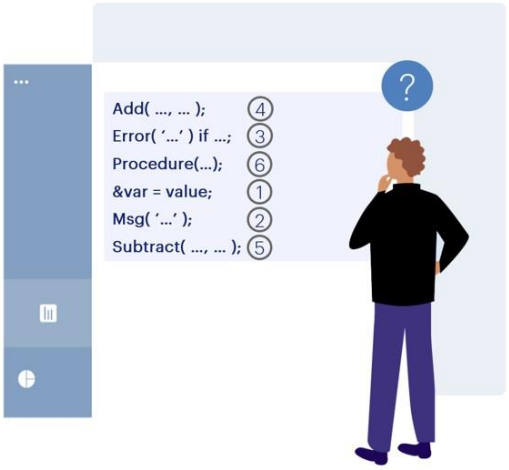
# Árbol de evaluación de disparo de reglas y fórmulas

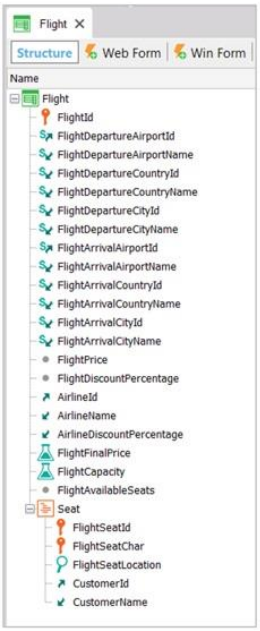
GeneXus

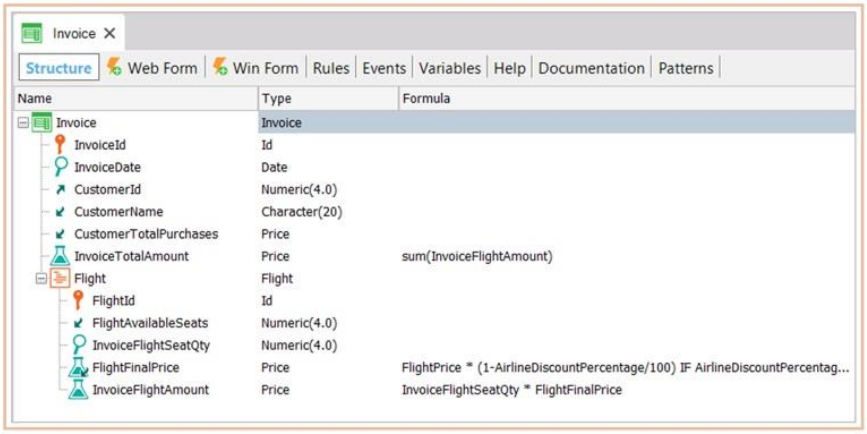



Sabemos que las reglas en una transacción se declaran en cualquier orden y es GeneXus quien determina el momento en que cada una se dispara. Esto a veces resulta confuso para los desarrolladores, porque sienten que pierden el control.

Pero en realidad se trata de una ventaja. Los desarrolladores solo debemos preocuparnos por declarar la lógica y GeneXus inferirá automáticamente, dónde y cuándo cada regla deberá dispararse.

# Reality

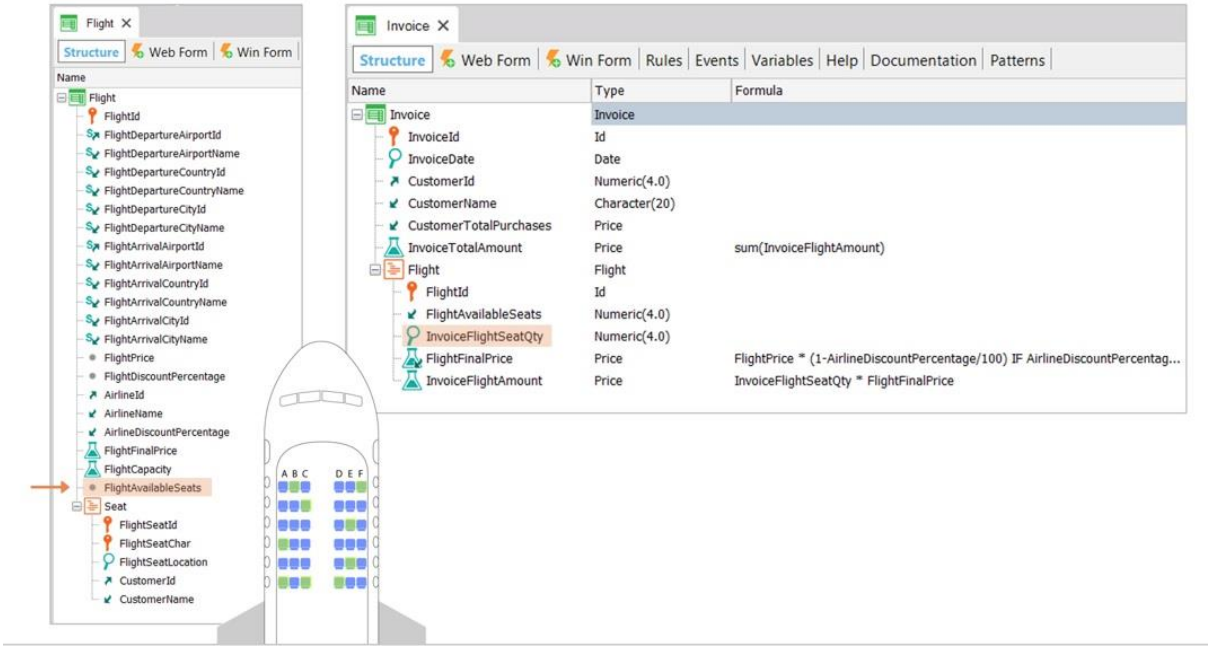



Para entender el tema, nos basaremos en la transacción de Facturas (Invoice) que cuenta con un segundo nivel (Flight), que representa los vuelos incluidos en la factura.

Nos vamos a concentrar en esta transacción y en el disparo de sus reglas.

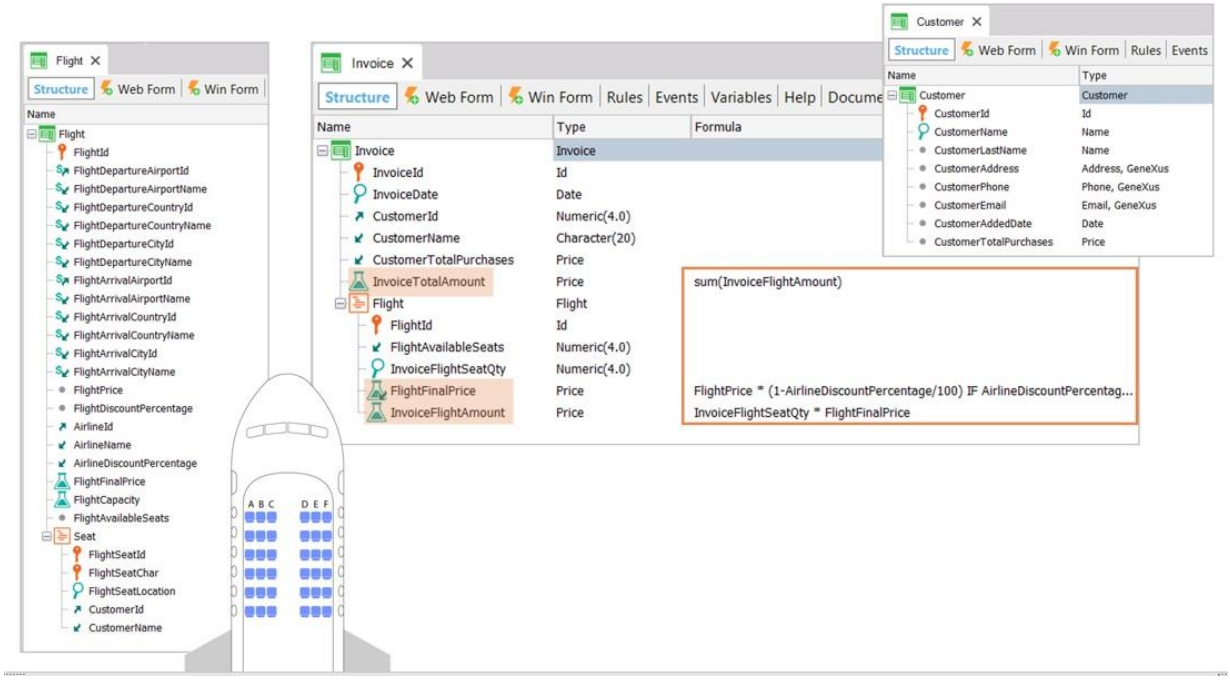
# Reality




Observemos que hemos agregado a la transacción Flight el atributo FlightAvailableSeats, que se usará para registrar los asientos disponibles de cada vuelo, que se irán decrementando cada vez que se realiza una factura para un cliente que compra una cantidad de asientos en el vuelo.

Lo hemos agregado, entonces, en este nivel. Será un atributo inferido.

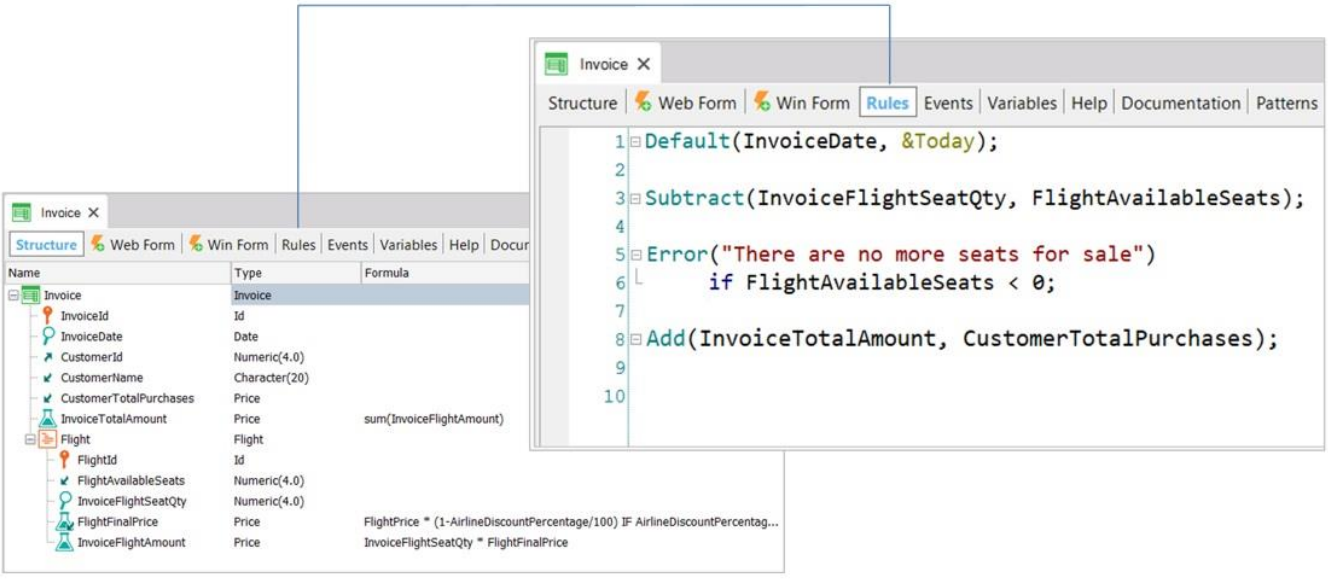
# Reality




Hemos agregado también a la transacción Customer el atributo CustomerTotalPurchases, para registrar el total comprado por el cliente por concepto de pasajes de vuelos. También lo agregamos a nuestra transacción como atributo inferido.

Los atributos InvoiceTotalAmount, FlightFinalPrice e InvoiceFlightAmount de la estructura de Invoice han sido definidos como fórmulas.

# Invoice rules

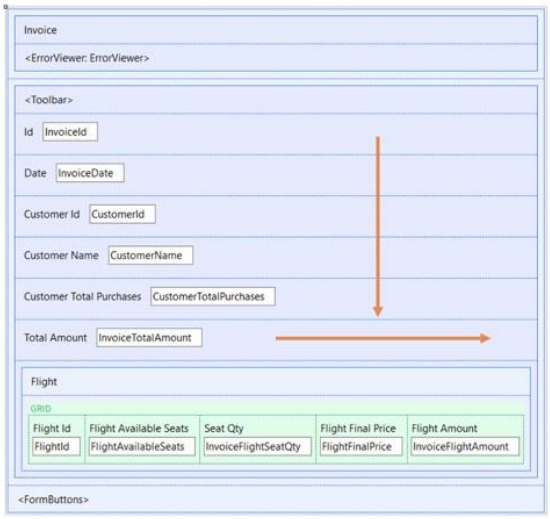



En la transacción Invoice, hemos definido las siguientes reglas para especificar su comportamiento:

La regla Default, que inicializa el atributo de la fecha de la factura con la fecha de hoy; la regla Subtract, que decrementa la cantidad de asientos disponibles del vuelo según la cantidad de asientos comprados en la factura -observemos que estará decrementando un atributo de la tabla Flight: FlightAvailableSeats es, aquí, inferido-; la regla Error, que muestra un mensaje de error si el vuelo ya no cuenta con asientos disponibles; y la regla Add, que suma el total de la factura al total de compras realizadas por el cliente -otra vez, atributo presente en una tabla de la extendida, 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;
```



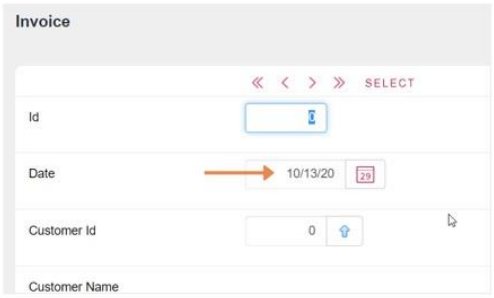

Resumiendo, tenemos todas estas reglas y fórmulas definidas en la transacción Invoice:

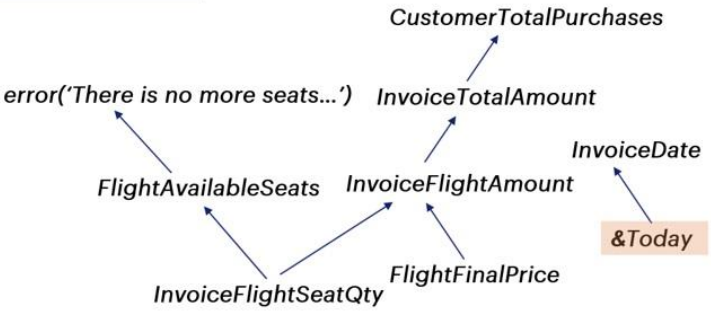
La gran pregunta es ¿cómo sabe GeneXus en qué orden debe dispararlas, y cuándo sí y cuándo no?

Por supuesto, hay un primer orden natural que es el que corresponde al ordenamiento de los atributos en la pantalla (de arriba hacia abajo y de izquierda a derecha).

## 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;
```

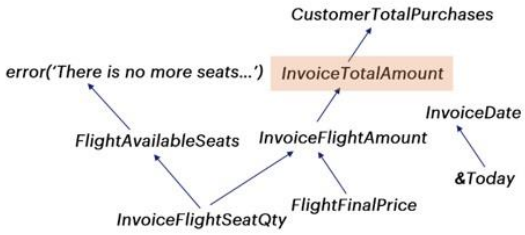


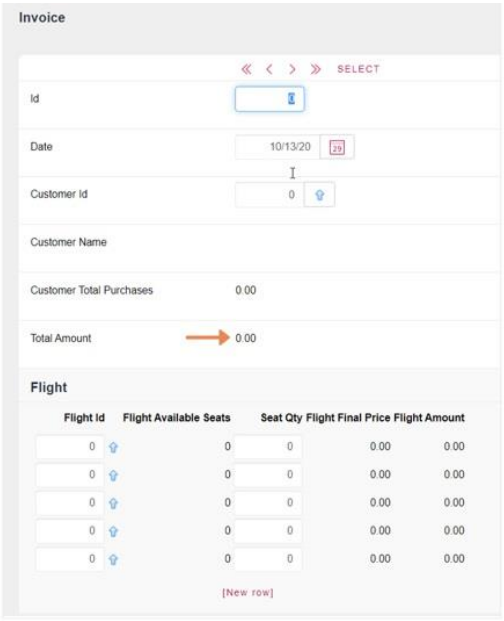




Una regla o fórmula se dispara ni bien se cuenta con la información que necesita. Por ejemplo, la regla Default solo necesita el valor de la variable &Today y saber que se está en modo Insert. Por eso apenas abrimos la pantalla en modo Insert ya vemos el valor en el campo, aún cuando ni siquiera hemos llegado a él (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;
```

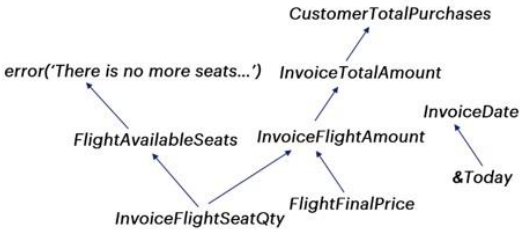



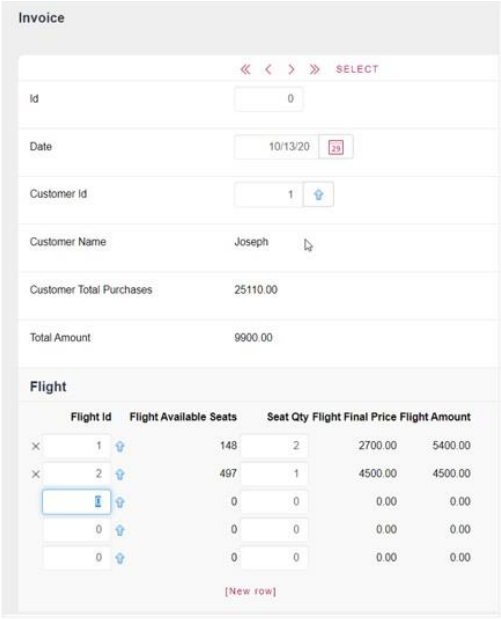




Pensemos qué pasa con la fórmula del primer nivel: InvoiceTotalAmount, que es un sum de un atributo del segundo nivel. Como la fórmula Sum necesita solamente el atributo InvoiceFlightAmount, se va a disparar para el cabezal incluso antes de haber podido ingresar la primera línea, dando 0.

## 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;
```

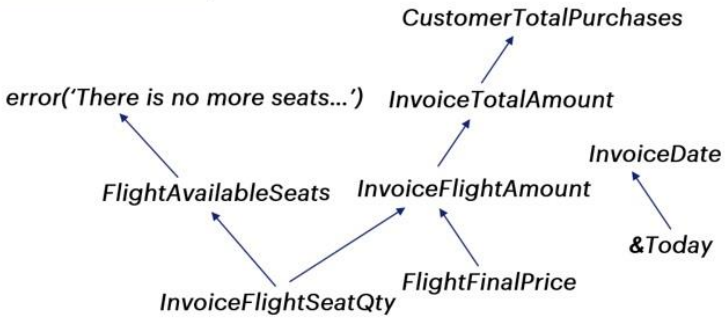




Pero luego, a medida que vayamos ingresando líneas, por cada una se va a redisparar. Pero, ¿y qué pasa si entramos en modo Update a la transacción y, por ejemplo, modificamos algo de una línea que no modifique para nada el InvoiceFlightAmount? (en este caso no tenemos ningún atributo para modificar que no modifique ese valor, porque los únicos dos atributos editables son el id de la línea, que no se puede cambiar por ser parte de la clave primaria, y luego el InvoiceFlightSeatQty, que sí modifica el valor de InvoiceFlightAmount, pero imaginemos que lo hubiera, por ejemplo, que deba indicarse si alguno de los pasajeros es diabético, y en el línea en cuestión teníamos que sí, y ahora queremos cambiarlo por no). Obviamente, en ese caso, la fórmula del cabezal no se recalcularía.

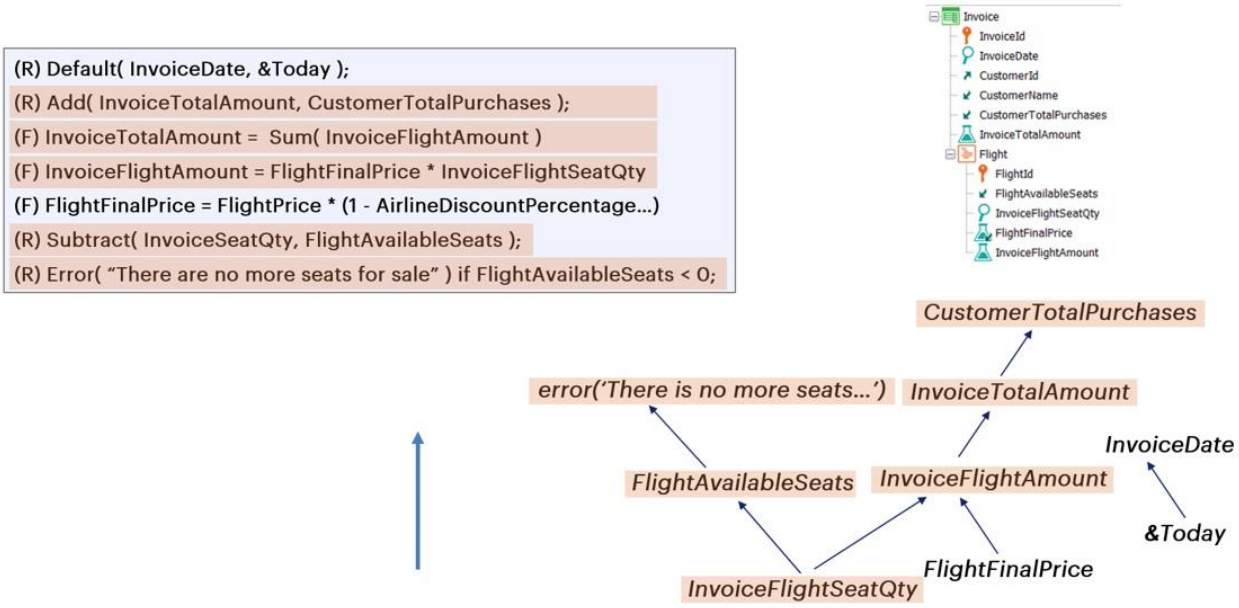
## Evaluation tree

```
(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;
```




Este "obviamente" también lo es para GeneXus, que lo que hace internamente es extraer las dependencias existentes entre lugares que asumen los controles en la pantalla, las reglas y las fórmulas, para construir un árbol de dependencias (conocido como árbol de evaluación) que es el que determinará qué reglas y fórmulas deberán redispararse ante cambios en atributos de la pantalla. En nuestro ejemplo será así:

## Evaluation tree

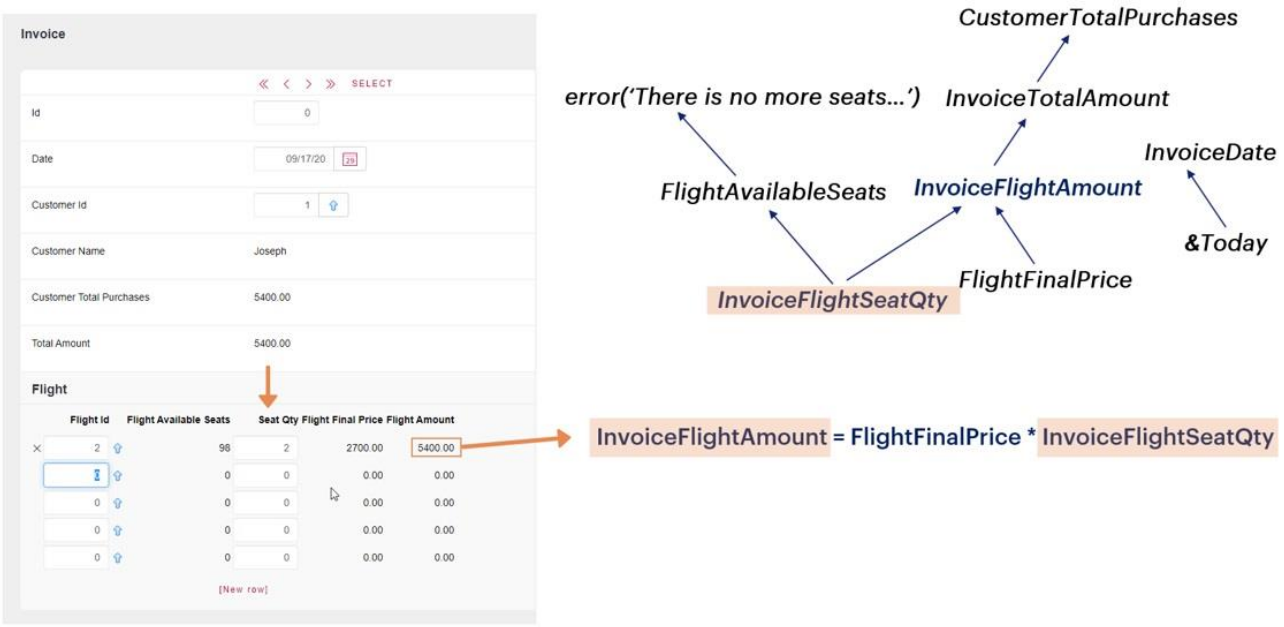



Por ejemplo, antendamos lo que sucede con el atributo InvoiceFlightSeatQty. De él depende la actualización con subtract del atributo FlightAvailableSeat, asientos disponibles del vuelo, del que depende, a su vez, el disparo de la regla error. Por ese motivo la condición de la regla error debe escribirse sabiendo que siempre, por esta dependencia, ya se habrá ejecutado el subtract, y es por ello que se coloca la condición 'menor que cero'. Recuerde que, como ya hemos estudiado, de quedar asientos negativos, se deshará todo lo hecho en el árbol que condujo a producir este error.

A su vez, también de InvoiceFlightSeatQty depende la actualización de la fórmula InvoiceFlightAmount, de la que, por su parte, depende la del cabezal, InvoiceTotalAmount, de la que depende la actualización del total de compras del cliente.

Podemos imaginar que el árbol se ejecuta de abajo hacia arriba, es decir que cada vez que cambia el valor de un atributo, se ejecutan todas las reglas y fórmulas que dependen de ese atributo (y que en el árbol se encuentran hacia arriba).

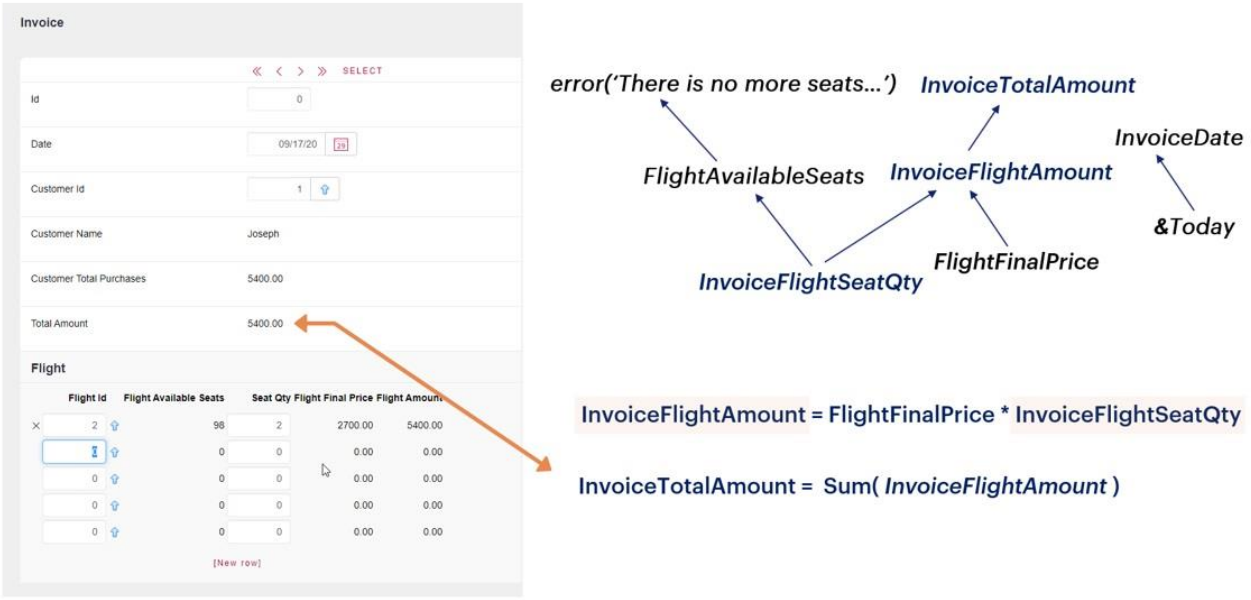
## Evaluation tree




Sigamos el ejemplo anterior:

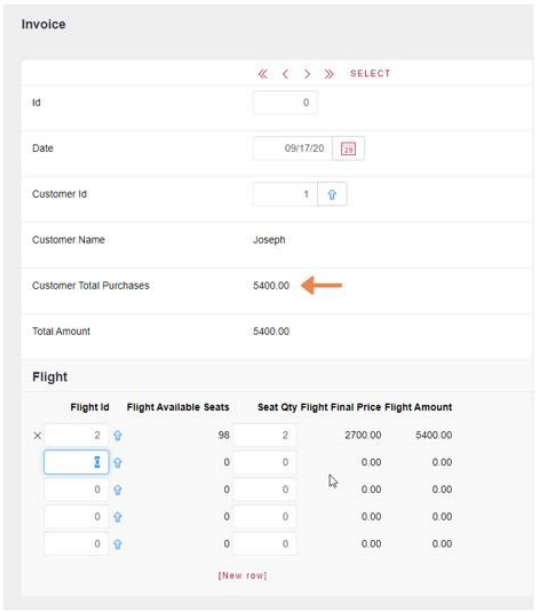
Si cambia la cantidad de asientos en una línea de una factura (InvoiceFlightSeatQty), como este atributo interviene en la fórmula que calcula el importe del vuelo (InvoiceFlightAmount), dicha fórmula se redisparará.

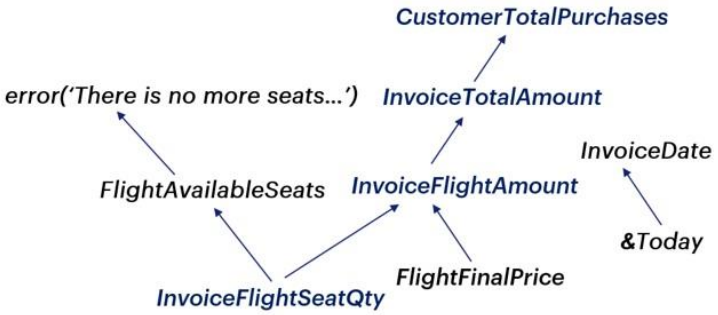
# Evaluation tree




Al redispararse, también deberá hacerlo la fórmula correspondiente al total de la factura (InvoiceTotalAmount).

Evaluation tree




InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty

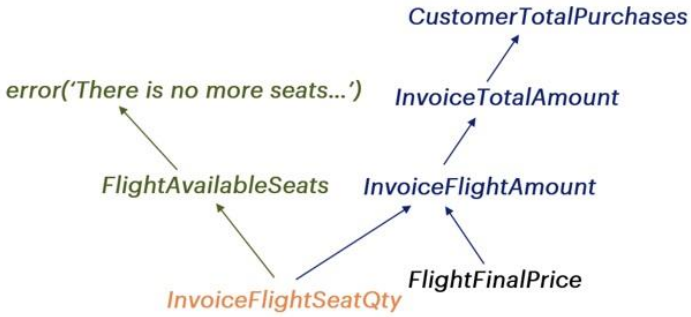
InvoiceTotalAmount = Sum( *InvoiceFlightAmount* )

Add( *InvoiceTotalAmount*, CustomerTotalPurchases );

Por último, por cambiar el total también se tendrá que disparar la regla Add(InvoiceTotalAmount, CustomerTotalPurchases) ya que deberá actualizarse el total de compras del cliente.

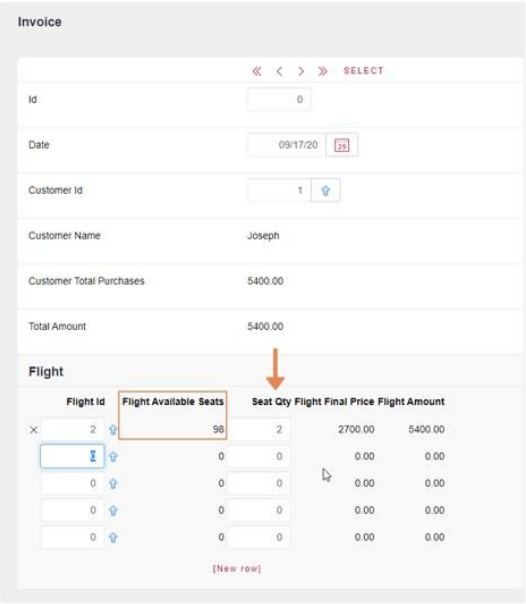
## Evaluation tree

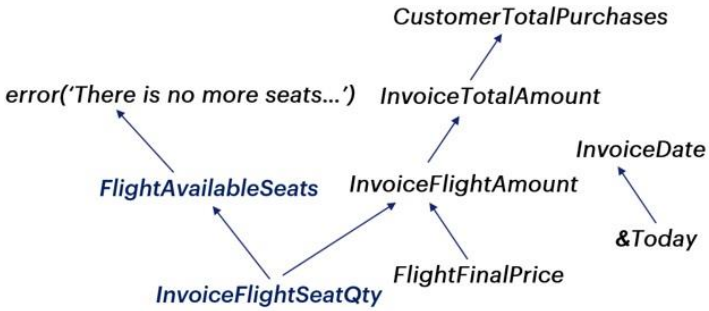
```
(R) Default( InvoiceDate, &Today );
(R) Add( InvoiceTotalAmount, CustomerTotalPurchases );
(F) InvoiceTotalAmount =  Sum( InvoiceFlightAmount )
(F) InvoiceFlightAmount = FlightFinalPrice * InvoiceFlightSeatQty
(F) FlightFinalPrice = FlightPrice * (1 - AirlineDiscountPercentage...)
(R) Subtract( InvoiceSeatQty,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




Además de dispararse todas las fórmulas y reglas involucradas en la rama derecha del árbol desde el atributo InvoiceFlightSeatQty, también se dispararán las fórmulas y reglas involucradas en la rama izquierda.

## Evaluation tree



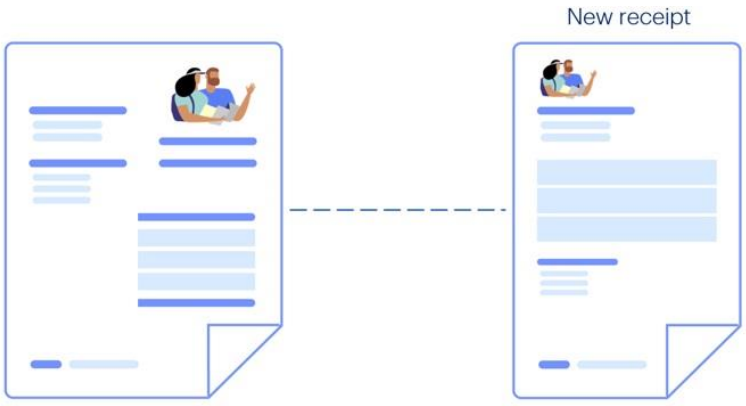

```
Subtract( InvoiceSeatQty, FlightAvailableSeats );

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

Como ya vimos, al cambiar el valor del atributo InvoiceFlightSeatQty, se redisparará también la regla Subtract(InvoiceFlightSeatQty, FlightAvailableSeats) que actualiza la cantidad de asientos disponibles en el vuelo (FlightAvailableSeats).

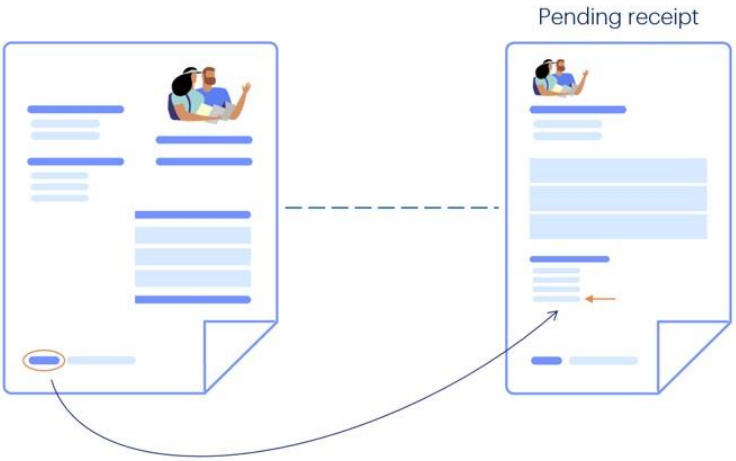
Y en consecuencia, por modificar esta regla, el valor del atributo FlightAvailableSeats se evaluará para saber si habrá que disparar la regla Error que indica que ya no hay asientos disponibles.

Si la condición para dispararse el error se satisface, todo lo hecho en el árbol desde el cambio en el atributo InvoiceFlightSeatQty será deshecho automáticamente, y los datos de la base de datos volverán al estado anterior a la ejecución de la regla error.

Reality



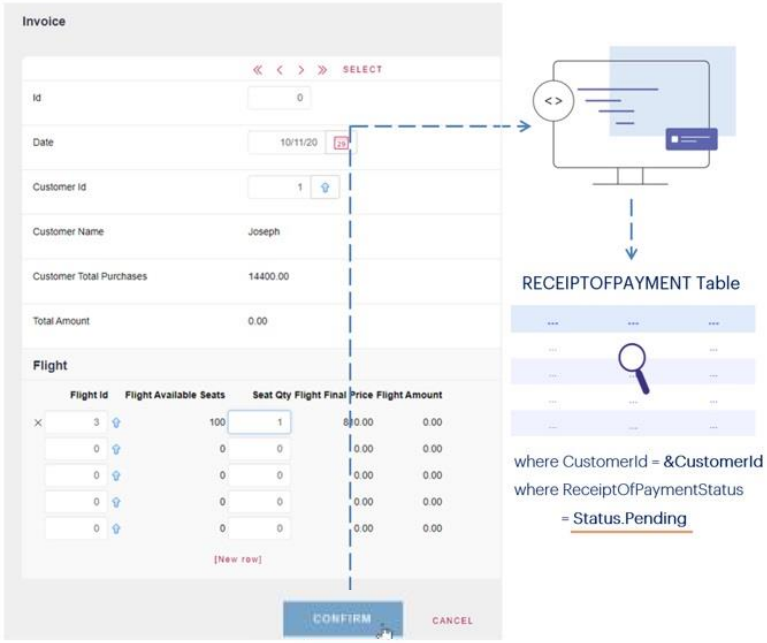

Veamos ahora un ejemplo en el que una regla no siempre se dispara en el momento deseado. Supongamos que inmediatamente de realizarle una factura a un cliente por una cantidad de asientos de vuelos comprados, queremos generarle un recibo de pago. Si el cliente está al día con los pagos, entonces le generamos un recibo nuevo,




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

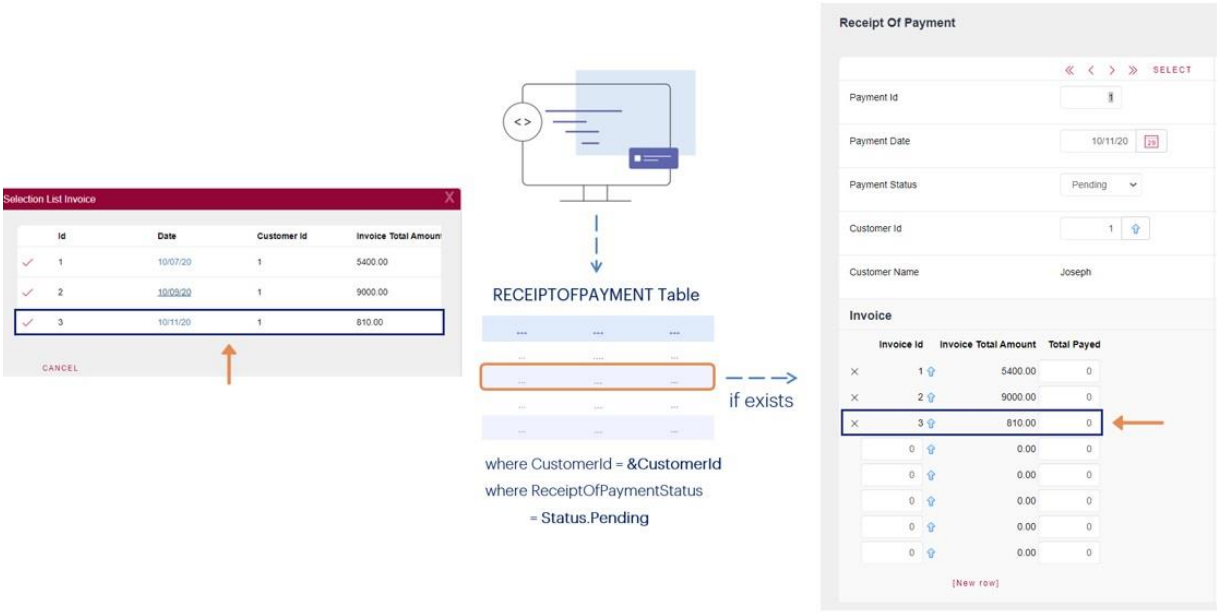
pero si no lo está, entonces agregamos el importe de esta factura al recibo pendiente anterior. Hemos creado una transacción ReceiptOfPayment. La misma se compone por el identificador del recibo, la fecha, el estado (que es un dominio enumerado con los valores pendiente y completado), el cliente, y un segundo nivel para registrar las facturas para las cuales se emite el recibo de pago.

# Reality



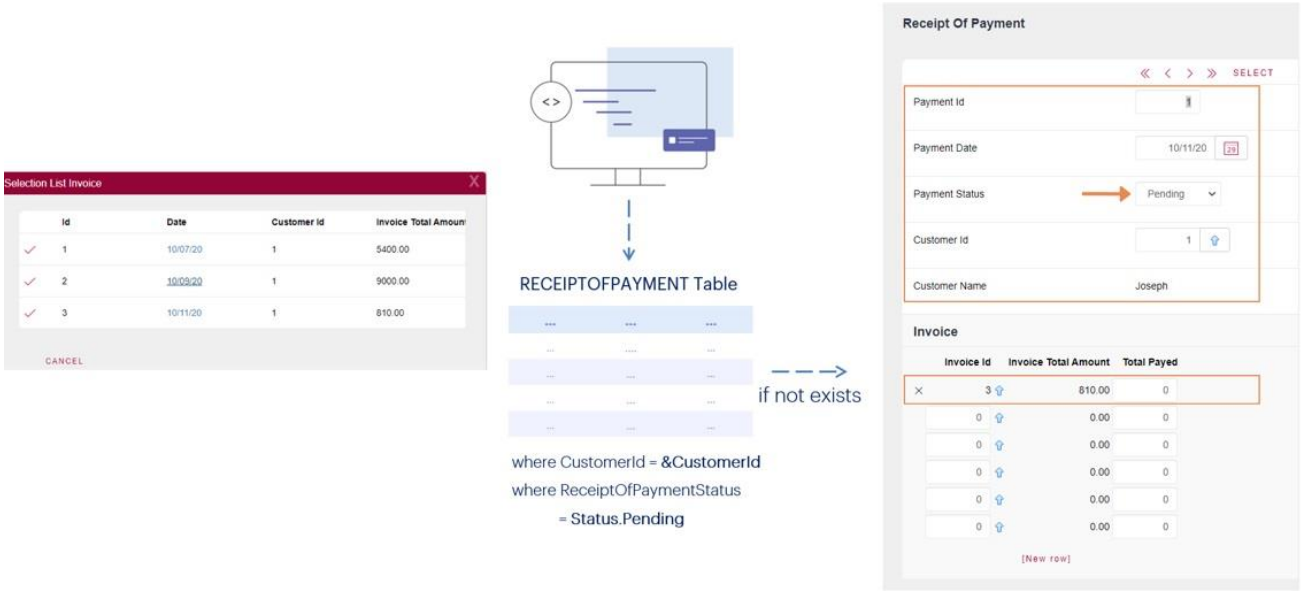

Entonces, cuando se ingresa una nueva factura, se busca por programa si ya existe un recibo de ese cliente en estado Pending.

# Reality

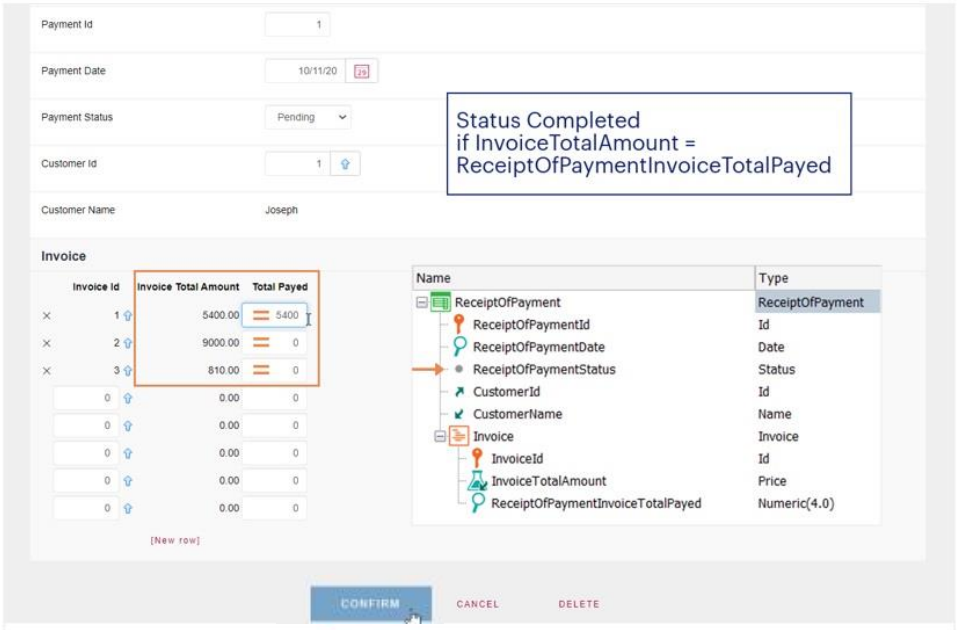



Si existe, se agrega una nueva línea con esta factura.

# Reality

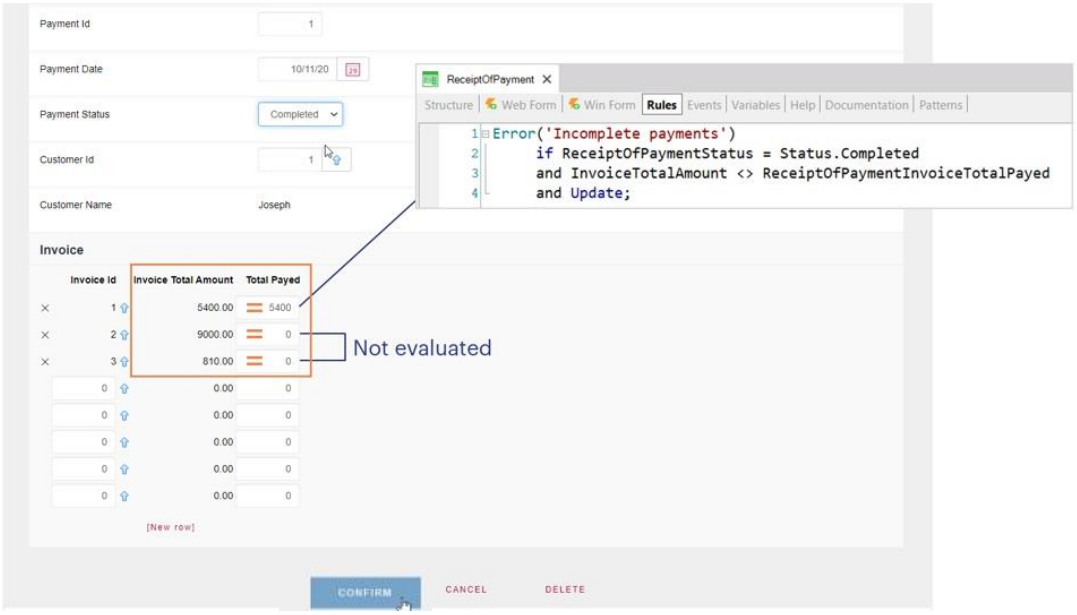



Si no existe, se crea el cabezal y la línea, y el cabezal queda en estado Pending.




Luego viene el cliente a pagar, por lo que el empleado abre la transacción y modifica el atributo ReceiptOfPaymentInvoiceTotalPayed para las facturas que el cliente desee pagar.

El estado del recibo (ReceiptOfPaymentStatus) sólo podrá pasarse a Completed si coinciden para todas las líneas el valor de InvoiceTotalAmount con el de ReceiptOfPaymentInvoiceTotalPayed. Supongamos que esa modificación es realizada por el usuario, es decir, que es él quien cambia el valor de ReceiptOfPaymentStatus, por lo que debe corroborarse que no se permita cambiar a Completed si quedó alguna factura impaga o mal paga.



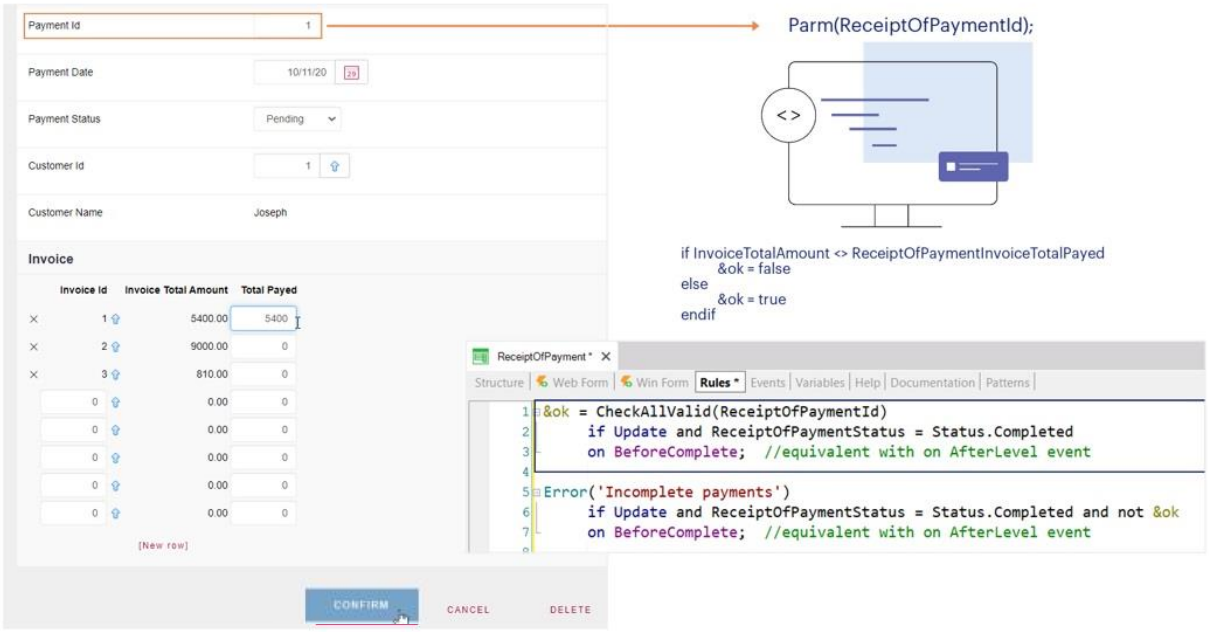

Podríamos pensar en colocar la siguiente regla de error, condicionándola a dispararse cuando estamos actualizando un recibo, el estado es Completed, y no coincide el valor del atributo que indica lo que debe pagarse y el que indica lo que se pagó para una línea:

Sin embargo... ¿cuándo se disparará esta regla?

Claramente si ingresamos a la transacción, cambiamos el estado a Completed y para una línea ingresamos un valor diferente del esperado, se disparará. Pero, ¿qué pasaría si para otra línea que tiene en 0 la cantidad pagada ni siquiera entramos? ¿Se disparará la regla?

La respuesta es no. Cuando ejecutamos una transacción en modo Update podemos no querer modificar el cabezal y cambiar solamente una línea. ¿En ese caso, qué se disparará? El cabezal se actualizará siempre, y luego solo la línea modificada. Para ella se disparará todo de acuerdo al árbol de evaluación.

Reality




¿Cómo resolveríamos, entonces este caso?

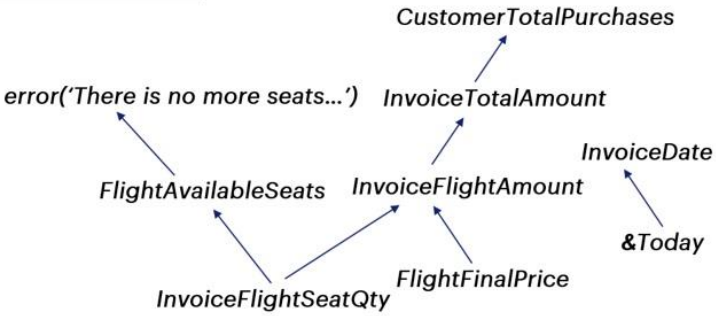
Una solución es llamar a un procedimiento al que le enviamos el id del recibo, una vez que dimos tiempo a modificar todas las líneas, y una vez que estas modificaciones se realizaron en la base de datos, pero antes del commit, así podemos deshacer. Ese procedimiento recorre TODAS las líneas y se asegura de que no quede ninguna con valor diferente para los atributos que nos interesan.

Si bien para la regla de Error podría parecer que no necesitamos condicionar al evento BeforeComplete dado que la variable &ok para evaluarse ya lo está, en verdad sí es necesario. Si no condicionáramos el error a exactamente el mismo evento, rompería las dependencias entre ambas reglas.

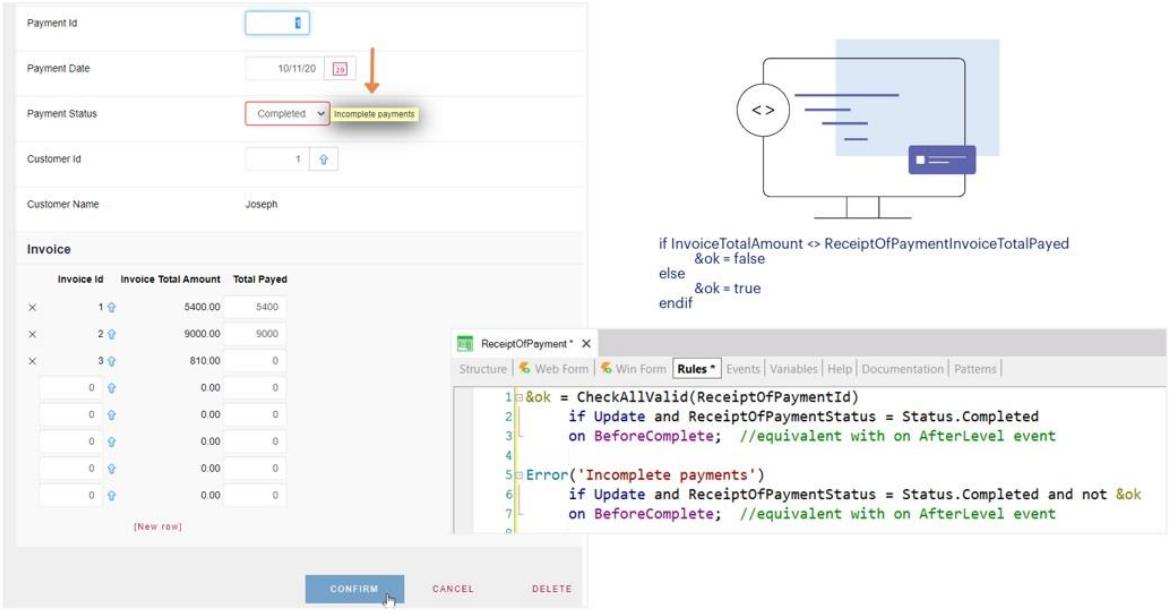
Cada evento de disparo tiene su propio árbol de evaluación, lo que significa que si condicionamos muchas reglas al mismo evento, las ordenará al momento de ocurrir el evento de acuerdo a sus dependencias, tal y como vimos antes.

## 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;
```

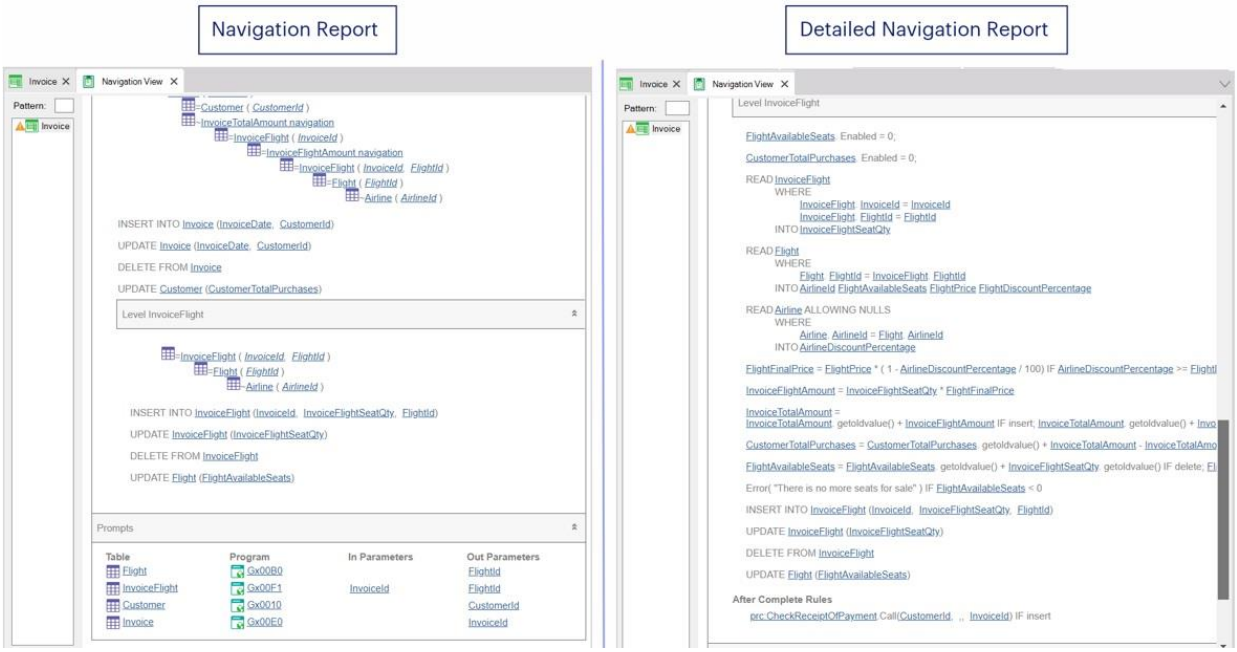



Concluyendo, las reglas y fórmulas que se definen en una transacción suelen estar interrelacionadas y GeneXus determina las dependencias entre ellas, así como su orden de evaluación.




En ocasiones, el árbol de evaluación no determina el orden de ejecución que deseamos: un claro ejemplo de ello es el que acabamos de ver, donde debimos retrasar el momento de disparo del procedimiento que chequea sobre los registros de las líneas y el error subsiquiente.

# Detailed navigation

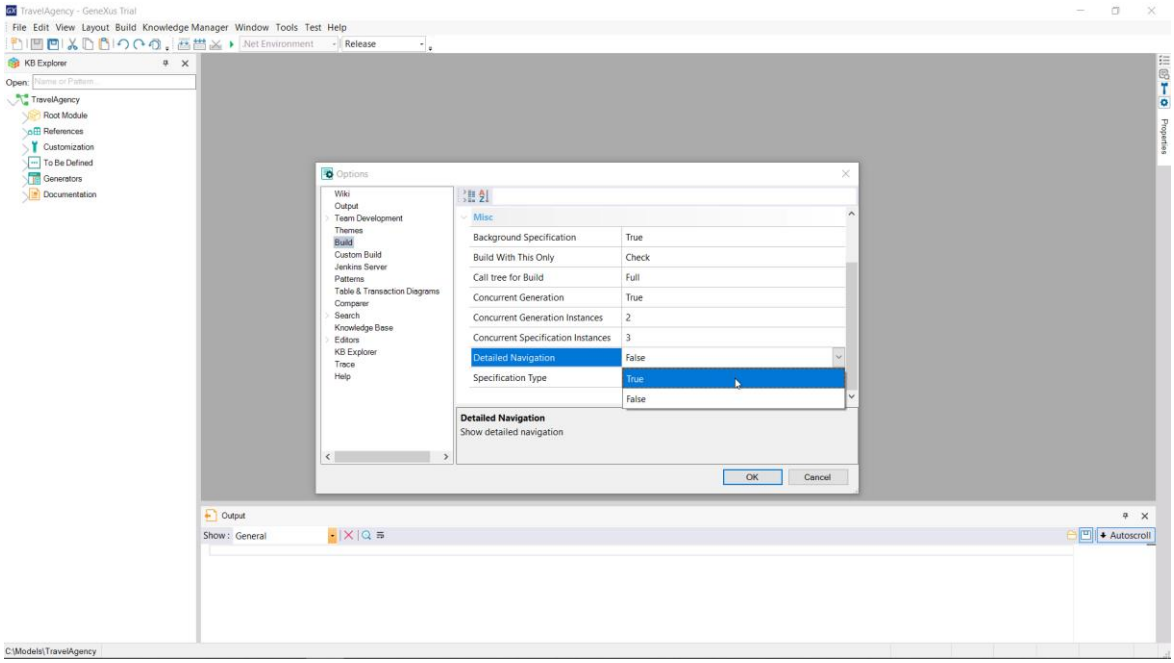



Si desea ver con más nivel de detalle el orden de las evaluaciones que dispara GeneXus, puede utilizar el listado de navegación detallado.

Aquí vemos la diferencia entre ambos... veamos, por ejemplo, que en la navegación detallada se nos muestran las reglas y los momentos en que se dispararán, lo que no sucede en el otro caso.

La navegación detallada puede ser útil en los casos en los que necesitamos entender bien dónde se está disparando una fórmula o regla, pero para lo normal lleva más tiempo de especificación, por lo que muchas veces no lo necesitamos.

# Detailed navigation




Para habilitarlo, debe ir al menú Tools > Options, y dentro de la categoría Build activar la propiedad Detailed Navigation.

GeneXus™

training.genexus.com
wiki.genexus.com