Application Object Tree (AOT)

Lifecycle Services (LCS)

Software Deployment Package (SDP)

(IDE) in Microsoft Dynamics AX is called MorphX.

programmable objects in Microsoft Dynamics AX—tables, forms, reports, classes, and so on—are organized in a tree structure called the Application Object Tree (AOT).

Creating and editing objects is enhanced by drag-and-drop functionality

The source code for methods on classes, tables, forms, and other objects is available to help you extend and customize Microsoft Dynamics AX functionality.

X++ is the programming language in Microsoft Dynamics A

Class Defines an object's interfaces.

instructs or explains how to construct an object of a particular type.

characteristic of a class is that you can create a new instance (an object) of the class.

Forms are an example of a class

Controls A graphical object, such as a text box, a check box, a command button, or a rectangle, that you place on a form or report when you are designing it to display data, perform an acti o n

Data Source Holds the data variables that a form or a query uses

data variables can be o ne or more tables, or they can be individual fields from tables .

Object

Any form or control is an object. The database is an object.

Properties are data that describe an object. Each type of object has different types of properties

A query is a filter mechanism to retrieve the data you want to see from your database tables.

Application object layer

Application object model

**Enums**

Click new item under our USR layer under our current solution in the solution explorer panel

Will then open a window for you what data type to choose which is I this case an enum, and once we specify the type we can then choose what type of enum.

Choose the base enum type and once item is created in our element panel right click the newly created enun and click new element from here we can add the values to our enum and their respective indexes from which each element will have

e.g.

DBFMServiceType

click new element for each of the ff: None, Registration, OilChange, Maintenance, Repair (I assume these are objects that we will create, or maybe not maybe like in C they are just placeholders for numbers/indeces)

To create an enum Expand the Data Dictionary node in the AOT. Right click the Base Enums node and select New Base Enum . Rename the enum. The literals in the enum are called elements. Right-click the enum and select New Element . Rename the element. Add as many additional elements as you need.

Declaring Enums You must create an enum type in the AOT before you can declare it. Enum declaration = enumname Variable { , Variable } ; Variable = identifier [ option ] Option = Arrayoptions | initialization

Lost highlights on this page:

Declaring Enums

You must create an enum type in the AOT before you can declare it.

Enum declaration – enumname Variable { , Variable } ;

Variable – identifier [ option ]

Option – Arrayoptions | initialization

Ex.

//A NoYes enum

NoYes done;

//An array of Criteria enums

Criteria crit[100];

Some more examples of declaring enums in code are:

1. DBFMVehicleType vehicle\_type {Car, Truck, Van, SUV};

DBFMVehicaleType::Car will return 0

1. DBFMFuelType fuel\_type {Gas, Diesel, Propane, Natural Gas, Electric};

enum Django {One, Two, Ps3, Four, Five} stuff; // Variable stuff created to work with

enum Django gamestation = Ps3; // gamestation is now assigned to the Django set, Ps3

gamestation now has a value 2 by default

Best practices:

1. Consider

Length

if it should be mandatory

if it should be a special type

1. Let constant values be enums
2. Never use numeric constants or other constants instead of enums
3. Never use relational operators on enums especially of enums of different types
4. Never assign enums to other enums of different types

**Extended Data Types**

Extended data types (EDTs) are user-defined types, based on the primitive data types boolean, integer, real, string, and date, and the composite type container. You can also base EDTs on other EDTs.

you could create a new EDT called Name and base it on a string

Code is easier to read because variables have a meaningful data type. For example, Name instead of string.

Declaration of EDT Variables In the AOT, the Data Dictionary > Extended Data Types node is used to create EDTs. The range of an EDT is identical to that of the base type it is based on. When you declare a variable in X++, use the syntax shown in the following table.

Extended declaration = Extendedtype Variable { , Variable } ;

Variable = Identifier [ option ]

Option = arrayoptions | initialization

where Extendedtype is the name of the extended data type in the AOT.

X++

// A UserGroupID (integer) variable is declared and initialized to 1.

UserGroupID groupID = 1;

// An Amount (real) variable is declared.

Amount currency;

EmployeeName some\_person = “<some persons name>”

Some EDTs in fleet management scenarios <Type>

* VehicleID or in c terms typedef int VehicleID
* InspectionDate date = <some date value>
* ThirdRowCheckBox trcb1 = <some enum value>
* DBFMGrossVehicleWeight gvw1 = <some int value>

You can create an extended data type in the Application Object Tree (AOT), and then base a table field on this type.

1. In the Application Object Tree, click Data Dictionary
2. Right-click Extended Data Types, click New, and then click a data type to base the extended data type on e.g. string is a the data type the extended data type can base on
3. Right-click the extended data type you created in step 2, and then click Properties
4. To base the extended data type on another extended data type, select an extended data type from the Extends property list

The list of available extended data types varies, depending on the base data type that you selected in step 2 e.g. EDT string, EDT date

1. Modify additional properties, as needed
2. Press CTRL+S to save the extended data type.

* AOT
* Data dictionary

1. Tables
2. Maps
3. Views
4. Extended Data Types
5. MyAddress (USR)
6. Array Elements
7. Address1
8. Address2
9. AdressN
10. Relations
11. Base Enums

Define an Extended Data Type as an Array

1. Expand the relevant extended data type, and then locate the Array Elements node
2. Right-click the Array Elements node, and then choose New Element in the shortcut menu
3. Display the property sheet for the element, and then rename the element by changing the Label property
4. To specify the plural name of the element, use the CollectionLabel property
5. To display a help string when using the extended data type in a form, use the HelpText property
6. Save your modifications.

**Tables, Maps, Views**

Like the Base Enum and Extended Data Types, tables, maps, and views are also elements under the application object tree

Tables, views, and maps are elements that X++ SQL statements can reference to read and write business data

specified in the Application Object Tree (AOT) under AOT > Data Dictionary . The following table describes these elements.

Tables store business data. Each table in the AOT has a corresponding table in the underlying Microsoft SQL Server database.

Method members – A table can have methods, just as a class in X++ or C# can have methods. Table inheritance – A table can extend, or be derived from another table. That same table can be the base table for several derived tables.

My assumption for this is like an entity in Django which is used as basis for creating the table in a data base, a table is also based on an entity, only now we are explicitly creating the entity itself using the table

**Tables**

are the foundation objects in Microsoft Dynamics AX and store data used by the system

made up of records (or rows) that contain information about a single entry in the table.

For example, a specific customer

ecord consists of one or more fields (or columns) that contain a discrete piece of data of a specific data type.

Each table contains the following primary elements:

Objectives in using tables

1. Add data types
2. Extend a pre-built table using inheritance, or by using an already user defined table and inheriting it
3. Set key properties
4. Create a query to create, read, update, patch, and delete an element from the table

Table fields are also based on a primitive data type or an extended data type.

Creating a Table

1. In the AOT expand the Data Dictionary node.
2. Right-click the Tables node, and then select New Table
3. Right-click the table, and then click Properties
4. Rename the table by modifying the Name property
5. To specify the table as temporary, set the Temporary property to Yes. For more information, see Table Properties
6. Modify additional table properties, as needed. For more information, see Table Properties
7. To delete the table, right-click it, and then click Delete

Adding Fields to a Table

Note You can delete only fields that do not contain data in any of the table records. You cannot modify the data type of an existing field

1. Right-click the Fields node of your table
2. Click New and then choose a primitive data type to base your field on. If you plan to base the field on a specific extended data type, you must choose a primitive data type that the extended data type is based on
3. To base the field on an extended data type, set the ExtendedDataType property in the properties panel I guess
4. Modify additional field properties, as needed. For more information
5. To delete the field, right-click it, and then click Delete

Changing table fields during development (like Django we have to find a way to make migrations to all the changes to our entities/tables/models)

When you insert data in a table during development, the SQL statement you use to insert the data is cached in the AOS. Next you might add a new field to the table and persist the change to the database. This causes the SQL statement in the cache to become stale, because the statement is not updated to include the new field. If you reuse the stale statement, the new field is ignored, or an error might occur.

To avoid this problem, restart the AOS after you persist table schema changes to the database. The cache is empty when the AOS restarts.

**fields**

Fields node contains all the fields in the table

Microsoft Dynamics AX performs data validation to ensure that only valid data is entered into each field in the table

Each field in a table has a number of properties that describe the behavior of the field

**Type** property contains the native data type of the field

**ExtendedDataType** property contains the extended data type value (if the field is based on an extended data type

These properties are akin I think to the kwargs we pass when instantiating a model class or the properties of the model class that we create that will be assigned to the necessary fields (in Django)

**Field groups**

Field groups are objects that group together fields that logically belong together, e.g. first\_name and last\_name

Creating a Field Group

1. In the AOT, expand the Data Dictionary node, and then expand the Tables node.
2. Locate the table that you want to add a field group to, and expand that node.
3. Right-click Field Groups, and then click New Group. This will add a new field group called Group1.
4. Right click the new field group, click Properties, and then type a name for the field group in the Name property.
5. Right-click the table, click Open New Window, and then expand the Fields node.
6. In the new window, drag fields from the Fields node onto your new field group.
7. Save the updated version of your table.

**Queries**

Querying data from existing tables will be needed for reports

There are two ways to create queries:

Use the Queries node in the Application Object Tree (AOT) to create static queries using a graphical interface. For more information about creating queries in the AOT, see How to: Create Queries by using the AOT.

Use the query system classes to create dynamic queries in code. For more information about creating queries in code, see How to: Create Queries by Using X++.

No matter which method that you use to create a query, the query can include a sorting order to sort the information that is returned. It can also include a range to filter the information that is returned based on your criteria. The filter funcitonality is like the filter method in javascript wherein only the objects that pass specific constraints are the only ones returned from the function, and like the function callback in sorting functions and methods in most PLs the objects in a collection/array must meet a set of constraints/rules in order to be sorted whether in ascending or descending order

Create the Query Entry

You create the query in the AOT by the following steps:

1. Expand AOT > Queries.
2. Right-click the Queries node, and then click New Query.
3. In the Properties window, edit the Name property value to be QryGbyHavgRange23.
4. Set the AllowCrossCompany property to Yes.
5. Right-click your QryGbyHavgRange23 node, and then click Save to confirm the node and property changes.
6. Expand the node for your new query, so that you can see its subnodes including its Data Sources subnode.
7. Proceed to create the subnodes that are described in the table that follows.

Adding a Data Source

Click the Subnode in queries called Data Sources

Steps:

1. Add the CustTable as a data source. Right-click Data Sources, and then click New Data Source. This creates a new node under Data Sources.
2. Click the new data source node to highlight it.
3. In the Properties window, set the Table property to CustTable. This action also changes the value of the Name property. This query reads data from only one table, the CustTable table.

In the following subnodes of data sources each will dictate how the data source will look like since it will be the one querying data from the table. The data returned will be dictated by a set of constraints again like a filter function/method in JavaScript or a comparator callback in sorting functions

Adding Subnodes Under Data Sources to dictate what rows from the indicated table should be returned based on a set of constraints dictated by the following subnodes

The following table shows the steps to create each subnode under AOT > Queries > QryGbyHavgRange23 > Data Sources. The table also shows the Properties window for each subnode.

Subnode – Steps

1. Fields
2. To add a field under the new CustTable data source node, expand the new data source node.
3. Right-click the Fields node, and then click New > COUNT.
4. In the Properties window for the new field node, set the Field property to RecId.

ANSI SQL typically uses an asterisk for the count aggregate function. But in Microsoft Dynamics AX a field must be used, and by convention the RecId field is usually used.

1. Ranges
2. To add a range node to exclude one particular value of the DestinationCodeId field, each range node applies to the Where clause of the SQL Select statement that is eventually generated, right-click the Ranges node, and then click New Range.
3. Set the Field property to DestinationCodeId.
4. Set the Value property to != "Gen\_5".

Notice that the Value property is set to both a comparison operator and a specific data value. Inclusion of an operator is optional.

1. Group By

Add a Group By clause. A query cannot have a Having clause unless it also has a Group By clause.

For each unique value in the DestinationCodeId field, the query counts all the CustTable records that share the same DestinationCodeId value with each other. This is accomplished by adding a group by clause.

Right-click the Group By node, and then click New Field.

In the Properties window, set the Field property to DestinationCodeId.

Implicitly the system adds the group by field to the fields list at run time, so that the group by field is also returned when the query is run.

Having -

Add a Having clause to filter the aggregate values that are generated by the Group By clause. In the present example, the COUNT(RecId) field contains the aggregate values that are filtered.

Right-click the Having node, and then click New > COUNT.

In the Properties window, set the Field property to RecId.

Set the Value property to > 6.

The value filter is compared against the COUNT aggregate result of each record that otherwise can be returned. The value is not compared against the RecId field.

Order By (this is akin to a comparator function callback in sorting methods)

Add an Order By node. The order by clause operates on the records that remain after all the filtering is completed.

Right-click the Order By node, and then click New Field.

In the Properties window, set the Field property to DestinationCodeId.

Set the Direction property to Descending.

Right-click the QryGbyHavgRange23 node, and then click Save.

Best Practices for queries:

1. use queries instead of select statements when structure of select is unknown
2. create queries in the application explorer instead of X++ code when possible
3. always use queries as the data sources for forms and reports

Table Indeces

An index is a table-specific database structure that speeds the retrieval of rows from a table

It's up to the database-specific query optimizer to use available indexes to facilitate efficient data retrieval

Indexes are associated with a single table and located in the Application Object Tree (AOT) under the Data Dictionary\Tables node underneath the specific table.

ike most database objects in Microsoft Dynamics AX, indexes are synchronized with the database

An index is defined by one or more fields

The system attempts to order the index according to the first field, and if there is more than one record with the same value in this field, the sorting conflict is resolved by looking at the next field and so on

When selecting table fields for an index consider the following: Fields that are often searched by a range. Fields that frequently participate in joins. Fields that are frequently used to order or group a result set.

There are two types of indexes: unique and non-unique. Whether an index is unique is defined by the index's AllowDuplicates property

set to No, a unique index is created

database uses the unique index to ensure that no duplicate key values occur

creating an index

1. In the AOT, locate the table that you want to add an index to, right-click the Indexes node, and then click New Index.
2. Right-click the new index, and then click New Field.
3. Right-click the field you added in step 3, click Properties, and then select a field from the DataField property list. Repeat steps 2 - 3 to add more fields to the index.

The order of the fields determines the sorting order of the records. If a sorting conflict occurs, the data is sorted on the next field.

Note: Any field of type String is transformed into lowercase.

1. Right-click the new index, and then click Properties.
2. Specify whether the index is unique or non-unique by doing one of the following:

To specify that the index is a non-unique index, confirm that the AllowDuplicates property is set to Yes. It is recommended that you create non-unique indexes for optimum database performance.

Note: If you include a RecId field in an index, the index will be unique. When you include this field and set the property to Yes, the Microsoft Dynamics AX compiler displays a warning.

To specify that the index is unique, set the AllowDuplicates property to No. When the index is unique, you cannot insert records with duplicate key values. A warning is issued to the user if he attempts to insert records with duplicate values.

1. To disable the index, set the Enabled property to No. When you disable an index, it is deleted from the database.

You can also delete an index by right-clicking the index, and then clicking Delete.

1. Modify additional properties as needed.

A primary key is one type of key. The other type of key is an alternate key

There is a maximum of one primary key per table

whereas a table can have several alternate keys

the primary key for every new table is always enforced by an index that has exactly one field

The one field is usually an incremented number or a completely meaningless number that is generated by the system

For new tables the default is a primary key based on the RecId field

This is represented as the surrogate key in the user interface

PrimaryIndex The drop-down list contains the surrogate key plus every index on the table that has its AlternateKey property set to Yes . CreateRecIdIndex This property controls whether the system creates a unique index on the RecId field. The default value is Yes . This is the basis of the surrogate key. No other field is added to this index, not even DataAreaId. ReplacementKey The drop-down list contains every index that has its AlternateKey property set to Yes . You might change the default blank value to an index whose field values within each record provide a name or other moniker that is meaningful to people. If a ReplacementKey is chosen, its fields can appear on forms to helpfully identify each record. The ReplacementKey should be a set of fields that represent the natural key. ClusterIndex The ClusterIndex value is given to the underlying Microsoft SQL Server database system as a performance tuning choice. This choice generally controls the physical sequence in which the records are stored in the underlying database.

A table can have several alternate keys. Any one alternate key can switch to being the primary key, if the alternate key is comprised of only one field

It is more common for a table to reference the primary key of another table, however a table can also reference the alternate key of another table

a unique index alone does not make an alternate key

The AlternateKey property must be set to Yes to make a unique index be an alternate key

AllowDuplicates No means that the combined fields of the index must together make a value in each record which no other record has. AlternateKey Yes means that other tables can create foreign key relations that reference this key, as an alternative to referencing the primary key. Indexes with two or more fields cannot have their AlternateKey property value set to Yes . ValidTimeStateKey A key that is marked as a valid time state key is not a candidate key for child tables to reference in their foreign key relations. Instead, this key is meant for managing date effective data in its own table. The default is No . This field can be Yes only if the ValidTimeStateFieldType property is Yes on the table. Yes means this key contains the ValidFrom and ValidTo fields. The ValidTimeStateKey property cannot be set to Yes when the AlternateKey property is set to No .

a relation represents a foreign key

The following image shows that the AtomStIdx alternate key of the AtomicState parent table is referenced by this foreign key of the AtomicElement child table

foreign key is comprised of the AtomicStateName field

image displays the AtomStIdx alternate key on the AtomicState table. The previous AtomStFkyRel relation references this alternate key

A replacement key is an alternate key that the system can display on forms instead of a meaningless numeric primary key value. Each table can have a maximum of one replacement key

The replacement key is chosen by setting the ReplacementKey property on the table. The drop-down list offers every alternate key as an available value. In the previous image of the AtomicElement table properties, the ReplacementKey property is SymIdx

foreign key In Microsoft Dynamics AX, an AOT node under MyTable > Relations represents a foreign key. For more information, see the previous Relations section in this topic. natural key A key whose value has meaning to people. Most replacement keys are natural keys. surrogate key A key whose value has no meaning to people. A large number generated by the system, such as RecId, could be a surrogate key. unique key A broad term that applies to primary keys and to alternate keys. It does not apply to foreign keys. This term emphasizes that all values for a given key must be unique within one table. All fields in a unique key must be not-nullable.

So indexing in the terms of lets say Django is like when we query a specific instance of an entity using the entities fields. E.g. if we had the entity Car with fields car\_id, model\_name, year\_manufactured, condition and searched one of its instances that had the value of “mustang” for the model\_name field we would have to use typically a self.get() method to query the instance, provided with the field that will be used as basis to look up the instance with the specific value assigned to its field that was used as basis for its look up. So if we had to search for an instance of a Car with the value mustang the query statement would be self.get(model\_name=”mustang”)

A cluster index in Django terms is when we use multiple kwargs that represent the fields that will be used as basis to look up the instance of an entity with specific values assigned to these fields. E.g.

If we had a cluster index in our indexes node in our table object, model\_name, car\_id, year\_manufactured, then we would have to query using these fields in order to retrieve the specific instance of an entity. In Django terms this would be self.get(model\_name=”<somevalue>”, car\_id=<someint>, year\_manufactured=<some date>)

Best practices in using indexing

1. Add an index to a table object if the speed gained by adding it is greater than the cost of updating it
2. Limit the use of too much columns/fields in the index node

**Full text indexes**

**Delete actions**

The DeleteAction element is used to maintain database consistency when a record is deleted. Define delete actions to specify what should occur when data being deleted in the current table is related to data in another table. The delete action values are None, Cascade, Restricted, and Cascade + Restricted.

I assume this is what the on\_delete kwarg does when it is set to models.CASCADE which is essentially as I’ve learned here just a delete action value called cascade. But what is cascade anyway???

The DeleteAction element helps maintain database consistency when a record is deleted

For example, use a cascading delete action to specify that the system is to delete a customer's address when that customer is deleted from the CustTable table

that means taht if we were to delete an entity instance/row from the table which has a field that uses a OneToOneField or a ForeignKeyField (in django terms) with the delete action set to cascade this means that I assume the object with this relation also has its field with that relation deleted

Adding a Delete Action

1. In the Application Object Tree (AOT), expand the Data Dictionary.
2. Expand Tables, and then locate the table that you want to add a delete action to.
3. Click the table, right-click DeleteActions, and then click New DeleteAction.
4. Right-click the new delete action, and then click Properties.
5. Select a related table from the Table property list.
6. Set the DeleteAction property. The following table describes the available values.

Delete Action – Description – Comments

1. None – Delete action disabled
2. Cascade – Deletes related records – Setting the DeleteAction property to Cascade extends the functionality of the table's delete method. As a result, super(), in delete, initiates a cascaded deletion, propagating the delete from table to table. A cascaded delete is implicitly protected by tts. Database changes aren't committed until the entire transaction is complete. E.g. On the CustTable table, a cascading delete action has been defined for the CustBankAccount table. When a customer is deleted from the CustTable table, the delete method also ensures that the corresponding bank account information is automatically deleted.
3. Restricted – Restricts deletion in the current table if data is present in related tables – Setting the DeleteAction property to Restricted extends the functionality of the table's validateDelete method.

As a result, super(), in validateDelete, checks whether records exist on related tables. If records do exist, validateDelete returns false. The forms system ensures that the deletion is not performed. In your own X++ code, check the return value of validateDelete. Don't delete the primary or related records if the method returns false. E.g. On the CustTable table, a restricted delete action has been defined for the CustTrans table. When a customer is deleted in the CustTable table, the validateDelete method ascertains whether transactions exist for the customer in the CustTrans table. If so, validateDelete returns false.

1. Cascade + Restricted – Cascade the delete, even though records exist on related tables. – Setting the DeleteAction property to Cascade + Restricted extends the functionality of the table's validateDelete and delete methods.

As a result, super(), in validateDelete, ascertains whether records exist on related tables. Whether deleting records from forms or X++, if validateDelete returns false, the primary record isn't deleted and the cascading delete isn't performed. You should first delete the records in the related table before deleting the primary record. If the primary record is being deleted as part of a cascading delete, the primary record and the records in the related table will be deleted.

e.g. The Cascade + Restricted delete action is used in the standard application for LedgerJournalTrans on LedgerJournalTable. This type of delete action is useful when you prefer a total clean-up—when you delete a customer, you also delete all the transactions associated with that customer.

**Relations**

Relations define the relationship between two tables that contain related data. Table relations are used to enforce referential integrity among other functions. Table relations are most commonly used in form fields to enable the look-up of information in another table. If a table relation exists, the Lookup button can be used to display a lookup list of values for a particular field

I assume these are akin to the PrimaryKeyField, ForeignKeyField, OneToOneField, ManyToManyField, OneToManyField, in Django

Table relations are most commonly used in form fields to enable the look up of information in another table

If a table relation exists, the lookup button can be used to display a lookup list of values for a particular field

The matching fields typically have the same name in each table.

a SalesOrder table containing orders might have a field called SalespersonID

The Salesperson table, containing the names of sales people, would also have a field called SalespersonID

To create a table relation, specify that the SalesOrder.SalespersonID field is related to the Salesperson.SalespersonID field

Table relationships are created, viewed, and edited in the Application Object Tree (AOT). When a table relation is created in Microsoft Dynamics AX, you must first specify the table involved in the relation and then define the fields in both tables that are related

Conditional relationships can be created by adding a condition to a table relation. Only records that fulfill the condition are included in the relation

Like relations in Django Entities/Models like ForeignKeyField and OneToOneField options for relations in dynamics 365 give the options Relation for OneToOne and ForeignKey Relation for ForeignKey relations

Add a Relation to a Table in the AOT

The initial steps for adding a relation are the same regardless of the relation type that you are adding. The later steps diverge based on the relation type. Remember to save your changes in the AOT.

1. In the AOT, move to Data Dictionary > Tables, and then expand the table that the relation will be added to.
2. Right-click the Relations node, and then select New Relation.
3. Right-click the newly added relation, and then select Properties.
4. Set the name of the new relationship by modifying the Name property.
5. In the Table property, select the related table.
6. Use the Validate property to determine whether the relation should be used to validate data when information is entered into forms.
7. Right-click the new relation, select New, and then click one of the following:
8. Normal to specify relation fields without conditions.
9. Field fixed to specify relation fields to restrict the records in the primary table.
10. Related field fixed to specify relation fields that restrict the records in the related table
11. ForeignKey to specify a correspondence between a foreign key field in the present table to the primary key field in another parent table.

Note: For this option to function, you must first set certain properties on the new relation. A following section explains more about this option.

1. Proceed to the subsection that corresponds to the relation type e.g. Normal, Field fixed etc. that you selected in the earlier step.

Types of OneToOne Relations

Normal Relation

1. In the Field property, select the field in the primary table that relates to a field in the present table.
2. In the RelatedField property, select the field in the related table.

Field fixed Relation

1. In the Field property, select the field in the primary table to use to restrict the records.
2. In the Value property, enter the value of the selected field as the filter. This relates only records in the primary table that match that field value. Only numeric values can be entered in the Value property. Field fixed relations can be created only on numeric fields. Each of the related fields are AND'ed in the table relation.

Related field fixed Relation

1. In the Value property, enter the filter value of the selected field. This causes only records in the related table that match that field value to be related. Only numeric values can be entered in the Value property. Related field fixed relations can be created only on numeric fields.
2. In the Field property, select the field in the related table to restrict the records. Each of the related fields are AND'ed in the table relation.

Types of ForeignKey Relations

ForeignKey Relation

We often use the term child to refer to a table that has a foreign key column. And we use the term parent to refer to the other table that supplies the value for the foreign key column. But we never use the terms parent and child to describe the base and derived tables in an inheritance relationship.

1. Set the Table property to the name of the parent table, the table that contains the primary key field.
2. Set the RelatedTableRole property to a word or phrase that describes the purpose of the parent in the relationship.

Note: This value is added automatically as a method name on the child table. This method is displayed by IntelliSense in the X++ editor, but you cannot see the method in the AOT. For more information about how to use this method, see How to: Use the UnitOfWork Class to Manage Database Transactions.

1. Set the Name property. A helpful value is a combination of the Table property and RelatedTableRole property values.
2. Right-click the node for your relation, click New, and then click ForeignKey. Next click either PrimaryKey based or Single field AlternateKey based. A new field is instantly added to the child table. This field stores the foreign key values.
3. Under the Fields node, click the new field, and then change its Name property value.
4. For performance benefits, you might decide to add an index on the new foreign key field.

Conditional table relations

Define conditional table relations to filter the records in either the primary or the related table. Following are the conditional table relations that can be specified when you define the fields in a table relation:

1. Field fixed
2. Related field fixed

As expected relation types that are otherwise the Normal Relation type will require conditions or constraints for its relation type, which is why when we created a relation it raised an error since we did not indicate that it would be otherwise either a Field Fixed or Related Field Fixed relation type which does require constraints

Relation Type – Format – Description

1. Field fixed – Field fixed (Table.Field == <EnumValue>) – Restricts the records selected in the primary table. Only records that meet the condition are selected. The condition is ANDed with your relation.
2. Related field fixed – Related field fixed (<EnumValue> == Table.Field) – Restricts the records selected in the related table. Only records that meet the condition are selected. The condition is ANDed with your relation.

Take it this way, a relation between two fields that either uses the ForeignKey or OneToOne relation only has a relation if it meets certain constraints

The Orders table has a CollectionTypeID column that stores which collection the order is related to. The possible CollectionTypeID values are defined by the CollectionType enum, which contains the Man, Woman, and Child values. There are conditional relations between the Orders table and each of the three collection tables (MensCollection, WomensCollection, and ChildrensCollection). One of the conditional relations is used according to which collection type the sales associate selects.

Using conditional relations makes it possible to look up information in three different tables from the same field in the Orders table. The table that Microsoft Dynamics AX uses is determined by the CollectionType enum value in the CollectionTypeID column in the Orders table.

e.g. Orders entity/row has CollectionTypeID value of 1, which is part of the CollectionType enum which for instance can be of indeces 1, 2, and 3 represented by MensCollection, WomensCollection, and ChildrensCollection, in C it would be enum CollectionType {MensCollection=1, WomensCollection, ChildrensCollection};

**Temporary Tables**

In Microsoft Dynamics AX, one type of temporary table is the InMemory table. We call them InMemory tables because their TableType property value is InMemory

InMemory tables are instantiated in the active memory of which ever tier the process is running on, either the client or the server tier. InMemory tables are never represented in the database management system. Meaning like variables stored in memory only it will always get deleted once the system is shut down or the app using the variables is closed

An InMemory table is instantiated when the first record is inserted

The instantiated InMemory table continues to exist only while a record buffer variable that references the table exists

The memory or disk space for the InMemory table is de-allocated as soon as the record buffer goes out of scope

To add data to an InMemory table, you must declare the record buffer or a variable based on a specific record/entity instance/row and call the insert method

The following code example uses the TmpCustLedger table which has its TableType property set to InMemory in the AOT.

static void TableTmpInsertRecord(Args \_args)

{

TmpCustLedger custTmpLedger;

;

custTmpLedger.Name = 'NameValue';

custTmpLedger.Balance01 = 2345000;

custTmpLedger.insert();

}

To free the memory and delete the file for the InMemory table, set the record buffer variable to null as follows.

custTmpLedger = null;

Another important difference between InMemory tables and containers is how they are used in method calls. When you pass an InMemory table into a method call, it is passed by reference. Containers are passed by value. When a variable is passed by reference, only a pointer to the object is passed into the method. When a variable is passed by value, a new copy of the variable is passed into the method. If the computer has a limited amount of memory, it might start swapping memory to disk, slowing down application execution. When you pass a variable into a method, an InMemory table may provide better performance than a container.

**Table inheritance**

Properties of a table in order to inherit other tables to create a new table:

Support Inheritance – when this is set to yes then we can also provide a value for the extends property of a table

Extends – when this is set to yes it allows us to use the functionality of the table we are working with whether user-defined or pre-built

Terminology:

A table can extend from or derive from another table. Each table has the SupportInheritance property and the Extends property, which together control table inheritance. I assume we do this much often in pre-built tables since pre-built tables may already have the fields and methods we already need like the models.Model class of django

The default for each new table is to implicitly extend from the Common table. The extension from the Common table cannot be set and seen in the Extends property

In the terminology for table inheritance, we say that the derived table extends its base table

1. ascendant - Class AA is an ascendant of CC.
2. base - Class AA is the base of BB.
3. child - Do not use child when discussing inheritance. This term is often used when discussing foreign key relationships between tables.
4. descendant - Class CC is a descendant of AA.
5. derived - Class BB is derived from AA.
6. extends - In X++, the extends keyword is used in the classDeclaration node of a class in the Application Object Tree (AOT). The extends keyword means that the present class derives from the class that is named on the extends clause.
7. Extends - At AOT > Data Dictionary > Tables > MyTable > Properties, the Extends property is used to derive MyTable from another table.
8. inherits - Class BB inherits from the AA class and the Object class.
9. parent - Do not use parent when discussing inheritance. This term is often used when discussing foreign key relationships between tables.
10. subtype - Class BB and class CC are both subtypes of class AA.
11. supertype - Class BB and class AA are both supertypes of class CC.

When you consider the use of inheritance between two tables, one table is the proposed base table, and the other is the proposed derived table which is the table we will be creating that will use the other table as basis

1. There is no thought that there might be a 1-to-many or many-to-many relationship between the two tables, if they were considered in isolation from all other tables.
2. The row in the proposed base table, and the corresponding row in the derived table, both refer to the same item in the real world. The two rows refer to different attributes about the item.
3. Each row in the proposed base table has exactly one corresponding row in the derived table. If one row is ever deleted from either table, the corresponding row must also be deleted.
4. The base table probably has at least two tables that derive from it. The two derived tables have fields for different kinds of attributes for different kinds of things. The two derived tables refer to different variations of the general items that are tracked together in the base table.
5. No item that is represented in a base table would ever be represented in more than one of its derived tables.

e.g. instead of entities Bird and Dog with fields BirthDate, Name, NumberOfTeeth and BirthDate, Name, BeakColor, respectively

we instead create another entity with fields BirthDate and Name since we now know that we have entities that have common fields and so in order to create a base entity we first have to know which fields, and methods perhaps of entities that we will are universal/common in all these entities, the entities Bird and Dog will inherit from this entity in order to be able to access the BirthDate and Name fields

Creating a Base Table and Derived Table

1. Create Two Tables. In this section you create a base table named TabPet, and you create a derived table named TabPetDog that extends the base table.

in django code

class Pet:

InstanceRelationType = Integer64Field()

BirthDate = DateField(mandatory=true)

Name = CharField()

to set inheritance properties on the tables, in the Properties window for your TabPet base table, set the InstanceRelationType property to InstanceRelationType. In the Properties window for your TabPetDog derived table, set the Extends property to TabPet. and now I understand, that if inheritance of a specific class or in this case a table is set to supportInheritance then other tables that will use it can. But on their end however their Extends property must be set to that of the table they will be inheriting from

**Mappings**

**Methods**

The Methods node displays all the methods available from a table

Add a New Method

1. Browse to the Data Dictionary, Tables node in the AOT.
2. Expand the table, right-click the Methods node, and then select New Method.
3. Enter your code in the Editor window and save your changes.

Override a Method

1. Browse to the Data Dictionary, Tables node in the AOT.
2. Expand the table, right-click the Methods node, and then select Override Method.
3. Select the method that you want to override.
4. Enter your code in the Editor window, and then save your changes.
5. Methods that have been overridden display an icon with an arrow.

Like entities/models in Django that you have created which have the ModelManager class, which have retrieve and create methods we too can override the pre-built methods of a table/entity/model or add a method to according to our needs

**events**

**View**

is an X++ SQL select statement that is given a name that is reusable in other X++ SQL statements

select statement of the view can reference one table, or it can join tables. Also, a view can reference other views, or a mix of views and tables. A view can also reference maps.

A map can unify the access to similar columns and methods that are present in multiple tables

ou associate a map field with a field in one or more tables

enables you to use the same field name to access fields with different names in different tables

Methods on maps enable you to create or modify methods that act on the table fields that the map references

A table name can contain letters and numbers but must begin with a letter. Spaces and special characters are not allowed

**Forms**

design patterns of a form

simple list and customer groups akin to retrieving a query set with objects that meet the constraints of having specific values of their fields using a GET http request method. Setup forms are like setting up a google account, or a bank account etc. which has fields that ask you for certain information, but what form pattern is this under?

setup pages are under table of contents design patterns

sub patterns - applied in a smilar manner as form patterns e.g. fields groups, image preview, fields, and custom filter, (don't forget a filter functionality is implemented through a form even in html, css, javascript). Now I understand; patterns of forms in this context is just the typical or usual ways of how users use forms to create, retrieve, update, and delete data

form properties:

view edit mode property values:

auto

view

edit

RootDatasource.AllowCreate (btw this simply means in OOP that the the our forms node/object can access the object RootDatasource and then has access to its property AllowCreate etc.)

RootDatasource.AllowDelete

Everything just has a visual interface we can now use

form pattern tab

Form patterns

1. Details form - You use the form to view, enter and update an individual record. In addition, the form enables you to perform actions on that record. This is akin to a GET, PUT, PATCH, or DELETE HTTP request which only acts on a single instance of an entity in the database
2. Details form with lines - You use the form to view, enter and update an individual record that is associated with one or more related lines. In addition, the form enables you to perform actions on that record and the lines. Maybe what is meant by lines here is the fields of a specific instance/record/row of an entity
3. Dialog form - You use the form to initiate a task or process where you must provide input. The form enables you to specify whether to continue or cancel the task or process.
4. Simple list and details - You use the form to view a list of records and a details form at the same time. The detail section of the form shows additional fields for the highlighted record in the list.
5. navigation forms on the other hand are used to find information, open forms, and perform actions. For example, you use a list page to find a single record or a collection of records that you want to work with. Unlike other forms, navigation forms open in the content pane of the client workspace.
6. Area page - You use the form to list links to list pages, content pages, forms, reports, classes, jobs, and queries for a module.
7. List page - You use the form to view a list of records. You use the list page to browse records, select one or more records, and perform an action upon the highlighted record or records.
8. Role center - You use the role center to shows a collection of information that is relevant to your Microsoft Dynamics AX role.

Templates

The template generates a new form that has the basic structure and components specified by the design pattern. The template reduces the number of steps that you have to complete to create the new form. To create a form with a template, right-click Forms in the AOT, click New Form from template, and then click the template that specifies the type of form you want to create. For example, you use the AOT to create a form and you click the SimpleListDetails template. If you expand the Design node of the new form, you see that the form already includes several controls. In addition, The Style property in the Design node of that form is set to SimpleListDetails. Like a form or perhaps serializer in django we can specify the properties/class attributes that our form will have in order to dictate its behavior.

1. DetailsFormMaster - Use the template to create a Details form to view, edit, and act on master data.
2. DetailsFormTransaction - Use the template to create details form with lines to view, edit, and act on master data that has line items.
3. Dialog - Use the template to create a dialog window that provides a response to a question.
4. DropDialog - Use the template to create a drop dialog form to perform an action with data.
5. ListPage - Use the template to create a list page you can use to find, analyze, and performs actions on master data.
6. SimpleList - Use the template to create a simple list form to view, edit, and act on dependent or reference data.
7. SimpleListDetails - Use the template to create a simple list and details form to view, edit, and act on dependent and reference data.
8. TableOfContents - Use the template to create a table of contents form to view and edit configuration or setup data.

Components

To complete the form, you add form components that retrieve data, specify how the data is used and displayed, and set security permissions for the form. The following table lists the components you find under each form node in the AOT.

1. Methods - You add or override X++ methods for the form. You can use X++ to customize the appearance and behavior of the form. For more information about form methods, see Methods on a Form.
2. Data Sources - You specify the database query, table, or view that the form uses to retrieve the data that appears in the form. For more information about the form data source, see Form Data Sources.
3. Parts - You add Parts that appear on the form. A Part is a specialized type of control that provides information related to the record that appears in the form. List pages and details forms have a FactBox pane or preview pane where the Parts appear. For more information about Parts, see Parts.
4. Designs - You add the controls that appear on the form to the Design node of the form. For more information about how to add controls, see, Using Controls in a Form Design.
5. Permissions - You specify access levels for the securable objects that appear in the form. For more information about security, see Security Permissions Properties for a Form.

Validation

You use form validation to identify differences between the design of the form you created and the form design pattern best practices for that type of form. The form style best practice tool compares the structure of the form and the values that you specified for form and control properties to a standard form design template. The form style best practice tool uses the value in the Design.Style property of the form to determine the template that is used to validate the form. To use the form style best practice tool, right-click a form in the AOT, click Add-Ins, and then click Check form style best practices. The Form style analysis window opens and lists the form style best practice violations for that form. You can use Fix violation button to quickly fix many design issues with the form. You can also use the Re-analyze button to repeat the validation of the form.

Adding a Data Source to the Form

1. Expand MyNewForm so that the Data Sources and Designs nodes are visible.
2. Press Ctrl + D to open a second AOT.
3. In the second AOT, expand AOT > Data Dictionary > Tables to see the list of tables.
4. From the second AOT, drag AssetTable onto the Data Sources node of the form.

Note: You can also create a form data source by right-clicking the Data Sources node of MyNewForm, and then clicking New DataSource. Use the Table property to select AssetTable.

1. Click the data source for the form. In the Properties window, click Name and type DataSourceAssetTable.

Add a Grid and Fields to the Form

1. Expand the Designs and the Design node of MyNewForm. In the properties window, click Caption and type Assets. The value in the Caption property appears in the titlebar of the form.
2. Click TitleDatasource and select DataSourceAssetTable from the drop-down list.
3. Right-click Design, click New Control, and then click Grid.
4. Click the grid in the Design node. In the Properties window, click Width and select Column width. Click Height and select Column height.
5. Expand the Data Sources > DataSourceAssetTable > Fields node of MyNewForm. Right-click Fields and then click Open New Window. The field list opens in a new AOT window.
6. In the AOT window that list fields, press Ctrl and then click AssetId, Name, SerialNum, and Model. Drag the highlighted fields to the Grid in the Design node of MyNewForm.

Add a Button to the Form

In this section you add a button control to the form. The button shows a list of assets based on the insured value of the asset. To create and show the list, you override the clicked method of the button control.

1. Right-click Design, click New Control, and then click Button.
2. In the Properties window, click Text and type Display Assets.
3. Expand the node for the new button, right-click Methods, click Override method, and then click clicked.
4. Right-click the clicked node, and then click View Code. The clicked method opens in the code editor.
5. Copy the following example code, paste the code into the Editor window.

X++

void clicked()

{

AssetTable assetTable;

while select assetTable order by AssetId

where assetTable.InsuredValue < 1000000

&& assetTable.InsuredValue >= 800

{

info(strFmt("%1 %2", assetTable.AssetId, assetTable.Name));

}

super();

}

1. Click the Save button and then close the editor.
2. To save your changes, right-click the form, and then click Save.

Testing the Form

1. Verify that the Data Sources and Design nodes of the form match what is shown in the following screen shot.
2. To view the form, right-click the MyNewForm, and then click Open. Click the Display assets button on the form. The form and the code output to the Infolog are shown in the following image.

**Form pattern tab**

Form design layout

container - description

Form.Design - The root of the page. It functions as a special kind of container

group - the general-purpose container control in MS Dynamix AX. Group controls can be nested as required

tab - A control that contains TabPage controls and has many possible Tab.Style values, such as Tab, FastTab, Vertical, Tab, Droplist and Panorama

tabPage - The appearance of each TabPage control depends on its Tab.Style value

buttonGroup - A special type of Group control that contains buttons

action pane tab - composed of buttons

e.g. button, command button, menu item button, drop dialog button, menu button

toolbars - represented by add, edit, map buttons. Previously known as the "action pane strip" (but what is the difference of this to the "action pane tab"?)

we can add buttons via a button group object

grid style - is set through the property of what object though? But the property is style that can takes in values Auto, Tabular, SimpleReadOnly, and List. Maybe this is akin to css grid display. Tabular Grid style is comprises of columsn and rows, but a List Grid style is composed only of rows and at most 1 column

Form

* needs data source which is a table
* needs a design pattern
* a design pattern can be implemented via a template so no code is repeated or done from scratch
* when ever we choose a pre-built design pattern by clicking the apply pattern option this will require certain objects also too in order to build the form pattern
* And objects under these form patterns like tabs, grids, lists, button groups, etc. which are added through the design pattern node also can be added the fields which are based on the EDTs we created earlier
* Forms can only have one design pattern, therefore like objects that we retrieve from our database through our view functions in Django which maybe a list of all objects or list of a single specific object. We make individual forms for these functionalities

Synchronizing database with project

Serving the project locally

**Menu**

Menus

* take in forms

menu parameters

menu extension

menu item type, menu item name, configuration key, normal range, label

Menu items

menu item parameters

menu item buttons

form view option, normal image, open mode, help text, label are the properties of a menu item

enum parameter, enum type parameter, linked permission object, linked permission type are the properties of an output menu item

display menu items

object property takes in the created form we have

action menu items

could this also have an object property that takes in a form or the http response when we click this menu item?

create permissions, delete permission, extended data security, configuration key,

output menu items

could this also have an object property that takes in a form or the http response when we click this menu item?

enum parameter, enum type parameter, linked permission type are the properties of an output menu item

extending a menu

OrganizationAdministration -> resources -> menu -> menu items -> form

Design pattern of form -> menu button -> menu item type -> created menu item

**Exception Handling**

Exception handling

file: DBFMVehicleEntry.xpp

DBFMVehicleTableTest vehicle\_table;

public void run()

{

this.setupDialog();

Dialog.run();

Dialog.wait();

if(Dialog.closeOk())

{

this.getFieldValues();

if(vehicle\_table.validateWrite())

{

vehicle\_table.insert();

DBFMVehicleEntry::tableInsertSuccess();

}

}

}

public void getFieldValues()

{

str make, model, VIN;

int vehicle\_id;

DBFMFuelLevel fuel\_level;

make = df\_make.value();

model = df\_model.value();

VIN = df\_vin.value();

fuel\_level = df\_fuel\_level.value();

try

{

if(strlen(VIN) < 18)

{

throw error("Invalid VIN. Cannot be fewer than 18 characters. Appending characters");

}

else

{

}

}

catch{

// see how many 0s are needed for the VIN to be valid

// e.g. VIN is 7 in length missing 11 0s

for(int i = 0; i < 18 - strlen(VIN); ++i)

{

VIN += "0";

}

retry;

}

vehicle\_table.DBFMMakeTest = make;

vehicle\_table.DBFMModelTest = model;

vehicle\_table.DBFMVINTest = VIN;

vehicle\_table.DBFMFuelLevelTest = fuel\_level;

}

// if VIN is less than 18 characters throw an error but retry the try block and append now the missing characters through 0s for the VIN field

public void setupDialog()

{

// to add

df\_fuel\_level = Dialog.addField(extendedTypeStr(DBFMFuelLevelTest), "Fuel level");

}

DBFMVehicleTable.xpp

public class DBFMVehicleTable extends common

{

public void initValue()

{

super();

}

public void insert()

{

super();

}

// method runs when modifying a field, takes in a field that was modified

public void modifiedField(Field \_fieldId)

{

super(\_fieldId);

switch(\_fieldId)

{

// fieldNum(<table name>, <table field>)

case fieldNum(DBFMVehicleTableTest, DBFMServiceTypeCodeTest):

info("service type field is updated");

break;

case fieldNum(DBFMVehicleTableTest, DBFMSeatingCapacityTest):

info("seating capacity field is updated");

break;

case fieldNum(DBFMVehicleTableTest, DBFMModel):

info("model field is updated");

break;

}

}

public boolean validateWrite()

{

boolean is\_valid = super();

if(this.DBFMFuelLevel < 10)

{

is\_valid = false;

error("fuel level must be greater than or equal 10");

}

return is\_valid;

}

}

file: CustCreateCustomer.xpp

public void run()

{

CustTable cust\_table;

Currency curr\_table;

Dialog dialog = new Dialog("Create new customer");

DailogField df\_cust\_acc = Dialog.addField(extendedTypeStr(CustVendAc), "Account number");

DialogField df\_group = Dialog.addField(extendedTypeStr(CustGroupId), "Group");

DialogField df\_credit\_rating = Dialog.addField(extendedTypeStr(CustCreditRating), "Credit rating");

const int MAX\_RETRIES = 5;

int retry\_count = 0;

if(dialog.run()){

try

{

cust\_table.AccountNum = df\_cust\_acc.value();

cust\_table.CustGroup = df\_group.value();

cust\_table.CreditRating = df\_credit\_rating.value();

if(!cust\_table.validateWrite())

{

// this will throw an error since the currency field is not supplied a value;

throw error("Please enter all required fields");

}

else

{

cust\_table.insert();

}

}

catch(Exception::Error)

{

// check if Currency field is indeed blank

if(!cust\_table.Currency)

{

select firstonly curr\_table;

cust\_table.Currency = curr\_table.CurrencyCode;

if(retry\_count <= MAX\_RETRIES)

{

++retry\_count;

retry;

}

}

else

{

error("An error occured. please try again");

}

}

}

}

**Class, Form, Table, Menu extensions**

next keyword is akin to super() method to call the original method of the parent class

next cannot also be written inside conditional statements scopes

The next keyword behaves like a super, and it will define when your extended logic executes

The next call after your code behaves like a Pre-event handler. Your logic executes first, and later on, the logic residing in the original method gets executed

* COC enabled strongly typed extension capabilities of public and protected methods
* it allows technical consultants to extend the application avoiding over-layering
* to use COC you must declare your class as final and your methods should always contain the next keyword
* only public and protected methods are allowed to be extended
* you can add custom logic that will run before and/or after the standar dcode runs
* you can access the protected and public methods, and variables of the base class
* you can only extend objects that are part of the referenced packages of your model. This is done through clicking the model parameters under the dynamics 365 menu, and updating the model parameters, the once in the window select the DevelopmentBasicsFMS option, next check the model we are trying to reference always e.g. CustTable for instance. This works like an import statement which adds the model that we check to our current project directory which we can always access at will
* methods with [Hookable(false)] or [Wrappable(false)] attribute cannot be extended
* methods that use the final keyword cannot be wrapped in an extension class
* class name must have an \_Extension suffix
* class declaration needs to ouse the final keyword
* add the ExtensionOf attribute on the class
* use COC to extend the following: table, class, form, etc.
* the next keyword is required. This will determine on which part of the code the base class will be called

we can also use other built or even pre-built objects like tables and forms in creating extensions of them e.g. [ExtensionOf(formStr(CustForm))] or [ExtensionOf(tableStr(CustTable))]

we can override methods of the tables we created and pre-built tables by right clicking the methods node of a table and navigating to the override method and selecting the method that we will override

this will then add our overriden method in this node and we can edit this method through x++ code

COCBaseClass.xpp

class COCBaseClass

{

public void new()

{

//

}

protected void test()

{

info("base class call");

}

public void runTest()

{

this.test();

}

}

COCChildClass.xpp

class COCChildClass extends COCBaseClass

{

// override test() method of parent class

protected void test()

{

info("child class before super");

// super() calls the original overriden method of the parent class

// where ever super() is scoped or the method it is currently in (usually an overriden method)

// super() assimilates the parent method of that of the overriden method it is inside of

// and returns (or not) its respective value, in this case the test() method is overriden

// and is the method it is inside in, therefore call the parent test() method

super();

info("child class after super");

}

}

COCChildClass\_Extension.xpp

[ExtensionOf(classStr(COCChildClass))]

final class COCCHildClass\_Extension

{

protected void test()

{

info("extension 1 before next call")

next test();

info("extension 1 after next call")

}

}

**Class extensions**

DBFMBaseClass.xpp

class DBFMBaseClass

{

public int curr\_state;

public int permState(int \_value=curr\_state)

{

curr\_state = \_value;

return curr\_state;

}

public void displayState()

{

info(strFmt("DBFMBaseClass state: %1", this.permState()));

}

}

DBFMBaseClass\_Extension.xpp

[ExtensionOf(classStr(DBFMBaseClass))]

final class DBFMBaseClass\_Extension

{

public void setCustomState(int cust\_state)

{

this.permState(cust\_state);

}

public void displayState()

{

warning("from extension class");

next displayState();

warning("from extension class");

}

}

DBFMChildClass.xpp

class DBFMChildClass extends DBFMBaseClass

{

public void displayState()

{

info("child class before super()");

super();

info("child class after super");

}

}

DBFMChildClass\_Extension.xpp

[ExtensionOf(classStr(DBFMChildClass))]

final class DBFMChildClass\_Extension

{

public void displayState()

{

info("Child class extension before next");

next displayState();

info("Child class extension after next");

}

}

DBFMExtendedClasses.xpp

class DBFMExtendedClasses

{

public static void main(Args \_args)

{

// we still call the normal name of a now extended class

DBFMBaseClass bc = new DBFMBaseClass();

bc.displayState();

bc.setCustomeState(10000);

bc.displayState();

DBFMChildClass cc = new DBFMChildClass();

cc.displayState();

cc.setCustomeState(50000);

cc.displayState();

}

}

**Table Extensions**

Example 1:

CustTableDBFM\_Extension.xpp

[ExtensionOf(tableStr(CustTable))]

final class CustTableDBFM\_Extension

{

void insert(DirPartyType \_partyType, Name \_name, boolean \_updateCRM)

{

if(str2Int(this.creditRating) < 200)

{

this.Blocked = CustVendorBlocked::All;

info("The customer has been placed on hold du to low credit rating. Please review");

}

next insert(\_partyType, \_name, \_updateCRM);

}

}

Example 2:

DBFMVehicleTable.xpp

public class DBFMVehicleTable extends common

{

public void initValue()

{

super();

}

public void insert()

{

super();

}

// method runs when modifying a field, takes in a field that was modified

public void modifiedField(Field \_fieldId)

{

super(\_fieldId);

switch(\_fieldId)

{

// fieldNum(<table name>, <table field>)

case fieldNum(DBFMVehicleTableTest, DBFMServiceTypeCodeTest):

info("service type field is updated");

break;

case fieldNum(DBFMVehicleTableTest, DBFMSeatingCapacityTest):

info("seating capacity field is updated");

break;

case fieldNum(DBFMVehicleTableTest, DBFMModel):

info("model field is updated");

break;

}

}

public boolean validateWrite()

{

boolean is\_valid = super();

if(this.DBFMFuelLevel < 10)

{

is\_valid = false;

error("fuel level must be greater than or equal 10");

}

return is\_valid;

}

}

Example 3:

DBFMVehicleEntry.xpp

class DBFMVehicleEntry

{

DBFMVehicleTableTest vehicle\_table;

Dialog dialog;

DialogField df\_make, df\_model, df\_VIN, df\_vehicle\_id, df\_fuel\_level, df\_year;

public static void main(Args \_args)

{

DBFMVehicleEntry ve = new DBFMVehicleEntry();

ve.run();

}

public void run()

{

this.setupDialog();

Dialog.run();

Dialog.wait();

if(Dialog.closeOk())

{

this.getFieldValues();

if(vehicle\_table.validateWrite())

{

vehicle\_table.insert();

DBFMVehicleEntry::tableInsertSuccess();

}

}

}

public void getFieldValues()

{

str make, model, VIN;

int vehicle\_id;

DBFMFuelLevel fuel\_level;

make = df\_make.value();

model = df\_model.value();

VIN = df\_vin.value();

fuel\_level = df\_fuel\_level.value();

try

{

if(strlen(VIN) < 18)

{

throw error("Invalid VIN. Cannot be fewer than 18 characters. Appending characters");

}

else

{

}

}

catch{

// see how many 0s are needed for the VIN to be valid

// e.g. VIN is 7 in length missing 11 0s

for(int i = 0; i < 18 - strlen(VIN); ++i)

{

VIN += "0";

}

retry;

}

vehicle\_table.DBFMMakeTest = make;

vehicle\_table.DBFMModelTest = model;

vehicle\_table.DBFMVINTest = VIN;

vehicle\_table.DBFMFuelLevelTest = fuel\_level;

}

public setupDialog()

{

vehicle\_table.init\_value();

df\_make = Dialog.add(extendedTypeStr(DBFMMakeTest), "make");

df\_model = Dialog.add(extendedTypeStr(DBFMModelTest), "model");

df\_VIN = Dialog.add(extendedTypeStr(DBFMVINTest), "vehicle identification number (VIN)");

df\_fuel\_level = Dialog.add(extendedTypeStr(DBFMFuelLevelTest), "fuel level");

df\_year = Dialog.add(extendedTypeStr(DBFMYearTest), "year");

df\_vehicle\_id = Dialog.add(extendedTypeStr(DBFMModelTest), "model");

df\_vehicle\_id.value(vehicle\_table.DBFMVehicleIdTest);

df\_vehicle\_id.allowEdit(false);

df\_year.value(vehicle\_table.DBFMYear);

}

}

DBFMVehicleTable.xpp

public class DBFMVehicleTable extends common

{

public void initValue()

{

this.DBFMYearTest = today();

this.DBFMVehicleIdTest = this.getMaxVehicleId() + 1;

super();

}

public void insert()

{

super();

}

// method runs when modifying a field, takes in a field that was modified

public void modifiedField(Field \_fieldId)

{

super(\_fieldId);

switch(\_fieldId)

{

// fieldNum(<table name>, <table field>)

case fieldNum(DBFMVehicleTableTest, DBFMServiceTypeCodeTest):

info("service type field is updated");

break;

case fieldNum(DBFMVehicleTableTest, DBFMSeatingCapacityTest):

info("seating capacity field is updated");

break;

case fieldNum(DBFMVehicleTableTest, DBFMModel):

info("model field is updated");

break;

}

}

public boolean validateWrite()

{

boolean is\_valid = super();

if(this.DBFMFuelLevel < 10)

{

is\_valid = false;

error("fuel level must be greater than or equal 10");

}

if(this.DBFMYearTest > today())

{

is\_valid = false;

error("year must not be greater than current date")

}

return is\_valid;

}

private DBFMVehicleIdTest getMaxVehicleId()

{

return (select maxof(DBFMVehicleIdTest) from vehicle\_table).DBFMVehicleIdTest;

}

}

Example 4:

DBFMVendTable\_Extension.xpp

[ExtensionOf(tableStr(VendTable))]

final class DBFMVendTable\_Extension

{

public void initValue()

{

VendGroup vendGroup;

// ah ok because fields tables have internally no values

// then the initValue() method is called to initialize the fields to these values

// if there is no setting of any field then we can place next initValue()

// either at the top or at the bottom

this.VendGroup = (select firstonly VendGroup from vendGroup).VendGroup;

next initValue();

// remember we can now dictate the behavior of the Vendtable because we have already extended it,

// so this is the class it now uses overall and the initValue() of the base class is not anymore

// the initValue() method being called but rather this declaration in our extension

}

}

Example 5:

* So add FMRental table to our project
* add the vehicle id of our DBFMVehicleTableTest to the fields
* add a primary key relation
* use set the properties like assocaiation, cardinality, relationship type and on delete to its respective values
* add a normal field to the relation and set field to vehicle id
* set the related field to the vehicle id as well

file: DBFMVendTable\_Extension.xpp

[ExtensionOf(tableStr(VendTable))]

final class DBFMVendTable\_Extension

{

public void initValue()

{

VendGroup vendGroup;

// ah ok because fields tables have internally no values

// then the initValue() method is called to initialize the fields to these values

// if there is no setting of any field then we can place next initValue()

// either at the top or at the bottom

this.VendGroup = (select firstonly VendGroup from vendGroup).VendGroup;

next initValue();

// remember we can now dictate the behavior of the Vendtable because we have already extended it,

// so this is the class it now uses overall and the initValue() of the base class is not anymore

// the initValue() method being called but rather this declaration in our extension

}

[SysClientCacheDataMethodAttribute(true)]

display str customVendGroup()

{

return "DBFM\_" + this.VendGroup

}

}

* create an extension of the VendTable form
* go to the pattern tab of the VendTable form and under the MainTab sub pattern di ko na alam kung ano add a string which will represent our custom group
* set the properties data method to the method that we made which is customVendGroup under our extended VendTable, DBFMVendTable\_Extension
* set hte data source to our VendTable
* set the label to "Custom Group"
* add vehicle id field from DBFMVehicleTableTest on VendTable, and FMRental tables (do this by creating the extension first)
* Add relationship to DBFMVehicleTable cardinality: zeroone, related table cardinality: exactlyone, relationship type: association
* set label of VendGroup field on VendTable to "Vendor group" by using label files
* open table browser of VendTable to check the additional field
* open all vendors to check the updated label

**Delegates**

Delegates allow us to access elements in higher models from a lower model. Below is a model heirarchy

Test

Application suite

Application foundation

Application platform

exhibit a: application foundation needs something from suite, suite needs to subscribe to application foundation

(1)delegate (2)void applyDiscountDelegate(real \_receiptTotal, EventHandlerResult \_result)

{

(3)

}

this is for application suite:

[SubscribesTo(classStr(Simpletax), delegateStr(SimpleTax, applyDiscountDelegate))]

public static void applyDiscountDelegateHandler(real \_receiptTotal, EventHandlerResult \_result)

{

real discountedTotal = \_receiptTotal \* (1 - DiscountRate);

\_result.result(discountedTotal);

}

for application foundation to use application suite:

public real calculateTotalTax()

{

real totalTax;

EventhandlerResult result;

this.applyDiscountDelegate(this.ReceiptTotal, result);

this.ReceiptTotal = result.result();

totalTax = this.ReceiptTotal \* this.TaxRate;

return totalTax;

}

**Event handling**

is exactly like the performCreate(), onDelete(), onInsert(), componentDidMount(), componentWillMount(), events and methods that act as events in the languages you've learned like JavaScript, and Python

we can override an event handler by creating a class

copy the onInserting event in the CustTable table that we have included in our project via importing

create a regular class named DBFMCustTableHandler and paste the onInserting event that we copied in the class

* import FMRental forms by checking the FleetManagement package/model
* copy event handler onInitialized
* create another class and paste the onInitialized method
* other events are also nested in the data sources node, besides the root form node itself like OnModified, OnActivated

Example 1:

file: DBFMFMRentalHandler.xpp

class DBFMFMRentalHandler

{

// under: FMRental

[FormEventHandler(formStr(FMRental), FormEventType::Initialized]

public static void FMRental\_OnInitialized(xFormRun sender, FormEventArgs e)

{

info("initialized form event handler");

FormRun form\_run = sender;

// access this like a tree with its nodes and subnodes

// to gain access to the field of the form "value" itself

FormControl fmrental\_state\_copy\_1 = form\_run.design().controlName(formControlStr(FMRental\_StateCopy1));

if(fmrental\_state\_copy\_1.valueStr() == "Complete")

{

fmrental\_state\_copy\_1.enabled(false);

}

info(strFmt("state is initialized as: %1", fmrental\_state\_copy\_1.valueStr()));

}

// under: FMRental\_StateCopy1 < InfoGroup < LineViewHeader < LineViewTab < LineView < TabDetails < TabPageDetails < Tab < FMRental Design Pattern

[FormControlEventHandler(formControlStr(FMRental, FMRental\_StateCopy1), FormControlEventType::Modified]

public static void FMRental\_StateCopy1\_OnModified(FormControl sender, FormControlEventArgs e)

{

//FMRental\_StateCopy1 fmr\_sc = sender;

// when valueStr() has value "complete" set the form to disabled

if(sender.valueStr() == "Complete")

{

sender.enabled(false);

}

info("state %1 has been modified", sender.valueStr());

}

// under: events < FMRental < Data Sources < FMRental

[FormDataSourceEventHandler(formDataSourceStr(FMRental, FMRental), FormDataSourceEventType::Activated)]

public static void FMRental\_OnActivated(FormDataSource sender, FormDataSourceEventArgs e)

{

FormRun form\_run = sender.formRun();

// access this like a tree with its nodes and subnodes

// to gain access to the field of the form "value" itself

FormControl fmrental\_state\_copy\_1 = form\_run.design().controlName(formControlStr(FMRental\_StateCopy1));

/\* if(fmrental\_state\_copy\_1.valueStr() == "Complete")

{

fmrental\_state\_copy\_1.enabled(false);

}

else

{

fmrental\_state\_copy\_1.enabled(true);

} \*/

/\* // shorthand of the above

fmrental\_state\_copy\_1.enabled(fmrental\_state\_copy\_1.valueStr() == "Complete" ? false : true);

info(strFmt("state is initialized as: %1", fmrental\_state\_copy\_1.valueStr())); \*/

FMRental fm\_rental = sender.cursor();

FormDataObject fm\_rental\_state = sender.object(fieldNum(FMRental, State));

fm\_rental\_state.enabled(fm\_rental.State != FMReservatioState::Complete);

info(strFmt("state is initialized as: %1", fm\_rental.State));

}

// under: events < state < fields < FMRental < data sources < FMRental

[FormDataFieldEventHandler(formDataFieldStr(FMRental, FMRental, state), FormDataFieldEventType::Modified)]

public static void State\_OnModified(FormDataObject sender, FormDataFieldEventArgs e)

{

// because the field is an enum we must compare the field\_val

// to an enum value

anytype field\_val = sender.getValue();

/\* if(sender.getValue() == FMReservationState::Complete)

{

sender.enabled(false);

}

else

{

sender.enabled(true);

} \*/

sender.enabled(sender.getValue() != FMReservationState::Complete);

info(strFmt("state has been modified to: %1", sender.getValue()));

}

}

* to do this we need to go to a specific entity instance and disable the field there. Because we dont have a mentioned specific instance, we are stuck in the form design itself
* we need to access a specific instance, and what instance is it, yes the instances only with fields set to Complete
* [<node all words with capitalized first letters>EventHandler(<node all words with capitalized first letters except first word with first lowercase first letter>Str(<...args>), <node all words with capitalized first letters>EventType::<EventType>)]

public static void <naem of the event handler or event to be overriden>(<...args>)

{

}

Example 2:

* using event handlers disable field "Group" on all vendors form if "Group" has a value, this means that even in cases of inserting a record, updating a record, upon the loading of a record the records must be checked to see if the their group fields are blank
* disable field "UPS zone" on all vendors > invoice and delivery > Delivery if "UPS zone" has a value

file: DBFMVendHandler.xpp

class DBFMVendHandler.xpp

{

[FormDataSourceEventHandler(formDataSourceStr(VendTable, VendTable), FormDataSourceEventType::Activated)]

public static void VendTable\_OnActivated(FormDataSource sender, FormDataSourceEventArgs e)

{

VendTable vend\_table = sender.cursor();

FormDataObject vend\_table\_fzone = sender.object(fieldNum(VendTable, FreightZone));

vend\_table\_fzone.enable(!vend\_table.FreightZone);

}

}