Object Oriented Design

OOD Principles and Patterns

Dr. Seán Russell

School of Computer Science,
University College Dublin

Lecture 07

Principles and Patterns

- We cannot make methodological recommendations to ensure we produce only good designs
- A large amount of experience of object-oriented modelling and design has now been gained
- With this experience some of the **properties** that can make a design successful or not are becoming better understood

Design Knowledge

- The knowledge accumulated by object-oriented designers falls into two distinct categories
  - Some widely accepted high-level **principles**
  - Some lower-level patterns seen in design
- The lower-level **design patterns** are more concerned with specific problems, and strategies for overcoming them

Table of Contents

S.O.L.I.D.

- S.O.L.I.D is an acronym for the first five object-oriented design (OOD) *principles* described by Robert C. Martin
- Together they make it easy for a programmer to develop software that are easy to maintain and extend
  - Single-responsibility Principle
  - Open-closed Principle
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency Inversion principle

Section Contents

- The Single-Responsibility Principle states that:

S.R.P.

A class should have one and only one reason to change, meaning that a class should have only one job.

- To understand this we will look at an example of an application to sum all of the areas of a number of shapes

Alternate Definition

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

- We will analyse this concept with an example relating to shapes
- An interface and several implementing classes will allow us to use polymorphism
- We will create a class that can calculate the area of shape objects in a data structure

```java
public interface Shape {
}
```

```java
public class Circle implements Shape {
    private int radius;
    public Circle(int r) {
        radius = r;
    }
}
```

```java
public class Square implements Shape {
    int length;
    public Square(int r) {
        length = r;
    }
}
```

```java
public class AreaCalculator {
    public List<Shape> shapes;
    public AreaCalculator(List<Shape> sps) {
        shapes = new ArrayList<Shape>();
        shapes.addAll(sps);
    }
    public double sum() {
        double sum = 0;
        for(Shape s: shapes) {
            if(s instanceof Circle){
                sum += Math.pow( ( (Circle) s ).radius, 2) * Math.PI;
            } else if(s instanceof Square){
                sum += Math.pow( ( (Square) s ).length, 2);
            }
        }
        return sum;
    }
}
```

- To use the `AreaCalculator`, we need to have output
  - This could be in JSON, HTML, plain text or some other format
- Adding this to the `AreaCalculator` class, the logic of which formats to use and how they are structured would have to be in it
- The `AreaCalculator` class **should not** care about formatting details
- Instead, we create another class responsible for outputting the result in the correct format

```java
public class AreaOutputter {
    private AreaCalculator calc;
    public AreaOutputter(AreaCalculator ac) {
        calc = ac;
    }
    public String getJSON() {
        ...
    }
    public String getHTML() {
        ...
    }
}
```

- If we follow this principle, then changes to our system will require only changes to parts of the code
- In this example, a change in the JSON format will require only a change to the output code
- This can make your code much easier to debug, test and maintain

- The single responsibility principle is straightforward, but still very easy to get wrong
- Applied too strictly and we will have many interconnected classes
  - Some containing only a single function
- Applied too loosely and we will have classes that contain lots of code

Section Contents

- The Open-Closed Principle states that:

> Objects or entities should be **open** for extension, but **closed** for modification.

- The open-closed principle was expressed by Bertrand Meyer in 1988 in the influential book Object Oriented Software Construction

- This principle is again concerned with the effects of **change** within a system, and insulating modules from the changes in others

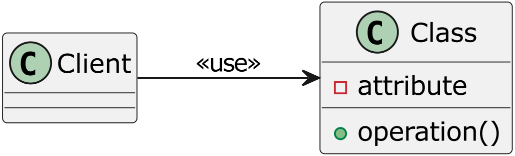
- Consider the situation where one module in a system makes use of the services provided by another
- It is usual to call the first module the **client** and the second the **supplier**

- A module is said to be closed if it is **not subject to further change**
- This means that client modules can use the supplier and not worry about future changes
- Closing a module means it is a **stable** component of the system
  - It should not change and adversely affect the rest of the design

- A module is open if it is still available for extension
  - This means adding to or enlarging its its capabilities
- Having open modules is desirable, because this will make it possible to extend and modify the system
- As system requirements are seldom stable, the ability to extend modules easily is an important aspect of keeping maintenance costs down

- We want modules that can be extended without being changed
- A solution to this is to distinguish between the interface and implementation of a module
- If client modules **only depend on the interfaces** of their suppliers, then the **implementations could be modified** without affecting clients
- OOP languages provide a number of ways to do this

- Data abstraction is intended to separate the interface of a data type or class from its implementation
- This might enable the construction of modules that are simultaneously open and closed
- To achieve data abstraction we assign an access level, such as 'public' or 'private', to each feature of a class



- From a clients point of view, the interface is simply those features that are visible
- Invisible features can be changed, removed or added to, as long as the visible interface is unchanged
- The example could be implemented as follows

```
public class Supplier {
  private int attribute;

  public void operation() {
    // Implementation of operation
  }
}
```

```java
public class Circle implements Shape {
    private int radius;
    public Circle(int r) {
        radius = r;
    }
    public int getRadius(){
        return radius;
    }
}
```

```java
public class Square implements Shape {
    private int length;
    public Square(int r) {
        length = r;
    }
    public int getLength(){
        return length;
    }
}
```

```java
public double sum() {
  double sum = 0;
  for (Shape s : shapes) {
    if (s instanceof Circle) {
      sum += Math.pow( ( (Circle) s).getRadius(), 2) * Math.PI;
    } else if (s instanceof Square) {
      sum += Math.pow( ( (Square) s).getLength(), 2);
    }
  }
  return sum;
}
```

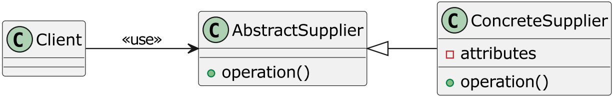
- The implementation of the public method can be changed
- Similarly, private fields can be added or removed as required
- What must be unchanged is the visible interface consisting of the **name** and **signature** of the public methods

- This approach to the implementation of the open-closed principle has a number of limitations
- Firstly, the client module is technically not even closed, as modifications to the system require changes to the code in the client class
- Secondly, in the data abstraction approach the interface required by client modules is left implicit
- Clients can make use of all features of the supplier that are visible to them, but need not do so

- In C++, for example, a class definition is typically split between a header file, which is physically incorporated into the client module, and an implementation file
- Any change to a header file, such as the addition of a new field, requires client modules to be recompiled, even if the change is invisible to them
- It would be preferable if an implementation of the open-closed principle could be found that was language independent

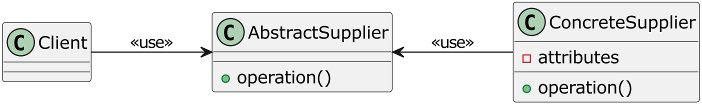
- In practice, different clients of a module may make use of different subsets of the visible interface of a module
- This makes it difficult to know exactly what changes to a module will affect a given client
- Both documentation and maintainability would be improved by an approach which explicitly documented the interface required by a client module

- An alternative approach is to use an abstract interface class
- A class diagram showing the general structure of the design using this technique is shown here



- Here the abstract supplier class defines the interface for the supplier class

- `AbstractSupplier` is abstract, the client actually uses a concrete subclass
- The `AbstractSupplier` contains only the implementation details that **will not be** changed
- This diagram shows the usage dependencies between the classes



- The `AbstractSupplier` class is an example of an open module, as it can be extended by subclasses
- These extra subclasses might provide alternative implementations of the interface, or might add new features
- The booking class in the restaurant system shows this feature of extensibility

- The `AbstractSupplier` is more closed than the supplier class in the first diagram
- Changes to the implementation of supplier objects will be made to its concrete subclasses
- However, changes to the interface may be required as the system evolves, and these will require changes to the abstract class
- The use of abstract interface classes is a fundamental technique in object-oriented design

```java
public interface Shape {
    public double getArea();
}
```

```java
public class Circle implements Shape {
    private int radius;
    public Circle(int r) {
        radius = r;
    }
    public int getArea(){
        return Math.pow(radius, 2) * Math.PI;
    }
}
```

```java
public class Square implements Shape {
    private int length;
    public Square(int r) {
        length = r;
    }
    public double getArea(){
        return length * length;
    }
}
```

```java
public class AreaCalculator {
    public List<Shape> shapes;
    public AreaCalculator(List<Shape> sps) {
        shapes = new ArrayList<Shape>();
        shapes.addAll(sps);
    }
    public double sum() {
        double sum = 0;
        for (Shape s : shapes) {
            sum += s.getArea();
        }
        return sum;
    }
}
```

Section Contents

Liskov substitution principle

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

- All this is stating is that every subclass class should be substitutable for their parent class

- If abstract interface classes are used, the implementation will make extensive use of polymorphism
- This means that the clients will call supplier operations through a reference of type ‘AbstractSupplier’
- When this code runs, the 'supplier' variable does not contain a reference to an instance of ‘AbstractSupplier’
- Instead, it will hold a reference to an instance of ‘ConcreteSupplier’

- If polymorphism is not to cause problems in programs, the semantics of the language must ensure that something like the following is true:

- If a client holds a reference to an object of class T, then it will work equally well when provided with a reference to an object of class S
    - where S is a specialization of T

- In this context the Liskov substitution principle can be stated as follows

Definition

Class S is correctly defined as a specialization of class T if the following is true: for each object s of class S there is an object t of class T such that the behaviour of any program P defined in terms of T is unchanged if s is substituted for t

- Less formally, this means that instances of a subclass can replace instances of a superclass without any effect on client classes or modules

- Although described in terms of types and subtypes, the Liskov substitution principle effectively defines the meaning of the idea of generalization
- In UML, the different forms of generalization that are defined, between classes, use cases and actors
- Exactly what this means will depend on the type of entity being considered

- Generalization between classes can only be correctly used where occurrences of the superclass can be substituted by occurrences of the subclass
- If a program would behave differently using a subclass object, then generalization is being used incorrectly
- It is possible to implement an operation in a subclass to undermine substitutability, by providing an implementation which causes subclass instances to behave in a completely different way from superclass instances

Section Contents

Interface segregation principle

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use

- Assume that we want to add functionality to our shape example from earlier
- We would like to develop a volume calculator similar to the area calculator
- The first step would be to add another method to the `Shape` interface

Shape Interface

```java
public interface Shape {
    public double getArea();
    public double getVolume();
}
```

- Any shape we create must implement the `getVolume` method
  - Even flat shapes like circles and squares
- This interface forces classes to implement methods that it has no use for
- The interface segregation principle says that this should not be done
- Instead, we should create a separate interface for the volume requirement

SolidShape Interface

```java
public interface SolidShape {
    public double getVolume();
}
```

```java
public class Cube implements SolidShape, Shape {
  private int length;

  public Cube(int r) {
    length = r;
  }

  public double getArea() {
    return 6 * length * length;
  }

  public double getVolume() {
    return Math.pow(length, 3);
  }
}
```

- This technique solves the problem because
  - The original `Shape` interface remains unchanged
  - The `SolidShape` interface can be used by clients
- If we extend the Shape interface in the SolidShape interface, then SolidShape can be used anywhere a Shape is used

SolidShape Interface

```
public interface SolidShape extends Shape {
    public double getVolume();
}
```

Section Contents

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

- The definition might sound complicated, but the principle is easy to understand
- The primary concern is that of decoupling
- Assume we have a class called `PasswordReminder`, that is used in a complex system to aid in password reminders
- In order to remind the user of their password, the class must have a connection to the database
- This introduces a dependency to a low level connection class

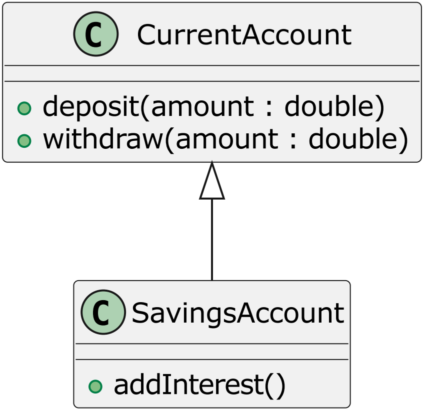
- The database connection (e.g. `MySQLConnection`) is a low level class and `PasswordReminder` is a high level class
- But the password reminder is forced to depend on this class
- Later if we decide to use a different database engine, we would have to update the code in the `PasswordReminder` class, as well as any other that uses the database
- This violates the open-closed principle

- The `PasswordReminder` class should not care what database engine we use
- Instead we use an interface or class to abstract the low level database connection
- We then use this high level class in our password reminder

Section Contents

- Another principle states that all subclasses in generalization relationships should be concrete
- Alternatively you could say that all non-leaf classes in a generalization hierarchy should be abstract
- This principle can be summarized in the slogan 'no concrete superclasses'

- Suppose a bank is implementing classes to model different types of account, and initially only the current account class is define with the normal withdraw and deposit operations
- Later, the bank adds savings accounts, which have the additional ability to pay interest
- These new accounts share much functionality with current accounts, so the savings account is defined as a subclass of the current account


C CurrentAccount
deposit(amount : double)
withdraw(amount : double)
C SavingsAccount
addInterest()

- The current account class is both concrete and a superclass
- Violation can lead to significant problems as a design evolves
- E.g. suppose the withdrawal operation originally defined did not permit accounts to become overdrawn
- Later, overdrafts are allowed on current accounts but not on savings accounts
- It is not easy to modify the design to provide this functionality

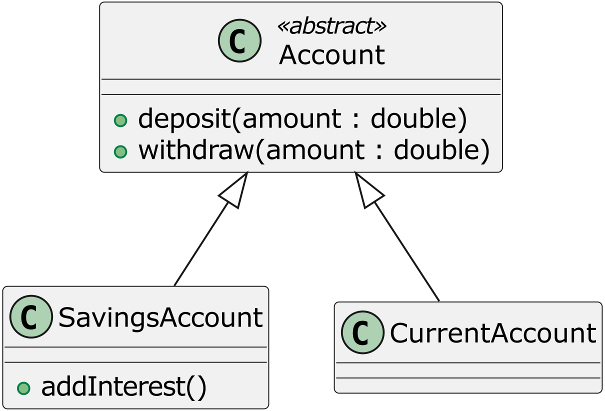
- If we modify the withdrawal in the current account, it is inherited by the savings account
- We could then override the operation in the savings account class, to provide the original functionality, but that is not a good solution
- It would have the consequence of replicating the code that actually performed the withdrawal in both classes
- Code replication is a strong sign of a design error

- We could check the run-time type of the account the withdrawal was being made from
- This style has serious drawbacks, primarily that the superclass **makes explicit reference** to its subclasses
- When a new subclass is added, the code in the current account class will have to be updated
- This style of programming is also taken as evidence of shortcomings in the design of a system

- Similar problems arise if a function has to be defined on current accounts only
- Suppose the bank wants the ability to cash cheques using a current account, but not a savings accounts
- The only obvious implementation of this function is for it to check the run-time type of its argument

```
void cashCheque( CurrentAccount a ) {
  if (a instanceof SavingsAccount) {
    return;
  }
  // Cash Cheque
}
```

- The problems in these cases arise from the fact that the current account class is performing two roles
  - As a superclass, it is defining the interface which must be implemented by all account objects
  - But it is also providing a default implementation of that interface
- In the cases we have examined, these roles conflict
- A general solution to this sort of problem is to adopt the rule that all superclasses should be abstract


C
«abstract»
Account
deposit(amount : double)
withdraw(amount : double)
C
SavingsAccount
addInterest()
C
CurrentAccount

- Now the withdraw operation in the 'Account' superclass can perform the basic withdrawal
- The code that checks that a savings account is not becoming overdrawn can be placed in the function that overrides it
- As 'SavingsAccount' is no longer a subclass of 'CurrentAccount', instances of the subclass can no longer be passed to functions like 'cashCheque'

Section Contents

- The Law of Demeter (LoD) or principle of least knowledge is a design guideline for developing software
    - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit
- The fundamental idea is that an object should assume as little as possible about the structure or properties of anything else
    - This is in accordance with the principle of "information hiding"

- In OOP the Law of Demeter can be called the "Law of Demeter for Functions/Methods" (LoD-F)
- An object `A` can call a method of an object instance `B`
- But object `A` should not "reach through" object `B` to access another object, `C`, to call its method
  - Doing so would mean that object `A` implicitly requires greater knowledge of object `B`'s internal structure
- Instead, B's interface should be modified so it can directly serve A's request, propagating it to any relevant subcomponents

- Alternatively, A might have a direct reference to object C and make the request directly to that

- If the law is followed, only object B knows its own internal structure

- More formally, the Law of Demeter for functions requires that a method `m` of an object `O` may only invoke the methods of the following kinds of objects:

1. `O` itself
2. `m`'s parameters
3. Any objects created/instantiated within `m`
4. `O`'s direct component objects
5. A global variable, accessible by `O`, in the scope of `m`

- In particular, an object should avoid invoking methods of a member object returned by another method
- For OOP languages that use a dot as field identifier, the law can be stated as "**use only one dot**"
- That is, the code `a.b.Method()` breaks the law where `a.Method()` does not

- The resulting software tends to be more maintainable and adaptable
- Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers
- Research suggests that a lower Response For a Class (RFC) can reduce the probability of software bugs
  - RFC is the number of methods potentially invoked in response to calling a method of that class
- Following the LoD can result in a lower RFC

- LoD may also result in having to write many wrapper methods to propagate calls to components
  - This may add noticeable time and space overhead
- Research also suggest that an increase in Weighted Methods per Class (WMC) can increase the probability of software bugs
  - WMC is the number of methods defined in each class
- Following the LoD can also result in a higher WMC

- Making use of the Spring-Boot framework requires the application a number of design patterns

- This section briefly introduces these patterns

Table of Contents

Section Contents

- The Model View Controller (MVC) pattern is an architectural pattern for the design of applications with user interfaces
- The core idea is the separation of the application into three interconnected parts
  - The **model** - which represents the internal data of the system
  - The **view** - which represents the user interface
  - The **controller** - which is the software linking the model and view

- We will be using spring-boot with the Thymeleaf plugin to create our application
- In this context, some of these components are slightly different
  - The **view** components are represented by HTML based templates
  - The **controller** can be seen as two separate roles
    - An object(s) responsible for implementing the logic required to complete use cases
    - A controller that is responsible for processing requests and formatting data

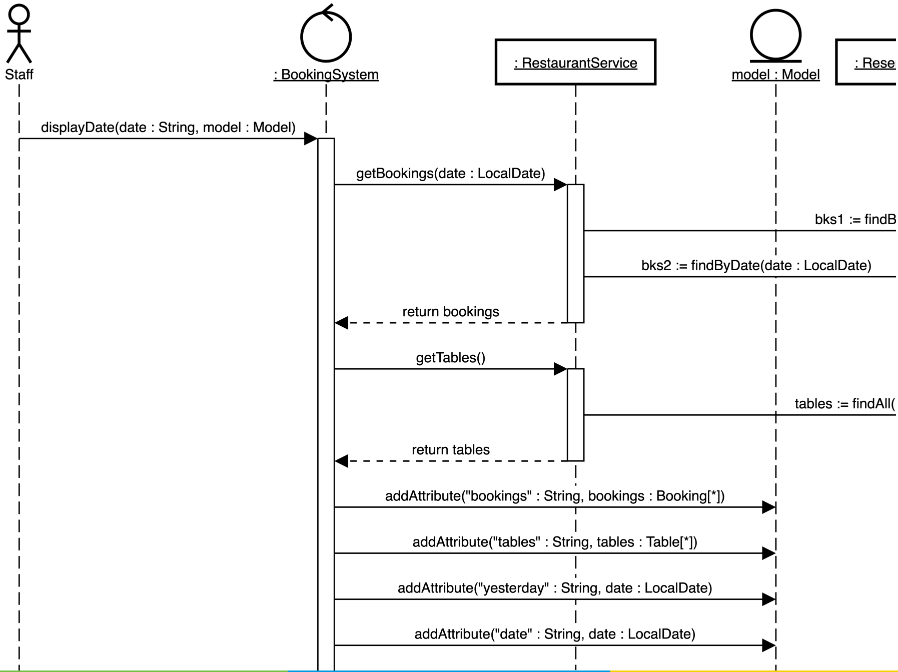
- Thymeleaf is a template engine
- Templates are provided in HTML with some modified syntax
- The template engine is provided data from the Java code to complete the page

- The thymeleaf syntax is embedded within HTML tags
- Relevant attributes are started with `th:`
- Here are some examples of some template syntax:

```
<a th:href="@{/(date=${yesterday})}">....
<tr th:each="table : ${tables}">
    <td th:text="${table.name}">name</td>
    <td th:text="${table.number}">number</td>
</tr>
```

- In order for these templates to successfully generate a page, the required data must be supplied
- This is done in the controller object
- This requires several steps to work
  - We must tell spring-boot that the class is a controller
  - We must tell spring-boot what URL the method is for
  - We must take a Model object as a parameter in the method

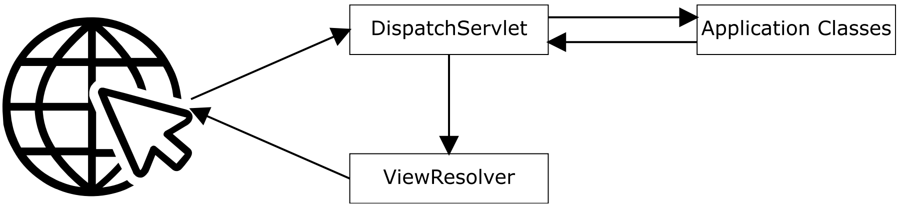
```java
@Controller
public class BookingSystem {
  @GetMapping("/")
  public String displayDate( @RequestParam(name="date")
  ↪  String date, Model model){
    ...
    List<Booking> bookings  = restaurant.getBookings(date);
    List<Table> tables = restaurant.getTables();
    model.addAttribute("date", date.toString());
    model.addAttribute("yesterday", date.minusDays(1));
    model.addAttribute("tomorrow", date.plusDays(1));
    model.addAttribute("bookings", bookin gs);
    model.addAttribute("tables", tables);
    return "index";
  }
  ...
}
```

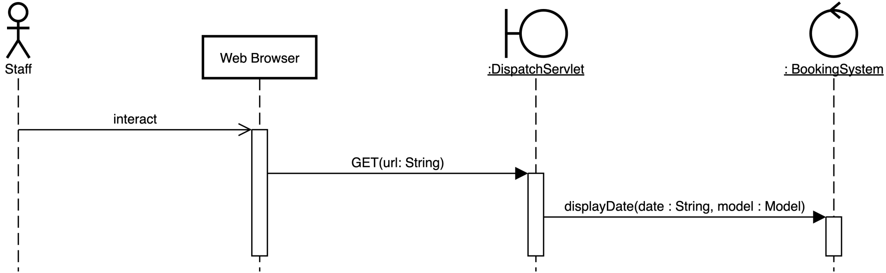

Staff
: BookingSystem
: RestaurantService
model : Model
: Rese
displayDate(date : String, model : Model)
getBookings(date : LocalDate)
bks1 := findB
bks2 := findByDate(date : LocalDate)
return bookings
getTables()
tables := findAll(
return tables
addAttribute("bookings" : String, bookings : Booking[*])
addAttribute("tables" : String, tables : Table[*])
addAttribute("yesterday" : String, date : LocalDate)
addAttribute("date" : String, date : LocalDate)

Section Contents

- The front controller pattern is commonly used in web-based applications
- The front controller is a controller that handles all requests for the application
  - These requests are then forwarded to the relevant part of the system to be handled
- This provides an interface for common behaviour like security and presenting different views

- The spring-boot framework is designed using the front controller pattern
- All HTTP GET and POST requests to the application are received by the `DispatchServlet`
  - Depending on the URL requested and the necessary authorisations, the request is then passed to the correct **controller in our code**


DispatchServlet
Application Classes
ViewResolver


Staff
Web Browser
:DispatchServlet
: BookingSystem
interact
GET(url: String)
displayDate(date : String, model : Model)

- Because the spring-boot framework is designed using the front controller pattern, we do not need to consider how to get requests to the correct controller
- Instead, we simply need to tell spring-boot which methods in our controller will respond to different URLs
  - This is done by annotating the relevant methods

Section Contents

- A Data Transfer Object (DTO) is an object that carry data between processes
  - The main purpose is to reduce transmissions of data
- These objects are typically distinct from the model classes representing the same information

- Spring-boot uses template engines to produce the web pages the user sees
- Pages are defined using HTML, CSS, and Javascript with some additional syntax
- This additional syntax allows us to insert data into the generated pages
- This data must be added to a model object to allow the page to be created

- There is no requirement to use data transfer objects in a spring-boot application
- However, they can be used to simplify interactions between our code and the front-end
- Rather than adding many individual values to the model, we can use a DTO object representing the information
- Similarly, when receiving a POST of a HTML form we can associate the form with a DTO and only require a single parameter

```java
@PostMapping("/walkin/")
public String addWalkIn(@ModelAttribute WalkInDTO wi,
  Model model) {

  // Code to add a new WalkIn booking to the system
```

```java
@PostMapping("/walkin/")
public String addWalkIn(
  @RequestParam(name="walkInDate") String date,
  @RequestParam(name="walkInTime") String time,
  @RequestParam(name="walkInCovers") int covers,
  @RequestParam(name="walkInTable") int tableNumber,
  Model model) {

  // Code to add a new WalkIn booking to the system
```

Section Contents

- The Data Access Object (DAO) pattern is a way of handling communication between our application and the database
- The idea is that our application should have an easy interface to perform CRUD operations
  - E.g. By calling a method like `User findByEmailAddress(String emailAddress);`

- Spring-boot allows for a number of different techniques to interact with a data store
- We will make use of the Jakarta Persistence API (JPA)
- JPA supports the generation of DAOs by defining repository interfaces
    - We do not have to write the database code!
- The classes we want to be remembered must be properly annotated in the code

- Queries are defined by correctly naming the methods in the repository and defining the correct parameters
- Method names make use of some keywords of SQL and instance variable names
- The instance variable names must match the spelling used in the class
    - E.g. For a String variable `phoneNumber` we would name the method `findByPhoneNumber(String pn);`

Keyword	Sample
Distinct	findDistinctByLastname
And	findByLastnameAndFirstname
Or	findByLastnameOrFirstname
	findByFirstname
Is, Equals	findByFirstnameIs
	findByFirstnameEquals
Between	findByStartDateBetween
LessThan	findByAgeLessThan
LessThanEqual	findByAgeLessThanEqual
GreaterThan	findByAgeGreaterThan

```
public interface ReservationRepository extends JpaRepository
 ↪ <Reservation, Long> {

    List<Reservation> findAllByDate(LocalDate date);

    List<Reservation> findAll();

    List<Reservation> findAllByDateAndTable(LocalDate date,
     ↪ Table table);

    Reservation findById(long id);
}
```

Section Contents

- Dependency injection is more of a programming technique than a pattern
- The basic idea is to **decouple** components of software from each other
  - This makes testing and integration easier
- This is done by having required objects be provided rather than having to create them

- Spring uses annotations to enable dependency injection
- These may be used by the system automatically
  - E.g. classes annotated with `@Controller` will be injected automatically into the `DispatchServlet`
- When the application is being started, spring-boot will find and create the appropriate objects to be injected where they are required

- We can also specify when we would like to have an object injected
  - This works for all of the classes that spring-boot creates for us
- This can be done using the `@Autowired` annotation
- It can be used with a field, setter method or constructor
  - Constructor is probably the best to use
- We just need define the parameters as normal and add the annotation

```java
@Controller
public class BookingSystem {
    private final RestaurantService restaurantService;

    @Autowired
    public BookingSystem(RestaurantService restaurantService) {
        this.restaurantService = restaurantService;
    }
```

```java
@Service
public class RestaurantService {
    private final CustomerRepository customers;
    private final ReservationRepository reservations;
    private final TableRepository tables;
    private final WalkInRepository walkIns;

    @Autowired
    public RestaurantService(CustomerRepository c, ReservationRepository r
    ↪   , TableRepository t, WalkInRepository w) {
        customers = c;
        reservations = r;
        tables = t;
        walkIns = w;
    }
```