Storage and retrieval of Software Components using Aspects

John Grundy
Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
john-g@cs.auckland.ac.nz

Abstract

While component-based software engineering technologies have become popular, finding and reusing appropriate software components is often challenging. We describe a software component repository that uses a concept of component "aspects" to index and query components based on their high-level systemic characteristics, including their user interface, persistency, distribution, security and collaborative work support. Software components are queried for aspects of a system they provide or require and these are used to automatically generate a high-level indexing system. Developers and end users can formulate high-level, aspect-based queries to retrieve components providing or requiring services appropriate to their needs.

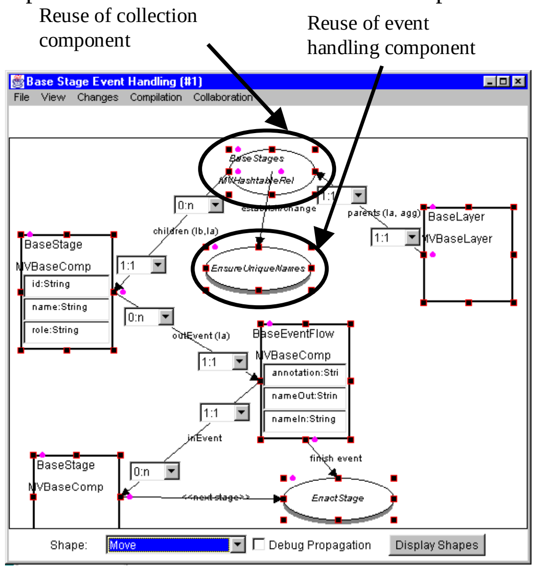
1. Introduction

Component-based software engineering technologies have become popular [14]. Examples of component-based architectures include OpenDoc [1], CORBA [18], DCOM [25], and JavaBeans [20]. Many tools have been developed to assist developers in constructing systems using these architectures, including JBuilder [2], VisualAge [9], and JComposer [3]. Tools to assist end users to configure and extend applications have also been developed, including Visual JavaScript [19], MET++ [26], and Serendipity-II [5].

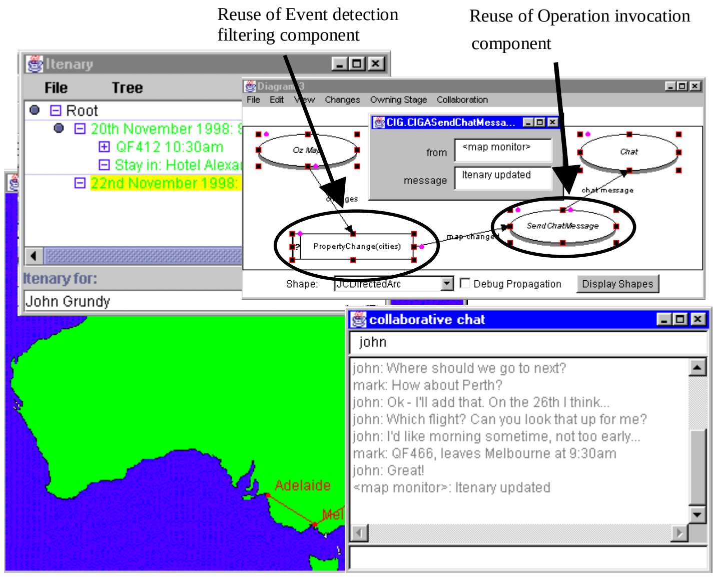
Component-based systems emphasise appropriate reuse and composition of software components as a key concept. However, finding and reusing appropriate software components is often very challenging, particularly when faced with a large collection of components and little documentation about how they can and should be used [15, 27]. This is a particular issue for end users of component-based systems who want to tailor and extend their environment, but have limited understanding of component functionality and implementation [8, 16, 19]. Many software component repositories have been developed, often extending the approaches used for software libraries. Examples include WiSeR [21], IBROW [17], and CodeFinder [8]. Key deficiencies of existing approaches include the need to use low-level, service-based queries, lack of high-level description of component capabilities, lack of validation or checking of retrieved component suitability, and lack of use of the context for which queries are being performed by the retrieval tool. Repositories using formal specification or execution-based retrieval mechanisms suffer from a need to exhaustively, formally specify parts of component services, with queries requiring formal specification techniques that may be difficult for many end users and developers to use.

We describe a software component repository that uses a concept of component "aspects" to index and query components based on their high-level systemic characteristics. These aspects describe a component's provided or required services and related non-functional constraints for capabilities like user interface provision, distribution and persistency management, collaborative work support, security and transaction processing, and configuration and inter-component relationship management. All of the software components in our component-based architecture advertise such high-level capabilities using aspects, and this is used to automate indexing of components. Our retrieval tool allows users to formulate high-level queries about component capabilities and takes account of the context in which a query is performed to assist query formulation. Component aspects provide validation functions to ensure sensible configurations result from using retrieved components.

Section 2 outlines the motivation for this work from two example systems: a CASE tool and workflow system software agent specification tool. Section 3 describes the concept of component aspects and our aspect-based component repository. Section 4 illustrates how component aspects are used to automate indexing of software components. Section 5 illustrates the use of our prototype querying tool to retrieve components and add them to component-based systems. Section 6 compares our approach with other component repositories and outlines areas for possible future research.



(a) Developing a component-based system.



john: Where should we go to next?
mark: How about Perth?
john: Ok - I'll add that. On the 26th I think...
john: Which flight? Can you look that up for me?
john: I'd like morning sometime, not too early...
mark: QF466, leaves Melbourne at 9:30am
john: Great!
<map monitor>: Itenary updated

(b) End user specifying a component-based software agent.

Figure 1. Examples of software developer and end-user component reuse.

2. Motivation

We have been developing various component-based systems, including CASE tools, Workflow Management Systems, and Collaborative Information Systems [3, 5, 4] When developing or extending such systems, developers and end users need to reuse software components from a component-based framework or developed for use on another project [3, 5].

Figure 1 (a) illustrates some reused software components in the JComposer CASE tool [3]. The developer has reused a collection management component (MVHashtableRel), and an event handling component that enforces unique keying for any type of collection component (EnsureUniqueNames). The developer needs to know where to find such components, what their capabilities are, and whether reusing them in the way shown is sensible to achieve their desired goals. They also want to be able to add newly developed components into the repository for future reuse.

Figure 1 (b) shows a software agent specification from the Serendipity-II workflow management system, being used to develop a collaborative Information System for travel itinerary planning [4]. This example shows a simple automated notfication agent which informs all users via a chat tool when the map visualisation has been updated. The end user of this system wants to build (usually) simple task automation agents to enhance their environment, but needs to reuse carefully packaged components in a compose and link way. End users (and often software developers) do not want to query for reusable components based on low-level operations and attributes, but on higher-level capabilities that fit in with the agent task they are trying to develop. Some key issues we have identified for a component repository include:

- Indexing should, as far as possible, be automatic, rather than require exhaustive developer and/or end user input. This ensures consistent, complete indexing by querying each component for information about itself.
- A high-level characterisation of capabilities should be used to index software components, rather than only operation and attribute-level names, types, formalism specfications or natural language comments. This is because, in our experience, multiple operations and attributes usually contribute to the determination of which components a user might wish to find and reuse.
- Querying will normally want to use the same, high-level language to describe component capabilities as indexing. Developers and end users usually want to query for abstract service provision and requirements rather than low-level interface characteristics.
- If possible, the context in which a component is to be reused should be used by the retrieval tool. If a component, or set of components, is required that provide a set of services for a component already identified, the requirements of this existing component can guide query formulation.
- Automatic component configuration and validation functions should be run by the retrieval tool to

ensure a component is appropriately initialised and validated for the context in which it is being reused.

3. Aspect-oriented Component Repository

We have been developing a methodology for the engineering of component-based systems called "Aspect-oriented Component Engineering" [6]. This uses a concept of high-level, systemic aspects of a software application to characterise component provided and required services. This includes the user interface-related services a component provides or requires from other components, component distribution and persistency management, collaborative work and user configuration capabilities, and security and transaction processing models. Each kind of aspect has a number of provided and required "aspect details", with each aspect detail having properties further characterising it (e.g. kinds of persistency or user interface elements, measurements of transaction processing or distributed systems performance, or constraints on usage of an aspect detail, and so on).

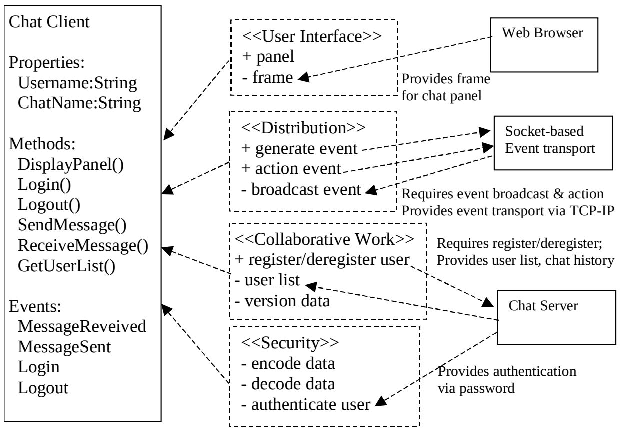


Figure 2. Concept of component aspects.

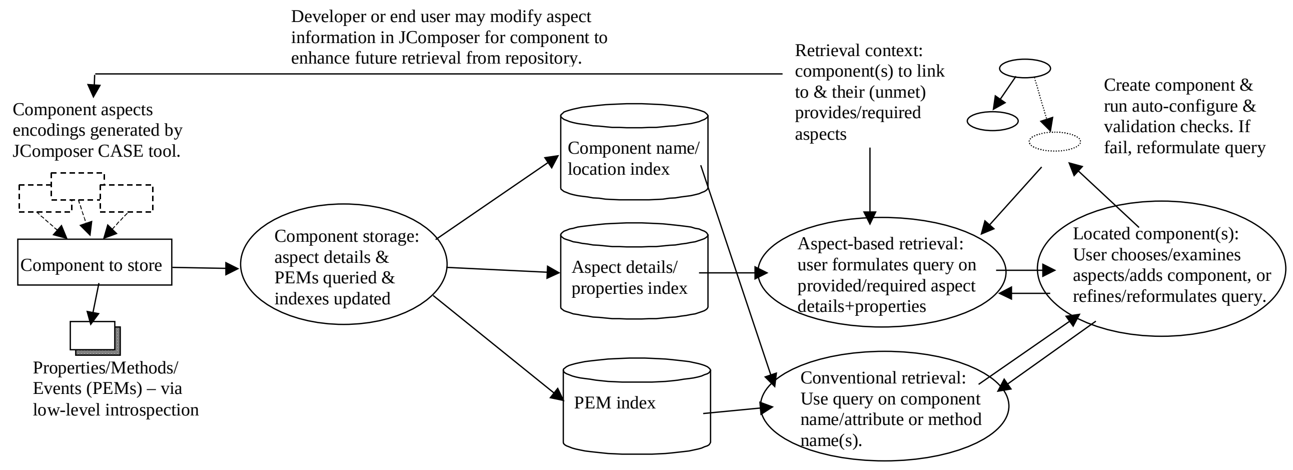
Figure 2 illustrates the concept of component aspects using a simple text chat client component. Components are in solid boxes, component aspects advertised by the chat client are dashed, and provided aspect details denoted by a '+' and required by a '-'. The chat client has a user interface (a panel), but requires a frame which manages this panel; it generates and actions events, but requires another component to handle actual network broadcasting (transport); it identifies the user but requires a component to provide a list of other users and chat history management, and requires security services from other component(s). Note that different components could be chosen to use or satisfy the provided/required aspect details e.g. a separate sectuirty management component, a CORBA or RMI event transport component, etc. Note that aspect details may relate to one or more component feature (methods, properties and/or events), and one component feature may be used by more than one aspect as necessary. This characteristic of aspects allows component developers to take multiple perspectives on a component's capabilities using aspects where these perspectives may naturally overlap.

We have used component aspects to aid developers when analysing, documenting and reasoning about component requirements [7], and when refining requirements into software component designs and implementations [6]. We have also developed support in a component-based architecture, JViews, for encoding aspect information in software components, for use at run-time by end users, developers and other components [6]. All JViews components advertise their aspects by using a set of AspectInfo class specialisations, similar to JavaBeans BeanInfo introspection classes and COM type libraries [20, 25]. In contrast, however, AspectInfo classes capture high-level information about component capabilities which are understandable by developers and, many of them, by end users. They also provide a set of standardised functions for invoking component operations and validating component configurations. Third party software components can have AspectInfo classes generated for them by our JComposer CASE tool.

We have used the advertising of component aspects by JViews-based components to generate indexes of component capabilities relating to different aspects of a software application, as shown in Figure 3. When a component is added to our repository, the component's aspects are queried and index entries entered for each aspect detail. Properties associated with each aspect detail are also stored to support further refinement of possible search criteria. We also index component names, properties, methods and events, in case users wish to query using them. Users formulate queries for components using either component, method or event names or aspects the component provides or requires.

4. Indexing Components using Aspects

A key aim of our component repository approach is to make it easier for developers and end users to formulate high-level queries for components and have access to high-level information about components retrieved. Our component aspects ontology for describing component capabilities provides a high-level language for users to describe desired component capabilities in queries and with which to review retrieved component capabilities. Thus every component added to the repository needs to have its aspect details queried and be indexed by these aspect details to facilitate retrieval. Component aspects are a three-level descriptive technique: aspects group aspect details which in turn have a variety of properties. Property values may be numeric, string literals, enumerated values or value ranges, depending on the aspect detail beging described.



Developer or end user may modify aspect information in JComposer for component to enhance future retrieval from repository.
Component aspects encodings generated by JComposer CASE tool.
Component to store
Properties/Methods/ Events (PEMs) – via low-level introspection
Component storage: aspect details & PEMs queried & indexes updated
Component name/ location index
Aspect details/ properties index
PEM index
Retrieval context: component(s) to link to & their (unmet) provides/required aspects
Aspect-based retrieval: user formulates query on provided/required aspect details+properties
Conventional retrieval: Use query on component name/attribute or method name(s).
Create component & run auto-configure & validation checks. If fail, reformulate query
Located component(s): User chooses/examines aspects/adds component, or refines/reformulates query.

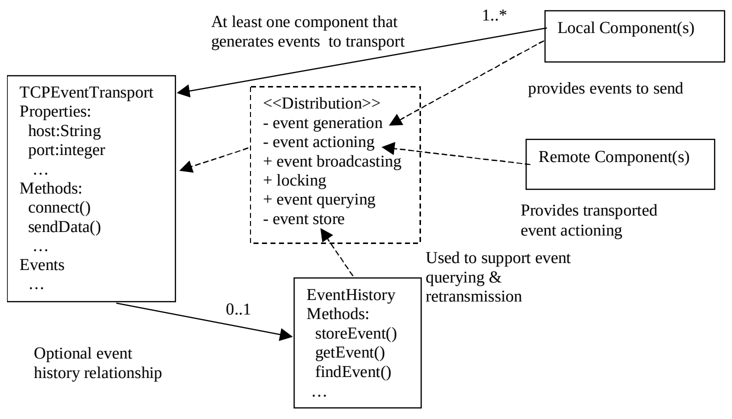
Figure 3. Aspect-based component repository architecture.

As an example, take the TCP/IP-based event transport component we have used extensively in a variety of JViews-based environments. This component is used to broadcast generated events to another user's JViews application, where generated events are propagated to components listening to its proxy. This component has only one aspect for which it provides and requires services, the Distribution aspect. Its specification using aspects is illustrated in Figure 4 (a). The Distribution aspect details it provides are event broadcasting, locking (using simple broadcast and wait for all remote components to finish responding before sending next event), and event querying.

It requires at least one component to generate events for it to transport, and at least one component to receive these transported events and act on them. It may be related to an event history component that is used to store events transmitted. Each of these provided and required aspect details has various properties, as illustrated in Figure 4 (b).

To index this component for effective retrieval we take each aspect detail and insert an entry in our aspect detail index indicating a component that provides distribution-related capabilities (event transport, synchronisation, broadcast type) and requires distribution-related services (event generation and consumption). For each of these aspect details, we take each detail property and create an entry indicating value(s) of the property possible for this component.

The property value may be a singleton (e.g. event_broadcasting.protocol = TCP/IP), an enumerated value (e.g. synchronisation.kind = Multi or Uni), or a value range (e.g. event_producers > 0). Some properties may have values which are dynamically computed depending on a component instance's configuration (e.g. retransmission=if event_history != null). Figure 5 illustrates the basic component indexing process we currently use.



```
component TCPEventTransport
  properties
     event_store : MVEventHistory
     host : String
     port : Integer
     …
  methods
     boolean connect(String,int)
     int sendData(Byte[])
     …
  events
     …
  aspect Distribution
     requires event_generation
        generate=after OR before
        transitive=true OR false
        event_producers > 0
     requires event_actioning
        event_consumers > 0
     provides event_broadcasting
        multicast=true OR false
        network_con=LAN OR WAN
        protocol=TCP/IP
        retransmission=if event_history != null
     provides locking
        kind=pesimistic
     provides event_querying
        store=event_store
        access=direct
  end aspect.
end Component.
```

(a) Event transport component aspect details. (b) Event transport aspect detail properties.

Figure 4. Distributed event propagation component's specification.

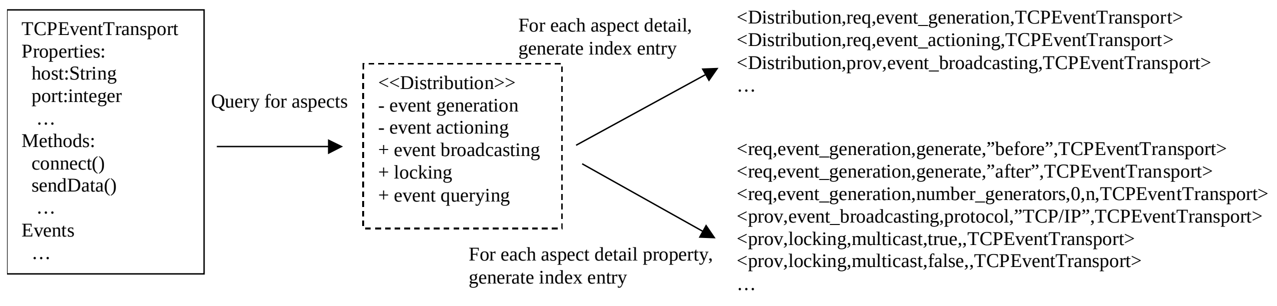

TCPEventTransport
Properties:
host:String
port:integer
…
Methods:
connect()
sendData()
…
Events
…
Query for aspects
<<Distribution>>
- event generation
- event actioning
+ event broadcasting
+ locking
+ event querying
For each aspect detail, generate index entry
For each aspect detail property, generate index entry
<Distribution,req,event_generation,TCPEventTransport>
<Distribution,req,event_actioning,TCPEventTransport>
<Distribution,prov,event_broadcasting,TCPEventTransport>
…
<req,event_generation,generate,"before",TCPEventTransport>
<req,event_generation,generate,"after",TCPEventTransport>
<req,event_generation,number_generators,0,n,TCPEventTransport>
<prov,event_broadcasting,protocol,"TCP/IP",TCPEventTransport>
<prov,locking,multicast,true,,TCPEventTransport>
<prov,locking,multicast,false,,TCPEventTransport>
…

Figure 5. Example of component indexing using aspects.

Some aspect details are "mandatory" if the component is to be reused. For example, an event generator must be supplied with an event source. Others are optional in some situations. For example, the event transport component can be linked to an event history which caches transported events to support asynchronous querying for prior events and storage of events for retransmission in case of temporary network failure. The event transport component can be used without this event store, but in this configuration it can not support asynchronous querying nor retransmission of events (as no history is maintained). When indexing components we indicate mandatory and optional details and properties.

5. Retrieving Components with Aspect-based Queries

To retrieve components developers and end users formulate queries based on the various aspects, aspect details and aspect detail property values the context in which they require a component indicates. The following steps to retrieving components are followed:

- The application invoking the repository querying tool can supply the "context" in which components are to be used, from which a partial query can be automatically constructed. For example, in JComposer and Serendipity, selected software components in a CASE tool design or agent specification can be queried for their aspects. Any currently "unmet" required aspects and "unused" provided aspects are usually to be partially fulfilled by new component(s) for which the user is searching the repository, and thus can be used to generate an initial set of query parameters.
- The user refines any defaulted query parameters or begins a new query. They select desired aspect details (grouped by aspect) that components they are searching for should either provide or require. Each aspect detail is added to the query, and only components that provide or require these details as appropriate will be retrieved.
- For each aspect detail selected, the user may optionally select one or more detail properties that the retrieved components should provide a value for. The user may specify a value for the property, may specify a range of values, or may just specify the property, indicating that retrieved components should at least supply some value for it. Note that some aspect details and detail properties are useful only for software developers (i.e. are "advanced" information about how a component works etc.). We allow end users of applications like Serendipity-II to request only a subset of all aspect details and properties are used, which provides a simplified description of component services and configuration.
- The query is run and all components in the repository providing or requiring the specified aspect details and having detail property values matching or within the range of those specified are retrieved.
- The user may view detailed aspect information for the retrieved components to help understand their purpose and how to configure them, may refine their search criteria if a large number of components are retrieved, may reformulate their query if no desired components are retrieved, or may request a new component instance be created.

The above steps are repeated each time the user wishes to retrieve and reuse new components. As an example, consider a software developer building a chat client, as shown in Figure 2, or an end user developing a distributed notification agent. The developer needs a component which can transport events generated by the chat client to other user's environments (to display the message in their chat clients), and the end user a component to transport events from components in their environment to components in another user's (or vice-versa). A query is formulated for such a component using the dialogue shown in Figure 6. Part of this query may be defaulted if e.g. the JComposer CASE tool determined the chat client component which the user wanted to link the new component has a Distribution aspect providing event generation and actioning but requiring event transport and locking. In the dialogue in Figure 6, the developer has formulated a query by specifying they want components providing event broadcasting and requiring event generation. For event generation properties they want events generated after a state change has occurred, and for event broadcasting they want uni-cast broadcasting. The query is then run and a short description of matching components are shown in the bottom pane. The user can view the different aspects and descriptions of aspect details and property values for any of these kinds of reusable components, can refine or reformulate their query, and can ask for one of these components to be created and added to the component specification view they are working on.

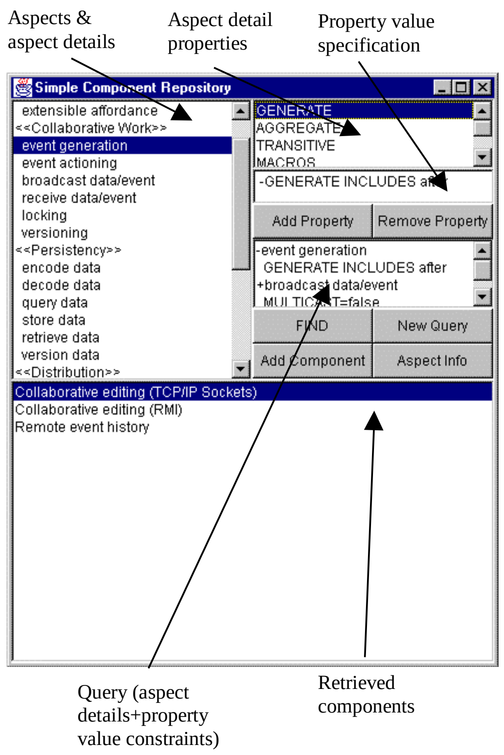
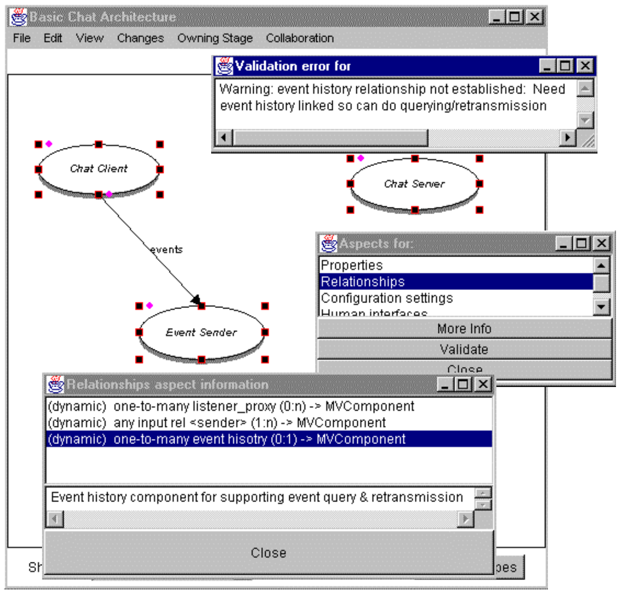


Figure 6. Component retrieval example.

Figure 7 shows an event transport component added to a JComposer component specification view. The user has connected the new component to the chat client component, and is viewing the aspect details for the new component. They have also asked for the aspect encodings to validate the configuration component. Note a warning is generated that this component has no event history component linked to it, so its event querying and retransmission facilities are unavailable in this configuration. The user could find an event history component from the repository (we have developed several versions of this kind of component, with different functional and non-functional characteristics), create one of these and link the event transport component to it.


Warning: event history relationship not established: Need event history linked so can do querying/retransmission

(dynamic) one-to-many listener_proxy (0:n) -> MVComponent
(dynamic) any input rel <sender> (1:n) -> MVComponent
(dynamic) one-to-many event hisotry (0:1) -> MVComponent

Event history component for supporting event query & retransmission

Figure 7. Retrieved component added in JComposer.

Developers and application end users can add their own newly developed or refined components and agent specifications (which are packaged as a single component interface) to our repository. In order to do this they need to specify appropriate aspects, aspect details and aspect detail property values for such components. This is done by using the JComposer CASE tool, which supports aspect information specification using a combination visual and textual languages. Wrappers for third party components can also be developed i.e. non-JViews components, which provide an interface to such component functionality. Aspect information is specified for these third party components and tools by specifying aspects for their JViews wrapper components.

6. Implementation and Integration

We developed our component repository as a JViews component. This allows it to be reused as a JavaBeans-compatible component in any JViews-based environment, and possibly in any JavaBeans-based component-based system. The component index is implemented as a hashtable of component names to component file locations and short descriptions (taken from the short description of components stored in their aspect information). This approach is used as all JViews components are implemented as extensions of JavaBeans components, and thus their implementation contained in .class bytecode files. Any third-party, non-JViews components have a JViews component "wrapper" which provides access to their capabilities. We have developed such components for interfacing to MS Word™, MS Excel™, Netscape™, Eudora™, a chat tool and a distributed file repository. Each of these wrappers has aspect characterisations of the tools they provide an interface to. When a retrieved component is to be created or its aspect information viewed by the user, the component's bytecode file is located using the component index and the component created and aspect information retrieved and displayed.

The aspect detail index is also managed as a persistent hashtable, with the provided/requried and aspect detail name as composite key. We have currently implemented the aspect detail property index as a set of relational tables using the mSQL simple relational database management system. Queries on aspect detail properties are translated into SQL queries which retrieve a set of component names. We plan to replace the mSQL engine and custom persistency mechanisms with an ObjectStore object-oriented database management system. This is because we are enhancing JViews-based applications to use ObjectStore for all object persistency management.

JViews-based applications like Serendipity-II and JComposer communicate with a component repository through its JViews component interface. They can request the repository dialogue be displayed to the user, can provide the repository component with the "reuse context" (a list of components currently selected in a view), and receive events from the repository component indicating a new component has been created and needs adding to the view. Currently the repository component supports configuration for developers (full aspect details and properties) and end users (implementation-level aspect details and properties not shown).

7. Discussion

A wide variety of research has been carried out on developing software component repositories [21], as effective storage and retrieval of reusable components is critical in managing and using large collections of reusable software [8]. The problem of managing and using large quantities of digitised information is not only relevant to software development, but to information management in general, as evidenced by the huge growth in research into digital libraries [28]. The need to support more effective software developer reuse of components has been an important area of research for some time [8, 12, 15], but the need to support end user reuse of software components has become more pressing in recent years as component-based systems become widespread [4, 13, 16].

Most component repositories adopt a form of syntactic indexing and querying, whereby component features (name, property and method names, type names, comments) are used to index components. Queries are made by specifying particular keywords or facets ("attributes") desired by reusers of components. Examples of such systems include CodeFinder [8], the Eiffel library [10], Phrasier [11], and those of JBuilder, Visual Age and Marvel [12, 2, 9]. Components are usually classified and organised by hierarchical groupings, as in EiffelBase, JBuilder and VisualAge, or by attributes (facets) of the components, as in CodeFinder [8]. Often natural language indexing and querying is also used, whereby typically comments associated with component features are extracted and indexed [8, 11]. All of these approaches suffer from a generally low-level view of component services, and consquently low-level indexing and retrieval queries which require users to be very familiar with component interfaces. Systems using structuring techniques to help guide users to appropriate collections of components do help, but these techniques require good understanding of the structuring mechanisms used and then still require typically low-level queries over subsets of components. Natural language-based searching can provide effective, higher-level access to component repositories, but is highly dependant on the quality of comments or user-provided indexing terms to be useful. Few syntax-based repository querying tools utilise the context a query is performed in to help guide searching. Fewer still allow users to modify component indexing terms in any straightforward way to allow users to provide extra information to help improve future requerying [8]. To our knowledge no syntax-based approaches utilise information about required component services/features, only provided features and services.

Our aspect-based indexing approach uses high-level conceptualisations of both provided and required component services to index and retrieve components. We have found it is much easier for both developers and end users to quickly formulate effective queries using these high-level service descriptions for components in our repository. Our technique is primarily a facet-based approach where users query for components based on particular systemic aspects ("attributes") of components. The ability to query components for both their provided and required services has proven to be very effective in quickly locating appropriate components, as when composing component-based systems often the required services of reused components are as important to understand as those they provide. Developers and end users can add additional information to component aspects using our JComposer CASE tool and then have the component reindexed in the repository, to improve future retrieval.

Semantics-based component retrieval techniques usually fall into one of three categories: type-based retrieval, execution-based retrieval and formal specification-based retrieval [21, 22, 23]. Type-based approaches try and locate components which match specified required types or groups of types, possibly using inferred types. Execution-based approaches use example input and output data, specified by users or generated from test cases or argument domains, and retrieve components whose outputs match those specified for the given input data. Formal specification-based approaches use some form of formal codification of a component's interface and behaviour to index components. All of these approaches have the potential to more or less automatically locate appropriate components for reuse with a minimum of user query specification. However, most of these techniques have so far proven successful in only functional programming domains, and not for more general software component applications. Our limited use of aspect-based configuration and validation functions of retrieved components could be viewed as a form of execution-based retrieval (or query filtering). As we incorporate more formal specification techniques into our aspect-oriented component engineering methodology, particularly with regard to specifying possible aspect detail property values, we hope to be able to more accurately index component functionality and especially non-functional constraints with our aspects.

A range of future research directions exist for this work. Incremental query execution, similar to that of Phrasier whereby as users specify index terms (aspect details and detail property value constraints), components matching the evolving query are returned and displayed. This would give the user more immediate feedback on the results of modifying query terms. A ranking scheme which uses the number and closeness of fit of the aspects of returned components to the queried aspect information would be useful in order to present "most likely matches" to users ahead of less closely matching components. A visual querying tool, similar to our JVisualise dynamic component visualisation tool [3], would be a possible enhancement to the query tool user interface. This would allow users to specify aspect-based queries using an extension of the component modelling language of JComposer, and possibly specify queries "in situ" in JComposer and Serendipity-II views.

Retrieval of multiple, related components is a major area of work we wish to persue. Typically some component provided and required aspects are obtained via other, related components (e.g. the event transport component and event history component together provide a remote event query service). At present, users of our repository query for each component they wish to reuse separately, with limited information from the previously found and reused component influencing the next component search. Showing users groups of related components that potentially match a query, or allowing users to more easily and naturally construct multiple queries for several related components would greatly enhance their ability to find appropriate collections of components for reuse. Semi-automated configuration and validation of these collections of components using our aspect encodings is an additional enhancement we wish to investigate.

8. Summary

We have described a new approach to storing and retrieving software components from a repositories to foster improved reuse of components by both developers and application end users. High-level characterisations of component provided and required services are used to generate indexes of these component capabilities. Components are retrieved from a repository by the formulation of queries over these provided and required services. Queries may be partially constructed automatically, based on the reuse context of the component. New components can be added to the repository with automatic indexing generated from their high-level aspect characterisations. Possible enhancements include supporting visual forms of component query construction and ranked result visualisation, supporting queries for multiple, related components at one time, and automatic configuration and validation of multiple retrieved components.

Acknowledgements

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

References

1. Apple Computer Inc., *OpenDoc Users Manual*, 1995.
2. Borland Inc, *Borland JBuilder*™, Borland Inc, http://www.borland.com/jbuilder/, 1998.
3. Grundy, J.C., Mugridge, W.B., Hosking, J.G. Static and dynamic visualisation of component-based software architectures, In *Proceedings of 10th International*

Conference on Software Engineering and Knowledge Engineering, San Francisco, June 18-20, 1998, KSI Press.
4. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
5. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, Vol. 2, No. 5, September/October 1998, IEEE CS Press.
6. Grundy, J.C. 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.
7. Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In *Proceedings of the 4th IEEE Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, IEEE CS Press, pp. 84-91.
8. Henninger, S. Supporting the Construction and Evolution of Component Repositories, In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996, IEEE CS Press, pp. 279-288.
9. IBM Inc, *VisualAge™ for Java*, 1998, http://www.software.ibm.com/ad/vajava.
10. ISE Inc, *EiffelBench Guided Tour*, 1999, htp://www.eiffel.com/.
11. Jones, S. Phrasier: An interactive system for linking and browsing within document collections using keyphrases, In *Proceedings of INTERACT'99*, Edinburgh, Scotland, September 1-3 1999, Kluwer Academic Publishers.
12. Mareek, Y., Berry, D., and Kaiser, G. An information retrieval approach for automatically constructing software libraries, *IEEE Transactions on Software Engineering* Vol. 17, No. 8, August 1991, 800-813.
13. Mehandjiev, N. and Bottaci, L. (1998): The place of user enhanceability in user-oriented software development, *Journal of End User Computing*, Vol. 10, No. 2, 4-14.
14. Meyer, B., Mingins, C., and Schmidt, H. Providing Trusted Components to the Industry, *IEEE Computer*, May 1998, pp. 104-15.\
15. Mili, H., Mili, F., Mili, A. Reusing software: Issues and research directions, *IEEE Transactions on Software Engineering* 21(6), June 1995, 528-561.
16. Morch, A. Tailoring tools for system development, *Journal of End User Computing* Vol. 10, No. 2, 1998, pp. 22-29.
17. Motta, E., Fensel, D., Gaspari, M., Benjamins, R. Specifications of Knowledge Components for Reuse, In *Proceedings of 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 36-43.
18. Mowbray, T.J., Ruh, W.A. *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley, 1997.
19. Netscape Communications Inc, *Visual Javascript™*, 1998, http://www.netscape.com/.
20. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
21. Pai, Y. and Bai, P. Retrieving software components by execution, In *Proceedings of the. 1st Component Users Conference*, Munich, July 1996, SIGS Books, pp. 39-48.
22. Podgurski, A. and Pierce, L. Retrieving reusable software by sampling behaviour, ACM Transactions on Software Engineering and Methodology 2 (3), 1993, 286-303.
23. Rittri, M. Using types as search keys in function libraries, Journal of Functional Programming, 1 (1), 1991, 71-89.
24. Rollins, E., Wing, J. Specifications as search keys for software libraries, In *Proceedings of the International Conference on Logic Programming*, 1991, MIT Press, pp. 173-187.
25. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
26. Wagner, B., Sluijmers, I., Eichelberg, D., and Ackerman, P., Black-box Reuse within Frameworks Based on Visual Programming, In *Proeedings of the. 1st Component Users Conference*, Munich, July 1996, SIGS Books, pp. 57-66.
27. Weyuker, E.J. Testing Component-based Sotware: A Cautionary Tale, *IEEE Software*, Sept/Oct 1998, pp. 54-59.
28. Witten, I., Moffat, A., Bell, T. Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Edition, Morgan Kaufmann, 1999.