System Platform Design Guidelines

Design guidelines for System Platform implementations
Read this first

This document is intended as a general systems design guideline. Guidelines are presented as a set of Best Practices. As a reference application, a simple Mixing Tank application is supplied with the document – this application is the basis for all examples in the document.

To fully use the application, the following steps must be performed:

- A PC (Virtual PCs or Machines are acceptable) with at least 1024 MB RAM
- Install the following software on it:
  - Application Server 3.0.
  - InTouch 10.0.
  - ActiveFactory 9.2.
- Access to a Historian is needed – it can be installed on the same PC.
- Access to an Alarm Database is needed – it can be installed on the same PC.
- Create a new Galaxy.
- Import the “Design Guidelines Application v1.0.aaPKG”.
- Open the GR_Platform instance:
  - On the General page, change the network address to the name of the PC.
  - On the Engine page, change the Historian address to the name of the Historian PC.
- Open the GR_AppEngine_001 instance
  - On the General page, change the Historian address to the name of the Historian PC.
- Set-up and start the Alarm DB Logger.
- Deploy the Galaxy
- Open the $DemoKitViewApp (InTouch Application)
- On the History and Pareto windows in InTouch, change the SQL Server set-up to the correct information.
- ArchestrA security can be enabled if required.
<table>
<thead>
<tr>
<th>BEST PRACTICE</th>
<th>RECOMMENDATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEVER</td>
<td>INstantiate objects directly from toplevel objects</td>
</tr>
<tr>
<td>ALWAYS</td>
<td>Have at least one base-level, one master-level and one application-level template before instantiating</td>
</tr>
<tr>
<td>CREATE</td>
<td>An index for each application and prefix application level objects with “A#_” where # is the index</td>
</tr>
<tr>
<td>CREATE</td>
<td>Self contained graphics in the graphic toolbox</td>
</tr>
<tr>
<td>WHEN</td>
<td>Embedding graphics create new properties for all requirements of the embedded graphic and pass them through</td>
</tr>
<tr>
<td>MAKE</td>
<td>The embedded graphic’s properties private</td>
</tr>
<tr>
<td>CREATE</td>
<td>A systems area on an engine on the gr and run all systems objects on this area</td>
</tr>
<tr>
<td>MAKE</td>
<td>Contained names descriptive of their role or function in the container</td>
</tr>
<tr>
<td>PREFIX</td>
<td>Script names with “Script” and give them descriptive names</td>
</tr>
<tr>
<td>NUMBER</td>
<td>All toolsets</td>
</tr>
<tr>
<td>ONLY</td>
<td>Instantiate application level objects</td>
</tr>
<tr>
<td>HIDE</td>
<td>Other levels (base and master) from application developers</td>
</tr>
<tr>
<td>IF</td>
<td>A symbol from the orchestra library is used as a standard and changes are to be made to it, make a copy of the symbol, and use the copy. If no changes are necessary - use the symbol directly</td>
</tr>
<tr>
<td>USE</td>
<td>The galaxy load function to create multiple instances</td>
</tr>
<tr>
<td>USE</td>
<td>Only the necessary columns when creating objects with the galaxy load function</td>
</tr>
</tbody>
</table>
# Table of contents

A. **Definition of Terminology** ......................................................... 5

B. **General** ......................................................................................... 6
   1. Design Philosophies ................................................................. 6
      1.1. Templates Philosophy ......................................................... 6
         1.1.1. Base objects ............................................................... 6
         1.1.2. Master objects ............................................................ 7
         1.1.3. Application objects .................................................... 7
      1.2. Graphics Philosophy ............................................................ 8
   2. Design Considerations ............................................................... 9
      2.1. Model ................................................................................. 9
      2.2. Derivation ......................................................................... 11
      2.3. Deployment ....................................................................... 12
   3. Naming conventions ...................................................................... 13
      3.1. Object naming ................................................................. 13
      3.2. Attribute naming ............................................................... 13
      3.3. Script names ..................................................................... 14

C. **Toolbox design** .......................................................................... 15
   4. Derivation .................................................................................. 15
      4.1. Base .................................................................................. 15
      4.2. Master ............................................................................... 15
      4.3. Application ....................................................................... 17
   5. Instantiation ............................................................................... 17
      5.1. General Instantiation ......................................................... 17
      5.2. Systems Area .................................................................... 18

D. **Graphics design** ......................................................................... 19
   6. Standards Graphics ..................................................................... 19
   7. Pop-ups ....................................................................................... 21
   8. Object Specific Graphics ......................................................... 23
   9. Overview graphics ..................................................................... 25

E. **Bulk object creation** ................................................................. 26
   10. Contained Templates ................................................................. 26
A. Definition of Terminology

Definition of terms used in this document

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>A specific implementation e.g. a specific project or site or PLC implementation.</td>
</tr>
<tr>
<td>Area</td>
<td>This refers to a part of a plant e.g. Mixing Plant.</td>
</tr>
<tr>
<td>Enterprise</td>
<td>This refers to the complete company e.g. Wonder Manufacturing. An Enterprise can contain several sites.</td>
</tr>
<tr>
<td>Site</td>
<td>This refers to the physical location of a plant e.g. Johannesburg Plant. A Site can contain several Areas.</td>
</tr>
</tbody>
</table>
B. General

General design guidelines

This document is designed for System Integrators and End Users who wish to implement System Platform. It assumes the following versions: Application Server 3.0 and InTouch 10.0.

The two major parts of a System Platform implementation is the functional and visual parts. Functionality is implemented via objects in the galaxy. Graphics can be standalone or implemented as part of these objects. These implementations are examined separately.

1. Design Philosophies

1.1. Templates Philosophy

Objects in the galaxy are instantiated from templates. When a galaxy is first created, only a handful of templates are available (e.g. UserDefined, Area, WinPlatform etc.). These objects are the highest level objects in a galaxy and cannot be modified. They are referred to as Top-level Objects.

Top level objects are divided into three types:

- **Application** – For designing the application
- **System** – For system type objects
- **Device Integration** – For PLC/field communication

Note that there are several top level objects in the Application toolset. The most important one is the UserDefined. For Field Devices the use of UserDefined objects is recommended above templates like Float etc.

**BEST PRACTICE:** NEVER INSTANTIATE OBJECTS DIRECTLY FROM TOPELEVEL OBJECTS

**BEST PRACTICE:** ALWAYS HAVE AT LEAST A BASE-LEVEL, A MASTER-LEVEL AND ONE APPLICATION-LEVEL TEMPLATE BEFORE INSTANTIATING

It is recommended that three levels of derivation be created…

1.1.1. Base objects

Base objects are derived from the Top-level objects. Since Top-level objects cannot be changed, this level provides a template that can be customised. Base objects are prefixed wit a “b_” – e.g. $b_UserDefined. Object functions that are global to the galaxy are created at this level.
1.1.2. Master objects

Master objects are the standards of the enterprise. They are prefixed with an "m_". Master objects are derived from base objects (not top-level objects). Master objects can be S95 Standards (e.g. $m_ProcessCell), S88 standards (e.g. $m_ControlModule) or Enterprise specific standards (i.e. an object representing a physical asset that is used in all parts of the enterprise and should be standard e.g. $m_LIMS). Master objects can also be lower derivatives of the abovementioned objects, but should contain no Application specific functionality (e.g. $m_Valve).

Generally IO assignment can be viewed as Application specific (since various sites/applications might have different PLCs). IO assignment can be present in Master templates, however, if the method of this assignment should be standardised.

Note that if a complex object is required to be standard across all Applications, it should be built in the Master level. This means that Master level templates can already be containers with contained objects.

1.1.3. Application objects

Application type objects are specific to an application – this might be functionally specific or PLC specific. Application objects are prefixed with “a#_” and are always derived from master objects. The # in the prefix should identify the Application. This can be a project code. Application level objects will be used to instantiate objects. This means that Application level objects can be contained objects.

---

1 Refer to Definition of Terminology
**BEST PRACTICE:** Create an index for each application and prefix application level objects with “A#_” where # is the index.

If application specific functionality (such as IO assignment) exists for a master object another a-level derivation is needed. In other words: If all valves in this application have something in common (such as PLC addressing) but differ from valves in other applications in the enterprise then an $a#_Valve$ is required. In this case the derivation will look as follows:

In the following example System Integrator A (SIA) designed a MixingTank with an Agitator. They designed their own Agitator with their custom code ($a0_Agitator$). Later System Integrator B (SIB) designed a Reactor for the company – it also requires an Agitator, but this one has some differences to that of SIA’s. Even though both SI’s standardised on m_Agitator, they have made different implementations of it.

1.2. Graphics Philosophy

Before moving to Graphic design, it is essential to understand that, while objects can be derived and changed, graphics cannot be derived. A graphic that already exist can only be embedded into other graphics, but changes cannot be affected to the graphic – the only changes that can be made to an embedded graphic is its public properties.

To ensure good reusability, it is recommended that most graphics be designed outside objects – that is: in the graphic toolbox. Design these graphics with public properties as required and remember to make internal properties private. In other words: Make these graphics self-contained.
2. Design Considerations

2.1. Model

It is desirable to have a standard for modelling and the S95 standard is recommended for the top levels.
The modelling part of the design aims to mimic the physical layout of the plant e.g. a mixing tank contains an agitator, an inlet valve, an outlet valve, a level transmitter, in- and outlet flow transmitters, a PID controller and a pump. A pump contains a flow transmitter and a pressure transmitter. Graphically this could be shown as on the next page.

In Application server this containment would be represented as:

```
MixingTank_001
  └ Agitator_001 [ Agitator ]
  └ InletValve_001 [ InletValve ]
  └ Level_001 [ Level ]
  └ OutletFlow_001 [ OutletFlow ]
  └ OutletValve_001 [ OutletValve ]
  └ PID_Controller_001 [ PID_Controller ]
  └ Pump_001 [ Pump_001 ]
       └ Flow_001 [ Flow ]
       └ Pressure_001 [ Pressure ]
```

The idea here is to contain the “sub-objects” inside the main object. This has the following advantages:

- Understandable format
- Containment:
  - If anyone needs to read the level of a specific Mixing Tank (say MixingTank_005), it can be referenced with `MixingTank_005.Level.PV`. He/she does not need to know that that level transmitter is called `Level_001`. 

10
A specific Mixing tank can reference the objects below it using me. For instance, a script inside the Mixing tank object that needs to check the level can reference it as me.Level.P. This means that no separate scripts are required for each Mixing tank – in other words: if the same script runs on all Mixing tanks, each one will get its own level.

A "sub-object" can reference its container with mycontainer. For instance, if the pump has a script that checks if the Mixing tank is in the starting condition, it can reference mycontainer.Status.Starting. This means that the specific pump need not know that it is part of Mixing tank 004.

Containment and well-designed models make implementation and maintenance very easy – if a model is well designed and contained; implementation consists of dragging the template to the model space, applying the correct Naming conventions, right-clicking and deploying it. Containment should be done on the template level where possible. A template for a Mixing tank should already contain templates for the Pump, valves etc. When a new Mixing tank is added to the model, the contained objects will already be present.

Note that the contained names are very important for containment to work efficiently.

2.2. Derivation

When working with derivation, the main thing to keep in mind is similar functionality. In our example above we have two completely different motors (an Agitator and a Pump) – but they do share some functionality. For instance all motors will probably have a start and stop command as well as a status indication (whether it’s running or not). These functions can be built into a type $m_Motor. Whenever a motor of any type is needed one can derive one from this template and it will already contain the required functionality (including scripting).

The argument can be further extended by realising that the two motors are of different types: One is a pump and the other an agitator. The agitator has a speed set-point for instance. The $m_Motor template can then be used to derive two new templates: $m_Agitator and $m_Pump. These can be used directly in the model or one can derive Mixing tank specific versions ($a0_MixingTank.Agitator for instance). If one follows the principal of building containment in a template, it is actually better to add these Mixing tank specific versions to the Mixing tank template.
To examine derivation, use the derivation view. Notice that in the above example, the Mixing tank specific version of $m_Agitator is called $a0_MixingTank.Agitator. The $a0_MixingTank part actually refers to the $a0_MixingTank template. The Agitator is already part of the template.

2.3. Deployment

Even though the ArchestrA model is designed independently from the infrastructure it is important to give some consideration to the deployment of objects. Deployment in ArchestrA requires a PC/Server. This is an object derived from $WinPlatform. To execute objects an engine is required. Two types of engines exist:

- $AppEngine (application engine) is used to execute objects.
- $ViewEngine is used to execute the InTouch application

Engines must be deployed on platforms. Application engines can directly run Device Integration objects. They can also execute normal objects, but these must be on an Area. This is important since it means that if an object is on the same area – they must be executed by the same application engine and therefore on the same server.

In a client-server set-up, the server machines will typically run a $WinPlatform and at least one $AppEngine. The client machines will have a $WinPlatform and a $ViewEngine. In a stand-alone environment their will be only workstation machines running a $WinPlatform, at least one $AppEngine and a $ViewEngine.

It is also recommended that at least a top level Systems area be deployed to the GR. This area should contain all the systems objects in the Model View (Platforms and Engines). This requires the GR to have at least one Application Engine.

**BEST PRACTICE:** CREATE A SYSTEMS AREA ON AN ENGINE ON THE GR AND RUN ALL SYSTEMS OBJECTS ON THIS AREA
It is also recommended to create derivations of the $b_WinPlatform for at least the following platforms:

- Galaxy Repository (GR)
- Application Server
- View Client

3. Naming conventions

3.1. Object naming

All objects have at least one name called the Tag name. Contained objects also have a second name called the Contained name. When referencing any object or tag in an ArchestrA galaxy the first object of the reference should always be a Tag name except if the reserved relative referencing words are used (me, mycontainer, myarea, myEngine etc). Only one Tag name can be used in a reference. An example would be: 

```
ATagname.ContainsName1.ContainsName2.ContainsName3.Attribute
```

The tag name is the objects identity – no other object may have the same tag name. A tag name uniquely identifies an object in the galaxy. A solid naming convention is highly recommended. A good example of this would be the ANSI ISA S5.1/1984 (R1992) standard.

A contained name uniquely identifies an object within its container – other objects in the galaxy may have the same contained name but not if they are in the same container. It is desirable to make the contained name descriptive of its role or function in the container e.g. Fan_Motor, Jacking_Oil_Pump, Inlet_Valve etc.

**BEST PRACTICE:**

MAKE CONTAINED NAMES DESCRIPTIVE OF THEIR ROLE OR FUNCTION IN THE CONTAINER.

3.2. Attribute naming

Attributes (whether Field Attributes or User Defined Attributes) are the public interfaces of an object. A strong naming convention is therefore suggested.

Such a convention is described here. Some changes might be required depending on the application/implementation. Name parts in square brackets ([ ]) are optional. Name parts in italics represent variable descriptive names.
<table>
<thead>
<tr>
<th>Description</th>
<th>Naming convention</th>
<th>Security/Category</th>
</tr>
</thead>
</table>
| Set-up parameter – set during design time (Typically Startup scripts use them) | Setup.[Attribute].Parameter  
* Setup.[XML].Folder  
(Folder is created the first time the object deploys) | Read Only |
| Configurable parameter – Object must be set off scan to change (typically OnScan scripts use them) | Cfg.[Attribute].Parameter  
* Cfg.[ORDERDATABASE].SERVERNAME  
(Connect is established every time the Engine goes on Scan) | Configure |
| Engineering Set Point – Only certain personnel can change these | Eng.[Attribute].SetPoint  
* Eng.[FEEDER].SPEED.MAXIMUM  
(Only Tun permitted users can change the maximum speed) | Tune |
| Calculate value – Cannot be changed by anything but a script | Calc.[Attribute].Parameter  
* Calc.[CABLE].LENGTH  
(Length is calculated from measured weight) | Calculated |
| Set point – Operational set points that can be changed by operators | SP.[Attribute].SetPoint  
* SP.[FEEDER].SPEED  
(Operating set point speed of feeder) | Operate |
| Commands – When set to TRUE, a function is performed. On completion the value returns to FALSE | Cmd.[Attribute].Command  
* Cmd.[FEED].START  
(Start the Feed sequence) | Depends on command |
| Mode Select – Indicates a specific mode. Stays in state until mode is changed. | Select.[Attribute].Mode  
* SELECT.[MODE].AUTO  
(Selects the Auto mode) | Depends on mode |
| Process value – Typically an Analogue value that is measured with and instrument. | [Attribute].PV  
* [LEVEL].PV  
(Process value of the Level) | Object Writable |
| Status value – Indicates status of something in the field (typically not an Analogue value) | Status.[Attribute].Status  
* STATUS.[INLETVALVE].OPEN  
(Indicates when the valve is open) | Object Writable |
| Alarms flag – Indicates that something is in alarm | [Attribute/Status/PV].AlarmType.Alarm  
* [TANK].LEVEL.HIGH.ALARM  
(Indicates when the valve is open) | Object Writable |
| Interlock flag – Indicates that something is interlocked | [Attribute/Status/PV].InterlockType.Interlock  
* [PUMP].VALVECLOSED.INTERLOCK  
(Indicates Pump is interlocked because valve is closed) | Object Writable |
| Simulation Attributes – Used to simulate something. An object should still be functional if all Simulation Attributes and Scripts are removed | Sim.RestofNameparts  
* Sim.ENG.SIMULATEDPV.MAXIMUM  
(Change the maximum simulated PV value) | Depends on simulation |

### 3.3. Script names

Prefix all scripts with an indication (such as “Script.” or “Scr.”). The rest of the name should be descriptive and the dot separator can be used to order the scripts. If the script calculates the result of an attribute, use the attribute name e.g. **Script.Status.State.Calculation**.

**BEST PRACTICE:** **PREFIX SCRIPT NAMES WITH “SCRIPT” AND GIVE THEM DESCRIPTIVE NAMES.**
C. Toolbox design

Toolbox design guidelines

4. Derivation

Templates should be organised and easy to browse. It is recommended that three types of toolboxes be created:

- Base objects
- Master objects
- Application objects

These toolboxes correspond to the three recommended levels of derivation (see Derivation (2.2)).

4.1. Base

Normally the Base objects toolkit will consist of a small number of objects that will form the base of the system. These can be organised in subsets but this can slow down the developer more than it helps.

4.2. Master

At the master level it becomes more useful to group objects into various subsets. Recommended subsets are:

- System templates – for all system objects
- S95 templates – for ISA 95 structuring
- Control Modules – for all master level control modules
- Transmitters – for all master level transmitters
Additional toolsets can also be created. Numbering and naming toolsets with descriptive names makes it easier to find templates when they are needed. Application Server 3.0 allows nesting of toolsets and this allows for even further hierarchies if required.

**BEST PRACTICE:** NUMBER ALL TOOLSETS
4.3. **Application**

As described earlier, Application level objects should be specific to the implemented application/project. Typically there will be a toolset for different types of units (for instance the Mixing tank). Keeping the numbering of toolsets everywhere the same makes navigating to templates easier.

5. **Instantiation**

5.1. **General Instantiation**

For consistency sake it is recommended that only application level objects be instantiated. This means that if an object from the master level is required directly, one should rather first derive it once to the a-level before instantiating. This might sometimes seem excessive but is the only way to ensure re-usability while still retaining enough flexibility for future expansion.

**BEST PRACTICE:** **ONLY INstantiate APPLIcATION level OBJECTS**
5.2. Systems Area

As mentioned earlier it is recommended that a Systems Area be created to hold system type objects in the model view. In other words – no deployed objects are kept in the Unassigned area. The systems area can be deployed on an Engine on the GR node.


D. Graphics design

Graphics design guidelines

A good practice is to create five levels of graphics namely:

- Standards.
- Pop-ups.
- Object specific Graphics.
- Overview Graphics

6. Standards Graphics

Standards graphics determine the styling of buttons, panels etc. Create these as stand-alone so that they can be embedded into any other graphic to perform a function. To change the complete application’s look-and-feel, one can then only change the standards.

A good example of this is for instance the standardisation of buttons in the application:

In the example button the standard button is drawn and given three public properties:

- GreyedOut [Boolean] – User can set this value to true to grey out the button.
- **MouseOver [Boolean]** – User can script this value to be true when the mouse is over the button.
- **Pressed [Boolean]** – User can script this value to be true when the button is pressed.

A single internal property exist: Highlighted. The highlighted Boolean is set to true internally if the mouse is over the button (MouseOver = TRUE) and the button is not greyed out (GreyedOut = FALSE).

Symbols for physical assets can also be Standards – e.g. a picture of a pump with all its contained animation. Note that Standards should not be object aware.

---

**BEST PRACTICE:** IF A SYMBOL FROM THE ARCHETRA LIBRARY IS USED AS A STANDARD AND CHANGES ARE TO BE MADE TO IT, MAKE A COPY OF THE SYMBOL, AND USE THE COPY. IF NO CHANGES ARE NECESSARY – USE THE SYMBOL DIRECTLY

Other Standards that can be built are:

- Panels
- Textboxes
- Headings
- Lights
- Switches
- Etc

Standards can also be used to generate other standards to provide a graphic function that can be re-used by different applications, objects or graphics. A good example is an
Auto/Manual switch – such a switch might be using standard buttons and text, but the complete primitive exposes only two properties (Auto and Manual). A statistics panel is shown as an example. The panel is reusable as the values need only be supplied to have them displayed.

One notable exception for building Standards outside objects is when one wants to use the Name Animation. The Name animation needs a connection to an object to work correctly. The name animation can be done in the $b_Userdefined.

Standards can also be object standard graphics (symbols) such as a pump, valve or motor. Note the use of properties: The property BladeCount is set to 18 (a constant) and made private – this becomes part of the standard i.e. **All graphics using this pump standard will use 18 blades.** The property Value is left Private and the user must tie the property to the correct tag – a default of me.Status.Running is supplied.

7. Pop-ups

Combine standards to create generic pop-ups. The Transmitter popup is shown.
Note that this popup has its own set of properties that should be linked to the properties in the standard and set to private.
8. **Object Specific Graphics**

Objects representing field devices normally contain only two types of graphics: The first should be the symbol representing the object (e.g., a picture of a pump); the second should be an assembled pop-up (Faceplate) window. The symbol graphic will contain the object symbol graphic configured for the object.

These graphics should be a customised group of embedded graphics (standards and pop-ups). All other (re-usable) graphics should be designed as separate graphics.
Object specific graphics can also be graphics representing the physical plant. In its simplest form, something like the previously described pump graphic can be directly embedded. This type of graphic can also be a collection of graphics. For instance a Pump set contains a pump, a flow transmitter and a pressure transmitter. The Pump set graphic can then be a complete collection of primitives already linked.
9. Overview graphics

These graphics represents the screen shown to the user. There are two options here:

- Build the screen inside traditional InTouch using object specific graphics.
- Build screens in Area objects that represent the screen. These overview screens can utilise object specific graphics as well as Standards.
E. Bulk object creation

Creating multiple instances of an object

10. Contained Templates

As mentioned earlier, a template can already contain other templates (e.g. a Mixing tank contains a Pump). If there should be multiple instances in the container (e.g. multiple pumps in a Mixing tank unit) it is recommended to only have one in the template. Multiple instances should be added after the first instantiation. The process is described here.

Instantiating the template will provide the first instance. Subsequent instances can be created with the Galaxy load option.

**BEST PRACTICE:** USE THE GALAXY LOAD FUNCTION TO CREATE MULTIPLE INSTANCES

The relevant instances can be dumped to a CSV file with the Galaxy Dump function. Remove all unnecessary columns before creating new rows for new instances. The CSV file is a good place to implement the correct Naming conventions.
BEST PRACTICE: USE ONLY THE NECESSARY COLUMNS WHEN CREATING OBJECTS WITH THE GALAXY LOAD FUNCTION.