Java Programming 3

Week 2

KdG

Agenda this week

Last week - Project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
Spring Framework: The Spring Container
Component Scanning - Autowiring

KdG

Agenda this week

Last week - Project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
Spring Framework: The Spring Container
Component Scanning - Autowiring

KdG

Last week - Project Review

- Topics last week: recap
- What is the state of the projects?
  - Problems?
  - What went ok?
  - What went wrong?
- Did you think about *the technical design*?
  - What package structure did you use?
  - What classes did you add?

KdG

Project Review

- JSON writing: custom serializer for LocalDate or LocalDateTime

```
public class LocalDateTimeSerializer implements JsonSerializer<LocalDateTime> {
   private static final DateTimeFormatter FORMATTER =
DateTimeFormatter.ofPattern("d-MMM-yyyy hh:mm");
   @Override
   public JsonElement serialize(LocalDateTime localDateTime, Type typeOfSrc,
JsonSerializationContext context) {
       return new JsonPrimitive(FORMATTER.format(localDateTime));
   }
}
```

- Register the Serializer with your builder

```
builder.registerTypeAdapter(LocalDateTime.class,
  new LocalDateTimeSerializer());
```

KdG

Project Review

- JSON writing: make List attributes (many-to-man relationships) transient
  - Otherwise: StackOverflow!

KdG

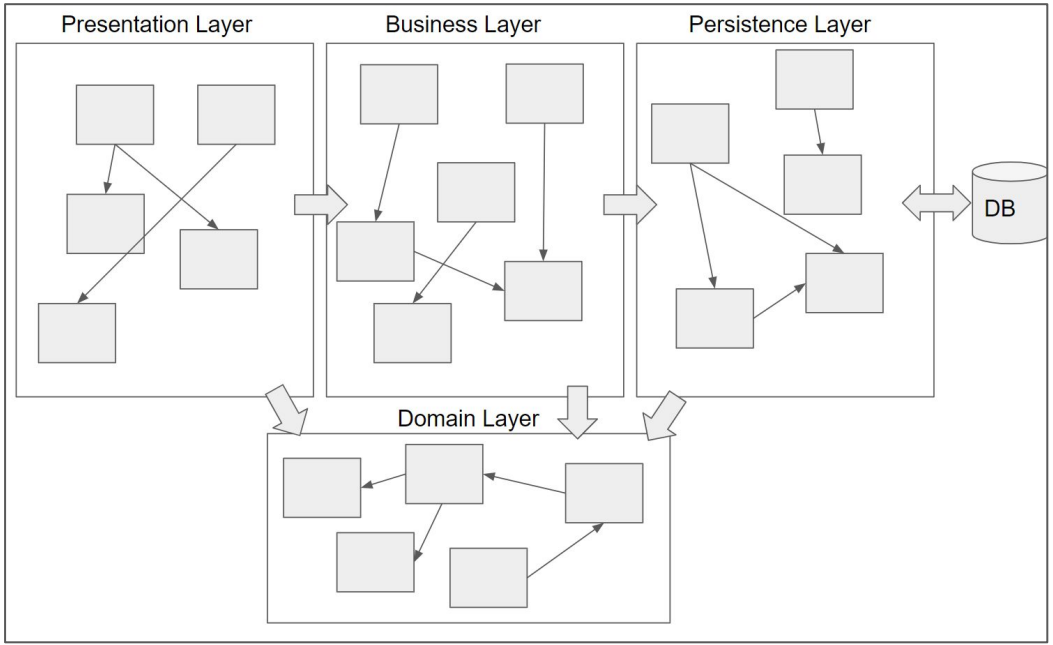
Agenda this week

Last week - project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
Spring Framework: The Spring Container
Component Scanning - Autowiring

KdG

3 - Layered Architecture

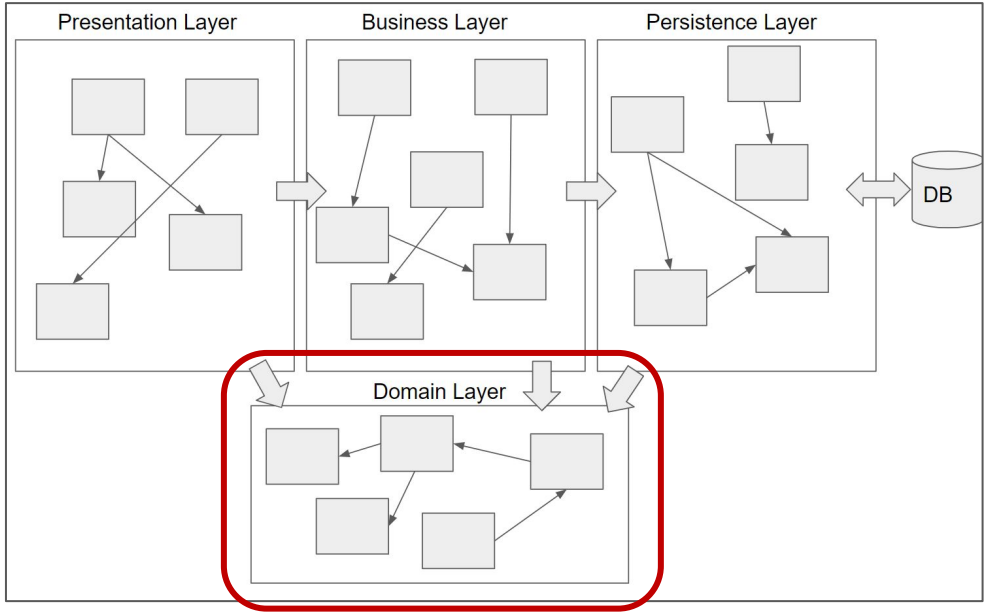
- Split the application into 3 layers + the domain layer
- Roadmap for your code
- Provides
  - Overview
  - Reusability
  - Maintainability
  - ...



KdG

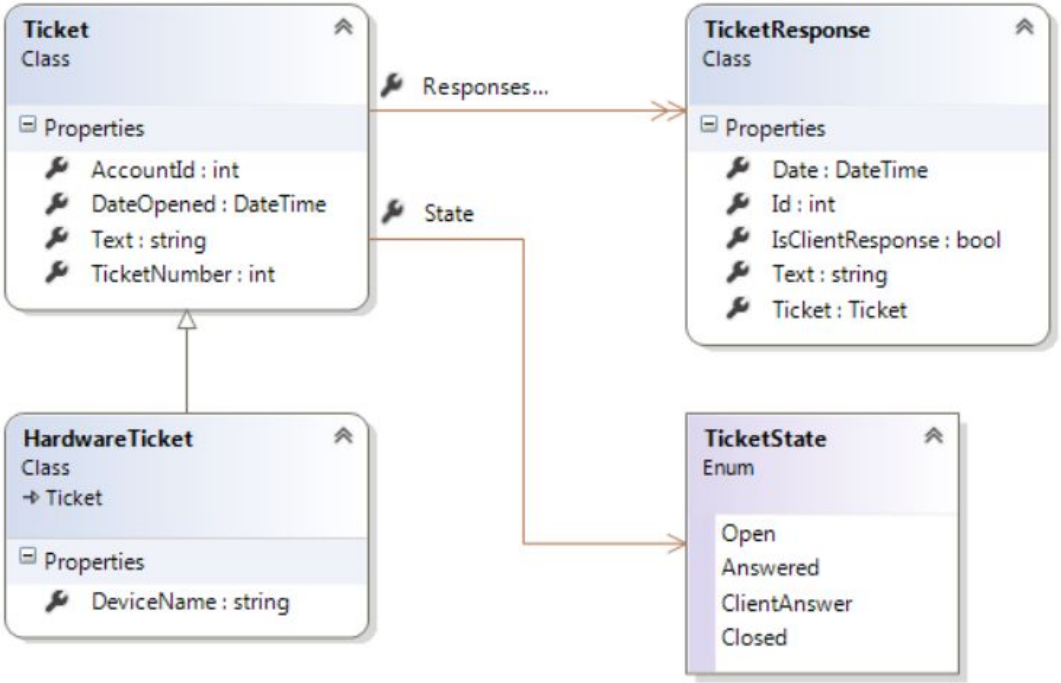
Domain Layer: based on the *Domain Model*

- The application is build around a ‘Domain Model’
- They are the foundation of the application
- They are also called ‘conceptual model’



KdG

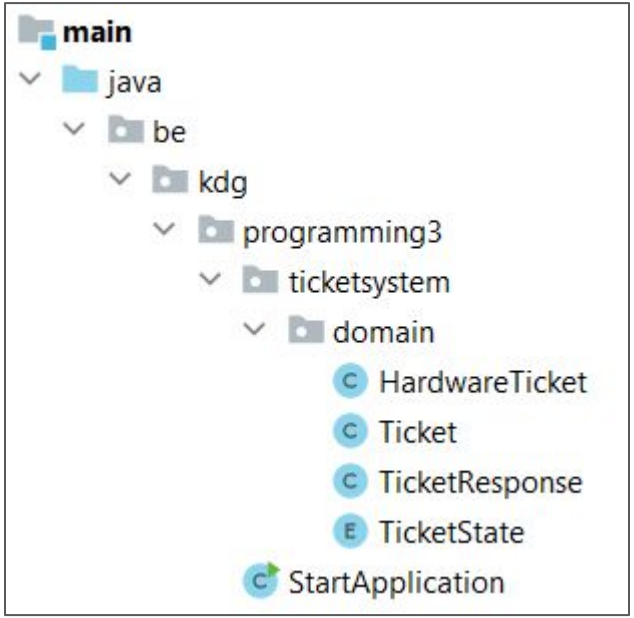
Domain Model: example (ticket system)



KdG

Domain Model → Domain Layer

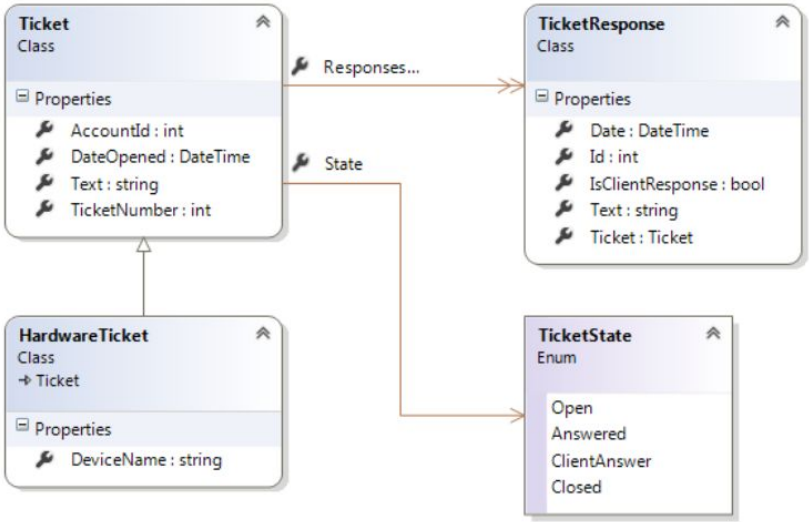
- Domain Model serves as a starting point for creating the classes in the *Domain Layer*
- Names of classes match the names of the entities of the Domain Model
- Relationships can translate into attributes (or extra classes)
- We create a java package **`domain`** to store the Domain Layer classes



KdG

Exercise 1: Domain Layer

- Support Center: we will implement a support center
- The Domain Model is given...
- Implement the Domain Layer!
- Create some valid data (in main method):
  - Make 2 tickets
  - Add responses to the tickets
  - Show all tickets (using toString)
  - Show all responses of a ticket



KdG

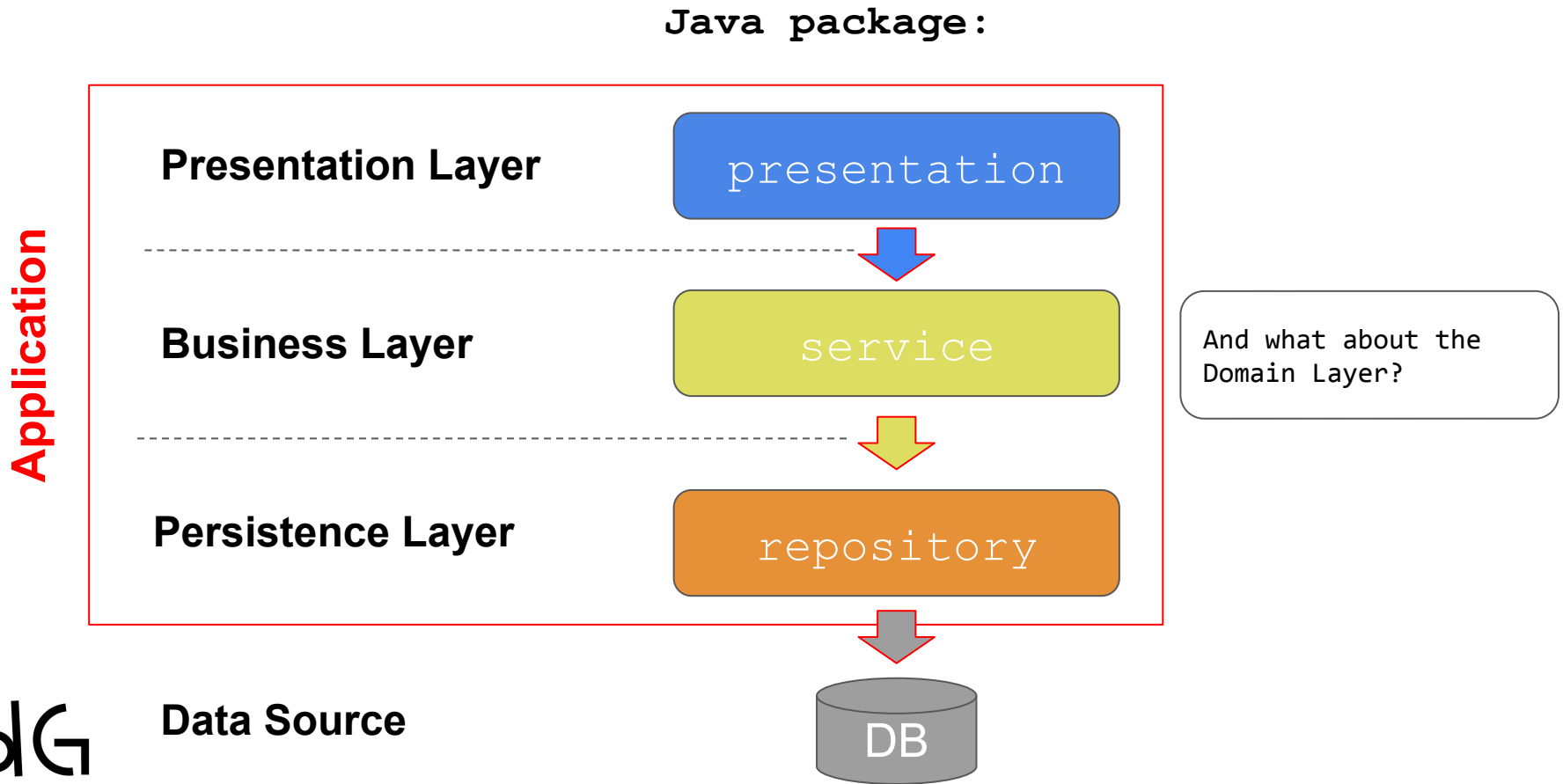
N-Layer architecture: dependencies…

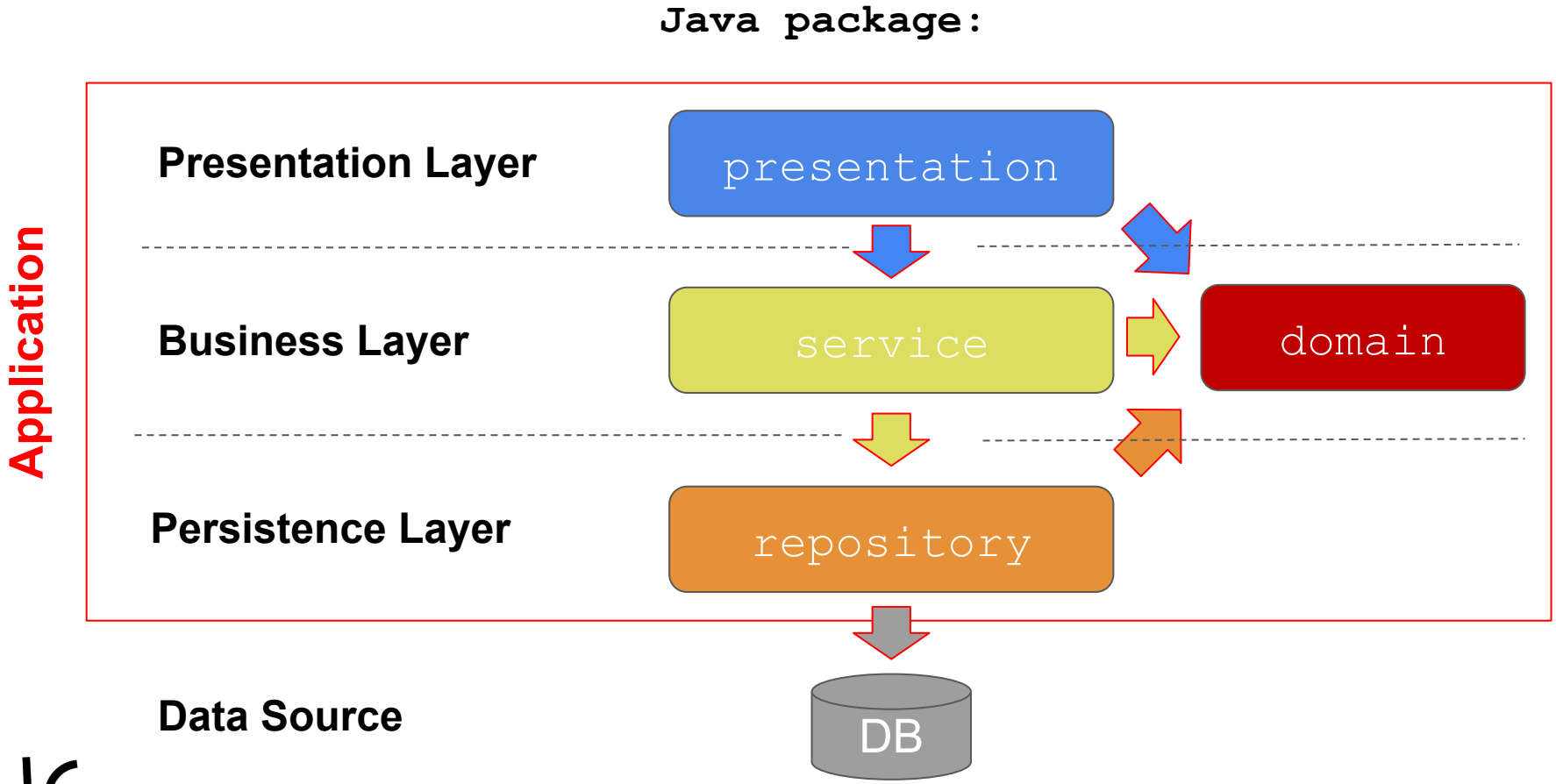
- Split up the application in logical components/layers
- Classical 3-layer model (+ domain layer):
  - **Presentation layer**
    - Visual representation
    - Interaction with the user
  - **Business Layer**
    - Domain models
    - Services
    - Business logic
  - **Persistence Layer**
    - Data access and data storage

A layer can only have a dependency on a lower layer!

KdG

N-Layer architecture: dependencies…




Java package:
Application
Presentation Layer
presentation
Business Layer
service
domain
Persistence Layer
repository
Data Source
DB

Demo 1: Layered Architecture

- In this demo we will
  - Create a small 3-layered console application (Student Management System)
    i. Create the domain package: a Student has an id, name, birthdate and length
    ii. Create the repository package with one repository class: StudentRepository. You can create and read students. We use a static List<Student> (represents the “Database”)
    iii. Create the service package with one service class: StudentService. You can add students and get all students.
    iv. Create the view package with one view class: View. It has a show method to show a menu to list all students and add a new student

KdG

Exercise 2: Layered Application (1/2)

- Upgrade the previous exercise (Support Center) to a small 3-layered application.
  - Create the repository package, it has one class TicketRepository:
    - Use a static List of Tickets and a static List of TicketResponses.
    - Create a seed method to seed the lists with data. Run from a static block.
    - Provide 2 (non-static!) methods:
      - Ticket createTicket(Ticket ticket): incoming ticket gets a unique ticketnumber (highest + 1)
      - List<Ticket> readTickets(): returns current list of Tickets

See next slide for the rest of the exercise...

KdG

Exercise 2: Layered Application (2/2)

- Create the service package, it has one class TicketService. This class has a TicketRepository attribute, created in the constructor. It has the following methods:
  - List<Ticket> getTickets(): returns all tickets. Use the repository...
  - Ticket addTicket(int accountId, String question): to create a Ticket. Use the repository...
  - Ticket addTicket(int accountId, String device, String problem): to create a HardwareTicket. Use the repository…
- Create a presentation package, it has one class View. This class has two attributes: a TicketService and a Scanner. It has one public method:
  - show(): shows a small menu (0:exit, 1:show all tickets, 2: add ticket). This method loops until the user chooses exit.
  - Add functionality to show the tickets and be able to add a ticket
- Create a View from the main method and run the show() method

KdG

Agenda this week

Last week - project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
Spring Framework: The Spring Container
Component Scanning - Autowiring

KdG

Loose coupling with Interfaces

- Try to make the dependencies *loosely coupled*:
  - It should be possible to switch to another implementation without affecting the dependent classes/layers
  - It should be possible to test classes/layers without the need for a full implementation of the classes/layers it depends on
- We can do this using interfaces
  - A class only talks to via an interface to it’s dependencies
  - The dependencies implement the interfaces

KdG

Demo 2: Loose Coupling with Interfaces

- In this demo we will
  - Demonstrate loose coupling with interfaces on the small Student Management System from previous demo
    - We extract an interface from the StudentRepository
    - The current class becomes an implementation of this interface (HardCodedStudentRepository)
    - We do the same for StudentService
    - We can provide a DummyStudentService implementation that only logs some output. That way we can test the View without using the real StudentService implementation

KdG

Exercise 3: Loose Coupling

- Refactor the Ticket System:
  - convert the repository and service classes into interfaces and use them
  - Provide some dummy implementations

KdG

Agenda this week

Last week - project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
Spring Framework: The Spring Container
Component Scanning - Autowiring

KdG

Dependency Injection

```
new Client()
```

- Example: class Client depends on class MyRobot.
- It could create the instance itself:
- Problem:
  - To replace the implementation of MyRobot, you have to alter the class Client!

```
public class Client {
   private final MyRobot myRobot;

   Client() {
       myRobot = new MyRobot();
   }

   public String greet() {
       return "Hello " + MyRobot.getName();
   }
}
```

```
public class MyRobot {
   @Override
   public String getName() {
       return "Johnny";
   }
}
```

Dependency Injection

- Solution:
  - Use an interface in Client
  - Inject the dependency (eg. via constructor parameter)

```
new Client(new MyRobot())
```

```
public class Client {
   private final Robot myRobot;

   Client(Robot robot) {
       myRobot = robot;
   }

   public String greet() {
       return "Hello " + MyRobot.getName();
   }
}
```

```
public interface Robot{
   String getName();
}
```

```
public class MyRobot implements Robot{
   @Override
   public String getName() {
       return "Johnny";
   }
}
```

Demo 3: Dependency Injection

- In this demo we will
  - We will refactor our Student Management System:
    - We will use Dependency Injection now
    - We have to add some ‘glue’ code to construct the different object
    - We can now swap the implementation of the service class without changing the view class.

KdG

Exercise 4

- Refactor the Ticket System:
  - Use Dependency Injection
  - Add the ‘glue’ code in your StartApplication
  - Does it still run?
  - Swap the implementations with dummy implementations and run again

KdG

Remarks

- Working with N-layer architecture and using interfaces and dependency injection is a guide, “design pattern”, the goal is:
  - **Separation of concerns**
    - Each layer has own layer-specific logic, described in interfaces
  - **Single Responsibility Principle**
    - Each class has small set of responsibilities
      - Business Layer and Persistence Layer can have more than one service- and repositoryclasses.
      - Use extra (helper-)classes for extra functionality

KdG

Sidestep: where is MVP?

- Remember: Model - View - Presenter
  - Model: contains the data
  - View: only view code
  - Presenter: contains the view logic
- Model: we can use our Domain classes and Service layer
- View: we remove all logic, only the System.out.println stuff
- Presenter: contains the UI logic

KdG

Demo 4: Introduce MVP

- In this demo we will
  - We will refactor our Student Management System:
    - We will create a Presenter class in the presentation layer
    - Move all UI logic to this class
    - There is a dependency from the Presenter to the View
    - There is also a dependency from the View to the Presenter
      - It would be better if the View was agnostic about the Presenter
      - We could introduce the *Observer* design pattern here: https://www.baeldung.com/java-observer-pattern

@Extra

KdG

Exercise 5

- Refactor the Ticket System:
  - Introduce MVP in your exercise
  - (Extra: try using the *observer* pattern to avoid a direct reference from the view to the presenter (not so easy…!))

KdG

Agenda this week

Last week - project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
Spring Framework: The Spring Container
Component Scanning - Autowiring

KdG

What is Spring?

- Spring is an open source framework to build *enterprise applications* in Java
- Main goal: Simplify Java Development
- Spring consist out of many *projects*: https://spring.io/projects
  - **Spring Framework**
  - Spring Boot
  - Spring Data
  - Spring Cloud
  - Spring Security
  - Spring GraphQL
  - Spring Session
  - Spring Integration
  - …

→ Each of the projects has its own version

KdG

What is the Spring Framework?

- **Spring Container with Dependency Injection**
- Web and remoting (Spring MVC, Webflux, …)
- Data Access and Integration
  - Less boilerplate JDBC with templates
  - Object Relational Mapping
- Aspect Oriented Programming:
  - implement application wide concerns
- Messaging
- Instrumentation: measure performance, diagnose errors,..
- Testing

KdG

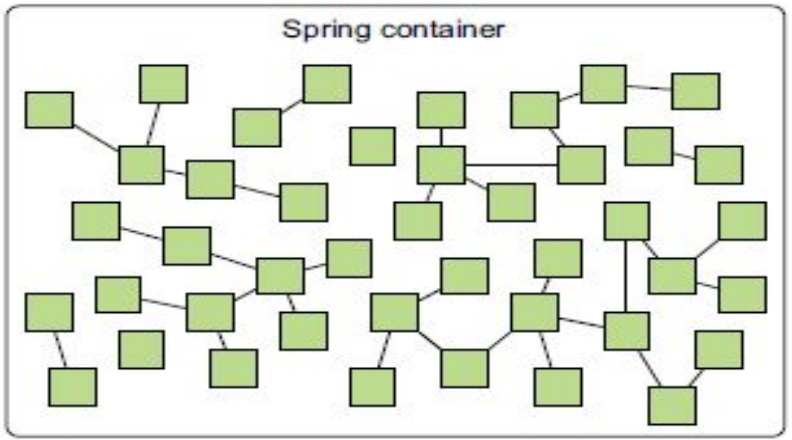
Spring Container with Dependency Injection

Enterprise application: is composed of many components that need to work together. Each component is responsible for a part of the functionality.

Components need to be created and work together

Spring Container (“Spring Application Context”) creates and manages these components (or *beans*)

This is done by using *Dependency Injection*



KdG

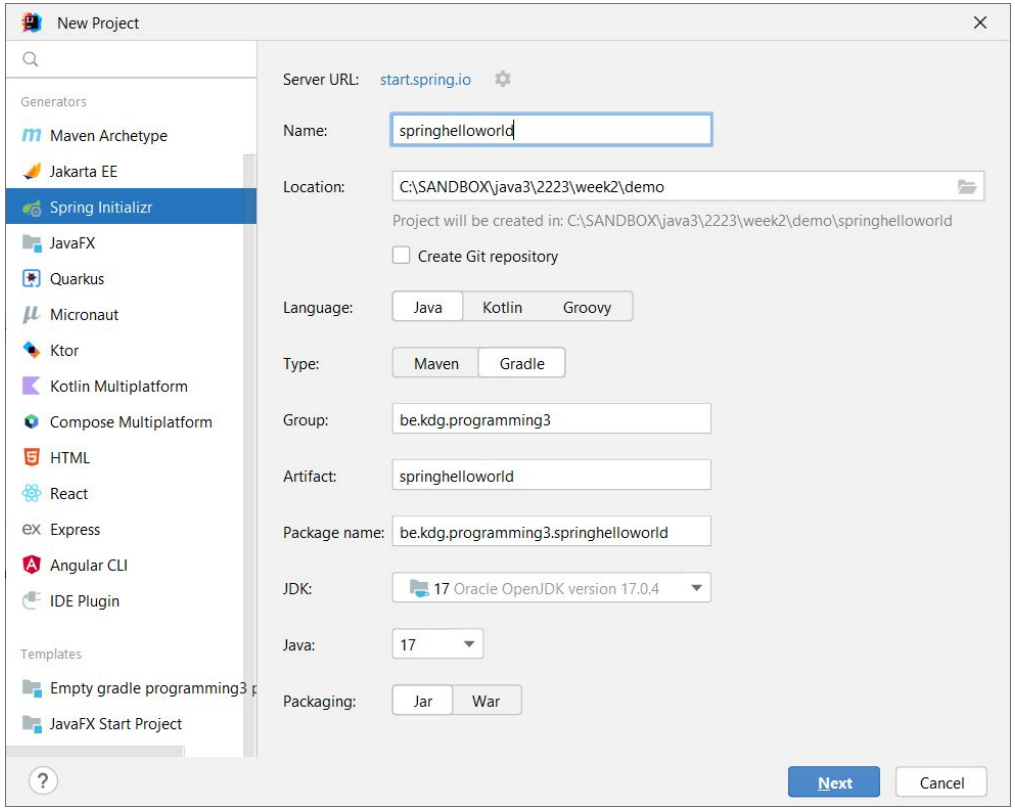
Dependency Injection Frameworks

- Also called Dependency Injection Containers
- Are in fact an implementation of the Abstract Factory design pattern
- Spring is not the only DI Framework
  - Enterprise Java Beans
  - Dagger (Android)
  - Micronaut
  - ...

KdG

Intellij Spring project

- File → new → Spring Initializr
  - Gradle project
- No extra Dependencies (for now)



build.gradle.kts

```
plugins {
 java
 id("org.springframework.boot") version "3.1.3"
 id("io.spring.dependency-management") version "1.1.3"
}
//…
dependencies {
 implementation("org.springframework.boot:spring-boot-starter")
 testImplementation
    ("org.springframework.boot:spring-boot-starter-test")
}

//…
```

KdG

Example: Knights and Quests

- We created a small demo project that uses DI:

The quest is injected in the HandyKnight

```
public interface Knight {
    void embarkOnQuest();
}
```

```
public interface Quest {
    void embark();
}
```

```
public class HandyKnight implements Knight{
    private Quest quest;

    public HandyKnight(Quest quest) {
        this.quest = quest;
    }

    @Override
    public void embarkOnQuest() {
        quest.embark();
    }
}
```

```
public class ErrandQuest
implements Quest{
    @Override
    public void embark() {
        //..do some errands
    }
}
```

The ‘glue’ code: making a quest and a knight

```
Quest quest = new ErrandQuest();
Knight knight = new HandyKnight(quest);
knight.embarkOnQuest();
```

Can Spring create the objects for us?

The quest is injected in the HandyKnight

KdG

Configure the container in @Configuration class

- @Configuration marks the configuration class
  - Configuration can also be done using XML files…
- Best delivered as source code → should be easy to alter
- @Bean annotation: marks the factory method for a bean.
  - You don't call this method, it is configuration!
- Spring sees a Quest parameter is needed → Spring will look for the first bean that implements the interface and inject it

```
@Configuration
public class KnightConfiguration {
    @Bean
    public Knight knight(Quest quest){
        return new HandyKnight(quest);
    }

    @Bean
    public Quest quest(){
        return new ErrandQuest();
    }
}
```

Using the @Beans

- @SpringBootApplication: class to start the application
- ConfigurableApplicationContext is the container that contains the beans and is managed by the Spring Framework
- Beans are by default Singleton: only one instance of the bean exists

```
@SpringBootApplication
public class SpringinjectionApplication {
   public static void main(String[] args) {
       ConfigurableApplicationContext context =
       SpringApplication.run(KnightConfiguration.class,
args);
       context.getBean(Knight.class).embarkOnQuest();
       context.close();
   }
}
```

KdG

What classes are @Bean candidates?

- Classes based on interfaces with possibly multiple implementations
  - Services, repositories, …
- You can use this configuration file to choose the correct implementation

KdG

Exercise 6

- Create the small Knights application that we showed in the slides and experiment!

KdG

Demo 6: Use Spring on Student System

- In this demo we will
  - Look at our previous demo application: the Student Management System
  - We will create a Spring application that uses the application context to create the Beans and run the application!

KdG

Exercise 7

- Now try to create a Spring application for the Support System!

KdG

Agenda this week

Project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
The Spring Container
Component Scanning - Autoconfiguration

KdG

Component scanning - Autowiring

- It is also possible to let Spring create and inject beans without a @configuration class!
- Instead of using @Bean in a @Configuration class, you can use @Component in the implementation class

```
@Component
public class ErrandQuest
implements Quest {
    @Override
    public void embark() {
        //do something
    }
}
```

KdG

Component scanning - Autowiring

- @Autowired tells Spring to find an implementation of Quest and inject it!
- You can also use @Inject

```
@Component
public class HandyKnight implements Knight {
   private Quest quest;

   @Autowired
   public HandyKnight(Quest quest) {
       this.quest = quest;
   }

   @Override
   public void embarkOnQuest() {
       quest.embark();
   }
}
```

KdG

Component scanning - Autowiring

- We no longer need the @Configuration class
- Spring will scan for @Component classes in packages and subpackages starting from the location of this class.
- You can combine component scanning and a @Configuration class

```
@SpringBootApplication
public class SpringinjectionApplication {
   public static void main(String[] args) {
        ConfigurableApplicationContext context =
        SpringApplication.run(SpringinjectionApplication.class, args);
        context.getBean(Knight.class).embarkOnQuest();
        context.close();
   }
}
```

KdG

What if there is more than one implementation?

- If there is more than one implementation of the interface, which one do we use?
  - Use the @Qualifier annotation!
    - Add it to the @Component annotation to give the component a name

```
@Qualifier("errand")
@Component
public class ErrandQuest implements Quest
{
    @Override
    public void embark() {
        //do something...
    }
}
```

```
@Component
public class HandyKnight implements Knight {
    private Quest quest;

    public HandyKnight(@Qualifier("errand") Quest quest)
        this.quest = quest;
    }
```

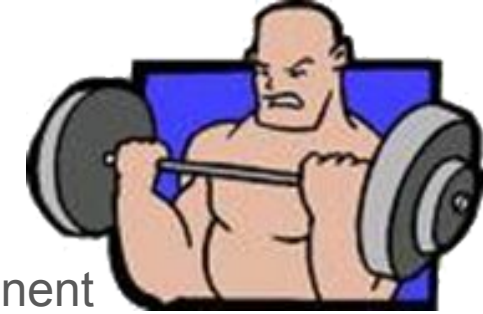
KdG

Exercise 8

- Refactor your previous exercise: use component scanning and autowiring!

KdG

Exercise 9 (1/3)



- If you have the Support System up and running with component scanning, try to add complete CRUD (Create, Read, Update, Delete) functionality to it:
  - In the repository:
    - Ticket readTicket(int ticketNumber)
    - void updateTicket(Ticket ticket)
    - void deleteTicket(int ticketNumber)
    - List<TicketResponse> readTicketResponsesOfTicket(int ticketNumber)
    - TicketResponse createTicketResponse(TicketResponse response)

  → Tip: deleteTicket also removes the responses of the concerning ticket, createTicketResponse creates an id.

KdG

Exercise 9 (2/3)

- In the business layer: TicketService
  - Ticket getTicket(int ticketNumber)
  - void changeTicket(Ticket ticket)
  - void removeTicket(int ticketNumber)
  - List<TicketResponse> getTicketResponses(int ticketNumber)
  - TicketResponse addTicketResponse(int ticketNumber, String response, boolean isClientResponse)

→ Tip: addTicketResponse adds a response to the Ticket but also updates the status of the Ticket using isClientResponse

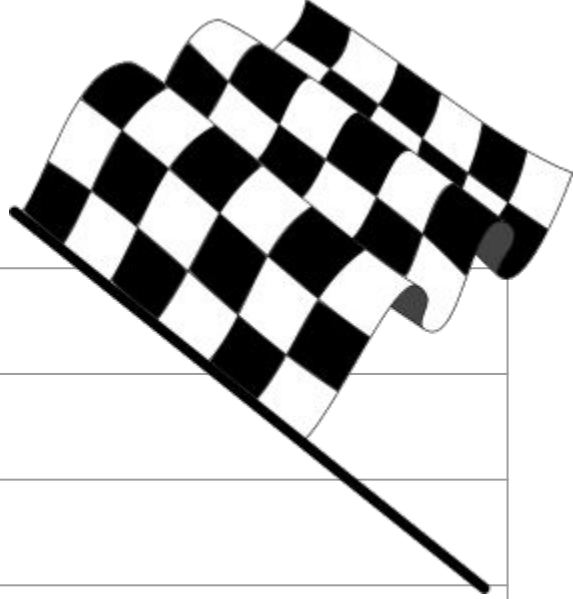
KdG

Exercise 9 (3/3)

- In the presentation layer
  - Expand the menu:
    1. Show all tickets
    2. Show details of ticket
    3. Show answers of ticket
    4. Make a new ticket
    5. Answer a ticket
    6. Remove a ticket
    7. Exit
  - Implement all functionality!

KdG

Agenda this week



Project Review
3-Layer Architecture
Loose Coupling With Interfaces
Dependency Injection
The Spring Container
Component Scanning - Autoconfiguration

KdG

Project

You refactor the application, it should use a **layered architecture:**

- presentation layer: contains the view code. Try to implement MVP here: the view code is separated from the view logic.
- business layer (services + domain): use two packages, implement a Service class for each of the two main entities.
- data access layer: contains the data classes (repositories).

Add **interfaces** to provide loose coupling between the layers. Use dependency injection to connect the different classes.

And finally convert it to a **Spring application** using the Spring Initializr. Use **component scanning** and **autowiring** to connect the components and configure the application context.

Push your results to a private repository on gitlab, tag it as 'sprint1' and submit the URL of your repository as a response to the canvas assignment. Be sure to add your teacher as Reporter to your project!

KdG