Administrator's and Developer's Guide

NetPhantom Gui-on-the-Fly

Version 7.7

Document Revision 1

5 June 2024

Mindus SARL

NetPhantom®

Version 7.7

Mindus SARL

1 Rue du Gabian
MC-98000 Monaco
MONACO

Telephone: **+377 99 90 32 66**
Web: **https://netphantom.com**
E-mail: **info@netphantom.com**

Support

Phone: **+377 99 90 32 66**
E-mail: **support@netphantom.com**

Contents

Preface – Intended Audience

If I wanted to become a tramp, I would seek information and advice
from the most successful tramp I could find.
If I wanted to become a failure, I would seek advice
from people who have never succeeded.
If I wanted to succeed in all things, I would look around me for those
who are succeeding, and do as they have done.

Joseph Marshall Wade

The *Administrator's and Developer's Guide to NetPhantom Gui-on-the-Fly* is not intended as end user documentation but rather as a guide and reference for NetPhantom application developers, system administrators and programmers.

This documentation illuminates the principles behind the Gui-on-the-Fly module. It also describes how to set up your own rules for processing, how to change the look of the controls by using a template and how to write your own identification classes.

Part of this document is quite technical and is mainly intended for developers planning to write their own identification classes. However, a deep understanding of the inner workings of the Gui-on-the-fly module makes it easier to understand any problem that might occur when using the default identification and makes it easier to configure the default identification using all the settings.

1 Gui-on-the-Fly

Nothing defines humans better
than their willingness to do irrational things
in the pursuit of phenomenally unlikely payoffs.

Scott Adams

Gui-on-the-Fly is a module in NetPhantom that can be used to automatically create graphical panels during runtime, without the need to first create graphical panels in the NetPhantom Editor.

1.1 General Description

The Gui-on-the-Fly module uses rules to identify the different controls from the host screen. This means that a host application should have host screens that have been designed with a well-defined look that should be repeated through all the screens. Otherwise, there is a risk that some functionality might be lost when the module tries to identify a complicated control, such as a list box. One way to avoid this problem with applications that do not have a well-defined look throughout all screens is to only identify simple controls such as entry fields or output texts.

No information will be saved when the panels are created, so this module cannot be used as a base for creating panels in the Editor.

Gui-on-the-Fly is designed with a lot of flexibility using settings that can be changed to suit the application in question. It is also designed with the possibility of custom-written identification classes in mind. All the identification classes are loaded dynamically from settings.

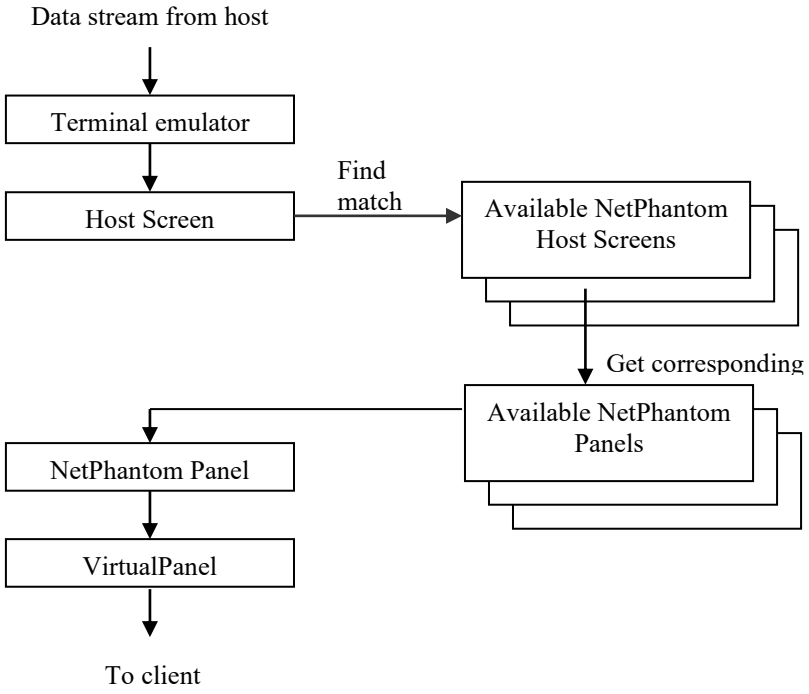
1.2 What Makes Gui-on-the-Fly Different?

To understand how to configure the default Gui-on-the-Fly settings or how to write your own Gui-on-the-Fly classes, it is helpful to understand how the Gui-on-the-Fly processing differs from normal processing.

Normal Mode of Operation

In the normal mode of operation, the server’s identification of host screens works like this. First the terminal emulator receives the data stream with terminal data. From this data, a Host Screen that stores data about all the host fields is created. The Host Screen's data is then compared to the NetPhantom Host Screen's belonging to the NetPhantom application that the user is running. The NetPhantom Host Screens are the host screens that have been created in the NetPhantom Editor when the NetPhantom application was created. When a match between a Host Screen and a NetPhantom Host Screen is found, the corresponding NetPhantom Panel, with information about all the controls, is fetched. From this data the server builds the VirtualPanel and its VirtualControls.

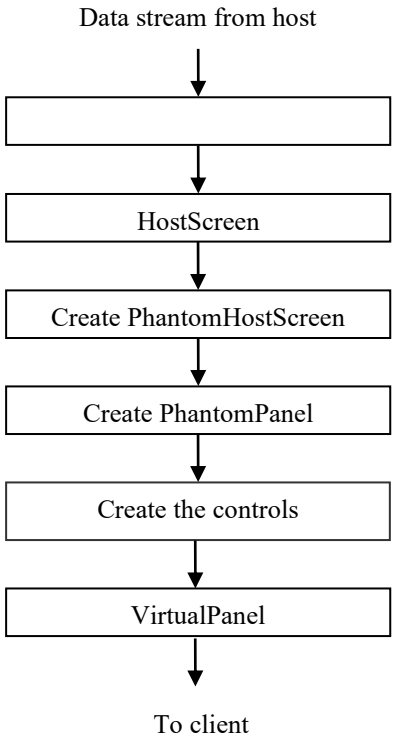
The above description is, of course, a little simplified and does not go into detail. The schematic diagram below illustrates the normal mode of operation.



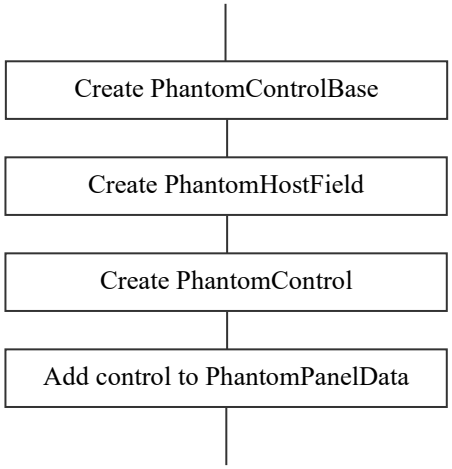
Gui-on-the-Fly Mode of Operation

In Gui-on-the-Fly mode of operation, the server's identification of host screens work like this. First the terminal emulator receives the datastream with terminal data. From this data, a HostScreen, that stores data about all the host fields, is created. The Gui-on-the-Fly module then creates the PhantomHostScreen, the data for the PhantomPanel and the data for all the controls on the panel. This means that the Gui-on-the-Fly module must create all the information in real-time that would normally be fetched from the application's runtime files. The server can then create the VirtualPanel and the VirtualControls in a normal way.

The steps performed by the Gui-on-the-Fly module are graphically illustrated in the following diagram.



The last step in the above diagram, called *Create the controls*, can in turn be further split into the several steps. These steps will be repeated until all the controls have been identified.



1.3 Gui-on-the-Fly Architecture

The Gui-on-the-Fly module consists of several distinctive building blocks. Some of the blocks are built into the server, while others are loaded dynamically during server startup.

The heart of the Gui-on-the-Fly module is the GofManager. The manager is responsible for managing all the Gui-on-the-Fly processing.

During startup of the NetPhantom Server, the GofManager loads all the Gui-on-the-Fly runtime files that are to be used as templates for the Gui-on-the-Fly processing. They contain all the required layout settings. Different NetPhantom runtime applications can use

different Gui-on-the-Fly runtimes as templates. For more information about Gui-on-the-Fly templates, see Chapter 4.

The GofManager also loads the basic Gui-on-the-Fly settings from the configuration file, named *gof.ini*. These are settings such as which class to use for the different identification tasks. The settings are then passed on to the correct GuiOnTheFlyRuntime instance, which is responsible for loading all the different classes it needs for the identification.

GuiOnTheFlyRuntime is the module that handles the Gui-on-the-Fly processing for a specific instance of a NetPhantom application. This means that every time a user starts an application that has Gui-on-the-Fly processing activated, a new instance of the GuiOnTheFlyRuntime will be started. The GuiOnTheFlyRuntime will normally load one GofHostFieldIdentifier class, a few GofHostAreaIdentifier classes and several GofControlIdentifier classes.

When a client runs using Gui-on-the-Fly, the GuiOnTheFlyRuntime gets the references to the different identifier classes for the application from the GofManager. It then processes the host screen and creates the PhantomPanelData with all the identified PhantomControls.

The various identifier classes identify different things on the host screen. The GofHostFieldIdentifier class identifies logical host fields from the physical host fields, the GofHostAreaIdentifier classes identify different areas on the host screen and the GofControlIdentifier classes identify the different controls in the different areas.

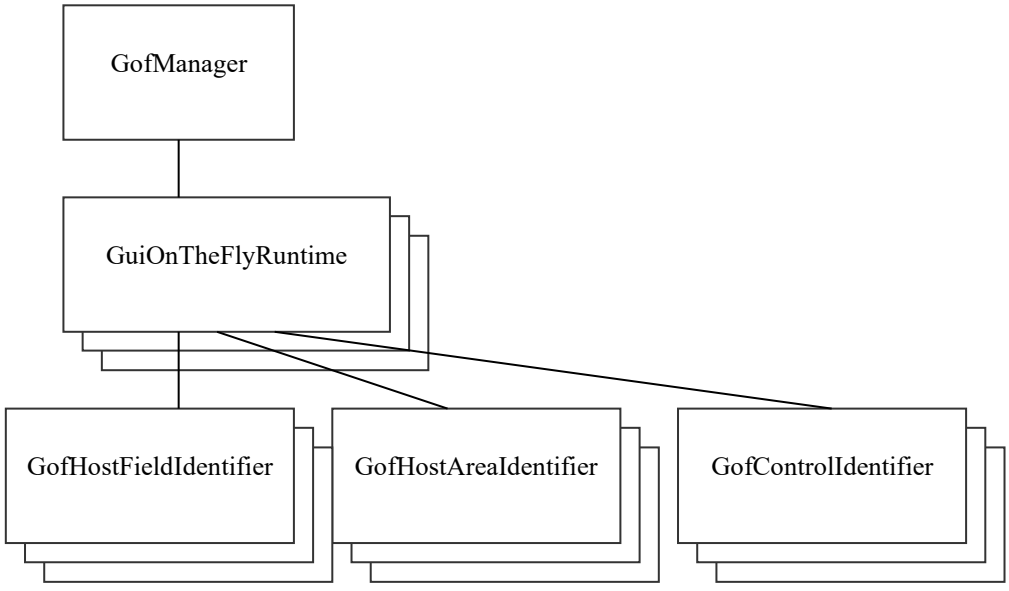


Fig. The basic building blocks of the Gui-on-the-Fly module.

1.4 GofManager

The GofManager is started when the NetPhantom Server starts up. The GofManager will try to set up as much as possible during startup, so that as little time as possible will be spent loading classes and setting up configurations when a client loads an application that uses Gui-on-the-Fly.

The GofManager will first look for a section in the configuration file named GuiOnTheFly. This section contains a list of applications that will use Gui-on-the-Fly and what setting they will use. Below is an example of how this section might look.

```
[GuiOnTheFly]
GOFTEST=NpGofTest
APP2=App2Gof
APP5=App5Gof
```

The item name (to the left of the equal sign) is the runtime name used in the Runtime Application Section of *server.ini*. The value (to the right of the equal sign) is the name of a specific Gui-on-the-Fly section.

The GofManager will then go through each of the Gui-on-the-Fly sections and read the basic settings from them.

The basic settings are:

- The runtime filename
- The class to use for host field identification
- The host screen area to identify
- Identification rule for partial fields
- The classes to use for identifying the different areas on the host screen
- The order in which the different areas should be identified
- The classes to use for identifying the different controls
- The identification order

These settings will be described in detail below.

The GofManager has a method that returns an instance of the GuiOnTheFlyRuntime that corresponds to a specific NetPhantom runtime application, based upon the NetPhantom application name. This is how the VirtualPanelSession gets a reference to the right GuiOnTheFlyRuntime instance.

The Runtime Filename

This setting is just the name for the NetPhantom compiled distribution (`JAR`) to be used as the Gui-on-the-Fly template. For more details about template files see Chapter 4. The name includes the path relative to the server's runtime directory.

```
gofruntime=gofgui/npgoftest/npgoftest.jar
```

Note that the extension is `jar`. The template compiled distribution file is a normal NetPhantom application, but just contains one or two panels created in the NetPhantom Editor, from which information about control layout will be fetched when needed. The template runtime application should be compiled using the menu item **File – Compile distribution**.

Class for Host Field Identification

This setting specifies the class used for host field identification. This class creates logical host fields (called GofHostFields) used by the Gui-on-the-Fly classes. This must be done in order to be able to split up a single host field into several logical fields, when necessary.

```
hostfieldidentifier=se.entra.NetPhantom.server.GofAS400HostFieldIdentifier
```

Host Screen Area Identification

This setting can be used when there is a need to exclude some part of the host screen from Gui-on-the-Fly processing. Note that this setting will be used on *all* the screens with the exception of popup windows.

The values for this setting are the start line, the start column, the end column and the end line of the area to be identified. The values can either be separated by commas, or by blanks.

The end column and the end line can have value 0 (zero), which means that the whole screen will be identified, regardless of the screen size.

The end column and the end line can also have a negative value. This is calculated as an offset from the right/bottom of the screen. For example, specifying a last column of -2, when the screen width is 80 columns, means that the last column to be included is column 77 (79 – 2 = 77, the first column is column 0).

Example of setting when the whole screen should be identified, with commas as separators:

```
hostscreenarea=0,0,0,0
```

Example of setting when fixed margins, regardless of screen size, should be removed. Here blanks have been used as separators:

```
hostscreenarea=1 1 -1 -1
```

Example of setting when a fixed area of screen, regardless of screen size, should be identified:

```
hostscreenarea=1,1,78,22
```

Identification Rule for Partial Fields

This setting determines what should happen with host fields not completely inside the HostScreenArea when the HostScreenArea specifies an area smaller than the whole screen.

```
identifypartialfields=0
```

Accepted values are 0 or 1. 0 means that host fields not completely inside the area should not be identified and 1 means that they should be identified.

The Classes for Identifying the Different Areas on the Host Screen

This setting consists of two parts, first a list defining the names of the areas to be identified and the second a list of the classes to be used for identifying the areas. The names can be any name, as long as they are unique.

```
areaidentifiers=TITLEAREA BUTTONAREA MAINAREA

TITLEAREA=se.entra.NetPhantom.server.GofAS400TitleAreaIdentifier
BUTTONAREA=se.entra.NetPhantom.server.GofAS400ButtonAreaIdentifier
MAINAREA=se.entra.NetPhantom.server.GofMainAreaIdentifier
```

The first list, as all other single line lists, must either have its items separated by blanks or by commas. Both alternatives below are valid.

The area identifier list separated by blanks:

```
areaidentifiers=TITLEAREA BUTTONAREA MAINAREA
```

The area identifier list separated by comma:

```
areaidentifiers=TITLEAREA,BUTTONAREA,MAINAREA
```

Order of Identification

This setting specifies in which order the different areas should be identified. The MAINAREA should always be the last area to be identified if the default classes are used, because any host field not included in any of the other areas will always be included in the MAINAREA area.

It is also possible to have overlapping areas, but in this case you should use user-written classes for area identification with rules for determining which host fields should be included in the specific area. If overlapping areas are used with the default classes, the first area identifier will grab all the host fields in the area.

```
areaorder=TITLEAREA BUTTONAREA MAINAREA
```

Control Identification Classes

This setting consists of two parts, first a list defining the names of the controls to be identified followed by a list of the classes to be used for identifying the controls. The names can be any name, as long as they are unique.

```
ctrlidentifiers=OTCTRL EFCTRL PBCTRL LISTCTRL LISTCONTMARKCTRL

OTCTRL=se.entra.NetPhantom.server.GofOutputTextIdentifier
EFCTRL=se.entra.NetPhantom.server.GofEntryFieldIdentifier
PBCTRL=se.entra.NetPhantom.server.GofPushButtonIdentifier
LISTCTRL=se.entra.NetPhantom.server.GofListBoxIdentifier
LISTCONTMARKCTRL=se.entra.NetPhantom.server.GofListContMarkIdentifier
```

Control Identification Order

This setting consists of two parts, a list defining the names of the control identification orders and a list of the different identification orders. Each identification order is made up of the area and a list of which controls to identify in that area. The order in which the controls are specified determines in which order the controls will be identified in that area. The identification order names can be any name, as long as they are unique.

```
identificationorder=TITLEIDENT BUTTONIDENT MAINIDENT

TITLEIDENT=TITLEAREA:TITLECTRL
BUTTONIDENT=BUTTONAREA:LISTCONTMARKCTRL PBCTRL
MAINIDENT=MAINAREA:LISTCONTMARKCTRL LISTCTRL OTCTRL EFCTRL
```

For example, in the setting above for the MAINIDENT identification order, the first name in the value (the name between the equal sign and the colon) specifies that this identification order be for the MAINAREA. The controls that should be identified in this area are: LISTCONTMARKCTRL, LISTCTRL, OTCTRL and EFCTRL, in the specified order.

When GuiOnTheFlyRuntime processes an identification order, it will first find the area the order is for. It will then loop through the list of controls in the identification order and try to find all the controls of the correct type in that area.

Other Settings

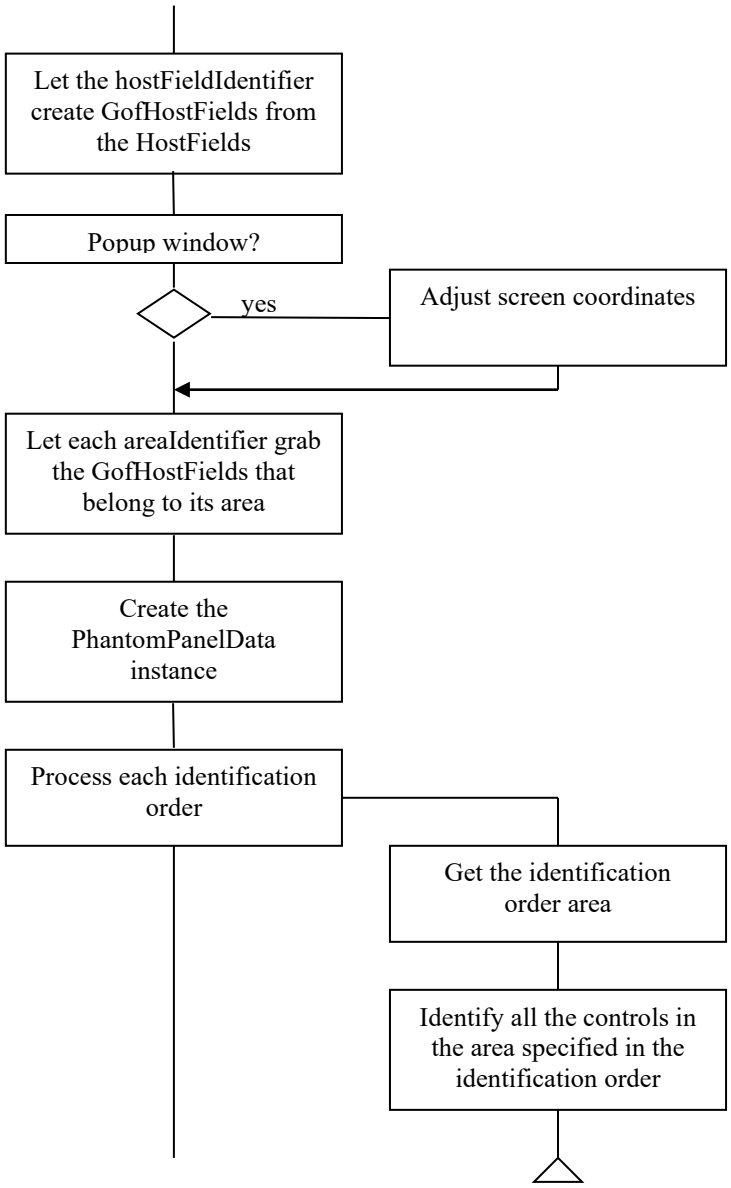
In addition to these basic settings, each class used for identification can have its own settings. The writer of the class must specify the names of the settings. It is also the responsibility of the class to read its own settings. For more details about how to write your own identification classes, see Chapter 5.

1.5 GuiOnTheFlyRuntime

The GuiOnTheFlyRuntime instance is responsible for saving all the basic settings. It is also responsible for loading and instantiating all the classes that will be used for identification.

The GofManager creates an instance of GuiOnTheFlyRuntime for each Gui-on-the-Fly section defined in the configuration file. These instances will be created during the startup of the NetPhantom Server.

When a client is running an application that uses Gui-on-the-Fly, this class uses the rules from the basic settings to identify host fields, then screen areas, and finally the controls in the different areas. It then creates a new panel with all the identified controls on it. This process is best described with the following diagram.



1.6 GofHostFieldIdentifier

The GofHostFields class that is used to create logical host fields from a screen's host fields must implement this interface. An adapter class, GofHostFieldIdentifierAdapter, which implements the GofHostFieldIdentifier interface, is available. If the need to write your own GofHostFieldIdentifier classes arises, it is recommended to extend from the adapter class, because it has default implementations for every method defined in the interface.

The GofHostFieldIdentifier makes it possible to split original host fields into two or more logical host fields, which might simplify the identification of controls. This might be done when a host field wraps from one line to another line, or when several non-editable list cells are included in a single host field, as is normal on mainframes.

Only one GofHostFieldIdentifier class can be used for each Gui-on-the-Fly setting.

The class implementing this interface must have a default constructor without any parameters, since it will be created dynamically from the class name.

The methods that the implementing class must implement are:

```
public void identifyHostFields( HostScreen screen );
public void doIdentifyPartialFields( boolean flag );
public Vector getHostFields( int x, int y, int w, int h );
public Vector getPopupWindowHostFields( int x, int y, int w, int h );
```

Below follows a short description of what should be implemented in these methods and what the adapter class has implemented.

The identifyHostFields Method

This method must create the GofHostFields from the HostScreen's HostFields. It is up this method to decide if a HostField should be split up to several GofHostFields. The newly created GofHostFields must be stored internally in the class, since the method GetHostFields must be able to return them.

This method must take a `HostScreen` as parameter.

The adapter class only implements an empty method. Classes extending this class must extend this method.

The doIdentifyPartialFields Method

This method must store a flag indicating what should be done with GofHostFields that are partially inside, partially outside the identification area. This flag is only meaningful when a part of the whole screen will be identified.

This method must take a `boolean` value as parameter.

The adapter class implements the above functionality.

The isPartialFieldsIdentified Method

This method queries if partial fields are identified when GofHostFields that are partially inside, partially outside the area should be identified. The returned value is only meaningful when a part of the whole screen is identified.

Return `true` if partial fields are identified, `false` otherwise.

The adapter class implements the above functionality.

The getHostFields Method

This method retrieves the GofHostFields created in the identifyHostFields method. It should take into account the value of the flag that was set in the doIdentifyPartialFields method.

This method takes four integer values as parameters. They are the start column, the start line, the width, and the height of the area to identify.

It should return a `Vector` containing all the GofHostFields.

The adapter class implements the above functionality.

The getPopupWindowHostFields Method

This method retrieves the GofHostFields created in the identifyHostFields method, which are inside a popup window. It should ignore the value of the flag that was set in the doIdentifyPartialFields method, since this should only be valid for complete screens.

This method takes four integer values as parameters. They are the start column, the start line, the width and the height of the area belonging to the popup window.

It should return a `Vector` containing all the GofHostFields in the popup window.

The adapter class implements the above functionality.

1.7 GofHostAreaIdentifier

The GofHostAreaIdentifier is an interface that must be implemented by all the classes that will be used to identify different areas on the host screen. An adapter class, GofHostAreaIdentifierAdapter, that implements the GofHostAreaIdentifier interface is available. If the need to write your own GofHostAreaIdentifier classes arises, it is recommended that you extend from the adapter class, because it has default implementations for every method defined in the interface.

By splitting the host screen into different areas, it will be possible to use different rules for control identification in these areas, and in some areas only identify certain controls.

Examples of areas that might be identified are:

- Title area
- Function key area
- Message area
- Status area
- Main area (the rest of the screen)

Each of the classes implementing this interface can load their own settings from the *gof.ini* configuration file.

The GofHostAreaIdentifier classes will also get a reference to all the controls created in their area, and will be able to implement their own layout rules.

The class implementing this interface must have a default constructor, without any parameters, since it will be created dynamically from the class name.

The methods that the implementation class must implement are:

```
public void getAreaSettings( IniFile serverIni, String subsection );
public Vector identifyArea( Vector gofHostFields, int x, int y, int w, int h );
public Vector getAreasGofHostFields( );
public void addControl( PhantomControl phantomControl );
public void layout( int type );
```

Below follows a short description of what should be implemented in these methods, and what the adapter class has implemented.

Instance Variables Defined in the Adapter Class

The adapter class defines a protected Vector called areaGofHostFields for the locale storage of GofHostFields belonging to this area. The areaGofHostFields is initialized as an empty Vector.

The getAreaSettings Method

This method should get all the settings from the *gof.ini* configuration file needed for this class.

This method takes a reference to the `IniFile` instance that holds the current server configuration file's settings, and a `String` with the name of the subsection used for the current Gui-on-the-Fly settings as parameters.

The adapter class only implements an empty method. Classes extending this class must extend this method.

The identifyArea Method

This method must implement the code for identifying the area that the class is supposed to identify. It should store the GofHostFields that belong to this area in a private `Vector`.

This method takes a `Vector` containing all the GofHostFields in the part of the screen area to be analyzed as a parameter. Normally it will be the whole screen area, but for a popup window only the popup window's area. It also takes four integer values as parameters. They are the start column, the start line, the width, and the height of the area to be analyzed.

The method must return a new `Vector` with all unused GofHostFields, that is, the GofHostFields from the first parameter not belonging to this area. These are then used as the first parameter for the next screen area and so on until all areas are identified.

The adapter class only implements an empty method that returns `null`. Classes extending this class must extend this method. They should use the areaGofHostFields Vector to store the GofHostFields belonging to this area.

The getAreasGofHostFields Method

This method must return a `Vector` containing all the GofHostFields belonging to this area.

The adapter class implements the above functionality. It returns the areaGofHostFields `Vector`.

The addControl Method

This method adds a new PhantomControl to this area. The method should store the control internally in the class. This makes it possible for this class to implement the laying out of the components.

This method takes a PhantomControl as a parameter.

The adapter class implements the above functionality. It stores the PhantomControls in the Vector phantomControls.

The Layout Method

This method should lay out the controls belonging to this area according to the layout rule specified by the parameter. If no layout rule is used, the control's location on the panel will be based on the host field's location on the host screen.

```
public static final int FLOW_LAYOUT = 1;
```

The adapter class only implements an empty method. Classes extending this class must extend this method if they want to implement their own layout handling.

1.8 GofControlIdentifier

The GofControlIdentifier is an interface that must be implemented by all the classes that will be used to identify different controls. An adapter class, GofControlIdentifierAdapter that implements the GofControlIdentifier interface is available. If the need to write your own GofControlIdentifier classes arises, it is recommended that you extend from the

adapter class, because it has default implementations for every method defined in the interface.

Each of the classes implementing this interface can load their own settings from the configuration file.

The class implementing this interface must have a default constructor without any parameters, since it will be created dynamically from the class name.

The methods that the implementing class must implement are:

```
public void setGofTemplatePanel( PhantomPanelData templatePanel );
public void getControlSettings( IniFile serverIni, String subsection );
public void identifyCtrls( GuiOnTheFlyRuntime gofRuntime,
                           GofHostAreaIdentifier areaIdentifier,
                           PhantomHostScreen phantomHostScreen,
                           HostScreen hostScreen,
                           PhantomPanelData newPanel,
                           int offsetX,
                           int offsetY );
```

Below follows a short description of what should be implemented in these methods and what the adapter class has implemented.

Constants Defined in the GofControlIdentifier Interface

The GofControlIdentifier interface defines several constants for calculating the positions and sizes of the controls in dialog units.

The unit for Gui-on-the-fly for GUI control positioning in X.

```
public static final int GOF_GUIUNITX  = 4;
```

The unit for Gui-on-the-fly for GUI control positioning in Y.

```
public static final int GOF_GUIUNITY  = 8;
```

The inter-line spacing for Gui-on-the-fly controls in X.

```
public static final int GOF_STEPX     = 4;
```

The inter-line spacing for Gui-on-the-fly controls in Y.

```
public static final int GOF_STEPY     = 10;
```

The margin for Gui-on-the-fly panels on the left and right side.

```
public static final int GOF_MARGINX   = GOF_STEPX/2;
```

The margin for Gui-on-the-fly panels at the top and bottom.

```
public static final int GOF_MARGINY   = GOF_STEPY/2;
```

How much can be offset before overlapping another field that is also offset in X.

```
public static final int GOF_OFFSETX   = 0;
```

How much can be offset before overlapping another field that is also offset in Y.

```
public static final int GOF_OFFSETY   = 2;
```

Instance Variables Defined in the Adapter Class

The adapter class defines a protected variable for PhantomPanelData called templPanel.

The getControlSettings Method

This method should get all the settings from the configuration file needed for this class.

The parameters for this method are a reference to the `IniFile` instance that holds the current server configuration file's settings and a `String` with the name of the subsection used for the current Gui-on-the-Fly settings.

The adapter class only implements an empty method. Classes extending this class must extend this method.

The identifyCtrls Method

This method is where the implementing class does all its work. The algorithm for identifying the control must be implemented here.

The classes implementing this interface should implement the following steps:

- Create PhantomControlBase
- Create PhantomHostField
- Create PhantomControl
- Add the control to PhantomPanelData
- Add the control to the appropriate GofHostAreaIdentifier
- Mark the GofHostField connected to the new control as processed

This class takes the current GuiOnTheFlyRuntime as the first parameter.

The second parameter is the area that is searched for the control type identified by this class.

The following parameters are the PhantomHostScreen, the HostScreen, PhantomPanelData for the template panel and the PhantomPanelData for the new panel that is being created.

The last two parameters are the offset in columns and lines for the current screen to be identified. This is important when calculating the position of the control for host popup windows.

The adapter class only implements an empty method. Classes extending this class must extend this method.

Instance Methods Defined in the Adapter Class

The adapter class has one additional method defined. This is the getVirtualSessionManager method, which can be used the access to the VirtualSessionManager.

The method takes one parameter, GuiOnTheFlyRuntime. The GuiOnTheFlyRuntime is available via the identifyCtrls method, so this method can easily be called from that method.

The method returns a handle to the VirtualSessionManager.

2 Description of the Default Identification Classes

The reason teaching has to go on
is that children are not born human;
they are made so.

Jacques Barzun

This section describes the default identification classes included in the distribution.

The description is very technical and will explain the actual implementation of the code. The administrators who are just going to use the default classes will probably not need to go into this level of detail. But for developers who are planning to write their own identification classes, understanding how the default classes work should be useful.

2.1 GofHostField

To make it easier to understand the implementation of the identifier classes, a short description of the GofHostField is called for.

The GofHostField is a wrapper object for the original host fields. Many of the GofHostField objects will correspond directly to a HostField object. But sometimes a HostField will be split into two or more GofHostFields, because logically it contains information that is not connected to each other. Typical examples are several output texts that have been written in the same host field with spaces between or on several lines using a line wrapping technique. This may happen if it was more convenient for the developer of the host system.

The GofHostFields are logical host fields that will be used during identification of the different controls.

The GofHostField keeps information such as start column, line, length, text, a reference to the original host field and a flag indicating if the original host field was empty. It also has a flag indicating if this GofHostField has been identified or not.

A new GofHostField is created by a constructor, which takes the HostField, the start column, the start line, the width, and a flag indicating if the host field is empty, as parameters.

There are methods for fetching all the information kept in the GofHostFields. In addition, there are methods for checking if the GofHostField's host field is protected or hidden. There is also a method for changing the GofHostField's text after it has been created.

The flag, indicating that the GofHostField has been identified, can be accessed directly. This flag is named `hasBeenProcessed`.

2.2 GofHostFieldIdentifiers

There are three GofHostFieldIdentifiers included in the default distribution. They are:

- GofAS400HostFieldIdentifier
- GofMainFrameHostFieldIdentifier
- GofMainFrameSplitHostFieldIdentifier

The GofAS400HostFieldIdentifier is for AS/400 based host applications. These applications usually have separate host fields for every control on the screen. Exceptions might be the function key information at the bottom of the screen, which often is described in a single host field.

The GofMainFrameHostFieldIdentifier is for mainframe host applications. These applications often have several output fields combined into a single host field. Many of these fields also wrap from one line to another. This class will split up a host field that wraps from one line to another into separate logical fields before control identification can take place.

The GofMainFrameSplitHostFieldIdentifier is for mainframe host applications. These applications often have several output fields combined into a single host field. Many of these fields also wrap from one line to another. Because of this many host fields have to be split up into several logical fields before control identification can take place.

GofAS400HostFieldIdentifier

This class takes all the host fields and creates GofHostFields from them. When creating the GofHostFields, it will not take into account if the area to identify is smaller than the actual screen, or if there is a popup window on the screen. These limitations will be taken into account when fetching the GofHostFields with one of the two methods for fetching.

This class extends the GofHostFieldIdentifierAdapter.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor doesn't do anything more than create the instance.

The method that creates the GofHostFields from the HostFields is the identifyHostFields method. It does this by looping through the HostFields. If it finds a HostField that wraps from one line to another, it will split this to two (or more, if it wraps to more than two lines) GofHostFields. For all other HostFields, a corresponding GofHostField will be created.

All the GofHostFields are stored in the gofHostFields Vector defined in the adapter class.

The getHostFields method is a method that returns all the GofHostFields on the screen. It takes a start column, start line, width, and height as parameters. The parameters determine the actual area of the host field that should be identified, so only GofhostFields inside this area will be returned.

When analyzing if a GofHostField is inside the area, it will consider the value of the flag doIdentifyPartialFields. If this flag is false, only GofHostFields completely inside the area will be included, otherwise GofHostFields partially inside the area will also be included.

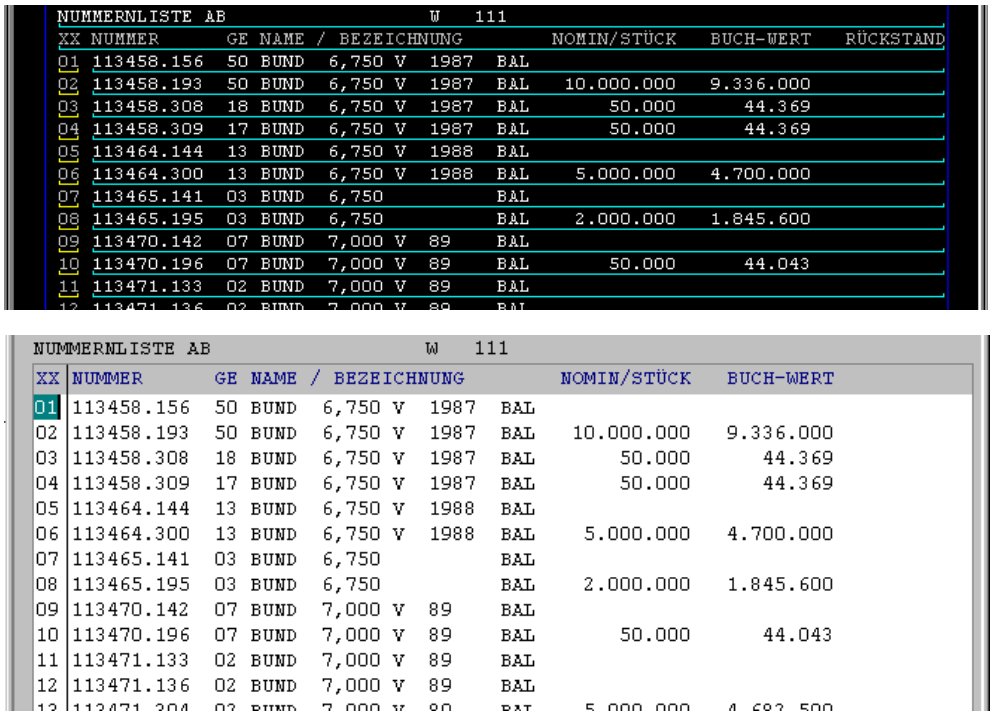
The GofHostFields that have been included will be returned in a Vector.

The getPopupWindowHostFields is the method to get the GofHostFields belonging to a popup-window. It takes a start column, start line, width and height for the popup window as parameters. It first saves the value of the doIdentifyPartialFields flag, sets it to false, since no partial fields should exist in a popup window. It then calls the getHostFields method, which retrieves the GofHostFields, and finally restores the doIdentifyPartialFields flag before returning the Vector with the included GofHostFields.

GofMainFrameHostFieldIdentifier

This is one of two GofHostFieldIdentifier classes written for use with mainframe host applications. This class takes all the host fields and creates GofHostFields from them. When creating the GofHostFields, it will not consider if the area to identify is smaller than the actual screen, or if there is a popup window on the screen. These limitations will be considered when fetching the GofHostFields with one of the two methods for fetching.

This class will not try to split up a HostField that is on one line into several logical GofHostFields. If this class is used in combination with list boxes that have several non-editable columns after each other, they will be created as a single column. See the screen capture below.



NUMMERNLISTE AB W 111

XX	NUMMER	GE	NAME / BEZEICHNUNG				NOMIN/STÜCK	BUCH-WERT	RÜCKSTAND
01	113458.156	50	BUND	6,750	V 1987	BAL			
02	113458.193	50	BUND	6,750	V 1987	BAL	10.000.000	9.336.000	
03	113458.308	18	BUND	6,750	V 1987	BAL	50.000	44.369	
04	113458.309	17	BUND	6,750	V 1987	BAL	50.000	44.369	
05	113464.144	13	BUND	6,750	V 1988	BAL			
06	113464.300	13	BUND	6,750	V 1988	BAL	5.000.000	4.700.000	
07	113465.141	03	BUND	6,750		BAL			
08	113465.195	03	BUND	6,750		BAL	2.000.000	1.845.600	
09	113470.142	07	BUND	7,000	V 89	BAL			
10	113470.196	07	BUND	7,000	V 89	BAL	50.000	44.043	
11	113471.133	02	BUND	7,000	V 89	BAL			

NUMMERNLISTE AB W 111

XX	NUMMER	GE	NAME / BEZEICHNUNG				NOMIN/STÜCK	BUCH-WERT
01	113458.156	50	BUND	6,750	V 1987	BAL		
02	113458.193	50	BUND	6,750	V 1987	BAL	10.000.000	9.336.000
03	113458.308	18	BUND	6,750	V 1987	BAL	50.000	44.369
04	113458.309	17	BUND	6,750	V 1987	BAL	50.000	44.369
05	113464.144	13	BUND	6,750	V 1988	BAL		
06	113464.300	13	BUND	6,750	V 1988	BAL	5.000.000	4.700.000
07	113465.141	03	BUND	6,750		BAL		
08	113465.195	03	BUND	6,750		BAL	2.000.000	1.845.600
09	113470.142	07	BUND	7,000	V 89	BAL		
10	113470.196	07	BUND	7,000	V 89	BAL	50.000	44.043
11	113471.133	02	BUND	7,000	V 89	BAL		
12	113471.136	02	BUND	7,000	V 89	BAL		

Fig. Typical list box result when using GofMainFrameFieldIdentifier. All the non-editable columns have been created as a single column, because all the contents on each line are in a single host field.

This class extends the GofHostFieldIdentifierAdapter.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor doesn't do anything more than create the instance.

The method that creates the GofHostFields from the HostFields is the identifyHostFields method. It does this by looping through the HostFields. If it finds a HostField that wraps from one line to another, it will split this to two (or more, if it wraps to more than two lines) GofHostFields. For all other HostFields, a corresponding GofHostField will be created.

All the GofHostFields are stored in the gofHostFields Vector defined in the adapter class.

The getHostFields method is a method that returns all the GofHostFields on the screen. It takes a start column, start line, width, and height as parameters. The parameters determine the actual area of the host field that should be identified, so only GofHostFields inside this area will be returned.

When analyzing if a GofHostField is inside the area, it will consider the value of the flag doIdentifyPartialFields. If this flag is false, only GofHostFields completely inside the area will be included, otherwise GofHostFields partially inside the area will also be included.

The GofHostFields that have been included will be returned in a Vector.

The getPopupWindowHostFields is the method to get the GofHostFields belonging to a popup-window. It takes a start column, start line, width, and height for the popup window as parameters. It first saves the value of the doIdentifyPartialFields flag, setting it to false, since no partial fields should exist in a popup window. It then calls the getHostFields method, which retrieves the GofHostFields, and finally restores the doIdentifyPartialFields flag before returning the Vector with the included GofHostFields.

GofMainFrameSplitHostFieldIdentifier

This is the second of two GofHostFieldIdentifier classes written for use with mainframe host applications. This class takes all the host fields and creates GofHostFields from them. When creating the GofHostFields, it will not consider if the area to identify is smaller than the actual screen, or if there is a popup window on the screen. These limitations will be considered when fetching the GofHostFields with one of the two methods for fetching.

This class will try to split up a HostField that is on one line into several logical GofHostFields. When doing this, it will try to look at the host fields above and below in order to try and identify logical columns. This is a very complex task and does not always produce the expected result. Even when it produces the expected result, it has the effect that if this class is used in combination with list boxes that have an empty, non-editable column between two other non-editable columns, it will not realize that there was an empty column, and this column will be missing from the list box. See screen capture below.

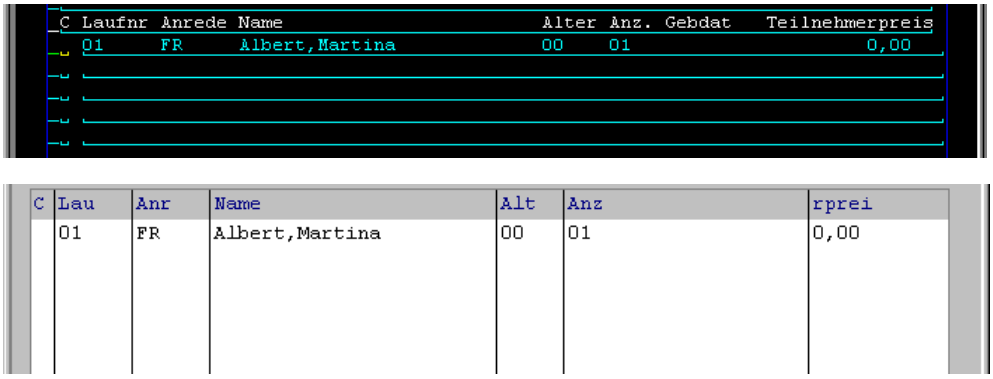


Fig. Typical problem with the GofMainFrameSplitHosFfieldIdentifier, where an empty column is missing from final list box. The missing column is the one with the header Gebdat.

This class extends the GofHostFieldIdentifierAdapter.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor doesn't do anything more than create the instance.

The method that creates the GofHostFields from the HostFields is the identifyHostFields method. It does this by looping through the HostFields. If it finds a HostField that wraps from one line to another, it will split this to two (or more, if it wraps to more than two lines) GofHostFields. For all other HostFields, a corresponding GofHostField will be created.

All the GofHostFields are stored in the gofHostFields Vector defined in the adapter class.

The getHostFields method is a method that returns all the GofHostFields on the screen. It takes a start column, start line, width, and height as parameters. The parameters determine the actual area of the host field that should be identified, so only GofHostFields inside this area will be returned.

When analyzing if a GofHostField is inside the area, it will take into account the value of the flag doIdentifyPartialFields. If this flag is false, only GofHostFields completely inside the area will be included, otherwise GofHostFields partially inside the area will also be included.

The GofHostFields that have been included will be returned in a Vector.

The getPopupWindowHostFields is the method to get the GofHostFields belonging to a popup-window. It takes a start column, start line, width, and height for the popup window as parameters. It first saves the value of the doIdentifyPartialFields flag, setting it to false, since no partial fields should exist in a popup window. It then calls the getHostFields

method, which retrieves the GofHostFields, and finally restores the doIdentifyPartialFields flag before returning the Vector with the included GofHostFields.

2.3 GofHostAreaIdentifiers

There are three GofHostAreaIdentifier classes included in the distribution. They are:

- GofTitleAreaIdentifier
- GofButtonAreaIdentifier
- GofMainAreaIdentifier

The GofTitleAreaIdentifier identifies an area at the top of the host screen that is used for special system and application information.

The GofButtonAreaIdentifier identifies the area at the lower part of the screen used for function key information. This information is often converted into push buttons, hence its name.

The last, the GofMainAreaIdentifier, takes what is left of the host area.

GofTitleAreaIdentifier

The identification of the area represented by this class is very simple. It just takes a start line and an end line from the configuration file and finds all the GofHostFields between these two lines, the specified start and end line included.

Fig. A typical terminal screen, where the two topmost lines should belong to the Title Area.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor doesn't do anything more than create the instance.

The settings for this class are read just after the instance of the class has been started, during server startup. Below is an example of how these two settings might look in the configuration file:

```
titlearea_startline=0
titlearea_endline=1
```

In this example, the area will be made up of the first two lines, suitable for the example in the figure above.

The current implementation of this area does not implement any layout managers for reformatting of the controls' positions.

GofButtonAreaIdentifier

The identification of the area represented by this class is also very simple. It just takes a start line and an end line from the configuration file and finds all the GofHostFields between these two lines.

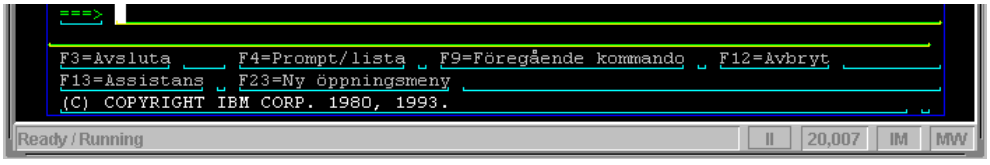


Fig. A typical AS/400 screen, where the last three lines should belong to the Button Area.

The constructor is a default constructor without any parameters. This makes it possible to easily load this class dynamically. The constructor only creates the instance.

The settings for this class are read during server startup just after the instance of the class has been started. The settings can either be absolute values or relative to the bottom of the screen. If they are relative to the bottom of the screen, they are specified as negative values. Note that absolute values are zero based.

Below are two examples of how these two settings might look in the configuration file. In the first example the position is calculated from the top of the screen, and in the second the position is calculated relative to the bottom of the screen.

Example 1:

```
buttonarea_startline=20
buttonarea_endline=23
```

Example 2:

```
buttonarea_startline=-4
buttonarea_endline=-1
```

There is also a setting for specifying a minimum window height. When the height of a popup window is lower than this value, the button area will not be identified at all. This setting looks like this:

```
buttonarea_minheight=5
```

The button area layout method can format the controls in flow layout. This means that they will be placed from left to right, in the same order as on the host screen, but ignoring their locations on the host screen. This setting looks like this:

```
buttonarea_layout=1
```

For example, if the lengths of the function key definitions are unequal, and the button layout has been cloned from the template panel (for an explanation of templates, see GofPushButtonIdentifier), every push button will be positioned the same distance from the previous button.

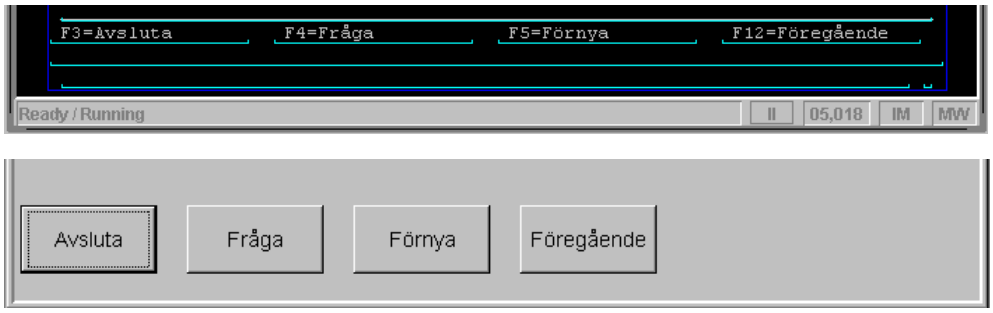


Fig. An example of a panel where the flow layout has been used for the button area. On the terminal screen the push button texts are spread evenly at the bottom of the popup area, but on the generated popup panel, the push buttons are aligned left.

The only valid values for the layout setting are:

```
0           no special layout (take position from host screen)
1           use flow layout
```

GofMainAreaIdentifier

Since this class represents everything that is left after all the other areas have been identified, it does not have any logic to identify itself. Therefore, it should work without a problem for any host application.

This class takes all the GofHostFields that have not been assigned to any other area and assigns them to the main area.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

The current implementation of this area does not implement any layout managers for reformatting of the controls' positions.

2.4 GofControlIdentifiers

These are the classes for identifying the different controls that are included in the default distribution. The classes are:

- GofSpinButtonIdentifier
- GofRadioButtonIdentifier
- GofCheckBoxIdentifier
- GofComboBoxIdentifier
- GofListContMarkIdentifier
- GofListBoxIdentifier
- GofHorLineIdentifier
- GofPushButtonIdentifier
- GofEntryFieldIdentifier
- GofOutputTextIdentifier
- GofTitleAreaTitleIdentifier

GofSpinButtonIdentifier

This class identifies spin button controls from unused GofHostFields. It will only create spin buttons with numerical ranges.

The identification of spin buttons is done by searching for a text prompt that indicates an entry field that could be replaced by a spin button. A text prompt indicating a spin button should have a start indicator, an end indicator and two values separated by a separator. The start indicator is usually a start parenthesis, the end indicator is usually an end parenthesis, and the separator is usually a hyphen. These are configurable in the configuration file.

After a text prompt has been identified as having a spin button indicator, the class checks whether the next or previous field is an entry field and whether it is on the same line. There is a configuration setting that specifies if the text prompt should be located before or after the entry field, or if both are accepted, and if so, in which order to search.

For the identification of spin buttons, the following settings are used:

```
spinrangestart    The start indicator for a value range.
spinrangeend      The end indicator for a value range.
```

```
spinvaluesep      The separator between the range values.
```

A typical example for these settings are shown below:

```
spinrangestart=(
spinrangeend=)
spinvaluesep=-
```

In the example above, a text prompt containing the text (1-99) would then indicate a spin button. The spin button would be created with the value range of 1 to 99. This means that the whole range must be included in the text that identifies the spin button.

There are two possibilities for the location of the spin button indicator relative to the spin button entry field. The indicator can either be in the preceding text prompt or in the trailing text prompt. The setting `spinsearch` will tell this class where to look for the spin button indicator, and in what order to search if multiple locations are to be searched.

Valid values are:

```
pre    Search the preceding text prompt .
post   Search the trailing text prompt .
```

It is possible to use both at the same time if both the preceding and trailing text prompt should be searched. The following example will first search the preceding text prompt. If no indicator is found there, the trailing text prompt will be searched.

```
spinsearch=pre,post
```

There is also a setting that specifies if a filler character should be used and what this character should be. This setting is called `spinfieldfiller`. If the filler character were the underscore character, it would look like this:

```
spinfieldfiller=_
```

This class also uses a setting that affects the look of the spin button. This is the `spinlayout` setting. Valid values for this setting are:

```
spinlayout=DEFAULT
spinlayout=FONT
spinlayout=COLOR
spinlayout=FONTANDCOLOR
```

DEFAULT means that no settings are taken from the template panel; default values will be used instead.

FONT means that the font is taken from a spin button with the Control ID `SB` (for non-editable spin buttons) or Control ID `SBE` (for editable spin buttons) in the template panel. If this control cannot be found or if it is not a spin button control, default values will be used.

COLOR means that the color is taken from a spin button with the Control ID `SB` (for non-editable spin buttons) or Control ID `SBE` (for editable spin buttons) in the template panel. If this control cannot be found or if it is not a spin button control, default values will be used.

FONTANDCOLOR means that the font and color are taken from a spin button with the Control ID `SB` (for non-editable spin buttons) or Control ID `SBE` (for editable spin buttons) in the template panel. If this control cannot be found or if it is not a spin button control, default values will be used.

Any other value will be treated as DEFAULT.

GofRadioButtonIdentifier

Identification of Radio buttons is not implemented in this version of Gui-on-the-Fly. Although identification of possible radio buttons can be done using rules similar to those for a combination box, there will be a problem with panel space for a group of radio

buttons, when the location of the controls is based on the location of the host fields on the host screen. Implementing a radio button will therefore require more complicated reformatting rules for the areas where radio buttons will be identified.

GofCheckBoxIdentifier

This class identifies checkbox controls from unused GofHostFields.

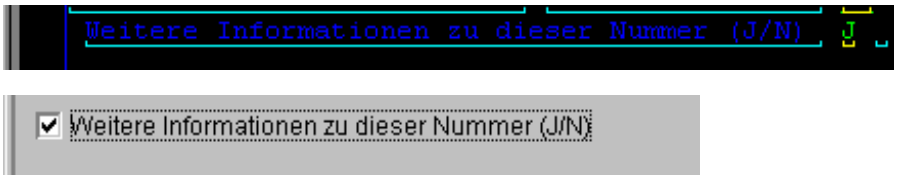


Fig. Checkbox created by the GofCheckBoxidentifier

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

The identification of checkboxes is done by searching for a text prompt that indicates an entry field that could be replaced by a checkbox. A text prompt indicating a checkbox should have a start indicator, an end indicator and two text strings separated by a separator. The start indicator is usually a start parenthesis, the end indicator is usually an end parenthesis, and the separator is usually a slash. But these are configured in the configuration file.

After a text prompt has been identified as having a checkbox indicator, the class checks if the next or previous field is an entry field and whether it is on the same line. There is a configuration setting that determines if the text prompt should be before or after the entry field, or if both are accepted, and if so, in which order to search.

For the identification of checkboxes, the following settings are used:

```
checkstart    The start indicator for a list of alternatives.
checkend      The end indicator for a list of alternatives.
checksep      The separator between the alternatives in the list.
```

A typical example of these settings is shown in the example below:

```
checkstart=(
checkend=)
checksep=/
```

There are also several possibilities for the location of the checkbox indicator relative to the checkbox entry field. The indicator can either be in the preceding text prompt or in the trailing text prompt. The setting `checksearch` will tell this class where to look for the checkbox indicator, and in what order to search if multiple places are to be searched.

Valid values are:

```
pre    Search the preceding text prompt.
post   Search the trailing text prompt.
```

It is possible to use both at the same time if both the preceding and trailing text prompt should be searched. The following example will first search the preceding text prompt. If no indicator is found there, the trailing text prompt will be searched.

```
checksearch=pre post
```

There is also a setting that specifies if there is a filler character to use, and what this character should be. This setting is called `checkfieldfiller`. If the filler character were the underscore character, it would look like this:

```
checkfieldfiller=_
```

To make it possible for the GofCheckBoxIdentifier to know which of the two text strings in the indicator is the select string and which is the unselect string, there are two settings where all the valid select strings and unselect strings can be listed. These two settings look like this:

```
checkselect=J Y
checkunselect=N
```

In the example above, both J (for Ja) and Y (for Yes) are valid select strings, and N (for No, Nein and Nej) is a valid unselect string.

This means that if there are checkboxes where there are presumptive checkboxes, where some have Yes as checked, and others have No as checked, checkboxes can only be used for one of these cases.

It is also possible for the text prompt where the checkbox indicator was found to be created as a separate output text/static text, or to be moved to the checkbox's text. See the figure below.

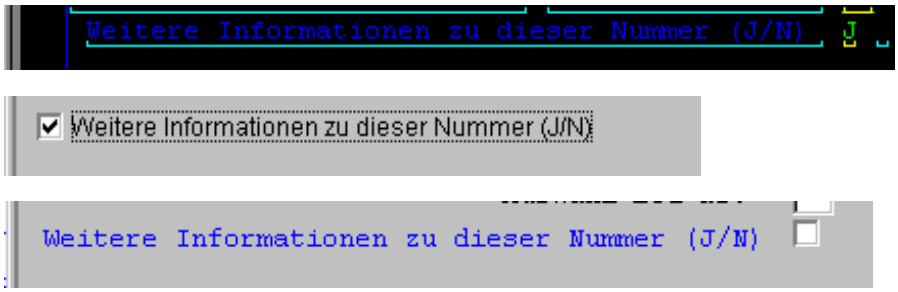


Fig. First panel has text prompt moved to the checkbox's text, in the second panel it has been left as the original output text.

The `checktextfromindicator` setting specifies which alternative should be used. To move the text to the checkbox, set this setting to 1, otherwise to 0.

```
checktextfromindicator=1
```

This class also uses a setting that affects the look of the checkboxes. This is the `checklayout` setting. Valid values for this setting are:

```
checklayout=DEFAULT
checklayout=FONT
checklayout=COLOR
checklayout=FONTANDCOLOR
```

DEFAULT means that no settings are taken from the template panel; default values will be used instead.

FONT means that the font is taken from template panel, from a checkbox with the Control ID `CHK`. If this control cannot be found, or if it is not a checkbox control, default values will be used.

COLOR means that the color is taken from template panel, from a checkbox with the Control ID `CHK`. If this control cannot be found, or if it is not a checkbox control, default values will be used.

FONTANDCOLOR means that the font and color are taken from template panel, from a checkbox with the Control ID `CHK`. If this control cannot be found, or if it is not a checkbox control, default values will be used.

Any other value will be treated as DEFAULT.

GofComboBoxIdentifier

This class identifies combination box controls from unused GofHostFields.

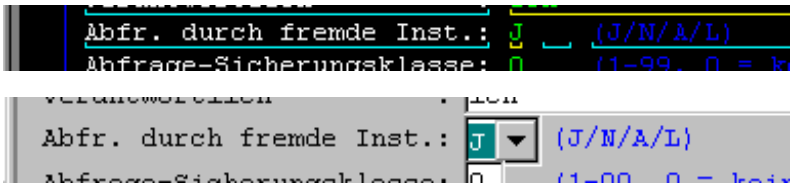


Fig. A combination box identified from the trailing text prompt by the GofComboBoxidentifier

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

The identification of combination boxes is done by searching for a text prompt that indicates an entry field that could be replaced by a combination box. A text prompt indicating a combination box should have a start indicator, an end indicator and two or more values (the combo list) separated by a separator. The start indicator is usually a start parenthesis, the end indicator is usually an end parenthesis, and the separator is usually a slash. These are configurable in the configuration file.

After a text prompt has been identified as having a combination box indicator, the class checks whether the next or previous field is an entry field, and if it is on the same line. There is a configuration that determines if the text prompt should be before or after the entry field, or if both are accepted, and if so, in which order to search.

For the identification of combination boxes, the following settings are used:

```
combolistst art    The start indicator for a list of alternatives.
combolistend       The end indicator for a list of alternatives.
combolistsep       The separator between the alternatives in the list.
```

A typical example of these settings is shown in the example below:

```
combolistst art=(
combolistend=)
combolistsep=/
```

There are also several possibilities for the location of the combination list indicator relative to the combination box entry field. The list can either be in the preceding text prompt or in the trailing text prompt. The setting `combosearch` will tell this class where to look for the combolist, and in what order to search if multiple places are to be searched.

Valid values are:

```
pre    Search the preceding text prompt.
post   Search the trailing text prompt.
```

It is possible to use both at the same time if both the preceding and trailing text prompt should be searched. The following example will first search the preceding text prompt. If no indicator is found there, the trailing text prompt will be searched.

```
combosearch=pre,post
```

To handle situations in which the original text in the host field is not part of the drop-down list, the setting `comboaddmissingtext` is used.

Valid values for this setting are 0 and 1. 0, which means that the original text should not be added to the list. In this case the combination box will be created as an editable combination box. 1 means that the text should be added to the list. Default is 0.

It is also possible to force combination boxes to always be created as editable boxes. This is specified in the `comboalwayseditable` setting.

```
comboalwayseditable=1
```

There is also a setting that specifies if there is a filler character to use, and what this character should be. This setting is called `combofieldfiller`. If the filler character were the underscore character, it would look like this:

```
combofieldfiller=_
```

This class also uses a setting that affects the look of the combination boxes. This is the `combolayout` setting. Valid values for this setting are:

```
combolayout=DEFAULT
combolayout=FONT
combolayout=COLOR
combolayout=FONTANDCOLOR
```

DEFAULT means that no settings are taken from the template panel; default values will be used instead.

FONT means that the font is taken from a combination box with the Control ID `CMB` (for non-editable combination boxes) or `CMBE` (for editable combination boxes) from the template panel. If this control cannot be found, or if it is not a combination box control, default values will be used.

COLOR means that the color is taken from a combination box with the Control ID `CMB` (for non-editable combination boxes) or `CMBE` (for editable combination boxes) from the template panel. If this control cannot be found, or if it is not a combination box control, default values will be used.

FONTANDCOLOR means that the font and color are taken from a combination box with the Control ID `CMB` (for non-editable combination boxes) or `CMBE` (for editable combination boxes) from the template panel. If this control cannot be found, or if it is not a combination box control, default values will be used.

Any other value will be treated as DEFAULT.

GofListContMarkIdentifier

This class identifies AS/400 subfile continuation marks.

A list continuation mark is used by the AS/400 at the end of a subfile to inform the user if there are more list pages in the subfile.

Typical list continuation marks are:

```
+      (plus sign, created with the Display File option *PLUS)
More   (created with the Display File option *MORE)
Bottom (created with the Display File option *MORE, when last page in
        list is displayed)
```

If a continuation mark is found, a flag is set to mark this, and the line on which the mark was found is saved as the last line in the list.

If a continuation mark has been found, the search will be terminated.

This class needs two settings from the configuration file. The first setting, called `listcontmark`, is used to specify the texts that are valid continuation marks. A typical example would be like the one below:

```
listcontmark=+,More,MORE
```

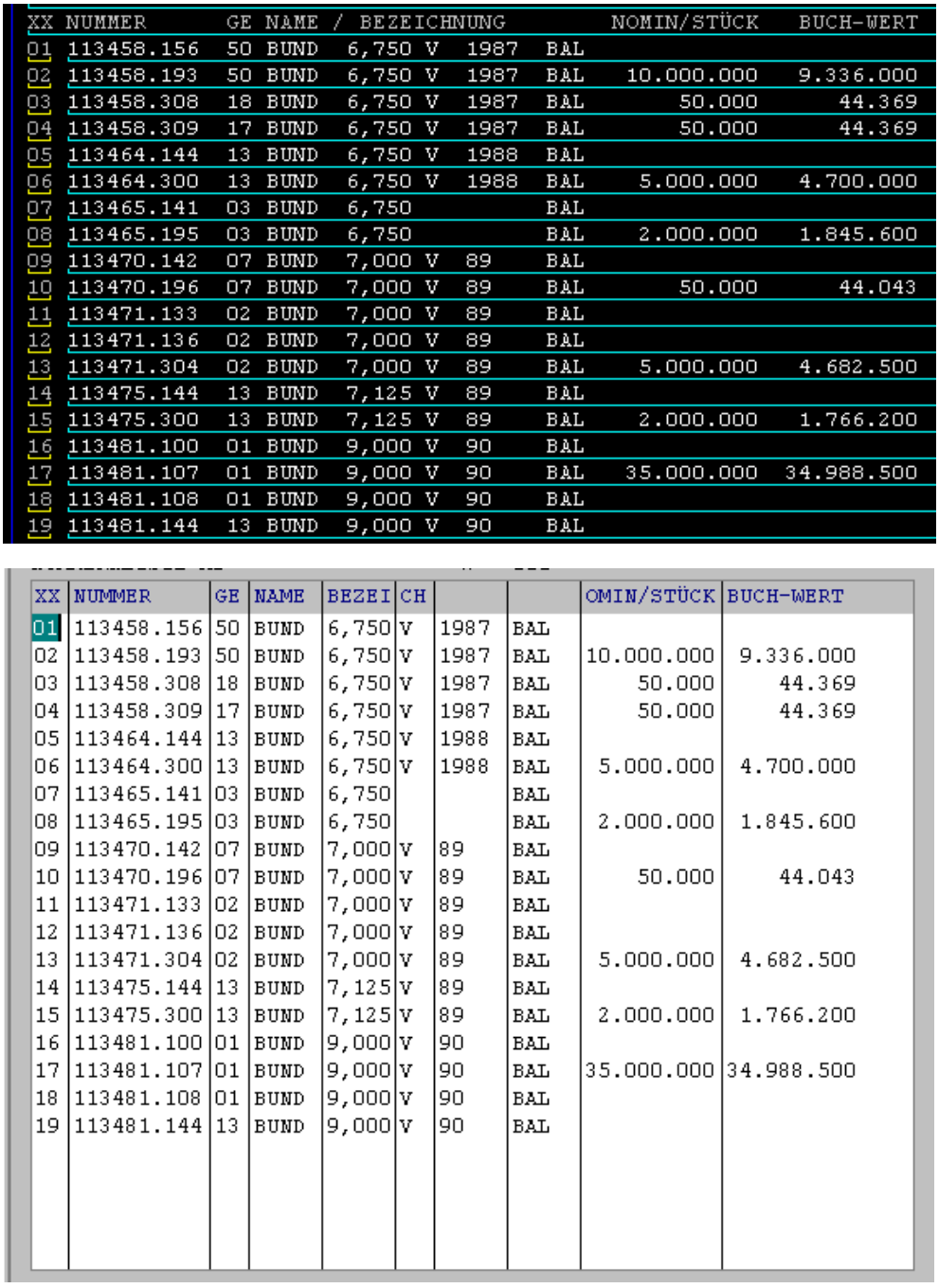
The second setting, called `listcontpos`, is used to specify from which column to start searching for the continuation mark. This can either be the real column, or a column relative to the right side. In the latter case it is specified as a negative value.

```
listcontpos=-13
```

In this example the search for the continuation mark would only be done in the last 13 columns.

GofListBoxIdentifier

This class identifies list box controls from unused GofHostFields. This class will only try to identify selectable lists.



XX	NUMMER	GE	NAME	BEZEICHNUNG				NOMIN/STÜCK	BUCH-WERT
01	113458.156	50	BUND	6,750	V	1987	BAL		
02	113458.193	50	BUND	6,750	V	1987	BAL	10.000.000	9.336.000
03	113458.308	18	BUND	6,750	V	1987	BAL	50.000	44.369
04	113458.309	17	BUND	6,750	V	1987	BAL	50.000	44.369
05	113464.144	13	BUND	6,750	V	1988	BAL		
06	113464.300	13	BUND	6,750	V	1988	BAL	5.000.000	4.700.000
07	113465.141	03	BUND	6,750			BAL		
08	113465.195	03	BUND	6,750			BAL	2.000.000	1.845.600
09	113470.142	07	BUND	7,000	V	89	BAL		
10	113470.196	07	BUND	7,000	V	89	BAL	50.000	44.043
11	113471.133	02	BUND	7,000	V	89	BAL		
12	113471.136	02	BUND	7,000	V	89	BAL		
13	113471.304	02	BUND	7,000	V	89	BAL	5.000.000	4.682.500
14	113475.144	13	BUND	7,125	V	89	BAL		
15	113475.300	13	BUND	7,125	V	89	BAL	2.000.000	1.766.200
16	113481.100	01	BUND	9,000	V	90	BAL		
17	113481.107	01	BUND	9,000	V	90	BAL	35.000.000	34.988.500
18	113481.108	01	BUND	9,000	V	90	BAL		
19	113481.144	13	BUND	9,000	V	90	BAL		

Fig. Part of a host screen that has been identified as a list box.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

The list box control is probably the control whose identification is the most difficult. The identification of list boxes is done in several steps.

First the identification process goes through all unused GofHostFields and creates column elements from them. GofHostFields on consecutive lines that start in the same column and have the same width will be placed in a single column element.

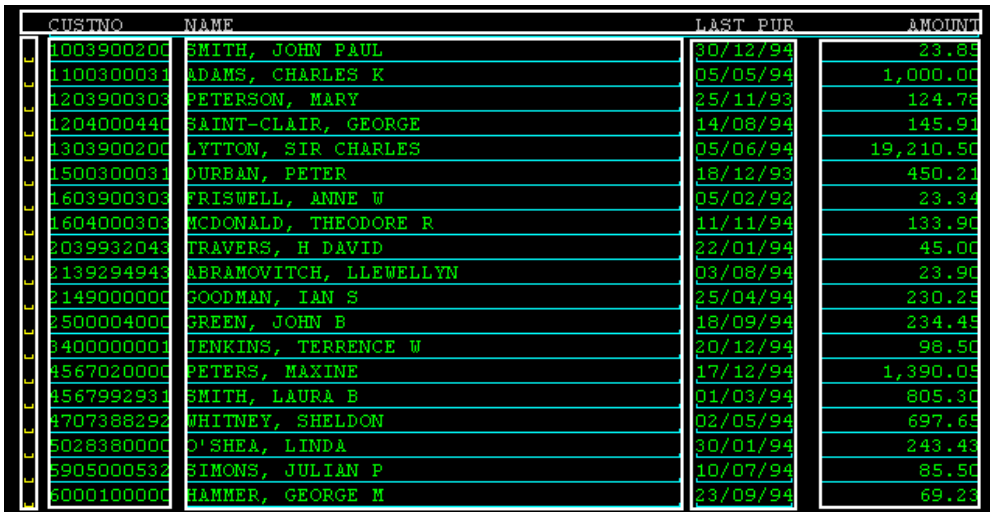


Fig. A potential list on a host screen, where the column elements are marked with white rectangles.

In the second step, the identification process will go through all column elements, and create list elements from them. Column elements starting on the same line and having the same height are considered to belong to a single list element.

When comparing the column element start lines, consideration is taken that a column header field might have been included at the top of some of the column elements. For example, if one column element starts line above the others but the end line is the same, it will be considered to belong to the same list element. The first line in that column will be used in the header.

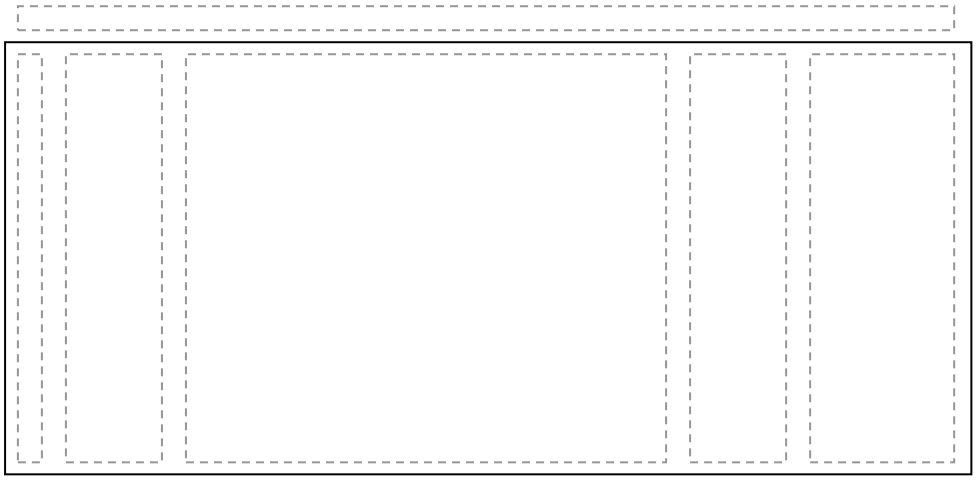


Fig. A schematic picture of the column elements from the previous picture shown with dashed lines. All column elements starting on the same line and with the same height are combined into a list box element, shown by the solid line.

Next, the identification process will check if a list element has the minimum number of lines and columns required for a list. The minimum number of lines and columns is specified in the configuration file. It will also check if the list element has texts or entry fields to the left, if so it will *not* be considered a candidate for a list box. This means that the list box should be the first control from the left. This will always be true for a list in an AS/400 application created as a subfile.

The remaining list box candidates will be checked to see if the first column is editable and has a maximum width no greater than the value specified for the `listboxmaxsellength` setting (see below). If everything checks out, this column is accepted as a selection column, and the list is accepted as a list box.

As a final step, the identification process will try to find the column headers before creating the list box.

There are four settings for the list box that can be specified in the configuration file. They are listed below with typical values.

```
listboxmincols=2
listboxminlines=2
listboxmaxheaderheight=2
listboxmaxsellength=2
```

The `listboxmincols` specifies the minimum number of columns required in a list box candidate for a list box control. The default value is two.

The `listboxminlines` specifies the minimum number of lines required in a list box candidate for a list box control. The default value is two.

The `listboxmaxheaderheight` specifies the maximum number of lines to search for the column headers. The default value is one.

The `listboxmaxsellength` specifies the maximum number of characters allowed for the selection field. The default value is two.

There is also a setting that specifies if there is a filler character to use, and what this character should be. This setting is called `listfieldfiller.` If the filler character were the underscore character, it would look like this:

```
listfieldfiller=_
```

This class has a setting that affects the appearance of the entry fields. This is the `listboxlayout` setting. Valid values for this setting are:

```
listboxlayout=DEFAULT
listboxlayout=FONT
listboxlayout=COLOR
listboxlayout=LINES
```

DEFAULT means that no settings are taken from the template panel; default values will be used instead.

FONT means that the fonts are taken from a list box with the Control ID `LISTB` in the template panel. If this control cannot be found, or if it is not a list box control, default values will be used. Both the header font and the list font are taken from the template list box.

COLOR means that the colors are taken from a list box with the Control ID `LISTB` in the template panel. If this control cannot be found, or if it is not a list box control, default values will be used. The colors that will be taken from the template list box are the background color and the foreground colors for both the header and the list.

LINES means that all the line settings are taken from a list box with the Control ID `LISTB` in the template panel. If this control cannot be found, or if it is not a list box control, default values will be used. The settings that are taken from the template list box are line between columns, line between headers, line between header and column and the line between lines.

Any other value will be ignored.

All these layout settings can be combined in a suitable way. For example, if FONT, COLOR and LINES were desired, the setting would look like this:

```
listboxlayout=FONT COLOR LINES
```

GofColumnElement

The GofColumnElement is a private inner class that is used to store information about the column elements created during the list box identification process. A column element is a column of one or more GofHostFields that starts in the same x-position, and has the same width.

In addition to some methods to set and retrieve information, it also contains some methods for retrieving column headers and one method that checks if the column element is editable.

The used GofColumnElements will be deleted when a new host screen is processed.

GofListElement

The GofListElement is a private inner class that is used to store information about the list elements created during the list box identification process. A list element consists of one or more column elements, where all column elements start on the same line, and have the same height.

The GofListElement has a method that checks if the list element's leftmost column has any other column elements to its left.

GofListColumnElement

The GofListColumnElement is a container class for all the data needed by the PhantomCListBox class to create its inner classes for the columns. By wrapping all the information into a class, the number of parameters to the PhantomCListBox's constructor is minimized.

GofHorLineIdentifier

This class identifies horizontal line controls from unused GofHostFields.

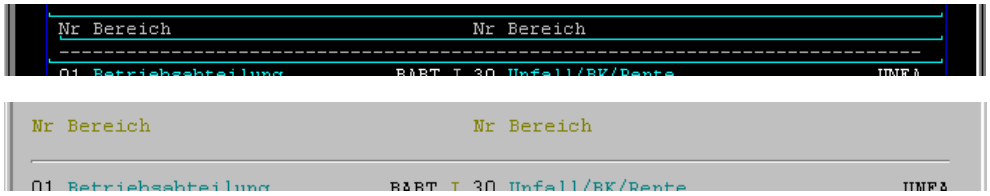


Fig. Example of horizontal line identified by the GofHorLineIdentifier.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

The class will check for a GofHostField that consists of only one, repeated character that belongs to the list of characters that have been specified as valid characters for the horizontal line. The number of times the character is repeated in the GofHostField must equal or exceed the minimum number specified in the configuration file.

The characters that may be converted into a horizontal line are specified in the configuration file. The setting might look like the one below. In this example, the only valid character is a hyphen.

```
horlinechars=-
```

The minimum number of times that this character must be repeated, before it is considered a horizontal line, is also specified in the configuration file.

```
horlineminlength=10
```

The horizontal line will be created as a PhantomCRectangle, with a height of 1, and a depth of 1.

GofPushButtonIdentifier

This class identifies push button controls from unused GofHostFields.

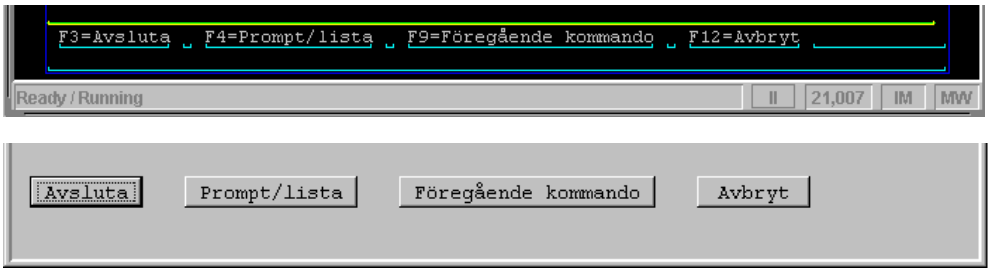


Fig. Example of push buttons identified by the GofPushButtonIdentifier.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

This class takes every unused protected host field and checks if it has a push button identification string. The push button identification strings are text strings that identify a function key definition. The push button identification strings are defined in the configuration file. When a host field is found to have one or more push button identification strings in its text, a push button element is created for each string. The push button element text will also be analyzed for the send key.

For each of the push button elements, a push button or a menu item will be created. There are settings in the configuration file that determine which type of control will be created.

Several settings that are needed during the identification of the push button controls are taken from the configuration file. There is one setting that should contain the push button identification strings. This setting is a comma-separated list, with all the identification strings. This means that the identification strings can't contain commas.

The identification strings will be combined with the value of the `pushbuttonequalsign` setting to create the complete identification string that will be searched for. The equal sign is specified separately because if push button combinations are allowed, the class requires an identification string without the equal sign.

It is possible to use a wildcard for numeric characters in the identification strings. The character used as a wildcard is the hash mark ('#').

The `pushbuttonequalsign` setting will usually be an equal sign ('=') or a colon (':').

Below is an example of pushbutton identification strings with wildcards, and an equal sign as the `pushbuttonequalsign`.

```
pushbuttonident=PF#,PF##,F#,F##
pushbuttonequalsign==
```

Remember that the search order for the host field text will be the same order as the order of the identification strings in the configuration file.

There is also a setting that specifies if push button combinations should be searched for. A push button combination is when two push button definitions have been nestled together, separated by separator character. An example of a pushbutton combination could be:

```
PF7/PF8=Previous/Next
```

If the setting `pushbuttoncombination` is set to 1, then this type of combination is searched for. This also requires that the `pushbuttoncombsign` be set to the separator character. In the above example, the separator character is the slash (‘/’). The setting below handles the above example:

```
pushbuttoncombination=0
pushbuttoncombsign=/
```

This class also has a setting that affects the look of the entry fields. This is the `pushbuttonlayout` setting. Valid values for this setting are:

```
pushbuttonlayout=DEFAULT
pushbuttonlayout=SIZE
pushbuttonlayout=FONT
pushbuttonlayout=SIZEANDFONT
pushbuttonlayout=CLONE
```

DEFAULT means that no settings are taken from the template panel. Size will be taken from the size used by the definition on the host screen. The panel's default font will be used.

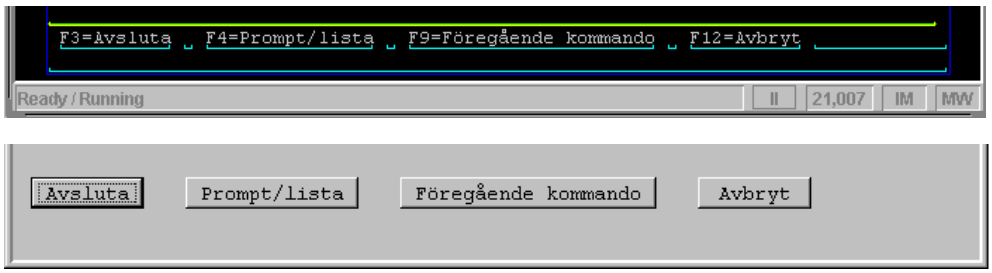


Fig. Example of push buttons created with pushbuttonlayout=DEFAULT and buttonarea_layout=0.

SIZE means that the size will be taken from a push button with the Control ID `PB_X` in the template panel. The panel's default font will be used. If this control cannot be found, default values will be used.

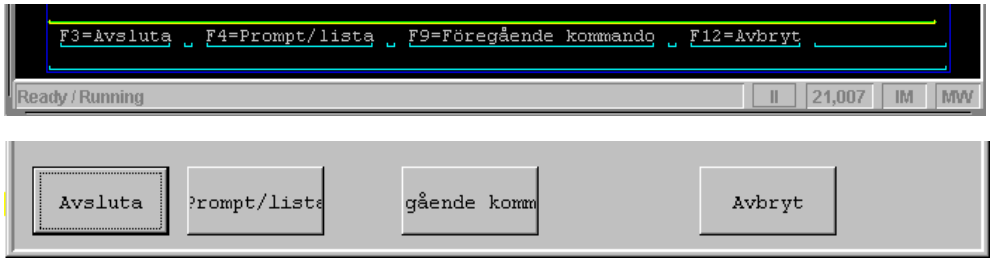


Fig. Example of push buttons created with pushbuttonlayout=SIZE and buttonarea_layout=0. In this case some of the button texts are too long for the size defined in the template panel.

FONT means that the font is taken from a push button with the Control ID `PB_X` in the template panel. If this control cannot be found, default values will be used.

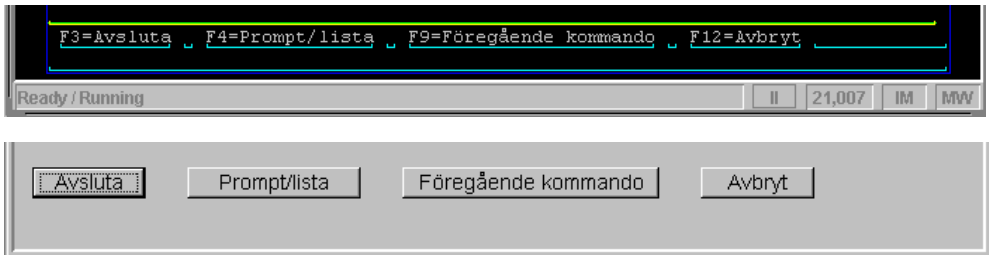


Fig. Example of push buttons created with pushbuttonlayout=FONT and buttonarea_layout=0.

SIZEANDFONT means that the font and size are taken from a push button with the Control ID `PB_X` in the template panel. If this control cannot be found, default values will be used.

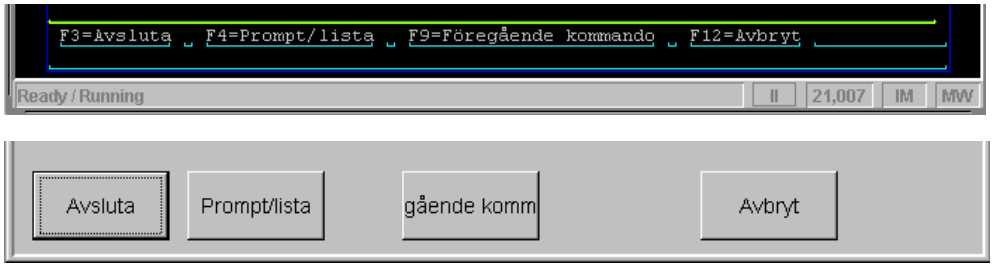


Fig. Example of push buttons created with pushbuttonlayout=SIZEANDFONT and buttonarea_layout=0. In this case some of the button texts are too long for the size defined in the template panel.

CLONE means that the button on the template panel is cloned. Cloned means that the size, font, text and an icon, if it exists, are taken from a push button with the Control ID `PB_F`*nn* (*nn* is one or two numeric characters, an exact match must be found). If this control cannot be found, default values will be used.

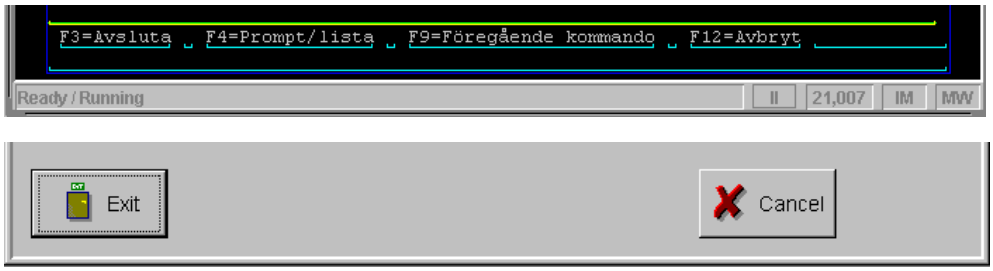


Fig. Example of push buttons created with pushbuttonlayout=CLONE and buttonarea_layout=0. Only the buttons with function keys defined in the template panel will be created as push button.
In the above example, only F2 and F12 are defined.

Fig. The same example as above, but with buttonarea_layout=1, which means that flow layout should be used.

It is possible to let push buttons not found in the template file be created as menu items instead. This is only possible when the push button layout is cloned from the template file, because push buttons not found in the template would have a different look from the cloned ones. For this there are two settings. The `nontemplatebutton` setting turns this option on and the `nontemplatemenu` setting specifies the menu name to use for the menu items. The value for the menu name can have a shortcut specified by placing a tilde (‘~’) in front of the character to be used as a shortcut. Below is an example of this setting.

```
nontemplatebutton=MENU
```

```
nontemplatemenu=~Functions
```

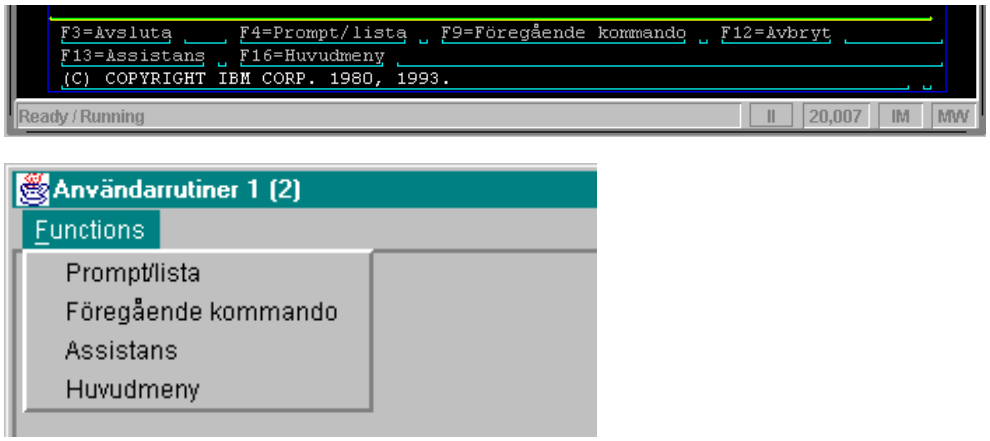


Fig. Example of function key definitions moved to a menu. In this case pushbuttonlayout=CLONE, nontemplatebutton=MENU and nontemplatemenu=~Functions, and F4, F9, F13 and F16 were not defined in the template button.

GofPushButtonElement

The GofPushButtonIdentifier will make use of a private inner class named GofPushButtonElement. This is used to store information about the push button element found during the matching of identification strings. This information is used during the creation of the push button control or the menu item.

The GofPushButtonElement also has a method to get the send key string that will be attached to the control.

GofEntryFieldIdentifier

This class identifies entry field controls from unused GofHostFields.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

This class takes each unused GofHostField that is not protected and creates an entry field. It will check if the current GofHostField points to the same host field as the previous GofHostField. If so, this is a continuation of the previous one and will not be created.

This class has a setting that affects the look of the entry fields. This is the `entryfieldlayout` setting. Valid values for this setting are:

```
entryfieldlayout=DEFAULT
entryfieldlayout=FONT
entryfieldlayout=COLOR
entryfieldlayout=FONTANDCOLOR
```

DEFAULT means that no settings are taken from the template panel; default values will be used instead.

FONT means that the font is taken from a template panel, from an entry field with the Control ID `E`, `ER`, `ENL` or `ENR` (depending on the host field's justification, and if the host field is numeric or not). If this control cannot be found, or if it is not an entry field control, default values will be used.

COLOR means that the color is taken from a template panel, from an entry field with the Control ID `E`, `ER`, `ENL` or `ENR` (depending on the host field's justification, and if the host field is numeric or not). If this control cannot be found, or if it is not an entry field control, default values will be used.

FONTANDCOLOR means that the font and color are taken from a template panel, from an entry field with the Control ID `E`, `ER`, `ENL` or `ENR` (depending on the host field's justification, and if the host field is numeric or not). If this control cannot be found, or if it is not an entry field control, default values will be used.

Any other value will be treated as DEFAULT.

There is also a setting that specifies if there is a filler character to use, and what this character should be. This setting is called `entryfieldfiller`. If the filler character were the underscore character, it would look like this:

```
entryfieldfiller=_
```

GofOutputTextIdentifier

This class identifies output text/static text controls from unused GofHostFields.

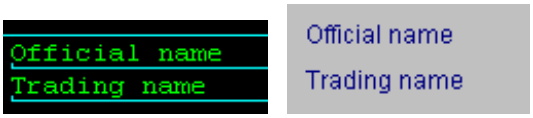


Fig. Example of output text.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

Every unused, protected GofHostField that is not empty will result in either an output text or a static text. The type of control that will be created depends on the settings for this class in the configuration file.

There is a setting for this class that makes it possible to remove trailing punctuation marks. These punctuation marks are either dots (.) or colons (:). Remember that if the text is changed, the control that will be created will always be a static text, not output text. The setting looks like this:

```
deltrailingpunctmark=1
```

It is also possible to specify that the controls created by this class should always be static texts. To do this, set the `alwaysusestatictext` setting to 1, otherwise to 0. In the example below, static texts will always be used.

```
alwaysusestatictext=1
```

Note: Keep in mind that using static text instead of output text requires more processing resources in the server.

There is also a setting that affects the look of the output texts/static texts. This is the `outputtextlayout` setting. Valid values for this setting are:

```
outputtextlayout=DEFAULT
outputtextlayout=FONT
outputtextlayout=COLOR
outputtextlayout=FONTANDCOLOR
```

DEFAULT means that no settings are taken from the template panel. Default values will be used.

FONT means that the font is taken from template panel, from an output text with the Control ID `O`, `OR`, `ONL` or `ONR` (depending on the host field's justification and if the host field is numeric or not). If this control cannot be found, or if it is not an output text control, default values will be used.

COLOR means that the color is taken from template panel, from an output text with the Control ID `O`, `OR`, `ONL` or `ONR` (depending on the host field's justification and if the host field is numeric or not). If this control cannot be found, or if it is not an output text control, default values will be used.

FONTANDCOLOR means that the font and color are taken from template panel, from an output text with the Control ID `O`, `OR`, `ONL` or `ONR` (depending on the host field's justification and if the host field is numeric or not). If this control cannot be found, or if it is not an output text control, default values will be used.

Any other value will be treated as DEFAULT.

Remember that the default color for an output text is black, and for a static text it is dark blue. This means that if the color for the output text with `O` is set to default, output texts and static texts will receive different colors on the generated panel.

GofTitleAreaTitleIdentifier

This class identifies the screen title from unused GofHostFields. If a title can be identified, the panel title will be the same as the screen title.

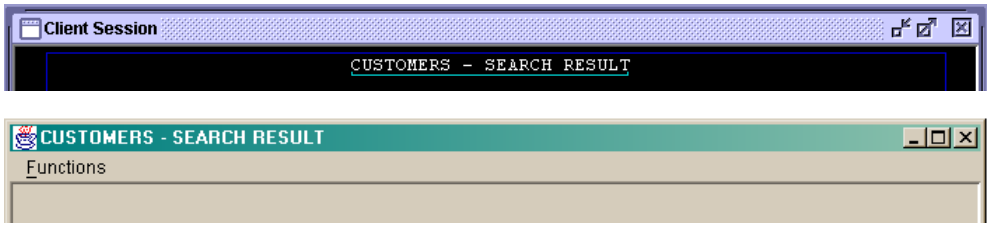


Fig. Example of a title identified by the GofTitleAreaTitleIdentifier. It has been placed in the panel's title bar by the identifier.

The constructor is a default constructor without any parameters. This makes it possible to load this class dynamically. The constructor only creates the instance.

The identification of the title is quite simple. The title is supposed to be on a specific location on the screen. The location is specified in the configuration file. If a protected host field is found at the specified location, it will be accepted as the screen's title.

The setting for title identifier allows two different ways of specifying the location of the title on the host screen. It can either be specified as combination of the line it starts on, the line it ends on, the first column that it can start on, the last column it can start on, a minimum length and a maximum length; or it can be specified as a combination of the line it starts on, the line it ends on and a location on the line.

The location can either be left, center or right. Left means that host field must start at the first column; center means that it must cross the center of the line and right means that it must end at the last column on the line.

The setting, which is named `title_title`, consists of a comma separated list consisting of the start line, the end line, the first allowed start column, the last allowed start column, the minimum width, the maximum width, and the position on the line. Values not needed must be marked with a hyphen ('-'). The format of this setting is:

```
title_title=sl,el,fsc,lsc,minw,maxw,pos
```

where the parameters are:

```
sl          start line (zero based)
el          end line (zero based)
fsc         first allowed start column (zero based)
lsc         last allowed start column (zero based)
minw        the minimum width in characters
maxw        the maximum width in characters
pos         position on line, valid values are L=left, C=center,
            R=right or -
```

Use either the `fsc, lsc, minw, maxw` combination or the `pos` parameters (set the unused to -).

An example of the first option could be:

```
title_title=0,0,20,30,20,40,-
```

In this example, the title must be on the first line, must start somewhere between column 20 and 30, and must have a width of between 20 and 40 characters.

And an example of the second option could be:

```
title_title=0,0,-,-,-,-,C
```

In this example the title must cross the center of the first line.

Normally the title will just be on a single line, so the first two values will be the same.

3 Checklist for Configuring the Default Identification Classes

A tiny change today brings us to a dramatically different tomorrow.
There are grand rewards for those who pick the high hard roads,
but those rewards are hidden by years.

Richard Bach

By changing the settings in the configuration file *gof.ini*, it is possible to change the way a panel created by the Gui-on-the-Fly module will look. This section tries to describe the most important things to consider when configuring all the different settings.

Go through each step and try to see what setting a good starting point might be.

3.1 Analyze the Host Applications

The first thing to do is to look through all the host screens (or as many as possible). This will give you a good feeling for how consistent the host screens are. Look for similarities in the host screens.

3.2 Identify Whole Screen or Partial Screen

First see if there is any area on the host screen that should always be removed from the identification. If so, set the `hostscreenarea` and the `identifypartialfields` settings accordingly. Remember that any area to exclude will be excluded from each and every one of the screens in the system.

Normally the whole screen should be processed, and the settings would the be:

```
hostscreenarea=0,0,0,0
identifypartialfields=0
```

3.3 Decide Which Host Field Identifier Class to Use

This should be very simple. If it is a normal AS/400 host application, use the GofAS400HostFieldIdentifier and if it is a mainframe host application, use the GofMainFrameHostFieldIdentifier.

Of course, for some unusual host application, it might be necessary to write a new host field identifier class.

Set the `hostfieldidentifier` setting to the chosen class. For an AS/400 this would be:

```
hostfieldidentifier=se.entra.NetPhantom.server.GofAS400HostFieldIdentifier
```

Remember that the whole package name for the class must be included, otherwise the class will not be found, and Gui-on-the-Fly will not be started.

3.4 Identify the Different Areas on the Host Screen

Try to visually identify distinct areas on the host screen. If these areas are always located on the same screen rows, either on absolute rows, or on rows relative to the bottom of the screen, they should probably be identified as host screen areas.

Typical areas should be the title area, the button area (or function key definition area), message area, and so on.

When these areas have been identified, give them unique names, and specify the names in the setting named `areaidentifiers`. A typical but very simple example is given below.

```
areaidentifiers=TITLEAREA BUTTONAREA MAINAREA
```

The order in which they are specified does not matter.

3.5 Choose Area Identifier Classes for the Host Areas

Once the different host areas have been specified, choose a suitable area identifier class for each of them. Specify the classes in the configuration file, one class for each area. In the above example this could be done as:

```
TITLEAREA=se.entra.NetPhantom.server.GofAS400TitleAreaIdentifier
BUTTONAREA=se.entra.NetPhantom.server.GofAS400ButtonAreaIdentifier
MAINAREA=se.entra.NetPhantom.server.GofMainAreaIdentifier
```

Remember that the whole package name for the class must be included, otherwise the class will not be found, and Gui-on-the-Fly will not be started. The order in which the lines are specified does not matter.

3.6 Specify the Settings for Each Area Identifier Class

Once the classes for identifying the areas have been chosen, read the documentation for these classes, and add all the required settings to the configuration file.

To continue the example above, the settings for the GofAS400TitleAreaIdentifier class could be as follows:

```
titlearea_startline=0
titlearea_endline=1
```

The settings for the GofAS400ButtonAreaIdentifier could be:

```
buttonarea_startline=-4
buttonarea_endline=-1
buttonarea_minheight=5
```

And finally, the GofMainAreaIdentifier doesn't have any settings.

3.7 Specify the Order in Which the Areas Should Be Identified

The order in which the different areas should be identified must be specified. Normally the different areas should not overlap, so the order should not matter, as long as the GofMainAreaIdentifier comes as the last area. The GofMainAreaIdentifier should always be the last area, because it takes all the unused area of the screen.

For example, the setting specifying the order in which to identify, should look like:

```
areaorder=TITLEAREA BUTTONAREA MAINAREA
```

3.8 Decide Which Controls Should Be Used on the Panels

Decide if all possible controls should be used on the panels, or if some can be left out. There might be different reasons for not using all control types. Trying to identify every possible control has a performance penalty, especially on screens with a lot of host fields. Unstructured and/or clogged host screens can make it difficult, if not impossible, to identify some controls such as list boxes in a correct way. Remember that it is always better to have a less fancy looking panel that has all its functionality intact, than to have a panel that didn’t build correctly and lost some of its functionality at the same time.

When a decision as to which controls should be included has been made, specify a unique name for each of them in the `ctrlidentifiers` setting. This setting might look like this, except that *it should be specified on a single line*:

```
ctrlidentifiers=OTCTRL EFCTRL PBCTRL TITLECTRL LISTCTRL
LISTCONTMARKCTRL CBCTRL SPINBCTRL HORLINECTRL
```

In this example most of the available controls will be used. The names used have been chosen to have some similarity to the control names, for example OTCTRL means output text control, EFCTRL means entry field control, and so on.

The order in which they are specified does not matter.

3.9 Choose Control Identifier Classes for the Controls

When a decision as to which of the controls should be identified has been made, suitable identification classes should be chosen for them. Specify the classes in the configuration file, one class for each control. In the example this could be done as:

```
OTCTRL=se.entra.NetPhantom.server.GofOutputTextIdentifier
EFCTRL=se.entra.NetPhantom.server.GofEntryFieldIdentifier
PBCTRL=se.entra.NetPhantom.server.GofPushButtonIdentifier
TITLECTRL=se.entra.NetPhantom.server.GofTitleAreaTitleIdentifier
LISTCTRL=se.entra.NetPhantom.server.GofListBoxIdentifier
LISTCONTMARKCTRL=se.entra.NetPhantom.server.GofListContMarkIdentifier
CBCTRL=se.entra.NetPhantom.server.GofComboBoxIdentifier
SPINBCTRL=se.entra.NetPhantom.server.GofSpinButtonIdentifier
HORLINECTRL=se.entra.NetPhantom.server.GofHorLineIdentifier
```

Remember that the whole package name for the class must be included, otherwise the class will not be found, and Gui-on-the-Fly will not be started. The order in which the lines are specified does not matter.

3.10 Specify the Settings for Each Control Identifier Class

Once the classes for identifying the controls have been chosen, read the documentation for these classes, and add all the required settings to the configuration file.

To continue the example, the settings for the GofOutputTextIdentifier class could be as follows:

```
alwaysusestatictext=0
deltrailingpunctmark=1
outputtextlayout=FONTANDCOLOR
```

The settings for the GofEntryFieldIdentifier class could be:

```
entryfieldfiller=_
entryfieldlayout=FONT
```

The settings for the GofPushButtonIdentifier class could be:

```
pushbuttonident=PF#,PF##,F#,F##
pushbuttonequalsign==
pushbuttoncombination=0
pushbuttoncombsign=/
pushbuttonlayout=CLONE
nontemplatebutton=MENU
nontemplatemenu=~Functions
```

The settings for the GofTitleAreaTitleIdentifier class could be:

```
title_title=0,0,-,-,-,-,C
```

The settings for the GofListBoxIdentifier class could be:

```
listboxmincols=2
listboxminlines=2
listboxmaxheaderheight=2
listboxmaxsellength=2
listfieldfiller=_
listboxlayout=FONT COLOR LINES
```

The settings for the GofListContMarkIdentifier class could be:

```
listcontmark=+,More,MORE
listcontpos=-13
```

The settings for the GofComboBoxIdentifier class could be:

```
combolistst art=(
combolistend=)
combolistsep=/
combosearch=pre,post
comboaddmissingtext=1
combofieldfiller=_
```

The settings for the GofSpinButtonIdentifier class could be:

```
spinrangestart=(
spinrangeend=)
spinvaluesep=-
spinsearch=pre,post
spinfieldfiller=_
```

The settings for the GofHorLineIdentifier class could be:

```
horlineminlength=10
horlinechars=-
```

3.11 Specify an Identification Order

The order in which the different areas will be searched for different controls must be specified. To do this, several identification orders will be specified, usually one for each area. Each identification order must have a unique name, which is specified in a blank or comma-separated list after the `identificationorder` setting. In our example, this could look like:

```
identificationorder=TITLEIDENT BUTTONIDENT MAINIDENT
```

Here names that have the areas name included have been chosen, for example TITLEIDENT means the identification order for the title area, and so on.

3.12 Specify Each of the Identification Orders

When the identification orders have been listed, each of them has to be specified in detail. Each identification order must have at least two items: first the name of the area, and then a comma separated list with all the names of the controls that should be identified in this area. The area name and the control list must be separated by colon (‘:’).

The specification of the identification order in our example will look like this, except that the specification for the MAINIDENT *should be on a single line*.

```
TITLEIDENT=TITLEAREA:TITLECTRL
BUTTONIDENT=BUTTONAREA:LISTCONTMARKCTRL,PBCTRL
MAINIDENT=MAINAREA:HORLINECTRL,LISTCONTMARKCTRL,SPINBCTRL,CBCTRL,
LISTCTRL,OTCTRL,EFCTRL
```

In this example, the title area will just be searched for a title control; the button area will just be searched for a list continuation mark and for push buttons, and so on.

3.13 Decide on a Template File for the Look of the Controls

Finally a template file must be specified. Either use the default one or copy it and change the copy until it has the look needed for your application. For more details about template files see Chapter 4.

The path and name of the NetPhantom runtime file that will be used as a template should be specified in the `gofruntime` setting. Note that the path is relative to the server's runtime directory.

The template NetPhantom runtime application must be compiled with the *Backward compatible* option.

This could look like:

```
gofruntime=gofgui/npgoftest/npgoftest.phr
```

3.14 Specify a Section Name for All These Settings

If all the settings that have been specified are completely new, a section name for these settings should be specified. This section name must be inserted before the first setting. This makes it possible to have several different Gui-on-the-Fly settings specified at the same time. The section name must be placed inside brackets, as in the example below:

```
[NpGofTest]
```

3.15 Specify all Section Names for the Gui-on-the-Fly Settings

All the applications that are using Gui-on-the-Fly and the section names used for the Gui-on-the-Fly settings must be specified under the GuiOnTheFly section. The application name must be the same as the one specified in the NetPhantom Administration Client on the Application tab, i.e. under the Application section in server.ini.

The Gui-on-the-Fly section could look like this:

```
[GuiOnTheFly]
APP1=NpGofTest
APP3=NpGofTest
```

In this example, both the APP1 and the APP3 applications will use the settings specified in the NpGofTest section.

3.16 If the Created Panels are Unusable

If the panels that have been created by the Gui-on-the-Fly module are unusable, then there are a few things that should be tested. First, try and get a rough idea of if most of the panels are unusable, or if it is just a few of the panels.

Just a Few of the Panels are Unusable

In this case, these panels are probably a bit too complicated to be created by Gui-on-the-Fly. The best thing to do then is to create panels for the problematic host screens in the NetPhantom Editor.

A Large Part of the Panels are Unusable

If this is the case, then the host application's host screens are either too complicated for the current settings used in Gui-on-the-Fly, or they are just too unstructured and inconsistent, so that it is difficult to find rules that apply to all screens.

The only thing to do then is to reduce the number of control types that should be created by Gui-on-the-Fly. In worst case try to use just output texts and entry fields, this should work for even the most complicated host applications.

4 Using a Template File to Change the Look of the Controls

If you don't like something, change it.
If you can't change it, change your attitude.
Don't complain.

Maya Angelou

It is possible to change the look of the controls by simply changing the corresponding control in a template file.

The runtime file must have at least one panel, onto which the different controls that should be used as templates have been placed. This panel, which is a Main Panel, must have the Panel ID defined as `BASE`.

It is also possible to use a second panel for popup windows. This panel should be defined as a Pop-up panel and must have the Panel ID defined as `POPUP`. By using a separate template panel for popup windows, it is possible to have different look for the controls on a popup window. If the Pop-up panel is missing, the popup windows will use the Main Panel as a template panel.

The extent to which control layout is taken from the template, is determined by the layout setting for each control that is specified in the *gof.ini* configuration file.

The controls that are used as templates all have their ID specified. Do not change the ID, because the identification classes use the ID to find the template control. The control IDs used by the default identification classes are:

```
O           Output text or static text, left aligned.
OR          Output text or static text, right aligned.
ONL         Numeric output text or static text, left aligned.
ONR         Numeric output text or static text, right aligned.
E           Entry field, left aligned.
ER          Entry field, right aligned.
ENL         Numeric entry field, left aligned.
ENR         Numeric entry field, right aligned.
LISTB       List box
COLSEL      Selection field in a list box.
COLL        Left aligned column in a list box.
COLR        Right aligned column in a list box.
HL          Horizontal line.
CMB         Non-editable combination box.
CMBE        Editable combination box.
SB          Non-editable spin button
SBE         Editable spin button.
CHK         Checkbox.
RB          Radio button.
PB_OK       A button corresponding to the Enter-key. Not used by the
            The default push button identifier.
PB_F1       A button corresponding to function key F1 (Help-button).
PB_F3       A button corresponding to function key F3 (Exit-button).
PB_F5       A button corresponding to function key F5
            (Refresh-button).
PB_F6       A button corresponding to function key F6 (Create-button).
PB_F12      A button corresponding to function key F12
            (Cancel-button).
PB_X        A default button, used for buttons when SIZE, FONT or
            SIZEANDFONT layout is used.
```

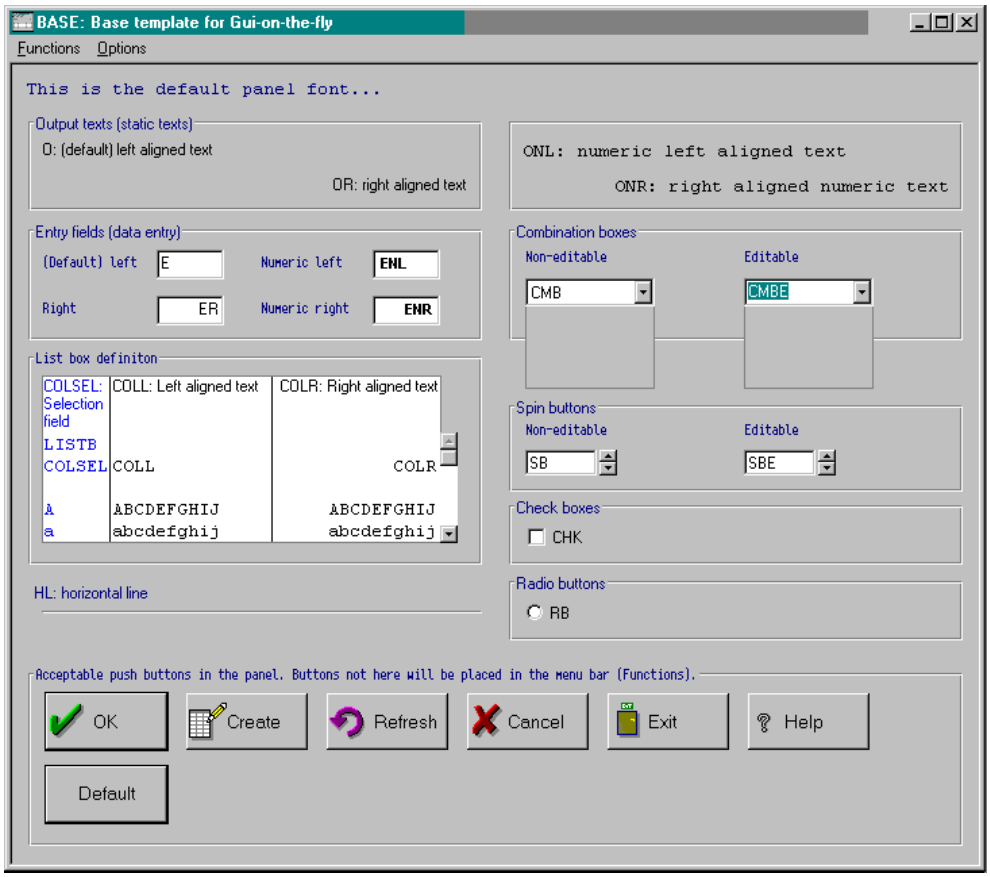


Fig. A typical template panel.

For most of the controls, the only changes that will affect the corresponding controls on the panels created by the Gui-on-the-Fly module are changes to the font and color.

The controls that can take more than the font and color from the template are:

- List box
- Push buttons

These controls are described in detail below.

The List Box in the Template File

In the list box the following settings can be taken from the list box in the template.

The font and color for the selection column’s header

The font and color for the selection column’s cells.

The font and color for the other column’s headers (can be separated for left and right adjusted columns).

The font and color for the other column’s cells (can be separated for left and right adjusted columns).

All the lines in the list box includes the line between columns, the line between headers, the line between the header and the column and the line between rows.

The push buttons in the template file

For the push buttons that exist in the template file, it is possible to either take the font, the size, both the size and font, or in some special cases to clone the button from the template.

When the cloning is used, the generated push button must correspond to a function key that equals the second part of the button's Control ID in the template file. For example, for a button to be cloned from the button with the Control ID `PB_F12`, it must correspond to the F12 function.

5 How to Write Your Own Identification Classes

It is common sense to take a method and try it.
If it fails, admit it frankly and try another.
But above all, try something.

Franklin D. Roosevelt

This section tries to give some hints on how to write your own identification classes, if the need should arise. The classes that can be replaced by your own identification classes can be grouped into three groups, the host field identification class, the host area identification classes and the control identification classes.

5.1 Writing Your Own Host Field Identification Class

To write your own host field identification class, you should extend the `GofHostFieldIdentifierAdapter` class, because this class implements the `GofHostFieldIdentifier` interface and has default methods for all the methods specified in the interface. The methods which can be overridden, are:

- public void identifyHostFields(HostScreen screen)
- public void doIdentifyPartialFields(boolean flag)
- public boolean isPartialFieldsIdentified( )
- public Vector getHostFields(int x, int y, int w, int h)
- public Vector getPopupWindowHostFields(int x, int y, int w, int h)

The `identifyHostFields` method must create the GofHostFields from the HostScreen's HostFields. It is up this method to decide if a HostField should be split up to several GofHostFields. This is where you implement your own algorithm for creating the GofHostFields. The newly created GofHostFields must be stored internally in the class since the method GetHostFields must be able to return them.

The `doIdentifyPartialFields` method must store a flag indicating what should be done with GofHostFields that are partially inside, partially outside the area that should be identified. This flag is only meaningful when a part of the whole screen is identified. This functionality is implemented by the default implementation.

The `isPartialFieldsIdentified` method must return the value of the flag indicating what should be done with GofHostFields that are partially inside, partially outside the area that should be identified. This functionality is implemented by the default implementation.

The `getHostFields` method retrieves the GofHostFields created in the `identifyHostFields` method. It should take in account the value of the flag that was set in the `doIdentifyPartialFields` method. This functionality is implemented by the default implementation.

The `getPopupWindowHostFields` method retrieves a popup window's GofHostFields created in the `identifyHostFields` method. It should ignore the value of the flag that was set in the `doIdentifyPartialFields` method, since this should only be valid for complete screens. This functionality is implemented by the default implementation.

5.2 Writing Your Own Host Area Identification Classes

To write your own host area identification class, you should extend the `GofHostAreaIdentifierAdapter` class, because this class implements the `GofHostAreaIdentifier` interface and has default methods for all the methods specified in the interface. The methods, which can be overridden, are:

- public void getAreaSettings(IniFile confFile, String subsection)
- public Vector identifyArea(Vector gofHostFields, int x, int y, int w, int h)
- public Vector getAreasGofHostFields( )
- public void addControl(PhantomControl phantomControl)
- public void layout( )

The `getAreaSettings` method should get all the required settings for this class from the configuration file.

The `identifyArea` method must implement the code for identifying the area the class is supposed to identify. This is where you implement your algorithm for identifying the area. It should also store the GofHostFields that belong to this area.

The `getAreasGofHostFields` method must return a Vector containing all the GofHostFields belonging to this area. This functionality is implemented by the default implementation.

The `addControl` method adds a new PhantomControl to the area. This method just needs to save a reference to the control, so that it can be accessed during the layout of this area. This functionality is implemented by the default implementation.

The `layout` method should layout the controls belonging to this area according to the layout rule specified by the parameter.

5.3 Writing Your Own Control Identification Classes

To write your own control identification class, you should extend the `GofControlIdentifierAdapter` class, because this class implements the `GofControlIdentifier` interface and has default methods for all the methods specified in the interface. The methods which can be overridden, are:

- public void getControlSettings(IniFile confFile, String subsection)
- public void identifyCtrls(GuiOnTheFlyRuntime gofRuntime,
  GofHostAreaIdentifier areaIdentifier,
  PhantomHostScreen phantomHostScreen,
  HostScreen hostScreen,
  PhantomPanelData newPanel,
  int offsetX,
  int offsetY)
- public VirtualSessionManager getVirtualSessionManager(GuiOnTheFlyRuntime gofRuntime)

The `getControlSettings` method should read all the required control identifier settings from the configuration file.

The `identifyCtrls` method is where the class implementing this interface does all its work, and where you implement your algorithm for identifying the control. This method should implement the following steps:

- Create PhantomControlBase
- Create PhantomHostField
- Create PhantomControl
- Add the control to PhantomPanelData
- Add the control to the GofHostAreaIdentifier it belongs to
- Mark the GofHostField connected to the new control as processed

The `getVirtualSessionManager` method must return a reference to the VirtualSessionManager instance. This functionality is implemented by the default implementation.

5.4 Examples of the Default Classes

To see examples of the source code for the default classes, see the *Source listings* under the menu *Gui-on-the-Fly* in the in the NetPhantom web server's default startup page.