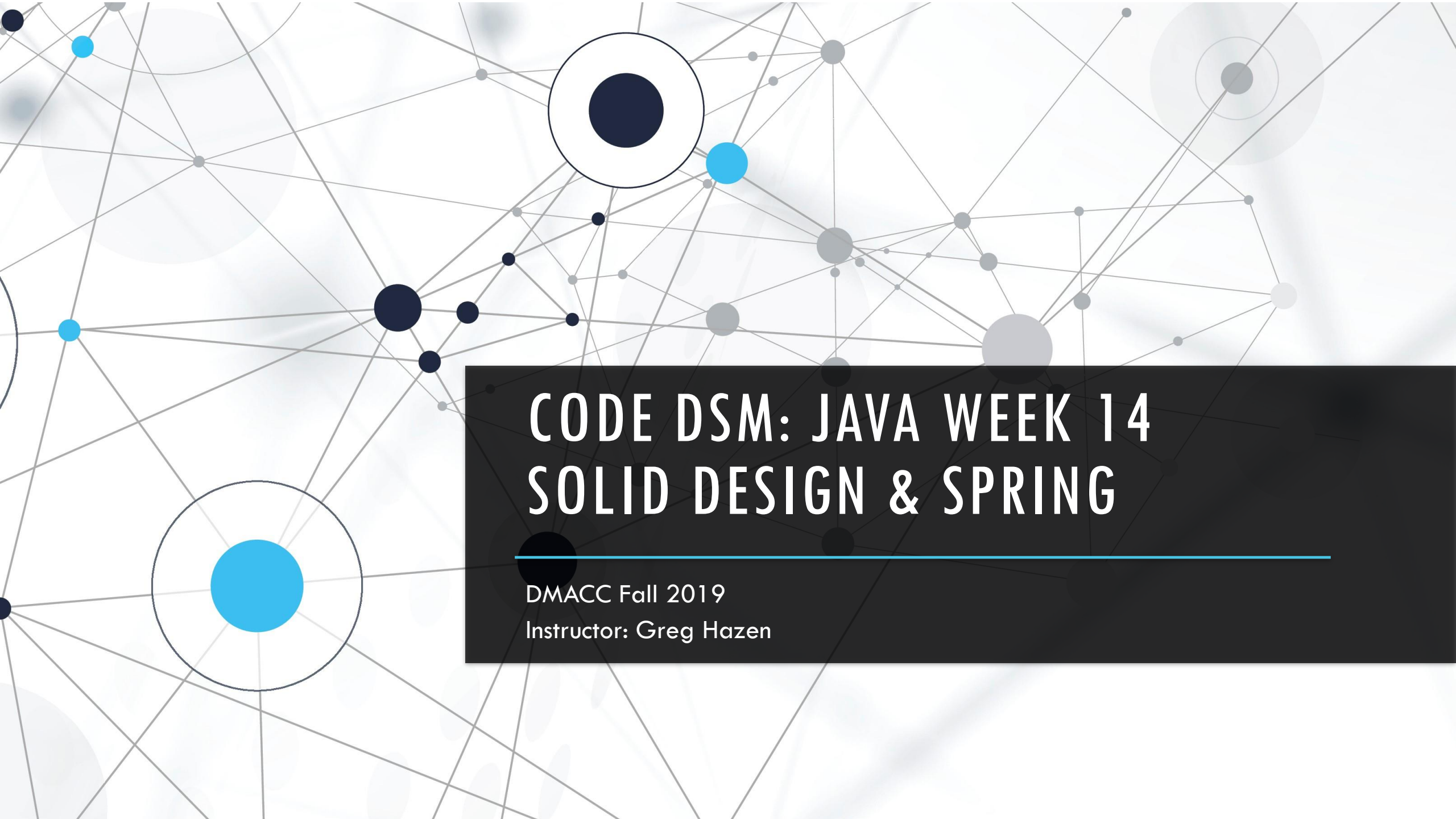


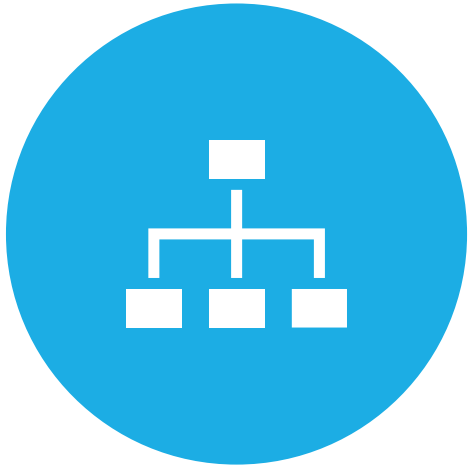
CODE DSM: JAVA WEEK 14 SOLID DESIGN & SPRING

DMACC Fall 2019

Instructor: Greg Hazen

AGENDA

SOLID DESIGN PRINCIPLES



SPRING DEPENDENCY INJECTION

TECHNICAL DEBT

We've discussed multiple ways to maintain robust and clean software including

- Object Oriented Design
- Unit Testing
- Refactoring Your Code
- Using Well Known Design Patterns

TECHNICAL DEBT

When you must make a change, you either

- make a fast change with less quality or
- make a quality change that takes more time

The quality change is more maintainable over time

Sometimes the quick change is the correct thing to do, but creates technical debt which should eventually get cleaned up

TECHNICAL DEBT

The longer you wait to clean up technical debt

- the more expensive it is to clean up
- the slower you'll be to resolve customer issues

TECHNICAL DEBT

No matter how good your team is, you will incur technical debt over time

The key is to constantly go back and keep the amount of technical debt low

SOLID PRINCIPLES

A tool for helping keep technical debt low

Introduced by Robert C. Martin (Uncle Bob) in 2000

Single Responsibility Principle
Open Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

SOLID PRINCIPLES: **S**INGLE RESPONSIBILITY

A class should only be responsible for one system functionality

Keeps the complexity of the code low by separating the logic into multiple classes

SOLID PRINCIPLES: **S**INGLE RESPONSIBILITY

```
public float getTotal(List<String> items) {
    float total = 0;
    for(String item : items) {
        total += menuDatabase.query("get price for " + item);
    }
    return total;
}
```

SOLID PRINCIPLES: **S**INGLE RESPONSIBILITY

```
public float getTotal(List<String> items) {
    float total = 0;
    for(String item : items) {
        total += menu.getPrice(item);
    }
    return total;
}
```

SOLID PRINCIPLES: **O**PEN CLOSED

Objects should be open for use, but closed to modification

We've already seen this using Encapsulation

Other objects should be able to use your class, but not directly modify your stored state

Makes your object's state/behavior more predictable

SOLID PRINCIPLES: LISKOV SUBSTITUTION AND INTERFACE SEGREGATION

These will be left for you to research on your own

Links are available in Course Contents for this week

SOLID PRINCIPLES: **D**EPENDENCY INVERSION

When objects are "tightly coupled", you can't change one object without needing to change the other

By **decoupling** the objects, you're loosening that coupling

You'll never completely **decouple**

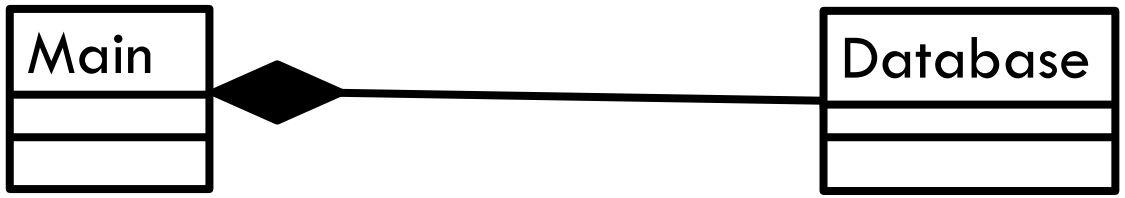
SOLID PRINCIPLES: **D**EPENDENCY INVERSION

Dependency Inversion is a principle for **decoupling** high-level objects from low-level objects

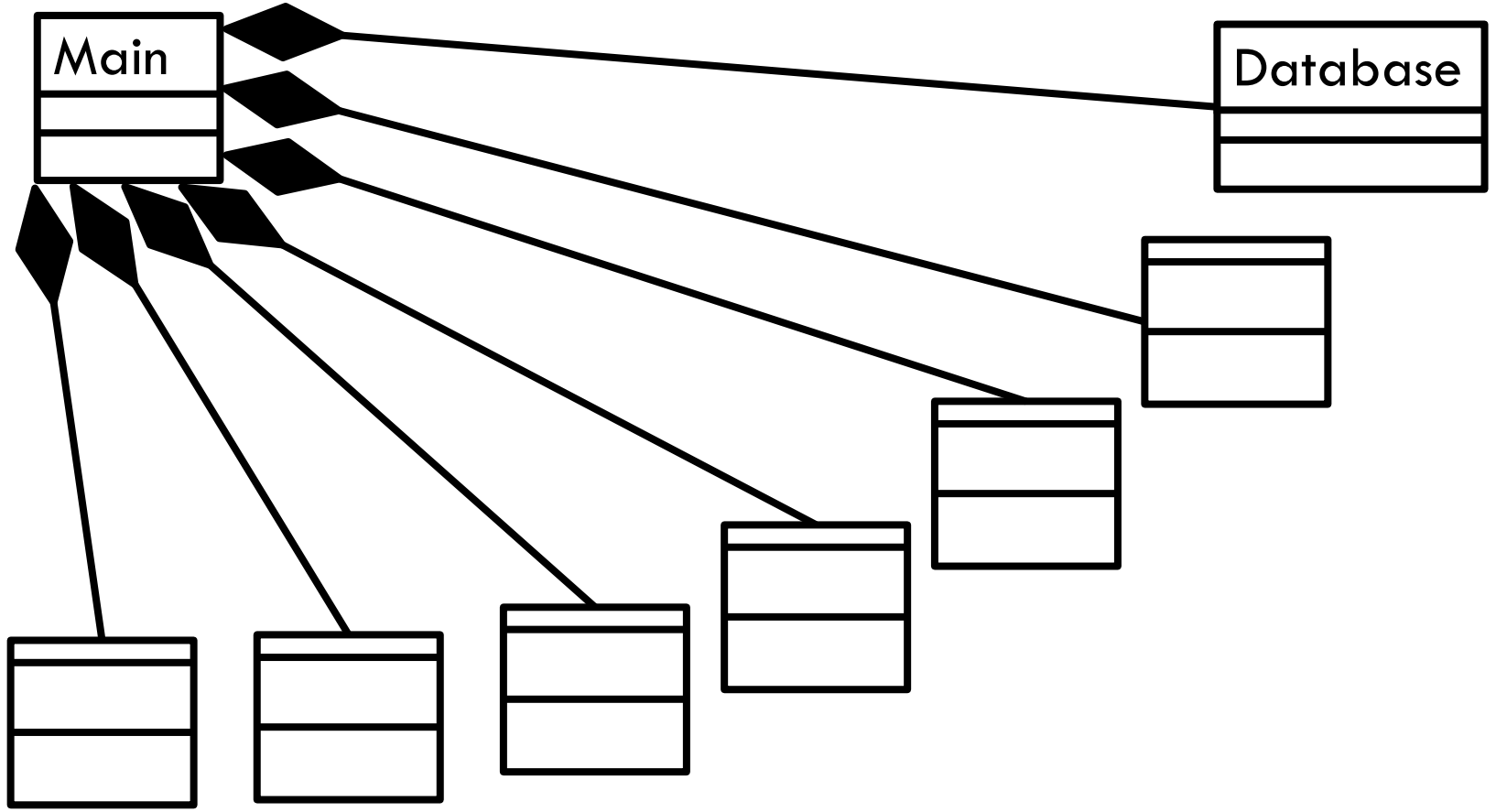
The high-level abstraction should not directly rely on the low-level details

Example: if the Main class runs the entire application, it shouldn't be responsible for initializing a low-level database

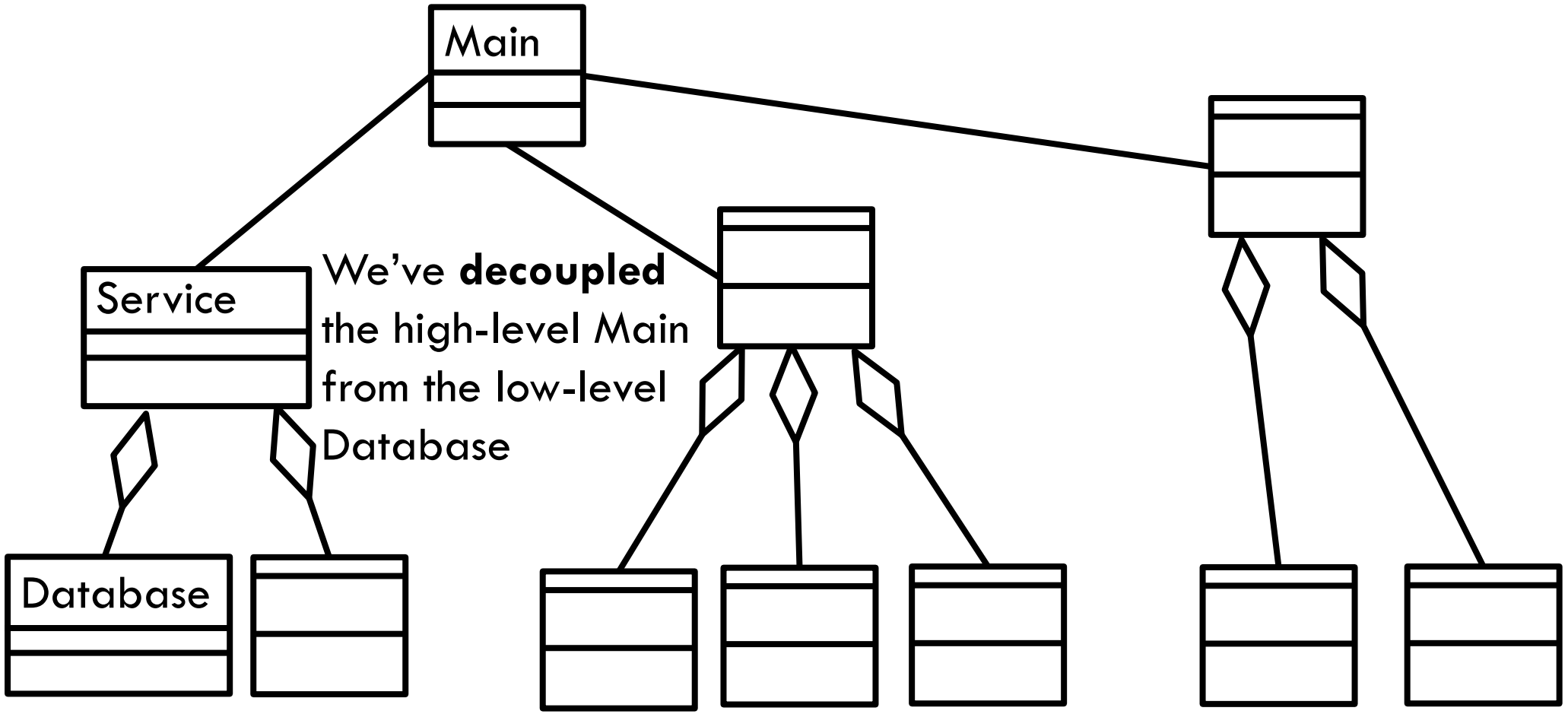
SOLID PRINCIPLES: **D**EPENDENCY INVERSION



SOLID PRINCIPLES: **D**EPENDENCY INVERSION



SOLID PRINCIPLES: **D**EPENDENCY INVERSION



INVERSION OF CONTROL

Inversion of Control principle (IoC) let's a container or framework manage the objects instead

IoC helps us follow the **Dependency Inversion** principle

- Our classes don't have to manage creating objects itself
- Classes are easier to maintain
- Classes are easier to test

DEPENDENCY INJECTION

Dependency Injection (DI) is a pattern for implementing the **IoC principle**

DI "injects" the objects that a class depends on

REVIEW THE TERMS!

Decoupling: Loosening the required maintenance between two objects

Dependency Inversion Principle: Decoupling high-level objects from low-level objects

Inversion of Control Principle (IoC): Let's a container or framework manage the objects for us

Dependency Injection (DI): A pattern for *implementing* the **IoC principle**

DEPENDENCY INJECTION

DI "injects" the objects that a class depends on

This can be as simple as *passing the object into* the constructor instead of *creating the object in* the constructor.

DEPENDENCY INJECTION

```
class Restaurant {
    private Menu menu;
    public Restaurant() {
        this.menu = new Menu();
    }
}
```

```
class Restaurant {
    private Menu menu;
    public Restaurant(Menu menu) {
        this.menu = menu;
    }
}
```

SPRING IOC

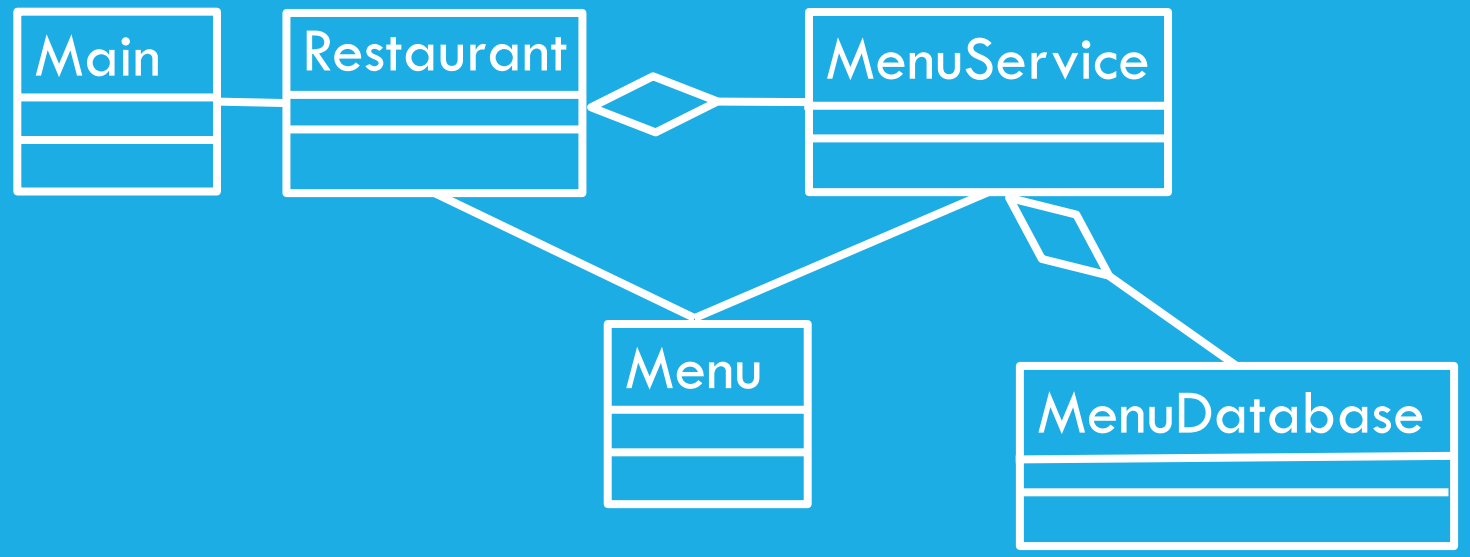
Spring framework is a library we can download, for example using Maven

Remember, **IoC** let's a container manage the objects for us

Spring IoC includes a container called **ApplicationContext**

SPRING IOC

ApplicationContext is responsible for creating and configuring objects, which Spring calls **beans**



LET'S TRY IT IN INTELLIJ

Let's start a Restaurant Application using Spring IoC

SPRING IOC

@Configuration tells Spring where the setup is

@ComponentScan(value = {"edu.dmacc.coma502"}) tells Spring to search for all annotations in the given package

new AnnotationConfigApplicationContext creates an IoC container to manage the annotations

context.getBean(Restaurant.class); gets a bean

SPRING IOC

@Component defines the class as a bean

@Service and @Repository are also @Component and simply make the code more readable

SPRING IOC

The ApplicationContext use to be configured using XML

```
<bean id="menuDatabase" class="edu.dmacc.coma502.database.MenuDatabase" />
<bean id="menuService" class="edu.dmacc.coma502.services.MenuService">
    <constructor-arg type="MenuDatabase" ref="menuDatabase" />
</bean>
<bean id="restaurant" class="edu.dmacc.coma502.Restaurant">
    <constructor-arg type="MenuService" ref="menuService" />
</bean>
```

SPRING IOC

XML-based makes the configuration more visible, but it can become hard to read and lengthy

```
<bean id=“menuService"
class="edu.dmacc.coma502.services.MenuService">
    <constructor-arg type="MenuDatabase" ref="menuDatabase" />
</bean>
```

Annotation-based is more more “magic”, but also more concise

```
@Service
public class MenuService {
    @Autowired
    public MenuService(MenuDatabase menuDatabase) {
```

SPRING FRAMEWORK

We only covered Spring IoC, but there's lots of other Spring libraries

These libraries provide templates and implementations for many common software needs, significantly cutting down on human error and reinventing the wheel

- Spring JDBC
- Spring MVC
- Spring Security
- Spring Boot
- And more…

FINAL PROJECT

Due this Sunday at noon!

In-class time now to work on your final project

I'm available for questions or design help

ASSIGNMENT

No Homework or quiz!

SEE YOU JANUARY 24TH!

Don’t forget to ask question early in the week

REFERENCES

- Baeldung.com. (16 Oct 2019). Intro to Inversion of Control and Dependency Injection with Spring. [online] Available at https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring [Accessed 10 Jan. 2020]
- Baeldung.com. (31 Dec 2019). A Comparison Between Spring and Spring Boot. [online] Available at https://www.baeldung.com/spring-vs-spring-boot [Accessed 10 Jan. 2020]
- Simon LH. (1 Jan 2019). SOLID Principles Explanation and Examples. [online] Available at https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4 [Accessed 10 Jan. 2020]