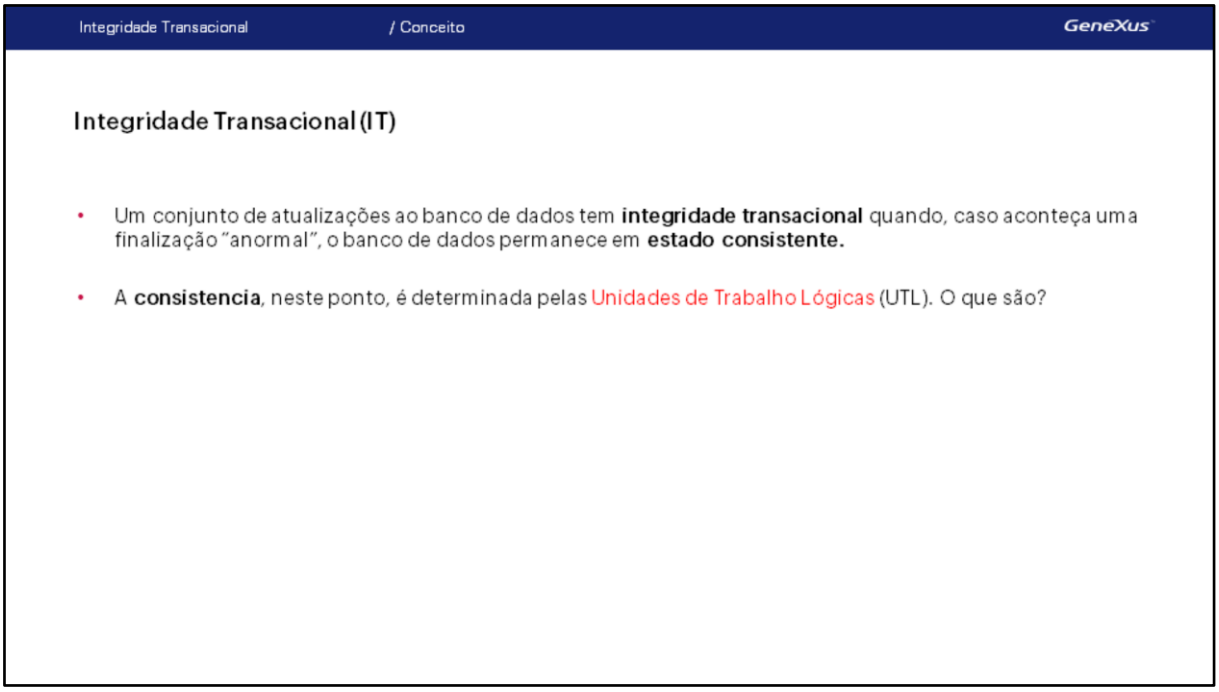
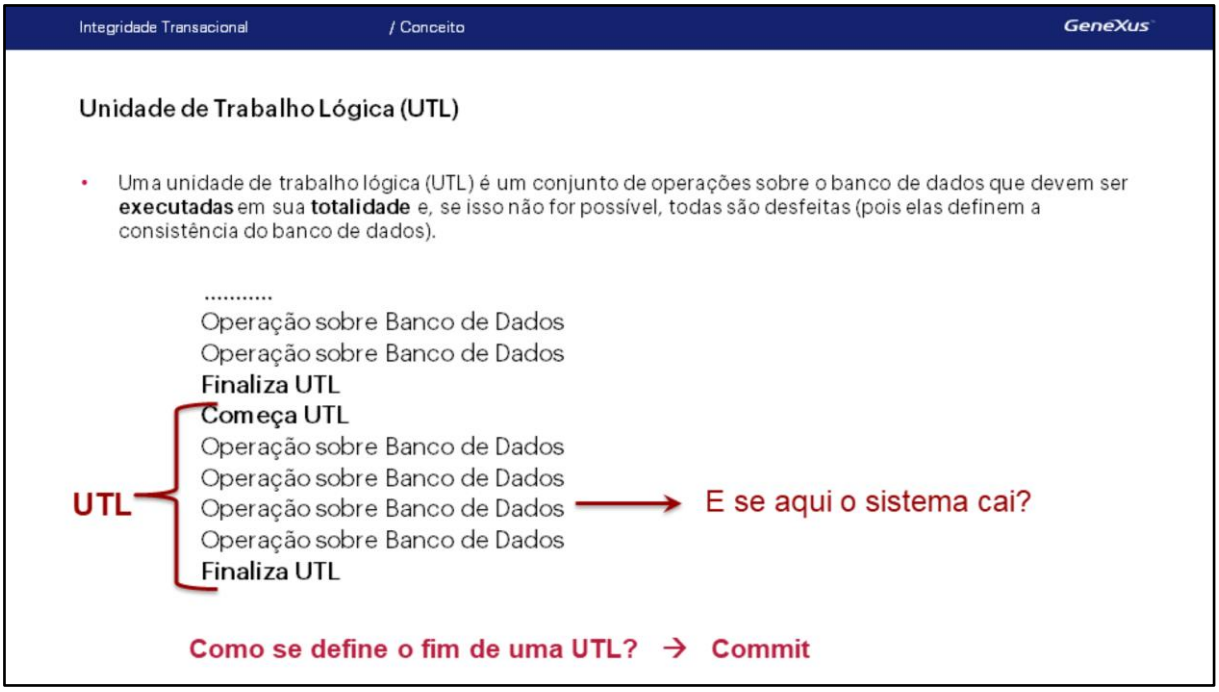
Integridade Transacional
Banco de dados consistente depois de falhas

GeneXus 16

Conceito



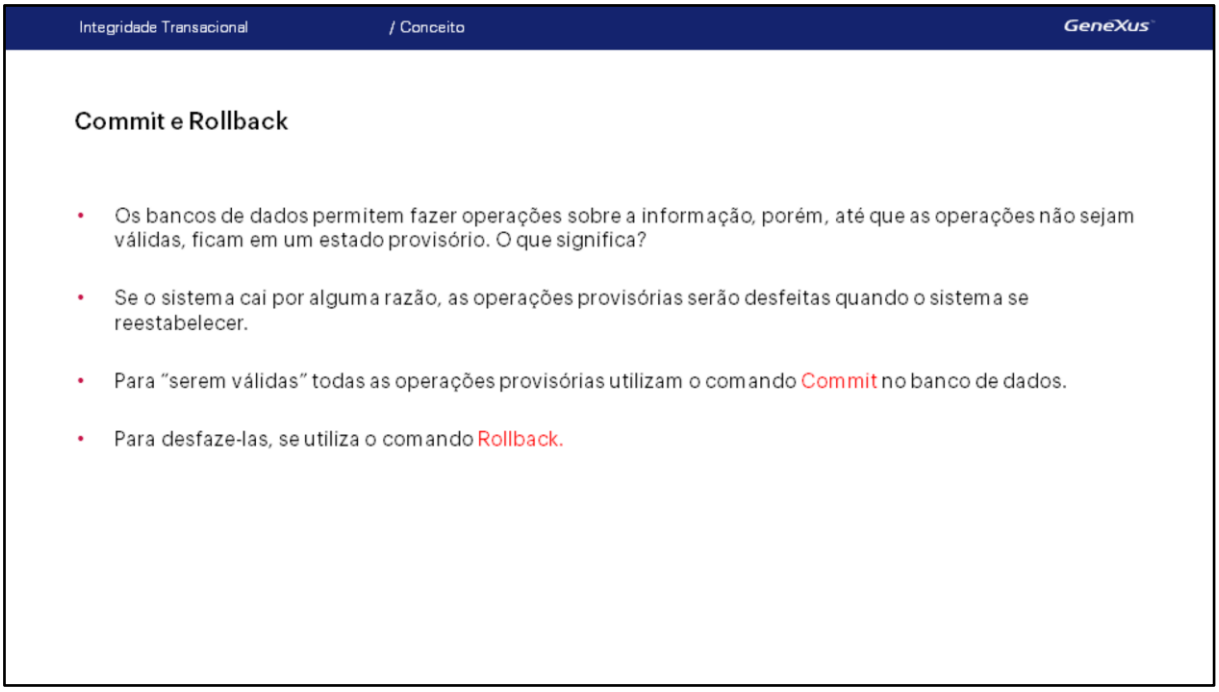
Muitos gerenciadores de bancos de dados (DBMSs) contam com sistemas de recuperação contra falhas, que permitem deixar o banco de dados num estado consistente quando ocorrem imprevistos tais como apagões ou quedas do sistema.



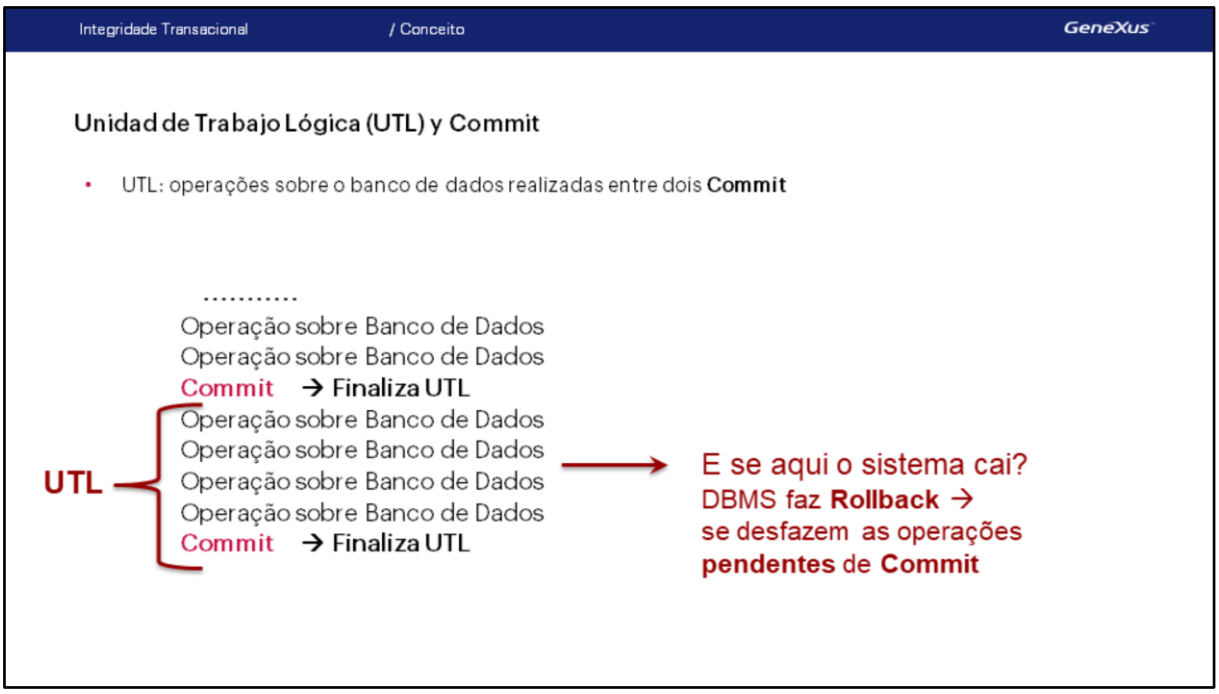
Os gerenciadores de bancos de dados (DBMSs) que oferecem integridade transacional permitem estabelecer unidades de trabalho lógicas (UTL), que correspondem nem mais nem menos que ao conceito de “transações” do banco de dados.

No exemplo, mostramos uma UTL que consiste de quatro operações no banco de dados. Vamos supor que as duas primeiras foram realizadas com sucesso, mas antes de executar a terceira o sistema cai. Como a UTL não foi finalizada totalmente, as duas operações que haviam sido concluidas deverão ser desfeitas. Se isso não acontecer, como são as UTLs que definem os estados consistentes do banco de dados a nível lógico, este ficaria inconsistente.

Como se define uma UTL?



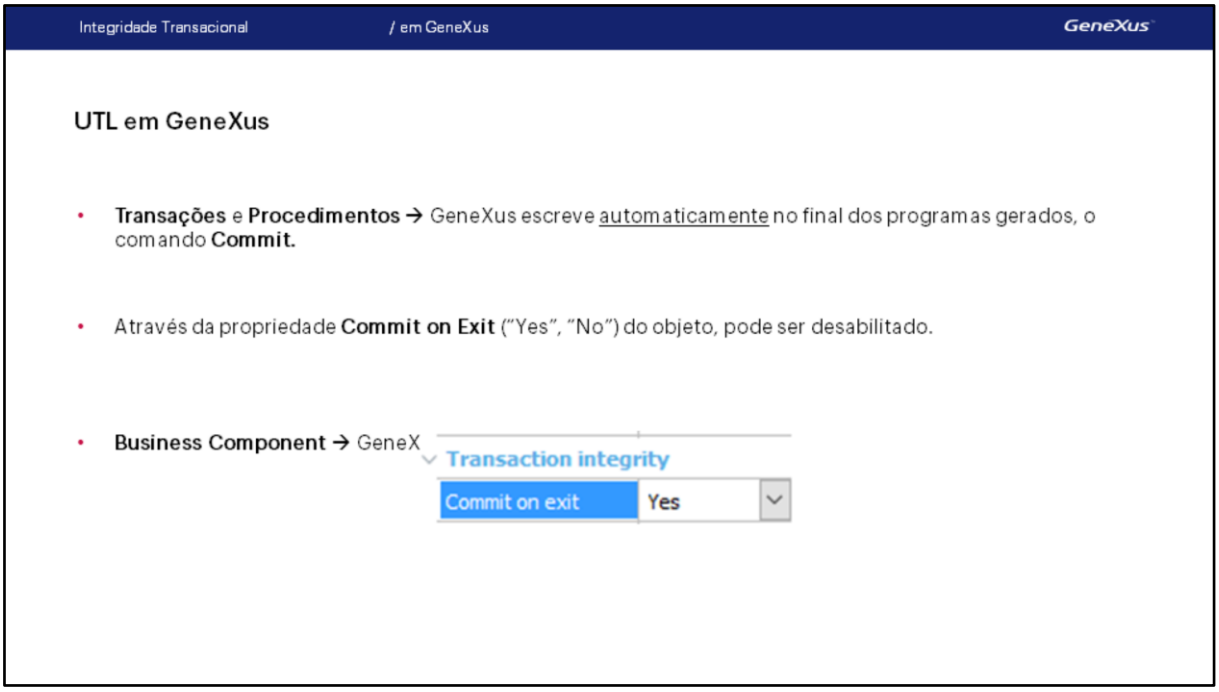
Quando o sistema cai e é reestabelecido, o DBMS realiza um Rollback para recuperar-se, mantendo o último estado consistente do banco de dados.



É o comando Commit que determina o fim de uma UTL. Dessa forma, uma UTL é definida pelas operações realizadas entre dois commits.

Se o sistema sofre uma queda onde está indicado na imagem, as duas operações realizadas depois do último Commit (que são as operações pendentes de Commit) são desfeitas com o Rollback automático que o DBMS realiza ao recuperar-se da falha.

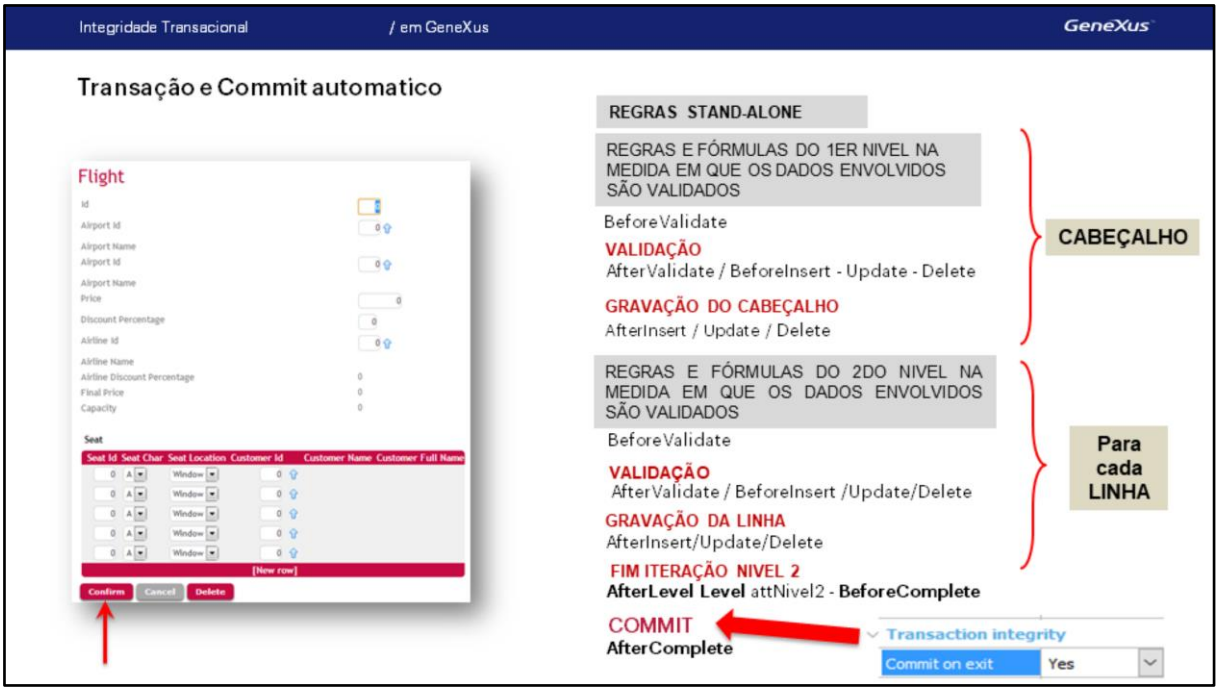
Integridade Transacional em GeneXus



As transações e os procedimentos são os objetos GeneXus **criados para atualizar** as informações do banco de dados. É por essa razão que GeneXus escreve o comando Commit quando gera os programas na linguagem configurada na KB. Onde?

- No objeto transação: no final de cada instância, imediatamente antes das regras com evento de disparo AfterComplete (quer dizer, depois de ter manipulado o cabeçalho e as linhas).
- No objeto procedimento: no final do Source.

Os Business Components, criados a partir das transações, não incluem o Commit, já que podem ser utilizados em qualquer objeto, e cabe ao desenvolvedor determinar o momento certo de "commitar".
Veremos isso em seguida.

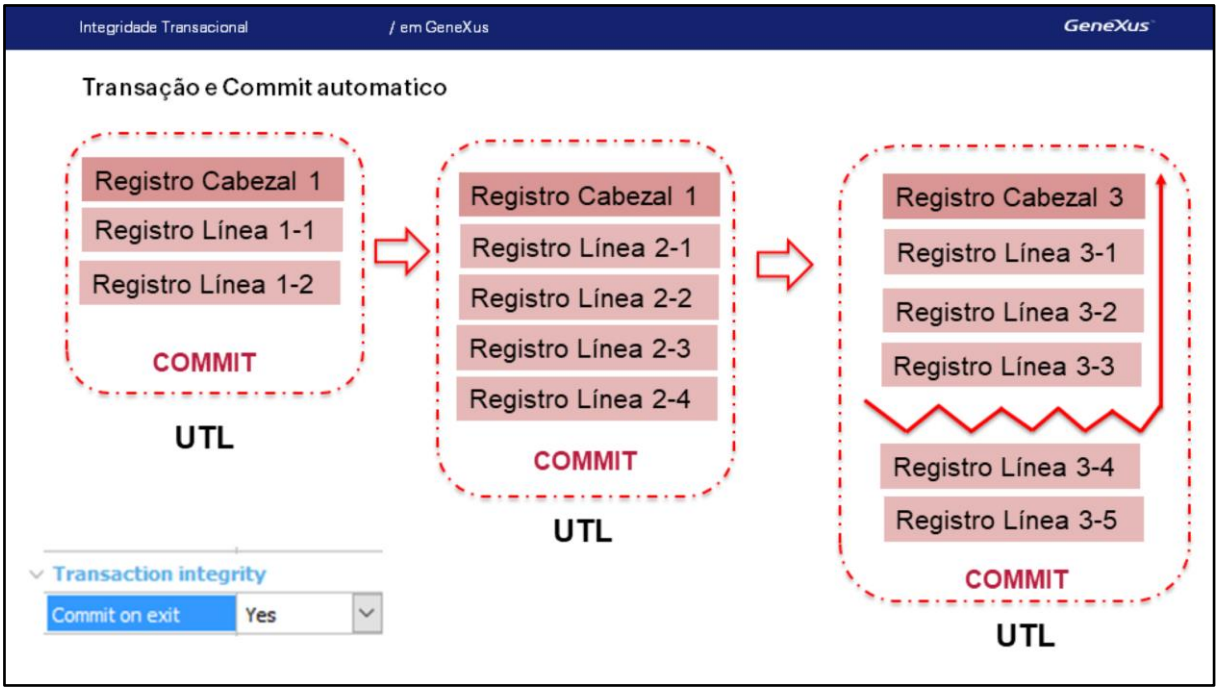


O usuário manipula o cabeçalho, as linhas da grid e pressiona o "Confirm". No servidor, segundo a árvore de avaliação, são disparadas as regras e fórmulas do primeiro nível. Depois, são disparadas as regras condicionadas ao evento BeforeValidate. Depois que as informações do cabeçalho são validadas, as regras condicionadas aos eventos AfterValidate são disparadas e, dependendo do modo, as condicionadas a BeforeInsert, BeforeUpdate ou BeforeDelete também. Depois disso, o cabeçalho é gravado e são disparadas as regras que estiverem condicionadas a AfterInsert, AfterUpdate ou AfterDelete, dependendo do modo.

Depois, para cada linha do grid:

- São executadas as regras que não tenham eventos definidos e que fazem referencia a atributos do segundo nível, como mostra a árvore de avaliação.
- São executadas as regras que tenham evento de disparo BeforeValidate, com atributos do segundo nível.
- É feita a validação da linha (ou seja, se ela é considerada válida).
- São executadas as regras que tenham evento de disparo AfterValidate ou, dependendo do modo, BeforeInsert, BeforeUpdate ou BeforeDelete.
- O registro validado, correspondente à linha do grid, é inserido/modificado/eliminado no banco de dados
- São executadas as regras que tenham evento de disparo AfterInsert, AfterUpdate ou AfterDelete, dependendo do modo da linha.

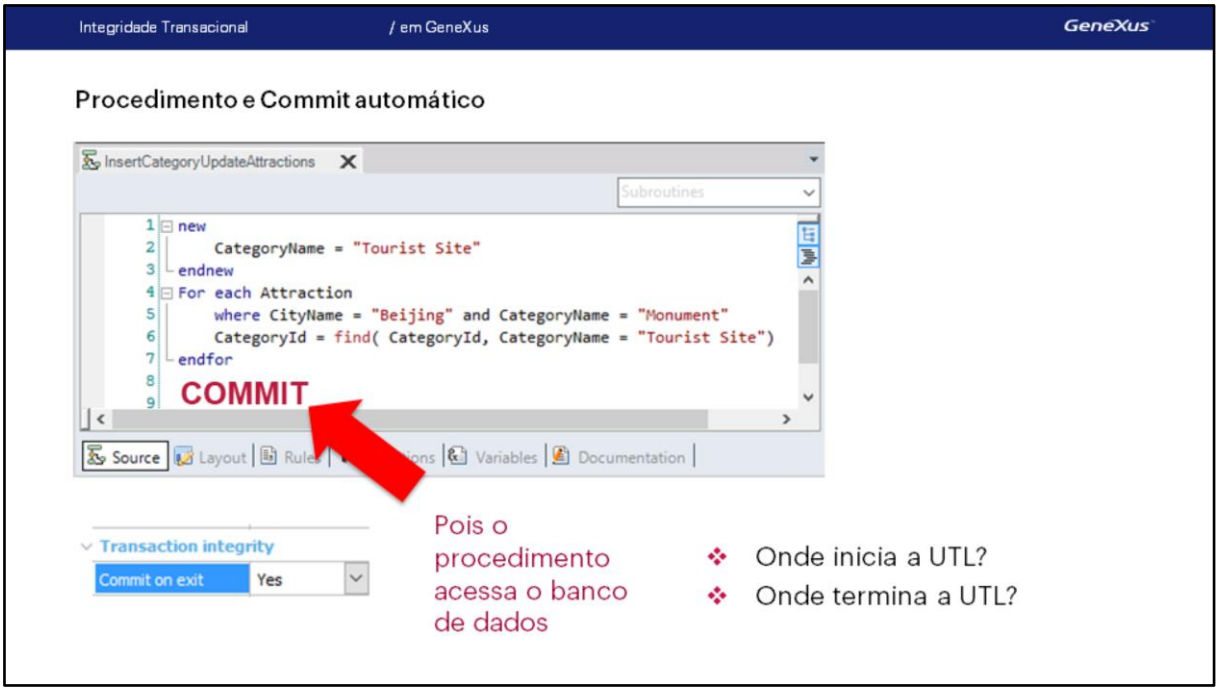
Depois de avaliar todas linha do grid, são executadas as regras que tenham evento de disparo After Level. Na sintaxe desse evento, é preciso definir um atributo do segundo nível. Se existir outro nível paralelo, todos esses passos são repetidos novamente para esse nível. Quando terminar com o último nível, são executadas as regras que tenham evento de disparo BeforeComplete. Depois disso é que GeneXus coloca o comando Commit de forma automatica. Portanto, nesse momento, o Commit é executado. Quer dizer, **a informação do cabeçalho e das linhas é commitada**. Em seguida, são disparadas as regras que estiverem condicionadas ao evento AfterComplete.



Vamos supor que o usuário precisa cadastrar 3 faturas no sistema. Cadastra a primeira, a segunda, e quando confirma a terceira, enquanto o sistema está processando a terceira linha, depois de tê-la gravado, acontece uma falha. Sabemos que o banco de dados irá recuperar-se, mas em que estado ele ficará?

Como o DBMS fará um Rollback, todas as operações que não foram “commitadas” serão desfeitas. Em nosso caso, serão eliminados os registros correspondentes ao cabeçalho e as três linhas da fatura 3.

Observem que, se tivéssemos desabilitado o commit automático da transação Invoice (propriedade “Commit on exit” = “No”), nenhum dos registros inseridos (nem os da fatura 1 e 2, e obviamente, os da 3) ficará registrado no banco de dados. Nste caso, todas as operações formaram uma UTL, enquanto que, se deixarmos o valor da propriedade Commit on Exit como “Yes” por padrão, cada cabeçalho com suas linhas formará uma UTL diferente.



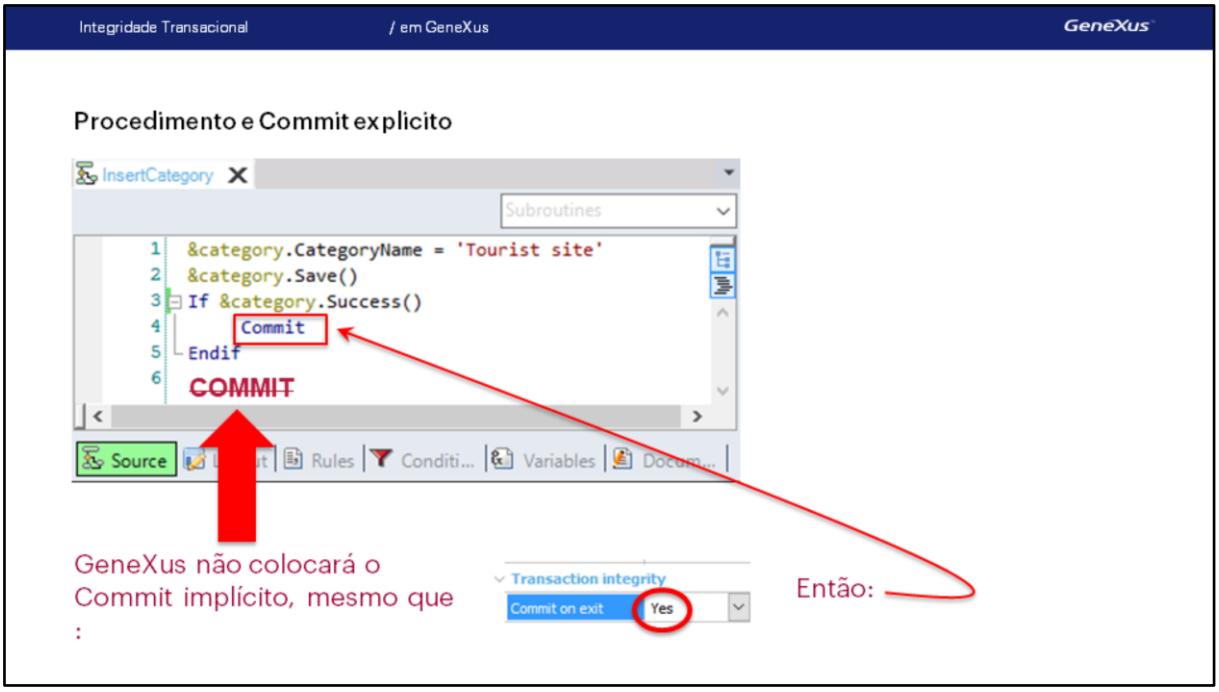
Em todo procedimento que acessa o banco de dados, GeneXus acrescenta automaticamente un Commit (a menos que se indique o contrario, através da propriedade Commit on exit).

Como os procedimientos são usados para outras coisas além de atualizar o banco de dados (por exemplo, para imprimir informações, ou realizar cálculos, ou simplemente consultar o banco de dados), GeneXus irá incluir o Commit automático no programa gerado caso entenda que esse procedimento tentar atualizar o banco de dados. Caso contrario, não o coloca, independente do valor da propriedade Commit on Exit.

Neste caso, em que estamos usando os comandos New para inserir uma categoria e o for each para atualizar o atributo CategoryId na tabela associada à transação Attraction, se a propriedade Commit on exit estiver definida como Yes, GeneXus escreverá o Commit automaticamente no programa gerado, no final do código. Com isso, depois que a nova categoria foi inserida na tabela CATEGORY, e, quando o terceiro monumento de Beijing estiver sendo modificado (substituindo a categoria atual pela nova), se o sistema cair, nenhuma das mudanças anteriores (nem a nova categoria, nem as alterações nos dois monumentos anteriores) ficará armazenada no banco de dados. Todas as operações deste procedimento serão parte da mesma UTL. Onde começa essa UTL?

Dependerá de quando foi executado o último Commit. Se, antes de chamar o procedimento, foi realizado algum Commit, então a UTL começa com o New deste procedimento. Caso contrário, todo este código será parte de uma UTL que começou antes. Onde? Imediatamente depois de onde foi executado o Commit anterior.

E onde termina a UTL? Se a propriedade Commit on Exit está definida como “Yes”, termina no final do procedimento. Se não, termina onde se encontra o próximo Commit (será preciso analisar o objeto que chamou o procedimento, e entender o código que vem depois dessa chamada).

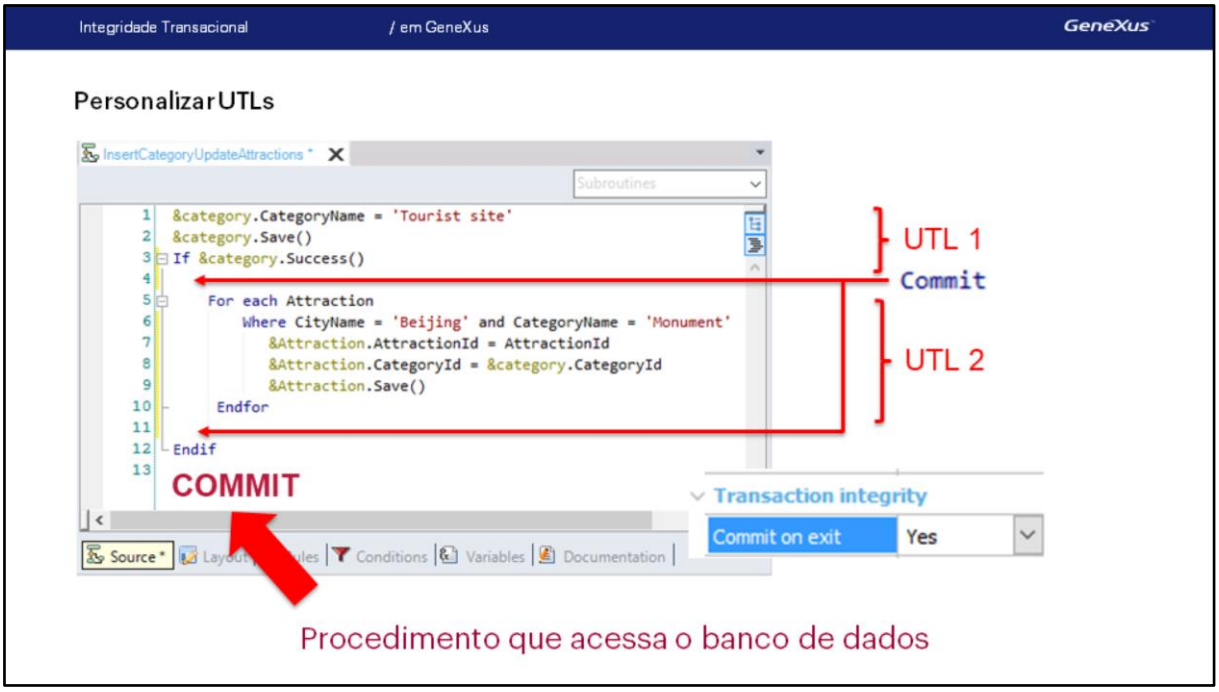


GeneXus reconhece que deve realizar um acesso ao banco de dados dentro de um procedimento, quando se utiliza New, For each para atualizar, Delete, ou os métodos e Delete() de um BC. Nesses casos, coloca o Commit implícito. Caso contrário, não identifica que houve acesso ao banco de dados. É por isso que, em um procedimento, quando somente escrevemos:

&BC.elemento1 = ...
&BC.elemento2 = ...
&BC.Save()

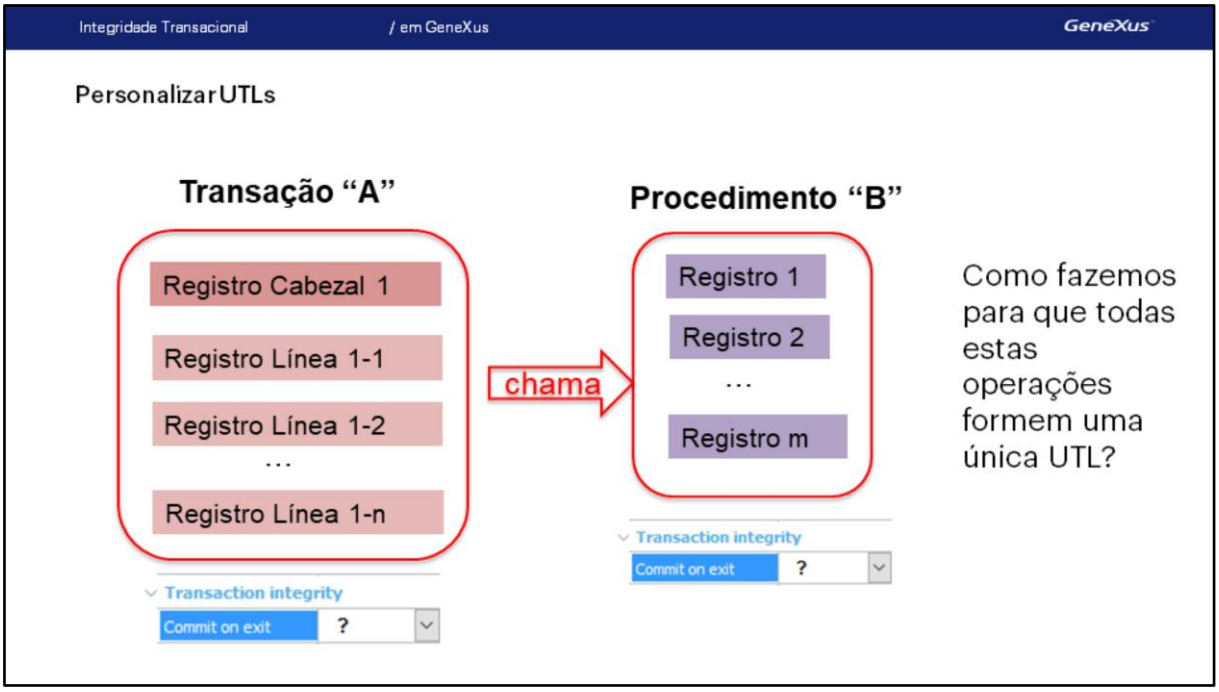
GeneXus não acrescenta o Commit. Seria nosso exemplo se não tivéssemos programado o Load da atração com a intensão de inserir uma nova categoria, como mostrado no Source acima. GeneXus não entende que estamos querendo fazer um Insert (trata o BC como se fosse um SDT qualquer) portanto que escrever o comando Commit explicitamente.

Se dentro do código do procedimento existir um New, For each que atualiza, ou um Load() (como no caso da página anterior) ou Delete, em qualquer um desses casos não seria preciso escrever explicitamente o Commit. Se, em nosso procedimento, temos somente o código anterior, aí sim teremos que acrescenta-lo de forma explícita.



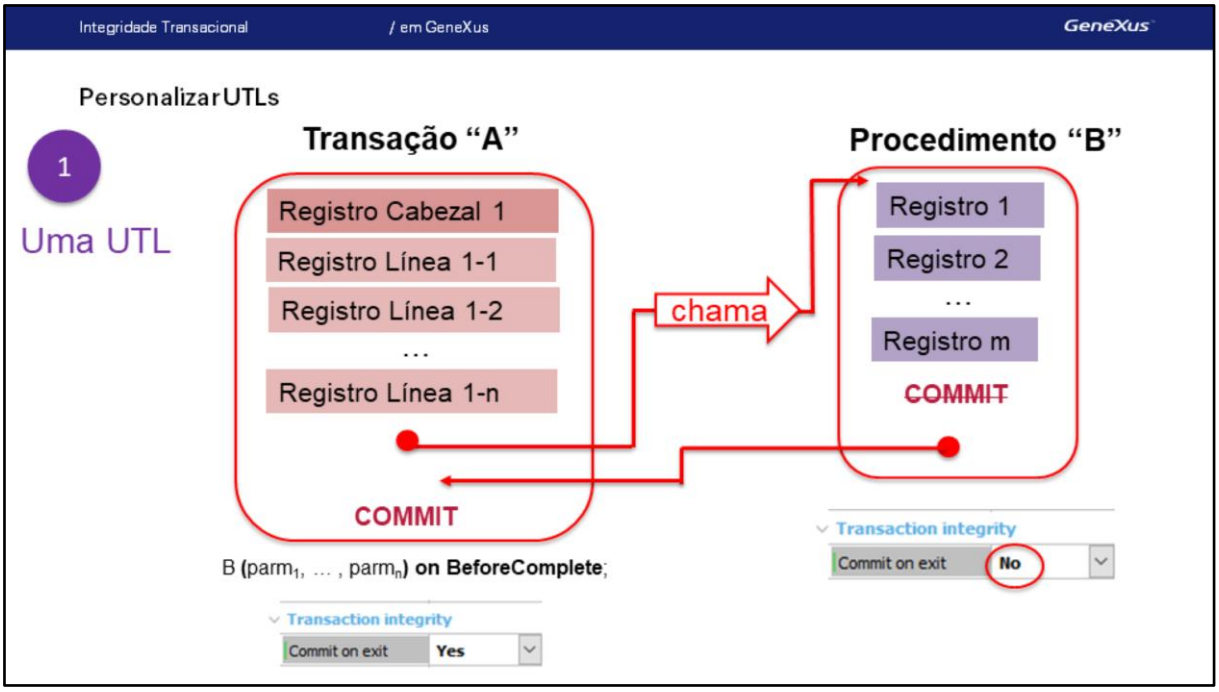
No exemplo que havíamos visto, Genexus colocava um commit automático no final (o Load faz com que se abra uma conexão com o banco de dados e ,portanto que acrescente um Commit no final). Porém, se quiséssemos que a gravação da categoria seja parte de uma UTL e as gravações das atrações sejam parte de outra, de modo que, se houver uma queda do sistema antes de terminar de modificar as atrações, a categoria fique registrada mas as atrações não, como fazemos?

Será através do uso do Commit. Teremos que escrever um Commit logo depois de ter inserido a categoria, e outro depois de inserir todas as atrações. Este segundo COMMIT não precisa ser escrito explicitamete se a propriedade Commit on exit do procedimento estiver como YES. De qualquer maneira, é uma boa prática escreve-lo, pois, se no futuro forem agregadas mais operações ao Source do procedimento, estas ficarão em outra UTL.



Se precisamos chamar um procedimiento "B" de dentro de uma transação "A", para realizar operações no banco de dados, de modo que as atualizações do registro do cabeçalho da transação mais todas as linhas, juntamente com todos os registros do procedimento formem uma única UTL (e, portanto, se acontecer uma queda do sistema, todas as alterações sejam desfeitas), como devemos programar?

Temos várias possibilidades de faze-lo.

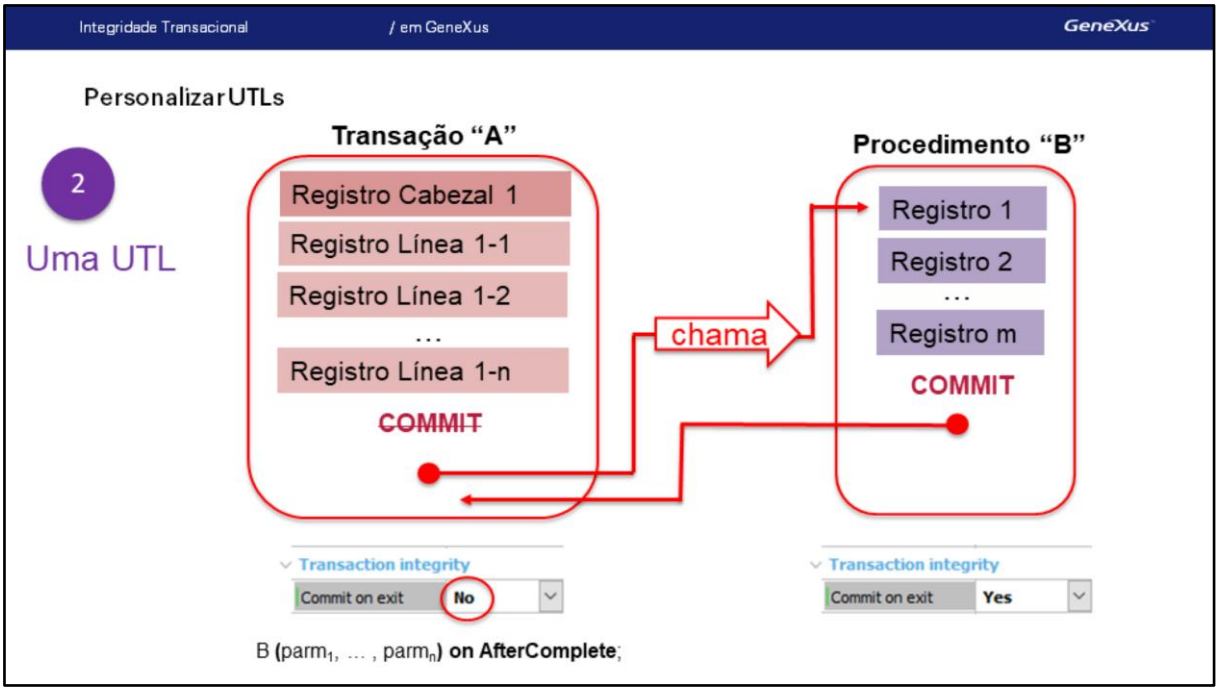


Uma alternativa é realizar os seguintes passos:

1. Chamar o procedimento en algum momento ANTERIOR ao Commit automático. Por exemplo:
   B($parm_1$, ... , $parm_n$**) on BeforeComplete**;

2. Desabilitar o Commit automático do procedimiento.

Desta manera, estaremos chamando o procedimento depois de ter gravado todos os registros (cabeçalho e linhas), tendo já disparado todas as regras condicionadas a eventos AfterLevel. Ou seja, o procedimento estará sendo chamado um instante antes do Commit. O procedimento irá fazer todas as suas atualizações de registros no banco de dados. Como desabilitamos seu Commit, se o desenvolvedor não declarou este comando explicitamente dentro do seu código, a UTL não será finalizada. Depois que terminar a execução do código do procedimento, retorna-se para o seu chamador, na linha imediatamente seguinte à da sua chamada. Aqui é onde se encontrará o Commit.



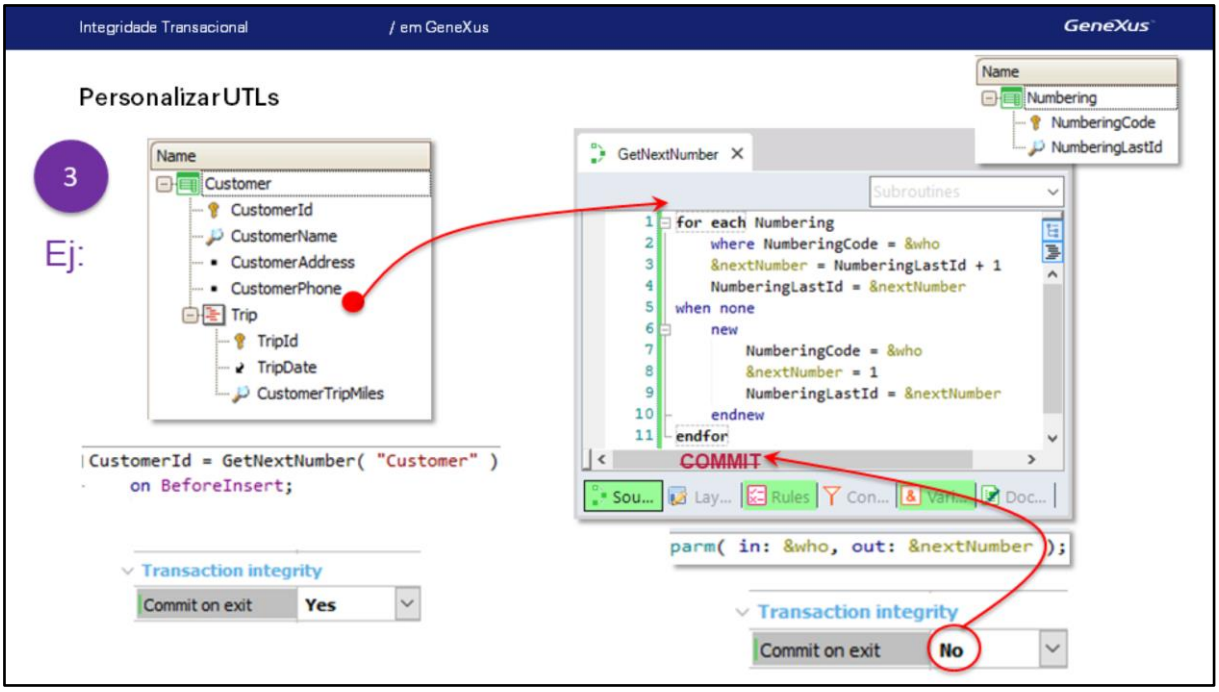
Outra alternativa é realizar os seguintes passos:

1. Não importa quando o procedimiento é chamado, contanto que seja sempre depois de gravar todos os registros (cabeçalho e linhas) da transação, quer dizer, depois de onde o Commit iria ser executado:
   B($parm_1$, ... , $parm_n$**) on AfterComplete;**

2. Sempre quando esse Commit automático da transação é desabilitado e deixa-se habilitado o commit automático do procedimento.

Chamando o procedimento entre o evento AfterLevel Level 2nd-level-attibute/Before Complete e o AfterComplete, estamos grarantindo que todos os registros do cabeçalho e linhas serão gravados. Depois, o procedimento realizará suas atualizações nos registros do banco de dados, e fará o commit, gravando fisicamente, portanto, todos os registros (os seus e os da transação).

Estas são só duas das várias alternativas. A escolhida dependerá da lógica que se quer implementar (normalmente, o momento da chamada do procedimento será indiferente).



Consideremos a transação Customer que registra os clientes e suas excursões contratadas.

Vamos supor que não queremos usar a estratégia de autonumeração do banco de dados, Ao invés disso, vamos criar uma tabela interna própria no banco de dados que registre o último número dado ao identificador de cada entidade.

Para isso, vamos criar uma transação chamada Numbering cujo atributo identificador NumeringCode registra o nome da entidade que queremos controlar (por exemplo, “Customer”, “Trip”, “Invoice”, “Category”, “Attraction”, etc.) e seu atributo NumeringLastId, irá guardar o último número dado para essa entidade.

Depois, programaremos um procedimento, GetNextNumber, que será o responsável por obter o próximo número a ser atribuido ao identificador da entidade que o está chamando.

Por exemplo, na transação Customer, quando o usuário quiser inserir um cliente novo, deixará o campo CustomerId vazio na tela e quando sair do campo, passando para o campo seguinte, CustomerName, a transação saberá que está sendo inserido um novo registro (ficará em modo “INS”, insert).
Como não temos o atributo CustomerId autonumérico, teremos que chamar o procedimento GetNextNumber, para obter o número a ser informado para CustomerId.
Devemos passar por parâmetro para o procedimento, o nome do objeto chamador, ou seja, neste caso, “Customer” para que o procedimento acesse a tabela Numbering e verifique o último número que foi dado para um “customer”, acrescente mais um a esse valor, atualize esse novo valor na tabela Numbering, e retorne esse novo número para o chamador do procedimento (Customer).
Se chamarmos esse procedimento através de uma regra de atribuição sem evento de disparo:

CustomerId = GetNextNumber(“Customer”) if Insert;

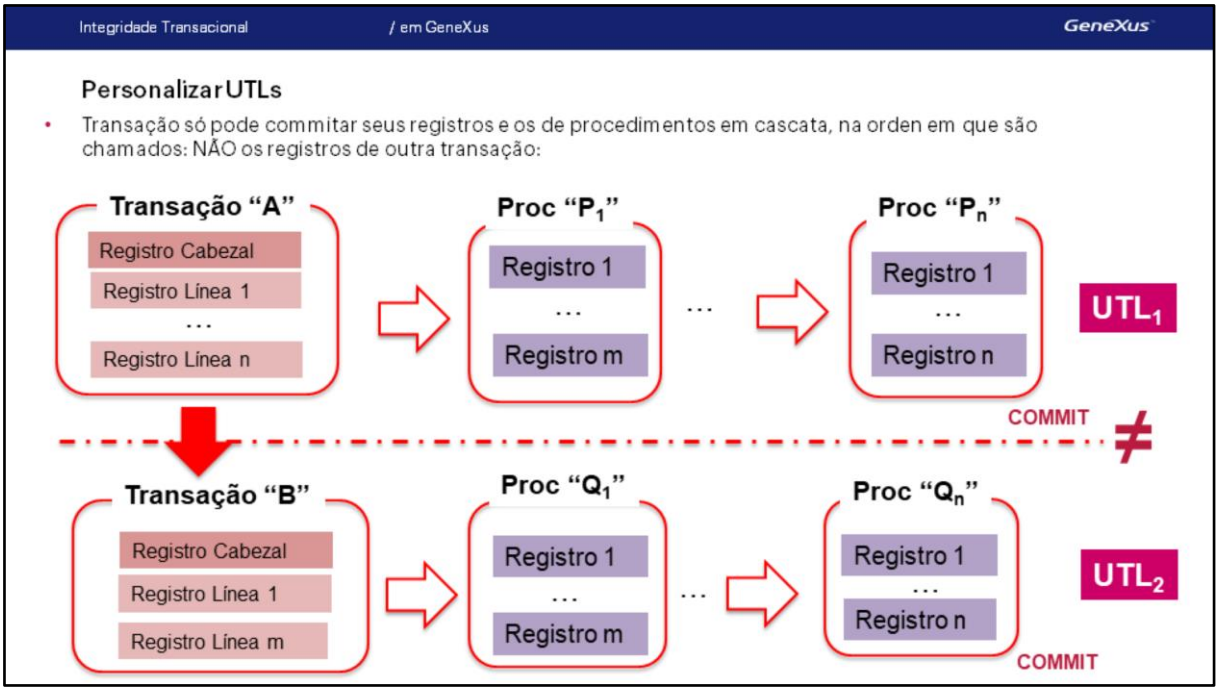
o procedimento será disparado:

1. Uma vez, quando o usuário passar pelo campo CustomerId, deixando-o vazio.
2. Uma segunda vez, quando o usuário pressionar o botão Confirm, porque, nesse momento, as regras voltam a ser disparadas, no servidor.

Imaginemos que o último id de Customer é o 5. No item 1., o procedimento será executado, atualizando o registro correspondente na tabela Numbering para 6. O novo valor (6) é mostrado imediatamente para o usuario na tela. Porém, o que acontece se o usuário se arrepende e nunca pressiona o Confirm, mas cancela a ação? O número 6 ficará perdido. Por outro lado, devido ao item 2., se o usuário confirma, o procedimento voltará a ser executado e o valor atribuido ao cliente será o 7. Dessa forma, o número 6 também se perderá.

Como resolvemos essa situação? Condicionando a chamada do procedimento a um evento de disparo (desta maneira, a regra de atribuição não será disparada enquanto o usuário trabalha na tela). Qual é o momento mais apropriado? O último momento possível para atribuir um valor é o BeforeInsert. Porque? Porque depois desse momento, o valor atribuido a qualquer atributo não terá efeito, uma vez que o registro já estará gravado.

Contudo, o procedimento GetNextNumber modifica um registro da tabela Numbering, ou insere um, se ele não existir. Então, por padrão, colocará um Commit automatico no final. Dessa forma, quando a execução retorna para a transação, vamos supor que o cliente é inserido em sua tabela, e quando a terceira excursão está sendo incluida, acontece uma queda do sistema. Quando o sistema volta, veremos que os registros associados ao cliente não ficaram registrados. Mas o número 6 ficará perdido, porque já havia sido commitado. Para que todas as operações realizadas pela transação e procedimento fiquem dentro da mesma UTL, basta definir o Commit on Exit do procedimento como NO, e deixar que o Commit de tudo seja feito pela transação.

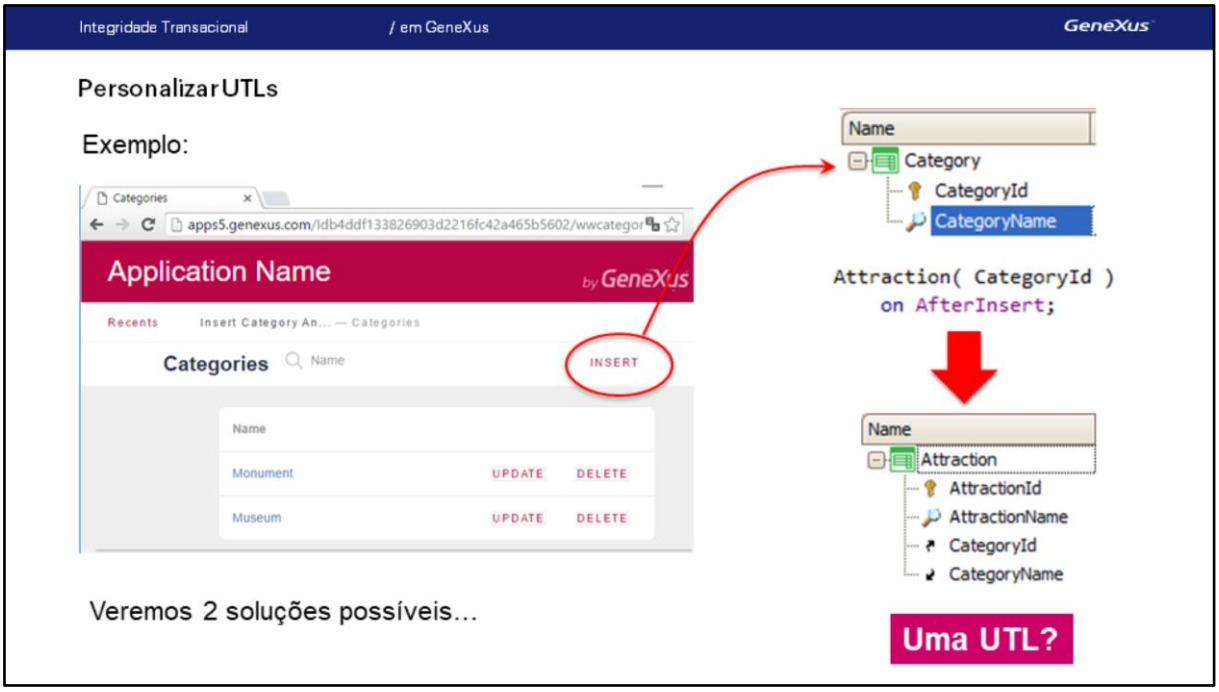


Não é possível haver uma única UTL entre transações diferentes.

Se, a partir de uma transação, chamamos um procedimento que chama outro procedimento que chama outro procedimento, todos com o Commit desabilitado, com exceção do último, seu commit gravará os registros da transação e de todos os procedimentos. Ou poderíamos ter o commit do último procedimento desabilitado também e, nesse caso, quem faria o Commit é a transação quando o controle retornasse para ela.

Contudo, se a transação ("A") chama uma outra transação ("B"), o commit da segunda ("B") **NÃO commitará** os registros manipulados pela primeira ("A"). São Commits independentes. Por isso, se desabilitamos o commit de "A" e chamamos a "B", que faz Commit, mesmo assim os registros de "A" não serão commitados!

Como fazemos, então, para conseguir cadastrar o cabeçalho e linhas de uma transação "A", chamar uma transação "B" para inserir informações relacionadas, e que todas estas operações formem uma única UTL?



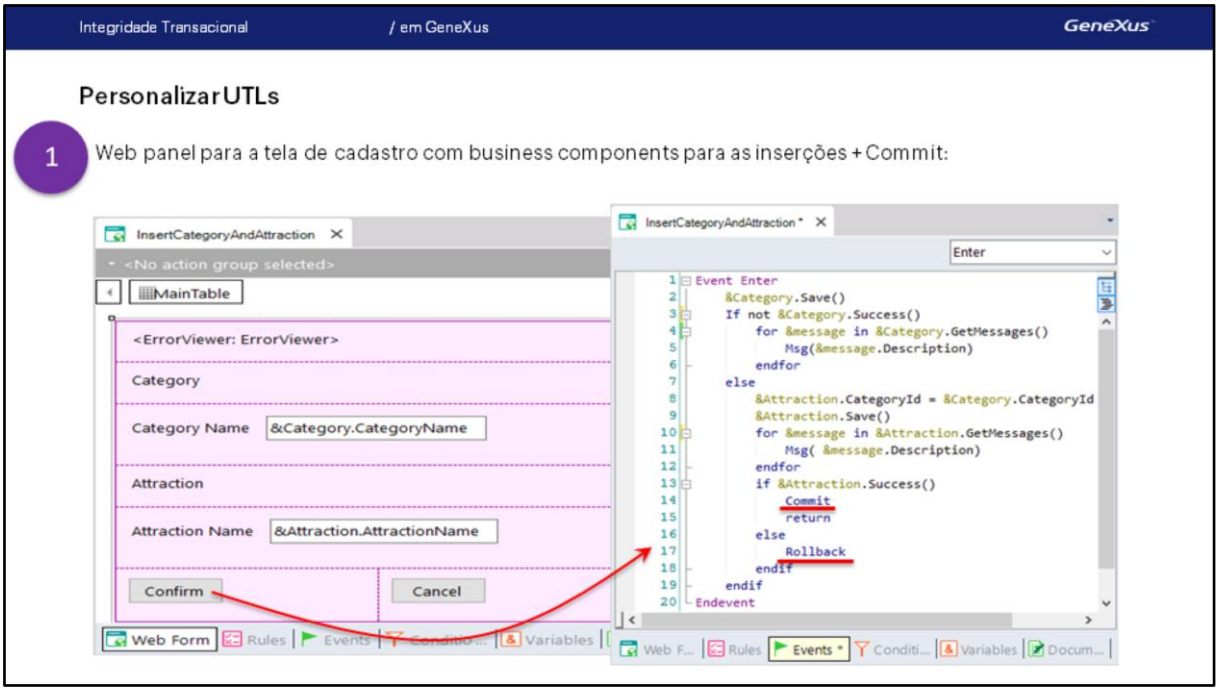
No objeto Trabalhar com Categories (wwcategories) queremos que, quando o usuário clicar em Insert, seja oferecida a possibilidade de inserir uma nova categoria e, em seguida, uma atração para essa categoría, porque não queremos criar categorias sem nenhuma atração associada.

Podemos, através do evento AfterInsert da transação Category, chamar a transação Attraction (para que o CategoryId já tenha o valor autonumérico correto, criado pelo banco de dados), e definir que Attraction receberá o valor de CategoryId através de um parâmetro.

Porém...o que acontecerá se o sistema cair depois que Attraction já tiver feito seu commit? Esse Commit, "commitará" também o registro de Category que havia sido inserido antes? A resposta é NÃO, pelo que já vimos antes.

Precisamos que a inserção de uma categoria, seguida da inserção de uma atração, formem uma mesma UTL e o Commit final sirva para os dois registros. Como conseguimos isso?

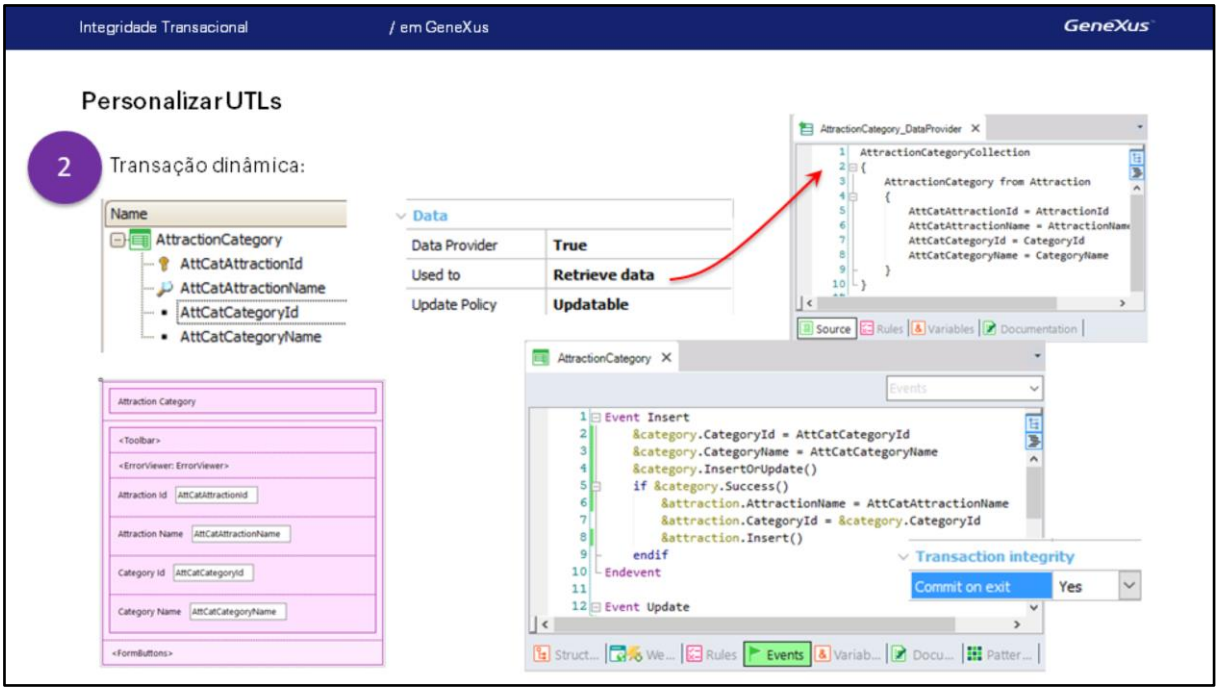
Veremos duas opções, de muitas que existem.



Uma solução é fazer com que o botão Insert do "trabalhar com" chame um web panel, que é um objeto com interface, onde os desenvolvedores podem programar seu comportamento livremente.
No web panel, incluimos duas variáveis (&category e &attraction) em seu Form. Serão Business Components de Category e Attraction, respectivamente.
Só nos interessa que o usuário insira o nome da categoria que está querendo criar, e o nome da atração dessa categoria. Ao fazer isso e pressionar Confirm, será disparado um evento associado ao botão, que estará programado como vemos na aba de eventos.
Primeiro queremos fazer o Save do BC de Category. Se falhar (por exemplo, se o usuario deixou o nome da categoria vazio na tela e a transação tem uma regra Error para prevenir essa situação) mostramos as mensagens geradas pelo BC através de um controle ErrorViewer.
Se não falhar, então atribuimos o valor da categoria recém-criada na variável BC de Attraction e tentamos fazer o Save. Mostramos as mensagens produzidas (sejam de Error ou Sucess, como "Data was successfuly added") e depois, se a operação teve sucesso, fazemos **Commit**. Depois disso, ambos registros ficarão commitados.
Se a operação falhou, então podemos contar com o **comando Rollback**, para desfazer a inclusão do registro de Category que já havia sido incluido com sucesso.

Personalizar UTLs

- GeneXus nos oferece os comandos: **Commit** e **Rollback**
- Eles nos permiten personalizar UTLs, em conjunto com o **Commit** automatico que podemos ligar ou desligar através da propriedade **Commit on Exit** em transações e procedimentos.
- É possível escrever em Procedimentos e Web Panels, combinados com Business Components. **Não** em Transações.



Uma segunda alternativa para resolver o mesmo problema é utilizar uma transação dinâmica em vez do web panel.

No exemplo, criamos a transação dinâmica AttractionCategory, cuja informação será originada da tabela de atrações, sabendo que cada atração tem uma categoria. Será como uma View de atrações.

Quando o usuário quiser inserir uma atração nessa transação, será possível cadastrar tanto os dados da atração como os da categoria. No evento Insert, utilizamos a variável &category que é Business Component de Category, e &attraction, Business Component de Attraction. Copiamos os valores que o usuário especificou nos atributos da transação dinâmica através do seu Web Form, para os membros dos Business Components.
Se a categoria não existe, ela é criada automaticamente antes de inserir a atração. Se existe, é possível atualizar seu nome. Depois, é feita a inserção da atração junto com a categoria.
Se deixamos a propiedade Commit on Exit da transação com o valor padrão (Yes), depois da execução do evento Insert, por tratar-se de uma transação com um só nível, o Commit será efetuado, e terá efeito sobre os dois registros incluidos através dos Business Components.

GeneXus™

The power of doing.

Videos	training.genexus.com
Documentation	wiki.genexus.com
Certifications	training.genexus.com/certifications