DCI Design Idea[1]: The Specialised Role

Sometimes a DCI Role is defined, which may need to be specialised. Every specific implementation has the same interface (the same role methods), so they can be considered the same role, just with different implementation details. The choice of specific role is made by the context during the assembly of data and assignment of roles. It is usually based upon attributes of the data. In such cases, the specialised role can be useful. In this design idea, the context deals with an abstract role, but knows how to specialise it, based on the data which needs to play the role.

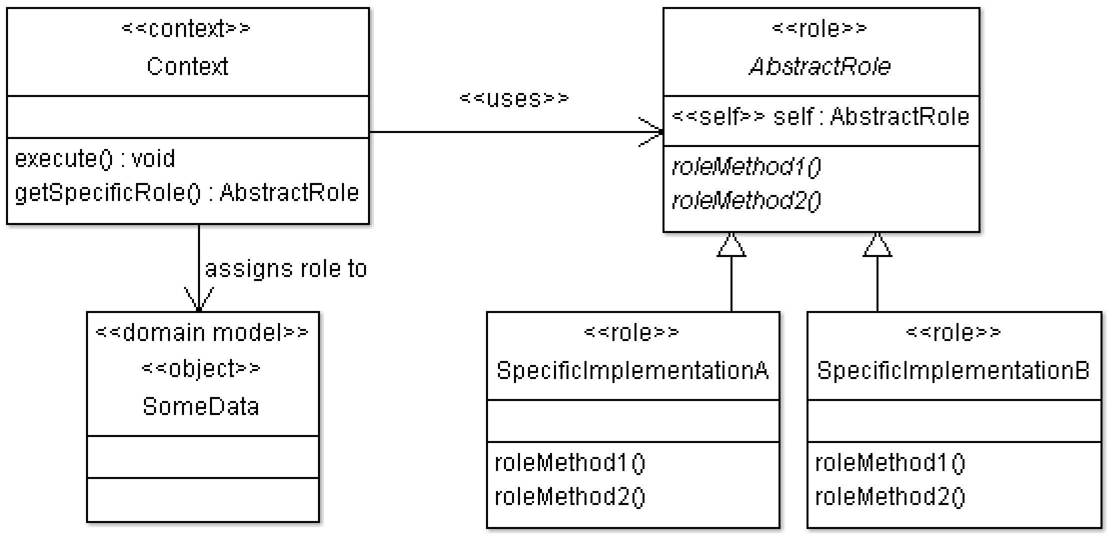


Figure 1: The Specialised Role

The `AbstractRole` does not need to be abstract; it can indeed also be a default implementation, which can be overridden as and when required.

This design idea could be implemented using individual Roles for each kind of specialisation, but doing so would mean that there is no place to put common code. This is an example where traditional object oriented inheritance is good. In cases where this design idea can be used, the role interface is identical between the specific roles; it is just the implementation which differs. Consider a waiter who is capable of serving drinks in the role of a drinks waiter. At times he will be in the role of the wine waiter (a type of drinks waiter), at other times, in the role of a soft drink waiter (also a type of drinks waiter). Different glasses and protocols are used for pouring different drinks and this specialisation is contained within the `serveDrinks()` method of the abstract role.

Example

An example[2] might be in an ordering system where an order takes on a role during order fulfilment, so that it can apply the relevant VAT / Sales Tax. In many countries, the amount of VAT to apply to each order item depends upon the item type. For example, luxury items may have a higher tax rate, or special items such as books, may be subject to less tax.

[1] To avoid the confusion between "Design Patterns" as defined by the Gang of Four and "Generative Patterns", the term "Design Idea" will be used.

[2] The idea for the example used here came from Petter Mahlen: http://pettermahlen.com/2010/10/02/dci-better-with-di/. See also http://groups.google.com/group/object-composition/browse_thread/thread/90e63ab085a602e6/f371b755a98ecb2a?lnk=gst&q=pattern#f371b755a98ecb2a

The role in this case would have the name "Taxable", that is, have the ability to add VAT to itself. To be able to do this, it would be able to determine the types of items it contains, and then add the relevant VAT to itself for each item type, with a user friendly description of its type.

This role can be represented abstractly, and specialised for each specific VAT rule needed. The UML would then look like this:

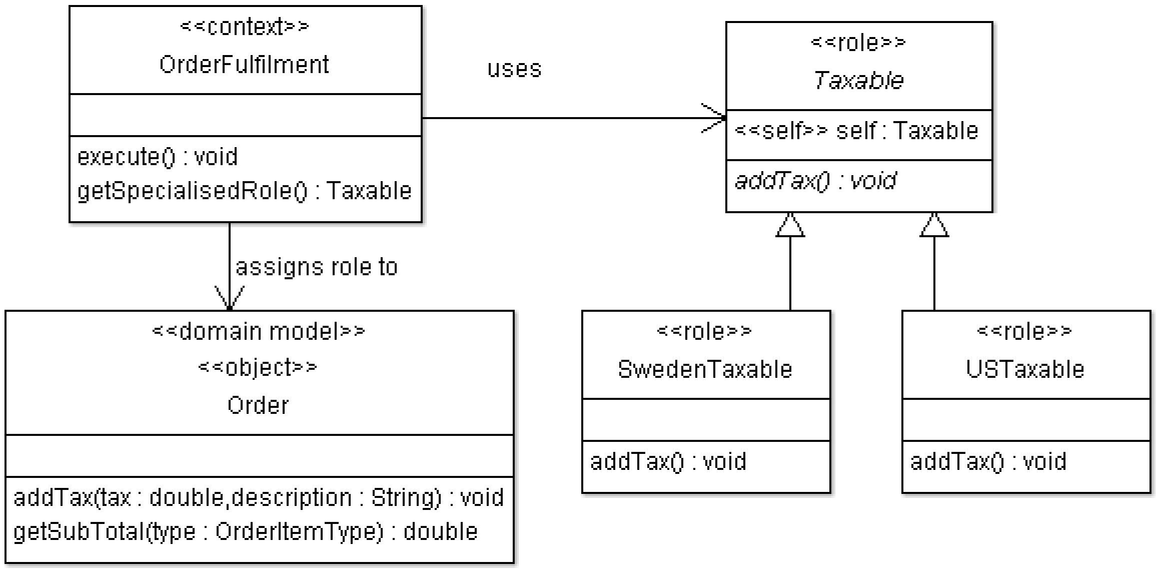


Figure 2: Example of The Specialised Role

In such a context, the data object being used to play the Taxable role, would be an Order object. This order object needs to be able to provide the `addVat()` role-method (in the `Taxable` Role) details about how much of the order is related to each VAT type it cares about. It does this via the `self.getSubTotal(OrderItemType)` method. The specific `Taxable` implementation applies the relevant tax for each item type, and adds it back to the order, using the `self.addTax(double, String)` method from the domain model, so that the order knows how to break down the VAT that has been applied, for example to print it on a receipt.

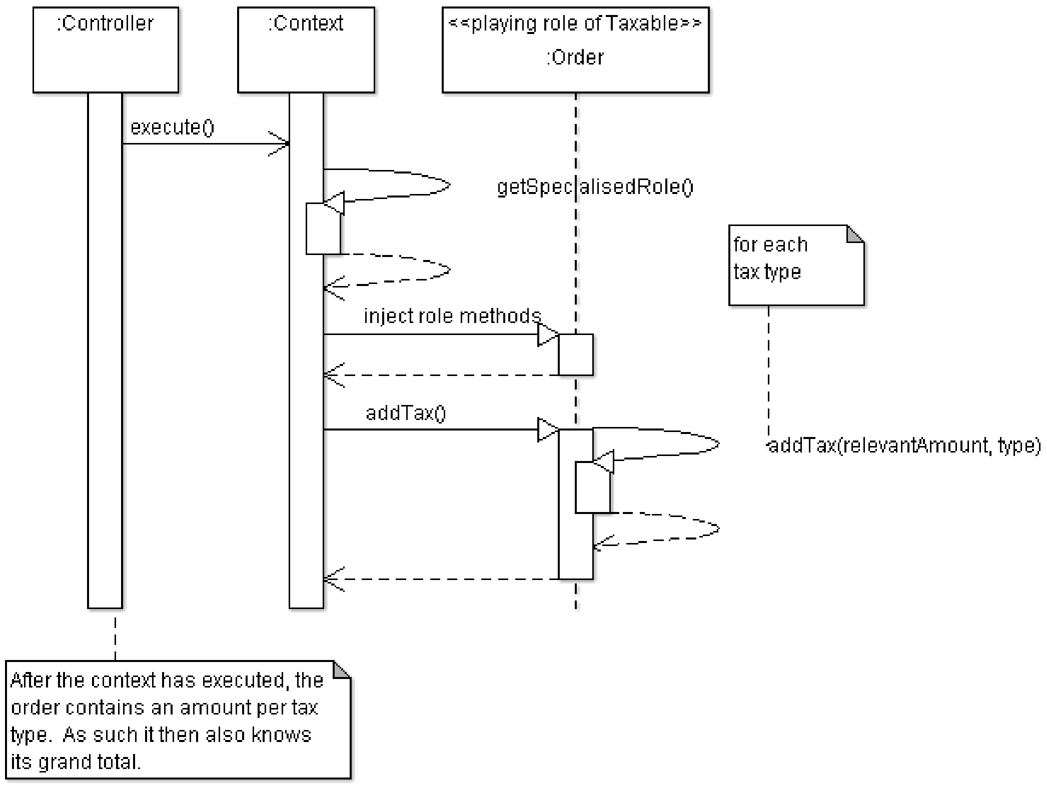

:Controller
:Context
<<playing role of Taxable>>
:Order
execute()
getSpecialisedRole()
for each
tax type
inject role methods
addTax()
addTax(relevantAmount, type)
After the context has executed, the order contains an amount per tax type. As such it then also knows its grand total.

Figure 3: Sequence Diagram for The Specialised Role

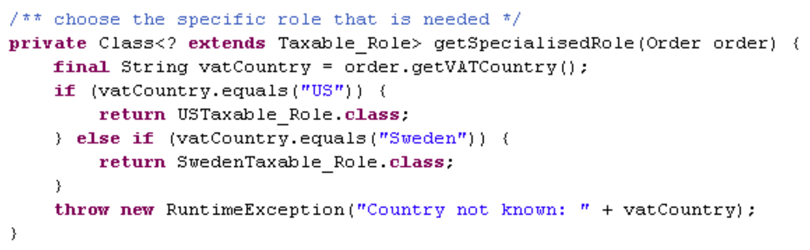
Java Implementation

See http://www.maxant.co.uk/tools.jsp for details about the `BehaviourInjector` class used in the example below.

A Java implementation of this example could look as follows.

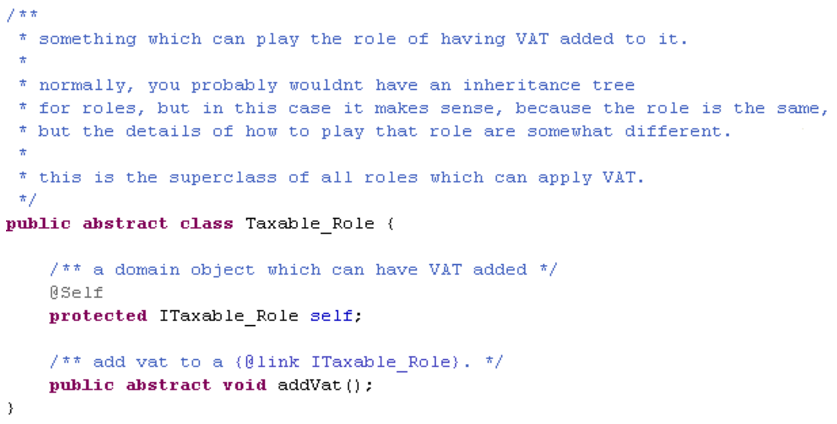
```
public void execute() {
    // prepare the injector
    BehaviourInjector behaviourInjector = new BehaviourInjector();

    // convert the domain object into a role, and inject the relevant
    // role methods into it
    ITaxable_Role vatable = behaviourInjector.inject(order, // domain object
            getSpecialisedRole(), // class providing all the impl
            ITaxable_Role.class); // the entire role impl

    // do the first part of fulfillment, by adding some VAT
    vatable.addVat();

    //TODO do other stuff to fulfill the order, perhaps using other roles...
}
```

Listing 1: Java example of the Order Fulfilment Context



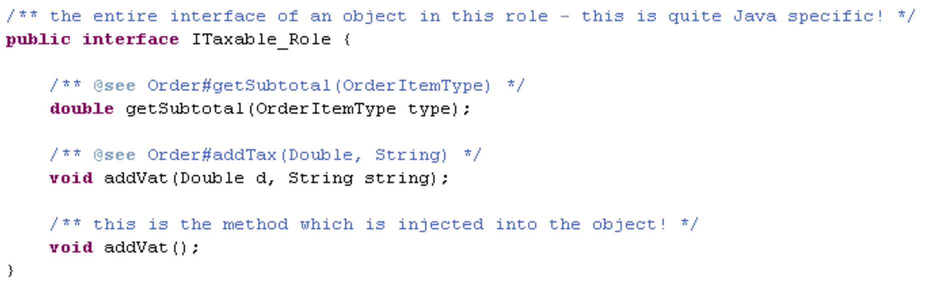
```java
/** choose the specific role that is needed */
private Class<? extends Taxable_Role> getSpecialisedRole(Order order) {
    final String vatCountry = order.getVATCountry();
    if (vatCountry.equals("US")) {
        return USTaxable_Role.class;
    } else if (vatCountry.equals("Sweden")) {
        return SwedenTaxable_Role.class;
    }
    throw new RuntimeException("Country not known: " + vatCountry);
}
```

Listing 2: Java example of the Order Fulfilment Context, showing selection of the specific role, based on information from the data object.



```java
/**
 * something which can play the role of having VAT added to it.
 *
 * normally, you probably wouldnt have an inheritance tree
 * for roles, but in this case it makes sense, because the role is the same,
 * but the details of how to play that role are somewhat different.
 *
 * this is the superclass of all roles which can apply VAT.
 */
public abstract class Taxable_Role {

    /** a domain object which can have VAT added */
    @Self
    protected ITaxable_Role self;

    /** add vat to a {@link ITaxable_Role}. */
    public abstract void addVat();
}
```

Listing 3: The abstract role.



```java
/** the entire interface of an object in this role - this is quite Java specific! */
public interface ITaxable_Role {

    /** @see Order#getSubtotal(OrderItemType) */
    double getSubtotal(OrderItemType type);

    /** @see Order#addTax(Double, String) */
    void addVat(Double d, String string);

    /** this is the method which is injected into the object! */
    void addVat();
}
```

Listing 4: The (Java specific) interface of the object playing this role[3].

[3] See the `BehaviourInjector` documentation for more information about this Java specific interface.

```java
/** VAT role played if its in Sweden */
public class SwedenTaxable_Role extends Taxable_Role {

    private static final double BOOK_RATE = 0.024;
    private static final double OTHER_RATE = 0.076;

    @Override
    public void addVat() {

        //in sweden, they have two different rates of VAT depending upon the item
        double bookAmount = self.getSubtotal(OrderItemType.BOOK);
        double otherAmount = self.getSubtotal(OrderItemType.NORMAL);
        self.addVat(new Double(BOOK_RATE * bookAmount), "VAT Books");
        self.addVat(new Double(OTHER_RATE * otherAmount), "VAT Other");
    }
}
```

Listing 5: The specific role for Swedish tax.

```java
/** VAT role played if its in the US */
public class USTaxable_Role extends Taxable_Role {

    private static final double RATE = 0.12;

    @Override
    public void addVat() {
        double subTotal = self.getSubtotal(OrderItemType.BOOK);
        subTotal += self.getSubtotal(OrderItemType.NORMAL);

        //in the US, there is one flat rate of VAT for everything.
        self.addVat(new Double(RATE * subTotal), "Sales Tax");
    }
}
```

Listing 6: The specific role for US tax.