HUMAN-COMPUTER INTERACTION ISSUES FOR THE CONFIGURATION OF COMPONENT-BASED SOFTWARE SYSTEMS

John Grundy and John Hosking

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
email: {john-g, john}@cs.auckland.ac.nz
http://www.cs.auckland.ac.nz/~john-g/

HUMAN-COMPUTER INTERACTION ISSUES FOR THE CONFIGURATION OF COMPONENT-BASED SOFTWARE SYSTEMS

ABSTRACT

Providing appropriate support for end-user configuration of component-based software systems is difficult. We discuss some of the key HCI issues we have identified to do with supporting this, and illustrate a variety of end-user configuration support approaches from a Collaborative Information System we have been developing. These range from visual structural composition and event handling specification to detailed parameterisation and semi-automated guidance. Novel approaches for understanding and retrieving components are also presented. We briefly discuss our approach to designing and implementing such component-based systems configuration facilities.

KEY WORDS

Component-based software, end user computing, component configuration and understanding, component repositories

1 INTRODUCTION

Component-based software systems, or "componentware", have become popular approaches to supporting more dynamic systems development (Sessions, 1998; Szyperski, 1997). Key concepts of using software components to develop applications are system development via composition from reusable parts and supporting end users in substantially extending and reconfiguring applications. An important issue is how developers can engineer components and component-based systems so that they effectively support such end user composition and configuration (Mehandjiev and Bottaci, 1998; Szyperski, 1997).

Some of the key HCI issues in supporting configuration of component-based systems we have identified include:

- providing appropriate metaphors and interfaces to support the structural configuration of components
- providing appropriate support for behavioural configuration of components
- providing assistance at appropriate levels of detail to aid understanding of how components can be configured
- providing assistance for locating appropriate components to reuse

In component-based systems we have developed we needed to provide a range of component configuration facilities. These include automated configuration, semi-automated configuration via wizards, parameter setting and high-level, visual component composition and behavioural specification, and low-level scripting. Additional support facilities we have developed provide users with high-level information about facilities components provide and require, and support for configuration validation. We have prototyped a component repository to assist users in storing and retrieving components for reuse. All of these facilities have a variety of HCI issues explored in this paper. We first introduce an example Collaborative Information System and use it to motivate this research. The following sections discuss component composition, behaviour configuration, component understanding and component retrieval. We conclude with a brief discussion of the architecture and implementation approaches we use to support component configuration.

2 MAIN ISSUES

Figure 1 (a) shows a screen dump from a collaborative travel itinerary planner in use, and Figure 1 (b) some of the components composed to produce this application, many of which have been reused from other applications (Grundy et al, 1998a). This itinerary planner supports collaborative, distributed planning of trips using a variety of interfaces including structured itinerary item editing (1), graphical visualisation (2), collaborative text chats (3), and web browsing. Users need to be able to configure the application. This includes adding notification agents to detect when others have modified the itinerary, integrating existing 3rd party tools (e.g. databases, spreadsheets and word processors), specifying task automation agents that support semi-automatic itinerary update or tool data exchange, and reconfiguring environment tools. These are supported using a variety of interfaces, including property sheets (4), visual structure and event specification tools (5), interactive wizards (6) and automatic configuration.

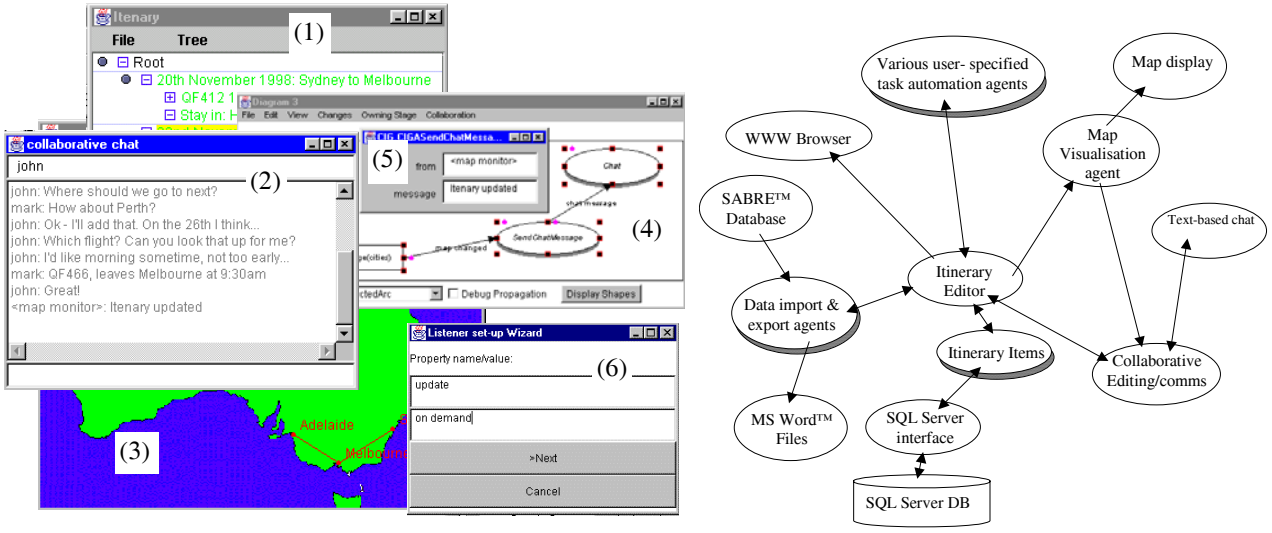

Figure 1: (a) An example component-based application. (b) Component structure of this application.

From both users' and software developers' perspectives a range of complex HCI issues present themselves when considering how to support users extend such systems:

- *Component composition*. Users need to add and remove components from their applications, and change the structural relationships between components. For example, in Figure 1 (b) a user may wish to add a new tool integration or data exchange component e.g. to export an itinerary to an email tool or spreadsheet.
- *Component behaviour configuration*. Users need to configure components to behave in various ways. This may range from simple parameter changes to complex programming of component event handling. For example, the timing of when a new tool integration agent exports data from the itinerary database may need to be specified.
- *Understanding Components*. Many software components provide a simple range of facilities and configuration options, but others may be very complex. Users need to be able to understand what components can do and how they can and should be configured, both structurally and behaviourally. For example, a user needs to understand what components a tool integration component may be connected to and how to set its configuration parameters.
- *Component Location*. Users need support for locating and reusing software components, and this need increases as the available collection of components grows. For example, when extending an application a user needs support for locating and reusing appropriate tool, task automation, notification and tool interface components.

The following sections address these HCI issues in further detail, including examples of our approaches to supporting them, and review other approaches.

3 COMPONENT COMPOSITION

The most fundamental support needed for configuring component-based systems is the ability to add, remove and connect software components. The key HCI issues here relate to providing users with appropriate metaphors for visualising software components and creating links between them. Components with their own user interface can often be "dropped into" the interface of other components using a containment metaphor. Abstract icon-glue representations of components, perhaps employing various iconic forms to distinguish different kinds of components, can be used to provide a more flexible component inter-connection mechanism.

Common approaches to this problem include drag-and-drop component composition using containment, as supported by OpenDoc (Apple Computer Inc, 1995) and Visual Javascript (Netscape Corp, 1997). These typically use the concrete appearance of a component via its user interface to support interactive composition. Other approaches include abstract visualisation of component objects and connection using annotated lines. Systems employing this approach include those of Wagner et al (1996) and ClockWorks (Graham et al, 1996).

Figure 2 shows examples of component composition by drag-and-drop and linking abstract representations of components from our collaborative itinerary system. A chat component can be created by selecting a tool bar icon, then its user interface panel dragged onto an itinerary editor component resulting in the chat being added as a sub-component of the itinerary editor's container (1). The linking of the chat component object to the itinerary editor component object as a sub-component is automated by this interactive composition. Users can also connect components using an abstract representation of an inter-component reference using drag-and-drop creation (2). Unlike the abstract representations of components used by many other systems, we provide several different iconic forms for components and connections to give users a richer representation (Grundy et al, 1998b). These include components that

are data stores, tools, event filters and event actioners. Connector representations distinguish structural, event passing and operation invocation relationships. In the manipulation of both concrete and abstract representations, the user is assisted by feedback from the tool indicating components it is possible to create and components it is possible to link to. Invalid configuration feedback is provided via highlighting and/or error messages in a message bar.

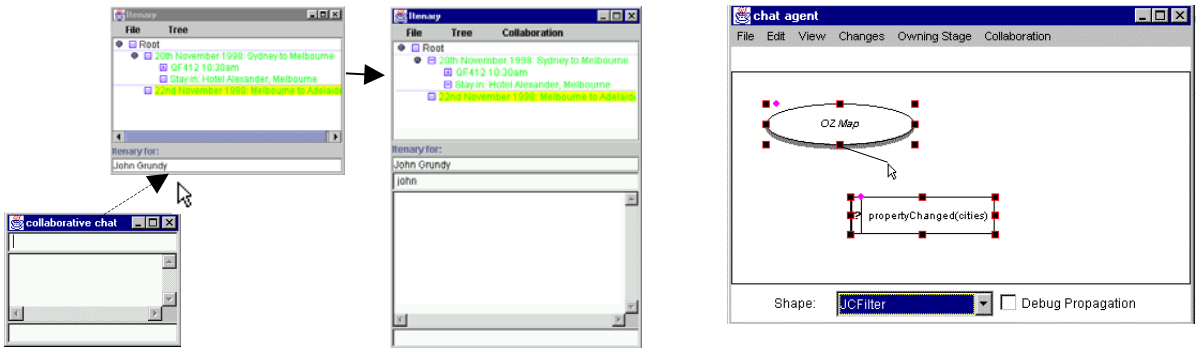

Figure 2: (a) Configuration using concrete representation (b) Configuration using abstract representation.

The advantage of visual approaches to component composition is that they tend to suit the user's mental model of composing a system from parts. Concrete component interface composition is particularly good in this respect. Creating and linking abstract representations is a slightly less direct metaphor, but has the advantage of greater flexibility and that components can be created and linked without needing a concrete user interface representation. A disadvantage of these approaches is that they tend not to support specification of dynamic structures i.e. structures that are changed programmatically. Some effort also needs to go into building such interfaces, and careful design of component representations is needed.

4 COMPONENT BEHAVIOUR CONFIGURATION

Composition of components supports the creation, linking and (if required) removal of components from a system. Some behavioural configuration of components can be automated by detecting component link creation or destruction and performing appropriate configuration functionality. For example, some component characteristics, such as colour and editing settings, can be defaulted from the container or component(s) to which they are linked. However, users also often need support for configuring component behaviour manually. Key HCI issues in configuring component behaviour relate to the level of detail of component behaviour the users must work with, the way they specify behaviour (declaratively vs. operationally), and the cognitive complexity of the behaviour specification interface.

Common approaches to specifying component behaviour include parameter setting, used in most systems, interactive configuration wizards and scripting (Sessions, 1998; Apple Computer Inc, 1995; O'Neil and Schildt, 1998). We have used all of these forms of behaviour specification, along with a novel visual scripting tool.

Generally the most abstract mechanism is the configuration wizard. An example of a simple notification component configuration wizard is shown in Figure 3. This allows users of our collaborative itinerary planner to specify a variety of event-based notification mechanisms, without needing any detailed knowledge of component inter-connection nor behaviour specification. The wizard interface is a set of question/answer dialogues with the user which configures a notification component based on user responses. The user is given a set of options to choose from, and can request more details about possible choices. These are used to create appropriate components to detect filter and action simple events other components might produce.

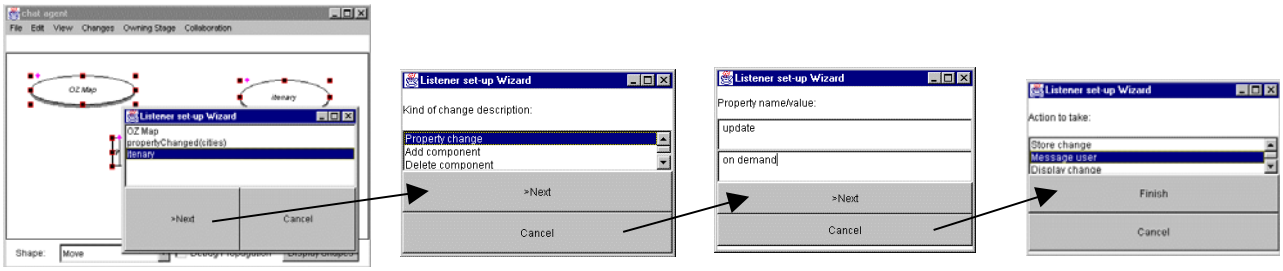

Figure 3: Wizard-based configuration assistance.

Wizards are an excellent tool for novice users of component-based systems, as they tend to hide away most details about components, component inter-connections and parameterisation. They also provide a clearly directed set of steps to perform configuration, with the ability to supply on demand more information and possibly different interfaces to

users. Wizards are by nature, however, inflexible, only supporting the configuration steps with which they are programmed. They may also provide inappropriate interfaces and over-simplified steps for more expert application users, and take considerable effort to program.

Parameterisation of software components is probably the most common form of behaviour configuration. Users set various component properties that the component uses, when necessary, to determine what it should do. Usually component properties hold simple values e.g. true or false, string literals or value ranges. A few systems allow users to supply expressions over values to configure components (Grundy et al, 1998b; Netscape Corp., 1997; IBM Corp., 1998).

Parameter setting is usually done via dialogues, often containing a list of name/value pairs. Figure 4 shows some examples of such "property sheets" from the map (back) and chat message sending agent (front) components in our itinerary planning system. Some of our components can also collect the properties of other components related to them and display a single dialogue containing multiple component parameters. We have found that composition of property sheets provides an improved interface for users where closely related components need to be concurrently configured. Some of our components also support the showing and hiding of a selection of properties based on user preferences e.g. novice users are not shown properties relating to complex component behaviour configuration. For example, the map component does not show the bottom two properties if the user has indicated they are a "novice".

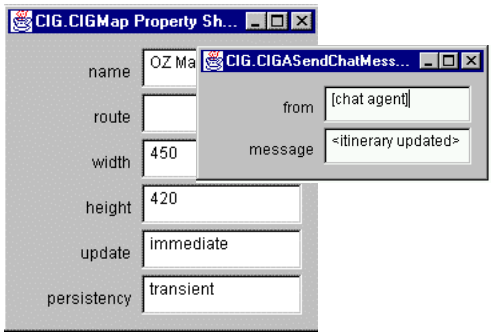

Figure 4: Component configuration properties.

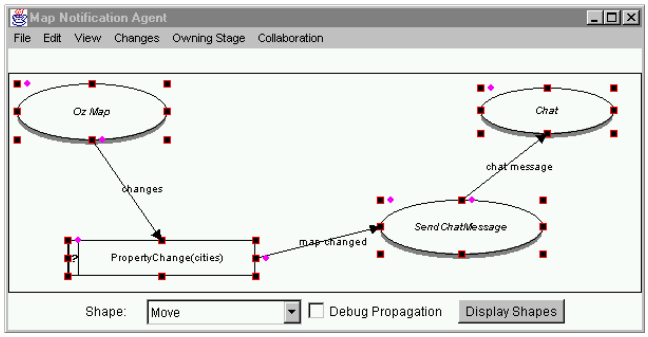

Figure 5: Configuring event-handling behaviour visually.

Property sheets provide a good access point for component parameters, and can sometimes be tailored so that they collect multiple component values or adapt to different kinds of users. They can also be used to provide context-dependent guidance to users and constrain parameter values using appropriate user interface elements. They do tend to support only limited kinds of behaviour tailoring, however, and care must be taken when combining or filtering component parameter lists that users are not confused and have access to all the values they may need to set. Understanding how multiple properties inter-relate, especially if they belong to different components, can be difficult.

Scripting, or programming, of components to control their behaviour gives the most flexible, but lowest-level of configuration control. We have developed a visual scripting tool that allows users to specify a variety of component event handling and some operation invocation (Grundy et al, 1998b). Figure 5 shows an example of a user building a notification agent that detects a map component property update and notifies all users of this via the collaborative text chat tool. Users create and compose components using abstract representations, but also specify the possible flow of events, filtering of events and actioning of events. A large collection of reusable event filter and action components supports the specification of such software agents for task automation, notification and tool integration.

Our visual event handling tool allows users to have more control over component event propagation and handling than wizards and parameter setting typically allow. It is also a higher level interface than the other common scripting approaches. It does require users to have an understanding of the events a component generates and responds to, and sometimes some of the operations and operation parameters a component interface supports. Users also need assistance from the tool to ensure valid configurations are specified, and it is often difficult to provide this for dynamic behaviour.

5 UNDERSTANDING COMPONENTS

In order to effectively compose and configure the behaviour of software components, users must be able to understand their purpose, facilities they provide and require, their parameters, and sometimes their events, event responses and operation interfaces (Grundy, 1999; Morch, 1998; Weyuker, 1998). Ideally such information needs to be presented in an appropriate manner and needs to match the user's level of expertise and context in which it is being requested. Most component-based systems do not provide such information at high enough levels of abstraction, especially when trying to characterise component service provision and requirements. Examples include JavaBeans BeanInfo objects (O'Neil and Schildt, 1998), COM type libraries (Sessions, 1998), and CORBA IDLs (Mowbray and Ruh, 1997).

We have been developing an approach to characterising provided and required services of components that we call aspects (Grundy et al, 1998a and 1999). These aspects are categorised into the systemic features of a system a component contributes to or requires from other components. Examples include user interface facilities, collaborative work support, data persistency and distribution mechanisms, component configuration details and component inter-relationships. We have been developing user interfaces to allow users to access aspect information when configuring component-based systems. Figure 6 shows an example of the aspects of the chat message sender component (1), details about the relationships it has with other components (2) and its parameters (3). Other information about this component that the user can access include its user interface (only a property sheet for configuration), and its low-level properties, methods and events. Some components have other characteristics such as their support for or need for collaborative work, security management, persistency or distribution services. Some basic validation support is also included, where the user can ask that the required or recommended configuration be checked and feedback given on what they need to do to correctly configure the component (4).

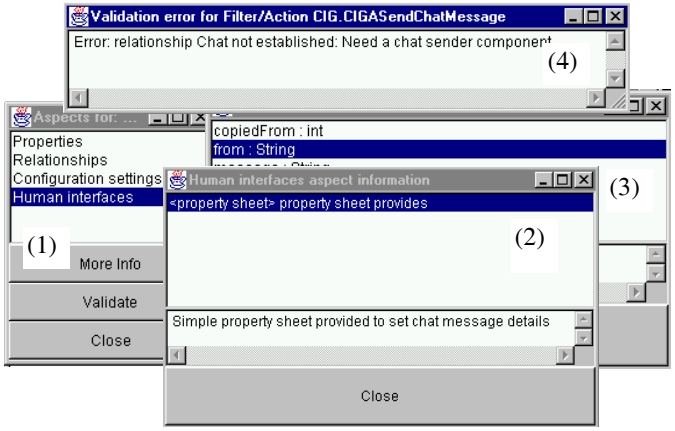

Figure 6: Using information about components.

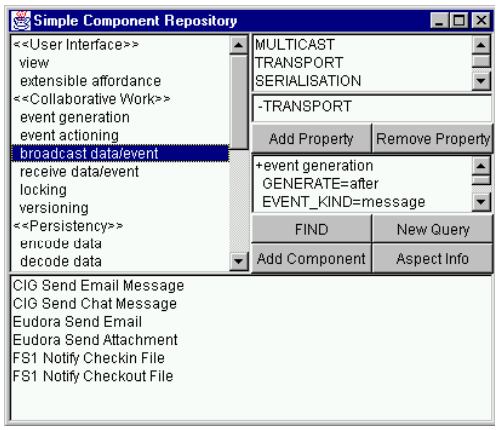

Figure 7: Locating components for reuse.

6 COMPONENT LOCATION

Users need support for finding appropriate components to reuse. Key HCI issues need to address the formulation of queries for components, visualising located components matching a user's query, adding located components to a component-based application and adding components to the repository or changing their indexing parameters. Most component repositories use categorisation, natural language or sematics-based indexing and querying (Park and Lai, 1996; Henninger, 1996; Mareek et al, 1991).

We have developed a prototype repository using our aspects categorisation of component provided and required services. Users formulate queries based on the provided and/or required aspect details the component they are wanting to reuse should support. Detailed property values or value ranges can also be specified to refine a query. A list of matching components is presented with the option to reformulate and/or refine the query, inspect the component aspect information, or add the component to the user's application. Figure 7 shows an example of using this repository while specifying the notification agent of Figure 5. The user is searching for a component that generates collaborative message events that are transported by another component. Such a component should provide collaborative event generation for "message" events, and require another component to do the message transport.

7 DESIGN AND IMPLEMENTATION ISSUES

In order to support the various component-based system configuration facilities illustrated above we have enhanced a component-based software architecture and implementation framework to add aspect information (Grundy, 1999). Configuration aspects are used by wizards, property sheet components and aspect validation functions to determine properties and relationships of a component that can be configured and to supply information about configuring them. Our visual component composition and event handling specification tools use relationship aspects to control inter-component connection, to automate some basic component parameterisation and linking and to support user access to aspect information. We have extended a CASE tool, JComposer, to support specification of component aspects, including JViews wrappers for 3rd party tools and components, using a visual formalism and dialogues (Grundy, 1999).

8 SUMMARY

There are many challenging HCI issues to do with supporting configuration of component-based, distributed, collaborative software systems. Examples include supporting component composition, behaviour configuration, access to information about component provided and required services, and support for locating components to reuse. We have addressed these requirements in component-based systems we have been developing using a variety of approaches ranging from property sheets and wizards, visual composition and event specification tools to aspect-based information dialogues and storage and querying of a component repository.

We are working on better characterisation of component aspects, including more formal functional and non-functional requirements capture and documentation. A more visual approach to querying our repository and especially visualising results is desired. Querying for multiple, related components at once and visualising query matches is an important enhancement we are working on. Improved automatic generation of wizard and property sheet-based configuration tools is desired. Refinements of our visual composition and event handling tools may include visual abstractions and editing mechanisms, combined with presenting aspect information and repository queries and results "in-situ".

ACKNOWLEDGEMENTS

Support for this research from the New Zealand Public Good Science Fund is gratefully acknowledged.

REFERENCES

1. Apple Computer Inc. (1995). *OpenDoc Users Manual*.
2. Graham, T.C.N., Morton, C.A. and Urnes, T. (1996). ClockWorks: Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages and Computing*, Academic Press, 175-196.
3. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. (1998). Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November 1998, IEEE CS Press.
4. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. (1998). An architecture for decentralised process modelling and enactment, *IEEE Internet Computing,* **2:5**, September/October 1998.
5. Grundy, J.C. (1999). Supporting aspect-oriented component-based systems engineering, In *Proceedings of 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 388-395.
6. Henninger, S. (1996). Supporting the Construction and Evolution of Component Repositories, In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, IEEE CS Press, pp. 279-288.
7. IBM Corp., *VisualAge™ for Java*, http://www.software.ibm.com/ad/vajava/, 22nd July 1999.
8. Netscape Corp., Visual JavaScript Developer's Guide, http://www.netscape.com/, 22nd July 1999.
9. Mareek, Y., Berry, D., and Kaiser, G. (1991). An information retrieval approach for automatically constructing software libraries, *IEEE Transactions on Software Engineering*, **17:8**, 800-813.
10. Mehandjiev, N. and Bottaci, L. (1998): The place of user enhanceability in user-oriented software development, *Journal of End User Computing*, **10:2**, 4-14.
11. Morch, A. Tailoring tools for system development*, Journal of End User Computing,* **10:2**, 1998, pp. 22-29.
12. Mowbray, T.J., Ruh, W.A. (1997). *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley.
13. O'Neil, J. and Schildt, H. (1998). *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill.
14. Park, Y. and Bai, P. (1996). Retrieving software components by execution, In *Proeedings of the. 1st Component Users Conference,* Munich, July 14-18, SIGS Books, pp. 39-48.
15. Sessions, R. (1998). *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons.
16. Szyperski, C.A. (1997). *Component Software: Beyond Object-oriented Programming*, Addison-Wesley.
17. Wagner, B., Sluijmers, I., Eichelberg, D., and Ackerman, P. (1996). Black-box Reuse within Frameworks Based on Visual Programming, In *Proceedings of the. 1st Component Users Conference,* Munich, July 14-18, SIGS Books, pp. 57-66.
18. Weyuker, E.J. (1998). Testing Component-based Software: A Cautionary Tale, *IEEE Software*, 54-59.