PERSONAL MULTIMEDIA CONTENT MANAGEMENT

A Project Report

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2002

0-612-72931-1

ABSTRACT

Personal Multimedia Content Management

Hon-Wai Chia

The explosive growth of information and the rising complexity of managing data increase the difficulties of determining how and where to store information and how to find and access it when it is needed. This project seeks to explore an alternative method for computer users to store and access personal information, by implementing a content management user interface with the use of filters and a time-based filing system based on concepts introduced in Semantic File Systems and the LifeStreams project.

ACKNOWLEDGMENTS

I would like to thank Dr H.F. Li for his patience and guidance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 INTRODUCTION

In this chapter, we briefly introduce the problem domain and give an overview of the contents of this report.

Motivation

The proliferation of computer and web usage, and the explosion of data storage, due to advances in technology and the continual reduction in cost of storage per megabyte, has created the awareness of the need for evaluation of alternative methods of managing this data. When discussing how user interfaces might evolve, there are two distinct issues to be addressed. Firstly, the issue of the look-and-feel of a computing environment, and secondly, the more critical issue of determining how a system stores user content.

By challenging some of the implicit assumptions and characteristics of current systems, and exploring new approaches to file systems and their associated user interfaces, it may be possible evolve these systems into more meaningful and intuitive models based on temporal or relational concepts, so that users are no longer forced to refer to their data content using unique file names.

This report proposes a new user-interface model and discusses how file systems could be changed to support such a model.

Overview of This Report

This report consists of two parts. In the first part, the problem is introduced. We describe two subject areas: File Systems and Multimedia Content. At the end of these chapters, the requirements presented by these subject areas are evaluated.

In the second part, our solution, called Atur (meaning "order" or "arrange"), is presented. In these three chapters, we discuss the main ideas behind the representation and the user interface, namely Semantic File Systems, and the LifeStreams project, and describe the Atur implementation itself.

In the final part, we discuss the current shortcomings of the existing implementation, and possible areas for future work.

Chapter 2 THE FILE SYSTEM

In this chapter, we discuss the limitations of current common File Systems from the end-user's point of view, and the requirements for possible improvements.

The User's Perception of the File System

Before a user can begin using a computer, users have to learn concepts behind common file systems. For example, they have to know how their information is stored, they have to understand what a "file" is, and they have to be taught to how to navigate the hierarchy of directories and subdirectories where these files are kept. These pre-requites immediately forces a user to understand how their data is stored inside the internals of the computer, and imposes a paradigm that may not mirror the way that a user conceptualizes his data should be organized. For users, the most basic unit they must deal with in any interaction with a computer is the file. However, this concept of a continuous file is a conceit; it is not a primitive of the file system at all but is actually an abstraction: the operating system already masks the fact that information is usually stored on non-contiguous sectors of the hard disk. Let us briefly look at the basic concepts that the user must be aware of:

- Information is partitioned into discrete units, usually a file. Every file is treated as a separate, distinct object.

- These units of information are kept in of a hierarchy or directories and subdirectories.

- Each information object is given a single, semi-unique name. This name identifies the units of information, and is used to access the information.

- These units of information are accessed and modified individually. These units of information cannot be jumbled arbitrarily with other units and manipulated together.

These concepts give us an idea of some of the fundamental limitations of traditional file systems. The next section will discuss the broader issues leading from these concepts.

Issues

Partitioning data

How should we partition data? If we put data into discrete objects, then what should be an object? An object is not necessarily a file; now it can also morph depending on context.

In older systems, when many user interfaces were implementing so-called WYSIWYG (What You See Is What You Get) systems, software was attempting to present a single representation of information objects. The idea was to closely couple the relationship between the content and the view. In this philosophy, output and the content were one and the same. With the explosion of hypertext and the World Wide Web, more current user interfaces now differentiate between the content and the view. The same content will often have many different associated views, depending on factors such as the context of the task or the medium of presentation. For example, users might want to retrieve e-mail on many different interfaces:

- on PDAs, where the display is small and limited
- on the telephone, through voice synthesis
- at their workstations, where the presentation limitations are the lowest and the user interface is rich and complex

In all of these different interfaces, the view is adjusted based on the needs of the user and the limitations and capabilities of the medium.

Another problem is that an object is usually a composite made up of other data. For example, a single Web page comprises of a HTML file and one or more subsidiary resources, such as Java applets, images, JavaScript, etc. These resources are rendered together when the page is ready to be displayed by the browser. Sometimes, it is difficult to define the object, even when taken to the smallest granularity. In the previous example, the HTML file could have been generated using an Extensible Stylesheet Language (XSL) definition with data expressed in Extensible Markup Language (XML). Such methods are used to serve up on-the-fly generation of content. The HTML file generated on demand by the server is based on parameters passed between the browser and the server. For example, if one downloads a Web page using a broadband connection, the HTML file may be formatted to full screen size, with full 1024 by 768 resolution, and using 32-bit colour images. If a slower connection, such as a PDA browsing the web over a wireless connection. is used. the HTML file may be generated with Compact HTML (HTML for small information devices) with abridged content. and using small. low-resolution images.

Thus. it is difficult to assign content to neat little boxes: it can change according to the properties and the context of the presentation environment.

Proliferation of Files

There are other problems. which continue to frustrate the user. A familiar problem is the task of installing a new application. In the days of single-file applications. a simple. file-based user interface will only require that you drag the application icon from the source. such as an installation floppy. to the target space. such as personal file folder. With the increasing complexity of software applications. it is now nearly impossible to find a modern application being limited to a single file: typically. the installation will copy into the file system many subsidiary files such as user document files. program configuration files. un-installer files. etc. all stored in installer-defined directories which are inconsistent from one application to the next.

Not only is the number of files a problem. this also means that a unit of information is associated with many files.

Single Parent Relationship

File systems are organized with strict hierarchies of directories and subdirectories. In the real world, however, the same information unit often belongs to multiple categories. A user's address information may appear on many emails or documents, and thus would belong to several objects across the system. A typical Windows user, using common word processors or desktop publishing software, will create documents using many pieces of information. His documents may be filled with many different kinds of information and media such as plain text, digital photographs, web pages, email addresses etc. A specific piece of information might thus logically be categorized as being part of many different documents in a user's system. Intuitively, if a piece of information is updated, all documents which contain that piece of information should also be automatically updated. However, with a single parent system, this kind of functionality would not be supported by the operating system.

Large hierarchies are problematical to conceptualize: it is difficult for the user to keep track of many directory names in order to traverse a large directory tree. This is analogous to forcing someone to remember a long string of numbers. With large hard disks, and large amounts of files, the single parent concept will fail because although a piece of data may have many possible classifications, the user had to choose use one in order to store the data in the hierarchy. In addition to the difficulty of maintaining consistency, classifying a file is a highly subjective and personal process, and that classification may not be easily recalled at a later date. This problem is further compounded if a different user attempted to discern where the files are. This is analogous to someone wandering into an unfamiliar office and trying to find a file from a large filing system.

File Names

In modern desktop operating systems, files are represented in the user interface by a name and some additional meta-data information. These names are usually chosen by the user who created them. These names are also used as the identifier of the file in the user interface, as they are usually unique (the operating system usually enforces the uniqueness when the file is being named). It is by these names, that a user will find and access these files later. These file names are not ideal for identification purposes for many reasons. Firstly, this is an inconsistent and arbitrary process. It is impossible for users to consistently generate meaningful file names, even in systems that are designed to allow long ones [1]. This is a subjective process, and is highly dependent on the whim of the user. Users in a hurry will not be likely to spend the extra time to concoct a more meaningful name. Different expressions and terms also vary from user to user, so these names will not be consistent in a system where files and information is being shared. Secondly, the naming may be premature. Since file names must usually be specified at creation, before any content is created, if the file name is created on assumptions which later prove to be incorrect, the name will no longer be suitable to help identify its contents. Third, users usually have difficulty later recognizing and remembering what a name represents. This is especially true when other factors are involved, such as the passage of time. the existence of many other similarly-named files. and the complexity and anonymity of a large file system. Last. but not least. is the unsuitability of names as identifiers of information units. Users are rarely able to define exactly what they are looking for. It is only through browsing and examining of the contents of different files. can the user find what he wants.

Summary

To summarize. we can derive the following requirements for an improved file system:

- An object-storage mechanism is needed where objects do not belong to single parent relationships.
- A compound-document architecture is needed.
- Context information must be provided to improve the presentation environment.
- An automatic mechanism is needed to augment or replace file naming so that it is systematic and un-arbitrary.

Chapter 3 MULTIMEDIA CONTENT

In this chapter, we discuss the various types of user content and their ramifications on the user interface and file system.

Multimedia Data

Non-textual information, such as aural and visual data, which represents multimedia content, such as speech, music, images and videos currently can not easily be extracted into structures useful for querying. Additional meta-data information is therefore required. This information may be manually provided through human intervention, by using a given set of meaningful keywords to classify the multimedia data, or through automatic processes, by using image processing software to automatically extract information such as colours and patterns [2].

Media Representation

There are some general data representation issues, which are relevant for the representation of the different media. The representation of alphanumerical data is straightforward, and formatting problems have already been mostly settled for this kind of representation. Moreover, the operating systems mostly guide the way by providing some standard set of data types (MIME). This is, at least for today and in the near future, not the case for multimedia data.

The basic data types like the alphanumeric ones are not appropriate to reflect the structure of multimedia data. New built-in data types like bitmap or audio sample have to be provided.

Furthermore, type constructors taking into account the temporal nature of multimedia data will be needed in some form. Additionally, appropriate support for processing these data types has to be provided. Similar to the standard operations associated with alphanumeric data (e.g., add integers, concatenate strings) access operations such as interactive video and audio editing and media playback and synchronization are needed.

Summary

To summarize, we can derive the following requirements for multimedia content:

- New built-in data types and operations for multimedia data are needed.
- Modular and efficient representation of different formats should be supported.
- Data representation should be transparent to the application/user.
- Different views on the same data should be possible.

Chapter 4 SEMANTIC FILE SYSTEMS

The concepts and design philosophies introduced by Semantic File Systems is discussed here, as a starting point for the approach undertaken by our implementation.

Semantic File Systems [3] is the concept of using higher-level information of files to create a unified data management framework.

Important higher-level information relating to a file include:

definitional :file extensions and magic numbers which define file type

associative :keywords in file that characterize content

structural : physical and logical arrangement of the data, including relationships between and within files

behavioural : viewing and modification semantics, change management

environmental : creator, revision history

Relying on the user interfaces of applications (for example, a word processor) to browse the file system will limit us to the presentation capabilities supported by the application. We can instead use views which are application independent, so that file system itself will provide a finer-grained, consistent and seamless interface to file data. This will also enable the ability to integrate file information with information collected in more useful and structured forms, such as databases.

Design Approaches

There are basically 2 possible design approaches to a Semantic File System: integrated and augmented [4].

Integrated Approach

Integrated file systems, such as Symphony [5], augment the file system by directly making improvements into the file system itself. These systems provide a unified data model, by merging the file data and meta-data into one; both types of data are tightly coupled and kept in synch. These systems may provide a means to define object structure, and even behaviour, by using a type definition language such as an Interface Definition Language (IDL). The data model is defined using this IDL so that content and meta-data, such as file type, file structure, author, access histories, content, etc can be stored together. Additional information such as the definitions access routines for different file type can also defined by the IDL. These routines can then be made available to all applications in the file system.

Augmented Approach

Instead of working in the file system layer, augmented semantic file systems adopt a layered approach. The improvements are introduced in a separate layer, on top of the traditional file system. This approach adopts an evolutionary philosophy to Semantic File System architectures. By leaving the underlying traditional file system interfaces untouched. the augmented files system provides an additional content abstraction view of file contents. Tools layered on the content abstraction can be more intelligent about querying and manipulating file information. The obvious drawback is the lack of a unified data model. since both systems are physically separate in different layers. The file data model is unaware of the meta-data model. Additional work also has to be done to make applications aware of the augmented layer.

The goal of an integrated approach is to attempt to provide an evolutionary way to get to a higher form of Semantic File Systems. without requiring radical changes to underlying operating systems interfaces. Augmented approaches do not impose on the underlying operating system. and provide less ideal version of the functionality of an integrated Semantic File System. However. augmented Semantic File Systems can leverage progress in operating systems to progressively evolve to gain the capabilities of a full-fledged integrated Semantic File System.

Chapter 5 LIFESTREAMS

This chapter provides a brief overview of the concepts and metaphors introduced by the LifeStreams project, which will be adapted for our implementation.

The LifeStreams project attempts to overcome difficulties encountered by users of common desktop systems, namely:

1. The struggle in organizing and finding information within hierarchical file systems.
2. The complexity of making use of archived information, which users normally discard so that they are not overwhelmed by it.
3. The confusion caused by the lack of a "big picture" view.
4. The difficulty of managing schedules and reminders.

LifeStreams provides a metaphor for organizing the electronic documents that users collect, whether in the form of electronic mail, downloaded images, pages gathered from the Web, or scheduling reminders, in "a fluid and natural way that reflects the way users work" [6]. This concept is based on earlier research studies such as the "Pile" project, which found that knowledge workers use physical space as temporary holders for information pieces which they cannot yet categorize. They often prefer to deal with information by creating physical piles of paper, rather than immediately categorizing it into specific folders [7]. Even when the user finds it difficult to categorize items, or to express how or why they are interconnected. they are still often able to meaningfully arrange them in space. Thus. computer interfaces should not force users to classify information in order to store it for later use.

The LifeStreams Interface

The LifeStreams interface presents the user with a stream of documents. LifeStreams presents the user with a view of the stream from the present to the past. The user can manipulate the mouse pointer or a scroll bar to backtrack into the past. or to scan the document representations of each document.

Users interact with LifeStreams via five primary operations: new. clone. transfer. find and summary. New and clone create documents. New creates an empty document and adds it to your stream. Clone duplicates an existing document. Transfer copies a document from your stream to someone else's. Find prompts the user for a search query and creates a Substream.[6]

LifeStreams introduces the concept of Substreams. Substreams. like the virtual directories demonstrated in the Prospero project [8]. present the user with a "view" of a document collection. for example. all the documents that are relevant to a search query. Rather than placing documents into fixed. "file folders" directory structures. Substreams create virtual organizations of documents from the stream. Documents are not actually stored in the Substream: the Substream is a temporary collection of documents that already exist in the stream [9].

Substreams can also be created and destroyed dynamically without affecting the contents provided by the stream or any existing Substream. If a Substream is allowed to persist, it will collect new documents that match search criteria as they are added to the stream. For example, a Substream created with the query "find all documents created by users other than me" would subsume your mailbox and automatically collect mail as it arrives. A Substream created from "all electronic mail I haven't responded to" would act as a collection that only contains unanswered mail. Two Substreams may overlap. Substreams can be created incrementally, yielding a nested set of collections. Semantically, this incremental stacking of Substreams results in a Boolean "and" of each new query with the previous Substream's query.

"Substreaming" thus provides a consistent information management framework and a way of finding information. We will see how some of these ideas will be applied in Atur in the next chapter.

Chapter 6 A PERSONAL MULTIMEDIA CONTENT MANAGER

In this chapter, we describe the concepts and requirements behind Atur and its principal features, and conclude with a usage scenario.

The purpose of this project is to address the issues brought about by the demands of personal multimedia data, in order to explore ways to improve the effectiveness and usability of file system user interfaces. To recapitulate on the requirements which were arrived at from previous discussions, the following requirements must be met by the system:

- An object-storage mechanism is needed where objects do not belong to single parent relationships.
- A compound-document architecture is needed.
- Context information must be provided to improve the presentation environment.
- An automatic mechanism is needed to augment or replace file naming so that it is systematic and un-arbitrary. New built-in data types and operations for multimedia data are needed.
- Modular and efficient representation of different formats should be supported.
- Data representation should be transparent to the application/user.
- Different views on the same data should be possible.

Principal Features

In the following we detail the features of the proposed solution.

Augmented Semantic File System:

This solution will attempt to implement a Semantic File System with the use of higher-level file information to add content management functionality. The solution will work on top of existing File Systems, and will therefore utilize information and functionality provided by the operating system. The traditional file system interfaces will remain unchanged, while providing a parallel content abstraction view of file contents.

This system will provide a content abstraction layer on top of the traditional file system so that higher-level tools can be constructed based on our content abstraction, which can be more intelligent about querying and manipulating file information.

Highly advanced implementations of augmented Semantic File Systems use the content abstraction layer for either query shipping or index shipping. Query shipping relays a repository-independent user query to the relevant files. Index shipping pulls the contents out of files, indexes them, and maintains them as meta-data for a user level querying engine. Our implementation will limit its scope to index shipping.

Files are abstracted into logical collections, or "Substreams". A Substream manager manages each Substream. Our definition of Substream for this project bears some resemblance to traditional database concepts such as view specification mechanisms.

However, it is not as precise as the database view, as traditional concepts in well-defined databases will not be present in our semi-structured problem space.

Simply put, Substream managers provide object-oriented views of the file contents in their domains. They also provide the engines to support querying on their domains. Substream managers contain both the content summarization engine, and the query engine.

The content summarization engine should consist of a file classifier and a collection of file content summarizers, based on file-type. The file classifier infers the type information of a file based on implicit information such its file extensions or explicit data such as magic numbers. The summarization engine processes all files to extract information about the file, based on the file's type. This information includes attributes such as author, access histories, content, relationships to other files, etc.

Due to the augmented nature of our layering approach, there will be implicit limitations for our system. Our augmented file systems will have to depend on operating systems which are not aware of the existence of the higher layer. This means that important events such as changes to files will not be passed on to the augmented file system layer to handle. There will therefore be a need to create a custom polling mechanism to get this information. This solution will most likely be less efficient than a simple and elegant file notification scheme. Moreover, the augmented Semantic File Systems layer will also have to do additional work to determine what incremental changes were made to the file content.

As for transactional capabilities. we will not implement this feature in our implementation. Although the system can easily provide transactional capabilities on the meta-data (by imposing it in the storage transaction in the database). the system normally cannot maintain an ACID transaction (i.e. transactions that support atomicity. consistency. isolation. and durability) across the file data and meta-data. since it is never in the context of the file update transactions. A caveat is that in more advanced operating systems. which feature transactional file systems and event notifications of file update events. this could still be implemented. However. this will be ignored as we will confine our implementation to a common desktop operating systems only.

The file type semantics for our implementation will be limited to a simple type system. Our approach will rely on the traditional weak implementation of file types. The traditional implementation of types in UNIX. for example has been to add an extension to the file name. and let programs "type check" their input based only on that extension. For example. the TEX processor would demand that files passed to it follow the naming convention "filename.tex": the Java compiler will demand "filename.java" and so forth.

Time is the Context - Chronology-based storage of data (Temporal Storage)

Time based ordering will be used. Time is a natural component of experience and is the most natural method of chaining things together in the mind. as suggested by time-based organization studies such as the 'Pile' project. File information will be maintained in a journaling system which will be an implementation based on "LifeStreams". Like

LifeStreams we will use a simple organizational metaphor and maintain all documents in a time-ordered stream. The user will not be required to interact with files and directories. Through the content summarization and query engines, the system will provide the content abstraction layer, so that the user will only interact through this paradigm.

A LifeStream is a time-ordered stream of documents that is conceptually equivalent to personal journal of a user's electronic life: every document that is created is stored in his LifeStream, as are the documents other people sent him. The tail of one's stream contains documents from the past, perhaps starting with the user's electronic birth certificate. This journal is purely time-ordered. The oldest documents would be placed at tail of the queue, with newest documents at the queue's head. So any current open projects and works-in-progress, email correspondence, currently-used software would be placed in the front of this stream.

We can see 2 major advantages of this simple yet powerful concept:

Firstly, the usefulness of this concept is apparent when we try to use time-based information as the context for a document. Instead of following links or guessing keywords, the user can simply scroll back in time, relying on our memory for hints to trigger our recollection. An important benefit to recognize here is that because of the temporal ordering, when we find the document, the context is also presented to us. In studies conducted at the Records Deposition Division at the National Archives of Canada, Terry Cook states that the key to effective record-keeping "lies in being able to determine, sometimes long after the fact, not only the content but also the context of a record in question." [10] Cook discovered many cases in which the lack of contextual information made electronic records largely unusable. Thus, the use of timestamp information, when presented in chronological order with other documents in a user's stream, can present valuable contextual information to the user.

Secondly, unlike other schemes, which still require users to classify their data, and which thereby generates poor quality classifications, due to the subjective and highly arbitrary decision-making that human categorization involves, chronological ordering is simple, consistent and indisputable. There is no ambiguity about classification decisions since there is no multiple possibilities for classification such as whether a document belongs in the "Research Papers" folder or the "Work Projects" folder.

Query/Filter Integration

The implementation will allow querying/filtering of file data and metadata with a simple unified query syntax (as defined in an XML file). This feature will provide the "intelligence" for the following 2 features:

1. The intelligence mechanism for fuzzy matching.

   The system will provide a mechanism for the user to specify keyword patterns for filters. When the system finds several viable categories for a filter, before it has had any feedback, it will select the category (or categories) which have amassed the highest internal score based on feature comparisons, and reject scores which are below a user specified threshold. In the event of equal scores, a category is selected arbitrarily.

2. Integration of feedback and improvement for user-directed "learning" by the system.

   The system will allow the user to provide feedback to the scoring mechanism, so that scoring information can be constantly tuned to improve results.

Reference Integration

The implementation will allow objects to contain references to other objects, as in composite objects. This will allow our system to construct objects which contain other nested objects. Filters can therefore be combined in a hierarchical fashion to construct more complex filters and object definitions.

Simplicity

This solution emphasizes simplicity for speed and performance. As this will be an augmented solution and meta-data will be limited to what the operating system can provide, data content will be limited to the basic alphanumeric data types.

Personal

The solution will emphasize ease of configuration and adaptability. Scalability will not be the prime consideration at this stage. Because our system is designed for low latency, high processing speed, it would occasionally receive incorrect answers from the data store while it was being written, due to lack of transactional consistency with the operating system information, or due to the imperfect nature of scoring. These occasional incorrect answers are acceptable for storage and retrieval of personal data, but not for business-critical data.

Let us now see how these concepts can be put to walk in a usage scenario.

Usage Scenario

User creates a stream. The system creates a database for the stream. Initially, the system will scan the root user directory to load information on all existing files and save this information to the database. All subsequent activities such as creation or modifications of objects are stored in the database. All usage is stored in a manner consistent the "LifeStreams" model, i.e. the information will be time-ordered.

User interacts with the system using the Graphical User Interface to create filters based on keywords. The user can create a Substream by associating it with a filter. The system picks the elements of the user's stream which fit the Substream based on its filter. Substreams are layered on other Substreams to become a hierarchy of Substreams. Using a tree diagram, the system provides the user with a hierarchy of Substreams, which presents a user-specific view of the contents of the user's stream.

Let us now walk though a more detailed example, with some example files.

Example

User A creates a stream, which currently contains 5 files:

1. An email containing a job ad for a java programmer : *JavaJob.txt.*
2. A java source code file for the "Atur" project: *StreamManager.java.*
3. A list of contact information for people working on the "Atur" project: *Contact.txt*
4. A java source code file for POP3 email client: *PopClient.java.*
5. A web page on Auto Maintenance and Repair: *AutoMaintenance.html*

The names of these files are created by the user. The semantics of these file types are derived from the file extensions, using the traditional UNIX-style simple type system.

The user creates 2 filters:

"Java Source File" filter – a filter which contains keywords and type information of java source files.

"Atur Project" filter – a filter which simply contains the "Atur" keyword, for all files related to the "Atur" project.

The user creates and structures his Substreams as demonstrated in Figure 6-1:

```
User's Stream
   |
   |-- Java Source Substream
   |
   |
   └--Atur Project Substream
         |
         └----Java Source Substream
```

Figure 6-1: An example of a User's Stream

Note that the user has used the same filter twice, once at the top level, and once as child Substream to the Atur Project Substream

Based on the user's stream, the system presents the following view in Figure 6-2:

```
User's Stream
   |
   |-- Java Source Substream: StreamManager.java  PopClient.java
   |
   |
   └-- Atur Project Substream: StreamManager.java  Contact.txt
         |
         └----Java Source Substream: StreamManager.java
```

Figure 6-2: Example User's stream, with filtered results

The web page on Auto Maintenance and Repair, *AutoMaintenance.html*, on the other hand, is rejected as it does not belong to any specified filter.

Basic Idea

The solution will create compound objects based on a simple object specification file. An object will have an associated view, and can be composed of other objects. The most basic object would be files. Files will be managed automatically by the system. File Naming, physical location of the file, storage of file and behavioural information will all be managed automatically behind the scenes. The files will be stored and managed in a variation of the FileStreams metaphor. Files created by the user will automatically be named and placed into stream according to chronological order.

Components

Viewing – User Interface

The interface allows the user to create compound objects based on a simple object specification file. The object specification file will behave like a filter, which specifies attributes of objects, which belong to the specified object. The interface will allow the user to manipulate objects/files by calling on its associated viewer/editor. The interface will interact with the object manager.

Referring again to the illustration in Figure 6-2, the Atur Project and Java Source Substreams are objects that are defined by an object specification file. In this example, the user can manipulate 3 different types of objects:

1. Substream objects: Java Source Substream, Atur Project Substream

   Selecting this object for manipulation will call up the program associated with this object type. The system will call up the filter/object specification editor.

2. Java source file objects: StreamManager, PopClient.java

   Since this is primitive object (a file in the operating system), selecting this object for manipulation will call up the program associated with this file type, be it the java compiler or the java source code editor. The type association is externally specified by the configuration of the underlying operating system.

3. Text file objects: Contact.txt

   Since this is also primitive object, selecting this object for manipulation will call up the program associated with this file type, such as a text editor.

Object Manager

The Object Manager will manage the database of information on all objects and files such as file names, the physical location of the files, meta-data and collected behavioural information. Files created by the user will automatically be named and placed into a stream according to chronological order.

Benefits

- Information is partitioned into meaningful units, which are principally dependent on content. For example, the flexibility will also allow the user to specify an object, which composes of all files and data on a particular project.

- The object-storage mechanism does not limit objects to single parent relationships. An object can belong to multiple compound objects. This eliminates the need to have multiple copies of the same or similar data, and also makes redundant updates on data unnecessary. (Example, updating an email address for various web pages).

- The user is protected from dealing with the file system directly. Names are no longer pertinent for data retrieval. Keywords and higher-level attributes would be the keys to retrieving data, which is more natural and consistent with usage patterns.

- The added flexibility will also make it possible to change the view for an object depending on the environment or context. For example, if the user is using a PDA with a limited displav, the view can be modified to a reduced detail.

Issues

Multi-users

How does the system support multiple users? Objects created in the system are linked to the user. i.e. each user has her own database repository. This means that unlike primitive objects (files). which are accessible to all. compound objects are not visible to other users. Since objects are defined using more abstract meta information. this information can be simply imported or duplicated into the personal systems of other users.

Access Conflicts

A related issue to multi-user support is resolving access conflicts. The Object Manager will support access locks at the operating system level. There will be no different from the scenario of a file in a shared network directory. Access conflicts will be resolved by whatever controls are implemented by the operating system.

Chapter 7 DESIGN AND IMPLEMENTATION

This chapter gives an introduction to the basic concepts of Atur. describes the main components and the design features. and concludes with an overview of the user interface.

Filters

Purpose

The purpose of a filter is to support querying and filtering of file data and metadata. This will be done using simple unified query syntax (as defined in an XML file)

Implementation

The filters are represented using Extensible Markup Language (XML). XML was chosen because of its ability to adapt to and support later improvements and extensions. Another important advantage is that it is the universal format for structured documents and data on the Web. and is widely supported by a wealth of software tools and programming languages.

The filter file uses XML tags to define the meta information for the objects defined by the user.

The Filter XML Tags

The filters are defined using XML tags. The structure of the tags is analogous to a HTML document. The basic tags are as follows:

1. filter
2. title
3. body
4. file type
5. keyword

The file type property defines the type of the file. In our primitive Unix-style simple type system, this simply defines the file extension of the file or files which belong to this filter. The property also recognizes wildcards such as "*", which specifies all file types.

The keyword property is the most important tag. It defines what the tokens which are defined for an object and its value for scoring purposes.

It has the following properties:

a. token

This is the string, which is defined for the keyword, made up of a variable length string of any ASCII characters.

b. scoreweight

This defines the scoring weight for this token. This is a positive integer, which can be of any number. The scoring will be normalized, so the relevance of the amount of the number is relative to the scoring weight numbers of the other keywords for the same filter.

c. scoretype

This defines the scoring type for this token. This is a single char. There are 3 possible values:

i. '1': a positive scoring weight. This is a moderately strong keyword for matches.

ii. '2': a positive scoring weight. This is a very strong keyword for matches.

iii. '-': a negative scoring weight. This word will be used as an undesirable keyword for matches.

A Filter Example

Here is a sample basic filter file:

```
<?xml version="1.0" standalone="yes"?>
```

```
<filter>
 <head>
  <title>java files</title>
 </head>

 <body>
  <file  type="java" />
  <keyword token="extends" suffix="" scoreweight="1" scoretype="1" />
  <keyword token="implements" suffix="" scoreweight="1" scoretype="1" />
  <keyword token="package" suffix="+" scoreweight="1" scoretype="1" />
  <keyword token="import" suffix="*" scoreweight="1" scoretype="1" />
  <keyword token="return" suffix="*" scoreweight="1" scoretype="1" />
 </body>

</filter>
```

In this example, the object is simply defined as an object which contains all data primitives which are java files that contain the following keywords: "extends", "implements", "interface", "package", "import", and "return". The suffix tag is used to define:

1. The length of the suffix after the word.

For example. if we are looking for Concordia University Student Ids . which currently have a length of 7 digits (e.g. 2640449) . and we are only interested in students of a particular year (let's say that they all have a ID number which begins with 310). then we can specify the criteria like this:

```
<keyword token="310" suffix="4" scoreweight="1" scoretype="1" />
```

2. Wildcard for suffixes of zero of more length: *

For example. if we are looking for items related to French culture. we may want to look for the following terms: "French". "Frenchman". and "Frenchmen".

In this case. we can specify the criteria like this:

```
<keyword token="French" suffix="*" scoreweight="1" scoretype="1" />
```

3. Wildcard for suffixes of 1 of more length: +

This is useful if we are looking for items related to a word with different endings. A simple example is a verb which ends in "y". such as "worry". In this case. there can be many suffixes. such as "worry". "worries". "worrying". "worried". "worrier" etc. But in all these cases. there is at least one char after "worr". In this case. we can specify the criteria like this:

```
<keyword token="worr" suffix="+" scoreweight="1" scoretype="1" />
```

Scoring

Purpose

The purpose of scoring is to support querying and filtering of file data and metadata. This will be used to determine the "fit" of a piece of a data primitive according to an object's filtering criteria.

Implementation

Scoring is done using the properties of the "keyword" tag which is explained in the previous section, in "Filters". Scoring is done on all data primitives in a defined Substream (or root stream, as the case may be). In our prototype implementation, a data primitive would be a single file. Using the criteria of the "token" property as prefix for matching, and using the suffix property to calculate the required length of an encountered word for a match, we then proceed to score the matching keyword. Using the scoring weight and scoring type properties defined for a keyword, we use a simple formula to calculate a score for the matched word: scoreWeight * scoreType. The scores are tallied for the data primitive, to be compared with scores of other data primitives. The results and then ranked, and data primitives with scores below a defined threshold are rejected from the result set.

Because of the demanding number of operations of disk I/O and string comparisons, the scoring engine has to be optimized in order to ensure efficiency. This is done by limiting the data to be stored in memory, and avoiding unnecessary reads from disk.

The final score as presented to the user is normalised, so that it will be an integer between 0 and 100. This part is actually independent of the scoring engine, and is dependent on the user interface.

Data Storage and Databases

Purpose

Data storage is needed to record user activity and to store the definitions of objects defined by the user.

Implementation

Data storage is implemented using an object database (ObjectStore PSE), instead of using the more familiar relational database. Relational databases require the developer to convert and map the data to be stored into relational tables consisting of rows and columns. The columns of data, or data fields, can only contain predefined data types. The relationships between rows in different tables have to be defined at the database level, and must be directly supported by the database engine. An object database, on the other hand is a more direct translation from the characteristics of runtime objects. The relationships between defined objects are directly supported from the object design. For example, the database is aware of the relationship between a base class and an extended class based on the class definitions. No further intervention is necessary. In object databases, querying objects simply becomes a process of traversing the class structure that defines the relationships between objects. Query traversals should therefore tend to be simple, straightforward, and very application specific. We can therefore retrieve, update and store the objects we need when we need them, while maintaining their object relationships. For relational databases, on the other hand, universal query languages like SQL will be forced to deal with references which are not directly related to our objects, and we would be forced to translate our objects to tables and rows. This would encumber the project with the addition of Data Access Object (DAO) classes and would unnecessarily increase code complexity.

The Time-based Stream

As previously discussed, all objects which are created are placed in a Stream. The Stream construct is used to enforce a consistent and clearly defined chronological ordering. The temporal ordering created here is also what is used to add contextual information to the system.

The database management classes are mainly in 2 packages: atur.stream and atur.stream.substream.

The atur.stream package is responsible for implementing our Chronology-based storage of data i.e. "Temporal Storage", and will provide the automatic chronological ordering and context functionality discussed earlier. All file creation and manipulation activities are stored into this stream. The chronological order preserves the order and the method of the creation, similar to a journal or a diary. Once notified of activity in the system (through the Atur user interface), the system will get the data pertaining to the activity and store it into the stream. Data, such as the name of the data primitive (in our implementation, this would be the file), and the location of the file (its path or Uniform Resource Locator), the timestamp, the duration of the activity, and the type of activity (creation, modification, deletion etc) will be stored into the record.

The User-defined Substream

The package atur.stream.substream handles the "Substreams" for the user of a system. The Substream is actually the implementation of the notion of a user-defined Object as discussed earlier in this report. The Substream stores the member set of primitives and other Substreams which belong to an object as defined by the user. The definition of a user-defined object is as specified by the use of the Filters as explained in the previous section. To recap, the filter defines the properties of what constitutes an object as defined by the user, each Substream is associated with a unique Filter, and the Substream contains all the data primitives (or files) and other Substreams which fall into this filter.

Data Management Strategy

For retrieval, the overall strategy of the data storage packages atur.stream and atur.stream.substream is to obtain a root object from the PSE object store, traverse it in

Java code to find the objects we're looking for and do our work on those objects, and then commit them back to the object store.

Processing the retrieved objects is carried out in StreamManager. This code traverses the object lattice and does the useful work. When a retrieved object needs to use another object that is still on the disk, the StreamManager will retrieve it transparently.

Because this example deals only with objects, instead of objects and relational tables, we can use more object-oriented techniques to build it. In particular, our stream and Substream objects are very flexible, and can be extended in many ways as our needs change.

User Interface

Purpose

The purpose of the user interface is to present the user with a simplified intuitive view of the system, according to the notion of Streams and Substreams model. The interface will also capture the interactions of the user in order to store useful data into the user's Stream.

Implementation

The user interface is implemented using Java 2 Swing [11] classes. The user interface classes are implemented in the atur.ui and atur.ui.statusbar packages. The interaction between the streams/Substreams and the tree display model in the User interface is done through the use of "bridge" classes such the "SubstreamModel" class which keeps the stream and user interface models in sync.

Currently, the user interface is presented in a window, with three sub windows:

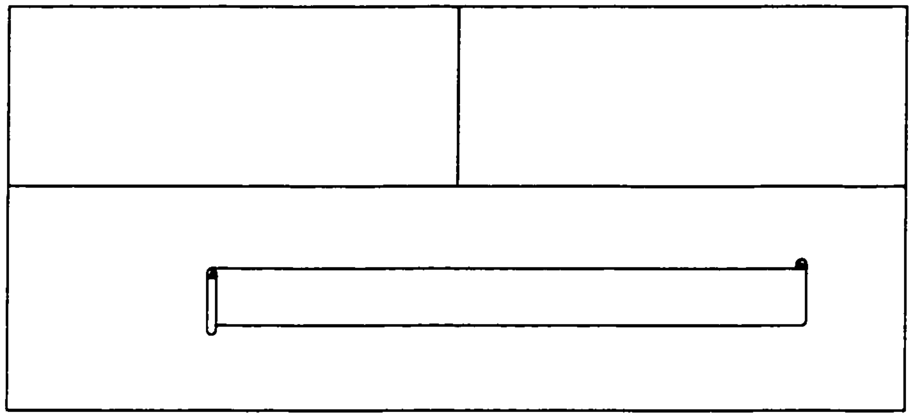


Figure 7-1: Basic Components of the "Atur" User Interface

The bottom window is used to present data stored in the main time-based Stream. The top left window is used to present objects as defined by the user's Substreams. The top right window presents all the user's defined Filters.

Example of System Usage

User A creates a stream, which currently contains 5 files:

1. An email containing a job ad for a java programmer: *Job.txt*.

2. A java source code file for the "Atur" project: *StreamManager.java*.

3. A list of contact information for people working on the "Atur" project: *Contact.txt*

4. The text for a seminar on Web Services in Microsoft Word format: *Web Services 101.doc*

5. A web page on Auto Maintenance and Repair: *AutoMaintenance.html*

The user creates 3 filters:

"Computer Science" filter – a filter which contains keywords and type information of files related to computer science.

"Java Files" filter – a filter which contains keywords and type information of files related to java technology.

"Hobbies" filter – a filter which simply contains the keywords related to personal hobbies.

The user creates and structures his Substreams as outlined in Figure 7-2:

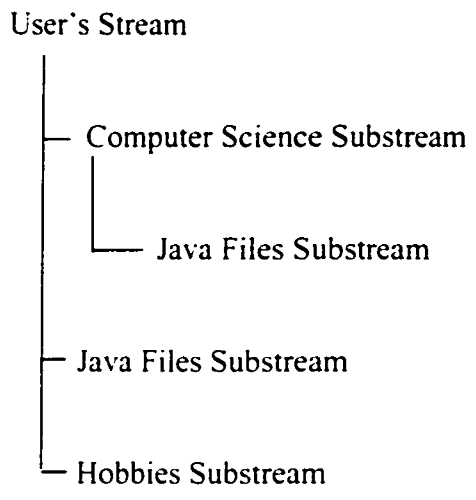


Figure 7-2: User's Stream and Substream structure

Note that the user has used the same filter twice (the "Java Files" filter), once at the top level, and once as child Substream to the Computer Science Substream

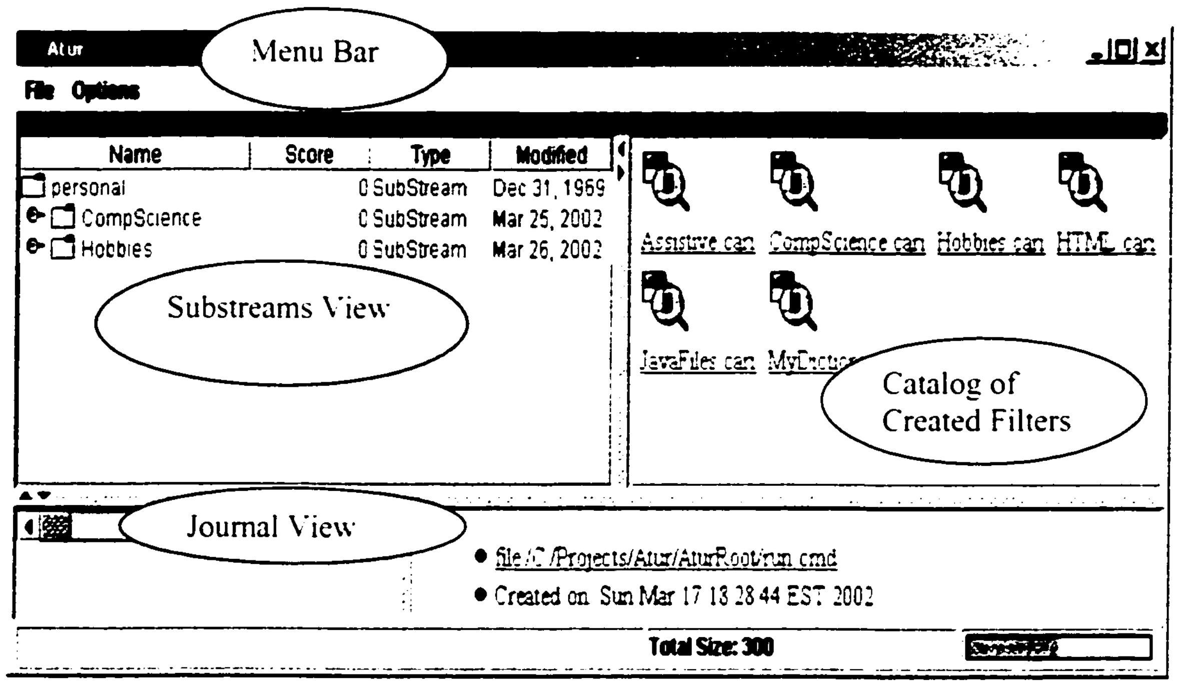
Based on the user's stream. the system presents the view as illustrated in Figure 7-3:

User's Stream

- Computer Science Substream: **Job.txt**
  - Java Files Substream: **StreamManager.java. Web Services 101 .doc**
- Java Files Substream: **StreamManager.java. Web Services 101.doc**
- Hobbies Substream: **AutoMaintenance.html**

Figure 7-3: Generated View of User's Stream and Substream structure

The figure below is an annotated screen capture of the Graphical User Interface the system presents user:



Name	Score	Type	Modified
personal		0 SubStream	Dec 31, 1969
CompScience		0 SubStream	Mar 25, 2002
Hobbies		0 SubStream	Mar 26, 2002

Assistive.can CompScience.can Hobbies.can HTML.can
JavaFiles.can MyPictur

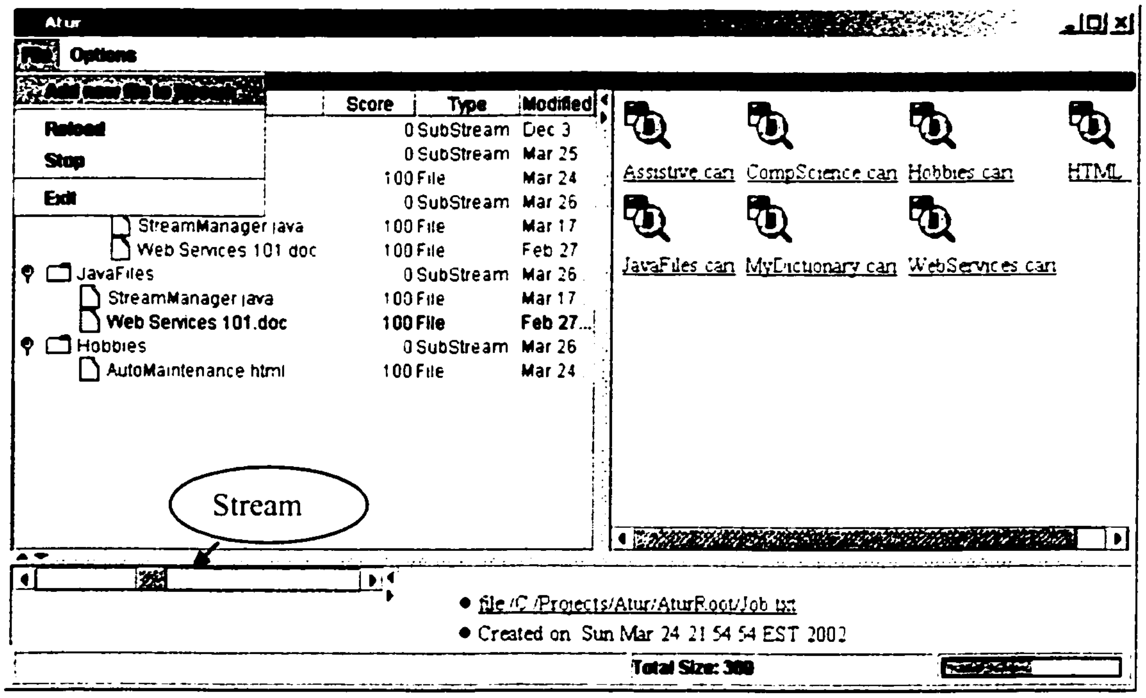
- file:/C:/Projects/Atur/AturRoot/run.cmd
- Created on Sun Mar 17 13:28:44 EST 2002

Total Size: 300

Figure 7-4: The "Atur" Graphical User Interface

Substreams are added to the root stream or to existing Substreams by selecting "File-Add new file to Substream" from the Menu Bar.

Note that the Journal view displays the user's Stream, which will record all known files in the system, i.e. all files created by the user. The scrollbar is used to quickly browse through all entries in the stream.



	Score	Type	Modified
	0	SubStream	Dec 3
	0	SubStream	Mar 25
	100	File	Mar 24
	0	SubStream	Mar 26
StreamManager java	100	File	Mar 17
Web Services 101 doc	100	File	Feb 27
JavaFiles	0	SubStream	Mar 26
StreamManager java	100	File	Mar 17
Web Services 101.doc	**100**	**File**	**Feb 27...**
Hobbies	0	SubStream	Mar 26
AutoMaintenance html	100	File	Mar 24

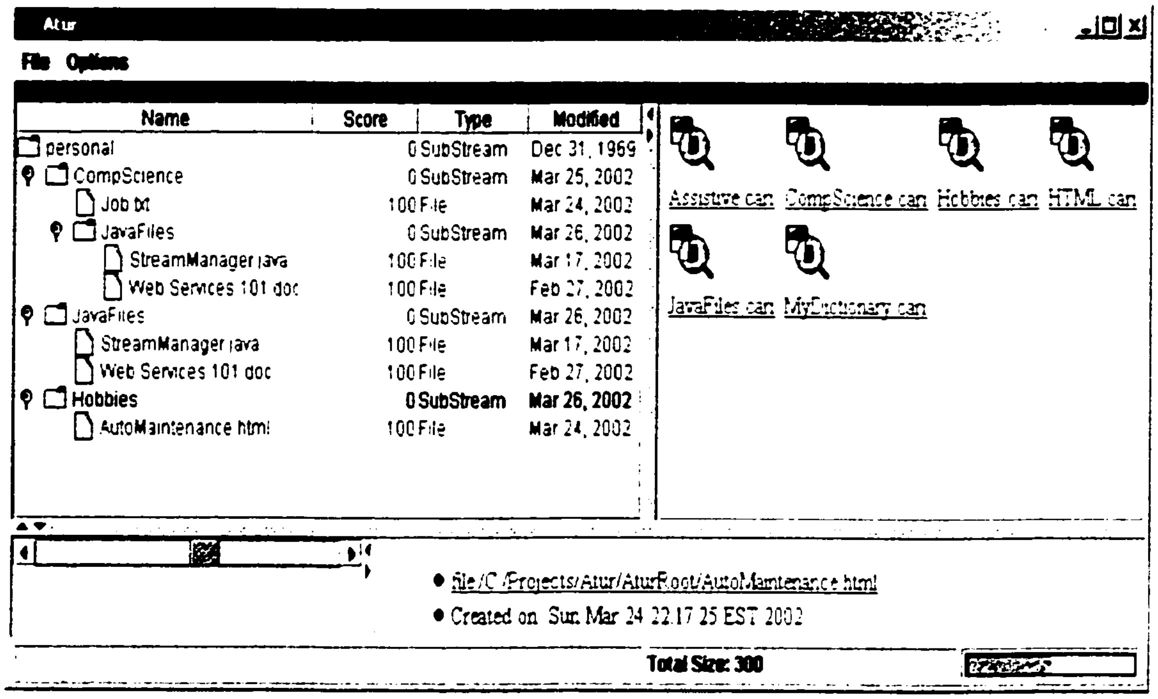
- file:/C:/Projects/Atur/AturRoot/Job txt
- Created on Sun Mar 24 21 54 54 EST 2002

Total Size: 300

Figure 7-5: Adding a Substream

Given the usage scenario for our example, the system will present the following Substreams

view



Name	Score	Type	Modified
personal	0	SubStream	Dec 31, 1969
CompScience	0	SubStream	Mar 25, 2002
Job txt	100	File	Mar 24, 2002
JavaFiles	0	SubStream	Mar 26, 2002
StreamManager java	100	File	Mar 17, 2002
Web Services 101 doc	100	File	Feb 27, 2002
JavaFiles	0	SubStream	Mar 26, 2002
StreamManager java	100	File	Mar 17, 2002
Web Services 101 doc	100	File	Feb 27, 2002
Hobbies	**0**	**SubStream**	**Mar 26, 2002**
AutoMaintenance html	100	File	Mar 24, 2002

- file/C /Projects/Atur/AturRoot/AutoMaintenance html
- Created on Sun Mar 24 22.17 25 EST 2002

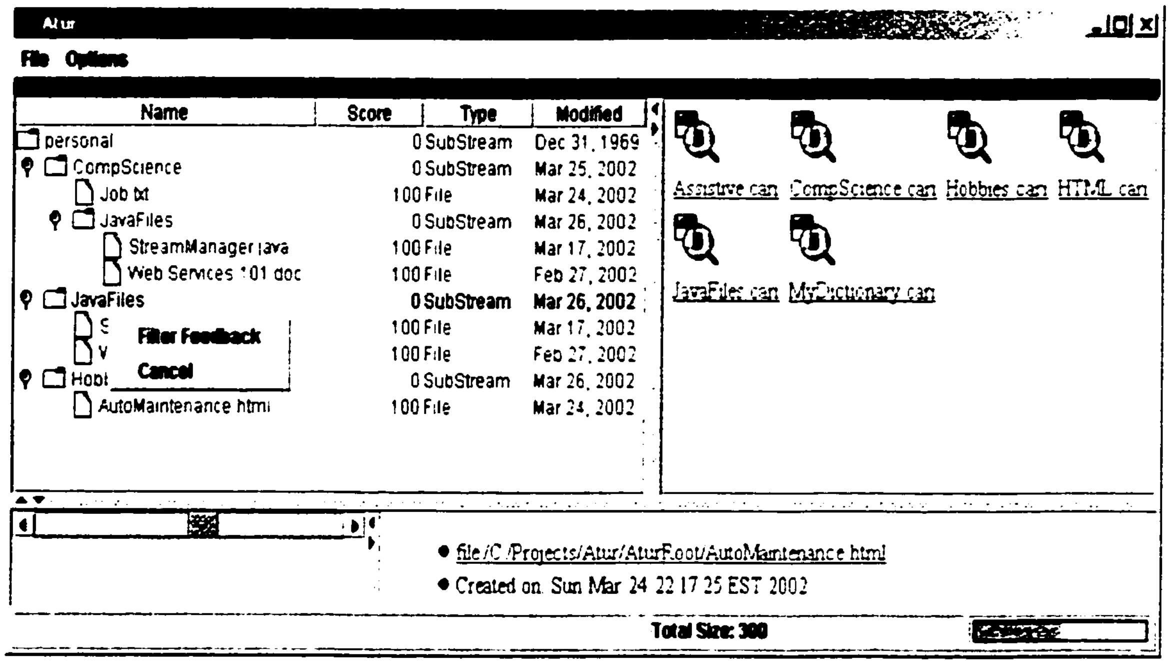
Total Size: 300

Figure 7-6: View based on example usage scenario

:

As can be seen in the "Score" column in the Substreams view, the system scores the files according to the filters, and groups them into the filters. We can also see that files belonging to a Substream also belong to all ancestor Substreams i.e. a Substream will only filter a set which belongs to the parent Substream.

Feedback can be passed back to the system through the GUI, by right-clicking the mouse on the relevant filter (Figure 7-7).



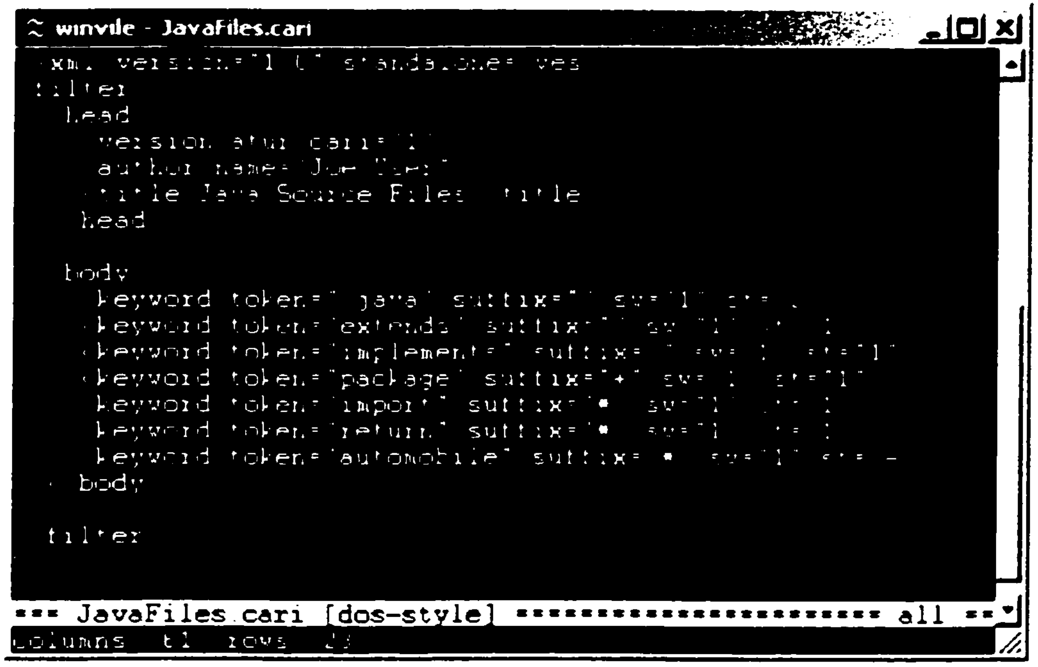
Name	Score	Type	Modified
personal	0	SubStream	Dec 31, 1969
CompScience	0	SubStream	Mar 25, 2002
Job.txt	100	File	Mar 24, 2002
JavaFiles	0	SubStream	Mar 26, 2002
StreamManager.java	100	File	Mar 17, 2002
Web Services 101.doc	100	File	Feb 27, 2002
JavaFiles	**0**	**SubStream**	**Mar 26, 2002**
S	100	File	Mar 17, 2002
V	100	File	Feb 27, 2002
Hobt	0	SubStream	Mar 26, 2002
AutoMaintenance.html	100	File	Mar 24, 2002

- file:/C:/Projects/Atur/AturRoot/AutoMaintenance.html
- Created on: Sun Mar 24 22:17:25 EST 2002

Total Size: 300

Figure 7-7: Feedback to update the filter

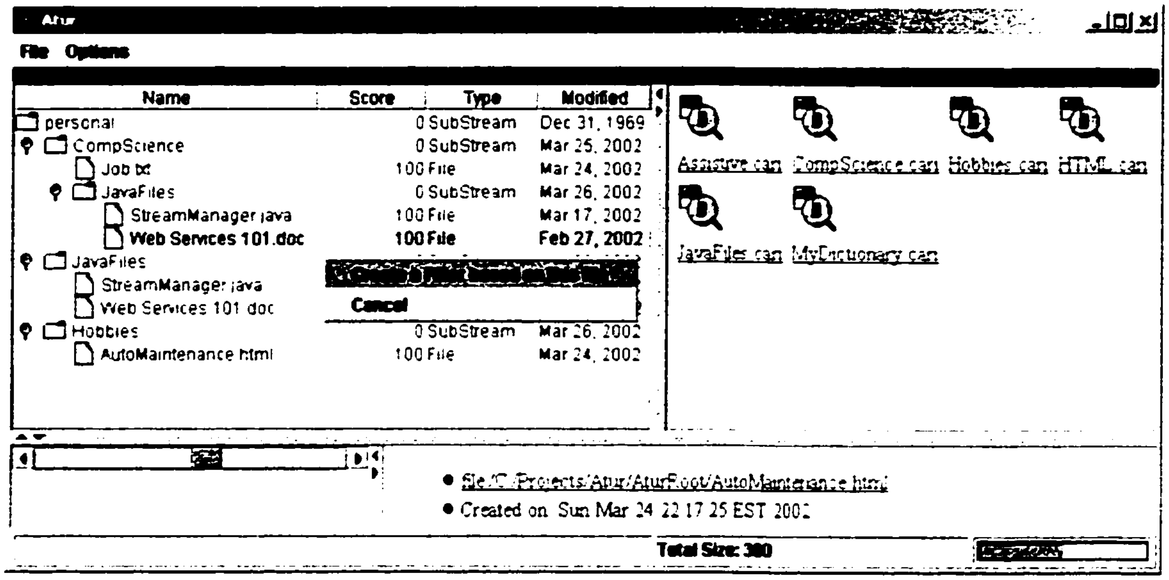
At present, the system's feedback mechanism will simply present the user with an editor to edit the filter file, as demonstrated in Figure 7-8:



```
winvile - JavaFiles.cari
 xml version="1.0" standalone="yes"
filter
  head
    version author cari="1"
    author name="Joe [illegible]"
    title Java Source Files title
   head

  body
    keyword token="java" suffix="" sw="1" [illegible]
    keyword token="extends" suffix="" sw="1" [illegible]
    keyword token="implements" suffix="" sw="1" [illegible]="1"
    keyword token="package" suffix="+" sw="1" [illegible]="1"
    keyword token="import" suffix="•" sw="1" [illegible]
    keyword token="return" suffix="•" sw="1" [illegible]
    keyword token="automobile" suffix="•" sw="1" [illegible]
   body

 filter

=== JavaFiles.cari [dos-style] ======================== all ==
columns 61 rows [illegible]
```

Figure 7-8: Editing the filter file

New filters can be created by selecting a file and right-clicking the mouse (Figure 7-9).



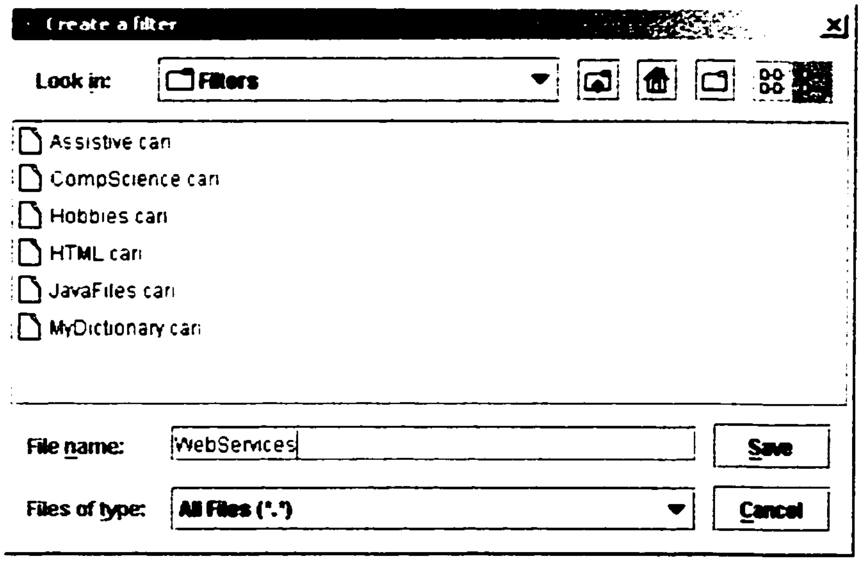
Name	Score	Type	Modified
personal	0	SubStream	Dec 31, 1969
CompScience	0	SubStream	Mar 25, 2002
Job txt	100	File	Mar 24, 2002
JavaFiles	0	SubStream	Mar 26, 2002
StreamManager java	100	File	Mar 17, 2002
Web Services 101.doc	**100**	**File**	**Feb 27, 2002**
JavaFiles			
StreamManager java			
Web Services 101 doc			
Hobbies	0	SubStream	Mar 26, 2002
AutoMaintenance html	100	File	Mar 24, 2002

Assistive can CompScience can Hobbies can HTML can JavaFiles can MyDictionary can

- file:/C:/Projects/Atur/AturRoot/AutoMaintenance.html
- Created on Sun Mar 24 22 17 25 EST 2002

Figure 7-9: Creating a new filter

This will call up the Create filter dialogue, as shown in Figure 7-10:



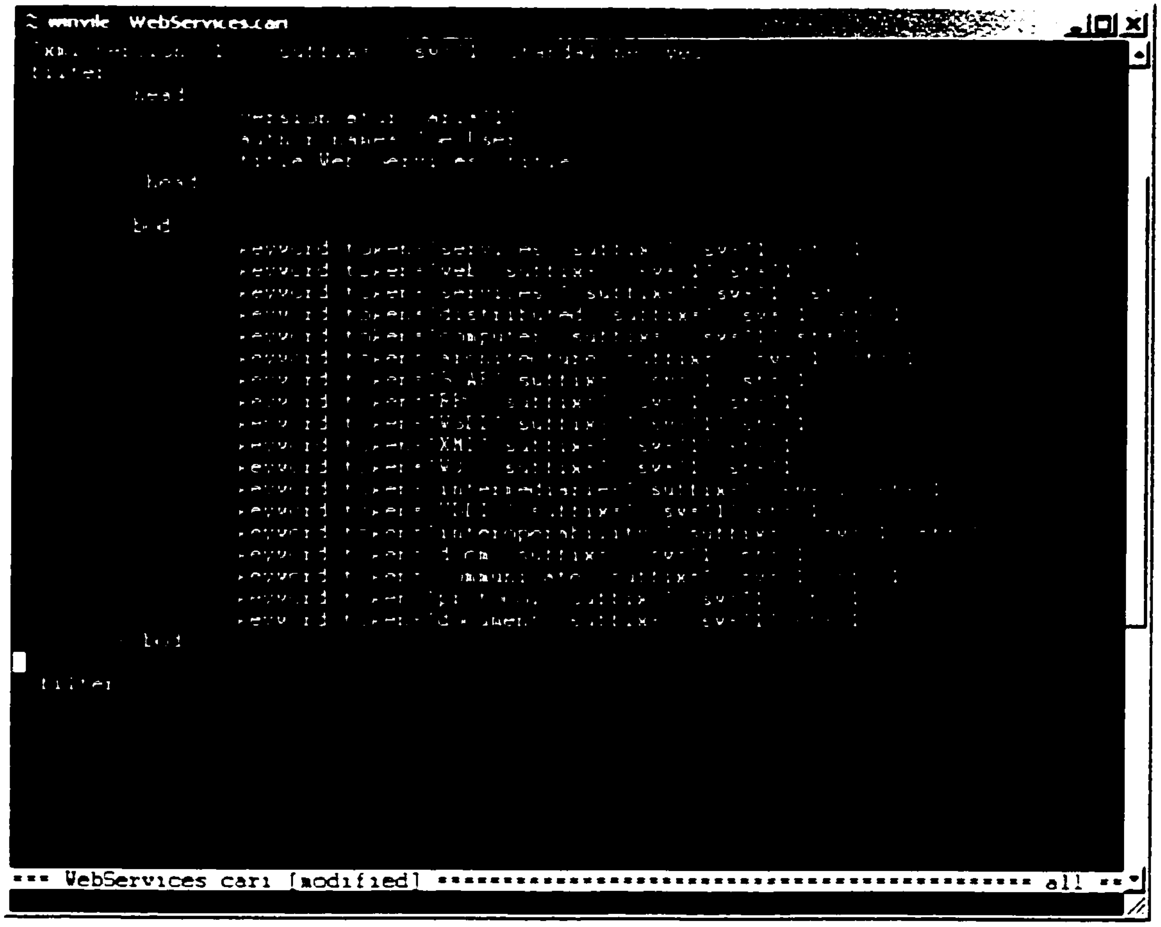
- Assistive can
- CompScience can
- Hobbies can
- HTML can
- JavaFiles can
- MyDictionary can

File name: WebServices

Files of type: All Files (*.*)

Figure 7-10: The Create Filter dialog

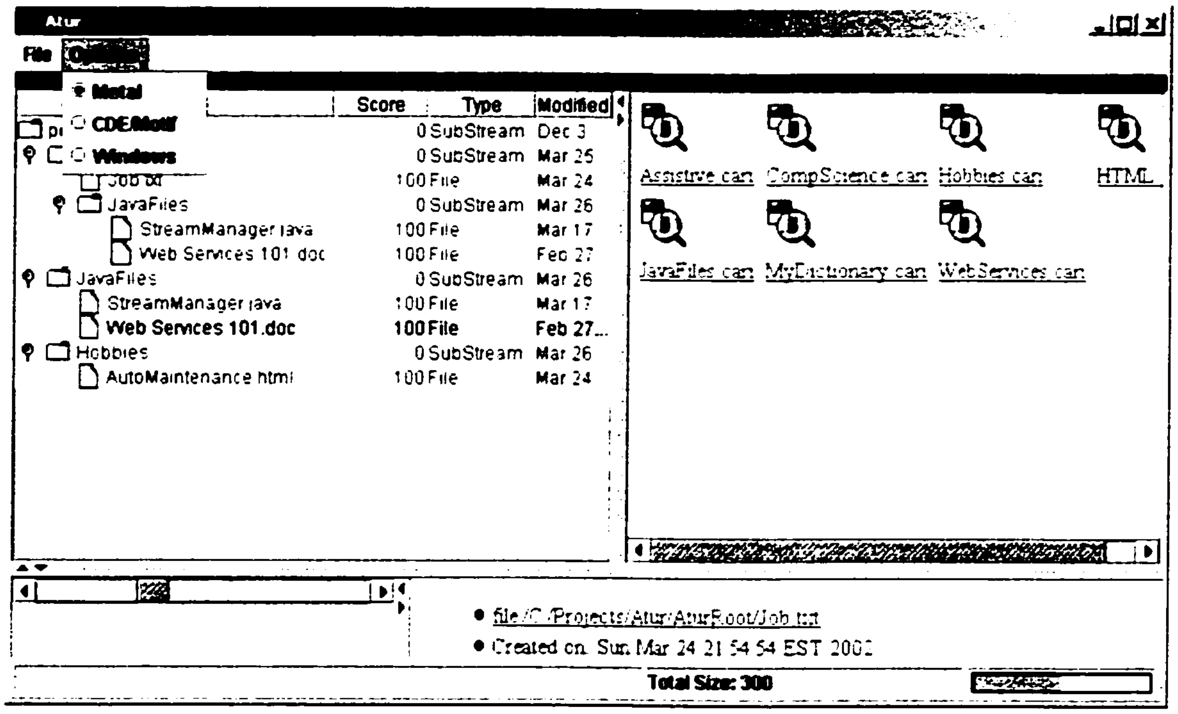
After a name for the new filter is typed-in by the user. the system parses the file and creates a filter of all unique words in an XML file (Figure 7-11).



```
xml [illegible] suffix [illegible]
filter
        head
                version [illegible]
                author name [illegible]
                title Web Services title
        head

        body
                keyword token= services suffix [illegible]
                keyword token= Web suffix [illegible]
                keyword token= services suffix [illegible]
                keyword token= distributed suffix [illegible]
                keyword token= computer suffix [illegible]
                keyword token= architecture suffix [illegible]
                keyword token= SOAP suffix [illegible]
                keyword token= RPC suffix [illegible]
                keyword token= WSDL suffix [illegible]
                keyword token= XML suffix [illegible]
                keyword token= W3 suffix [illegible]
                keyword token= intermediaries suffix [illegible]
                keyword token= UDDI suffix [illegible]
                keyword token= interoperability suffix [illegible]
                keyword token= [illegible] suffix [illegible]
                keyword token= communicate suffix [illegible]
                keyword token= [illegible] suffix [illegible]
                keyword token= document suffix [illegible]
        body
filter
*** WebServices cari [modified] ********************************************** all **
```

Figure 7-11: Defining unique words in a filter

For customization of the user interface, there is an "Option" menu item to select the "Look and Feel" of the system. In the example in Figure 7-12, the OS-neutral "Metal" "Look and Feel" is selected.



	Score	Type	Modified
pı	0	SubStream	Dec 3
	0	SubStream	Mar 25
Job txt	100	File	Mar 24
JavaFiles	0	SubStream	Mar 26
StreamManager java	100	File	Mar 17
Web Services 101 doc	100	File	Feb 27
JavaFiles	0	SubStream	Mar 26
StreamManager java	100	File	Mar 17
Web Services 101.doc	**100**	**File**	**Feb 27...**
Hobbies	0	SubStream	Mar 26
AutoMaintenance html	100	File	Mar 24

Assistive can CompScience can Hobbies can HTML

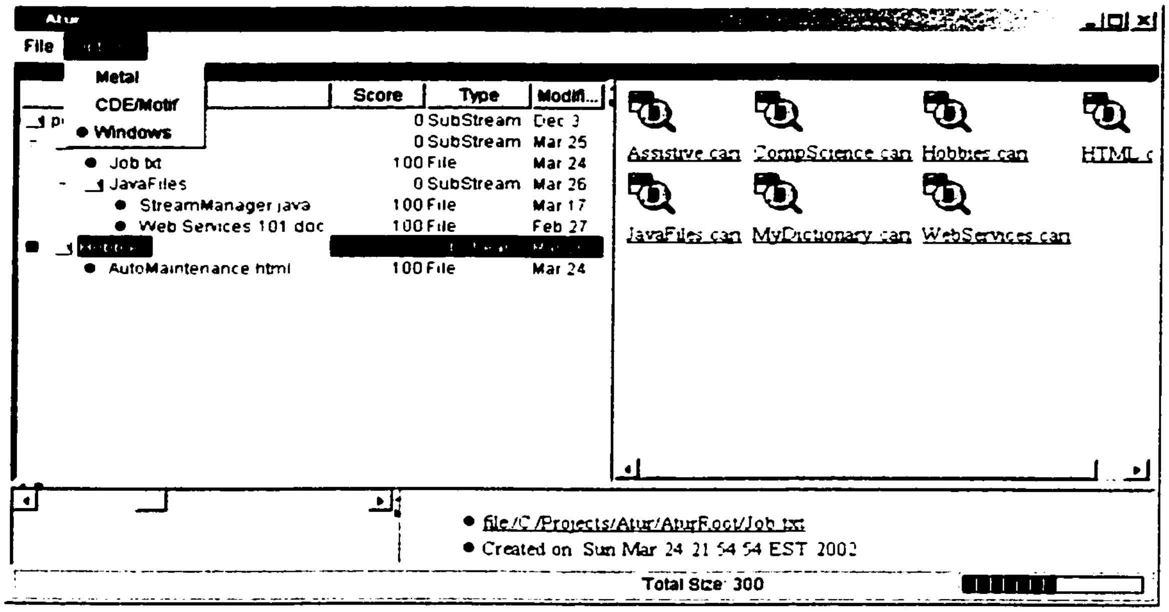
JavaFiles can MyDictionary can WebServices can

- file:/C:/Projects/Atur/AturRoot/Job txt
- Created on Sun Mar 24 21 54 54 EST 2002

Total Size: 300

Figure 7-12: Selecting the "Look and Feel"

If the user selected the windows "Look and Feel", she would be presented with the following view (Figure 7-13):



```
                          Score    Type         Modifi...
                              0 SubStream   Dec 3
                              0 SubStream   Mar 25
   ● Job txt                100 File        Mar 24
 - JavaFiles                  0 SubStream   Mar 26
   ● StreamManager java     100 File        Mar 17
   ● Web Services 101 doc   100 File        Feb 27
   ● AutoMaintenance html   100 File        Mar 24
```

● file:/C/Projects/Atur/AturRoot/Job txt
● Created on Sun Mar 24 21 54 54 EST 2002

Total Size 300

Figure 7-13: Selecting an alternative "Look and Feel"

Chapter 8 FUTURE WORK

This chapter discusses some of the current limitations of Atur. and areas for improvement in future work.

Performance Considerations

Performance is one of key factors in the usability of the interface. Due to performance demands, some components of Atur must be tuned to attain faster processing times. The parsing engine for the filters is a prime example. Due to the high overhead of XML parsing, commonly available schema-based XML engines, such as the Xerces XML parser from the Jakarta Apache Project, which use call-back mechanisms, could not be used. Instead, a custom parsing engine was built to ensure speedy performance of the filter engine. However, this creates the drawback of limiting the ease of extending the filter grammar. Any extensions to the filter grammar would entail changing the custom parsing engine, instead of simply adding call-back handlers to handle new XML tags. The performance overhead of the parsing engine will be likely be less important in the future: at the current rate of progress in the performance of client desktop systems, the time would soon come when high-performance client systems would present the opportunity for the parsing engine to use a commonly available XML engine.

Improvements to User Interface

Another limiting factor is the simplicity of the feedback engine. At present, feedback involves adding filter keywords to a filter file, which can later be manually edited by the user. The feedback interface and mechanism could be improved together the interface for creation of Substream filters. Improvements could be made with front-end interfaces with "wizard"-like properties that will guide the user through the filter-making and feedback processes.

Decoupling the Presentation Layer

Currently, the presentation layer is tightly coupled with the summarization and filtering engines. User data filters and processing can and should be abstracted into a server layer. The presentation or client layer can then be hosted on a separate machine.

This coupling also results in severely limiting the system's capabilities in the creation of multiple views for an object. Because the view is tightly coupled, extending the object definition language in order to support adding and extending dynamic view and access methods to different object types is problematic and time consuming.

Another significant benefit of refactoring the presentation layer would be to open up the possibility of using the same interface on different display hosts. This would also make it possible for all data and filtering preferences to exist on the same host.

Integration with Operating System

The Substreams layer works on top of the operating system layer. Thus, this makes it difficult for the user interface to be available to other applications on the system.

The Substreams layer can be integrated into the operating system layer, as an alternative to the traditional directory structure. This would enable all applications on a system to utilize the same Substreams interface when using file-system services.

With the continuing progress and maturity of the Java Accessibility API [12] and assistive technology on the Java platform, investigation could be done on the feasibility of Atur leveraging user interface information generated directly by the Java Virtual Machine through the Java Accessibility interfaces, as a replacement for the polling mechanism for file system activity.

REFERENCES

[1] G. Ballintijn and M. v. Steen, "Scalable Naming in Global Middleware," Vrije Universiteit, Amsterdam, The Netherlands, IR-464, Oct. 1999.

[2] T. C. Rakow, E. J. Neuhold, and M. Lohr, "Multimedia Database Systems - The Notions and the Issues," presented at Datenbanksysteme in Buro, Technik und Wissenschaft (BTW), GI-Fachtagung, Dresden, 1995.

[3] D. K. Gifford, P. Jouvelot, M. Sheldon, and J. OToole, "Semantic File Systems," presented at 13th ACM Symposium on Operating Systems Principles, 1991.

[4] P. Pazandak and V. Vesudevan, "Semantic File Systems," OBJS Technical Reports and Publications 1997.

[5] P. J. Shenoy, P. Goyal, S. S. Rao, H. M. Vin, and S. A. I. M. F. System, "Symphony: An Integrated Multimedia File System," presented at ACM SIGMETRICS Conference on Modeling and Evaluation of Computer Systems, 1998.

[6] S. Fertig, E. Freeman, and D. Gelernter, "Lifestreams: An Alternative to the Desktop Metaphor," presented at ACM SIGCHI Conference on Human Factors in Computing Systems, 1996.

[7] R. Mander, G. Salomon, and Y. Y. Wong, "The 'Pile' Metaphor For Supporting Casual Organization Of Information," presented at ACM CHI'92, 1992.

[8] B. C. Neuman, "The Prospero File System: A Global File System Based on the Virtual System.," *Computing Systems*, vol. 5(4), pp. 407--432, 1992.

[9] E. Freeman and D. Gelernter, "Lifestreams: A Storage Model For Personal Data," *ACM SIGMOD Bulletin*.

[10] T. Cook, "It's Ten O'Clock, Do You Know Where Your Data Are?" *MIT Technology Review 98*(1), pp. 48-53.

[11] "Creating a GUI with JFC/Swing," *The Swing Connection, Sun Microsystems*, 1999.

[12] "The Java Accessibility API Interfaces and Classes," *Java Accessibility, Sun Microsystems*, 2002.

APPENDIX

Class Diagrams

The following class diagrams illustrate the class design diagrams for the the following packages:

Package Name	Purpose	Figure
atur.cari	Sorting & Filtering Engine	Figure A-1
atur.filesystree	Creation and management of object trees for the user interface	Figure A-2
atur.stream atur.stream.substream	Data management and storage for Streams and Substreams	Figure A-3 Figure A-4
atur.ui atur.ui.statusbar	User Interface classes	Figure A-5 Figure A-6
atur.util	Utility and helper classes for common routines, debugging	Figure A-7

Table A-1: Java class packages for Atur

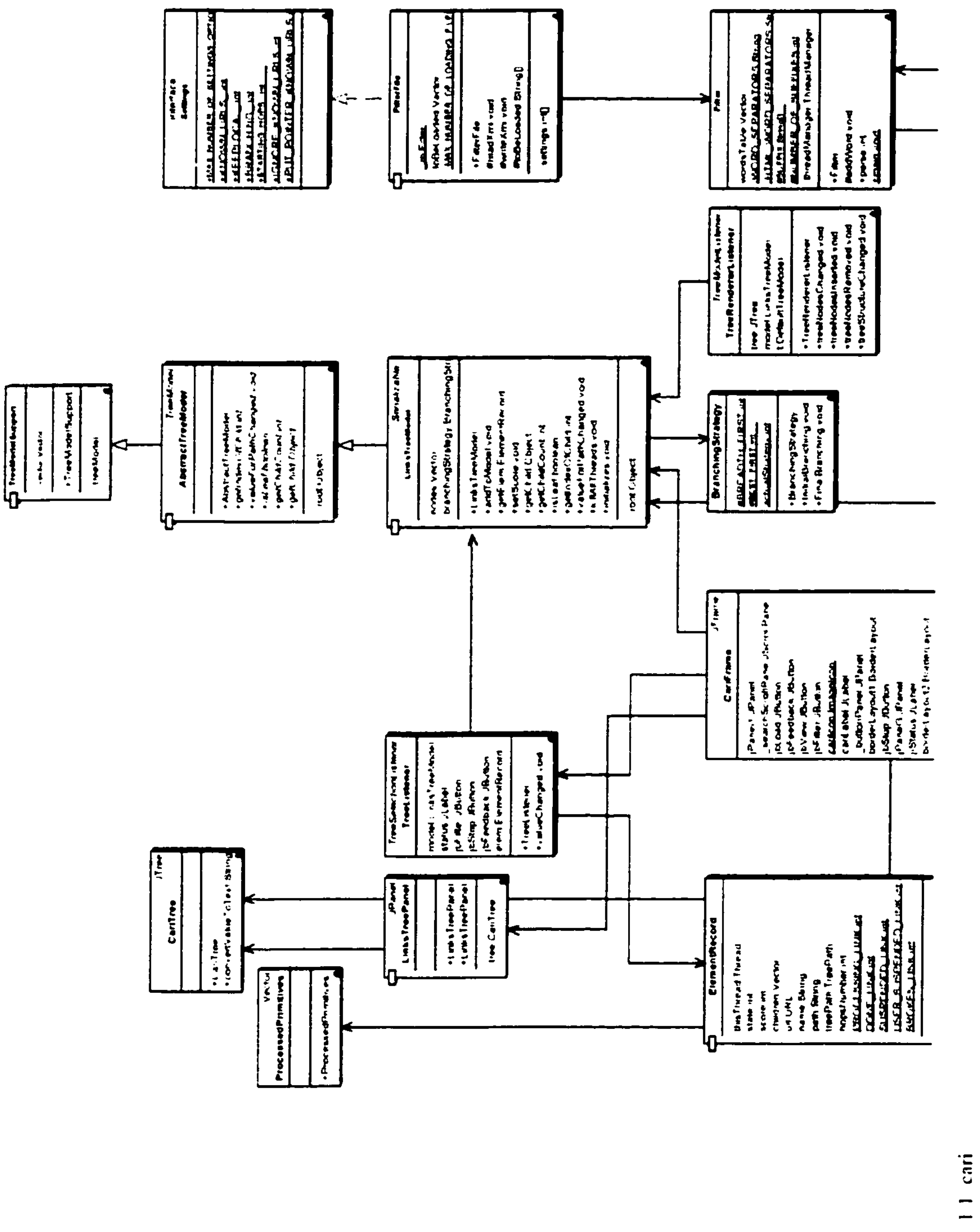


Figure A-1 (a): Class Diagram for atur.cari

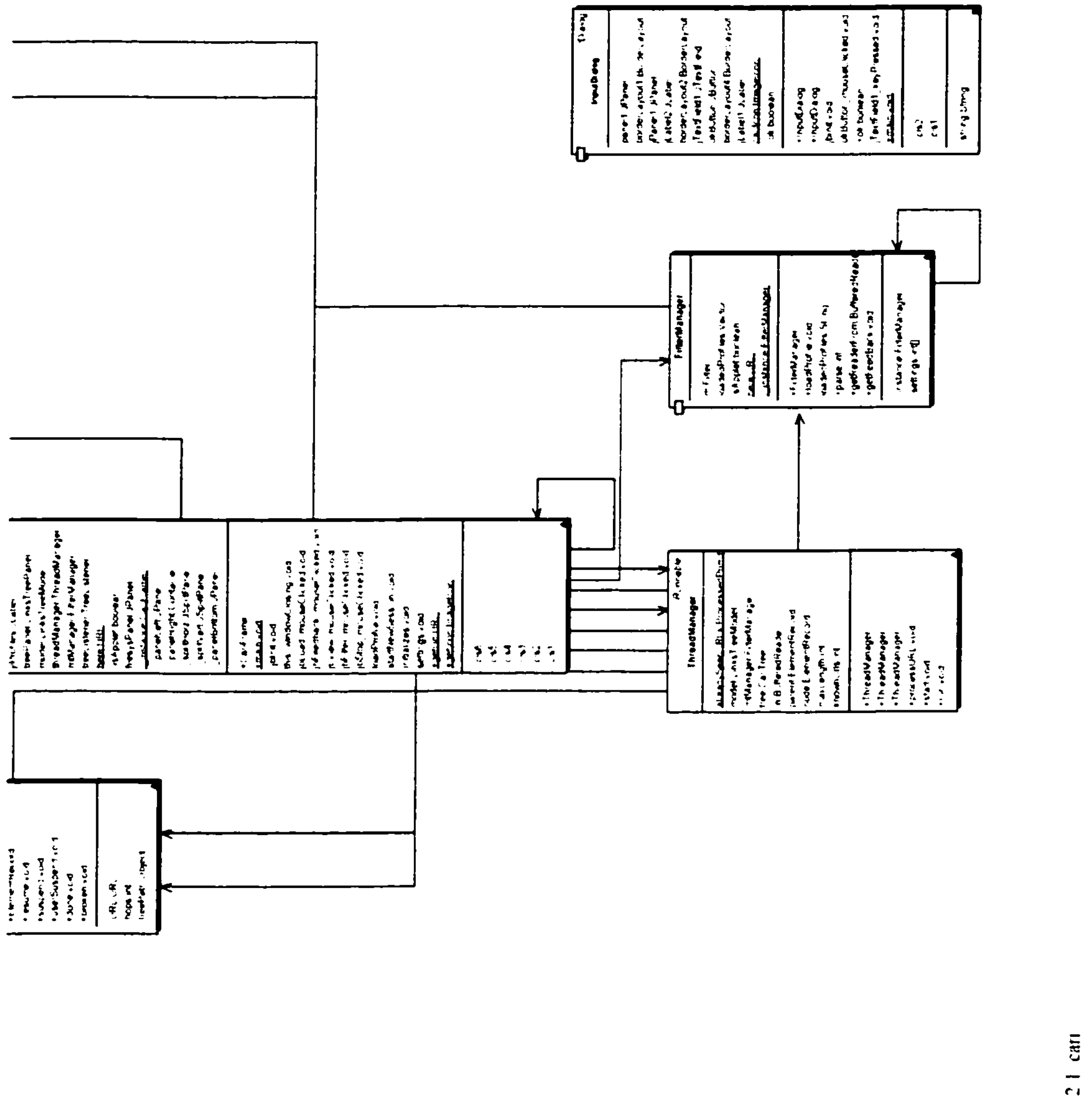


Figure A-1 (b): Class Diagram for atur.can

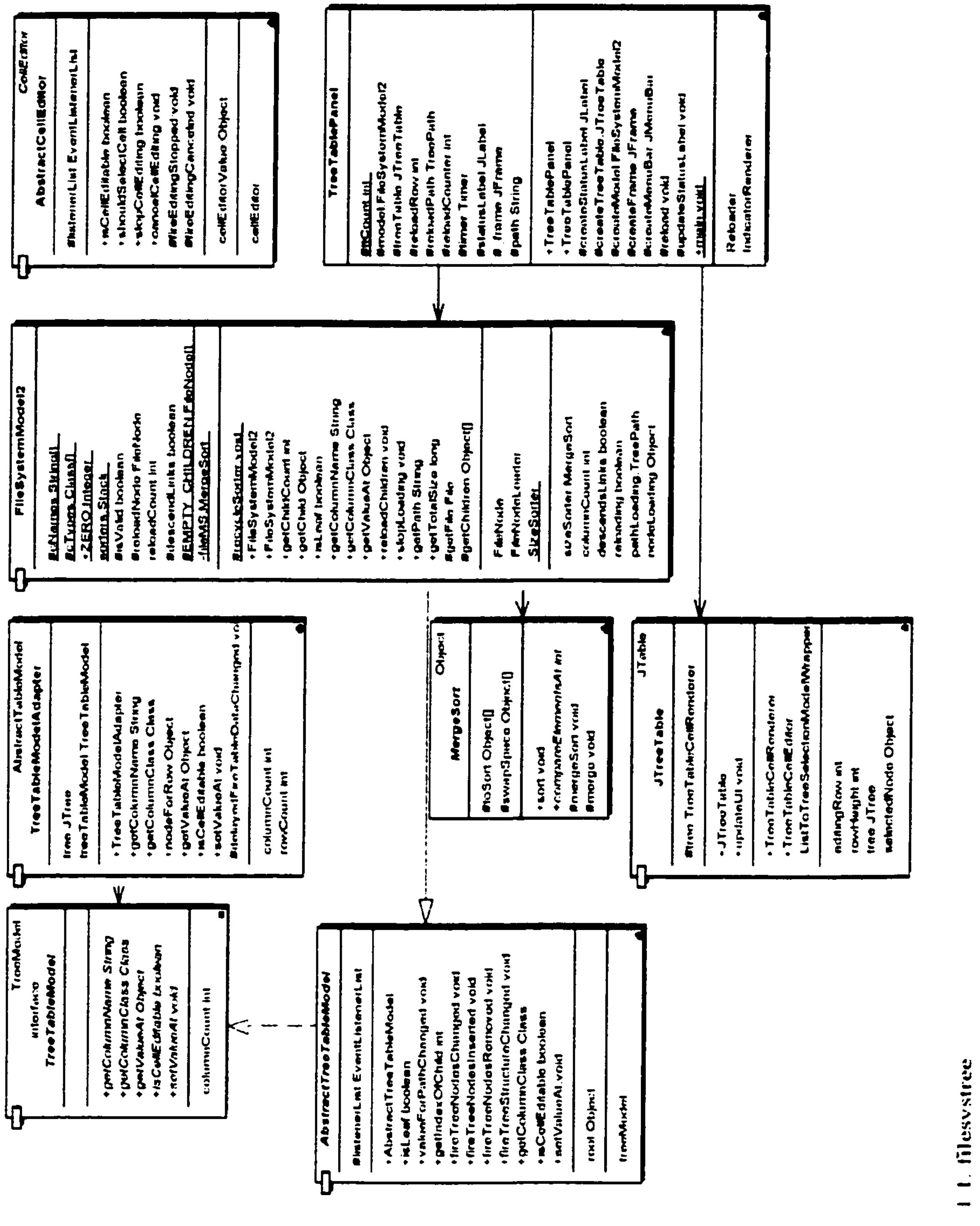


Figure A-2: Class Diagram for atur.filesystree

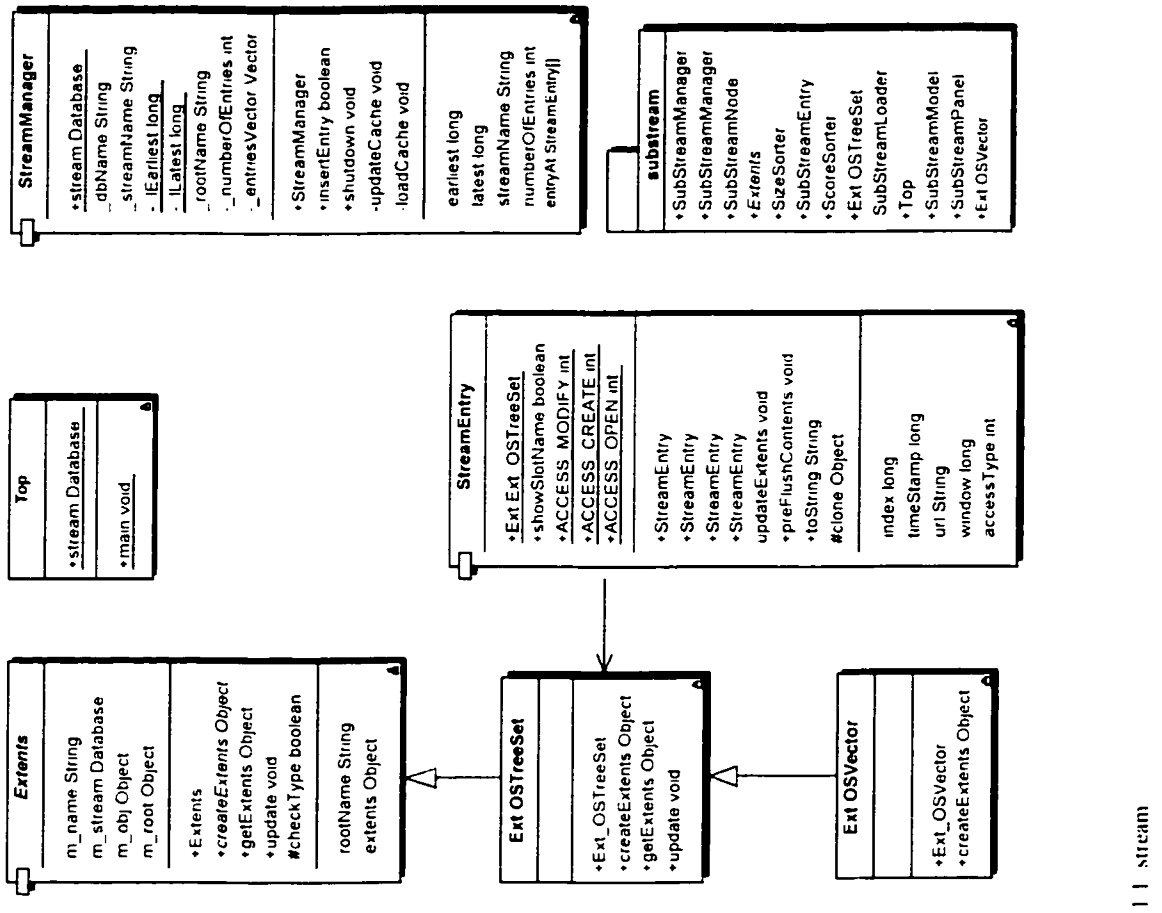


Figure A-3: Class Diagram for atur.stream

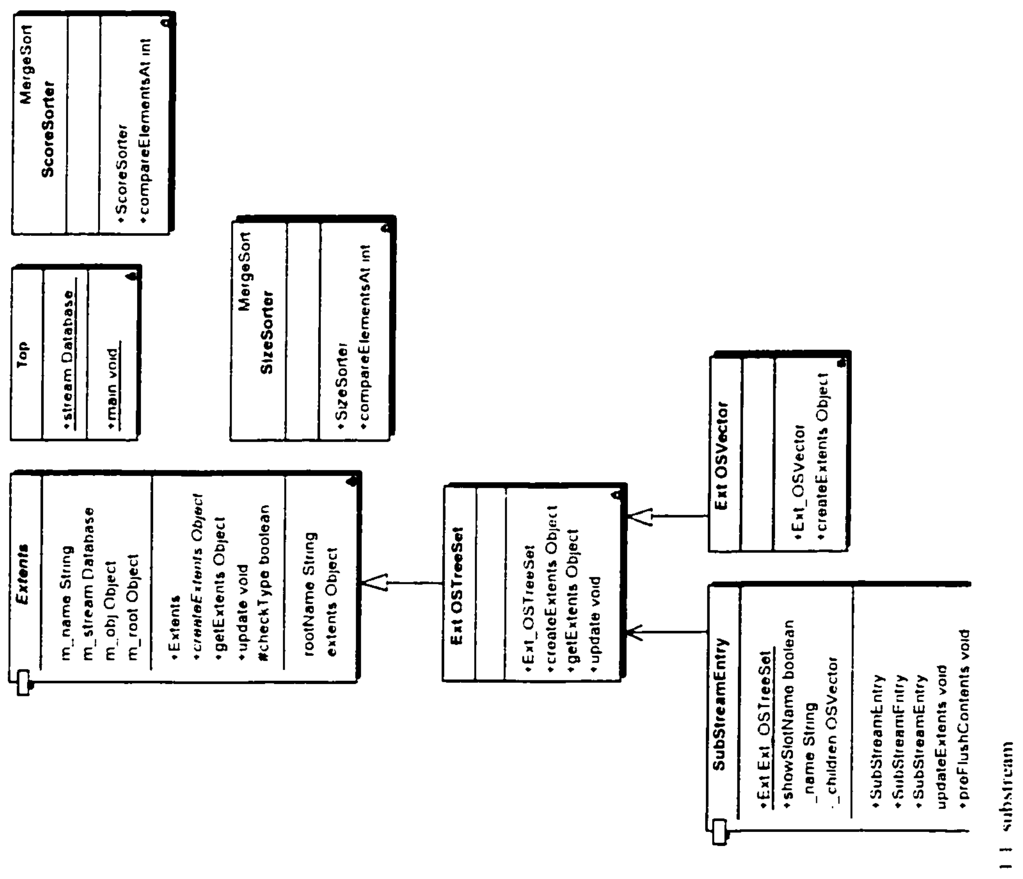


Figure A-4 (i): Class Diagram for atur..stream.substream

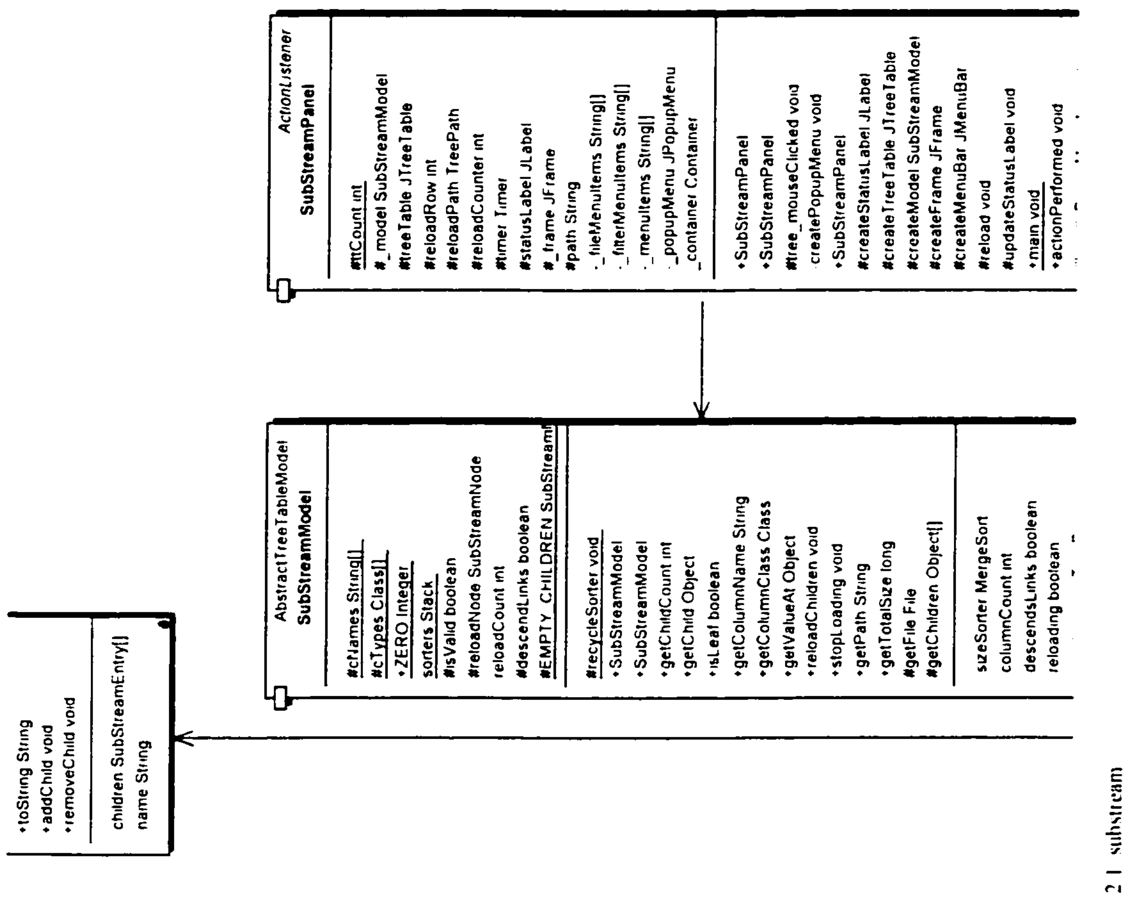


Figure A-4 (ii): Class Diagram for atur.stream.substream

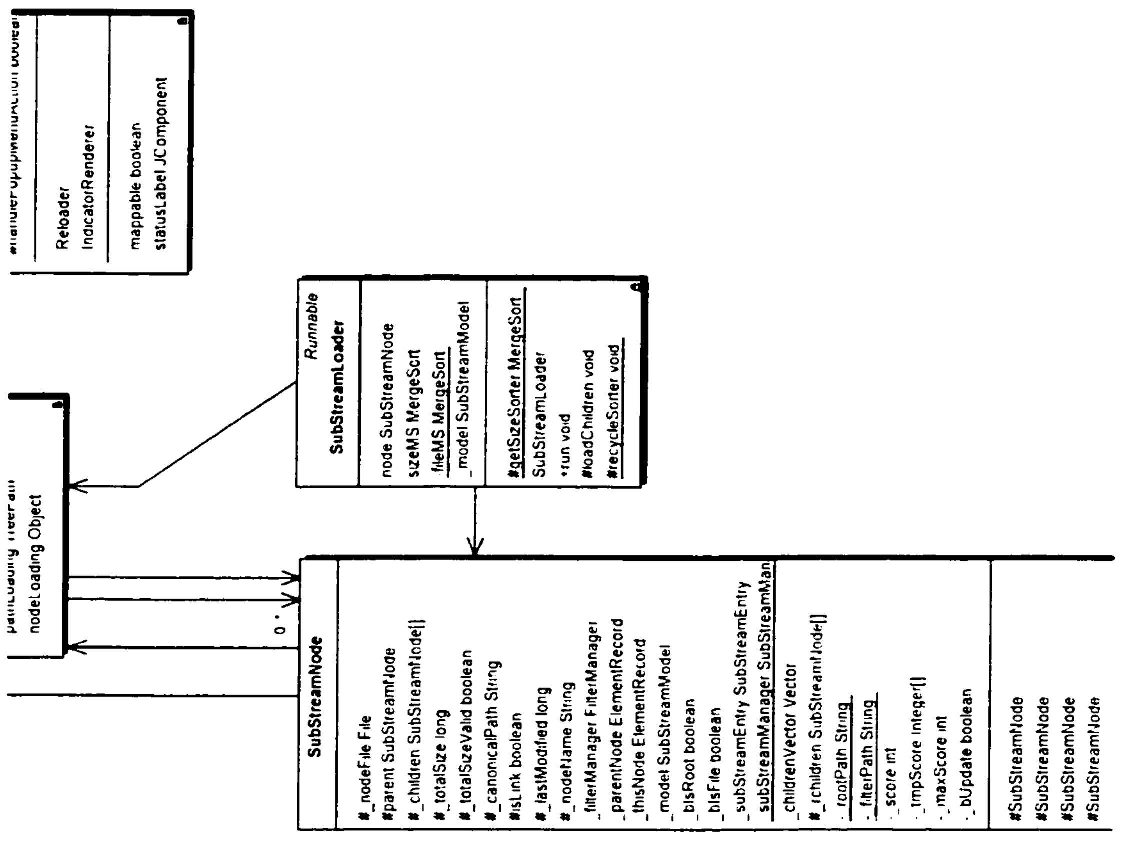


Figure A-4 (iii): Class Diagram for atur.stream.substream

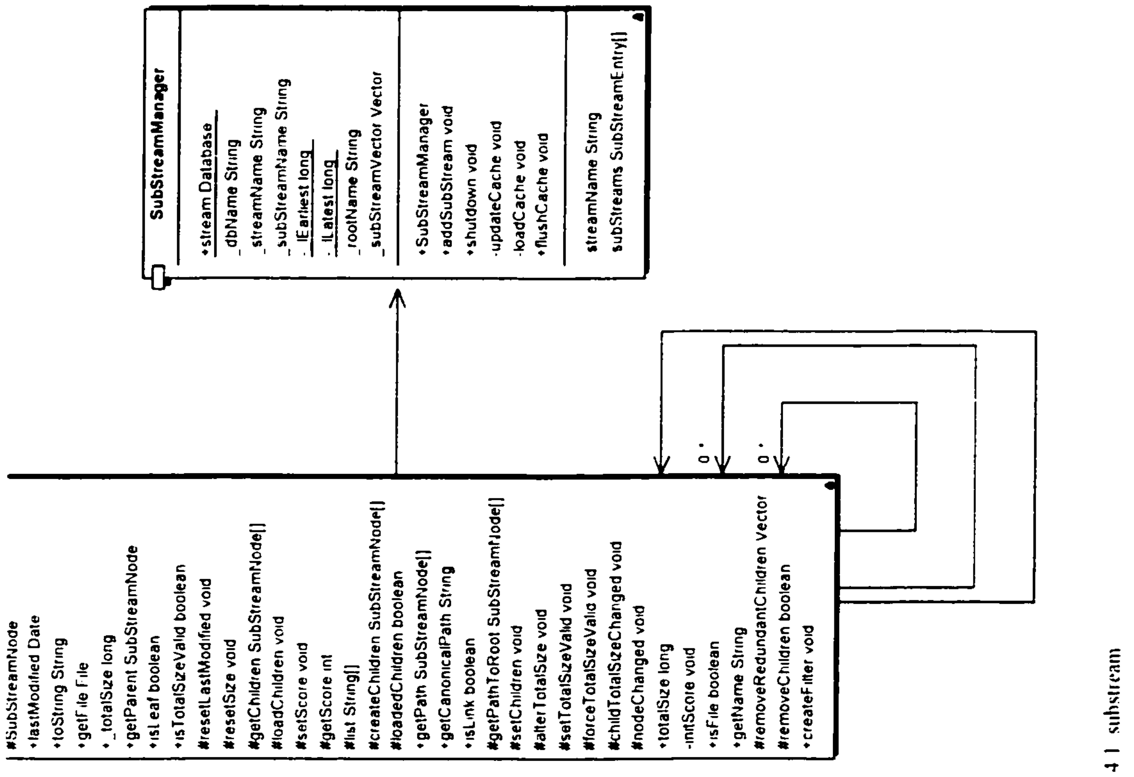
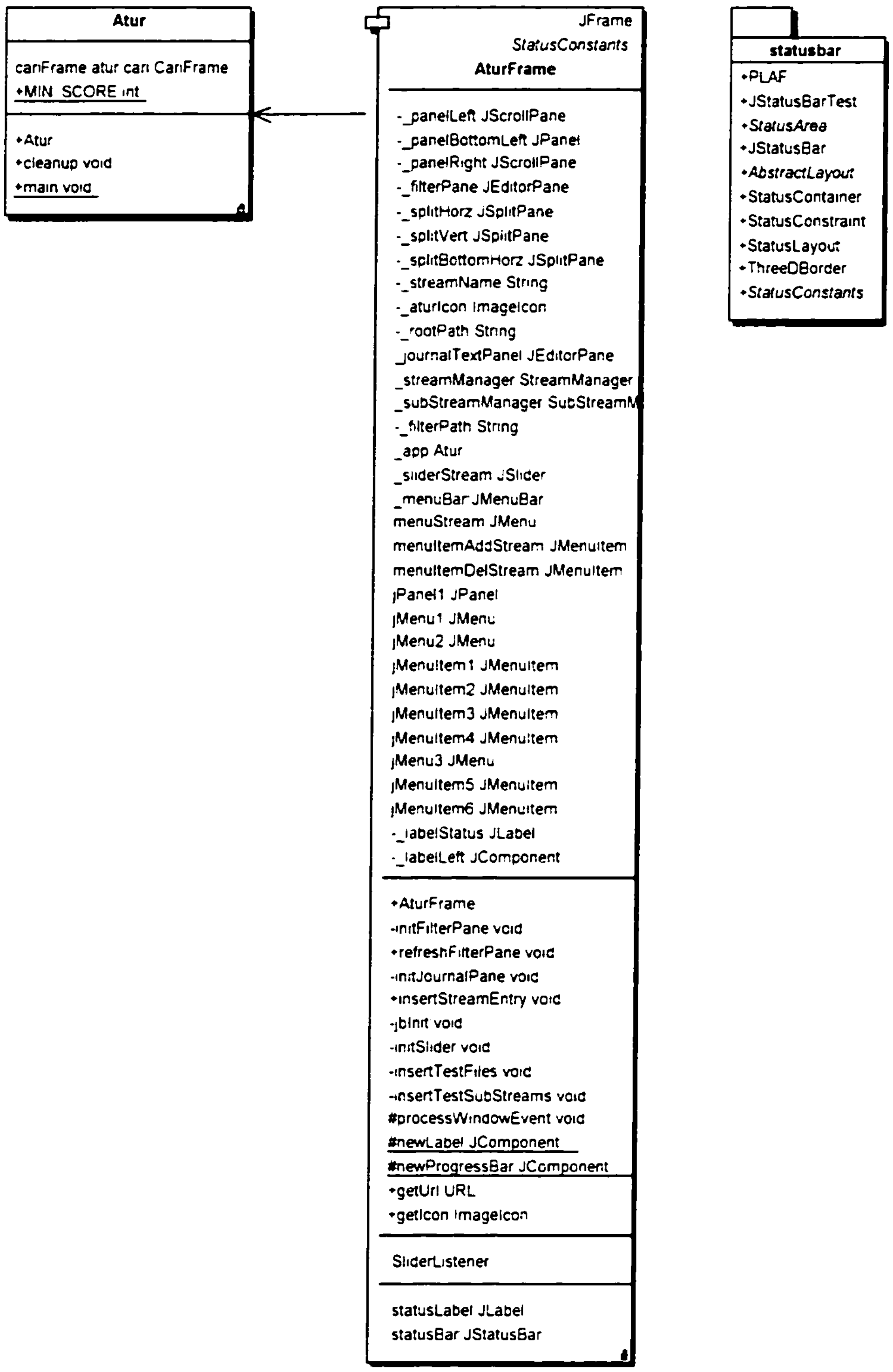


Figure A-4 (iv): Class Diagram for atur.stream.substream



1.1 ui

Figure A-5: Class Diagram for atur.ui

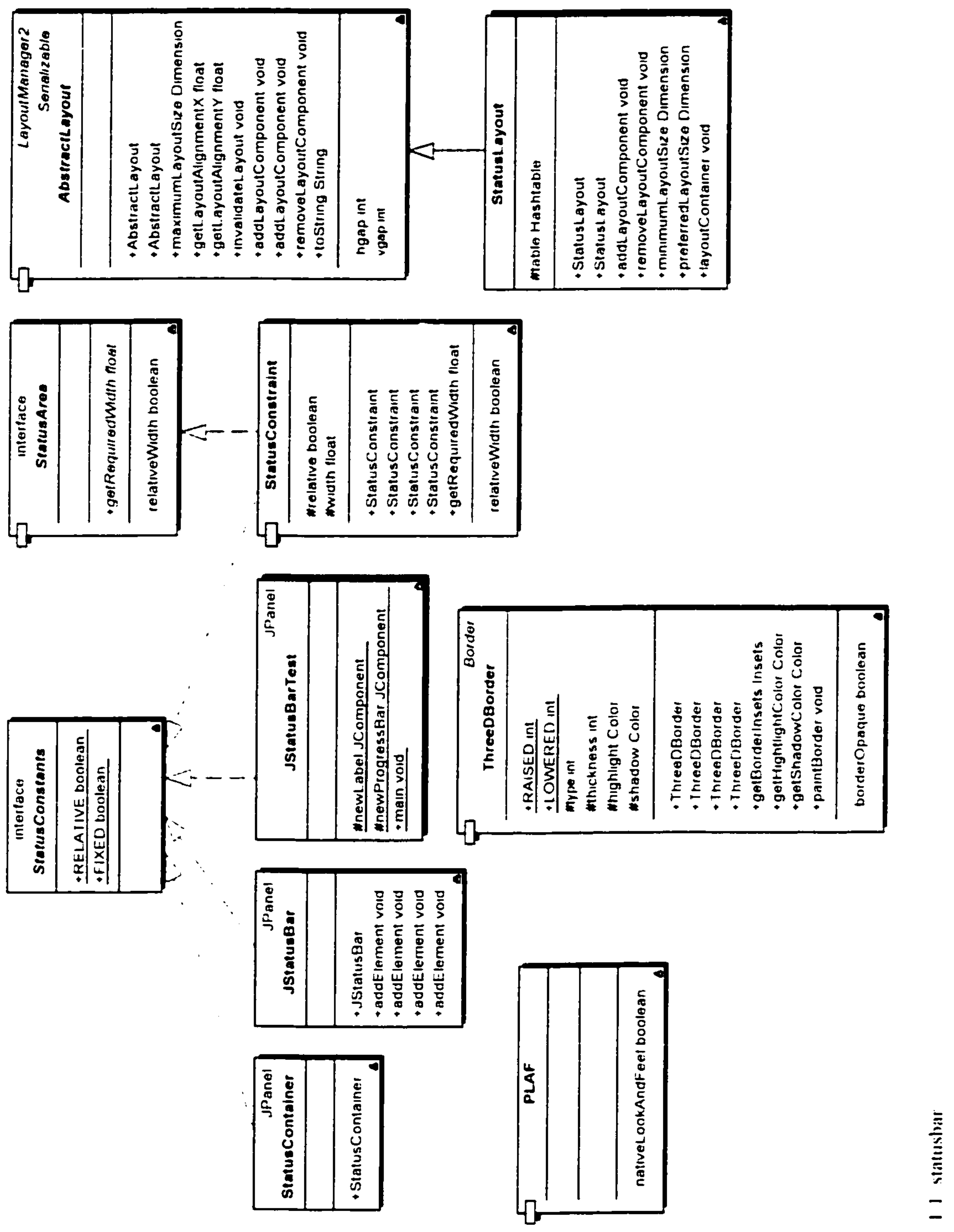
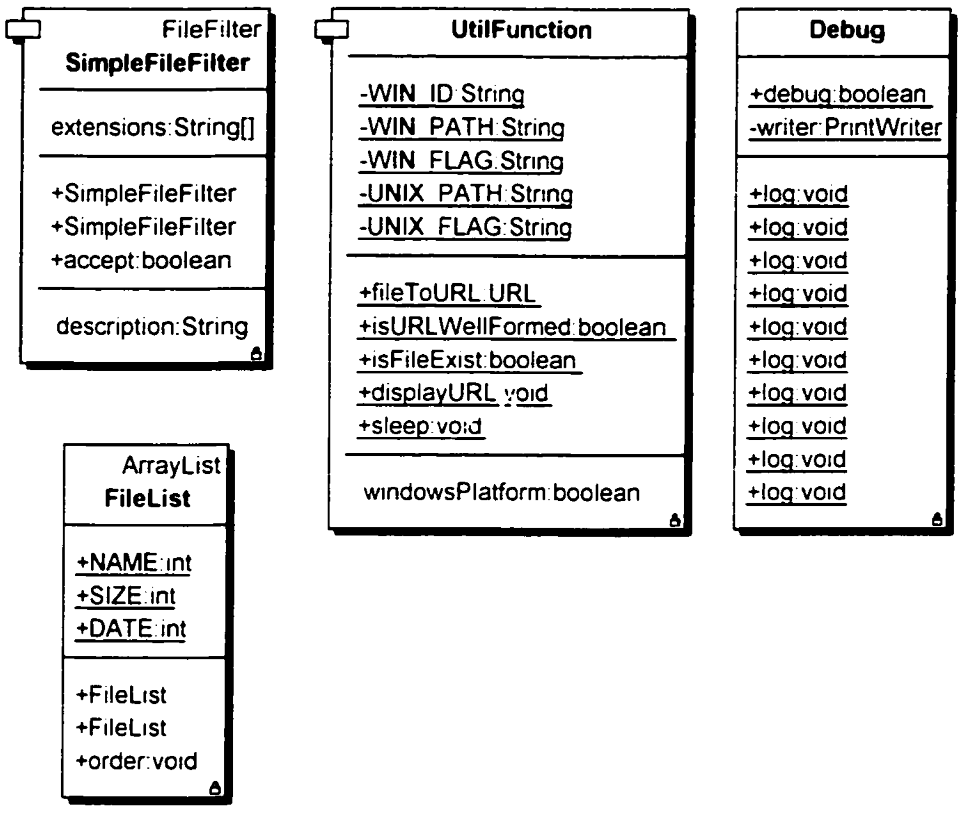


Figure A-6: Class Diagram for atur.ui.statusbar



1 1. util

Figure A-7: Class diagram for atur.util utility classes