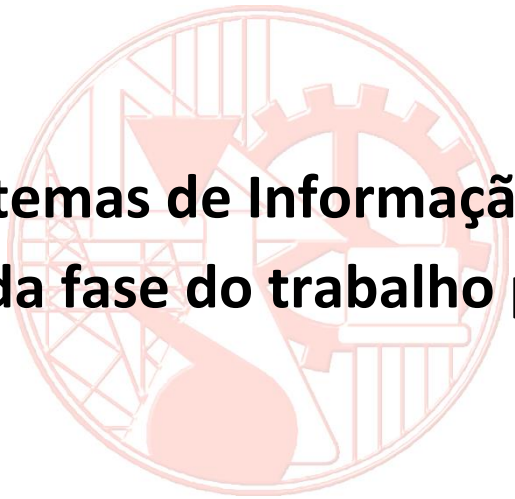
Instituto **S**uperior de **E**ngenharia de **L**isboa

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2015/2016



Sistemas de Informação II
Segunda fase do trabalho prático

LI51N

Trabalho elaborado pelo Grupo 1:
Sara Sobral N.º 40602
Rute Marlene N.º 40531
Filipa Fernandes N.º 39129

Índice

Introdução

Pretende-se realizar um Sistema de Informação para a empresa ISto É Lindo (ISEL) que permita gerir um sistema de vendas *online*.

É necessário desenvolver o modelo de dados adequado às necessidades da empresa, tal como utilizar corretamente os conhecimentos adquiridos de *Transact* SQL e os do Processamento Transacional.

Etapas de desenvolvimento

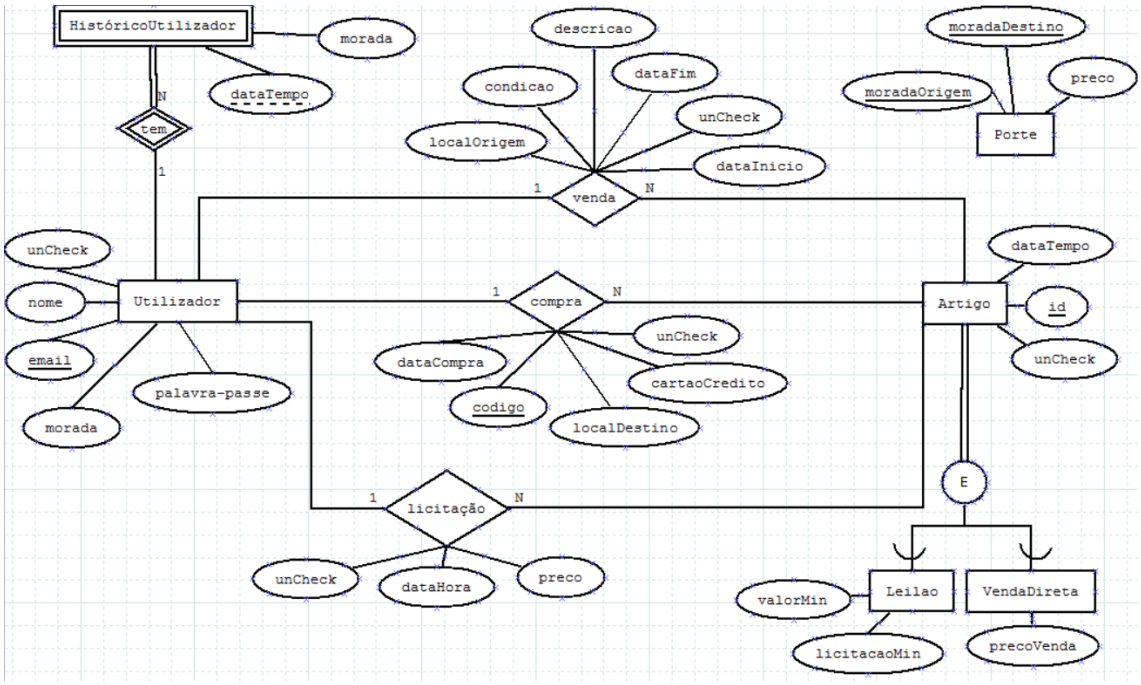
FASE I

Modelo de dados conceptual

Para o desenvolvimento do modelo Entidade-Associação (EA):

1. Nesta etapa deve-se identificar as entidades relevantes, representando-as conforme os requisitos pretendidos.
2. De seguida atribuir às entidades os atributos descritivos consoante o domínio das entidades e atributos.
3. Associar as entidades com associações, respeitando as obrigatoriedades do enunciado e indicando as cardinalidades.
4. Identificar os requisitos e restrições que não conseguem ser garantidos no modelo EA, para garantir a captura de um maior número de restrições possíveis.

Diagrama entidade-associação



Descrição das entidades e atributos e associações

Entidade representativa dos portes a cobrar por localidades:

- Porte
  1. moradaOrigem, cadeia de caracteres, não nulo
  2. moradaDestino, cadeia de caracteres, não nulo
  3. preco, inteiro, não nulo

Entidade representativa do Utilizador do sistema de vendas:

- Utilizador

1. nome, cadeia de caracteres
2. morada, cadeia de caracteres, não nulo
3. email, cadeia de caracteres, não nulo
4. palavra-passe, cadeia de caracteres, não nulo
5. unCheck, bit, não nulo

Entidade representativa de alterações de morada do Utilizador:

- HistoricoUtilizador
  1. email, cadeia de caracteres, não nulo
  2. dataTempo, data, não nulo

Entidade representativa dos artigos disponíveis no sistema de vendas:

- Artigo
  1. id, inteiro, não nulo
  2. dataTempo, data, não nulo
  3. unCheck, bit, não nulo

Entidade representativa dos artigos do tipo de venda leilão:

- Leilao
  1. valorMin, int não nulo
  2. licitacaoMin, int, não nulo, representante do incremento mínimo a efetuar após a última licitação, no caso da primeira licitação considera-se 0.

Entidade representativa dos artigos do tipo de venda direta:

- VendaDirecta
  1. precoVenda, int, não nulo

Entidade representativa da venda dos artigos:

- Venda
  1. localOrigem, cadeia de caracteres, não nulo
  2. condição, cadeia de caracteres, não nulo
  3. descrição, cadeia de caracteres
  4. dataFim, data, não nulo
  5. dataIncio, data, não nulo
  6. unCheck, bit, não nulo

Entidade representativa da licitação dos artigos:

- Licitacao
  1. Código, int, não nulo
  2. localDestino, cadeia de caracteres, não nulo
  3. dataCompra, data, não nulo
  4. cartaoCredito, cadeia de caracteres, não nulo
  7. unCheck, bit, não nulo

Entidade representativa da compra dos artigos:

- Compra
  1. dataHora, data, não nulo
  2. preco, inteiro, não nulo
  3. unCheck, bit, não nulo

Restrições não capturadas pelo Modelo EA

RI1. Obrigatório inserir preço no Porte.
RI2. Obrigatório inserir o valor minino de licitação e o valor de licitação mínima do Leilão.
RI3. Obrigatório inserir o preço de venda da VendaDirecta.
RI4. Quaisquer preços, como licitacaoMin e valorMin) devem sempre ser superior a 0€.
RI5. O valor da licitação mínima deve estar entre 1 euro e 10% do valor de venda.
RI6. Obrigatório inserir morada e palavra-passe no Utilizador.

RI7. A palavra passe deve ter uma dimensão entre 6 e 50.
RI8. Obrigatório inserir o email no HistóricoUtilizador sempre que há mudança de morada.
RI9. O atributo de Venda condição pode ter os valores "novo", "usado", "como novo" ou "velharia vintage".
RI10. Para garantir um correto ordenamento das licitações, todas as datas têm de ser inseridas na base de dados com precisão de milissegundos.
RI11. A data de início da venda deve ser inferior à data de fim da mesma.
RI12. A data de licitação deve estar entre a data de início e a data de fim de venda.
RI13. A data de compra pode ser no máximo superior a 2 dias da data de fim da venda.
RI14. O atributo *unCheck* deve ter os valores 0 ou 1.
RI15. Os locais dos portes devem estar de acordo com as normas ISSO 3166-1.

Descrição das regras de negócio aplicáveis

1. A palavra passe não será guardada em claro, ficando registado o resultado da função de *hash* MD5.
2. No caso de venda direta, uma licitação só é válida se valor de licitação for igual ao valor de venda, e só pode ser feita uma licitação.
3. No caso de leilão, uma compra só é válida se valor de licitação for igual ou superior ao valor de venda.
4. Quando em leilão, se o valor mínimo não for atingido o artigo não será vendido.
5. Todos os artigos podem ser licitados
6. Não são permitidas inserções de novas licitações sempre que o tempo definido para a venda tenha terminado.
7. Em qualquer momento deve ser possível simular o custo total de compra, usando um valor de licitação válido e uma localização de entrega.
8. É ainda possível um utilizador retirar uma licitação que efetuou, desde que dentro do período de venda. No entanto, essa informação não pode ser removida do sistema. Em casos como este é utilizado uma flag *unCheck* que indica se o tuplo está ou não presente.

Modelo de dados lógico

O modelo de dados lógico foi obtido por transformação do modelo conceptual apresentado.

PK –> primary key;
AK –> alternative key;
CK –> compose key.

Utilizador (email, palavraPasse, nome, morada, unCheck)
PK = {email}

HistoricoUtilizador (dataTempo, morada, email)
FK = {email} ref Utilizador(email)
PK = {email, dataTempo}

Artigo (id, dataTempo, unCheck)
PK = {id}

Leilao (artigoId, dataTempo, licitacaoMin, valorMaxn)
FK = {artigoId, dataTempo} ref Artigo (id, dataTempo)
PK = {artigoId}

VendaDirecta (artigoId, dataTempo, precoVenda)
FK = {artigoId, dataTempo} ref Artigo (id, dataTempo)
PK = {artigoId}

Venda (descricao, dataInicio, dataFim, localizacao, condicao, unCheck, email, artigoId)
FK = {email} ref Utilizador(email)
FK = {artigoId} ref Artigo(id)
PK = {artigoId, email}

Licitacao (dataHora, check, preco, unCheck, email, artigoId)
FK = {email} ref Utilizador(email)
FK = {{artigoId} ref Artigo(id)
PK = {email, artigoId}
PK = {artigoId, email}

Compra (dataCompra, codigo, localizacao, cartaoCredito, unCheck)
FK = {email} ref Utilizador (email)
FK = {{artigoId} ref Artigo(id)
PK = {codigo}

Porte (preco, moradaOrigem, moradaDestino)
PK = {moradaOrigem, moradaDestino}

Restrições não capturadas pelo Modelo Lógico

RI1. A data de licitação deve estar entre a data de início e a data de fim de venda.
RI2. A data de compra pode ser no máximo superior a 2 dias da data de fim da venda.
RI3. Os locais dos portes devem estar de acordo com as normas ISSO 3166-1.

Normalização até à terceira forma normal

- Tendo em conta a 1ª forma normal podemos dizer que, como nenhuma das tabelas tem atributos não atómicos, o modelo está normalizado segundo a 1ª forma normal.

- Segundo a 2ª forma normal, podemos concluir que este está normalizado pois não temos dependências parciais funcionais, ou seja, atributos não chave que dependam de apenas parte da chave primária.

- Por fim a com 3ª forma normal, podemos aferir que o modelo se encontra normalizado segundo a 3ª forma normal pois não temos dependências transitivas, ou seja, não existem, em tabela alguma, atributos não chave que dependam de outros atributos não chave

Tendo em conta os três pontos acima apresentados, podemos então concluir que o modelo lógico apresentado se encontra normalizado até, pelo menos, à 3ª forma normal.

Dependências funcionais

Entidades fraca:

HistoricoUtilizador:	email, dataHora -> morada

Entidades forte:

Utilizador:	email -> morada, palavra-passe, nome
Porte:	moradaDestino, moradaOrigem -> preco
Artigo:	id -> dataTempo
Leilao:	id -> valorMin, licitacaoMin
VendaDirecta:	id -> precoVenda

Associações:

Venda:	artigoId -> descricao, condicao artigoId, email -> dataInicio, dataFim email -> localOrigem
Licitacao:	artigoId -> preco email -> dataHora
Compra:	codigo -> dataCompra email -> localDestino, cartaoCredito

Base de dados

Resolução do problema para a criação/eliminação da base de dados:

```
USE [master];
IF EXISTS (SELECT name FROM master.dbo.sysdatabases WHERE name = N'ISEL')
BEGIN
  print('Removing database named ISEL');
    ALTER DATABASE ISEL SET SINGLE_USER WITH ROLLBACK IMMEDIATE
    DROP DATABASE ISEL;
END

GO
CREATE DATABASE ISEL;
print('Database named ISEL is created');
```

Questões

(a) Criar o modelo físico;

create_DB.sql – Cria a Base de Dados (BD).
create_Tables.sql – Cria as tabelas correspondentes ao modelo relacional.

(b) Inserir, remover e atualizar informação de um artigo;

Artigo.sql – cria três processos.

Um processo (`inserirArtigo`) que insere um artigo com a data passado por parâmetro na BD, outro processo (`apagarArtigo`) que remove o artigo passado como parâmetro da BD e outro (`atualizarArtigo`) que atualiza a informação passada como parâmetro de um artigo também passado como parâmetro (b).

(c) Inserir, remover e atualizar informação de um utilizador;

Utilizador.sql – Cria três processos.

Um processo (`inserirUtilizador`) que insere um utilizador com os valores passados por parâmetro, outro processo (`apagarUtilizador`) que remove o utilizador passado como parâmetro da BD e outro (`atualizarUtilizador`) que atualiza a informação passada como parâmetro de um utilizador também passado como parâmetro (c).

(d) Inserir, remover e atualizar informação de um local;

Local.sql – Cria três processos.

Um (`inserirLocal`) insere portes e preços na tabela de portes, outro (`apagarLocal`) remove o porte com os locais passados por parâmetro e outro que (`atualizarLocal`) atualiza o preço de um local (d).

(e) Inserir uma licitação; (f) Retirar uma licitação

Licitacao.sql – Cria dois processos.

O `inserirLicitacao` verifica se a licitação é feita sobre um leilão ou uma venda direta, verifica o valor de licitação, no caso de ser venda direta verifica se já existe alguma licitação sobre o artigo. O `removerLicitacao` remove a licitação com os dados passados por parâmetros (e), (f).

(g) Concluir a compra de um leilão; (h) Realizar a compra de um artigo de venda direta;

VendaDireta.sql – Cria dois processos que apresentam os dados de uma compra sobre um determinado artigo, verificando se os parâmetros são válidos e apresentando o valor total (preco+porte) a pagar (g), (h).

(i) Determinar o valor da licitação de um artigo;

ValorLicitacao.sql – Cria uma função que retorna uma tabela com o ultimo valor da licitação de um artigo passado como parâmetro;

(j) Obter as n últimas licitações;

nLicitacoes.sql – Cria uma função retorna uma tabela com as n últimas licitações do artigo passado como parâmetro (j);

(k) Obter os portes, dadas duas localizações;

Porte.sql – Cria um procedimento que retorna o valor do porte dadas duas (k);

(l) Listar os leilões que não foram concluídos;

LeilaoNconcluido.sql – Cria uma vista para apresentar os leilões que não foram concluídos, ou seja, um artigo do tipo leilão que não chegou a estar à venda, que não tenha sido licitado, que a data de fim de venda já tenha expirado, que ao existir compra possível o artigo não foi comprado dois dias apos a data de fim de venda e que o valor mínimo da licitação não tenha sido atingido (l);

(m) Verificar a *password* de um utilizador;

verificarPP.sql – Cria um procedimento que verifica se a palavra passe passada como parâmetro corresponde à palavra passe do email passado como parâmetro (m);

(n) Testar todos os requisitos anteriores.

Foi realizado um ficheiro RunTestScripts.bat para a execução de todos os ficheiros de teste (TestX.sql, em que X se refere ao teste).

Em vez do *rollback* no final de cada ficheiro para não "estragar" a BD, são limpos os tuplos das tabelas.

Foi ainda criado um script que contêm todos os *triggers* a utilizar:

create_Triggers.sql – Cria *triggers* que correm nas tabelas geradas no script acima.

utilizadorIU – *trigger* que depois de uma inserção ou atualização do utilizador coloca a palavra passe em formato md5.

deleteUtilizador – *trigger* que em vez de apagar o utilizador da BD coloca todos os tuplos em que esse utilizador aparece com a flag *unCheck* a 0.

deleteArtigo – trigger que em vez de apagar o artigo da BD coloca todos os tuplos em que esse artigo aparece com a flag *unCheck* a 0.

deleteLicitacao – *trigger* que em vez de apagar a licitação da BD coloca esse tuplo com a flag *unCheck* a 0.

deleteCompra – *trigger* que em vez de apagar a compra da BD coloca esse tuplo com a flag *unCheck* a 0.

deleteVenda – *trigger* que em vez de apagar a venda da BD coloca esse tuplo com a flag *unCheck* a 0.

Foi realizado um ficheiro de extensão ".bat" que executa os scripts relativos ao preenchimento da BD.

Uso de transações

Na realização do trabalho utilizam-se os níveis de isolamento *Repeatable Read* e *Read Committed*.

Read Committed para as inserções, remoções e atualizações para que não haja outra transação a ler os dados de uma tabela enquanto se está a efetuar estas operações numa dada tabela.

Repeatable Read para as operações de processamento para que não exista uma outra transação a modificar dados enquanto se efetua uma atualização aos dados.

FASE II

Nota: Para testar a fase 2 podem-se preencher algumas tabelas da Base de Dados, para tal execute o ficheiro RunSqlScriptsFillInTables.bat

Obedecendo ao XML Schema, exportar a informação das licitações de um leilão que tenha terminado

Através da aplicação (na classe XmlConverter.cs) é gerado um XML de acordo com o conteúdo da base de dados e de acordo com o *Schema* apresentado no enunciado. O xml gerado fica guardado num ficheiro com o nome "`exportedResult`.xml".

Foi usado o seguinte *website* para a sua validação:
http://www.utilities-online.info/xsdvalidation/#.V3vCz_krJD8

Criar uma aplicação usando Entity Framework

Para a utilização de Entety Framework é necessário usar contextos que são adicionados de forma automática pelo ADO.NET Entety Data Model. A classe dedicada a esta parte é a AppEF.cs.

Criar uma aplicação ADO.NET usando objetos "conectados"

Para a utilização de objetos conectados, é necessário ter acesso à base de dados, para tal com os dados introduzidos pelo utilizador é estabelecida uma conexão. A classe dedicada a esta parte é a App.cs.

Facilidade de programação e desempenho de Entity Framework e ADO.NET

Em *ADO.NET* necessitamos de abrir uma conexão à base de dados. Por outras palavras sempre que executamos uma instrução SQL necessitamos de abrir uma conexão executar o comando e fechar a conexão.

O *Entity Framework* é uma API de acesso a dados para o mapeamento do objeto relacional, que permite aos utilizadores trabalhar com bases de dados, tornando transparente o acesso aos mesmos, eliminando a necessidade de escrever código SQL, como *Select*, *Insert*, *Update*, *Delete* na aplicação.
Para aceder aos dados da base de dados apenas temos que instanciar um objeto "*Context*" sobre o qual acedemos aos objetos referentes à base de dados.

O resultado da aplicação em *Entity Framework* é igual ao obtido com *ADO.NET*, mas neste caso não é necessário abrir conexão à base de dados nem saber o comando *SQL* necessário e o fecho da ligação é gerido pela *Entity Framework*.

Podemos concluir que existe uma maior facilidade na utilização da *Entity Framework* devido a já ter todos os dados de todas as tabelas não sendo necessária a criação de código *SQL* (*querys*) dentro da nossa aplicação.

Vantagens e desvantagens de *Entity Framework* e ADO-NET em termos de garantir a consistência de dados

Em relação à consistência dos dados, a *Entity Framework* vai sempre verificar a por alguma alteração que possa ter ocorrido fora do contexto atual e só vai proceder às alterações caso não tenha ocorrido nenhuma alteração, se existirem alterações vai ser lançada uma exceção.

Em *ADO.NET* é usado controlo transacional pelo utilizador de modo a garantir toda a consistência dos dados.

Conclusão

Na primeira fase foram gerados:

O modelo EA foi realizado com base nos requisitos apresentados no enunciado, adicionados atributos *flag* para melhor cumprimento dos requisitos.

O modelo de dados foi também realizado de acordo com os requisitos da empresa e pelo modelo EA, a normalização do mesmo foi realizada com sucesso, pois aquando a construção do modelo somo impelidos a normalizá-lo.

Na criação do modelo físico foram implementadas vistas, procedimentos armazenados, gatilhos, funções, níveis de isolamento e controlo transacional que permite o bom funcionamento da base de dados criada para o propósito do trabalho.

Na segunda fase foi possível fazer melhoria ao modelo relacional.

Foi gerada uma aplicação que permitiu entender como é que o XML e o XSchema se relacionam e funcionam.

Com esta aplicação foi possível entender as diferenças da utilização de *Entity Framework* e de *ADO.NET*.

Anexo teórico

Vistas

Vistas ou *Views*, são conhecidas como tabelas virtuais que são definidas por uma consulta e utilizadas como uma tabela. A *view* apesar de ser semelhante a uma tabela o que faz é armazenar uma instrução SQL no banco de dados.

Procedimentos armazenados

Um procedimento armazenado, também conhecido como *store procedure*, é um conjunto de uma ou mais instruções SQL, que é armazenado no sistema de gestão de base de dados.

Os procedimentos armazenados devem implementar aspetos que estejam relacionados com a natureza intrínseca dos dados como, validação de regras de negócio associados a restrições de integridade dos dados, não suportadas diretamente pelo SGBD.

Existe igualmente um melhor desempenho, isto porque quando um procedimento é compilado na primeira vez em que é executado cria um plano de execução que é reutilizado em execuções subsequentes, assim sendo, o processador de consulta demora menos tempo para processar o procedimento.

Gatilhos

Triggers são procedimentos que são executados automaticamente, como resultado de um evento gerado no SGBD – execução reativa.

Existem dois tipos de *triggers*, do tipo *After* e do tipo *Instead Of*. Os *After* são executados após a execução com sucesso da instrução que os despoletou, não após cada tuplo manipulado. Os *Instead Of* são executados em vez da instrução que os despoletou.

Funções

Uma função é uma rotina que aceita parâmetros, realiza uma ação e retorna um resultado da execução dessa ação. O valor retornado pode ser escalar ou uma tabela.

Níveis de isolamento

Existem vários níveis de isolamento de uma transação:

- ***Serializable***: As instruções não podem ler dados que foram modificados, e que ainda não foram confirmados por outras transações. Nenhuma outra transação pode modificar dados lidos pela transação atual até que a transação atual seja concluída (*commit/rollback*). Outras transações não podem inserir linhas novas com valores chave que estejam no intervalo de chaves lido por qualquer instrução da transação atual até que esta seja concluída.
- ***Repeatable Read***: Uma transação não pode ler dados que foram modificados por outra transação sem esta terminar (*commit*/*rollback*). Nenhuma outra transação pode modificar informação lida pela transação corrente até esta estar completa.

- ***Read Commited***: Uma transação não pode ler dados que foram modificados por outra transação sem esta terminar (*commit*/*rollback*). A informação lida pela transação corrente pode ser modificada por outras transações antes desta estar completa.
- ***Read Uncommited***: Podem ser lidas linhas pela transação corrente que foram modificadas por outra transação sem esta terminar (*commit*).

Controlo transacional

Existem os seguintes comandos para controlar transações:

- ***Commit***: Salva todas as transações no banco de dados desde o último *commit* ou *rollback*;
- ***Rollback***: Para reverter as alterações. Só pode ser usado para desfazer operações desde o último *commit* ou *rollback*;
- ***Savepoint***: É um ponto numa transação para quando se quer reverter a transação até um certo ponto.
- ***Set Transaction***: Pode ser usado para iniciar uma transação no banco de dados. Este comando é usado para especificar as características da operação que vem de seguida. Por exemplo, pode servir para especificar se uma transação é de leitura, escrita ou leitura e escrita.