Decision Management Community Challenge Jan 2018- Order Promotions A solution using Drools

(Bob Moore, JETset Business Consulting, 25 Jan 2018)

1 Problem Statement (cut and pasted from the web site)

The objective of this challenge is to help merchants to define various promotions for their sales orders and to automatically decide if an order is eligible to a promotion. An order usually consists of order items with known price and quantities. A promotion defines minimal quantities of certain items in the eligible orders.

A simple example of promotion: *reduce the total cost of the order by $3.50 if it contains at least 5 items 1108 and at least 4 items 2639*.

2 Interpreting the Challenge

Almost all specifications leave some area open to interpretation, but the challenge here leaves more up for grabs than is often the case. Of all the challenges I've looked at so far, this proved the most interesting and challenging because filling in the gaps leads to some interesting outcomes. In this case it threw up some issues in building a decision table-based solution. And having overcome these, some more issues emerged which prompted me to kick out decision tables altogether and to exploit the synergy of objects with rules to give a solution which it seems to me puts a more real-world flavour to how one might manage promotions.

The vagueness of "An order *usually* consists …", I think is fair to ignore. It seems safe to assume an order *always* consists of order items with known price and quantities. What is much open ended is that we only have an example of a "*simple*" promotion. What are we to assume a typical or a complex promotion looks like? To start, let's make some additional assumptions to clarify what we are aiming at. First, we will assume that a promotion is always defined as:

- A set of item types (each identified by an SKU)
- For each SKU a qualifying quantity for that SKU
- A cash value for the discount

Next we will assume an order will comprise a number of 'order lines' each order line specifies the type of item (an SKU) and the quantity being ordered. An order qualifies for a promotion if for each SKU in the promotion, the order contains an order line for that SKU, and the number of SKU on the order line is at least the qualifying quantity for the SKU in the promotion.

Before we go further, let's jot down some of the open questions which come to mind:

- From the simple example we have, suppose the customer orders 10 items 1108, and 8 items 2639. Does she get a $7.00 discount (she'd get two $3.50 discounts if she made two orders)?[1].
- Can I have two separate promotions involving the same item? For example. $4.00 discount for 6 items 1108, and 2 items 1121, in addition to the promotion in the challenge. If so, and the order qualifies for both discounts, does the customer get one, or both? And if only one, which one?
- Following on from this, if an order qualifies for multiple promotions, are all applied or only one (or something else)?
- Can promotions refer to a category of product rather than a specific SKU? For example. the order might be for several ballpoint pens, some red and some green (so they will have different SKUs). Can I define a promotion which involves the total number of pens regardless of the specific SKU of each type?
- Can we get additional items instead of a discount (e.g. a free fountain pen with every order of 50 red ballpoints)?
- Can we get percentage discounts as well as discounts of a fixed amount (and is the discount over the whole order or just the promoted items)?

It would take a fair bit of time to handle all these ideas, so I'm only going to look at the first three. But even that is a bit complicated to start with, so let's start off nice and simple. We'll assume that:

1. An order only contains order lines for distinct SKUs (so it won't have an order line for 2 1108 items and then another order line one for 3 1108 items, I'll just have one order line for 5 1108 items)
2. A promotion is based on no more than 3 different types of SKU
3. Any promotion is applied only once
4. A customer receives discounts for *all* the promotions the order qualifies for

The second assumption is a bit arbitrary, but thinking about it, customers are likely to get annoyed if the promotions get too complex. We could define promotions with 5 or even 50 types of item involved, but specifying a customer needs even as many 4 different kinds of items to get a discount seems a bit of overkill. Once we've looked at this basic solution we'll look at relaxing all but the first assumption which is essentially about data consistency.

3 A Basic Solution

Let's speculate what a solution might look like. Something like this is what sprang to my mind when I was initially thinking about it. First, I need something which tells me what my promotions are, which I visualised as looking something like this:

NAME	Product 1	Quantity 1	Product 2	Quantity 2	Product 3	Quantity 3	Discount
Promo 101	1108	5	2639	4			3.50
Promo 102	1001	3	2001	4	3001	6	4.50
Promo 103	2002	4					5.50

[1] Mike Parrish's solution explores this idea in some detail – see https://dmcommunity.files.wordpress.com/2018/01/corticon-order-promotions.pdf

Secondly the ‘outcome’ of applying the promotion(s) would look something like this:

```
Congratulations your order Order 1   qualifies for discounts
Total discounts   64.00
Qualifying promotions:
        Promotion Prom 3    giving discount of   4.00
        Promotion Prom 1c   giving discount of  30.00
        Promotion Prom 4c   giving discount of  18.00
        Promotion Prom 1b   giving discount of  12.00
```

Let’s start by looking in detail at how the promotions are represented. What I drew out above is basically an Excel spreadsheet, with each promotion shown as a row, with the name, the SKUs and quantities (up to three as assumed above) and the applicable discount in each column. The first promotion “Promo 101” is the given example. The second promotion “Promo 102” is more complex requiring an order has three specific products. Finally, the promotion “Promo 103” only requires that the order includes an order line for at least 4 2002 items. At this stage the examples don’t involve any overlap

Ideally an Excel spread sheet in this tabular form should be translatable to a decision table each row defining a rule which determines if an order qualifies for a discount. If we can do this then we can apply the decision table to the order and then we’ll find the discounts.

However, a little bit of work is needed. The first problem is that in a decision table the condition columns are ‘independent’ of one another. The truth value associated with a cell in one column of a row does not depend on the value in another cell in the same row. However, in the spreadsheet above the ‘truth values’ have a dependency spreading across adjacent cells (so the cells containing ‘1108’ and ‘5’ combine to give the single condition “*there is an order line for 5 or more 1108 items on the order*”)[2].

If we merge the Product and Quantity columns we have something we can work with:

NAME	Product 1 / Quantity 1	Product 2 / Quantity 2	Product 3 / Quantity 3	Discount
Promo 101	1108, 5	2639, 4		3.50
Promo 102	1001, 3	2001, 4	3001, 6	4.50
Promo 103	2002, 4			5.50

If we consider this a DMN decision table, the ‘hit’ policy is going to be ‘Collect’ as we want to know all the promotions which can apply (assumption 3 above).

However, the cells representing the conditions are not representing conditions as seen in ‘simple’ DMN decision tables, they are not a test on a primitive value like a string, or

[2] Mike’s solution gets around this by separately having attributes in his decision table for an SKU being on the order and the number of items of that SKU – so there is a ‘coffee.sku’ test and a ‘coffee.qty’ test – but this does double the number of attributes involved. Jacob (Feldman) has presented a solution (https://openrules.wordpress.com/2018/01/03/decision-model-for-sales-order-promotions/) that nimbly skips around this problem by separately enumerating the SKUs and qualifying quantities, but (as least without a bit more work) this prevents us defining multiple promotions involving the same SKU, (unless the qualifying quantity for that SKU is always the same).

a number, but on composite entity - a quantity of a certain type of item. Likewise, the different columns do not represent tests on distinct ‘simple’ attributes of the input item (which in this case is the order), but a test of the item types (and their quantities) on the set of order lines on the order.

Extant examples of DMN decision table don’t actually seem to address conditions like this, although my reading of the DMN spec is that falls within the scope of what is permitted, and we could build such a table using a suitable function in FEEL However lacking access to a level 3 compliant DMN tool, I leap straight to implementing the table in Drools.

Drools, in common with some other BRMS, uses a ‘template’ approach to decision tables, which means that the condition satisfied by a cell in a decision table depends on the ‘template’ used to define it. You can define your own templates and the contents of a condition cell just provide the parameters to the template. The upshot is that it is quite easy to build an Excel spreadsheet which looks identical to the one above (having hidden a few rows and columns), and have it directly consumed by the Drools compiler.

Specifically, if we reveal the hidden rows in the excel above we see:

RuleSet	jetsetbc.promotion			
Import	jetsetbc.promotion.datamodel.*			
RULEFLOW-GROUP	DetermineApplicablePromotionsDT			
RuleTable Discount rates				
NAME	**CONDITION**	**CONDITION**	**CONDITION**	**ACTION**
	$order:Order			
	getQuantityOfItem("$1 ") >=$2	getQuantityOfItem("$1 ") >=$2	getQuantityOfItem("$ 1") >=$2	modify($order) { addPromotiontoApply(new Promotion(drools.getRule ().getName(), $1))};
NAME	Product 1 / Quantity 1	Product 2 / Quantity 2	Product 3 / Quantity 3	Discount
Promo 101	1108, 5	2639, 4		3.5
Promo 102	1001, 3	2001, 4	3001, 6	4.5
Promo 103	2002, 4			5.5

We’ll explore the object model in more detail later as it is reused in the more complex model described below, but the template for the condition cells:

```
getQuantityOfItem("$1")  >= $2
```

basically says, use the first value ($1) in the cell (e.g. 1108), to look up the number of items of this SKU on the order, and test if it is at least as big as the second value ($2) in the cell (e.g. 5) which is the qualifying number of that SKU for the promotion to apply.

Before we get onto the output, let’s delve a little more into the problem. Using the decision table above, if we take the following order:

Order 1		
Line	SKU	Quantity
1	1001	4
2	1108	8
3	2001	4
4	2002	4
5	2639	4
6	3001	4

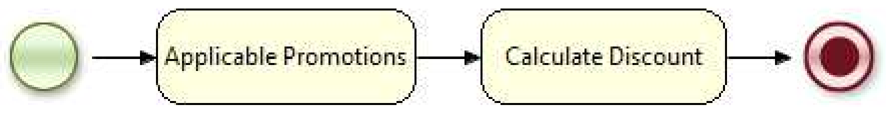
Lines 2 and 5 satisfy the conditions for promotion 101, line 4 satisfies the conditions for promotion 3, but promotion 102 requires the ordering of more of item 3001 for it to kick in. So, we should get a discount of \$3.50 for “promo 101” and an additional discount of \$5.50 for “promo 103”. Adding these up the customer gets a \$9.00 discount.

Are we done? Not quite yet! Putting aside the various ideas about how promotions might interact, we do at least need to be able to explain were this \$9.00 came from. So, part of the output of the decision table not only needs to be the discount, but also the name or identifier of the promotion the discount is associated with. So, in the same manner that the condition cells are of a ‘complex’ rather than a ‘simple’ values, the output of the decision tables should be a collection of ‘complex’ rather than ‘simple’ values. We can ‘cheat’ a little, since the value in the ‘NAME’ column becomes the ‘name’ of the Drools rule. We can pick up this up using the Drools run-time, so we only need the simple value of the discount in the output column. Thus we get the template for the discount is:

```
modify($order) { addPromotiontoApply( new Promotion(drools.getRule().getName(), $1 ))};
```

Which says add a new promotion to the order initialising it with the rule name (in the left most column – using the Drools API) and the discount (\$1) from right most column.

This need for multiple outputs also prevents us from using something corresponding to a ‘sum’ hit policy in DMN. As a consequence, the discount calculation needs to be something separate and distinct from the decision table and we see there are at least two separate decision steps in the decision process. The first determines which promotions apply to the order, the second determines the effect of the promotions on the overall order price. Of course under our initial assumptions for the ‘basic’ solution, all we need to do is list them out and add them up as per my envisaged output. In building a solution in Drools we can link the steps using a (very) simple rule-flow:



The first step invokes our decision table, the second outputs the results (the output logic is described in more detail in section 6.7)

If we look back the ‘open questions’ in section 2, it looks like we can address many of these by inserting an extra step into this decision process. We have worked out which promotions the order satisfies the prerequisites for. To address the open questions, what we now need is some logic about how these qualifying promotions interact, and if all or only some should be applied, before moving onto the final step of adding up the discounts from the set of promotions we have settled on.

4 Decision Tables or Objects?

I must admit I'm not the greatest fan of decision tables. When I started building decision systems back in 1989[3], the tools I was using didn't support them, but they did support pattern matching across multiple objects – objects which could be readily plucked from relational databases if so desired. When decision tables were first introduced into Blaze Advisor (the tool I was exclusively using at the time) around 2000, I did struggle to see what they could do, which I couldn't do just as easily by pattern matching over a bunch of objects as I had been doing for at least a decade with multiple tools.

The most plausible answer I've come up with is that with decision tables, business users can visualise the rules better, because BRMS systems give you nice editors for decision tables, but not for collections of object instances[4] [5].

In terms of what you *can't* do with a decision table, but you *can* do with a bunch of objects and some pattern matching rules, the list is much longer. For this particular problem we run into one of these immediately as soon as we try to figure out how to implement our 'middle' third of the logic, determining the interaction of promotions with one another. Let us say for example I add another promotion to my decision table:

NAME	Product 1 / Quantity 1	Product 2 / Quantity 2	Product 3 / Quantity 3	Discount
Promo 101	1108, 5	2639, 4		3.50
Promo 102	1001, 3	2001, 4	3001, 6	4.50
Promo 103	2002, 4			5.50
Promo 104	1108, 3	2001,4		15.00

This promotion also involves items of SKU 1108 and 2001, so the order satisfies the qualification criteria for promotion 'Promo 104'. If we take our simplistic approach of applying every discount the satisfies the qualification criteria we apply this one also. This may be what is wanted, but we are 'double counting' the items of SKU 1108 and 2001 which somehow doesn't seem correct. If we don't want to double count, we have several alternatives. Two options which come to mind are:

- If two (or more) qualifying discounts 'overlap' we apply just one (typically the one with the biggest discount) and do not apply the others
- If two (or more) qualifying discounts 'overlap' we select the one giving the biggest discount, and reduce the quantities of qualifying SKUs by the qualifying quantities and see if the order would still qualify for the next best discount (so in our example order 3 of the 8 items of SKU 1108 with the 4 items of SKU 2001 give promotion 'Promo 104', and the remaining 5 items of type 1108 SKU can combine with 4 of SKU 2639 to still allow the order to qualify for 'Promo 101'

[3] The first system I worked on was calculating promotional discounts. Improbable but true!

[4] There are performance considerations. If you have a decision table with static values and simple tests you can (at least in principle) do a lot of optimisation of the generated code which will give you significant performance improvements over working with a set of objects where the rule compiler can't be sure if the attribute settings are going to change over time. While this could be an issue with large tables, with smaller tables flexibility and maintainability should be the more important consideration

[5] Of course, if you are plucking your objects from a database, it's easy to build a nice front end to your database. The argument against this is you now are using two tools a DBMS and a BRMS (my counter argument is that in any real system, a DBMS somewhere will eventually be playing a key role – so use it)

The first scheme requires that ‘Promo 101’ and ‘Promo 104’ ‘know’ they overlap. The second requires not only that they ‘know’ they overlap, but also the qualifying quantities of the overlapping item types. However, if we use a decision table approach, this information is embedded in the cells of the decision table. The only way we can reason about this information elsewhere in the decision service is if we duplicate it. We need to hold facts like ‘Promo 101’ involves item types 1108 and 2639 both in the decision table and somewhere else (another decision table?). A promise of a maintenance nightmare!

On the other hand, if we use an object-based approach we define a promotion once as an object, and then can access its characteristics at any point in the decision-making process we like. Indeed, once we make the promotions ‘first-class’ rather than being inextricably embedded in a decision table it means we can be much more flexible with the way they work. We may still use the basic ‘qualification’ process but adding ideas like free additional items, BOGOF offers, percentage discounts, promotions which always apply regardless of overlapping promotions and so on is greatly simplified

Another point – which I only realised as I read Mike Parrish’s solution is that making the promotions objects rather than rules makes them much more accessible. Mike makes the point that a customer wants to know if a promotion has *not* been applied, why it hasn’t. This seems a plausible question to ask – but it only makes sense if the customer knows that the promotion exists in the first place.

To do this the customer needs to know what the promotions are up front – and if they do, they can tailor their order to maximise their discounts (and if the promotion scheme is any good, this should ultimately end up generating more revenue for the merchant). It’s not easy to do that if the promotions are embedded in a decision table (we’d need to give the customer some kind of read-only view of our decision table). On the other hand, if the promotions are objects stored outside the decision service (say in a database), it would be easy to display these to customers on the merchant’s web site while they are pondering what to order and encourage them to order more to get the discounts (ultimately the objective of discounts is to get the customer to spend more).

5 The Expanded Challenge

Before getting to work on a solution which addresses these ideas, it’s probably sensible to decide what exactly we are trying to solve. There are a couple of questions here:

- When does an order qualify for a promotion?
- If an order qualifies for multiple promotions which of these should be applied?

The first question we have already addressed, assuming an order qualifies if it has the right set of SKUs and the right quantities of each. But what about the second? There are lots of different options, but for our purposes we will confine ourselves to a model which expands on the ‘simple promotion’ but captures some ideas about how promotions might interact with one another.

First of all, we will say two promotions ‘overlap’ if they share at least one SKU in common. It is assumed that any qualifying promotion which does not overlap with any of the other qualifying promotions will be applied. On the other hand, if two qualifying promotions do overlap, we need to specify how they interact. We suppose there are three different types of promotion with the following characteristics:

- Promotions which are always applied if the order qualifies, regardless of any overlapping promotions – these we denote as 'A' (for Always) promotions.
- Promotions which if they are applied exclude the possibility of other qualifying but overlapping promotions being applied – these we denote as 'X' (for eXclusive) promotions.
- Promotions which if they are applied 'allocate' the qualifying quantity of each SKU to the promotion so no individual item on the order is 'double' counted – these we denote as 'Q' (for Quantity) promotions.

'Always' promotions are pretty self-explanatory. Although the overlaps may be more complicated, it's easiest to understand 'Q and 'X' promotions in terms of quantity discounts. The first open question posed was if I have a promotional discount of $3.50 if my order contains at least 5 items 1108 and at least 4 item 2639, do I still only get $3.50 if my order contains 10 items 1108 and 8 item 2639, or do I get 2 x $3.50 = $7.00 discount, or something else?

The idea of 'Q' and 'X' promotions are that I can define a set of overlapping promotions like this:

NAME	Product 1 / Quantity 1	Product 2 / Quantity 2	Product 3 / Quantity 3	Discount
Promo 101a	1108, 5	2639, 4		3.50
Promo 101b	1108, 10	2639, 7		8.00
Promo 101c	1108, 20	2639, 14		18.00
Promo 101d	1108, 40	2639, 25		40.00

The promotions give the customer progressively better discounts with increasing orders of SKUs 1108 and 2639 (we could keep the ratios of one to the other equal, but in this example the relative number of 2639 reduces). An order of 30 items of 1108 and 20 items of 2639 thus qualifies for Promo 101a, Promo 101b and Promo 101c.

If the promotions are all 'X', then the promotion Promo 101c is the one which is applied, and the other promotions are excluded – we get a discount of $18.00. This approach has the effect of putting a ceiling on the overall discounts provided – for example if the customer orders 400 items of 1108 and 250 items of 2639 they still only get a $40.00 discount[6].

If the promotions are all 'Q', then Promo 101c is applied and 20 items of SKU 1108 and 14 of SKU 2639 are allocated to it. These leaves 10 items of SKU 1108 and 6 of SKU 2639. These satisfy the qualification conditions of Promo 101a, so this also can be applied. Once the allocation of items to this promotion is make we are only left with 5 items of SKU 1108 and 2 of SKU 2639, and these do not qualify for any promotions. In this case, we get a discount of $3.50 + $18.00 = $21.50[7]

[6] Although the representation is quite different, the extended promotions Mike explores in his solution are in effect 'X' ones

[7] While 'Q' promotions may look as if they are always better for the customer than 'X' ones, the actual benefit will ultimately depend on the discount structure and typical order quantities.

In some scenarios, with ‘Q’ promotions, it may be possible to apply the same promotion multiple times – on an order of 400 items of 1108 and 250 items of 2639, Promo 101d should be applied 10 times giving a $400.00 discount (which feels good to the customer, but with a big order like this it’s going to be good for the retailer too).

6 An Object Based Solution

So, let’s roll up the sleeves and build an object-based approach which tries to implement promotions with ‘A’, ‘Q’ and ‘X’ type interaction as described above.

6.1 The Object Model

We start with an object model. We obviously need at least two types of object, one describing orders and one describing promotions. Since we will be using Drools to build the solution, the object types will be Java classes. It turns out that a third object type an ***ItemQuantity*** is also good to have which expresses we have a certain number of a certain type of object (e.g. 5 items of SKU 1108).

An order is going to comprise a lot of things like date, customer, status (requested, accepted, picked …) total price and so on. However, for our purposes the only things of interest are the order lines (the input) and the applied discounts and the total discount (the outputs).

At first sight a promotion just needs to include some kind of identifier and/or description, a list of required SKUs and their qualifying quantities and the associated discount, though it turns out we need a little more as we will see.

Since both orders and promotions rely on a list of SKUs and quantities in the spirit of the object approach, we might as well group these ideas in a superclass – ***ListOfItemQuantities*** – which supports the idea of a ‘named’ list of items (SKUs) with associated quantities. Then both orders and promotions can inherit from it. This should have methods to do things like:

- Get the name of the list – the order id or the promotion name
- Get a list of the SKUs on the list
- For a given SKU get the quantity of the SKU on the list (which may be zero of course)
- Set the quantity of an SKU on the list
- Get a list of the SKU and their associated quantities (as ***ItemQuantity*** instances)

As discussed above, we are going to have to work out which promotions qualify to be applied to an order, so we need somewhere to record this, and then separately we need to record which promotions have been selected to be applied. So, we give the ***Order*** class a couple of extra list attributes, ***applicablePromotions*** which contains the promotions which *could* be applied, and ***promotionsToApply*** which contains the promotions which *will* be applied.

The ***Promotion*** class also needs some additional attributes – there is the size of the discount and the way the promotion interacts with overlapping promotions (‘A’, ‘Q’ or ‘X’) evidently. Additionally, we need to know the promotion is still (potentially) available or if it has already been used or excluded (we can use ‘Q’ promotions multiple times, but ‘A’ and ‘X’ promotions at most once). So, we give promotions a ***status*** – either

‘OPEN’ or ‘CLOSED’. Finally, since a “Q” promotion might be applied more than once we should be able to show this in the output. There is a choice, we can output the usage of the promotion several times:

```
Congratulations your order Order 2   qualifies for discounts
Total discounts    8.00
Qualifying promotions:
        Promotion Prom 1a   giving discount of   4.00
        Promotion Prom 1a   giving discount of   4.00
```

Or we can aggregate the usage of the same promotion giving something like:

```
Congratulations your order Order 2   qualifies for discounts
Total discounts    8.00
Qualifying promotions:
        Promotion Prom 1a   giving discount of   4.00 (used  2 times)
```

The former might get confusing as the same promotion could appear several times, so the latter approach has been adopted by giving promotions a ***useCount*** attribute.

The full code of the object model is provided in the appendix for those interested.

6.2 Definition of the Promotions and Orders

In the decision table approach, promotions are defined by the rows and columns of a decision table. In the object-based approach we have a collection of objects which do this. However, we still want to be able to maintain them. In a real system they would be stored (and maintained) in a database. For this solution they are stored in a simple tab delimited file, which can be easily read in by a few lines of Java code[8]. For test purposes we will use the following promotions:

Name	Discount	Interaction	SKU	Quantity	SKU	Quantity	SKU	Quantity
Prom 1a	4	Q	1001	4	1002	2		
Prom 1b	12	Q	1001	10	1002	5		
Prom 1c	30	Q	1001	20	1002	10		
Prom 2	10	Q	1002	4	1003	6		
Prom 3	4	A	1001	2	1002	2	1003	2
Prom 4a	3	X	1003	3	1004	6		
Prom 4b	8	X	1003	6	1004	10		
Prom 4c	18	X	1003	12	1004	15		
Prom 4d	24	X	1003	12	1004	20		

This gives us examples of ‘A’. ‘X’ and ‘Q’ discounts and various sorts of overlaps between the product SKU used in the discounts.

Likewise, orders would be stored separately in some kind of database. An even simpler text file format has been used for the test data in this case, with the order name followed by lines giving the SKUs and the ordered quantities:

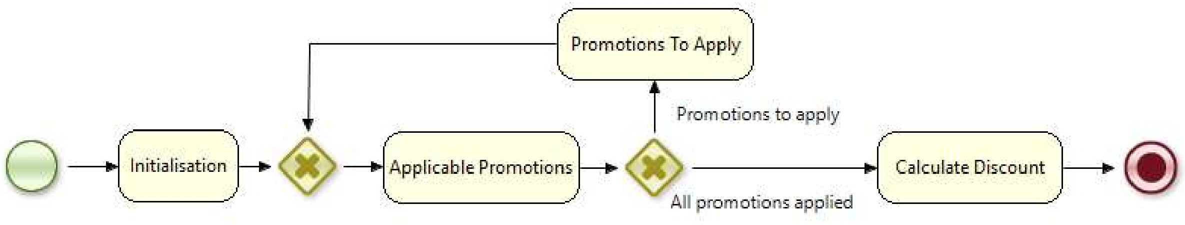
```
Order 1
```

[8] Note this format means we no longer have the restriction of only 3 products in the promotion we imposed for the decision table solution. However, it still makes very little sense from a usability perspective to allow a promotion to be based on more than about 3 items

```
1001    30
1002    20
…
Order 2
1001    9
…
```

6.3 The Rule-Flow

The decision-making process comprises several steps, so a rule-flow is used to orchestrate the steps. This is more complex than for the decision table solution:



In the decision table solution our decision table stores no ‘state’ but moving to an object-based solution the instances of the ***Promotion*** class change state during the decision-making process[9], so we need to have an initialisation step to reset them before processing each ***Order***. Having determined the qualifying promotions, the rule-flow has a branch, either one or more applicable promotions have been identified, in which case we determine which ones to apply (“Promotions to apply”), otherwise (“All promotions applied”) we’ve determined all the promotions to apply and move on the calculate the overall discount on the order. As we will see when we apply some promotions (specifically ‘Q’ type), this generally means we need work out which if any promotions still apply, so there is a loop in the logic[10].

6.4 Initialisation

We just need a simple rule to manage this, but it is a taster for things to come, so we can warm up gently:

```
rule "Initialise Logic"
   ruleflow-group "InitialiseLogic"
   when
      $promotion: Promotion( status != "OPEN" || useCount > 0 )
   then
      modify($promotion) { reset() }
end
```

[9] We could create the ***Promotion*** objects from scratch with each execution, but assuming they don’t change very often, it makes more sense to load them into working memory once and reuse them for each ***Order*** we need to process.

[10] With the defined types of interactions, we only need to loop because of ‘Q’ promotions, but this general structure should also support more complex types of interactions between the promotions.

We just reset any Promotion which has been used (so either its status is not “OPEN” or it’s use count is not 0)[11]

6.5 Determining the applicable promotions

A decision table approach involves several rules (rows) and essentially no objects, while an object-based approach involves several objects and (usually) a lot fewer rules. In terms of determining the applicable promotions, Drools expressive capabilities mean we only need one rule – though it is a bit of a mouthful:

```
1     rule "Determine Applicable Promotions"
2       ruleflow-group "DetermineApplicablePromotions"
3       when
4         $promotion: Promotion($itemsRequired: itemsInList, status == "OPEN")
5         $order: Order($itemsOnOrder: itemQuantities,
6             applicablePromotions not contains $promotion )
7         $onOrder : List() from collect (
8             ItemQuantity(description memberOf $itemsRequired ,
9                      $promotion.getItemQuantity( description ).quantity <=  quantity)
10                                   from $itemsOnOrder)
11        eval($itemsRequired.size() == $onOrder.size())
12      then
13        modify($order) { addApplicablePromotion($promotion) }
14    end
```

To explain:

Firstly (line 4), we look for ***Promotion*** instances which are (still) “OPEN” and bind the variable ***$itemsRequired*** to the list of SKUs that define the promotion (just the SKUs not the associated quantities).

Secondly, we look for an ***Order*** instance (of which we assume there is only one) and check we have not already added this promotion to it’s set of applicable promotions as well as binding the variable ***$itemsOnOrder*** to the list of ***ItemQuantity*** instances associated with the order (i.e. the SKUs *with* their associated quantities) (lines 5 & 6).

The third condition (stretching from line 7 to 10), is the let’s face it rather challenging. In effect, it says “gather together the ***ItemQuantity*** instances on the order (***$itemsOnOrder***), whose description (i.e. their SKU) matches the description (SKU) of one of the required items for the promotion to be valid (***$itemsRequired***), *provided that* the quantity of the item on the order is at least the required quality for the promotion to apply (fetched from the order by the ***getItemQuantity*** method).” The end result – ***$onOrder*** – is bound to a list of the SKUs on the order which match one of the SKUs defining the promotion and also satisfy the promotion’s quantity condition for that SKU.

The final condition (line 11) simply asks if the number of items in the list ***$onOrder*** is the same as the number of items in the list ***$itemsRequired***. If it is, then we have matched all the SKUs in the ***Promotion***, with an SKU on the ***Order***, where the required quantity on the order is at least the quantity required in the ***Promotion***, so the ***Order***

[11] Why not reset all the promotions instead of just the ones which have changed? Well each time you ‘modify’ a Promotion object, it potentially puts rules on the agenda and if you don’t put in qualifying conditions to prevent this rule being re-evaluated Drools goes into an infinite loop (as I found)

qualifies for the ***Promotion*** to be applied, otherwise either a required SKU is not on the order, or it is, but not enough has been ordered. Either way, if the list lengths are not equal it means the ***Order*** does not qualify for the ***Promotion***.

If all the conditions are satisfied, then we add the ***Promotion*** to the list of promotions applicable for the ***Order*** (line 13). The modify statement is required to ensure the rule engine is aware of the order has been updated, so changes can be propagated through inference network.

This single rule (combined with the ***Promotion*** objects) has virtually identical effects as the decision table we built in section 3. The only difference is that here the Promotions are added to the ***applicablePromotions*** list rather than the ***promotionsToApply*** list.

6.6 Deciding which applicable Promotions should be applied

It is at this point where we see the benefit of swapping from decision tables to objects and rules, because it turns out we need to know again about the SKUs and quantities involved in each promotion to do this.

It turns out we need we need four rules to decide which promotions to use.

The first simply sorts out the 'A' type promotions – which by definition we always apply if the order qualifies:

```
1    rule "Manage Promotions which are always applied"
2       ruleflow-group "PromotionsToApply"
3       when
4          $promotion : Promotion(interaction == "A")
5          $order : Order($promotion memberOf applicablePromotions)
6       then
7             modify($promotion) {setStatus("CLOSED") }
8             modify($order) {
9                    removeApplicablePromotion($promotion),
10                   addPromotiontoApply($promotion)
11            }
12   end
```

If we have a ***Promotion*** of type 'A' (line 4), and it is one of the applicable promotions of the ***Order*** (line 5), then we set the status of the Promotion to "CLOSED" (line 7) (to ensure it is not used again), remove it from the list of applicable promotions (line 9) and add it to the list of promotions which will actually be applied (line 10).

For 'Q' and 'X' promotions we assume a strategy that given a choice of promotions we always pick the one which gives the biggest discount. So, we need a rule to work out what is the biggest discount[12] and which promotion offers (for both 'Q' and 'X'

[12] This strategy does not guarantee that the customer gets the maximal discount in all possible scenarios, but provided the overlapping promotions are (as per the examples) primarily organised as tiers of 'bulk' discounts for the same set of SKUs (be they all 'X' or all 'Q'), then it should generally do so.

promotions)[13] offers this biggest discount. We are also interested in which (if any) applicable promotions overlap with this 'best' promotion. Best practise really would be to do this last part in a separate rule, but here is bundled in with first part in one rule:

```
1     rule "Find Best Promotion"
2        ruleflow-group "PromotionsToApply"
3        when
4           $order: Order()
5           accumulate( Promotion($discount: discount,
6              this memberOf $order.applicablePromotions); $maxDiscount: max($discount))
7           $promotion: Promotion(
8              this memberOf $order.applicablePromotions, discount == $maxDiscount)
9           $overlappingPromotions : List() from collect (
10             Promotion(intersects($promotion), this != $promotion)
11                from $order.applicablePromotions)
12       then
13             insert(new BestPromotion($order, $promotion, $overlappingPromotions));
14    end
```

First, we simply bind ***$order*** to the ***Order*** of interest (line 4). Then we use Drools' *accumulate* mechanism to search through the ***Order***'s applicable promotions seeking out the maximum discount, and binding this to ***$maxDiscount***. (lines 5 & 6). Next (lines 7 & 8) we bind ***$promotion*** to an applicable ***Promotion*** of the ***Order*** which gives this maximal discount. Finally, in lines 9 to 11 we create a list of promotions which overlap with the selected Promotion (***$overlappingPromotions***)[14].

We want to hang onto our knowledge of ***$promotion***, ***$overlappingPromotions*** and ***$order*** so we can reuse it in subsequent rules. To do this a little internal Drools class is used ***BestPromotion***, with three attributes, ***order***, ***promotion*** and ***overlappingPromotions***. The rule builds an instance of this binding, saving the intermediate results.

Now we can move onto processing 'Q' promotions:

[13] There could be multiple promotions which offer the same 'best' discount, here we just allow the rule engine's default strategy to pick the one but obviously we could create a specific tie break mechanism

[14] We make use of a method on ***Promotions*** – ***intersects*** – which is simply a thin wrapper around the ***disjoint*** method in Java's ***Collections*** class to identify if two ***Promotions*** overlap.

```
1     rule "Manage Promotions which use allocation of parts of order to promotion"
2       ruleflow-group "PromotionsToApply"
3       when
4         $bestPromotion: BestPromotion($order: order, $promotion: promotion,
5           $overlappingPromotions: overlappingPromotions, $promotion.interaction == "Q")
6       then
7         modify($order) {
8           removeAllApplicablePromotions($promotion),
9           addPromotiontoApply($promotion)
10        }
11        for(String item: $promotion.getItemsInList()) {
12          int quantityToRemove =
13            $order.getQuantityOfItem(item) - $promotion.getQuantityOfItem(item);
14          modify($order) { setQuantityOfItem(item, quantityToRemove) };
15        }
16        delete($bestPromotion)
17    end
```

The **when** part of the rule simply rebinds the information on the ***BestPromotion*** instance we created in the "Find Best Promotion" rule, but checking we are dealing with a 'Q' promotion.

The logic in the **then** part involves a few steps. Firstly (line 8), we remove *all* the applicable promotions from the Order. We remove all of them because we are going to 'modify' the order line quantities, so we need to start again from scratch to decide which if any promotions still apply (including the current one). Next, we add the current promotion to the set of promotions to apply (line 9). As noted earlier if the same promotion is added multiple times, it's count is incremented rather than the list of promotions being extended, but this is handled internally by the Order class. The next step (lines 11 to 15) is to look at each SKU on the promotion and reduce the quantity of the SKU on the order by the designated quantity on the promotion[15]. The last step is to dispose of our temporary ***BestPromotion*** instance, so it does not interfere when we next pass through this set of rules.

Finally let's see how the 'X' promotions are handled:

[15] Obviously we can't really go about reducing the quantities on the order lines, so implicitly we are working on 'shadow copies' of the order lines for the purposes of establishing the discounts rather than the 'real' order lines

```
1     rule "Manage Promotions which cannot be used with other promotions of lesser value"
2        ruleflow-group "PromotionsToApply"
3        when
4           $bestPromotion: BestPromotion($order: order, $promotion: promotion,
5              $overlappingPromotions: overlappingPromotions, $promotion.interaction == "X")
6        then
7           modify($order) {
8              removeAllApplicablePromotions($promotion),
9              addPromotiontoApply($promotion)
10          }
11          modify($promotion) {setStatus("CLOSED") }
12          for(Object overlappingPromotionasObj: $overlappingPromotions) {
13             Promotion overlappingPromotion = (Promotion)overlappingPromotionasObj;
14             modify(overlappingPromotion) { setStatus("CLOSED") }
15          }
16          delete($bestPromotion)
17    end
```

This starts off looking very similar to the ‘Q’ promotions rule above. The only difference in the **when** part is that we seek an ‘X’ rather than a ‘Q’ promotion. The **then** part starts by clearing the applicable promotions and adding the current promotion to the list of promotions to apply as before. Things diverge at line 11. For an ‘X’ promotion we set its status to “CLOSED”, so it is only used once. Then we enter a loop (lines 12 to 14) where we set the status of all the overlapping promotions to “CLOSED” as well according to the definition of ‘X’ promotions[16]. Finally, we dispose of our temporary ***BestPromotion*** instance as we did when working with ‘Q’ promotions.

6.7 Presenting the results

We could do this procedurally of course, but for simplicity we use a couple of rules, one where there are no discounts and one where there are. The logic here is the same for both the ‘simple’ decision table solution as well as the more complex object-based solution. The No discounts rule looks like this:

```
rule "No discounts"
   ruleflow-group "DiscountCalculation"
   when
      $order: Order(promotionstoApply.size == 0)
   then
      System.out.println("**************************************************");
      System.out.println(
         String.format("Order %-9s  does not qualify for any discounts",
            $order.getDescription()));
      System.out.println("**************************************************");
end
```

The rule for outputting results when there are discounts looks like this:

[16] I couldn’t figure out how get Drools to make $overlappingPromotions a list of Promotions rather than a list of Objects hence we need to cast each entry into a Promotion on line 13 before using it on line 14

```
rule "Applied discounts"
  ruleflow-group "DiscountCalculation"
  when
    $order: Order(promotionstoApply.size > 0)
  then
    System.out.println("**************************************************");
    System.out.println(
      String.format("Congratulations your order %-9s qualifies for discounts",
        $order.getDescription()));
    System.out.println(
      String.format("Total discounts  %6.2f", $order.getTotalDiscount()));
    System.out.println("Qualifying promotions:");
    for(Promotion p: $order.getPromotionstoApply()) {
      System.out.println(
        String.format("\tPromotion %-9s giving discount of %6.2f (used %2d times)",
          p.getDescription(), p.getDiscount(), p.getUseCount()));
    }
    System.out.println("**************************************************");
end
```

7 <u>Test Results</u>

7.1 Decision Table based solution

Using the promotions in the decision table in section 3, against the following orders:

Order DT 1		Order DT 2		Order DT 3	
1108	8	1108	4	1108	5
2639	4	2639	4	2639	3
1001	4	1001	4	1001	4
2001	4	2001	4	2001	3
2002	4	2002	2	2002	2
3001	4	3001	6	3001	6

Gives the following output:

```
**************************************************
Congratulations your order Order DT 1 qualifies for discounts
Total discounts    9.00
Qualifying promotions:
        Promotion Promo 101 giving discount of   3.50 (used  1 times)
        Promotion Promo 103 giving discount of   5.50 (used  1 times)
**************************************************

**************************************************
Congratulations your order Order DT 2 qualifies for discounts
Total discounts    4.50
Qualifying promotions:
        Promotion Promo 102 giving discount of   4.50 (used  1 times)
**************************************************

**************************************************
```

```
        Order Order DT 3  does not qualify for any discounts
        **************************************************
```

Which are the expected outcomes.

Object/Rule based solution

Using the promotions defined by the decision table in section 6.2, and following orders:

```
Order 1      Order 2      Order 3      Order 4
1001   30    1001   9     1001   2     1001   30
1002   20    1002   5     1002   5     1004   30
1003   12                 1003   9
1004   15
```

The output was:

```
        **************************************************
        Congratulations your order Order 1   qualifies for discounts
        Total discounts   64.00
        Qualifying promotions:
                Promotion Prom 3    giving discount of   4.00 (used  1 times)
                Promotion Prom 1c   giving discount of  30.00 (used  1 times)
                Promotion Prom 4c   giving discount of  18.00 (used  1 times)
                Promotion Prom 1b   giving discount of  12.00 (used  1 times)
        **************************************************

        **************************************************
        Congratulations your order Order 2   qualifies for discounts
        Total discounts    8.00
        Qualifying promotions:
                Promotion Prom 1a   giving discount of   4.00 (used  2 times)
        **************************************************

        **************************************************
        Congratulations your order Order 3   qualifies for discounts
        Total discounts   14.00
        Qualifying promotions:
                Promotion Prom 3    giving discount of   4.00 (used  1 times)
                Promotion Prom 2    giving discount of  10.00 (used  1 times)
        **************************************************

        **************************************************
        Order Order 4    does not qualify for any discounts
        **************************************************
```

Again, these are the expected outcomes.

8 Conclusion

One of the fun aspects of the DMC challenges is how seeing many different takes people can have on the same ‘problem’. And usually the shorter and less specific a challenge statement is, the more it allows free-rein to one’s imagination.

This is definitely the most interesting challenge I’ve had a go at. Taking the challenge statement at it’s basic level raises some interesting points. Addressing the detection and treatment of competing promotions proved trickier than I expected but rewarding.

It’s also odd to find myself revisiting a problem I first tackled nearly 30 years ago. But it was no surprise to find going that bit deeper into the problem, I needed the extra power and flexibility the combination of objects and rules gives you over straight decision tables.

9 Appendix – Java Object Model

9.1 ItemQuantity Class

```
package jetsetbc.promotion.datamodel;
/**
 * Simple class to define a quantity of a certain type so we know
 * we are not comparing Apples with Oranges.
 * @author Bob
 *
 */
public class ItemQuantity {
	private String description;
	private int quantity;

	public String getDescription() { return description; }

	public int getQuantity() { return quantity; }

	public ItemQuantity(String description, int quantity) {
		this.description = description;
		this.quantity = quantity;
	}

}
```

9.2 ListOfItemQuantities Class

```
package jetsetbc.promotion.datamodel;

import java.util.*;
/**
 * Class representing a named collection of quantities of distinct types of item
 * @author Bob
 *
 */
public class ListOfItemQuantities {
	private String description; // the name
	// we use a hash map to ensure we only have one entry for
	// any item type
	protected HashMap<String, ItemQuantity> itemQuantities;

	public ListOfItemQuantities(String description) {
		this.description = description;
		itemQuantities = new HashMap<String, ItemQuantity>();
	}

	public String getDescription() { return description; }

	// get the item type and quantity of a particular item type
	public ItemQuantity getItemQuantity(String itemId) {
		return itemQuantities.get(itemId);
	}

	// get the quantity of a particular item type. If the collection
	// does not hold the item return 0
	public int getQuantityOfItem(String itemId) {
		ItemQuantity quantity = getItemQuantity(itemId);
		return quantity == null ? 0 : quantity.getQuantity();
	}

	// set the quantity of a particular item type overwriting any
	// existing value
	public void setQuantityOfItem(String itemId, int quantity) {
		itemQuantities.put(itemId, new ItemQuantity(itemId, quantity));
	}

	// get the item types in the collection
	public Set<String> getItemsInList() {
		return itemQuantities.keySet();
	}

	// get the item type/ quantity combinations from in the collection
	public ArrayList<ItemQuantity> getItemQuantities() {
		ArrayList<ItemQuantity> itemQuantitiesAsArray =
				new ArrayList<ItemQuantity>(itemQuantities.size());
		for (String itemId: itemQuantities.keySet()) {
			itemQuantitiesAsArray.add(itemQuantities.get(itemId));
		}
		return itemQuantitiesAsArray;
	}
}
```

9.3 Order Class

```
package jetsetbc.promotion.datamodel;

import java.util.*;

```

```
/**
 * A class representing an order and any eligible or applied promotions
 * @author Bob
 *
 */
public class Order extends ListOfItemQuantities {
	private ArrayList<Promotion> applicablePromotions = new ArrayList<Promotion>();
	private ArrayList<Promotion> promotionsToApply = new ArrayList<Promotion>();

	public Order(String desciption) {
		super(desciption);
	}

	// add a (new) promotion this order qualifies for
	public void addApplicablePromotion(Promotion promotion) {
		if(!applicablePromotions.contains(promotion)) {
			applicablePromotions.add(promotion);
		}
	}

	// remove a promotion from the list the order qualifies for
	public void removeApplicablePromotion(Promotion promotion) {
		applicablePromotions.remove(promotion);
	}

	// clear the list of promotions the order qualifies for
	public void removeAllApplicablePromotions(Promotion promotion) {
		applicablePromotions.clear();
	}

	// add a promotion to the list which will be applied to this promotion
	// if the promotion is already on the list it is not added again,
	// but the useage count is incremented
	public void addPromotiontoApply(Promotion promotion) {
		if(!promotionsToApply.contains(promotion)) {
			promotionsToApply.add(promotion);
		}
		promotion.incrementUseCount();
	}

	public ArrayList<Promotion> getApplicablePromotions() {
		return applicablePromotions;
	}

	public ArrayList<Promotion> getPromotionstoApply() {
		return promotionsToApply;
	}

	// calculate the total discount by adding up the discount * useCount
	// for all the promotions which are to be applied to the order
	public double getTotalDiscount() {
		double totalDiscount = 0.0;
		for (Promotion discount : promotionsToApply) {
```

```
                    totalDiscount += discount.getDiscount() * discount.getUseCount();
            }
            return totalDiscount;
    }

}
```

9.4 Promotion Class

```
package jetsetbc.promotion.datamodel;

import java.util.Collections;

/**
 * Class representing a promotion and its status in an order
 * @author Bob
 *
 */
public class Promotion extends ListOfItemQuantities {
        private String interaction = "Q";
        private double discount;
        private String status = "OPEN";
        private int useCount = 0;

        public Promotion(String description, double discount, String interaction) {
                super(description);
                this.discount = discount;
                this.interaction = interaction;
        }

        public Promotion(String description, double discount) {
                super(description);
                this.discount = discount;
        }

        public String getInteraction() { return interaction; }

        public double getDiscount() { return discount; }

        public String getStatus() { return status; }

        public int getUseCount() { return useCount; }

        public void setStatus(String status) {this.status = status; }

        public void incrementUseCount() { this.useCount += 1; }

        public void reset() {
                this.useCount = 0;
                this.status = "OPEN";
        }

        // determines if two promotions interact with one another by comparing the
```

```
		// SKUs they require
		public boolean intersects(Promotion otherPromo) {
			boolean answer =
				!Collections.disjoint(getItemsInList(), otherPromo.getItemsInList());
			return answer;
		}

	}

```