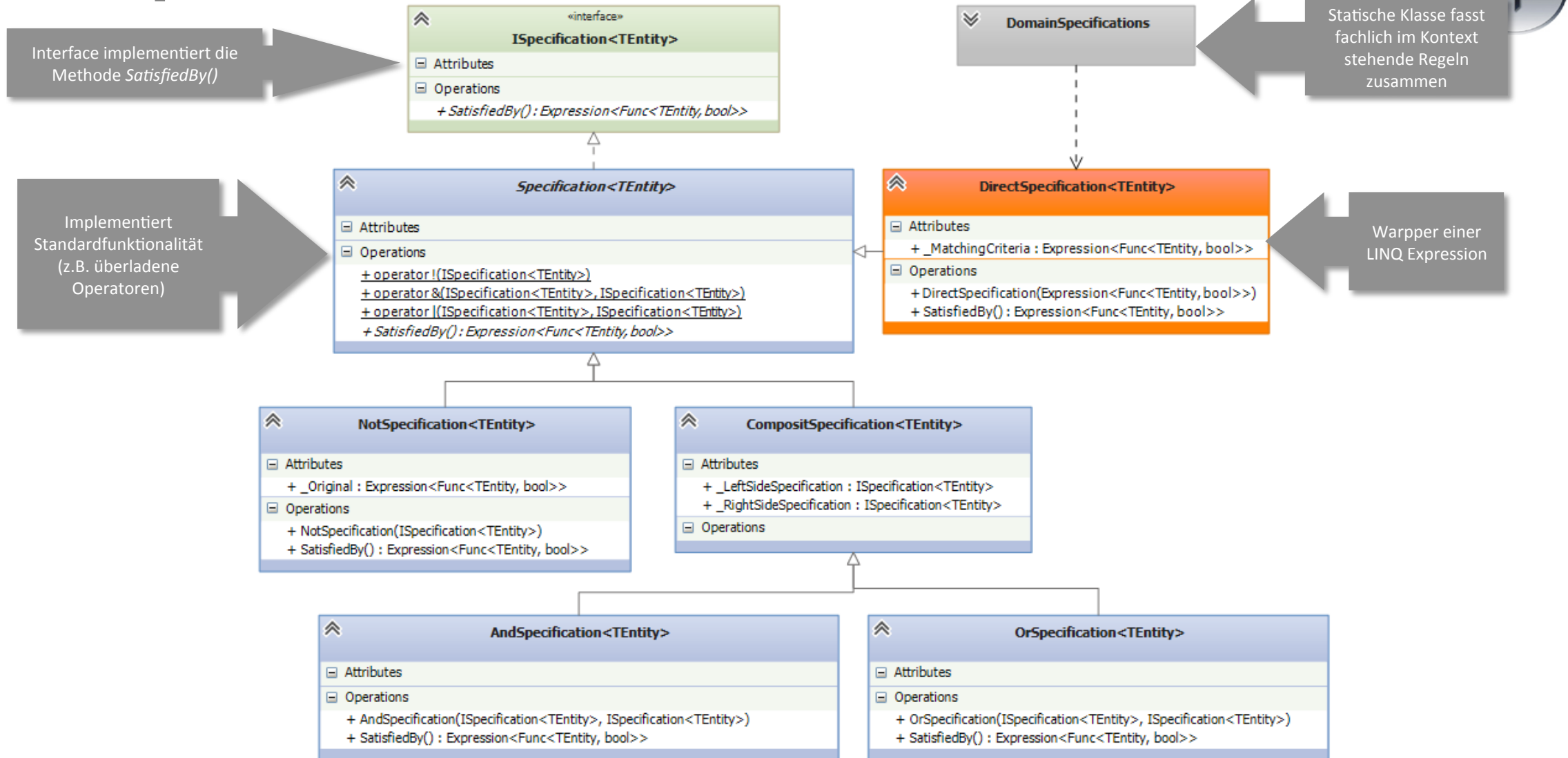
Specification



Anordnung im Architektur Model



Specification



Specification



Beispiel

```
public sealed class DirectSpecification<TEntity> : Specification<TEntity> where TEntity : Entity
{
    readonly Expression<Func<TEntity, bool>> _MatchingCriteria;

    public DirectSpecification(Expression<Func<TEntity, bool>> matchingCriteria)
    {
        _MatchingCriteria = matchingCriteria;
    }

    public override Expression<Func<TEntity, bool>> SatisfiedBy()
    {
        return _MatchingCriteria;
    }
}

public static class BankAccountSpecifications
{
    public static ISpecification<BankAccount> BankAccountIbanNumber(string ibanNumber)
    {
        return new DirectSpecification<BankAccount>(
            b => b.Active == true
        ) &=
            new DirectSpecification<BankAccount>(
                b => b.Iban.ToLower().Contains(ibanNumber)
            );
    }
}
```

Wrapper einer LINQ
Expression

Statische Klasse fasst mehrere im
fachlichen Kontext stehende Regeln
zusammen

Logische Verknüpfung

Specification

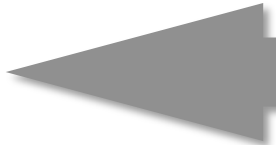


Beispiel

```
public class ApplicationService
{
    private readonly IDomainService _service;
    private readonly IBankRepository _repository;

    public ApplicationService(IDomainService service, IBankRepository repository)
    {
        _service = service;
        _repository = repository;
    }

    public IEnumerable<BankAccount> GetBankAccounts(int iban)
    {
        return _repository.GetAllMatching(BankAccountSpecifications.BankAccountIbanNumber(iban));
    }
}
```



Nutzung einer Specification zur
Ermittlung von Entitäten

Agenda



Einführung

N-Layer Domain-Oriented Architecture

Architecture Patterns

Visual Studio – Architecture Validation

Fazit

Visual Studio – Architecture Validation



DEMO

Agenda



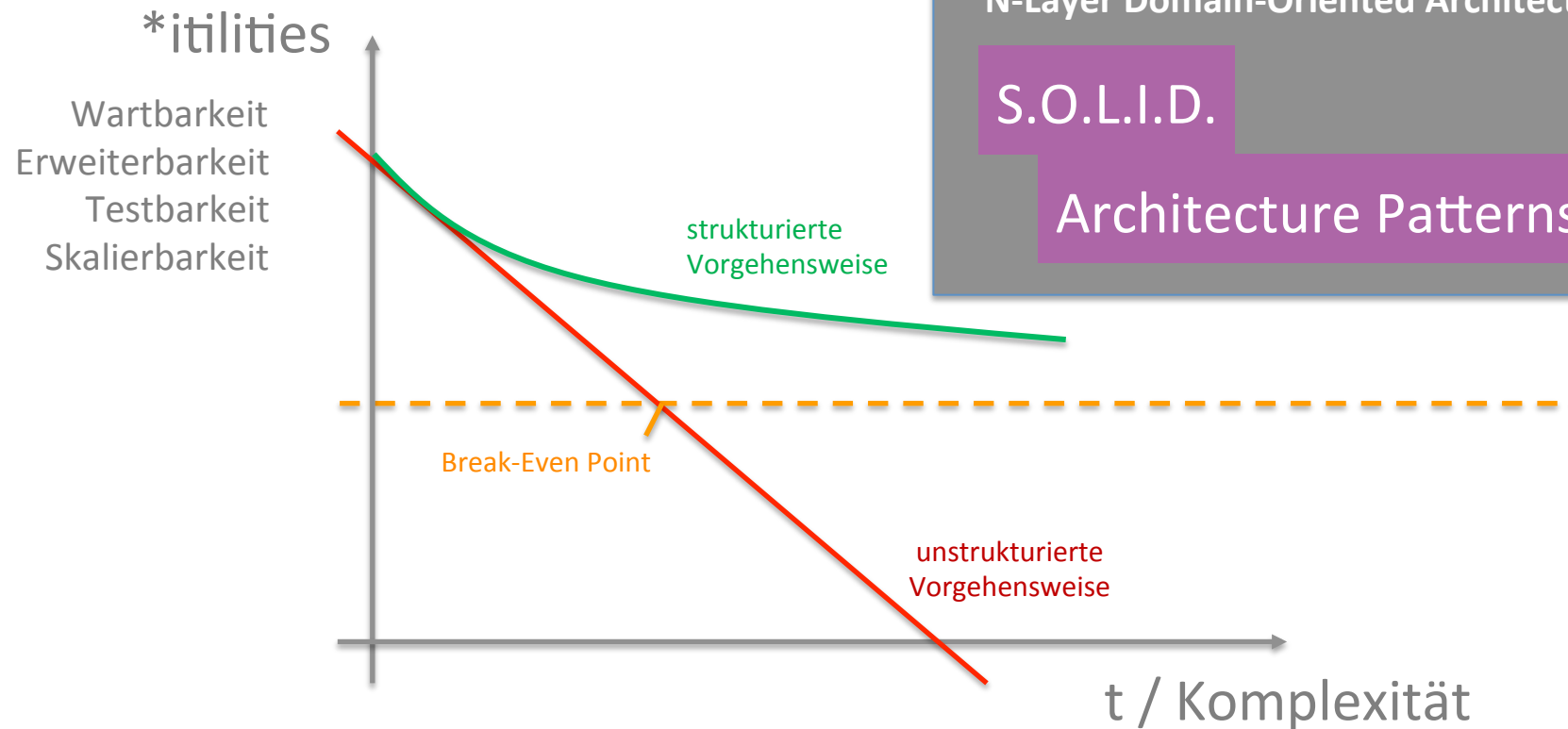
Einführung

N-Layer Domain-Oriented Architecture

Architecture Patterns

Visual Studio - Validation

Fazit



N-Layer Domain-Oriented Architecture Guide

S.O.L.I.D.

Architecture Patterns



Kritik

-  Teilweise unnötige Abstraktion hinsichtlich verwendeter Technologien (*UoW* vs *EF* / *Specification* vs *Lamdas*)
-  Realisierung *Repository* / *Unit of Work*
-  Hilft Zusammenhänge und Auswirkungen komplexer und flexibler Architekturen zu verstehen



Vielen Dank für Ihre Aufmerksamkeit !!!

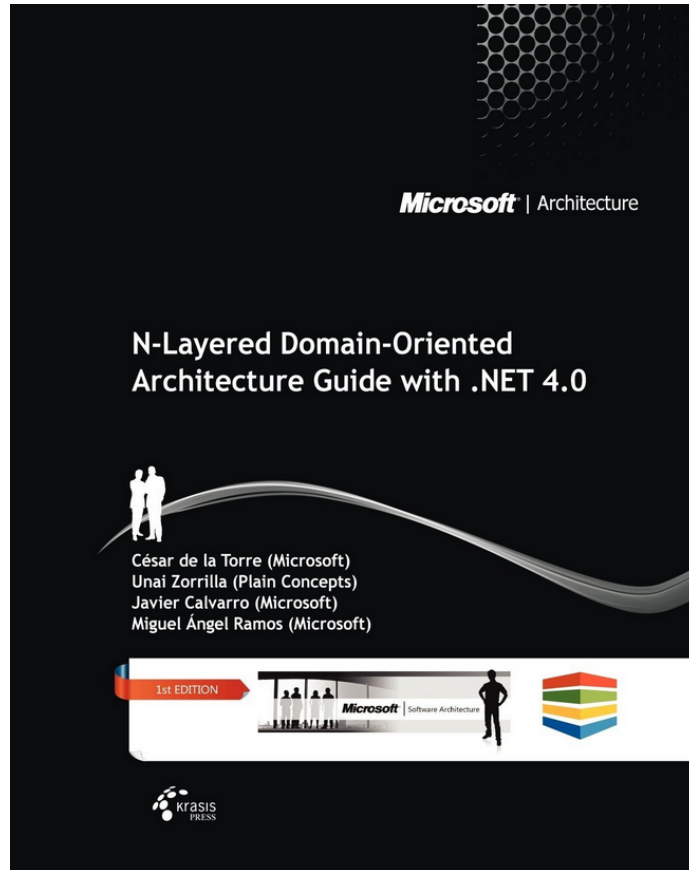
Tobias Krügel

Senior Software Engineer / Architect

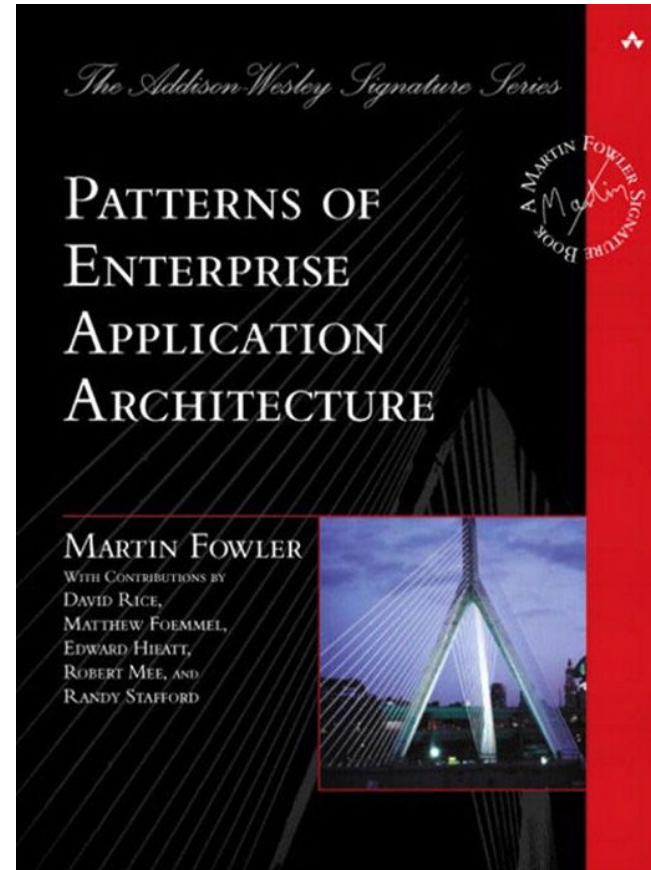
complement AG

tobias.kruegel@complement.de

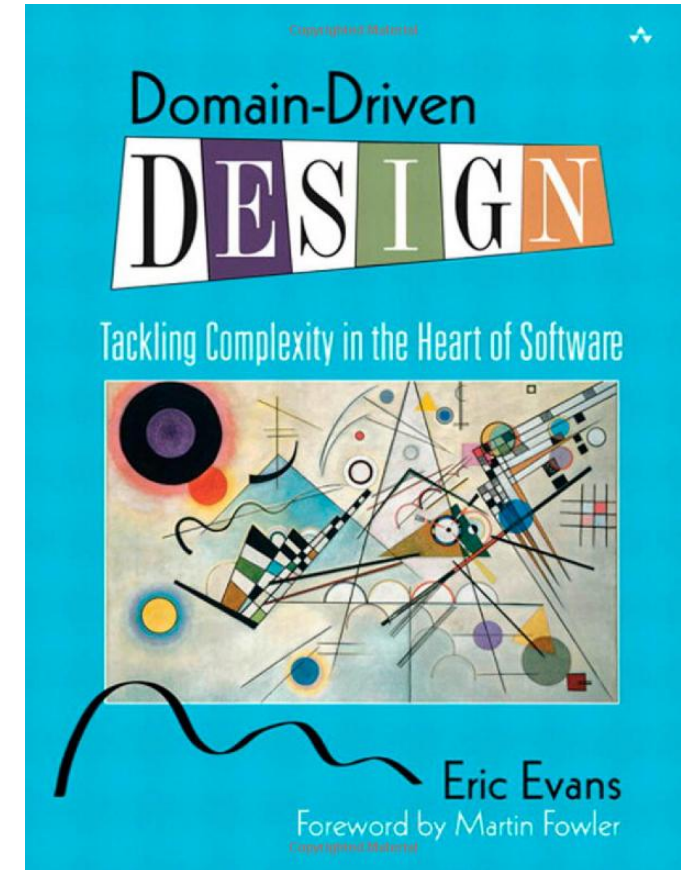




ISBN-13: 978-8493903619



ISBN-13: 978-0321127426



ISBN-13: 978-0321125217



Codeplex

<http://microsoftnlayerapp.codeplex.com>

<http://unity.codeplex.com>

<http://automapper.codeplex.com>