NOVATEC

Flexible Software Architecture

Understand and Apply Clean and Hexagonal Architecture

Matthias Eschhold

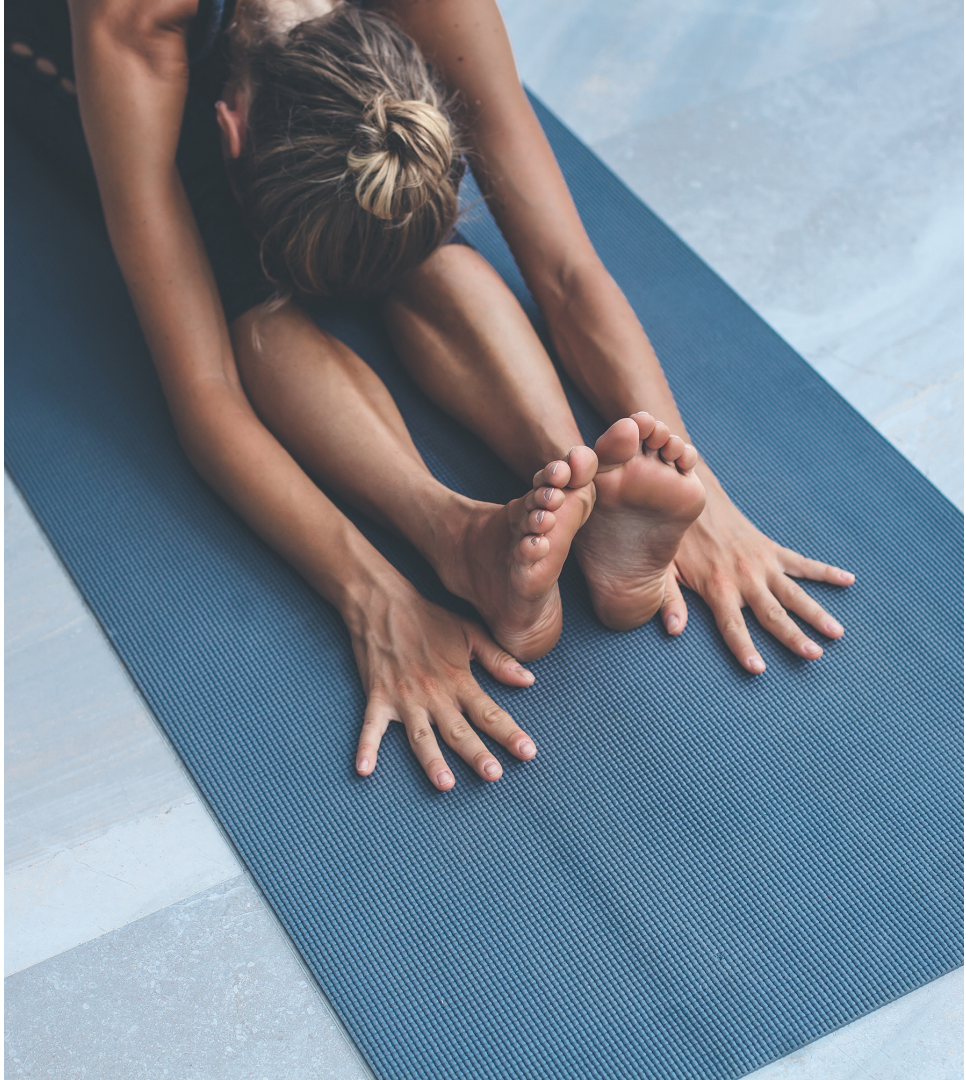
- Software Architect & Trainer & Enabler
- Domain-driven Design enthusiast
- Father of two girls (3 & 5 years old)
- Studied business information system in Ingolstadt
- Opponent of technical debt

iSAQB ACCREDITED Trainer

NOVATEC

Agenda

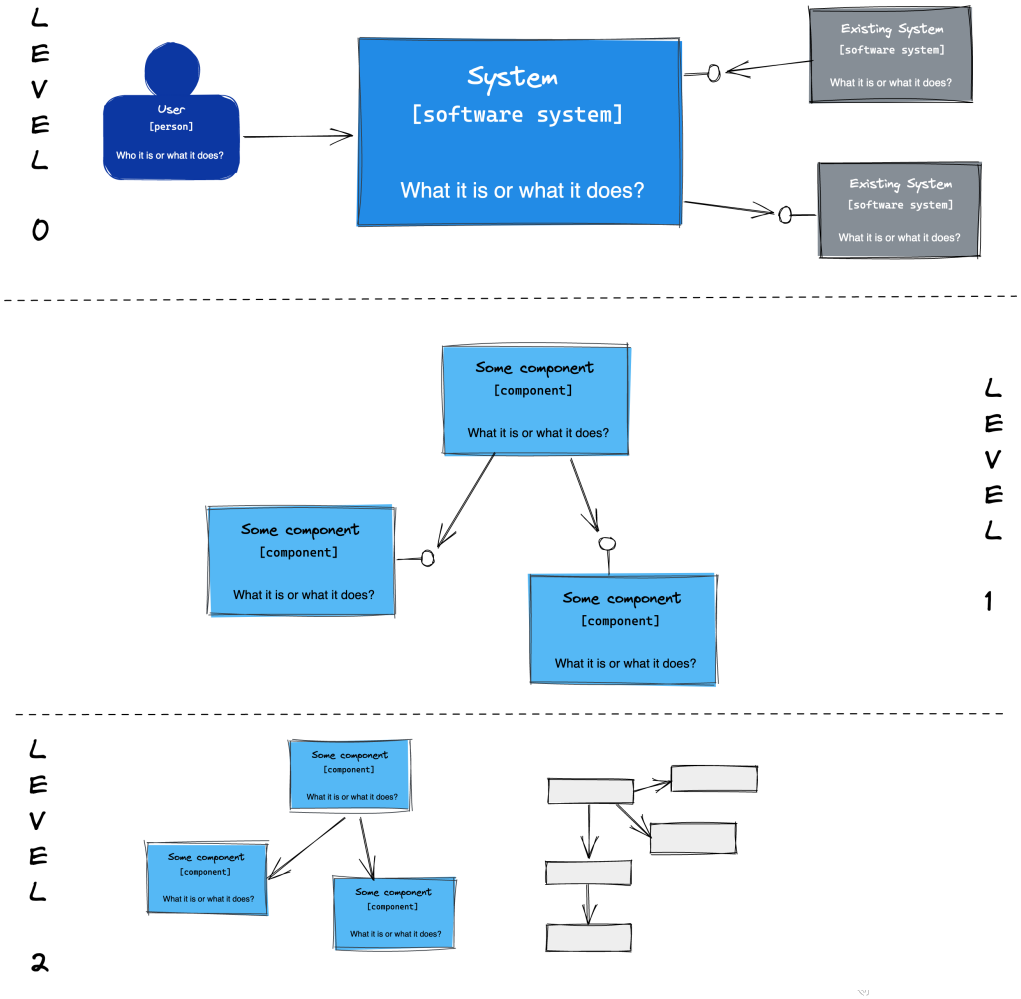
1. Software Architecture and Flexibility
2. Ports & Adapters Pattern
3. Hexagonal Architecture Pattern
4. Clean Architecture Pattern
5. Mappings Between Boundaries
6. Slicing Use Cases
7. Domain Modules and Dependencies Between
8. Evolutionary Architecture & Fitness Functions



Software Architecture

What is Software Architecture?

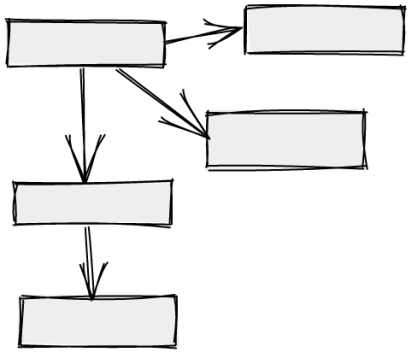
- Abstraction on different levels
- Blackbox and whitebox thinking
- Reduction of complexity

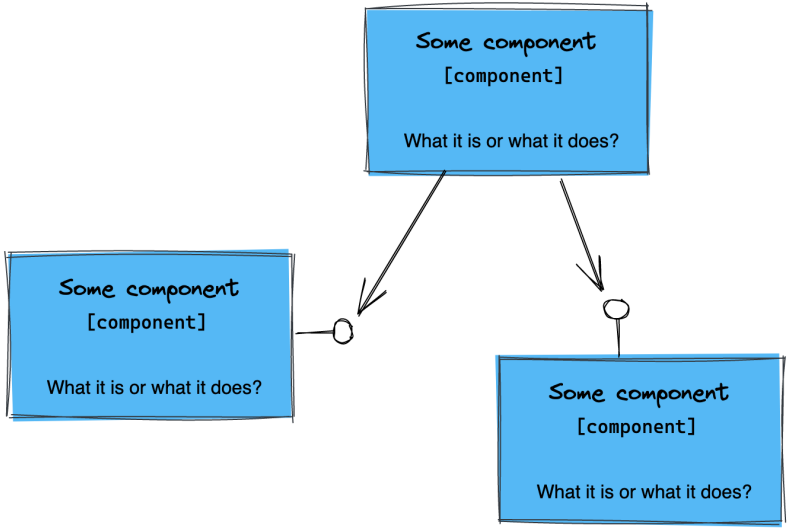


What is Software Architecture?

Architecture is the fundamental **organization** of a **system** embodied in its **components**, their **relationships** to each other, and to the **environment**, and the **principles guiding** its design and evolution.

[IEEE 1471]





NOVATEC

What is Software Architecture?

An architecture is the set of significant decisions about the **organization** of a software **system**, the selection of the **structural elements** and their **interfaces** by which the system is composed, together with their behavior as specified in the **collaborations** among those elements, the **composition** of these structural and behavioral elements into progressively larger **subsystems**, and the architectural style that guides this organization --- these elements and their interfaces, their collaborations, and their composition.

[The Rational Unified Process]

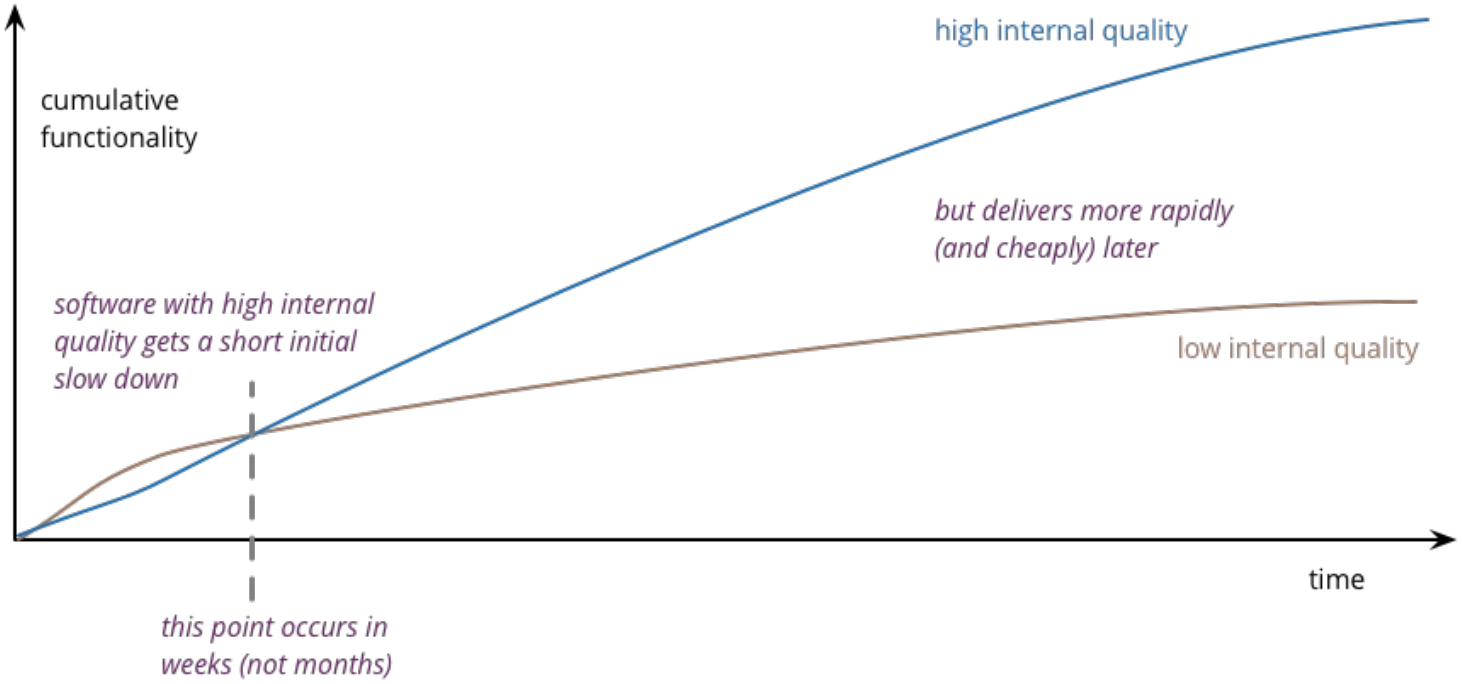
Software Architecture is the **sum of decisions** that are **hard to change later.** *[Stefan Toth]*

Hard to change means **time-consuming** and **cost-intensive**.

Time-consuming

Cost-intensive

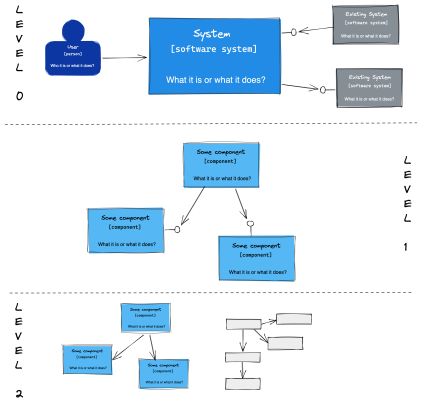
Why Does Software Architecture Matter?

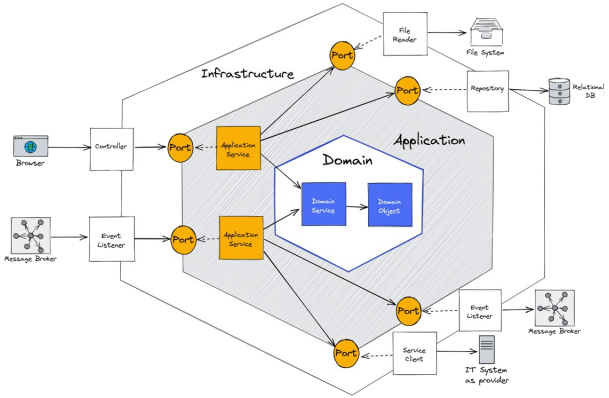


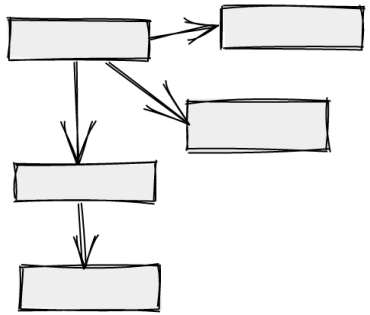
https://martinfowler.com/architecture/

NOVATEC

Three Pillars of Pragmatic Software Architecture







1 Functional Decomposition

2 Architecture Pattern

3 Pattern Language

NOVATEC

Flexibility as Software Quality Characteristic

Software Quality Characteristics Maintainability Based on ISO 25010

Maintainability

- refers to how well a **system** can be **modified** to **improve**, **correct**, or **adapt to changes** in the environment as well as requirements.

Modularity

The degree to which a **system is composed** of discrete **components** such that **a change to one component** has **minimal impact** on **other components**

Analyzability

The degree of **effectiveness** and **efficiency** with which it is possible to **assess the impact on** a **system** of an **intended change** […]

Modifiability

The degree to which a **system can be** effectively and efficiently **modified without introducing defects** or **degrading** existing **quality**

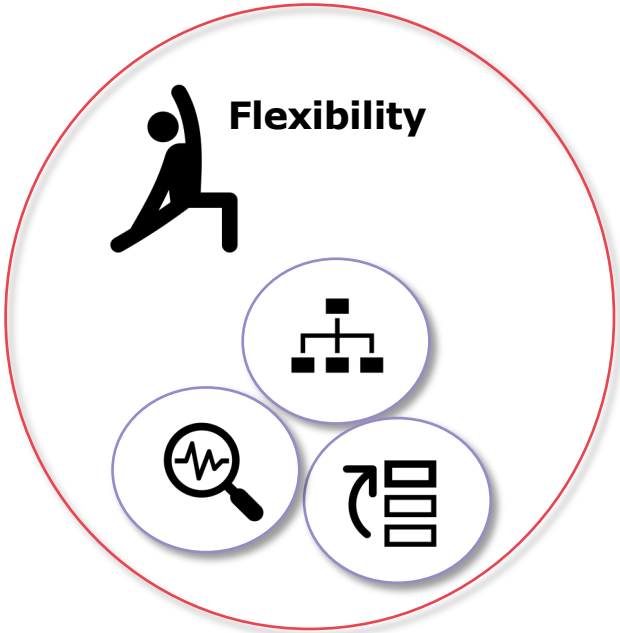
https://blog.codacy.com/iso-25010-software-quality-model/

https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=6

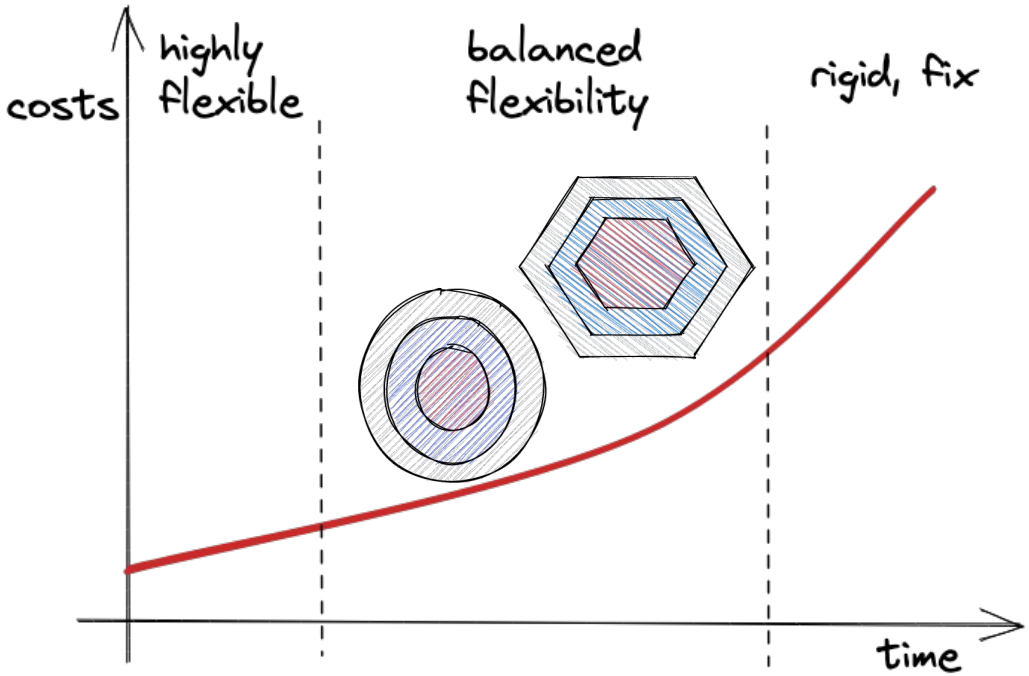
Flexibility as Software Quality Characteristic

- A flexible software architecture is **always prepared** for **modifications** and **designed** to **adapt fast**

- **Flexibility** is the **capability** of a system to **develop** from a **current state** to a **new state** based on changing
  - functional requirements,
  - quality requirements or
  - infrastructure



The Degree of Flexibility is Determined Based on Time and Costs



degree of flexibility: the lower the better

Interface Segregation Principle

Dependency Inversion Principle

Dependency Injections

Ports & Adapters

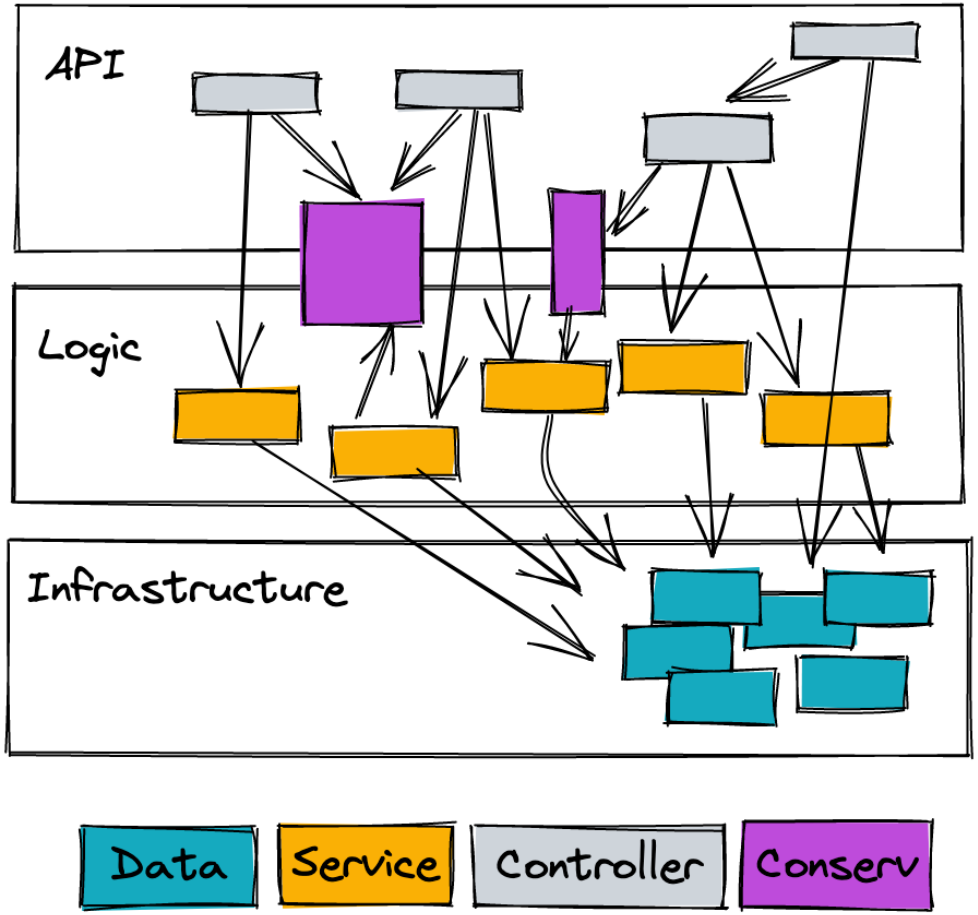
Domain Modules

Root Entity

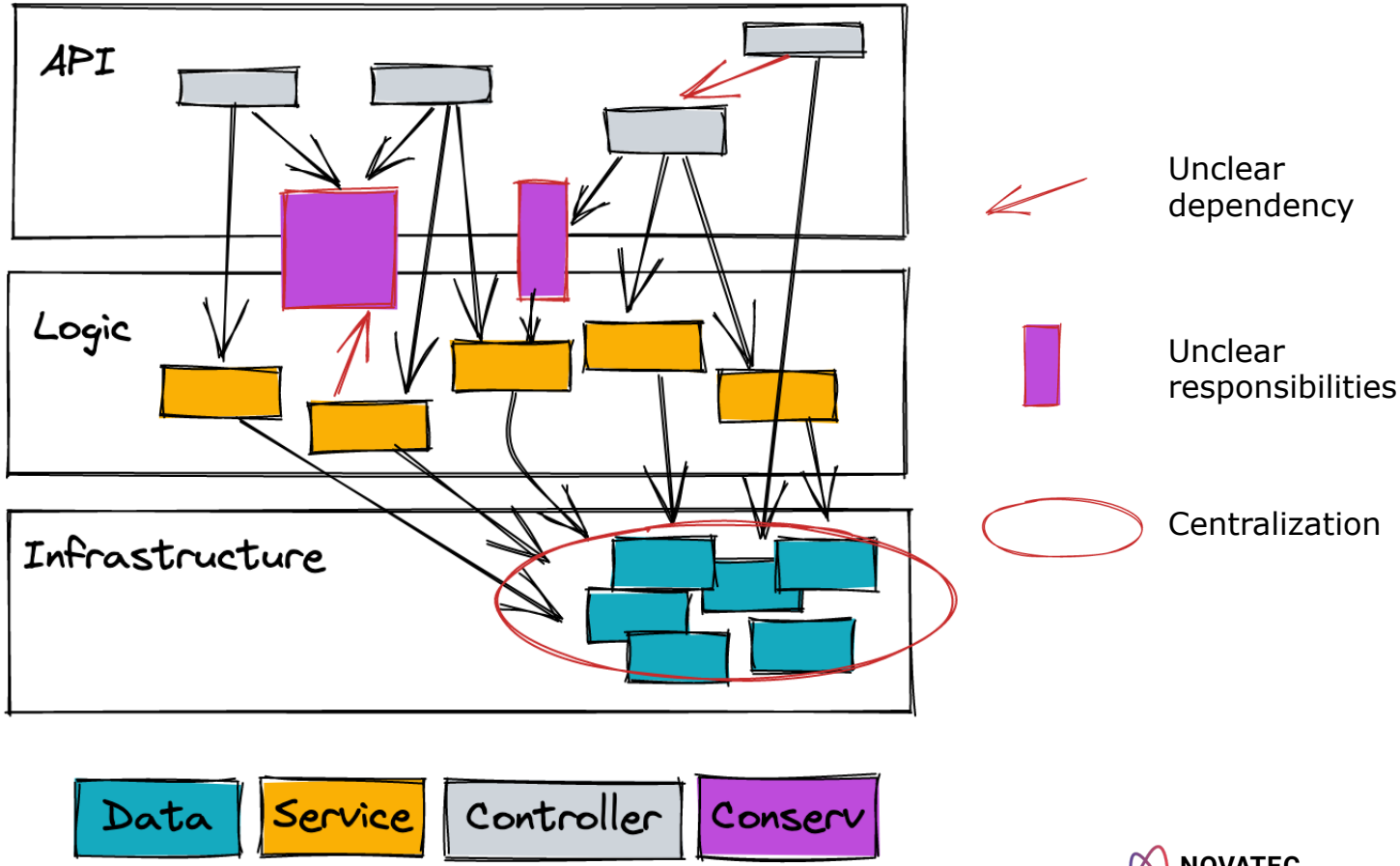
NOVATEC

The Problems with Layered Architecture

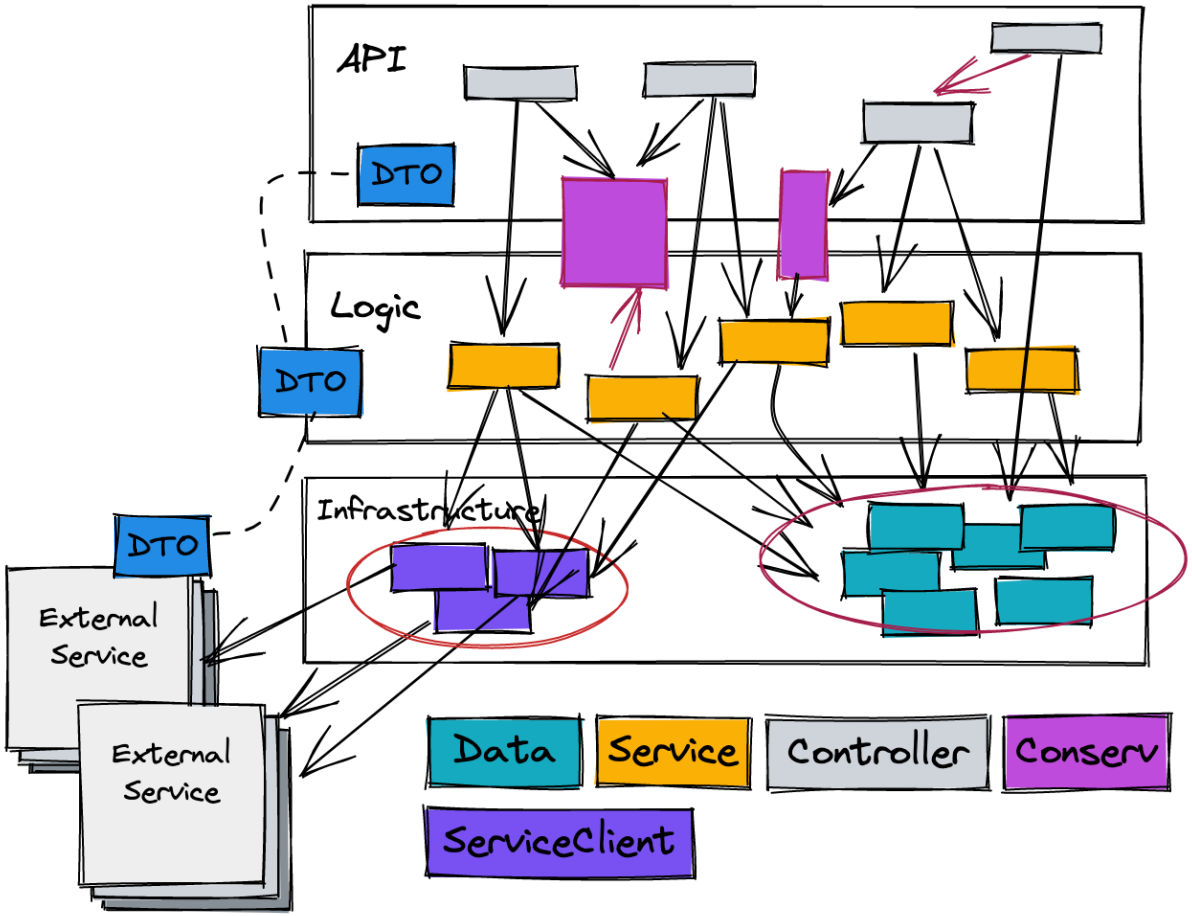
Missing separation of domain and technical aspects



NOVATEC


API
Logic
Infrastructure
Unclear dependency
Unclear responsibilities
Centralization
Data
Service
Controller
Conserv

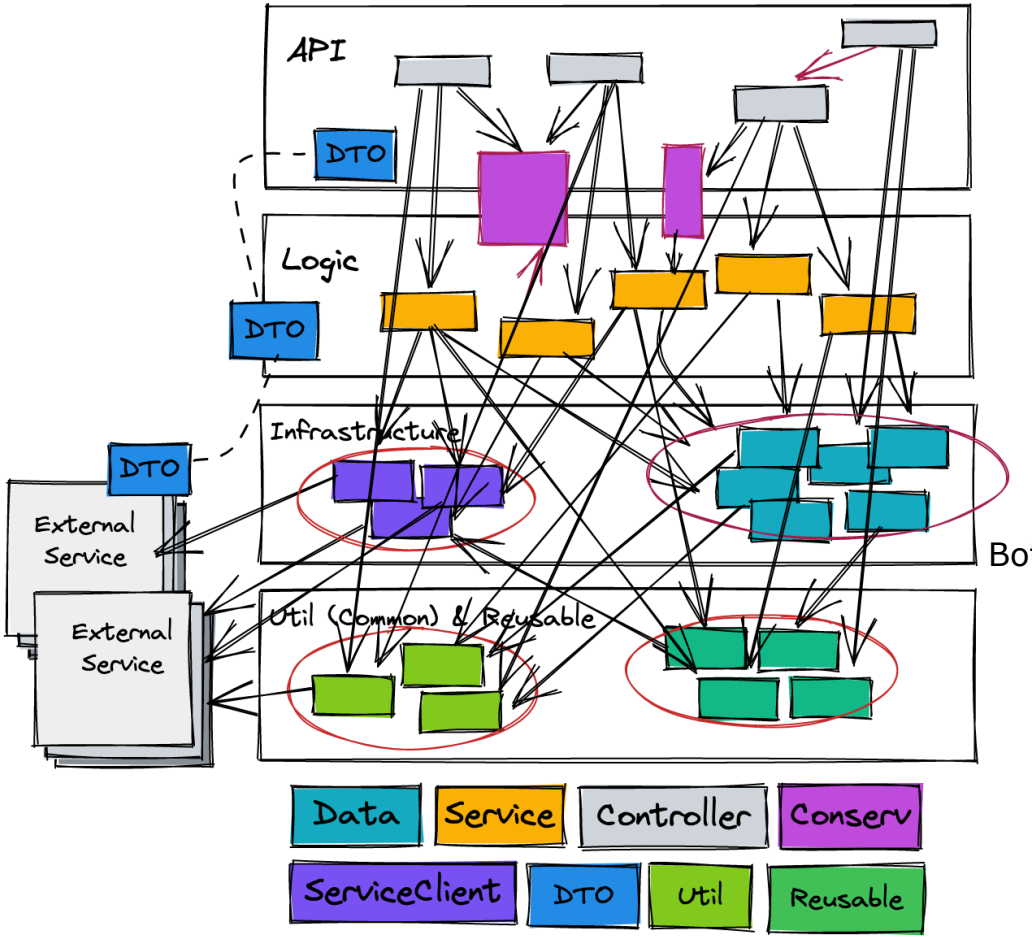
NOVATEC



DTO

Direct usage of external models inside our domain

NOVATEC



Bottom-most layer phenomenon which causes coupling and side effects

NOVATEC

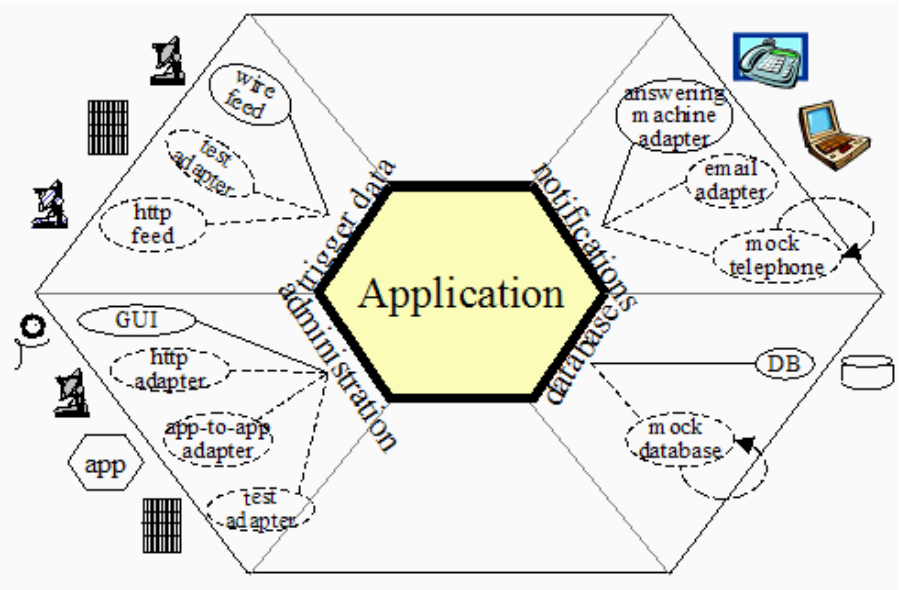
Alistair Cockburn's Ports & Adapters Pattern

Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

- Cockburn 2005

Pattern Name
Ports & Adapter

Working Title
Hexagonal Architecture



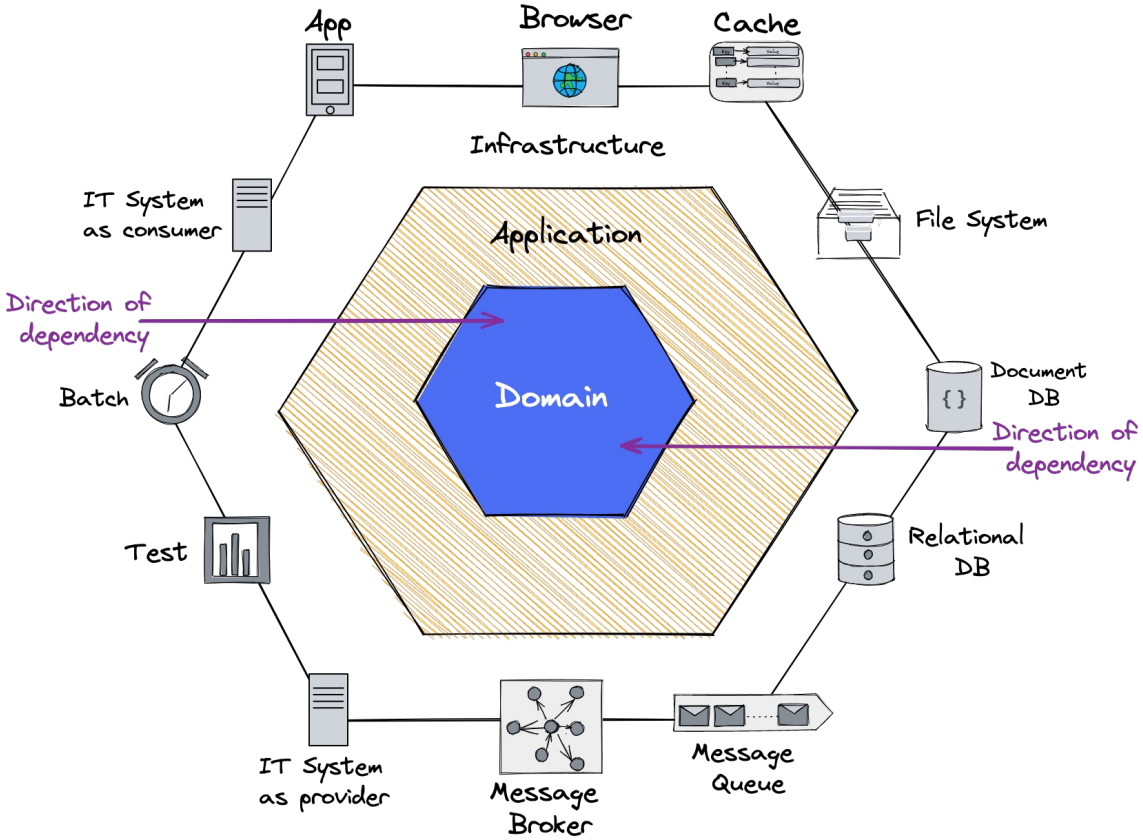
https://alistair.cockburn.us/hexagonal-architecture/

NOVATEC

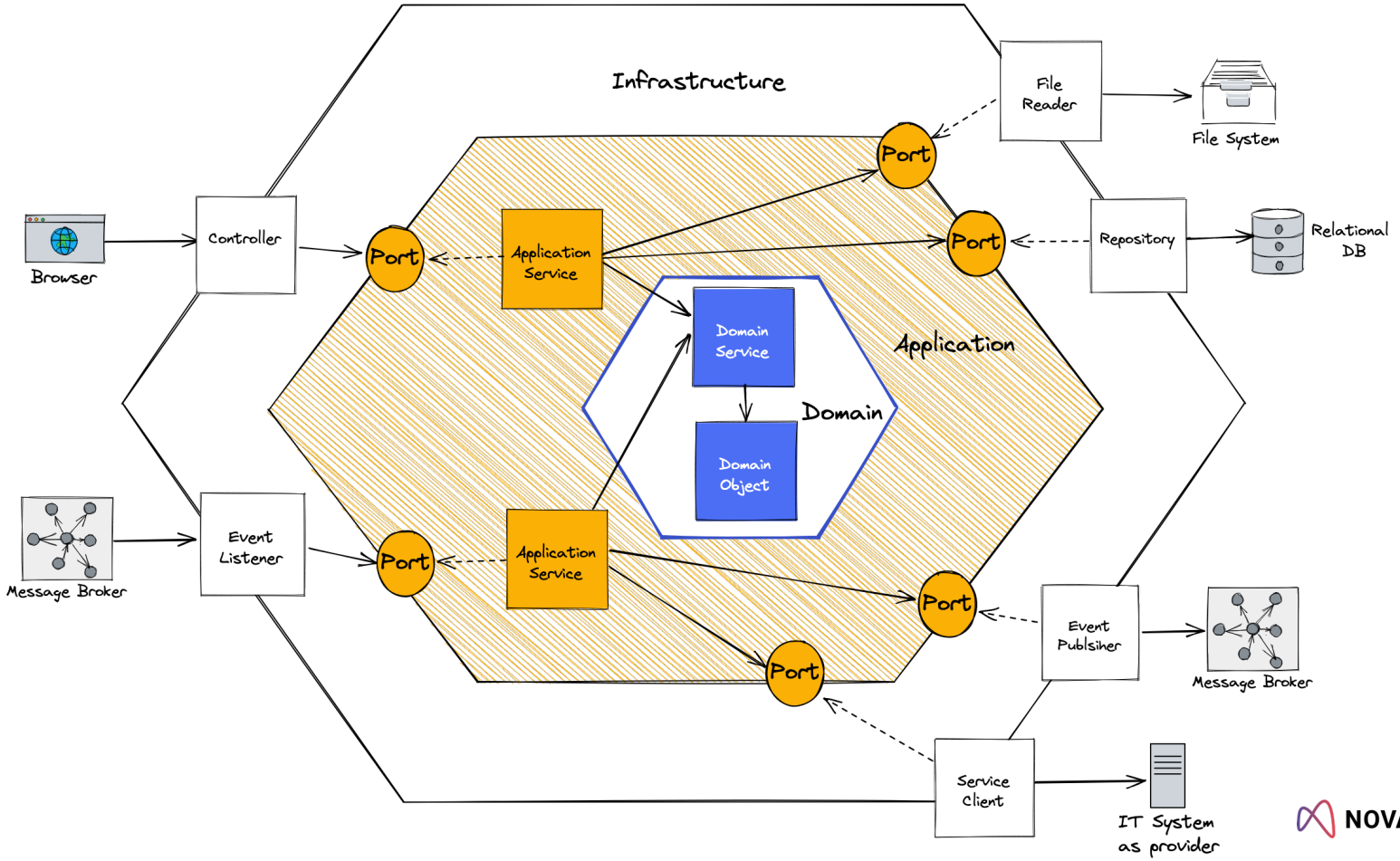
Hexagonal Architecture

According to Michael Plöd[2]

Hexagonal Architecture Pattern



NOVATEC


Infrastructure
File Reader
File System
Port
Browser
Controller
Port
Application Service
Port
Repository
Relational DB
Domain Service
Application
Domain
Domain Object
Message Broker
Event Listener
Port
Application Service
Port
Port
Event Publsiher
Message Broker
Service Client
IT System as provider

NOVATEC

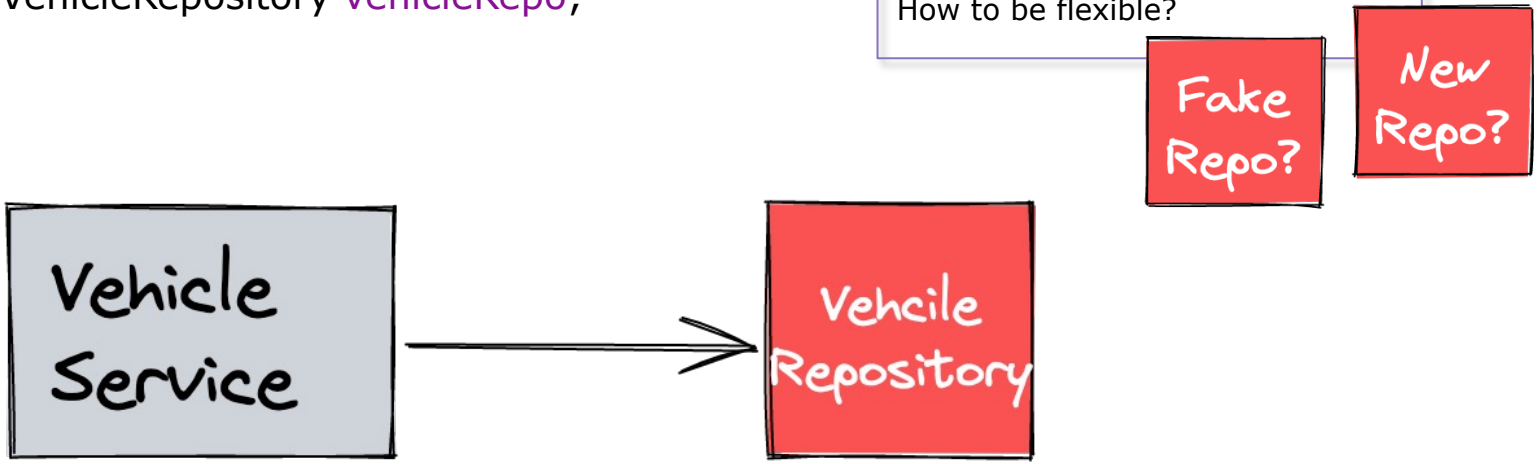
Hierarchical Dependency Structure

```
import de.vehicle.infra.db.VehicleRepository;

public class VehicleService {

    private VehicleRepository vehicleRepo;

    //...
}
```

How to add a fake implementation?

How to modernize based on a new implementation?

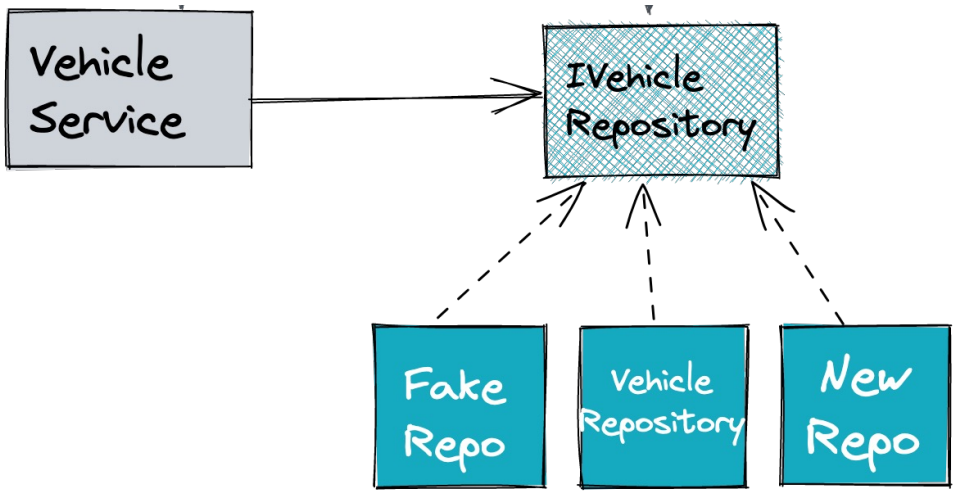
How to be flexible?



Dependency Inversion Principle

```
import de.vehicle.infra.db.IVehicleRepository;

public class VehicleService {

    private IVehicleRepository vehicleRepo;

    //...
}
```

Abstraction **should not depend** on details.
Details **should depend** on abstractions.



NOVATEC

Dependency Inversion Principle

```
import de.vehicle.infra.db.IVehicleRepository;
import de.vehicle.infra.db.VehicleRepository;

public class VehicleService {

    private IVehicleRepository vehicleRepo;


    public VehicleService() {
        this.vehicleRepo = new VehicleRepository();
    }


}
```

This is not working!
The dependency **is not inverted!**

Dependency Inversion Principle

```
import de.vehicle.infra.db.IVehicleRepository;

public class VehicleService {

    private IVehicleRepository vehicleRepo;

   @Inject
    public VehicleService(IVehicleRepository vehicleRepo) {
          this.vehicleRepo = vehicleRepo;
    }

    //...
}
```

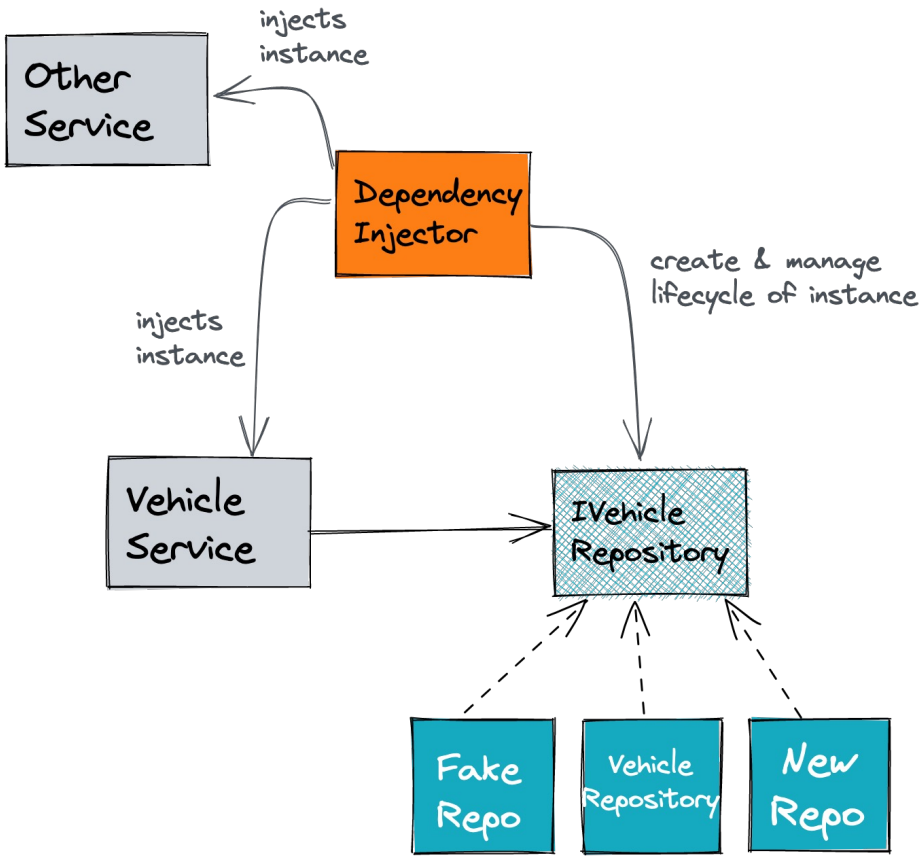
*Inversion of Control is too generic a term, and thus people find it confusing. As a result with a lot of discussion with various [Inversion of Control] advocates we settled on the name **Dependency Injection**.*

- Fowler [7]

But only the combination of the idea of dependency inversion and the pattern of dependency injection makes the inversion of the dependency possible.

Dependency Injection

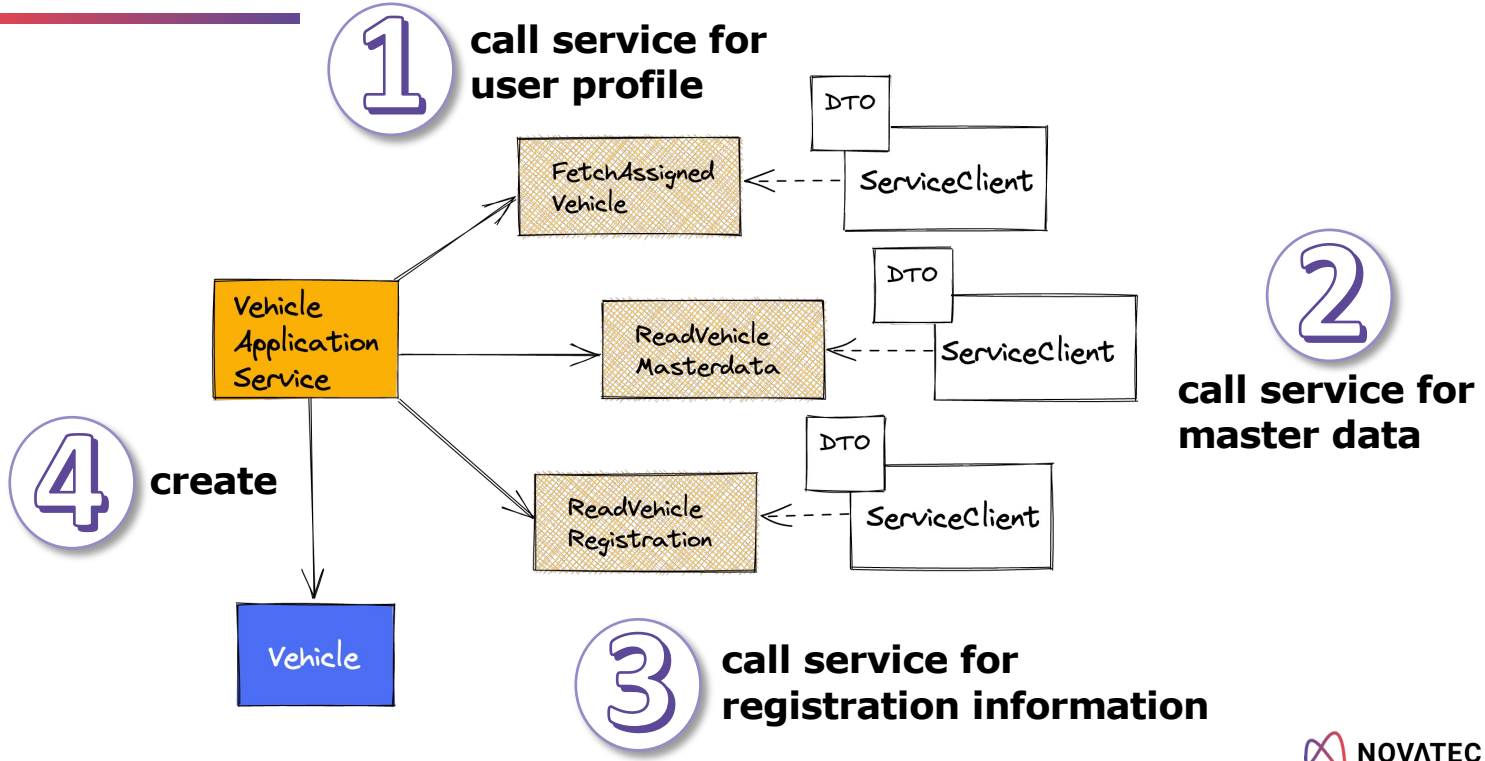
A central component, the dependency injector, manages the lifecycle of class instances and injects these instances into several consumers.



Types of dependency injection:

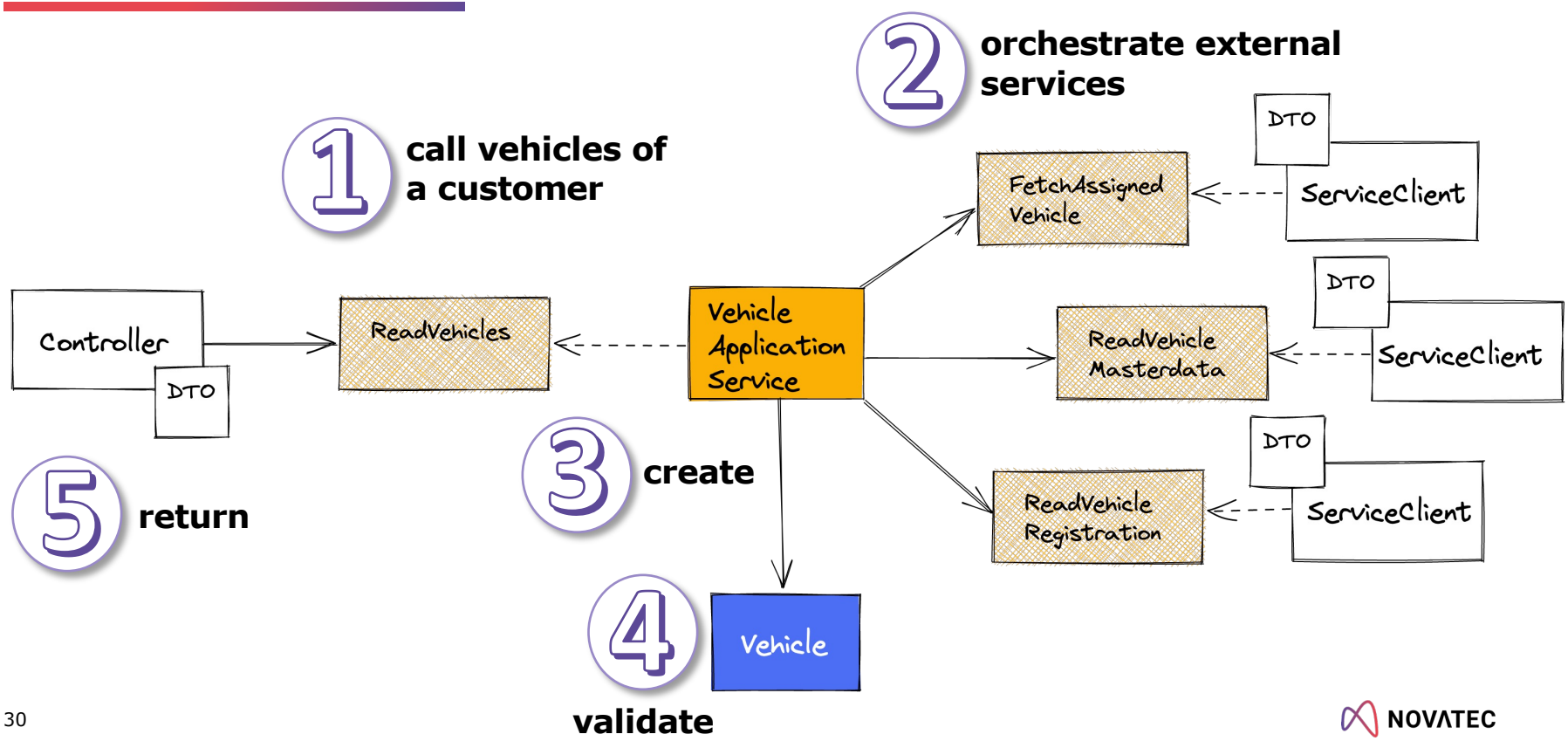
- Constructor Injection
- Setter Injection
- Field Injection
- Method Injection

Dependency Inversion within Hexagonal Architecture



NOVATEC

Flow Through the Hexagon



NOVATEC

Domain Model

Based on the philosophy of domain-driven design

Entities Should Ensure Validity

Validation of Domain Values

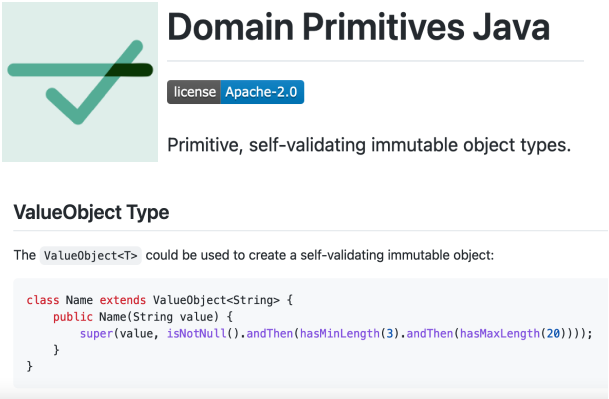
- Vehicle Identification Number
- WVWZZZ1JZ3W386752 or W0L000051T2123456

```
public class Vin {

    private String value;

    public Vin(String value) {
        this.value = value;

        if(!value.matches("(?=.*\\d|=.*[A-Z])(?=.*[A-Z])[A-Z0-9]{17}")) {
            throw new ValidationException("Vin", "is not valid!");
        }

    }
}
```

Self-Validating Domain Objects

Validation of Domain Values

```
import io.github.domainprimitives.type.ValueObject;
import static io.github.domainprimitives.validation.Constraints.isPattern;

public class Vin extends ValueObject<String> {

    public Vin(String value) {
        super(value, isPattern("(?=.*\\d|=.*[A-Z])(?=.*[A-Z])[A-Z0-9]{17}"));
    }

}
```



Domain Primitives Java

license Apache-2.0

Primitive, self-validating immutable object types.

ValueObject Type

The `ValueObject<T>` could be used to create a self-validating immutable object:

```
class Name extends ValueObject<String> {
    public Name(String value) {
        super(value, isNotNull().andThen(hasMinLength(3).andThen(hasMaxLength(20))));
    }
}
```

https://github.com/domain-primitives/domain-primitives-java

NOVATEC

Entities Should Contain Behavior That Can Be Realized by Themself

Calculations & Detections

```
public class GarageOrder {

    private List<VehicleService> vehicleServices;
    //...

    public TotalTimeRequired calucateTotalTimeRequired() {

      TotalTimeRequired totalTimeRequired = new TotalTimeRequired();

      this.vehicleService.forEach(vehicleService ->
        time.addTimeRequired(vehicleService.getTimeRequired());

      return totalTimeRequired;
    }
}
```

When Consumers Calls the Domain with Invalid Domain Values

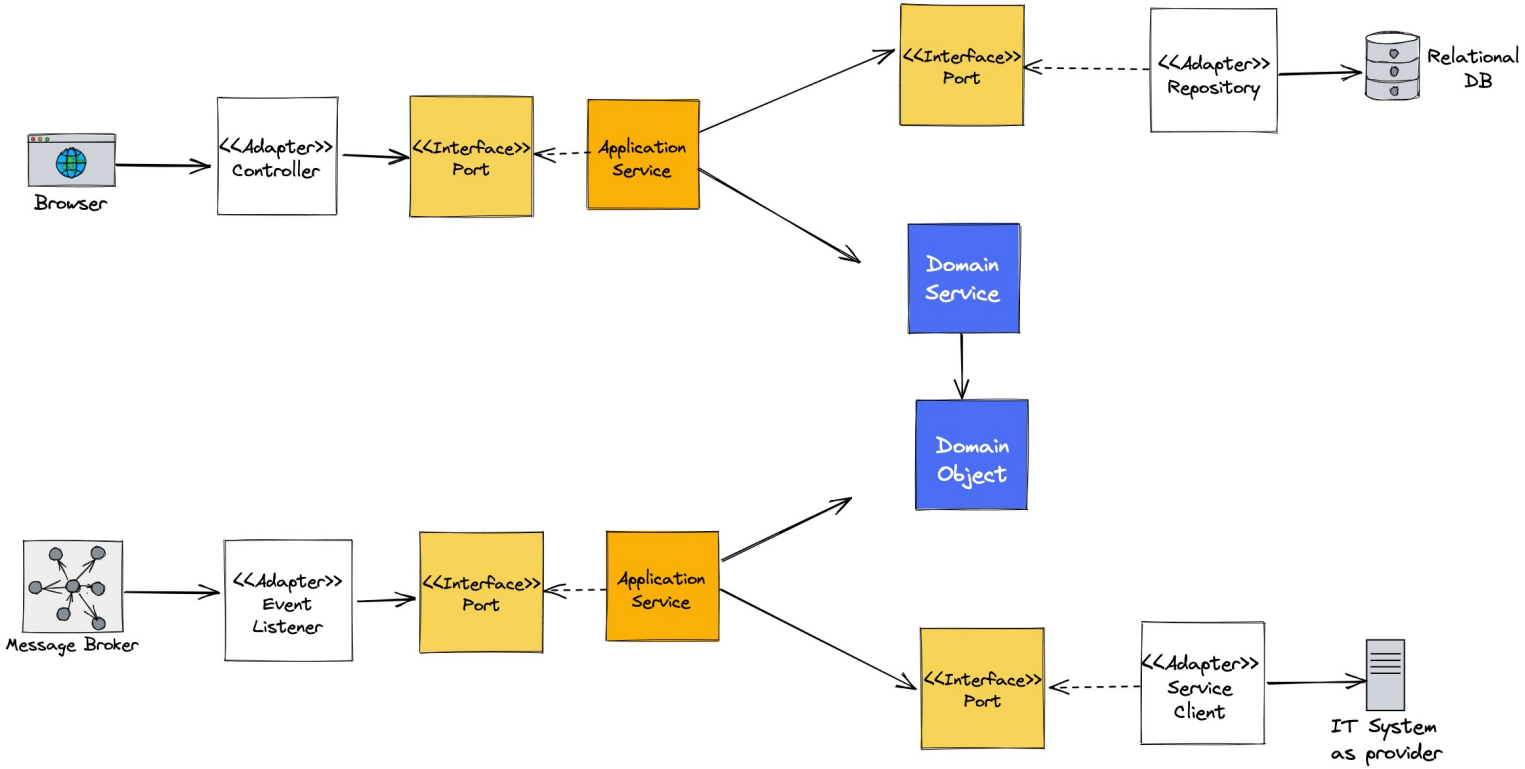
```
@GetMapping("/{vin}")
ResponseEntity<VehicleResource> readVehicle(@RequestParam(value = "vin") String vin)
{
    return ResponseEntity
            .ok()
            .body(mapper.mapToDto(
                    vehicleQuery.getVehicleByVin(new Vin(vin))
            ));
}
```

Early validation with meaningful message!

Domain Service and Application Service

Based on the philosophy of domain-driven design enthusiasts

Pattern Language for Hexagonal Architecture



NOVATEC

Domain Services Contain Domain Logic That is not Assignable to an Entity and Nothing Else As The Domain Model is Needed

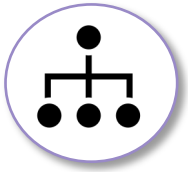
- To detect the *sales channel color* of a *vehicle*, the *vehicle dealer id* is important
- But the *sales channel color* is not part of the *vehicle* domain model

Calculation & Detections

```
public DetectSalesChannelColorService {

    public SalesChannelColor detect(Vehicle vehicle) {

        if(vehicle.getDealerId().startsWith("E") {
            return SalesChannelColor.RED;
        }

        return SalesChannelColor.BLUE;
    }

}
```

Application Services Connect All Together

- Connect input adapters with the domain core by implementing input ports
- Orchestration of output adapters

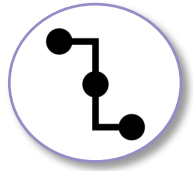


Connect & Orchestrate

```
public class VehicleApplicationService implements VehicleQuery {

  //...

  public List<Vehicle> getVehicles(CustomerId customerId) {

    List<Vehicle> pairedVehicles = pairedVehiclesQuery
      .fetchVehiclesByCustomerId(customerId);

    for (Vehicle vehicle : pairedVehicles) {
      Vin vin = vehicle.getVin();

        vehicle.addVehicleMasterdata(
          vehicleMasterdataQuery.getVehicleByVin(vin));

        vehicle.addVehicleRegistration(
          vehicleRegistrationQuery.getVehicleRegistration(vin));
    }

    return pairedVehicles;
  }
}
```

Application Services Connect All Together

- Delegate execution of domain logic to domain services or entities
- Solve domain-related dependencies between root entities



Connect & Delegate

```
public class GarageOrderApplicationService implements GarageOrderCommand {

    private final ReadGarageOrder readGarageOrder;
    private final CalculateWorkingTime calculateWorkingTime;
    private final SaveGarageOrder saveGarageOrder;
    private final VehicleQuery vehicleQuery;

    public GarageOrder addVehicleServices(GarageOrderId garageOrderId, List<VehicleService> services) {
      GarageOrder order = readGarageOrder.findOrderById(garageOrderId);
      String vin = garageOrder.getVin();

       vehicleServices =  calculateWorkingTime.calculate(services, vin, vehicleQuery.findByVin(vin).getEquipment());

       order.addVehicleServices(vehicleServices);

       order.calculateTotalTimeRequired();

      return saveGarageOrder.save(order);
    }
}
```

Characteristics of a Pattern Language

- It defines
  - a clear responsibility for each class stereotype
  - dependency rules between class stereotypes

- Helps to
  - avoid cycles because a pattern language follows a uni-directional hierarchy
  - ensure conceptual integrity between domain modules
  - establish a common architecture understanding within the developer team

- Applies good-practices like
  - Single Responsibility Principle
  - Common Closure Principle
  - Information Hiding & Encapsulation

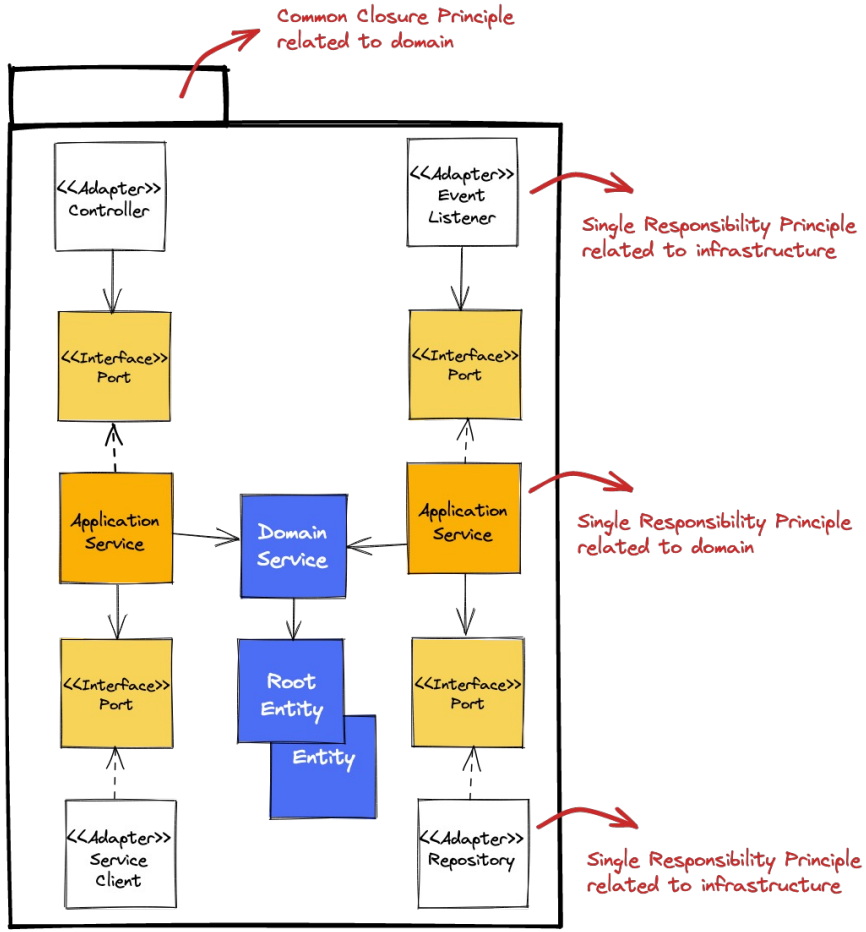
Applied Architecture Principles

- **Common Closure Principle**
  Pack together what changes together

- **Single Responsibility Principle**
  One reason to change

- **Dependency Inversion Principle**
  Invert the dependency between classes

- **Separation of technical and domain-related aspects**



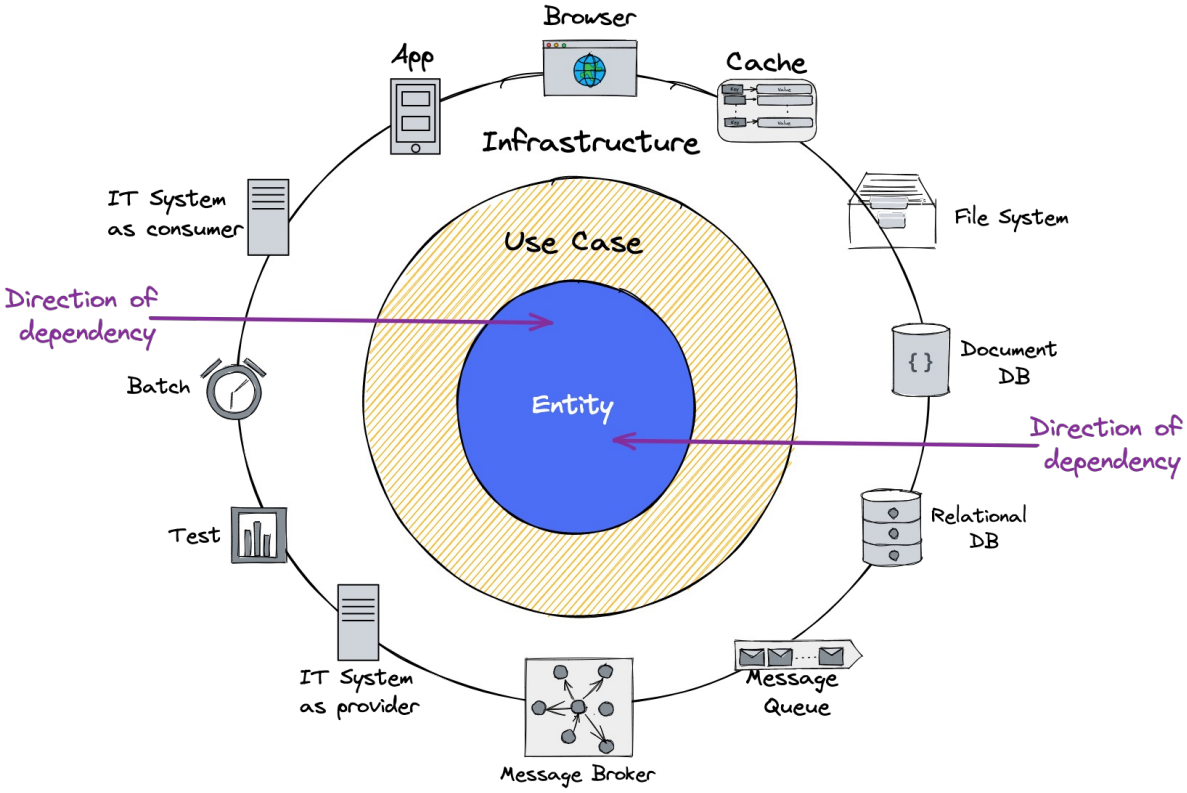
NOVATEC

Clean Architecture

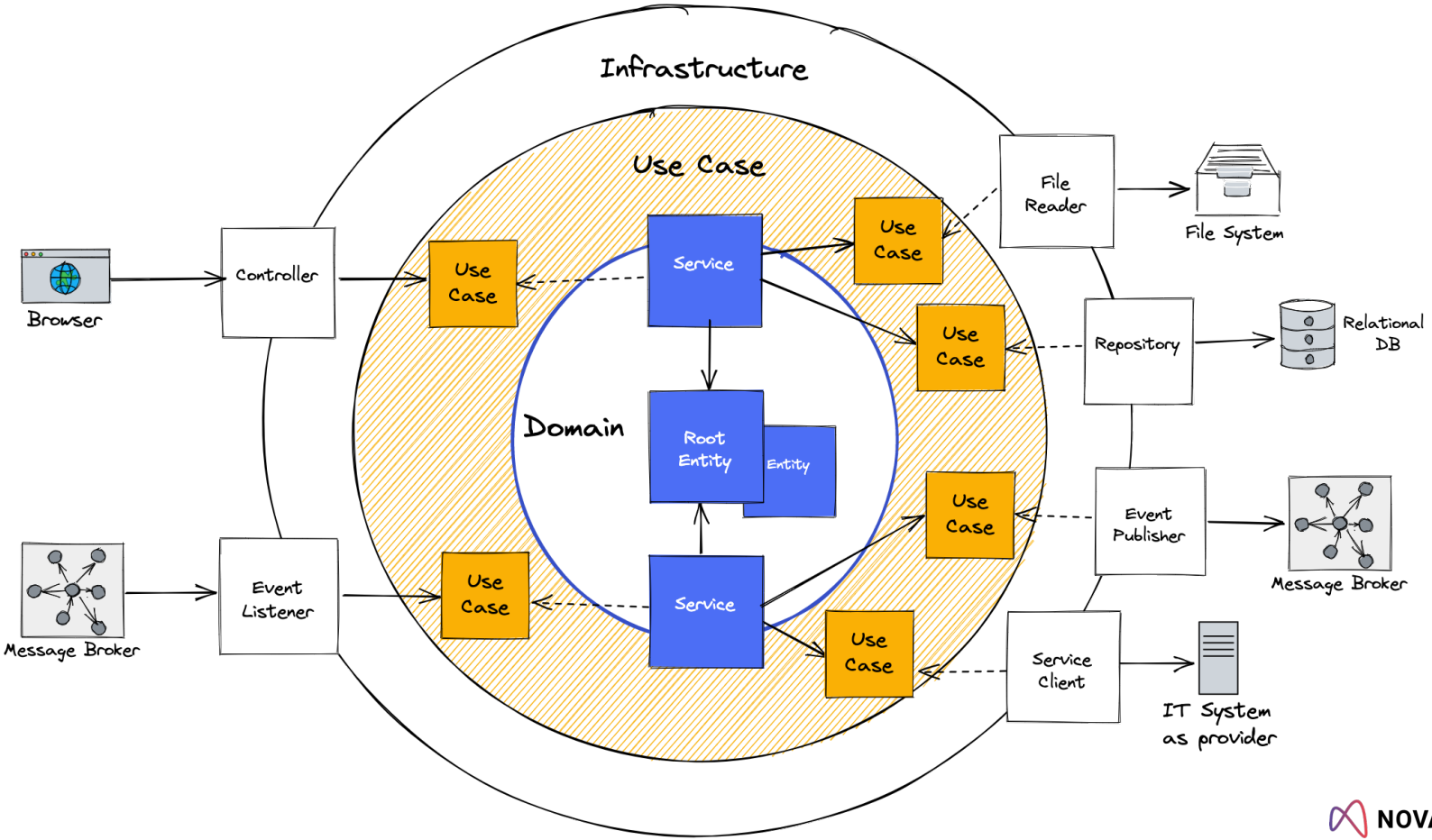
According to Robert C. Martin[3] and Tom Hombergs[1]

NOVATEC

Clean Architecture Pattern

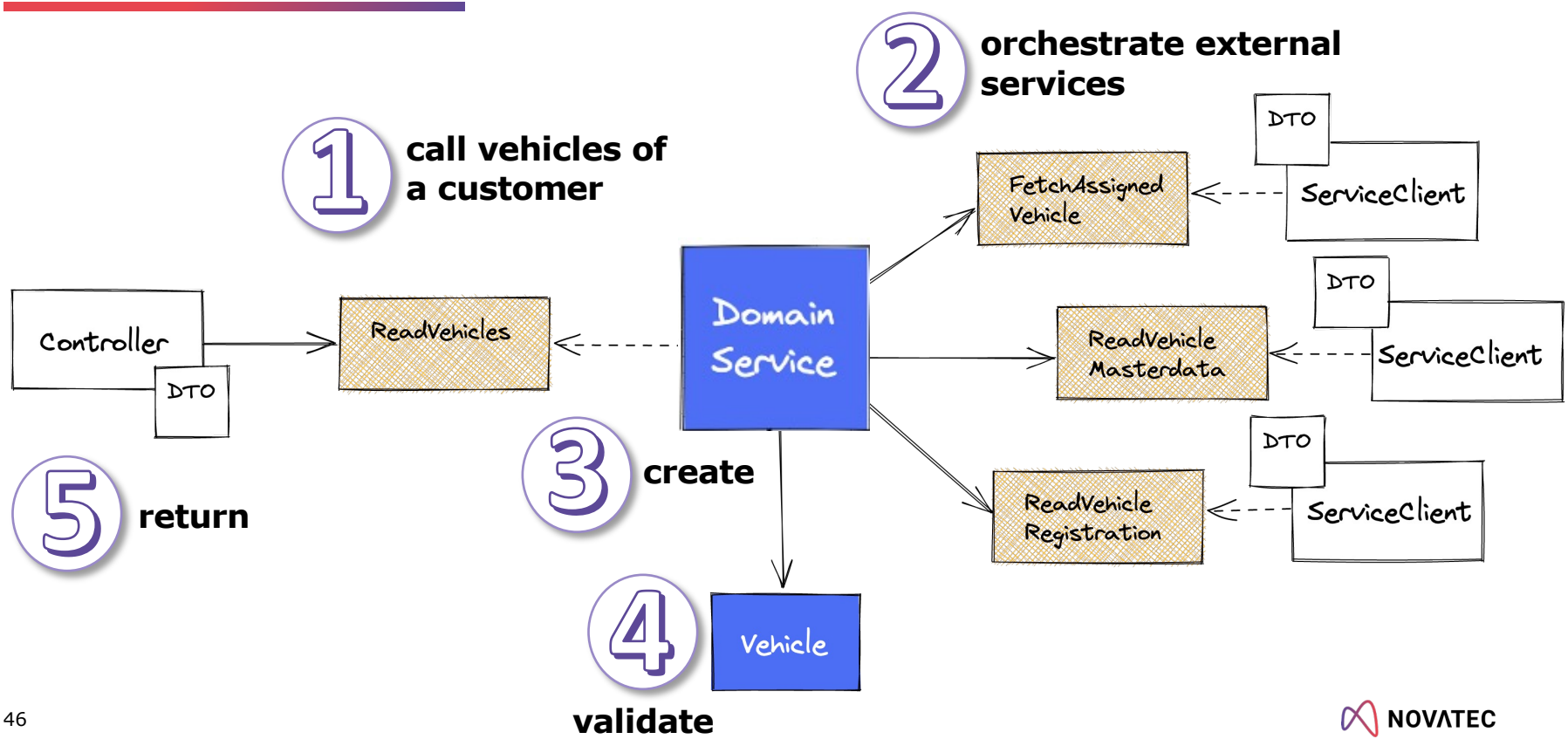


NOVATEC


Infrastructure
Use Case
Domain
Browser
Controller
Use Case
Service
Use Case
Use Case
Root Entity
Entity
Use Case
Use Case
Service
Use Case
Message Broker
Event Listener
File Reader
File System
Repository
Relational DB
Event Publisher
Message Broker
Service Client
IT System as provider

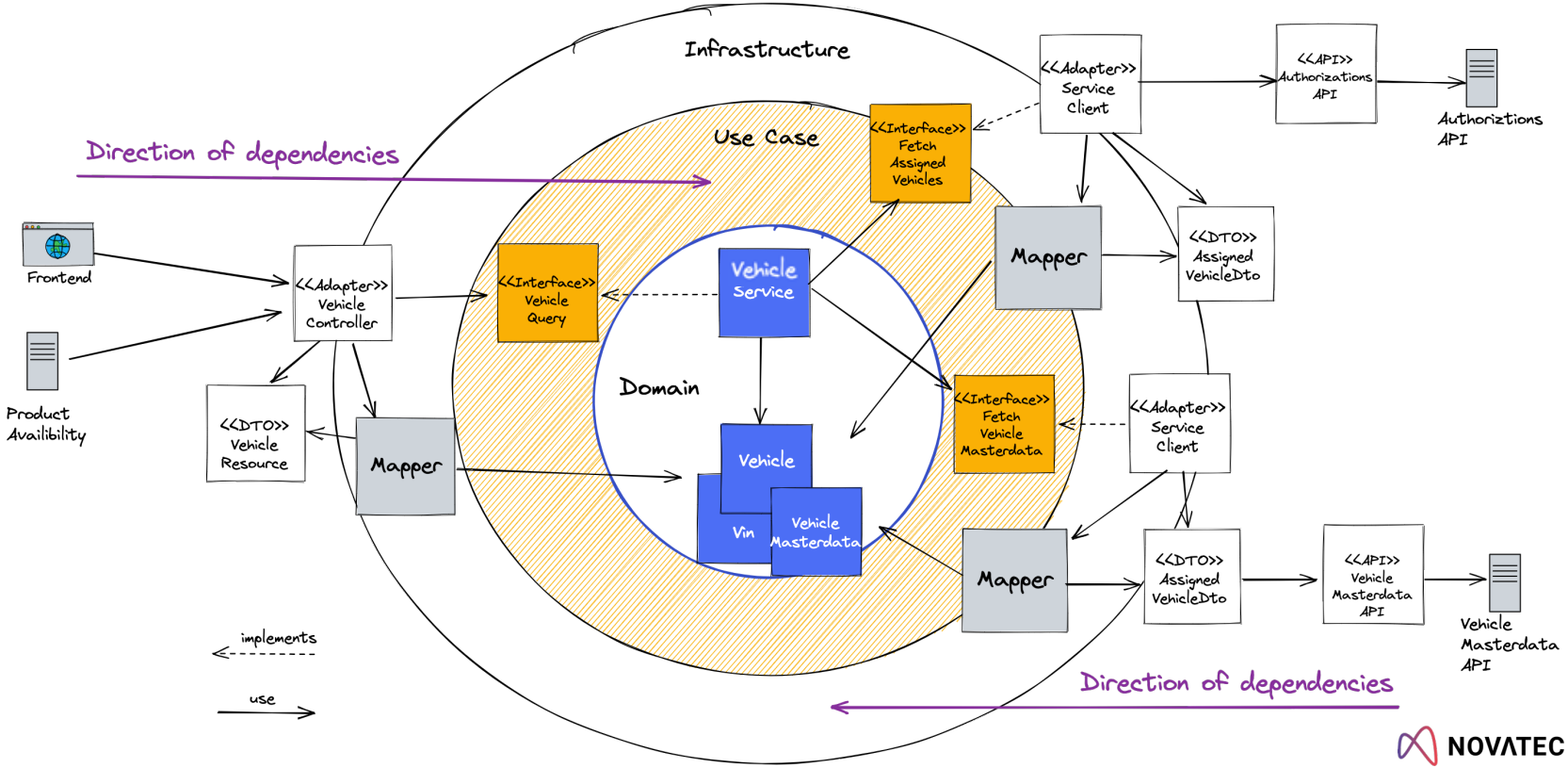
NOVATEC

Flow Through the Rings



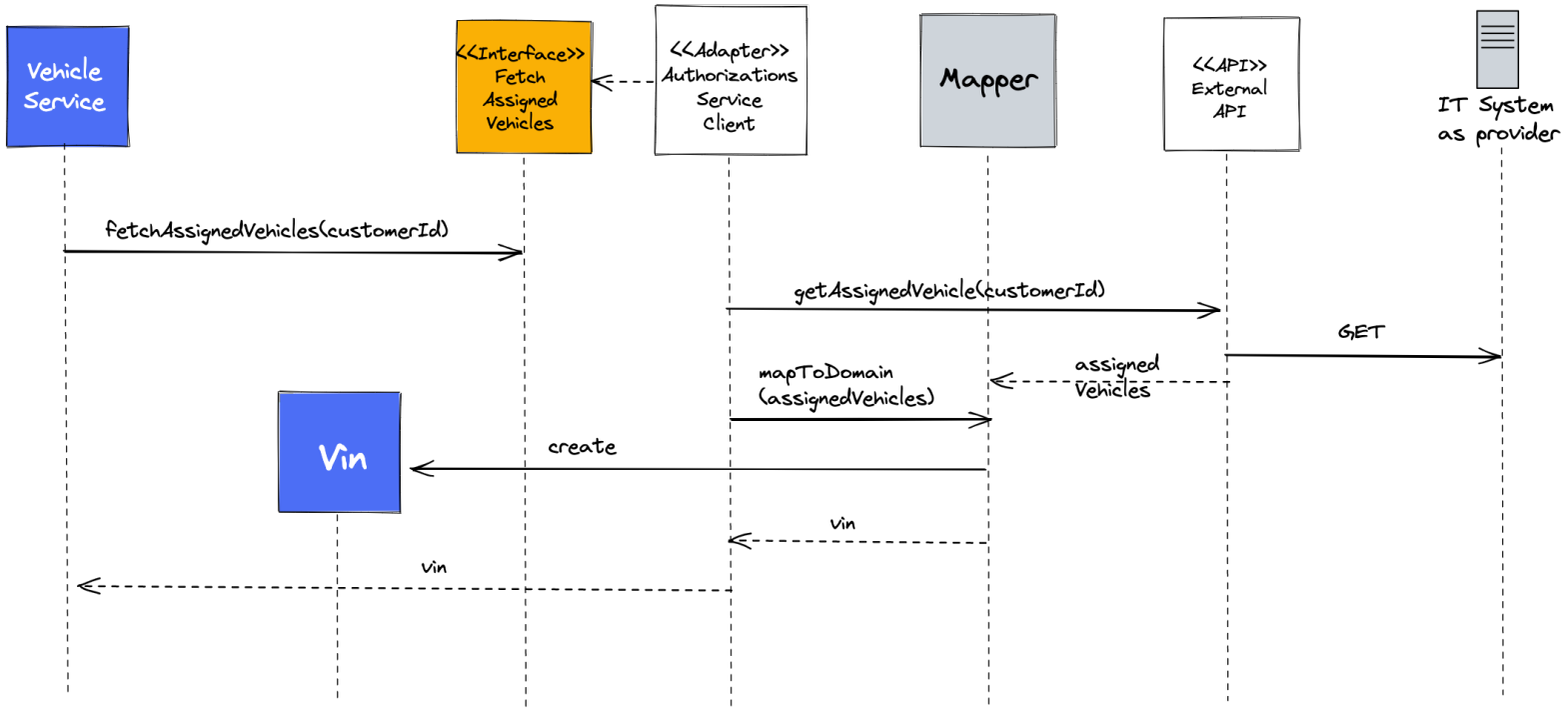
NOVATEC

Class Structure Based on the Ring Model



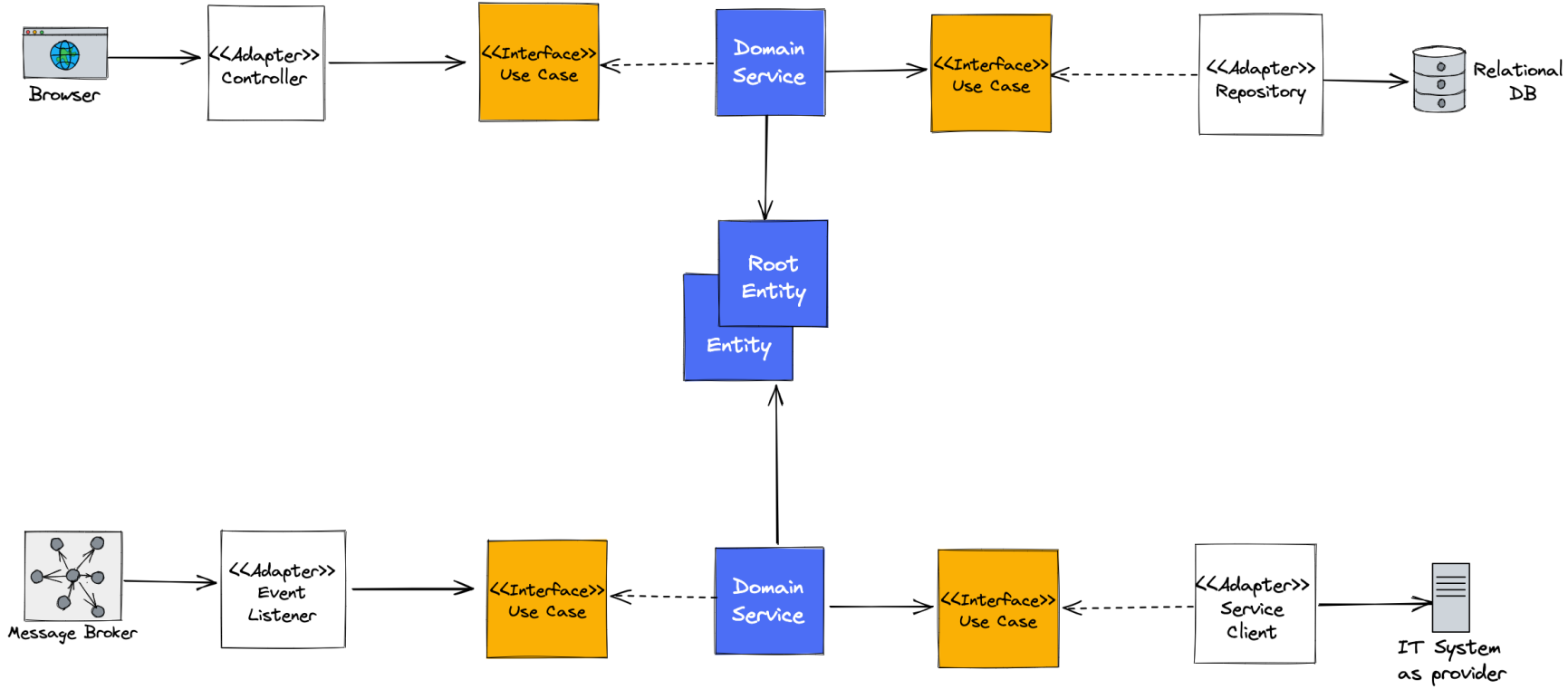
NOVATEC

The sequence of the Outgoing Use Case *FetchAssignedVehicles*



NOVATEC

Class Stereotypes of Clean Architecture



NOVATEC

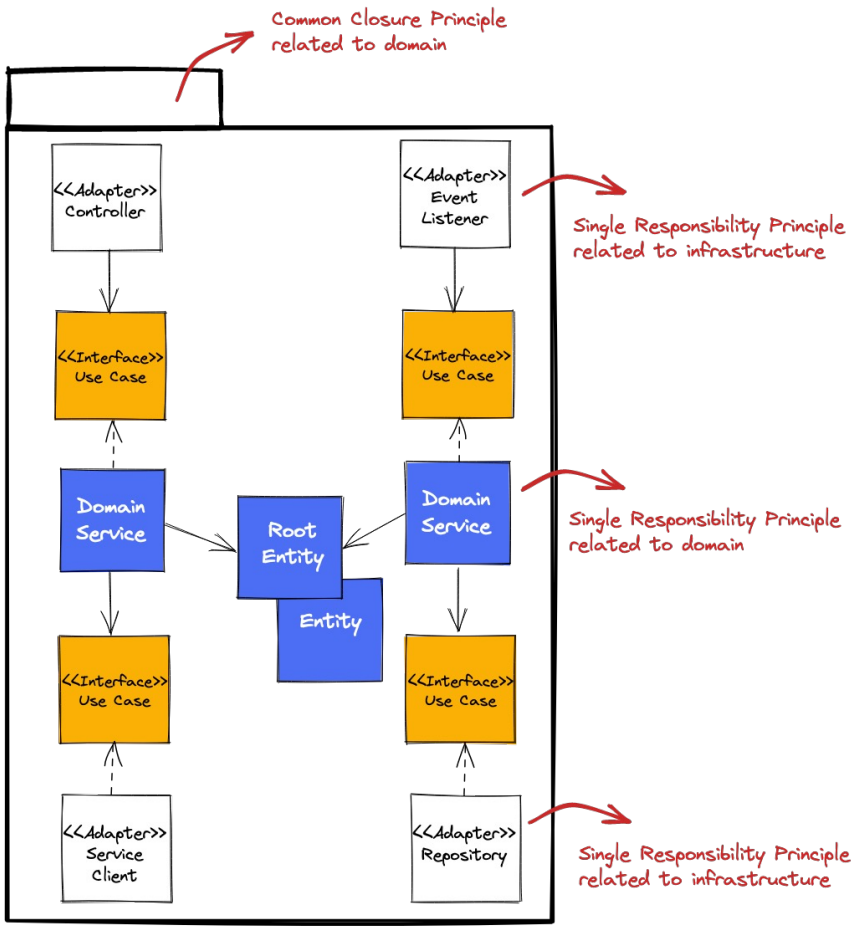
Applied Architecture Principles

- **Common Closure Principle**
  Pack together what changes together

- **Single Responsibility Principle**
  One reason to change

- **Dependency Inversion Principle**
  Invert the dependency between classes

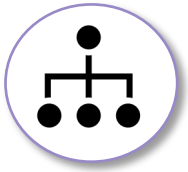
- **Separation of technical and domain-related aspects**



NOVATEC

Domain Services Connect All Together

- Connect input adapters with the domain core by implementing incoming use cases
- Orchestration of output adapters

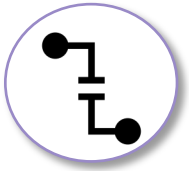


Connect & Orchestrate

```
public class VehicleQueryService implements VehicleQuery {

  //...

  public List<Vehicle> getVehicles(CustomerId customerId) {

    List<Vehicle> pairedVehicles = pairedVehiclesQuery
      .fetchVehiclesByCiamId(customerId);

    for (Vehicle vehicle : pairedVehicles) {
      Vin vin = vehicle.getVin();

        vehicle.addVehicleMasterdata(
          vehicleMasterdataQuery.getVehicleByVin(vin));

        vehicle.addVehicleRegistration(
          vehicleRegistrationQuery.getVehicleRegistration(vin));
    }

    return pairedVehicles;
  }
}
```

Domain Services Connect Almost All Together

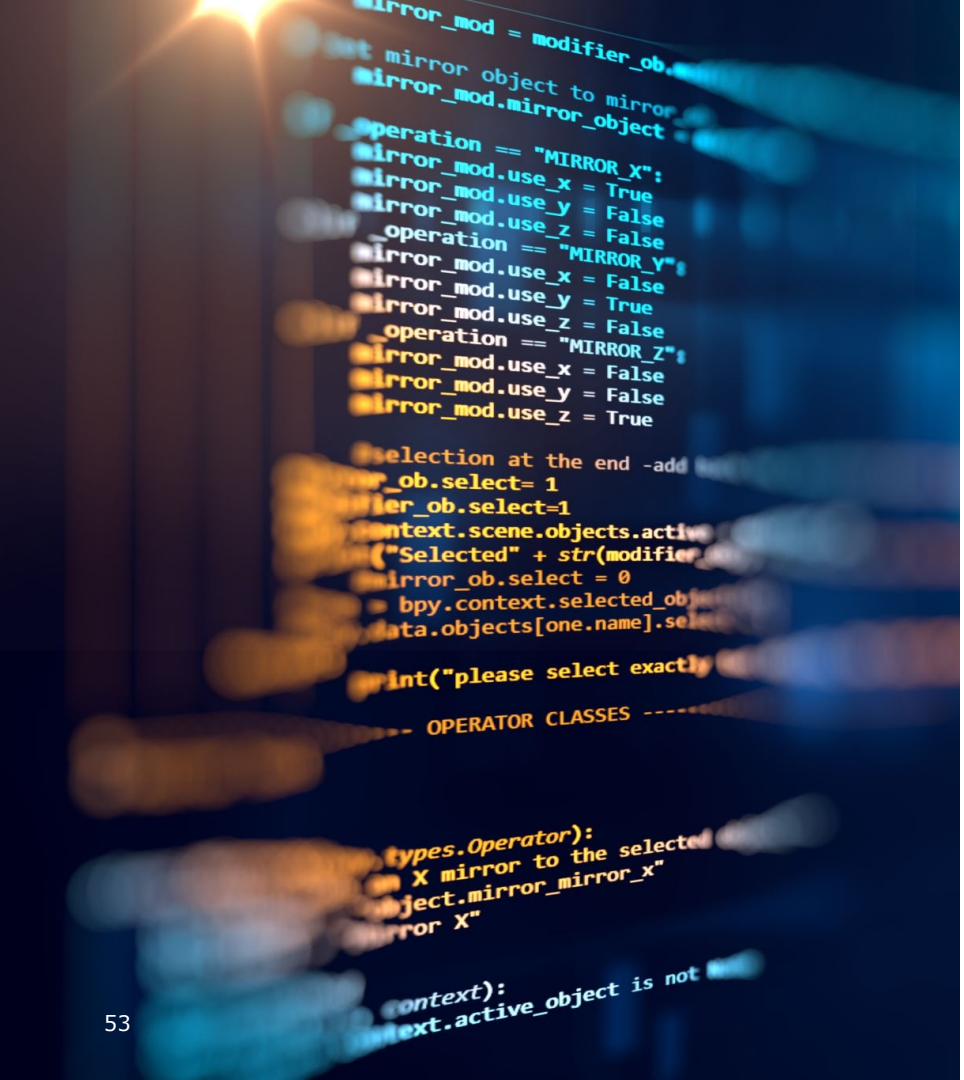
- Delegate execution of domain logic to domain services or entities
- ~~Solve domain-related dependencies between root entities~~



Connect & Delegate

```
public class GarageOrderApplicationService implements GarageOrderCommand {

    private final ReadGarageOrder readGarageOrder;
    private final CalculateWorkingTime calculateWorkingTime;
    private final SaveGarageOrder saveGarageOrder;

    public GarageOrder addVehicleServices(GarageOrderId garageOrderId, List<VehicleService> services,
                List<Equipment> equipmentList) {
      GarageOrder order = readGarageOrder.findOrderById(garageOrderId);
      String vin = garageOrder.getVin();

       vehicleServices =  calculateWorkingTime.calculate(services, vin, equipmentList)

       order.addVehicleServices(vehicleServices);

       order.calculateTotalTimeRequired();

      return saveGarageOrder.save(order);
    }
}
```

Equipment and *VehicleService* belongs to the domain model of *GarageOrder*.



Coding Challenge

- Lab 1
- Group discussion
- Lab 2
- Group discussion

NOVATEC

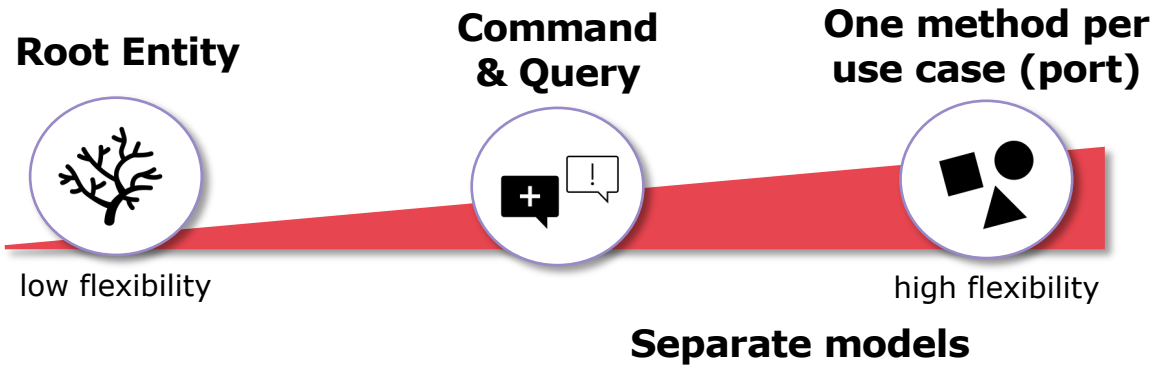
Slicing Use Cases

According to Tom Hombergs[1]

Applying the Interface Segregation Principle for Slicing Use Cases Removes Unnecessary Dependencies and Makes the Existing Dependencies More Visible

- *"Depending on something that carries baggage that you don't need can cause you troubles that you didn't expect."*
  Robert C. Martin[3]

- Interfaces should be separated into small responsibilities so that consumers only depend on the functionality they really need



NOVATEC

Use Cases Based on (Root) Entity

```
public interface VehicleUseCase {

    List<Vehicle> fetchVehicleByCustomerId(CustomerId customerId);

    VehicleMasterdata fetchVehicleMasterdata(Vin vin);

    VehicleRegsitration fetchVehicleRegistration(Vin vin);

    Vehicle saveVehicle (Vehicle vehicle);
}
```

Use Cases Based on Command and Query

```
public interface VehicleQuery {

    List<Vehicle> fetchVehicleByCustomerId(CustomerId customerId);

    VehicleMasterdata fetchVehicleMasterdata(Vin vin);

    VehicleRegsitration fetchVehicleRegistration(Vin vin);

}


public interface VehicleCommand {

    Vehicle saveVehicle (Vehicle vehicle);

}
```

Use Cases Based on Functions

```
public interface FetchVehicle {
   List<Vehicle> fetchByCustomerId(CustomerId customerId);
}

public interface FetchVehicleMasterdata {
    VehicleMasterdata fetchMasterdata(Vin vin);
}

public interface SaveVehicle {
   Vehicle save (Vehicle vehicle);
}
```

Recommendations

- Prefer business rules and validation inside (root) entities
- Use rich entities for calculations & detections
- Use domain services for calculations & detections not assignable to a root entity
- Connect and orchestrate infrastructure only with application services

The Dark Side of Decoupling

Mapping between boundaries

NOVATEC

A Definition for Mapping

Mapping

- The **transformation** of the **state** of an **entity model <A>** to an **entity model <B>** between **layers** (or **rings**).

- Eschhold

Pros, Cons and Tradeoffs

- If we **don't map between layers**, we have to use the same model in both layers, which **means that the layers** will be **tightly coupled**.

- Hombergs [1]

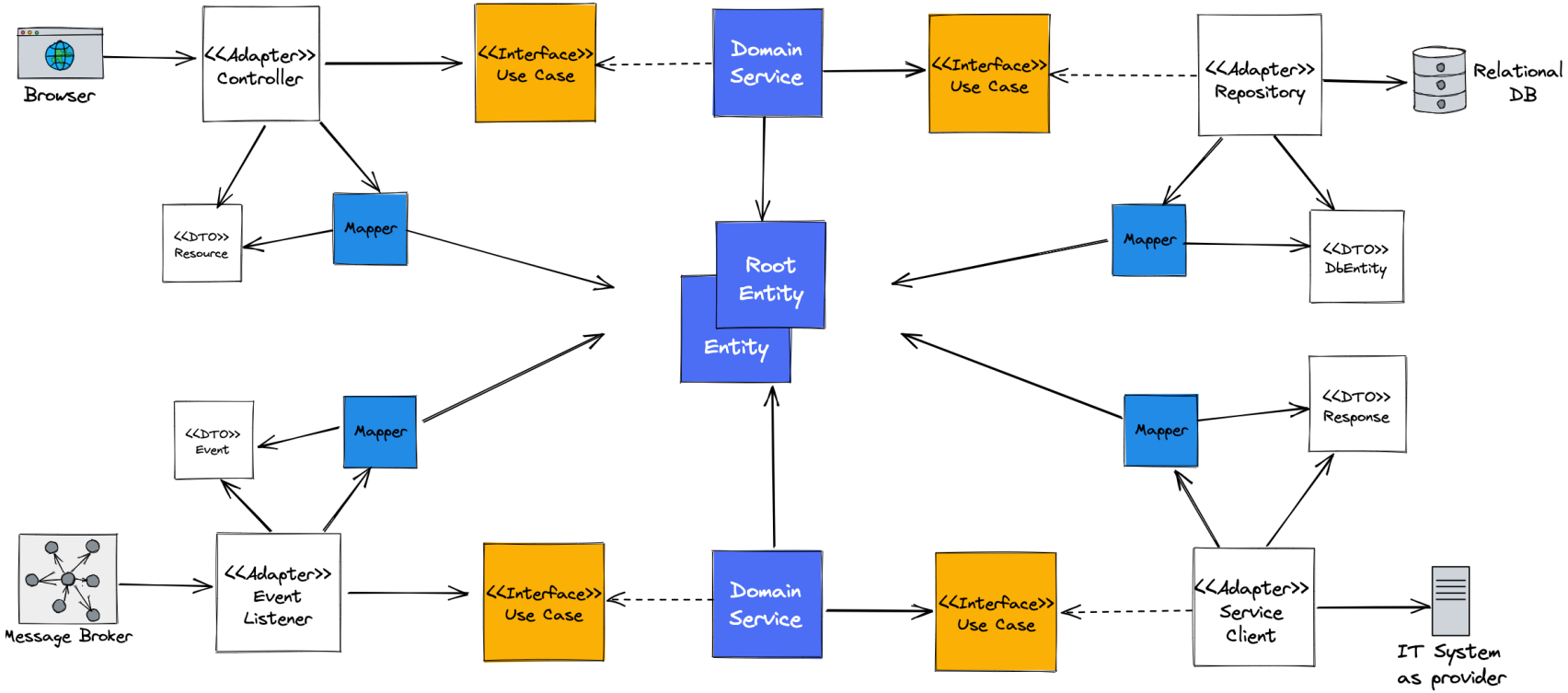
- But if we **do map** between layers, we produce a lot of **boilerplate code**, which is **overkill** for **many use cases**, since **they're only doing CRUD** and have the **same model across layers** anyway.

- Hombergs [1]

- If we **don't map between layers,** how could the domain model be envolve independently from the infrastructure.

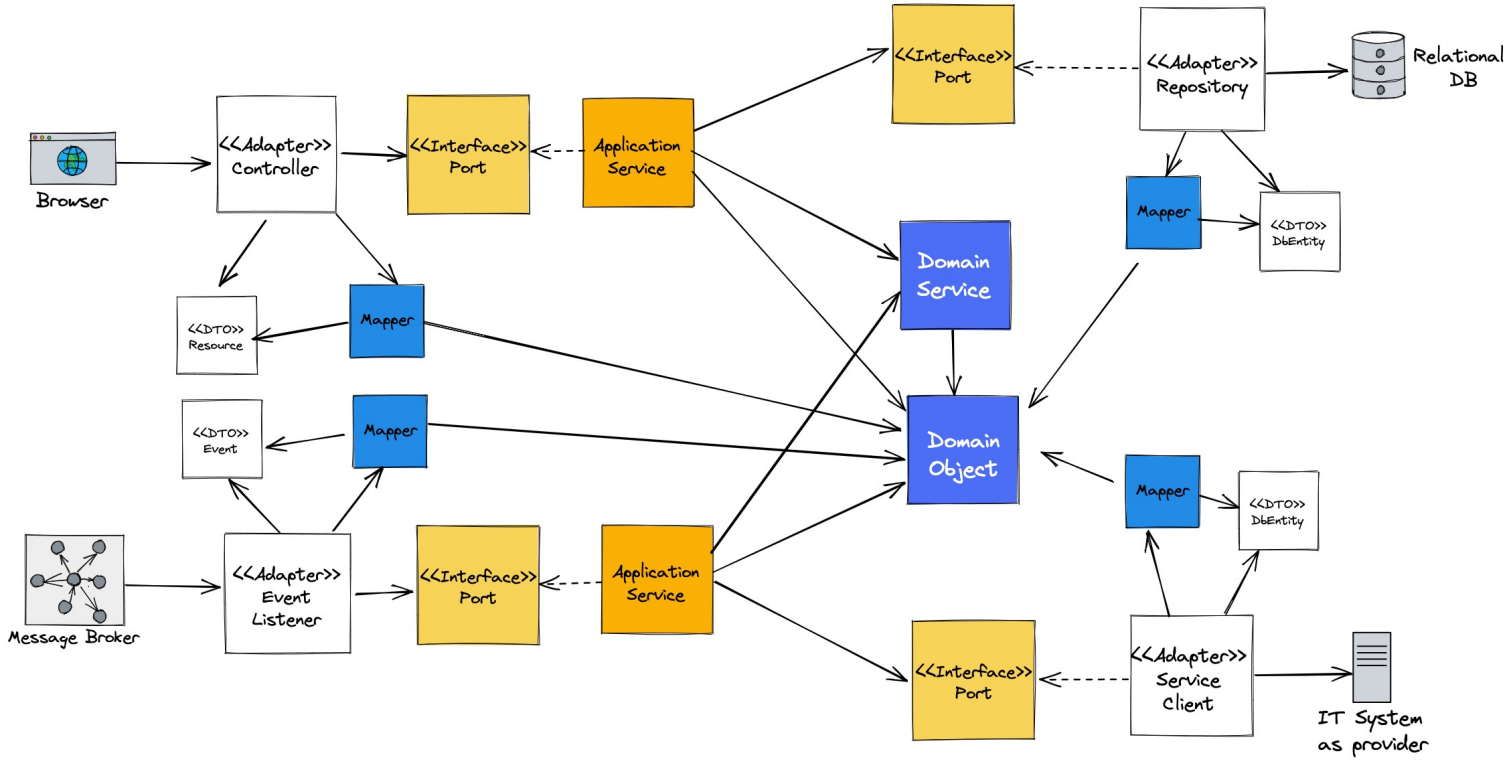
- Eschhold

Two-Way Mapping Strategy within Clean Architecture



NOVATEC

Two-Way Mapping Strategy within Hexagonal Architecture



NOVATEC

Native Mapping Between Objects

```
public class VehicleToVehicleDtoMapper {

    Vehicle mapToDto(VehicleDto vehicleDto) {
      Vehicle vehicle = new Vehicle();
      vehicle.setVin(vehicleDto.getVehicleId());
      vehicle.setVehicleModel(vehicleDto.getModelType());
      return vehicle;
    }

    VehicleDto mapToVehicleDto(Vehicle vehicle) {
      VehicleDto dto = new VehicleDto();
      dto.setVehicleId(vehicle.getVehicleModel());
      dto.setModelType(vehicle.getVehicleModel());
      return dto;
    }

}
```

Mappings with MapStruct

```
@Mapper
public interface VehicleToVehicleDtoMapper {

    @Mapping(target = "vin", source = "vehicleId")
    @Mapping(target = "vehicleModel", source = "modelType")
    Vehicle mapToDto(VehicleDto vehicleDto);

    @Mapping(target = "vehicleId", source = "vin")
    @Mapping(target = "modelType", source = "vehicleModel")
    VehicleDto mapToVehicleDto(Vehicle vehicle);

}
```

https://mapstruct.org/

Mappings with MapStruct and Domain Primitives Java

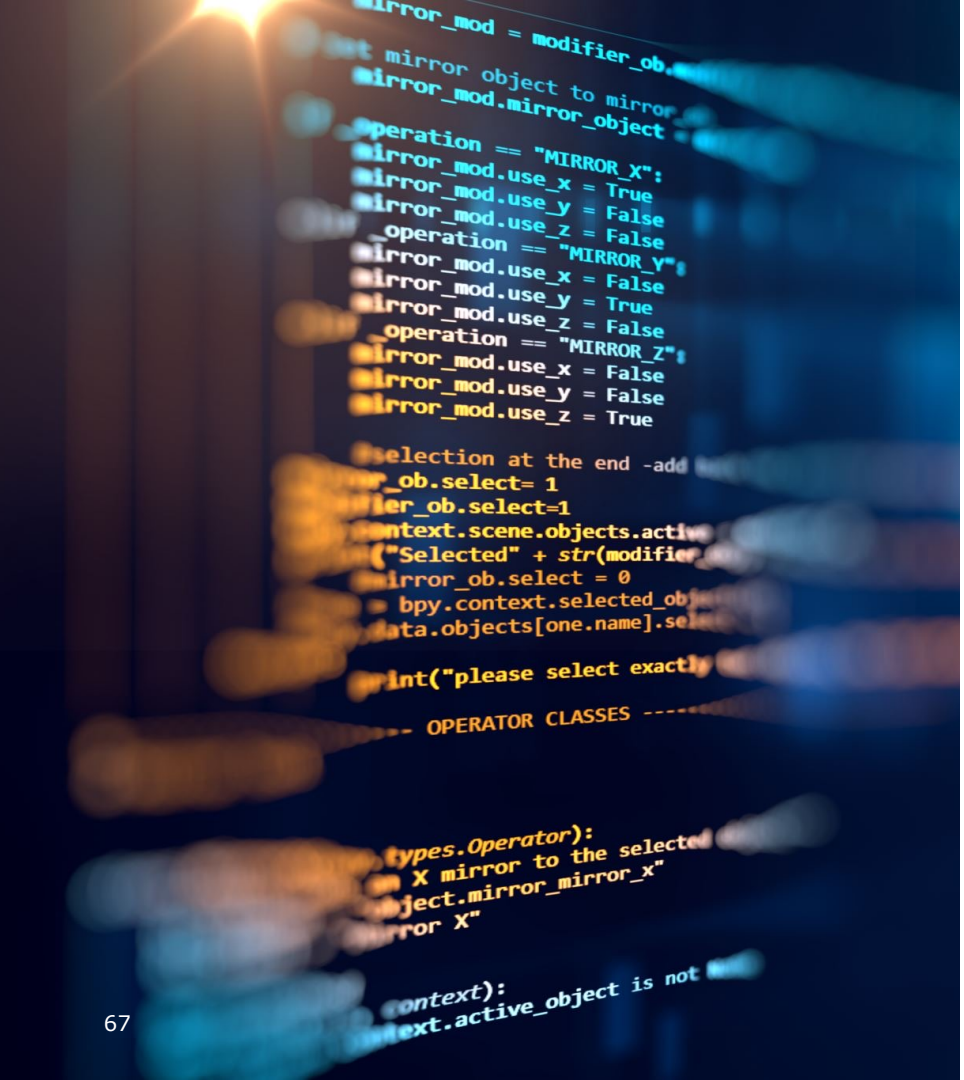
```
@Mapper(uses = ValueObjectMapperFactory.class)
public interface VehicleToVehicleResourceMapper {

    VehicleDto mapToDto(Vehicle domainObject);
    Vehicle mapToDomain(VehicleResource dto);

}
```

Domain Primitives Java

license Apache-2.0

Primitive, self-validating immutable object types.

NOVATEC



Coding Challenge

- Lab 3
- Group discussion
- Lab 4
- Group discussion

NOVATEC

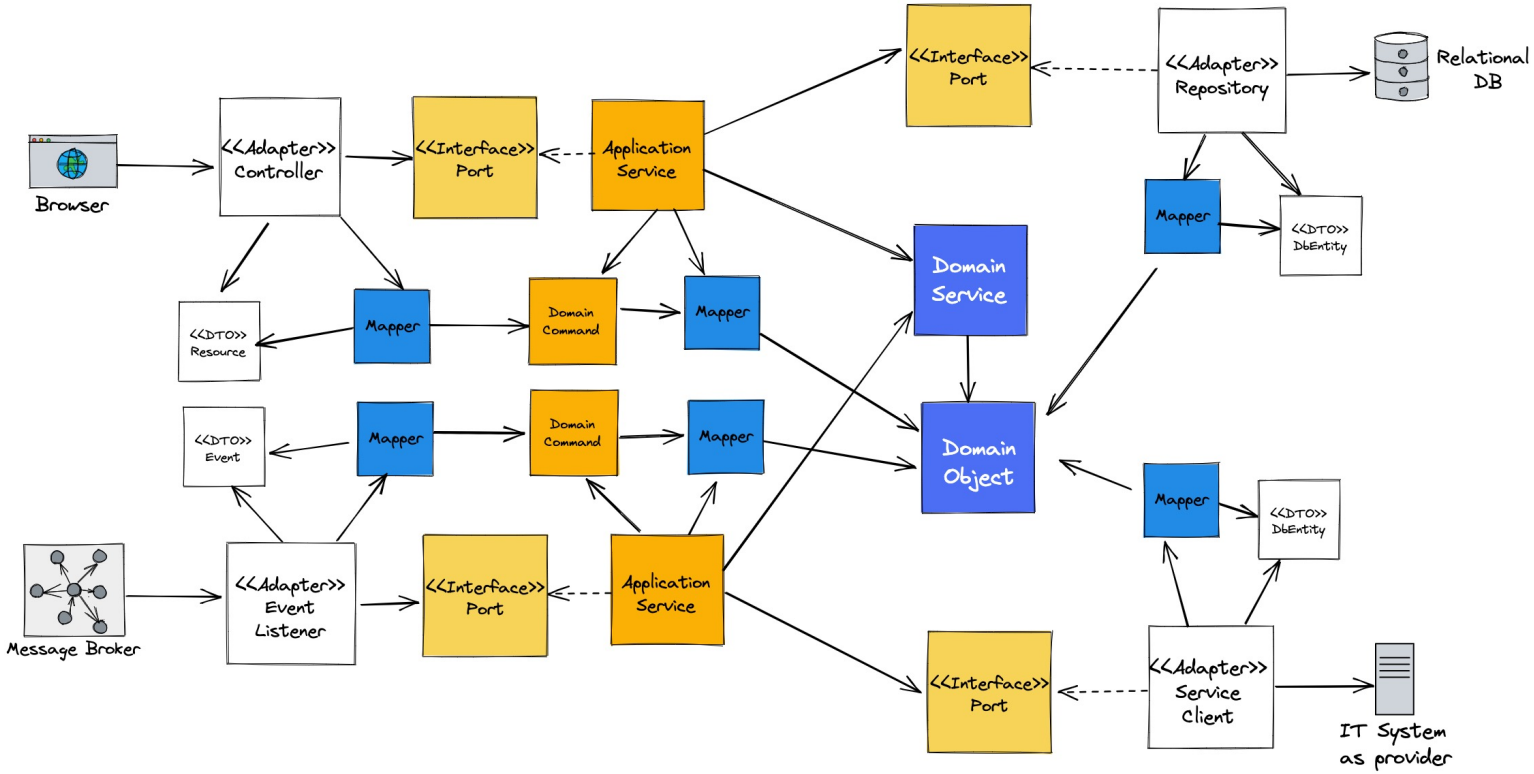
Domain Logic and More Complex Scenarios

Clean Domain Model & The Dark Side of Decoupling II

Mapping Strategies

NOVATEC

Full Mapping Strategy



NOVATEC

When Full Mapping Strategy Make Sense …

Separation Between Domain, Use Case Model, and Consumers

- Separate model for each use case and / consumer
  - result in increased flexibility,
  - avoid side effects based on independent use cases,
  - that are easier to change

- But it costs a mapping between
  - (Root) Entity and
  - Use Case Model (aka Domain DTO)

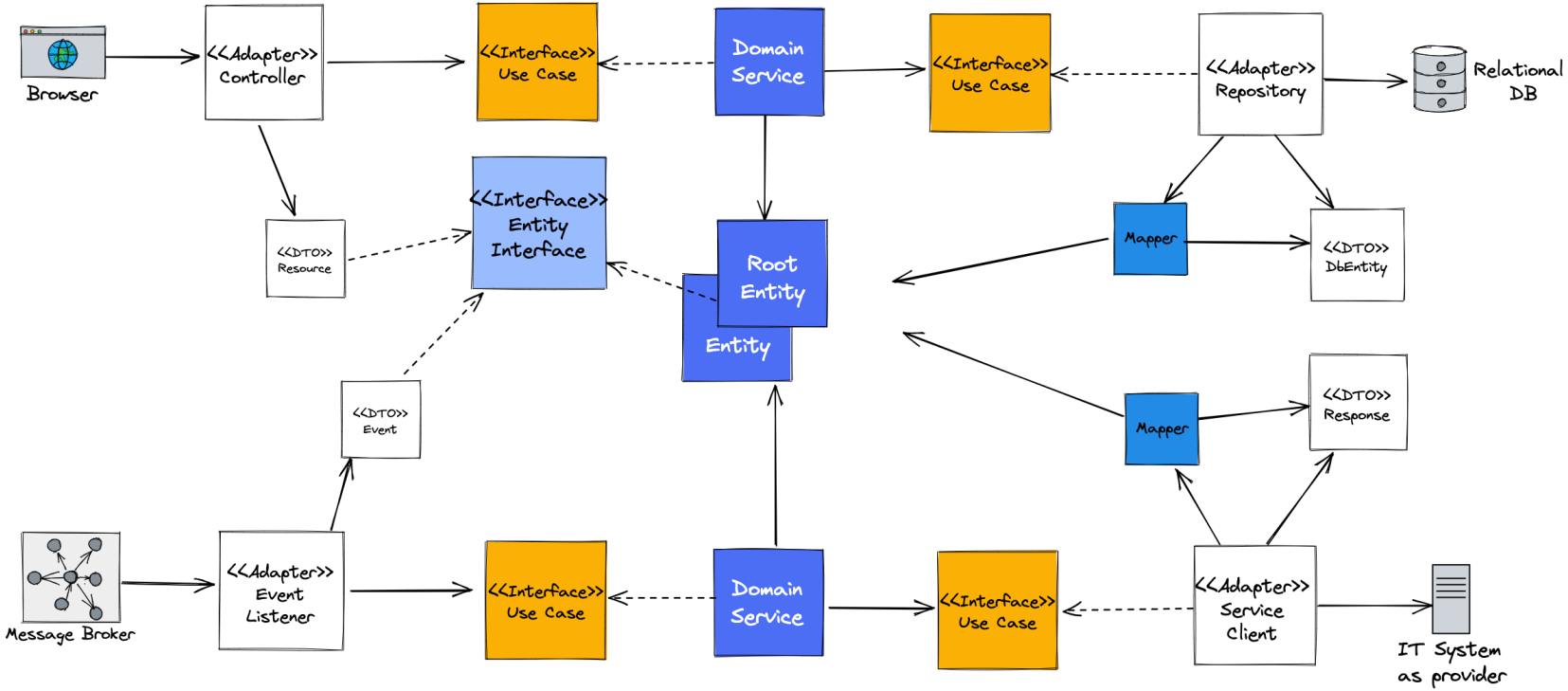
Ensure a Clean and Independent Domain Model

Domain-related and technical aspects could only be clearly separated from the domain based on an additional layer (ring or hexagon).

For example:

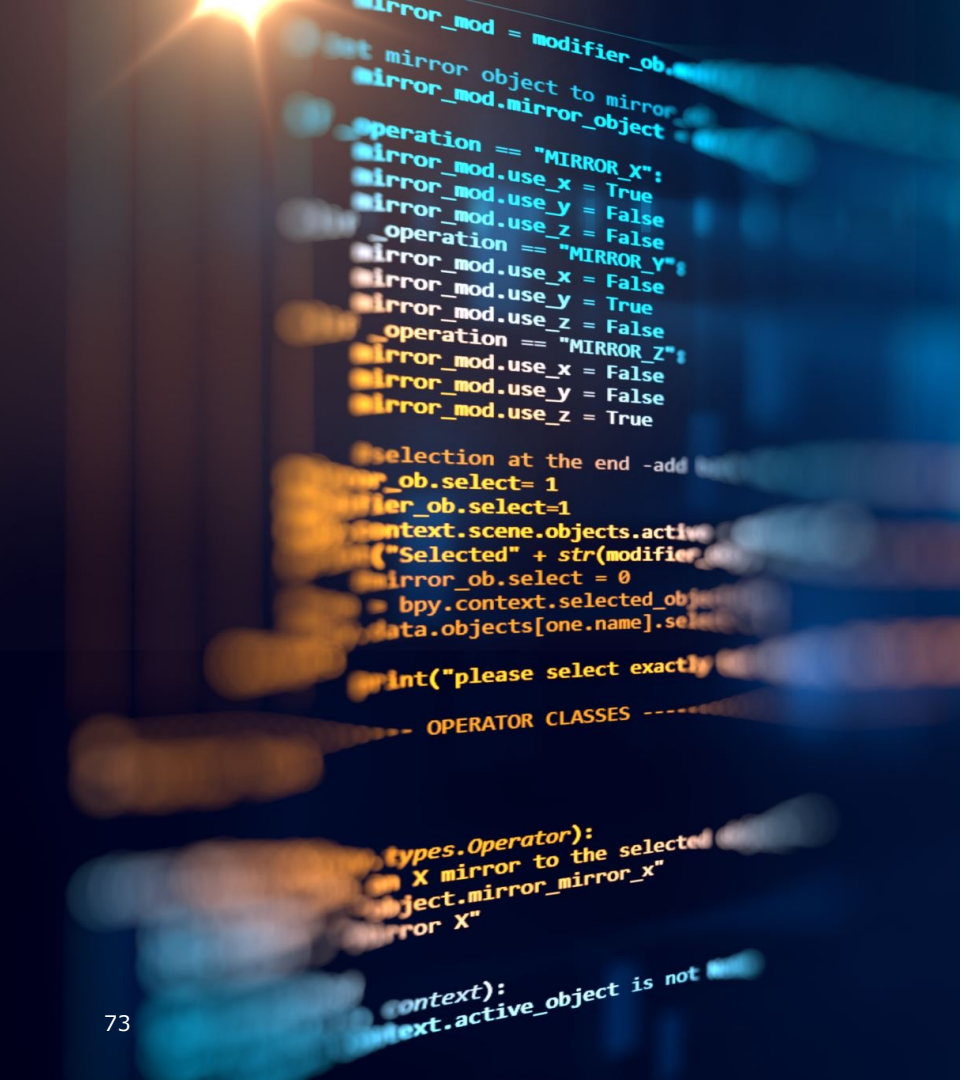
- High complexity in orchestrating infrastructure or external data
- enriched domain values based on domain logic

One-Way Mapping Strategy



Encapsulate domain rules and logic without mapping!

NOVATEC



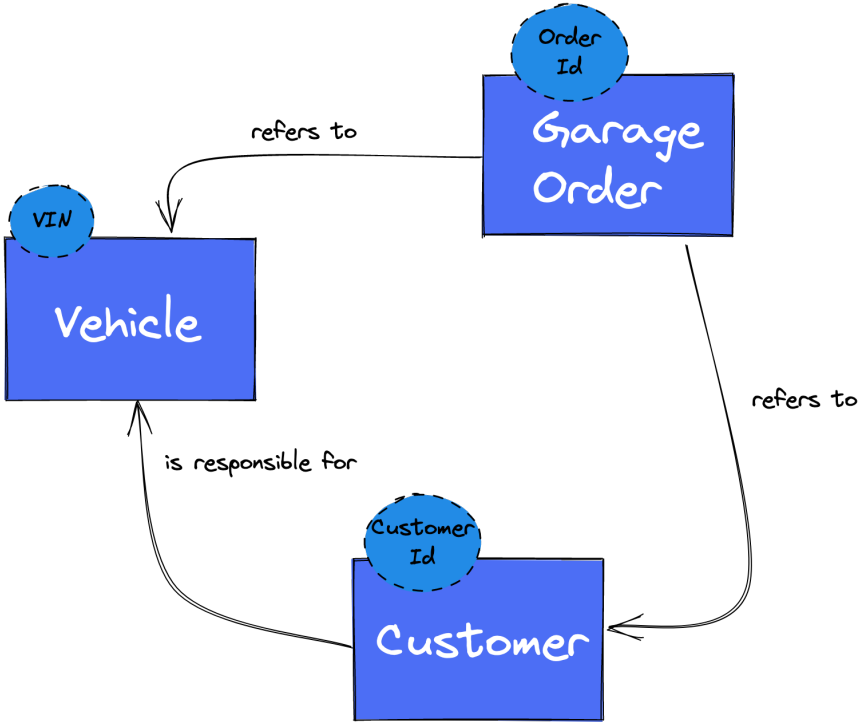
Coding Challenge

- Group discussion

NOVATEC

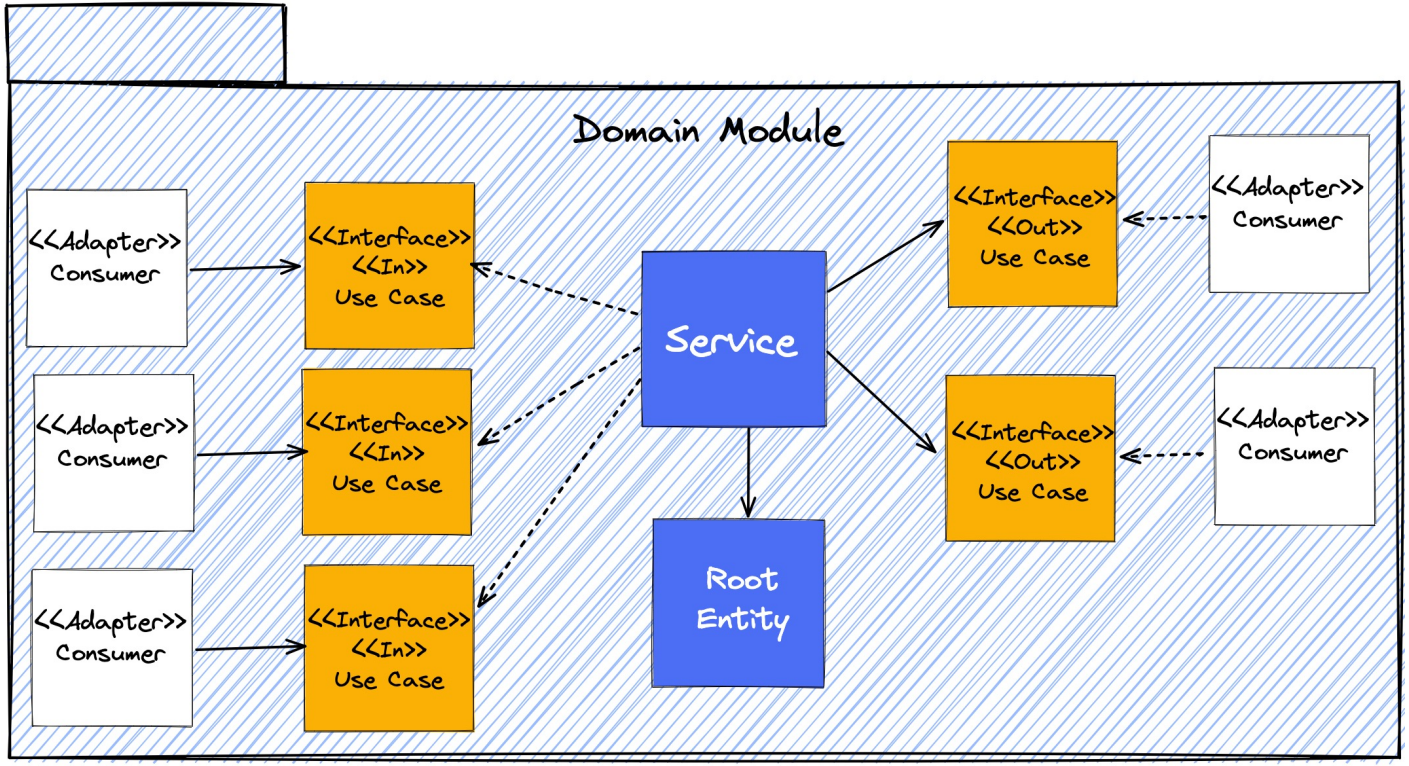
Domain Modules and Dependencies Between

Modularization around root entities



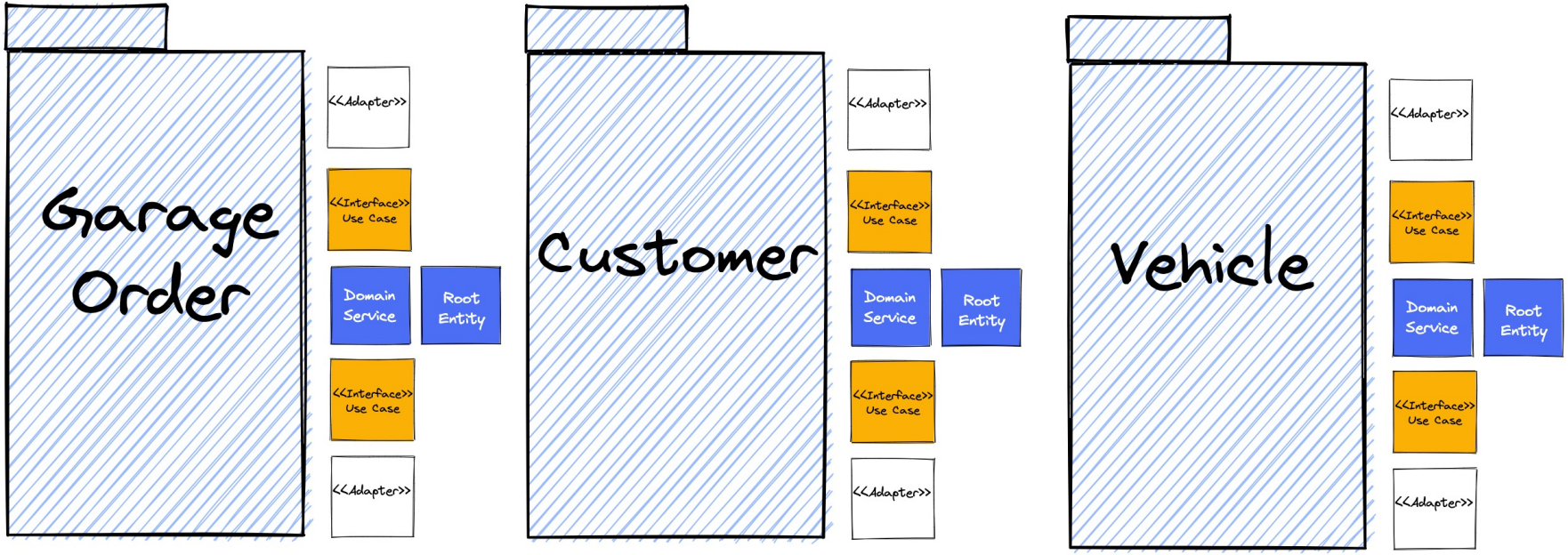
- *A garage order refers to a customer and a vehicle*
- Root entities have independent lifecycles
- Root entities symbolize domain-related consistency boundary

Modularization Around Root Entities



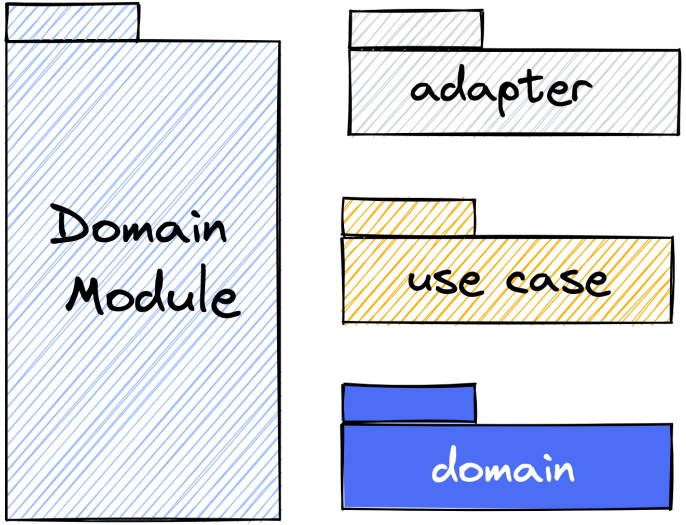
NOVATEC

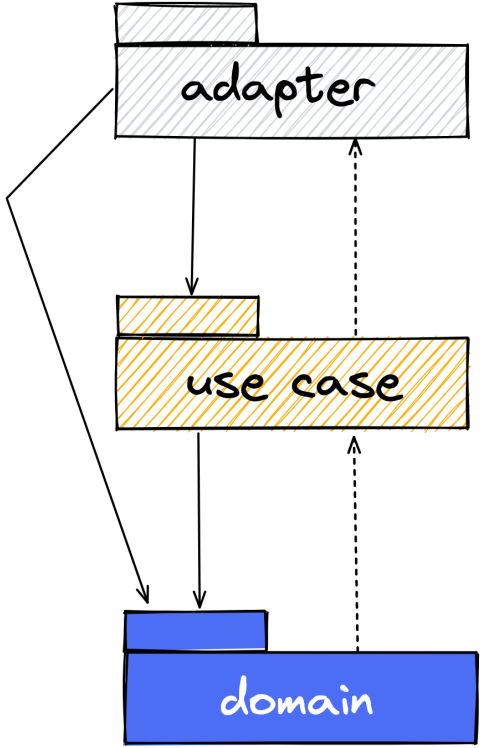
Modularization Around Root Entities



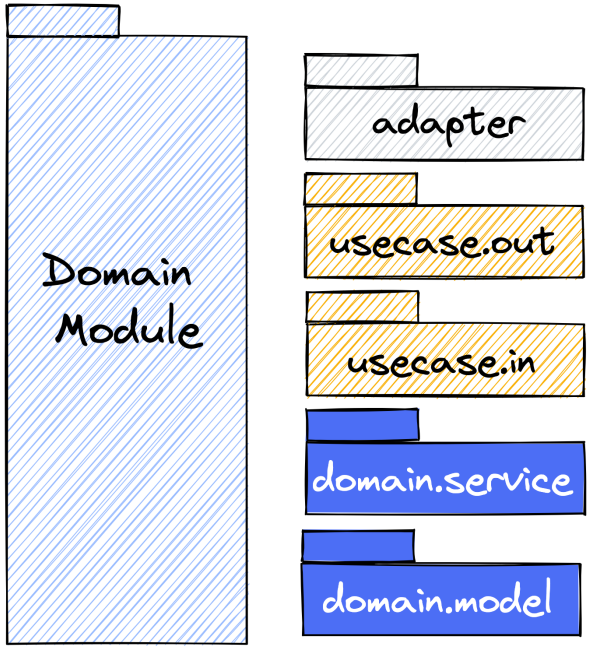
NOVATEC

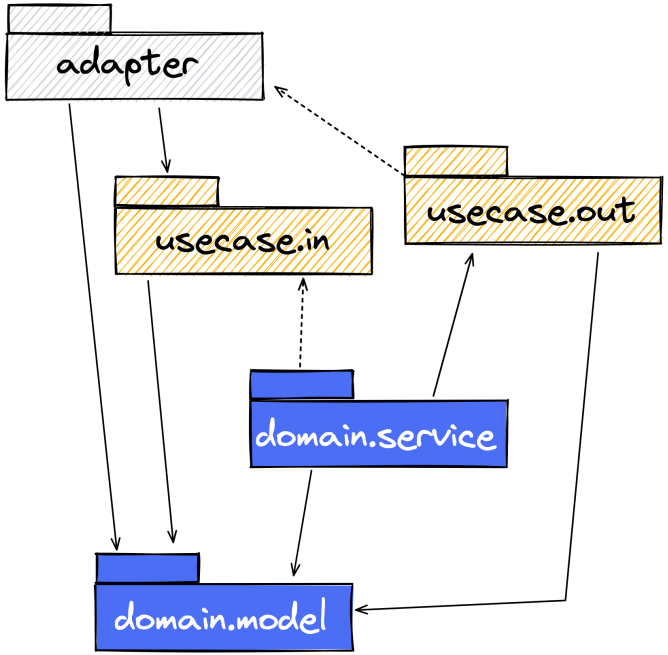
Rings As Layers





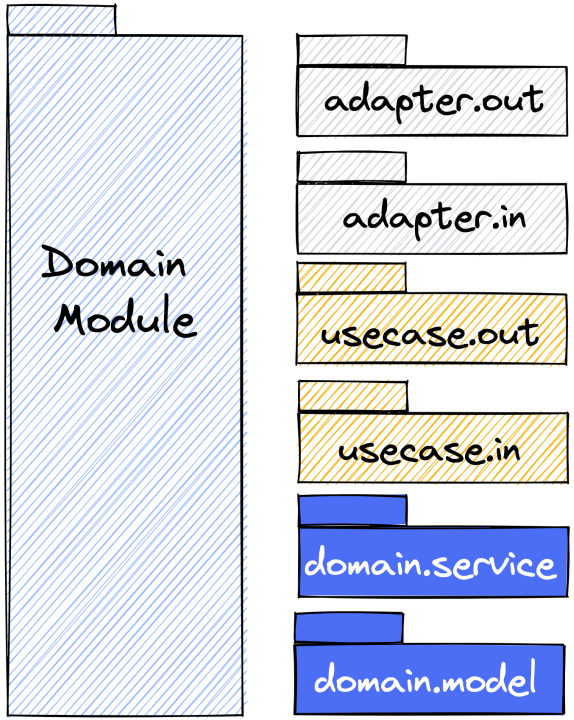
Architectural Expressive Domain

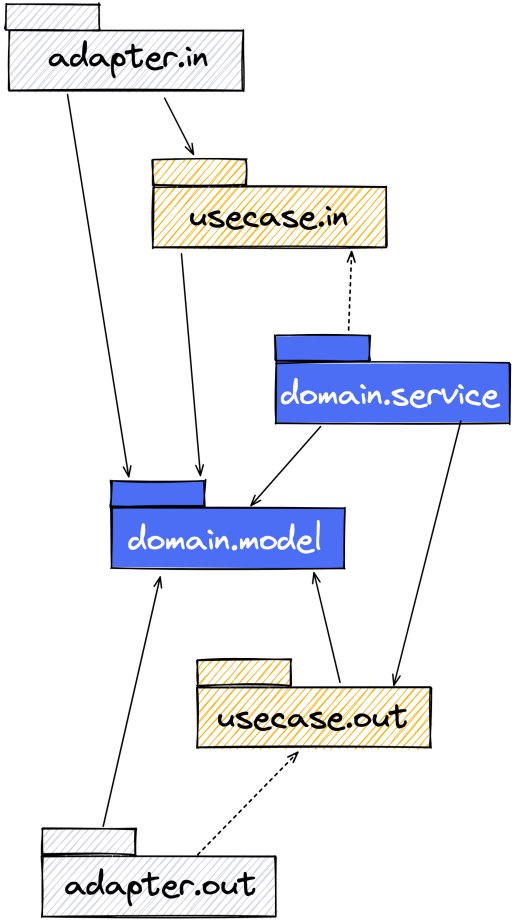




NOVATEC

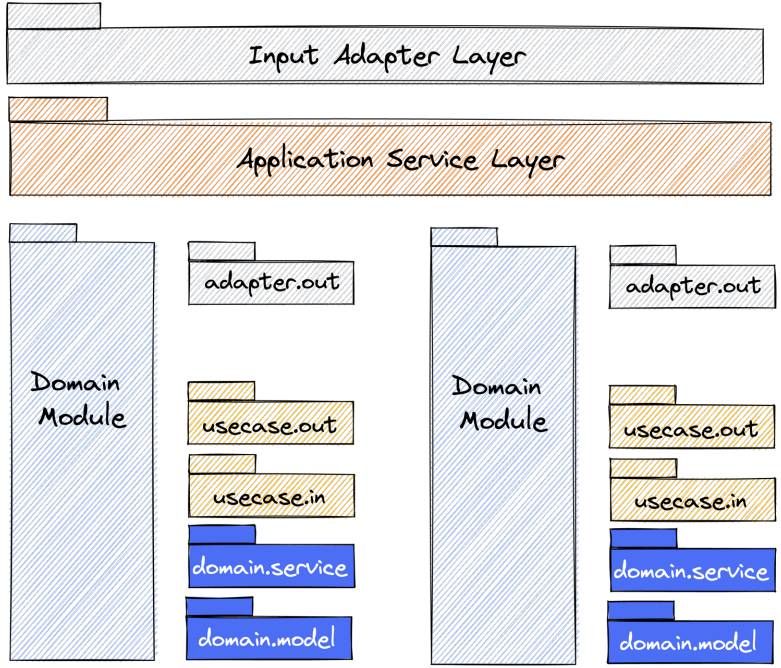
Architectural Expressive

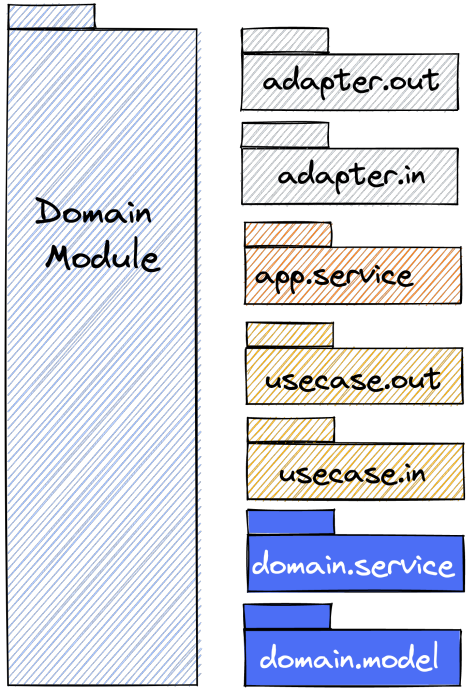




NOVATEC

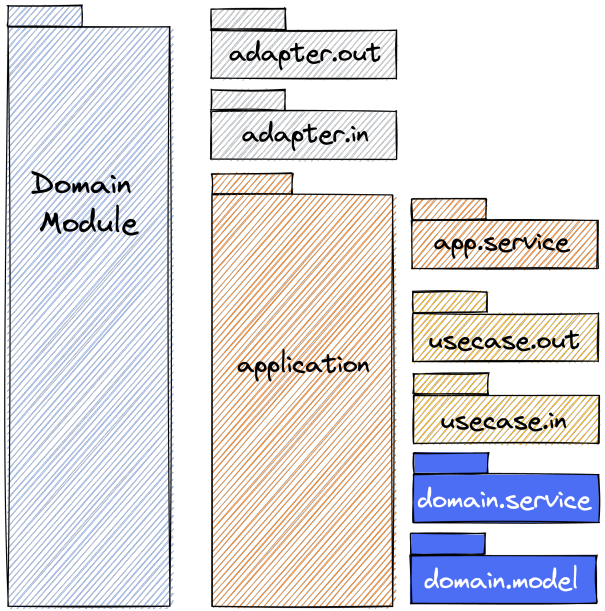
Modularization with an Application Service Within Hexagonal Architecture

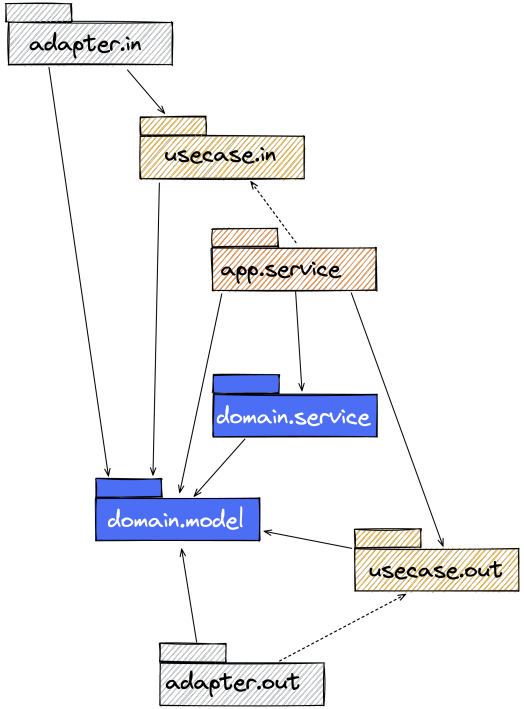


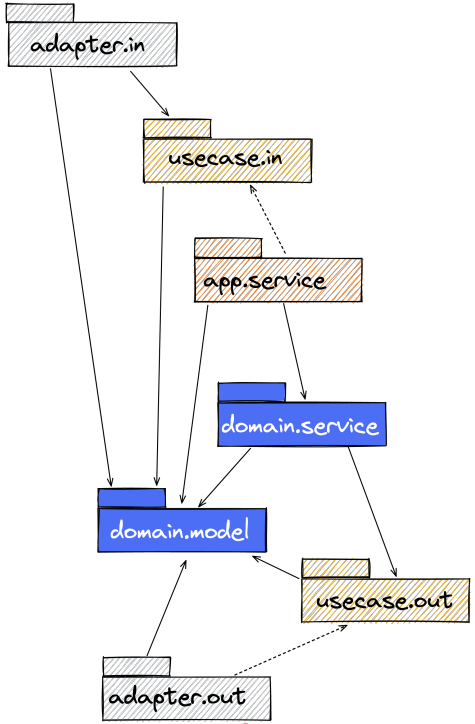


NOVATEC

Modularization with an Application Service Within Hexagonal Architecture

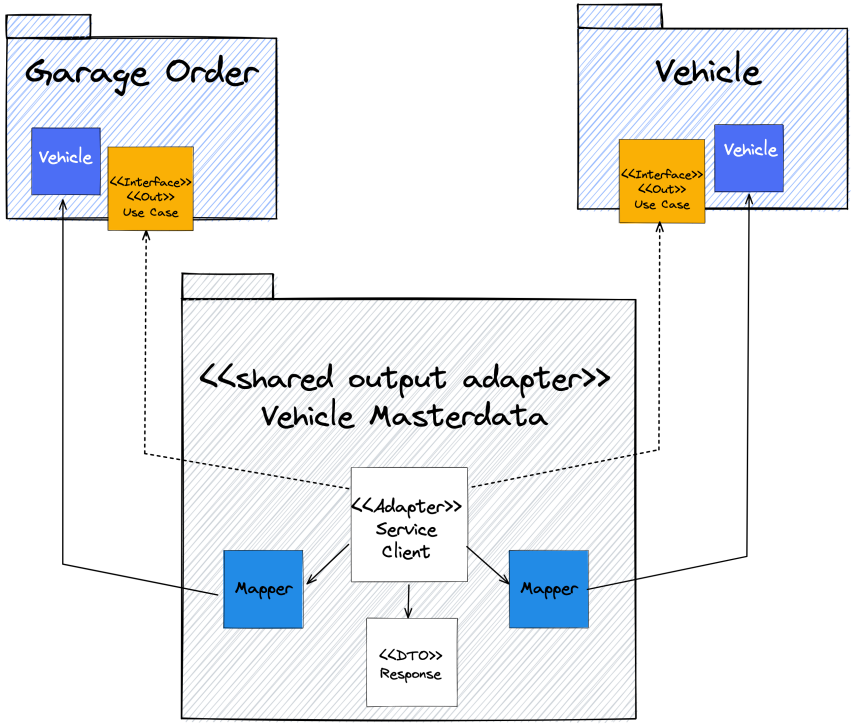






NOVATEC

Shared Output Adapter



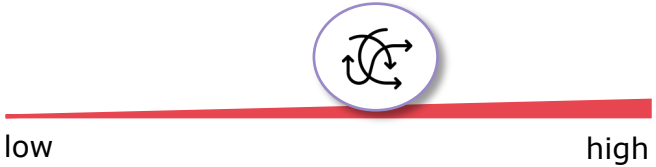
Intended use

- Integration of infrastructure is trivial, (very) similar for each consumer, and stable

Implementation

- Decoupling based on different models for each use case
- Each domain module owns its use case
- For each consumer exists a mapper

Coupling



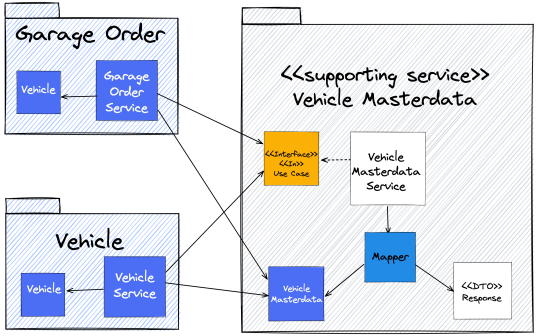
Supporting Service

Intended use

- Functionality is trivial and part of several primary functions
- Technical functions, for example sending email
- Subordinated but domain-related functionality
- Use when the costs of reusability are lower than the costs of decoupling

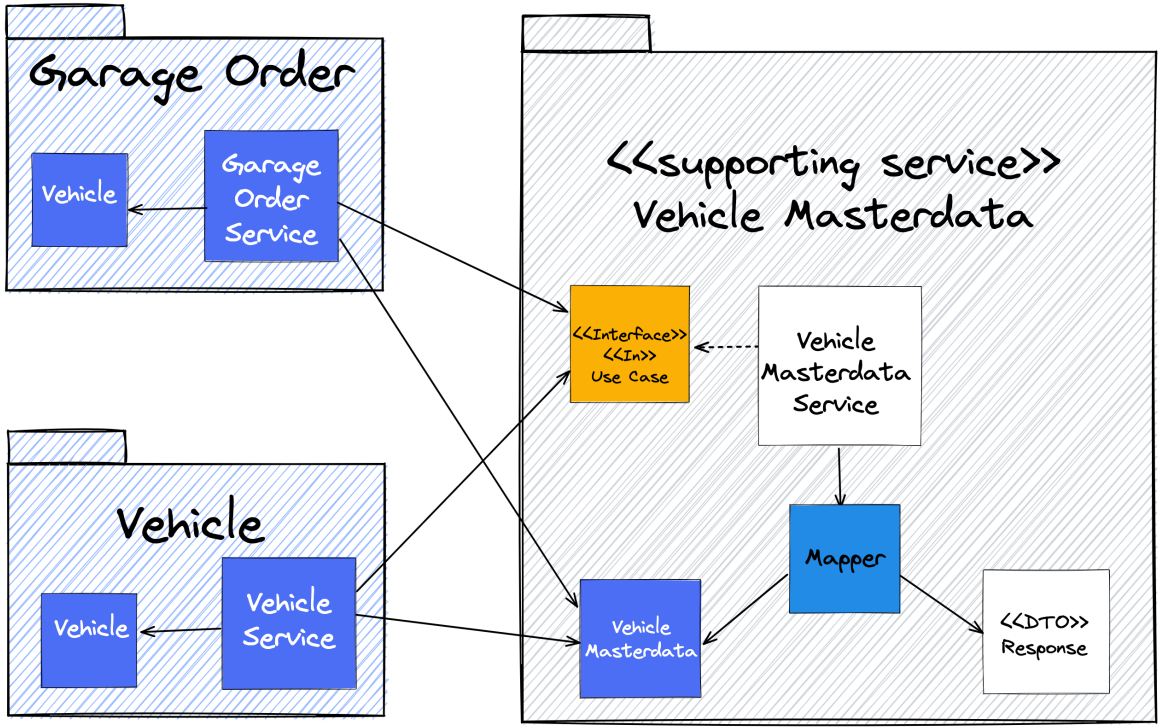
Implementation

- The supporting service offers the functionality through an incoming use case
- The shared functionality relays on a shared model



NOVATEC

Supporting Service



NOVATEC

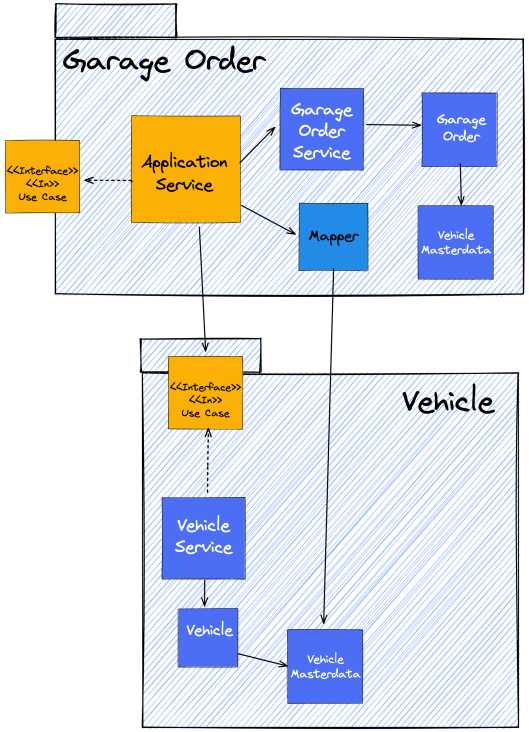
Application Service

Intended use

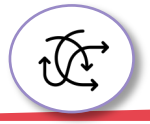
- Dependencies between domain modules (which also means between root entities) will be resolved by an application service so that the domain has no technical dependency on each other

Implementation

- The application service implements the use case, orchestrates domain services, and maps between entities of domain modules



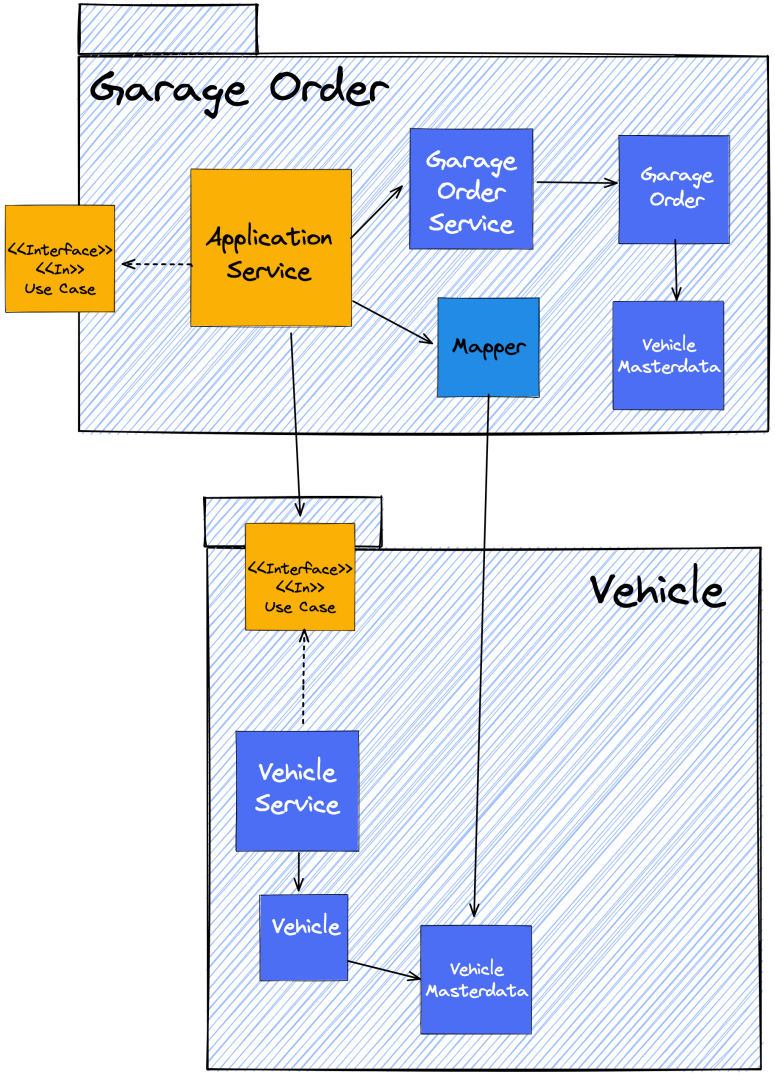
Coupling



low high

NOVATEC

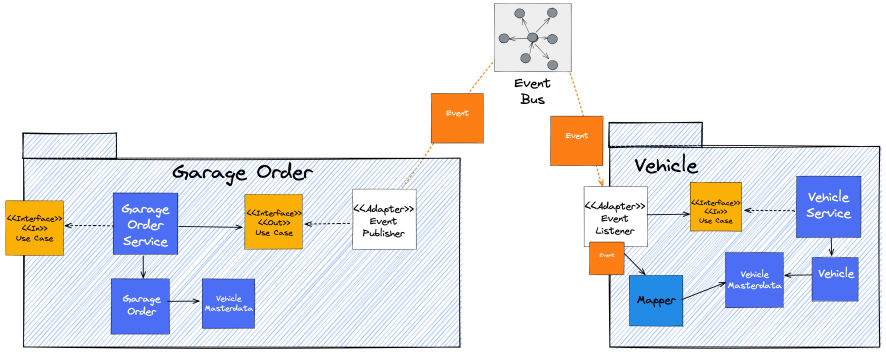
Application Service



Publish / Subscribe with Domain Events

Intended use

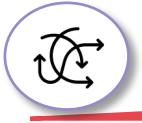
- Dependencies between domain modules (which means also between root entities) will be resolved by domain events



Implementation

- The technical coupling is shifted to pub/sub framework (e.g. spring events)
- The subscriber has to implement an input adapter and a mapper
- The publisher has to implement an output adapter and a mapper

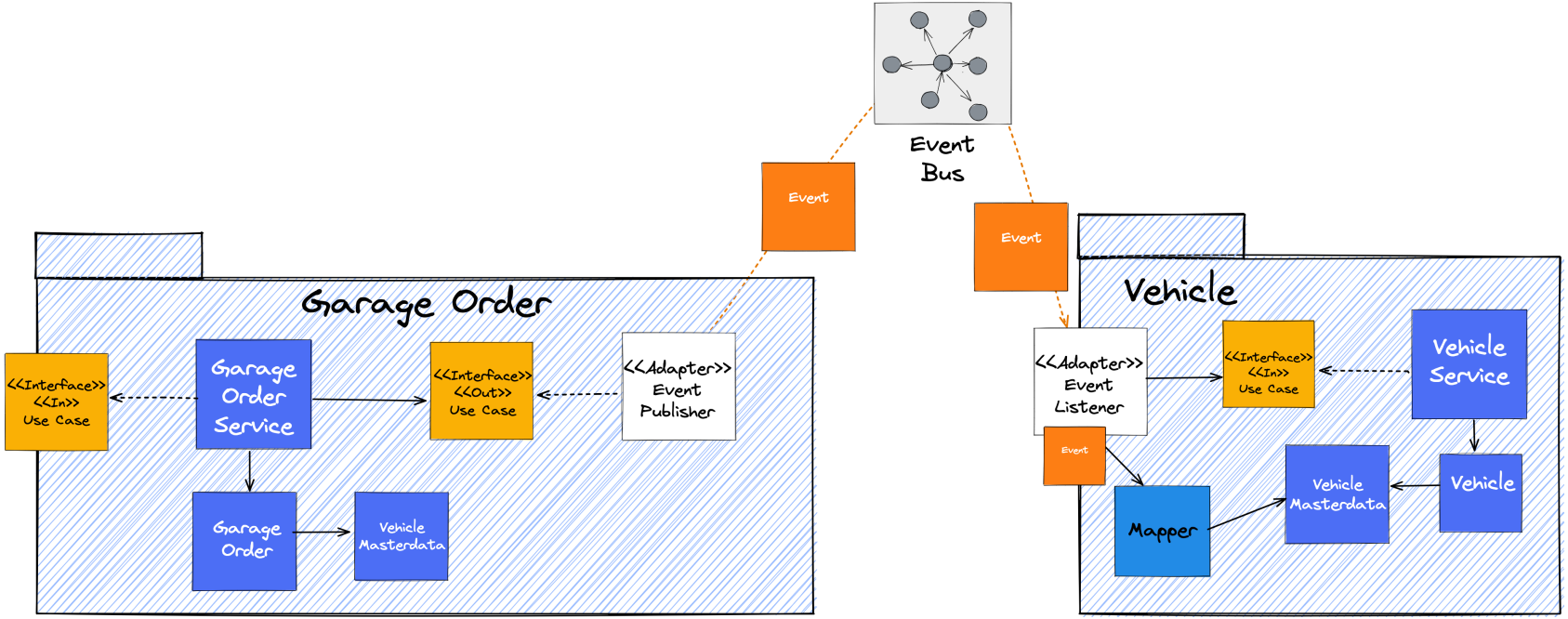
Coupling



low

high

Publish / Subscribe with Domain Events



NOVATEC

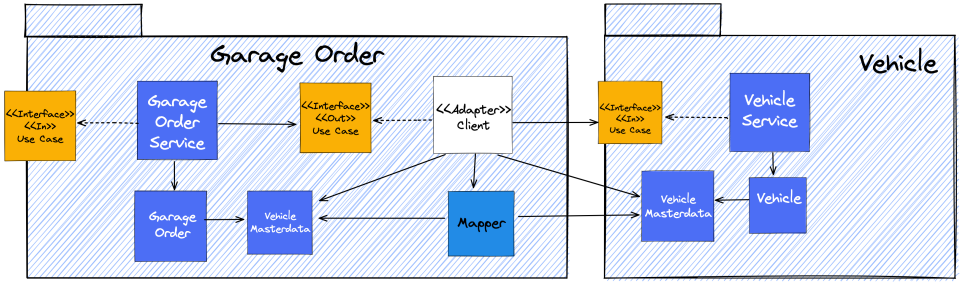
Adapter.Out – UseCase.In

Intended use

- Dependencies between domain modules (which means also between root entities) will be resolved using the ports and adapters pattern

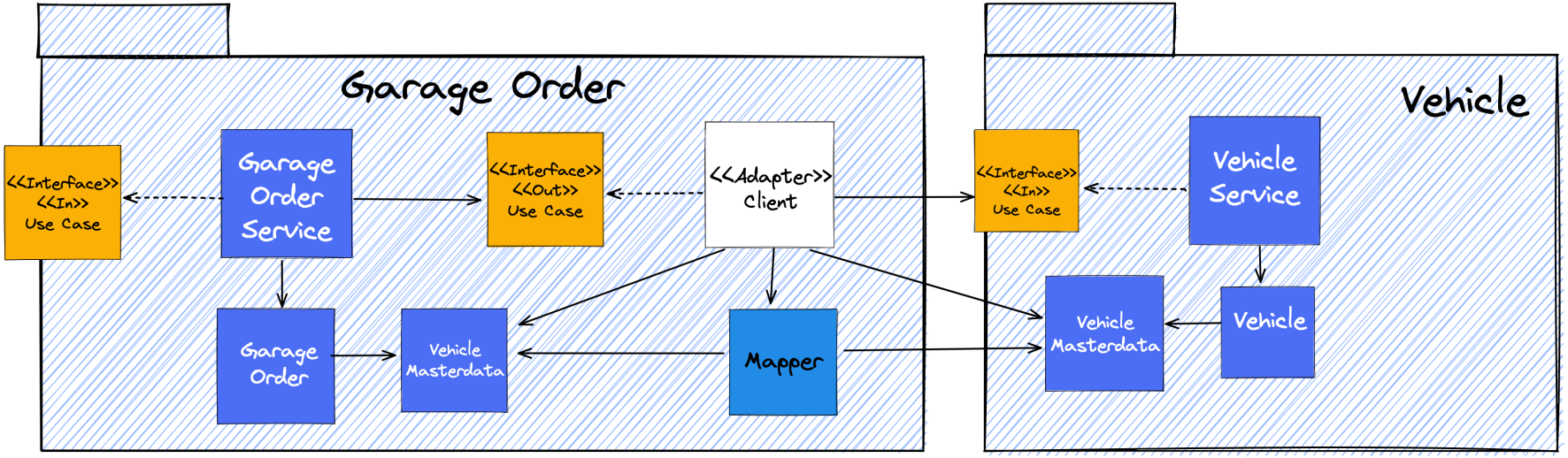
Implementation

- The technical coupling is only located in the adapter implementations
- The consumer has to implement an output adapter and a mapper
- The provider offers an exclusive use case (input port) for the consumer

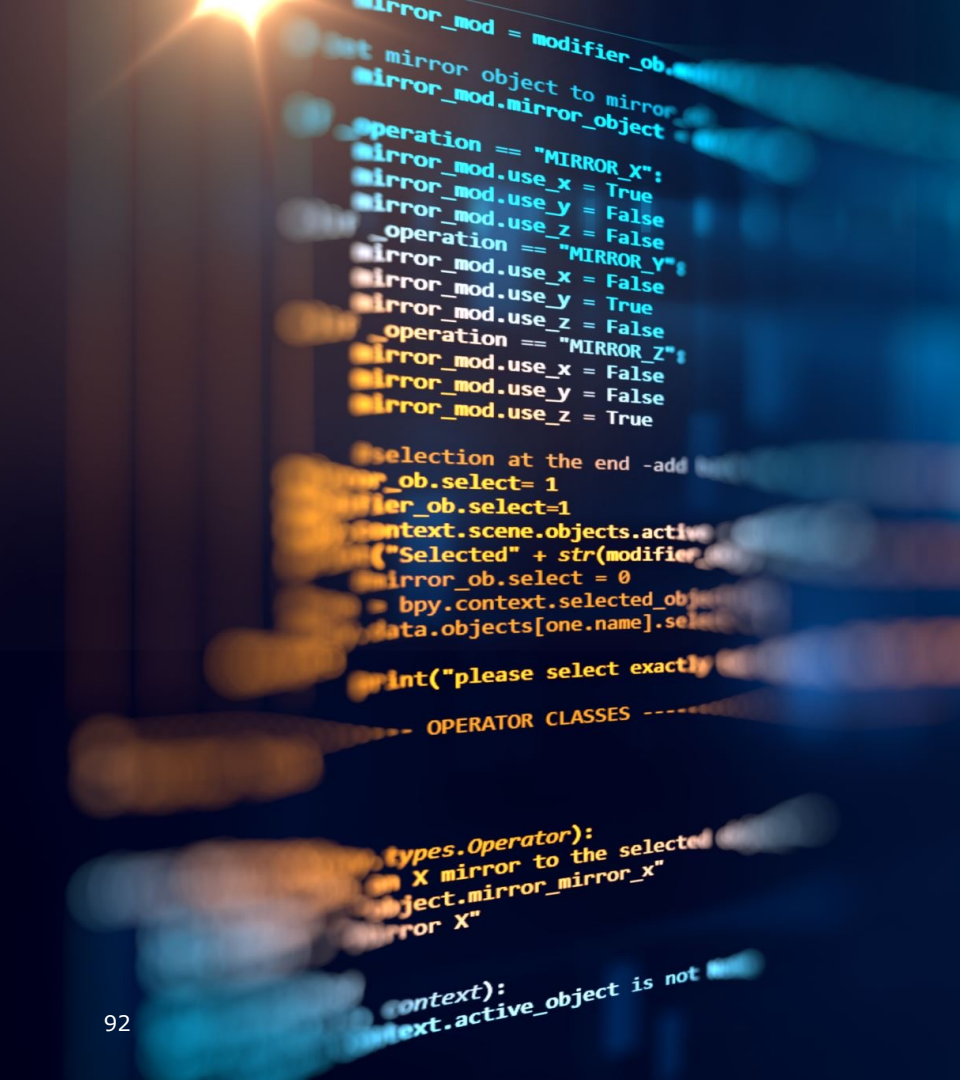


NOVATEC

Adapter.Out – UseCase.In



NOVATEC



Coding Challenge

- Lab 5
- Group discussion
- Lab 6
- Group discussion

NOVATEC

Evolutionary Architecture and Fitness Functions

Broken Windows Theory

As soon as something looks run-down, damaged, [insert negative adjective here], or generally untended, the human brain feels that it's OK to make it more run-down, damaged, or [insert negative adjective here].

- Hombergs [1]

https://en.wikipedia.org/wiki/Broken_windows_theory

Broken Windows Theory

- In a neighborhood where vandalism is common, the threshold to loot or damage an untended car is low.
- When a car has a broken window, the threshold to damage it further is low, even in a "good" neighborhood.
- In an untidy bedroom, the threshold to throw our clothes on the ground instead of putting them into the wardrobe is low.
- In a group of people where bullying is common, the threshold to bully just a little more is low.
- When working on a low-quality code base, the threshold to add more low-quality code is low.
- When working on a codebase with a lot of coding violations, the threshold to add another coding violation is low.
- When working on a codebase with a lot of shortcuts, the threshold to add another shortcut is low.

- Hombergs [1]

***Starting clean** is a **key success factor**!*

A Clean Architecture Helps to Ensure the Equilibrium

- The software development ecosystem consists of all the tools, frameworks, libraries, and best practices […].
- **This ecosystem forms an equilibrium -** much like a biological system - that developers can understand and build things within.
- However, that **equilibrium is dynamic** - **new things** come along constantly, **initially upsetting the balance** until a **new equilibrium emerges**.
- […] In the software development ecosystem, each **new innovation** or **practice** may **disrupt the status quo**, **forcing** the establishment of a **new equilibrium.**

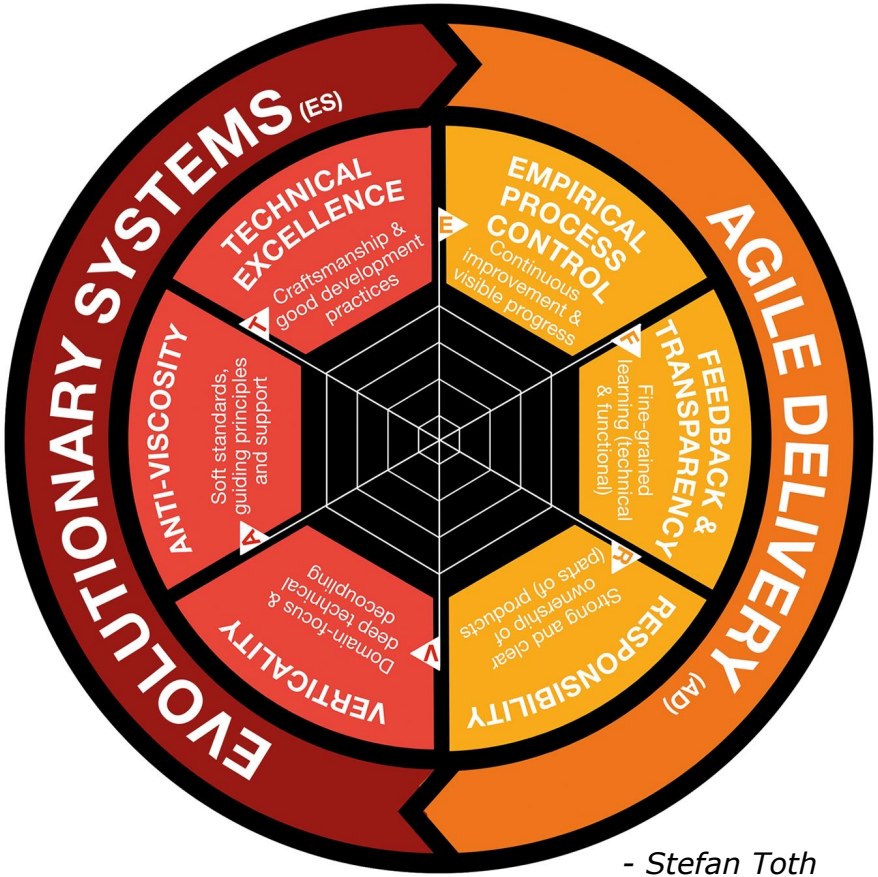
- Pearson, Ford, Kua [6]

NOVATEC

The Mindset of Evolutionary Architecture

*An evolutionary architecture supports **guided**, **incremental** change across multiple dimensions.*

- Pearson, Ford, Kua [6]



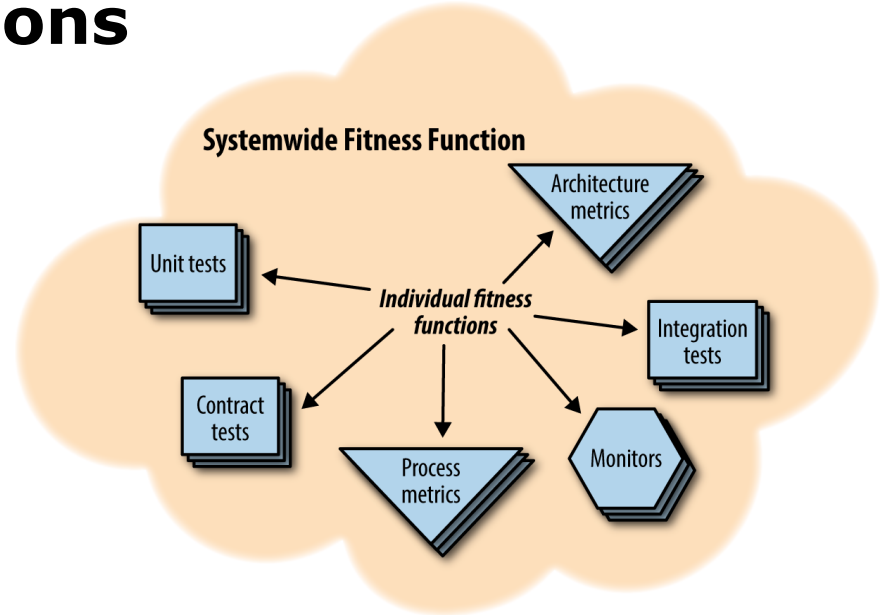
- Stefan Toth

https://kiosk.entwickler.de/java-magazin/java-magazin-11-2019/bewegung-im-architektur-genpool/

The Mindset of Evolutionary Architecture Enabled by Fitness Functions

An architectural fitness function provides an **objective integrity assessment** of some **architectural characteristic(s)**.

- Pearson, Ford, Kua [6]



Fast feedback for each increment if the state of quality changed (positive or negative)!

Continuously, fine-grained and automated

NOVATEC

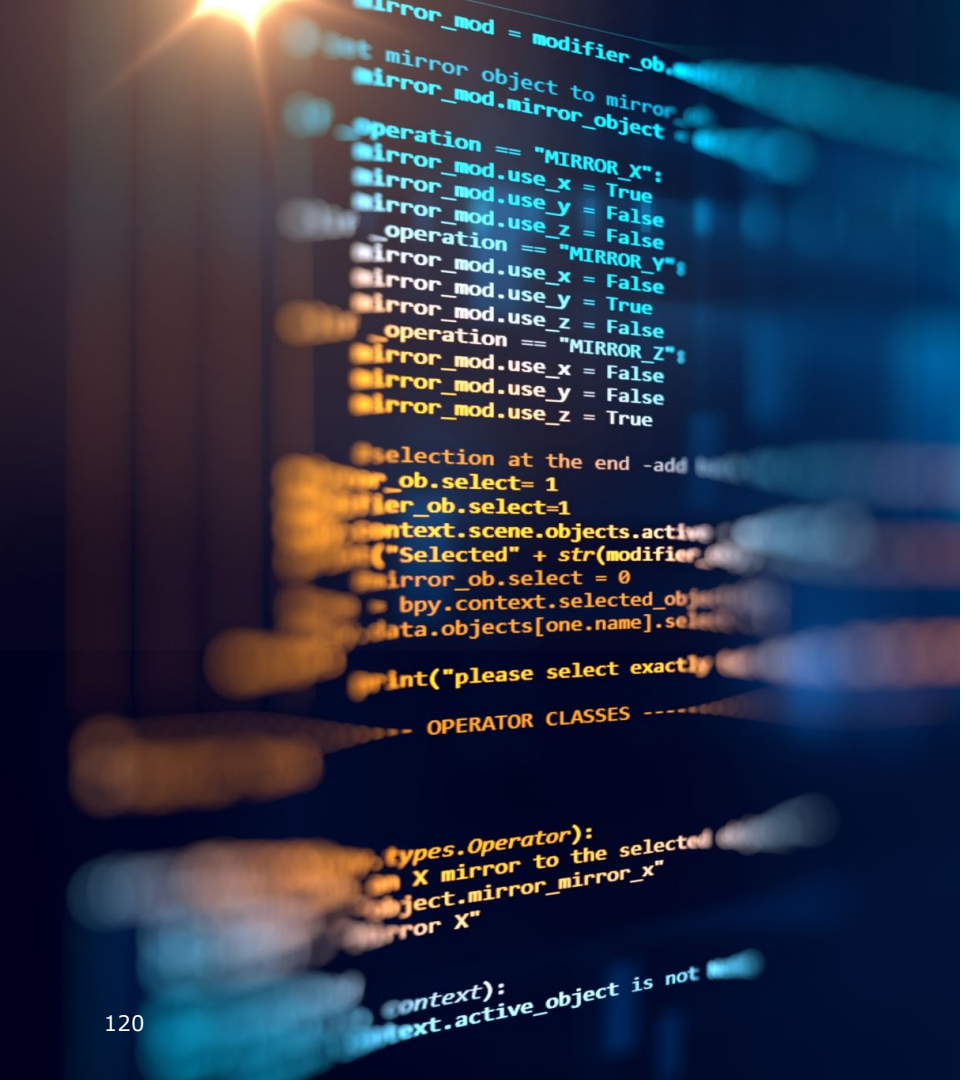
Evolvability is more than flexibility. A flexible software architecture is a crucial foundation that could also only be reached sustainability based on fitness functions, which ensures the health of the application architecture by continuously inspecting and adapting the implementation based on the rules of the used architecture pattern and pattern language.

Dependency Rules of Clean Architecture Pattern

Modul	Modularisierungsvariante	Beziehungsregeln
Domain.Service	Architektonisch ausdrucksstarke Ringe Architektonisch ausdrucksstark in der Domäne	Klassen des Moduls *Domain.Service* dürfen optional ein Interface des Moduls *UseCase.In* implementieren und Klassen der Module *UseCase.Out* und *Domain.Model* nutzen.
Domain.Model	Architektonisch ausdrucksstarke Ringe Architektonisch ausdrucksstark in der Domäne	Klassen des Moduls *Domain.Model* dürfen keine Klassen anderer Module nutzen.
Domain	Ringe als Schichten	Klassen des Moduls *Domain* dürfen nur Klassen des Modul *UseCase* nutzen.
Adapter.In	Architektonisch ausdrucksstarke Ringe	Klassen des Moduls *Adapter.In* dürfen nur Klassen der Module *UseCase.In* und *Domain.Model* nutzen.
Adapter.Out	Architektonisch ausdrucksstarke Ringe	Klassen des Moduls *Adapter.Out* dürfen optional ein Interface des Moduls *UseCase.Out* implementieren sowie Klassen des Moduls *Domain.Model* nutzen.
Adapter	Architektonisch ausdrucksstark in der Domäne	Klassen des Moduls Adapter dürfen nur Klassen der Module *Domain.Model, UseCase.In* und *UseCase.Out* nutzen.
Adapter	Ringe als Schichten	Klassen des Moduls *Adapter* dürfen Klassen der Module *UseCase* und *Domain* nutzen.
UseCase.In	Architektonisch ausdrucksstarke Ringe Architektonisch ausdrucksstark in der Domäne	Klassen des Moduls *UseCase.In* dürfen nur Klassen des Moduls *Domain.Model* referenzieren.
UseCase.Out	Architektonisch ausdrucksstarke Ringe Architektonisch ausdrucksstark in der Domäne	Klassen des Moduls *UseCase.In* dürfen nur Klassen des Moduls *Domain.Model* referenzieren.
UseCase	Ringe als Schichten	Klassen des Moduls *UseCase* dürfen nur Klassen des Moduls *Domain* referenzieren.

Tabelle 1: Beziehungsregeln der Module in der Clean Architecture

https://entwickler.de/reader/reading/java-magazin/11.2022/c471d56dececbf67d54169ee



Code Demo

- ArchUnit
- Clean Architecture Fitness Functions

NOVATEC

Bibliography

1. Get your hands dirty on clean architecture
   [Tom Hombergs]
2. Hands on Domain-driven Design
   [Michael Plöd]
3. Clean Architecture
   [Robert C. Martin]
4. Hexagonal Architecture
   [Alistar Cockburn]
5. Suistainable Software Architecture
   [Carola Lilienthal]
6. Building Evolutionary Architecture
   [Rebecca Pearson, Neal Ford, Pat Kua]
7. Dependency Inversion and Dependency Injection
   [Martin Fowler]
   https://martinfowler.com/articles/dipInTheWild.html &
   https://martinfowler.com/articles/injection.html
8. https://www.archunit.org/

GitHub

- https://github.com/MatthiasEschhold/clean-architecture-and-flexibility-patterns
- https://github.com/MatthiasEschhold/clean-architecture-csharp-demo
- https://github.com/MatthiasEschhold/clean-architecture-java-demo
- https://github.com/MatthiasEschhold/domain-driven-design-werkstatt
- https://github.com/MatthiasEschhold/clean-architecture-fitness-functions
- https://github.com/domain-primitives

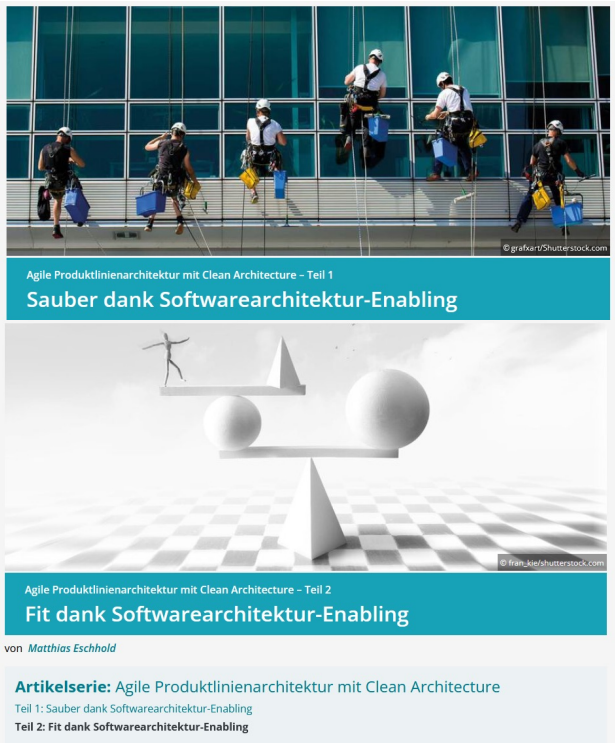
NOVATEC

Articles

Flexible Anwendungsarchitektur mit der Clean Architecture – Teil 1: Die Basis für mehr Flexibilität

https://github.com/MatthiasEschhold/clean-architecture-and-flexibility-patterns

Java Aktuell (02/22, 03/22 & 04/22)



Windows Developer (9/22 & 10/22)

Java Magazin (10/22 & 11/22)

NOVATEC

NOVATEC

Matthias Eschhold
Managing Consultant

0170/1166664
matthias.eschhold@novatec-gmbh.de

Novatec Consulting GmbH
Bertha-Benz-Platz 1
D-70771 Leinfelden-Echterdingen

T. +49 711 22040-700
info@novatec-gmbh.de
www.novatec-gmbh.de