Personal Picture Finder

A Softbot for the World Wide Web

Diplomarbeit
von
Christoph Endres

angefertigt nach einem Thema von
Prof. Dr. Wolfgang Wahlster
am Fachbereich 14 - Informatik -
der Universität des Saarlandes

Saarbrücken, Mai 1999

Eidesstattliche Erklärung

Hiermit versichere ich an Eides Statt, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, den 22.05.1999

Christoph Endres

Acknowledgements

It would have hardly been possible to accomplish my goal without the support and help of the following people. I would like to thank:
Wolfgang Wahlster for coming up with the interesting idea of this thesis and supporting the development. Also, for encouraging me to write, along with him, a paper about the *Personal Picure Finder* for an important national conference.
Markus Meyer for his critical and inspiring supervision.
The members of the projects PAN and RAP at DFKI GmbH for giving me the opportunity to do some practical AI work. Mathias Bauer and Dietmar Dengler for sharing their experience in AI and publishing matters. I was able to noticeably improve this work through their critical remarks, our long discussions, and their continuing support by sharing with me the latest news in AI research.
Interesting talks with members of other projects were as helpful.
I would like to thank Jochen Müller for tips and tricks about latest Java versions and state-of-the-art programming. His knowledge about Java seems to be infinite, as well as his eagerness to share it.
Wilken Schütz for introducing me to HyQL script writing.
Jens Haase for his cooperation in building an interface between *Personal Picture Finder* and the *Bitmap Information Tool*.
Markus Bolz and his team made a good job at administrating the computers.
James Hendler for recommending the *Personal Picture Finder*'s URL for the *Netwatch* column of the american *Science Magazine*.
Robert Wirth and Markus Burkhart for inspiring ideas and helpful discussions.
Ana García for proofreading this thesis and introducing me to the basics of layout for the first version of the webpage. Renato Orsinsi and Alexander Kowalski for spending a lot of time helping me find the final.
And especially, I would like to thank my parents for their support troughout the years.

Whatever the background, one is face to face with an inscrutable positronic brain, which the slide-rule geniuses say should work thus-and-so.

Except that they don't.

Isaac Asimov [Asi50]

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Today, the World Wide Web (WWW) is the biggest known information system. At the moment it contains about 300 million documents and is used by more than 40 million users.
In 2002, the—only 10 years old—WWW will have one billion users. Figure 1.1 shows the explosion of the internet after Tim Berners-Lee and Robert Cailliau [BLCG92] invented the WWW in 1992[1].

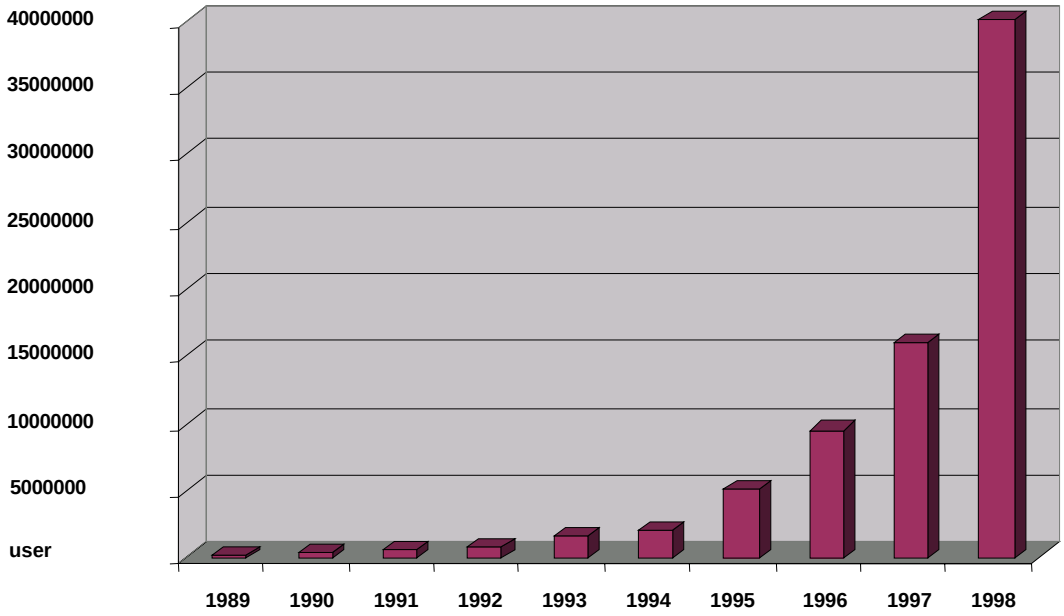

Figure 1.1: Internet expansion 1989 - 1998

Finding relevant information is an emerging problem which seems to get worse by the minute. An information system turns practically useless, if the information required is

[1]Actually the first submission of a Web proposal at CERN dates back to august 1990, but until the release of the Viola browser by Pei Wei in July 1992 this was a rather theoretical work.

stored, but not available to the user. In the case of the internet, information might get hidden because of *information overload*. A user then might need assistance while searching the *World Wide Web* in order to find the information he is looking for. Index-based search engines were first used to solve this problem and continue to be a good approach. Clever indexing, on the other hand, is an enormous problem in for a huge amount of frequently changing data. No search engine has achieved this goal so far. The best indices today, cover up to 40 percent of documents. Output of scripts and database-generated pages are not indexed at all.
The next step in the evolution of tools assisting the user on his way of finding information on the internet were metasearch engines. The idea is to simply request several search engines and collect their results. Still, not all the available information is covered[2] and the links provided may not work, eventhough the chances of finding the desired information were increased.
Filtering, sorting and compressing the resulting information still is time consuming. *Dedicated services* took place in the process of finding information. Its goal is to provide one specific piece of information, e.g. by looking up the email address of a person.
A higher level of assistance for the user is provided by so-called *softbots*.
Being intelligent agents with learning capabilities, softbots can autonomously transform transactions in the way intended by the user. Their aim is to facilitate the users access to the internet[3]. With a manyfold of techniques, it is now possible to evaluate given information, to compress it, combine it with other information and present it on *virtual webpages*. Those pages are created on the fly by the softbot during its research aiming only to answer exactly the users question. Animated characters can be used to present the result (*presentation agent*, see [ARM98]) or to communicate with the user in order to narrow down his request. The evolution and improvement of softbots will change the appearance of the web over the next few years and it will make it accessible to an even larger group.

1.2 The Personal Picture Finder

Personal internet assistants (PIA) are programs with the following features:

- Knowledge about the internet and the WWW.
- Ability to recognize what the user wants to know, by interpreting his requests in the right way and transforming them into the appropriate queries to the different information services and document sources.
- Capability to process information (*information processors*) for collecting, filtering, combining and presenting search results.

[2]Talking about the percentage of information covered one should keep in mind that a full coverage is not necessarily desirable. Several human maintained search engines explicitly *exclude* for example pages containing racistic material.

[3]It is not necessary to restrict the softbots domain to the WWW only. Other internet services can be used as well for acquiring information, for example ftp or gopher, or even a simple finger demon.

We call these PIA *netbots*, an acronym for inter**net** ro**bots**[4]. Like robots, netbots perform work for and in interaction with their user. The difference is the restriction of the netbots' domain to the internet and of his work to be solely intellectual and not physical. The purpose of a softbot is to access internet services in commission of the user, to use and to interact with those services. According to the "Less is more"–philosophy the netbot looks up the user-specific relevant information only.
In this thesis I will describe a new netbot called *Personal Picture Finder* [EMW99].
The *Personal Picture Finder* was developed within the project PAN[5] at the German Research Center for Artificial Intelligence, DFKI GmbH. Its objective is to look up portrait pictures of persons on the web. Given the name of a person, the *Personal Picture Finder* consults relevant information sources, extracts and analyzes pictures from webpages, to then present the resulting pictures with a reference to the pages containing them. The *Personal Picture Finder* is available online (http://finder.dfki.de:7000/). The user has the possibility to give a feedback to the results and thus improving the performance of the *Personal Picture Finder*.
As a personal internet assistant the *Personal Picture Finder* can support journalists in the search of pictures of prominent people on the internet. It is checking out the whole internet for photos of the requested person, and eventually presents them in a suitable way.
Also, for an illustrator wanting to put the photo of a prominent person into a graphic, the *Personal Picture Finder* is a time and labor saving tool. An interesting application of the *Personal Picture Finder* is finding copyright violations of information providers. The copyright-holder of an image simply checks if the photo occurs somewhere else without his permission.
For private or business use it might be useful when preparing a meeting (conference, picking up somebody at the airport, etc.) with a person one can not identify visually.
Fans or fanclubs can use the *Personal Picture Finder* in order to find pictures of their idol and maybe collect them on CD-ROMs or in fanzines.
Tracing criminals is another buzzword in this context. It is obvious that criminals will not put a private homepage on the WWW, but one might find pictures of a larger group of people in another context containing the person one is looking for. This will surely work only if the name is mentioned in a textual annotation of picture. Another source for pictures of a criminal are the webpages of institutions like the FBI in the US or the Bundeskriminalamt in Germany. At least on the internet, the international cooperation and collaboration of police might work.

[4]The word *robot* was first used in 1921 by the czech writer Karel Čapek in his play "RUR - Rossum's Universal Robots". Its origin are the czech words *robota* (compulsory labor) and *robotnik* (slave).[Kur90]
[5]PAN is an acronym for **P**lanning **A**ssistant for the **N**et.

1.3 Problem specification

The specification for this thesis was to describe and implement the *Personal Picture Finder* including the following features:

- The *Personal Picture Finder* provides a WWW service for finding portrait pictures of persons which uses existing search engines and information infrastructures publicly available.
- Parallel pull technology is used.[6]
- The implementation is modular and realized in the Java programming language.
- Communication overhead is avoided by using server side computation (servlets).
- The system includes filter for graphics file formats on the web. The decision which pictures to be presented to the user is based on checking parameters like width, height, depth and compression of the graphics file.
- If possible, the HyQL interpreter developed in PAN is integrated in the *Personal Picture Finder* system.
- Requests and user feedback are protocolled. This information is stored in a database.
- Using low-level machine learning, the *Personal Picture Finder* should improve its graphics filters by learning from the collected data.

The original problem specification did not include the interface to the *Bitmap Information Tool BIT* developed by Jens Haase that turned out to be a useful extension.

1.4 Overview

The rest of this document is organized as follows.
Chapter 2 gives a survey on the *State of the Art* of implementing internet agents and search engines, and building intelligent user interfaces. Different kinds of search engines and agents will be introduced.
In Chapter 3, the underlying technology will be discussed: *BIT*, a *Bitmap Information Tool* currently under development at project *AIA*[7] adds useful information to the analyze of pictures. The *Hypertext Query Language HyQL* developed and prototyped in project *PAN* is a powerful tool for specifying information sources and information extraction operations. The programming environment *Java* used for the implementation of the *Personal Picture Finder* offers several features that facilitate the development of a softbot. A primer of the several *graphics file formats* is given, and explained why there are so many different formats to store graphics data in. The graphics file formats used on the web

[6]Parallel pull refers to concurrent WWW access and is described in Section 4.1.2.
[7]AIA is an acronym for Adaptive Communication Assistant for Effective Infobahn Access.

are discussed in detail. In order to store previous results, a *database* is needed and will be introduced as well. *Machine Learning* provides an interesting approach to automated improvement of the agent. *Persona*, a life-like character, was used for introducing the user to the *Personal Picture Finder* in an older implementation.
Chapter 4 provides an in-depth look at the *Personal Picture Finder*. The mode of operation, the concept of *parallel pull*, the architecture, and an elegant trick for accessing a database under difficult circumstances are explained in detail. The following sections explain special features like the *Minifinder*, and a former release of the *Personal Picture Finder* containing *Persona*. The *Personal Picture Finder* available to the public was designed to be fast, hence some features could not be added there. Section 4.4 informs about an experimental version of the *Personal Picture Finder*. This version is not accessible online and has some interesting new features and interfaces to other systems. Experiments with *Machine Learning* is described next. Some statistics try to add a bit more transparency to the average user of the *Personal Picture Finder*. Some other applications, which emerged as by-products through the development of the *Personal Picture Finder*, are presented in Section 4.7.
At the end of Chapter 4, I compare the *Personal Picture Finder* with related services.
Chapter 5 summarizes the results, and gives a visionary outlook on features which could be added in the future.
Due to the object oriented structure of the Java programming language, the *Personal Picture Finder* contains a lot of reusable code. Some of the most useful classes are presented in Appendix A.
Appendix B contains a user manual explaining how to use the *Personal Picture Finder*, answering frequently asked questions and giving tips on how to fix occurring problems.
Some general remarks about this document:

1. By using the words *he* and *his* when talking about the user, I refer to male and female users.

2. Throughout this document, I use the name of *Alan Turing* as example for looking up pictures on the internet. Of course the *Personal Picture Finder* can be used for looking up any person on the internet, and is neither restricted to special names nor to famous persons.

Chapter 2

State of the Art

This Chapter is about the state of the art in giving the user assistance while browsing the web. The *Personal Picture Finder* is a personal internet assistant based on the search infrastructure provided by search engines. The use of intelligent technologies makes it a helpful tool for looking up specific information on behalf of the user. Technologies discussed here are search engines, internet agents and intelligent user interfaces. The distinction of the sections "search engines" and "internet assistants" indicates two different levels of the quality of service, namely those using intelligent technologies and those simply looking up information. The line between those two kinds of services is not drawn strict, boundaries blur and there is no unique definition. Some search engines could as well be described as *index search agents* and therefore fit in both categories. The objective of this Chapter is providing a survey on current technologies. There is no claim at all for a final or complete classification.

2.1 Search engines

As soon as the World Wide Web started growing over the level of a few pages of information, some guidance for the user became inevitable. In 1994, Jerry Yang and David Filo published their collection of bookmarks entitled *Jerry's Guide To The World Wide Web*. This page was first used by some of their friends only, but til the end of the year reached 10000 daily visitors. In the same year, the first web catalogue *Yahoo!* was published. It simply provided a survey of categories that lead to more specific categories, where the user could simply specify with a few mouse clicks what he is looking for. For example, looking for information about the soccer world championship, the user chooses the category sports, then soccer, then championship and found the desired information. This technology was sufficient at this time, but could not catch up with the staggering growth of information provided.
Search engines tried to automate the indexing process. By using *spider* and *crawler*, a huge amount of webpages can be retrieved, classified by keywords and stored in a database. The user can query this database using a web interface and retrieve the URLs of pages classified with these keywords. A lot of search engines were developed using this technology, and it was a very common opinion, that the information provided basically was the

same. In 1996, Selberg and Etzioni [SE96] proved the contrary. Search engines differ in many ways. Some rate the important keywords by position in the document, some other evaluate the keywords by the frequency of appearance and some others simply check the *meta*-tags at the beginning of a document. Using six search engines in parallel results in average 3.5 times more URLs than using a single one.
The idea of a *metasearch engine* is to provide access to many search engines and collect the results. The advantage for the user is a uniform interface for placing his query and a lot more of information returned. Furthermore *metasearch engines* like the Metacrawler[1] virtually add features to search engines like search for phrases. If a search engine does not provide this feature, Metacrawler simply requests it with the phrase as keywords and then checks the results it gets in return for the requested phrase. If the keywords do not appear in the desired order, it will not present this URL to the user. Further post processing on the references like checking if the pages referred to really exist, eliminating duplicates, and sorting the references is another service provided by *metasearch engines*. Despite all these sophisticated services, the user still has to check out manually all the references he get in order to find the specific information he is interested in. One approach for facilitating this task are intelligent internet agents.

2.2 A survey on internet agents

In this section I show which requirements an *internet agent* should meet. After that I give a survey on the different kinds of internet agents.
The following characteristics are desirable agent qualities[EW95]:

- **Autonomous**: an agent is able to take initiative and exercise a non-trivial degree of control over its own actions:
  - **Goal-oriented**: an agent accepts high-level requests indicating what a human wants and is responsible for deciding how and where to satisfy the requests.
  - **Collaborative**: an agent does not blindly obey commands, but has the ability to modify requests, ask clarification questions, or even refuse to satisfy certain requests.
  - **Flexible**: the agents actions are not *scripted*; it is able to dynamically choose which actions to invoke, and it what sequence, in response to the state of its external environment.
- **Communicative**: the agent is able to engage in complex communication with other agents, including people, in order to obtain information or enlist their help in accomplishing its goals.
- **Adaptive**: the agent automatically customizes itself to the preferences of its user based on previous experience. The agent automatically adapts to changes in its environment.

[1] www.metacrawler.com

- **Mobile**: an agent is able to transport itself from one machine to another and across different system architectures and platforms.

Usually, internet agents do not meet all the requirements mentioned above. Mobility, in particular can hardly be achieved due to security restrictions.
The *Personal Picture Finder* is autonomous and communicative. It performs his task only based on the name provided by the user without asking for further instructions but instead autonomously starts and performs its search. In order to achieve its goals, the *Personal Picture Finder* communicates with other information infrastructures available on the web like search engines and image databases.
The most famous agents in the WWW are *index search agents* like Lycos[2], WebCrawler[3] and InfoSeek[4]. Those agents autonomously browse the WWW and store an index of words from titles and content of documents. By querying such an agent, the request is used as a keyword to look up webpages containing this term in a database.
These kinds of agents have a lot of restrictions:

- The WWW cannot be indexed completely. Only a part of the documents is covered.
- Dynamic generated information is not considered. Documents generated by cgi-scripts are dismissed.
- The query is restricted to words instead of concepts. If one is looking up "Pan American Freeway", one might miss all pages about "Route 66".

Other members of the family of internet agents are *presentation agent*, *shopbots*, *information-gathering agents* and *search agents*:

- Presentation agents generate online presentations adapted to the user profile. Furthermore, they guide the user interactively through the presentation [ARM98].
- Shopbots are virtual shopping assistants and represent the narrow combination of information gathering and electronic commerce.
- Information-gathering agents facilitate the former toilsome way of looking up information on the internet. The user roughly tells the agent about the information he needs and the agent determines and executes the necessary actions to eventually present it in the desired way. The first generation of shopping agents does not offer any interaction with the user, but instead is restricted to comparing the price of a product from different vendors and show the result in the appropriate way. Examples for this kind of shopbots are Jango[5] and the Yahoo-Visa-Shopping-Guide[6].
  The shopbots of the next generation should be personal internet assistants, they should know all relevant individual information about the user and be able to perform a whole deal autonomously.

[2] *www.lycos.com*
[3] *www.webcrawler.com*
[4] *www.infoseek.com*
[5] *www.jango.com*
[6] *shopguide.yahoo.com*

2.3 Intelligent User Interfaces

Modern interface technology has advanced from initial command line interfaces to the esablished use of direct manipulation or WIMP (windows, icons, menu and pointing) interfaces in nearly all applications. Additional benefits to the user will be provided in the next generation of interfaces, often called "intelligent interfaces".
Maybury and Wahlster define:

> "**Intelligent user interfaces** are human-machine interfaces that aim to improve the efficiency, effectiveness, and naturalnes of human-machine interaction by representing, reasoning, and acting on models of the user, domain, task, discourse, and media [MW98]."

The development of *intelligent user interfaces* is an interdisciplinary area. It includes human-computer interaction (HCI), ergonomics, cognitive science, and artificial intelligence.
Contemporary research in this field mainly includes:

- Multimedia Input Analysis
- Multimedia Presentation Design
- Automated Graphics Design
- Automated Layout
- User and Discourse Modeling [KW89]
- Model-Based Interfaces
- Agent Interfaces
- Evaluation

The development of the *Personal Picture Finder* took place in the Intelligent User Interfaces research department of DFKI and is an examplary use of the research in *Agent Interfaces*. An in-depth look at the state of the art in developing intelligent user interfaces can be found in [MW98].

Chapter 3

Underlying Technology

The *Personal Picture Finder* relies on basic technologies as well as on experimental work or current research. In this Chapter, the underlying technology is discussed:

- The *Bitmap Information Tool BIT* is a current research topic of project AIA. It was used to add some more information about the pictures in the experimental version of the *Personal Picture Finder*.

- *HyQL - the Hypertext Query Language* is a main research interest of project PAN. It provides mechanisms for information extraction from HTML documents. For the public version of the *Personal Picture Finder*, the prototype of the HyQL interpreter is not used yet.

- *Java* is the programming environment used for the implementation of the *Personal Picture Finder*.

- *Graphics File Formats* are the kind of data the *Personal Picture Finder* is evaluating.

- A *database* provides the infrastructure for storing collected data.

- The field of *Machine Learning* has some useful approaches for automated filter optimization.

- *Persona* is a presentation agent developed in projects PPP and AIA and was used in an early version of the *Personal Picture Finder* as presentation host.

3.1 BIT - Bitmap Information Tool

The Bitmap Information Tool (BIT) [Haa99] is a powerful tool for analyzing graphics files. It is currently under development at DFKI GmbH by Jens Haase.
Unfortunately analyzing pictures is a rather time consuming operation. One of the main purposes of the publicly accessible release of the *Personal Picture Finder* on the other hand was looking up pictures on the web fast, so a detailed analyze cannot be used.

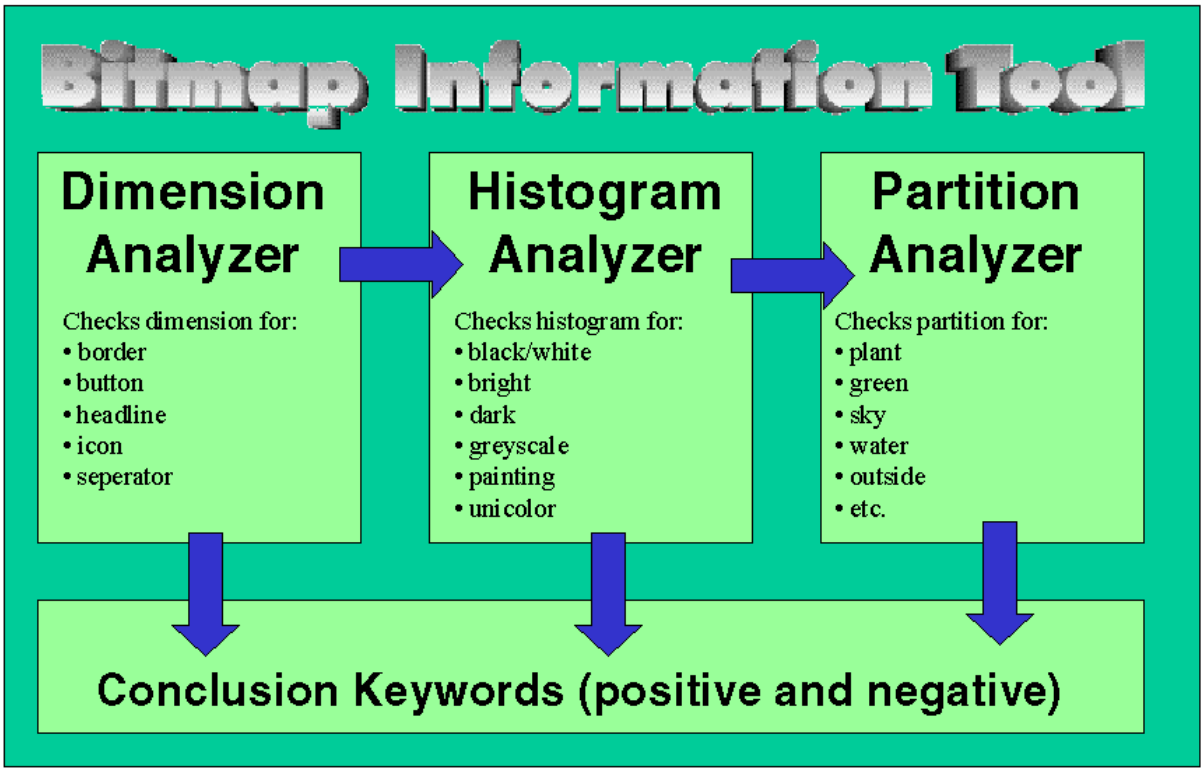

Figure 3.1: Bitmap Information Tool BIT

As I will show later on, combining the *Personal Picture Finder* and *BIT* is a promising approach for providing better results to more patient users.
BIT is implemented in C++ and based on the libraries of *Image Magick*. It provides the user with a highly configurable interface as well as with many options for the command line version.
Given one or several graphics files, *BIT* reads the data and performs up to three analyzing levels:

1. *Dimension / Metadata:*
   The dimension information includes besides parameters like horizontal and vertical dimension and resolution and color depth some interesting features like:

   - Detection of animation
   - Classification of the picture in categories like button, headline, icon, separator or border. The classification includes positive and negative information (what the picture is and what the picture is not respectively).

2. *Histogram:*
   The histogram counts the colors of a picture and performs statistic testing on lightness, saturation, and the amount of colors used compared with the maximal possible colors in the specific graphics file format.
   The histogram information requires a more detailed analysis and may take up to

two minutes depending on picture size and computational power.
Keyword classification on this level includes positive and negative detection of:

- Blackwhite:
  The picture has only two colors: black and white.
- Bright:
  The amount of bright pixel in the picture exceeds by far the amount of dark pixel.
- Dark:
  The amount of dark pixel in the picture exceeds by far the amount of bright pixel.
- Grayscale:
  The picture has only gray colored pixel.
- Painting:
  The picture uses only a few of the possible colors.
- Unicolor:
  The majority of pixel in the picture are unicolored pixel.

3. *Partition:*
   This final test divides the picture in squares and counts the colors in them. A statistical evaluation of specific regions of the picture based on empirical testing provides a high level keyword classification. The result of this filter states if the picture is or is not of the following kind:

   - Plant
   - Green
   - Sky
   - Water
   - Outside
   - Inside
   - Room
   - Snow
   - Beach
   - Sunset

   If the classification is ambiguous, *undefined* is returned as result.

As an output, the *Bitmap Information Tool* generates a summary of the conclusions drawn on each level of analysis.
Other features will be added on soon and described in detail in the original specification.

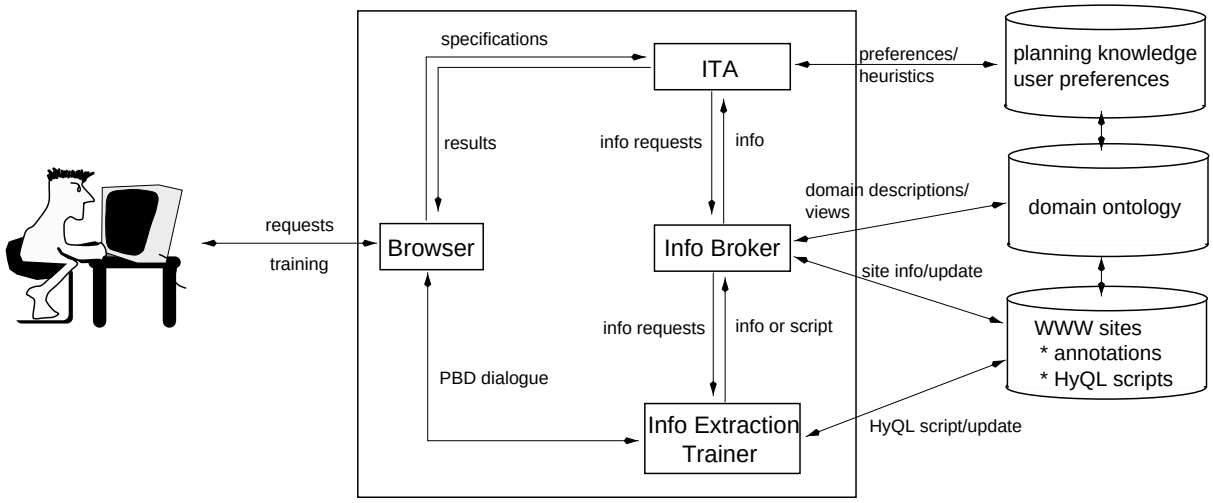

Figure 3.2: TriAs architecture developed in PAN

3.2 HyQL - A Hypertext Query Language

The World Wide Web contains a huge amount of data easily accessible for everybody. Almost any information can be found. Due to the lack of structure this turned out to be a double-edged sword. Finding a specific information without any assistance at all is next to impossible. Access-mechanisms as well as a language to specify higher level requests than just simple keyword search become inevitable. An SQL-style query language seems to be the appropriate way. There have been several approaches concerned with WWW query languages.
In 1998 Bauer and Dengler presented TrIAs, an approach for cooperative problem solving using **Tr**ainable **I**nformation **As**sistants [BD98] as a research result of project *PAN*. A robust prototype interpreter evolved out of this theoretical work and now is successfully used in several applications. Since most search engines frequently change the layout of their output, the architecture described in this section seems to be a useful extension for the *Personal Picture Finder*.
In the following, after shortly introducing project PAN and its goals I give a survey of the TrIAs architecture and the underlying technology, namely the **P**rogramming **b**y **D**emonstration paradigm (PbD) and the **Hy**pertext **Q**uery **L**anguage [Den99b] HyQL. A much more detailed introduction can be found in [BD99b].

The name of project PAN stands for **P**lanning **A**ssistants for the **N**et. It is part of the IUI[1] research department of the DFKI GmbH. In the tradition of the ancestor projects *PHI*[2] and *RAP*[3] research about planning is a main objective. A new aspect is the usage of the WWW as domain. The goal of project PAN is the development of plan-based information assistants for the internet. Main purposes are generation and execution of plans

[1]IUI is an acronym for **I**ntelligent **U**ser **I**nterfaces.
[2]PHI is an acronym for **P**lanbasierte **Hi**lfesysteme (plan-based assistance systems).
[3]RAP is an acronym for **R**easoning **A**bout **P**lans.

in dynamic environments, handling of the underlying domain model and the implementation of a cooperative and adaptive behaviour of the assistant. The latest development of PAN is a tool for constructing and developing information assistants for personal use. Furthermore these assistants can be trained for the extraction of relevant information (*InfoBean-concept*; [BD99a]).

One crucial point is the simplified adaptation of these assistants to changing structures

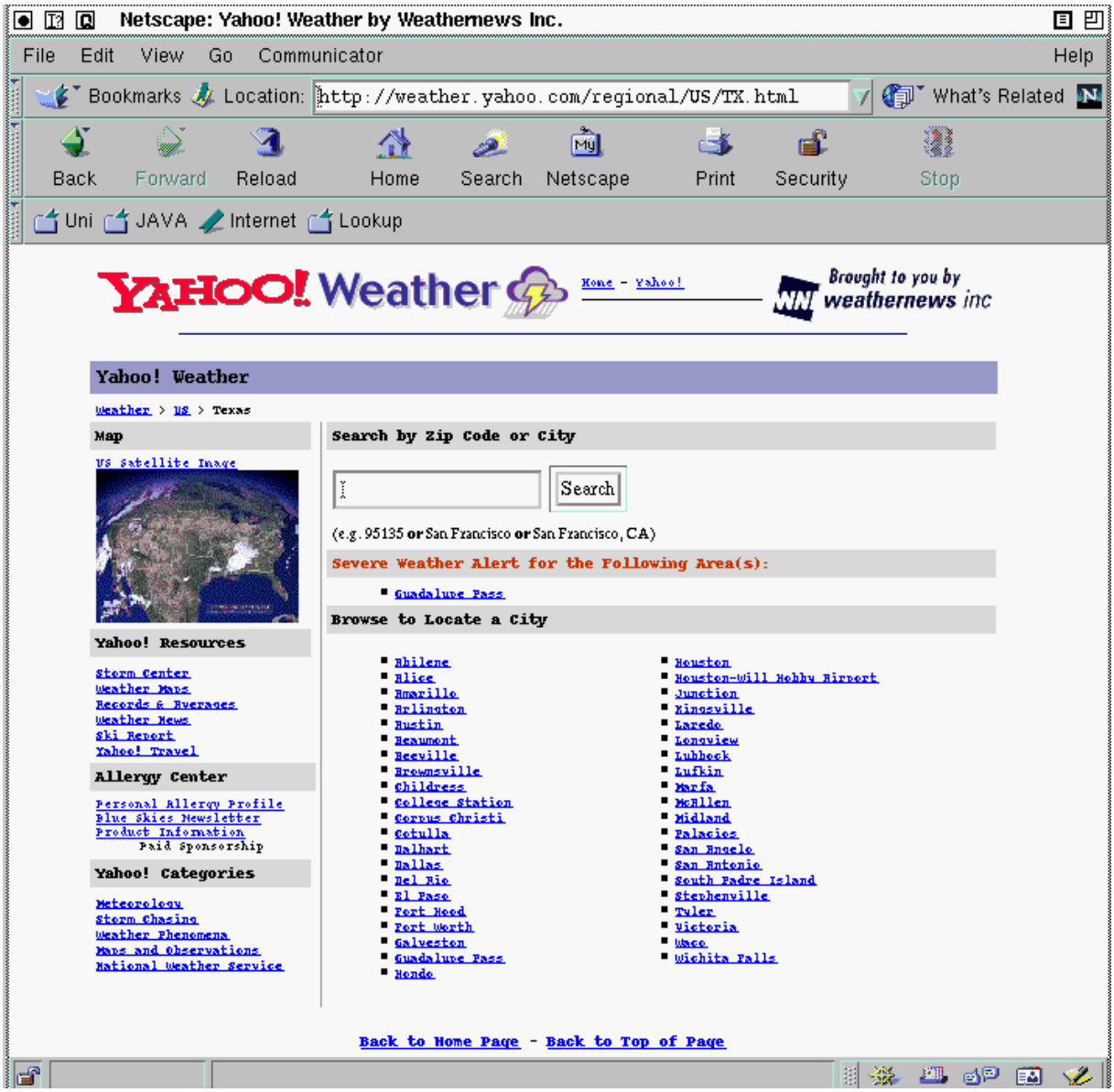

Figure 3.3: Regional weather forecast page

and new domains. This can be achieved using the TrIAs technology described below. Research in PAN focuses on:

- planning and plan execution in dynamic environments

- systematic domain model development
- user modeling for Net applications
- learning interface agents

The *Personal Picture Finder* is an exemplary prototype of an interface agent.
Software agents are intended to autonomously perform certain tasks on behalf of their users. In a highly dynamic domain like the Net this can hardly be achieved, since the agent's competence might not be sufficient to produce the desired outcome. Instead of just giving up and leaving the whole task to the user, the *Trainable Information Assistants (TrIAs)* approach identifies the problems of the autonomous agent and tries to improve its capabilities in dialog with the user. The user here is expected to be able and willing to help, since he is interested in obtaining a useful response from the system, even at the cost of having to intervene from time to time.
In the following I describe the TrIAs architecture using the example scenario ITA[4]. The architecture of this example is shown in Figure 3.2. Its three core components are the *application module* (here: the trip planner ITA), the *Information Broker* and the *Information Extraction Trainer* (IET).
Whenever the trip planner has an information gap that cannot be filled using current domain knowledge it requests the Information Broker for appropriate information. The Information Broker maintains a database containing specifications of the information sources to be used along with operational descriptions how to extract information from these sources. These descriptions are represented by HyQL scripts. Executing these scripts, the Information Broker obtains and forwards the information requested by the application module.
The Information Broker's search for information however can fail from time to time due to one of the following problems occurring:

- The site containing the information is not available.
- The structure of the document changed and therefore cannot be understood by the HyQL script.

The first case is no real challenge. The Information Broker can handle this situation by simply accessing the desired information from an alternative source. The later case is much more interesting. The script producing the failure is passed to the Information Extraction Trainer that starts a Programming by Demonstration (PbD) dialog with the user.
The objective of the PbD dialog is the generation of a working script for a modified document or at least the identification of the relevant information for the application module. The former enables the system to deal successfully with this website and the later avoids a failure of the current process. In the TrIAs architecture described above specification of information sources including operational extracting instructions are stored in scripts implemented in the **Hy**pertext **Q**very **L**anguage HyQL. In the HyQL approach, the WWW

[4]ITA is an acronym for **I**nternet **T**ravel Arrangement **A**ssistant.

is considered to be a computable dynamic graph structure. The nodes can be static documents or dynamic generated ones, like database queries or the output of a CGI script or application. Links are the edges in this graph.
The documents are represented as tree structures based on the opening and closing HTML-tags.
HyQL and PbD do not intend the construction of a completely autonomous agent (which can hardly be achieved on the WWW), but it can help the user a lot to deal with the dynamically changing environment and will—by interaction—increase its functionality.
Features of the HyQL language are:

- detailed specification of WWW navigation and programmed search,
- detailed and flexible access to document structure and content,
- flexible referencing and selection scheme to reach robustness of queries against layout changes of documents,
- specification and use of user-defined abstractions (macros),
- homogenous language definition fulfilling the needs of naive as well as expert users.

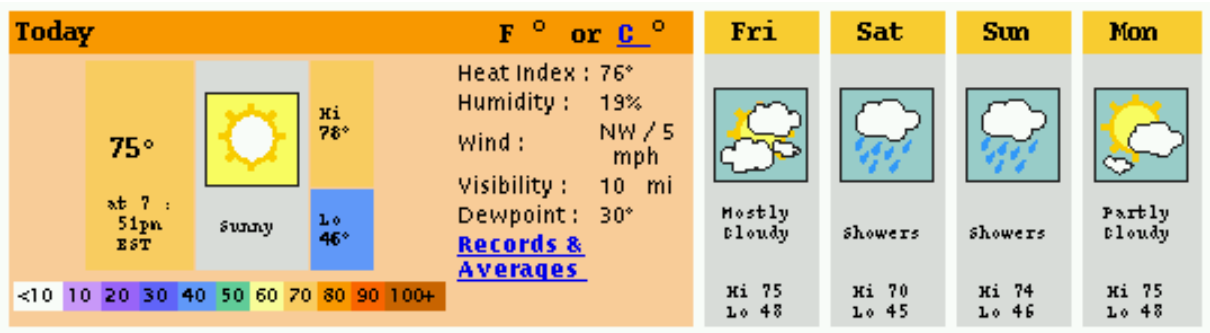

Figure 3.4: Result returned from HyQL sample script

The following example gives an impression of HyQL and its possibilities.
Example:

```
select valid_html(root,descendant(1,body)(4,table)) from
  select content from document d1 such that
    document d in http://weather.yahoo.com/regional/US/TX.html
    document d -> document d1
    d1.url = select root,descendant(1,a)href from
    { select info i1 := root,descendant(all,a)
      from document d
      where i1 match "El Paso" }
```

The objective of this script is to obtain the weather forecast for a city in Texas by following a link from a regional weather forecast survey (see Figure 3.3.).
The script selects *valid html* from a specified position of a document specified later on. *Valid html* means, that the piece of html code extracted will be "fixed" in terms of adding missing closing tags in order to obtain a tree structure. Besides that, a full html page is created out of it by adding the document's original head tag, and by setting the document base to the page's original source[5].
The position in the document is described as *(root,descendant(1,body)(4,table))*, meaning from the document root the first subtree labeled *body* is selected and there the fourth subtree labeled *table*. Or short: The fourth table in the document body.
The desription of the document to obtain this information from is more difficult and shows some of the possibilities of HyQL.
The second *select* statement says that the desired document is document *d1*, and it can be obtained by downloading another document *d* given by an URL. The constraints to be solved then are:

- There is a local link from document *d* to document *d1*. Local links are denoted by $->$, global links by $=>$. Link chains or alternative paths can be denoted as well in HyQL [Den99a].

- The URL of document d1 can be obtained from a *href* in the first *a*-tag that matches the keyword *El Paso*.

The output produced by the sample script is shown in Figure 3.4. Some more sample scripts for obtaining information from search engines will be shown later.

3.3 Java

In this section I describe Java, the programming environment used for implementing the *Personal Picture Finder*. It provides the full functionality of a modern, object-oriented programming language like C++, but is reduced to a simple and easy to handle structure. It is platform independent and provides the user with a lot of useful, predefined libraries. Java was developed at Sun Microsystems in 1991. It is described as "simple, object-oriented, statically typed, compiled, architecture neutral, multi-threaded, garbage collected, robust, secure, and extensible." [GJS96]:

- **Simple.** Java's developers deliberately left out features like implicit type casting, operator overloading, header files, or multiple inheritance.

- **Object-oriented.** Just like C++, Java uses classes to organize code into logical modules. At runtime, a program creates objects from the classes. Java classes can inherit from other classes, but multiple inheritance is not allowed.

[5]By setting the code base the browser is able to include pictures and links from the original document.

- **Statically typed.** All objects used in a program must be declared before they are used.
- **Compiled.** A Java program has to be compiled to a so called *byte-code* before running it. This byte-code can be interpreted on any platform using a Java Virtual Machine that translates the byte code into machine language commands.
- **Multi-threaded.** Java programs can contain multiple threads of execution, which enables programs to handle several tasks concurrently. The *Personal Picture Finder* takes advantage of this feature by downloading several webpages in parallel and at the same time analyzing graphics data in parallel too.
- **Garbage collected.** Java programs do their own garbage collection.
- **Robust.** The interpreter checks all system access. When an error is discovered, the program throws an exception that can be captured and managed by the program. It is not possible to crash any serious operating system with a Java program.
- **Secure.** Since Java does not support pointers, it is not possible to access any part of a system without authorization.
- **Extensible.** Java programs support native methods, which are functions written in another language, usually C++.
- **Well-understood.** The Java language is based upon technology that's been developed over many years.

The focus in the following is on features of Java used for the development of the *Personal Picture Finder*.

3.3.1 Applets - the java.applet package

Applets are probably the most famous Java programs. An applet is a Java program running on the Virtual Machine (VM) of a web browser. When downloading a page containing an applet, the browser also downloads the applet code and executes it on the local computer. Applets are used for creating special effects like animation or little games on a webpage. Applets do not have the full functionality of Java programs. A *SecurityManager*-object controls the execution. Depending on the browser, the user has several possibilities to change the SecurityManager's restrictions. Typically the main restrictions for a program executed in a browser are:

- No access to the local file system is allowed.
- Network connections are restricted to connecting the server providing the applet.

It is possible to sign an applet so that the user can verify who wrote it and that it was not modified at download time. The user can decide which developers he trusts in and give their applets more rights.
In the *Personal Picture Finder* an applet is used for connecting back to the server, displaying information and building a virtual webpage.

	Application	Applet	signed Applet	Servlet
stand alone	yes	no	no	no
context	none	Browser	Browser	Server
GUI	possible	yes	yes	no
Networking	yes	restricted	yes	yes
file access	yes	no	yes	yes
Computer	server	client	client	server

Table 3.1: Features of Java programs.

3.3.2 Servlets - the javax.http package

A servlet is a server side application using the JVM[6] of a webserver. It can be accessed via a *http request* and return some output like a CGI[7]-script. Unlike an applet running on client side, the servlet has no default security restrictions. It can access the local file system, connect to every location on the web, execute programs and access databases. Using the SecurityManger, the user could of course add restrictions. The servlet is running in an environment controlled by the developer, who can always install the latest JDK[8] release. This is obviously not possible for an applet running on client side.
Servlets are used in several parts of the *Personal Picture Finder* architecture. Details will be discussed later.

3.3.3 JDBC - the java.sql package

The *java.sql* package in the Java API is usually referred to as the JDBC API[9]. It was developed as a separate package from JDK 1.02 and is an integral part of JDK release 1.1.
The objective of JDBC is providing database connectivity to Java programs. That does not sound too spectacular, but in connection with other Java features provides a powerful tool:

- Database access can be provided over the net.
- Tools accessing and manipulating a database can be implemented platform independent.
- Webpages can be created on the fly using information from a database.
- When replacing a database with another, one simply needs to load another driver in the Java program but not reimplement the whole software.

In the *Personal Picture Finder* JDBC is used to store information about graphics data found on the net.

[6]JVM is an acronym of Java Virtual Machine.
[7]CGI stands for Common Gateway Interface.
[8]JDK is the Java Developers Kit.
[9]JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity [HCF97].

3.3.4 Networking - the java.net package

When implementing an internet agent it is crucial to have a powerful programming language in terms of which networking capabilities it offers. The *java.net* package provides a powerful infrastructure for networking [Fla97]. It provides two different kinds of interprocess communication, the simple datagram socket and the more complex stream socket. The stream socket (or connected socket) is a socket over which data can be transmitted continuously[10]. Continuous activity distinguishes the stream socket from the datagram socket that is used for one-time communication. Stream sockets can be used for TCP/IP connections. Java provides streamed socket programming primarily through two classes: Socket and ServerSocket. The difference between these classes is that the later can be used for implementing a Server while the former is suitable for clients only.
On a higher level, Java provides the classes URL, URLConnection and HttpURLConnection representing the *Uniform Resource Locator*, the connection to a *Uniform Resource Locator* and the Hypertext Transfer Protocol respectively. Using these classes makes it easy to access and manipulate webpages.
The *Personal Picture Finder* uses the HTTP protocol for accessing webpages and for the initial communication between applet and servlet. The communication between applet and a stand alone application controlling the applet is established via TCP/IP over stream sockets.

3.4 Graphics File Formats

> "File formats can be complex. Of course they never seem complex until you're actually trying to implement one in software." (J. D. Murray, [Mv96])

In order to decide, which pictures are to be presented to the user, the *Personal Picture Finder* has to analyze the pictures it found. This section is about the files pictures are stored in, about *graphics file formats* [KL95].

3.4.1 General remarks

This section provides some general remarks about graphics file formats, the way graphics are stored in files, why there are so many different formats and how to choose the appropriate format for an application. Later on, the graphics file formats used on the web are discussed.

3.4.1.1 Basics

A *graphics file format* (GFF) is the format in which data describing an image is stored in a file. graphics file format have come about from the need to store, organize, and retrieve graphics data in an efficient and logical way ([Mv96]). Depending on the application using

[10]Continuously does not necessarily mean that data are send all the time but that the socket itself is active and ready for communication all the time.

an graphics file format this "efficient and logical way" can vary significantly. Choosing the right graphics file format for an application raises a lot of questions:

- Should the way of storing the graphics data be efficient in terms of disk space or download time? Or should it rather be fast and easy to be displayed?
- Is scalabilty necessary?
- How many colors are needed?
- Is the graphics file format used for displaying a picture on the screen or for printing it out?
- Are special effects like interlacing, animation or transparency necessary?
- Is compatibility to other applications necessary?
- How much quality is needed?
- Or, subsuming all the above questions: What do I need the graphics file format for?

A wide variety of graphics file formats evolved out of these questions over the last decades. After shortly explaining the traditional main classification of graphics data, I describe the graphics data one has to deal with on the internet, namely the JPEG and GIF formats and the PNG format which is anticipated to become the future standard format for graphics on the internet.

3.4.1.2 On Vector formats versus Bitmap formats

Traditionally, graphics file formats are divided into vector formats and bitmap formats. A vector format is a format containing vector data. Vector data refers to what one usually associates with the mathematical or scientific term vector: the data gets stored as a description of lines, polygons or curves along with additional information like thickness and color of lines. Vector data are very convenient if one needs to scale graphics or print them on a plotter. Using vector data is not very useful if one wants to rapidly display a graphic on the screen, since rendering vector data is a time consuming operation. Bitmap data, on the other hand, suits this purpose perfectly. The term "bitmap" is quite confusing in this context. In older usage, it really referred to an *array of bits* representing a monochrome picture. Its meaning has changed over the past years and the usage of this term now includes, besides arrays of bits, also arrays of pixels or integers (to represent a color from a given color palette), and even compressed formats that can be uncompressed to obtain an array of pixels.

3.4.2 Graphics File Formats on the WWW

By looking at the classification of graphics file formats it is obvious that compressed bitmap formats perfectly meet the requirements of the WWW, which are: fast download

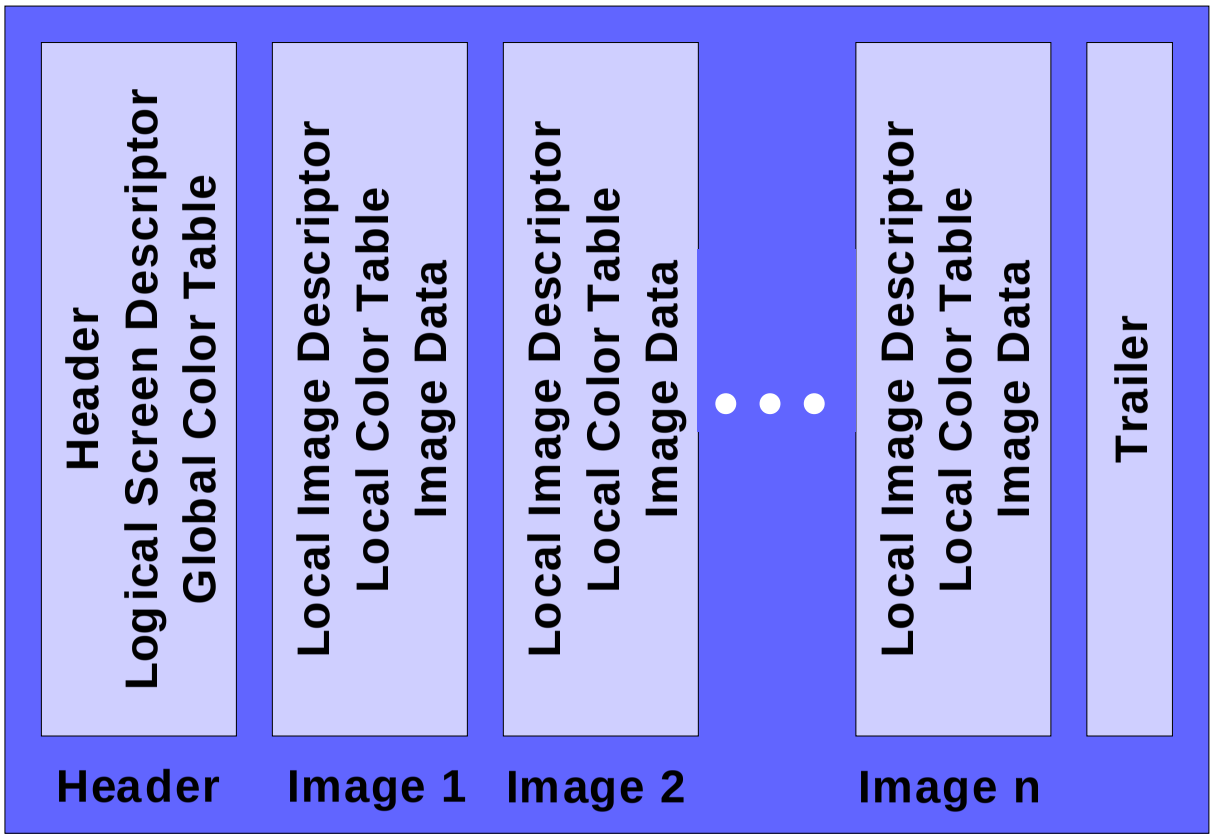

Figure 3.5: Structure of a GIF87a

(provided by small files due to compression) and fast and unscaled displaying. The commonly graphics file format used on the WWW[11] are GIF, JPEG and PNG as a standard to be[12]. In this section I describe these graphics file format. A Java application analyzing these formats can be found in Appendix A.2.4. A description of all those formats can be found in [Mv96] and [Bor97]. Further information about specific formats will be quoted in the subsections below. It turned out that the best specification for the formats JPG and PNG were the libraries libjpg.h and libpng.h respectively.

3.4.2.1 Graphic Interchange Format GIF 87a

There are two revisions of the Graphic Interchange Format *GIF*, which became very popular and widely distributed. The original revision was *GIF87a*, named after the year in which it was released by CompuServe Inc. Two years later, in 1989, the current revision with enhanced capabilities was released. In this subsection I present *GIF87a* as

[11]This refers to the graphics file format included in a webpage using the IMG tag. One surely can put a link to any file format, including a graphics file format, on a webpage and provide other formats for

Format	GIF 87a	GIF 89a	JPEG	PNG
Type	Bitmap	Bitmap	Bitmap	Bitmap
Compression	LZW	LZW	JPEG	LZ77 variant
Colors	1 to 8 bit	1 to 8 bit	Up to 24 bit	1 to 48 bit
Maximum size[13]	64K x 64K	64K x 64K	64K x 64K	2G x 2G
Multiple Images[14]	no	yes	no	no
Num. Format	Little-endian	Little-endian	Big-endian	Big-endian
Originator	CompuServe	CompuServe	C-Cube	Boutell et. al.

Table 3.2: Graphics File Formats on the WWW

described in [Com87]. The GIF-format uses the LZW (Lempel-Ziv-Welch) compression algorithm. This algorithm is based on the commonly used algorithms LZ77 and LZ78[15]. LZW is a general compression algorithm capable of working on almost any type of data. It is fast in both compressing and decompressing data and therefore suits the needs of an efficient storing of graphics data.
Unfortunately the LZW algorithm is not freely available, meaning that every developer using LZW for compressing or decompressing data has to obtain a license from CompuServe and pay a royalty on each copy of their product sold.
The layout of a GIF is (roughly) a header followed by a logical screen descriptor and a global color table. The image date and a trailer follow that (see Figure 3.5).

3.4.2.2 Graphic Interchange Format GIF 89a

This current version of GIF is similar to the 87a revision, but it contains several additional blocks of information (see Figure 3.6). These additional information blocks are used for four so-called *Control Extensions*, which are:

- *Graphics Control Extension:* The most popular feature from this extension is the transparency-option.
- *Plain Text Extension:* This extension allows human-readable text which is actually part of the bitmap itself.
- *Comment Extension:* Also used for additional human-readable text. Unlike the Plain Text Extension, this text is embedded in the data-stream.
- *Application Extension:* Additional information is stored here as well, in order to help the displaying application to properly and fast display the picture.

The format is specified in detail in [Com89].

download or write plugins to display them.

[12]Some browsers are able to display XBM and XPM as well, but these files are really rare on the WWW and most likely will become extinct.

[13]Image size in pixels.

[14]Multiple Images per file possible.

[15]The compression tools compress, zoo, lha, zip, and arj for example use the LZ77 algorithm.

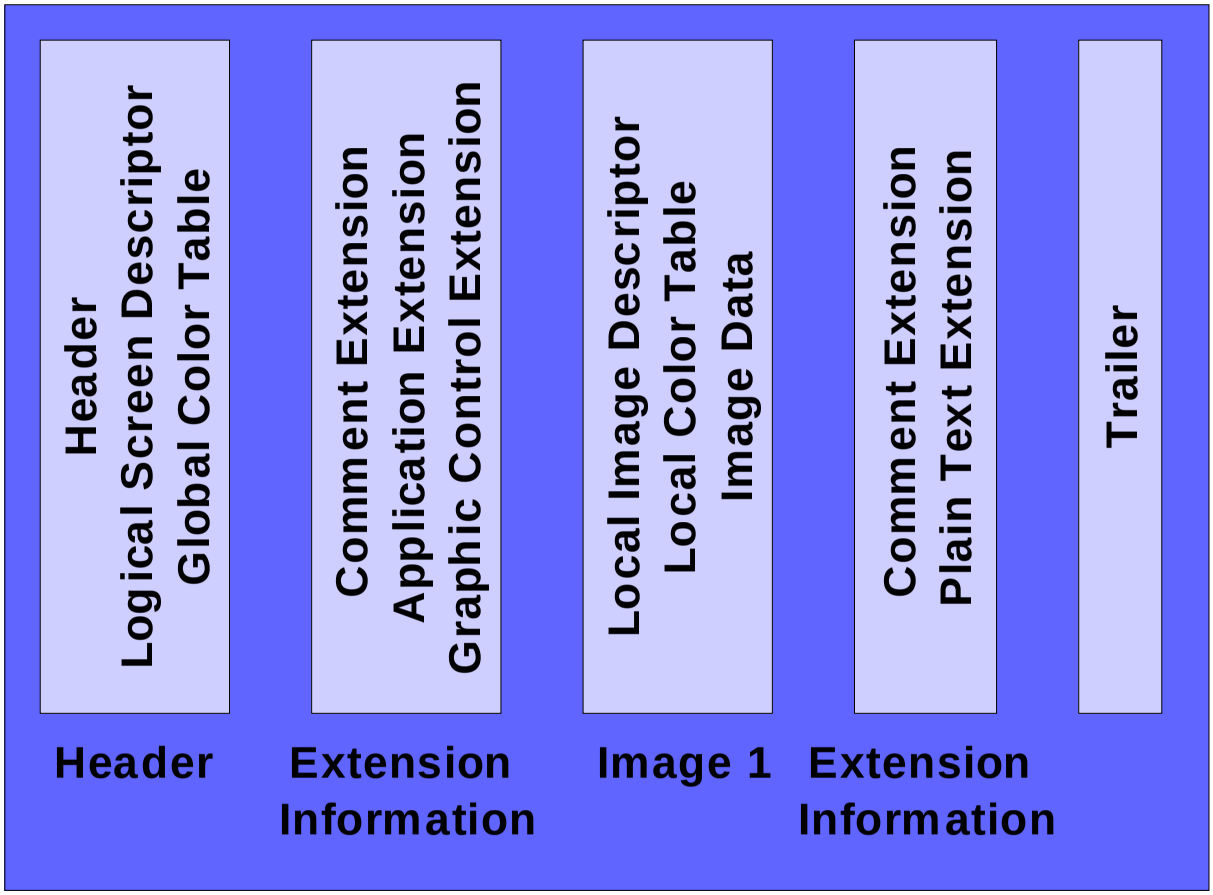

Figure 3.6: Structure of a GIF89a

3.4.2.3 JPEG File Interchange Format

JPEG File Interchange Format is a graphics file format based on JPEG compression. The usage of the acronym JPEG[16] is a bit confusing, since it refers to a standards organization, a method of file compression and sometimes to the graphics file format itself too.
The current revision of JPEG is 1.02, released by C-Cube Microsystems in 1992. Files are stored here as a stream of blocks, each starting with a specific marker to identify its content. It does not posses a formally defined header, but it always starts with the blocks SOI[17] and APP0[18] which serve as a de-facto-header. Unfortunately these two blocks do not contain information about the dimensions of the image. This information can be *anywhere* in the file. Since the public version of the *Personal Picture Finder* only downloads the first 400 bytes of a file, the chance of missing this important information is about 40 percent. Newer graphic tools, usually store this information block at the beginning of the file, but there is still a lot of old images available on the internet.

[16]JPEG is an acronym for **J**oint **P**hotographic **E**xperts **G**roup.
[17]SOI is an acronym of **S**tart **O**f **I**mage.
[18]APP0 is the Application Marker Segment.

3.4.2.4 Portable Network Graphics PNG

The intention of the implementation of PNG (pronounced "ping") was offering an alternative to CompuServe's GIF[19]. The main differences to the GIF format are enlarged width, height and depth parameters of the picture as well as the usage of the LZ77 compression algorithm which is available at no charge.

3.5 The Image Data Base

This section is about the image database. The *Personal Picture Finder* uses an Oracle database for storing information about analyzed pictures and the feedback given by the user.
A *database* is a *collection of information.* A simple computer file is a collection of information as well. But there are some fundamental differences that make the use of a database especially interesting:

- A database comprises not only data but a plan, or *model* of the data.
- A database can be a common resource, used concurrently by many people.

The principal difference between information collected in a database and information in a file is the way the data is organized. While in a file the information is organized physically (i.e. certain items precede or follow other items), the contents of a database are organized according to a *data model.* A data model is a plan, or a map, that defines the units of data and specifies how each unit is related to the others.
The data model is designed when the database is created. The units of data are inserted according to the plan specified in the model. Sometimes the data model is referred to as *schema* as well.
Modern databases are usually *relational databases*, meaning that they are organized according to the *relational calculus* by E.F.Codd. In this calculus all data is presented in tables comprising rows and columns.
The most common language for accessing and querying a relational database is *SQL*[20]. SQL and the relational model were invented at IBM in the 1970s. The ANSI[21]-SQL1 standard was defined in in 1986 using a core set of SQL features. Beside this core standard most databases today use slightly differing features.
The JDBC API described in [HCF97] provides a powerful tool for accessing databases using Java programs. A more detailed introduction to databases can be found in [O'N94].

[19]The unofficial recursive derivation of the name "PNG" is "**P**NG's **N**ot **G**IF".
[20]SQL is an acronym for **S**tructured **Q**uery **L**anguage.
[21]ANSI is an acronym of **A**merican **N**ational **S**tandard **I**nstitute.

3.6 Machine Learning

In a highly dynamic environment like the internet it is desirable for a personal assistant to be as independent as possible. The netbot should learn from experience instead of being instructed over and over again by the user or developer. One approach to reduce human intervention at the agents work is *learning* [Mit97]:

Definition: A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Applied on a checkers learning problem one could define:

- Task T: playing checkers
- Performance measure P: percent of games won against opponents
- Training experience E: playing practice games against itself

Machine learning was influenced by a lot of scientific disciplines, like artificial intelligence, psychology, statistics, philosophy and information theory. It was successfully used over the last few decades in many applications, like speech recognition, autonomous drivers for vehicles, recognition of hand writing, playing backgammon at world champion level, etc. An introduction to the field of learning computer systems can be found in [WK91].
In the *Personal Picture Finder* application, machine learning was used for experiments on automated filter optimization. Details will be discussed later on in the next chapter.

3.7 A Life-Like Presentation Agent: Persona

One important approach when building intelligent user interfaces, is the usage of life-like characters as presentation agents. At DFKI, the character *Persona* [Mül99] was developed in the projects PPP[22] and AIA[23]. Its objective is to lead the user through a multimedia presentation.
Persona is integrated in a complex presentation system, which generates user-specific adapted presentations. The intention is to give the user a contact person he can consult when he needs help.
The required features of a presentation agent in a multimedia presentation are:

- *hosting the presentation:* Like a human host, the agent leads his "guest" through the presentation. He explains, gives hints, points out important parts, etc. All these actions have to be scheduled and coordinated. For this purpose, it is of advance for the agent to have an anthropomorphic appearance.

[22] PPP is an acronym for **P**ersonalized **P**lan-based **P**resenter.
[23] AIA is an acronym for **A**daptive Communication Assistant for Effective **I**nfobahn **A**ccess.

Figure 3.7: DFKI Persona

- *extensive presentation possibilities:* In order to use the presentation agent in a multitude of different applications, he has to offer a lot presentation possibilities. This refers to the possibilities of the agent itself, as well as to the possibilities of interaction between agent and other presentation objects.
- *application adaptivity:* When being integrated in a presentation, the agent needs an interface, which provides an easy access for the application. Furthermore, the agent should be easily adapted to application-specific requirements.
- *reactive behaviour:* The presentation agent must be able to adapt to a dynamic environment.
- *interaction with the user*: The agent should be able to provide the user with a possibility to communicate with the system he represents.

In the *Personal Picture Finder* context, the *Persona* was used to explain the user interface, lead the user through the query process and to present the resulting pictures.

Chapter 4

The Personal Picture Finder

This Chapter discusses—as core part of this thesis—the *Personal Picture Finder*. Sections 4.1 - 4.4 introduce the different versions, two of them are available to the public. Some comments about my experiments with machine learning and statistical information follow in Sections 4.5 and 4.6 respectively.
Some other applications emerged as by-products in the development of the *Personal Picture Finder* and are presented in Section 4.7.
At the end of this Chapter, the *Personal Picture Finder* is compared with related work.

4.1 The publicly accessible version

In this Section, the version of *Personal Picture Finder* available to the public is presented. The mode of operation, the concept of *parallel pull*, the architecture, and an elegant trick for accessing a database under difficult circumstances are explained in detail.

4.1.1 Mode of operation

The following chapter explains the idea and the mode of operation of the *Personal Picture Finder* using an example.
One can imagine the following situation: A journalist is looking for pictures of Alan Turing. Using the traditional approach he consults a search engine. The result he gets is a list of webpages associated with the term from the query (here: Alan Turing). This is where the journalist really starts working now: He has to download all those pages sequentially in his browser to check them for appropriate pictures. After some minutes of rather boring work, he might come across this page of Alan Turing (Figure 4.1).
Besides the picture there is a lot of—in this context—uninteresting information: icons, banner, textual information, etc. All this undesired information slows down the download process. Furthermore, the journalist only has one picture now and it would be much nicer for him to have several pictures to choose from. So he has to go ahead checking more pages.
The strategy just described is obviously time consuming and tiring. The *Personal Picture Finder* evolved out of a situation like this: One would like to have an internet agent which

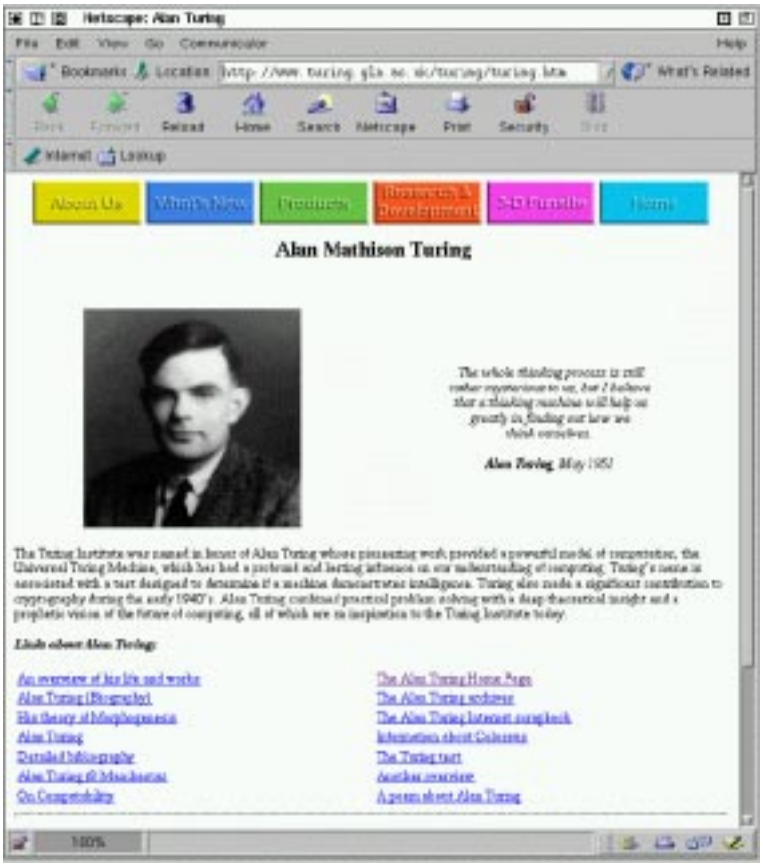

Figure 4.1: A webpage about Alan M. Turing

collects pictures of a person autonomously and fast on the internet given only the first and last name of a person and which presents all the material on one page for the user to choose from.

This is the exact purpose of the *Personal Picture Finder*. As shown in 4.2, the user types the name of a person in a form and gets a collection of several photos of this person after a few seconds. Additionally, he can click on the photos to see the page where they were found. Furthermore, the user can leave a feedback for every result, stating whether it matches his query or not.

During the search, several displays in an applet inform the user about the state of his query. A clock is counting the seconds elapsed since the search started. Other displays are:

- *Pages*: Number of webpages matching the query found so far.
- *Pictures*: Number of pictures already shown.
- *Rejected*: Number of rejected pictures.
- *Stack*: Number of requested, but not yet evaluated pages.

Figure 4.2: *Personal Picture Finder* - a search result

- *Engines*: Number of requested search engines which have not yet answered the query.

The mode of operation of the *Personal Picture Finder* can be shortly described as follows (see Figure 4.3): The user enters the name of the desired person, this name is used for parallel requests to several search engines on the WWW with the purpose of finding pages containing the requested name. The *Personal Picture Finder* then downloads the resulting pages in parallel. Furthermore, it queries several picture databases. Several filters check and sort the results of the database queries and the pictures extracted from webpages. Icons, drawings and banners are recognized and rejected. Pictures passing the filters eventually are gathered in one webpage and presented as result.

4.1.2 Parallel Pull

When manually browsing the internet for information, one ends up with a lot of results or links from search engines or other information sources. Checking them all out sequentially can be time consuming. Even for a softbot which processes information much faster, this

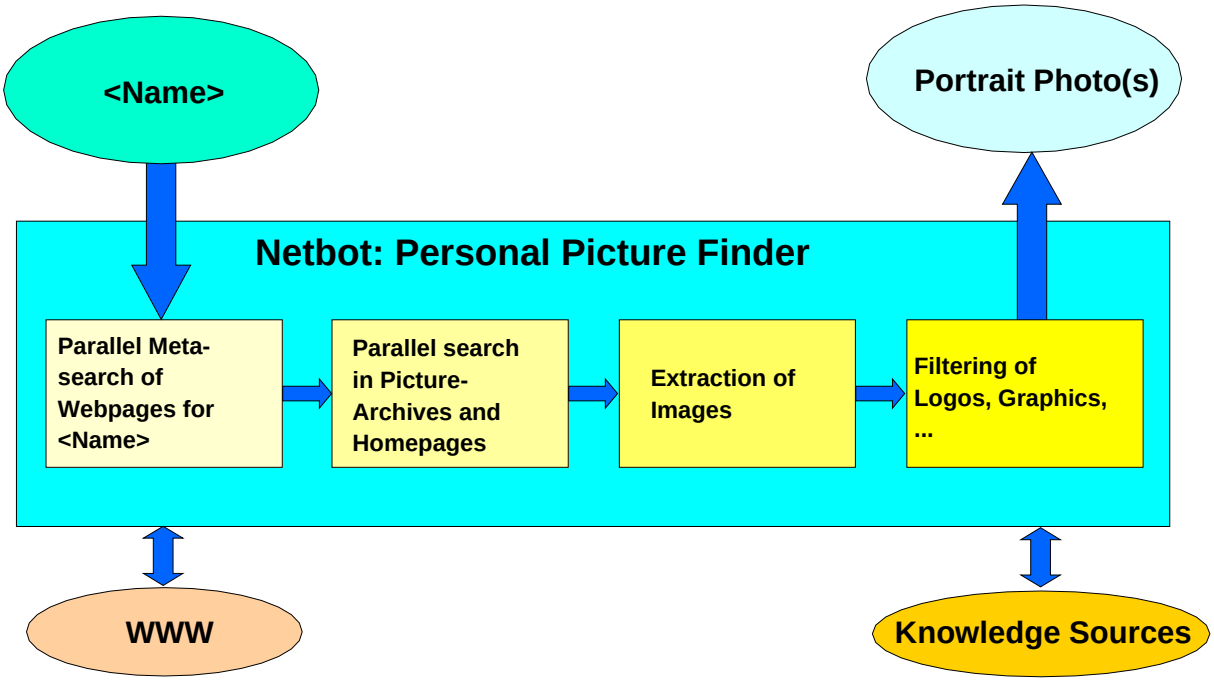

Figure 4.3: *Personal Picture Finder* - Mode of Operation

is a suboptimal operation mode.
Parallel processing seems to be a good way to avoid this problem. It can be achieved in two different ways:

- Running several processes concurrently.
  This approach has some advantages. It can be achieved in any programming language, but by using it one runs into the problem of setting up process communication based on TCP/IP, a communication overhead and an inacceptable waste of operating system resources.

- Running several threads in one process.
  Fortunately the Java programming language provides this feature, which makes the program elegant and efficient. Two problems occur in this approach:

  - Making sure every thread is destroyed when terminating the application: this is important, especially when running the application on server-side, since the Java Virtual Machine will not shut down until the server is restarted. Accumulating unused threads over a few hours or over a week will eventually kill the whole server.

  - The operating system limits the number of threads of a process. E.g. on a UNIX-platform this restriction can be set in /etc/system, appearing there as number of file descriptors. The default value on a Solaris platform is 64, for the *Personal Picture Finder* the value was set to 256. It is not advisable to use

a higher value. (Other processes on this machine would grow as well which is not desirable for the performance.)

4.1.3 The Architecture of the System

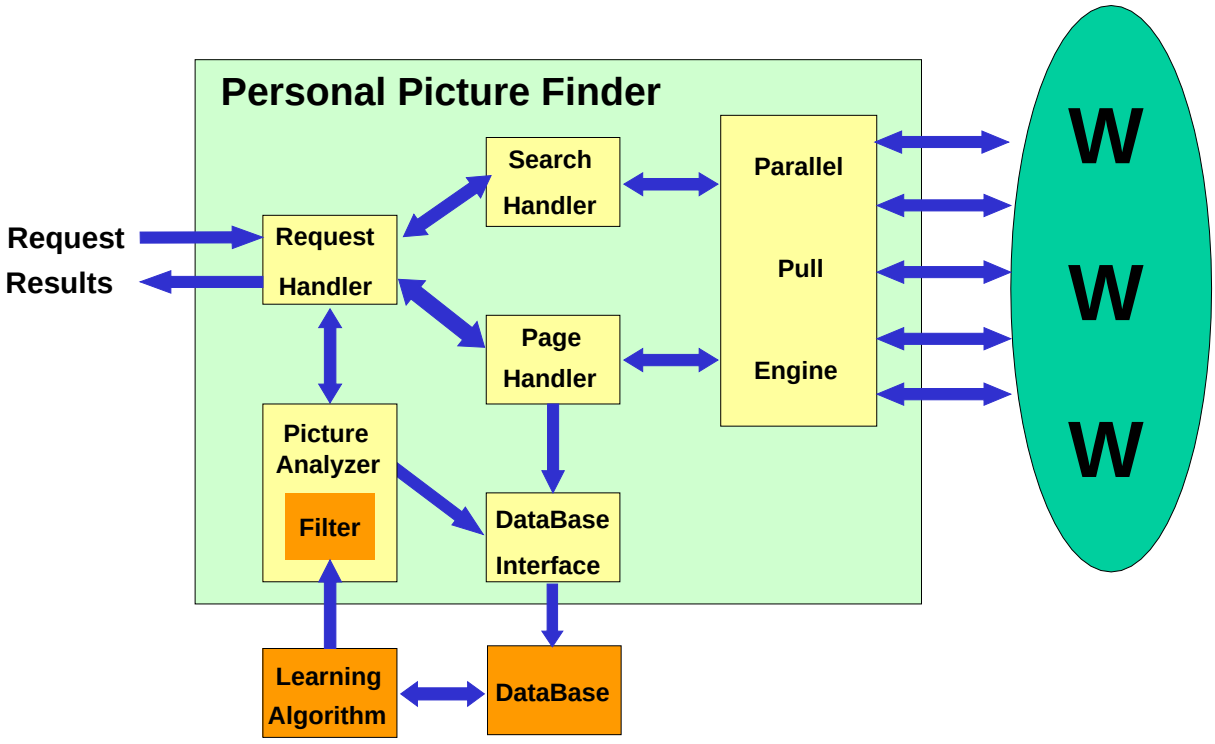

Figure 4.4: Architecture of the *Personal Picture Finder*

The previously described mode of operation will now be examined more detailed. Some additional features of the system are presented. Figure 4.4 shows a schematic representation of the system.
The *Personal Picture Finder* consists of the following modules:

- Request Handler
- Search Handler
- Page Handler
- Parallel Pull Engine
- Picture Analyzer
- DataBase interface

The *request handler* is concerned with the user's request by delegating and scheduling tasks and collecting and forwarding intermediate results. When the user finishes his request, the request handler takes care of the proper termination of all running threads. This provides stable behaviour of the system even under a lot of requests or simultaneous requests.
The request handler delegates tasks to *search handler*, *page handler* and *picture analyzer*. The *search handler* creates the appropriate query to the requested search engines out of the data given by the user query and sends it to the *Parallel Pull Engine*. The Parallel Pull Engine[1] is an essential module of the *Personal Picture Finder*. It simultaneously schedules and performs request with the objective of not having too many concurrent threads at one time.
As soon as HTML-pages are available from the Parallel Pull Engine, the search handler extracts essential information, which is either URLs of webpages associated with the request or—when requesting picture databases—the URLs of pictures.
Using the Parallel Pull Engine the *page handler* requests those webpages of which it obtained addresses from the request handler and it extracts the locations of pictures. The URLs of the pictures are passed on to the request handler which forwards it to the picture analyzer.
Using the header information of the graphic files and knowledge about its origin, the *picture analyzer* decides which pictures are to be presented to the user and which are not. The current version of the *Personal Picture Finder* uses the name of the picture and picture-specific information only to make this decision. Former versions also checked if the page where the picture came from contained the requested name. Due to refinement of the queries this is not necessary anymore. Picture information considered are parameters like size, format, compression and color depth.
The span of attention of the average WWW user is rather short. Therefore, a main implementation goal of the public version of *Personal Picture Finder* was speed. Expensive picture analyses are too slow for this purpose. The heuristics used so far provide good results. The filters reject:

- Icons: Icons can be easily recognized due to their size. Less than 64 pixels in height or width indicate that a picture is not a portrait, at least not the kind one wants to have as a result when searching for photos of a person.

- Banner: Their specific format makes it easy to detect banners. A banners width is at least twice its height.

- Drawings: Unlike scanned photos or pictures from a digital camera, drawings usually have a low color depth. We reject pictures with less then 5 bits in depth.

- Thumbnails and Previews: The product of height, width and depth indicates the size of the picture as uncompressed bitmap. By comparing this to the actual size of the file, assumptions about the quality of an image can be made. Checking the

[1]The functionality of a Parallel Pull Engine will be discussed in detail later on using the MultiHttpServer example [End99].

compression works good for pictures in GIF or PNG format, but unfortunately not for JPEG. Even highly compressed JPEGs can look very good.[2]

In order to save time and avoid unnecessary downloads, data which has been already evaluated gets stored. Java provides the JDBC[3]-interface, which is a comfortable tool for accessing databases.
In addition to the completely in Java implemented main *Personal Picture Finder* architecture, the system contains two other components. One of them is the DataBase, accessed via the JDBC-interface. It provides a fast, efficient and robust possibility of dealing with a huge amount of data. This could have not been achieved by using Java only, at least not with an appropriate amount of work and additional code. The other component is a learning algorithm whose purpose is to improve the filters of the picture analyzer. This algorithm works offline. Attempts to use machine learning techniques are described in chapter 4.5.
Stability is a major concern in the design of an internet agent. Good performance under a lot of simultaneous requests should be guaranteed. Design decisions of the *Personal Picture Finder* are based on the following considerations:

1. The agent should, as much as possible, be independent from browser, platform and operating system.

2. Client-side computations should be minimized. Also the data flow from the server to the browser should be minimal.

3. Access to the WWW should be provided without preparations on client-side. Signed applets, plugins and changes of the browsers security restrictions should be avoided.

4. Problems occurring in one request should not have any impact on concurrent or later requests.

The dataflow is shown in Figure 4.5.
The user's request entered in the HTML-form is passed by a JavaScript method to the applet. The applet accesses a servlet via http request. The servlet starts a *Personal Picture Finder* application and returns the number of a TCP/IP-port to an interface with this application. The applet opens a connection to this port and reads data from it. The information obtained here is evaluated and visualized by the applet. It either contains commands for updating the user interface or displaying a picture. In the later case, the applet calls a JavaScript function, by generating and writing the required HTML-code to the output-frame. The user can terminate this process at any time using the applet's STOP-button. This architecture is robust even under high load and is independent from platform, operating system, and browser[4]. Problems occur when the user is sitting behind a firewall not allowing a TCP/IP-connection.

[2]The JPEG-compression filters information which is invisible for the human eye. The file loses a lot of information but still looks the same to the human regarder.

[3]JDBC stands for Java Database Connectivity. It is not an acronym.

[4]Browser refers here to newer versions of popular browsers. Older browser revisions may not apply.

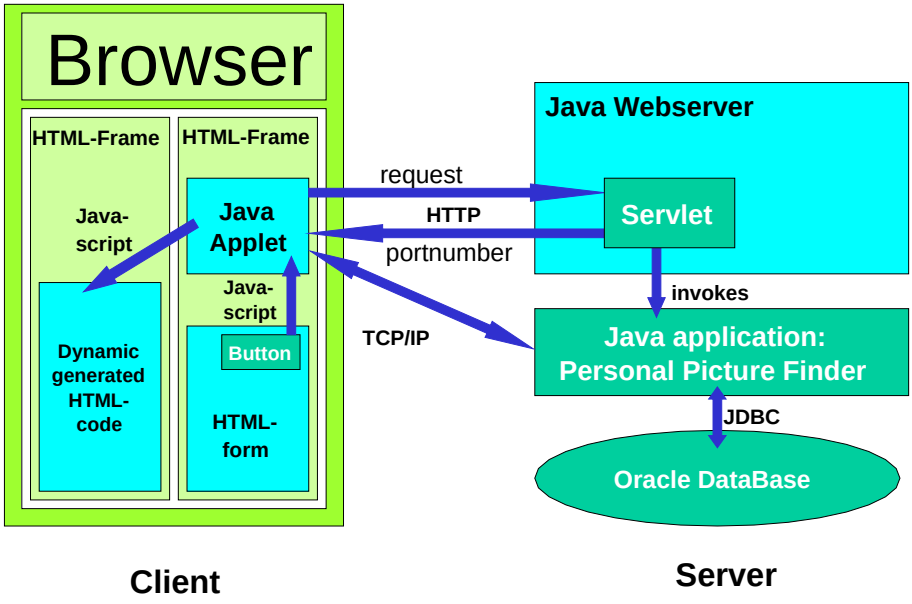

Figure 4.5: Dataflow in the *Personal Picture Finder*

4.1.4 User Feedback and Database Access

In order to store data about previously found and evaluated pictures as well as feedback from the user, the *Personal Picture Finder* uses a database. This database can be accessed on serverside using JDBC, a database interface for Java developers.
When obtaining a query, the *Personal Picture Finder* requests a servlet with the parameters of the query. The servlet consults the database and generates a file, similar to logfiles used in an older implementation.
Every picture is represented in this file as:

```
[#]<first name>,<last name>,<picture URL>,<page URL>[,<feedback>]
```

The optional hashmark (#) at the begin of a line indicates, that this picture should not be presented to the user again, either because of application of filters, or because of negative feedback.
The lines are sorted by user feedback, pictures with best feedback first.
Since the main application of the *Personal Picture Finder* is executed on serverside, storing data about new pictures can be done easily as well.
Problems occur, when providing the user with the possibility to give feedback about pictures. This mechanism should hold the following conditions:

- The user gives his feedback on clientside in a dynamic generated HTML page. The HTML page is not necessarily downloaded (or rather: generated) completely.
- The user feedback has to be entered on serverside in a database.
- The user should be provided with a fast visible feedback that his opinion has been noted.

- The current page should not be reloaded. As a matter of fact it cannot be reloaded, since most browsers do not cache dynamically generated HTML code.
- Multiple feedback should be ignored.
- The user should not be annoyed with additional windows opening etc.

The solution that holds all those conditions is a bit tricky and should be clarified using an example.
The Javascript functions generating the HTML page with the results keep track of the number of pictures displayed using a counter. This counter is necessary in order to generate a unique identifier for the picture.
As example, I assume the picture http://www.a-ten.com/alz/tur1.gif from page http://www.a-ten.com/alz/aturing.htm is the third picture to be displayed as result.
The HTML code generated to display this picture now is:

```
<table>
  <tr valign=top>
    <td>
      <a href=http://www.a-ten.com/alz/aturing.htm
         target=new>
         <img src=http://www.a-ten.com/alz/tur1.gif
              ALT=http://www.a-ten.com/alz/aturing.htm>
      </a>
    </td>
    <td>
      My comment:<p>
      <a href="javascript:void(0);"
         onMouseOver="self.status='This picture matches my query';
                      return true;"
         onMouseOut="self.status='';
                     return true;"
         onclick="window.document.good3.src=
                  'http://.../vote?pic=http://.../tur1.gif&vote=1';
                   return false;">
        <img src=up_yes.gif name=good3 align=center border=0>
      </a>
      <br>
      <a href=\"javascript:void(0);\"
         onMouseOver="self.status='This picture does not match my query';
                      return true;"
         onMouseOut="self.status='';
                     return true;"
         onClick="window.document.bad3.src=
                  'http://.../vote?pic=http://.../tur1.gif&vote=-1';
                   return false;">
        <img src=down_yes.gif name=bad3 align=center border=0>
      </a>
    </td>
  </tr>
```

```
</table>
<br>
```

Figure 4.6: The minifinder

This HTML code generates a table with one row and two columns. The left column contains the picture to be displayed, along with a link to its original source and the *alt*-tag giving the name of the page where the picture came from as additional information (for example when moving the mouse over the picture). In the right column, the icons for positive and negative feedback are loaded (an empty checkbox with a *thumbs up* or *thumbs down* icon respectively). The problem now is, that the only obvious way to send the feedback the user gives by clicking on one of these items is to make a http-request to a cgi-script, e.g. a servlet, which connects and updates the database. Every http-request on the other hand returns a result. If no result is returned, the browser at least gives an error message like "document contains no data!". There is no way of making a http-request in HTML and to ignore the returned answer. In the worst case, the browser tries to load the non-existent answer in the current frame, which simply clears the frame and destroys the current result page. As mentioned before, this page is not cached and hence lost.

If the answer can not be ignored, it has to be displayed somewhere. The trick here is to return an image instead of a HTML or text file. The servlet is requested when clicking on a feedback icon, but not using an *a*-tag but by setting the source of the current image to the servlet. Using the counter, the icons have a unique *id* specified in the *name*-parameter of the *img*-tag and hence can be substituted.
The servlet now checks its parameters. If the feedback is positive, it loads the *thumbs up*-icon with a checked checkbox, otherwise the *thumbs down*-icon with checked checkbox to an array of bytes. The servlet then sets its content type to *image/gif*, writes the array of bytes on its output stream, flushes its output stream, and then makes a connection to the database to enter the user feedback. The first part, returning the image, is very fast, so the user knows almost immediately his comment has been entered in the database.
The servlet can be called exactly once per session for a specific set of parameters. After that, the picture it returns will be cached, so multiple clicks on the same icon will cause exactly one entry in the database.
The *br*-tag at the end of the generated HTML code looks a bit unnecessary at first sight, but is very important, since some browser versions do not start displaying a table or picture until the end of a line is indicated[5].

4.2 The Minifinder

A special feature of the *Personal Picture Finder* is the *minifinder*. It provides the user with a small interface, which he can place in a corner of his screen, while browsing the web as usual (see Figure 4.6). When coming across a name he wants to look up, the user can query the *minifinder*. As soon as matching pictures are found, the *minifinder* pops up a window with results and the possibility for feedback (see Figure 4.7). After checking out the results, the user can use the *clear*-button to clear the interface and destroy the window with the results.
The idea of *minifinder* is not new. Metacrawler, for example, offers the possibility to launch a *minicrawler* with similar functionality. The main functional difference between *minifinder* and *minicrawler* (besides providing different services) is, that the *minifinder* presents the results in a new window, while *minicrawler* shows a resulting page in an already existing window.

4.3 Persona

As described in Section 3.7, an important component of *intelligent user interfaces* are *presentation agents*. An early version of the *Personal Picture Finder* used the *Persona* developed in PPP and AIA as presentation agent (see [Mül99]). *Persona* explained the interface for the user, showed him how to use it, made some comments during the search and eventually presented the resulting pictures to the user (see Figure 3.7).

[5]This is the reason why the pictures presented as results are displayed in a row and not in a column. It would simply not work to generate the HTML dynamically without know how many pictures are still to be expected.

Figure 4.7: Results of minifinder popping up

Despite all the advantages of having a presentation agents, there were some reasons to eventually remove *Persona* from the webpage again:

- Having *Persona* and the *Personal Picture Finder* (back then with more client-side computation than now!) on one page dramatically decreased the performance on slower machines.
- Special features of *Persona*, like the highly dynamic creation of presentations or interactivity could not be used in this context. *Persona* usually showed the same introduction and similar comments during the presentation of the pictures, which made it look like a simple movie clip included in the page.
- The *Personal Picture Finder* could not give enough interesting information to *Persona* to create an interesting presentation of the results.

In the next Section, I will describe an experimental version of the *Personal Picture Finder*, which collects and analyzes pictures as a background process, and presents the results to the user after completely finishing the query. Considering to include *Persona* in the

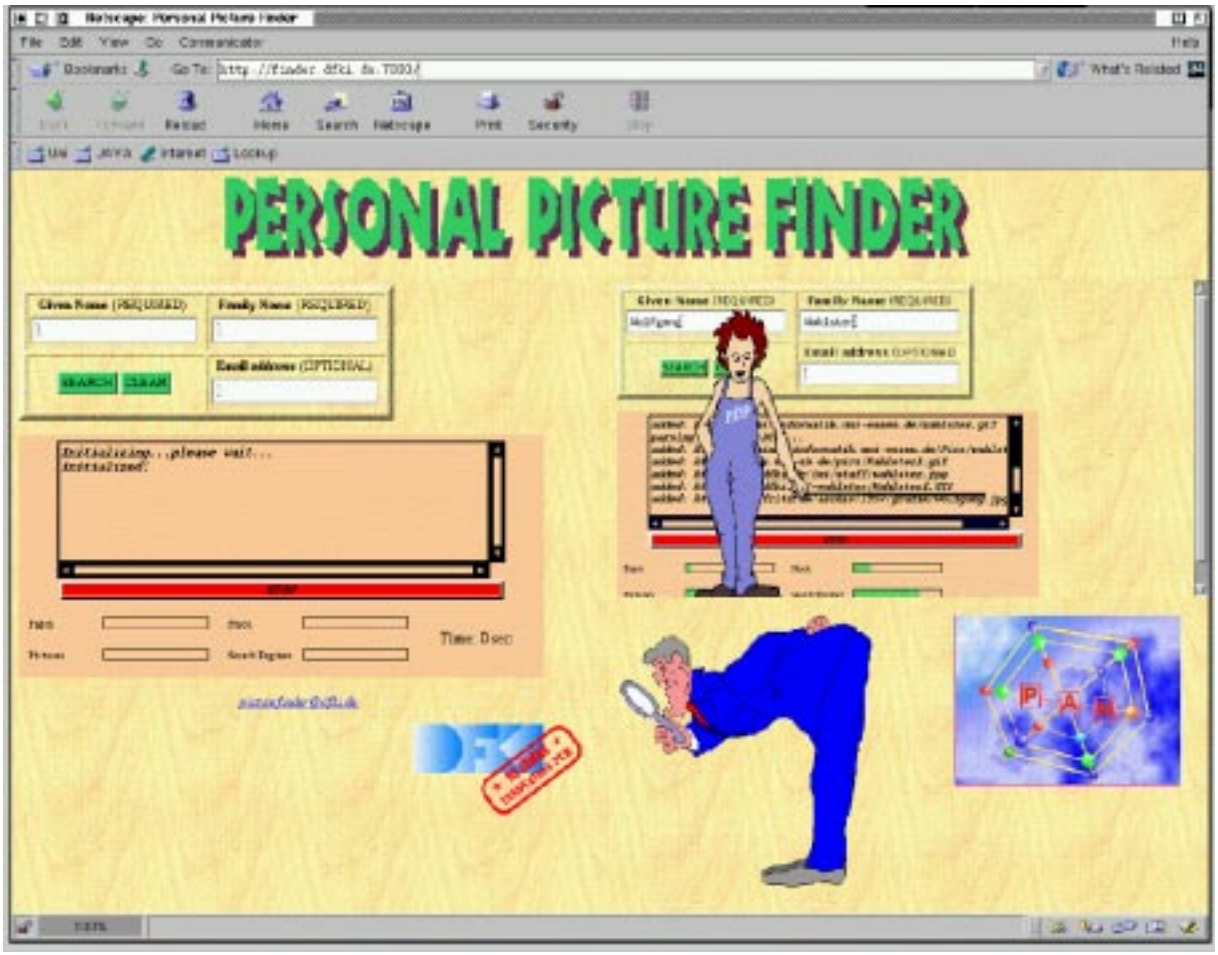

Figure 4.8: Persona explaining the *Personal Picture Finder*

Personal Picture Finder, this version would meet the requirements much better than the one previously used.

4.4 The Experimental Version

Another version of the *Personal Picture Finder* not available to the public will be described in this Section. The main differences between this experimental version and the public *Personal Picture Finder* are the strictly sequential way of programming and the presentation of the results after the search, not at runtime. Parallel pull is in its results, but debugging a parallel program sometimes is next to impossible. Especially when a multithreaded process hangs, it usually is not obvious, what caused the process to hang. By presenting the result at the very end of the search, execution time becomes less relevant and the whole process becomes more transparent. It is easier to keep track of the time needed for every step of the program.
In this Section, I describe the usage of *HyQL* and a URL generator in the experimental *Personal Picture Finder*, and its *mode of operation*.

4.4.1 HyQL and the Personal Picture Finder

As shown in Section 3.2, the *Hypertext Query Language HyQL* provides the user with an easy way of updating internet agents by using the *Programming by demonstration* paradigm. This is an enormous advantage in a highly dynamic environment like the World Wide Web.
In case of the *Personal Picture Finder*, it is useful to use *HyQL* for requesting search engines and picture databases. The way those services return their data changes frequently, which leads to a lack of information and worse results for the user. Using *HyQL*, an update can be done fast.
Extracting pictures from webpages on the other hand will always be done in the same way (unless if HTML language changes dramatically, which is not to be expected), and therefore does not require the usage of HyQL.
In this section I present the *HyQL* scripts used for two picture databases and two search engines.

4.4.1.1 Metacrawler

Metacrawler is a metasearch engine. It requests several search engines, collects, and sorts the results. The HTML code of the returned page is rather big, usually between 30 and 40 Kilobytes. The information needed form it is up to 20 links matching the query.
The following script extracts the desired information, by first extracting all links from the page and then choosing those not being local links, e.g. links leading to other services on the same site or to commercial presentations. The results are returned to the user as links separated by newlines.

```
{let    info i1 := root,descendant(all,a)href
 from {select content from
         document d1 in
           http://search.go2net.com/crawler?general=Alan+Turing&... }
 where i1 nomatch "*go2net.com*"},
{select textnl (root,descendant(all)) from i1}
```

4.4.1.2 Ahoy!

Ahoy! is a search engine specialized in looking up private homepages. The resulting links are represented not only as links, but as well on the page itself, printed in italic. The following script simply looks up all italic parts of the text form the returned document and returns them separated by newlines.

```
select  textnl(
        root,descendant(1,body)(alltext,i))
from select content from
        document d1 in
           http://ahoy.cs.washington.edu:6060/cgi-bin/...
```

4.4.1.3 Lycos

Lycos is a picture database. It provides the user with pictures and links to the source of the picture, but usually not with a link to the page where the picture was found. The *Personal Picture Finder* in this case simply returns the address of Lycos as origin of the picture.
In the following script, all links from the page returned from the request are checked for the extensions *gif* or *jpg*. The URLs again are returned separated by newlines.

```
{let    info i1 := root,descendant(1,body)(all,a)href
 from {select content from
        document d1 in
           http://www.de.lycos.de/cgi-bin/pursuit?query=Alan+Turing&...}
 where i1 match "*jpg" or i1 match "*gif"},
{select textnl (root,descendant(all)) from i1}
```

4.4.1.4 Altavista

Altavista is another picture database. Unlike Lycos, it returns links to the picture and to the page where the picture was found. The following script works in two steps. In the first step, all links matching *image/* are collected. In the second step, *src* and *href* of these links are extracted and returned, separated by newlines.

```
{let info i1 := root,descendant(all,a)
 from {select content from
         document d1 in
           http://image.altavista.com/cgi-bin/avncgi?query=%2BAlan%20... }
 where root,descendant(2)src applies to i1 matches "*image/*"},
{select         textnl(
        root,descendant(all,a)href,
        root,descendant(all,img)src)
from i1}
```

Detailed information about the *Hypertext Query Language HyQL*, along with some more sample scripts, can be found in [Den99b] and [Den99a].

4.4.2 The URL generator

A *URL generator* is a function which generates the possible URL of a page using known information about this page. Based on the email address of a person for example, an assumption about this persons homepage can be made. Especially after collecting a lot of email addresses and URLs of persons, patterns how the URL can be retrieved form the email address can be found.
Example:
The URL of a person with email address <lastname>@dfki.de usually is http://www.dfki.de/~<lastname>/

The *Personal picture Finder* has no information about the email address of a person,

which makes the generation of URLs a bit more difficult, since the generation is based on the persons first and last name only. At least for prominent persons, it works successfully anyway. The currently used *URL generator* produces, given the name Alan Turing, the following URLs:

```
http://www.alanturing.com/
http://www.alanturing.de/
http://www.alanturing.org/
http://www.alan-turing.com/
http://www.alan-turing.de/
http://www.alan-turing.org/
http://www.turing.com/alan.html
http://www.turing.com/pic/alan.html
http://www.turing.com/pics/alan.html
http://www.turing.com/alan/
http://www.dfki.de/staff/karte.pl?turingalan
```

Two out of these 11 URLs provide useful information, the rest of these pages simply does not exist and will be ignored.

4.4.3 Mode of Operation

The span of attention of the average WWW user is rather short, hence the implementation of the version of *Personal Picture Finder* available to the public had to return results fast. This version of the *Personal Picture Finder* is not available to the public. Results are not returned until the whole search is finished. Its objective is a better quality of service instead of fast results.
From the implementors point of view, this approach avoids a lot of problems, especially with the correct termination of concurrent threads and the generation of a not-cached webpage on the fly.
The implementation based on a Bourne shell script, doing all the necessary steps sequentially and eventually gathering the results in one webpage to be presented to the user by launching a browser.
The execution of the script takes between 15 and 90 minutes, depending on the amount of collected data to be examined.
Another advantage of the script-form is the simple adaptivity and enhancement with new features. New search engines or filters can be added easily, just by adding one or two lines to the script.
In this section, the script is presented and explained. The line numbers are not part of the script.
At DFKI, this program is installed on serv-203 and can be used by calling

```
/home/endres/ppftools/search <firstname> <lastname>
```

The names should not contain '-' or whitespaces, '+' can be used instead.

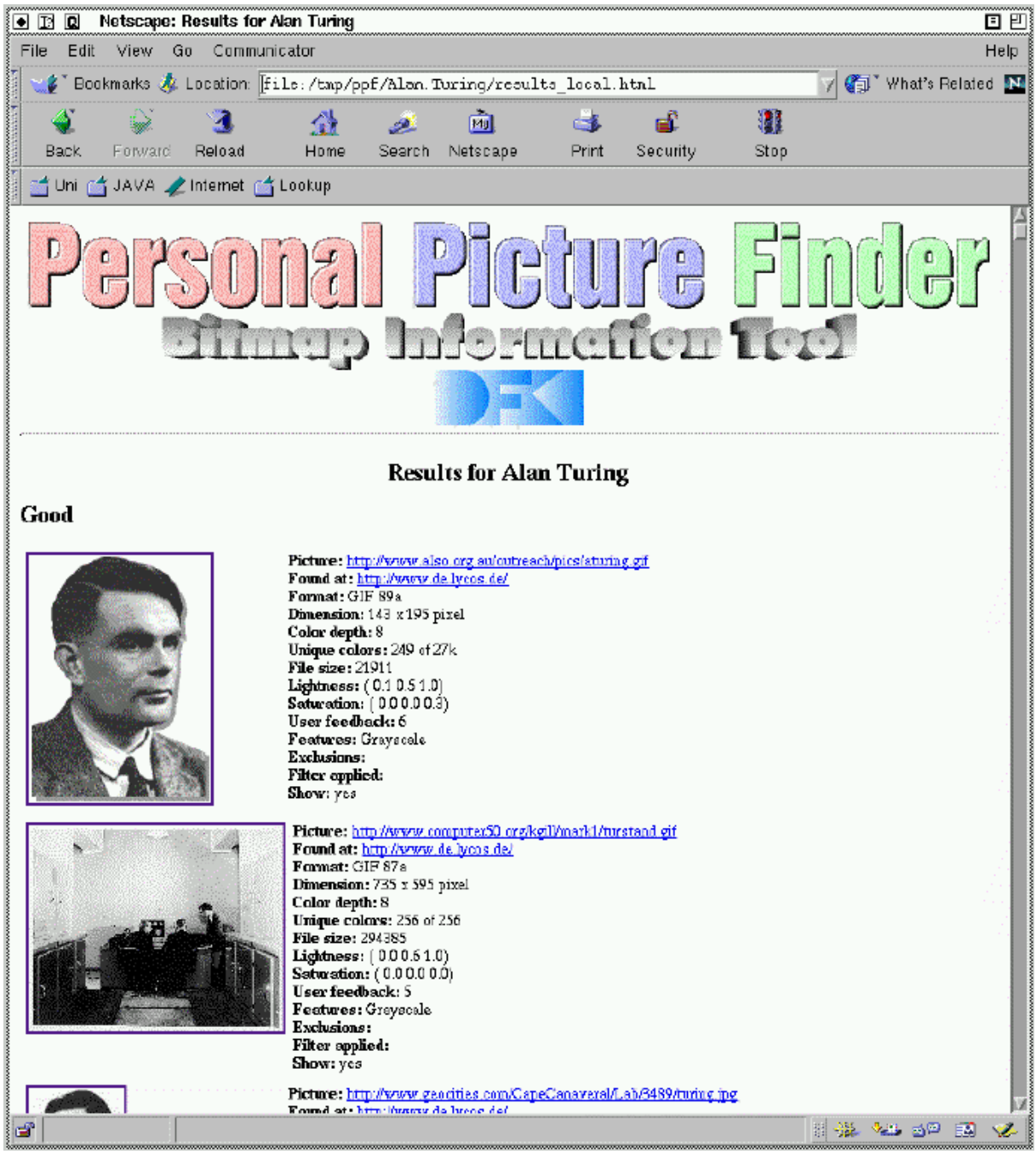

Figure 4.9: Pictures found while searching for Alan Turing

```
1  #!/bin/sh
2
3  START=‘date‘
4
```

The first line calls the Bourne shell as interpreter of this script. After that, the time when the script execution was started gets stored in variable START.

```
 5  SCRIPT=‘/bin/basename $0‘
 6
 7  usage() { echo "usage: $SCRIPT <first name> <last name>"; }
 8
 9  if [ $# -ne 2 ]; then
10    echo "Personal Picture Finder" 1>&2
11    usage 1>&2
12    exit 1
13  fi
14
```

Lines 5-14 make sure that the user gets information about the usage of the script in case he calls it with a wrong number of arguments.

```
15  PATH=/usr/bin:/bin:/opt/java/bin:/opt/X11/bin:$PATH:/home/haase/pkg/bin
16  BITDB=/tmp/.bit/picdb
17  BITWD=/tmp/.bit
18  CLASSPATH=.:/home/endres/ppftools:/home/endres/lib/java
19  export CLASSPATH
20  PPFTEMP=/tmp/ppf
21
22  umask 000
23
```

In line 15-23, the necessary pathnames and variables are set. PATH should include the location of all programs called at runtime, in this case especially Java, netscape, and bit. BITBD and BITWD are the paths for the database and the working directory of bit respectively. CLASSPATH points on the necessary Java libraries, here to the ppftools written for this script and my other libraries containing the *Personal Picture Finder* API. A temporary directory for the script execution is set, and the umask is set to 000 to avoid problems occurring when two users execute the script at the same time using the same working directory.

```
24  WORKDIR=$PPFTEMP/$1.$2
25  mkdir -p $WORKDIR
26  mkdir -p $BITWD
27  cd $WORKDIR
28
29  rm -f *
30  touch pages.txt
31  touch pictures.txt
32
```

The necessary directories are created, the working directory is cleaned from previous files and the files pages.txt and pictures.txt created.

```
33  ## consult database
34  dbURL="http://finder.dfki.de:7000/generateLogfile?first=$1&last=$2"
35  echo "requesting database"
36  java GetFile $dbURL dbentry.txt
37
```

The content of the database for the requested name is downloaded in the file dbentry.txt.

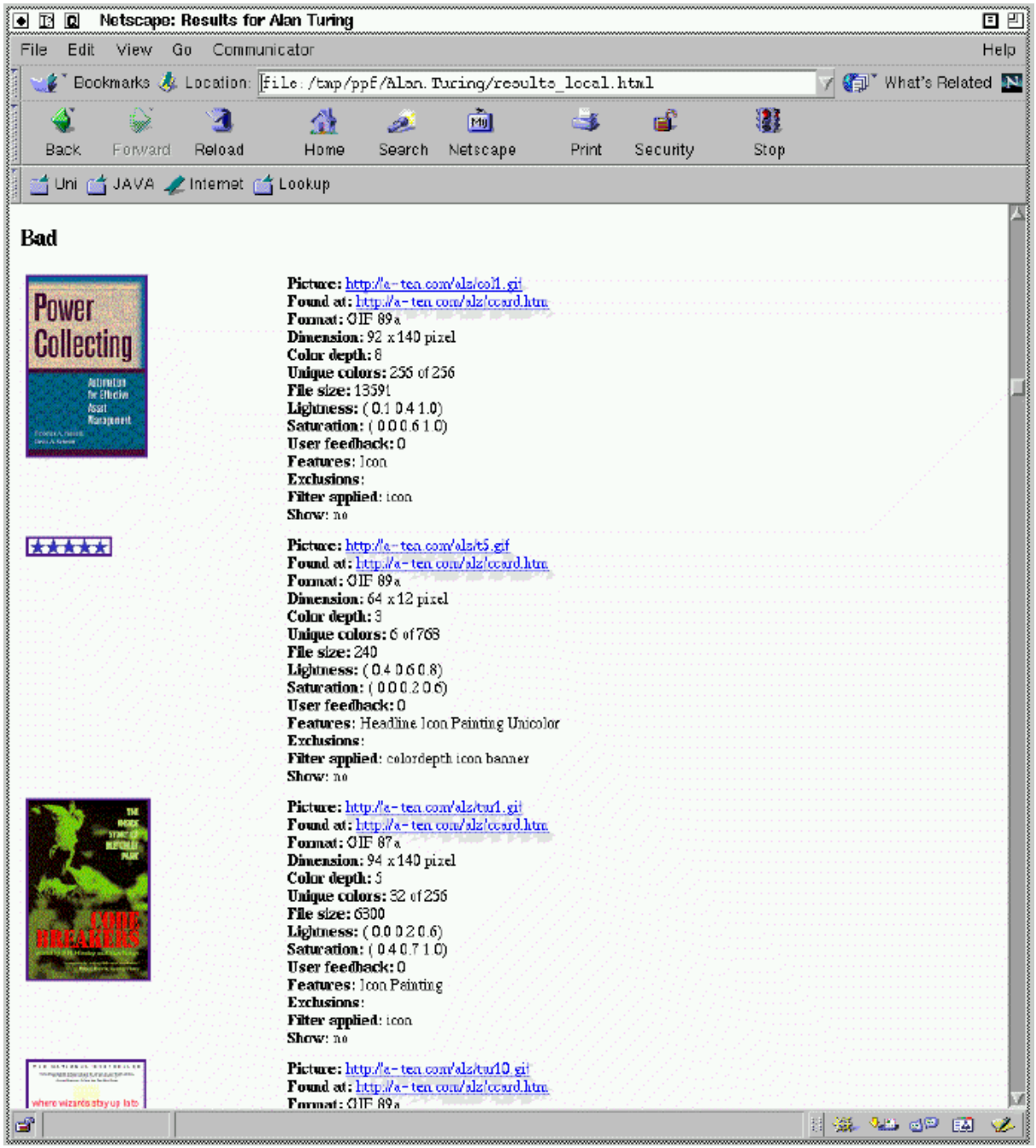

Figure 4.10: Rejected pictures

```
38  ## consult urlgenerator and search engines
39  echo "guessing urls..."
40  java urlgenerator $1 $2  >> pages.txt
```

```
41  echo "requesting metacrawler."
42  java  metacrawler $1 $2 >> pages.txt
43  echo "requesting ahoy."
44  java ahoy $1 $2 >> pages.txt
45  echo "requesting lycos."
46  java lycos $1 $2 >> pictures.txt
47  echo "requesting altavista."
48  java altavista $1 $2 >> pictures.txt
49  echo "all available data from search engines collected."
50
```

The *URL generator*, the search engines, and the picture databases are consulted sequentially. Information about pages is stored in the file pages.txt (each URL in a separate line), the information from the picture databases in pictures.txt.

```
51  ## remove multiple occurrences (pages)
52  echo "sorting pages"
53  sort -u -o pages.txt pages.txt
54
55  ## set new field separator
56  OLD_IFS=$IFS
57  IFS='
58  '
```

Multiple occurrences of pages are removed using the *sort -u* command. The field separator is set to newline.

```
59  ## extract pictures from pages
60  for URL in `cat pages.txt`; do
61    echo "extracting pictures from $URL"
62    java extractPictures $URL >> pictures.txt
63  done
64
```

For each URL in the file pages.txt, the Java application *extractPictures* is called. The information about pictures extracted from the page is appended at the end of file pictures.txt.

```
65  ## check database
66  echo "checking database"
67  java checkDataBase dbentry.txt >> pictures.txt
68
```

Information about pictures matching the query is appended at the end of file pictures.txt.

```
69  ## remove multiple occurrences (pictures)
70  echo "sorting pictures"
71  sort -u -o pictures.txt pictures.txt
72
```

Multiple occurrences are removed from file pictures.txt, again by using the *sort -u* command.

```
73  ## download and analyze pictures
74  COUNTER=0
75  for LINE in `cat pictures.txt`; do
76    PICTURE=`echo $LINE | awk -F',' '{print $1}'`
77    PAGE=`echo $LINE | awk -F',' '{print $2}'`
78    COUNTER=`echo $COUNTER+1 | bc`
79    SUFFIX=img_`echo 000$COUNTER | sed -e 's/^.*\([0-9]\{4\}\)$/\1/'`
80    echo "downloading picture $SUFFIX"
81    java getPicture $PICTURE $PAGE $SUFFIX dbentry.txt
82    PICTURE=`awk '/^local:/{ print $2 }' $SUFFIX'.txt'`
83    bit -v -h -c -p -d -s -f $PICTURE | tail +3 | strings >> $SUFFIX.txt
84  done
85
```

All pictures from file pictures.txt are downloaded sequentially. The format of file pictures.txt is:

```
<picture-URL>,<page-URL>
<picture-URL>,<page-URL>
<picture-URL>,<page-URL>
...
```

From each line, the URL of a picture and the URL from the Page where this picture was found is stored in the variables PICTURE and PAGE respectively. A new basename for the picture is generated using a COUNTER and stored in SUFFIX. The new basenames of the pictures are img_0001, img_0002, etc.
The pipe to the UNIX command *bc* in line 78 is necessary, since the bourne shell does not include any arithmetic operations at all.
In line 81, the Java application *getPicture* is called with PICTURE, PAGE, SUFFIX and the name of the file containg the database entry. *getPicture* downloads the picture to <basename>.<extension>, with the new basename and the extension *.gif*, *.jpg*, or *.png* depending on the format of the picture. Since the picture is downloaded completely anyway, the former restriction to the 400 bytes for analyzing the picture does not apply. This is useful especially for the older JPEG version 1.00 or 1.01, which usally could not be analyzed using the begin of the file only and therefore caused bad results sometimes. Besides downloading the picture, *getFile* creates a file <basename>.txt containing information about the picture, i.e. its parameters and information from the database. In line 82 the variable PICTURE is set to the name of the picture on the local filesystem. This information is extraced from <basename>.txt using the *awk* tool[6]. In line 82, *bit* [Haa99] is called with PICTURE as parameter and the resulting information appended at the file <basename>.txt.

```
86  ## set field seperator to previous value
87  IFS=$OLD_IFS
88
89  ## optional: delete pictures not to be shown
```

[6]awk is an acromym for **A**ho, **W**einberger, **K**ernighan.

```
90  # for file in *.txt; do
91  #   if [‘awk ’/^show:/{ print $2 }’ $file‘ = ’no’] ; then
92  #      rm -f ‘basename $file‘’.*’
93  #   fi
94  # done
95
```

After finishing the iteration over lines in files, the field seperator is set back to its old value in line 87. Lines 89-94 contain a special feature to delete every image containing the line *Show: no* in its textual description. This feature is currently not used, and therefore commented out. It should be used when only matching are expected as result.

```
96  ## collect results
97  touch results.txt
98  echo "collecting results..."
99  for file in ‘/bin/ls -1 img*.txt | sort‘; do
100   cat $file >> results.txt
101   echo ’*’ >> results.txt
102 done
103
```

In lines 96-102, the results for every picture are collected in one file, *results.txt*. A $\star$ is added as seperator.

```
104 ## create HTML-page
105 echo "preparing htmlpage..."
106 ## usage: java collectResults <infile> <local/global> <first> <last>
107 java collectResults results.txt local $1 $2 > results_local.html
108 java collectResults results.txt global $1 $2 > results_global.html
109
```

The Java application *collectResults* creates the results as HTML pages (see Figures 4.9 and 4.10). The pictures are divided in two categories, *good* and *bad*, pictures bigger than 200 pixels in width are resized, links to the original source of the picture and the page where it came from as well as the information available about the picture are added. The images in the category *good* are sorted in order to display the best ones first.
The file *results_local.html* includes the pictures from the local file system, while *results_global.html* uses the original source. The latter one is much slower, but can be mailed to somebody else without including all the image files from the working directory.

```
110 ## print out information about search time
111 echo "Search started at $START"
112 echo "finished at "‘date‘
113
```

For the information of the user, the time when the search was started and ended is displayed.

```
114 ## show results
115 netscape results_local.html &
```

Finally, the results are displayed by launching the users browser.

Example:

```
% ./ppftools/search Alan Turing
requesting database
guessing urls...
requesting metacrawler.
requesting ahoy.
requesting lycos.
requesting altavista.
all available data from search engines collected.
sorting pages
extracting pictures from http://aleph0.clarku.edu/~djoyce/mathhist/webresources.html
extracting pictures from http://ei.cs.vt.edu/~history/Turing.html
...
checking database
sorting pictures
downloading picture img_0001
downloading picture img_0002
...
downloading picture img_0206
downloading picture img_0207
gathering results...
preparing htmlpage...
Search started at Wed Apr 28 18:07:12 MET DST 1999
finished at Wed Apr 28 19:15:14 MET DST 1999
```

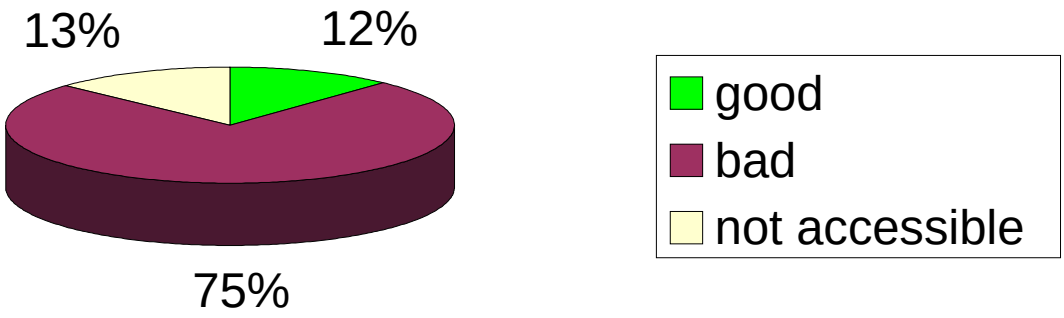

Figure 4.11: Results of the evaluation of pictures

The execution of this request took 68 minutes. 207 URLs of pictures were found on pages containing the name Alan Turing. 31 could not be downloaded. Out of the 176 pictures successfully contained, 28 were evealuated as *good*, and 148 as *bad* (see Figure 4.11). 43 pages were consulted.

4.5 Experiments with Machine Learning

According to the problem specification, the *Personal Picture Finder* should improve its graphics filters by learning from the collected data using low-level machine learning.

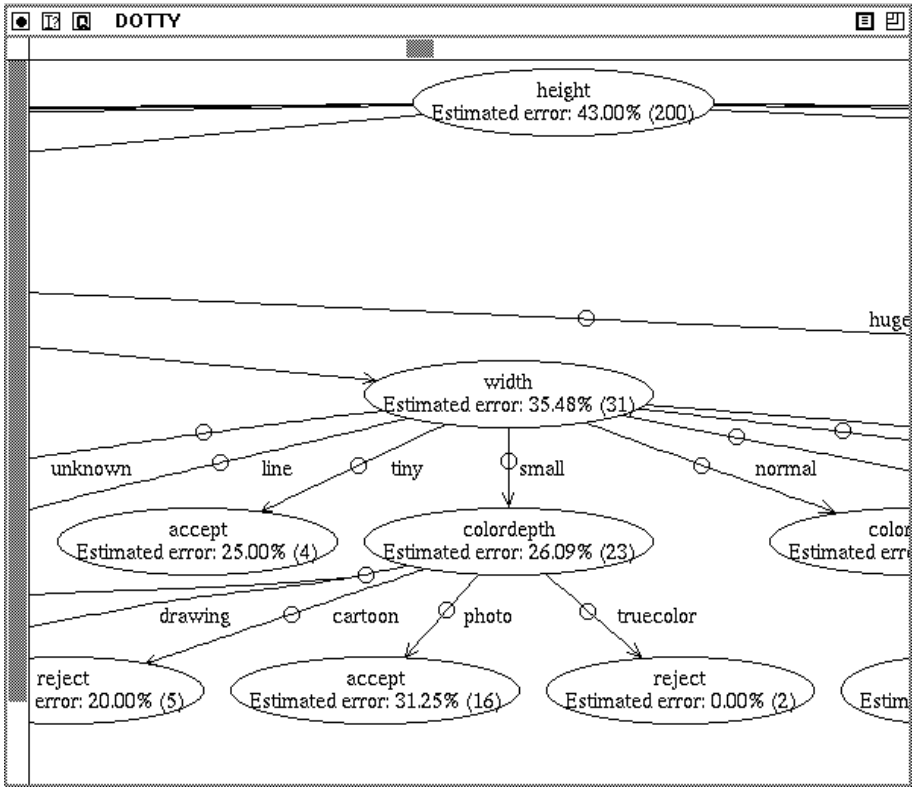

Figure 4.12: Decision tree

The collected data contain information about previously found pictures and user feedback. The simplest way of learning from those data is:

- Remembering results from previous requests with the same name.
- Remembering the user feedback about pictures previously displayed.

When the *Personal Picture Finder* is queried, the database is looked up for previous queries with the same name. The information obtained can be used for adding previously found pictures to the result set (*memoization effect*), rejecting previously rejected pictures and thus saving time, rejecting pictures rejected by user feedback and sorting pictures to be shown in order to show the best pictures first.
When the user gives a feedback to a picture, the value +1 or -1 is entered in the database for good or bad feedback respectively. Obviously it is not advisable to reject a picture after the first bad user feedback. If the feedback was wrong and the picture is rejected in further requests, there is no way of correcting this mistake. Looking up the database, the sum over those values is calculated, and pictures with a feedback value lower than -2 are rejected.
Besides this low-level instance-based learning, I tried to find some general new filters using *supervised classification learning*. Suitable libraries written in C++ are provided by *MLC++*, a *Machine Learning Library of C++ classes* developed at Stanford university [Koh95].

The *induction task* here is to obtain a *classifier*, which, given the parameters of a picture, labels the picture with *accept* or *reject*, by making an assumption about the user's feedback for this picture. The task is inductive (as opposed to deductive), because there will be no formal justification (i.e., a proof) of the resulting *classifier*.
In order to induce such a *classifier*, the *induction algorithm* must have some input. This type of supervised learning is sometimes called *tabula rasa learning*, as no domain knowledge is given to the learning algorithm. Here, the input will be a set of picture parameters, called *instance*. In order to train the algorithm, each instance has to be labelled with category the *classifier* should categorize this instance, i.e. *accept* or *reject*.
Each instance is a list of *attribute values*, such as *filesize*, *colordepth*, *width*, *height*, and *shape*. These attributes can be obtained from a database, and their values have to be *discrete*, meaning all information about an instance must be expressible as a fixed list of values. For example, the width of a picture is not expressed in pixel, but instead as one of the values *unknown*, *line*, *tiny*, *small*, *normal*, *large*, *big*, and *huge*.
The set of labelled instances given to the induction algorithm is called a *training set*. Using this *training set*, the induction algorithm is able to generate a classifier. A *classifier* can have many forms, like *table*, *nearest neighbor*, *decision tree*, or *boolean formula*. Here, I choosed the *decision tree*.
After obtaining the *classifier*, it is important to estimate, how good its perfomance is. If the topic to be classified is well know, it is possible to consult a domain expert and ask for his opinion. Another possibility is to test the *classifier* with new instances that were not used for training, and evaluate its performance. The set of instances used for this is called *test set*. It is usually assumed that the training set and the test set are random samples from some underlying distribution. This applies here, since I choosed randomly 1422 instances from the database and split them up in two sets, one with 1222 instances as the *training set*, and one with 200 instances as *test set*.
The *MLC++* library offers a variety of induction algorithms. I choosed the classic *ID3* algorithm [Qui86].
A part of the resulting tree is shown in Figure 4.12.
For the evaluation, two cases have to be distinguished: first, how many good pictures from the *test set* were rejected and how many bad pictures were displayed.
The resulting decision tree has an average error probability on the *test set* of 42.5%. Out of the 200 instances of the *test set*, 73 have been accepted, 23 of them by mistake. The rate of falsely accepted pages results as 31.5%. More important are in this concept are good pictures, that have been falsely rejected. Out of a total of 127 pictures rejected by the decision tree, only 65 had to be rejected. 48.8% of the rejected pictures were okay, which is obviously an inacceptable quote.
Due to this, the low-level machine learning for the filters from the problem specification relies only on instance-based learning.
About the reason for the failure of the *supervised classification learning*, two assumptions can be made:

1. The data provided by the user may be biased, i.e. maybe it was not completely obvious to the user, on which criteria his feedback should be used.

2. Training the inducer by formal parameters of the pictures may be not sufficient.

The first problem could be solved, by explaining the meaning of the feedback to the user a bit more as well as with checking the feedback before giving it to the inducer. For the latter, additional information about the relative position of a picture in the document and the requested name perhaps could increase the quality of the results.

4.6 Statistics

An important aspect when providing a web service available to the public, is to find out about the users preferences. In Appendix C, the evaluation of the log files is visualized. Figure C.1 shows an almost constant rate of 600-700 requests for January until April 1999. Much higher values in November (1634 requests) and December (1162 requests) can be explained by two press releases, namely a long article in *Computerwoche*[7] and a short note with the URL of the *Personal Picture Finder* in *Künstliche Intelligenz*[8], a german AI magazine, in a survey on intelligent agents.
Figure C.2 shows the usage of the *Personal Picture Finder* over the week. The access rate is almost constant from Monday til Friday, with peaks on Monday and Thursday, and significantly decreasing at the weekend.

In Figure C.3, the usage of the *Personal Picture Finder* gets related to the time of the day[9]. Least usage can be found at 6 a.m., from then constantly increasing—with a peak before lunchtime at 11 a.m.—until 3 p.m. and then decreasing again until 6 a.m. An interesting aspect is the almost constant usage between 4 a.m. and 7 a.m. So after finding out when the *Personal Picture Finder* is requested, the next question is concerned with the users location. In Figure C.4, the origin of the requests is sorted by top-level domains. As expected, most of the requests[10] (3847) are from Germany (due to the press releases mentioned before and appearance on nationwide TV news in July 98). Next are the domains *.net* (613) and *.com* (399), followed by *.se* (Sweden, 327) and *.es* (Spain, 109).

4.7 By-products

This Section is about the applications developed in the context of the *Personal Picture Finder*. Two netbots, a tool for analyzing graphics file formats and the *MultiHttpServer*—an implementation of the *parallel pull* concept—are introduced.

[7] Computerwoche 44/98, October, 30th, 1998, pages 25-26
[8] Künstliche Intelligenz, 3/98, September 1998, page 61
[9] Time refers here to MET and MET DST, since most of the users live in this timezone.
[10] About top-level-domains, more statistic data was available than for the access time, hence the sum over the statistic by top-level domains is not equals the sum over the requests in the time statistics.

4.7.1 Netbots

While trying to find the appropriate architecture for the *Personal Picture Finder*, I examined several other possibilities as well. One way for building a netbot is the usage of a servlet based architecture only. The user interface consists of a simple HTML form only. After filling the form in, a servlet with the specified parameters is called. The output produced by the servlet is a dynamic generated HTML page. The advantage of this architecture is the lack of any security critical operation. Also, the user needs no additional capabilities in his browser besides displaying HTML pages. All computations are performed server-side. The disadvantage is—compared with the *Personal Picture Finder*-architecture—the rather simple display. Another disadvantage is producing a high load on the server by performing all computations there.
The *Personal Picture Finder* could have been implemented like that too[11].
In this section I present two simple netbots based on servlet.

4.7.1.1 A shopbot for CDs

The objective of the netbot presented here deals with buying CDs on the web. Given a timeout and the title and artist of a CD, the netbot requests several CD shops in germany[12], collects the results it gets in return before the specified timeout and presents the results in a unique way in a table. The decision which provider is providing the specified item most economic-priced is still a difficult task, because of different conditions for shipping etc. and therefore left to the user.
A special feature of this netbot is its configureability. When getting a request, the servlet reads a config-file specifying several shops. This specification includes the name of the shop, information how to request it and how to parse the resulting page, the name of the currency and the URL for further information of the business conditions of this shop.

Example 1:

```
name: recordplanet
request: http://www.recordplanet.de/cn/euroshop.dll?command=parsequery
                  &mandant=11215&dlgtext1= $QUERY &submit=Schnellsuche
main: http://www.recordplanet.de/
agb: http://www.recordplanet.de/cn/euroshop.dll?command=sendpage&page=Agb.htm
currency: DM
table: 5
skiplines: 0
wrapper:
TD
TD
FONT Text Price
TD
TD
```

[11] As a matter of fact there was an experimental version of the *Personal Picture Finder* implemented in a servlet only.

[12] I restricted the search to one country to avoid comparing different currencies.

```
FONT Text Description
TD
FONT Text Number
*
```

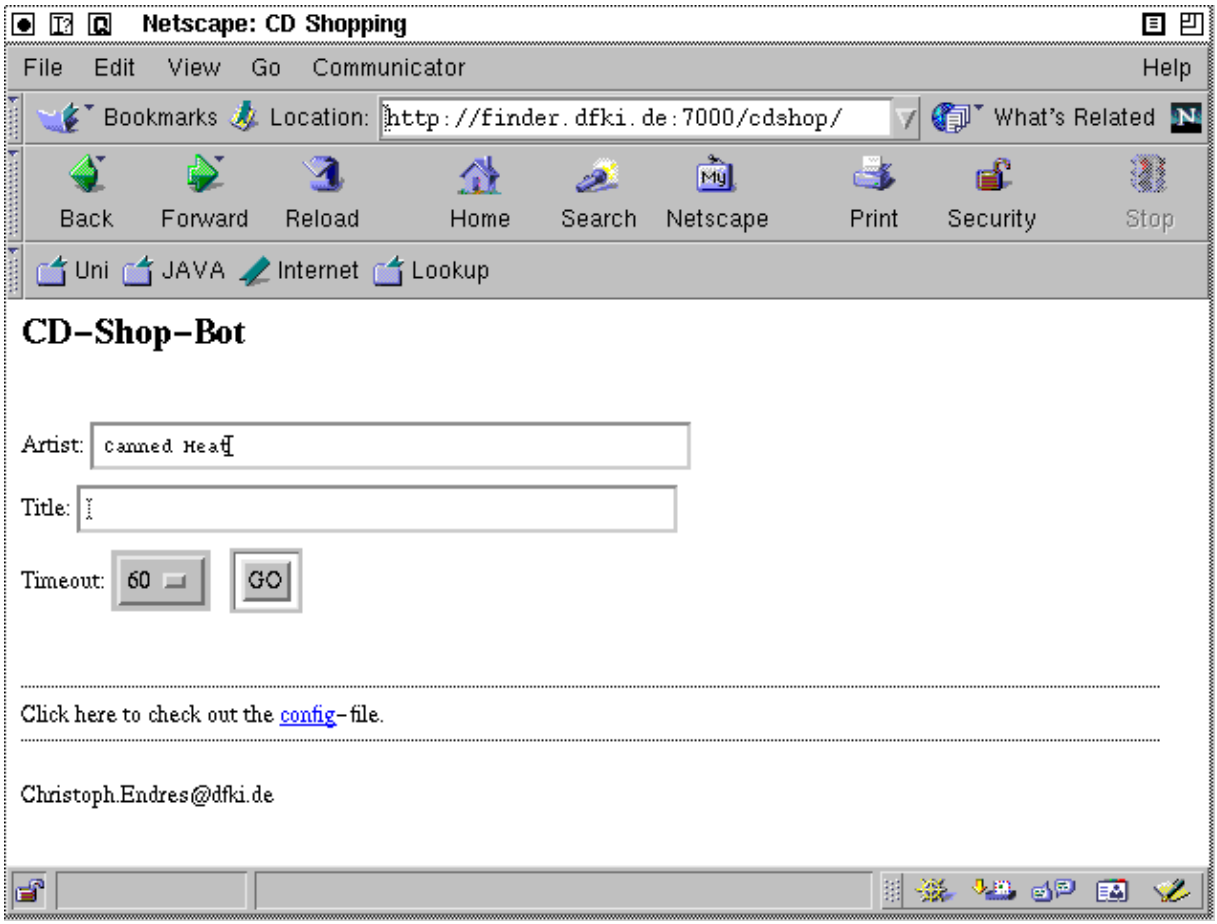

Figure 4.13: User interface of the CD shopbot

The request is performed by taking the string specified in the *request:* line and evaluating the variables. In this example, the provider does not distinguish between artist and title but instead offers a keyword search in its database. The netbot takes the parameters specified by the user, concatenates it and puts and substitutes the variable *$QUERY* with it. The resulting request string is used for a HTTP access in the World Wide Web. When a page is returned, the netbot uses a simple wrapper to find the desired information. In this case, the fifth table is looked up. This table contains no header lines to be skipped. The specified tags are looked up in every line. In the FONT tag following the second TD tag the information about the price is found as Text included in this tag. The description of the item and the item number can be found in a similar way.

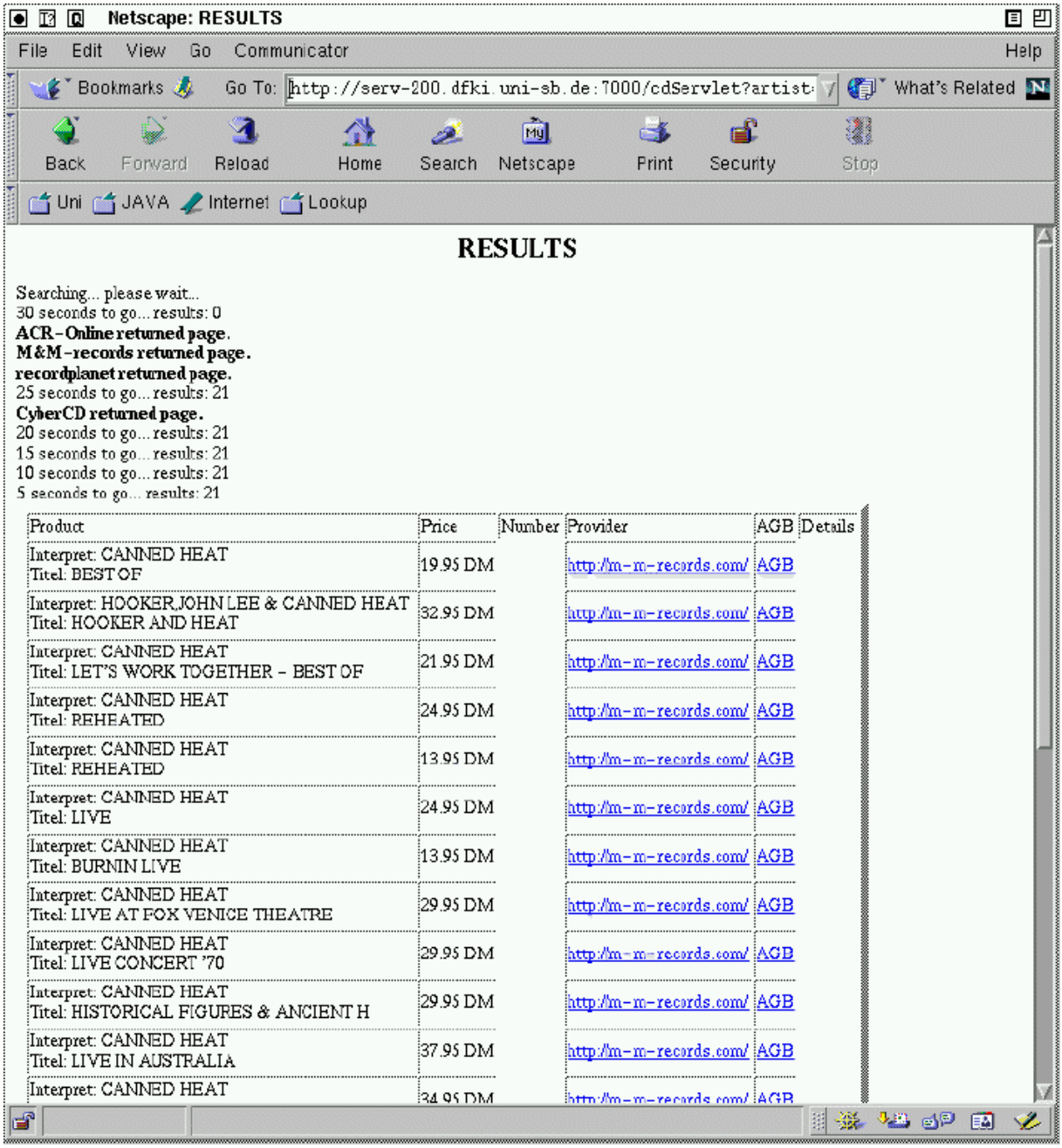

Figure 4.14: Result of a query for 'Canned Heat'

Example 2:

```
name: jpc
request: http://www.aeon-plaza.de/MallParser?
         code=aeon.mall.servlet.plugin.SearchCD&providerid=10?cd=cd
         &track=track&artist= $ARTIST &title= $TITLE
main: http://www.aeon-plaza.de/providers/jpc/search_cd.mall
agb: http://www.aeon-plaza.de/providers/jpc/ShopInfo.mall
currency: DM
```

```
table: 2
skiplines: 1
wrapper:
TD
TD
A Href Info Text Artistname
A Text Title
TD
FONT Text Price
*
```

This example differs from the previous in the way the request is build. Here, the provider distinguishes two variables for database lookup, namely artist and title. The information specified by the user is filled in in the variables *$ARTIST* and *$TITLE* in order to perform the request. The wrapper takes the result, locates the second table, skips the first line of this table and then checks every line for the specified tags. In the first *A* tag after the second *TD* tag, the information about the artist's name is found in the text included in this tag. Furthermore a link to more detailed information about this item is provided in the *HREF* parameter of the same tag. The price of the item is specified in the first *FONT* tag following the next *TD* tag.
The advantage of this architecture is the easy way of configuring it: Adding some lines specifying another provider to the config file is all one has to do for providing another source of information. As well, just by exchanging the config file, one could make a shopbot for books as well without any changes in the program code at all.
This netbot uses libraries of the *Personal Picture Finder*. As an advantage of that, the implementation of this netbot took less than one day and needed less than 300 lines of new code.
It is publicly available at *http://finder.dfki.de:7000/cdshop/*.

4.7.1.2 An information gathering agent

Another netbot based on the libraries based on the libraries of the *Personal Picture Finder* is a simple information gathering agent. Its objective is to find and extract the meal plan of the day from the university's canteen. Problems occuring in this task are the frequent change of the URL and the fact that there is not one specific URL to bookmark. The meal plans are provided by one page for every week, every week with another URL.
The agent described here does not expect any information provided by the user. When the servlet is called, it downloads the university's main page, checks it for keywords, and follows matching links. Doing this recursively it finally reaches a page with links for mealplans for several weeks. The agent checks the current date and tries to find a link to a week including the current date. It follows this link, checks the next page for the table with the mealplans, again searches for keywords indicating the days of the week, cuts out and returns the plan for the current day.
Using the classes from the *Personal Picture Finder*, the core of this agent was implemented in 80 lines of program code. It is publicly available at *http://finder.dfki.de:7000/mealplan* since March 99 and works robust so far.

4.7.2 The *Whatsit?* tool

In this section, I will describe a simple tool called *whatsit?* which analyzes different graphics file formats used on the World Wide Web.
Given a list of URLs or local files the *Whatsit?* loads the specified images, checks the signature, in order to find out the format[13] and according to that, it parses the file content, in order to find parameters for width, height and depth of the image. While developing the *Personal Picture Finder*, I revised several newsgroups for Java developers to find out if such a tool is already in existance and publicly available. Coming across a great variety of similar requests, but no answers I decided to publish my implementation in the appendix of this thesis (see appendix A.2.4). Due to its object oriented design, the API can be reused for a variety of other applications, especially to enable agents living on the internet to automatically find out what kind of data[14] they are dealing with[15].
Using the list of files or URLs given in the command line, *whatsit?* loads them sequentially, checks the header signature and parses the files depending on their format in order to find out width, height and depth of the image.

Example:[16]

```
$ java whatsit /home/endres/pictures/ana.gif \
               http://rimbaud/gif/apache_logo.gif \
               Photo.jpg
/home/endres/pictures/ana.gif:       GIF 87a, 546x536x8, 132490 bytes.
http://rimbaud/gif/apache_logo.gif:  GIF 89a, 400x148x8, 23439 bytes.
Photo.jpg:                           JPG 1.01, 493x373x24, 112556 bytes.
$
```

Let us take a closer look at the program code. After importing the packages *java.io* (for I/O operations like accessing files from the local file system), *java.net* (for establishing connections to the WWW) and *endres.graph* (for analyzing graphics file formats[17]), the definition of class whatsit follows. As an application it needs a main-method. This main method checks the command line parameters and sequentially creates a new instance of the whatsit-class for every parameter. The constructor of this class then reads the file via the *readfile()*-method and in case the file is accessible, it executes the *checkFile()*-method. *readfile()* checks if the given parameter describes a URL or the name of a file on the local file system and then opens an according DataInputStream and reads the file in a

[13] Especially when working on heterogenous sources like the WWW one should not assume that every file is what it seems to be according to its extensions. GIFs with a .jpg-extension or vice versa occur and usually are displayed properly on common browsers.

[14] An older version of *whatsit?* recognized some more file formats used on the web, like java classes or postscript documents. The code example described in this thesis is restricted to graphics file formats only.

[15] For example a spider or crawler has to know whether the link he follows leads to another webpage or to another file, like for instance a postscript document.

[16] Java is platform independent. I assume a UNIX shell for this example. One might as well execute it on any other platform.

[17] The package *endres.graph* is explained in detail in the next section.

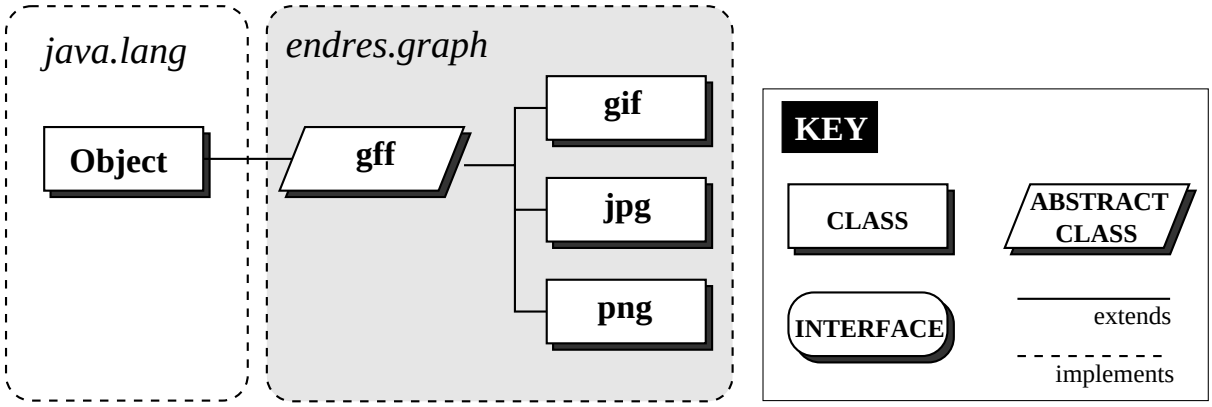

Figure 4.15: Classes of the *endres.graph* package

byte-array.
checkFile() checks the signature of the file, calls the appropriate analyzer from the *endres.graph* package and eventually prints out the result. The package *endres.graph* is crucial here. An overview of this package is given in Figure 4.15. It consists of the abstract class *gff*[18] and subclasses for the formats *gif*[19], *jpg* and *png*.
The class *gff* is abstract and therefore cannot be instanciated. Here, the variables used in every subclass, like the integers for width, height etc. are defined, as well as the desired methods like *getHeight()*, *getVersion()* and several others.
Furthermore, some useful static methods for working on a byte-array representing a file are defined.
In class *gif* the advantage from defining the abstract class *gff* first is obvious. The code became very short. It consists of three methods only: one static method to check the file signature, the constructor which simply calls the constructor of the superclass and the *analyze()*-method. The header of the *gif*-format is static, so a where to get the desired information and how to interpret it is sufficient.
Class *jpg* consists of the same three functions as *gif*, but it is a bit bigger. Unfortunately, the static header consisting of the *SOI*[20] and the *APP0*[21], which contains information about format and version only. The dimension of the picture is denoted in a *SOF*[22]. After identifying the first *SOF* we use the markerlength value to find the following *SOF* until we find one containing the information we need.
The Portable Network Graphics format *png* is similar to *gif*. Again, just the desired information at static positions is checked. In theory, a *png* could contain a picture with up to 4G pixels in height and width. Accessing height and width in this code example works up to 2G pixels only, which should be sufficient for any practical use.
There is only one version of *png*, so the method *getVersion()* will return an empty string.

[18] *gff* is an acronym of graphics file format.
[19] In this example it is not necessary to distinguish GIF87a and GIF89a. There are a lot of differences between these two formats, but the information required here can be found in the same way.
[20] SOI is an acronym for **S**tart **O**f **I**mage.
[21] APP0 is the Application Marker Segment.
[22] SOF is an acronym for **S**tart **O**f **F**rame.

Subclasses for other graphics file formats can be inserted easily in the object model described above by subclassing *gff*.

4.7.3 MultiHttpServer

Internet agents require fast, concurrent access to many web pages. This service should be stable, central, and easy to access independently from the actual implementation language. In this Chapter, I describe the *MultiHttpServer* (MHS), a parallel pull engine implemented in Java with a TCP/IP interface for communication with other programs. A second TCP/IP interface provides information for administration purposes. A simple config-file allows application-oriented tuning of the *MultiHttpServer*. An optional Java-Servlet can remotely start up the *MultiHttpServer* on demand.
Internet Agents and Netbots usually deal with numerous information sources, e.g. webpages. Downloading a page is a time consuming operation. On the other hand it is desirable to collect all information required as fast as possible to produce an acceptable runtime behaviour.
The idea of parallel pull is to save time by performing download tasks in parallel. It has been successfully used in several applications developed in project PAN.
The following sections describe the underlying idea, implementation details, and the specification of the *MultiHttpServer*'s communication protocol.

4.7.3.1 Architecture of the MultiHttpServer

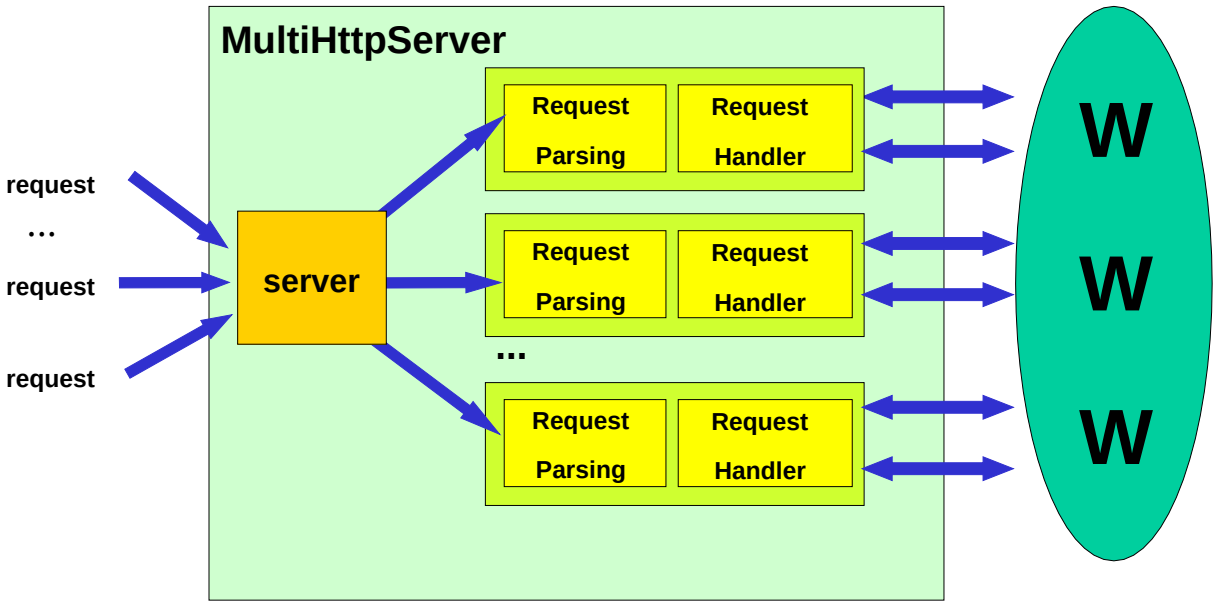

Figure 4.16: Architecture of a single *MultiHttpServer* process

In this section I introduce the architecture of a single *MultiHttpServer* instance. As discussed later it can be useful—depending on the required capacity—to have several

instances of a *MultiHttpServer* running. At the moment I focus on the single server instance shown in Figure 4.16.
It has two core components, a server module and several service modules. The server accepts requests from clients[23] by using a TCP/IP interface. For every client connection a service module is generated. The service module consists of two parts. A *Request Parsing module* controls the dialog protocol with the client and a *Request Handler module* executes the client's request by accessing the World Wide Web (WWW) in parallel.

4.7.3.2 Stability Problems

Parallel execution of requests can be done in two different ways: multithreading or concurrent execution of processes. Both approaches are limited by the operating system. Former versions of the *MultiHttpServer* used multithreading only, which lead to stability problems when the maximum number of threads for one process was reached. Therefore it was necessary to provide a mechanism for creating multiple instances of the multithreaded *MultiHttpServer* and scheduling the queries. Still it is important to keep in mind that no matter how clever the system resources are used they will always be limited.

4.7.3.3 Scheduling and forking processes

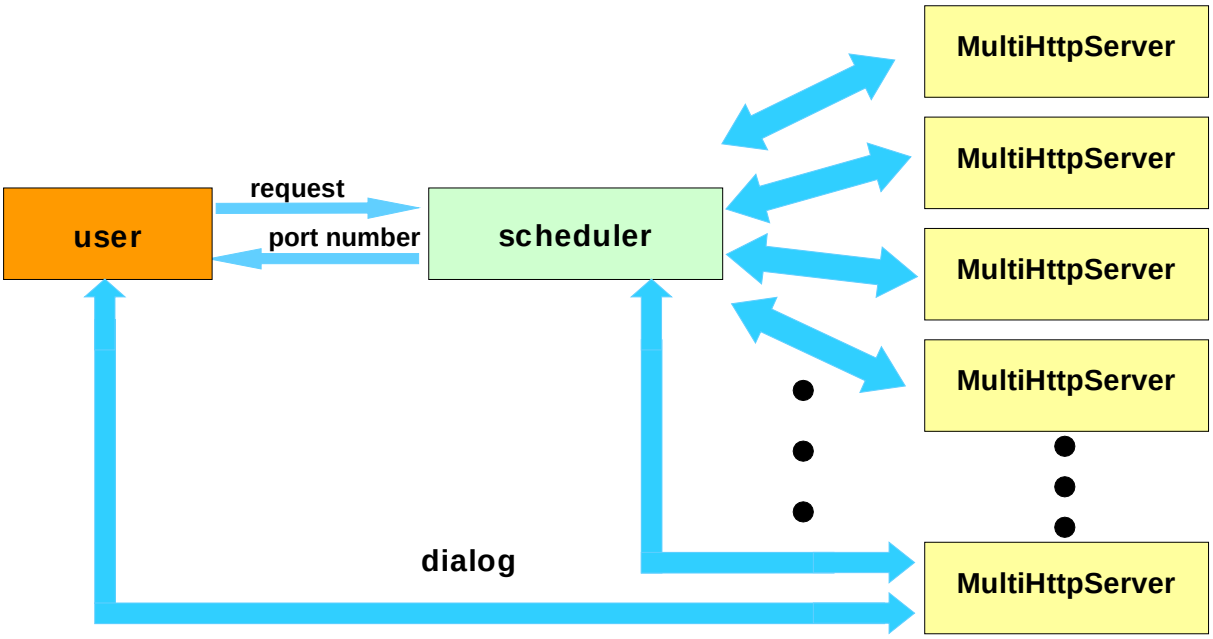

Figure 4.17: Request Scheduling

The current version of the *MultiHttpServer* includes a *scheduler* which supervises all running *MultiHttpServer* instances and schedules user queries. Before actually performing

[23]In this context I do not distinguish between a human user and a client application when using the word *client*.

his queries the user asks the *scheduler* which instance of the *MultiHttpServer* he should use. The *scheduler* answers by telling the port number of the least busy *MultiHttpServer*. The user then connects to this port and goes ahead with placing his query.
The *scheduler* regularly (e.g. every three seconds) collects information from the *MultiHttpServers* about their status (i.e. number of running threads). If for any reason a server does not answer anymore, the *scheduler* kills it and starts a new one. In general, servers are created and killed depending on how busy the whole parallel pull engine is. The information about how this generating and killing of processes should be done is provided by a config-file which the user should set up to his requirements and system resources as described in Section 4.7.3.6.
The architecture of the cooperation between the *scheduler* and the instances of the *MultiHttpServer* is shown in Figure 4.17. In order to establish a communication to the *MultiHttpServer*, the user connects to the *scheduler* via TCP/IP. The *scheduler* checks the capacities of all *MultiHttpServer* running, decides which one should handle the user request, returns the port number, and disconnects the user. The user then connects to the specified port and runs the server protocol as described in the following section.

4.7.3.4 The server protocol

In this section I describe the protocol used for communication between the server and the client. It can be used as a short reference for developers. Sample sessions are shown in the next section.

- mode:[24] <one/all>
  There are two different modes for the server protocol, *one* and *all*. When starting a session the first thing one should do is specify the protocol mode to be used.
  Mode *one* means that only one of the pages requested is interesting. This is useful if there are several URLs of webpages providing equivalent information, e.g. three different weather forecast information services.
  In the *one*-mode, they are all requested, but the result of one of them is sufficient and information from other sources no longer interesting. As soon as one requested page returns, all other requests will be cancelled and all other running threads killed.

  Information already received from other pages will get lost.
  Mode *all* does not kill running threads autonomously. All of the requested pages will be downloaded (unless killed using the *kill*-command). This mode is used if the information on the requested pages is not equivalent.
  If no mode is specified, mode *all* is used by default.

- get: <URL> [<timeout> [+]]
  Request a URL. Timeout (in seconds) and additional parameters (in case the requested URL is a cgi-script) can be specified.
  Examples:

[24] Please not that the colon is part of the command.

- get: http://www.dfki.de/
  requests the webpage of *DFKI*. The information should be obtained no matter how long it takes.
- get http://www.microsoft.com/ 2
  requests the Microsoft webpage. If it takes more than 2 seconds to download, the information is not interesting anymore.
- get: http://www.info.edu/pplsearch.cgi 30 +
  first=Arthur +
  last=Rimbaud +
  email= +
  country=fr
  (the '+' at the end of a line indicates that further parameter specifications follow).
- get: http://www.info.edu/pplsearch.cgi?first=Arthur&last=Rimbaud

Let us take a closer look at the last two examples. Both request the same page. The first one by specifying the parameters separately, one in every line. The parameters are concatenated and written on the URL connection. This is the **POST** method of the HTTP protocol. The second example directly codes the parameters in the request string using the **GET** method. Most CGI scripts handle POST and GET methods in the same way if only a few parameters are specified. One of the main differences is that the POST method can handle much longer parameter inputs, e.g. text.
A detailed description of those methods can be found in the HTTP specification[25].

- authget: <URL> <timeout> <login> <password>
  Get a password protected URL using specified login and password.
- info: <URL>
  Get available information about a requested URL.
- show: <URL>
  Show the output of a requested URL (if available).
- kill: <URL>
  Remove a no longer needed page or cancel downloading it.
- shortinfo
  Get a short overview of the status of all requested pages.
- stack
  Show the URLs of all requested pages.
- stacksize
  Return the amount of requested pages.

[25] See www.w3c.org.

- available
  Return the amount of already received pages.

- rest
  Return the addresses of pages still to be expected, i.e. all pages of the stack besides those reaching timeout or not accessible for any other reason.

- more
  Return the maximum amount of pages still to be expected.

- success
  Return one URL of a received page. If there is no page available yet the return value is 'no'.

- waitsuccess
  Return one URL of a received page. Wait until a value can be returned.

- waitsuccess <timeout>
  Return one URL of a received page. Wait up to <timeout> seconds for a return value.

- status: <URL>
  Show the status of a requested page. Return values are

  - connecting
  - connected
  - receiving
  - received
  - timeout
  - Error <errorcode>

- accesstrend: <URL>
  Show the access trend of a requested page. Return values are

  - increasing
  - constant
  - decreasing

- accessrate: <URL>
  Show the access rate of a requested page in bytes per second.

- poud: <URL>
  Percentage of unfetched data of a requested page. If the size of a page is unknown[26] the return value is set to -1.

[26]The size of a document is an optional header field in version 1.0 of HTTP.

- size: <URL>
  Show the size of a requested page in bytes.

- help [<command>]
  help shows a general help including a list of all available commands, *help* <*command*> explains the usage of <command>.

- version
  Display version and copyright information.

- bye
  Terminate session and close TCP/IP connection.

4.7.3.5 A sample session

This section shows an example of a *MultiHttpServer* session. After obtaining a TCP/IP port number from the *scheduler*, the client connects to a *MultiHttpServer* instance. At the beginning of a session, the server displays a prompt.

```
+-------------------------------------------------------+
| MultiHttpServer version 1.0                  april 99 |
| Christoph Endres, DFKI GmbH   Christoph.Endres@dfki.de |
+-------------------------------------------------------+
Type 'help' for more information
-Ok-
```

The client now requests three webpages.

```
get: http://www.dfki.de/
-Ok-
get: http://www.microsoft.com/ 1
-Ok-
get: http://www.whitehouse.gov/ 10
-Ok-
```

Using the *shortinfo*-command, the client checks the current status of the pages he requested.

```
shortinfo
http://www.microsoft.com/ timeout
http://www.dfki.de/ received
http://www.whitehouse.gov/ received
-Ok-
```

The *success*-command is now used by the client in order to obtain the URL of one of the successfully received pages. The page is displayed using the *show*-command.

```
success
http://www.dfki.de/
-Ok-
show: http://www.dfki.de/
<HTML>
<HEAD>
<TITLE>DFKI - WWW: New Version 13.04.99</TITLE>
</HEAD>
<frameset cols="144,*" border=0 frameborder=0 framespacing=0>
  <frame src="dfkiweb/menu.htm" name="links" noresize>
  <frame src="dfkiweb/start.htm" name="rechts">
</frameset>
</HTML>
-Ok-
```

After obtaining the page, the client removes it from the server and then closes the connection.

```
kill: http://www.dfki.de/
-Ok-
bye
bye
-Ok-
```

4.7.3.6 Configuration

The *MultiHttpServer* is configured by editing a config file. Here is an example:

```
#
# Configfile for the MultiHttpServer
#

# TCP/IP ports for administrator and user
# If no values are specified, the ports 2000 and 2001 are used by default.
Admin: 2001
User:  2000

# minimum and maximum instances of MultiHttpServer running in parallel
# If unspecified, a minimum of 1 and a maximum of 10 is assumed.
Instancemin: 2
Instancemax: 5

# set critical value for the load of a MultiHttpServer instance.
# Unit is pages per server.
# Default value is 50
Load: 25 pps
```

The values to be specified so far are port numbers for administrator and user, a minimal and maximal number of *MultiHttpServer* instances running concurrently, and a critical value for the load of a server. The latter is used as an indicator for creating new instances of *MultiHttpServer* at runtime.

4.7.3.7 Using the administrator port

By connecting to the administrator port some useful information about the status of the system is provided, e.g. the number of *MultiHttpServer*-instances currently running and the number of their threads.

Example:

```
-------------------------------------------------------
MultiHttpServer 1.0
(c) DFKI GmbH 1999
running on serv-200.
You are connecting from serv-200.
Adminport is 2001, userport is 2000.
From 2 up to 5 MHS instances can be created.
Critical load is 5.

2 instances running:
34937:  3 connections, 12 pages.
34956:  5 connections, 17 pages.
-------------------------------------------------------
```

4.7.3.8 Start on demand

Using a servlet it is very easy to start up the *MultiHttpServer* on demand. A Java-Servlet enabled Webserver is necessary. An application wanting to use the *MultiHttpServer* simply creates a http-connection to the *startMHS* servlet, for instance to:
http://www.myserver.edu/startMHS?command=start.
The servlet starts the *MultiHttpServer* (in case it was not running yet) and returns the number of the client port and the administrator port (which usually is not necessary since one usually knows which ports one specified in the config file).

4.7.3.9 Clients

Clients in several programming languages for the *MultiHttpServer* have been implemented. Sample clients in the languages Java, Perl 5, and Eclipse Prolog can be found in the original *MultiHttpServer* documentation [End99].

4.8 Related work

Attempting to compare the *Personal Picture Finder* with similar internet services turns out to be difficult, since there is nothing directly comparable at the moment. In this section I present some services sharing either the metasearch paradigm or the dedication to find pictures with the *Personal Picture Finder*.
The standard tools for finding information on the internet are index search engines that are based on a index (usually a database). Metasearch engines, on the other hand, access

	Metacrawler	Lycos	Altavista	PPF
dynamic	√			√
data evaluation	√	√	√	√
data compression		√	√	√
metasearch paradigm	√			√

Table 4.1: *Personal Picture Finder* and related services.

several index search engines, evaluate the results and present a final result in a unique way. The most famous example of a metasearch engine is Metacrawler[27] [SE96]. The *Personal Picture Finder* is a metasearch engine. Unlike other metasearch engines, its purpose is not only to collect and evaluate data, but also to compress and analyze resulting data. The aspect of searching pictures is shared with other internet services, which are in contrast to the *Personal Picture Finder* static. Two well known examples are:

- AltaVista Photo Finder[28]
  The Altavista Photo Finder is a picture database. Additional to the information of where to find a picture (which is either a URL somewhere on the WWW or a reference to its own database) a short description of the content of the picture is given. The picture descriptions are browsed for the requested keywords. The search results are shown as thumbnails. The user has the possibility to see the original picture, along with the textual description. The system is static, at runtime only the local database is accessed, no requests to the WWW are performed.

- Lycos Bildkatalog[29]
  The Lycos Bildkatalog[30] is a static service as well. Like with the Altavista Photo Finder, at runtime a database is accessed. The classification of the pictures does not include a textual description of the picture. The database does not include own pictures, but provides references to pictures on the WWW instead. The user can submit own URLs to be included in the database. User-submitted pictures are a great advantage when dealing in a dynamic environment like the WWW, on the other hand it is based on trust in every user providing information and can lead to incorrect data. For example when looking up *Boris Becker* (a german tennis player), one gets a huge variety of easter eggs as result, which does not match the query at all.

Static information look up provides fast results, but can never be up to date in a dynamic environment. Dynamic search for information is a bit slower, but it meets the requirements of the dynamic environment. By using the metaseach paradigm, the *Personal Picture Finder* accesses the static databases mentioned above as well as dynamic

[27] www.metacrawler.com
[28] image.altavista.com
[29] www.de.lycos.de
[30] The word Bildkatalog is german and means Picture Catalogue

search in pages obtained from *Metacrawler* and *Ahoy!*. Using Metacrawler makes the *Personal Picture Finder* a meta-metasearch engine. It combines fast static information with dynamic information lookup. A survey on these properties is shown in Table 4.1.

Chapter 5

Conclusion

5.1 Summary

In this thesis I presented the *Personal Picture Finder*, a netbot with dedication to find portrait photos on the web. I implemented the *Personal Picture Finder* system as practical part of my thesis. The system is available online at *http://finder.dfki.de:7000/* since the first public presentation at the 10th anniversary of DFKI GmbH in July 1998 and successfully used ever since then. The press, TV, several presentations and a national Internet-conference made it popular. More than 500 requests are processed every month. The technology developed for the *Personal Picture Finder* is a powerful tool for building other netbots. The MultiHttpServer, a by-product of the *Personal Picture Finder*, and the Parallel Pull technology became an essential module for every major product in project PAN. Due to the object-oriented paradigm of the Java programming language a lot of reusable code was produced.
An experimental version of the *Personal Picture Finder* increases the *quality of service*. Furthermore it provides interfaces to other filters, e.g. the *Bitmap Information Tool* and additional information sources. An extensible, pattern-based URL generator is included as well. Using HyQL scripts, the *Personal Picture Finder* can be adapted easily in the dynamic environment of the World Wide Web using the *Programming by Demonstration* dialog developed in PAN. Besides providing a multitude of possible applications, the *Personal Picture Finder* is a prototype of an internet agent. The basic internet agent technology developed here can be used for other applications as well.

5.2 Outlook

Developing a system like the *Personal Picture Finder* was an interesting and inspiring work. It lead to a lot of nice ideas for new netbots or features that could be added. Some could be added, like the *minifinder* or the *URL generator*, some others should be mentioned here as an outlook on future work.
A mail interface could be included in the experimental version. The user then simply sends a mail with his request to the *Personal Picture Finder* and gets the results as at-

tachment of the reply some hours later.
Face recognition algorithms could be added too. Some programs are freely available on the internet and could be added to the script easily.
The new video file format *MPEG 7* will set new standards for encoding comments in a video stream. These features could be used to look up video clips containing the requested person, maybe those clips could be post processed to obtain single matching pictures out of the video.
The intention of *Personal Picture Finder* was to build a prototype. Other applications could be assembled using its API to look up any information, like email addresses, documents, sound files, or even banners, email icons, backgrounds, etc.
At DFKI GmbH, the reuse of technology developed for the *Personal Picture Finder* is discussed in at least two new projects. One of them is concerned with looking up documents, the other one with real estate business.
Surely a lot of other applications will emerge here, the limitations are set by the information provided on the World Wide Web and the imagination of the developer only.

Appendix A

Reusable Code Examples

An important part of the *Personal Picture Finder* implementation was the access of graphics files over the web and the analyze of their parameters. The tool *whatsit?* described in section 4.7.2 performs this task. The source code of this tool is added here. Due to is modular structure, it could be used in other applications as well.

A.1 *Whatsit?* - A graphics file format analyzer

```
import java.io.*;
import java.net.*;
import endres.graph.*;

public class whatsit {

  public static void main(String args[]) {
    for (int i=0; i<args.length; i++) {
      new whatsit(args[i]);
    }
  }

  String file;
  boolean unavailable = false;
  byte content[];

  public whatsit(String file) {
    this.file = file;
    readfile();
    if (!unavailable) {
      checkFile();
    } else {
      System.err.println(file+" is not available!");
    }
  }

  void readfile() {
    try {
      DataInputStream in;
```

```
      int filesize;
      if (file.indexOf(":/")>0) {
        URL url = new URL(file);
        URLConnection urlc = url.openConnection();
        in = new DataInputStream(urlc.getInputStream());
        filesize = urlc.getContentLength();
      } else {
        in = new DataInputStream(new FileInputStream(file));
        filesize = in.available();
      }
      if (in != null) {
        content = new byte[filesize];
        in.readFully(content);
        in.close();
      }
    } catch (Exception e) {
      unavailable = true;
    }
  }

  void checkFile() {
    gff g = null;
    if (gif.check(content)) g = new gif(content);
    if (jpg.check(content)) g = new jpg(content);
    if (png.check(content)) g = new png(content);
    System.out.println(file+":\t"+
                       g.getFormat()+" "+
                       g.getVersion()+", "+
                       g.getWidth()+"x"+
                       g.getHeight()+"x"+
                       g.getDepth()+", "+
                       g.getLength()+" bytes.");
  }

}
```

A.2 The endres.graph API

After the main application of the *whatsit?*-tool described in the previous section, the source code from the package *endres.graph* follows here, including analyzers for the formats GIF, JPEG and PNG.

A.2.1 endres.graph.gff

```
package endres.graph;

public abstract class gff {

  byte content[];
  int width = -1;
```

```
int height = -1;
int depth = -1;
int length;
String format = "";
String version = "";

public gff(byte content[]) {
  this.content = content;
  fileLength();
  analyze();
}

public abstract void analyze();

public void fileLength() {
  length = content.length;
}

public int getWidth() {
  return width;
}

public int getHeight() {
  return height;
}

public int getDepth() {
  return depth;
}

public int getLength() {
  return length;
}

public String getFormat() {
  return format;
}

public String getVersion() {
  return version;
}

static int bytetoint(byte b) {
   int i = (int)b;
   if (i<0) i+=256;
   return i;
}

static String getContentString(byte content[], int start, int length) {
  StringBuffer accu = new StringBuffer();
  for(int i = start; i<(start+length) ; i++) {
    accu = accu.append((char)content[i]);
  }
```

```
    return accu.toString();
  }

  static int getInt(byte content[], int start, int length) {
    int accu = 0;
    for(int i = start; i<(start+length) ; i++) {
      accu = 256*accu+bytetoint(content[i]);
    }
    return accu;
  }

  static int getWord(byte content[], int start) {
    return bytetoint(content[start])+(256*bytetoint(content[start+1]));
  }

}
```

A.2.2 endres.graph.gif

```
package endres.graph;

public class gif extends gff {

  public static boolean check(byte content[]) {
    return getContentString(content,0,3).equals("GIF");
  }

  public gif(byte content[]) {
    super(content);
  }

  public void analyze() {
    width = getWord(content,6);
    height= getWord(content,8);
    depth = (bytetoint(content[10]) & (int)7)+1;
    format = "GIF";
    version = getContentString(content, 3, 3);
  }

}
```

A.2.3 endres.graph.jpg

```
package endres.graph;

public class jpg extends gff {

  public static boolean check(byte content[]) {
    return getContentString(content,6,4).equals("JFIF");
  }
```

```
  public jpg(byte content[]) {
    super(content);
  }

  public void analyze() {
    format = "JPG";
    // check version
    int ver = getInt(content,12,1);
    if (ver < 10) {
      version = getInt(content,11,1)+".0"+ver;
    } else {
      version = getInt(content,11,1)+"."+ver;
    }
    // search first 'FF'-marker
    int i = 10;
    while((bytetoint(content[i])!=255) && (i<length-7)) {
      i++;
    }
    // use markerlength indicator to find next marker
    // until required data is found
    boolean found = false;
    while(!found && (i<length-10)) {
      int markerlen = getInt(content,i+2,2);
      int identifier = bytetoint(content[i+1]);
      if ((identifier==0xC0)||
          (identifier==0xC1)||
          (identifier==0xC2)||
          (identifier==0xC0)||
          (identifier==0xCA)) {
          found = true;
          height=getInt(content,i+5,2);
          width=getInt(content,i+7,2);
          depth=8*bytetoint(content[i+9]);
      } else {
        i+=(markerlen+2);
      }
    }
  }

}
```

A.2.4 endres.graph.png

```
package endres.graph;

public class png extends gff {

  public static boolean check(byte content[]) {
    return ((bytetoint(content[0])==0x89) &&
            (bytetoint(content[1])==0x50) &&
            (bytetoint(content[2])==0x4E) &&
            (bytetoint(content[3])==0x47) &&
```

```
            (bytetoint(content[4])==0x0D) &&
            (bytetoint(content[5])==0x0A) &&
            (bytetoint(content[6])==0x1A) &&
            (bytetoint(content[7])==0x0A));
  }

  public png(byte content[]) {
    super(content);
  }

  public void analyze() {
    width = getInt(content,16,4);
    height = getInt(content,20,4);
    depth = getInt(content,24,1);
    format = "PNG";
    // PNG does not have a version identifier.
  }

}
```

Appendix B

User Manual

B.1 Usage

The public version of the *Personal Picture Finder* is easy to use. It works with any Java and Javascript enabled browser, like *Netscape Communicator* version 4.0 or higher or Microsoft Internet Explorer version 4.0 or higher. It was tested under Solaris, Linux, Macintosh, and Windows.
Before downloading the webpage of the *Personal Picture Finder*, Java and Javascript should be enabled.
In case of the *Netscape Communicator*, this can be done in menu *edit*, menuitem *preferences*. A new window is popping up with a list entitled Category. The listitem *Advanced* contains a menu with checkboxes *Enable Java* and *Enable Javascript*. Both must be checked.
For the *Microsoft Internet Explorer*, this is similar. In menu *View*, menuitem *Internet Options* pops up a new window. Here, in menu *Advanced*, a list of checkboxes can be found. Under the keyword *Java VM* are three checkboxes, labelled *Java logging enabled*, *Java JIT compiler enabled* and *Java console enabled*. All of them have to be checked (default is the second out of these three only). A restart of the *Microsoft Internet Explorer* is required.
The next step is to download page *http://finder.dfki.de:7000/*. On the left side of the page, the user interface appears (see Figure B.1). You can enter first and last name of the required person, then click *GO* and wait for results to appear on the right side of the page. Alternatively, you can choose the *minifinder* by clicking on the link *minifinder*. The same user interface pops up in a separate, small window. The main window of the browser can be used as usual now. As soon as results for the query return, a new window with those results pops up.

Along with the results, checkboxes with *thumbs up* and *thumbs down* icon appear. In case you want to leave a feedback about the picture, check one of these boxes for every picture. The feedback should indicate if the picture matches the query, not your opinion whether this person looks good on this picture or not. To find out about the source of a displayed picture, move the mouse pointer over it. The name of the page where it was found is displayed then. After clicking on the picture with the left mouse button, a new

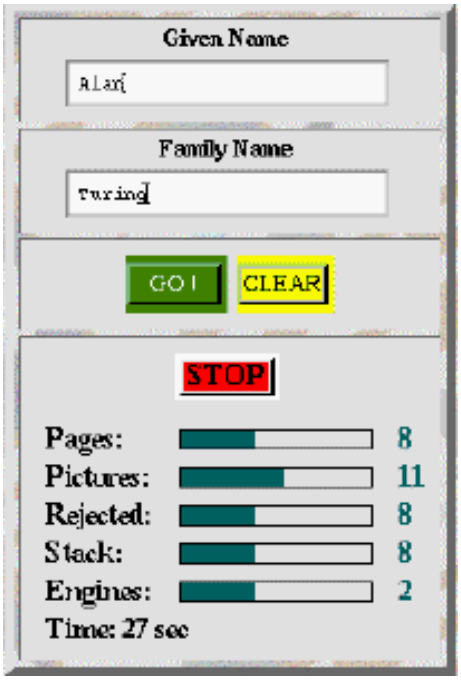

Figure B.1: User interface on the webpage

browser window with this page will be opened. In case you want to use this picture, please contact the webmaster of this page. The *Personal Picture Finder* just looks up pictures, it does not provide them for further usage. The pictures presented as result are usually not copyright-free.
During the search, some information is shown on the applets display. Their meaning is explained in detail in Chapter 4.
After finishing your request, please click the *STOP* button. Before starting a new request, please click the *CLEAR* button. If the search is not terminated, a useless process is idling on the server and has to be cleaned up (this works automatically with a script cleaning up the server every 10 minutes, but can be avoided anyway by the user).

B.2 Fixing problems

When using the *Personal Picture Finder*, some errors may occur. Some known problems are:

- *Firewalls*:
  Since the *Personal Picture Finder*'s webserver uses port 7000 instead of the default port 80, some *firewalls* won't download this page. Even those downloading the page will most likely not allow a TCP/IP connection to the *Personal Picture Finder* application running on serverside.
  In case you want to use the *Personal Picture Finder* anyway, please choose another

computer or ask your system administrator to reconfigure the *firewall* with special rights for the computer with IP-address *134.96.188.66*, alias *serv-200.dfki.uni-sb.de* or *finder.dfki.de*.

- *Javascript error*:
  Sometimes Javascript reports an error which says, that a function is undefined. This problem occurs, when the browser window is resized while the page is loaded. A reload of the page or the frame with the applet usually solves this problem.

- *No applet*:
  If the applet can not be loaded, please make sure, that your browser was installed completely and Java is enabled. On *Microsoft* operating systems, you can also try a restart of your browser.

- *Results disappear*:
  When using the *minifinder*, the result window should not be resized. The page contained in it is not cached, and hence can not redisplayed. When opening the result window, the parameter *resizable* is set to *no*. On some platforms, *netscape* simply ignores it and pops up a resizable window.

- *No scrollbar*:
  Sometimes no scrollbar appears in the result window or frame. In this case, you can try to click on the background of this window or frame and start scrolling down with the cursor keys. Usually the scrollbar appears then.

Please keep in mind that these problems are not bugs in the implementation of the *Personal Picture Finder*, but general problems every platform and browser independent service has to deal with.
For any question not answered here, please contact *picturefinder@dfki.de*. When reporting an error, please specify the problem, and include information about your operating system and your browserversion.

Appendix C

Access statistics

This Section contains the usage statistics mentioned in Section 4.6. The diagrams represent the access of the *Personal Picture Finder*, sorted by months, days, time, and top-level domains.

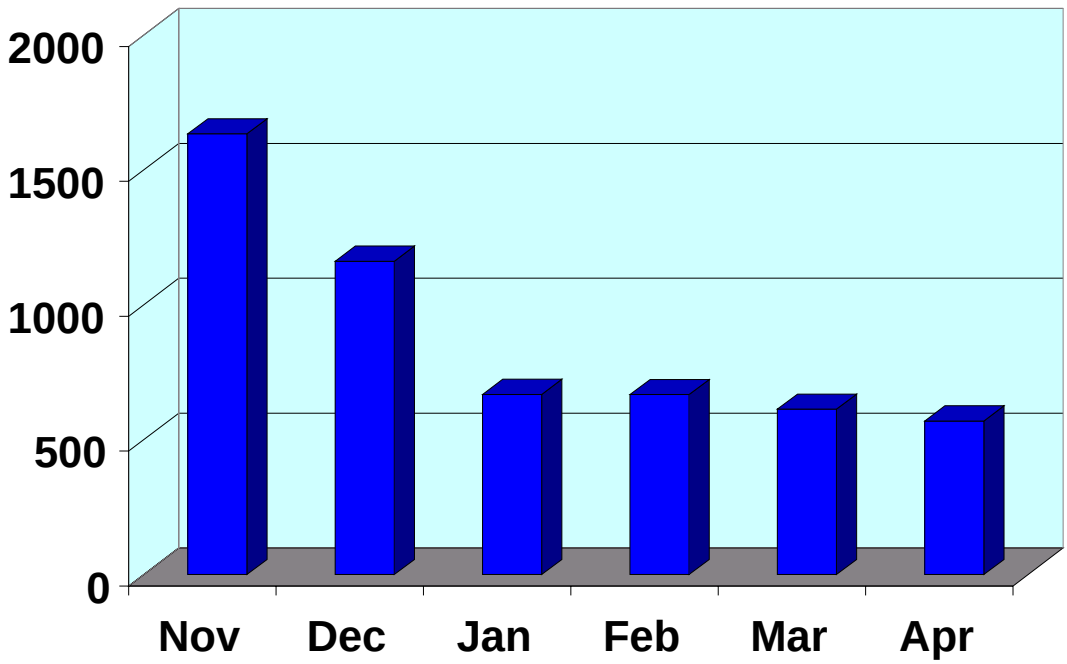

Figure C.1: Access (sorted by months)

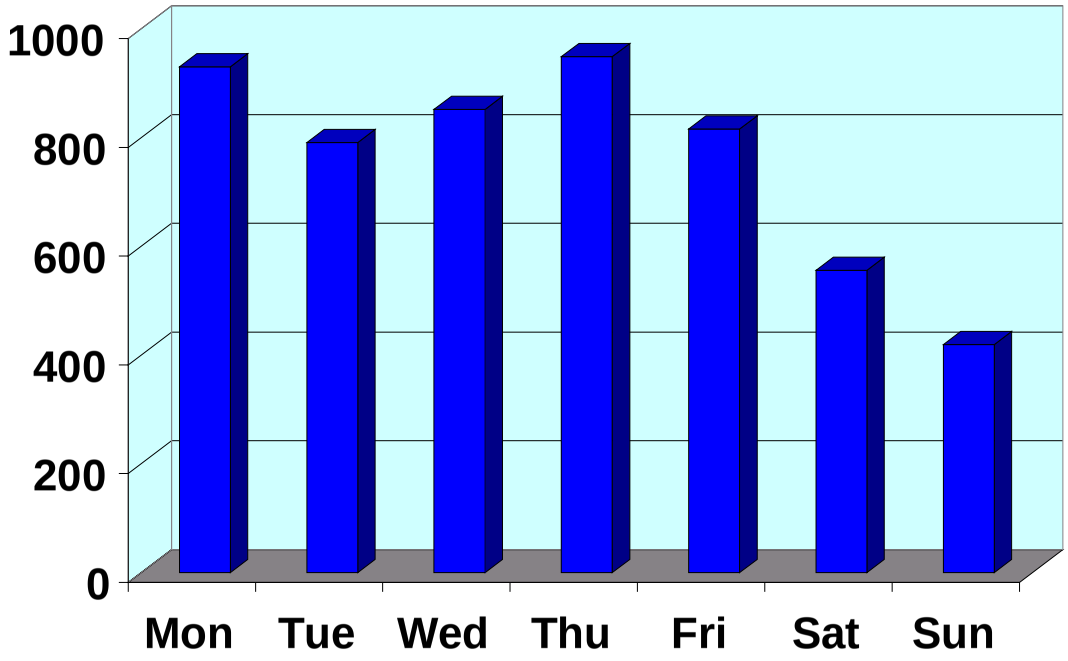

Figure C.2: Access (sorted by days)

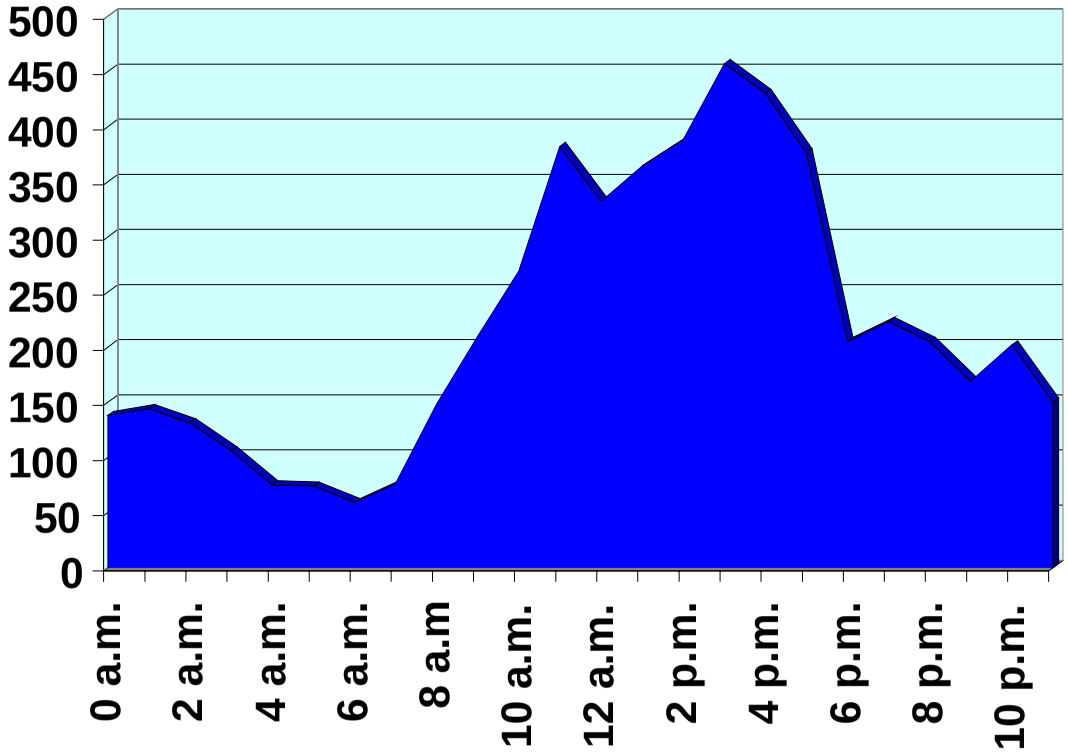

Figure C.3: Access (sorted by time)

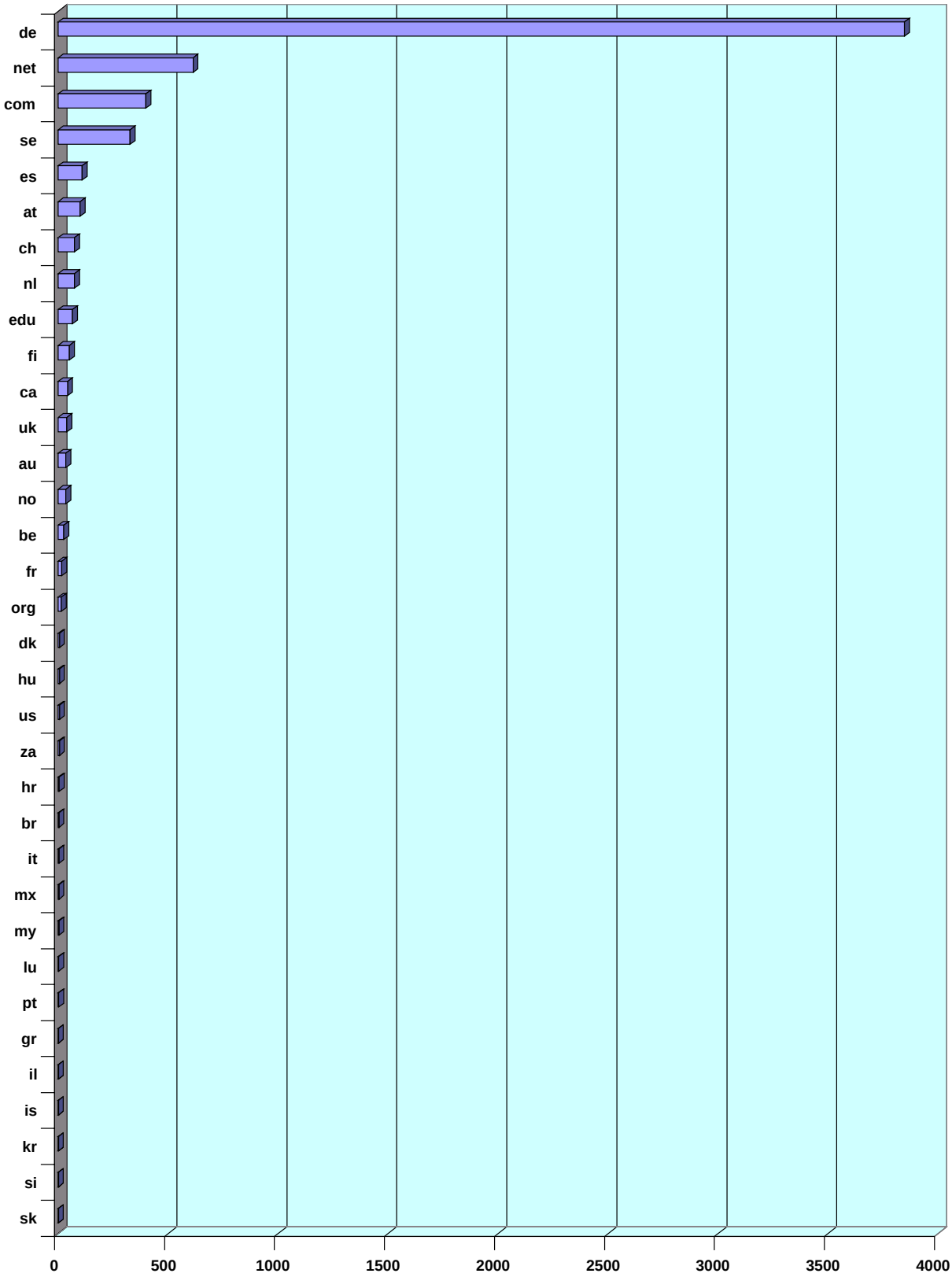

Figure C.4: Access (sorted by top-level domains)

Appendix D

Additional Information

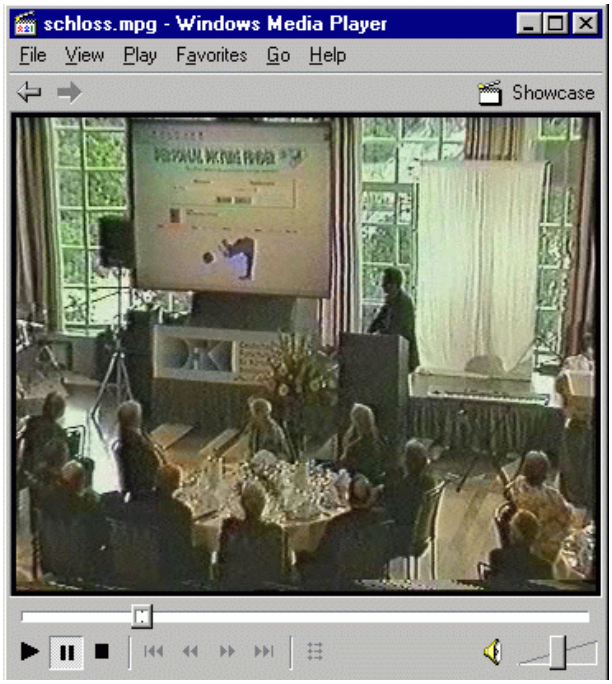

Figure D.1: Video clip: Presentation at the castle

Some additional information in this context can not be represented in form of a printed document. The *CD ROM* included with this thesis contains:

- Some samples of search results of the experimental version of the *Personal Picture Finder*.
- This document as postscript and pdf-file and some of the articles quoted here.
- The source code of the *whatsit?* tool.
- A video clip of the first public presentation of the *Personal Picture Finder* in July 1998 at the castle in Saarbrücken.

- Newspaper articles about the *Personal Picture Finder* and the 10th anniversary of DFKI GmbH.
- Video clips of the news programs *heute* and *Aktueller Bericht* (excerpts) showing the *Personal Picture Finder*.
- Photos of some public presentations of the *Personal Picture Finder*.
- The slides I used for my speech at the *Informatikforum 98*.

Bibliography

[ARM98] Elisabeth André, Thomas Rist, and Jochen Müller. Integrating Reactive and Scripted Behaviors in a Life-Like Presentation Agent. In *Proceedings of the Agents98 Conference*, pages 261–268, 1998.

[Asi50] Isaac Asimov. *I, ROBOT*. Gnome Press, 1950.

[BD98] Mathias Bauer and Dietmar Dengler. TrIAs: An Architecture for Trainable Information Assistants. In *Proceedings of the Agents98 Workshop on "Personal Information Assistants"*, 1998.

[BD99a] Mathias Bauer and Dietmar Dengler. InfoBeans - Configuration of Personalized Information Assistants. In *Proceedings of the 1999 Conference on Intelligent User Interfaces*, 1999.

[BD99b] Mathias Bauer and Dietmar Dengler. TrIAs: Trainable Information Assistants for Cooperative Problem Solving. In *Proceedings of the Agents99 Workshop on "Personal Information Assistants"*, 1999. to appear.

[BLCG92] Tim Berners-Lee, Robert Cailliau, and Jean-Francois Groff. The World-Wide Web Computer Networks and ISDN Systems. 25:454–459, 1992.

[Bor97] Günther Born. *Referenzhandbuch Dateiformate*. Addison-Wesley, fifth edition, 1997.

[Com87] CompuServe Incorporated. *Graphics Interchange Format*, 1987.

[Com89] CompuServe Incorporated. *Graphics Interchange Format*, 1989.

[Den99a] Dietmar Dengler. HyQL - A Tutorial. 1999. to appear.

[Den99b] Dietmar Dengler. The HyQL Language Specification. 1999. to appear.

[EMW99] Christoph Endres, Markus Meyer, and Wolfgang Wahlster. Personal Picture Finder: Ein Internet-Agent zur wissensbasierten Suche nach Personenphotos. In *In: Vogt, F. (ed.): Online'99, Congressband VI*, pages 301–315. Velbert: Online-Verlag, 1999.

[End99] Christoph Endres. The MultiHttpServer - A Parallel Pull Engine. Technical Report TM-99-04, Deutsches Forschungszentrum für Künstliche Intelligenz, 1999.

[EW95] Oren Etzioni and Daniel Weld. Intelligent Agents on the Internet: Fact, Fiction and Forecast. *IEEE Expert*, pages 44–49, 1995.

[Fla97] David Flanagan. *Java in a nutshell.* O'Reilly, second edition, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Haa99] Jens Haase. BIT – Bitmap Information Tool. 1999. to appear.

[HCF97] Graham Hamilton, Rick Cattell, and Maydene Fisher. *JDBC Database Access with Java.* Addison Wesley, 1997.

[KL95] David Kay and John Levine. *Graphics File Formats.* Windcrest/McGraw-Hill, second edition, 1995.

[Koh95] Ronny Kohavi. *MLC++ Tutorial.* Stanford university, 1995.

[Kur90] Raymond Kurzweil. *The Age Of Intelligent Machines.* Massachusetts Institute of Technology, 1990.

[KW89] Alfred Kobsa and Wolfgang Wahlster, editors. *User Models in Dialog Systems.* Springer Verlag, 1989.

[Mit97] Tom Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[Mül99] Jochen Müller. *Ein planbasierter Präsentationsagent.* PhD thesis, Deutsches Forschungszentrum für Künstliche Intelligenz, 1999.

[Mv96] James D. Murray and William vanRyper. *Encyclopedia of Graphics File Formats.* O'Reilly & Associates Inc., second edition, 1996.

[MW98] Mark Maybury and Wolfgang Wahlster, editors. *Readings in Intelligent User Interfaces.* Morgan Kaufmann Publishers, Inc., 1998.

[O'N94] Patrick O'Neil. *Database - Principles, Programming, Performance.* Morgan Kaufmann Publishers, Inc., 1994.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, (1:81-106), 1986.

[SE96] Erik Selberg and Oren Etzioni. The MetaCrawler Architecture for Ressource Aggregation on the Web. 1996.

[WK91] Sholom Weiss and Casimir Kulikowski. *Computer Systems That Learn.* Morgan Kaufmann Publishers, Inc., 1991.

Index