uWIRES: a Software Framework for the Rapid Development of Web-Related Tools

Petros A. Demetriades[a,b]
[a]King's College London
The Strand, London WC2R 2LS, UK
petros@dcs.bbk.ac.uk

Alexandra Poulovassilis[b]
[b]Birkbeck, University of London
Malet Street, London WC1 7HX, UK
ap@dcs.bbk.ac.uk

ABSTRACT

This paper presents uWIRES, a framework that aims to facilitate the rapid design and development of web-related tools by providing an architectural layout, a set of design and development guidelines, an information model and a comprehensive class library. uWIRES has been used to develop a number of tools to support our research into visualisation of the web, including WebIR2, an end-user meta-search tool evaluated in a real-world context by 25 evaluation participants over a period of 4 months. We discuss our experiences of using uWIRES for the development of these tools and present evidence indicating that uWIRES can indeed meet its design goals and objectives of enabling the rapid development of production-quality web-related tools.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures–domain-specific architectures; D.3.3 [Programming Languages]: Language Constructs and Features–frameworks; D.2.13 [Software Engineering]: Reusable Software–reusable libraries, Java.

General Terms

Design, Experimentation, Standardisation, Performance.

Keywords

Component Architecture, Software Framework, Web Tools.

1. INTRODUCTION

The World Wide Web has become a ubiquitous information dissemination, communication, entertainment and commerce resource with an ever growing user base. These users (the number of which is currently estimated to be greater than one billion, with the two billion milestone expected in 2011 [1]), depend on the web to satisfy a wide variety of daily information needs. An integral task of much current research into web technologies is the development of software tools to prototype some new technique (e.g. web visualisation, web search, web data mining), or to determine web-related metrics (e.g. size and growth of the web, overlap of search engine indexes, extent of coverage of search engines, freshness and age of web documents), or more generally to test and evaluate new web-related algorithms and hypotheses. Many such tools have similar functional needs including: collection of the required data; temporary, and possibly persistent, storage of data collected; processing of the data collected (for example analysis, synthesis, aggregation and derivation of new data, frequently with multiple dependent processing stages); display and visualisation of the data; and interaction with users.

Furthermore, especially if they are end-user tools, tools are likely to have significant non-functional requirements such as portability, high performance, highly responsive user interfaces, robust error-handling, adequate logging of errors, comprehensive instrumentation to record data that (after appropriate analysis) would enable conclusions relating to the research for which they are developed to be reached, and so on. Not only are such non-functional requirements non-trivial and do they demand substantial design and development effort, but they are also likely to be so similar across tools that they could be satisfied by identical code.

A final, possible common characteristic, is that the experimental nature of many of these tools could require a certain degree of "trial and error" with different approaches to implementing some new algorithm or paradigm of interacting with the web. To complicate matters further, such different approaches may only become apparent after an initial working version of the tool is produced rather than at the outset when the tool is being designed.

We encountered such a situation while undertaking research on visualisation of the web: we required a number of tools to use as test-beds for our research, including a production-quality end-user tool to be used for formal evaluation of our proposed techniques. At the outset of our research, there were many unknown details such as the nature of the components that our tools would require, the data that they would need, what the sources of the data would be, effective ways of visualising the data and which type of data model (e.g. relational, graph-based, object-oriented, hierarchical) would be best suited for modelling the web and persisting the appropriate data. These uncertainties presented two main challenges: Firstly, each component could not be developed in isolation without considering the other potentially necessary components and, therefore, thought needed to be given to an overall system architecture. Secondly, the amount of work that would be required to design, build and test each component and different implementations of similar components would have been substantial and could have exceeded the time available to undertake the research and possibly affected the currency and timeliness of conclusions drawn.

In order to resolve these issues, we investigated the availability of existing frameworks and code libraries to use as a starting point. We searched for frameworks that could help resolve these two issues, that provided a suitable "best practice" architecture, and that satisfied if not all at least the majority of the non-functional requirements described above. We found an abundance of general approaches to component-based architectures, frameworks and class libraries for facilitating user interface development and for building systems that target specific aspects of interaction with the web. These include: FLAIR [2], one of the earliest frameworks for building general user interfaces; FIRE [3], an IR framework that focuses on providing re-usable indexing and retrieval facilities; InfoGrid [4], a framework for building IR applications that provides a UI design and an interaction model; and Terrier [5], a framework for building high performance and scalable IR systems which focuses on providing indexing and retrieval facilities with associated features such as pseudo-relevance feedback. However, none of these were suitable for our purposes, for a number of reasons, including:

- they were not sufficiently general and thus not well-suited or applicable to our needs;
- many were no longer available (i.e. it was not possible to obtain a copy of the compiled frameworks or source code for most of the ones referred to in the literature);
- they were developed in non-readily portable languages that do not support "compile-once-run-anywhere" as Java does (for example Lisp or C++);
- they focussed on very specific areas such as user interface creation or web indexing and querying.

The apparent lack of an appropriate and readily available framework motivated the design and development of our End-**u**ser **W**eb **I**nformation **RE**trieval **S**upport [1] (uWIRES) framework. Features of uWIRES are that it:

- specifies an application architecture;
- meets all the functional requirements stated above;
- is based on a class hierarchy that promotes re-use and enables changes to be easily propagated to the entire hierarchy;
- specifies interfaces between components and decouples components as much as possible, in order to minimise the ripple effects of change and to enable the use of components or services that cannot be statically linked into a tool;
- provides comprehensive data management facilities, and models the data entities that a typical web-related tool requires;
- incorporates a broad range of architectural and infrastructural services;
- makes use of existing "off-the-shelf" components and code libraries to minimise development effort;
- is readily available and aims to be of use to researchers and developers working on a wide range of web-related topics.

In Section 2 we discuss the design goals of uWIRES, outline its architecture and the facilities it provides, and describe a set of development and design guidelines which informed the design and development of the framework and which we recommend to others wishing to develop tools using uWIRES. In Section 3 we detail the uWIRES class library and some of the more significant technical and implementation details. In Section 4 we discuss to what extent uWIRES meets its intended design goals and objectives and can assist the rapid development of web-related tools. Section 5 concludes and briefly describes future plans and research directions.

2. THE uWIRES FRAMEWORK

The uWIRES framework consists of:

- an architectural layout, which the framework itself follows, and which should be followed by tools built using the framework;
- a set of design and development guidelines for developers;
- a class library (containing a total of 7,776 lines of code, 58 classes and 843 methods);
- an extensible information model that provides comprehensive data management facilities and a default set of data entities.

[1] The name of the framework reflects the fact that it was developed as part of a research project that was investigating visualisation of the web in the context of meta-search. However the framework is more generally applicable to web-related tools.

We begin our description of uWIRES by first discussing, in Section 2.1, the design goals that guided its development. Section 2.2 then describes the architectural layout, Section 2.3 the design and development guidelines, Section 2.4 the class library and Section 2.5 the information model.

2.1 Design Goals

The framework was designed with the following goals in mind, so as to meet the objectives discussed in Section 1 and to ensure that it can be applied to other situations where the rapid development of web-related tools is required:

1. *Minimise development and maintenance time*: Given the potentially substantial amount of functionality that a tool may require and the typically limited amount of time available, the framework should serve to decrease the overall amount of development by, among other things, facilitating the use of existing code that offers desired functionality, enabling the development of constituent system components by more than one person, and minimising the ripple effects of change.
2. *Facilitate experimentation*: The architecture should facilitate experimentation with different approaches for achieving some objective.
3. *Deliver high performance*: In order to avoid a situation where the positive effects of proposed solutions or novel ways of achieving some goal are masked by slow-performing software, the architecture should encourage practices that promote high performance and should simplify multi-threading and multi-processing.
4. *Facilitate incremental visualisation of data*: In order to improve performance and reduce users' "idle waiting time", the framework should facilitate the incremental visualisation of data so that users can start to view output and interact with tools as soon as some data is available (as opposed to being forced to wait until all processing stages are entirely completed on the whole dataset).
5. *Be scalable*: The framework should be able to cope with large amounts of data so that tools could be put to use in real-world scenarios.
6. *Support development of both interactive and non-interactive tools*: The framework should be applicable to the development of interactive user tools but also to non-interactive tools, e.g. data gathering and analysis tools.
7. *Be portable*: The framework should be portable to other platforms so that it could be used irrespective of platform choice, and that tools developed using it could be deployed to multiple types of platforms.

2.2 Architectural Layout

In order to meet design goals 1 and 2 above, the framework needed to be modular, to encourage the development of systems with a modular architecture, and to provide clearly defined types of components and interfaces between them so that components could be decoupled and made independent as necessary.

By analysing the targeted functional requirements discussed earlier, it was evident that the framework should provide four classes of components specifically tailored to (i) data collection, (ii) data processing, (iii) temporary and persistent data storage, and (iv) data visualisation or user interaction. This led to the architectural layout shown in Figure 1.

In this architecture, **Collectors** are components that gather

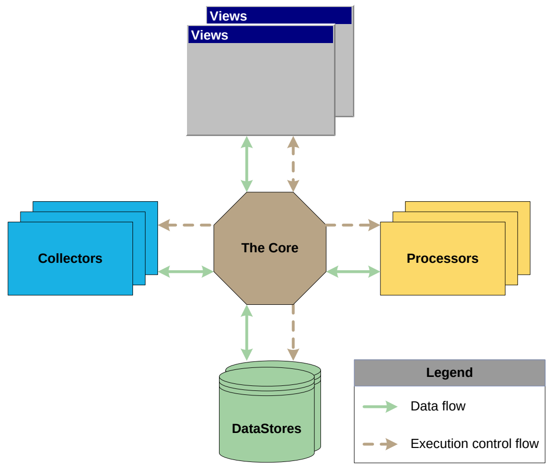

Figure 1: uWIRES Architectural Layout

required data such as search results, webpage contents, website maps, and so on. **Datastores** are components that store data temporarily in transient in-memory structures, persistently in appropriate databases conforming to defined models and schemas, or both. **Processors** are components that process the data gathered by collectors in ways that help achieve the objectives of a tool, e.g. ranking search results. **Views** can be visible components that display data and serve as user interfaces, or non-visible control execution components that guide a tool through an automated series of data collection and processing steps.

Multiple such components can exist in a tool, all operating simultaneously. In the case of datastores they can be implementations of entirely different data models (e.g. relational, graph-based, object-oriented, etc) and can store the same data simultaneously in the different data models. In the case of views, they can display different visualisations of the same data and can be synchronised.

As its name suggests, the **Core** component is central to the uWIRES framework and to tools built using it. The core can be thought of as the middleware that binds together all the other components. It was introduced in order to meet design goals 3 and 6 and to further facilitate design goals 1 and 2. The core provides the following services:

- *Tool composition*: The components comprising a tool are registered with the core, which then fully takes over their control.
- *Tool start-up and shutdown*: Once components have been registered, the core performs the appropriate system start-up sequence. At the request of a view (e.g. user chooses to exit the tool) or if a fatal system error occurs, the core shuts down the system.
- *Inter-component interaction and communication*: Acts as a hub through which components can interact and communicate with other components. This interaction is either via events or via Application Programming Interface (API) calls.
- *Standard control flow for commonly used operations*: Incorporates functions that implement the standard control flow for web-related operations that may be commonly used by more than one component or different experimental implementations of particular components. For example, downloading the contents of Uniform Resource Locators (URLs).
- *Tool and component preferences repository*: Provides a centralised repository of preferences and settings.
- *Deployment control*: Incorporates functions that enforce expiry of tools (e.g. by a certain date) in order to prevent widespread distribution of superseded experimental or prototypical versions of tools and to ensure that users are always using the latest versions.

The uWIRES architectural layout borrows from the Model-View-Controller (MVC) [6] classic design pattern which is often used to guide the design of interactive applications. MVC partitions applications into three separate components: models for maintaining and storing data, views for displaying the data and accepting user input, and controllers either for handling events or for dispatching events and controlling execution flow. In some respects, uWIRES can be considered a specialisation of MVC. uWIRES differs from MVC however in that it is not just a design pattern but a concrete application framework targeted specifically to the development of web-related tools that are fast, scalable, portable, and highly-interactive and responsive.

Although some of the research into uWIRES predates them, uWIRES also borrows from other component technologies, such as Enterprise JavaBeans (EJB) [7] and Component Object Model (COM) [8], in that it enables independently developed components to be integrated into a single system through registration with the core, which then proceeds to initialise, bind them and begin executing them as a single system. Unlike EJB, and COM technologies, however, uWIRES is specifically targeted towards research and development of web-related tools. The uWIRES components can only be one of the five specific types described above and must be derived from one of the uWIRES component templates.

2.3 Design and Development Guidelines

In addition to adhering to the architectural layout depicted in Figure 1, we recommend that tools developed using the uWIRES framework should follow, as much as possible, the design and development guidelines described below. The design and development of the framework's class library itself followed these guidelines, and adherence to these guidelines will increase the degree to which the framework's design goals are met.

1. *Java as the programming language*: To the extent possible, Java should be used for development of all parts of a tool (although this is not absolutely compulsory as components written in other languages can be integrated via the use of Java wrappers and JNI).
2. *Components should be independently executing entities*: Each instance of a component should be able to execute in its own thread as a stand-alone executing entity and should not assume execution within the thread of some other component or the core.
3. *Concurrency synchronisation at the data level*: As multiple threads will be accessing and acting upon the same data, access to this data must be controlled to ensure that no two threads attempt to modify the same data simultaneously and no thread attempts to read some data that is being updated by some other thread.
4. *Inter-component communication only through the core*: As indicated by the architectural layout, there should be no direct data or control flow communication between components except through the core. The core's public API and the inter-component messaging and data exchange facilities provided should be used for this purpose.
5. *The functional segregation of components in the architectural layout should not be violated*: The architectural

layout implies that each component must perform a specific type of function. Although the framework encourages this functional segregation, it does not include any controls to enforce it and a tool developer could choose to disregard it. Violation of this functional segregation should be avoided as it can negate some of the benefits of using this framework to build a tool (see discussion below).

6. *Avoidance of platform-specific functions, services, and components*: No platform-specific functions and services should be used if possible. Where this cannot be avoided, a Java wrapper to the native functions, services or components should be created.
7. *Incorporate instrumentation*: Comprehensive instrumentation should be incorporated to facilitate testing, debugging and effectiveness evaluation of tools.

We now summarise some of the ways in which the above guidelines help meet the design goals of Section 2.1:

1. *Java as the programming language*: Java is arguably the most portable software development language available today, and comes with a very rich class library which can reduce development effort and thus timescales. There are a vast number of free and open-source code libraries which can be used to further reduce development times.
2. *Components should be independently executing entities*: This allows components to be executed in separate threads or processes, thus enabling different tasks to be performed concurrently. Since many of the delays of a web-related tool are likely to be with network access (e.g. waiting for a website to respond or a page to download), rather than long computations, multi-threading increases performance.
3. *Concurrency synchronisation at the data level*: This is not only dictated by the fact that the framework promotes multi-threading, but can also prevent obscure concurrency-related defects and significantly increases performance. One approach to concurrency synchronisation is to use database concurrency control mechanisms, such as locking. But this would reduce flexibility as it would require a datastore that supported locking. Another way would be to ensure that all functions that access or modify data reside in a single class and are all synchronised. However, this would be rather crude as the entire object in which these functions reside would be locked whenever any single function executed and only one such function would be able to execute at any time. For a heavily multi-threaded tool, this could decrease performance as the multiple threads would compete for access to the object lock. An alternative approach that eliminates these issues is to use synchronised blocks of code within the functions that read and modify data, at the precise locations where such reads and modifications occur, using different lock objects (i.e. objects on which the locks should be acquired) for each distinct data entity. This approach increases performance and also facilitates incremental data visualisation as it prevents the display of data that is in the process of being updated.
4. *Inter-component communication only through the core*: This prevents tight coupling between components, which in turn minimises the ripple effects of change and allows entire components to be easily replaced by other experimental components (e.g. alternative processor or view components). It also facilitates the incremental visualisation of data by enabling the core to intercept data updates (irrespective of which component initiated them) and to issue appropriate "data update events" to other components as and when new data is available. This arrangement also increases scalability by enabling components to be executed on a different physical machine, with the core component taking care of all the necessary inter-process and inter-machine communications. Finally, it also facilitates portability as unavoidable platform-specific code could be isolated within one or more components (with an appropriate wrapper to act as the interface with the core), thus eliminating the need for other components to have any knowledge of where a component is executing or the type of platform on which it is executing.
5. *The functional segregation of components in the architectural layout should not be violated*: The proposed functional segregation encourages decoupling, which in turn can minimise the ripple effects of change. It can simplify experimentation (e.g. alternative ways of collecting, processing, storing, or visualising data) as each of these "functional groups" exist in separate "modules" and are thus easily replaced by new experimental modules. Furthermore, as the user interface exists in a separate module (a view component), this could easily be replaced by a non-interactive view component which simply instructs the system to perform certain actions through appropriate command events. Such a view could, for example, be used to perform an analysis of the overlap of search results by different engines by executing, without any user intervention, several hundred or thousand queries and determining the number of common results. Similarly, a non-interactive non-visible view could be used to automatically test a tool by simulating the actions of a user issuing commands and interacting with the tool.
6. *Avoidance of platform-specific functions, services, and components*: Greater portability can be achieved by using Java functions, services and components, and avoiding platform-specific equivalents.
7. *Incorporate instrumentation*: Comprehensive debugging instrumentation (e.g. logging and tracing) can reduce the time needed to troubleshoot and correct defects and other issues.

2.4 Class Library

The uWIRES class library is one possible implementation of the architectural layout described in Section 2.2. It was implemented in Java and follows the guidelines described in Section 2.3. Figure 2 lists all the classes in the uWIRES class library as well as the hierarchical relationships between them.

The services and functions provided by these classes are grouped into six categories:

1. *Component templates*: These are either concrete classes which implement components that can be instantiated, or abstract classes and interfaces which provide the templates (and specify the methods that must be implemented) from which component classes can be derived.
2. *Inter-component messaging and communication*: Classes that enable communication between components even if these components are running in separate threads or processes.
3. *Data management and default web-related data entities*: Classes providing data management functions that tools may typically require (such as reading, inserting, updating, caching, looking-up, bulk loading, sorting, filtering and so on). They also model a set of default data entities that a typical web-related tool would require.
4. *Common web-related services*: Classes which provide

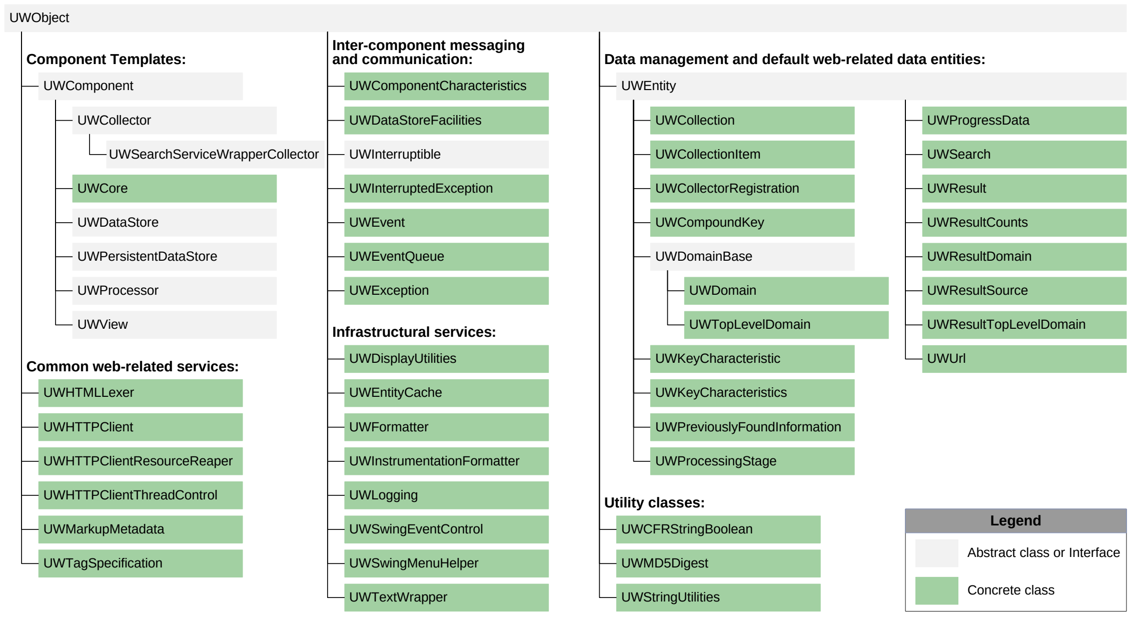

Figure 2: The uWIRES class library

services such as HTTP clients and HTML Parsers.

5. *Infrastructural services*: Classes which provide fundamental infrastructural services such as error handling, logging, thread management and other convenience functions.
6. *Utility classes*: Classes implementing frequently needed functions not available in the standard Java libraries, for example calculation of MD5 digests and some advanced string manipulation functions.

2.5 Information Model

uWIRES incorporates a comprehensive architecture for data management as well as a set of default data entities that web-related tools are likely to require. The internal model that uWIRES employs to store and manage data can be regarded as object-oriented. Each primitive data entity is modelled as a single class. Complex entities can be composed from primitive entities and treated as individual entities (rather than as a collection of separate primitive entities). Where this is done, the primitive entities that form the complex entity may be linked to the complex entity via entity id referencing, or they may be fully embedded within the complex entity – it is up to a tool developer to determine which approach is best suited to a given purpose. The linked approach is advantageous as it means that the primitive entities may be cached in memory (irrespective of whether the complex entity is cached) thus helping to meet design goal 3 by improving performance; memory utilisation would also be lower as the proliferation of identical copies (object instances) of the same entity is avoided thus helping to meet design goal 5.

3. CLASS DETAILS

This section describes the purpose of the main classes in the uWIRES framework, the services provided by them and some significant technical implementation details. We begin by describing the ancestor of all classes (the UWObject class) in Section 3.1. Section 3.2 describes the *Component Template* classes which either implement concrete components that can be instantiated "as is" or abstract classes from which each of the five component types in the framework must be derived. Section 3.3 describes the classes that implement the *Inter-component Messaging and Communication* facilities. Section 3.4 describes the classes that provide *Data Management* facilities and a set of *Default Web-Related Data Entities* that implement the uWIRES information model. Section 3.5 describes the classes that implement a number of commonly needed web-related services. Finally, Section 3.6 describes the classes that implement the *Infrastructural Services* and *Utility* functions.

A comprehensive description of all the classes, attributes, methods and facilities provided by uWIRES can be found in the API documentation accompanying uWIRES available from http://www.dcs.bbk.ac.uk/~petros.

3.1 UWObject Class

The **UWObject** class is the abstract parent class of all classes in the framework. It enables the propagation of attributes and methods to the entire class hierarchy and to every custom class of a tool that is derived from UWObject. Some of the services it provides are for logging, error handling, abnormal termination, and many convenience methods for object comparisons.

Comprehensive logging services are provided that simplify debugging by supporting global, class-specific and log-level or keyword-based[2] logging. Three conceptual logs are made available to each object: a global log, a class log, and an evaluation log. The class and global logs are intended for error and general debugging messages while the evaluation log is reserved for data related to the evaluation of a tool[3] or for metrics collected by a tool. The presence and logging level for the class and global logs are at the control of developers. Irrespective of the number of objects in a tool, all log output is written to two files, one for entries related to evaluation of a tool and one for everything else. Both logs have a consistent format and can easily be loaded into a spreadsheet or database application for analysis.

[2] Keyword-based logging is very helpful when debugging issues or defects that are specific to one area of functionality as only log entries with a keyword matching globally registered "log keywords" will be stored in a log file.

[3] Storing evaluation-related data in this log rather than a persistent datastore, facilitates harvesting of these logs especially if the tool is distributed to many remote evaluation users.

3.2 Component Templates

The **UWComponent** class is the abstract parent of all classes that implement the five architectural components of the framework, namely the core, views, collectors, processors, and datastores. This class provides the following three common facilities to all components that extend it.

(i) Component identification
Encapsulation of data elements that identify the component type (collector, store, view, etc), its name, and its group. Each component must be given a unique name and must belong to one group. Both are used primarily to facilitate communication and interaction with components (e.g. for sending events to a specific component or a group of components at once).

(ii) Component events infrastructure
This class furnishes each component (via use of the **UWEventQueue** class) with a thread-safe, FIFO event queue, and appropriate methods for event submission and retrieval. This enables components to communicate with other components through events that broadcast commands, state information, or information relating to data updates to all components, a group of components or a specific component. Furthermore, it provides a full implementation of an event loop method, the `run()` method, which continuously checks for new events and dispatches them to appropriate (abstract) event handling methods. This arrangement means that derived classes must implement each of the event handling methods but they need not worry about the event retrieval and dispatching mechanisms—this is taken care of by the framework.

(iii) Component execution control
In order to facilitate control of multiple independently executing components and to simplify component development, components can be in one of five execution states at any one time. Figure 3 shows these five states and the legal state transitions between them. State transitions are either automatic (i.e. dictated by the core or triggered by events received) or explicitly triggered by components. State transitions and the determination of component states is done exclusively by the framework: although components can issue events that can alter their state, they do not have direct control over the value of their state. This approach not only reduces the amount of code in derived components but also reduces the possibility of programming errors thus helping to meet design goal 1. The meaning of the five states is as follows:

- *Initialising*: This is the state that a component enters when it is constructed, while initialising and before it is instructed to begin executing its main event loop.
- *Active*: A component is in this state if it is performing its primary function. For example, a collector component would be in the active state while performing collection of the data it was designed to collect, but not while performing any other task such as event processing.
- *Paused*: A component is in this state when it has been instructed not to enter the active state even if events that would normally cause such a transition are received. Transition from the paused to the active state can only occur if an explicit "continue" command event is received.
- *Inactive*: A component is in this state whenever it is performing a task other than its primary function, for example, while it is idle waiting for events.
- *Terminated*: A component is in this state if it has exited its event loop and can no longer accept or process events.

The **UWCollector** class is the abstract class from which all

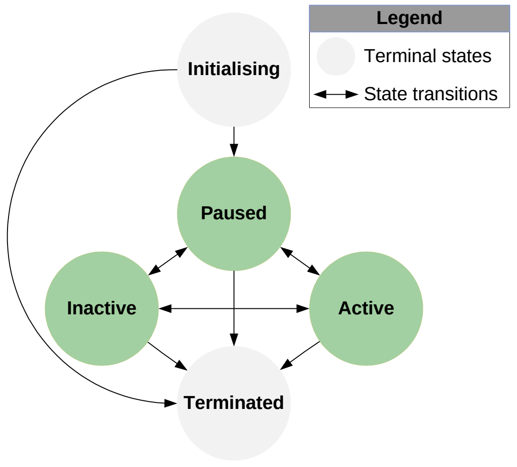

Figure 3: Component state transition diagram

collector components must be derived. It extends UWComponent by including collector-specific attributes and methods. These are:

- *Component type definition*: Definition of the component as a collector, thus avoiding repetition of this in every derived collector class;
- *Registration with a datastore*: Collectors must be registered with datastores before attempting to store any data. This is so that the datastore can create the appropriate data structures required to accommodate data from all collectors, but also so that the datastore can assign a unique identifier to each collector which is used to identify the originator of some data to the datastore in a more efficient way than using component names. This class performs this on behalf of all derived collector classes. (The **UWCollectorRegistration** data entity class encapsulates the data of these registrations).

The **UWSearchServiceWrapperCollector** class is the abstract parent class of all search result collectors (i.e. wrappers to search services). It provides several facilities that simplify the development of wrappers, such as URL generation (generation of the URL to obtain a specific page of results for some search), HTTP handling, search control flow (performing in the correct order all the steps required to obtain a page of results), and so on.

The **UWCore** class is a concrete class that implements the core component. Tools can either directly instantiate and use this class or they can extend it and override its non-final methods to create customised cores. The core provides the facilities discussed in (i) to (vi) below:

(i) Tool composition
In order to serve its function as a communications hub between components, the core must be aware of all the components comprising a tool. Therefore, a tool is essentially created by instantiating objects for each of its components and registering them with the core. That is all the explicit initialisation that a tool needs to do: all the rest is done by the core component using events and direct calls to methods inherited from UWComponent or more specific abstract sub-classes such as UWCollector.

When registering a component, a tool also specifies whether or not the component should run in its own thread. If a component should run in its own thread then a thread is automatically created by the core and added to an appropriate thread group. Once initialisation of a component is complete, it enters the paused state and awaits the appropriate signal to begin execution.

(ii) Tool start-up and shutdown

At the request of the tool's initialisation routine, the core starts-up the tool by instructing all components to enter their main event loop and begin to respond to events. The tool does this by calling the core's `startApplication()` method and does not need to worry about threads or thread invocation. Once a tool's initialisation routine calls `startApplication()`, it exits and execution control is transferred to the core. The tool is terminated by the core upon receipt of a "terminate" system control event or a "system fatal" error event. Termination is achieved by gracefully stopping all threads after calling appropriate thread termination preparation methods inherited from UWComponent (or more specific sub-classes). In the case of abnormal termination, caused by a fatal error for example, appropriate debugging information is also automatically written by the core to the global log.

(iii) Inter-component interaction and communication

The core enables inter-component interaction and communication via a number of methods that act as interfaces to the components attached to the core. The core performs the required interaction or communication on behalf of the requesting component either by broadcasting events to the appropriate components (if they have subscribed to the appropriate event type) or through direct method calls. In general, when the interaction is implemented as events, the requisite processing is performed by each component's thread; when the interaction is implemented as direct method calls, the requisite processing is performed by the thread of the component requesting the interaction. Therefore, interaction implemented as events is asynchronous (except for components not executing in their own threads) whereas interaction implemented as direct method calls is synchronous.

(iv) Tool and component preferences repository

The core provides a number of methods that enable components to set and retrieve preferences and other settings either associated with the entire tool or specific to each component. Doing this in the core instead of in each individual component serves two purposes: firstly, it avoids the need for preferences and settings to be stored independently by each component, thus helping meet design goals 1 and 6. Secondly, it also allows different components (including experimental versions of similar components) to access preferences and settings created by other components thus helping meet design goals 1, 2, and 6.

(v) Standard control flow for commonly used operations

Many web operations require a number of steps to be performed by components in an identical sequence. For example performing a new search (a request typically initiated by a view component) requires the following steps to be performed by the specified components:

a) All collectors must be instructed to gracefully stop any active data collection, submit all pending data to a datastore, and reset themselves;
b) All processors must be instructed to complete processing of the current data unit, submit all pending data to a datastore and reset themselves;
c) Once (a) and (b) have completed successfully, the datastore must be instructed to accept data related to a new query;
d) Once the datastore is ready, all components can be instructed to continue with their primary activity: all search result collectors will immediately proceed to obtain the data related to the new search and start performing the search.

Since interaction and communication with components is done either via events or via standardised methods that exist in each component of the same type (derived from the UWComponent class or more specific sub-classes), it is possible to define methods in the core that perform these sequential tasks in the correct pre-determined order irrespective of how many or what types of components are attached to the core. Given that developers can create a customised core by deriving a new core class from the one in the framework, it is easy to incorporate many such standard control flows that would be available to all components in the tool without having to implement them in each component. This facilitates design goal 2 without negatively affecting design goal 1.

(vi) Deployment control

It was envisaged that the framework would be particularly useful for developing experimental and prototypical tools, so it was important to ensure that no superseded versions of tools were in use by evaluators or beta-test users. In order to avoid this, the core incorporates a method of "expiring" tools beyond a certain date. Instantiation of the core object requires an "expiry date".

The **UWDataStore** class is the abstract parent from which all in-memory datastore components must be derived. It extends UWComponent by including datastore-specific data elements and methods to enable the components of a tool to interact with the uWIRES information model. The UWDataStore class specifies the prototypes (abstract methods) for all the functions that a datastore component must implement. However, in order to avoid restricting flexibility or penalising performance, it does not dictate implementation details such as data model and data structures but leaves these to the judgment of derived datastore designers as they are in a better position to decide what best suits a given tool.

The **UWPersistentDataStore** class is the abstract parent class from which any persistent storage datastore components must be derived. It specifies the prototypes for all the methods that must be implemented in order to create a persistent datastore and provides facilities that simplify such implementations. Some of the facilities provided are handling of database connections, transactions, automatic preparation and caching of database statements, mapping between DB data types and Java data types, assignment of parameters to database statements, validity checking of parameters assigned to database statements, releasing of database resources, automatic logging of warnings and exceptions and so on. Similarly to the UWDataStore class, this class does not dictate the type of data model, DBMS or the physical schema used, which are all left to tool developers.

The **UWProcessor** class is the abstract parent class from which all processor components must be derived. It extends UWComponent by including processor-specific attributes and methods and defining derived components as processors. Processors become aware of new data available that they may be able to process via events (assuming they have registered interest for the appropriate event types). A multi-threaded processing pipeline of multiple concurrent processors can be created by using the framework-provided **UWProcessingStage** class. Similarly, a tool can keep track of overall processing progress (e.g. for indication of progress to users) by using the **UWProgressData** class.

The **UWView** class is the abstract parent from which all view components must be derived. It extends UWComponent by including view-specific attributes and methods and by defining derived components as views. Views are the only component in addition to the core that can control execution flow within a tool. The core is able to control execution flow so that it can perform initialisation, termination and to perform standard execution control flows as described earlier. Views are able to control

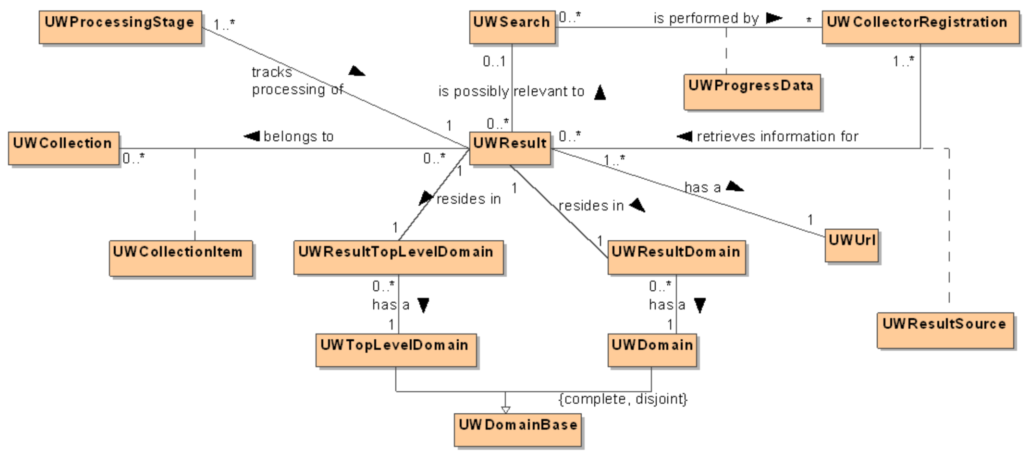

Figure 4: uWIRES default data entities

execution flow so that they can perform user requests or dictate other required processing. There is no need for the other components to be able to control execution flow and allowing them to do so could jeopardise fulfilment of the design goals. We recall that the primarily event-based approach for inter-component communication enables multiple views to be attached to a single core all of which could display different visualisations of the data and all updated simultaneously to reflect data changes.

The framework makes no assumptions as to the nature of views and it is up to the designer of a tool to determine their precise nature. The architectural layout and the design of the framework allow views to take many different forms including: visible graphical or textual views displaying data visualisations of data in the information model in any way required by a tool and appropriate controls to allow users to interact with the visualisations and tool components; control execution non-visible modules that guide a tool through a series of steps that perform some analytical function (e.g. determining the stability of search results over time for different search systems); wrappers to graphical, textual or purely control execution modules written in a programming language other than Java or executing on a different physical machine; or interfaces to pre-existing visualisation systems.

3.3 Inter-component communication

This group of classes is integral to the framework's inter-thread communication and event-based paradigm: the UWEventQueue class, as already described, implements a FIFO event queue; the **UWEvent** class encapsulates attributes that model all events that can be exchanged within the framework and methods to create and access event data; the **UWComponentCharacteristics** and **UWDataStoreFacilities** are used to communicate certain characteristics (such as supported facilities) of components to other components; the **UWException** class forms part of the error handling mechanism of the framework and is also used to communicate errors to the entire framework. Finally, the **UWInterruptible** abstract class (in fact a Java interface) and the **UWInterruptedException** class provide a mechanism for the instant interruption of interruptible uWIRES components such as collectors and processors. The **UWInterruptible** class is instrumental in ensuring a highly-responsive user interface as it can be implemented on any object that belongs to one of the components that can control execution flow and the methods provided by it are used by interruptible components to determine whether they should instantly interrupt their processing.

3.4 Data management and default entities

UWEntity is the abstract parent class of all classes that model data entities. It extends UWObject by defining an enumeration of entity types (which is used throughout the framework to identify the type of data entity a class defines) and abstract methods that enforce behaviours that all data entities must implement.

Figure 4 shows the primary default entities provided by the uWIRES information model. Entity attributes and entities that have multiple composition relationships with almost all of the entities, such as **UWCompoundKey** and **UWKeyCharacteristic**, are not shown to avoid clutter (the former models and simplifies working with simple and complex entity keys while the latter defines the types and other characteristics of entity keys). Tool developers can create additional entities either by using one of the default concrete entity classes as a starting point, or by extending the abstract UWEntity class.

The data management facilities provided by uWIRES (embodied within the **UWDataStore**, **UWPersistentDataStore** and **UWEntity** classes) include entity key validation and management, searching for entities by any of their keys or sub-keys, bulk loading into memory of all entities that match specified conditions, persistence of entities, transparent automatic caching of entities, and facilities for interacting with a DBMS system via JDBC.

3.5 Common web-related services

This group of classes implement a number of common services required by web-related tools. The classes **UWHTTPClient**, **UWHTTPClientResourceReaper** and **UWHTTPClientThreadControl** implement an HTTP protocol client that can access and download the contents of URLs. The **UWHTMLLexer** and **UWTagSpecification** classes implement an HTML lexical analyser and an HTML parser. The **UWMarkupMetadata** class provides useful facilities for handling HTML mark-up, such as separating text from mark-up in a block of HTML and allowing independent manipulation of both while maintaining the intended positioning of mark-up.

3.6 Infrastructural services and utility classes

This group of classes provides a number of fundamental infrastructural and architectural services common to most tools. For example: the **UWDisplayUtilities** class facilitates detection of, manipulation of and interacting with all available screen displays in a computer system; the **UWFormatter**, **UWInstrumentationFormatter** and the **UWLoggingClass** classes form part of the instrumentation and logging facilities of the framework by formatting log output to a common standard and writing it to the appropriate log file; the **UWSwingEventControl** and **UWSwingMenuHelper** classes facilitate (over and above those provided by the standard Java libraries) the creation of comprehensive menu structures; the **UWTextWrapper** and **UWStringUtilities** provide a number of useful string manipulation features not available within the standard Java library such as wrapping text within a particular width (taking into account any formatting that will be applied to the font) so that it can be correctly and appealingly displayed on the screen; and the **UWMD5Digest** class provides convenience methods for calculating an MD5 digest for strings.

4. DISCUSSION

We now discuss to what extent the uWIRES framework meets the design goals set out in Section 2.1.

Design Goal 1: minimise development and maintenance time

uWIRES was heavily used by the authors between February 2005 and September 2006 to develop a number of tools, including: ***WebIR2***, a user-centred meta-search tool that prototypes novel approaches to visualising web search results; several analytical tools that determine a number of web-related metrics and investigate certain characteristics of search results (for example a tool which investigates the incidence of broken links within search results); and a number of tools that helped us optimise WebIR2 and some of our visualisation approaches (for example a tool which determined the number of collector and processor threads that best balanced performance with memory utilisation).

Our observations during this usage of uWIRES were that we were able to save considerable development effort and time by:

- being able to focus on the specific algorithms and details of the functionality required to meet objectives without having to worry about developing any of the mandatory underlying infrastructure;
- introducing new facilities and functionality to the WebIR2 tool, as guided by feedback from the tool's evaluators and the data collected through instrumentation, simply by adding these to the appropriate framework classes;
- easily identifying the root causes of defects; this was very much simplified by the ability to restrict logging to specific keywords or classes before undertaking analysis. If the defects resided within the framework itself then fixes needed to be applied only to the appropriate location in the framework without the need to, for example, replicate them within the code for each component;
- using the extensive Java library and many free and open-source code libraries and classes to provide required functionality without having to develop it ourselves; some of the "off-the-shelf" libraries used include: an HTTP client [9], an HTML lexical analyser [10], an NTP client [11], a class library that implements a comprehensive set of string similarity algorithms [12], a relational database system [13], and a set of classes that facilitated interaction with web browsers and e-mail clients on Microsoft Windows [14] [15]. The framework approach of uWIRES allowed us to make all these services available to all components irrespective of the language they were developed in or the physical system on which they were executing.

Design Goal 2: facilitate experimentation

We were able to experiment with different visualisation approaches by easily and quickly developing new view components within a few hours, as we only needed to focus on code to display the visualisations. We could also investigate and interact with the various visualisation techniques side-by-side simply by registering multiple views with the uWIRES core.

Design Goal 3: deliver high performance

We were able to substantially optimise the performance of the WebIR2 tool by introducing additional parallelism by registering multiple identical collectors and processors with the core – the combination of the architectural layout, event-based paradigm and the thread-safe data management facilities meant that all that was necessary to speed up certain tasks was to instantiate more "workers" to perform the tasks in parallel. By using a non-visible view that simulated a user issuing commands to the tool, and by taking advantage of the instrumentation facilities within the framework, we were able to easily determine the number of collector and processor components (as well as HTTP client threads) that provided the optimum balance between performance and memory utilisation. By analysing the logs sent to us by the WebIR2 evaluation participants (which included data on the elapsed time for every invocation of operations that were likely to be performance bottlenecks), we were able to introduce further performance optimisations during the evaluation period.

Analysing the data collected from the evaluation exercise indicates that on average WebIR2 was able to retrieve, process and insert into a persistent database 157 results from three search engines in less than 6 seconds (including all network-related delays). The acceptability of the WebIR2 tool to real end-users was validated by our evaluation group: in response to a post-evaluation question regarding the performance of the tool, 88% of the evaluation participants stated that they agreed or agreed strongly with the statement "WebIR2 is quick".

Design Goal 4: facilitate incremental visualisation of data

The ability to execute multiple tasks concurrently, the event-based approach to communicating data availability and status to all WebIR2 components, and the concurrency synchronisation at the data level, allowed us to display newly acquired search results to the users as soon as these were available. By analysing the data collected from the evaluation exercise, we were able to determine that in practice this meant that users were able to start exploring search results within, on average, 3 seconds after initiating a search (by which time the first batch of results, typically 64, was available and displayed in the views). Meanwhile WebIR2 continued to process the results in the background.

Design Goal 5: be scalable

WebIR2 was evaluated by 25 users over periods ranging from two weeks and three months. They performed a total of 1,189 web search sessions and, as can be seen in Table 1, many of them used the tool for a significant number of their web searches. This indicates that WebIR2, and by implication the uWIRES framework that was used to build it, are able to cope with real-world daily usage scenarios.

Design Goal 6: support both interactive and non-interactive tools

76% of the WebIR2 evaluation participants stated that they agreed or agreed strongly with the statement "I would like to continue using WebIR2 for searching the web after the evaluation study is completed". This suggests that uWIRES is well-suited to the development of interactive tools. We showed that it is also suitable for non-interactive tools by using it in a number of our automated experiments including, as discussed earlier, determination of the number of collector and processor component "worker" threads required to achieve an optimum balance of performance and memory utilisation, and an

Table 1: Extent of utilisation of WebIR2 by evaluation users

Participant	Search sessions per week (from pre-eval questionnaire)	Tool Usage period (weeks)	Expected sessions given pre-eval response	Sessions performed using Tool	% of expected sessions
1	14	13.0	182	208	114%
4	21	2.0	42	48	114%
7	3	11.7	35	57	162%
11	21	9.4	198	48	24%
13	14	12.1	170	121	71%
14	14	12.3	172	78	45%
18	14	10.6	148	49	33%
27	14	12.1	170	33	19%

experiment to determine the incidence of broken links in search engines' results.

Design Goal 7: be portable
Since uWIRES is entirely written in Java it is portable to any platform for which a Java Runtime Environment exists. We designed and tested the WebIR2 tool on Windows XP and all the evaluation participants used it on either Windows XP or Windows 2000. We showed its portability to other platforms by porting the Windows-specific portions (all of which were isolated in a single package) to Linux and executing this on Suse and Ubuntu Linux flavours.

5. CONCLUSIONS

In this paper we have described uWIRES, a software framework that aims to facilitate the rapid development of high-performance, portable, production-quality web-related tools. We discussed how the framework meets its design goals and its use in the development of a number of research tools. One of these tools, WebIR2, was a substantial user-centred meta-search application that prototyped a novel approach to visualising search results and which was evaluated in a real-world context by 25 users over a period of four months.

uWIRES is related to other frameworks and code libraries (such EJB, COM, FLAIR, FIRE, InfoGrid and Terrier) in that it provides an architectural layout, enforces certain design and coding disciplines, and furnishes developers with a number of classes that meet some functional and non-functional requirements. Unlike these other frameworks and code libraries however, uWIRES does not just provide a general architectural layout and the services needed to support that layout (as is the case for EJB and COM). Nor does it just provide classes that meet certain very specific functional or non-functional requirements, such as user interfaces or indexing and searching the web (as is the case for FLAIR, FIRE, InfoGrid and Terrier). Instead, uWIRES is a fully-fledged framework, applicable to a wide range of web-related tools and it provides, in an "off the shelf" manner, most of the non-functional and many of the functional requirements that such tools need.

The WebIR2 tool developed using uWIRES was of sufficient quality that, after a trial period of a few weeks, an investor consortium offered funding for its commercialisation, and this activity is ongoing. Future work on uWIRES itself includes (i) to further optimise the performance of uWIRES, e.g. reduce object creation and destruction overheads by incorporating more extensive caching and pooling of very frequently used objects; and (ii) to increase its applicability to the development of tools for evaluating new web-related algorithms, interaction techniques or paradigms e.g. by enabling the near real-time submission of evaluation data and metrics to a central repository when a connection to the repository over an appropriate network is detected, and a feature to "auto-update" evaluation tools when new versions become available.

6. REFERENCES

[1] Computer Industry Almanac Inc.. Worldwide Internet Users Top 1 Billion in 2005 (Press Release). Available at http://www.c-i-a.com/pr0106.htm, Jan 2006.

[2] Peter C. S. Wong, Eric R. Reid. FLAIR - User Interface Dialog Design tool. In proceedings of the 9th Annual conference on Computer Graphics and Interactive Techniques, 1982.

[3] Gabriele Sonnenberger, Hans-Peter Frei. Design of a Reusable IR Framework. In proceedings of the 18th International ACM SIGIR conference on research and development in information retrieval, 1995.

[4] Ramana Rao, Stuart K. Card, Herbert D. Jellinek, Jock D. Mackinlay, George G. Robertson. The Information Grid: A Framework for Information Retrieval and Retrieval-Centered Applications. In proceedings of the 5th Annual ACM symposium on User interface software and technology, Dec 1992.

[5] Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, Christina Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In proceedings of the 2006 Workshop on Open Source Information Retrieval Systems.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture - A System of Patterns. John Wiley & Sons, 1996.

[7] David Blevins. Overview of the Enterprise JavaBeans Component Model. In Component-based software engineering: putting the pieces together, 2001.

[8] Tim Ewald. Overview of COM+. In Component-based software engineering: putting the pieces together, 2001.

[9] Apache Jakarta commons HTTP client library. Available at http://jakarta.apache.org/commons/httpclient/.

[10] Open source HTML Lexer and Parser library. Available at http://htmlparser.sourceforge.net/.

[11] Apache Jakarta commons net library. Available at http://jakarta.apache.org/commons/net/.

[12] Sam Chapman. simMetrics Similarity Metrics Library. Available at http://sourceforge.net/projects/simmetrics/.

[13] Apache Derby, Open-Source relational Database Implemented Entirely in Java. At http://db.apache.org/derby/.

[14] JACOB Java to COM Bridge. Available at http://sourceforge.net/projects/jacob-project/.

[15] JDesktop Integration Components (JDIC). Available at https://jdic.dev.java.net/.