# A Multithreaded Java Framework for Information Extraction in the Context of Enterprise Application Integration

Stefan Kuhlins and Axel Korthaus
Chair of Information Systems III, University of Mannheim
D-68131 Mannheim, Germany
{kuhlins|korthaus}@uni-mannheim.de

**Abstract**

In this paper, we present a new multithreaded framework for information extraction with Java in heterogeneous enterprise application environments, which frees the developer from having to deal with the error-prone task of low-level thread programming. The power of this framework is demonstrated by an example of extracting product prices from web sites, but the framework is useful for numerous other purposes, too. Strong points of the framework are its performance, continuous feedback, and adherence to maximum response times. The description of the framework uses UML modeling techniques for visualizing multithreading. Moreover, we tackle Java problems of stopping running threads.

## 1. Introduction

The software application landscape of today's enterprises, their business partners, suppliers, and customers is often characterized by a mixture of heterogeneous systems that have to collaborate and exchange data in order to perform the required business functions. Thus, interoperability and enterprise application integration are the concepts of the hour.

Hence, a typical problem is the integration of disparate data sources into a single coherent framework for real-time reporting and detailed analysis. Usually, it is desirable to have multiple extraction processes perform concurrently in order to optimize the total time needed for this task. In this paper, we describe the design and an implementation of a Java framework that allows for the parallel extraction of data from different data sources with the help of so-called "Wrappers", controlled by a "Mediator" which is responsible for collecting and returning the results. Special wrappers for different resources provide resource compatibility by making sure that on the next level, i.e., the mediator's level, data appears in a unified format for further processing. Using our framework, it is quite easy to develop specific wrappers for the integration of legacy systems to extract their data for use in new applications. In order to provide efficient data query functionality in real-time, the wrappers work concurrently, because time consumption for network accesses usually exceeds the time needed for data processing by far, so that network accesses should always occur in parallel.

For the implementation of our framework, we chose the Java programming language, because it provides very comfortable features for implementing multithreading and networking functionality etc. However, stopping threads in Java is not trivial, because the `stop()` method of Java class `Thread` is inherently unsafe and is therefore deprecated [13], so that it should not be used anymore. This can be quite problematic, e.g., if a computation started by a thread seems to "hang" because it does not react to interrupt requests. One example of such a situation is the improper use of regular expressions for searching and extracting information in a web-based environment. If the data source is modified and does not match the structure assumed by the regular expression any longer, the evaluation process might employ a backtracking strategy resulting in extremely long response times [4]. In the design of our framework, we have followed an interesting solution to lessen this Java problem, described by

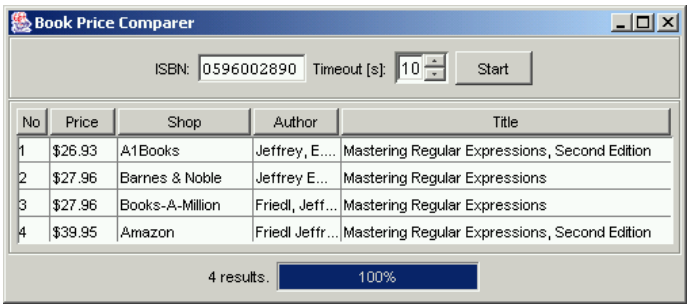


**Fig. 1: Screenshot of our Book Price Comparer example application**

work, we have followed an interesting solution to lessen this Java problem, described by Lea [10].

As an example of the practical use of our framework, we present a simple application of the framework in a web-based environment, where the different data sources provide web front ends. The problem domain we focus on is the well-known field of price comparers (shopping bots), i.e., tools that extract price information from HTML pages offered by different shop web sites [6, 2]. A screenshot of our example application is shown in Fig. 1.

## 2. Requirements for the Framework

A basic requirement for our information extraction framework is the possibility for users of the framework to specify maximum response times. Wrappers that do not deliver any results within a specified period of time are to be stopped. However, as mentioned before, stopping threads in Java is not trivial, because the stop() method of class Thread is deprecated. The reason for this is that "*stopping a thread with* Thread.stop *causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked* ThreadDeath *exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior.*" [13] Therefore, we had to find another way to stop threads in a safely manner to be able to meet the requirement mentioned before. However, since this is an inherent Java problem, there is no solution that works for all circumstances, especially if library methods are involved.

Results collected by wrappers are to be presented to the user immediately without delay, so that the user receives an adequate feedback about the processing state. The mediator which controls all the wrapper threads should not perform a "busy wait" to get the results of the information extraction. Instead, we use a special iterator which allows to receive the results of wrappers immediately, as they become available.

## 3. Related Work

Our paper basically describes a new combination and further development of existing approaches from different areas. For example, one of the first contributions to the field of wrapper/mediator structures was a fundamental paper titled "Mediators in the Architecture of Future Information Systems" by Wiederhold [14]. The special problem domain employed to illustrate the use of our framework in this paper is information extraction from the World Wide Web. A similar approach referring to information extraction from heterogeneous databases can be found in [5]. Nice surveys of existing wrapper-based techniques are provided in [3] and in [9]. The concepts described in wrapper-based approaches are often manifested for practical use in so-called "wrapper toolkits" (see for example an overview in [7]) and concrete

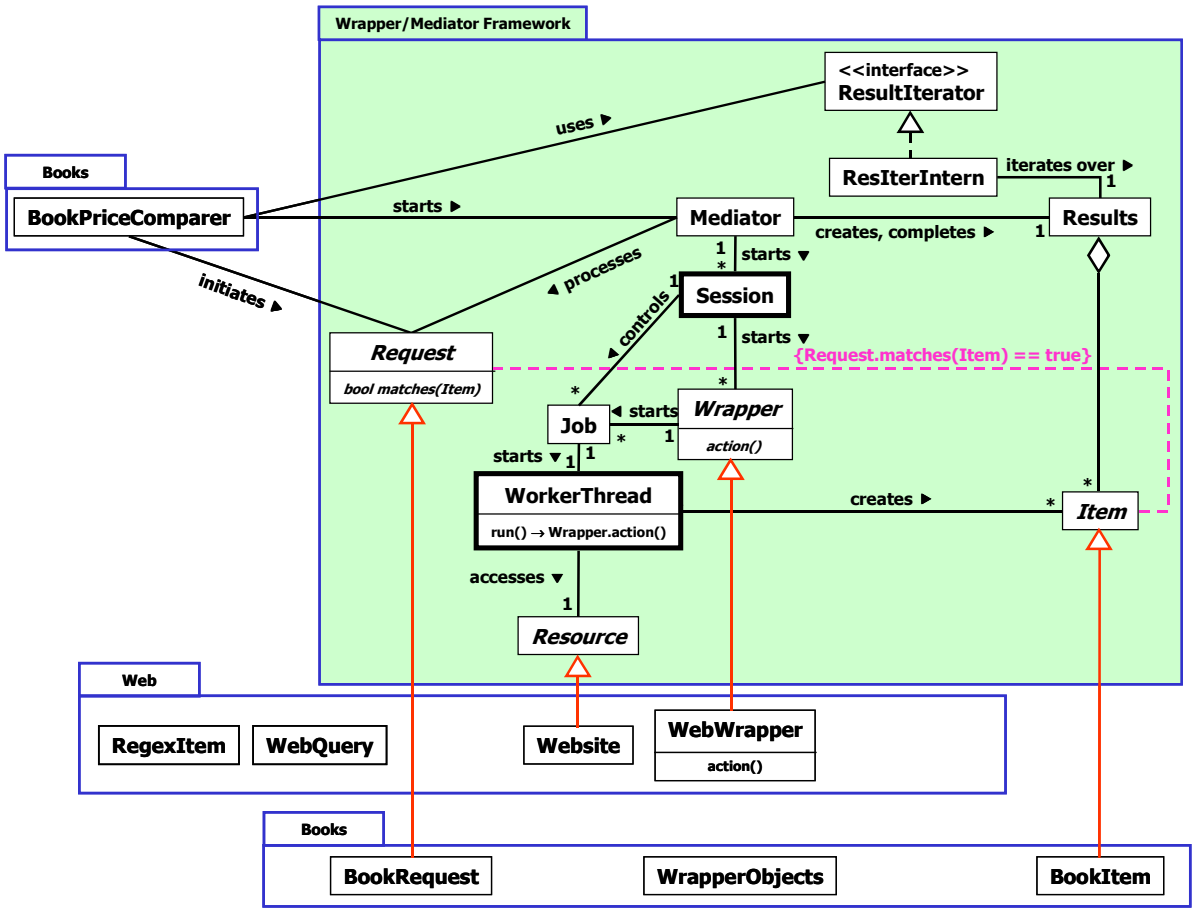


**Fig. 2: Simplified class diagram of the information extraction framework**

products such as shopping bots (see for example [2], where an interesting learning algorithm for shop wrappers is described). However, since there are several proprietary commercial solutions, source code or design patterns for these systems are generally not available, which was one of the motivations we had for writing this paper. Among the resources providing technical basics for this work were [10], where multithreading in Java is described in detail, and [12] which contains information about modeling threads with the Unified Modeling Language.

## 4. Design and Implementation of the Framework

Fig. 2 shows a UML class diagram depicting the constituent classes of our Java-based information extraction framework and user-defined classes implementing our specific problem domain. A simple exemplary dynamic scenario of our Book Price Comparer application is modeled in the UML sequence diagram of Fig. 3.

The central class in the framework is Mediator. The mediator manages wrapper objects, i.e., for every request the mediator creates a Session object, which starts and controls wrappers or their jobs, respectively. After a specified time period, wrapper jobs that have not delivered their results are stopped (see below). The mediator sessions register as Listeners with associated wrappers, providing the completed(Wrapper.Job) method. This method represents a callback method for wrapper jobs, or their threads, respectively, to indicate that the wrapper's data extraction work is done. In this case, the wrapper job will be added to the list of finished wrapper jobs. Using the start(Request, int) method, a new request can be started

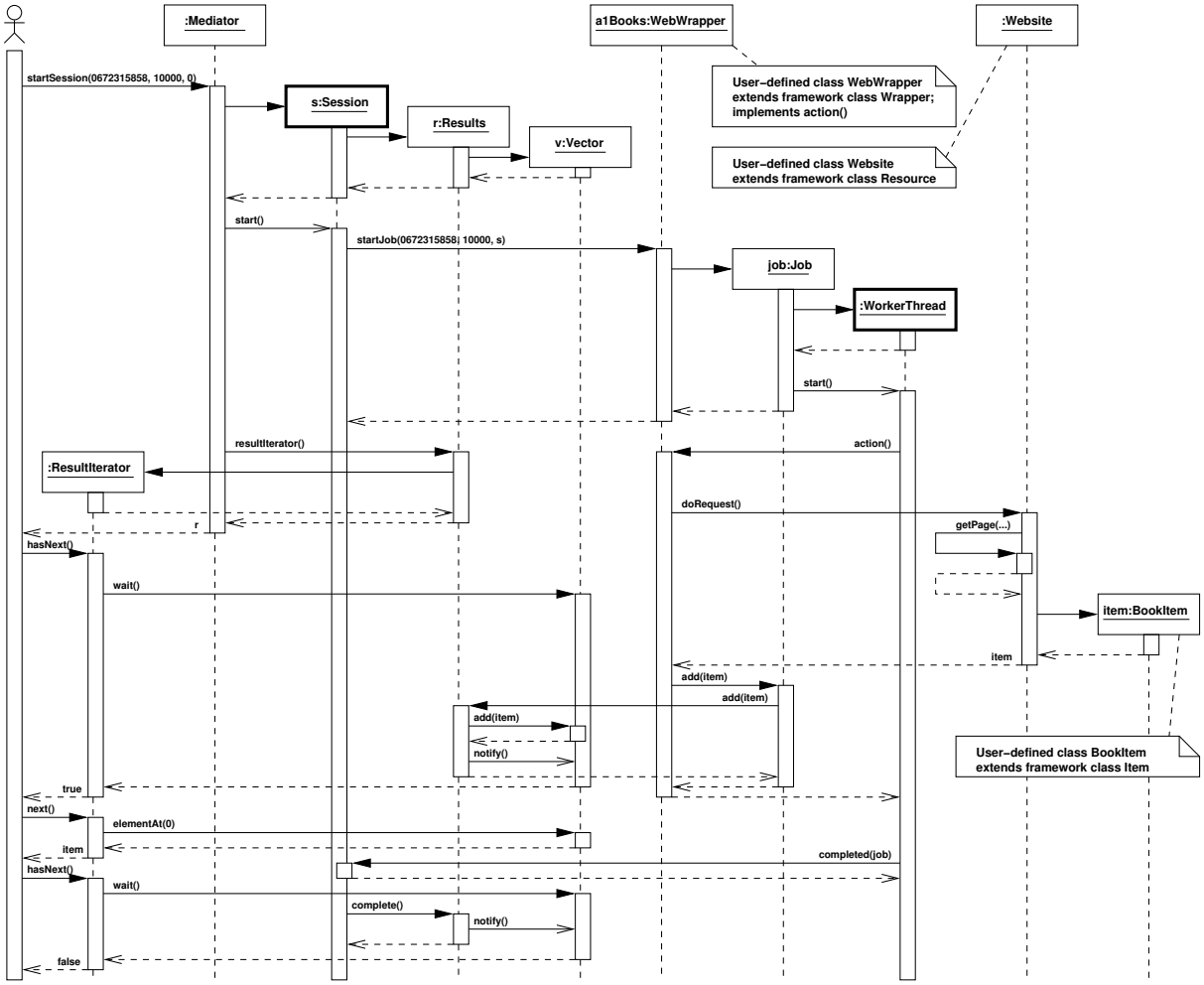


**Fig. 3: Simplified sequence diagram of an example scenario**

which may not exceed the period of time passed as the second argument, and the results of the request are accessible through the returned Results object. The result list could be traversed by a ResultIterator immediately. For this purpose, the hasNext() method waits for results, if necessary.

The abstract Request class is used to represent a request directed to specific resources, i.e., data sources. A request may refer to anything like, for example, books, digital cameras, stocks, news etc. The matches(Item) method checks whether an actually found item matches the requested properties.

The abstract Item class is a superclass for concrete classes instantiating objects, which are received by wrappers from resources due to specific requests. The subclasses implement entities such as books, digital cameras, stocks, news etc. To be able to sort the result lists of items, subclasses have to implement the Comparable interface. The getResource() method of the Item class returns the resource the item object originates from.

The abstract Resource class describes resources, which serve as a provider of items requested by wrappers. Potential subclasses include, for example, web sites, databases, news tickers etc. The name of the resource can be determined by calling the getName() method.

Objects of subclasses of the abstract Wrapper class are used to direct requests to a resource and to return the results subsequently. The wrappers activity is started by an invocation of the startJob() method, which returns a newly created Job object. Using stopRunning(), the mediator session can try to stop the Job's WorkerThread prematurely. We will show the implementation of the respective mechanism below.

A wrapper job object accesses its wrapper's resource object and stores references to the request to be handled and the result list where it inserts items returned from the resource. A wrapper job uses a worker thread (inner class WorkerThread) to become active having its own flow of control. The worker thread triggers the actions specified in the action(Wrapper.Job) method. Within this method, new elements are added to the result list by a call to the add(Item) method.

This result list of items is managed by objects of the Results class, mainly by delegating to a subordinate Java Vector. Most methods are synchronized to enable concurrent accesses. As long as the result list is not complete, new objects can be inserted by an invocation of the add(Item) method. Whether the list is complete or not is indicated by the result of isComplete(). As the name suggests, the waitUntilComplete() method waits until the result list is complete. In case this method has been called, the final result list can subsequently be sorted. There are two sort methods for rearranging the result list, based on the comparison method for items or based on a specific comparator object, respectively. Also, the result list can be traversed with a usual Iterator returned by the iterator() method. The number of objects contained in the result list can be determined by calling size(). For immediate navigation through the result list a call to resultIterator() returns a special ResultIterator object. This iterator, implementing interface ResultIterator which is similar to Iterator, can be used to iterate over the elements of the mediator session's result list even if the whole data extraction process is not complete, i.e., wrapper jobs are still inserting results. The basic idea for the iterator is similar to that described in [8].

If a mediator session wants to stop a wrapper job, the first try used to be the invocation of the stop() method of class Thread. However, since this method is now deprecated, there is no forcible way of stopping ongoing activities of a thread. A workaround to this problem has been presented by Lea [10]. Lea's approach uses "*a generic multiphase cancellation facility that tries to cancel tasks in the least disruptive manner possible and, if they do not terminate soon, tries a more disruptive technique.*" For our framework, we have combined Lea's proposal with a technique described by Chan [1]: "*The proper way to stop a running thread is to set a variable that the thread checks occasionally. When the thread detects that the variable is set, it should return from the run() method.*" The result is the cancel() method of class Wrapper.Job that must be called if the thread has to be stopped:

```
boolean cancel() {   // of class Wrapper.Job
    if (!workerThread.isAlive()) return true;
    stopRunning();   // set Job.running to false (according to Chan – see text)
    workerThread.interrupt();   // try to interrupt and wait a bit
    try { workerThread.join(100); } catch (InterruptedException ignore) {}
    if (!workerThread.isAlive()) return true;
    workerThread.setPriority(Thread.MIN_PRIORITY);   // minimize damage
    return false;
}
```

## 5. Conclusion and Future Work

In this paper, we have presented the design and an implementation of a generic Java-based framework for the concurrent extraction of data in a heterogeneous environment. The framework fits especially well for collecting data from different web sites, as has been demonstrated by an example implementing a price comparison shopping bot extracting price information from HTML pages, which could also be done by calling suitable web services if available, as for example at Amazon's servers. However, there are countless other application domains for the framework, as for example

- querying stock, bond and share quotes in order to be able to react to specific conditions (e.g., by sending an email if a specified limit is exceeded etc.)
- querying news tickers (economy news and stock quotes can be combined)
- querying several different databases on different platforms (see, for example, the GARLIC system described in [11])
- querying other systems based on CORBA, RMI, etc.

Generally, our wrapper/mediator framework for information extraction can be employed in any case where several different systems have to be queried in parallel. We have explained how a common problem of Java thread programming can be avoided by applying suitable workarounds. The basic structure and the dynamic behavior of the framework have been illustrated using UML class and sequence diagrams. Our future work on the framework will include the integration of different new kinds of functionality, such as logging, caching etc. in order to extend and further improve the services provided by our solution.

## References


[1] Chan, P. (2002): The Java Developers Almanac 1.4, Volume 1: Examples and Quick Reference, e93. Stopping a Thread, http://javaalmanac.com/egs/java.lang/StopThread.html

[2] Doorenbos, R.B., Etzioni, O., and Weld, D.S. (1997): A Scalable Comparison-Shopping Agent for the World-Wide Web, in: Proc. ACM Conf. Autonomous Agents, ftp://ftp.cs.washington.edu/pub/etzioni/softbots/agents97.ps

[3] Eikvil, L. (1999): Information Extraction from World Wide Web – A Survey. Norwegian Computing Center, P.B. 114 Blindern, N-0314 Oslo, Norwegen, Rapport Nr. 945

[4] Friedl, J.E.F. (2002): Mastering Regular Expressions, 2nd edition, O'Reilly & Associates

[5] Hull, R. (1997): Managing Semantic Heterogeneity in Databases – A Theoretical Perspective. Tutorial. Bell Laboratories. Lucent Technologies. http://www.db-research.bell-labs.com/user/hull/pods97-tutorial.html

[6] Krulwich, B.T. (1996): The BargainFinder Agent – Comparison Price Shopping on the Internet, in: Williams, Joseph (ed.): Bots and other Internet Beasties, Sams.net Publishing (Macmillan), pp. 257–263

[7] Kuhlins, S. and Tredwell, R. (2003): Toolkits for Generating Wrappers – A Survey of Software Toolkits for Automated Data Extraction from Websites, in: Aksit, M., Mezini, M., and Unland, R. (eds.): Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays (NODe 2002), Oct. 7-10, 2002, Erfurt, Germany, Lecture Notes in Computer Science (LNCS 2591), Springer, pp. 184-198, http://www.wifo.uni-mannheim.de/~kuhlins/paper/ wrapper.pdf

[8] Kushmerick, N. (1998): (Toward) an Extensible Wrapper Repository Standard, in: Proc. Workshop on AI & Information Integration, AAAI-98 (Madison), http://www.cs.ucd.ie/staff/nick/home/research/ download/kushmerick-aaai98-aiii-panel.ps.gz

[9] Kushmerick, N. (2002): Gleaning Answers from the Web. Position paper, AAAI 2002 Spring Symposium on Mining Answers from Texts and Knowledge Bases.

[10] Lea, D. (1999): Concurrent Programming in Java – Design Principles and Patterns, Second edition, Addison-Wesley; "Multiphase cancellation": http://gee.cs.oswego.edu/dl/cpj/cancel.html

[11] Roth, M.T. and Schwarz, P. (1997): A Wrapper Architecture for Legacy Data Sources. IBM Almaden Research Center.

[12] Schader, M., and Korthaus, A. (1998): Modeling Java Threads in UML. In: Schader, M., and Korthaus, A. (eds.): The Unified Modeling Language – Technical Aspects and Applications. Physica, Heidelberg, New York, pp. 122-143

[13] Sun Microsystems (2003): Java 2 Platform, Standard Edition, v 1.4.2, API Specification, Class Thread, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html#stop()

[14] Wiederhold, G. (1992): Mediators in the Architecture of Future Information Systems, in: IEEE Computer, 25(3), pp. 38-49, http://www-db.stanford.edu/pub/gio/1991/afis.ps