May 22

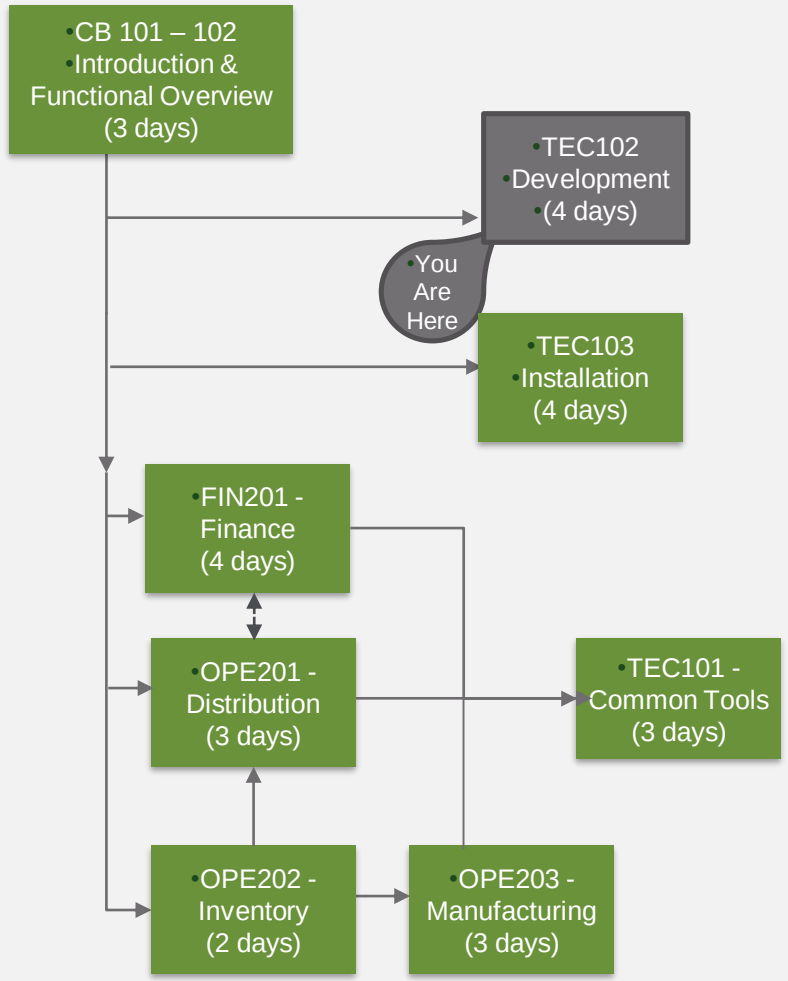
Sage ERP X3

Training

TEC102 - Development - Fundamentals

sage

Sage ERP X3 Fundamentals



•Sage ERP X3 Advanced

Distribution
- Contracts (Purchasing and Sales)
- Loan Orders
- Packaging & Shipment Preparation
- Purchasing Signature Management
- Sales and Purchasing Prepayments
- Sales Reps and Commissions
- Sourcing: RFQs, Responses
- Carrier Management
- Inter-Site and Intercompany Transactions
- Invoice Elements (Purchasing & Sales)
- Kits, Options & Variants
- Price List Management (Advanced)

Manufacturing (4 days)
- Advanced Global Manufacturing
- Production Costing and WIP Posting

Finance
- Bank Communication & Reconciliation
- Budget Control/ Analytical Budget
- Financial Data Extraction
- Statements & Reminders
- Automatic Journals & Dimension Defaults
- Advanced Payment Transactions
- Month and Year End Processes
- Fixed Assets

Inventory (3 days)
- Allocation Rules (Advanced)
- Stock Counts
- Serial Number Management
- Reodering Replenishment Storage Plan
- Valuation Methods and Price Adjustments and cost calculations
- Quality Control & Sampling

Tools & Development
- Advanced Common Tools
- Advanced Development

•Sage ERP X3 Expert

Distribution and Inventory
- Purchasing Subcontracts
- Pre-Allocations (Pegging Function)
- Radio Terminals

Manufacturing (5 days)
- Configurator
- Weighing
- Optimisation ILOG

Finance
- Multi-Legislation Set Up
- Operating Budgets
- Analytical Allocations
- Factoring

Tools & Development
- Connectors
- Java Bridge Serveur Components
- Eclipse
- LDAP integration
- Interface Import/Export
- Netvibes
- Crystal Reports

Course Map

•TEC102 - Development - Fundamentals

Objectives

At the end of this training, you will be able to :

- Get familiar with the X3 development framework
- Start using the different fundamentals tools
- Reach a first level of skills to be autonomous

Contents

- 1. Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- 3. Introduction To The Sage X3 Language
  - 3.1 – Variables And Variable Classes
  - 3.2 – Operators
  - 3.3 – Instructions
  - 3.4 – Functions
  - 3.5 – System Variables
  - 3.6 – Functions, Subprograms And Scope
  - 3.7 – Reusable Subprograms
  - 3.8 – Performances
- 4. Basic Development: The Object Template
  - 4.1 – The Object Template
  - 4.2 – Field Actions
  - 4.3 – Object Actions
  - 4.4 – STD snippet: Database transaction management
- 5. Debugging Sage X3 Code
  - 5.1 – The Debugger

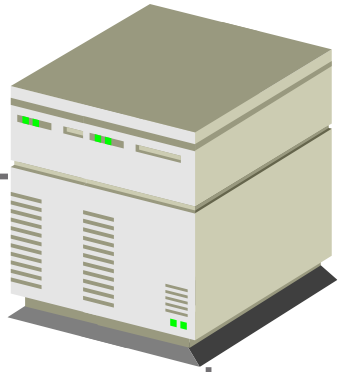
Contents

- 1. Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- 3. Introduction To The Sage X3 Language
  - 3.1 – Variables And Variable Classes
  - 3.2 – Operators
  - 3.3 – Instructions
  - 3.4 – Functions
  - 3.5 – System Variables
  - 3.6 – Functions, Subprograms And Scope
  - 3.7 – Reusable Subprograms
  - 3.8 – Performances
- 4. Basic Development: The Object Template
  - 4.1 – The Object Template
  - 4.2 – Field Actions
  - 4.3 – Object Actions
  - 4.4 – STD snippet: Database transaction management
- 5. Debugging Sage X3 Code
  - 5.1 – The Debugger

1.1 – General Architecture

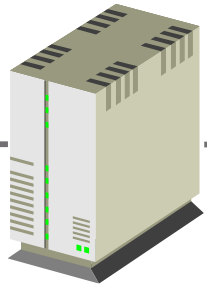
Client/Server/Web Native Architecture

Client Workstation
(Windows XP/NT/2k)



Database:
Oracle 9i
SQL Server 2000

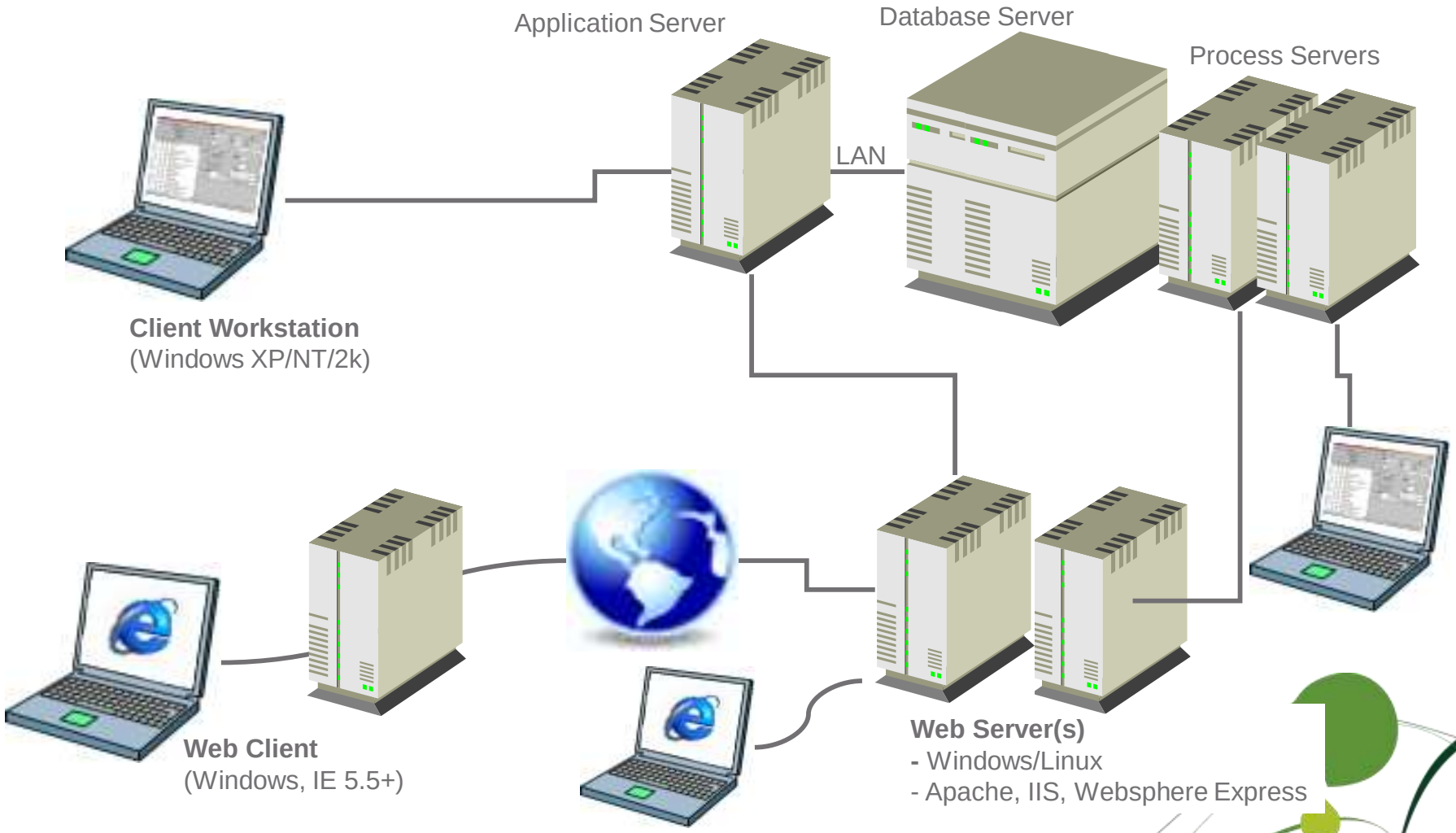
X3 Server(s)
- Windows NT/2k/2003
- Unix (AIX/HP-UX)
- Linux (Redhat/Intel)



Web Server(s)
- Windows/Linux
- Apache, IIS, Web sphere Express

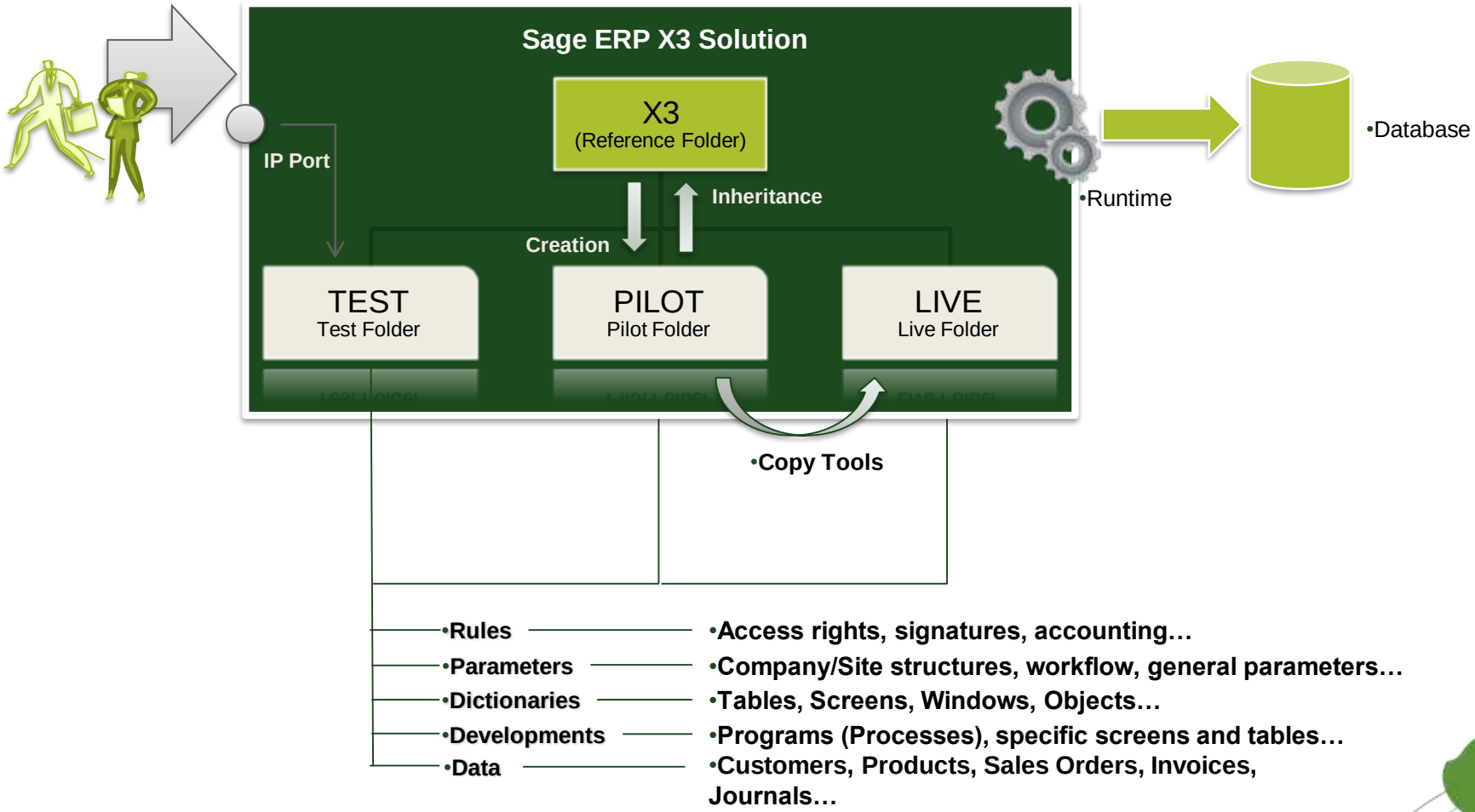
Web Client
(Windows, IE 5.5+)

Client/Server/Web Native Architecture

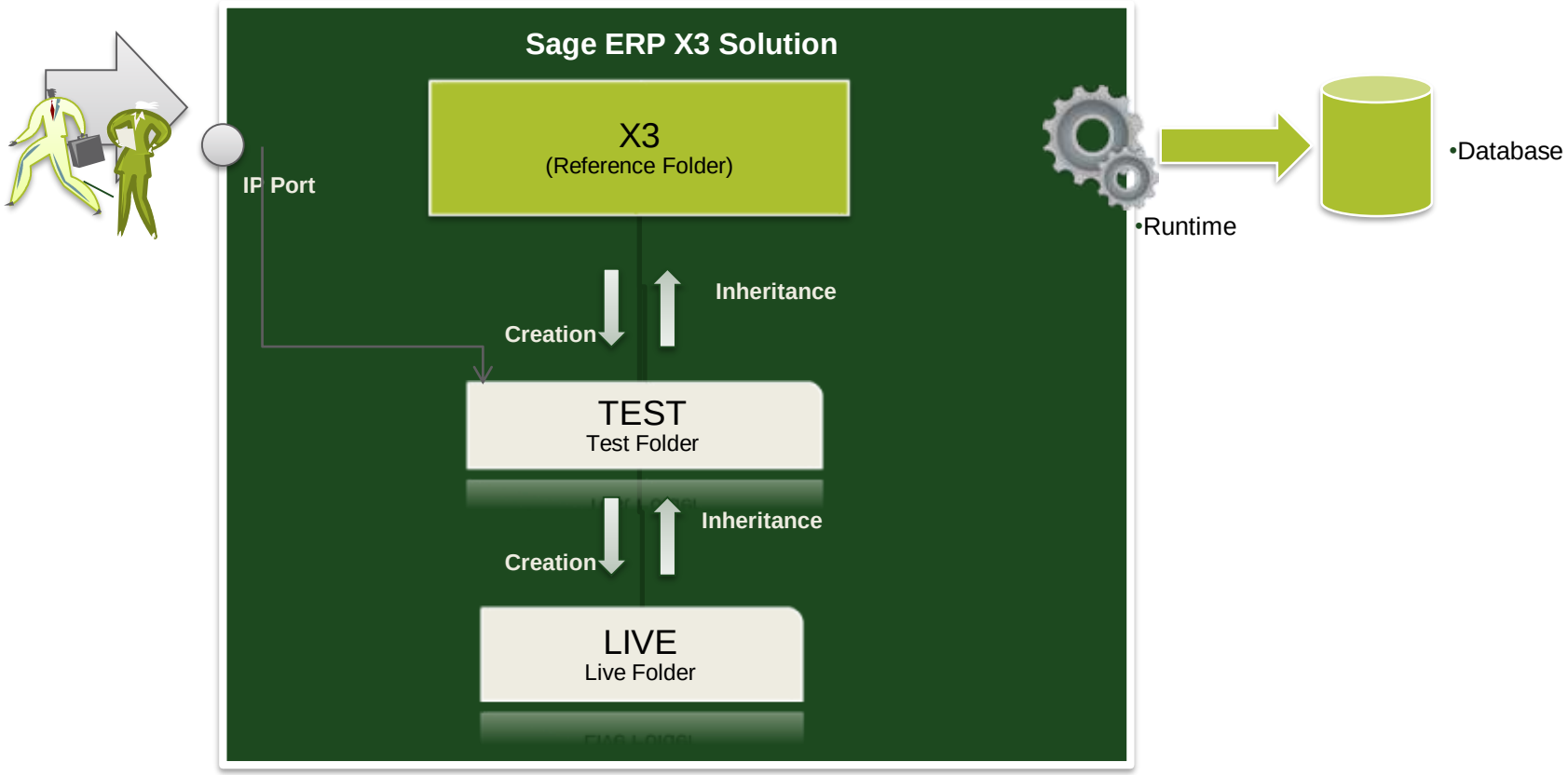


1.2 – Folder Management

Solutions and Folders (2 tier)

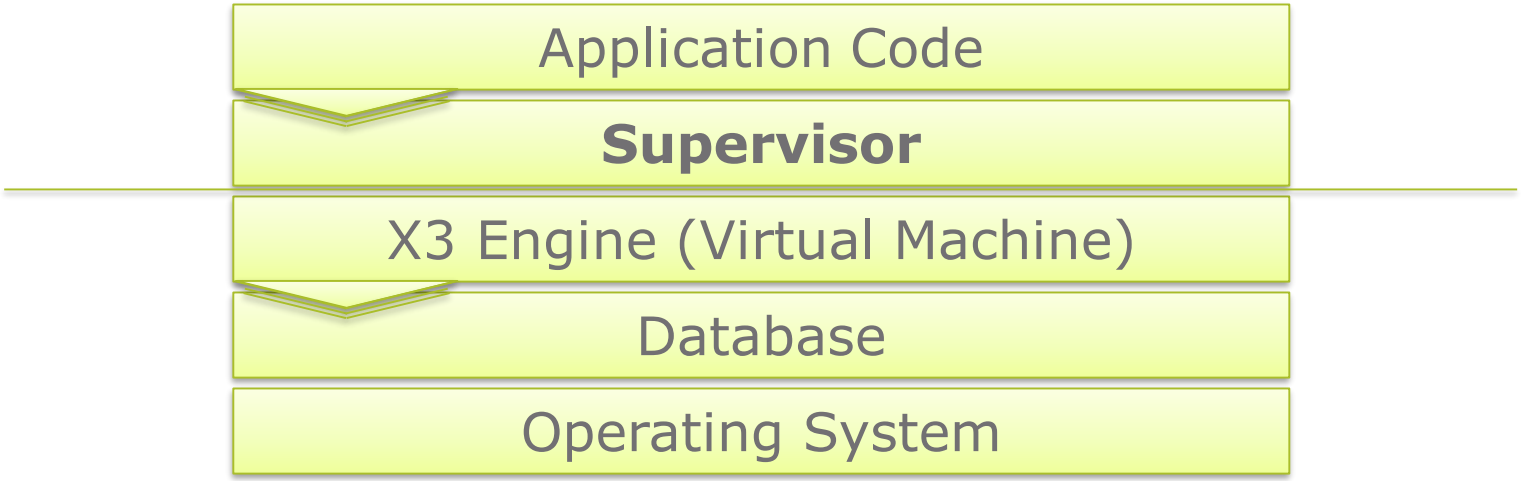


Solutions and Folders (3 tier)



1.3 – Application Architecture

The Supervisor As An Application Layer



A set of Global Services

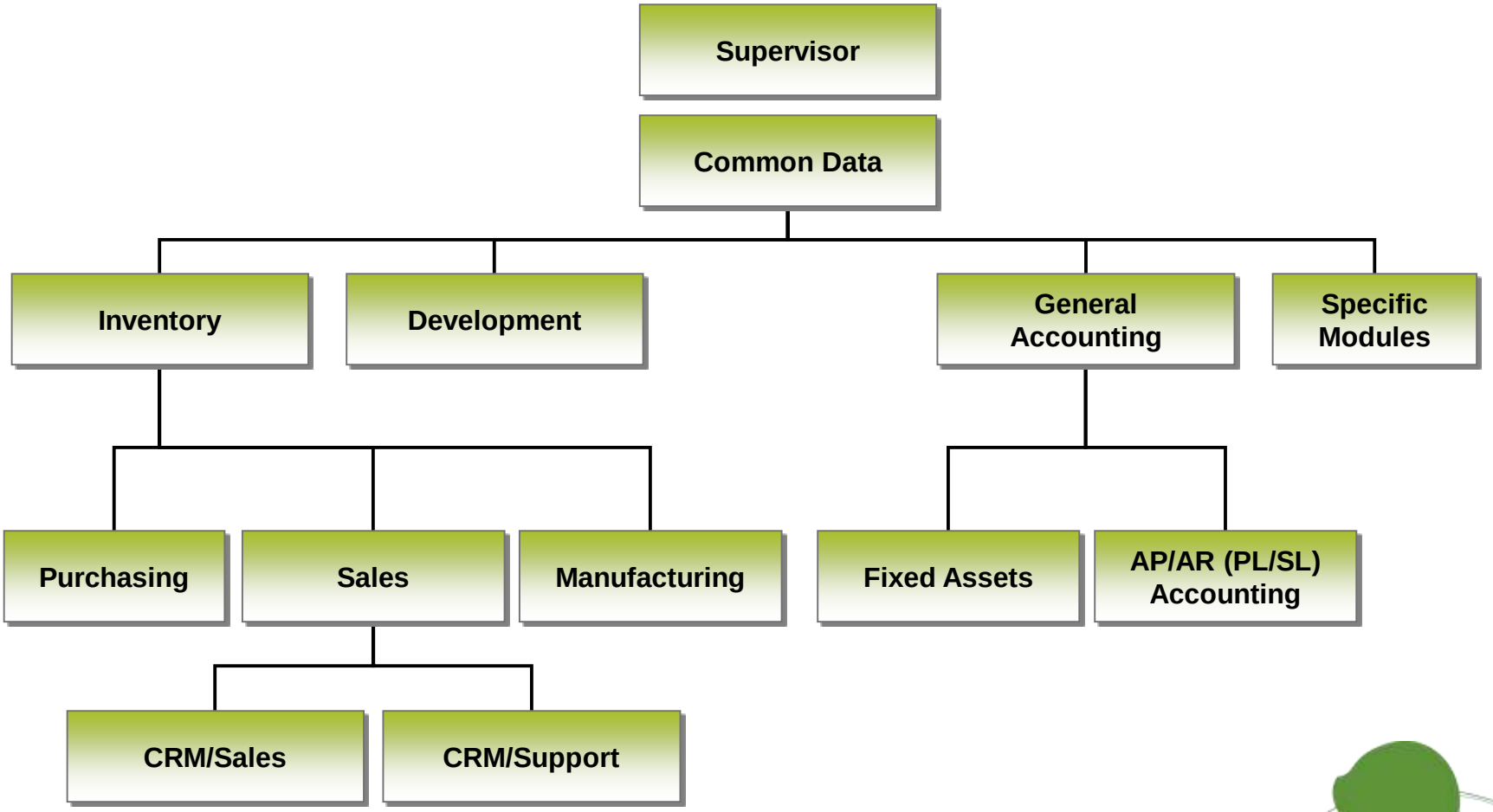
Supervisor

- Database and Configuration
- User and Access Rights
- Data Inquiry Management (Statistics, Reports, Data Access…)
- Application Parameters
- Batch Job Management
- Version & Upgrade Management
- Development Platform for Specifics and Verticals

Sage X3 Functions

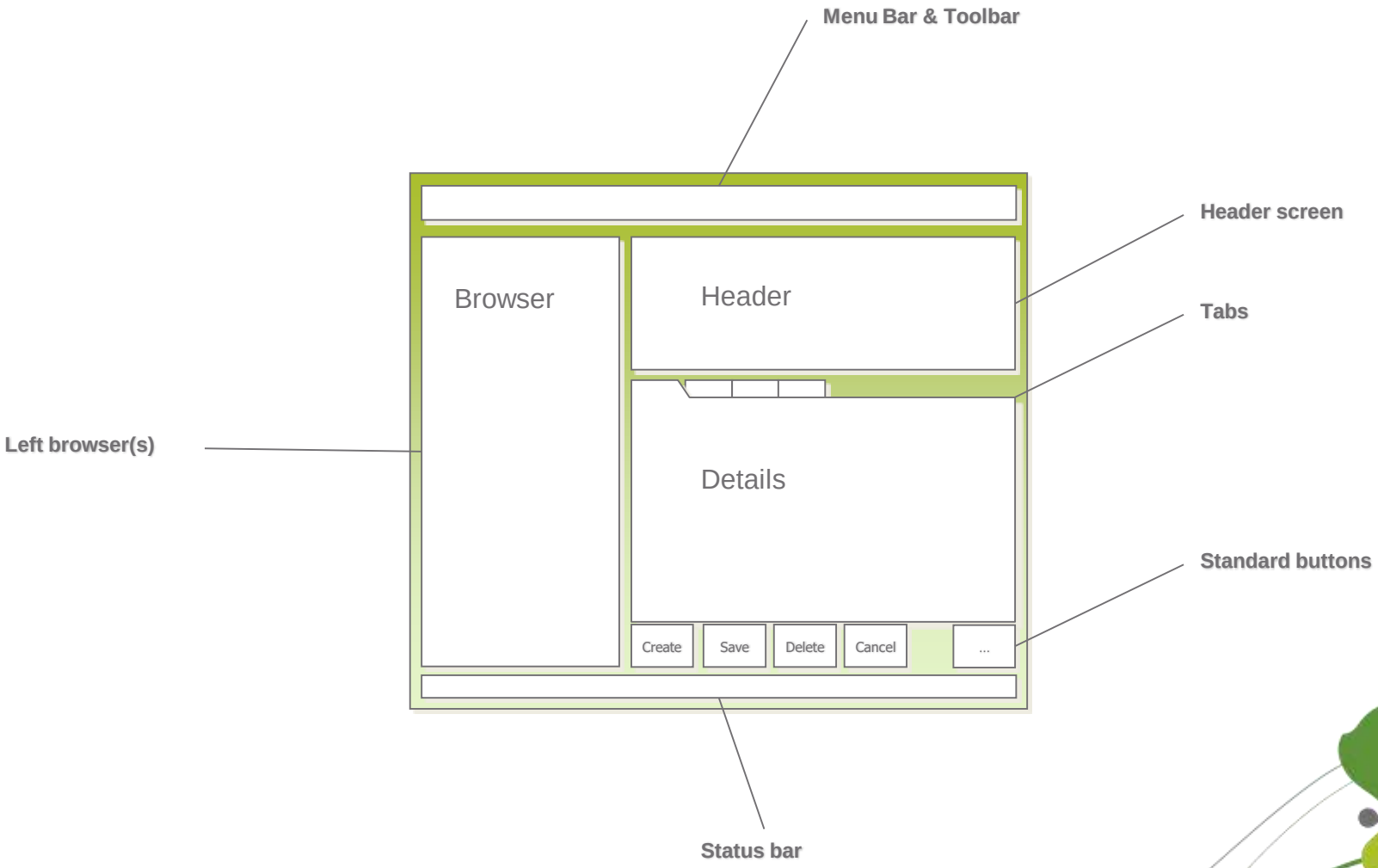
- Sage X3 is built into functions that may be called:
  - From a menu item
  - From an html page (e.g. Portal, Wizards…) – via a link
  - From other functions (Tunnels)
- Functions serve as a basis for:
  - Access rights (defined for User/Function combinations)
  - Menu organisation
- Functions are identified by a code (**`GFONCTION`** variable, tip text in Sage X3 menus).
- They are generally built based on an Sage X3 function template that can be customised.

Functions Belong to Modules



1.4 – The User Interface

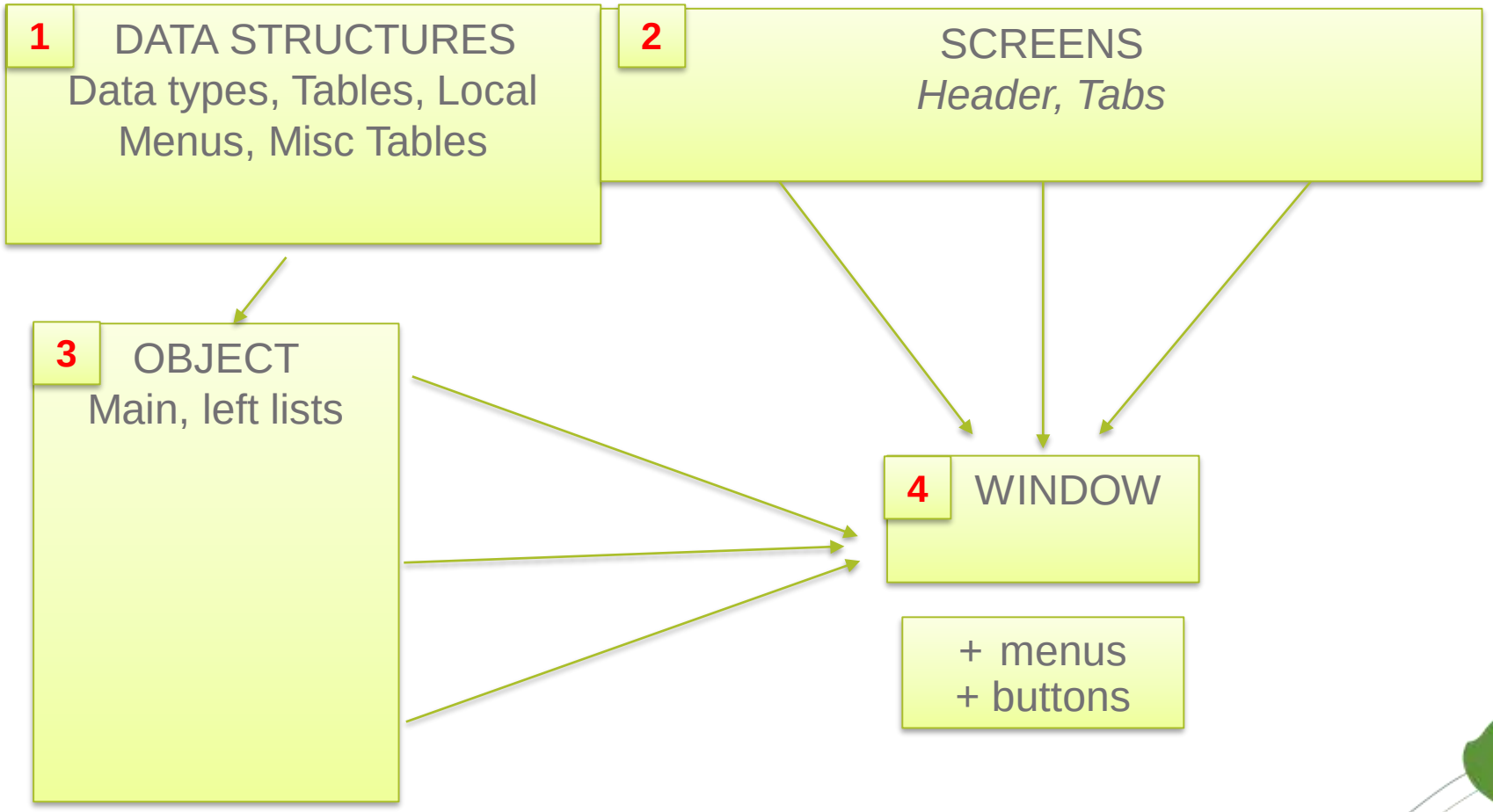
Common User Interface Elements



Contents

- 1. Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- 3. Introduction To The Sage X3 Language
  - 3.1 – Variables And Variable Classes
  - 3.2 – Operators
  - 3.3 – Instructions
  - 3.4 – Functions
  - 3.5 – System Variables
  - 3.6 – Functions, Subprograms And Scope
  - 3.7 – Reusable Subprograms
  - 3.8 – Performances
- 4. Basic Development: The Object Template
  - 4.1 – The Object Template
  - 4.2 – Field Actions
  - 4.3 – Object Actions
  - 4.4 – STD snippet: Database transaction management
- 5. Debugging Sage X3 Code
  - 5.1 – The Debugger

General Development process



DATA STRUCTURE

2.1.1 – Tables & Views

Tables

Sage X3 Tables

- Sage X3 tables are described in the Table Dictionary. They represent and describe physical database tables.
- Dictionary table validation (*Validation* button):
  - Creation or
  - of *.srf* and *.fde* table definition files in the FIL directory.
  - **valfil** utility creates the table in the database, from table definition files.
  - The **valfil** utility may be run manually and accepts a number of options.

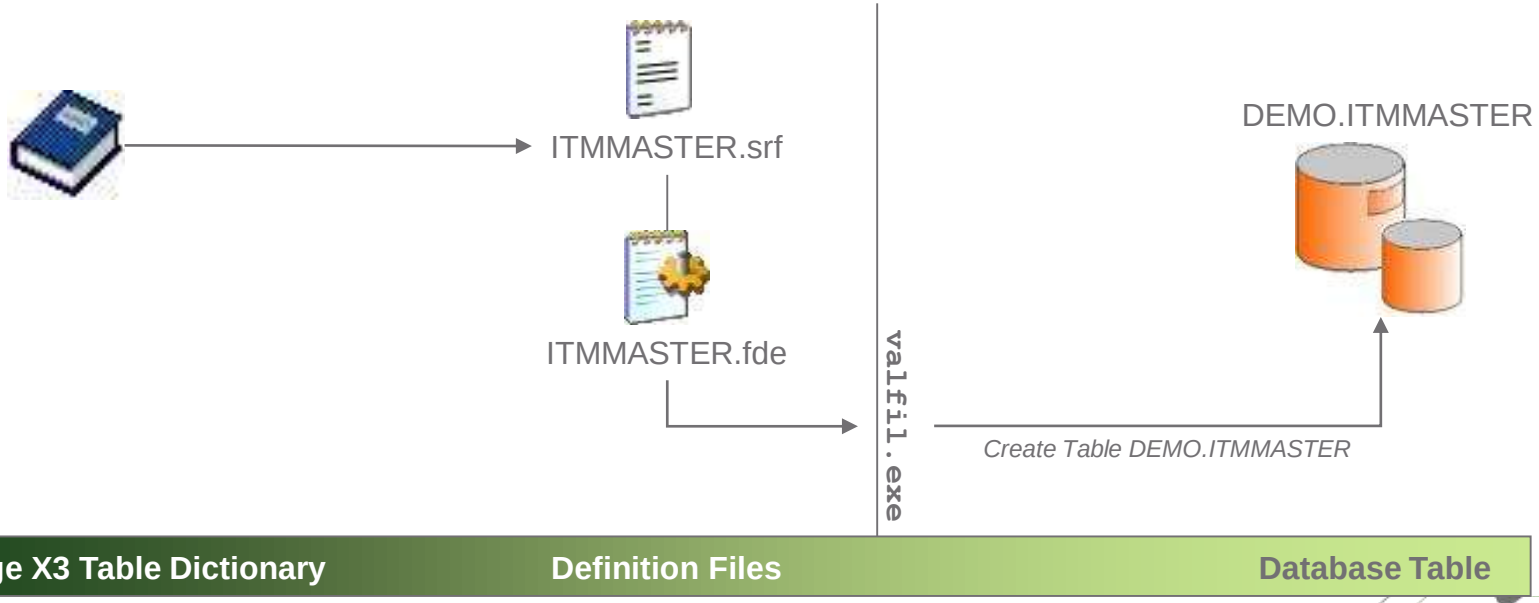


Table Fields

- Fields (Table columns) are identified mainly by the following elements:
  Code
  Data type and length
  Designation (Normal / Long / Short)
  Dimension
- Fields can be linked to a reference table (*Linked Table*)
  Link Expression contains the values to link for linked tables with complex indexes.
  Cancellation options specify data integrity checks to perform when linked record is deleted
- Strong or Weak Types may be used with table fields
  Strong Types are mainly linked to Sage X3 Objects (e.g. Products (ITM), Customers (BPC) etc.)
  Main Weak Types:
  - **A** (String)
  - **C** (Short integer)
  - **L** (Long integer)
  - **D** (Date)
  - **DCB** (Floating Point)
- Each occurrence of a dimensioned field is a column in the database table. The maximum number of columns (Field*dimension) is 512 .

Table Indexes

- Table Indexes may have to 16 components and their description may span 256 characters.
- The primary index is the first index defined against the table.
- Index component sort order:
  - Ascending order (+ sign)
  - Descending order (- sign)
  - The first component has no sign and is always ascending.
- *Duplicates* specifies unique or non-unique indexes.

Special Fields

- Special fields may be added to main object tables. They are automatically managed
by the system.

  CREDAT, **UPDDAT**: Creation and update date (*Created On*…, *Updated On*…)

  CRETIM, **UPDTIM**: Creation and update time

  CREUSR, **UPDUSR**: Creation and update user code (*Created By*…, *Updated By*…)

  EXPNUM: Export number for chronological exports (See *Imports / Exports* chapter)

  ENAFLG: *Active* flag. Specifies if the record is active or not. Used in selections, controls etc.

- All fields except ENAFLG are usually invisible (not included in object screens).

Special Fields

- Special fields may be added to main object tables. They are automatically managed
  by the system.

  CRESTP, **UPDSTP**: Creation and update timestamp

Table Buttons

- ***Validation***: Validates the table.
- ***Process***: Creates a temporary process that may be used to initialise fields when the table is validated.
- ***Copy***: Copies the table definition from one folder to another folder (no validation). Actual data is not copied.
- ***RTZ***: Deletes the table data (Empties the table).
- ***Delete***: Deletes the table from the database and the table definition from the Sage X3 dictionary.

DATA STRUCTURE

2.1.1 Tables & Views

Views

Views

- Target

- Create views in the database
  Accessible from Crystal Reports
  Accessible from Formula Wizard
  Accessible from processes (read only)

- Restrictions

-The view code should not be an existing table code

DATA STRUCTURE

2.1.2 Data Types

Data Types

- **Data Types** fall into 3 categories:

Basic data types

- Built into the software and not modifiable
- Examples: A (Alphanumeric), DCB (Decimal), D (Date), L (Long integer)…

User defined types

- Defined by specifying the length, format, entry options and optional actions (Events)
- Examples: MD1 (Currency Amount), BID (Bank ID Code)…

Object data types inherit the following items from their object

- Controls
- Automatic zooms and selection boxes
- Data integrity checks

Advanced Data Types

- **User Defined** data types

  Internal Type

  - Local Menu
  - Short/Long Integer
  - Double Precision / Floating Point Decimal
  - Clob/Blob
  - Alphanumeric
  - Date

  Sage X3 Format:

  - Sage X3 expression for complex formats
  - Variable formats may be specified by entering an Sage X3 expression preceded by the '**=**' sign

  Additional **Options** are available depending on the basic data type.

- **Object** data types

  The **Object Code** links the data type to an Sage X3 object

  - Automatic tunnels and selection options when defining fields with that data type
  - Automatic data integrity controls

DATA STRUCTURE

2.1.3 Local Menus

Local Menus

- **Local Menus** are lists of application texts that are used:
  - In drop-down lists, radio buttons and checkboxes
  - For application messages (Information boxes, errors, warnings etc.)

- **Organisation** of local menus:
  - Local menus are grouped in **chapters**
  - Each text in a chapter is identified by a sequential **number** starting with 1

- In the **database**:
  - Local menus are stored in table **APLSTD**
  - When a field contains a local menu item, the text number is stored in the database rather than the actual text
  - Local menus are hence externalised and translatable

Using Local Menus

- Using local menus

The **M** data type identifies a Local Menu, along with the chapter number.

- Fields with that data type will present a choice of text values
  As a drop down list
  As Radio buttons
  As a check box for the Yes/No menu (Chapter 1)

The **MM** data type identifies a Local Menu changeable by a user.

Local Menus used as application messages may be retrieved using the **`mess`** instruction.

- **`mess(1, 110, 1)`** retrieves text 1 from chapter 110 in the connexion language.

Specific local menus may be created in chapters 1000 to 1999. Specific menus used as messages may be created in chapters 160 to 169.

Using Local Menus

- Local Menus VS Messages

If a local menu is not explicitly flag as “local menu”, then it is a message.

- Still stored in APLSTD table
- Values can’t be changed by users
- Not uploaded on the user workstation (client server)
- Tend to be used for handling messages in source code (error message, etc.)
- Still translatable.

Using Local Menus

In a single chapter, texts are identified by numbers that span 1 to 123.
Chapters 1 to 999 are standard.

- Local menu ranges

  1000 - 1999 & 5200 – 5999 : verticals
  6200 - 6999 : customization

  4000 - 4999 : Add-ons

- Messages ranges

  160 - 164 & 6000 - 6199 : Customization
  165 - 169 & 5000 – 5199 : verticals

DATA STRUCTURE

2.1.4 Activity Codes

Activity Codes

- **Activity codes** are signature codes that are used to:

Protect and identify specific developments (Type = **Functional**):

- Specific activity codes start with **X**, **Y** or **Z**
- Protection from patches and upgrades
- Identification for patch extraction and dictionary validation

Activate or deactivate Sage X3 elements (Type = **Functional**):

- Tables, table indexes, table fields
- Screens, screen blocks, screen fields
- Objects, individual object parameters
- Window tabs
- …

Attach a customisable dimension to array elements (Type = **Dimension**)

- Table fields
- Screen fields (lines)

Identify localisations (Type = **Localisation**):

- Localisation activity codes start with letter **K**

Managing Activity Codes

- Managing activity codes in the ***Folders*** function:
  - Specific activity codes appear in the *Modifications* tab in the *Folders* function
  - Standard functional activity codes and localisation activity codes appear in the *Options* tab
  - Dimension activity codes are listed in the *Screens* tab

- After an activity code has been modified, the elements it links to must be **revalidated**:
  - Through full folder validation (*Folders* function)
  - Through the *Dictionary Validation* function (Development / Utilities / Dictionary / Validation) with a filter on the activity code.

- The *Activity Code Search* function (Development / Utilities / Searches / Activity Code) lists all the Sage X3 elements linked to a single activity code.

Using Activity Codes

- Protecting your specific developments using activity codes

  From patches (Conflicts with patches containing the same elements)

  From upgrades (Folder validation and/or changing application version)

  It is **mandatory** to sign and protect all specific developments using specific activity codes (starting with **X**, **Y** or **Z**).

  - Fields inserted in existing tables or screens
  - New blocks in standard screens
  - New tabs in standard windows
  - As a general rule, all new elements in the development dictionaries (All elements under
    the *Development* menu)

  It is **not necessary** to protect customisation elements that are considered as parameterisation.

  - Some of the object parameters such as selection parameters (*Object Personalisation* function)
  - New requesters, workflow rules, statistical parameters etc. or modifications to standard ones
  - As a general rule, all elements under the *Parameters* menu.

CREATING THE DATA STRUCTURE

- LAB1: Domain management

Exercice 1.1: Activity Code

Exercice 1.2: Local Menu

Exercice 1.3: Table

EXERCICES

DATA STRUCTURE

2.1.4 Miscellaneous Table

Miscellaneous Tables

Miscellaneous Tables

Miscellaneous tables are *logical* tables used to store a small sets of records without creating a physical table for them:

- Incoterms
- Payment Terms
- Tax Codes
- …

All miscellaneous table data are stored in one physical table, **ATABDIV**.
Miscellaneous table definitions are stored in table **ATABTAB**.

Miscellaneous table definition

Miscellaneous tables are identified by a **number**.

They store a code, short description and long description by default for each record.

Additional columns with various data types may be added to a miscellaneous table.

Miscellaneous Table Hierarchies

- Miscellaneous tables may be dependent, in which case their data is hierarchical.
  - Parent table is defined in child table record
  - Child table records reference the parent record.

Table 440 – Customer Stat Group 1

Code	Designation
TYP	Type
REV	Revenue
LOC	Location

Table 441 – Customer Stat Group 2

Code	Designation	Linked to Table 440 Record...
PER	Personal	TYP
COM	Company	TYP
GRP	Group	TYP
LRG	+ 10 000 K$	REV
MED	+ 1 000 K$	REV
SML	+ 100 K$	REV
EUR	Europe	LOC
AFR	Africa	LOC
MEA	Middle East	LOC
ASI	Asia	LOC

Using Miscellaneous Tables

- Using miscellaneous tables

A miscellaneous table may be attached to a field using data type **ADI**, with the following parameters:

- The table number as a data type parameter in (***Table Number*** parameter).

Miscellaneous table dependency is defined by specifying the parent table data as a
data type parameter (***Dependency Table*** parameter):

- Field data will be controlled according to the expression entered against that parameter.
- For more details on miscellaneous table dependency, please review the Sage X3 Supervisor training course.

Contents

- 1. Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- 3. Introduction To The Sage X3 Language
  - 3.1 – Variables And Variable Classes
  - 3.2 – Operators
  - 3.3 – Instructions
  - 3.4 – Functions
  - 3.5 – System Variables
  - 3.6 – Functions, Subprograms And Scope
  - 3.7 – Reusable Subprograms
  - 3.8 – Performances
- 4. Basic Development: The Object Template
  - 4.1 – The Object Template
  - 4.2 – Field Actions
  - 4.3 – Object Actions
  - 4.4 – STD snippet: Database transaction management
- 5. Debugging Sage X3 Code
  - 5.1 – The Debugger

GRAPHICAL STRUCTURE & INTERFACE

2.2.1 Screens

Sage X3 Screens

- Sage X3 screens are the building blocks for windows that are used to edit data. Screens may be window headers window tabs or standalone forms.

- Screens are described in the *Screen Dictionary*. When a screen is validated, its description file is created in a directory on the server.

- Screens are made of one or several **blocks**.

General Information

- A screen's ***Size*** may be initialised with default values by selecting a screen type. Sizes may be changed manually.

- *Template Screen*: Used to define non-usable screens that serve as templates or cache structures.

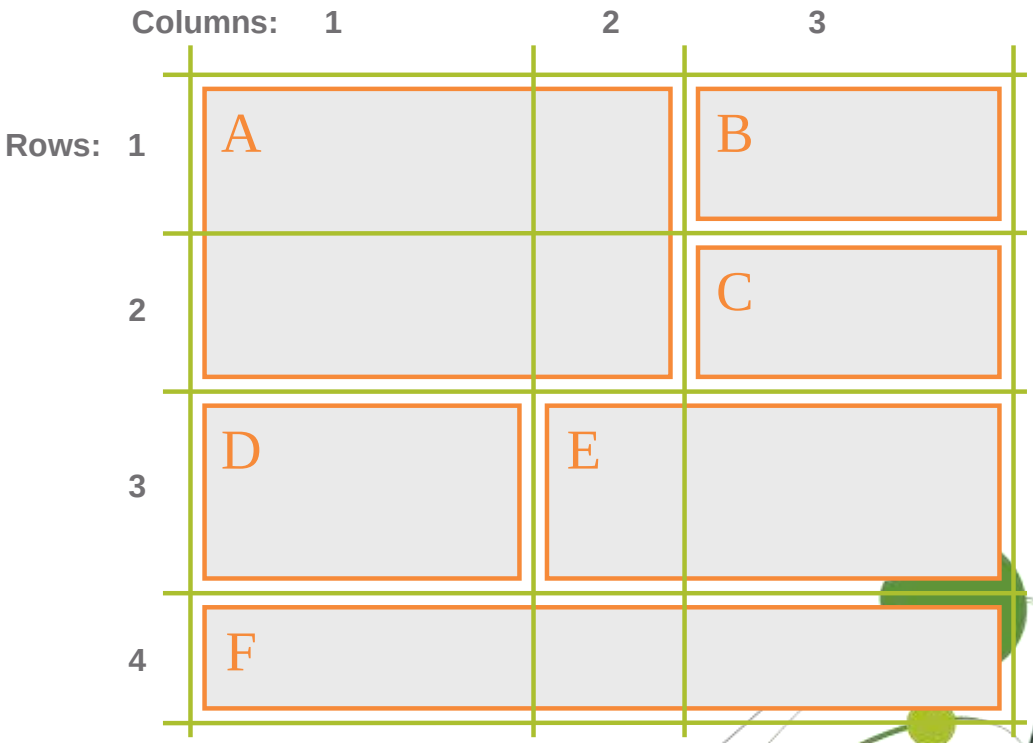
- Block types:
  List: Normal block with individual fields
  Table: Cell block that contains arrayed fields (dimension > 1)
  Text: Contains one text (or rich text) field
  Hidden: Invisible normal block

- Block definition:
  The block ***Row*** identifies the block in the screen. It may be used in Sage X3 code instructions to point to the block.
  Length is the total width reserved for field titles.
  Line identifies the number of lines for table blocks
  Bottom of Page contains the name of the variable that holds the total number of lines for table blocks, typically NBLIG.

Block Layout

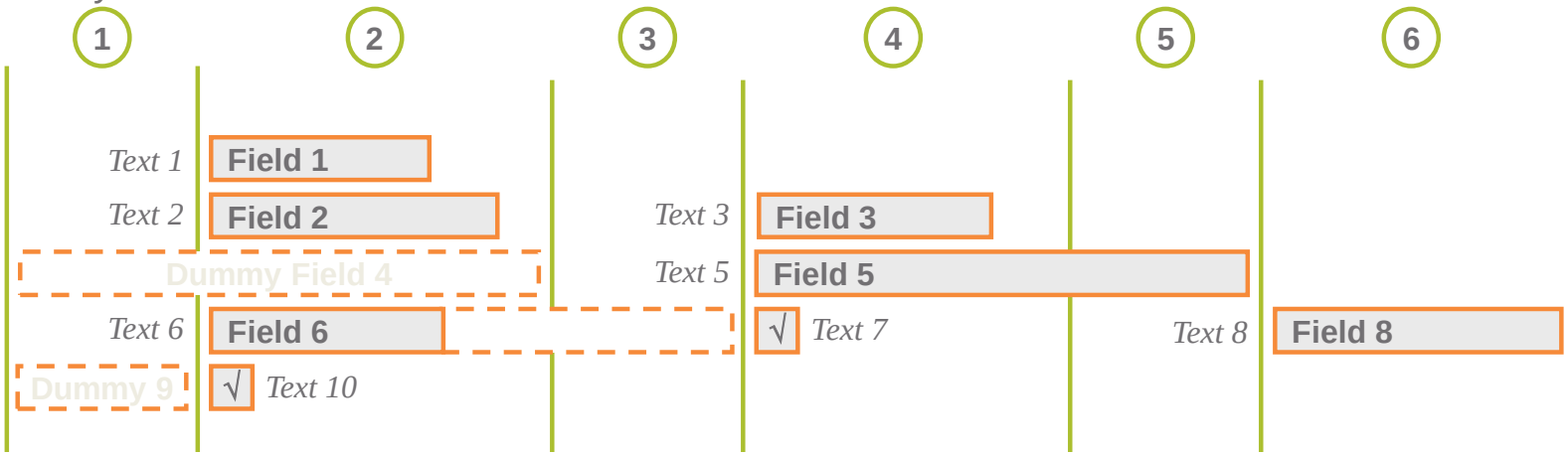
- Blocks are ordered in a screen using a grid that adapts to block boundaries.
  - Each block is placed by specifying its row/column on the grid.
  - The number of cells occupied by each block is specified in rows and columns.
  - The grid is flexible and blocks will extend or retract if the screen size is reduced.

	Block Type	Position	Line	Col.
A	List	1,1	2	2
B	List	1,3	1	1
C	Table	2,3	1	1
D	List	3,1	1	1
E	Text	3,2	1	2
F	List	4,1	1	3



Screen Fields

- Fields are placed in the block by specifying a row/column position. Exact arrangement of the fields is managed automatically.
- The width of a field may be specified in number of columns: This means the field ‘books’ a number of columns for itself, even if does not physically occupy them all.
- Field layout:



Clob And Blob Fields

- Clobs (**C**haracter **L**ong **Ob**jects) contain rich text data and are stored directly in the database.
- Data type: **ACB** or **AC0**.
- Graphical object: *Multi-line Text*.
- Parameters:
  Number Of Lines (height of text box)
  Physical Columns (width of text box)
  Text Type: Rich text (RTF), plain text (TXT), undetermined (Chosen during entry via context menu)
- Blobs (**B**inary **L**ong **O**bjects) contain binary data and are stored directly in the database.
- Data Type: **ABB** or **AB0**.
- Graphical object: *Photo*.
- Parameters:
  Number Of Lines (height of text box)
  Physical Columns (width of text box)
  Photo Type: Normal, Stretched, Proportional.
- A *Photo* object has default context menus:
  Select…
  Save As…

Icon Fields

- An *icon* is a character field with length = 3. It displays as a normalised square button anywhere in the block.
- The field value indicates the icon to be displayed.
- An icon may be clicked to trigger an event/action.

Development > Processes > Screens > Screens (GESAMK)

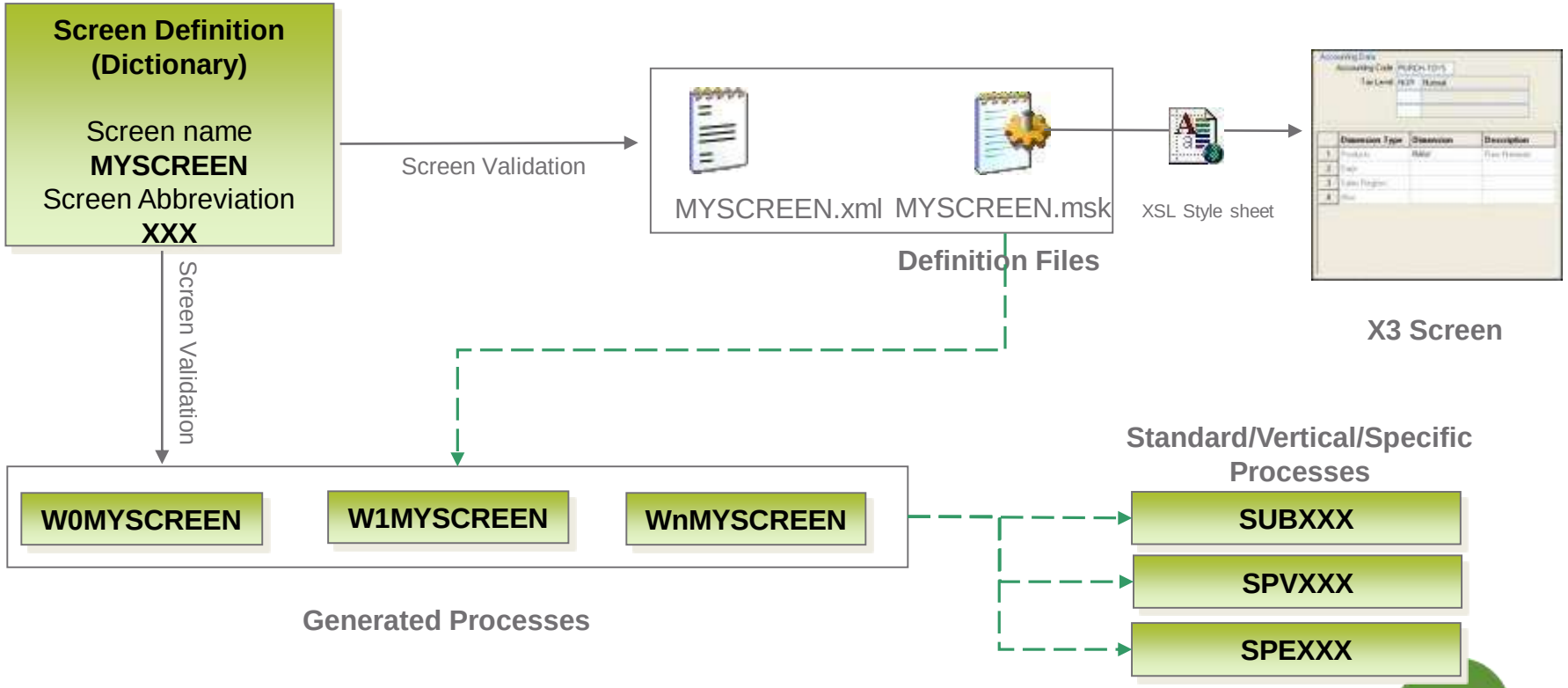
- Option > Icons

Table Blocks

- Table blocks contain arrayed fields (Dimension > 1) in a cell form (Columns).
- They are managed by a *Grid Control* variable, typically ***NBLIG***.
  Referenced in the block definition
  Data type: ***ABS*** (Absolute integer)
  Mandatory field placed first in the block
  At any time, that field contains the total number of lines in the block
  Field *Entry Type* defines the entry mode for the whole block: *Entered*, *Displayed*, *Hidden, Technical*
- Various options may be added to a table block. They define cell and line behaviour:
  A: Cancellation (Line deletion)
  I: Insertion (Line insertion)
  # (Any digit): Number of fixed columns…
- Fields in the block are referenced by their index: FIELD(0) for line 1, FIELD(1) for line 2, FIELD(2) for line 3 etc.

Screen Validation

> Screen validation creates the actual screen definition file (XML) and all generated *processes* that are linked to the screen.



CREATING THE GRAPHICAL STRUCTURE & INTERFACE

- LAB1: Domain management

  Exercice 2.1: Screens

EXERCICES

GRAPHICAL STRUCTURE & INTERFACE

2.2.2 Objects

Objects

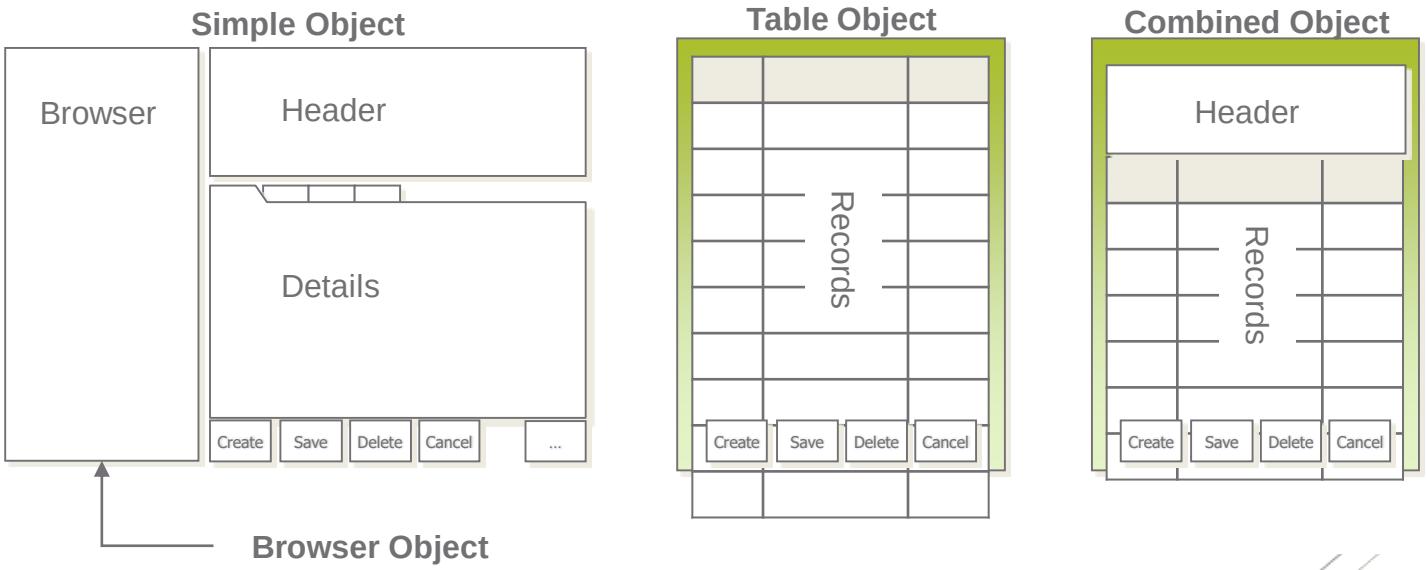
- Objects represent the core of Sage X3 X3 data maintenance. An *Object* manages one main table and defines the general behaviour of the data for that table.
- There are three types of object interfaces:

  Simple: Header/Tab management, record by record.

  Table: All records are viewed in one shot, in a table screen.

  Combined: First N parts of index is viewed in header screen; Last part is viewed in cell block (equivalent to viewing record groups for a *Table* object).

  Browser objects are used to define left lists (browsers).



Objects

- An object manages a single **main table**, or reference table. Other tables may be linked to it (See below) but must be managed by code.

- **Selection** parameters define

  A number of ***Selection Options*** that may be used as filters when linking data to the object.

  The columns that should be used to display selection boxes for records managed by that object.

Object Environment Tab

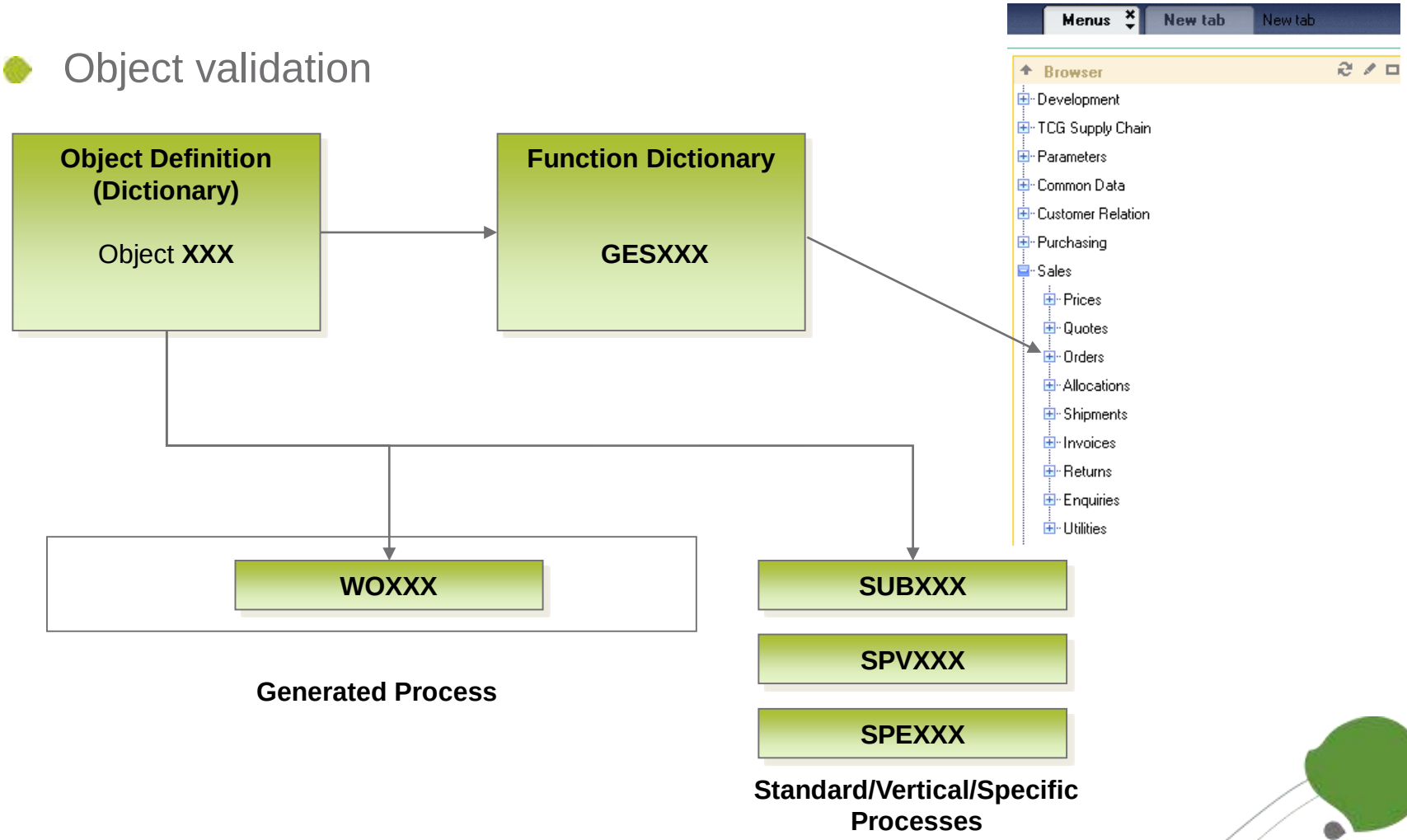
- The Environment tab includes additional tables that should be opened by the system

Automatic joins are possible through the Link Expression column.

Linked tables must be updated by code. The object does not manage them automatically.

Object Validation

- Object validation



CREATING THE GRAPHICAL STRUCTURE & INTERFACE

- LAB1: Domain management

  Exercice 2.2: Object

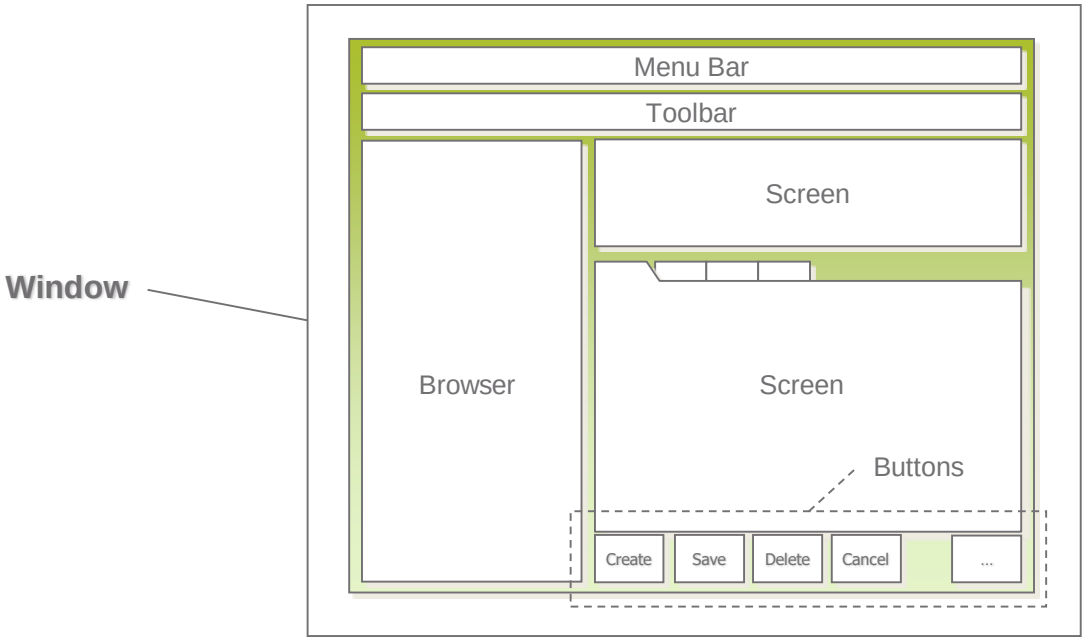
EXERCICES

GRAPHICAL STRUCTURE & INTERFACE

2.2.3 Windows

Windows

> **Windows** group a number of screens together with a browser, toolbar, menu bar and buttons to form a full-fledged Sage X3 function. Windows linked to objects are usually named OXXX where XXX is the object code.

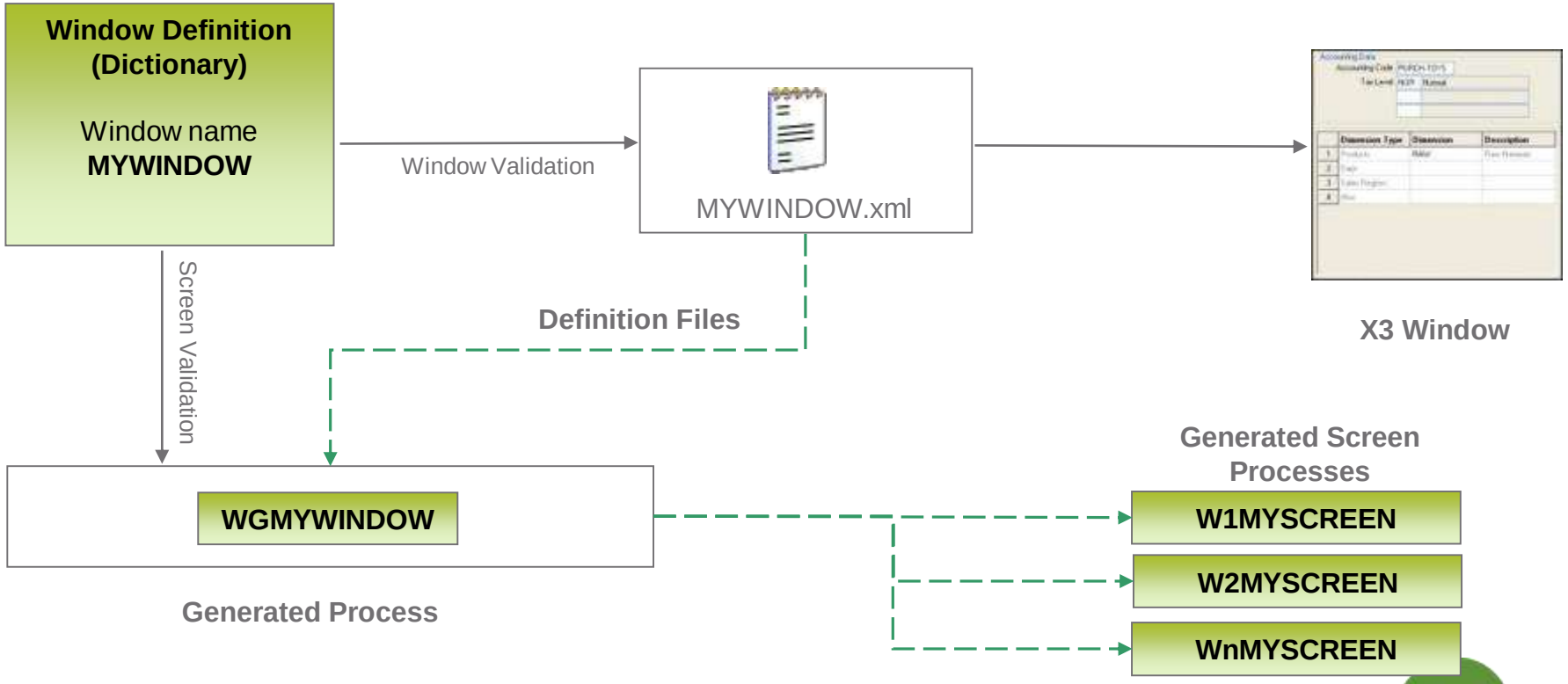


Window Definition

- A window contains one or more **Screens**:
  - A unique header screen (optional)
  - One or more tabs
- The **Buttons/Menus** tab contains the list of user-defined buttons and menu items
  - Menu items and buttons are identified by a code letter
  - They are attached to actions from the Action Dictionary or directly handled by code
- **Predefined** buttons may be enabled/disabled
  - Predefined buttons depend on the Window's use (Development Template)
- The **Browsers** tab includes a list of the window's left browsers
  - Identified by the browser object code
  - Options to view/hide the Last Read browser
  - Browser types:
    - Simple and hierarchical lists: Display information in flat or hierarchical arrangements
    - Picking lists: Used to pick a number of records from the browser (Additional actions and indicators)
    - Recursive lists: Data is accessed by drilling in from the top to lower levels

Window Validation

> Window validation creates the actual screen definition file (XML) and all generated *processes* that are linked to the window.



CREATING THE GRAPHICAL STRUCTURE & INTERFACE

- LAB1: Domain management

  Exercice 2.3: Window

EXERCICES

GRAPHICAL STRUCTURE & INTERFACE

2.2.4 Functions

Functions

Functions

Functions links Sage X3 home page menus to

- Sub-menus
- Sage X3 code, through the Action Dictionary and development templates.

Functions also encapsulate access right management, help definitions and other parameters.

Process type functions must be linked to an action from the Action Dictionary

- The Action defines which development template is to be run and which programs will be executed.
- This is the main type of functions, that links an Sage X3 menu to an actual functionality.

CREATING THE GRAPHICAL STRUCTURE & INTERFACE

- LAB1: Domain management

Exercice 2.4: Menu

Exercice 2.5: Function

EXERCICES

Contents

- Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- **3. Introduction To The Sage X3 Language**
  - **3.1 – Variables And Variable Classes**
  - **3.2 – Operators**
  - **3.3 – Instructions**
  - **3.4 – Functions**
  - **3.5 – System Variables**
  - **3.6 – Functions, Subprograms And Scope**
  - **3.7 – Reusable Subprograms**
  - **3.8 – Performances**
- 4. Basic Development: The Object Template
  - 4.1 – The Object Template
  - 4.2 – Field Actions
  - 4.3 – Object Actions
  - 4.4 – STD snippet: Database transaction management
- 5. Debugging Sage X3 Code
  - 5.1 – The Debugger

3.1 – Variables And Variable Classes

Variable Declaration

- Sage X3 variables represent memory values (classical variables), table fields or screen fields.
- Variable declaration syntax:

Global Local	**Integer (-2^31+1 to 2^31-1)** **Shortint(-2^15 to 2^15-1)** **Decimal (< -2^255 to 2^255)** **Char (30 by default,255 max)** **Date** **Clbfile (512Ko max)** **Blbfile (512Ko max)** **1024 Ko max**	*Variable_name (64 char max)* 	(Nb) or (Nb1..Nb2) (Nb) or (Nb1..Nb2) (Nb) or (Nb1..Nb2) (Nb) (Nb) (Nb)	Defines a variable of the corresponding type. Initialises the variable to a null (or zero) value Indexes start with 0 unless otherwise specified. Starting and ending indexes may be specified.
	File	*Table_Code (12 char max)*		Opens a table with the specified abbreviation.
	Mask	*Screen_Code (12 char max)*		Opens a screen with the specified abbreviation.

```
Local Date MYDATE                    Declares a Date variable [DD/MM/YYYY]
Local Integer XMYINT(10)             Declares an array of 10 integers (0 to 9)
Local Integer XMYINT(1..10)          Declares an array of 10 integers (1 to 10)
Global Char YMYGLOBAL(250)           Declares a string variable with length = 250
Local Mask ZSCREEN [ZSC]             Opens a screen with abbreviation ZSC
Local File XTABLE [XTB]              Opens a table with abbreviation XTB
Local File XTABLE                    Opens a table with default abbreviation
```

Variable Classes

- Variables are grouped into *classes* identified by the [*] prefix where * is the class identifier

Variable Class	Syntax	Examples	Returns
Table Field (value for the current record)	`[F:ABV]FLD` **ABV** = Table abbreviation **FLD** = Field name	`[F:BPC]BPCNUM`	Value contained in the table field
Screen Field (visible or not)	`[M:ABV]FLD` **ABV** = Screen abbreviation **FLD** = Field name	`[M:SIH0]BPCINV`	Value contained in the screen field

Global Variable	`[V]VAR`
	VAR = Variable name (class is not mandatory)
		`GVERSION`
		`GUSER`
		`GFONCTION`
		`GWINDOW`
			Variable value
			(Names usually start with '`G`')

System Variable	`[S]VAR`
	VAR = Variable name (class is not mandatory)
		`Datesyst`
		`nomap`
		`adxtcp`
			Variable value

Local Variable (depending on the context)	`[L]VAR` **VAR** = Variable name (class is not mandatory)	`XMYINT(8)` `[L]MYDATE`	Variable value

Table Class Declaration

- A table may be opened with additional options specifying
  - Record filters (*Where* clause)
  - Sort orders/indexes (*Order By* clause)

```
[Local] File <Table_Name> [<Abbreviation>] [, <Table Name> [Abbreviation]]...
&           Where <Where_Condition>
&           Order By [Key <Index_Code> | <Field_Name> [Asc | Desc] ]
```

- Table Close

```
Close [Local] File [<Abbreviation>][, <Abbreviation>][, <Abbreviation>]...
```

Screen Class Declaration

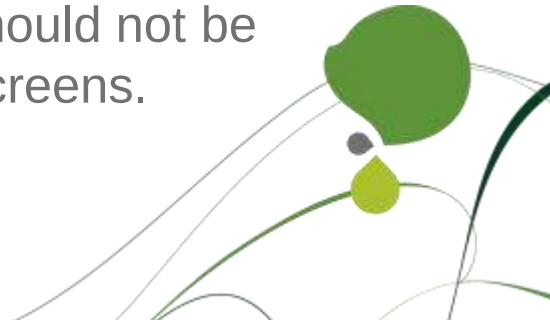
- Opening a screen:

```
[Local] Mask <Screen_Name> [<Abbreviation>] [, <Screen_Name> [<Abbreviation>]]...
```

- Closing a screen:

```
Close [Local] Mask [<Abbreviation>][,<Abbreviation>][,<Abbreviation>]...
```

- **Note**: Screens are usually automatically opened by the function template (See ad hoc chapters in this course) and should not be opened manually unless used as invisible working screens.



The `Clalev` Instruction

- The **`Clalev`** instruction shows if a class is accessible or not
  - Return value = 0: The class is not accessible
  - Return value positive: The class is accessible

```
# Is my table opened?
Local File BPCUSTOMER [BPC]
...
If Clalev([BPC]) > 0
    # We can close the table
    Close Local File [BPC]
Endif
```

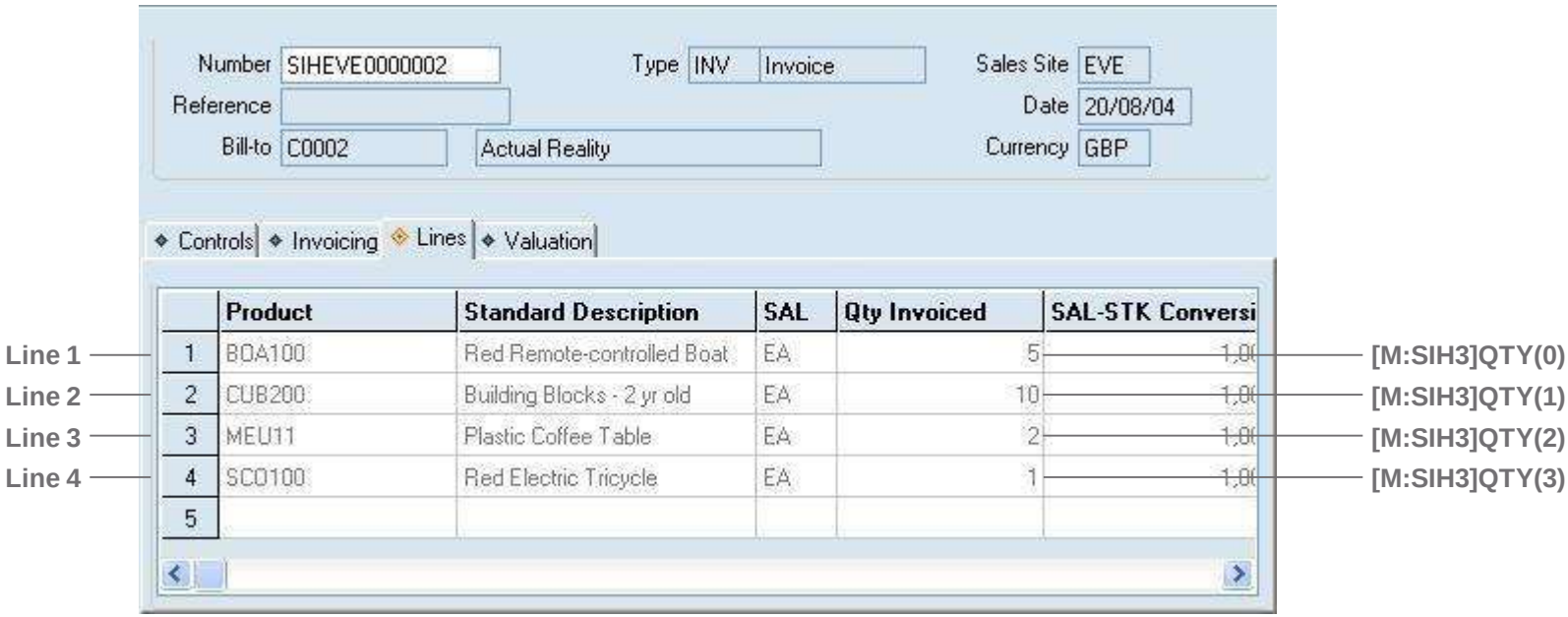
```
# Is my screen opened?
Local Mask WRKSCREEN [WRK]
...
If Clalev([WRK]) > 0
    # We can close the screen
    Close Local Mask [WRK]
Endif
```

Indexed Variables (Arrays)

```
# Declaration of local integer array with 8 elements (From 0 to 7)
# TABLNG(5) contains the value of the 6th element in the array
    Local Integer TABLNG(8)

# Declaration of local integer array with 8 elements (From 1 to 8)
# TABCOL(5) contains the value of the 5th element in the array
    Local Integer TABCOL(1..8)

# Declaration of local string with 5 characters
    Local Char STRING(5)

# Declaration of array of strings with 8 elements, each having 10 characters
# STR_ARRAY(5) contains the 6th string in the array
    Local Char STR_ARRAY(10)(8)

# Declaration of array of strings with 8 elements, each having 10 characters
# STR2_ARRAY(5) contains the 5th string in the array
    Local Char STR_ARRAY(10)(1..8)

# Declaration of a 2 dimensional string array
    Local char STR_ARRAY_2(10)(10,10)

# Declaration of 3-dimensional array of short integers
    Local Shortint INT_ARRAY(10,100,100)

# Arrays can be declared dynamically, by range of 128.
    Local Integer DYN_ARRAY(1..)
    DYN_ARRAY(20) = 5  # Segment 1..128 is allocated
    DYN_ARRAY(300) = 5 # Segment 256..384 is allocated. Segments 128..256 are empty
```

Indexed Screens (Grid Blocks)

- Screens with Grid Blocks contain arrays of fields
  - Field indexes start with 0
  - Line numbers start with 1



- The **`nolign`** variable indicates the current line number. Field indexes are referred to using expression **`nolign-1`**.

Assigning Values To Variables

```
# Assigning December 31, 2007 to a date variable
    Local Date ENDDAT
    ENDDAT = [31/12/2007]

# Assigning values to integers or decimals
    Local Integer DAYS
    Local Decimal HOURS
    DAYS = 7
    HOURS = DAYS*24

# Assigning values to a string
    Local Char MESSAGE(250)
    MESSAGE = "This operation is not allowed."
    MESSAGE += " Please check with the system administrator"

# Assigning a value to a table field
    Local File BPCUSTOMER [BPC]
    […]
    [F:BPC]BPCNAM = "Sage Group"
    [F:BPC]CREDAT = [01/01/2006]

# Assigning a value to a screen field
    [M:BPC1]BPCNAM = "Sage Group"
    [M:SIH3]DISCRGVAL(nolign-1) = 2.0
```

Clob And Blob Variables

- Clobs: Long text object, may be rich or plain text
- Blobs: Long binary objects (e.g. images)
- Sizes (Dimensions) are expressed in powers of 2

```
# Declaration of text Clob with default size
    Local Clbfile WCLOB1

# Declaration of text Clob with size = 32k
    Local Clbfile WCLOB2(5)

# Declaration of Blob with size = 64k
    Local Blbfile WBLOB(6)

# Array of 8k Clobs
    Local Clbfile WCLOBARRAY(3)(10)

# Assigning a Clob to another Clob
    [M:XXX]ClOB = [F:XXX]CLOB
    [M:XXX] = [F:XXX] :# Class transfer also works with binary objects

# Transferring a Clob into an array of strings
    Local Char TEXT(250)(50)
    Setlob TEXT With [M:XXX]CLOB

# Transferring an array of strings into a Clob
    Setlob [F:XXX]CLOB With TEXT(0..49)
```

3.2 – Operators

Common Operators

- Numerical:
  `+ - / * ^`
  Logical:
  `| (Or) - & (And) - ! (Not) - ?(Xor)`

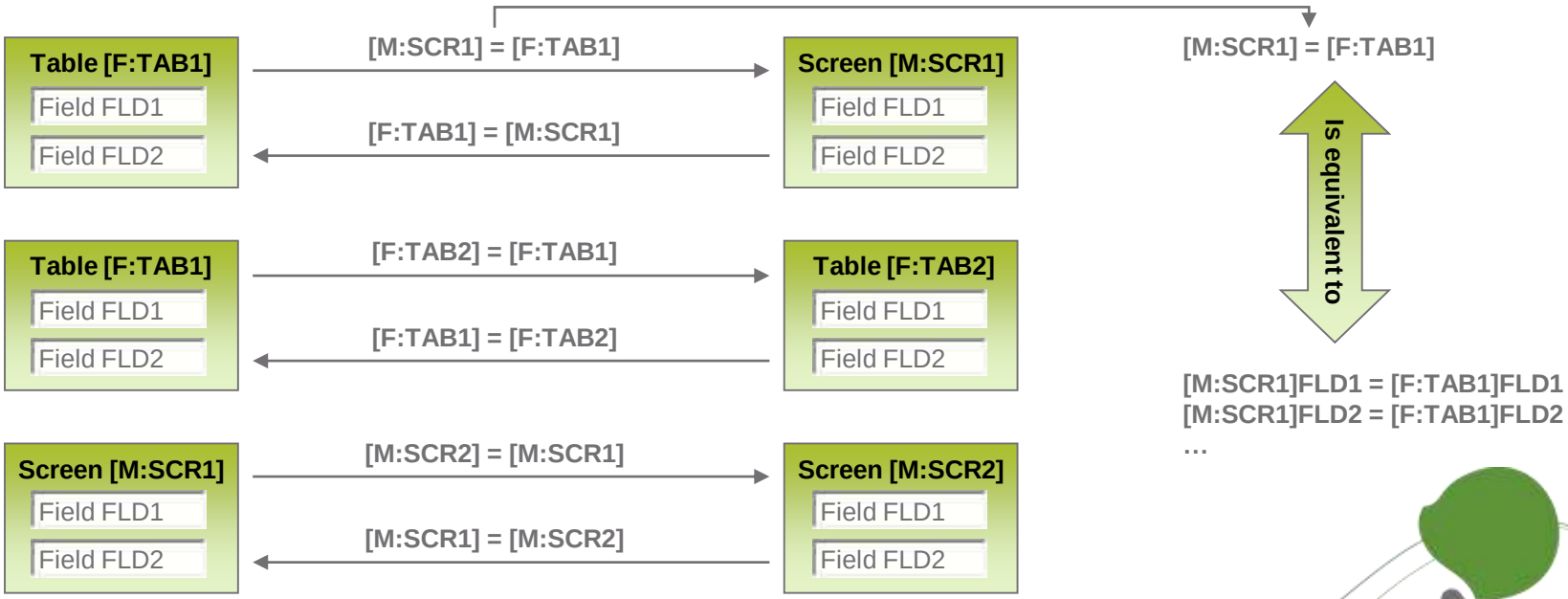
- Date:
  `Date1 + N` (Adds a number of days to a date)
  `Date1 – Date2` (Computes difference between two dates in number of days)
  `Date1 – N` (Subtracts a number of days from a date)

- String:
  `String1 + String2` (String concatenation without space)
  `String1 – String2` (String concatenation with space)

```
# Numerical operations
    Local Decimal I, J
    I = 100
    J = I/33  :# This populates J with 100/33 = 30.0303...

# Dates
    Local Date DATE1, DATE2
    DATE1 = [30/01/2007]
    DATE2 = DATE1 + 5  :# This sets DATE2 to 04/02/2007

# Strings
    Local Char STR1(150) : STR1 = "This customer cannot be assigned to site"
    Local Char STR2(250)
    STR2 = STR1 – "SITE01" :# Returns the following string in STR2:
                           # "This customer cannot be assigned to site SITE01"
```

Class Transfers

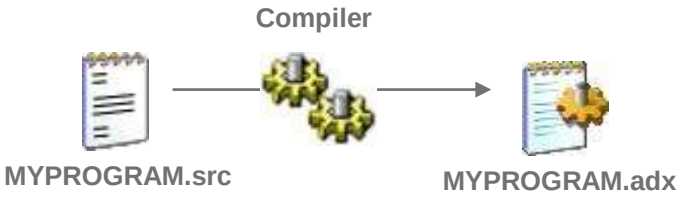
- Variable values in a class may be transferred in one shot:
  - From a screen to another screen
  - From a table to another table
  - From a screen to a table and vice versa
- Fields in the first class are populated with fields from the second class that have the same name.



3.3 – Instructions

Code Structure

- Sage X3 code is organised into **processes** (programs):
  Source code is stored as a text file (With .src extension)
  Compiled as binary p-code (corresponding file with .adx extension)



- Sage X3 processes are structured into three types of code "sections":
  Labels
  - Identified by the **$** symbol and a name
  - Terminated by the **Return** instruction
  - Called using the **Gosub** instruction

  Subprograms
  - Identified by the **Subprog** instruction, a name and optional parameters
  - Terminated by the **End** instruction
  - Called using the **Call** instruction

  Functions
  - Identified by the **Funprog** instruction, a name and optional parameters
  - Terminated by the **End** instruction and a return value
  - Called using the **Func** instruction

- Code statements are separated by line breaks or by colons (**:**)
  Statements may be continued on a subsequent line by starting that line with the **&** character.

- Any statement or line starting with # is identified as a comment.

Code Structure

- **Labels** organise the code into chunks *within the same scope*

EXAMPLE1.src

```
...
    Local File BPCUSTOMER [BPC]
    Read [BPC]BPC0 = "ADONIX"
    Local Decimal WAMOUNT

    Gosub READ_AMOUNT
    ...

End

##########################
$READ_AMOUNT
    # Get Customer Authorised Credit

    # Table [F:BPC] and
    # variable WAMOUNT are within scope
    WAMOUNT = [F:BPC]OSTAUZ

Return
```

EXAMPLE2.src

```
...
    Local File BPCUSTOMER [BPC]
    Read [BPC]BPC0 = "ADONIX"
    Local Decimal WAMOUNT

    WAMOUNT = [F:BPC]OSTAUZ
    ...

    End
```

Code Structure

- **Subprograms** isolate the code in a new scope

  Parameters may be passed to the subprogram by value or by address

  - By value: Declared using keyword **`Value`**. May not be modified.
  - By address: Declared using keyword **`Variable`**. May be modified.

  Subprograms do not return any value

EXAMPLE3.src

```
...
    Local Integer A, B, RESULT
    A = 2 : B = 14

    Call ADD(A, B, RESULT)

    EXAMPLE3::add(A,B,RESULT)  # New syntax available from V6.2

    # At this point variables V1 and V2 do not exist anymore
    Infbox "My Result is"-num$(RESULT)
End

############################
Subprog ADD(V1, V2, RES)
Value Integer V1
Value Integer V2
Variable Integer RES
    # At this point variables A and B are out of scope
    RES = V1+V2
End
```

EXAMPLE4.src

```
Local Integer A, B : A = 183 : B = 13
Local Integer RESULT
Call ADD(A, B, RESULT) From EXAMPLE3.src
```

- Subprograms may be called from another process using the From keyword

Code Structure

- Same declaration, definition and scope behaviour as subprograms
- **Functions** isolate the code in a new scope but have a return value

EXAMPLE3.src

```
...
    Local Integer A, B, RESULT
    A = 2 : B = 14

    RESULT = func ADD(A, B)

    # At this point variables V1 and V2 do not exist anymore
    Infbox "My result is"-num$(RESULT)

    # Or:
    Infbox "My result is"-num$( func ADD(A, B) )

End

############################
Funprog ADD(V1, V2)
Value Integer V1
Value Integer V2

# At this point variables A and B are out of scope

End V1+V2
```

EXAMPLE4.src

```
Local Integer A, B : A = 183 : B = 13
Local Integer RESULT
RESULT = func EXAMPLE3.ADD(A, B)
```

- Functions may be called in another program by adding the program name as a prefix

Code Structure

- Error handling and the **Onerrgo** statement

  Onerrgo routes program execution to a specified label when a low level error occurs (Such as a system error or exception)

  The error handling section should end with the **Resume** statement

```
# Open a sequential file on the server
# If the file is not there or there is a system error, try
# something else.

Onerrgo ERROR
Openi "E:\Sage X3\Temp\Tempfile.txt"
...
Onerrgo
...
End

$ERROR
    Infbox "Error n."-errn-errmes$(errn)-errm
    Infbox "Error occurred in program"-errp-"line"-errl
    Infbox "Please contact the system administrator."
    ... :# Try handling the error
Resume
```

Conditions

- **`If`** statement

```
If I=1
    Infbox "I is equal to 1"
Elsif I > 1 Then Infbox "I is greater than 1"
Elsif I > 0
    Infbox "I is between 0 and 1"
Else
    Infbox "I is negative"
Endif
```

- **`Case`** statement

```
Case I
    When 1 : Infbox "I is equal to 1"
    When 3
        Infbox "I is equal to 3"
        I += 1
    When 5,7,11 : Infbox "I is a prime number between 5 and 11"
    When WAMOUNT+[F:BPR]TAXES : Infbox "I is exactly the +Tax amount!"
    When Default
Endcase
```

Loops

- **For** statement

```
Local Integer I
Infbox "These are the even numbers between 1 and 100:"
For I = 1 to 100
    If mod(I,2) = 0 : Infbox num$(I) : Endif
Next I
```

```
Infbox "These are the even numbers between 1 and 10:"
For I = 2 to 10 Step 2
    Infbox num$(I) : Endif
Next
```

```
Infbox "These are the even numbers between 1 and 10:"
J = 10
For I = J-8, J-6, J-4, J-2, J
    Infbox num$(I) : Endif
Next
```

Increments in ranges, using a custom step value or through a value list

`Next` statement with variable specified or not

Loops

- **`While`** statement

  Loops while an expression is true

```
While <Expression>
    <Statements>
Wend
```

- **`Repeat`** statement

  Repeats a loop until an expression becomes true

```
Repeat
    <Statements>
Until <Expression>
```

Loops

- The **Break** statement

  Breaks a loop or several loop levels

  Used with **For**, **While** and **Repeat** instructions

```
# Search for value 25 in 2-dimensional matrix
Local Integer MATRIX(10,100)
For I = 0 To dim(MATRIX, 1) - 1
    For J = 0 To dim(MATRIX, 2) - 1
        If MATRIX(I,J) = 25
            Break 2 :# Value found – break out of 2 loops
        Endif
    Next J
Next I
```

```
# Check values in a screen with table block
For I = 0 to 500
    If [M:SIH3]QTY(I) > 100
        # Break out of loop
        Break
    Endif
Next
```

- LAB2: Playing with the X3 language

  Exercice 1: Playing with integers
  Exercice 2: Playing with Strings
  Exercice 3: Playing with Arrays

EXERCICES

Table Instructions

- Opening a table

```
[Local] File     <Table_Name> [ <Abbreviation> ]
[&      Where    <Where_Condition>  ]
[&      Order By         <Sort_Order>  ]
```

- Filtering a table

```
Filter <Abbreviation> Where <Where_Condition> [ Order By <Sort_Order> ]
```

- Cancelling the filter

```
Filter <Abbreviation>
```

- Closing a table

```
Close [Local] File [<Abbreviation>][,<Abbreviation>]...
```

```
# Open Products table with restriction on items with Statistical Family = "TOY"
Local File ITMMASTER [ITM1] Where TSICOD = "TOY"

# Open Products table and restrict to items with base price > 100
Local File ITMMASTER [ITM2]
Filter [ITM2] Where BASPRI > 100

# Close classes
Close Local File [ITM1], [ITM2]
```

Table Instructions

- The **Where** clause

Syntax 1:

```
... Where <Expression>
```

Syntax 2:

```
... Where Key[(<Index>)] = <Value1>[;<Value2>][;<Value3>]…
```

```
# Filter records in table ATABDIV
# Index named CODE is composed of fields CODE and NUMTAB

    If !clalev([ADI]) : Local File ATABDIV [ADI] : Endif

# First syntax
    Filter [ADI] Where NUMTAB = 9
    For [ADI]CODE
        Infbox ( num$( [F:ADI]NUMTAB ) - [F:ADI]CODE )
    Next
```

```
# Second syntax with similar result
    Filter [ADI] Where Key CODE(1) = 9
    For [ADI]CODE
        Infbox ( num$( [F:ADI]NUMTAB ) - [F:ADI]CODE )
    Next
```

Table Instructions

- The **`Order By`** clause

Syntax 1:

```
... Order By Key <Index_Id> [ (<Index_Part>) ] [ Asc | Desc ]
```

Syntax 2:

```
... Order By Key <Index_Id> = <Index_Field>[;<Index_Field>][;<Index_Field]...
```

Syntax 3:

```
... Order By <Field>[;<Field>][;<Field>]...
```

Syntax 4:

```
... Order With Key <Index_Id> = <Expression>
```

Syntax 5:

```
... Order With <Expression>
```

```
# Filter table SPD using index SPD0 (Already exists)
Filter [SPD] Where VCRTYP = 4 & VCRNUM = [F:SPH]VCRNUM & PACSEQ = [F:SPH]PACSEQ
&            Order By Key SPD0


# Filter table SQH1 using a new index, SQHX
Filter [SQH1] Where BPCORD = [F:SQH]BPCORD
&             Order By Key SQHX = BPCORD ; QUODAT


# Filter table SDH using an index constructed from an expression
Local Char INDEX(250), FLD1, FLD2, FLD3
FLD1 = "BPCORD" : FLD2 = "SDHNUM" : FLD3 = "ORDREF"
INDEX = FLD1 + ";" + FLD2 + ";" + "FLD3"

Filter [SDH] Where CREDAT >= date$-30
&            Order With Key SDHX = INDEX
```

Table Instructions

- The **Link** instruction (Table joins)

  Links join two or more tables based under a unique abbreviation, so that they may be read in one go with a unique **Read** or **For** statement.

```
     Link <Class> With <Class><Index> [ (<Index_Part>) ] =
<Expression> [ , ... ]
As <Abbreviation>
[ &  Where <Condition> ]
[ &  Order By <Sort_Order> ]
```

```
# Link table ITF to two other tables,
# read info and populate SBI2 screen

Link [ITF] With [ITV]ITV0=[F:ITF]ITMREF;[F:ITF]STOFCY,
&               [ITM]ITM0=[F:ITF]ITMREF
&          As [LNK]
&          Where [F:ITF]CUNFLG = 2 & [F:ITF]STOFCY = [M:SBI1]FCY
&          Order By Key ITF1

For [LNK]
    [M:SBI2] = [F:ITM]
    [M:SBI2] = [F:ITV]
    [M:SBI2] = [F:ITF]
Next
```

Database Access

The **For** instruction

The For statement scans records in a table using a specified index

When no index is specified, the default index is used:

- Last index used with the table in previous code
- First index defined against the table (Primary index) if none

```
For <Class>  [ <Index_Id> [ (<Index_Part>) ] ] [ From <Start_Value> To <End_Value> ]
[ & Where <Condition> ]
[ & With Lock | With Stability ]
...
Next   [ <Index_Id> ]
```

```
# Process sales order table
Local File SORDER [SOH]

For [SOH] Where ORDSTA = 1 & ORDDAT >= [01/01/2006]
    # Default index used to browse the table: SOH0 (Primary index for SORDER)
    ...
Next
```

```
# Process sales order price details
Local File SORDERP [SOP]

For [SOP]SOP0(1)
    # First loop on first part of index
    For [SOP]SOP0
        # Loop on details
    Next
Next
```

Database Access

- The **Read** and **Look** instructions

  Reads records using specified index and corresponding values
  The **Read** instruction loads the record into the [F] class
  The **Look** instruction reads the record (Tests its existence) but does not load the [F] class with data
  Return value **fstat** shows read status (OK / Not found etc.)

```
      Read [<Class>][<Index_ID>][ (<Index_Part>) ] [First | Curr | Next | Prev | Last]
Look                                              [= <Expression>[;<Expression>]...]
                                                  [> | < | >= | <= <Expression>>[;<Expression>]...]
```

```
# Product-Site table
Local File ITMFACILIT [ITF]

# Read first record
Read [ITF] First   :# Default index, ITF0 = ITMREF+STOFCY is used

# Read precise record
Read [ITF]ITF0 = "1INCHSCREW";"S01"

# Read next one
Read [ITF] Next :# Or Read [ITF]ITF0 Next
```

Database Access

- The **Read** and **Look** instructions with the >= / <= operators, with Multi-part indexes

  The following option used with a two-part index, for example:

  ```
  Read [XXX]KEY >= VALUE1;VALUE2
  ```

  Will read the following records:

  - Records where Part 1 = VALUE1 and Part 2 >= VALUE2
  - Records where Part 1 > VALUE1, independently from Part 2

```
# Product-Site table
Local File ITMFACILIT [ITF]

# Read first record with index >= "B";"S01"
Read [ITF]ITF0 >= "B";"S01"

# This will find, for example, a record with
#     Product code = "BATCH001"
#        Site code = "S01"
```

Database Access

- The **Columns** instruction

  Allows a table columns (fields) to be restricted to those specified

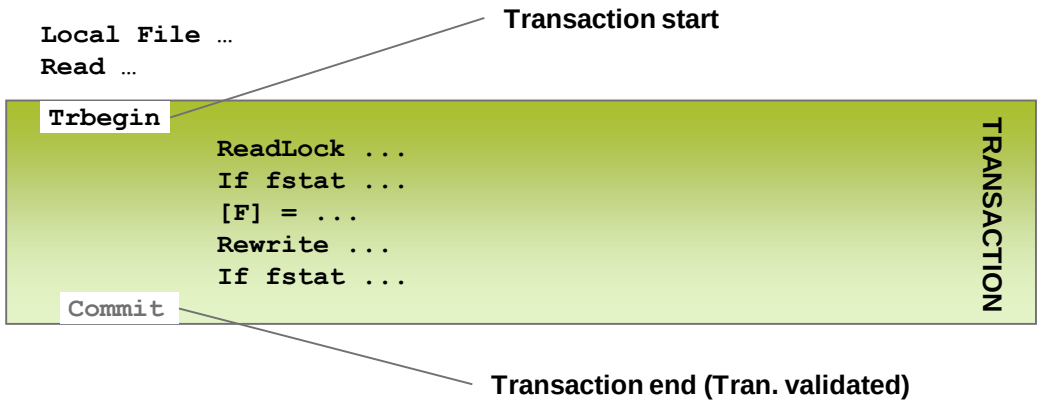
  Used to constrain the amount of data transferred from the database to the application

```
# Process products in WIP table

  Local File ORDERS [ORD]
  Local File ITMMASTER [ITM]

  Link [ORD] with [ITM]ITM0=[F:ORD]ITMREF as [ORI]

# Only need item reference and item designation in ITMMASTER table
# Need all the ORDERS table
  Columns [ORI] ([ITM]ITMREF,[ITM]ITMDES1,[ORD])
  For [ORI]
      # Process records
      ...
  Next

# Remove columns filter
  Columns [ORI]
```

```
# Populate a details screen [M:SOX] with sales orders data
Local File SORDERQ [SOQ]

# Only need pertinent data to populate the screen
Columns [SOQ] With Mask ([M:SOX])

For [SOQ]
    ...
```

Database Update

- A database table may be updated using the following instructions:
  `Write / Rewrite`
  `Update`
- An update process must be encapsulated inside a ***transaction*** to guarantee data integrity in case the transaction is aborted
  `Trbegin`
  `Commit`
  `Rollback`
- To avoid conflicts, existing records should be ***locked*** before they are modified
  `Lock`
  `… With Lock`
  `Readlock`



Database Update

- Transaction instructions

```
# Open Customers table and start
transaction
Trbegin BPCUSTOMER [BPC]

# Start transaction on Sales Orders
table
Local File SORDER [SOH]
Trbegin [SOH]

# Start transaction on table name in
variable
MYTABLE = "BPCUSTOMER"
Trbegin =MYTABLE [XXX]
```

`Trbegin`

- Signals the start of a transaction
- May open tables in Read/Write mode
- A transaction spans multiple tables

`Commit`

- Instructs the system to validate a transaction and finalise all modified data within the transaction (since the `Trbegin` statement)
- Releases all locks initiated within the transaction

`Rollback`

- Aborts a transaction
- All fields modified within the transaction are restored to their initial state
- Releases all locks initiated within the transaction
- When a process aborts before the `Commit` statement has been executed, the transaction
  is implicitly rolled back.

Database Update

- **Locking** a record: The **`Lock`** / **`Unlock`** and **`Readlock`** instructions

The **`Readlock`** instruction locks **a single record** in a table class

- Syntax is the same as the **`Read`** statement
- The lock is released by a **`Commit`**, **`Rollback`** or **`Unlock`** statement

The **`Lock`** instruction locks **all records** in one or several classes

```
Lock <Class>[,<Class>][,<Class>]... [With Lockwait =
  <Wait_Time>]
```

The **`Unlock`** instruction releases all locks on one or several tables

```
Unlock <Class>[,<Class>][,<Class>]...
```

Database Update

- The **`Write`** statement

Creates a **new record** in the table from the data contained in the [F] class

```
# Write new customer record
Local File BPCUSTOMER [BPC]

# Populate [F] class from [M:BPX] screen
[F:BPC] = [M:BPX]
[F:BPC]BPCNUM = "MYNEWCUSTOMER"

Trbegin [BPC]  :# Transaction start

# Write table
Write [BPC]
If fstat
   # Record already exists
   ...
   Rollback
Else
   Commit
Endif
```

Database Update

- The **Rewrite** statement

Modifies an **existing record** in the table from the data contained in the [F] class

```
# Modify existing customer record
Local File BPCUSTOMER [BPC]

Trbegin [BPC]  :# Transaction start

# Read record and lock.
Readlock [BPC]BPC0 = "SAGE"
If fstat > 1
    # Record not found
    Rollback
    ...
Elsif fstat = 1
    # Record already locked by another user
    Rollback
    ...
Else
    # Modify authorised credit
    [BPC]OSTAUZ = 15000
    # Rewrite table
    Rewrite [BPC]
    Commit
Endif
```

Database Update

- The **Update** statement

Modifies a group of existing records with optional filters
This statement automatically reads, locks, modifies and unlocks the records

Update [*<Class>*] **[Where** *<Condition>*] **With** *<Field>* **=** *<Value>*
[, *<Field>* **=** *<Value>***]...**

```
# Increase all customers' authorised credits by 1000
# For English customers
Local File BPCUSTOMER [BPC]

Trbegin [BPC]

Update [BPC] Where BPCCRY = "GB" With OSTAUZ = OSTAUZ + 1000
Infbox "Records modified:"-num$(adxuprec)

Commit
```

Database Update

- The **Delete** statement

Deletes a record or a group of records from the table

- Syntax 1 (Delete one record)

```
Delete [<Class>][<Index_ID>][ (<Index_Part>) ] [First | Curr | Next | Prev]
                                               [= <Expression>[;<Expression>]...]
                                               [> | < | >= | >= <Expression>]
```

- Syntax 2 (Delete a group of records)

```
Delete [<Class>][<Index_ID>][ (<Index_Part>) ] [Where <Condition>]
```

```
Trbegin BPPCUSTOMER [BPC]

# Delete all non-active customers
Delete [BPC] Where BPCSTA = 1
If fstat = 1
    # Locking problem
    Rollback
    ...
Else
    Infbox "Records deleted:"-num$(adxdlrec)
Endif

# Delete one customer
Delete [BPC]BPC0 = "CUST01"

# Delete current customer
Readlock [BPC]BPC0 = "CUST02"
Delete [BPC]

Commit
```

Working With Screens

- Refresh and erase instructions

Affzo	Refreshes a screen field (displays the new value if any)
Effzo	Erases a screen field (resets the field and un-initialises it)

```
# Working with customer screen BPC3

# Modify value and refresh field
[M:BPC3]TAXRAT = 19.6
Affzo [M:BPC3]TAXRAT

# Refresh several fields
Affzo [M:BPC3]TAXRULE, MODCPT, NBCOPIES

# Erase all fields from TAXRATE to TAXVMP
Effzo [M:BPC3]TAXRAT – TAXVMP

# Erase screen block with rank = 10
Effzo [M:BPC3]10

# Refresh whole screen
Affzo [M:BPC3]
```

Working With Screens

- The **Raz** instruction

Raz General RTZ (**R**eset **T**o **Z**ero) instruction used with all classes (Screens, Tables, Variables etc.)

```
# Reset whole BPC3 screen
Raz [M:BPC3]

# Reset some fields in BPC1 and refresh
Raz [M:BPC4]BPCGRU, BPCBPSNUM
Affzo [M:BPC4]BPCGRU, BPCBPSNUM

# Reset variable MYINT
Local Integer MYINT
Raz MYINT

# Reset last item in array of decimals
Local Decimal MYARRAY(10)
Raz MYARRAY(9)

# Reset all local variables
Raz [L]
```

Working With Screens

- Field activation and de-activation

Grizo	Completely disables a screen field or group of fields
Diszo	Locks screen fields (locks field data) but maintains other possibilities
Actzo	Activates a screen field or group of fields

```
# Activate fields in BPC1
Actzo [M:BPC1]BPCGRU, BPCBPSNUM

# Disable blocks with ranks 10 to 25 in BPC2 screen
# Field data appear with grey font over grey background
Grizo [M:BPC2]10 - 25

# Lock fields in BPC1
# Field data appear with black font over grey background
# Right-click and function key possibilities remain active
Diszo [M:BPC1]BPCGRU, BPCBPSNUM
```

Working With Sequential Files

- Sequential files may be opened, read from and written to on any machine that is visible from the Sage X3 server on the network.

- Opening a file:

Output mode:

```
Openo <Filename>, <Option> Using [<Abbr>]
```

Input mode:

```
Openi <Filename>, <Option> Using [<Abbr>]
```

Both modes:

```
Openio <Filename>, <Option> Using [<Abbr>]
```

Reading text data:

```
Rdseq <Variable>[;[<Variable>]...] Using [<Abbr>]
```

Writing text data:

```
Wrseq <Variable>[;[<Variable>]...] Using [<Abbr>]
```

Reading binary data:

```
Getseq <Nb_Elements>,<Variable>[,<Variable>]... Using [<Abbr>]
```

Writing binary data:

```
Putseq <Nb_Elements>,<Variable>[,<Variable>]... Using [<Abbr>]
```

- Format system variables:

adxirs Record separator (end of line)

adxifs Field separator represented by the **;** character with Rdseq and Wrseq instructions

adxium Encoding format (UTF-8, ISO, UCS)

Working With Sequential Files

- Getting path information: The **`filpath()`** function

  `filpath` returns the full path of a file according to the specified parameters:

  `filpath(`*`<Dir>, <Filename>, <Ext>`***`[, `***`<Application>`***`[, `***`<Vol>`***`[, `***`<Machine>`***`] ] ] )`**

```
# Get file path on server in tmp application directory for current application
Local Char MYFILE(250)
MYFILE = filpath("tmp", "myfile", "txt") :# Searching for myfile.txt in tmp directory
Infbox MYFILE

# Get path of Sage X3 client directory on client workstation
Infbox filpath("!", "", "", "", "", "#") :# Hash symbol represents client
```

- Getting file information: The **`filinfo()`** function

  `filinfo` returns one of the properties of the file that is given as an argument to the function. The second argument is the property that `filinfo` is to return:

  `filinfo(`*`<Filename>, <Property_Num>`***`)`**

```
# Get file size
Local Integer SIZE : SIZE = filinfo(filpath("FIL", "BPCUSTOMER", "fde"), 1)
If SIZE < 0 :# File does not exist
    ...
```

3.4 – Functions

String Functions

- **String** manipulation functions

`left$ ( CStr, iNb )`	Extracts the first iNb characters from a string CStr
`right$ ( CStr, Pos )`	Extracts a string's characters from character Pos
`mid$ ( CStr, Pos, iNb )`	Extracts iNb characters starting from Pos
`seg$ ( CStr, Pos1, Pos2 )`	Extracts the substring between characters Pos1 and Pos2
`len ( CStr )`	Returns the length of a string
`val ( CStr )`	Converts a string representing a number into a numerical value
`num$ ( iNum )`	Converts a number into a string
`tolower ( CStr )`	Converts the string to lowercase
`toupper ( CStr )`	Converts the string to uppercase
`ctrans ( CStr [, CIn, Cout ] )`	Filters and/or replaces characters in a string
`vireblc ( CStr, iOption )`	Deletes white space in a string
`format$ ( CFmt, CStr )`	Formats CFmt using a format string CFmt
`pat ( CStr, CPattern )`	Matches a string to a pattern
`instr ( Pos, CStr, CSch )`	Searches for a substring inside a string
`space$ ( iNb )`	Generates a string made of spaces
`string$ ( iNb, CStr )`	Generates a string made of iNb occurrences of CStr
`ascii ( CStr )`	ASCII code of first character in string CStr
`chr$ ( iAscii )`	Returns the character corresponding to ascii code iAscii
`parse ( CStr [, btoken] )`	Parses the string (Returns nonzero value if error)
`evalue ( CStr [, btoken] )`	Evaluates the expression in the string and returns the result
`getenv$ ( CStr )`	Returns the value of the system variable in string CStr

Date Functions

- **Date** manipulation functions

date$	Returns the current server date
time$	Returns the current server time in the HH:MM:SS format
time	Returns the current server time in seconds since 00:00:00
day (Date)	Extracts the day number
day$ (Date)	Extracts the day name
month (Date)	Extracts the month number
month$ (Date)	Extracts the month name
year (Date)	Extracts the year
gdat$ (iDay, iMon, iYear)	Constructs a date from a day, month and year
nday (Date)	Returns the number of days since 01/01/1600
nday$ (iNbDays)	Converts a number of days since 01/01/1600 into a date
dayn (Date)	Returns the day number in the week (1 to 7)
week (Date)	Returns the week number in the year (1 to 52)
aweek (iWeek, iYear)	Date of the beginning of the week indicated by iWeek
eomonth (Date)	Returns a date corresponding to the end of the month
addmonth (Date, iNb)	Adds a number of months to a date
format$	Date formatting according to a format string

Numerical Functions

Numerical functions

`abs ( iNb )`	Absolute value
`mod ( iInt, iDiv )`	Modulo – Returns the remainder of a division
`rnd ( iNb )`	Random number between 0 and iNb (excluded)
`sgn ( iNb )`	Returns the sign of a numerical value (-1, 0, 1)
`sqr ( iNb )`	Square root
`ar2 ( iNb )`	Rounds to 2 decimals
`arr ( iNb, nPrec )`	Rounds to a precision of nPrec (e.g. 0.0001)
`fix ( iNb )`	Trucates decimals from iNb
`int ( iNb )`	Integer part of a numerical value
`fac ( iInt )`	Factorial
`anp ( iN, iP )`	Statistics: Arrangements of **iP** in **iN**
`cnp ( iN, iP )`	Statistics: Combinations of **iP** in **iN**
`format$`	Formats a numerical value according to a format string

Miscellaneous Functions

Miscellaneous Functions

`find ( Value, List )`	Searches for a value in a list or array. Returns the rank where found or 0.
`uni ( List )`	Returns index of first duplicate value found in list or array.
`min ( List )`	Minimum value in a list or array
`max ( List )`	Maximum value in a list or array
`avg ( List )`	Mean Average of values in a list or array
`var ( List )`	Variance of a list or array
`sum ( List )`	Sum of a list or array
`prd ( List )`	Product of a list or array
`sigma( [Var=]iStr, iEnd, Expr )`	Sum of Expr with Var going from iStr to iEnd
`[S]indcum`	Default loop variable for sigma function if not specified

User Defined Functions

- **User defined** functions

User defined functions (**Funprog**) may be used in the same way as Sage X3 functions or reserved words

- In the Sage X3 code
- In parameters and expressions throughout the system
- In the calculator

Syntax:

func *<TRT>.<FUNCTION>***(...)**

Where TRT is the program name and FUNCTION is the function name

```
# Find cubical root of number
Local Decimal I
Local Decimal CUBROOT

I = 278.456
CUBROOT = func XUTIL.CROOT(I)
Infbox "The cubical root of"-num$(I)-"is"-num$(CUBROOT)
End
```

XUTIL.src

```
# Cubical root function
Funprog CROOT(VAL)
Value Decimal VAL
End VAL^(1/3)
```

3.5 – System Variables

System Variables ([S])

Tables

adxdlrec	Number of records deleted using `Delete` instruction
adxuprec	Number of records modified using `Update` instruction
fstat	Return status for all Table instructions (`Read`, `Write`, `Readlock` etc.)
fileabre(5)(101)	Abbreviations of all open tables
filename(255)(101)	Names of all open tables
lockwait	Number of seconds to wait for locking trials

Sequential Files

adxifs	Field separator
adxirs	Record separator
adxium	File type (Ascii, UTF-8, UCS-2)

Dates

datesyst	System date (client workstation)
adxdcs	Base date for 2-digit year numbering (1940 by default)

Application

nomap	Current application folder
adxmother	Parent application folder
adxdir	Sage X3 runtime installation path on the server

System Variables ([S])

- **Display and user interface**

currbox	Current active left browser
indice	Current index for array fields
nolign	Current line number for table blocks
mkstat	Error status in field entry
status	Return status after user action
adxgtb	Entry or exit status in a datasheet

- **Resources**

adxmpr	Maximum number of resident programs
adxmso	Maximum number of concurrent sequential file channels
adxmto	Maximum number of concurrent open tables
adxmbm	Maximum number of message buffers
adxmxl	Number of elements displayed at a time in left browsers
maxmem	Maximum allocated memory for each session
dbgstr	List of default classes (files/screens)

3.6 – Functions, Subprograms And Scope

Variable Scope

- Local variables are local to a section, subprogram or function
- Global and system variables span the entire code

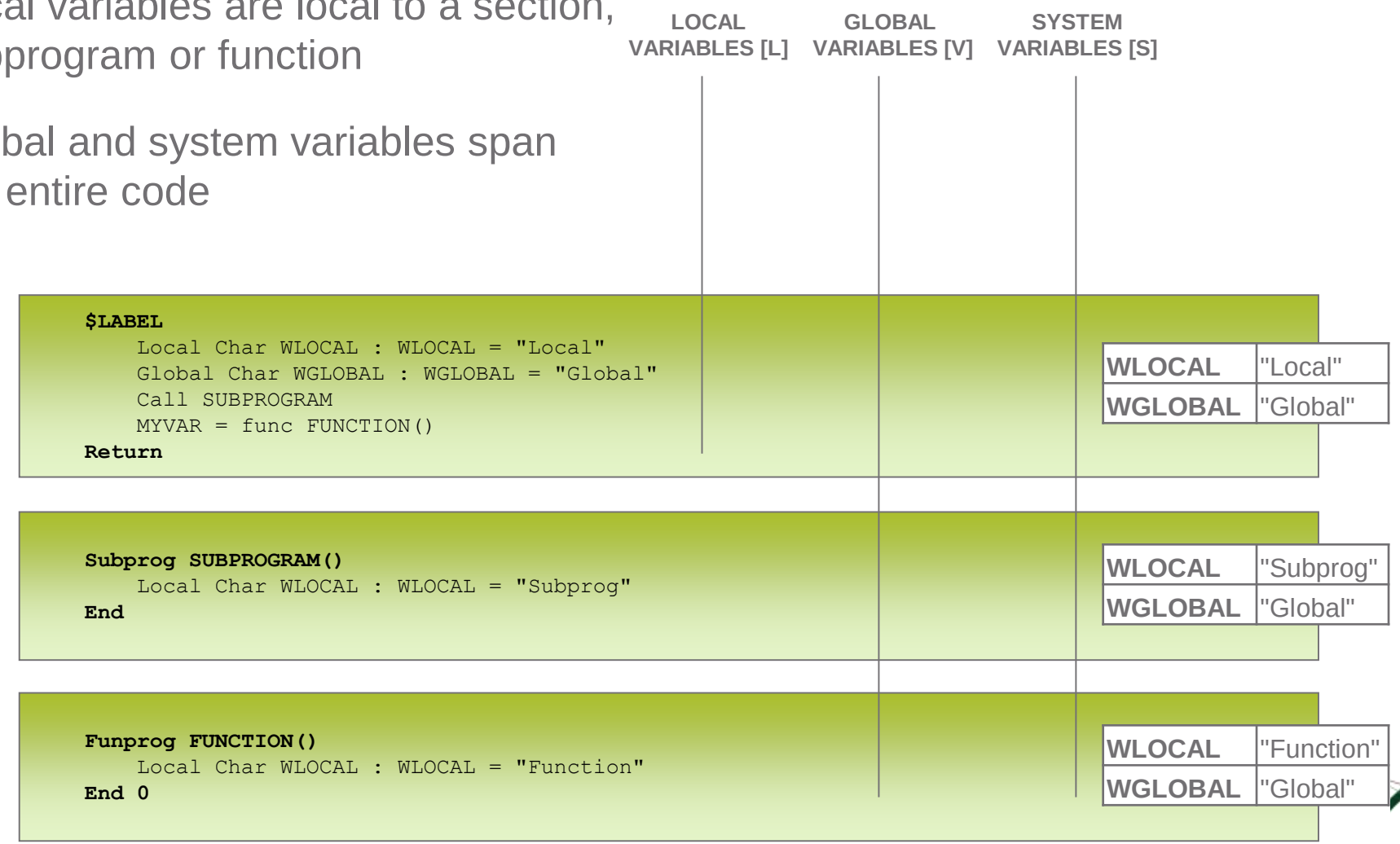


Table Scope

- Table classes are global to all sections, subprograms and functions.
- When a table is declared in different scope sections, with the same abbreviation, a different [F] buffer is opened locally.

Screen Scope

- **Screen classes are global to all sections, subprograms and functions.**
- When a screen is declared in different scope sections, with the same abbreviation, the *same [M] buffer* is used, which means additional **`Mask`** statements are disregarded.

```
$LABEL
    Local Mask BPR0 [BPR0]
    [M:BPR0]BPRNUM = "CUST01"
    Call SUBPROGRAM()
    MYVAR = func FUNCTION()
Return
```

[M:BPR0]BPRNUM	"CUST01"
[M:BPR0]BPRNUM	"CUST02"
[M:BPR0]BPRNUM	"CUST03"

```
Subprog SUBPROGRAM()
    Local Mask BPR0 [BPR0]
    [M:BPR0]BPRNUM = "CUST02"
End
```

[M:BPR0]BPRNUM	"CUST02"

```
Funprog FUNCTION()
    [M:BPR0]BPRNUM = "CUST03"
End 0
```

[M:BPR0]BPRNUM	"CUST03"

3.7 – Reusable Subprograms

Trace File Management

- **Trace file** management

Opening a trace file:

```
# Open trace file with title
Call OUVRE_TRACE("Development Course Example") From LECFIC
```

Inserting statements in a trace file:

```
# Information message in trace file
# If no trace file is opened, message will display on screen as information box
Call ECR_TRACE("This is a simple message.", 0) From GESECRAN

# Error message in trace file (special formatting)
# If no trace file is opened, message will display on screen as error box
Call ECR_TRACE("Error!!", 1) From GESECRAN
```

Closing and viewing trace files:

```
Call FERME_TRACE From LECFIC    :# Close trace file
Call LEC_TRACE From LECFIC :# Trace view screen opens. User has several choices
Call SUP_TRACE From LECFIC :# Force deletion after user has viewed the file
```

Message Boxes

- Simple **message boxes**

Information message:

```
# An information message
Call ECR_TRACE("This is an information message", 0) From GESECRAN

# Another way:
Call MESSAGE("This is another info") From GESECRAN
```

Error messages:

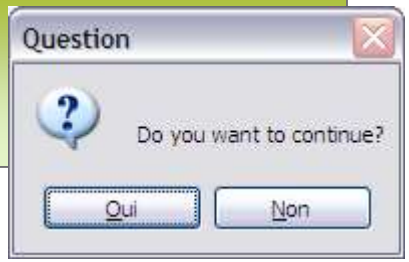
```
# Error message
Call ECR_TRACE("This is an error message", 1) From GESECRAN

# Another way:
Call ERREUR("This is another error message") From GESECRAN

# Error message with custom title:
Call ERRTIT("Application error", "And the error is...") From GESECRAN

# Fatal error:
Call ERREND("Fatal error!!") From GESECRAN
```

Question Boxes

Question boxes

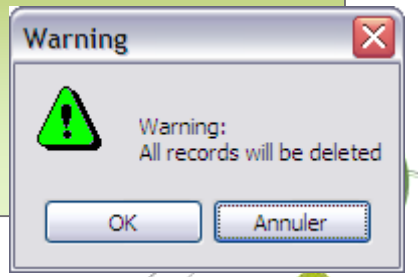
Yes/No boxes:

```
# Default value = Yes (2)
Local Integer YESNO : YESNO = 2 :# Yes
Call OUINON("Do you want to continue?", YESNO) From GESECRAN
If YESNO = 2
   ...
```



Ok/Cancel boxes:

```
# Default value = Cancel
Local Integer OKCAN : OKCAN = 1 :# Cancel
Call AVERTIR("Warning:\All records will be deleted", OKCAN) From GESECRAN
If OKCAN = 1
   ...
Else
   ...
Endif
```

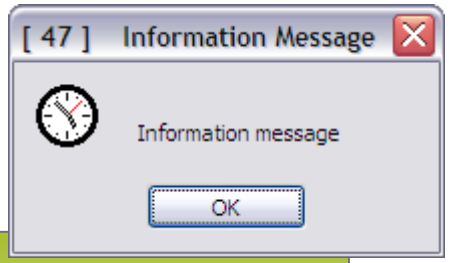


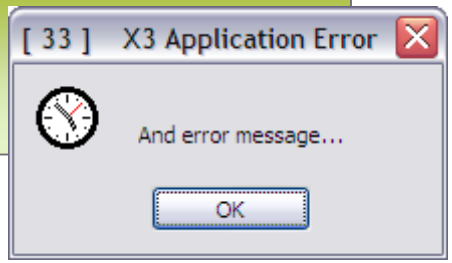
Timeout Subprograms

- **Timeout** subprograms

Information / warning:

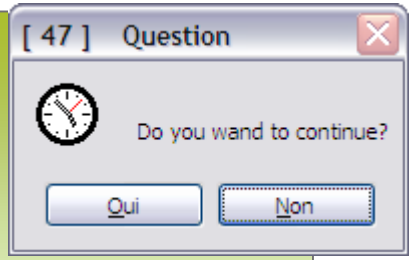
```
# Information message
Call ERREURT("Information message", 0) From GESECRAN

# Error message
Call ERREURT("And error message...", 1) From GESECRAN
```





Question and Ok/Cancel boxes:

```
# Default value = No (1)
Local Integer YESNO : YESNO = 1 :# No
Call OUINONT("Do you wand to continue?", YESNO) From GESECRAN
...

# Default value = Ok (1)
Local Integer OKCAN : OKCAN = 1 :# Ok
Call AVERTIRT("This is dangerous.", OKCAN) From GESECRAN
```



- LAB2: Playing with the X3 language

  Exercice 4: Playing with trace files

  Exercice 5: Playing with boxes

EXERCICES

3.8 – Performances

General rules

- Prefer using the SQL translated instructions and operators

  len, ascii, chr$, toupper, tolower, or, and, not, left$, right$, mid$, seg$, >,>=,<,<=,<>,=,+,-,*, /

- Specific case of ***Find***

  For [XXX] Where Find(CHAMP,3,5,9) <> 0

  Is translated to

  Select ... Where CHAMP in (3,5,9)

  For [XXX] Where Find(CHAMP,3,5,9) = 0

  Is translated to

  Select ... Where CHAMP not in (3,5,9)

Requests optimization

- Send a minimum number of requests to the database
  Use ***Link*** instead of several ***Read***
  Use ***For*** instead of several ***Read***
- Prefer using **AND** rather than **OR**
  For [XXX] Where MYINT>=3 and MYINT<=5
  is better than
  For [XXX] Where MYINT=3 or MYINT=4 or MYINT=5
- Limit the data on complex and costly algorithms
  Use ***Columns*** before ***For*** and ***Rewrite*** to avoid a *Select * From*
  Use ***Update*** to update set of data, on a defined number of fields

Contents

- Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- 3. Introduction To The Sage X3 Language
  - 3.1 – Variables And Variable Classes
  - 3.2 – Operators
  - 3.3 – Instructions
  - 3.4 – Functions
  - 3.5 – System Variables
  - 3.6 – Functions, Subprograms And Scope
  - 3.7 – Reusable Subprograms
  - 3.8 – Performances
- **4. Basic Development: The Object Template**
  - **4.1 – The Object Template**
  - **4.2 – Field Actions**
  - **4.3 – Object Actions**
  - **4.4 – STD snippet: Database transaction management**
- 5. Debugging Sage X3 Code
  - 5.1 – The Debugger

4.1 – The Object Template

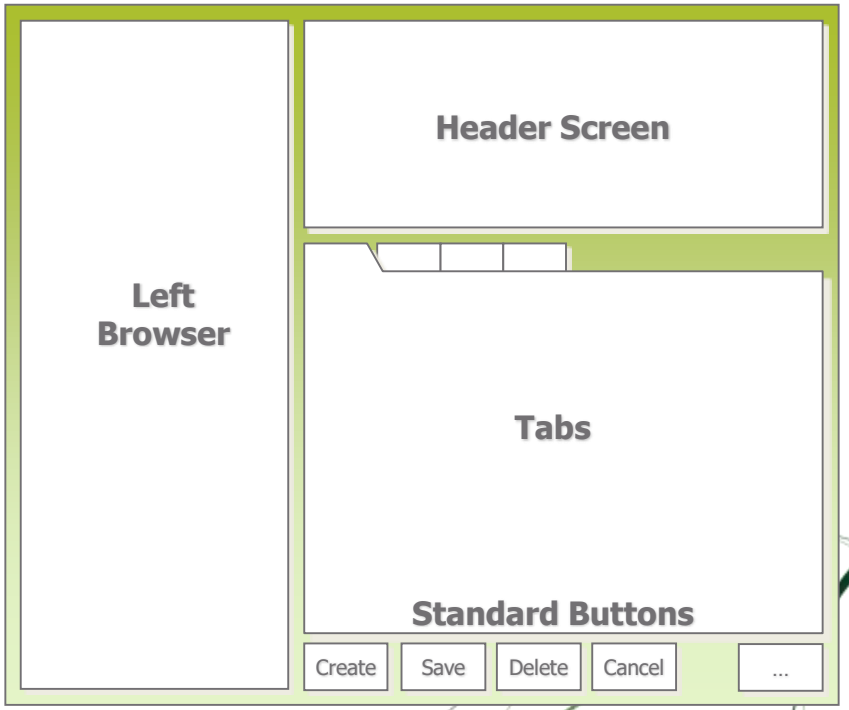
Introduction

- The Object Template performs standard maintenance (Creation/Modification/Deletion) of Sage X3 records.

It is based on one or several Sage X3 tables.

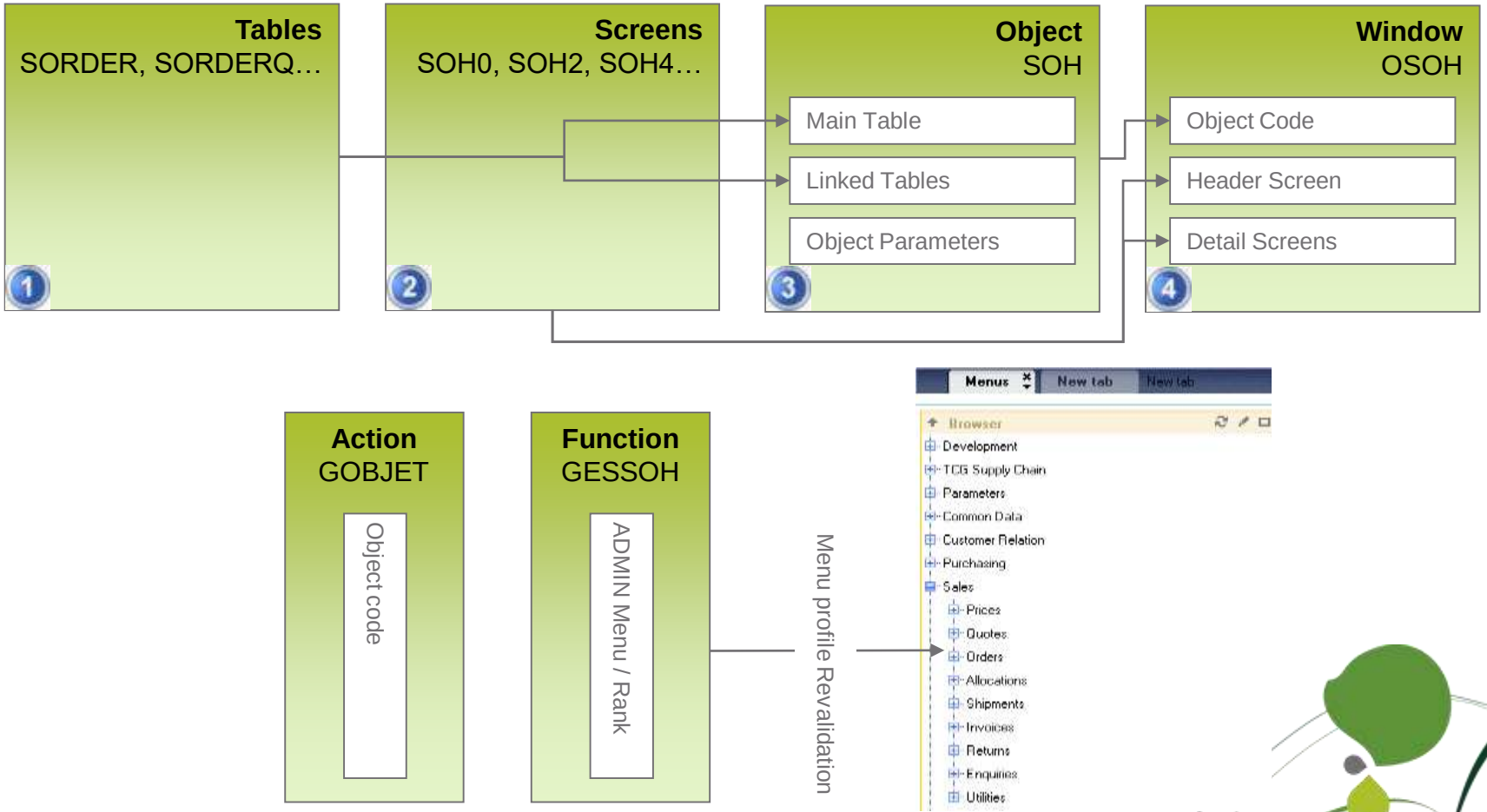
- The Object Template encapsulates access right management, default left browsers (Selection, picking, links), filtering possibilities, advanced selection etc., and above all database transactions.

- It may be customised (Parameterisation):
  - Left browsers
  - Detail Tab contents (Transactions)
  - Links to other functions
    (Navigation, tunnels, 'File' menu operations etc.)
  - Miscellaneous parameters
    (Object properties, reports used etc.)

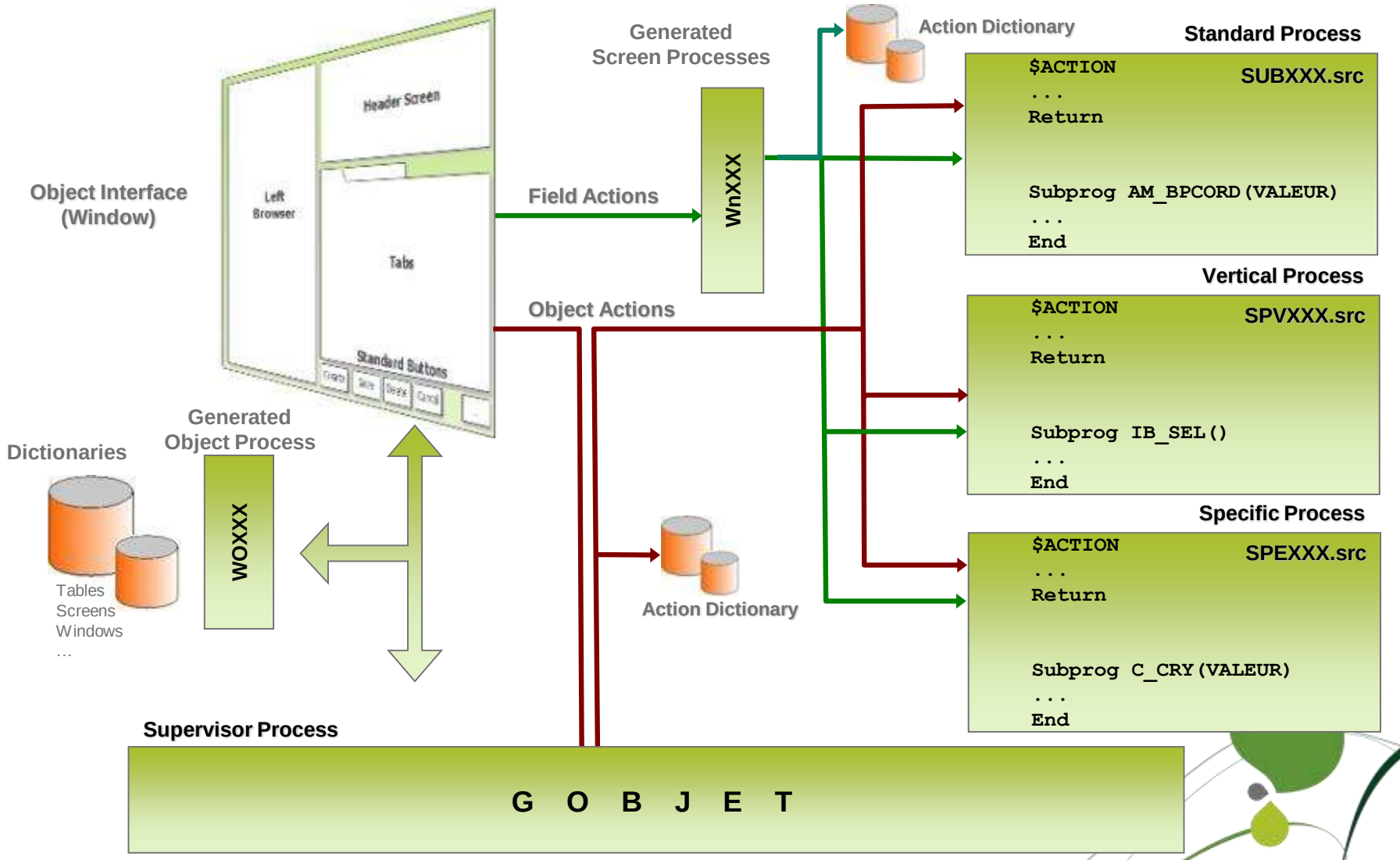


Creating An Object

- Creating a new Object and the function/menus that link to it



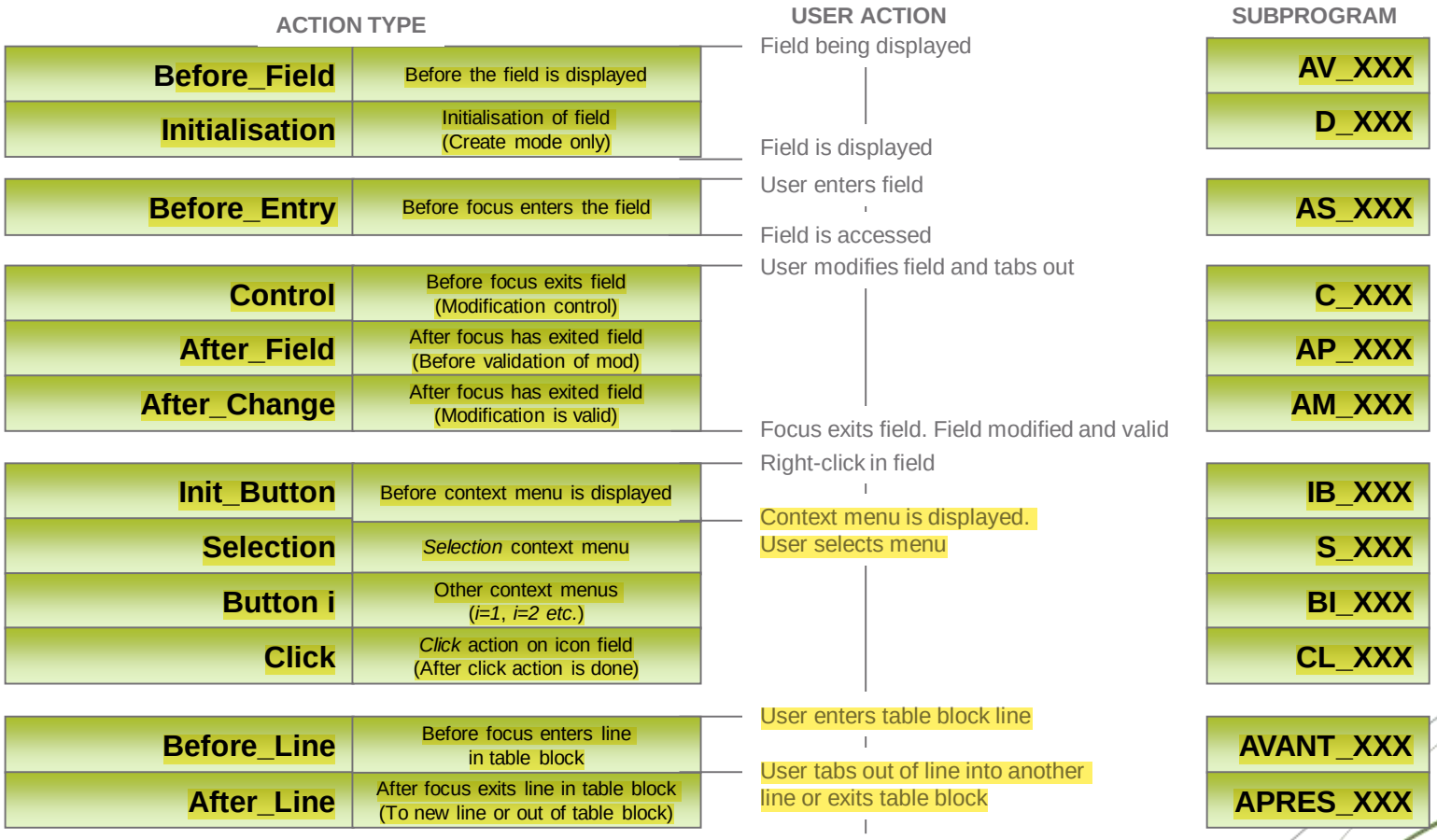
Objects: Behind The Scene



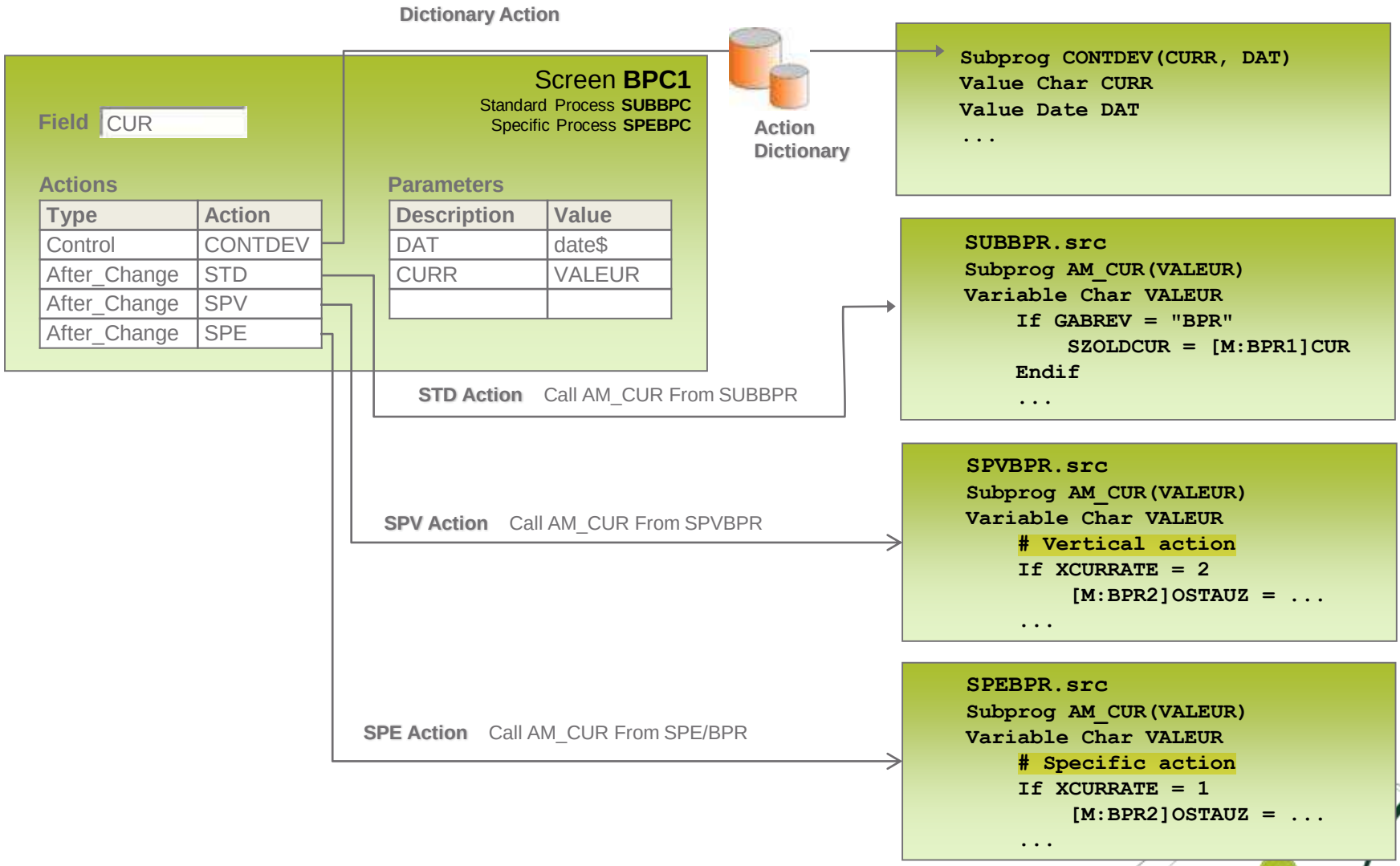
4.2 – Field Actions

Action Types

- Field actions are triggered by different events when the user navigates through screen fields.
- Field actions may be linked to direct code (subprograms in the screen processes) or to actions from the action dictionary.



Defining Field Actions



Variable VALEUR

- Local variable **VALEUR**

  Contains current field value (modification in progress)

  `[M]XXX` contains the value before the field was modified (For all actions until `AM_` included)

  After the `AM_` action, `[M]XXX` is then changed to the content of `VALEUR`

```
##############################
# Field BPCORD (Customer)
# in Sales Orders
#
Subprog C_BPCORD(VALEUR)
Variable Char VALEUR

    If VALEUR = "BERT" & [M]BPCORD = "MARY"
        GMESSAGE = "Changing customer MARY to BERT!"
    Endif

End
```

New value

Previous value

Field Access: mkstat

- System variable **[S]mkstat** pilots field access and field value validation.
- It is set inside field actions for the purpose of:
  Allowing / preventing the user from accessing the field
  Allowing / preventing the user from exiting the field after modifying it.
- Possible values:

mkstat = 0	Normal behaviour
mkstat = 1, 2	Error: Field is locked. User is locked in or out.

```
################################
# Field BPCORD (Customer)
# in Sales Orders
#
Subprog C_BPCORD(VALEUR)
Variable Char VALEUR
    # Check BP country
    Read [BPC]BPC0 = VALEUR
    If [F:BPC]CRY = "US"
        mkstat = 2 :# Lock user in field
        GMESSAGE = "USA orders prohibited."
    Endif
End
```

Enabling / Disabling Fields

- Field status instructions may be used to enable or disable screen fields depending on values in the current field, mainly, in **After_Modification** or **Control** actions for the current field.
- Field status instructions:

`Grizo`	Completely disables a screen field (Or a complete screen, or a block)
`Diszo`	Locks a screen field (Preserves context menus and focus)
`Actzo`	Activates a screen field (Or a complete screen, or a block)

- Additional actions to refresh or reset screen fields:

`Raz`	Deletes a screen field value
`Affzo`	Refreshes a screen field (Necessary to display new values)
`Effzo`	Deletes a screen value and refreshes (**`Raz + Affzo`**).

Global Variables

- Global variables available during field actions

Variable	Type	Meaning
GREP(1)	Char(1)	Standard action in progress (A / C / D / M)
GIMPORT	Integer	Import mode indicator (0 / 1)
GSERVEUR	Integer	Batch mode indicator (0 / 1)
GCOUL(0..7)	Integer	Field colour
GBOUT(1..20)	Char(35)	Title of context menu item
GBOUTS	Char(35)	Title of selection context menu
GBOUTA	Char(35)	Advanced selection context menu title
GBOUTI	Char(35)	Icon click title (tooltip)
GMESSAGE	Char(250)	Information, warning or error message
GERR	Integer	Message type (Warning, critical error etc.)
GMENLOC(0..123)	Integer	Hides a local menu item
GOK	Integer	Return status *inside transactions only*

System Variables

- System variables available during field actions ([S] class)

Variable	Type	Meaning
nolign	Integer	Current line number in table blocks
indice	Integer	Current index of array variable (normal blocks)
status	Integer	Return status at the end of an entry
mkstat	Integer	Status when exiting a field
fstat	Integer	Status for table operation (Read, Write, Rewrite, Lock etc.)

Common Field Actions

- **Before_Field**
  Prevent the field from being displayed and entered
  Reset the field and prevent it from being displayed and entered
  Assign a font colour (Table block fields only)
- **Init**
  Initialise the field value
- **Before_Entry**
  Prevent user from entering field (Focus jumps to next field)
  Delete field value and prevent user from entering field
  Disable local menu (Combo box) items
- **Control, After_Field, After_Modification**
  Display an information/warning/error message
  Lock user inside the field
- **After_Field, After_Modification**
  Populate and refresh other fields
  Disable, enable other fields depending on current value

4.3 – Object Actions

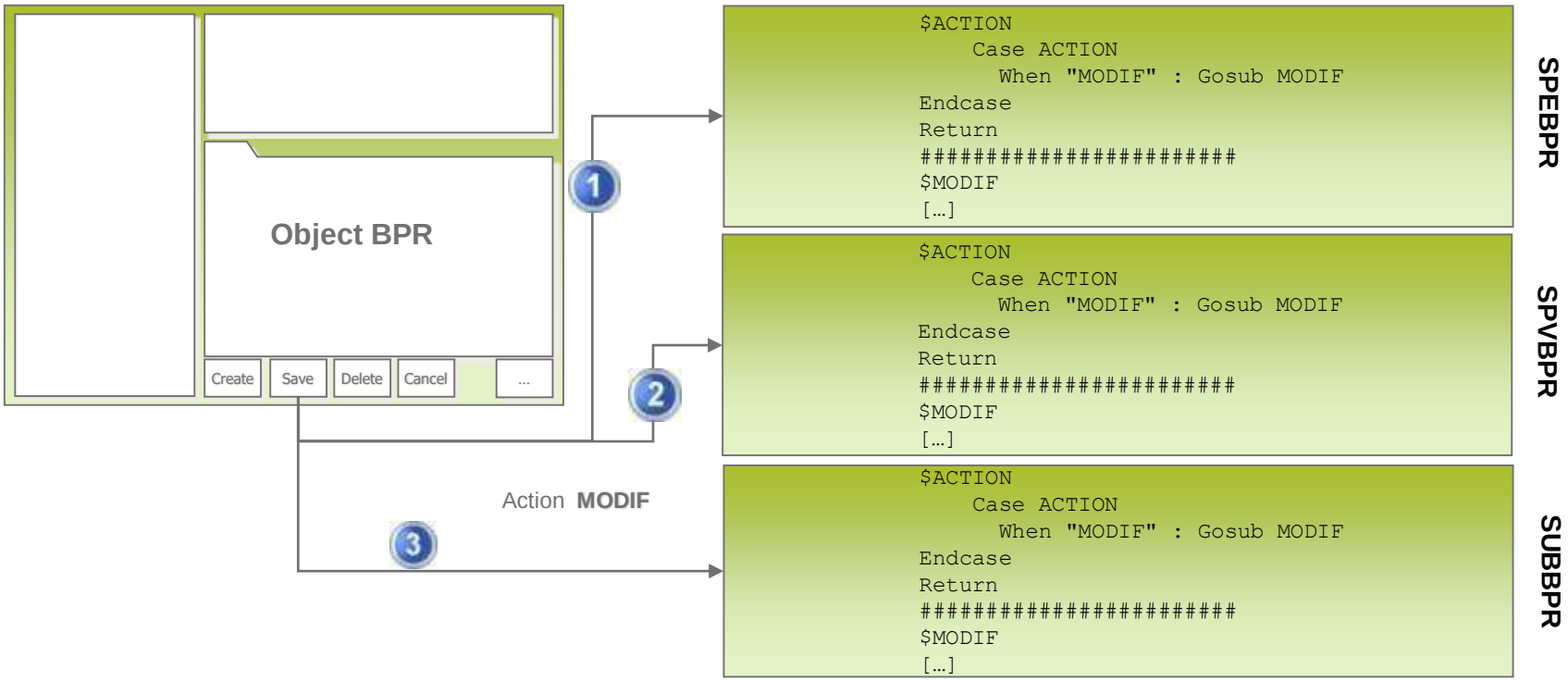
Object Actions

- Object actions are global events that are executed when users maintain or navigate through records in an object.
- They are called via the **$ACTION** section in the following programs, where XXX is the object code. Variable **ACTION** holds the current action code:

  SUBXXX for standard actions

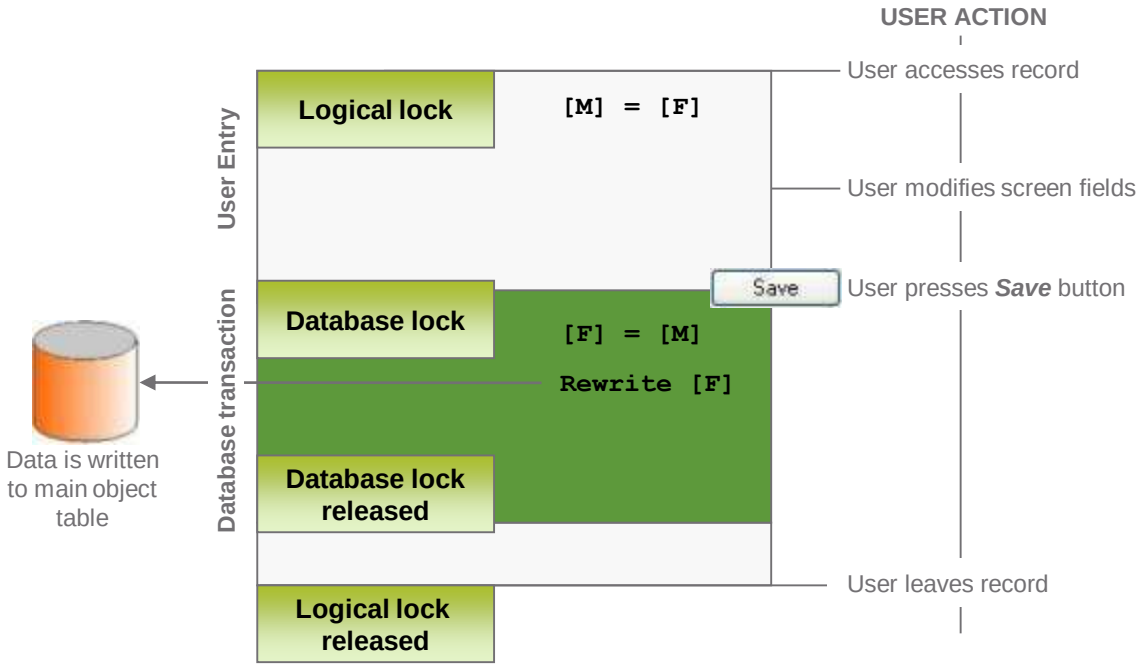
  SPVXXX for vertical actions (Called before the standard action if the SPVXXX program exists)

  SPEXXX for specific actions (Called before the vertical action if the SPEXXX program exists)



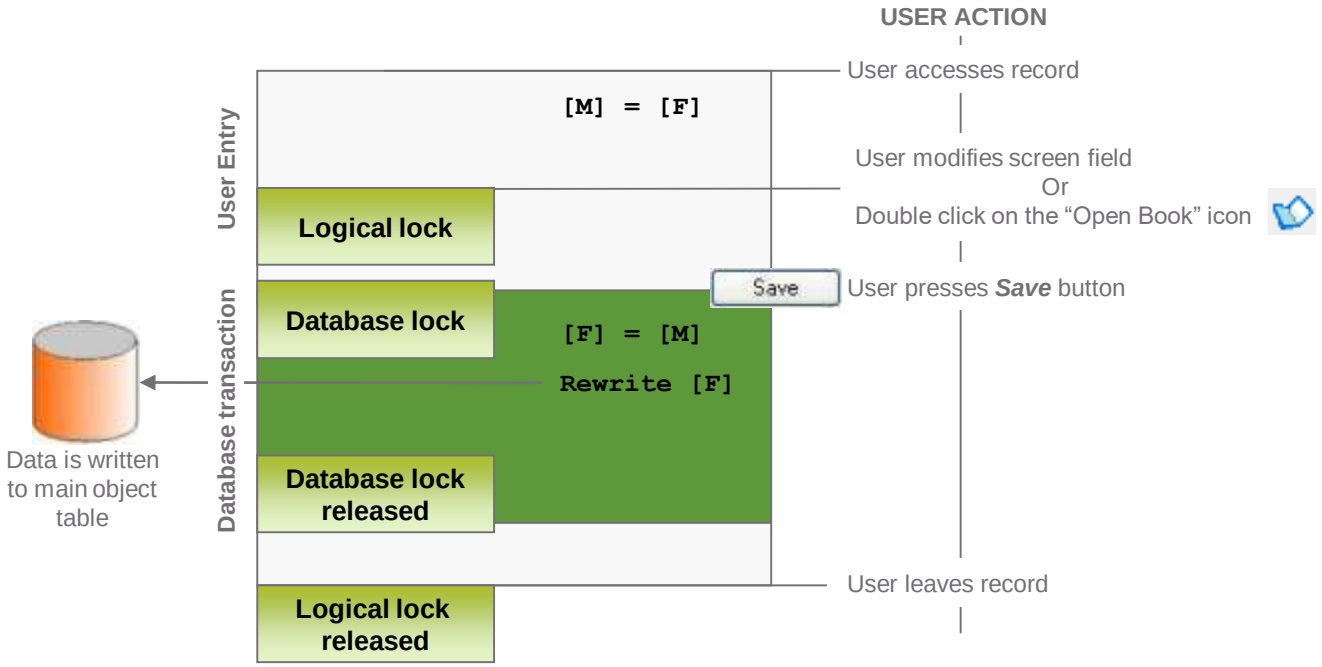
Object Kinematics

- Object behaviour: Tables and screens



Object Kinematics

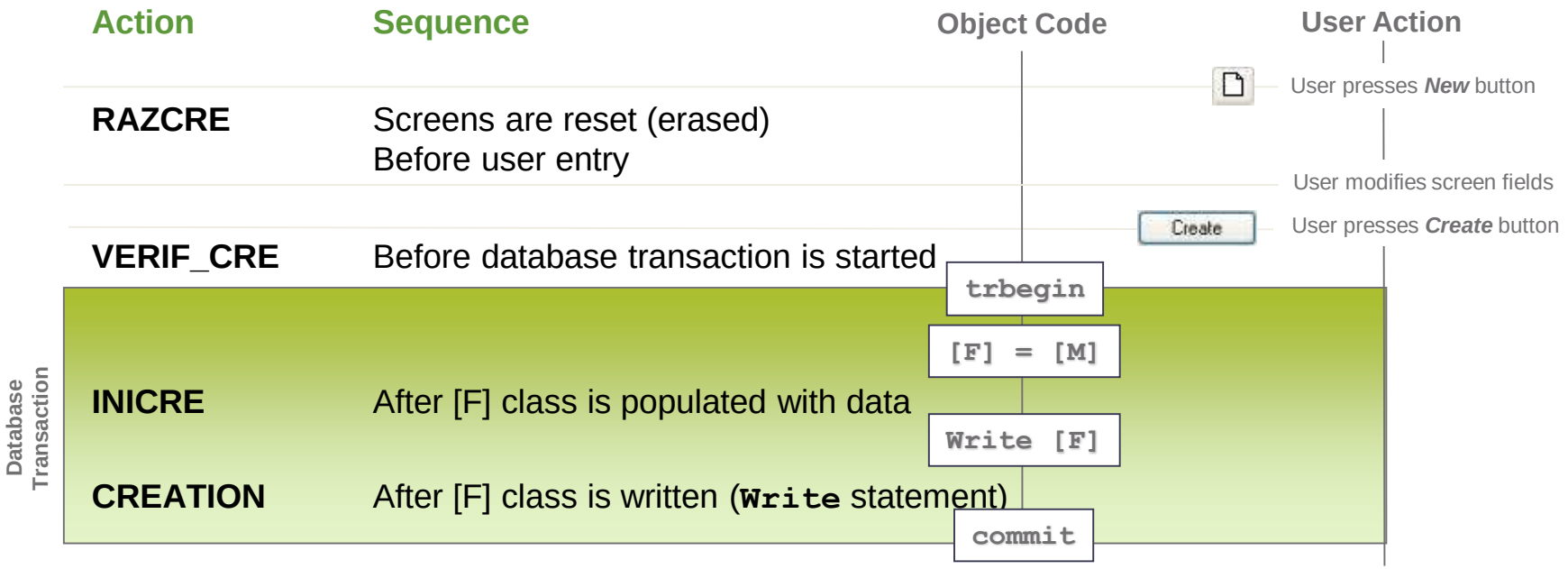
- Object behaviour: Lock in modification



Creation Mode

- Simple objects – **CREATION** mode

Action	Sequence
RAZCRE	Screens are reset (erased) Before user entry
VERIF_CRE	Before database transaction is started
INICRE	After [F] class is populated with data
CREATION	After [F] class is written (**`Write`** statement)
APRES_CRE	After database transaction has been committed
AB_CREATION	When transaction is abandoned (Rollback)



Updating Other Tables During CREATION

- The object automatically manages error messages, rollbacks and retries when there is a database error or locking conflict during the transaction.

  Database status is communicated to the object using variable **[V]GOK**.

```
$CREATION

    # Modify record in linked table
    Readlock [XXX]XXX0 = KEY
    If fstat = 1 :# Locking problem
        GOK= -1
        GLOCK = "My table"
        Return
    Elsif fstat  :# Record not found
        GOK = 0
        Call RSTA("XXX", KEY) From GLOCK
        Return
    Endif
    Rewrite [XXX]
    If fstat
        GOK = 0
        Call FSTA("XXX") From GLOCK
        Return
    Endif
Return
```

Instructs the object that a lock conflict has occurred. The object will automatically manage error messages, retries and rollbacks.

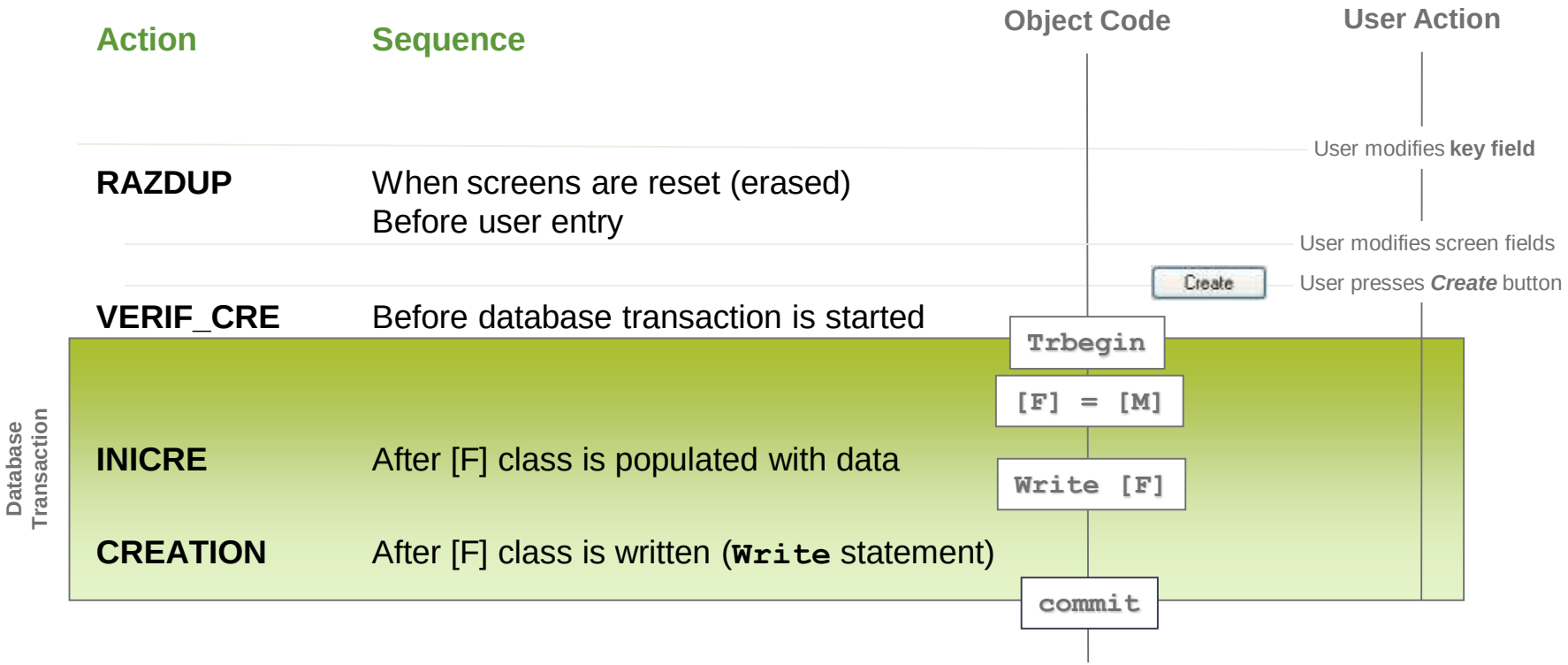
Instructs the object that database error has occurred. The object will automatically manage rollbacks.

Instructs the object that database error has occurred. The object will automatically manage rollbacks.

Duplication Mode

- Simple objects – **DUPLICATION** mode

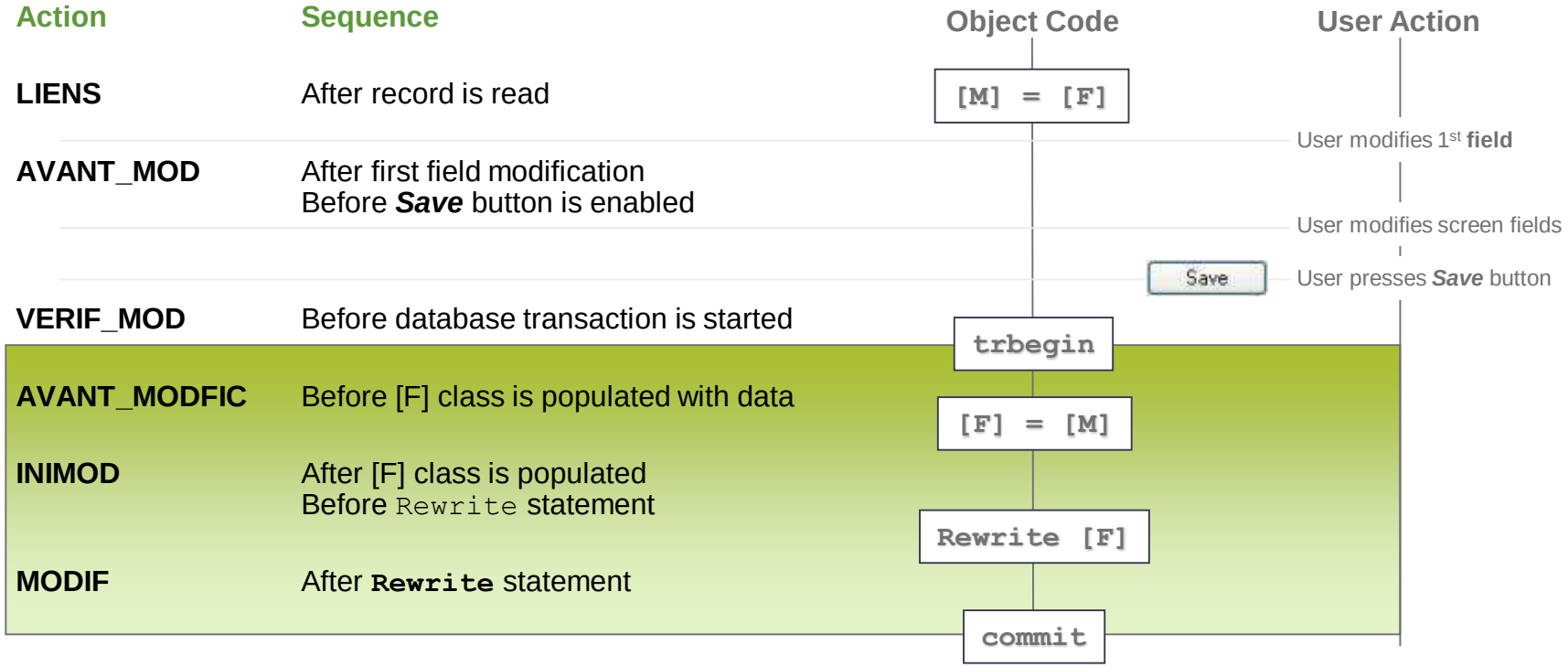
Action	Sequence
RAZDUP	When screens are reset (erased) Before user entry
VERIF_CRE	Before database transaction is started
INICRE	After [F] class is populated with data
CREATION	After [F] class is written (**`Write`** statement)
APRES_CRE	After database transaction has been committed
AB_CREATION	When transaction is abandoned (Rollback)



Modification Mode

- Simple objects – **MODIFICATION** mode

Action	Sequence
LIENS	After record is read
AVANT_MOD	After first field modification Before ***Save*** button is enabled
VERIF_MOD	Before database transaction is started
AVANT_MODFIC	Before [F] class is populated with data
INIMOD	After [F] class is populated Before `Rewrite` statement
MODIF	After **`Rewrite`** statement
APRES_MOD	After database transaction has been committed
AB_MODIF	When transaction is abandoned (Rollback)



Updating Other Tables During MODIFICATION

- Writing data to linked tables during **MODIF** action

```
$MODIF

    Readlock [XXX]XXX0 = INDEX_VALUE

    If fstat = 1          :# Locking conflict (Record is locked)
       GOK = -1           :# Instructs object to retry transaction
                           # according to retry parameters
        GLOCK = "Table name/code + key"
        Return
    Elsif fstat
       GOK = 0            :# Read error: Cannot find record
        Call RSTA("Table name/code + key", INDEX_VALUE) From GLOCK :# Generates proper
                                                                   # error message
        Return
    Endif

    Rewrite [XXX]
    If fstat              :# Write error (duplicate records, etc.)
       GOK = 0
        Call FSTA("Table name/code + key") From GLOCK :# Generates proper
                                                      # error message
        Return
    Endif

Return
```

Instructs the object that a lock conflict has occurred. The object will automatically manage error messages, retries and rollbacks.

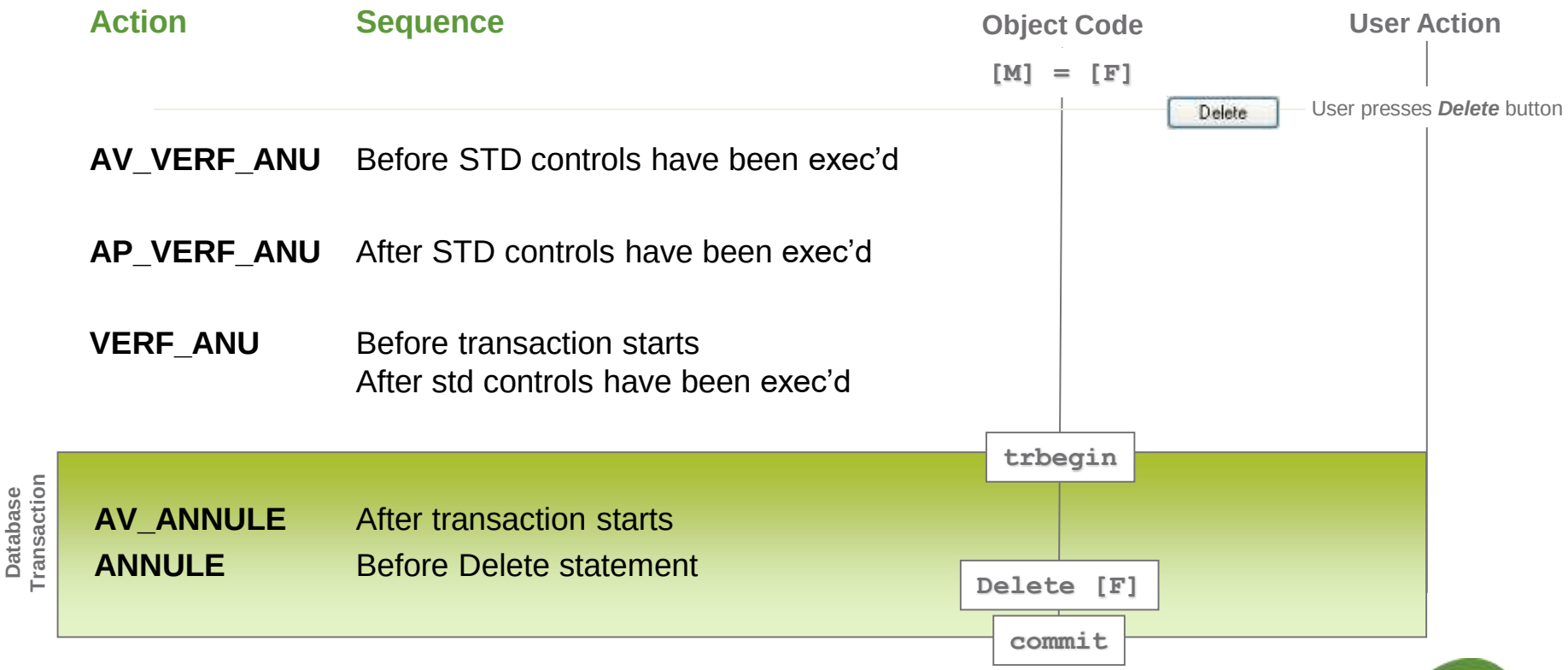
Instructs the object that database error has occurred. The object will automatically manage rollbacks.

Instructs the object that database error has occurred. The object will automatically manage rollbacks.

Deletion Mode

- Simple objects – **DELETION** mode

Action	Sequence
AV_VERF_ANU	Before STD controls have been exec'd
AP_VERF_ANU	After STD controls have been exec'd
VERF_ANU	Before transaction starts After std controls have been exec'd
AV_ANNULE	After transaction starts
ANNULE	Before Delete statement
AP_ANNULE	After deletion transaction has been committed



Buttons

- Button actions

SETBOUT

- Button activation/deactivation
- Called whenever the context changes and buttons must be redrawn

AVANTBOUT (Specific buttons)

- Called when a *specific* button is pressed, before the button action/code is executed
- Button code available in variable **`BOUT`**

AVANT_XXX (Standard buttons)

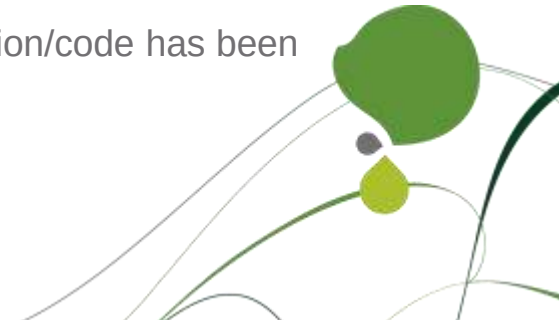
- Called when a *standard* button is pressed, before the button action/code is executed
- Where XXX is the standard button code (**END**, **ABA**, **NEW** etc.)

EXEBOUT (Specific buttons)

- Called when a *specific* button is pressed, after the button action/code has been executed
- Button code available in variable **`BOUT`**

XXX (Standard buttons)

- Called when a *standard* button is pressed, after the button action/code has been executed
- Where XXX is the standard button code (**END**, **E**, **C**, **S** etc.)



Menus

- Menu actions (Menu bar)

SETBOUT

- Button activation/deactivation
- Called whenever the context changes and buttons must be redrawn

AVANTBOUT

- Called when a menu item is selected, before executing the action/code attached to the menu
- Menu code available in variable **`BOUT`**

STATUT

- Called after the menu action/code has been executed
- Menu code available in variable **`BOUT`**

The GOK Variable

- The **[V]GOK** variable

GOK is used to manage transaction status *inside object transactions*

GOK = -1

- Lock in progress on another workstation
- Transaction is rolled back and retried

GOK = 0

- Read/Write or other critical error
- Transaction is abandoned

GOK = 1

- Normal value (No error)

The OK Variable

- The `[L]OK` variable

`OK` is used for some of the object actions to manage return status *outside transactions*

`OK = 1`

- Normal value (No error)

`OK = 0`

- Error status
- Object will react in various ways depending on context and action in progress

The APRES_MODIF Action

- The **APRES_MODIF** action

This object action is executed when a field is modified

- After the field AM_ (After Change) action
- Before the [M] class is updated with the new value

Available variables

- **COUZON**: Name of current field
- **CZ**: Current value of field
- **COUIND**: Current index (Array fields)

General Object Actions

- Object being opened

AVANT_OUVRE	Before object is opened
DEFTRANS	Before entry transaction is chosen (For objects with more than one active window)
VARIANTE	Analysis of each active entry transaction for object
SETTRANS	After entry transaction has been chosen (For objects with more than one active window)
OUVRE	After tables linked to object have been opened
BOITE	Before object window is opened
TITRE	Before active tab is selected and window / browser titles are displayed
OUVRE_BOITE	After window has been opened
AFFMASK	After screens and tables are opened, before populating the screens
FILTRE	Before table records are filtered
DROIT	Before access rights to current record are checked
CLE_GAUCHE	Before left browser (Simple or picking) is displayed (Except for Last Read)
VERROU	After current record has been locked by current user (Symbolic lock)
LIENS	After linked tables have been read
STYLE	Before fields are displayed (Used for display styles)
SETBOUT	When window/buttons are redrawn (After user action or refresh)

General Object Actions

- Object being closed

User presses END or closes

APRES_CHOI	After user has selected window option / Before the action is executed
FIN	END button pressed (See XXX button action)
AVANT_DEVERROU	Before unlocking the record
DEVERROU	After unlocking the record

Record unlocked

FERME	After object has been closed

Object window is closed

General Object Actions

- Menu item selected

User selects menu item

APRES_CHOI	After user has selected window option / Before the action is executed
AVANTBOUT	Before menu item or button is executed

Menu action is executed

STATUT	After menu item action is executed
FIN_ACTION	End of user action (Button, menu, left list record selection etc.)
SETBOUT	When window/buttons are redrawn (Window is refreshed)

Window menus refreshed

- Button pressed

User selects menu item

APRES_CHOI	After user has selected window option / Before the action is executed
AVANTBOUT	Before menu item or button is executed

Menu action is executed

EXEBOUT	After button action is executed
SETBOUT	When window/buttons are redrawn (Window is refreshed)

Window menus refreshed

General Object Actions

- ‘Save’ button pressed

APRES_CHOI	
AVANT_ACT	Std button has been activated (*Create*, *Save* etc.)
EXEACT	Before standard action is executed
VERIF_MOD	
AVANT_MODFIC	
INIMOD	
MODIF	
APRES_MOD	
FIN_MOD	After transaction has been committed
DEGRISE	After object key field has been enabled
FIN_ACTION	After standard button action has been executed
SETBOUT	

Standard **modification** mode actions

General Object Actions

- Screen field modified (First modification)

APRES_MODIF	
AVANT_MOD	Called for first modification, before object enters modification mode
GRISE	After object key field has been disabled
SETBOUT	

HEADER and DETAIL Handling

- The Object manager only handles one table.
- If you want to manage header and detail tables, the supervisor provides you with some library calls to handle this situation.

Prerequisites:

In the detail screen, you have to add a line counter. (type L , hidden)

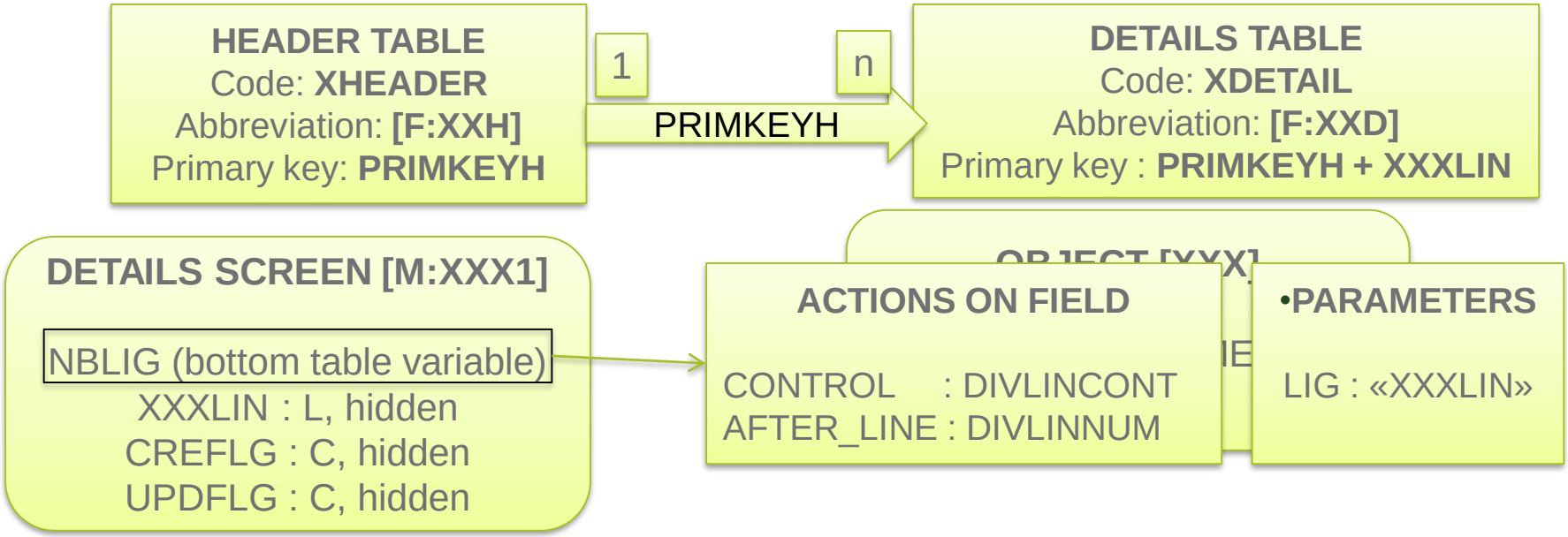
This variable should exist in the detail table, and be part of the index for this table. On the grid variable, 2 events should be added:

Control : **DIVLINCONT**

After_Line : **DIVLINNUM**

They require one parameter : <LIG> , which should point to the line counter variable.

In the grid, 2 extra (hidden) variables should be added: **CREFLG** and **UPDFLG**, both of type C.



PROCESS : SPEXXX

ACTION TO USE	CODE TO USE
OUVRE	GOSUB DECLARE FROM TABLEAUX
FILTRE	Default File [XXH]
LIENS	GOSUB LIENS FROM TABLEAUX
CREATION	GOSUB CREATION FROM TABLEAUX
MODIF	GOSUB MODIF FROM TABLEAUX
ANNULE	GOSUB ANNULE FROM TABLEAUX
APRES_CRE	GOSUB LIENS FROM TABLEAUX
APRES_MOD	GOSUB LIENS FROM TABLEAUX
INICRE_LIG	[F:XXD]PRIMKEYH = [M:XXX0]PRIMKEYH
DEFLIG	See next slide

HEADER TABLE
Code: **XHEADER**
Abbreviation: **[F:XXH]**
Primary key: **PRIMKEYH**

1 — PRIMKEYH — n

DETAILS TABLE
Code: **XDETAIL**
Abbreviation: **[F:XXD]**
Primary key : **PRIMKEYH + XXXLIN**

DETAILS SCREEN [M:XXX1]

NBLIG (bottom table variable)
XXXLIN : L, hidden
CREFLG : C, hidden
UPDFLG : C, hidden

ACTIONS ON FIELD

CONTROL : DIVLINCONT
AFTER_LINE : DIVLINNUM

•PARAMETERS

LIG : «XXXLIN»

PROCESS : SPEXXX

ACTION TO USE	CODE TO USE

DEFLIG	Default Mask [XXX1] #The screen holding your detail information
	Default File [XXD] #The table holding your detail information
	
	#Filtering expression to read all detail lines related to one header
	CRIT = 'PRIMKEYH="'+[M:XXX0]PRIMKEYH+'"'
	FICLIG = "XDETAIL" #The name of the detail Table
	ABLIG = "XXD" #It's abbreviation
	ZONLIG = "XXXLIN" #The line counter field

HEADER AND DETAIL MANAGEMENT

- LAB1: Domain management

Exercice 3.1: Activity Code
Exercice 3.2: Local Menu
Exercice 3.3: Table
Exercice 3.4.a Process

EXERCICES

4.4 – STD snippet: Database transaction management

Transactions

- **Transactions** as they are managed by the standard object template

  Transaction main procedure:

```
# Update customer ACETEX credit +1000

Local File BPCUSTOMER [BPC]

Call DEBTRANS From GLOCK                        :# Set global variables

$TR1
    [V]GOK = 1                                  :# All OK
    Trbegin [BPC]                               :# Start transaction
    Readlock [BPC]BPC0 = "ACETEX"
    If [S]fstat=1
        [V]GOK=-1
        [V]GLOCK = "BPCUSTOMER"-"ACETEX"
        Goto ROL_TR1                            :# Rollback and retry
    Elsif [S]fstat
        [V]GOK=0
        Call RSTA("BPC","ACETEX") From GLOCK    :# Display proper error mess
        Goto AB_TR1                             :# Abandon transaction
    Endif

    [BPC]OSTAUZ += 1000
    Rewrite [BPC]
    If [S]fstat
        [V]GOK=0
        Call FSTA("BPC") From GLOCK             :# Display proper error mess
        Goto AB_TR1                             :# Abandon transaction
    Endif
    Commit
Return
```

Transactions

- **Transactions** as they are managed by the standard object template

Retry procedure:

```
$ROL_TR1
    Rollback
    Call ROLL From GLOCK                                  :# See how many retries until now
    If [V]GROLL
        Call ECR_TRACE(mess(17,107,1),1) From GESECRAN :# Max number of retries reached
    Else
        Goto TR1                                          :# Can still retry one more time
    Endif
Return
```

Abandon procedure:

```
$AB_TR1
    Rollback                                          :# Roll back
    Call ECR_TRACE(mess(17,107,1),1) From GESECRAN :# Proper error message
Return
```

EXERCICES

HEADER AND DETAIL MANAGEMENT

- LAB1: Domain management

Exercice 3.4.b Process

Contents

- Basic Principles
  - 1.1 – General Architecture
  - 1.2 – Folder Management
  - 1.3 – Application Architecture
  - 1.4 – The User Interface
- 2. The Development Dictionary
  - 2.1 – Data structure
    - 2.1.1 – Table & Views
    - 2.1.2 – Data Types
    - 2.1.3 – Local Menus
    - 2.1.4 – Activity Codes
    - 2.1.5 – Miscellaneous Tables
  - 2.2 – Graphical structure & Interface
    - 2.2.1 – Screens
    - 2.2.2 – Objects
    - 2.2.3 – Windows
    - 2.2.4 – Functions
- 3. Introduction To The Sage X3 Language
  - 3.1 – Variables And Variable Classes
  - 3.2 – Operators
  - 3.3 – Instructions
  - 3.4 – Functions
  - 3.5 – System Variables
  - 3.6 – Functions, Subprograms And Scope
  - 3.7 – Reusable Subprograms
  - 3.8 – Performances
- 4. Basic Development: The Object Template
  - 4.1 – The Object Template
  - 4.2 – Field Actions
  - 4.3 – Object Actions
  - 4.4 – STD snippet: Database transaction management
- **5. Debugging Sage X3 Code**
  - **5.1 – The Debugger**

5 – DEBUGGING SAGE X3 CODE

5.1 – The Debugger

The Sage X3 Debugger

- The Sage X3 debugger is accessible from the *Tools* menu or by code:

  Debugger menu item + user action

  Toggle Debug to directly open the debugger

  `Dbgaff` instruction by code.

- User parameter value **DBG** enables/disables debugger access from the *Tools* menu.

The Sage X3 Debugger

- Debugger features

Debugging the code:

- Step-in (*Line* button)
- Step-over(*Continue* button)
- Breakpoints (Breakpoint expression or code line + *Continue* button)

Variable spy:

- Up to 20 variables or expressions to evaluate
- Automatic display of subprogram parameters and of their value

Global spy (***Detail*** button):

- Memory state
- Details of all variable classes including tables, screens and global variables
- Modification of class values by right-click item *Change*.

Trace file option:

- The debugger may be run in the background and print its results in a log file.
- ***Log*** checkbox + log file path (by default *user.dbg* in the TMP application directory)

THANK YOU

End of the Fundamental Training Course