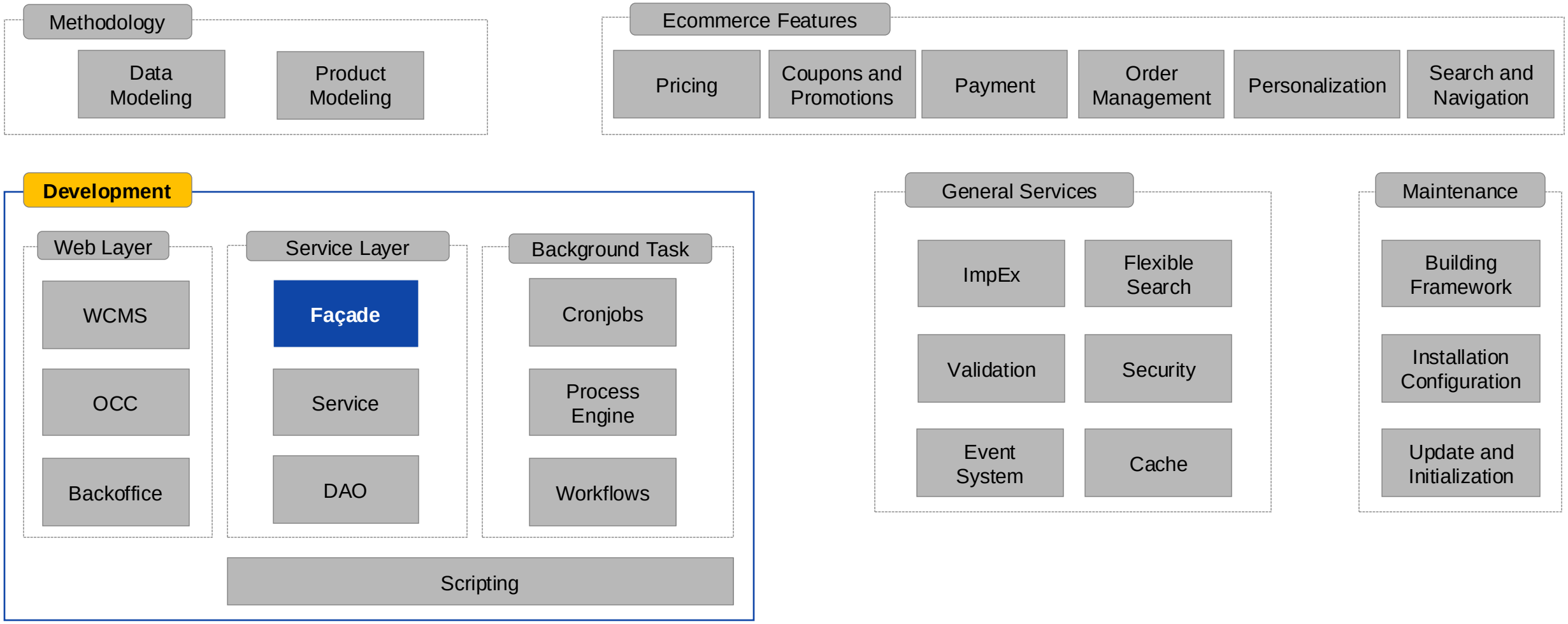
SAP Customer Experience

Façades

What we will cover in this topic



The Context

- The responsibility of a single Facade is to **integrate existing business services** from the full range of the SAP Commerce Cloud extensions and **expose a Data object** (DTO) response adjusted to meet the storefront requirements.

Preparation

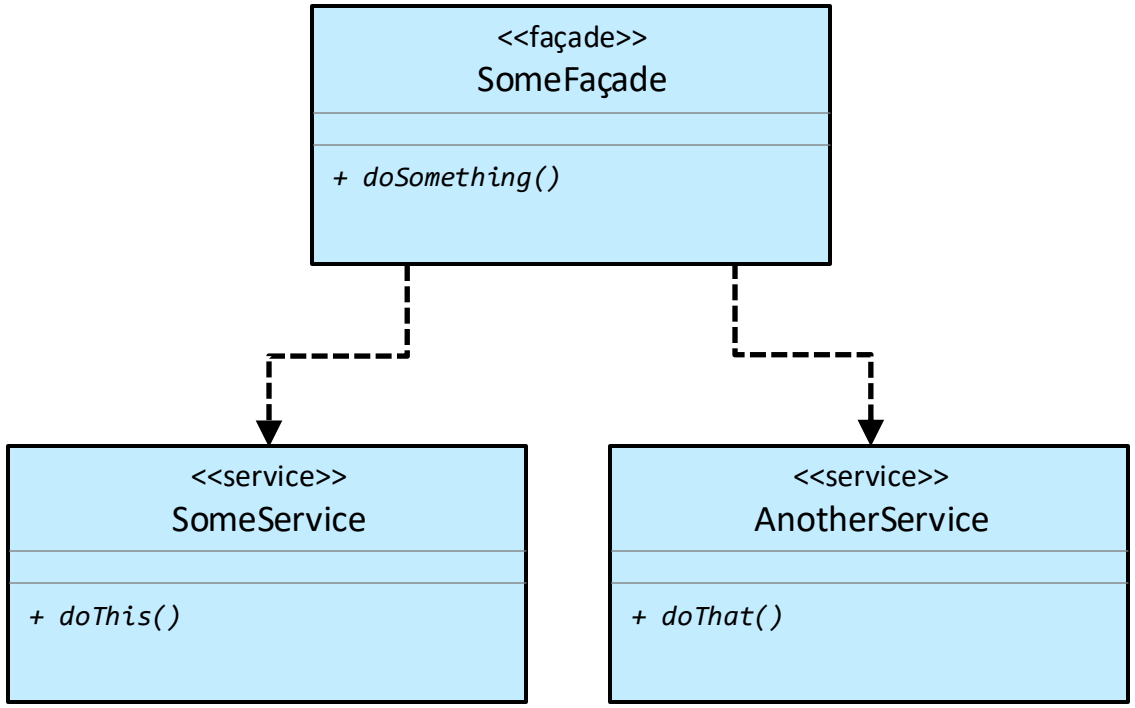
Complete the step P1 of the Façades exercise

(The setup ant target will compile your system during the lecture)

Façades Overview

The Façades Design Pattern

- A Façade offers a simplified interface to more complex codes/interfaces.
- It is presented as a high-level business operation.
- Internally, a Façade can orchestrate calls to other business operations or to lower-level operations.
- A Façade typically returns simpler objects than the ones returned by the underlying services.
  - These objects are called Data Objects or Data Transfer Objects (DTOs).



Challenges on Implementing a Façade in SAP Commerce Cloud

- Services usually return Model items, while Façades return DTOs.
- Extra code needs to be generated, e.g.:
  - Defining the DTOs
  - Providing conversion from Models to DTOs or from DTOs back to Models
- SAP Commerce Cloud
  - provides an easy way of defining and generating DTOs.
  - offers a re-usable solution for the conversion between Models and DTOs.

```
public class SomeFacadeImpl implements SomeFacade
{
    private SomeService someService;
    private AnotherService anotherService;

    public SomeData doSomething()
    {
        // Model items returned by underlying services
        SomeModel someModel = someService.doThis();
        AnotherModel anotherModel = anotherService.doThat();

        // Data Transfer Object (DTO)
        SomeData someData;

        // Conversion from SomeModel and AnotherModel
        // into SomeData goes here

        return someData;
    }
}
```

Bean Generation

The Use of Bean Instances as MVC Data Objects

- Custom data objects (instances of JavaBean classes) carry data to the view
  - Populated with only the display-ready values that the target view needs
- Data objects are attached to a view by its controller
- To simplify the Spring MVC controller, we typically create a façade class with a method that obtains the Data Objects for the controller to send to the view
  - Typically, this method obtains its data from services that return ServiceLayer model objects (e.g. ProductModel, CategoryModel)

E.g. `List<MovieDetailData> getMovieDetailViewData(String movieID)`

Data Objects are also known as DTOs (Data Transfer Objects)

Auto-Generated JavaBean Classes – A Declarative Approach

- We can have JavaBean (and Enum) source code generated for us during ant builds
  - For each JavaBean class to be generated, a declaration must exist inside a **`resources/<extensionName>-beans.xml`** file
  - A JavaBean class declaration includes the fully-qualified class name, its superclass (optional), and the bean's “properties” (property names and Java types)
  - Each extension may contribute its own **`*-beans.xml`** file

`commercefacades-beans.xml`

```
<bean class="de.hybris.platform.commercefacades.order.data.DeliveryModeData">
   <property name="code" type="String"/>
   <property name="name" type="String"/>
   <property name="description" type="String"/>
   <property name="arriveByDate" type="java.util.Date"/>
   <property name="deliveryCost"
             type="de.hybris.platform.commercefacades.product.data.PriceData"/>
</bean>
```

What Gets Generated?

- Generate Java Beans from declarations within a **`*-beans.xml`** file

```
<bean class="org.training.data.MyPojo">
    <property name="id" type="String"/>
</bean>
```

```
public class MyPojo implements java.io.Serializable
{
    private String id;
    public MyPojo() {} //no-argument constructor
    public String getId() {...}
    public void setId(String id) {}
}
```

Why a Declarative Approach?

- A single JavaBean class definition can be split-up across multiple extensions
  - All partial declarations having the same class name are merged (from all extensions participating in the build) and generate a single JavaBean class
  - This way, an extension can be made optional
  - New, custom extensions can extend existing JavaBean definitions
- Java Enum classes (with singleton member values) can be defined similarly
- Generated classes are placed in **`platform/bootstrap/gensrc`**

Does this sound familiar?

It should! items.xml and beans.xml share the same paradigm

How Bean Definitions Get Merged

extension1-beans.xml

```
<bean class="org.training.data.MyPojo">
    <property name="id" type="String"/>
</bean>
```

extension2-beans.xml

```
<bean class="org.training.data.MyPojo">
    <property name="timeStamp"
              type="java.util.Date"/>
</bean>
```

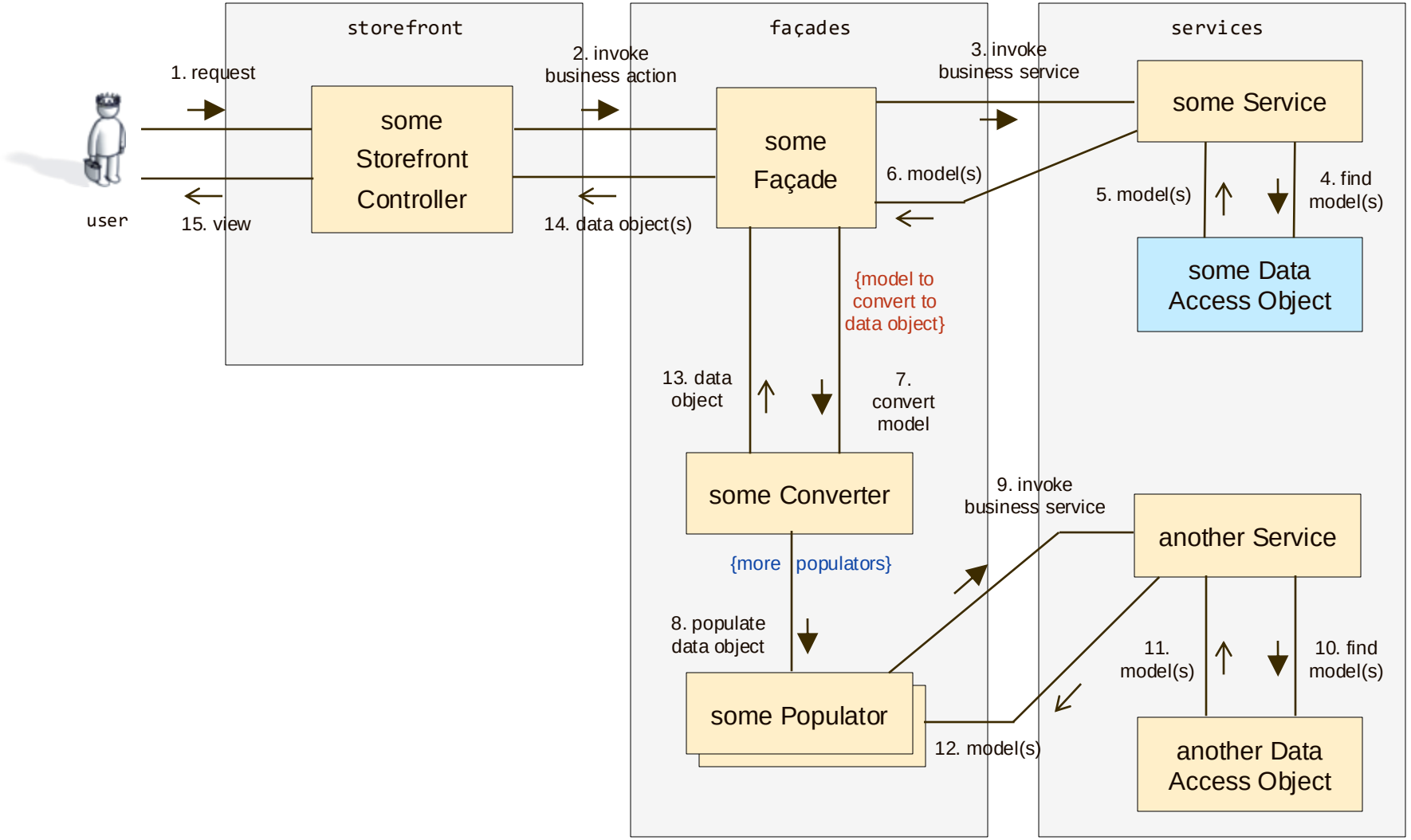
Triggered by: `ant all`

Generated in platform/bootstrap/gensrc

```
public class MyPojo implements java.io.Serializable
{
    private String id;
    private java.util.Date timeStamp;
    public MyPojo() {}
    public String getId() {...}
    public java.util.Date getTimeStamp() {...}
    public void setId(String id) {...}
    public void setTimeStamp(java.util.Date timeStamp) {...}
}
```

Conversion Process

Conceptual Interaction Diagram



Converters and Populators

Implementation of `Converter<SOURCE, TARGET>`

- Transforms an object of type `SOURCE` into an object of type `TARGET`
- Primary callback method is: `TARGET convert( SOURCE )`
  1. Instantiates a new, empty instance of `TARGET` (typically a DTO)
  2. Delegates the population to a list of Populators - passing in `SOURCE` and `TARGET` (see below)
  3. Afterwards return the populated `TARGET` instance

Implementation of `Populator<SOURCE, TARGET>`

- Sets values in `TARGET` instance based on values in `SOURCE` instance
- Primary callback method is: `void populate( SOURCE, TARGET )`
  - Uses values from `SOURCE` instance to populate values of `TARGET` instance

Type conversion is typically broken down into a sequence of population steps

- Converter delegates to one or more assigned Populators, in the assigned order
- Converters are normally just configured via Spring (a custom converter is **rare**)

A Typical Populator

CarBasicPopulator.java

```
public class CarBasicPopulator implements Populator<CarModel, CarData>
{
    @Override
    public void populate(final CarModel source, final CarData target)
     throws ConversionException
    {
        target.setDescription( source.getDescription() );   // E.g. String-to-String exact copy
        // populator can leverage services to get source price (double)
        double priceValue =
            getPriceService().getWebPriceForProduct(source).getPriceValue().getValue()
        DecimalFormat currencyFmt = DecimalFormat.getCurrencyInstance( getLocale() );
        String displayPrice = currencyFmt.format( priceValue );
        target.setPrice( displayPrice );   // target price: formatted String, source price: Double
        // . . . etc.
    }
    // property getters, setters, and private attributes not shown
}
```

myfacadesextension-spring.xml

```
<bean name="defaultCarBasicPopulator"
      class="org.training.facades.populators.CarBasicPopulator">
    <property name="priceService" ref="mySimplePriceService" />
</bean>
```

Use Case 1: Reorganize the Conversion Process via a New Converter

- The `platformservices` extension provides a base `abstractPopulatingConverter` bean
  - Allows you to define a new converter bean without having to write a Java class
  - Allows for easy reuse of populators (as beans backed by custom Java classes)

myfacadesextension-spring.xml

```
<alias alias="carConverter"
       name="defaultCarConverter" />

<bean  id="defaultCarConverter"
       parent="abstractPopulatingConverter">
    <property name="targetClass"
              value="my.project...data.CarData"/>
    <property name="populators">
        <list>
            <ref bean="carBasicPopulator"/>
            <ref bean="carFeaturesPopulator"/>
        </list>
    </property>
</bean>
```

Populators injected into converter bean here:

Use Case 2: Extend The Conversion Process via a New Populator

- How can type conversion be hooked-into without rewriting the basic code or existing converters?
  - Use a `modifyPopulatorList` to modify existing populator lists
    - defined in `converters-spring.xml of the` platformservices extension
    - available operations: **add** and **remove**
    - Processed by BeanPostProcessor

No real need for bean ID:

Name of converter bean defined in another, pre-existing extension:

myfacadesextension-spring.xml

```
<bean   parent="modifyPopulatorList">
    <property name="list" ref="productConverter"/>
    <property name="add"  ref="fooProductPopulator"/>
</bean>
```

Use Case 3: Extend the Conversion Process for Subtyping – 2 Approaches

- New attributes of extended types are to be transferred to view, e.g.:

```
<itemtype code="FooProduct" extends="Product" ...
    <attribute qualifier="bar" type="java.lang.String" ...
```

- Approach 1
  - Define a new converter bean whose “parent” is base type’s converter bean instead of the usual, `parent="abstractPopulatingConverter"`
  - Spring’s `<list merge="true">` can *merge* new populators with ’inherited’ ones, if desired

```
<bean id="fooProductConverter" parent="productConverter" >
    <property name="populators" >
        <list merge="true" >
            <ref bean="fooProductAdditionalAttribsPopulator" />
        </list> . . .
```

Inherits populator list; merges two lists instead of replacing

  - Façade must decide which converter to use per `SOURCE` instance, based on its type

- Approach 2
  - Merge new subtype’s attributes/properties into base type’s (e.g., `ProductData`) DTO and Converter
  - Keep existing converter, but add additional populator using a `modifyPopulatorList`
  - Newly added populator must check `SOURCE` item type; accesses new attributes only if appropriate

The Façade Class: Implementation to Include the Conversion Process

- The façade class needs to be written – it typically looks like this:

```
public class DefaultCarFacade implements CarFacade
{
   private CarService carService;
   private Converter<CarModel, CarData> carConverter;

   public CarData getCarOfTheYear(final int year)
   {
      CarModel car = carService.getFeaturedCar(year);
      CarData carData = carConverter.convert(car);
      return carData;
   }

   // other facade methods

   // getters & setters (for carConverter and carService injections) not shown
   //    these will be injected in <extensionname>-spring.xml
}
```

Converter injected here (using the corresponding setter) by *myfacadesextension*-spring.xml

Façade Declaration: Associating the Converter

- Declare the façade as a Spring bean
- Inject the converter bean, along with all the other service beans the façade will need

myfacadesextension-spring.xml

```
<alias alias="carFacade"
       name="defaultCarFacade" />
<bean id="defaultCarFacade"
      class="my..commercefacades.car.impl.DefaultCarFacade">
    <property name="carService" ref="carService"/>
    <property name="customerReviewService" ref="customerReviewService"/>
    <property name="userService" ref="userService"/>
    <property name="modelService" ref="modelService"/>

    <property name="carConverter" ref="carConverter"/>

</bean>
```

Converter injection within myfacadesextension-spring.xml

Using the Façade

- Within the controller class:

```
@Controller
@RequestMapping(value = "/**/car")
public class CarPageController extends AbstractPageController
{
    @Resource(name = "carFacade")
    private CarFacade carFacade;

    //...


    @RequestMapping(value = YEAR_PATH_VARIABLE_PATTERN, method = RequestMethod.GET)
    public String showCarDetail(@PathVariable("year") final String carProductionYear, final Model model,
            final HttpServletRequest request, final HttpServletResponse response)
    {
        final CarData carData = carFacade.getCarOfTheYear( carProductionYear );
        //...
    }
}
```

Controller classes are generally NOT Spring beans, so...

Façade is typically "wired" into controller using an annotation

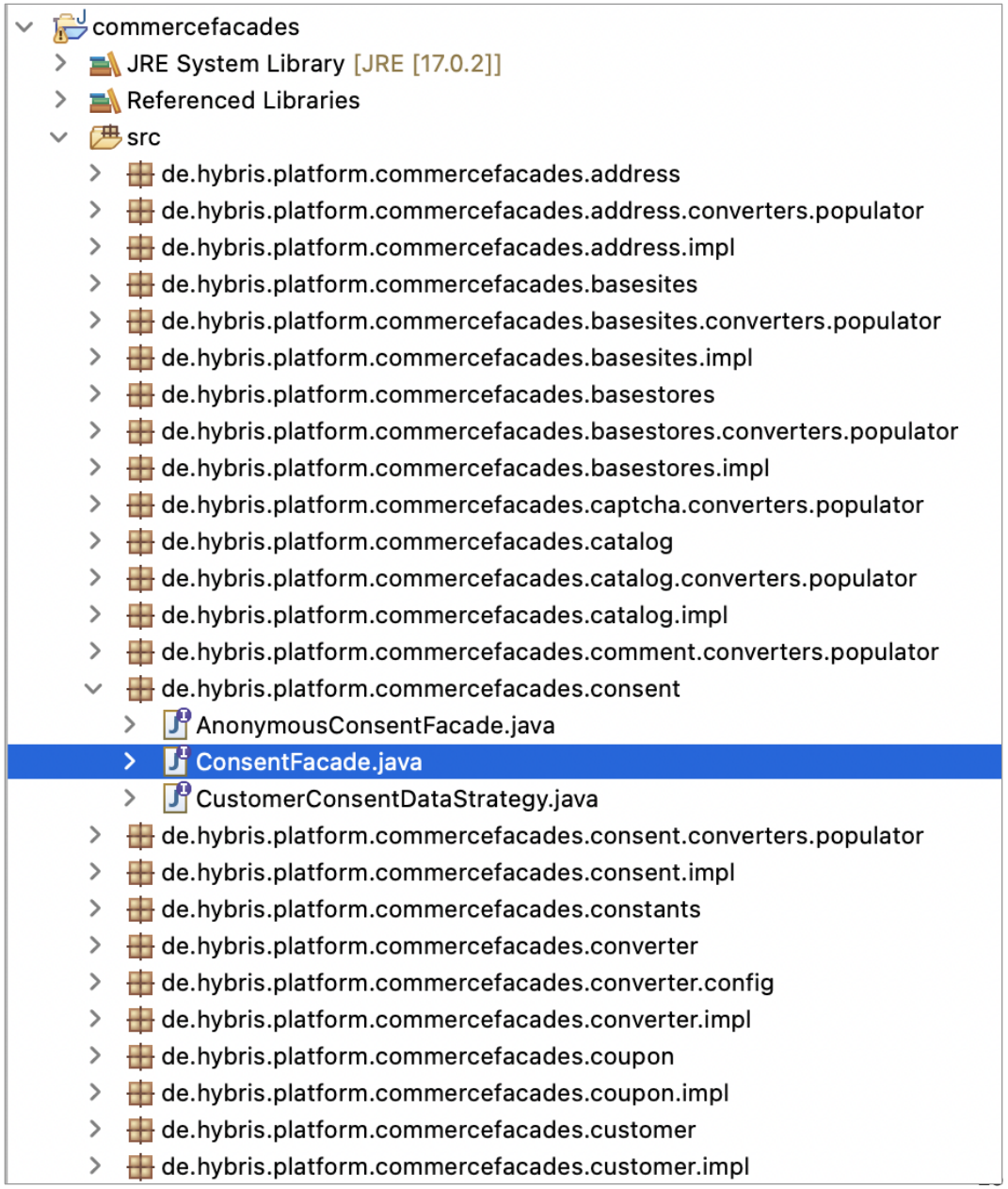
Façade is used here

Commerce Façades

commercefacades extension

Typical suite of Storefront Actions that make up a unified multichannel storefront API:

- Search for products with a free text search
- View product details
- Add a product to a cart
- Manage customers and related data
- Access consents and consent templates
- Access related promotions and coupons
- Add a delivery address during checkout
- Obtain cart and order data
- And more…



References

- Bean Generation
  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/8bc53579866910149472ccbef0222ec5.html

- Converters and Populators
  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/e1391e5265574bfbb56ca4c0573ba1dc/8b937ff886691014815fcd31ff1de47a.html

- Façades and DTOs – Best practices
  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/8c7acd1986691014a5f4b5880d032474.html

- commercefacades extensions
  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/e1391e5265574bfbb56ca4c0573ba1dc/8b832e6286691014a050d0863227d73b.html

Key Points

1. The Façade layer is responsible for **converting models** to **data transfer objects**
2. Concrete conversion is implemented by a **converter** and its associated **populators**
3. Different ways exist to populate attribute values of Models to DTOs:
   - Create a new converter
   - Reuse an existing converter and add new populators using the modifyPopulatorList bean
   - Extend a parent converter and merge new populators with inherited populators
4. The commercefacades extensions contain major functionality to support B2C features. They also provide a good example to demonstrate the relationship among converters, populators, façades, services, models, and data transfer objects.

Facades Exercise

Thank you.