New Wave Searchables: Changing the Paradigm of Internet-Scale Search

Axel Uhl and Horst Lichter

Abstract— **Internet search engines today are facing problems in keeping up with the pace of web growth. Two facts are responsible: bandwidth bottlenecks due to central indexing; *deep web* (or *invisible web*) contents that are inaccessible for search engines. The *New Wave Searchables* framework provides an approach that can solve these problems.**

Keywords— **Internet search, distributed information retrieval, software architectures, modeling, searchability, UML, New Wave Searchables**

I. Introduction

TODAY'S Internet search engines compute their centralized index by crawling web contents. This approach implies two major problems:

- large and relevant parts of the Internet content are not reachable by crawling and thus remain inaccessible for search engines [1]
- bandwidth and its growth impose harsh limits on central index currency and indexable share of vastly growing available information

Other publications support these findings, e.g. [2]: "...search engines are increasingly falling behind in their effort to index the web..."

The obvious solution is a distributed approach to information retrieval that better leverages the available bandwidth in order to achieve higher index currency and improved coverage, including deep web contents. Forward knowledge [3] — like keyword indices — have to be stored "closer" to the searchable information sources than the central index approach currently does. Furthermore, their updating has to happen in a more bandwidth-efficient manner as compared to the change detection heuristics and "brute force" crawling methods used today.

A typical search scenario could look like this: A user directs a web browser at a search portal. First, he or she configures which parts of the content accessible to him or her shall be searched. This may, i.e., include "standard" web pages, product descriptions, multimedia archives, scientific publications, person directories, a local mail database, or a corporate intranet with all its documents and databases.

This selection will imply the set of different types of queries the user may enter, depending on the selected information sources' query processing capabilities. Query types may range from simple keyword queries over natural language queries to complex multimedia queries like image similarity or sound sample matching. Deciding for a query type will further confine the set of searchable sources to which queries of that type are applicable. Furthermore, it will present the user with a web interface for creating a query of the selected type.

Once the query is entered and submitted it will travel through the network towards the selected information sources. Once the query reaches a source the search results will be computed for that particular source and will get sent back to the network node where the query originated from.

This process can then be optimized by the already mentioned distribution of forward knowledge. This knowledge may be used to answer a query without actually having to send it to the original source which is a typical way to save bandwidth.[1]

Sections II and III will introduce an object-oriented framework — the *New Wave Searchables*^{TM} — facilitating the construction of systems that resemble the example sketched above. Subsequently, an approach is introduced in sections IV and V with which instantiations of this framework can be automatically generated from UML application models that are annotated with searchability information. The generation can be customized such that the resulting implementations are integrated with modern application architectures like those based on J2EE/EJB including a web-centric user interface.

II. An Infrastructure for Scalable Distributed Search

The basis for all further discussions in this article is the top level of the *New Wave Searchables* framework as depicted in figure 1[2]. Searchable data and information sources are represented by the abstraction `Searchable`; query types that can be applied to searchable sources obey to the common abstraction `Query`. Results of a search are collections of objects implementing the `SearchResult` interface. The querying capabilities that a source supports can be expressed by instances implementing the `Production` interface.

Note, that queries, search results, the capability description objects, and the user-interface accessors used for query creation (see below) are designed to be value-types (implementing the `Serializable` interface) whose instances can be passed as arguments and results of remote operation

A. Uhl is with Interactive Objects Software GmbH, Freiburg, Germany. E-mail: axel.uhl@io-software.com

H. Lichter is Professor at the Aachen Technical University, Aachen, Germany. E-mail: lichter@informatik.rwth-aachen.de

[1] An extreme and "degenerated" form of forward knowledge management is central index-keeping where indices are build using crawling techniques.

[2] Note the component icons labeled with a capital *R*. These are classifiers with stereotype *Resource* as defined in [4], using *Organizations*, *Processes* and *Resources* as base abstractions for all kinds of business models.

Proc. of International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet SSGRR 2001, L'Aquila, Italy.

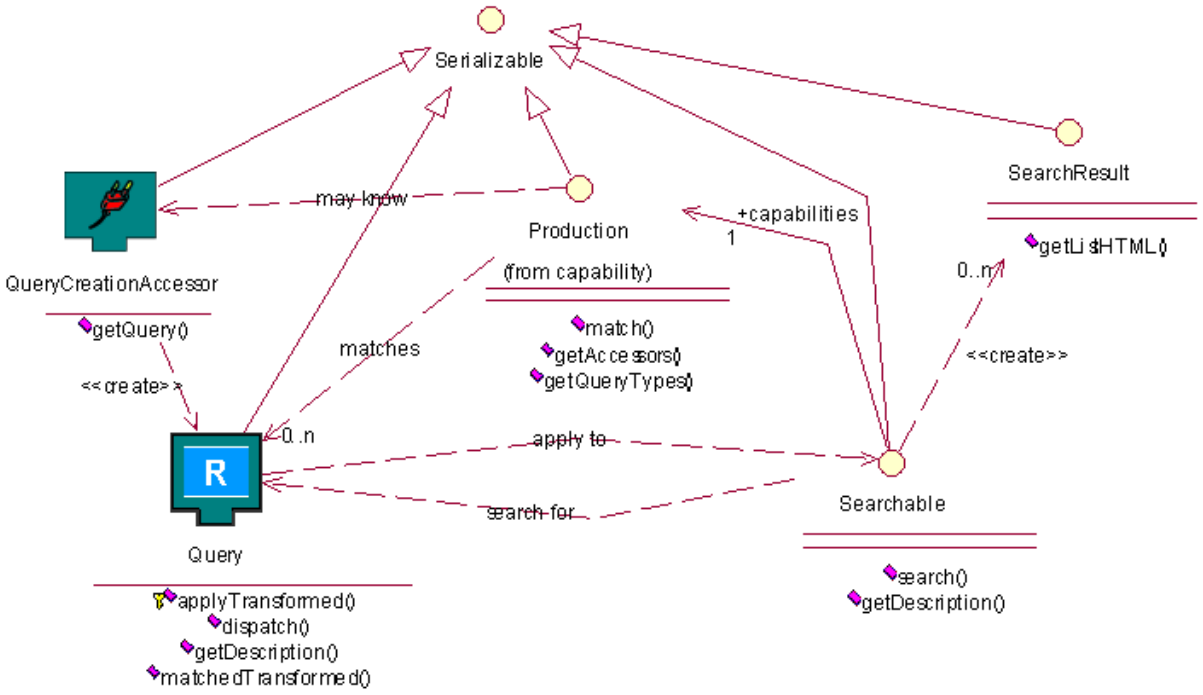


Fig. 1. UML model of the top-level abstractions of the *New Wave Searchables* framework

invocations. Using Java for implementation, this feature comes with the language's serialization framework.

Even with these fairly simple base definitions it is possible to provide a variety of query types, beginning with easy-to-use keyword query types that may be combined using boolean expressions, and reaching as far as image similarity queries where reference image data is part of the query. The design lends itself well for flexible extension of the set of possible query types, mainly due to the object-oriented principle of type inheritance.

`SearchResult` as part of the top-level object model can be extended by proper specializations, e.g. `WebSearchResult` which contains typed attributes like the URL of the found result, the title, an excerpt, etc. As search results are designed as serializable value types these objects won't lose their structure when being passed to clients. This is an advantage over HTML-based search "architectures", especially when the search results are to be processed further by electronic systems and are not only intended for display to human users.

In order to make implementing the `Searchable` interface as easy as possible, a default implementation `AbstractSearchable` is provided (see figure 2). It leaves only two major issues up to the developer of a new `Searchable` implementation: Defining the search capabilities and implementing the corresponding `search` methods. Many other issues like default query transformation and dispatching, or ranking of results if the query provides a ranking function are handled generically by the framework. Specializations of these default algorithms may be implemented as needed.

Accessing `Searchable` objects that are distributed

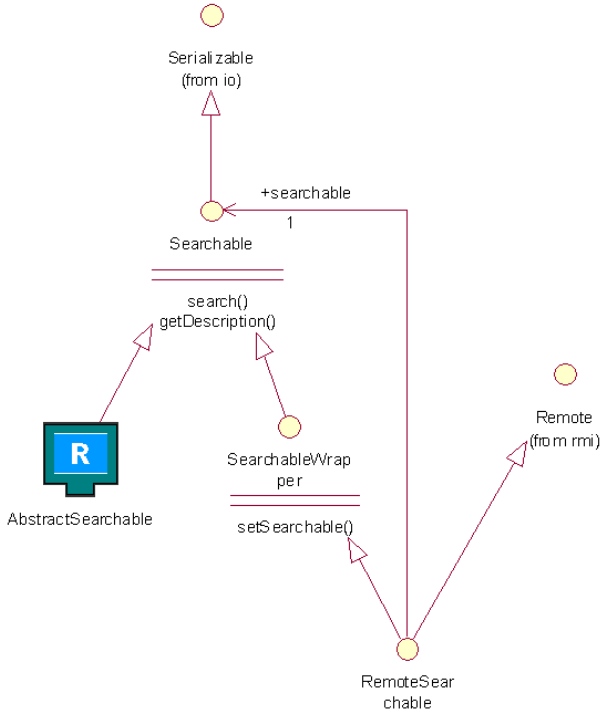


Fig. 2. Default implementation and remote-enabling of `Searchable`

across a network is also made possible by the framework. It provides the `RemoteSearchable` specialization of the `Searchable` interface which is a Java RMI remote interface (see the relationship to interface `Remote` in figure 2). Again, using Java here proves very helpful; besides transparent remote method invocations including marshalling

and demarshalling of arguments, results and exceptions the Java runtime environment also addresses issues like network security and firewalls. RMI's protocol can be run over HTTP connections or can be restricted to use dedicated TCP ports for communication, giving the network administrator fine-grained control.

In order to embed the framework into a web-centric environment with HTML-based user interfaces the framework has to specify ways in which users can create `Query` objects from their web browsers. Examples are a simple text entry field in an HTML form where users can input a keyword query string with boolean operators; or for creating an image similarity query a user could select an image file on his or her local file system and transmit it to the server that is creating the `Query` object using an HTTP PUT request.

These user interface application elements responsible for the creation of `Query` objects are termed *Query Creation Accessors* in accordance with the definition of user interface *accessors* in [5], [6]. They are depicted in figure 1. Again, objects of this type can be passed by value together with their implementation, even across remote calls, which makes it easy to flexibly embed their functionality into web applications like those running on portal web sites providing powerful search features.

Another important concept in the framework is that of a *Trader*. Traders implement a *composite* pattern [7] regarding `Searchable` objects. A trader implements the `Searchable` interface itself and knows a set of references to other `Searchable` objects which can also be traders again. Queries directed to a trader are conceptually sent out to all `Searchable` objects known by the trader that support the given query type. Due to the formal query capability descriptions of each `Searchable` object a trader can decide efficiently where queries must be sent — in case the serializable `Production` objects are cached in the trader even without requiring any remote calls for this check. The search results are merged in the trader, possibly according to a ranking function if one was provided together with the query, and then returned as an ordered collection of `SearchResult` objects.

Obviously, actually distributing queries to *all* `Searchable` objects supporting the query type doesn't scale for large networks. A good example for this are the problems of the *Gnutella* network, e.g. described in `http://www.tch.org/gnutella.html`. Therefore, trader implementations may — supported by `Searchable` implementations — opt to maintain forward knowledge about some or all of the referenced searchable contents. This will enable a trader to answer queries on behalf of its referenced `Searchable` objects without having to use bandwidth to distribute the query and to collect the results. Implementing a distributed *observer* pattern [7], [8] is a bandwidth-efficient approach for updating the forward knowledge stored in a trader.

In addition to acting as smart caches for information retrieval, traders are also the place where intelligent, bandwidth-aware reconfiguration algorithms may be hosted. Analyzing the dynamics of query-response behavior of the trader's referenced `Searchable` implementations the trader can establish "shortcuts" to searchable sources that, e.g., understand many query types or return many results. This can save bandwidth, as fewer intermediary transmissions of query and result objects are required.

III. The *New Wave Searchables* Prototype

The framework described in the previous section has been implemented prototypically in Java. Its source code is freely available at `http://www.NewWaveSearchables.com` under the GNU General Public License (GPL). The search portal that runs as a prototype at the mentioned URL is based on these sources. It has proven the concepts as feasible, and it is briefly described in this section.

For several reasons of which some were already mentioned in section II Java was chosen as implementation language for the prototype. Most importantly:

- portability
- rich library support for Internet access
- RMI framework for distributed objects
- serialization allowing pass-by-value for complex objects including their implementation
- built-in security, policy as well as subject based

The prototype mainly consists of four parts: the framework implementation, hand-written wrappers and query types for several searchable web sources, a portal web site based on the framework, and an example for an automatically generated search adapter based on a UML application model. The latter will be described in section V, after the concepts for modeling and generating searchability support have been introduced.

The framework implements all key abstractions as shown in figures 1 and 2. The concept of *query creation accessors* has been drafted as a set of Java interfaces, but has not yet been integrated with the portal web site. Several text query types (`KeywordQuery`, `PhraseQuery`, `RegexQuery`), a proximity query type, a query type based on an OQL-related query language, attribute queries, composite queries like `AndQuery`, `OrQuery`, and decorated queries like `RankingQuery` have all been implemented.

With this query type set some query transformation algorithms were implemented. Proximity queries may be transformed into relaxed `AndQuery` objects, whose search results are then postprocessed by a proximity matching algorithm. The boolean composite query types `AndQuery` and `OrQuery` may be decomposed into their constituents who then get executed separately with the results being merged according to the logics of the boolean operator.

Formal query capability description is implemented in the prototype as well. Several standard `Production` implementations are given, e.g. `PrimitiveProduction` matching queries as long as their type is assignment compatible to a specified query type; or `CompositeProduction` recursively describing how a composite query like an `AndQuery` may be constituted. In order to express search capabilities for attribute-searchable sources, an `AttributeModel` class is provided. With it an arbitrarily nested tree of associations and attributes can be described as metadata objects.

Instances of class `AttributeQuery` can then be validated against such an `AttributeModel` for syntactical correctness.

The framework also provides a very simple `Trader` default implementation. Its main purpose is to implement the composite pattern for `Searchable` objects. It does so by delegating incoming query objects to all referenced `Searchable` objects whose query capability descriptions match the query. The returned results are collected and merged. If the query was a `RankingQuery`, the `Comparator` implementation provided together with the query is used for merging the results.

The trader's query capabilities are also computed according to the composite pattern: they are the combination of all contained `Searchable` objects' query capabilities. The provided trader implementation does not yet support any automatic reconfigurations nor any caching of forward knowledge.

With this framework a set of wrappers for existing searchable web sources has been implemented, among others for popular web and usenet search engines, the *Java Technology Site Search*, the *Computer Science Bibliography* at the University of Karlsruhe, and for different air-travel web sites allowing to search for cheap airfares. For the latter, the additional query type `FlightQuery` has been created, which has been used embedded into `RankingQuery` objects with an `AirfareComparator` that compares resulting `TripSearchResult` objects by their associated price tags.

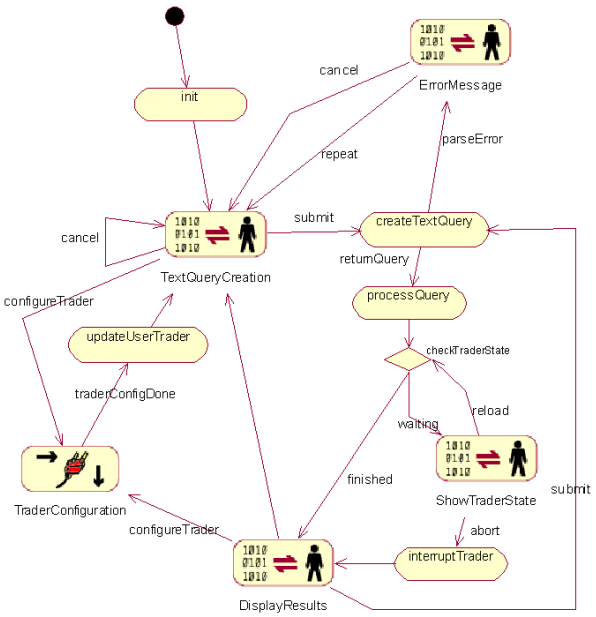


Fig. 3. UML activity diagram for the prototype's web application

In order to demonstrate integration with web-centric environments we created a web application through which users can enter queries and view the search results. The web application was modeled as a UML activity diagram (see figure 3), and a servlet together with a set of Java server pages (*JSPs*) was generated from that using the tool *ArcStyler*. Each user gets assigned a *personal trader* that can be configured with the web application. References to `Searchable` objects can be added to or removed from the trader by the user.

As only query creation accessor the prototype currently supports one for text queries and boolean combinations thereof. It is presented as a single-line text entry field in an HTML form. The entered query string is analyzed by a parser that was generated using the *JavaCC* parser generator toolkit. Resulting from this analysis is a query object that is then sent to the user's personal trader.

All `SearchResult` objects are capable of rendering themselves as HTML list entry. This capability is used by the web application to display the set of results delivered by the trader. In case of `WebSearchResult` objects as returned by typical web search engine wrappers the HTML representation consists of a link to the found document, a brief description, if available an excerpt from the found document, and a link to the search engine that found the link.

IV. Searchability as Aspect of Internet-Centric Application Architectures

A fast growing share of the publicly available web content is no longer being served from static HTML documents but rather from online applications that are often database-driven. Given this trend it turns out that more and more web content appears in the deep web that — as mentioned in section I — is not amenable to search engines' crawlers.

The framework presented in section II can be used to also search deep web contents. The content providers have to contribute by means of providing a `Searchable` implementation that searches their specific information. One way to accomplish this is, of course, a manual implementation of the interface. But this is tedious and prone to break whenever the business processes or the formats of the contents made searchable change. Yet, companies like *Equero AG* pursue such an approach, offering implementation of search wrappers for deep web sources as a service.

An architecturally more solid alternative is to make *searchability* an integral part of the overall application architecture. Like we are used to regarding persistence, distribution, and transactionality standard items of "real-world" application architectures we should add searchability as just another item to this list. It should be as natural to apply changes to an application's searchability as it is to modify its transactional behavior or the physical distribution of its components.

Using a model-driven approach to application development we can start to integrate searchability into application models, as we do already today for the other aspects mentioned above [9]. Not only will this enable automatic integration with global Internet search as will be shown in the next section, but also it will become possible to utilize this model information for automating implementation of information retrieval support within the application. This is illustrated in figure 4. Examples are a customer relations management tool where searchability plays an important role, or a warehouse management application providing one

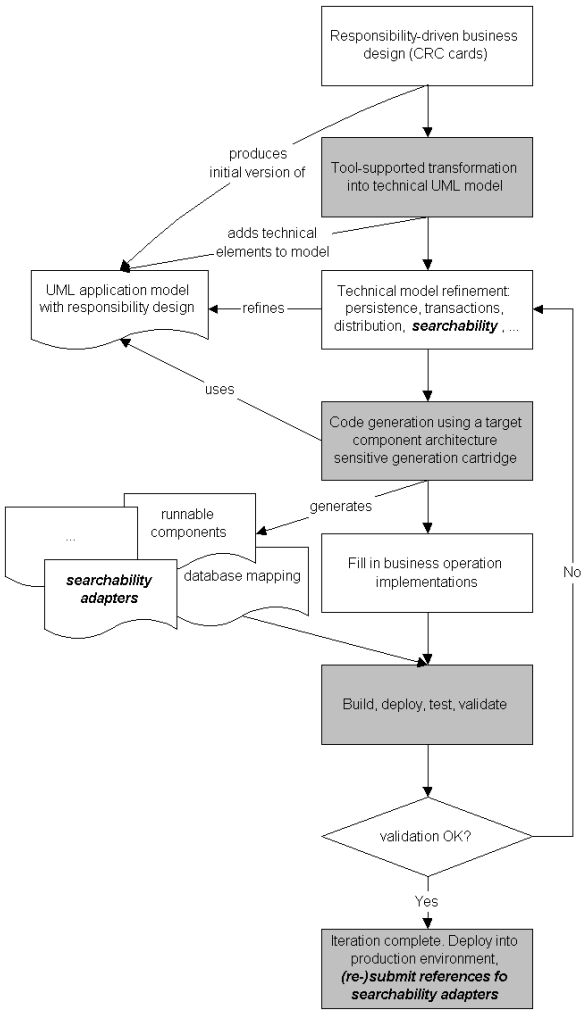


Fig. 4. Steps in using a model-driven application development approach. Searchability related activities are highlighted in **bold** print, fully or partly automated steps are displayed shaded.

set of search features for its internal users and another set of search features for its external Internet integration.

V. Generating Searchability Support from UML Models

Model-driven application development is especially useful in areas where the target architecture is complex and has to deal with several different aspects. What may be a single component in the model that represents some business logic may end up as many physical artifacts, e.g. compilable source code files for the component's business dimension, its technical dimension, sources for client-side proxies or *personalities*, deployment descriptors, developer documentation, and the like. We also call this few-to-many relation between model elements and physical artifacts the *generation fan-out*. Complex component technologies typically are characterized by a high generation fan-out[3]. Translative code generation [10] can be used to transform the models into the physical artifacts mentioned above.

The remainder of this section will shed light on how the *Unified Modeling Language* (UML) can be used to express searchability as an integral part of an application model. It will be shown how from these parts of the model an implementation for the modeled searchability can be generated automatically. Obviously, there is no "one and only" way to do this. Therefore, a simple example is given, based on which future searchability modeling styles may thrive.

From now on it is assumed that an application model is given in which business components are represented as UML classifiers, together with their operations, attributes and associations with other components.

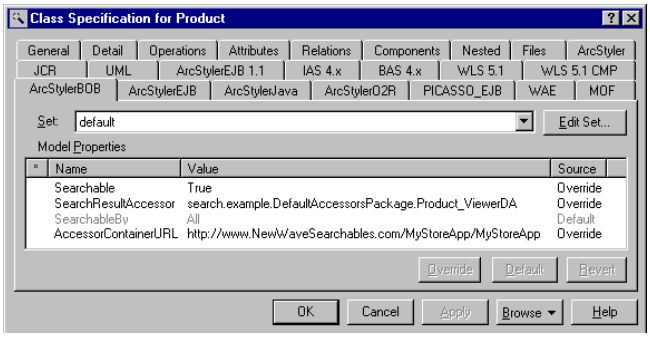


Fig. 5. Searchability properties of a component

Figure 5 shows a set of properties pertaining to a UML classifier that specify the general searchability for a component. The first is the boolean flag `Searchable` which, in this case, is set to `True`, indicating that this component is a searchable component. In this case an adapter will be generated for this particular component that implements the `Searchable` interface.

The adapter will support the `AttributeQuery` query type which is also part of the *New Wave Searchables* framework. Queries of this type consist of a set of pairs, where each pair consists of an *attribute path* specification — a dot-separated list of attribute or association end names with an attribute at its end — and a subquery to be applied to the value of the attribute denoted by the path. A component is considered matched by such an `AttributeQuery` if for all contained pairs the inner query matches the corresponding attribute, starting attribute path navigation from the component at hand[4].

In the searchability modeling style example explained here it is assumed that a search result must be displayable as an HTML table entry, containing a link that leads to a human-readable representation of the found component instance. In order to be able to generate a complete implementation the model must contain information on how to assemble those result objects. Here, these are the two properties `SearchResultAccessor` and

[3]This, by the way, makes true round-trip engineering close to impossible, for there is typically no unambiguous reverse mapping of changes to single artifacts.

[4]It may occur that an attribute path traverses a to-many association. In this case the subquery is conceptually applied to *all* associated instances, and the subquery is considered a match if at least one of the associated instances produces a match for the subquery.

`AccessorContainerURL`. They define the name of a server-side Java class (a user interface *accessor*, as explained in section II) capable of rendering a component instance as HTML, and the URL of a servlet that can execute this accessor, respectively.

With this information at hand an implementation can be generated that creates `SearchResult` objects that can render themselves as HTML table entries with a link included that is based on the servlet URL, the accessor class name and additionally on a reference to the found component instance. When such a link is traversed, it will invoke the servlet which in turn activates the specified accessor and passes the reference encoded in the link to the accessor. This reference can then be used by the accessor to extract any information from the component required for rendering its user interface which is then transmitted to the client by the servlet.

Next, searchability has to be specified for attributes and associations of the searchable component. This will decide which sorts of `AttributeQuery` objects are accepted at all. Obviously, queries providing subqueries for non-existing attribute paths are invalid. But queries are also invalid if they use attribute paths that contain elements that have not been marked as searchable in the model.

Each attribute and each association end in the model may — just like classifiers — be tagged as searchable. This information will be used to generate an `AttributeModel` implementation for the classifier to which the tagged attribute / association end belongs, defining exactly the set of valid attribute paths.

For each searchable component a method is generated into the component's life-cycle management interface that is responsible for finding component instances based on a given query. Using EJB as component technology this would be the component's *home interface*, and the method would be a *finder* method. This finder method takes a query object as argument which is then transformed into a query that can be executed against the component's persistence manager. This could be a database query, an EJB-QL query, or — if an object-to-relational (*O2R*) mapping tool is used — a query in the O2R tool's query language.

This transformation can be implemented generically, parameterized only by the particular `AttributeModel` for the searchable component. Thus, the transformation algorithm does not need to be generated for each searchable component but can be coded once and then be reused by all searchable components, e.g. by subclassing or delegation.

A prototype has been implemented for the *BEA Weblogic* EJB application server in conjunction with the *TopLink* O2R mapping tool. *Tomcat* was used as servlet execution engine. This combination was particularly suited for the task at hand because of TopLink's powerful dynamic querying capabilities which makes the transformation of framework queries into TopLink queries straightforward.

The prototype handles all kinds of text queries (keyword and phrase), boolean combinations thereof, and proximity queries, furthermore attribute queries complying with the searchable component's attribute model. They are transformed into TopLink's association and attribute traversal methods (`anyOf`, `get`), text search expressions (`containsSubstring`), and boolean operators (`and`, `or`). Searchable attributes in the prototype currently have to be of type `String`, but obviously, other attribute types could easily be supported as well.

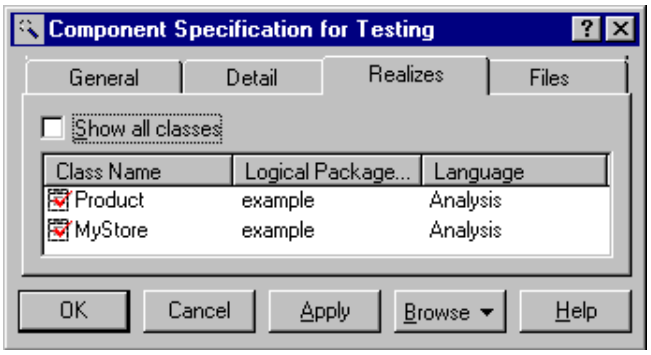


Fig. 6. Modeling a `Trader` component and assigning a set of classifiers marked as searchable (in this case `Product` and `MyStore`)

If several components have been modeled as searchable, and a single point of entry is to be provided for searching the application, sets of searchable components can be assigned to UML *components* with stereotype `<<Searchable>>` in the model (see figure 6). This will cause the generation of a `Trader` implementation that creates and references the generated `Searchable` adapters for the assigned searchable components.

Once again: the searchability modeling style presented here is just an example of how searchability of an application can be expressed in UML. A prototype has proven the feasibility of the concepts. Now further research has to show how far this approach may reach, what other searchability modeling styles can be conceived of, and what the limits and constraints of the approach are.

VI. Runtime Environment

Section V has delivered a technique for automatically generating implementations for application searchability support. This section will show how the results can be embedded into a global information retrieval environment, and which aspects have to be considered in order to ensure reliable operation of the search infrastructure.

The adapters and traders generated by the procedure presented in section V have to be instantiated by a running process, and references to the instances have to be added to other — typically remote — traders. This will integrate the generated `Searchable` objects into a global mesh of interconnected, federated traders.

This procedure resembles the steps a web site operator performs in order to have the site indexed by crawling-based search engines where the operator will submit links to selected documents on the site to several search engines, hoping they will crawl their contents and make them appear in the search engine's index.

Search portals may then set up their trader by referencing all `Searchable` objects — including other traders —

they want to include in searches performed through the portal. At this point the search portal can benefit from the query creation accessors specified by implementations of the `Searchable` interface: They can be automatically embedded into the portal's user interface. Of course, security policies may apply, with the portal accepting accessors from specific organizations only, or restricting runtime privileges of the accessors. Java's policy-based descriptive standard security, together with its *JAAS* framework for authentication and authorization, makes subject-based privilege enforcement easy to implement.

When distributing object references across the Internet and storing them in traders, it is important that the implementation of these references to the `Searchable` objects are robust against things like process restarts, server reboots or class evolution. Two techniques have been evaluated in the prototype:

- using *Java RMI Activation* and the *activation daemon* `rmid`
- implementing *loose references* that are bound to objects only via a naming service

RMI activation allows the use of so-called *activatable object references*. Such references are robust against process or server restarts. The activation subsystem of the Java runtime environment persistently stores information that can be used to activate objects on demand, whenever an activatable reference to an object of that kind is used.

Unfortunately, it is hard to make these references "survive" major changes to the implementation of the referenced objects. Partly, this is due to limits in compatibly evolving the serialized activation information together with the changes in the `Searchable` implementations. Another challenge is the management of *activation IDs*. Those are part of activatable object references and are used by the activation subsystem for deciding which class to instantiate and from which persistent source to retrieve the activation information. By default, the activation subsystem generates new and unique IDs for each object that gets registered as activatable. Significant effort would have to be spent in assigning dedicated activation IDs, because then uniqueness and association with persistent activation information would have to be managed explicitly.

In the prototype it turned out to be easier to use *loose references* bound to objects by naming. This can be combined with RMI activation by caching an activatable reference after retrieving it from the naming service. A naming lookup then only has to take place in case the activatable reference goes stale. The resulting reference implementation performs comparably to activatable references as long as the reference doesn't go stale, and only one naming service lookup is required to update the reference in case using the cached activatable reference fails. With this reference implementation even the class of the referenced object may change[5] dynamically without having the reference break.

[5] as long as it remains assignment compatible to `Searchable`

VII. Conclusions and Outlook

Global search across all publicly accessible Internet contents represents an enormous technical challenge. Today's central index based approaches have problems regarding access to the information, especially in the deep web, as well as keeping up with the content growth and change frequency, due to bandwidth constraints.

In order to get closer to the goal of making all available information searchable the paradigm of Internet search has to be reversed. While crawling still may play a role for static parts of the web, content providers will have to contribute by explicitly providing search interfaces to their web-enabled applications and other web contents.

This article has presented a framework – the *New Wave Searchables* — that defines searchability interfaces and implements many best practices from the field of distributed information retrieval. A comprehensive prototype has been used to evaluate the concepts and to test feasibility.

A novel approach was described that integrates *searchability* as one more integral aspect of modern application architectures that can, like other aspects, be expressed in models, using e.g. UML. Given a searchability modeling style, adapters can be generated automatically using translative code generation that integrate the application with the global search framework.

Integration with interconnected, federated networks of traders and search portals can easily be implemented using this approach, as has also been validated with the prototype.

Further research will have to be focussed on bandwidth-efficiently distributing and updating forward knowledge between traders and intelligent, dynamic, bandwidth-aware, and automatic reorganization of trader meshes. It will be interesting to look closer at the forces affecting the evolution of the set of available query types; query power, easy of use, and standardization are just a few of them. Not least, UML searchability modeling style alternatives and extensions to the example presented in this article have to be investigated; potentials and limits of the approach are to be researched.

References

[1] Michael K. Bergman, "The deep web: Surfacing hidden value," July 2000.

[2] Steve Lawrence and C. Lee Giles, "Searching the World Wide Web," *Science*, vol. 280, no. 5360, pp. 98, 1998.

[3] C. Weider, J. Fullton, and S. Spero, "Architecture of the whois++ index service," Feb. 1996.

[4] David A. Taylor, *Business Engineering with Object Technology*, John Wiley & Sons, Inc., 1995.

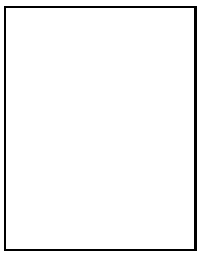
[5] Jens Heiderich and Richard Hubert, "Verfahren zur modellbasierten objektorientierten Entwicklung von externen Schnittstellen für verteilte Softwaresysteme, Deutsches Patent, Anmeldenummer 00123321.2," Oct. 2000.

[6] Jens Heiderich and Gisela Hillenbrand, "Arcstyler 2.5 accessor guide," .

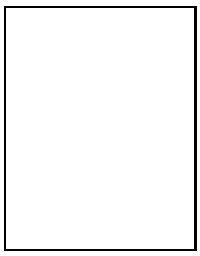
[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison Wesley, Reading, 1996.

[8] Axel Uhl, "Verfahren zur Informationsübertragung, Patent Nr. DE19832482 A 20000127, Aktenzeichen DE19981032482 19980720, Prioritätsaktenzeichen DE19981032482 19980720, Klassifikationssymbol G06F13/14 ; H04L12/24," Jan. 2000.

[9] Richard Hubert, "White Paper: Convergent Architecture & The ArcStyler Tool Suite," 2000.
[10] Rodney Bell, "Code generation from object models," *Embedded Systems Programming*, vol. 11, no. 3, Mar. 1998.



Axel Uhl received a diploma degree in computer science from the University of Karlsruhe, Germany in 1995, where he specialized in distributed and parallel computing as well as in software engineering. For the next four years he was with Asea Brown Boveri (ABB) Corporate Research where he focused on Internet-enabling business processes. Since January 2000 he is with Interactive Objects Software GmbH where he works as a software architect in a team developing the architectural IDE *ArcStyler*^{TM} and pursues a PhD study in the area of software architectures for scalable Internet search.



Horst Lichter received a diploma degree in computer science from the University of Kaiserslautern, Germany. Then he was a member of the software engineering group at ETH Zurich and University of Stuttgart. After receiving a PhD degree from the University of Stuttgart he was with the Union Bank of Switzerland, Zurich, and the ABB Corporate Research Centre, Heidelberg. Since 1998 he is a professor for computer science at Aachen Technical University (RWTH Aachen) heading a research group focusing on software construction and quality assurance.