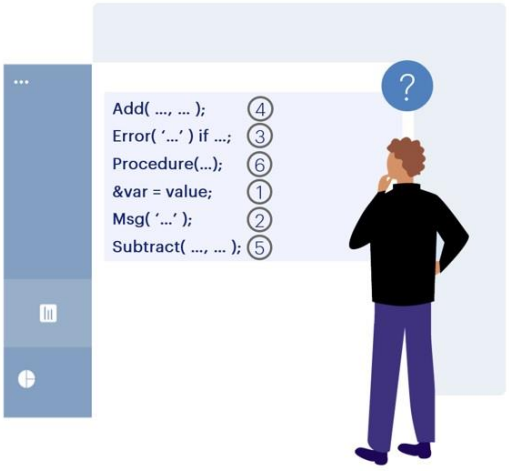
Árvore de avaliação de disparo de regras e fórmulas

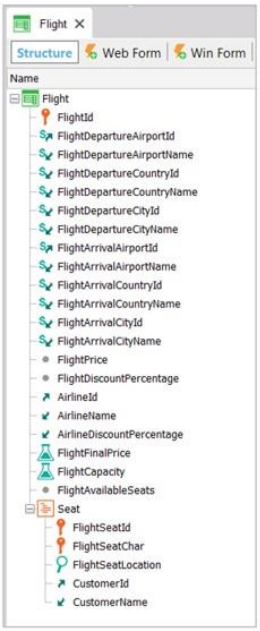
GeneXus™

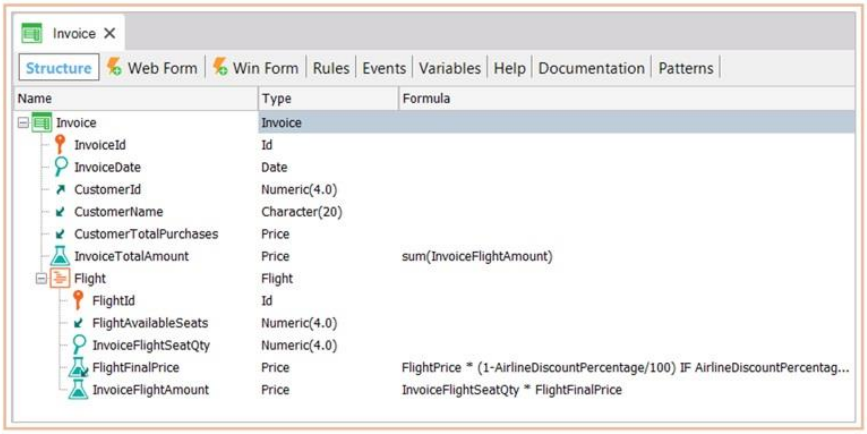


Sabemos que as regras em uma transação são declaradas em qualquer ordem e é GeneXus quem determina o momento em que cada uma é disparada. Isto às vezes é confuso para os desenvolvedores, porque sentem que perdem o controle.

Mas na verdade trata-se de uma vantagem. Os desenvolvedores só precisam se preocupar em declarar a lógica e GeneXus inferirá automaticamente onde e quando cada regra deverá ser disparada.

Reality

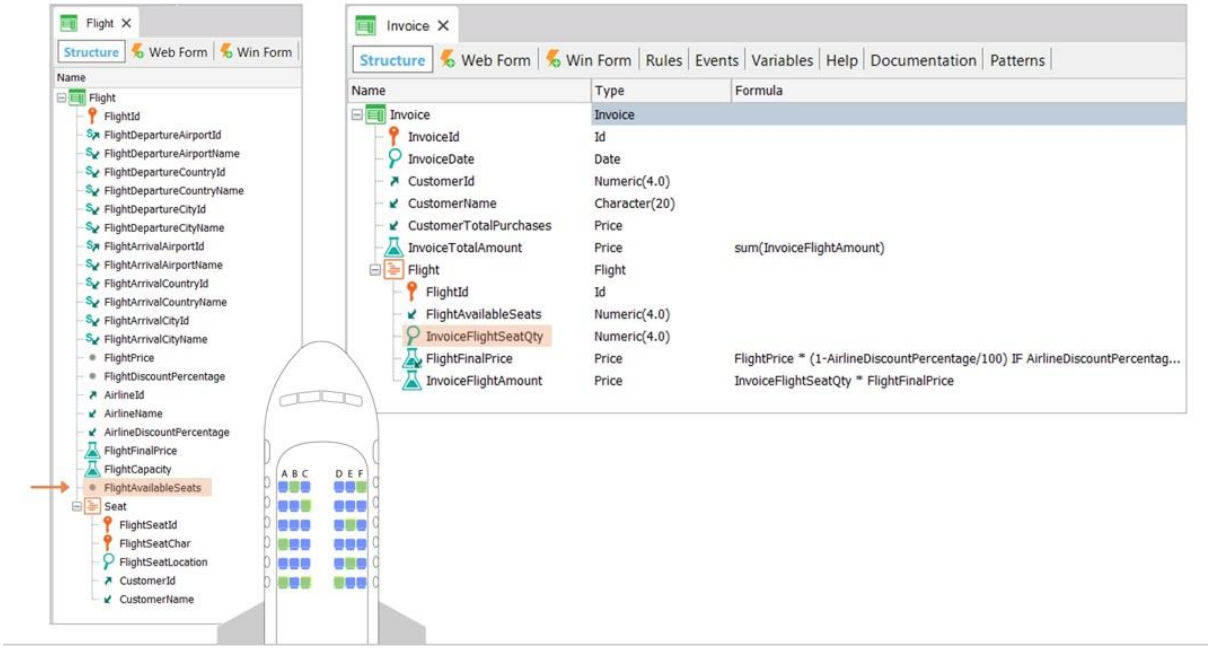




Para entender o tema, nos basearemos na transação de Faturas (Invoice), que possui um segundo nível (Flight), que representa os voos incluídos na fatura.

Vamos nos concentrar nesta transação e no disparo de suas regras.

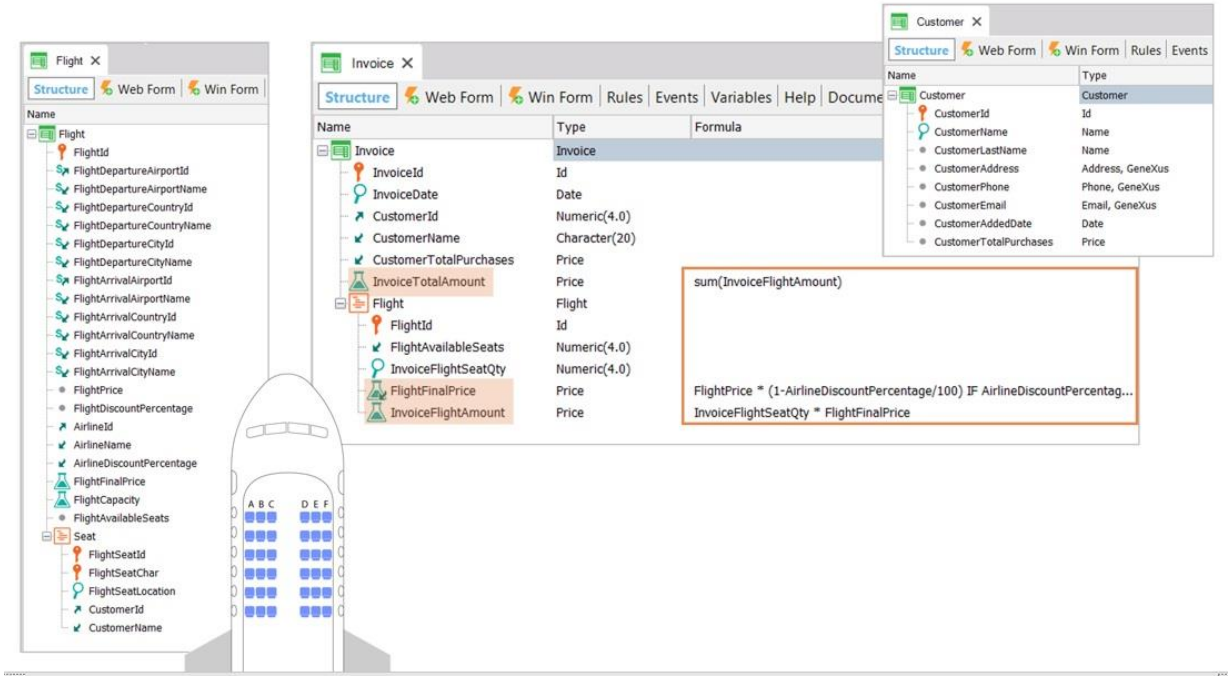
Reality



Observemos que adicionamos à transação Flight o atributo FlightAvailableSeats, que será usado para registrar os assentos disponíveis de cada voo, que serão diminuídos cada vez que seja feita uma fatura para um cliente que compra uma quantidade de assentos no voo.

O adicionamos, então, a este nível. Será um atributo inferido.

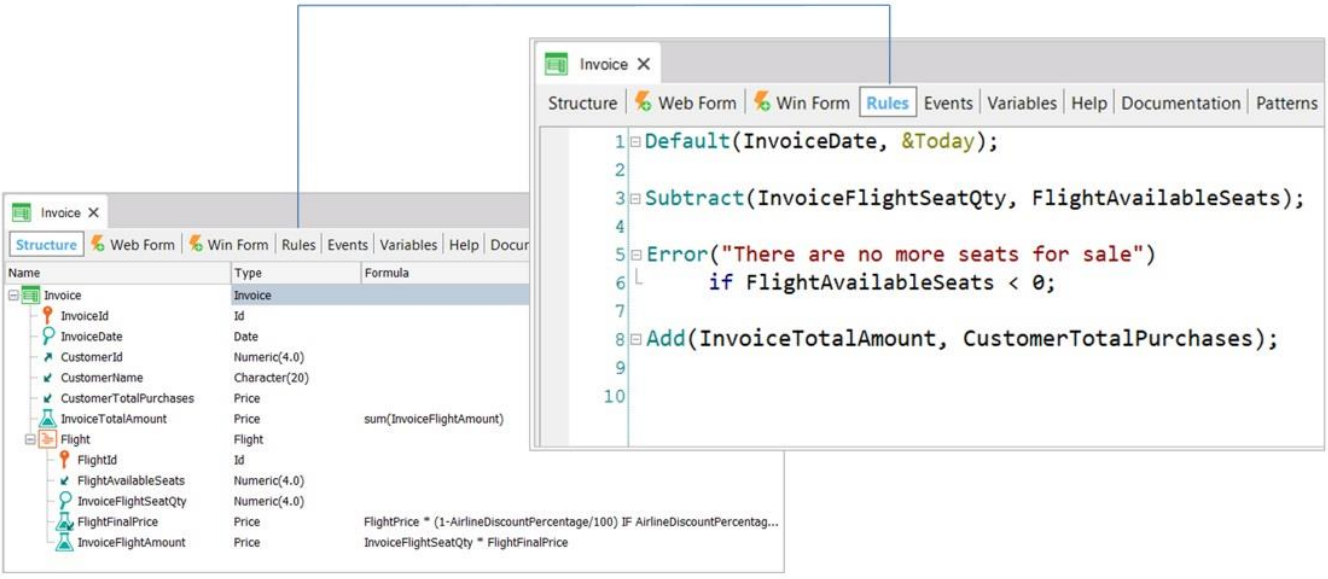
Reality



Também adicionamos à transação Customer o atributo CustomerTotalPurchases, para registrar o total comprado pelo cliente de passagens aéreas. Também o adicionamos à nossa transação como atributo inferido.

Os atributos InvoiceTotalAmount, FlightFinalPrice e InvoiceFlightAmount da estrutura de Invoice foram definidos como fórmulas.

Invoice rules

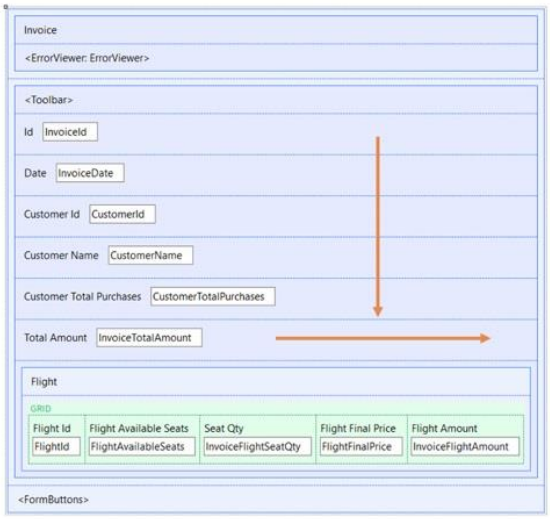


Na transação Invoice, definimos as seguintes regras para especificar seu comportamento:

A regra Default, que inicializa o atributo da data da fatura com a data de hoje; a regra Subtract, que diminui a quantidade de assentos disponíveis do voo de acordo com a quantidade de assentos comprados na fatura - observemos que estará diminuindo um atributo da tabela Flight: FlightAvailableSeats é, aqui, inferido-; a regra Error, que exibe uma mensagem de erro se o voo não tiver assentos disponíveis; e a regra Add, que soma o total da fatura ao total de compras realizadas pelo cliente - novamente, atributo presente em uma tabela da estendida, Customer-.

Evaluation tree

```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```



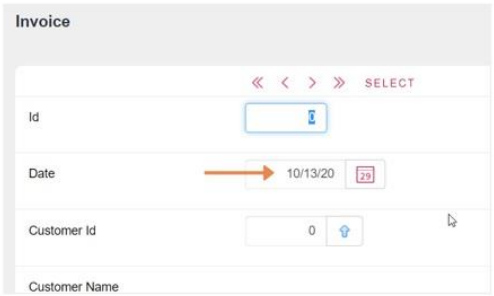
Resumindo, temos todas estas regras e fórmulas definidas na transação Invoice:

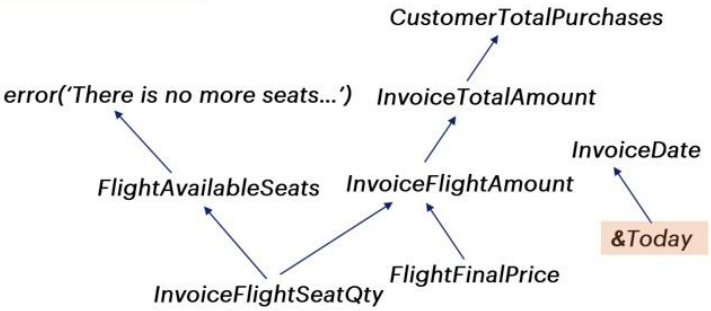
A grande questão é: como GeneXus sabe em que ordem deve dispará-las, quando sim e quando não?

Claro, há uma primeira ordem natural que é a que corresponde à ordem dos atributos na tela (de cima para baixo e da esquerda para a direita).

Evaluation tree

```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```

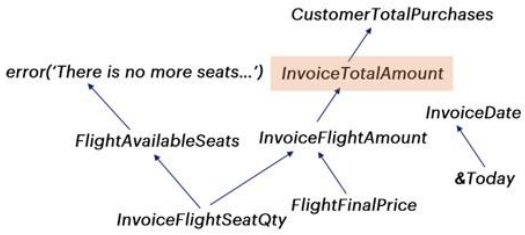


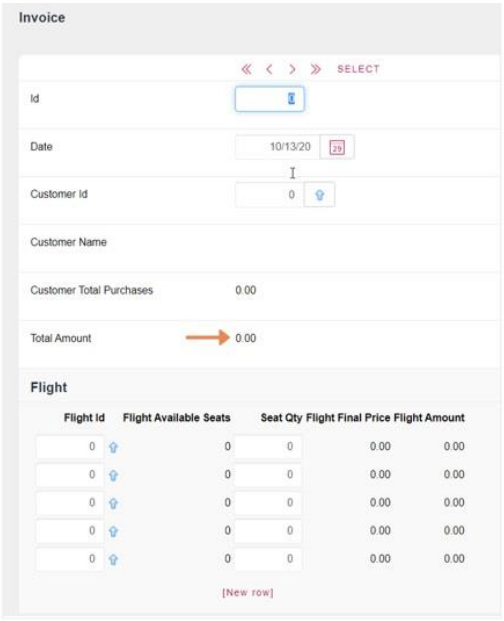


Uma regra ou fórmula é disparada assim que se tem a informação de que precisa. Por exemplo, a regra Default precisa apenas do valor da variável &Today e saber que está em modo Insert. Por isso apenas abrimos a tela em modo Insert e já vemos o valor no campo, mesmo que ainda não tenhamos chegado nele (apenas estamos posicionados sobre InvoiceId).

Evaluation tree

```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```

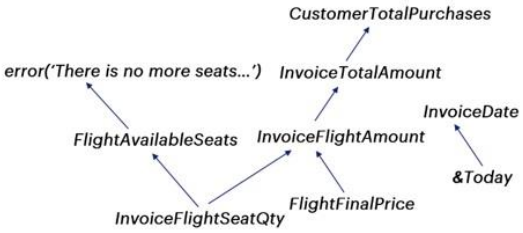


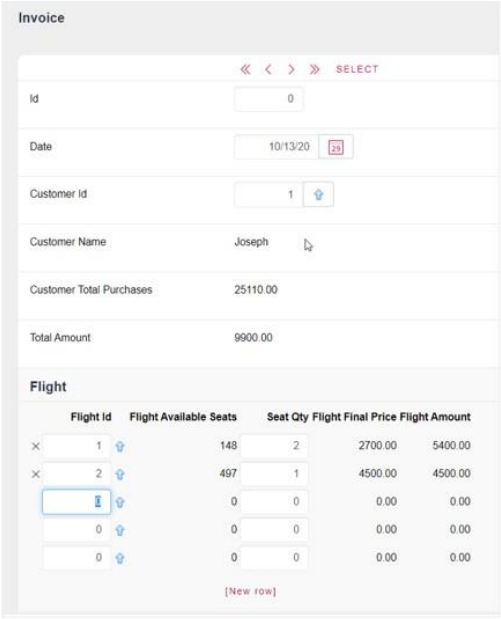


Pensemos no que acontece com a fórmula do primeiro nível: InvoiceTotalAmount, que é uma soma de um atributo do segundo nível. Como a fórmula Sum necessita apenas do atributo InvoiceFlightAmount, será disparada para o cabeçalho antes mesmo de ter sido possível inserir a primeira linha, dando 0.

Evaluation tree

```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```





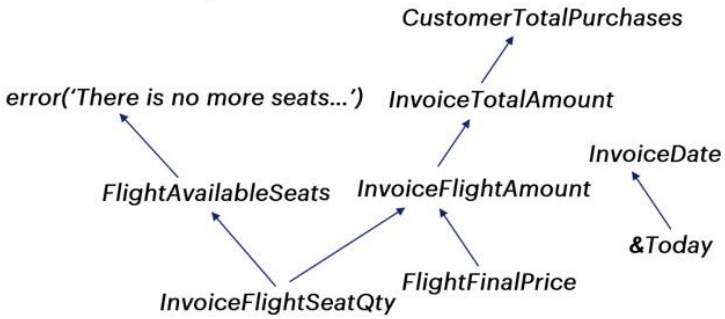
Mas em seguida, conforme inserimos linhas, para cada uma será novamente disparada. Mas, o que acontece se entramos em modo Update na transação e, por exemplo, modificamos algo de uma linha que não modifica de forma alguma o InvoiceFlightAmount? (neste caso não temos nenhum atributo para modificar que não modifique esse valor, porque os únicos dois atributos editáveis são o Id da linha que não pode ser modificado por ser parte da chave primária e depois o InvoiceFlightSeatQty que sim, modifica o valor de InvoiceFlightAmount, mas imaginemos que houvesse)

Por exemplo, que deva ser indicado se algum dos passageiros é diabético e para a linha em questão tínhamos que sim e agora queremos mudar para não.

Obviamente, nesse caso, a fórmula do cabeçalho não seria recalculada.

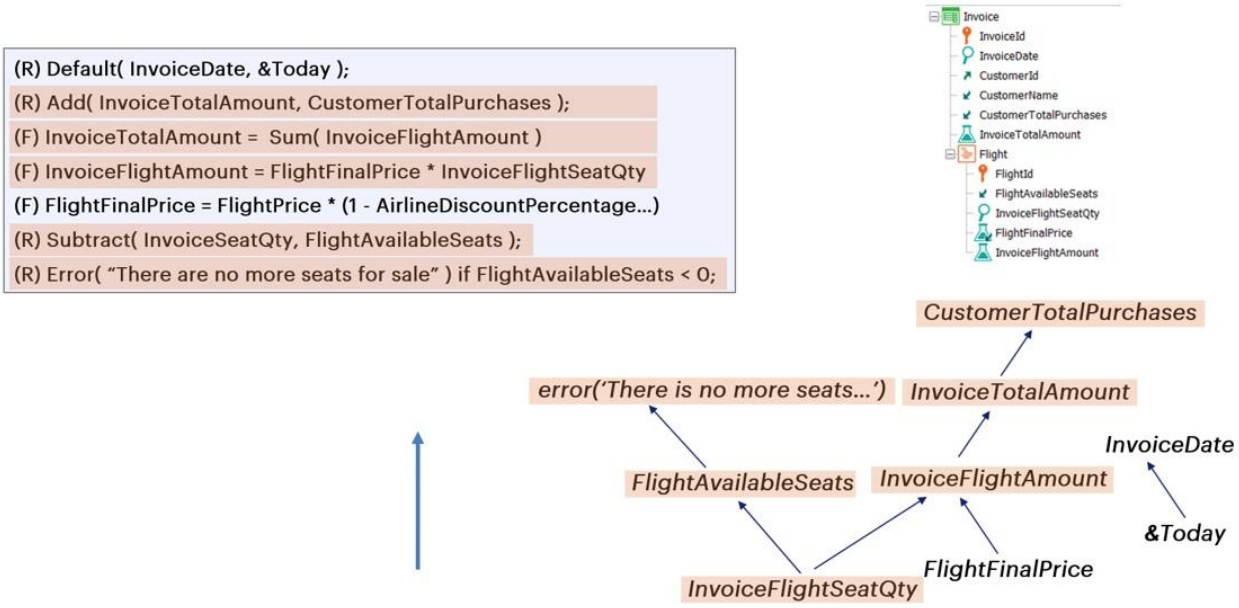
Evaluation tree

```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```



Isto "obviamente" também é verdadeiro para GeneXus, que internamente o que faz é extrair as dependências existentes entre lugares que assumem os controles na tela, as regras e as fórmulas, para construir uma árvore de dependências (conhecida como árvore de avaliação) que é a que determinará quais regras e fórmulas deverão ser novamente disparadas diante de mudanças em atributos da tela. Em nosso exemplo, será assim:

Evaluation tree



Por exemplo, vejamos o que acontece com o atributo InvoiceFlightSeatQty. Dele depende a atualização com subtract do atributo FlightAvailableSeats, assentos disponíveis do voo, do qual depende, por sua vez, o disparo da regra error.

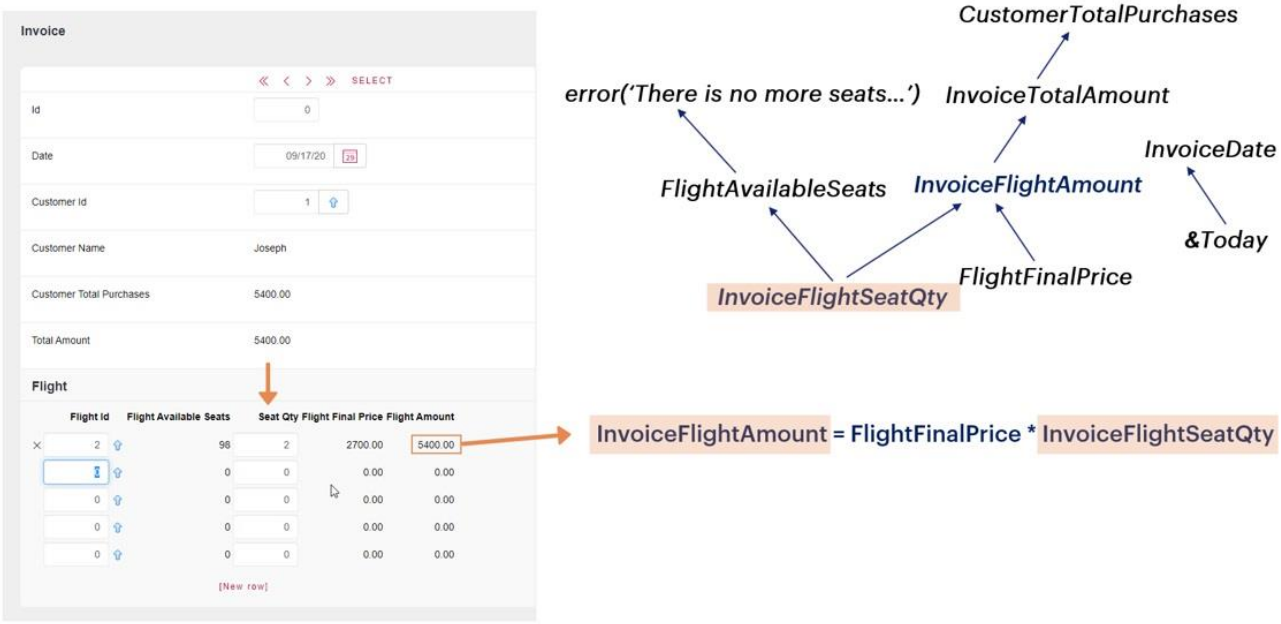
Por esse motivo a condição da regra error deve ser escrita sabendo que sempre, por esta dependência, já terá sido executado o subtract e é por isso que se coloca a condição "menor que zero".

Lembre-se de que, como já estudamos, caso dê assentos negativos, se desfará tudo o que foi feito até ocorrer este erro.

Por sua vez, também de InvoiceFlightSeatQty depende a atualização da fórmula InvoiceFlightAmount, da qual, por sua vez, depende a do cabeçalho, InvoiceTotalAmount, da qual depende a atualização do total de compras do cliente.

Podemos imaginar que a árvore é executada de baixo para cima, ou seja, que cada vez que muda o valor de um atributo, são executadas todas as regras e fórmulas que dependem desse atributo (e que na árvore se encontram acima).

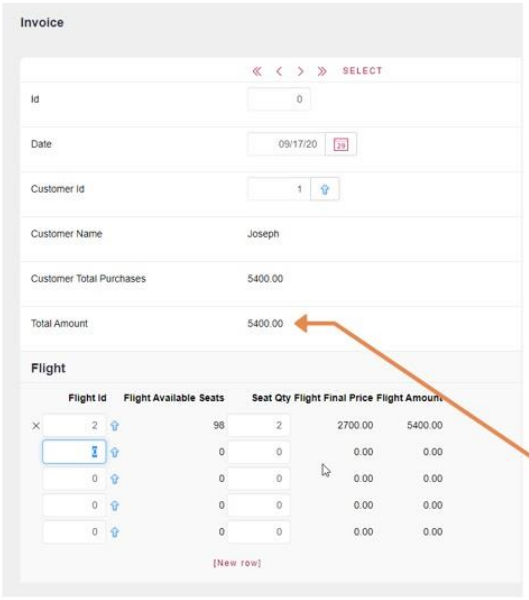
Evaluation tree

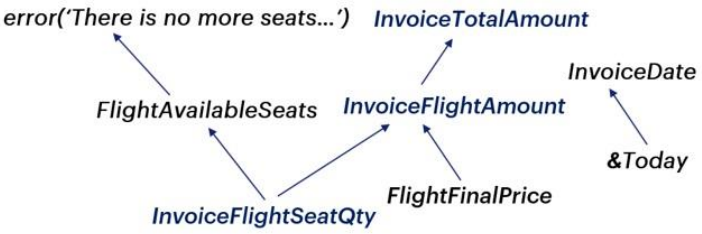


Vamos seguir o exemplo anterior:

Se altera a quantidade de assentos em uma linha de uma fatura (InvoiceFlightSeatQty), como este atributo intervém na fórmula que calcula o valor do voo (InvoiceFlightAmount), esta fórmula será novamente disparada.

Evaluation tree



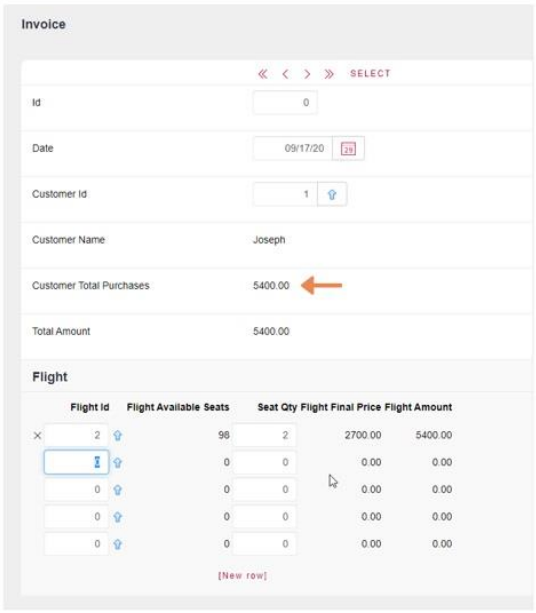


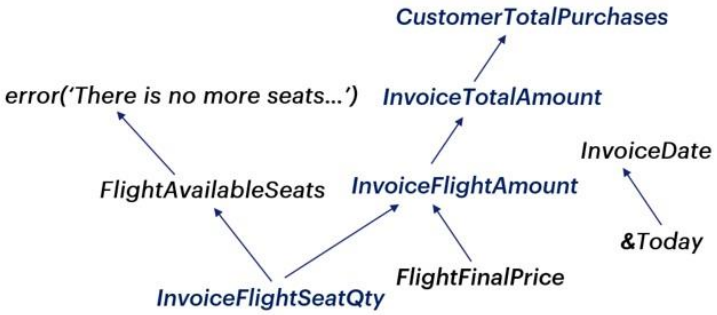
InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty

InvoiceTotalAmount = Sum(*InvoiceFlightAmount*)

Quando disparada novamente, também deverá ser novamente disparada a fórmula correspondente ao total da fatura (InvoiceTotalAmount).

Evaluation tree





InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty

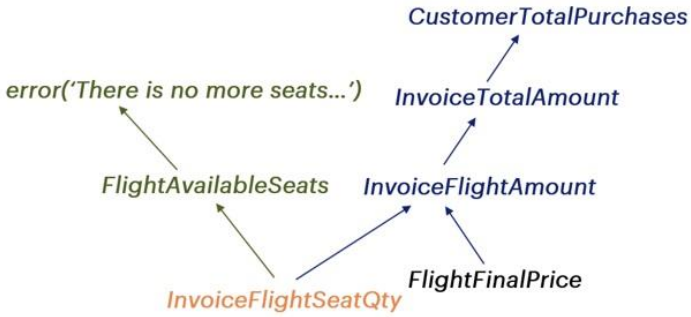
InvoiceTotalAmount = Sum(*InvoiceFlightAmount*)

Add(*InvoiceTotalAmount*, CustomerTotalPurchases);

Por último, por alterar o total, também deverá ser disparada a regra Add(InvoiceTotalAmount, CustomerTotalPurchases), uma vez que deverá ser atualizado o total de compras do cliente.

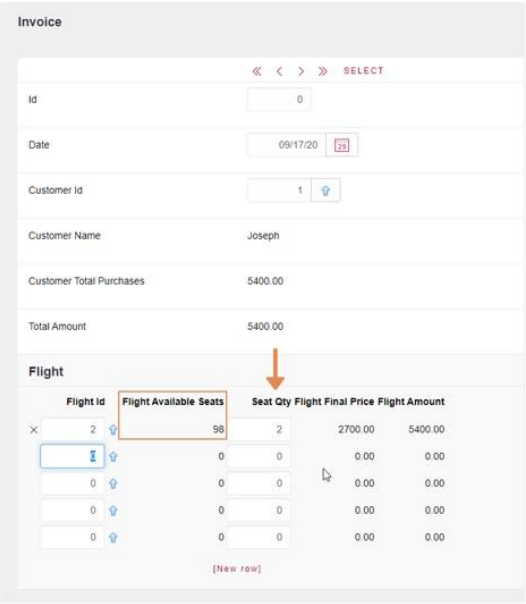
Evaluation tree

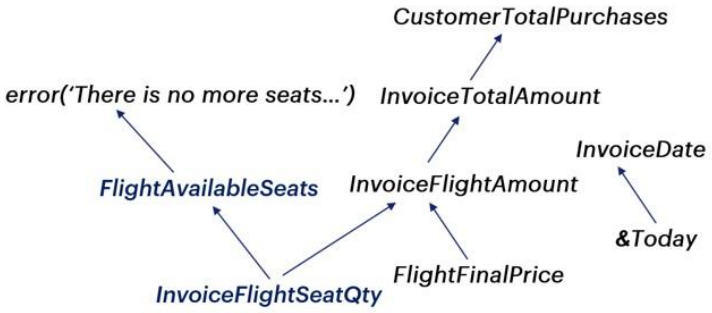
```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```



Além de serem disparadas todas as fórmulas e regras envolvidas na ramificação direita da árvore a partir do atributo InvoiceFlightSeatQty, também serão disparadas as fórmulas e regras envolvidas na ramificação esquerda.

Evaluation tree






```
Subtract( InvoiceSeatQty, FlightAvailableSeats );

Error( "There are no more seats for sale" )
if FlightAvailableSeats < 0;
```

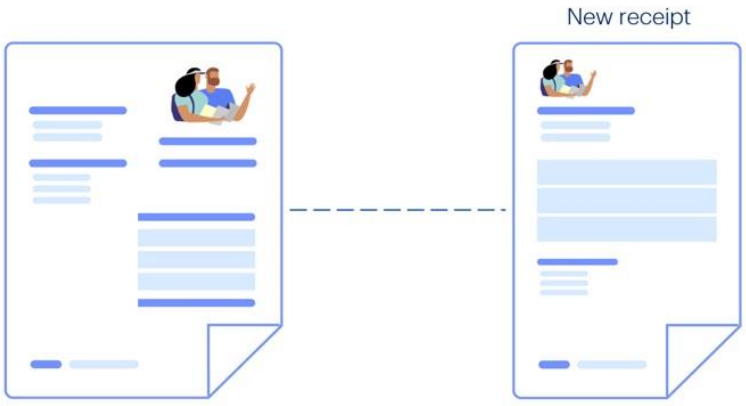
Como já vimos, ao alterar o valor do atributo InvoiceFlightSeatQty, também será novamente disparada a regra Subtract(InvoiceFlightSeatQty, FlightAvailableSeats) que atualiza a quantidade de assentos disponíveis no voo (FlightAvailableSeats).

E consequentemente, por modificar esta regra, o valor do atributo FlightAvailableSeats será avaliado para determinar se terá que disparar a regra Error, que indica que não há mais assentos disponíveis.

Se a condição para disparar o erro for satisfeita, tudo o que foi feito na árvore desde a mudança no atributo InvoiceFlightSeatQty será desfeito automaticamente.

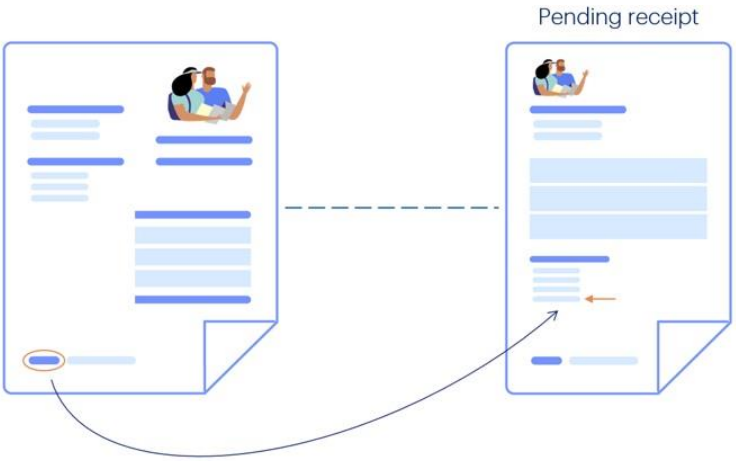
E os dados da base de dados voltarão ao estado anterior à execução da regra Error.

Reality



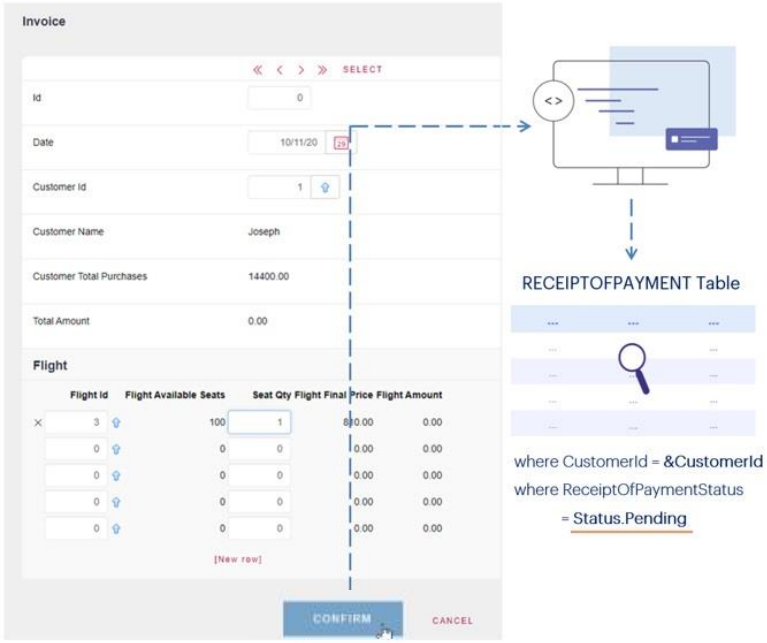
Vejamos agora um exemplo em que uma regra nem sempre se dispara no momento desejado.

Suponhamos que, imediatamente após fazer uma fatura para um cliente por uma quantidade de assentos de voo comprados, desejamos gerar um recibo de pagamento. Se o cliente estiver em dia com os pagamentos, então geramos um novo recibo,



Name	Type
ReceiptOfPayment	ReceiptOfPayment
ReceiptOfPaymentId	Id
ReceiptOfPaymentDate	Date
ReceiptOfPaymentStatus	Status
CustomerId	Id
CustomerName	Name
Invoice	Invoice
InvoiceId	Id
InvoiceTotalAmount	Price
ReceiptOfPaymentInvoiceTotalPayed	Numeric(4.0)

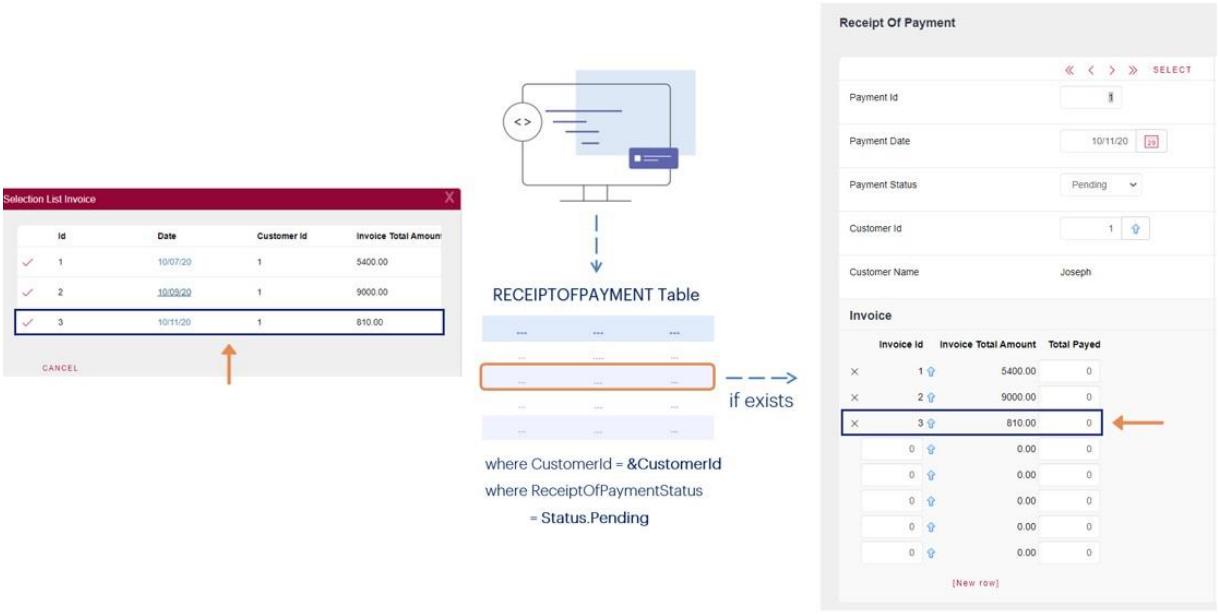
mas se não está, então adicionamos o valor desta fatura ao recibo pendente anterior. Criamos uma transação ReceiptOfPayment.



A mesma é composta pelo identificador do recibo, a data, o status que é um domínio enumerado, com os valores Pendente e Concluído, o cliente, e um segundo nível para registrar as faturas para as quais se emite o recibo de pagamento.

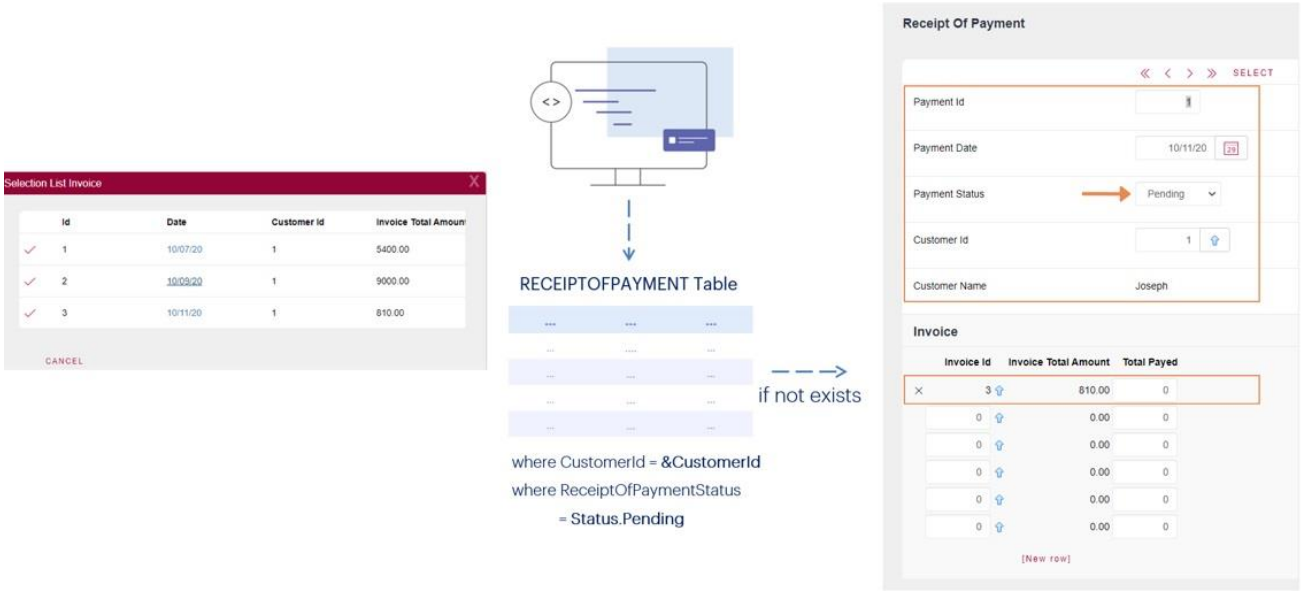
Então, quando é inserida uma nova fatura, é pesquisado por programa se já existe um recibo desse cliente com status Pending.

Reality

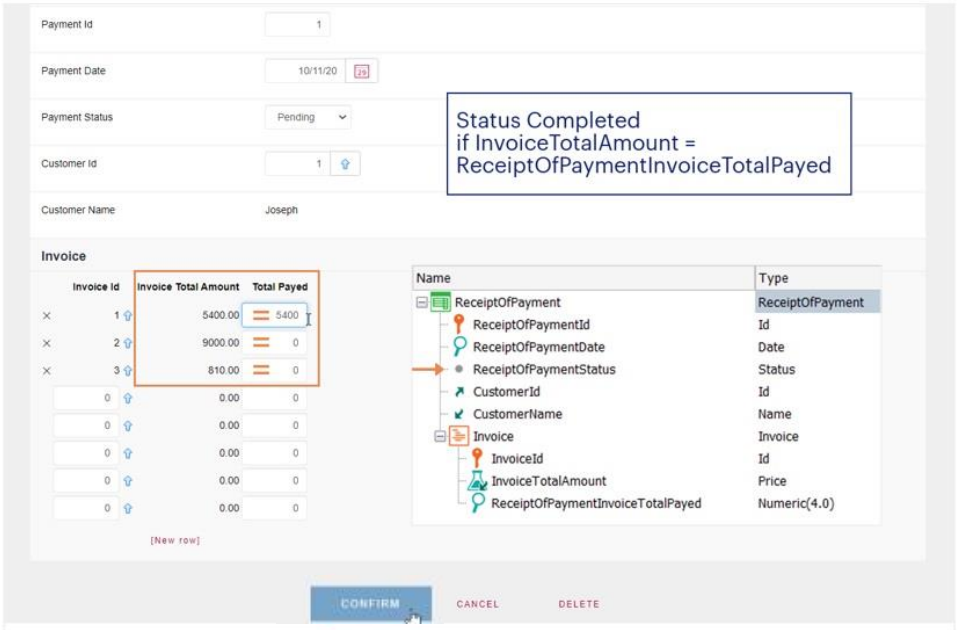


Se existir, é adicionada uma nova linha com esta fatura.

Reality

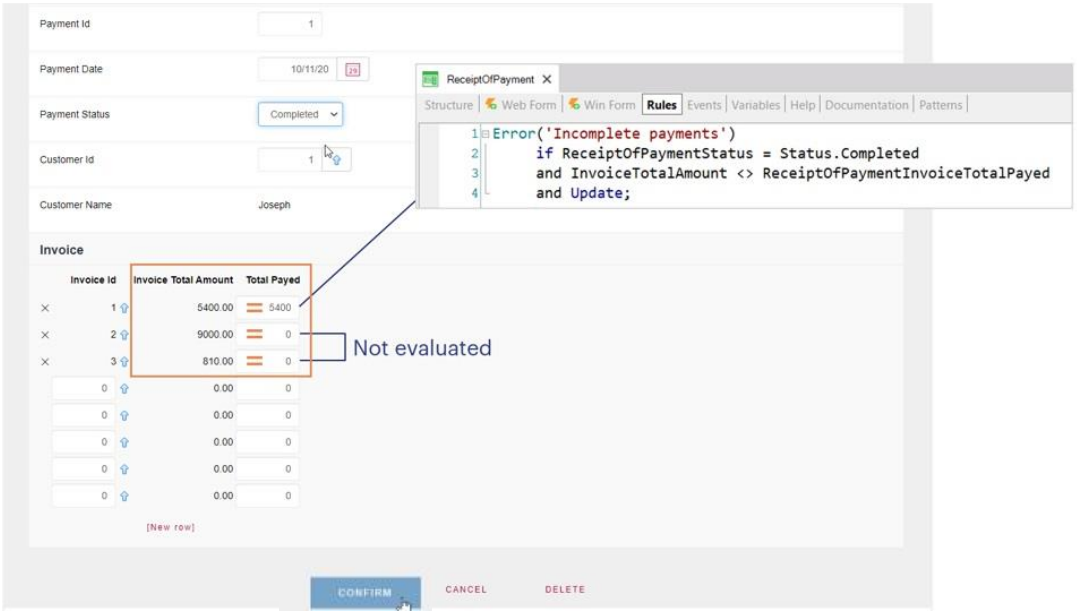


Se não existir, são criados o cabeçalho e a linha, e o cabeçalho fica com status Pending.



Em seguida, o cliente vem para pagar, então o funcionário abre a transação e modifica o atributo ReceiptOfPaymentInvoiceTotalPaid para as faturas que o cliente deseje pagar.

O status do recibo (ReceiptOfPaymentStatus) só poderá ser definido como Completed se coincidem para todas as linhas o valor de InvoiceTotalAmount com o do ReceiptOfPaymentInvoiceTotalPaid. Suponhamos que essa modificação seja realizada pelo usuário, ou seja, que é ele quem modifica o valor de ReceiptOfPaymentStatus, portanto, deve-se confirmar que não seja permitido mudar para Completed se houver alguma fatura não paga ou paga incorretamente.



Poderíamos pensar em colocar a seguinte regra de erro:

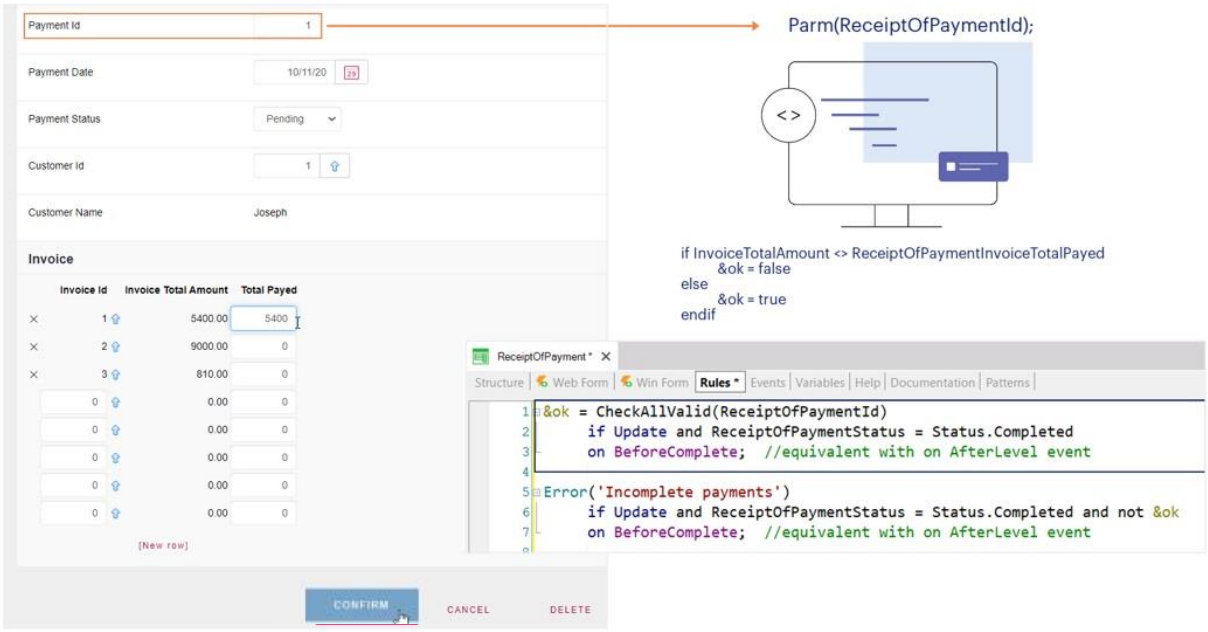
Condicionando-a para disparar quando estamos atualizando um recibo, o status é Completed e não coincide o valor do atributo que indica o que deve ser pago e o que indica o que se pagou para uma linha.

No entanto... quando será disparada esta regra?

Claramente se entramos na transação, mudamos o Status para Completed e para uma linha inserimos um valor diferente do esperado, será disparada.

Mas, o que aconteceria se para outra linha que tem zero na quantidade paga, nem sequer entramos? A regra será disparada?

A resposta é não. Quando executamos uma transação em modo Update, podemos não querer modificar o cabeçalho e alterar apenas uma linha. Nesse caso, o que será disparado? O cabeçalho se atualizará sempre. E então apenas a linha modificada. Para ela, tudo será disparado de acordo com a árvore de avaliação.



Como resolveríamos este caso então?

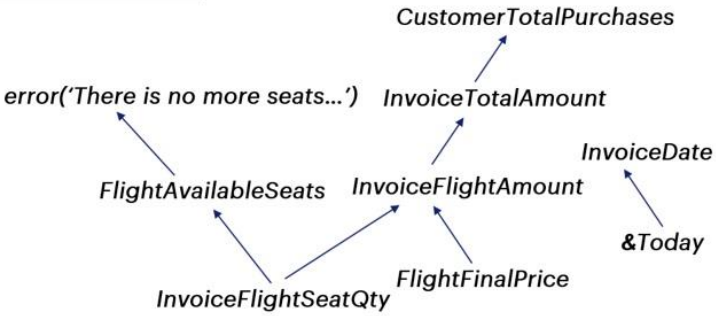
Uma solução é chamar um procedimento para o qual enviamos o id do recibo, uma vez que demos tempo para modificar todas as linhas, e uma vez que estas modificações foram realizadas na base de dados, mas antes do commit, assim podemos desfazer. Esse procedimento percorre TODAS as linhas e garante que não ficou nenhuma com valor diferente para os atributos que nos interessam.

Embora para a regra Error poderia parecer que não precisamos condicionar ao evento BeforeComplete uma vez que a variável &ok a ser avaliada já está, na verdade é necessário. Se não condicionássemos o erro para exatamente o mesmo evento, isso quebraria as dependências entre as duas regras.

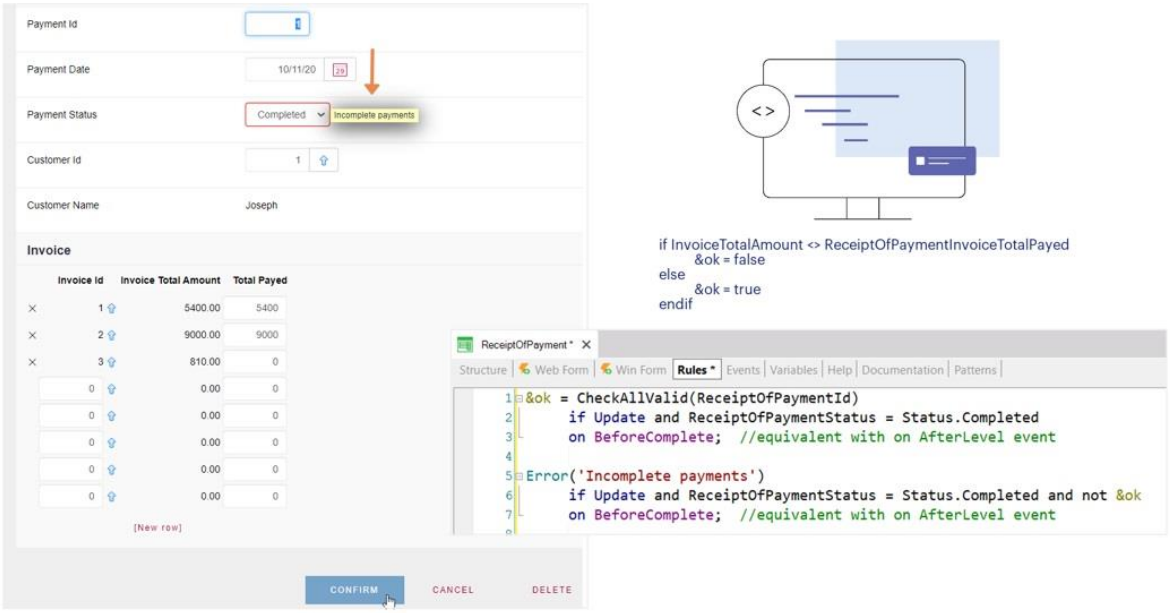
Cada evento de disparo tem sua própria árvore de avaliação, o que significa que se condicionamos muitas regras para o mesmo evento, as ordenará no momento em que o evento ocorrer de acordo com suas dependências, tal como vimos antes.

Evaluation tree

```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty, FlightAvailableSeats );
(R) Error( "There are no more seats for sale" ) if FlightAvailableSeats < 0;
```

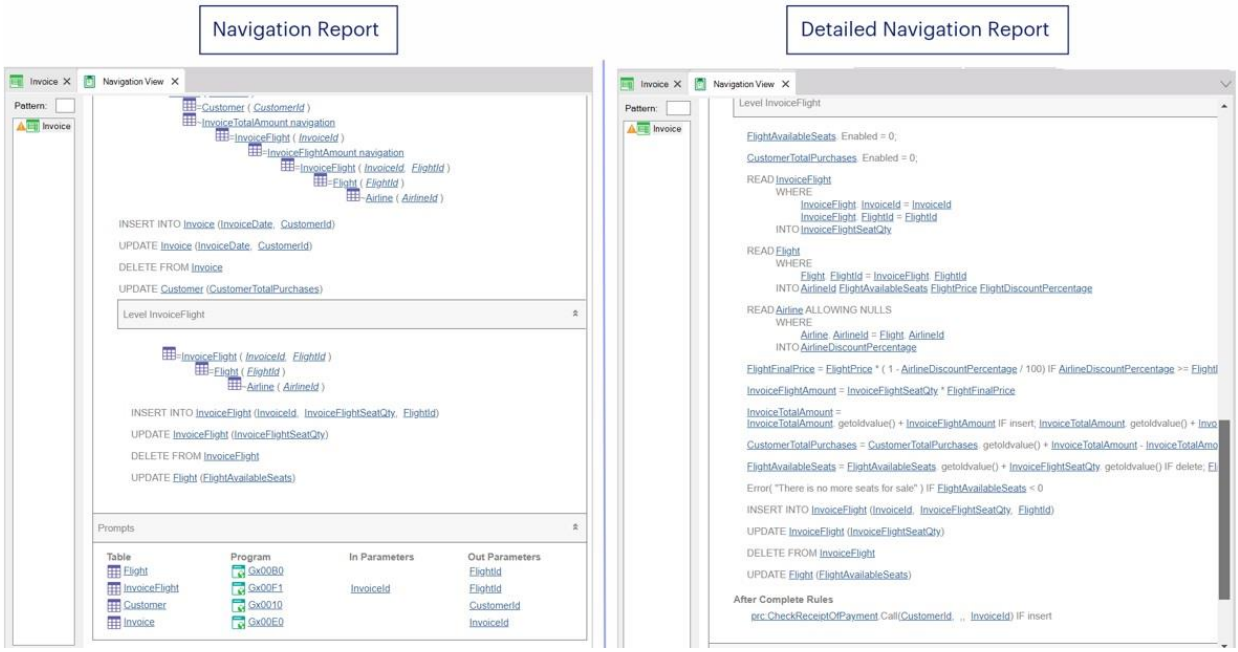


Concluindo, as regras e fórmulas que se definem em uma transação costumam estar inter-relacionadas e GeneXus determina as dependências entre elas, bem como sua ordem de avaliação.



Às vezes, a árvore de avaliação não determina a ordem de execução que queremos: um exemplo claro disso é o que acabamos de ver, onde devemos atrasar o momento de disparo do procedimento que verifica os registros das linhas e o erro subsequente.

Detailed navigation

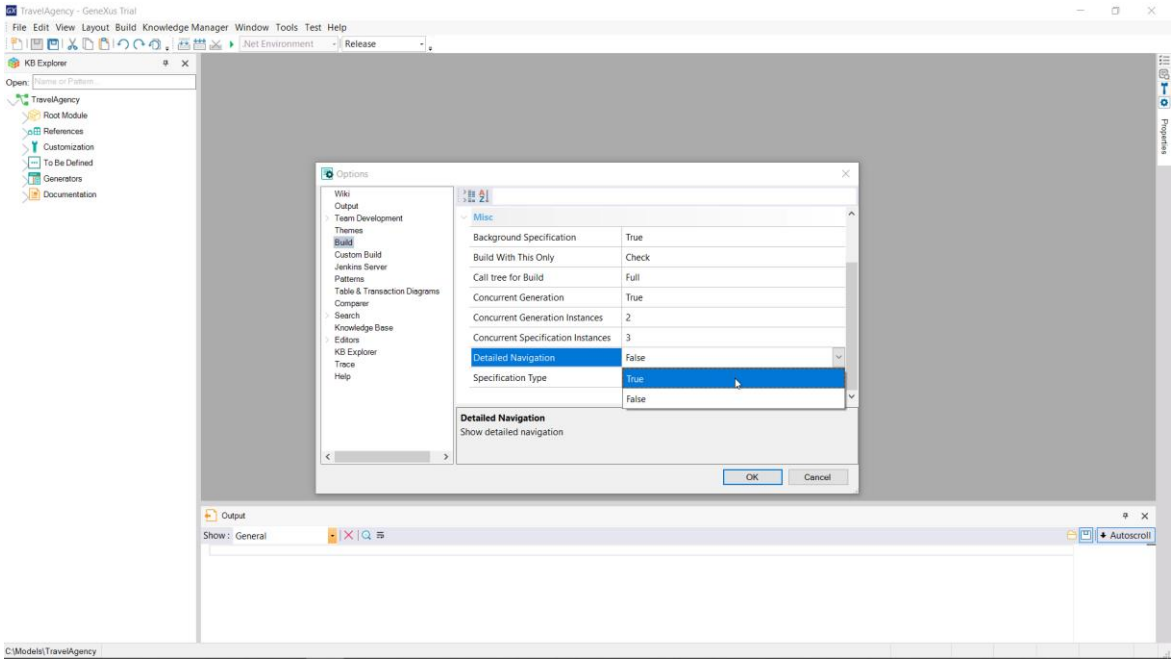


Se deseja ver mais detalhadamente a ordem das avaliações disparadas por GeneXus, pode utilizar a lista de navegação detalhada.

Aqui vemos a diferença entre ambas... vejamos, por exemplo, que na navegação detalhada nos são mostradas as regras e os momentos em que serão disparadas, o que não ocorre no outro caso.

A navegação detalhada pode ser útil nos casos em que precisamos entender bem onde está sendo disparada uma fórmula ou regra, mas normalmente leva mais tempo de especificação e, muitas vezes, não precisamos disso.

Detailed navigation



Para habilitá-la, deve ir ao menu Tools > Options e, dentro da categoria Build, ativar a propriedade Detailed Navigation.

GeneXus™

training.genexus.com
wiki.genexus.com