

MAGIC SOFTWARE ENTERPRISES LTD.

Magic eDeveloper

Part of the Magic eBusiness Platform

Developing with
Magic Components

The information in this document is subject to change without prior notice and does not represent a commitment on the part of MSE.

MSE makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose.

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms and conditions of the license agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement.

No part of this document and/or databases may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or information recording and retrieval systems, for any purpose other than the purchaser's personal use, without the prior express written permission of MSE.

All references made to third-party trademarks are for informational purposes only regarding compatibility with the products of Magic Software Enterprises Ltd.

Unless otherwise noted, all names of companies, products, street addresses, and persons contained herein are part of a completely fictitious scenario or scenarios and are designed solely to document the use of Magic.

Magic® is a registered trademark of Magic Software Enterprises Ltd.

PC/TCP® Network Software is a registered trademark of FTP Software Inc.

Microsoft® is a registered trademark, and Windows™, WindowsNT™ and ActiveX™ are trademarks of Microsoft Corp.

Clip art images copyright by Presentation Task Force, a registered trademark of New Vision Technologies Inc.

All other product names are trademarks or registered trademarks of their respective holders.

04 03 02

6 5 4 3 2 1

Copyright © 2002 by Magic Software Enterprises Ltd. All rights reserved.

Contents

1 Components

What is a Component?	4
Why Should I Use Components?	5

2 Magic Components

Magic Components are More Effective	6
How Magic Reveals Objects	7
Magic Component Types	7

3 Creating and Loading Components

Creating an Interface	8
Loading Components into Host Applications	9
Developing with a Component	9

4 Effective Application Components

When Should I Use a Component?	11
--------------------------------------	----

5 Component Runtime Behavior

When Does a Component Load?	13
The Runtime Task Tree	14
Where do Variables Fit In?	16

6 Events and Handlers

How are Events Handled?	17
-------------------------------	----

7 Nested Magic Components

What are Nested Components?	20
-----------------------------------	----

8 Afterword

Components

What are components, and why should we use them?

Components let us develop and deploy applications more rapidly than ever before. A key to assembling applications quickly is the ability to reuse existing application components to meet new application requirements.

What is a Component?



A component is a part of a larger structure that contributes to the composition of the whole structure. The same is true of a software component, which can be defined as an identifiable application that describes and delivers a set of meaningful services using predefined interfaces.

Components are encapsulated and only reveal the services they provide in their interfaces. As a result, the actual implementation of the service within the component is hidden from the outside world.

A component can be an entire business-logic module, such as an accounting module, but a component can also be much smaller in scope, such as checking the validity of an identification number.

When using components, it is important to focus on what a component does and not how it does it. This separates the **What** from the **How**, isolating changes to the component from the applications that use it.

Why Should I Use Components?



Every application developer has the same goal -- to produce the most effective applications in the least possible time. And every developer knows that exploiting the services available in the development environment is essential for achieving that goal. This is where components play a big part.

The heart of component-based development is that business services designed as components are inherently reusable, and easily upgraded and deployed.

A component can be modified and upgraded without the host application because it exists as an independent entity. This makes upgrading, customizing, and maintaining applications a lot easier. For example, if a programming glitch or quirk is found, you just need to fix the faulty component, not the entire application. Only this component needs to be sent to your customers. Your applications no longer become outdated and unable to take advantage of changing situations or new technologies.

Deploying with components makes your application easy to configure. Let's say a customer purchases a component-based package from you and initially decides to purchase only one module. If the customer wants to buy another module in the future, all you have to deploy is the new component. For example, a customer who buys an accounting package with a bookkeeping module and a tax module can decide at any later stage to also purchase a stock-tracking module.

As your component library grows, your ability to customize new applications increases along with it. For example, if at one time the stock-tracking module was only part of the accounting application, it may now also become part of an ecommerce application, significantly reducing the ecommerce application's development time.

In short, developing with components not only means that time-to-market requirements are easily met, but also that you can rapidly respond to changes and meet today's dynamic business requirements.

Magic Components

What are Magic components? How are they different from other components?

A Magic component is a regular Magic application. As a Magic developer, you do not have to learn any new skills to create Magic components. You only need to choose the properties you want the outside world to see. This chapter gives a brief overview of how to do this. For more information, see the *Magic Reference Guide*.

Magic Components are More Effective



Although a Magic component conforms to all the rules of a regular component, it provides much more because it is first and foremost a Magic application. A Magic component can be used simply as a component that provides a small feature of the host application, or it can be a full-scale module in its own right that is sold and used without a host application.

Magic components can also be host applications for other components. In other words, Magic applications can have nested components. Magic components can be either Magic Control Files or Magic Flat Files.

You Can Reveal More

Most components mainly reveal their methods, or programs, Magic lets you reveal:

- Models
- Tables
- Programs
- Helps
- Rights
- Events
- Environment settings

How Magic Reveals Objects

Revealing objects is simple with Magic components. All you have to do is provide a public name for the item you want to reveal. Figure 2-1 below provides an example of a program that will be revealed to the outside world. In this case, the program, *Send to Excel*, will be revealed to the outside world as *SendToExcel*.

Program Repository					
#	Name	Folder	Public Name	Last Update	Time
1	Main Program			08/01/2002	14:44:35
2					
3	Prepare to send data to Excel			07/11/2001	17:23:54
4	Actual conversation with Excel			07/11/2001	17:42:39
5					
6	Send to Excel		SendToExcel	09/01/2002	09:25:34
7	Data display			08/01/2002	14:35:05
8					

Figure 2-1 Example of a Program with a Public Name

In this way, you can turn any Magic application into a component without learning any new skills. .

The Main Program

The main program cannot be given a Public Name and will therefore not be revealed. Only the main program's events can be revealed because they behave as application events.

Magic Component Types

A Magic component does not have to be hosted by a Magic application. Magic components can also be hosted by J2EE applications as EJBs, or they can be Web Service providers, but this is a subject for another paper.

J2EE and Web Services

If the component is created as an EJB or a Web Service, only programs may be revealed.

Creating and Loading Components

How do I create a component interface?

This chapter explains how to create an interface and load a component into a host application. You construct the component interface file using the Magic Component Interface Builder, which you can access from the Magic Tools menu.

Creating an Interface



The Component Interface Builder lets you build an interface to a host application by creating a Magic Component Interface file (MCI), which is actually a pure text file. You can use the Component Interface Builder to reveal properties or to remove previously revealed properties.

The Component Interface Builder stores the components in a database, allowing easy access to create or modify the interface. When the builder is loaded, all objects that have public names are added to the database. In addition, the Component Interface Builder lets you reveal various environment and setup properties, including:

- Servers
- Services
- Databases
- Logical Names
- Environment Settings

The full list of the environment settings that may be revealed can be found in the *Magic Reference Guide*.

The Component Builder also lets you supply a Help file to assist developers who use the component.

When Should I Change the Interface?
 A new interface should be created when:

- An object that has been revealed is deleted
- A new object needs to be revealed

Loading Components into Host Applications



You load components from within the host application using the new Component repository. When you zoom on an empty entry, or request Load/Reload from the menu, Magic lets you select the component you want to load from a list of all the Magic Component Interfaces (MCIs) that have already been created. Once an MCI is selected, Magic loads the corresponding component into the current application, and the properties that the component revealed are now available for use in the application.

Component Repository			
#	Name	Description	Folder
1	Excel connectivity		

Figure 3-1 A Loaded Component Named ExcelConnectivity

Sometimes the host application has to be changed when changes are made to the interface, but this is not always necessary. When new objects are revealed, there is no need to reload the component unless you want to use those new objects.

Developing with a Component

When a component is loaded using the Component repository, all repository items that were revealed are available in the current application. Let us take the example of the *Send to Excel* program on page 7. How do you call that program? Because this program is now an integral part of the application, you can just call the program.

As shown in Figure 3-2 below, the program is called like any other Magic program, with two arguments and a return value. The only difference here is in the name. The name of the program is preceded by the name of the component.

Operation : Handler On Ctrl+F1									
#	Operation	Name	Init:	Arg:	Fmc:	Ret:	Ret:	Ret:	End
1	Select	Virtual	: 1	Return Code	0	0	0	0	Yes
2									
3	Call	Prog	: 5	Excel connectivity.Send to Excel	2	Fmc:	0	Ret: K	Yes
4									
5	Evaluate		: 3	MMSTOP ()				Ret: ???	4

Figure 3-2 Component Usage Example

The same methodology used for programs is valid for all revealed objects.

Setup and Environment Properties

As mentioned on page 6, various Setup properties can be revealed. In the case of Servers, Services, Databases, and Logical Names, an extra line is added to the Magic.ini file of the host application for each reference. These properties are generally used for configuring or localizing the application, and they should be amended to suit the current environment. Except for Logical Names, all the properties are loaded as they were revealed together with all their values. The developer must fine-tune the values. Since the translation values of the Logical Names are on the development machine, the Logical Names will probably not be the same as on the computer where the component is implemented. Magic therefore creates an entry without the translation values, allowing the developer on the host computer to add the correct values.

All of these settings can be accessed from within the host application because the component is now part of the host application.

It is important to remember that if component environment properties that are different from the properties in the current application, such as Date Mode and Start of Century, were revealed, the revealed values will only be valid within a program in the component. These values are generally only relevant during runtime. If a component program is called, these different values will only be valid for this program. When a program in the host uses a table from the component, it will use the environment values of the host application.

Note

Magic accesses objects in the component by Public Name. This means that despite the fact that we call the program *Send to Excel*, we are actually calling *SendToExcel*. If a developer wants to call a different program, the Public Name *SendToExcel* must be removed from the *Send to Excel* program entry and moved to the new program. No change is necessary to the interface.

Important

As stated above, Magic accesses objects in the component by Public Name. If an object is removed from the interface, it may still be called using the Public Name. If you no longer want this object to be called, you must also remove the Public Name.

Effective Application Components

How do I build my applications with components?

You now know what a component is, what a Magic component is, and how to use a Magic component. Now you need to understand how to go about building your applications with components, which will be explained in this chapter.

When Should I Use a Component?



The key to assembling applications rapidly is the ability to reuse existing code, in other words a *write once, use again and again* strategy.

There are no hard and fast rules on what to place in a component or what factors should be taken into account when designing a component-driven application. Nevertheless, we can supply some guidelines and techniques.

If your application has various modules, consider making each one a component. If the module can exist on its own and be sold separately, that module should definitely be created as a component. For example, if your application consists of a Sales module, a Financial module, and a Payroll module, consider making each module a component.

- Put all frequently used models into a component.
- Put all frequently used helps and rights into components.
- Put all frequently used tables, and all frequently used programs based on those tables, into a component.
- Put all frequently used events into components.
- If you need to provide default handlers for the entire application, it is good practice to make components using global handlers, as explained on page 18.
- Look for behavior that exists in more than one place in the system, and create a

utility component that can be used in multiple applications. For example, you can make a program that checks the parity digit of a number into a component.

- Customer customizing – If the application is the same for all customers but there are a few modules, or interfaces, that can be different for each customer, this is a good opportunity to take advantage of components. An example of this would be a customized package. If the main application remains the same but each customer has different reports, the reports can be written in a component.
- If a program or feature is modified more often than others, this may also be a good opportunity to take advantage of components. An example in an accounting application would be when the base remains the same but the taxation module changes periodically.
- Outsourcing – When you outsource part of the application,, it is good practice to make the outsourced module a component.

Let's illustrate this with a simple example using a model. Take a simple case of the use of a serial number. You want this number to be numeric with 9 digits without negative values. The number appears in a few tables, and you want to use it in various programs as a virtual variable. You could define this each time. But if you wanted it to be displayed in a specific format, you would have to remember exactly how you set it up each time. In this case, you would probably choose to use a model. Assuming that each application you write uses that same serial number in the same format, it would make sense to place this model into a component, thereby defining it in one place but having it available for every application.

Limitations

If an object has an interconnected reference to another object, it is not valid to break these into components. This can be explained with an example: If you have created a model for a combo box and the combo box uses a table as its data, revealing the model without the corresponding table is invalid. If you plan to have all models in one component and all tables in a different component, and if you have models reference tables and tables reference models, you will run into difficulties. This is an invalid practice.

Component Runtime Behavior

What happens to a component and all its variables at runtime?

How do components behave at runtime? At what point does a Magic component load? Where do the component variables fit into the environment? These and other runtime issues are discussed in this chapter.

When Does a Component Load?



In the Component repository properties, you can determine when the component will be loaded during runtime. The component can either be loaded together with the host application, as specified in the Immediate setting, or when the first call is made to it, as specified in the On Demand setting. When to load the component depends on the nature of your application and how it is constructed.

When a component loads, its main program loads as well. If operations are performed in the main program, such as variable initialization or memory table initialization, these operations will always be executed. Because the main program is only executed once, whether the component load property is Immediate or On Demand makes a big difference.

- If a component has the load property Immediate and a host application program calls a component application program, the component's main program will not be called again.
- If a component has the On Demand property and a host application program calls a component application program, the component's main program is called because the component is loaded and initialized at the same time. Nevertheless, during subsequent calls to any component program, the component main program will not be called.

It is important to give some thought as to when a component should be loaded. If you have many components and all of them are loaded upon application startup, the initial load may take a long time. All component resources will be immediately available when

the application loads, but a lot of memory will be consumed. If some of those components are rarely accessed or accessed only by certain users, loading them initially may be counter-productive. However, if your components contain models, tables, events, programs, and other items that are used widely and frequently, it makes sense to load them initially.

The Runtime Task Tree

What is the runtime tree? Where does the current task fit in the global task tree?



When programs call other programs or tasks, a task tree is formed in runtime. This tree begins with the main program, which is the first program to run, and continues until the current program. A running task can query certain values of any ancestor task in the tree by using a set of predefined functions. You can direct a function to a specific task in the tree by specifying its generation, a sequential number defined by its location in the task tree. The bottom of the tree, the last task to be called, is generation 0, the parent of that task is generation 1, and so forth.

In a Magic application without components, Program A calls Program B, as shown in Figure 5-1.

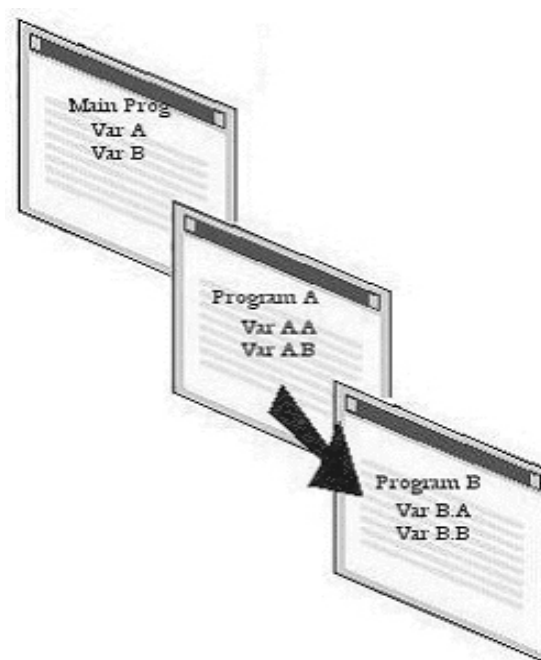


Figure 5-1 Example of a Regular Runtime Task Tree

The runtime task tree in this case is:

Main Program > Program A > Program B

If you query the task tree in Program B, you find that Program B is generation 0;

Program A is generation 1; Main Program is generation 2. What happens when the called program is in a component, as shown in Figure 5-2?

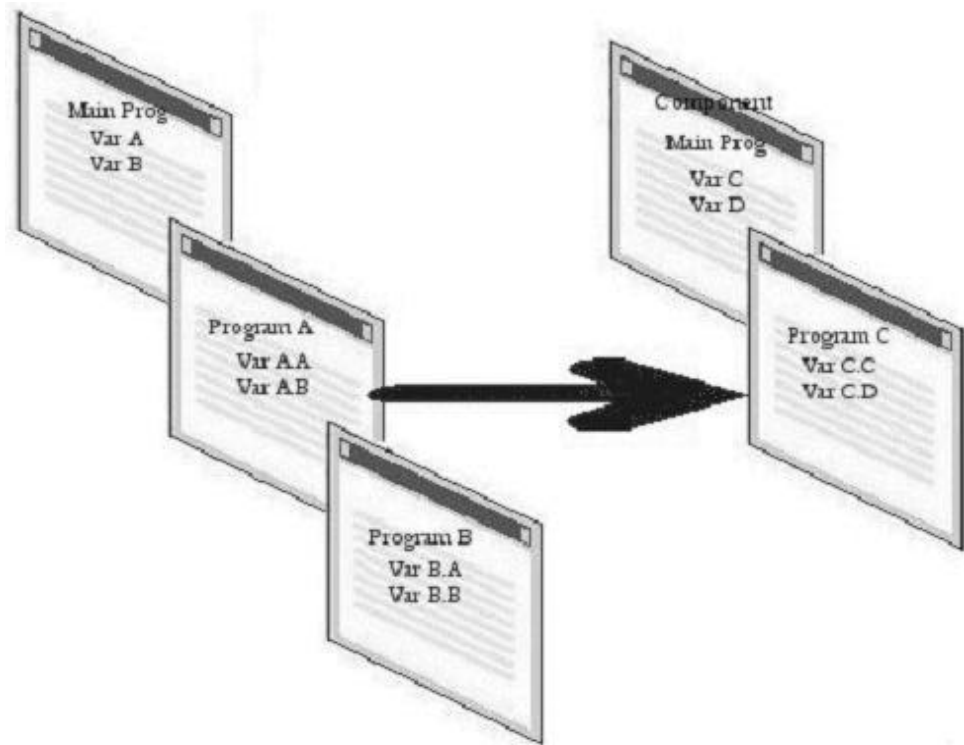


Figure 5-2 Example of a program calling a program in a component

Here the component's main program may complicate matters. How will it fit into the global scope of things? In this case, the runtime task tree is:

Main Program > Program A > Component Main Program > Program C

If you query the task tree in Program C, you find that Program C is generation0; Program A is generation 1; Main Program is generation 2. The main program of the component is not included in the generations.

The Magic functions that deal with task generation are:

CHEIGHT, CLEF, CLEFTMDI, COUNTER, CTOP, CTOPMDI, CURROW, CURRPOSITION, CWIDTH, DBCACHE, EOF, EOP, LASTPARK, LEVEL, LINE, PAGE, ROLLBACK, STAT, VARINP, VIEWMOD, WINBOX, WINHWND.

Where do Variables Fit In?

We often need to access variables from an ancestral program in a called program or task. This is done in a way that is similar to the runtime task tree. The first variable in the main program is 1 or A, and the subsequent variables are numbered sequentially until the current program. Therefore, in the case of a regular Magic application, as shown in Figure 5-1 on page14, the variable tree would be:

A	1	A
B	2	B
A.A	3	C
A.B	4	D
B.A	5	E
B.B	6	F

But what happens when the component is called, as shown in Figure 5-2 on page15? In this case, the component's main program variables are added to the tree before the last program. The variable tree would then be:

A	1	A
B	2	B
A.A	3	C
A.B	4	D
C	5	E
D	6	F
C.C	7	G
C.D	8	H

As we have seen, unlike the generations, the component's main program variables are taken into account. The Magic functions that deal with variable generations are: VARATTR, VARCURR, VARCURRN, VARINDEX, VARINP, VARMOD, VARPIC, VARPREV, VARSET.

Events and Handlers

How are events handled in a component-based application?

The backbone of an event-driven engine is the ability to use and handle events. In this chapter we will talk about how events and handlers fit into a component-based application.

How are Events Handled?



As mentioned earlier, only the main program's events can be revealed for the use of host applications. The revealed events then become part of the host application in the same way that a revealed program becomes a part of the application. Component events are handled in the Runtime task tree in the same way as regular events are handled.

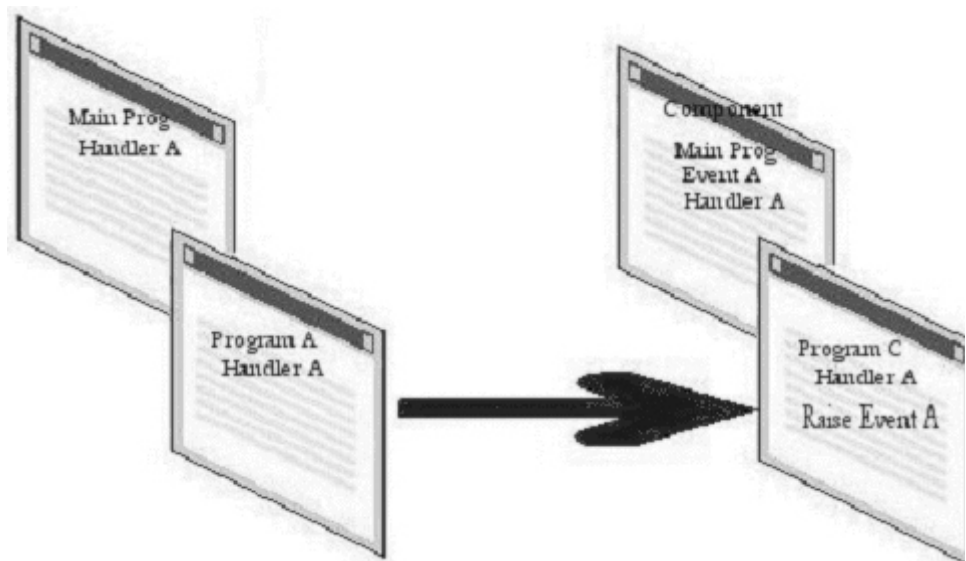


Figure 6-1 A component program raises an event

As an example of the way an event is handled, let's take a look at Figure 6-1. Here, a component program raises an event, which is handled along the task tree. In other words, a handler for this event will first be searched in Program C, then in the main program of the component, then in Program A, which called Program C, and finally in the host main program. When there is a handler in each program, as in the current example, the handler will be handled accordingly along the task tree. If the handler propagate property is set to No in any of the programs, that handler will be the last handler for that event in the task tree.

If Program A were to raise the event, a handler would only be searched in the host application, in other words in Program A and in the host main program.

More information about the way events are handled may be found in the MSE White Paper on Event-Driven Architecture.

Global Handlers

If a program in the host calls a component event and the handler for the event is in the main program of the component, the event will not be handled because the handler will not be in the current runtime task tree. However, if the handler has a global scope, the event will be handled, as shown in Figure 6-2.

Task:1 - Main Program							
#	Level	Event	Details	Scope	Prop	Enable	Oper
1	Task	Prefix					0 ▲
2	Task	Suffix					0
3	Record	Main					0
4	Handler	EventA	Ctrl:	Global	No	<input checked="" type="checkbox"/>	1

Figure 6-2 Global Handler

The question is, when will the global handler be handled? Let's use the previous example, but this time we will substitute a global handler for the event handler in the main program of the component.

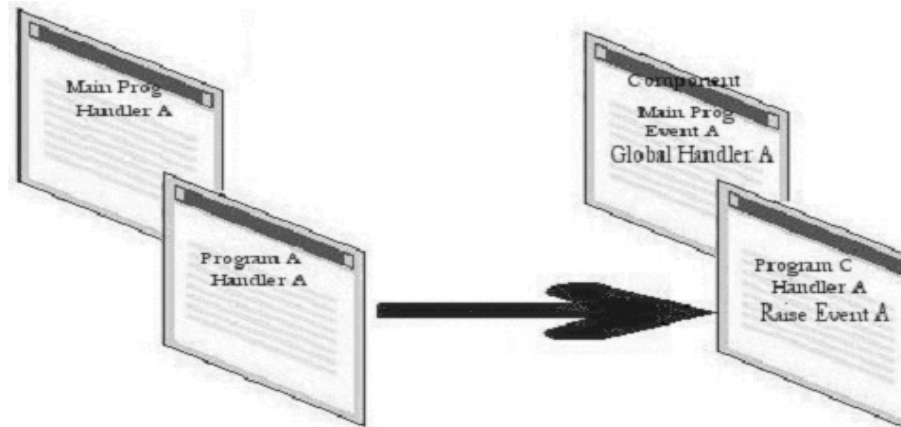


Figure 6-3 Global Handlers

Once again, Program C raises the event, but the search for a handler, and the propagate sequence, is slightly different. The sequence is:

- Program C
- Program A
- Host Main Program
- Component Global Handler

As you can see, the global handlers are handled last in the runtime tree.

The main advantage of a global handler is to provide event default behavior for the entire application.

Important

A global handler may be defined for an event that has not been defined in this component. You can even create a global handler for an Internal or a System event.

Nested Magic Components

How are nested components handled?

Using the Magic component methodology described in this paper, it is not inconceivable that you will have a system of components within components, known as nested components, in your application. This chapter explains how to work with nested components.

What are Nested Components?



You may have a component that is used in an application. Then you may use the application that contains the original component as another component within another application. Or, you may have a component that contains all of your frequently used models, tables, and programs, and then you create a module using those revealed objects. Next you create a package that uses that module. In both of the cases above, you have created nested component systems. Figure 7-1 below displays a more complex nested component system.

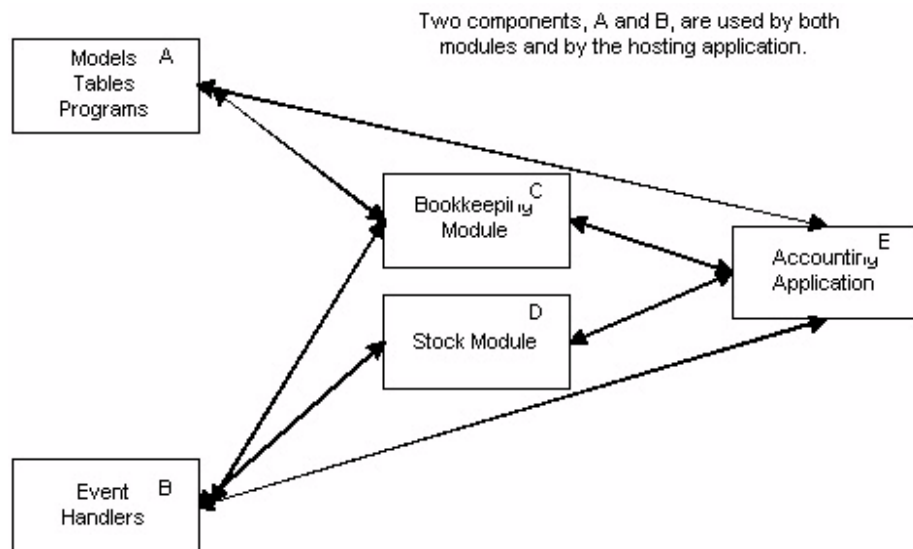


Figure 7-1 An example of a nested component system

In this example, we have Component A, with frequently used models, tables and programs, and Component B, with frequently used events and global handlers. These two components are used in the Bookkeeping Module C and the Stock Module D. These two modules are used as components of the Accounting Application E, which also uses components A and B. In this case the same component is used three times and, in theory, would therefore need to be loaded three times, but this is not what really happens. The Magic engine is smart and knows when a component has been loaded into memory and does not need to be loaded again for each instance.

Note

Magic recognizes a component by its full name. If the component is loaded as `c:\Magic\comp.mcf` in one container or host and is loaded as `\\myserver\Magic\comp.mcf` in another container, Magic will identify these as the names of two different components even though they may point to the same component. The same is true if two different databases are used.

Limitation

You cannot reveal objects from a nested component. In other words, a host application cannot reveal objects from one of its components.

Recursive

Recursive components generate a runtime error. This may happen when you have a Host Application A, which contains a Component B, which contains a Component A that is identical to the host application. .

Chapter

8

Afterword

A few words before you begin using this new style of Magic programming.

We hope this paper will help you get started with component-based Magic programming. You should now be more familiar with Magic component terminology, the limitations and the advantages of component-based programming, how to create Magic components, how Magic handles the component objects, and how components behave at runtime.

We are confident that Magic eDeveloper's component-based programming benefits will help you create more sophisticated and robust applications in a shorter period of time, thereby bring you more RROI – Rapid Return on Investment.