Hexagonal Architecture

Łukasz Soszyński

Read Me

- Łukasz Soszyński
  - Spring
  - Clean Code
  - TDD
  - DDD
  - Functional Programming
  - Reactive Programming
- Expirience: 9 years of professional programming in Java
- Impaq: CC
  - Leader: Łukasz Ciechanowski
  - Java / Ruby
  - DevOps
  - Android

Agenda

- Introduction: The most important part in Your application
- Dependency Inversion Principle on example of N layer architecture
- Hexagonal architecture
- Examples

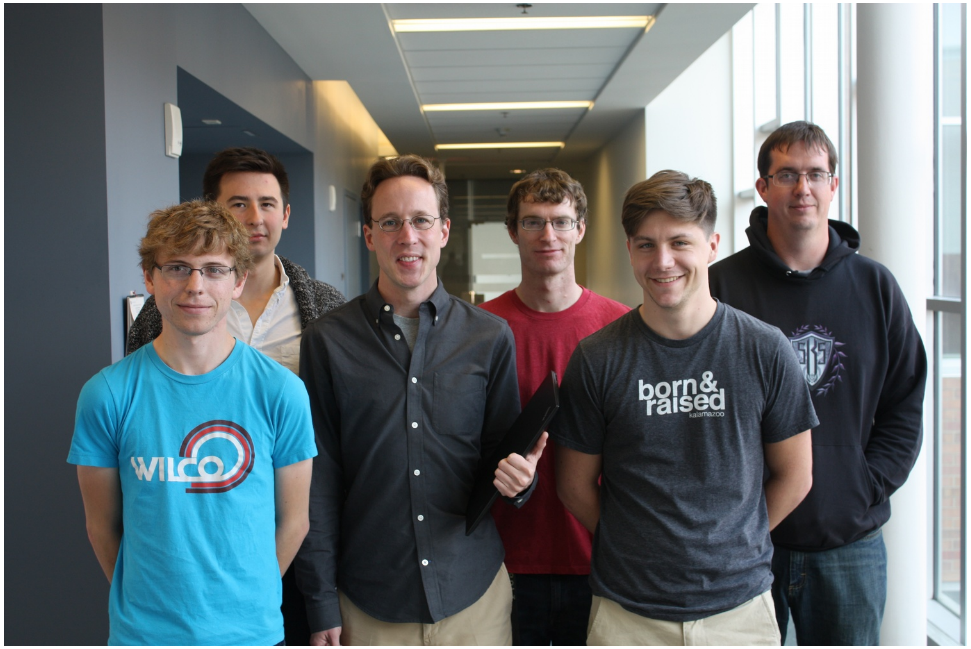
Introduction

Łukasz Soszyński

What is architecture for

Do I need any architecture?

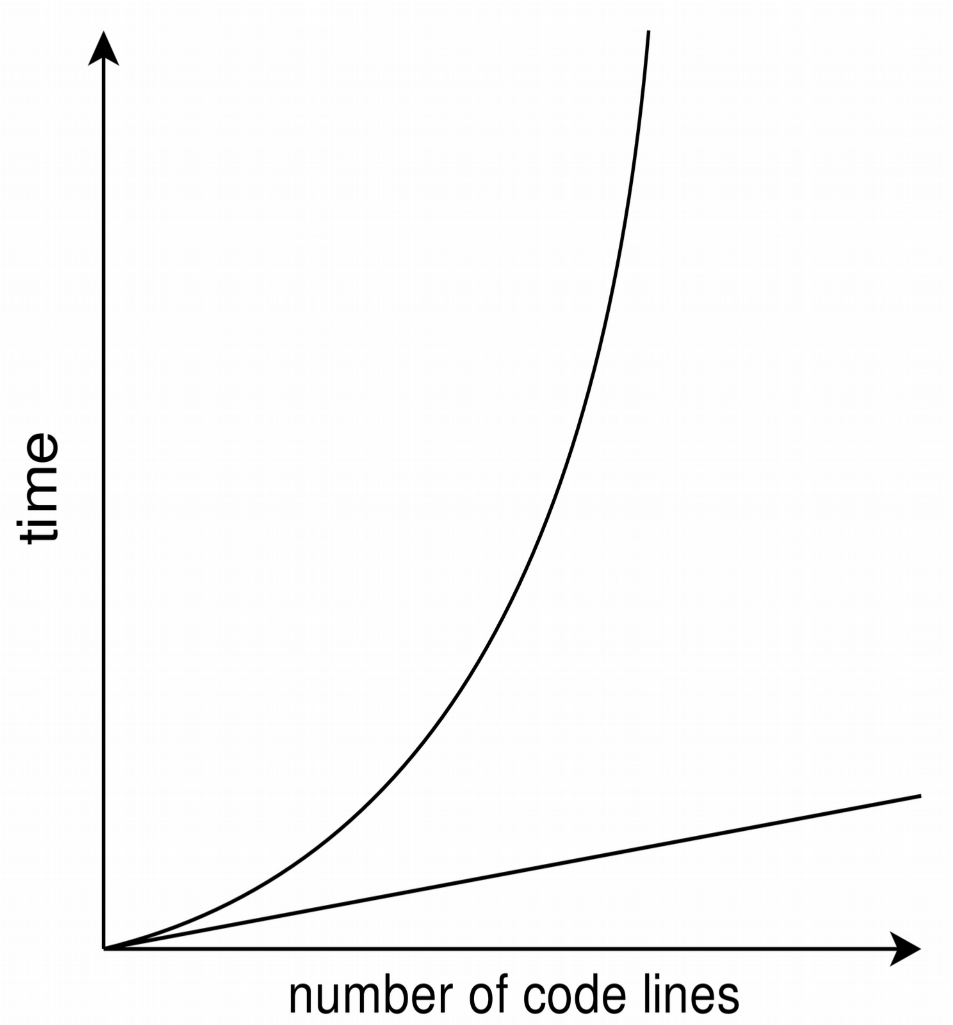
We need small additional checkbox



100 story points/mdays

Goals of software architecture

- Easy changes
  - Lowest development cost
- Maintability
- Reusability



The goal of software architecture is to minimize the human resources required to build and maintain the required system*

Because software is not a hardware...

*Clean Architecture, Robert C. Martin

What is not architecture

- Database
- Framework
- Libraries
- Deployment structure

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.*

Architecture is everything what is expensive to change.

* IEEE 1471

Good Architecture

- Good architecture leaves as many options open as possible, for as long as possible.*
- Reduce software development and maintainmance cost

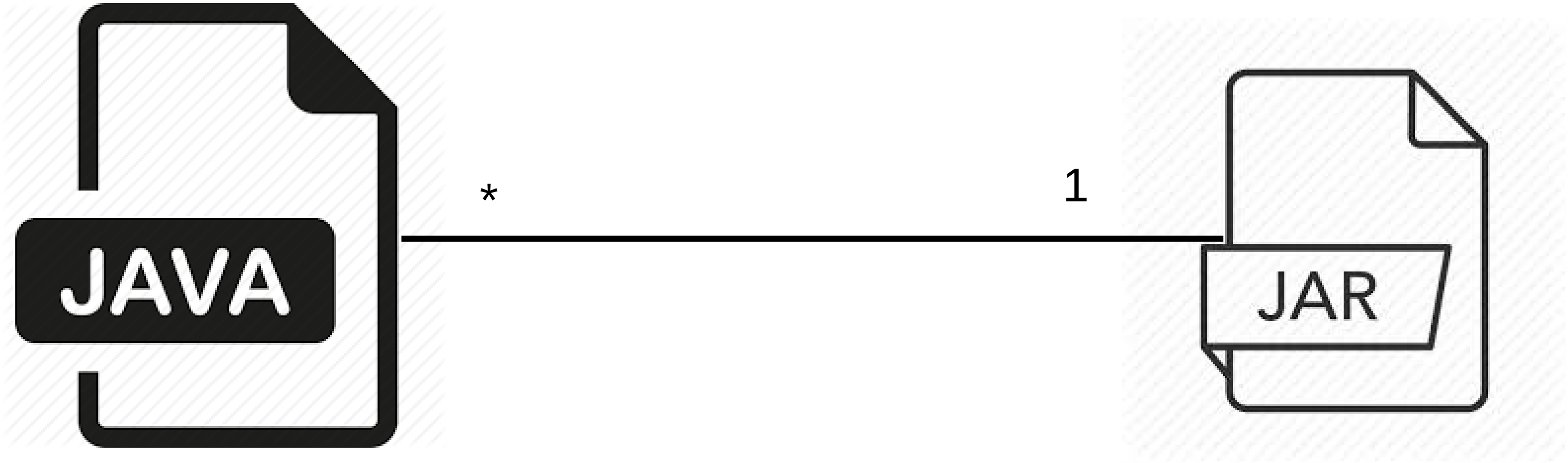
- Does not impose usage of
  - Database
  - Framework
  - Library
  - Deployment structure

*Clean Architecture, Robert C. Martin

Glossary

- Module:
  - ... is just a source file*
  - file with extension „java”
  - module is just a cohesive set of functions and data structures*

- Component:
  - Components are the units of deployment*
  - file with extension „jar”



*Clean Architecture, Robert C. Martin

Glossary

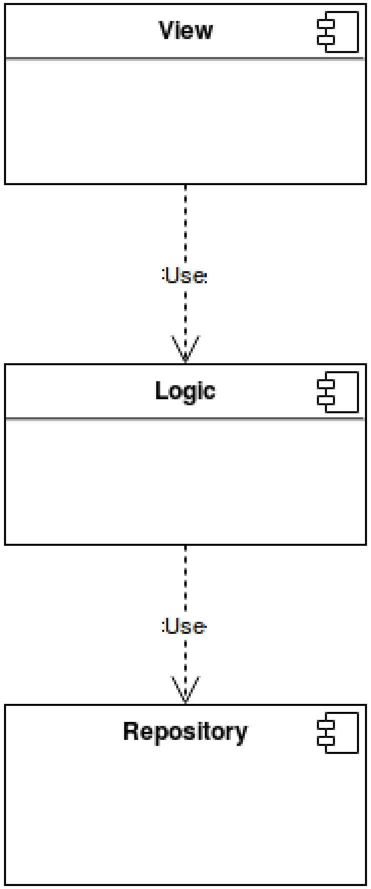
- Business Logic
- Domain Model
- Model

Glossary

Component View

Component Logic

Component Repository



```xml
<project>
    <groupId>com.impaqgroup</groupId>
    <artifactId>View</artifactId>
    <dependencies>
        <dependency>
            <groupId>com.impaqgroup</groupId>
            <artifactId>Logic</artifactId>
        </dependency>
    </dependencies>
</project>
```

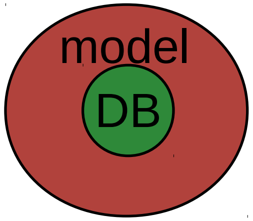
```xml
 <project>
     <groupId>com.impaqgroup</groupId>
     <artifactId>Logic</artifactId>
     <dependencies>
         <dependency>
             <groupId>com.impaqgroup</groupId>
             <artifactId>Repository</artifactId>
         </dependency>
     </dependencies>
 </project>
```

Old trusty software design (I)

Step 1

- Software design starts from database level
  - Database model is prepared

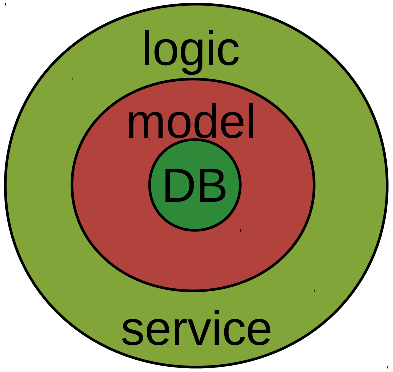
Old trusty software design (II)



Step 2

- Create anemic model which corresponds with database table

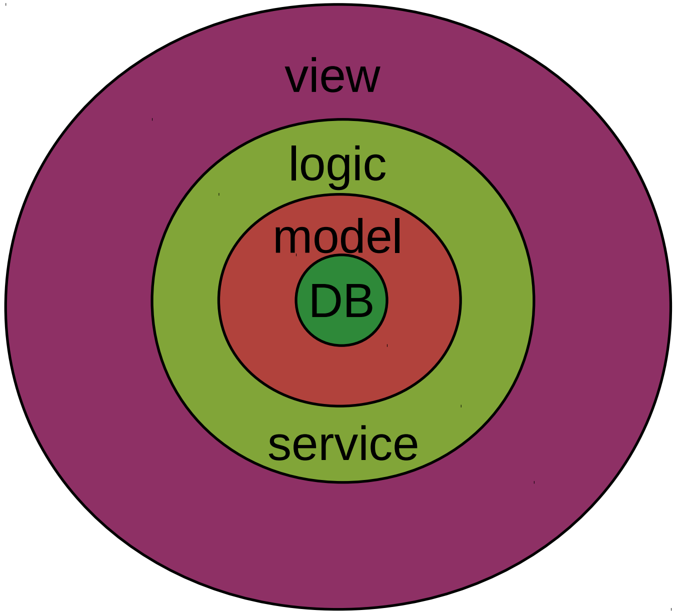
Old trusty software design (III)

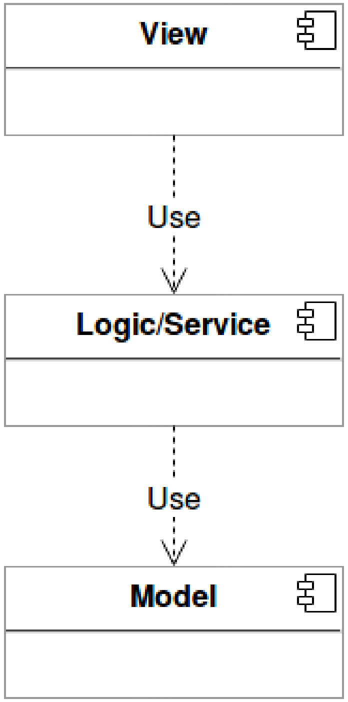


Step 3

- Create service layer which incorporates business logic

Old trusty software design (IV)

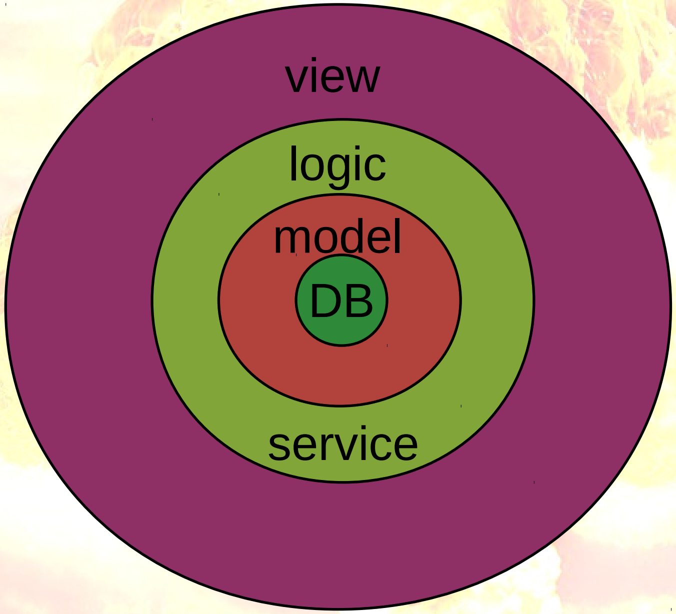




Step 4

- Ceate GUI (JSP/JSF/GWT/Vaadin/etc.)

Old trusty software design (V)



Step 5

- Maintenance
  - Relational database become performance bottleneck
  - Task: Use noSql database instead of relational one

Old ~~trusty~~ software design (VI)

Old trusty software design drawbacks

- Application is build around database
  - Database is most important in design
- Every component depends on (directly or indirectly) on database
- Database cannot be replaced quickly and cheaply

Technical details

- Good software design does not depends on technical details like:
  - Databases
  - Frameworks
  - Libraries
  - Deployment structure
- Good software design allow replacement of technical details

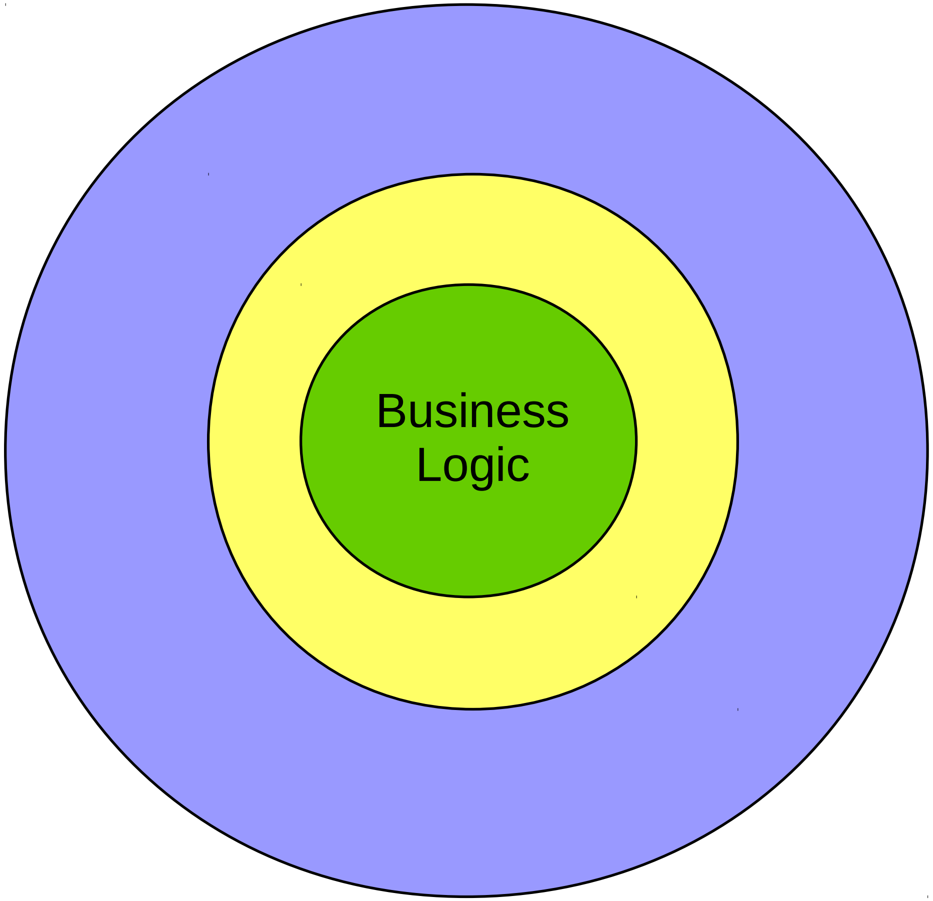
Bussines logic

- Domain model which contais **business logic is the most important part of applicaion**
- It is unlikely to replace busineslogic
  - eg. Turn anti-fraud system into pizza ordering system
- Good software design should be focused on business logic to ensure:
  - Extensibility
  - Maintainability
- It is probable to replace technical details of application
  - eg. database

Dependency inversion principle

- Can be used to protect business logic against polution
  - So that business logic does not depend on implementation details
- Relevant ingredient of:
  - Three layer architecture
  - Hexagonal Architecture

Most important part of application



Dependency inversion principle

Based on Tree Layer Architecture

Tier „atchitecture” evolution

Mainframe

One-tire

PC

Two-tire

Modern server

Three-tire

Evolution

One-tire „architecture”

- Time of mainframes
- Whole application is located on a single machine
- Access via simple text terminals

Two-tire „architecture”

- Time of PCs
- Client server architecture
- Logic is executed on client
- Server is responsible for data storage

Three-tier „architecture”

View

Logic

Data

Physical separation on three deployment units:

- View
  - Web Browsers
  - Mobile devices
- Logic
- Data

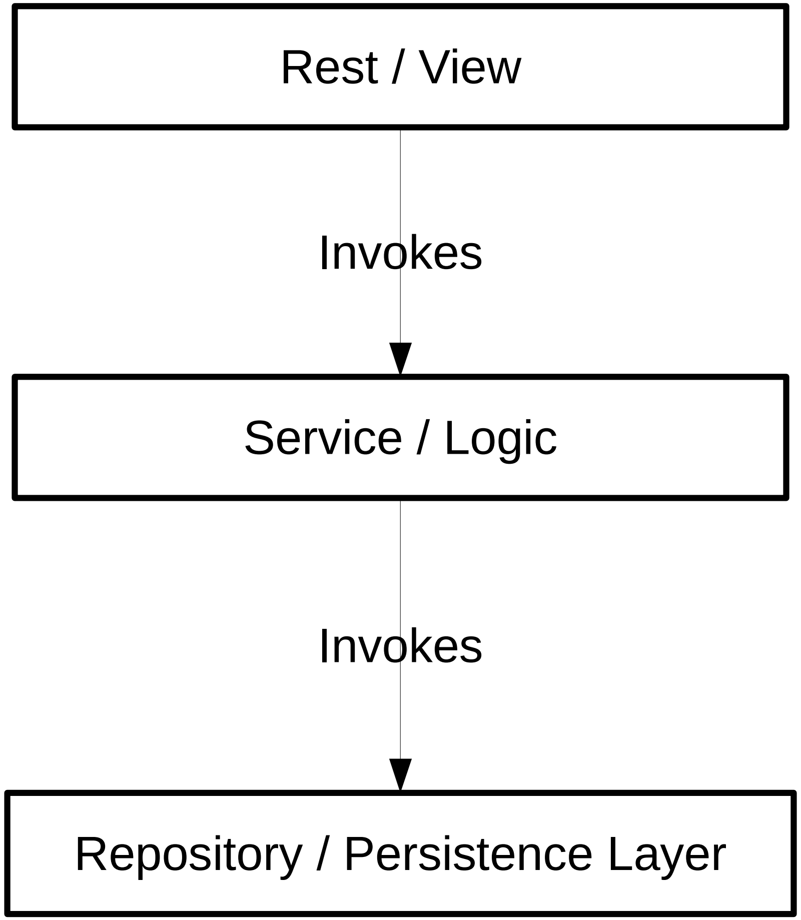
Tree Layer Architecture

Business Tier

- **Logical** separation
  - View
  - Logic
  - Data
- Why application is split into three layers?

Three layers architecture

Busines Tier



Rest / View

Invokes

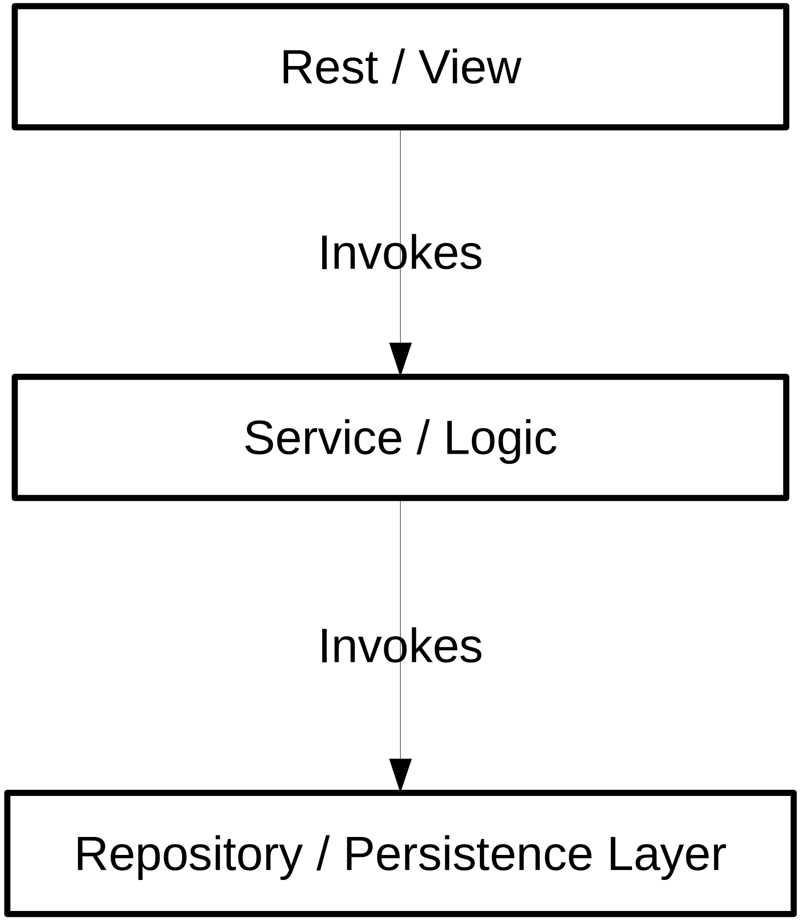
Service / Logic

Invokes

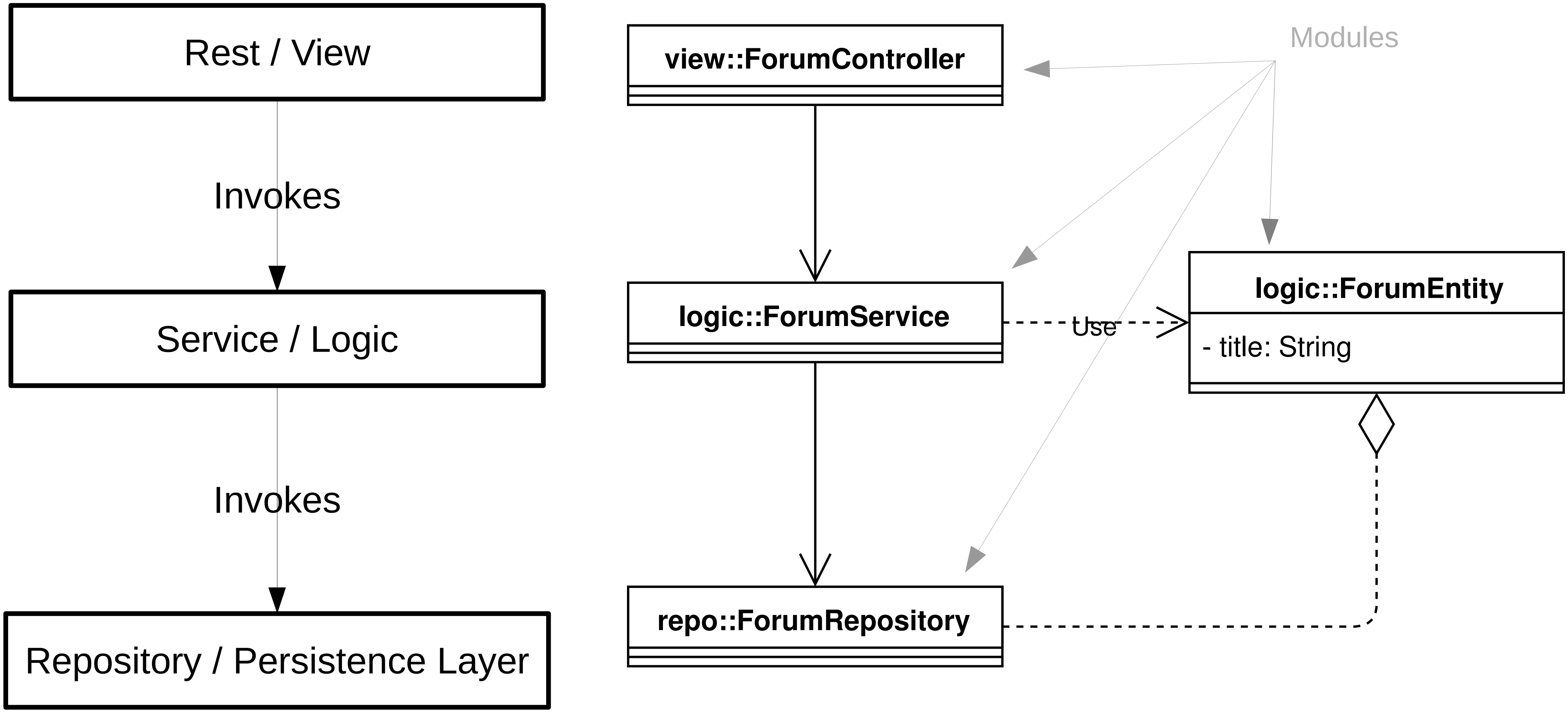
Repository / Persistence Layer

Three layers architecture

- View layer is responsible for invocation of business logic
- Business Logic layer is responsible for invocation of repository



Three layers architecture. Implementation (I)



All application modules can be placed in a single component (a deployment unit).

ForumEntity

```
public class ForumEntity {
    private String title;
    //many others
}
```

- ForumEntity is part of damain model
- Contains businesslogic:
  - Creating new subforum
  - Creating new post
  - Etc.

ForumEntity – stored in database

```
public class ForumEntity {
    @Column(name = "title", length = 436, nullable = false)
    private String title;
    //many others
}
```

Forum Entity - Validated

```java
public class ForumEntity {
    @NotNull
    @Pattern(regexp = "^[A-Z]{1}[a-z ]{435}$")
    @Column(name = "title", length = 436, nullable = false)
    private String title;
    //many others
}
```

ForumEntity - JSON

```java
public class ForumEntity {
    @JsonProperty("title")
    @NotNull
    @Pattern(regexp = "^[A-Z]{1}[a-z ]{435}$")
    @Column(name = "title", length = 436, nullable = false)
    private String title;
    //many others
}
```

ForumEntity - XML

```
public class ForumEntity {
    @XmlAttribute(name = "title")
    @JsonProperty("title")
    @NotNull
    @Pattern(regexp = "^[A-Z]{1}[a-z ]{435}$")
    @Column(name = "title", length = 436, nullable = false)
    private String title;
    //many others
}
```

ForumEntity class depends on

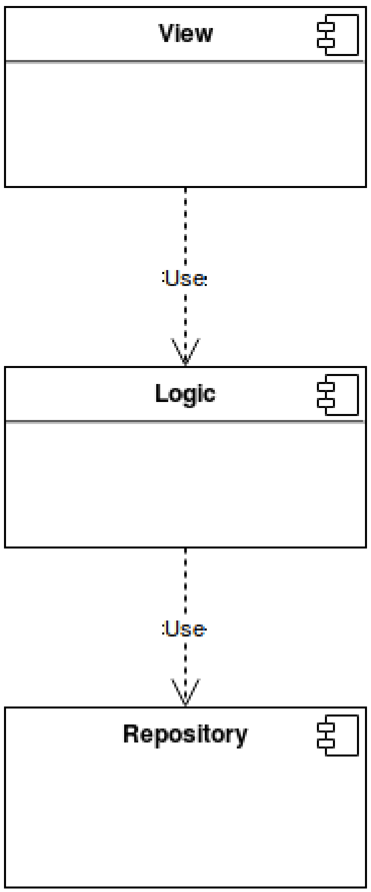
- JPA (`@Column`)
- JSR380 / Hibernate validator (`@NotNull, @Pattern`)
- Jackson (`@JsonProperty`)
- JAXB (`@XmlAttribute`)

ForumEntity

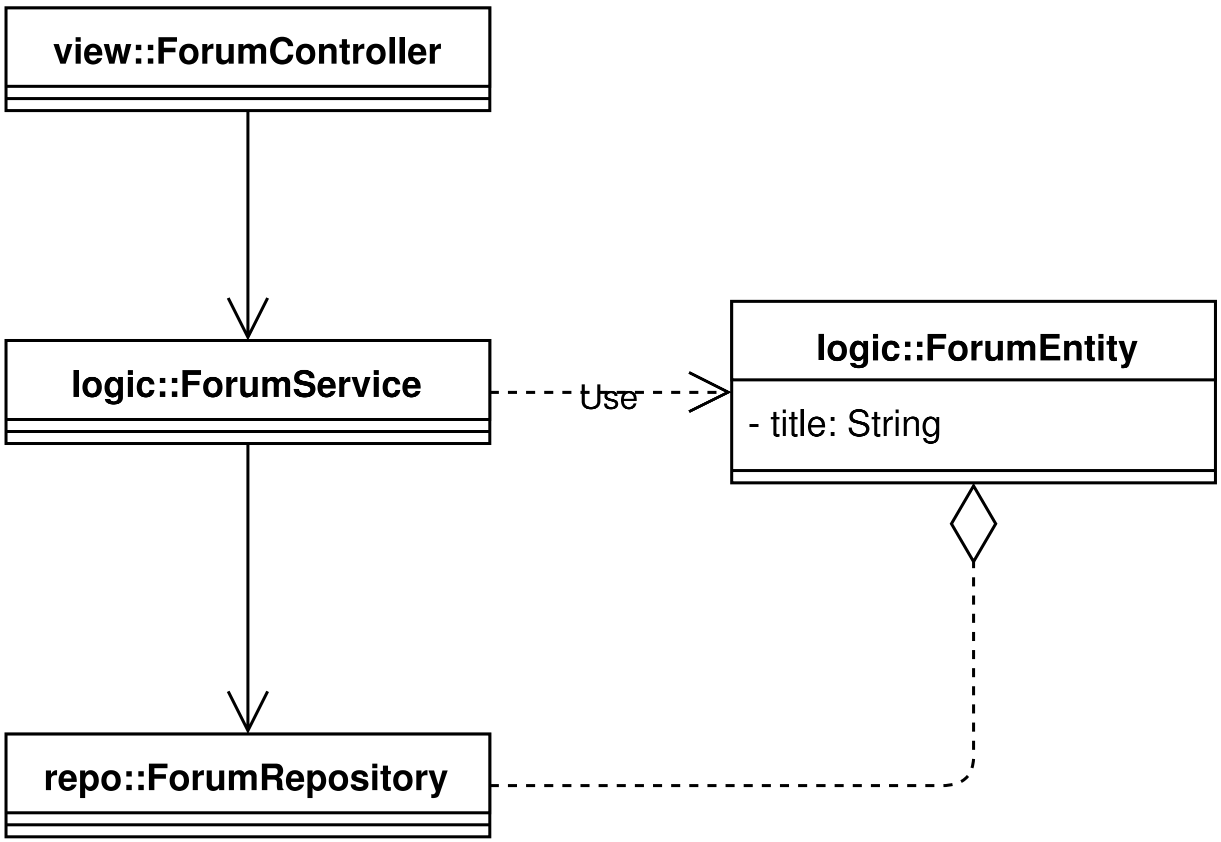
- Is a domain object
- Contains business logic
- **Should not depend on** any libraries which are **implementation details**!

Three layers architecture. Implementation (II)

Components



Modules



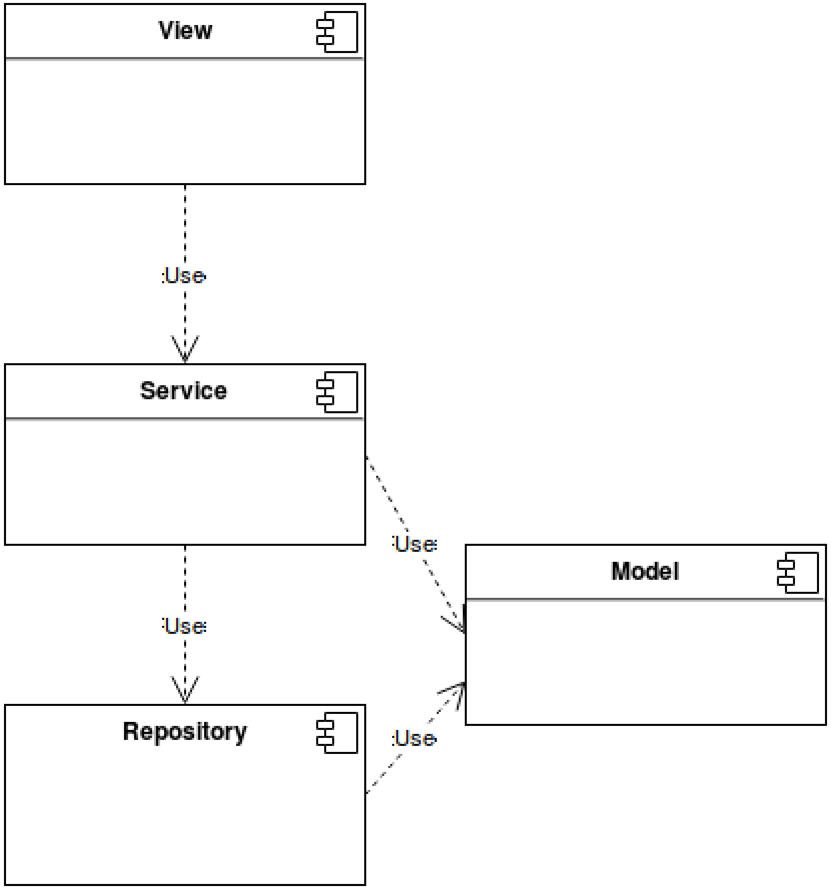
Application is split into three components

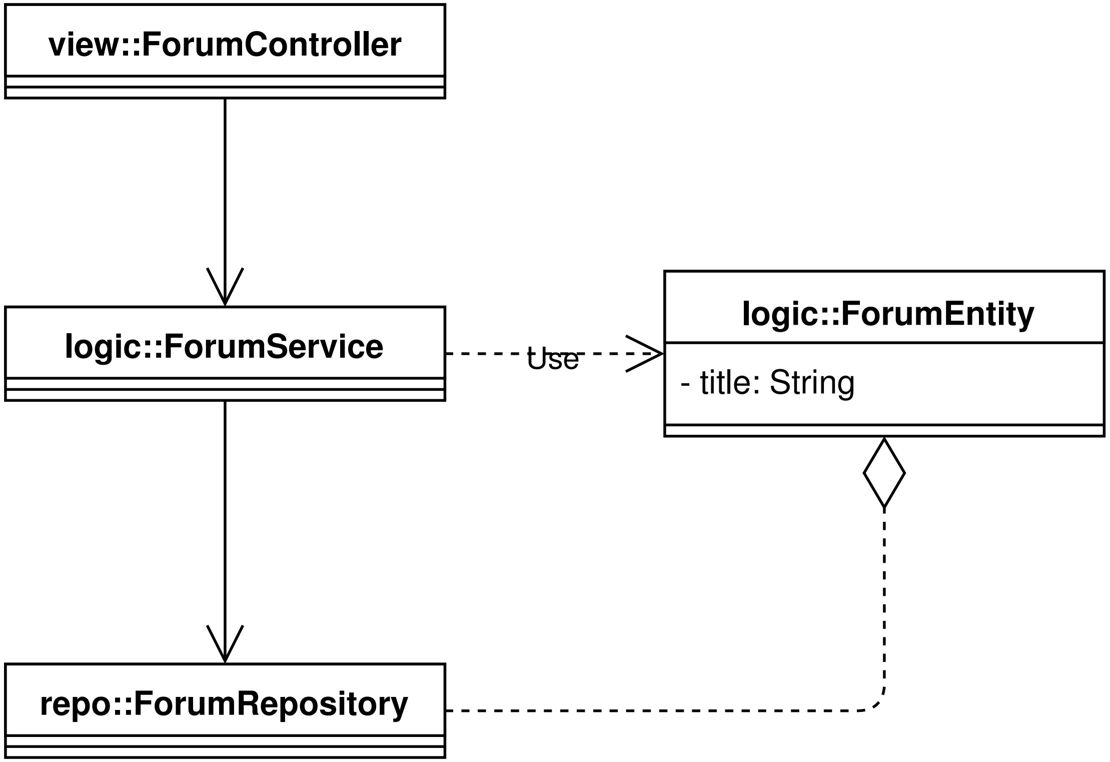
ForumEntity

```
public class ForumEntity {
    @XmlAttribute(name = "title")
    @JsonProperty("title")
    @NotNull
    @Pattern(regexp = "^[A-Z]{1}[a-z ]{435}$")
    @Column(name = "title", length = 436, nullable = false)
    private String title;
    //many others
}
```

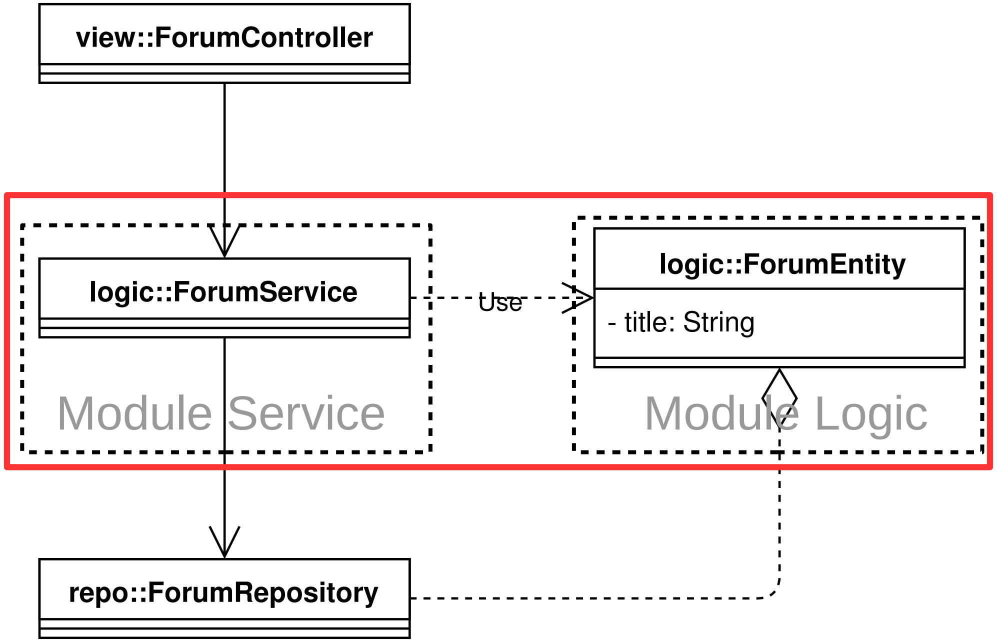
- Domain object ForumEntity can be still poluted by technical details
- Partitioning application into three componens do not improve application design

Three layers architecture. Implementation (III)



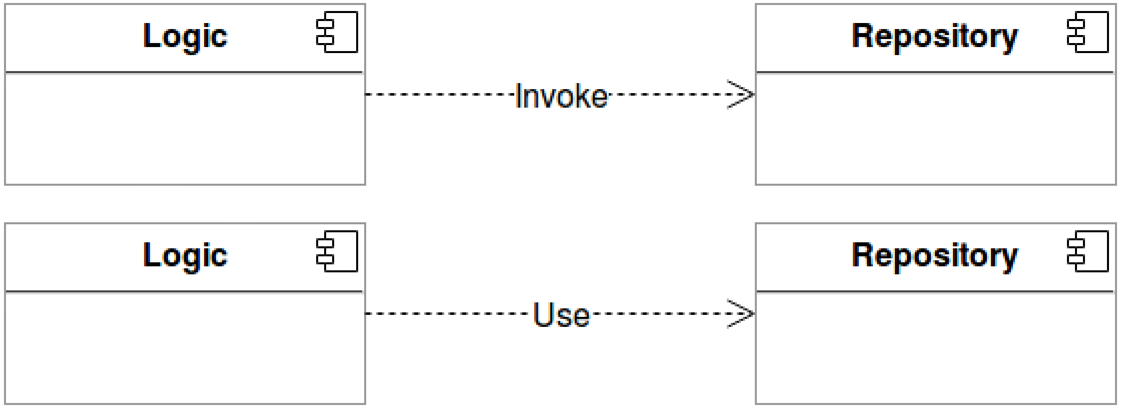


Three layers architecture. Implementation (III)



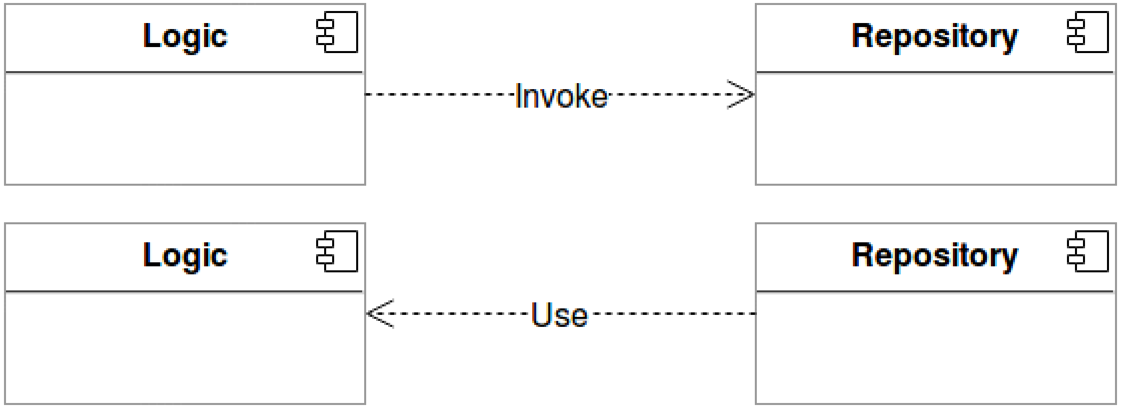
- Business logic still depends on impementation details (ForumRepository)
- Business logic is spread between two modules/components
- It is hard to write unit test in order to test business logic

Components protection (I)



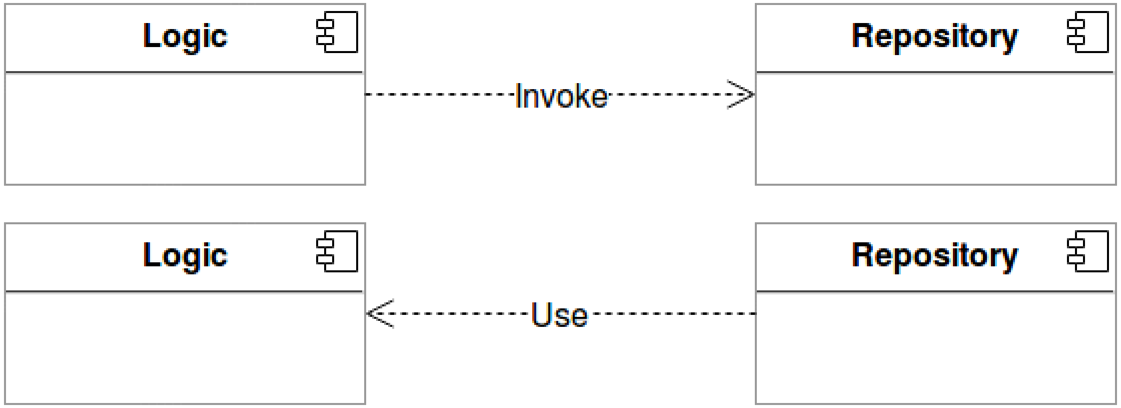
- Logic is a stable component
- Repository is an unstable component and is likely to undergo changes
    - Repository is the only example. Actually, it can be any unstable component
- Goal: **To protect stable component Logic against changes in unstable component Repository**

Components protection (II)



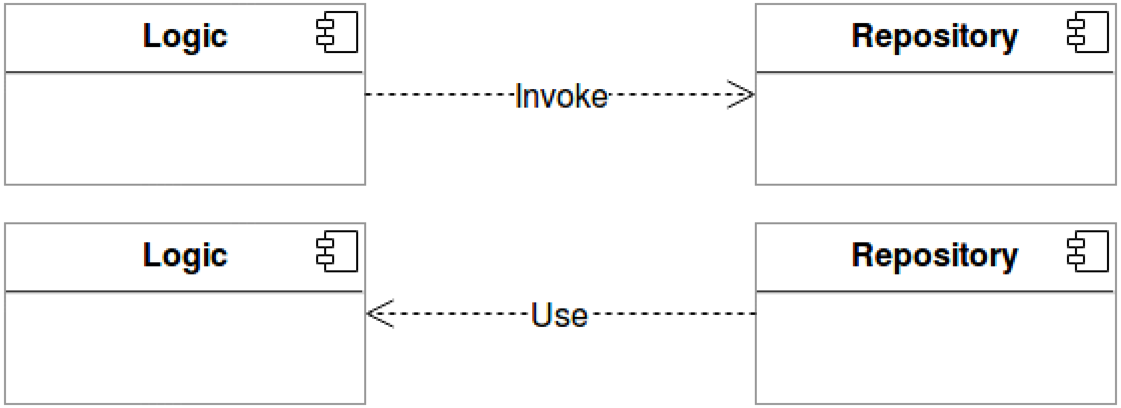
- Stable component Logic does not depend on unstable Repository component
- Changes inside Repository component do not affect Logic component
- Invocation direction is opposite to dependency direction. Thus, this principle is called:
  - **Dependency inversion principle**
  - Inversion of control

Components protection (III)



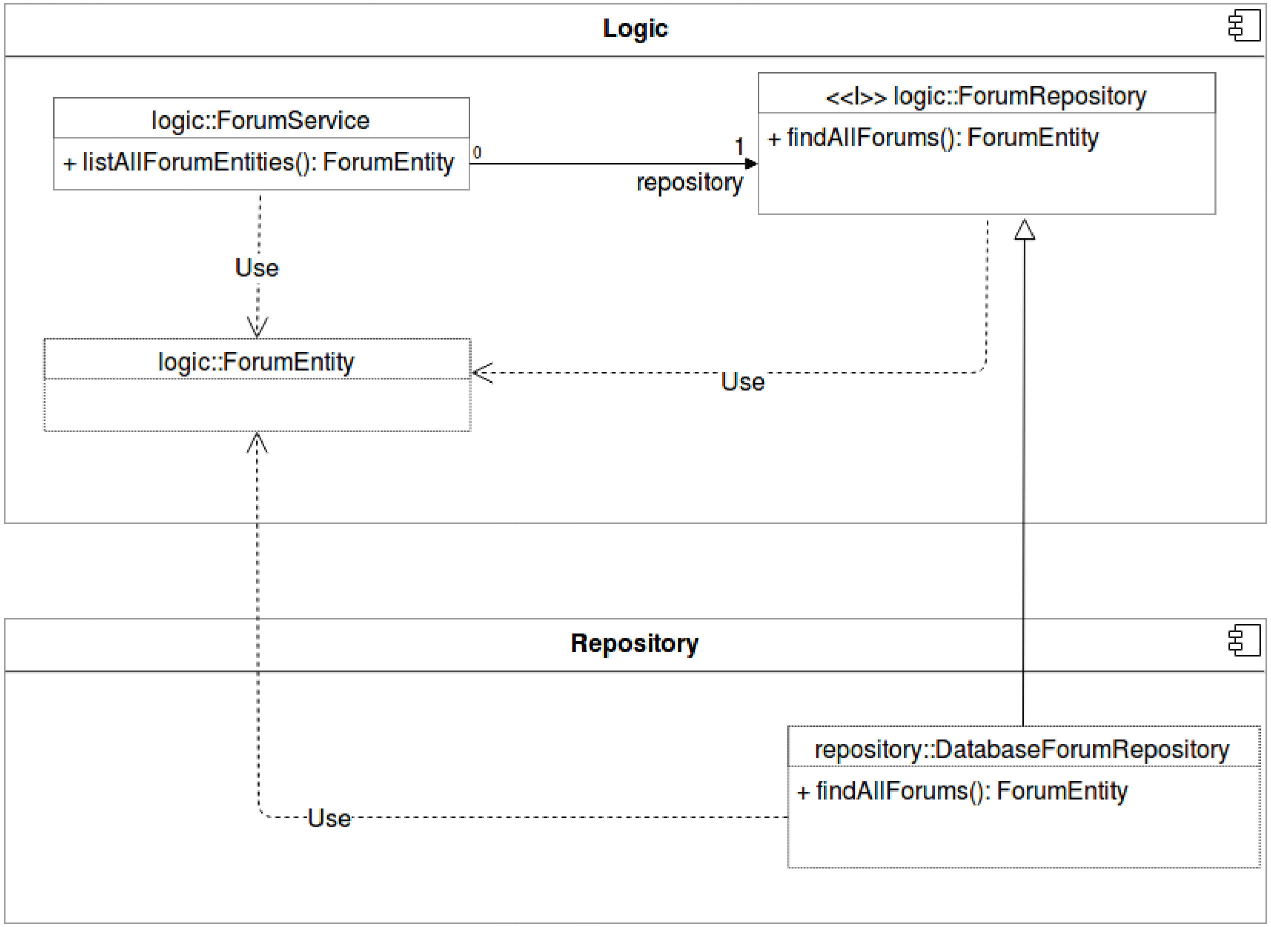
- Component Logic does not depend on Repository component
- Logic component cannot directly invoke function from Repository component

Components protection (IV)

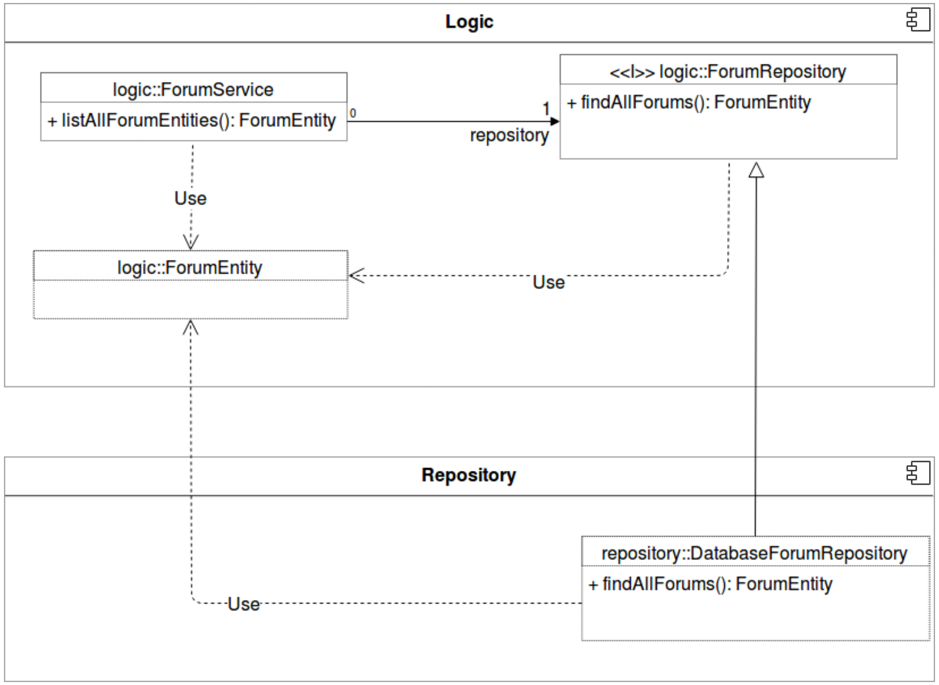


- Logic component defines interfaces which are implemented inside Repository component
- Modules (Classes) inside Logic component invoke function from Repository component using interfaces defined inside Logic component

Components protection (V)


Logic
logic::ForumService
+ listAllForumEntities(): ForumEntity
0
1
repository
<<I>> logic::ForumRepository
+ findAllForums(): ForumEntity
Use
logic::ForumEntity
Use
Repository
repository::DatabaseForumRepository
+ findAllForums(): ForumEntity
Use

Components protection (VI)


Logic
logic::ForumService
+ listAllForumEntities(): ForumEntity
0
1
repository
<<I>> logic::ForumRepository
+ findAllForums(): ForumEntity
Use
logic::ForumEntity
Use
Repository
repository::DatabaseForumRepository
+ findAllForums(): ForumEntity
Use

How to create ForumRepository implementation:

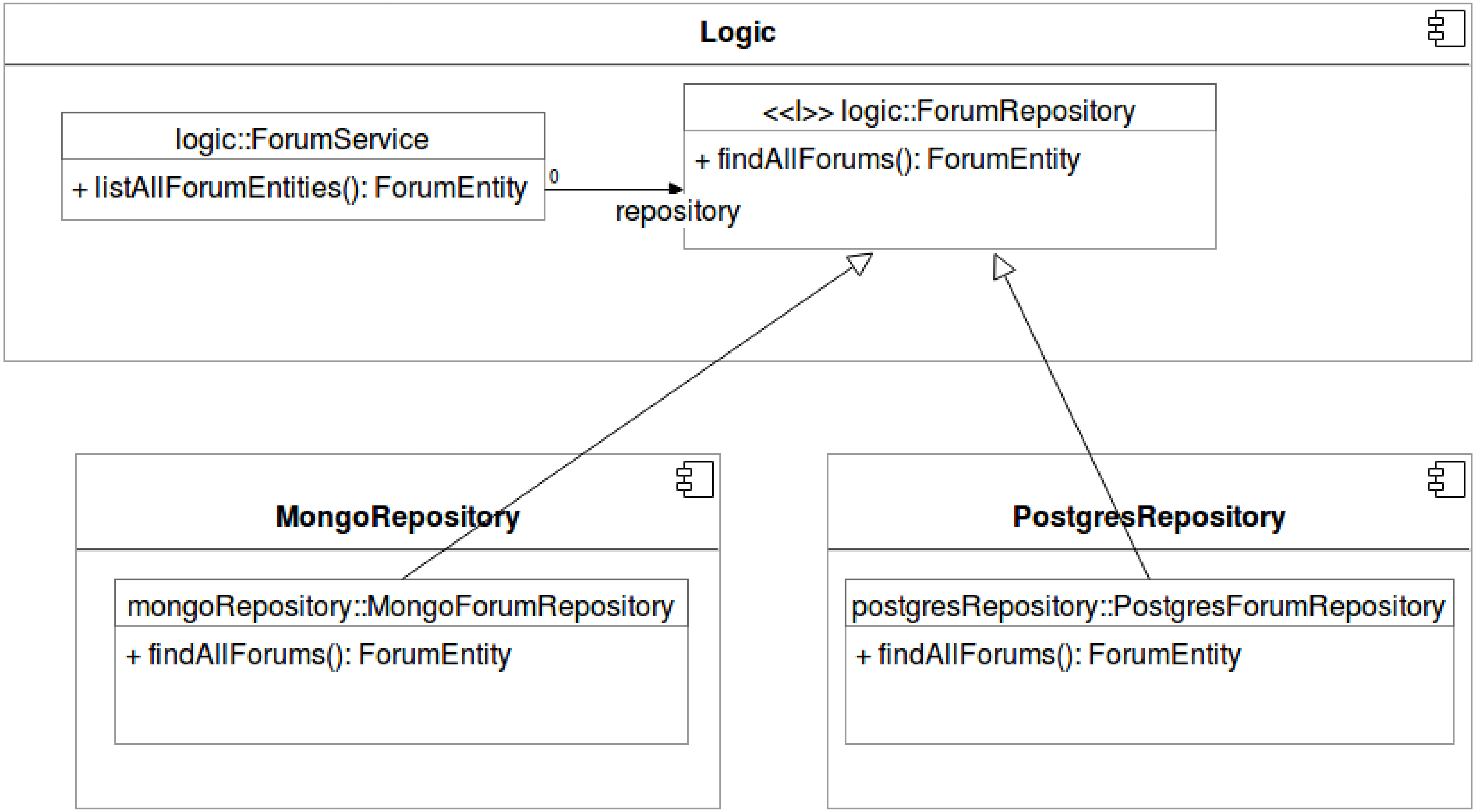
- Reflections
- Abstract Factory
- java.util.ServiceLoader
- ~~@ConponentScan~~
- Spring Boot auto configuration
- org.springframework.core.io.support.SpringFactoriesLoader
- OSGI

https://blog.frankel.ch/java-service-loader-vs-spring-factories/

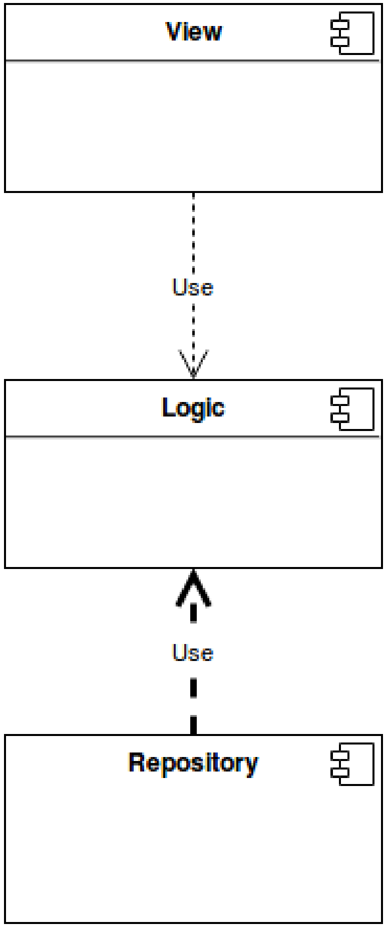
Components protection (VII)

```
public interface DomainService {}



public class DomainServiceDerivative implements DomainService {

}

DomainService s;

// Java 6-8: Reflection API
s = (DomainService) Class.forName("impaqgroup.ha.DomainServiceDerivative").~~newInstance~~();

// Java 9 Reflection API
s = (DomainService) Class.forName("impaqgroup.ha.DomainServiceDerivative").getConstructor().newInstance();

// Java 6 JDK, configuration file name:
// META-INF/services/impaqgroup.ha.DomainService
s = ServiceLoader.load(DomainService.class).stream()
        .map(Provider::get)
        .findFirst()
        .orElse(null);

// Spring 3.2
// Configuration file name:
// META-INF/spring.factories
List<DomainService> domainServices = SpringFactoriesLoader.loadFactories(DomainService.class, null);
```

Plag-in architecture

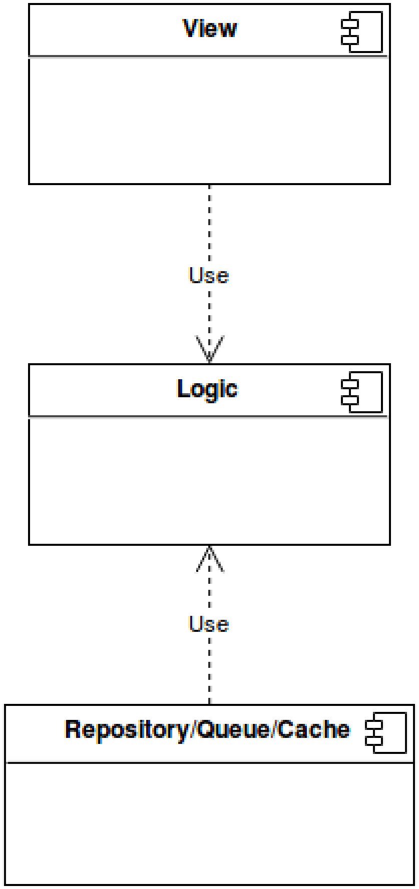

Logic
logic::ForumService
+ listAllForumEntities(): ForumEntity
0
repository
<<I>> logic::ForumRepository
+ findAllForums(): ForumEntity
MongoRepository
mongoRepository::MongoForumRepository
+ findAllForums(): ForumEntity
PostgresRepository
postgresRepository::PostgresForumRepository
+ findAllForums(): ForumEntity

Three layers architecture. Implementation (III)



- Business Logic does not depend on the implementation detail which is repository
    - Due to dependency inversion
- Repository can be replaced without changes inside Logic component
- Many implementations of Repository module can co-exist.

Three layers architecture. Implementation (III)

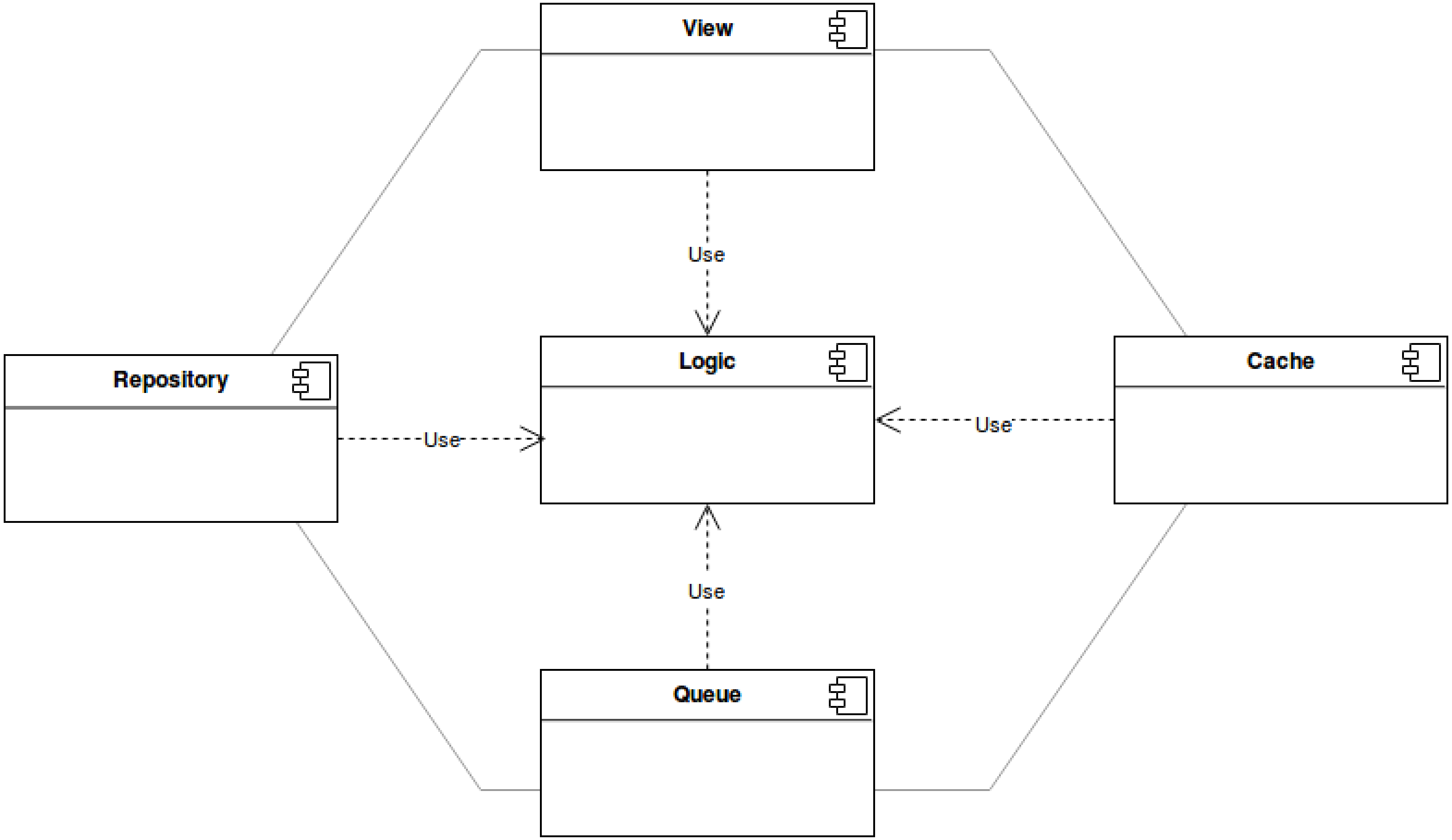


- Is designed to protect business logic
- All implementation details are moved to component which is usually called „Repository”
  - Repository
    - Postgresdb
    - Mongo
  - Queue
    - RabbitMQ
    - JMS
  - Cache
    - Guava
    - Redis

Łukasz Soszyński

Hexagonal Architecture

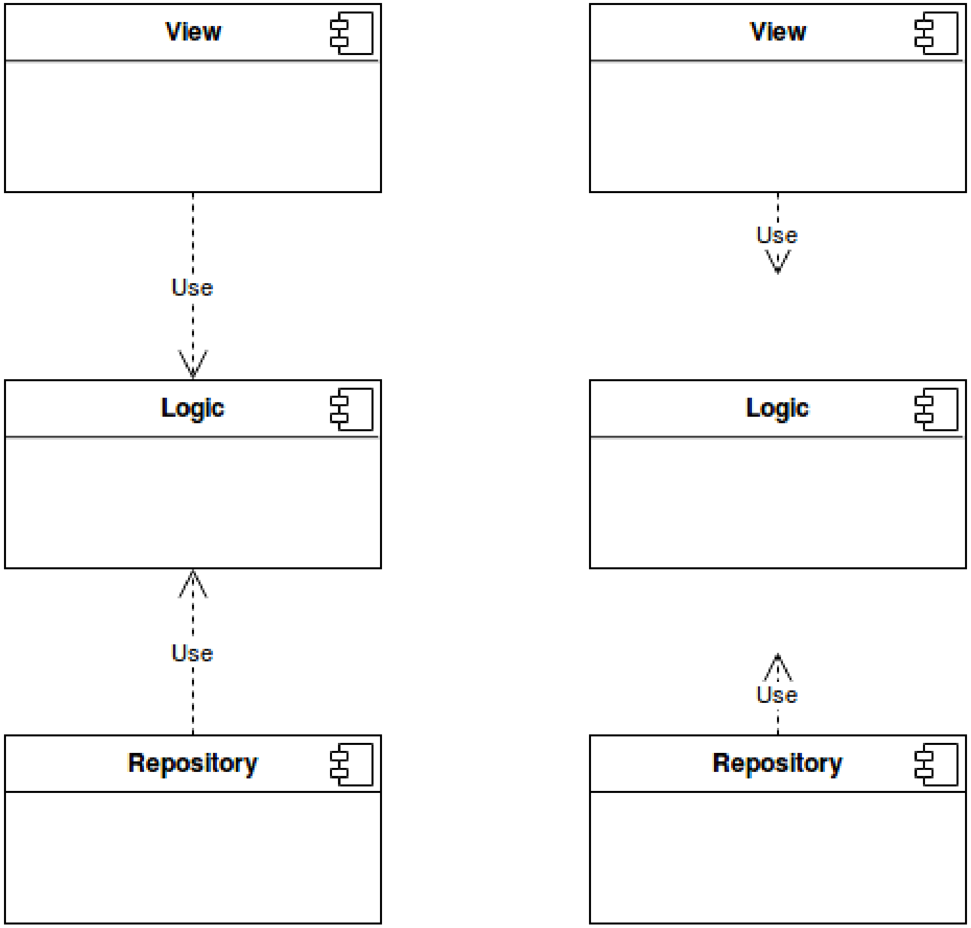
Almost Hexagonal Architecture



Hexagonal Architecture Names

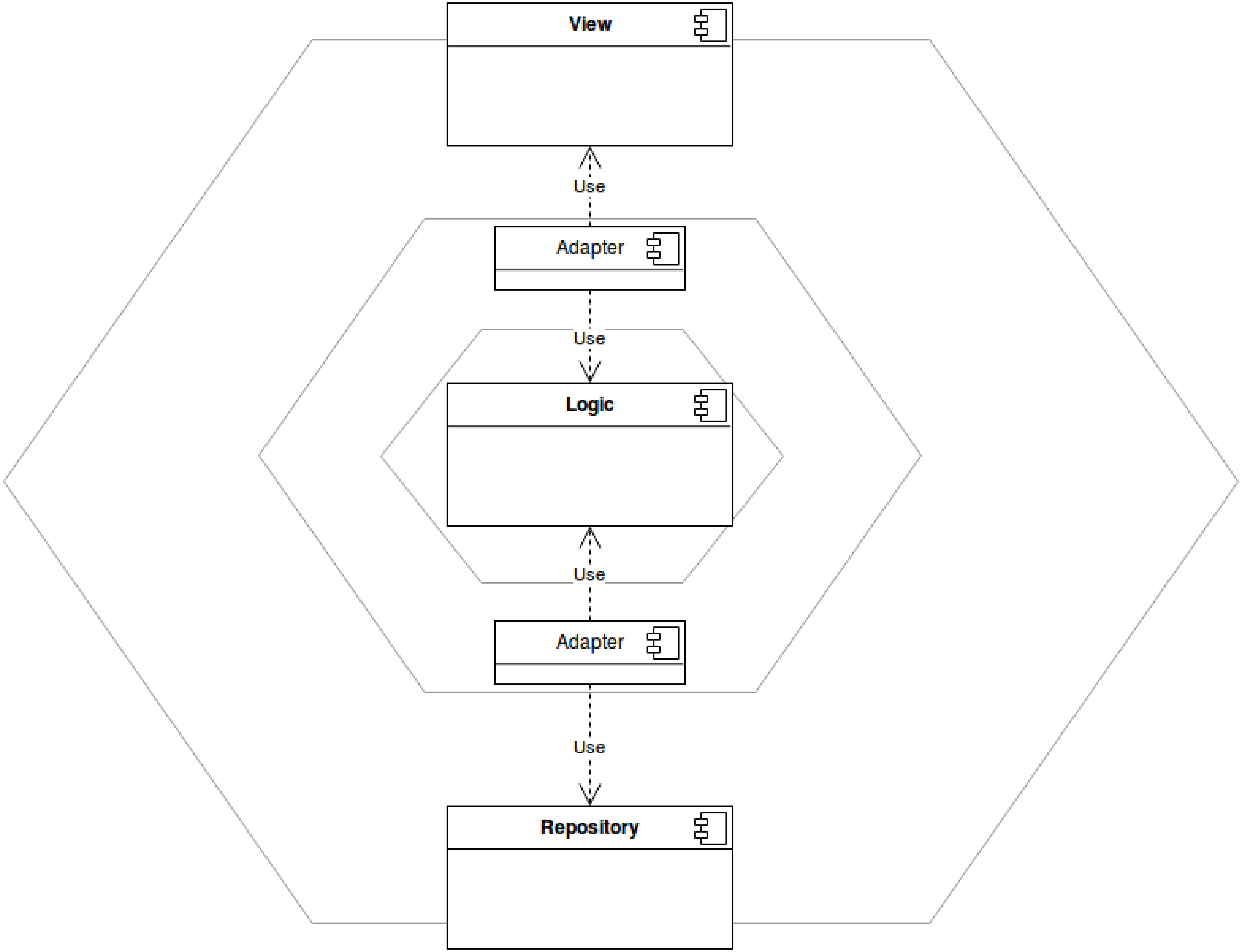
- Hexagonal Architecture
- Ports & Adapters
- Onion Architecture
- Clean Architecture

Ports & Adapters (I)

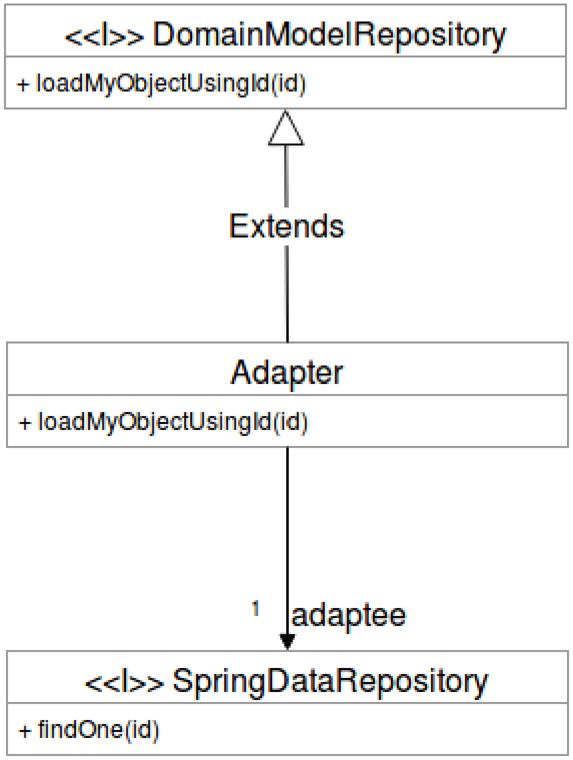


How to loosen dependency between components?

Ports & Adapters (II)

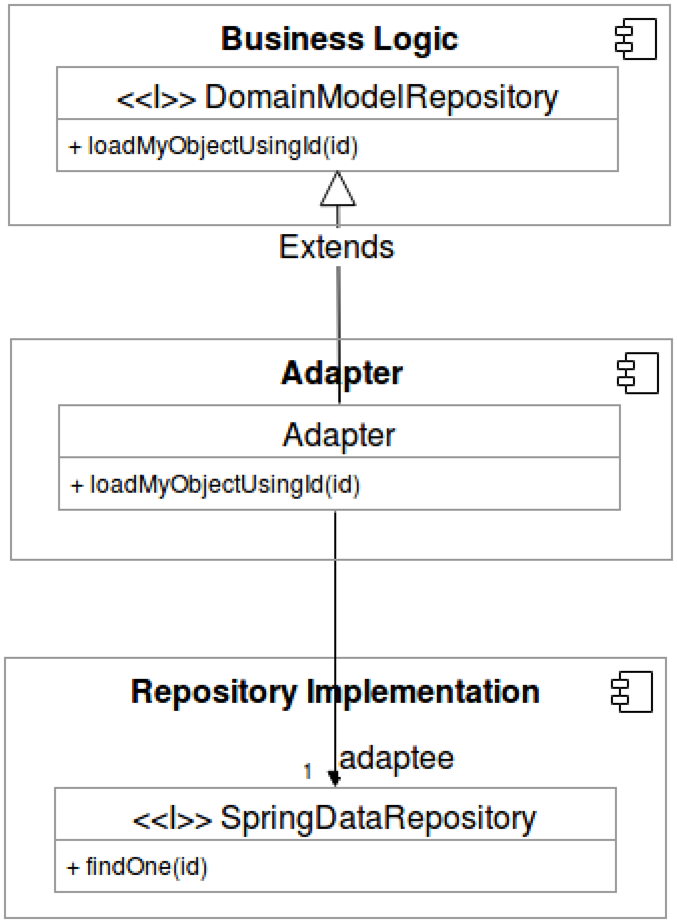


Ports & Adapters (III)

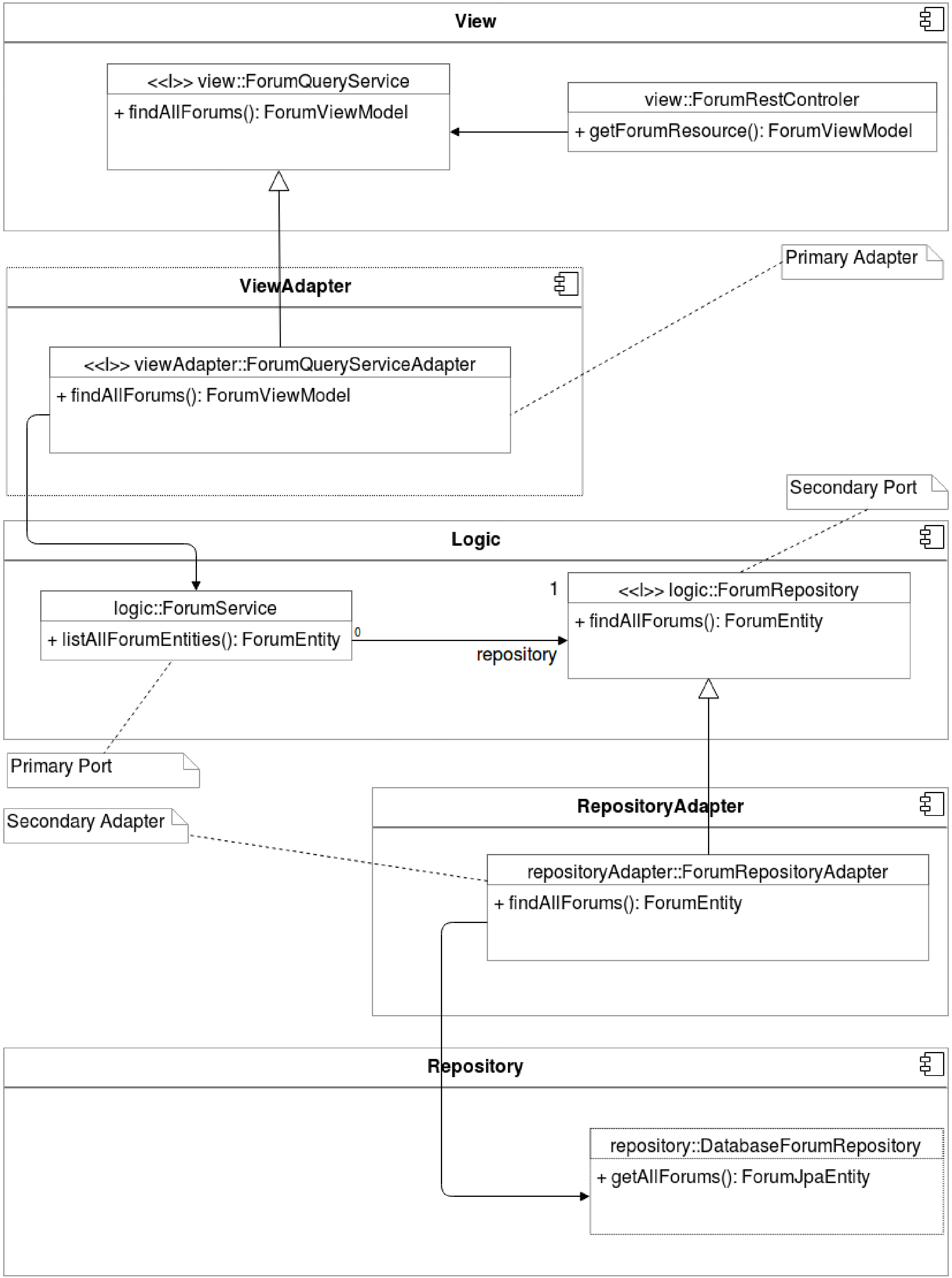

<<I>> DomainModelRepository
+ loadMyObjectUsingId(id)
Extends
Adapter
+ loadMyObjectUsingId(id)
1
adaptee
<<I>> SpringDataRepository
+ findOne(id)

Adapter design pattern

Ports & Adapters (III)


Business Logic
<<I>> DomainModelRepository
+ loadMyObjectUsingId(id)
Extends
Adapter
Adapter
+ loadMyObjectUsingId(id)
Repository Implementation
1
adaptee
<<I>> SpringDataRepository
+ findOne(id)

Ports & Adapters (IV)


View
<<I>> view::ForumQueryService
+ findAllForums(): ForumViewModel
view::ForumRestControler
+ getForumResource(): ForumViewModel
Primary Adapter
ViewAdapter
<<I>> viewAdapter::ForumQueryServiceAdapter
+ findAllForums(): ForumViewModel
Secondary Port
Logic
logic::ForumService
+ listAllForumEntities(): ForumEntity
0
1
repository
<<I>> logic::ForumRepository
+ findAllForums(): ForumEntity
Primary Port
Secondary Adapter
RepositoryAdapter
repositoryAdapter::ForumRepositoryAdapter
+ findAllForums(): ForumEntity
Repository
repository::DatabaseForumRepository
+ getAllForums(): ForumJpaEntity

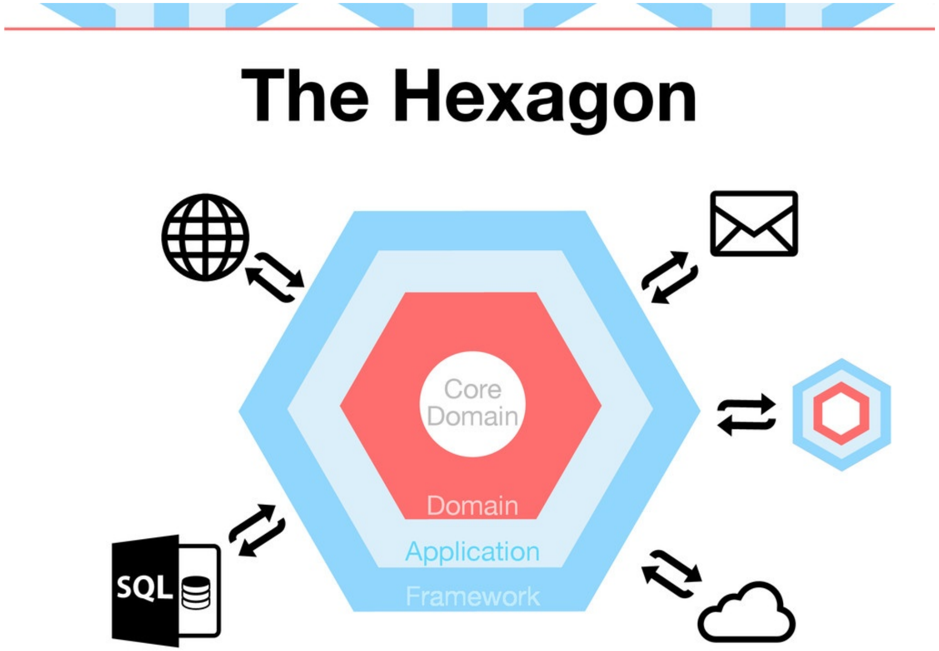
Ports & Adapters (V)

- Primary Port
  - Concret Class
    - More-less
  - Located inside model
  - *ForumService*
- Primary Adapter
  - Invokes Primary Port
  - eg. „controller-like function”
  - *ForumQueryServiceAdapter*

- Secondary ports
  - Interface
  - Located inside model
  - Invokes Secondary Adapter
  - *ForumRepository*
- Secondary Adapter
  - Implementation of interface which is a Secondate port
  - ForumRepositoryAdapter

http://www.dossier-andreas.net/software_architecture/ports_and_adapters.html

You Can Add More Layers

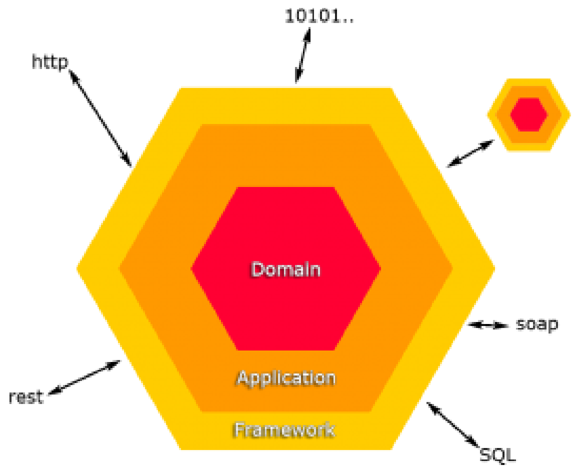


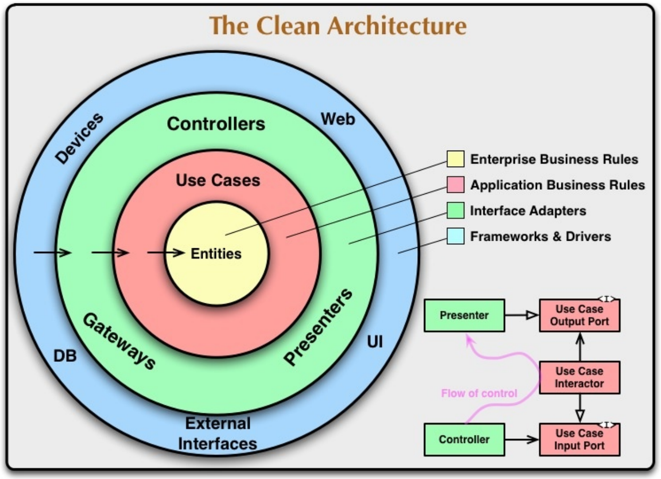
Application Layer

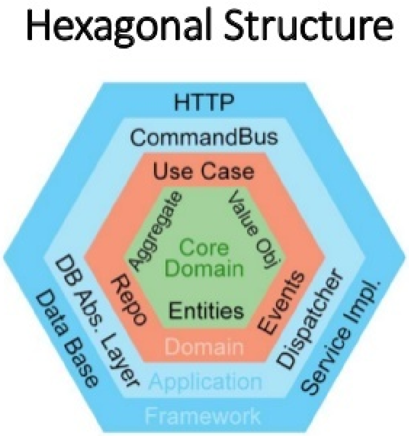
- Security
- Transactions

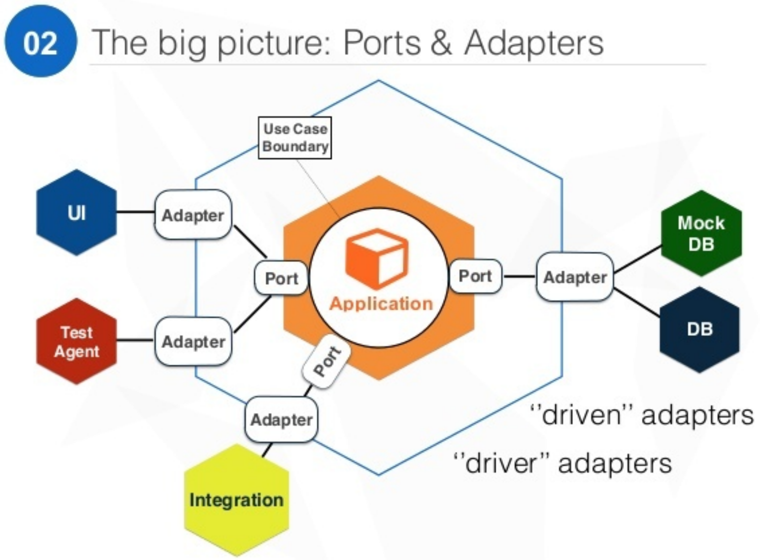
Source: http://fideloper.com/hexagonal-architecture

You can adapt it









When To Use Hexagonal Architecture

- Long-living applications
- Application with sophisticated business logic
- Medium and large applications
- Command part of CQRS architecture

Disadvantages

- Conversion
  - During invocation Port ↔ adapter
- A lot of boilerplate code
  - You can try grouping all adapters in one component
- Verbose

Example

Example

- Repository:
  git@github.com:lukaszsoszynski/hexagonal-architecture.git
- Run With:
  java --add-modules java.xml.bind,java.xml.ws -jar install-0.0.1-SNAPSHOT.jar
- Tags
  - WHOLE_LOGIC_IN_SERVICE_MODULE
  - REPOSITORY_AMQP_MODULE
  - REPOSITORY_AMQP_MODULE_DI
  - HEXAGONAL_ARCHITECTURE
  - HEXAGONAL_ARCHITECTURE_SOAP
- Domain
  - Feature rich discussion forum

Technical details

- Java 9
- Speing Boot 2.0.0.RC1
- JPA
- AMQP
- REST
- WS (xml based web services)

WHOLE_LOGIC_IN_SERVICE_MODULE

- Model Component
  - Does not depend on anything
  - Contains domain model (business logic)
  - Does not contain **whole** business logic
  - orm.xml
- Module Service
  - Contains part of business logic
    - Events are emitted after post creation
    - Persistence
  - Depends on implementation details
    - Spring Data
    - Spring AMQP

REPOSITORY_AMQP_MODULE

- Component AMQP created
- Component Repository created
- Component Service
  - Still contains part of business logic
  - Depends on Amqp Component
    - Transitive dependency on Spring AMQP
  - Depends on Repository Component
    - Transitive dependency on Spring Data

REPOSITORY_AMQP_MODULE_DI

- Service component
  - Does not depend on Spring AMPQ
  - Does not depend on Spring Data
  - Contains part of business logic
- Dependency Inversion principal used