Vrije Universiteit Brussel

FACULTY OF SCIENCES AND BIO-ENGINEERING
SCIENCES

An RSL-based Associative Filesystem

Gregory Cardone

Promotor: Beat Signer

Thesis submitted for obtaining the degree of Master in Applied Computer Science

Academic Year 2009-2010

Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN EN BIO-INGENIEURSWETENSCHAPPEN

An RSL-based Associative Filesystem

Gregory Cardone

Promotor: Beat Signer

Eindwerk ingediend voor het behalen van de graad van Master in de Toegepaste Informatica

Academiejaar 2009-2010

Abstract

The persistent storage capacity of personal computers in the form of disk space is growing every year, and we are already dealing with terabytes of data. People use their computer not only for work, but also for entertainment. However, since the number of files is increasing, problems of organizing and retrieving data are arising. With current filesystems, it is possible to organize files in folders and to define complex hierarchies of folders. But over the time and with an increasing number of files, an efficient organization becomes more complex to realize. The general issue is that users can have difficulties to remember in which folders their data is located and how it is organized. This is because files can be located in deep folder hierarchies. Search functionality do not help us in most of these situations since files often have non-trivial names. Therefore, we propose a mechanism to classify files and folders within multiple folders, called *multiple classification*, in order to organize our data and access it from multiple folders, and to avoid creating deep folder hierarchies.
Furthermore, in existing filesystems we do not see whether files are semantically related to each other (for example a photo and a video that have been created at the same geographical place and in the same context). Also, we encounter sometimes situations where it is difficult to create a correct organization of our files with folders. We introduce the concept of *semantic links*, which enables us to semantically link data in order to understand how a set of data is related, and make it possible to flexibly organize our data (i.e. not only organizing files and folders within folders).
Moreover, we often create "temporary" files that become irrelevant after some time. For example, if we want to insert a part of an image in a document, we may have to create a cropped version of the image before inserting it into our document. In many cases, we forget to later remove those "temporary" files. Thus, they pollute our collection of files and waste unnecessary space on our hard disk drive. To solve the problem, we introduce the concept of *content recycling*, which consists of reusing data by pointing to (parts of) the content of files and render it within other files.
In addition, current filesystems add their own predefined metadata to files or folders, but do not allow us to define our own metadata. Furthermore, search functions can only be based on this limited system-defined metadata. Therefore, we propose

to let users create user-defined metadata, called *properties*, on any data such as files or partial content of files.
Last but not least, we introduce the concept of *content traveling* and explain how it is useful when we want to consult the content of different information sources in a non-linear way. We also explain the concept of *content outsorters* in order to adapt the view (or environment) of our data. Finally, we show how we can grant access rights to our data, and how we can combine it with content outsorters in order to adapt the presentation of data.

To achieve these goals, we have extended the Resource-Selector-Link (RSL) model, a metamodel that has the purpose to link and compose data, with an extension called the RSL-based Associative Filesystem (RBAF) model. We have implemented an initial prototype of the RSL-based Associative Filesystem as a proof of concept.

Acknowledgments

I would like to thank my promotor, Professor Beat Signer, for supporting me throughout the period of my thesis. He made an effort to review my thesis several times. His comments on my research results, and his advices were invaluable.
I also would like to thank my parents for giving me the opportunity to study and to obtain my Master's degree at the Vrije Universiteit Brussel.
Finally, I would like to thank somebody special from whom I got the physical and mental strength to persevere until the finish.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Since the 60s, computers have evolved fast and many filesystems have been developed in order to store and retrieve data on different types of data storage. Those filesystems have evolved as well, and gradually introduced new features. While the first purpose of a filesystem was to store data and retrieve them from a small data storage, the main changes where the support for a higher data storage size, the introduction of folders in order to organize files, and more system-predefined metadata attached to files and folders. Extra features such as shortcuts and user policies have been introduced, and search functions helped users to speed up the search of files and folders. Nowadays, files are not only a stream of characters, but can be any type of data we know today: photos, videos, e-mails, documents and other data for personal usage are such examples. With the help of graphical user interfaces to represent folders, we can organize this data in an intuitive way.
Unfortunately, the evolution has stopped there and we are still using filesystems with the same features. Because the storage size of data storage does not cease to grow, we are starting to face problems in organizing and retrieving data in an efficient way.
First, folder hierarchies can become deep and complex, and we often have to browse folders in a recursive way to find a file. We can use search functionality to find files, but it often happens that we did not give a proper filename or gave a shortened filename during the creation of our files, which does not provide much information about the content. Therefore, in many cases the search function does not help and we are forced to manually search the desired file. The reason why we have to recursively browse folders to find a file is because a file is uniquely located in one folder. Also a folder can only be uniquely located in another folder. We can avoid to put files in deep hierarchies of folders and access them from different folders by classifying our files within multiple folders. A file is thus no longer located in a single folder, but in multiple folders at the same time.

Second, we cannot see whether files are semantically related to each other. For example, a photo and a video that have been created at the same geographical place and in the same context. In our brain, all information is stored and associated with each other. This is why we can remember something when we see a visual instance or when we think about a related event. Suppose that one tries to remember a particular person. One probably can remember the person by its hairstyle, or because the person told a funny joke. We can try to mimic the brain by introducing the concept of semantically linking files and folders. Users can on one hand make use of this concept in order to understand how a set of data is related, and on the other hand create a flexible organization of their files and folders. This means that we are not limited to organize our data in folders, but we can also make it clear that data belongs together via semantic associations. For example, suppose that a user has a folder containing biology documents and a folder containing mathematics documents. While reading biology documents, they know that regularly consulting mathematics documents is necessary in order to understand calculations mentioned in the biology documents. Moving the folder of mathematics within the folder of biology, or the other way around is not logical (because biology and mathematics are two different domains). Therefore, they can use semantic links in order to semantically relate the two folders. With the word "flexible", we mean that we are not organizing data in a direct way (such as we do when creating subfolders within folders). We know that a collection of data "has something to do with" another collection of data (i.e. just as they are grouped within folders).

Third, we cannot reuse (parts of) the content of files (for example a part of an image) and integrate them into other files. Reusing (parts of) the content of files has the advantage that we avoid creating unnecessary additional or replicate information on our disk drive. This can be solved with a concept invented by Ted Nelson called *transclusions*[10], which are structural relationships between (parts of) files. The idea is that a file, for example a document, does not contain an element such as an image or a piece of text, but to compose its content by linking to (a part of) another document. When opening the document, the linked elements are rendered in the document itself, and not physically merged (i.e. copy and paste) with the document. The concept of content recycling is inspired by Ted Nelson's idea, and has the purpose to create and compose data by reusing existing data on the filesystem.

Fourth, finding a file is only possible via its filename, via some system-predefined properties or via the content of text-based files. Other files like videos and photos cannot be retrieved by entering keywords of a certain sequence of a video or parts of a photo. Also files and folders contain limited information, beside system-predefined properties (such as the creation date), about their content, which makes it difficult

to understand what data they contain. It would be nice if one could have more freedom in augmenting data with supplementary metadata by, for example, adding properties in the form of keywords as a description.
Fifth, suppose that we consult different websites one after the other in our web browser. We use tabs in order to switch between those webpages. When suspending the search, copying and pasting the content of webpages in a separate document is not necessary. Before closing the web browser, we can save the session together with those tabs in order to reopen them next time.
Similarly, when we regularly consult the same content in different files in an ordered way, we probably do not want to create a separate document and copy and paste pieces of their content. With the concept of content traveling, we can let users create navigational links between files (or folders) in order to jump from one piece of content to another piece of information.
Sixth, we can let users adapt the presentation or environment of their files and folders (i.e. how it looks like or what it contains) with different criteria such as properties. For example, a user might want to view only files with properties *banking transactions* and *Dexia* (which is a Belgian bank). He can define a set of rules that must match a set of criteria and apply those rules in order to only view the desired files. We can compare this with the *Thunderbird* mailclient, which uses virtual folders. In virtual folders, a user can define a set of rules such as for which e-mail account they want to apply those rules, and what e-mail to view that match criteria such as 'job'. The virtual folder will be viewed as a regular folder, and when the user clicks on it, it will contain only e-mails that match the criteria. We propose to integrate a similar feature, which we call *content outsorters*, in a filesystem where users can define their rules for a set of files and apply them when they want.
Last but not least, we can grant access rights to our data in order to define what can be viewed/accessed or not. For example, who can view/access a semantic link to a folder. The user can configure access rights in two ways: via *policies*, which are the ownerships and access rights to data, and *settings*, which define parameters on what the RBAF system or applications should be considered before accessing data. We can also combine content outsorters and access rights in order to adapt the environment depending on the user, policies and settings.

The goal of this master thesis was to investigate an alternative way to create, organize and retrieve files with the introduction of a filesystem called RSL-based Associative Filesystem (RBAF). A prototype has been realized based on the RBAF model, which is an extension of the Resource-Selector-Link (RSL) metamodel.

In Chapter 2 we discuss the evolution of filesystems, related work and how people use their brain to organize and retrieve information in their memory. Chapter 3 discusses two scenarios, each addressing specific problems in dealing with current filesystems. Chapter 4 introduces the main concepts and architecture of the RSL-based Associative Filesystem in combination with illustrative examples. Chapter 5 provides an overview of the implementation of the prototype of the RSL-based Associative Filesystem. Furthermore, Chapter 6 introduces 2 applications that have been developed to demonstrate the functionality of the RBAF system. Finally, some conclusions and comments about future work are provided in Chapter 7.

Chapter 2

Background

2.1 The Memex

In 1945, Vannevar Bush [2] had a dream to help scientists to store, organize and retrieve all kind of information they collected during their work in a collective way by allowing every scientist to collaborate together. One of the main ideas was to record the path of thoughts that scientists used during their research. Those paths can be used to navigate from one page to the other similar to hyperlinks on the Web to navigate from one webpage to another. The paths are called "trails". He came with this idea because of the problems of classifying and retrieving information.

> "When data of any sort are placed in storage, they are filed alphabetically or numerically, and information is found (when it is) by tracing it down from subclass to subclass. It can be in only one place, unless duplicates are used; one has to have rules as to which path will locate it, and the rules are cumbersome. Having found one item, moreover, one has to emerge from the system and re-enter on a new path. The human mind does not work that way. It operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain".([2] p.43).

Therefore, he wanted to mimic human associative thoughts with trails instead of via hierarchical classification. Nowadays, we use the hierarchical classification system with folders in an abundant way, and this will probably not change soon. However, we can use the idea of trails in our system for multiple applications. One application can be that our search behavior is tracked and saved as a navigational linkage between folders and files. Next time, instead of browsing the previous path by clicking recursively on subfolders and searching for the next subfolder to click on,

we can follow the navigational linkage and browse the items one by one until we find the file we are looking for. It is possible that we are looking for another file that is on the same navigation linkage.
Also, every scientist had to be able to annotate the information (in the visual form of encyclopedia paper) he was consulting and save it. Bush introduced a machine called the *memex*. The memex looks like a desktop as illustrated in Figure 2.1a. It has at the bottom front side drawers were scientists can put persistent storage units in the form of microfilms. In the top middle we find two displays which have almost the same size of an A4 paper format as we can find it in most encyclopedias. Those two displays are used to view the information in the form of pages, stored on microfilms which will be accessed by the memex and projected to the two displays. At the bottom of each display we find a switch which can be moved from left to right and vice versa. These switches are used to browse the pages that are stored on the microfilms.

(a) The Memex in its entirety **(b)** The "Yellow Pages"

Figure 2.1: The Memex

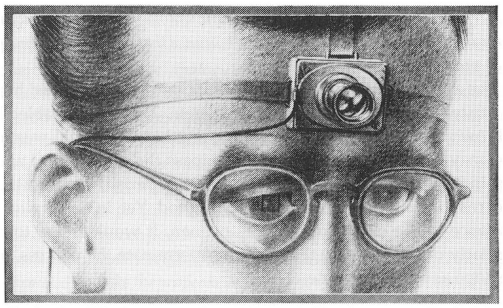
On the top right hand side of the memex we find four other tools. The first tool is a mechanical display that can maximally show seven letters or numbers. The letters and numbers are used to compose a unique code to identify a page. The second tool is a sort of quick-viewer, like the Yellow Pages we use to find the telephone number of a person, where the desired category of information is associated with a code. The third tool is a keyboard used to input the code of the book. Whereas the fourth tool is a pen to write on the two displays. On the left-hand side of the memex, we find an extra display which is used to view other pages that are on other trails. A scientist who wants to search information for his research works as follows: first he may have to add new microfilms to the drawer depending on which microfilms store the required information. Since the number of information is huge, the user will first look where that information is located on

the microfilms by consulting the "Yellow Pages" (see Figure 2.1b) to find the code of the page he is looking for. Once he has found the code of the page in the "Yellow Pages", he will have to introduce the code manually on the keyboard. After having actioned the switch to proceed in order to view the page, the code of that page will first be displayed on the mechanical display above the "Yellow Pages". Then the image of the page will be projected on one of the two displays in the middle of the memex. The user can repeat the same procedure to find another page and display it on the other display. Furthermore, the scientist has the possibility to register a trail between the two pages. The trail has a unique identification code and is stored permanently. The pen is used to annotate the pages with handwritten comments and draws. The main trail can also be linked with a side trail in order to reach a particular item. A few years later, the scientist can search for a specific trail by entering its code and display the item linked with that trail. This is similar to the examples mentioned before, except that if trails would exist in our filesystem, we would store metadata on a trail to search for it. The trail can be shared with another scientist by photographing the whole trail, save it on a microfilm and pass it to the other scientist.

Some ideas of Bush's memex have been taken into account, especially the idea of "linking" pages, for example in the Xanadu project. Nowadays, the idea of linking pages is a concept used on the World Wide Web. Webpages link to other pages of other websites or to different webpages on the same website. Since the introduction of the Web 2.0, people can easily create content on the Web and annotate webpages, in a similar way to the idea of Bush's memex. For example, we can add tags to a webpage which can then be used when searching for a specific webpage or we can attach comments to webpages. The Web has become an immense source of information where everybody works, updates information and collaborates with other people on the Web. We find all kind of information in a lot of formats including sound and video. Unfortunately, these features that are available on the Web are not found in current filesystems. We cannot tag files with properties, neither attach comments. It would be useful to have the features to add metadata or annotate a file with personal comments in order to understand its content. We can compare the usefulness to adding comments to methods when we are editing the source code of an application, in order to understand what the functionality of a method is.

The memex can be seen as the pioneer of todays Web, and the Web could be seen as a network-enabled memex. Some ideas of the memex have been extended, and others have been distorted over the time. The memex originally was not meant to be used on a network, even if it would have been a good idea if the technology existed at that time. It was meant to be used as a sort of desktop where all microfilms are

stored locally (but which can be transported). Some projects have tried to build an information system with the idea of linking resources in a special way. For example, project Xanadu proposed to use bi-directional links. Unfortunately bi-directional links have not been realized so far on the Web, neither on filesystems. Bi-directional links on filesystems can be useful in some situations. For example, suppose that a file is being used by another file and that the two files are bi-directionally linked together. If we try to delete the target file, probably because we think that is not useful anymore, we would see that this file is used by another file (i.e. is linked by), and thus prevent the target file to be deleted. The source file of the bi-directional link will therefore not point to a nonexistent file. To illustrate this with a concrete example, suppose that an application A is installed together with `.dll` files (dynamic link library). Later, an application B is installed and needs the same `.dll` files as application A. Therefore, we let application B point to those `.dll` files. After a while, we do not need application A anymore and decide to remove it. With bi-directional links, we would see that application B uses the same `.dll` files as application A. Thus we would just remove application A and leave the `.dll` files in order for application B to still work. An existing software management system using a similar idea to check the dependencies between applications is the Redhat Package Manager (RPM) [21]. We do not intend to only use bi-directional links for software management systems, but for the entire filesystem.
The idea of storing data permanently was imagined by Bush by using microfilms. Since of Bush's time photo cameras with microfilms existed, he thought of using that same idea for taking snapshots of articles and pages of books (or other relevant information sources). The photos would have been taken by fixing a small camera fixed on the forehead of the scientist as shown in Figure 2.2.



A scientist of the future records experiments with a tiny camera fitted with universal-focus lens. The small square in the eyeglass at the left sights the object (*LIFE 19*(11), p. 112).

Figure 2.2: A camera fixed on the forehead of a scientist

When the scientist sees a page or a visual instance he is interested in, he just takes a picture of it. When finished, the microfilm tape of the photo camera would probably have been attached to a bigger microfilm that would fit into the drawers of the memex, and use the memex to project the photos from the microfilms on a higher resolution. We can draw inspiration from taking snapshots of visual instances and integrate it in a filesystem. For example, we can select a part of the content of any file (like a video sequence) we are interested in and save it. The saved "file" is not a file on its own, but a link to a part of another file. When we open that "file" (or link to it from another file), we would only view the selected part.

Although the memex at that time was a good idea and integrating the concept of linking our documents would have been great, it misses some useful features. The memex does only link entire pages, and not parts of pages such as pieces of text. For example it would be handy if we do not have to copy and paste in our HTML webpage some text of another webpage and restructure everything by removing tags like `<H1>`, `<P>` that do not fit in our webpage. An example is that we have a website and some webpages that should embed content of other webpages. A reason for doing that is if we have to extend or restructure the content of the website, we can instead of making new webpages and copy and paste the new content, just reference to the content on other websites and render it in our webpage. In a similar situation on a filesystem, instead of copying and pasting text from a document A to another document B, we can link to the piece of text we want and render it in our document A itself. There exist alternative technologies like mashups in order to extract parts of websites we are interested in and integrate them automatically into our webpage. Unfortunately, this functionality is not integrated in HTML. What is surprising with HTML is that it is possible to embed any kind of media like video, sound and images of other websites and render them in our webpage, but it is not possible to do that with text.

There are two other features that are missing in the memex. The first one is that there is no possibility to search for a page with keywords, without searching the code in the "Yellow Pages" and insert it manually. The second missing feature is the lack of metadata which can describe the content of a page.

The memex has never been realized, but other projects such as Xanadu have been inspired by Bush's idea of associative memory and come up with similar ideas like *hypertext*. It would be nice to integrate the idea of trails (navigational links) and bi-directional links in a filesystem. As mentioned before, we can take advantage of those two concepts for some applications. Also the idea of selecting parts of a file and be able to link to it can be useful, as it has the advantage to reuse data instead of duplicating it. Although search functionality and metadata exist in filesystems, they are very limited. It is not possible to add our own metadata to files, which

can be useful to search for them. Also the search functionality is limited to search files by name and system-predefined metadata (properties).

2.2 Overview of the History of Filesystems

Mankind has always discovered new things and had the need to keep their knowledge in order to be reused on the long term. The evolution of the way how humans "wrote their knowledge down" before the 20th century was quite slow, in the sense that we used first at a certain time rolls, for example in the old Kingdom. Books that we know today had many more advantages, like the containment of more content, the logical separation of the content by using chapters, the easiness to consult that content and even organize those books (for example by author or subject). They have been introduced thousands of years later during the Middle Ages.
At the end of the 20th century, new discoveries and the related knowledge started to emerge very quickly. This was possible especially with the introduction of computers, because they had a promising future to save a lot of information in a compact and, later, portable way. Saving and retrieving data was so important that since the early 60s, many filesystems have been introduced in order to achieve that purpose. Some players in the filesystem market that will be discussed are Microsoft, Apple, Unix, Linux and BeOS [13].
The popular *File Allocation Table* (FAT), was first introduced by Microsoft. The first version was *FAT-12*, because it could only address 12 bit at the time to count the clusters on the persistent storage (at that time, floppy disks were used as standards). In total, FAT-12 could only address a space of 32 MB ($2^{12} = 4096$,with 8KB clusters). The naming convention of files was "8.3", meaning that a file could contain a name of maximum eight characters, and three characters used to distinguish the type of file with an extension. The filesystem initially did not support the use of directories at all. It only stored each files next to the other. Later, the storage space grew, and nested directories were introduced because it would have been impossible to arrange all that data without any hierarchical logic. Microsoft introduced *FAT-16* after FAT-12. That filesystem could address a maximum of 2 GB storage space, but still had the limitation of naming files with eight characters and three characters for the extension. Together with Windows 95, *VFAT* has been introduced, which let the user to enter a filename up to 255 characters. To solve the problem of backward compatibility with older filesystems, VFAT needed a workaround. The trick was to let the user enter a filename up to 255 characters and automatically shorten it to eight characters (the remaining letters were hidden and stored elsewhere in the system). For example a file called "vacation in Paris.gif" is shorten to "VACATI1̃.gif". VFAT was not able to solve the problems originating

from FAT-16 (problems with space waste and fragmentation). Instead of refining VFAT, Microsoft introduced *FAT-32*, which could address a storage space of 32 GB maximally (this was a limitation made by Microsoft even if the filesystem could address 8 TB of storage space). Together with the introduction of Windows NT, the *NTFS* filesystem saw the daylight, which was a 64-bit filesystem. The particularity of the filesystem was an enhanced security by using ACL (Access Control Lists), disk encryption and disk quotas. Compared to FAT-12, -16 and -32, which had some predefined metadata, NTFS introduced more predefined metadata. Another particularity was the indexed search of files to speed up the searches.
Apple also wanted to become a major player on the market, so it introduced the *Macintosh FileSystem*, or *MFS*, which originally had a limit of 20MB and 4,096 files. The user could create graphical "folders" and drag files into them, but there were no folder hierarchies. MFS was replaced by a system with proper hierarchical directories in 1985 . Because of this, it was called the *Hierarchal FileSystem*, or *HFS*. HFS used 512KB clusters with a 16-bit pointer, so the maximum size of a drive was 32GB. MFS and HFS introduced an innovative way of handling files, called "forks". Instead of storing metadata separately, HFS represented each file by two files: the file itself (the "data fork") and an invisible "resource fork" that contained structured data, including information about the file, such as its icon. The file was not readable anymore when it was sent to another computer that did not know about forks. HFS used a massively four-letter "type code" and another creator code, which were stored in the filesystem's metadata, treated as a peer to information such as the file's creation date. But HFS had some technical limitations. As long as it was used in a single tasking system, the filesystem was fast. But later, multitasking caused some problems, which were fixed by HFS+. It used a 32-bit number for block numbering.
Unix almost completely dominated the market for scientific workstations and servers before being neatly replaced by a work-alike clone called Linux, which started out as a pun on Unix. The filesystem used was called *Unix FileSystem* (UFS). *Inodes* were used, which are pointers that locate files. Unix-like systems have introduced an interesting way to organize files and folders beside hierarchical folders : *symbolic* and *hard links*. A symbolic link is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path, and that affects the pathname resolution. Programs which read or write to files named by a symbolic link will behave as if operating directly on the target file. However, if a symbolic link is deleted, its target remains unaffected. But if the target is moved, renamed or deleted, any symbolic link that used to point to it continues to exist but now points to a non-existing file. Hence, those symbolic links are broken[1]. A

[1] http://en.wikipedia.org/wiki/Symbolic_link

hard link is a directory entry that associates a name with a file. Multiple hard links can be created for the same file. This has the effect of creating multiple names for the same file, causing an aliasing effect. In contrast to symbolic links, they are not a link to a file itself, but to a filename. This also creates aliasing, but in a different way. Most operating systems do not allow to create hard links for directories to prevent endless recursion, which is a limitation[2]. *Shortcuts*, another type of link used in Windows systems, may resemble symbolic or hard links. But the main difference is that applications must be aware of shortcuts (i.e. how to access them) in order to reach target files, whereas symbolic or hard links are transparent for applications when accessing target files. A common limitation for the three mentioned link types in most operating systems is that they do not give the possibility to delete the target file or folder immediately (at least in graphical environments): the link will be deleted, but the target file or folder will remain (this case for hard links is only true if there is more than one hard link to a same file). *NTFS Junction Points*[3] partially solve the problem by allowing to link to directories and to delete the target directories immediately. However, they are only usable on NTFS and cannot be created by users (unless the system is tweaked). Operating systems like Windows Vista, 7 and Mac OS X now also support symbolic and hard links. In the beginning, there was no journaling in Unix filesystems, which means that if the system is for example shutdown while manipulating data, then the data was corrupt. The solution to this was to run a utility called `fsck` (for filesystem check). Many variants have been introduced like ext2, ext3, ext4 and ReiserFS. A few of those filesystems have succeeded to be faster and more reliable, for example those support journaling.

The operating system BeOs has been introduced together with the new hardware platform BeBox. The BeOs used initially a filesystem called *OFS*. It was particular in the sense that it used a relational database to store metadata of each file. Users could add metadata as much as they wanted. A drawback of the filesystem is that it was slow. Later, the *BFS* was invented that supported journaling, and was a much faster filesystem. Unfortunately, this filesystem has not survived long on the market because of the huge competition with Microsoft and Apple and the lack of knowing how to sell the product on the market.

A new filesystem called *Windows Future Storage* (WinFS)[4] [6], is currently being developed by Microsoft. The aim of this filesystem is to associate (user-defined) metadata to files and integrate an indexed search functionality via a relational database, just like the BFS. The major drawback of the attempt of Microsoft to create a new filesystem is that it is proprietary. It cannot be used by other filesys-

[2] http://en.wikipedia.org/wiki/Hard_link

[3] http://en.wikipedia.org/wiki/NTFS_junction_point

[4] http://en.wikipedia.org/wiki/WinFS

tems like ext4 and HFS+. In contrast, the RBAF system aims to work with any filesystem.

As seen in the history of filesystems, the aim was to make the best filesystem in terms of reliability and performance. However, less attention has been spent on the organization of data. Symbolic and hard links are indeed additional ways to organize files, but symbolic links are too fragile because they can be broken without warning the user, and hard links are not intended for folders. Both link types do not give full access to the target file such as deleting it through the link itself. Other link types such as shortcuts and NTFS junction points are used in specific operating systems. Since NTFS, ext4 and HFS+, no major new release has been introduced for years. Unfortunately, BeOS has been abandoned for his idea of using a relational database and user-defined metadata. To complete this section, a comparison Table 2.1, shows the usage of concepts of folders, metadata, symbolic and hard links for the organization of files in past and current filesystems.

Table 2.1: Comparison of filesystems[a]

Filesystem name	Company or Developer(s)	Hierarchical folder support	Metadata support	Symbolic links	Hard links
FAT-12	Microsoft	Introduced later	Limited and system predefined	No	No
FAT-16	Microsoft	Yes	Limited and system predefined	No	No
VFAT	Microsoft	Yes	Limited and system predefined	No	No
FAT-32	Microsoft	Yes	Limited and system predefined	No	No
NTFS	Microsoft	Yes	Limited and system predefined	Yes	Yes
MFS	Apple	No	Limited and system predefined	No	No
HFS	Apple	Yes	Limited and system predefined	Yes	No
HFS+	Apple	Yes	Limited and system predefined	Yes	Yes
UFS	CRSG	Yes	Limited and system predefined	Yes	Yes
ext2	Rémy Card	Yes	Limited and system predefined	Yes	Yes
ext3	Stephan Tweedie	Yes	Limited and system predefined	Yes	Yes
ext4	Mingming Cao and other	Yes	Limited and system predefined	Yes	Yes
ReiserFS	Hans Reiser	Yes	Limited and system predefined	Yes	Yes
OFS	Be Inc.	Yes, relational database adds extra flexibility	User defined	Yes	Yes
BFS	Be Inc.	Yes, relational database adds extra flexibility	User defined	Yes	Yes

[a]partially taken from http://en.wikipedia.org/wiki/Comparison_of_file_systems

2.3 Project Xanadu

In this section, we first give a short introduction of the Xanadu project. Next, we discuss the model which is used in the architecture of Xanadu, and finally we discuss the idea of *transclusion* used in the project.

Ted Nelson created Xanadu[5] [9], a project of information system for instant and universal sharing of computer data. Ted Nelson has been influenced by the ideas of Vannevar Bush, when introducing *hypertext*. Hypertext is text which contains links to other texts. The idea of hypertext is inspired by the idea of Bush's "trails". Just as Bush, Ted Nelson believed that linear text was a limited medium for accessing information. The principal idea of the Xanadu project was a networked system that would store and index all of the world's literature and other public and private information. Users would access information, for example documents, via the hypertext system and access one document via a document's link. Ted Nelson also included other visions within Xanadu. It would be a system that automatically bills a user for the delivery of copyrighted material, and rewards the author with for example money. Also, the user can link documents and has a view of two linked documents on one screen, with full support for versioning. In Bush's memex, the user can only follow a trail between documents in a sequence, expect if the trail has different paths. Ted Nelson proposed to the user the choice to freely move between links.

2.3.1 The Transcopyright Model

Ted Nelson has proposed a different way to create and publish content online. The idea is shown in a model called *Transcopyright*[6] as shown in Figure 2.3.

It is an extension to a previous model called *Xanadu model*[7]. The difference is that the Transcopyright model adds to the Xanadu model a copyright mechanism. The purpose of creating content is to not create files that are a sequence of bytes. Instead, we can create (a) "content(s)" as stand-alone data such as a pieces of text or images, located separately and be accessed by anyone, which are grouped in order to form a source of information. We can easily mix content from different sources. Content is stored in an abstract space called a *pool*. Any content can be created separately and is unified in a so called *virtual file* to represent the desired contents. A virtual file is nothing more than a list containing pointers to all content located in the pool. That virtual file has the extension `.xvf`. Ted Nelson knew that copyright issues play an important role, and therefore added in his model

[5] http://en.wikipedia.org/wiki/Project_Xanadu
[6] http://xanadu.com/tco/index.html
[7] http://xanadu.com/xuTheModel/index.html

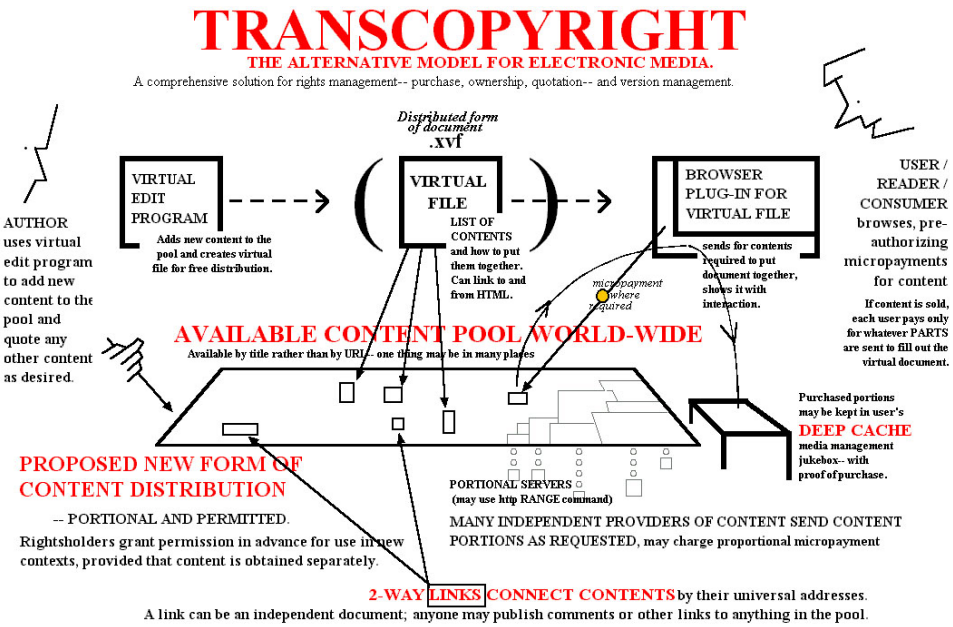


Figure 2.3: The Transcopyright model

a copyright mechanism to restrict access to specific contents. The cycle from the creation of content to the reading of the content by the end-user is as follows and shown in Figure 2.3. The author works with a virtual editor application which lets him create content and publish it in the pool. He also can read content of other authors and freely quote them, which is similar to the idea of annotating pages in Bush's memex. When the content is created, a virtual file will be generated that will hold the structure of all contents. The end-user can read the virtual file with a special browser plug-in for virtual files. The plug-in will read the virtual file and find the content associated with the virtual file, and put all contents together and finally present them to the end-user. We can use this idea to implement a similar feature in a filesystem: when a user wants to create a new file, for example a PowerPoint presentation, they will first check whether content can be reused in other files, such as entire PowerPoint slides in other PowerPoint presentations. Slides are not copied and pasted, but the content and structure of the PowerPoint presentation are created by linking to those external PowerPoint slides. Once the PowerPoint presentation is finished, the next time it is opened, the content of the external slides will be rendered and structured in the PowerPoint presentation itself.

This is how an author can publish content and a reader can read it freely. If content should be copyrighted, then the following approach is proposed. Any rights holder grants permission in advance to content. When the reader wants to access some contents in the virtual file, which are not free, one or multiple micro payment first

have to be done in order to have access to the desired content and so fill out the virtual file. Once the reader has purchased the desired content, the proof of purchase is stored in a deep cache. Next time, the reader can access the previously consulted content again if the proof of purchase is retrieved in the deep cache. Again, we can draw inspiration of this idea for a filesystem as well. We can, for example, adapt the access to linked content depending which user has to access it, or in which context (for example on which computer and operating system) the user is viewing it. Suppose that a student, which is logged in with a student account, has to view a document containing instructions for a course exam. In this document, there is linked content for students and linked content for teachers (for example instructions of how to correct the exam). The view of the document can be adapted by hiding the linked content of teachers from the students.
The pool is a transparent unification of different servers that host different content. Those servers send their content from requests of the reader when opening the virtual file.
Content can be linked together with two-way links (bi-directional links). The meaning is that we can see which content is connected to the other and vice versa. This concept is interesting to use in a filesystem. For example, we can make shortcuts, used in Windows systems, bi-directional in order to know which file is being linked by which other files, and prevent to accidentally delete a referenced file. Links are treated as content, and can be created and published freely in the pool. To search any content in the pool, it will not be addressed via its URI, but via its name.

2.3.2 The Xanalogical Structure

The functionality of linking documents (which also includes composing documents with other content, as with virtual files) is based on the Xanalogical structure [10]. It is a symmetric connective system that enables to link multiple forms with each other and view those links as connections between two forms. Ted Nelson calls this *parallel documents*, which has been introduced in 1965 to represent abstract documents that can be associated in two ways. The first way is by linking documents with each other, which is called *links*. The second way is a more complex linkage that links content of a document and incorporates it in another, which is called *transclusion*. One of the main purposes is to be able to view documents side by side and compare them. Graphically we can view two documents next to each other, and see what is being linked and what is extracted from one document that is incorporated into another. The graphical representation of parallel documents is called *transpointing windows* as shown in Figure 2.4. The necessity of working with transpointing windows can be varied. Two main valid contexts that come to our mind is first when building documents that are the continuity of previous versions.

When writing the following version of a document, for example version 2, it is handy to view version 1 next to version 2 to compare and add content in version 2 from version 1. Second, another situation is simply when studying literature. An example is when we study geometrical optics, it is possible that there are some study fields or terms that we do not know. Therefore, there can be links that point to other documents that treats the subject or term. In geometrical optics, we refer to subjects like polarization and terms like rays and imaging. When the reader selects one of those links, the corresponding document page will appear next to the document about the geometrical optics. The reader will be able to read gradually the part of the subject or term that he wants to read about while keeping an eye on the chapter of geometrical optics.

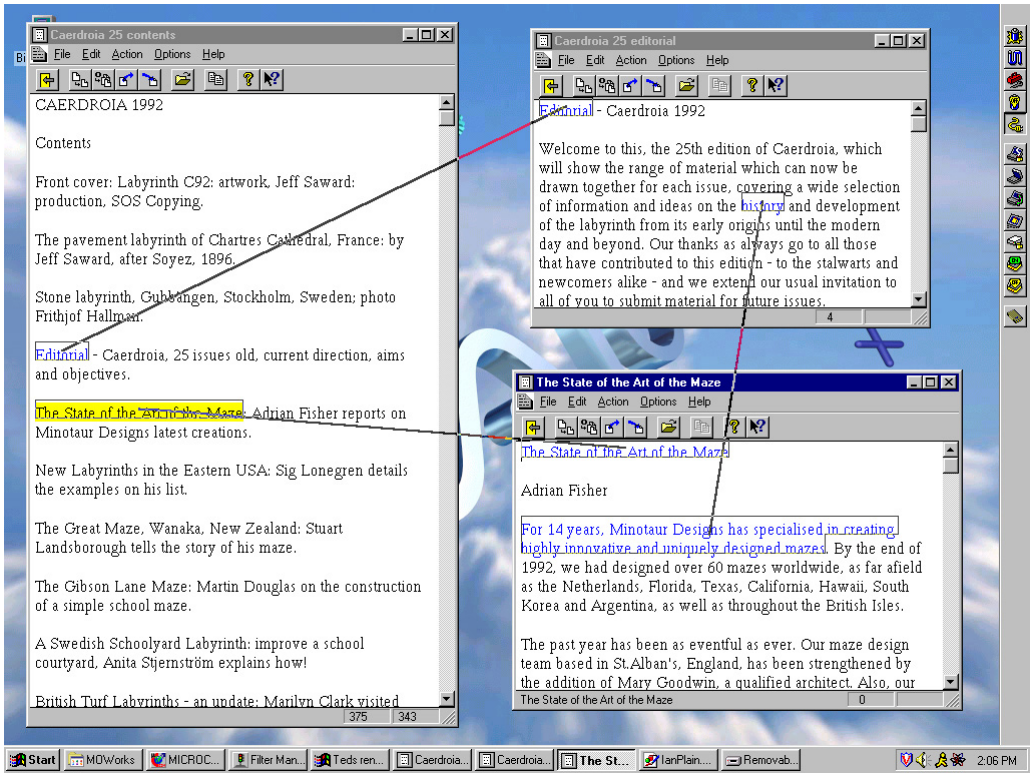


Figure 2.4: Transpointing windows

As we have seen previously, documents are considered as a virtual representation of content entities in a "space" (which is the pool mentioned before). In other words they are pointers to content entities located somewhere. Figure 2.5 shows pointed contents that together form a document.

Content is created by everyone, and a document contains a list of all the content that should be shown in an order. In transpointing windows, we use the concept of transclusion. The idea is that we can reuse parts of documents by linking to those

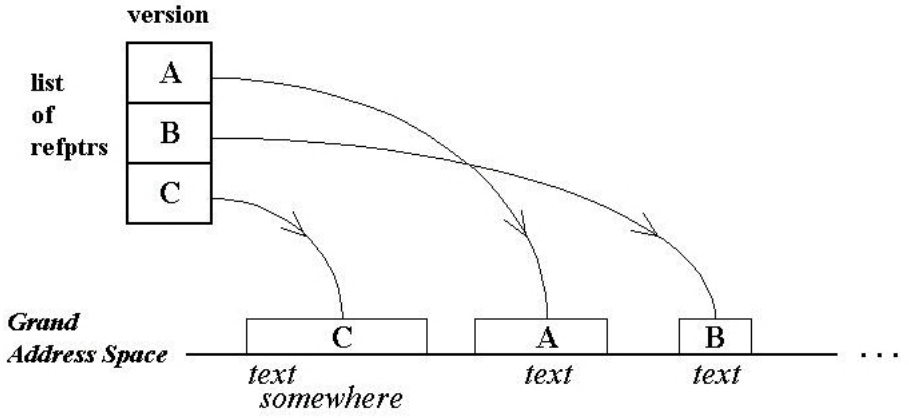


Figure 2.5: A reference pointers list is a virtual document that points to external contents

parts from a document and render the linked content in that document. Suppose that somebody, called Mister X, has a document called B and it contains a part that explains a mathematical definition of a formula. He can select and then point that part to the piece of information about the formula in document A. When he wants to visualize document B he will see that it is composed of the content of B itself, and the link that points to document A. Figure 2.6 shows that document B links a part of its content to a piece of information of document A.

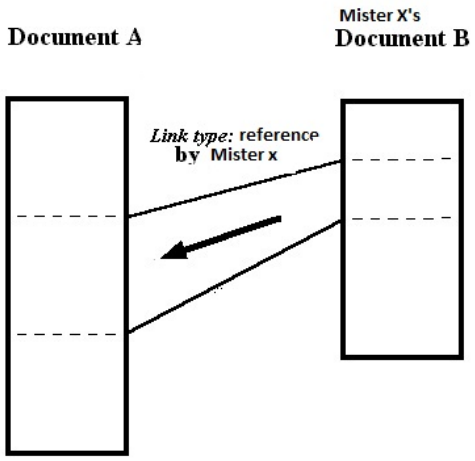


Figure 2.6: A piece of text of document B is linked to a piece of text of document A

The aim of links used in transcluded content is that they are unbreakable. Suppose that document A has extended the part that explains about a mathematical formula. The updated part adds new lines in the paragraph itself. It is like the paragraph has been split in two and the new lines are pushed between the two

splitting parts. The link to the transcluded part of document A will be updated by creating two links that point to the separated lines. Figure 2.7 shows the link of document B pointing to the extended piece of information of document A.

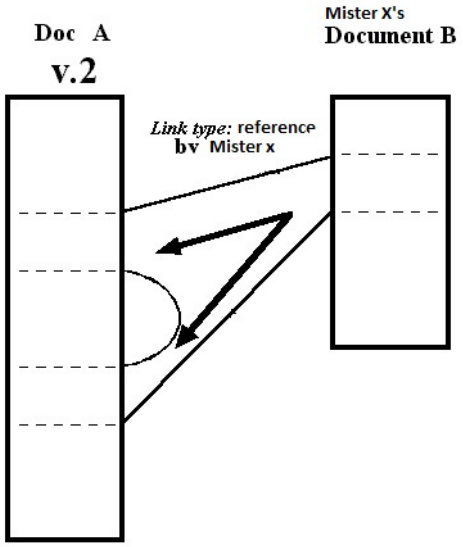


Figure 2.7: Document B still points to the old same piece of text of document A after an update

When a part is deleted, the link will also be updated by pointing to the remaining lines. Figure 2.8 shows that document B still links to the piece of text of document A even if a part of that text has been deleted.

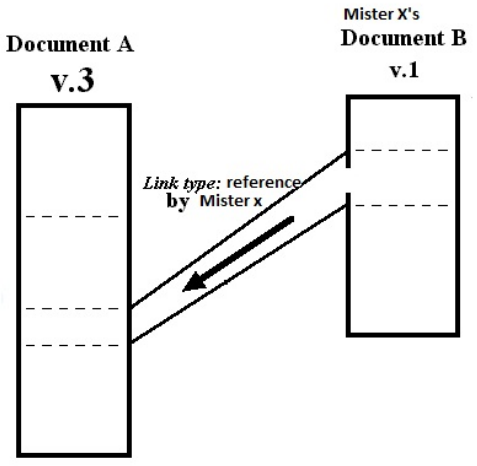


Figure 2.8: Document B points to the reduced piece of text of document A

Using the idea of transclusion in a filesystem is also interesting. From the previ-

ous example of linking PowerPoint presentations to PowerPoint slides, we can link to *parts* of the slides in order to reuse the same content in our PowerPoint presentation, such as a piece of text, and not an entire PowerPoint slide. In this way, we can avoid replicating or creating files for a temporary usage to finally save storage space efficiently. A consecutive advantage is that we can reduce fragmentation on our disk drive, since we do not create, copy, paste or move content between different files.

Many ideas of Ted Nelson are reality today. However, Ted Nelson's system was never realized because it was unclear at that time how it should be realized. Input/output issues, filemanagement issues, facility sharing and networking were things that needed to be figured out before this system could become reality[8]. Today we have the technology to realize some of Nelson's ideas. The idea of transclusion does not only need to be used for the Web, but also on filesystems. Just as the transpointing windows of Ted Nelson, we can use that concept to view in a filesystem which files are linked by which one and what is linked. In this way, we are aware how files are related to each other, and thus we understand how files are composed, forming a unified set of data.

2.4 The oN Line System (NLS)

The oN Line System (NLS)[9] [1] has been invented by Douglas Engelbart in the 60s. The NLS was a system to let scientists and professionals collaborate on a project whose activities were to resolve complex problems. He has mainly been inspired by the memex of Vanevar Bush, who also wanted that scientists could collaborate and exchange their discovery. The NLS was the first system that introduced a set of mechanisms for information sharing. He also introduced inventions like the video-conference, windowing system (in a graphical user interface), the mouse and keyboard. The NLS was the first system that implemented a functional system that used the concept of linking (parts of) documents to other douments. It was possible to have internal links in documents by linking parts of the document to itself and only view those linked parts by selecting their link. Linking was not only possible in text, but also in graphics such as pictures. One could select a part of graphics and for example link to a piece of text in a document. So a user can for example select a part of a map and link it to a piece of text in a document. When clicking on that part of the graphic, the user jumps to that piece of text. Just like in the memex, it was possible to create chains of views. In other words we can link pieces of text in different documents and jump from one text to the

[8] http://www-personal.umich.edu/ mattkaz/history/hypertext2.html

[9] http://sloan.stanford.edu/mousesite/1968Demo.html

other. An application of the link concept used in the NLS was the *Journal*. The Journal was conceived as an electronic version of a scientific journal, consisted of a permanent database, which consists of a set of published articles, where users could access for consultation. The NLS facilitated the organization and retrieval of items through a system of indexes, catalogs and cross-references. The Journal was a primitive hypertext-based groupware program which can be seen as a predecessor of all contemporary server software that supports collaborative document creation (like wikis)[10].
As Douglas Engelbart used links in his system, we can also make use of a similar feature in a filesystem. The idea is to link (parts of the content of) files and travel from (the content of) one file to the other.

2.5 Human Memory Types and the Synapse-State Theory of Mental Life

In this section we first discuss three types of human memory that we can draw inspiration from and implement features for a filesystem[11]. Finally, we shortly explain a theory of Paul Naur [8] about the general working of human's nervous system and what inspiration we can draw from it.

2.5.1 What types of Human Memory can we draw Inspiration from for a Filesystem?

The human memory has three different common memory types (which are memory processes) of the long term memory[12].
One of those types is the *semantic memory*. The semantic memory is concerned with abstract definitions and facts to understand something. For example when we are in a school class following a lecture about rhinoceroses, although we probably have never seen a real rhinoceros in Africa, we do not have to remember a specific event of having seen a rhinoceros in order to understand the lecture. We can make use of the semantic memory in order to remember what is it (it has four legs, it lives in a warm country, it is a mammal, and so forth).
The second type of memory is the *episodic memory*. This memory is related with chronological events such as time and associated emotions. For example, remembering the name of our child we gave to him at his birth.
We also have a third type of memory, which is the *procedural memory*. This type of memory is used to remember how to accomplish things. For example, how does

[10] http://en.wikipedia.org/wiki/NLS_computer_system
[11] http://en.wikipedia.org/wiki/Semantic_memory
[12] http://www.wisegeek.com/what-is-semantic-memory.htm

one open a book. The semantic memory, together with the two previous mentioned types of memory gives us the possibility to remember objects or concepts that have a meaning to us and relate them in order to give it a sense, from the phase of understanding it to the phase of how do we use or do it. For example, when we want to read a book, our procedural memory remembers how we read it, but is it the semantic memory that will make it possible to understand what the different letters mean and that relating specific letters in a specific order will result in a meaningful word or sentence. The episodic memory lets us remember in which context we are in the story of the book and what happened before that story. With semantic memory we can also read every book with a different type of font because we understand the concept of a letter, and do not have to remember a specific example of a letter (e.g. with the font type Times New Roman). Figure 2.9 shows an example of each memory type.

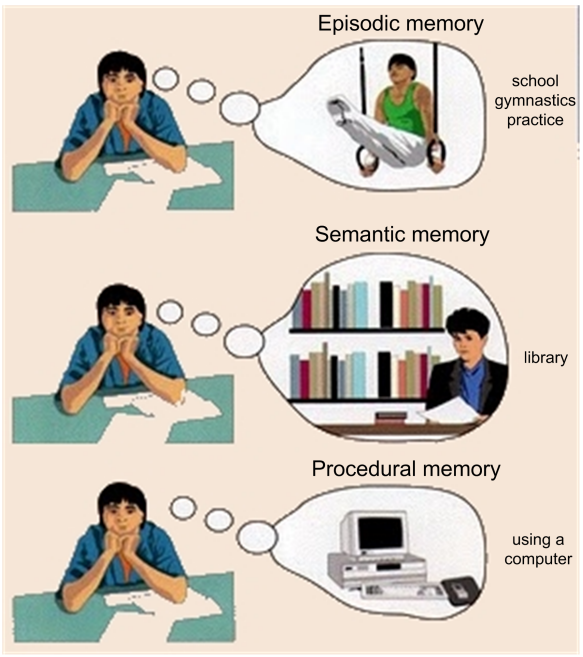


Figure 2.9: Example of each memory type[13]

For each type of memory, we can draw inspiration and extend functionality in a filesystem. For the episodic memory, we can use metadata in files as a description to remember its content. Not only the date would be stored, but also other properties such as in which context it has been created, why and what content it has. Another functionality can be linking files to semantically relate them. For example, we can relate a photo to a video because a photo has been taken of a historical monument, and the video of that monument has been created as well. For semantic memory,

[13]http://www.abacon.com/slavin/t58.html

we can use multiple classification of files within different folders [16]. The idea is that a same file (not a copy) can be contained in multiple different folders. Suppose that we have a folder containing photos of the sea during our vacation, and another folder about images of nature. We can classify photos of the sea as images of nature, because the sea is part of nature. Therefore we can classify photos of the sea both in the folder of 'vacation' and in the folder of 'nature'. For procedural memory, we can again use metadata stored in files and give instructions to applications on how they should manipulate a file.

2.5.2 What Inspiration can we draw from the Human Nervous System (Synapse-State Theory) for a Filesystem?

Peter Naur [8] argues that there is no similarity with computers and human thinking. He states that human thoughts are very complex and the way of a human to think is very flexible and variable. This means that we do not think about something twice in the same way. In order to show how flexible and complex human thoughts are, he has created a model that represents the cooperation of neurons that are influenced by many factors in different layers. Neurons are cells in our brain that are influenced by millions of other neurons via plastic conductors called *synapses*. An habitual connection between those neurons is called *association*.

The model is divided into five layers shown in Figure 2.10. Those layers are the *item-layer*, the *attention-layer*, the *specious-present-layer*, the *sense-layer* and the *motor-layer*. The item-layer is in the middle of the model, the attention-layer is in the left corner, the specious-present-layer is in the right corner, the sense-layer is on the top of the model, and the motor-layer is on the bottom of the model. The nodes are shown with boxes, neurons are shown with lines and the synapses are shown with short words in capital letters. Those words are ITEM,ATT,SPEC,SENS, and MOT.

The Item-layer

This layer contains all habits and information that stands on the long term in our organism. The first thing that attracts our attention are the rectangles which are nodes. The nodes are the end-points of neurons. When enough neurons, which are connected with a node, are active then this node represent an acquaintance objects,which is simply a thought. The nodes are connected with other nodes in a specific order. The path from a node A to a node B is composed of a neuron, a synapse and a neuron again. The synapse on the model is represented in the form of ITEM-x. The synapses play an important role because they will make it possible that two (or more) neurons become unified, which will have a sense for possibly each individual person. To clarify this, a newborn has synapses that are

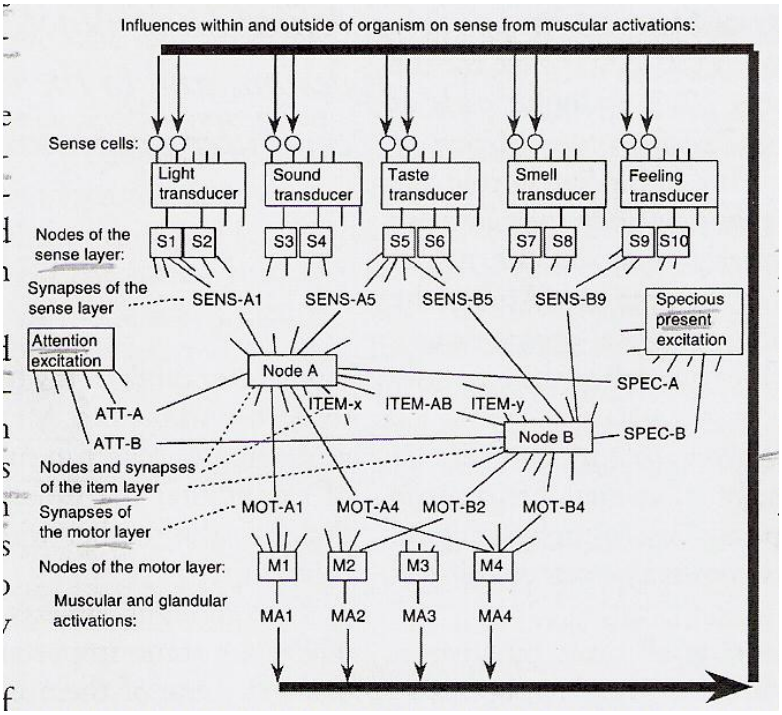


Figure 2.10: The Synapse-State model of the nervous system

not conductive. While it is exposed in front of external factors, multiple nodes will be excited. When in a random situation two nodes are significantly excited at the same time, the synapse that relates those two nodes will be excited too and thus will be more conductive. The more those two nodes are excited, the more the synapse will be conductive. When the synapse is conductive enough, the newborn will have learned that two nodes have something to do with each other. We say then that those nodes are associated. Once the two nodes are no more excited, the synapse that relates those node will be less conductive with the years, which probably will result of having forgotten about the relation between the two nodes. If we want to mimic that behavior in a filesystem, we can compare a node with a file, a neuron with a link and a synapse with properties attached to a link. When we search for a file, but with no results, we can try to search for properties on links (i.e. metadata on links) between files that match the search criteria. Once a link is found with properties based on the search criteria, we can be asked if a file on the other side of the link is what we are looking for. If this is the case, a property which represents a search "hit" will be updated. Next time we start a search process with similar search criteria, the search results will first show the "hit" file from the previous search process.

The Attention-layer

The attention node, which is called the attention excitation, is a particular node that will result in the attention of an acquaintance object. Every node is connected with the attention excitation node which contains its synapse in the form of ATT-x. If for example a node A is excited beyond a certain level, possibly by external factors like senses (for example eyes and taste) or by other nodes that are connected with node A, then the synapse ATT-A between the attention excitation node and node A will be conductive for a short duration. The summation of impulses will be influenced and we say then that the attention sits in node A. The synapse ATT-A will get weaker very quickly, in terms of seconds, and needs to recover before it can be excited again. In a filesystem, we can mimic a similar feature with properties on a file. Suppose we often use a file located in a folder. Every time we open that same folder, we will see that file highlighted in order to attract our attention and immediately perceive that file, instead of looking where that file in the folder is. This feature would work for any file in any folder.

The Specious-present Layer

This layer is almost similar with the attention-layer. We also have a node called the specious present excitation node that is connected to nodes with each their own synapse in the form of SPEC-x. The difference between the two layers is that the synapses of the specious present excitation node are slower. The nodes connected with the specious-present excitation node need to have a certain level of excitation to excite the synapse SPEC-x itself. There can be other nodes that are excited at the same time and will excite their synapse SPEC-x too. Once a synapse is excited, it will last for a short time, which is longer than a synapse of the attention excitation node. We can compare this process with the short term memory.

The Sense-layer

External factors signals are captured by nodes in the sense-layer called transducers. Those transducers are translated and excite a specific or multiple nodes called which have the form of Sx, for example S1,S2,... where S stands for sensation. When those nodes are excited there will be a tendency where some nodes in the item-layer are excited. When a lot of the nodes Sx are excited and excite in their turn a particular node with a synapse in the form of SENS-x, then that node by summation of impulses will be strongly excited. When the excitation is strong enough to excite the attention-synapse, we say then that the corresponding acquaintance object is in our perception, and thus the nodes Sx stay excited as well in the same time. It is also possible that the conductivity between the ITEM node and the SENS node

occurs in the opposite direction. We can imitate that behavior in a situation where a file is structurally linked to an image. i.e. the file that links to the image will render that image in its own content, but does not physically contain that image. If in this file we often modify the image, and later browse the folder containing the image, we would see (with for example a notification) that the image has been modified multiple times by the file.

The Motor-layer

The item nodes are connected with nodes in the form of Mx in the motor-layer through synapses `MOT-<item name><number>`, for example an item node A can be connected with a node M2 with the synapse MOT-A2. When a synapse of the motor layer is excited, it will activate a muscle contraction in our organism. A MOT node will last less longer, more or less a few weeks, compared to an item node, which can last for a few years. For example when we play the piano, we need to train often if we do not want to lose the ability to play it well. This means that the nodes in the motor-layer need to be excited regularly if we do not want them to faint. The conductivity of the synapses in the motor-layer are increased with training. In a filesystem, we can use properties attached to a link between files. A property would contain an instruction of what to do when a certain event occurs, for example "onDelete".

2.6 The RSL Model

Many hypermedia models have been proposed in order to be used as an architecture to implement extra navigational functionality in hypermedia systems. These models have the purpose to be used as a certain design style and representation of the general architecture of an implemented hyper media systems. An example of a popular hypermedia system is the World Wide Web, which is a distributed hypermedia system that works on top of services provided by the Internet. The World Wide Web is based on a client/server model. This model defines protocols, like the hypertext transfer protocol, in order to exchange information between World Wide Web client and servers. The metamodel called the resource-selector-link (RSL) model [19] has been developed to be general and flexible in order to be used for hypermedia systems that evolve. The RSL model is based on the concept of linking arbitrary resources. Although the RSL model has the initial purpose to be used in hypermedia systems, we can use this model as a basis to realize extended functionality in a filesystem (which we will discuss in Chapter 4). There are three types of concepts that are important in the model: resources, selectors and links.

2.6.1 RSL Links

The RSL model is expressed with the OM data model [11]. All the concepts are grouped in rectangles. Those rectangles are collections of object instances. For example a `link`, which is an instance, belongs to the classification of `Links`. We distinguish six types of concepts shown in Figure 2.11. We will start with the concept of *entities*.

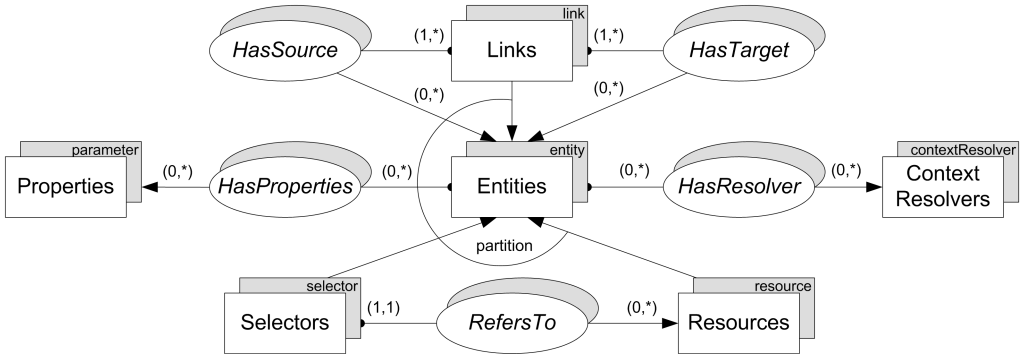


Figure 2.11: RSL links

An entity from the `Entities` collection is an abstract representation of any object that will exist in the hypermedia system. Because it is abstract, it needs to be extended by other subtypes. We distinguish three subtypes of an entity.
The `resource` subtype, which is another abstract concept, must be also extended in order to address concrete types of media, for example an image.
The `selector` subtype is a concept that is used to select a part of a resource. It is also an abstract concept that should be extended in order to create specific selectors to address parts of certain media types. We can imagine to create a selector to be able to select parts of a specific media. The association `RefersTo` has two cardinality constraints between a selector and its resource. The constraint (1,1) on the selector means that a selector is associated with exactly one resource. The other constraint (0,*) means that a resource can have zero or more selectors that refer to it.
The `link` subtype is a concept that has the purpose to link entities to each other. This means that we can have different combinations of linkages between different types of entities. For example, we can link from a resource to a selector of a resource, link a resource with another resource or link two links with each other. The link concept has an association `HasSource` with the `Entities` collection, and an association `HasTarget`. An entity as a source must have at least one target, but it can be linked to multiple target entities. Furthermore, a target entity can be linked by multiple entities.

An entity can only be extended by one subtype as indicated by the `partition` constraint, which means that we cannot have an entity that is a selector and a resource at the same time.
Next we have the concept of *context resolvers* from the `ContextResolvers` collection. A context resolver tells us whether an entity can be accessed or not in a certain context. Each entity can have multiple context resolvers as indicated by the `HasResolver` association. An example of the usage of context resolvers is a resource that is linked to multiple target resources. By coupling a context resolver on each link, we can define in which context each link will be visible or not. Suppose that we have a photo linked to a video and we are in a context where we only want to view photos, then it will be not relevant to show a link pointing to videos.
Finally, we have the concept of *properties* that we find in the `Properties` collection. A property is a key/value string tuple. We can compare a property with an attribute. The 'key' is the description of the property. The 'value' is the information that the 'key' contains.

2.6.2 RSL Structures

The link concept described above has the possibility to be extended with one of two other subclasses. Those subclasses are the `StructuralLinks` and the `NavigationalLinks`.
Structural links are links that are used by resources that want to be composed with other resources whom they are linked to, as shown in Figure 2.12.

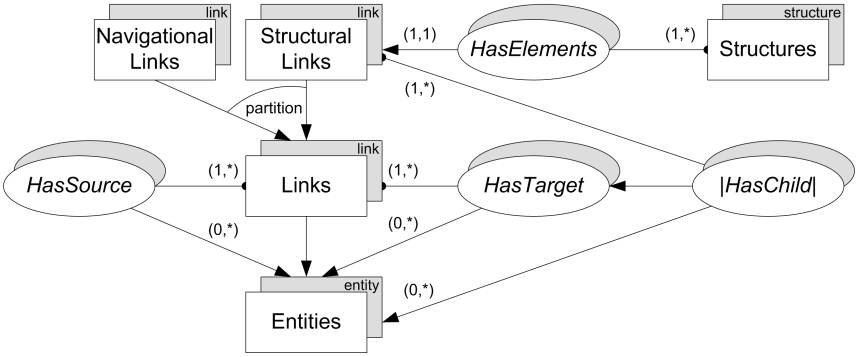


Figure 2.12: RSL structures

The `StructuralLinks` collection is associated with the `Structures` collection via the `HasElements` association.
The responsibility of the `Structures` collection is to keep the elements that are referenced via structural links. For example, we can link a document to multiple images. The links will make it possible to structurally compose the document of

images. That idea is similar to Ted Nelson's transclusion concept. We also need to know the order of such entities since we are talking about structures. This is achieved via the ordered | `HasChild` | association between the structural links and the entities collection. An example of an order is the order of a document that contains chapters, which in their turn contain paragraphs. We can not only define a structure over data, but also over other structures. This means that we can for example take a substructure of a document like taking one chapter with all its paragraphs, and make that substructure part of another structure within another document. Structural links can also form structures of regular associative links.
Navigational links from the `NavigationalLinks` collection are used to navigate between different entities. We can compare it with hyperlinks on the Web, where we can navigate from one webpage to another. We can also combine the concept of structural links together with navigational links. For example, we can define an ordered list of navigational links where the user can select to navigate orderly from one entity to another. This idea is similar to Bush's trails, and the links used in NLS to jump from document to document. We can use that feature to create navigation trails which guide the user to visit entities in an order.

2.6.3 RSL Users

We can define access policies for each entity that is meant for users only.Four concepts are distinguished beside the entities concept, as shown in Figure 2.13.

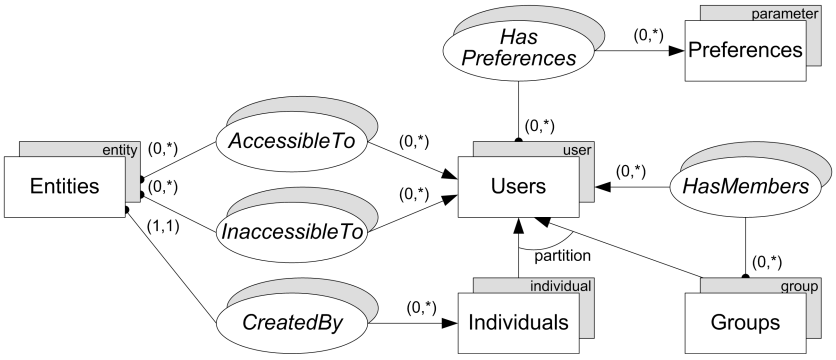


Figure 2.13: RSL users

First we have the `Users` collection. This collection contains all the users that can be in contact with an entity. The `Users` collection is an abstract concept, and must therefore be extended with one of two other subcollections.
The first subcollection of the `Users` collection is the `Individuals` collection. This collection contains every people individually.
The second subcollection is the `Groups` collection. This collection deals with a

greater range of users that can all be grouped. A group can contain individuals or other groups. Note that an individual and a user should not be confused. A user can be an individual or a group. An individual is somebody in person. From the `Entities` collection, there are three associations with the `Users` collection.
The first one is the `AccessibleTo` association. This association represents the accessibility of an entity by a certain user (either individual or group).
The second association is the `InaccessibleTo` association. This association has the same constraints as the previous one.
The third one is the `CreatedBy` association, which is associated with the `Individuals` collection. The `Users` collection has two other associations. One with the `Preferences` collection and one with the `Groups` collection.
The association with the `Preferences` collection means that every user has personal applied policies for each entity it is in contact with. The `Preferences` collection groups all parameters of every user. For example which entity can be accessed and which not.
The last association with the `Groups` collection is the `HasMembers` association : a user can be part of no group or many other groups.

2.6.4 RSL Layers

The RSL model deals with the problem of how to visualize and choose selectors that overlap each other on a same resource. Figure 2.14 shows the layers part of the model.

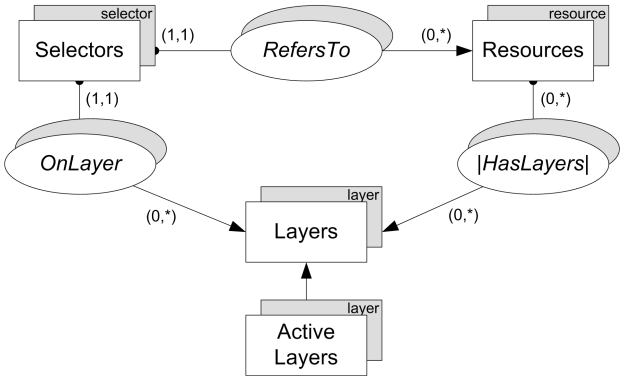


Figure 2.14: RSL layers

We identify two concepts in the layers model that deal with that problem.
First, we have the `Layers` collection that maintains the layers of different overlapping selectors on a same resource. Every selector is on a layer. The constraint (1,1) on the association `OnLayer` means that each selector is situated in exactly one layer. The association | `HasLayers` | from the resources type to the layers type is

an ordered association, and means that a resource can have multiple layers.
The subcollection of the `Layers` collection is the `ActiveLayers`. This collection is responsible to maintain the active layers in the `Layers` collection. We can activate or deactivate layers depending which selector we want to view and depending which context we are.

2.7 Summary

We conclude this chapter with a summary of which ideas and concepts from the previous discussions are interesting to implement in a filesystem. The concept of trails in Bush's memex can be used in order to follow a navigational path from a file or folder to another file or folder. Also, navigating from a part of the content of a file to another is an interesting idea, which was done in the NLS. Next, we have seen the evolution of filesystems where the focus was on data organization. The most important features are hierarchical folders, metadata, symbolic and hard links (in combination with other link types such as shortcuts and NTFS Junction Points, which are system specific). Unfortunately, no further improvement has been seen for making hierarchical folders more flexible and let users create their own metadata. An interesting idea, which has been abandoned, was the use of a relational database as in the OFS and BFS filesystems, to make flexible folders possible and add user-defined metadata. Symbolic links are fragile because they can be broken and thus dangle in the filesystem, and hard links are in most operating systems prohibited to link to directories. Other link types have advantages and disadvantages, such as shortcuts and junction points, but are only usable for specific systems. Further, the following concepts of the Xanadu project are nice. Transclusion, which can be used to link to (parts of) the content of files and render that content in another file. From the idea of a copyright mechanism and check if the user has paid for the copyrighted content, we can draw inspiration of it by implementing a similar mechanism, but in this case it is used to adapt for example the content of a file depending which user is accessing it or in which context the user is viewing the file (or what files he desires to view). Furthermore, we have seen how we can draw inspiration from the human memory and nervous system which motivates to implement features such as multiple classification. Finally, we have explained the RSL model, which forms the basis of our proposed filesystem discussed in Chapter 4.

In the next Chapter, we show two motivating scenarios that highlight some drawbacks of current filesystems in organizing and creating data. In Chapter 4, we then introduce the RSL-based Associative Filesystem (RBAF) system and its model.

Chapter 3

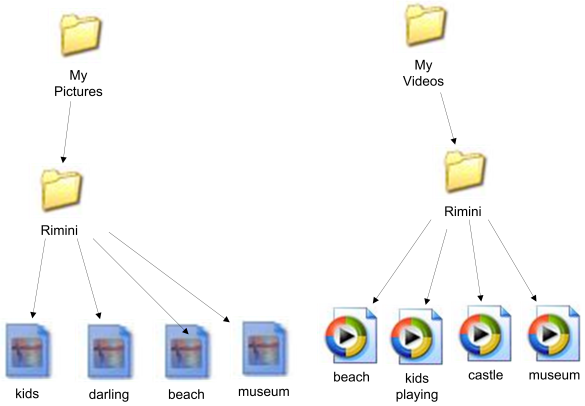
Scenarios

In this chapter, everyday examples are shown to demonstrate how we organize and create our data, followed by a discussion of some of the drawbacks of current filesystem-based solutions.

3.1 Scenario 1 : Souvenirs from Italy

For this example, we invent a fictive journalist named Charles Fricker and his family who are going for a holiday on the beach in Rimini. He and his family decide to go to Riccione in July 2010 for the first time. They plan to stay in a hotel for three weeks. The first week, they visit the local place of Riccione, where every day they go to the beach and enjoy a good time while swimming in the sea. Charles takes quite a lot of photos when his wife and kids are swimming, walking on the beach, having a chat and eating in their beach cot. His wife is filming the beach with her video camera to see how the children are playing, how the sea is waving and to capture the ambiance of the local beach place where there are a lot of tourists. During that week, they also visit the rest of the city, do some window-shopping, walking during the evening to see all the open stores and buy some local stuff. The second week arrives and the family decides to go visit the museum of Territory. There they take photos of the archeological finds. They film the guide when he gives explanations to other visitors. A few days later, they want to enjoy the nice weather in the La Perla Gardens. They enjoy the wonderful show of the fountains where they take photos of their children splashing their hands in the water. At the end of the day, Charles and his family walk around in the gardens and film his wife and kids.The third week they go to the Agolanti Castle. They take some photos and also film the interior of the castle. At the end of their vacation, Charles and his family go back home. In the evening, Charles organizes all the data that he and his wife captured with the photo and video camera on his personal computer. He uses Windows XP

as his operating system and creates two folders with the name Rimini in the folders My Pictures and My Videos. Each of these folders contains a number of files that he and his wife made during their vacation as shown in Figure 3.1.



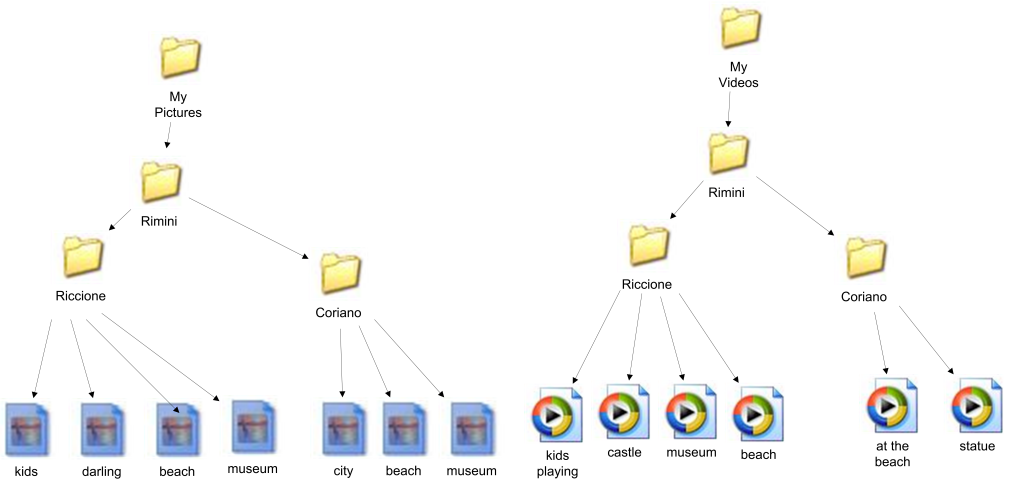
(a) Photo organization **(b)** Video organization

Figure 3.1: Initial data organization

Charles liked the climate in Rimini. The year after, he decides to do another trip to Coriano in Italy. There he stays on his own for two weeks. He visits some nice places and takes photos and videos to show to his family. When he returns home, he organizes the data on his personal computer. He faces the problem that he would like to distinguish the photos and videos that he made in Coriano and Riccione. Therefore, he has to create two subfolders Coriano and Riccione. He puts the older photos and videos in the Riccione folder whereas the new resources are stored in the Coriano folder, as shown in Figure 3.2.

Then a few years later, in 2014, he remembers how much he liked Riccione and decides to return there with his family. This time, they visit the archaeological site in San Lorenzo. The day is shiny, so they take their photo and video camera with them. They do not take a lot of pictures, but prefer to film some interesting parts of the site. This is the only place where they went to do some tourism. They prefer to enjoy the beach and the sea every day. The parents also like to spend time with their children since they could not do this as much as they wanted during the year because of their work.

When the whole family returns home, Charles saves his data on his personal computer. When he looks at the folder hierarchy, he realizes that he would like to differentiate the photos taken in Riccione in 2010 and in 2014. So he has no choice rather than to restructure his files. He creates two subfolders named 2010 and 2014 and puts the older photos and videos in the folder '2010' and the new content in the '2014' folder. Figures 3.3 and 3.4 show how the organization looks like. Note that



(a) Photo organization **(b)** Video organization

Figure 3.2: Second data organization

one image has not been given a name and therefore has the default name 'photo1'.

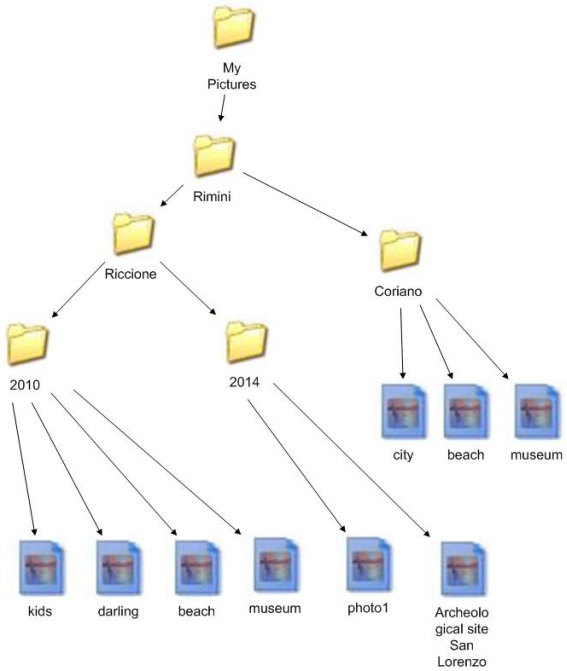


Figure 3.3: Third data organization in the My Pictures folder

As shown in this example, when our personal data space grows, we encounter all kind of problems. If we add new files from a different event (e.g. assignment or vacation), the chances are high that we must restructure all the files by creating

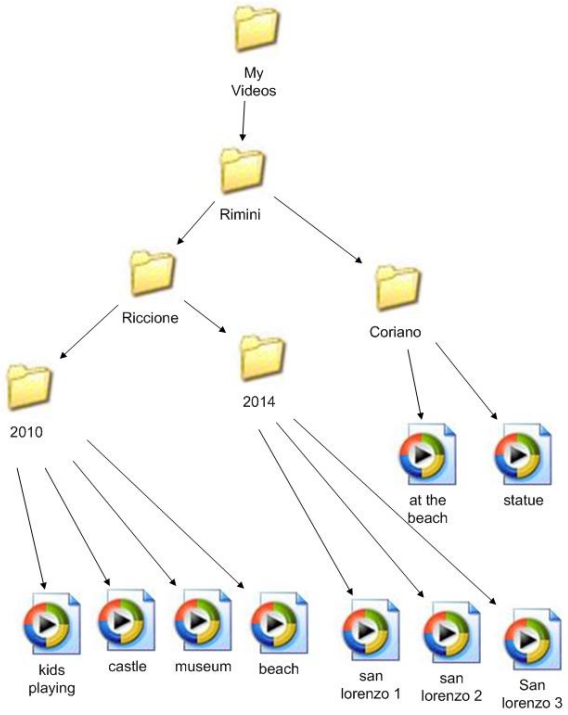


Figure 3.4: Third data organization in the My Videos folder

new subfolders and move those files to the new subfolders. Over the time, our personal data space can become very complex and rather difficult to browse. A complementary way to enhance the organization and retrieval of the files is to add properties to them and group those files that are the most related (e.g. by place or by year). Properties add more meaning to files and can be used for different purposes like the semantic regrouping of files when we search after them with the help of a search function or give instructions to an application. Even if the photos and videos have been taken at the same place and time, we currently cannot see that those data have something in common. We can try to give consistent names to the files. For example, suppose that Charles Fricker made a photo and a video of the fountains of the La Perla Gardens in the previous scenario. If we want to show that a photo and video have been taken on the same place, we can name the photo and video 'the fountains with kids'. But if Charles has multiple photos that have been taken in the La Perla Gardens and wants to know that those photos have been taken while also filming, he cannot rely on a naming convention anymore. For example he cannot name a photo 'the fountains when kids are running' and another one 'the fountains when kids splashing in water'. It is not a good idea to give a description of the image as the name for the file, because the names can be too short or too long to provide enough information about the photo.

Actually, there is no way to relate the photos and videos. We can guess that the video and some photos have a semantic relation only if we view the video and the photos, and remember that they have been taken at the same place (as in Riccione from the scenario).

3.2 Scenario 2 : Preparing PowerPoint Presentations

Mister John Kennon is a fictive doctor in computer science. He is invited at the conference of the MICT (ministry of information and communication technology) to present to visitors the use of firewalls in today's computers. He prepares a PowerPoint presentation simply entitled 'The firewall'. He thinks about what he will talk about and which subjects he will cover (we will not talk about the content of the presentation in this example). Finally, he comes to the conclusion to use these subjects as seen in Figure 3.5.

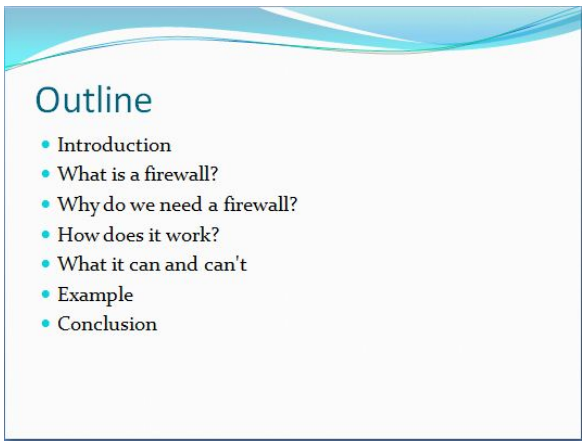


Figure 3.5: Outline of 'The firewall' presentation

While writing the appropriate content, he searches for some images on the Internet about the architecture of firewalls to paste on his slides. Finally, when all is done, he saves his PowerPoint presentation. A few months later, he has to give a lecture about firewalls in Windows. Again, he prepares his presentation and at the end of the day he finishes the presentation containing the subjects in Figure 3.6.

John Kennon realizes that his presentation has more or less the same content as the previous presentation he made at MICT. Nevertheless, he wants to avoid repeating the same thing in a slightly different way and therefore decides to copy and paste some of the slides of his previous presentation.

Two years later, John Kennon must give a lecture about the Netfilter architecture in Linux. Like always, he prepares his presentation for the next day, by copying some slides, modifying some images and pasting them in the new presentation. His

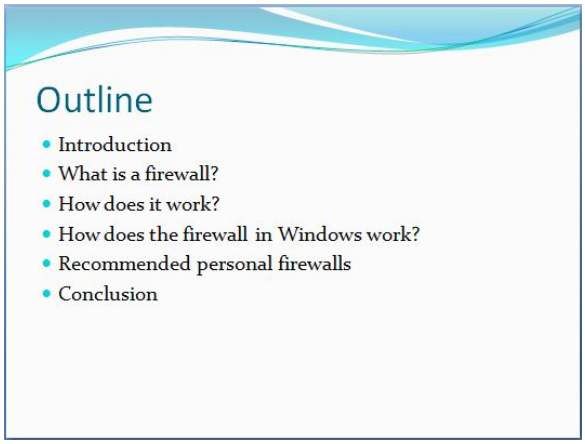


Figure 3.6: Outline of a second presentation about Windows firewalls

presentation has the subjects highlighted in Figure 3.7.

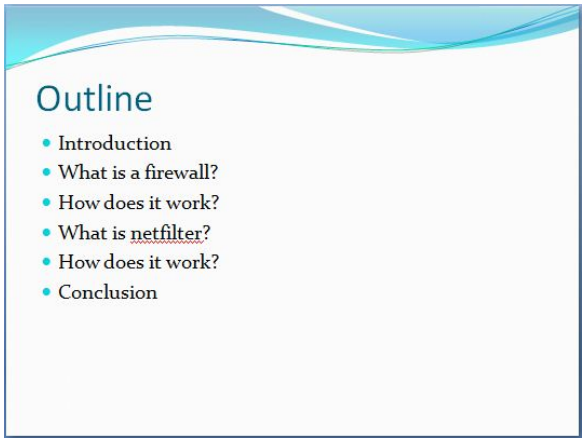


Figure 3.7: Outline of a third Linux firewall presentation

In that same year he must also give the same lecture about Windows firewalls. Again, since this presentation has been given two years ago, he recognizes that the slides 'What is a firewall?' and 'How does it work?' in his presentation are a bit out dated. He gives a look at the most recent presentation of Netfilter architecture in Linux to check the slides that have the same subject name and copies those slides into his old presentation.

In the previous scenarios, we have seen that doctor Kennon does quite some repetitive work. He had to copy and paste the same content of previous presentations in more recent ones. He also feels that his data is never up to date, even if he works on it. As seen in the previous scenarios, he has often worked on similar subjects (e.g. firewalls). In each new presentation, he has possibly updated some

sentences about those subjects from previous presentations. In the last scenario, he had to give a second presentation of a PowerPoint presentation he delivered a few years ago. There the content, especially the two often referenced subjects in his presentations, were outdated. He had no other choice but to look at a presentation that contains the same most updated subjects and to copy and paste the content into an older presentation.

The previous two scenarios are examples of daily work practices that could be accomplished in a different way. The current filesystems do not help us to organize our data in a way that we can remember or find those data more easily. In the first scenario, mister Fricker will probably on the long term have a complex folder hierarchy. The way how he gives names to subfolders are not necessarily intuitive, and the depth of subfolders can be of more than three levels. In this case, he has two choices. He can "dive" in the folder hierarchy and click on each subfolder until he meets the target, or he can make use of a search function of his operating system to find an item with a technical search criteria like name, date and file extension. The latter option can be a temporary solution to search in a small amount of data. When the amount of data becomes higher and search criteria like the filename are not relevant anymore, probably because the filename has not a significant name, the search process can take much more time and maybe end with no results. All the data is spread in two different parts, namely in photos and videos. Even if they have been created on the same place and have something in common, mister Fricker does not have a lot of options to make those relations visible. In this case, he can only try to give unique and meaningful filenames. In the second scenario, doctor Kennon cannot reuse or update data with minimal efforts. He is forced to search for PowerPoint presentations which have similar content and to update one of them manually.

In the next chapter, we introduce our RSL-based Associative Filesystem that can help to solve some of these problems.

Chapter 4

RSL-based Associative Filesystem

In this chapter we introduce the RSL-based Associative Filesystem (RBAF system) based on the RBAF model (which extends the RSL model introduced in Chapter 2), and explain how it can help to overcome some of the problems mentioned in Chapter 3. Other illustrative examples will accompany the different concepts of the RBAF system.

Existing filesystems have seen an immense improvement in storage space addressing and performance during the last years. We start to use solid state disks (SSD) where the read speed is higher than the traditional hard disk drive. But during the last years, we almost forgot that the task of filesystems is not only to store digital information, but also to be able to organize and retrieve the information. As mentioned in the introduction, we saw that while the storage space evolved together with filesystems, we needed a way to separate and categorize all data. We have introduced directories, also graphically known as folders in order to bring a minimum order in our personal data. An effort has been put to bring symbolic, hard links, shortcuts and NTFS junction points (which all have advantages and disadvantages). Most filesystems now support system-predefined metadata on files and folders which can be used to search and sort data. But still, we did not pay further attention on how to improve the organization and retrieval of data for users with their increasing number of files for every system.
We have chosen to build the RSL-based associative filesystem on top of an existing filesystem as a current solution. The reason is because we should give the choice to operating systems between using the functionality of the RSL-based associative filesystem or use their usual filesystem for backward compatibility, which is a shared opinion as in [15]. In contrast to [12], which proposes to study and come with a new

solution in the long term to replace the existing hierarchical filesystem, we provide a short term solution with the RBAF system. In the long term, we can implement the RBAF system as a new filesystem if all operating systems use it exclusively. We can compare the idea of working on top of an existing system with the *BumpTop*[1] application shown in Figure 4.1. This application works on top of the Windows and Mac OS X desktops. The purpose of the application is to enhance the user experience by providing a 3D illusion that their files and folders are on a real desktop. One can create files or folders, move them, pile them, hang photos or sticky notes on a wall, and so forth. The user is free to profit from the functionality offered by BumpTop, and abandon it if they are not interested to use the application anymore. In that case, the user will not benefit from the customizations made via BumpTop, but will still access their files on the desktop as usual.

Figure 4.1: BumpTop works on top of an existing desktop to emulate behaviors and interaction with the 3D desktop like in the real world[2]

4.1 Files and Folders

In the RBAF model, any object can be addressed via the abstract `entity` type, and should therefore be extended with three subtypes. The first subtype will be discussed in this section, whereas the two other subtypes in the next sections.

The first subtype is the `resource` type. A resource can be any instance located

[1] http://www.bumptop.com/
[2] http://blog.weebo.ro/bumptop-desktop-3d-asa-cum-nu-ati-mai-vazut/

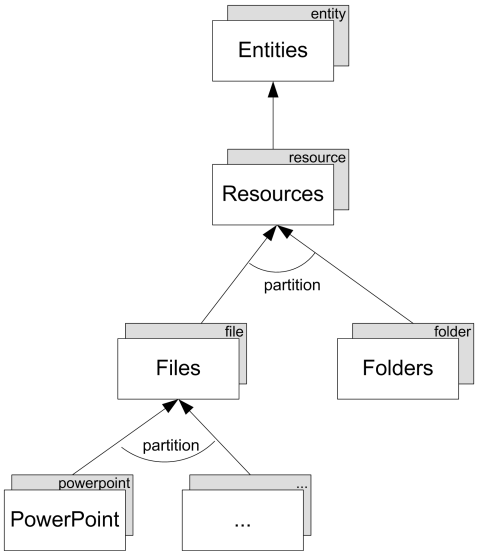


Figure 4.2: Files and Folders represented in the RBAF model

on the filesystem. It is an abstract concept that should also be extended with other subtypes. The RBAF model extends the RSL model with two different subtypes shown in Figure 4.2. The first subtype of the resource type is the `file`. It is a more precise type that addresses any file on the filesystem, and can access information such as system-predefined metadata (e.g. creation date). Although we can address any type of files, the `file` type in the RBAF model should be extended with concrete file types in order to know what file we are manipulating. An example of a concrete file is a PowerPoint presentation with the extension of `.ppt`. Beside files, the RBAF model also extends the resource type with the subtype `folder`. The `folder` type represents any folder that is located on the filesystem. Just as in most filesystems, we can for instance create folders, list or access their content such as files and other folders. Also system-predefined metadata such as the creation date of a folder can be accessed.

With system-predefined metadata, we mean that the user cannot add or modify metadata attached to a file and a folder in the current filesystem. For instance, we cannot add a property *artist* to an image file. In the next section, we will show that adding metadata to files and folders can be useful.

4.2 Metadata on Files and Folders

In the RBAF model, we represent metadata with the `property` type, which is associated with the `entity` type. A property gives a meaning or describes an

entity. It is a key/value string tuple. Users, the RBAF system or other applications can make a different use of properties depending on their needs.

4.2.1 Adding Properties to Files and Folders: Why?

As already mentioned before, folders help us to logically order data, but the drawback of this approach is that a file can be located in a deeply located folder (i.e. a file that is located in a folder, which in its turn is recursively located in another folder). Thus, once a file reaches a certain depth in a folder hierarchy tree, the user will probably never look after that file by recursively clicking on a folder one after the other. Therefore, search functionality is available in modern operating systems in order to quickly retrieve these files or folders, without browsing a complex hierarchy of folders. However, there are other related problems. First, every file can only be uniquely identified via its filename together with the filepath. In order to retrieve a file, we must remember its name. Giving a correct and understandable name is not always simple, especially when we have a group of files in the same folder, which have some similarities (e.g. what name does one give to multiple photos that show the same person in the same context such as on the beach?). In the example of the journalist named Charles Fricker from the first scenario in Chapter 3, we have seen that some photos were left with a default name followed by a number(i.e. photo1) because he did not know what name to give to the photo. Second, if we type in the name of a file in a search field, it often happens that we get a long list of search results. We probably have seen that there are even files that share the same keyword in their name, but have absolutely nothing in common. For example, suppose that we are looking for an HTML file. The only thing we remember is that it has the word 'web' in its filename. So we type in the keyword 'web' in the search field hoping to find it. Astonishingly, there is a long list of search results with files we did not know they are located on our hard drive. In that situation, we probably will have to browse the long list of search results to find the file that we are looking for.

To clarify, it is not easy to remember all files and search after them only by their filename. We should be able to remember a file according to aspects we remember the best. For example a photo that we took with our Sony photocamera. In order to describe a file in more than one way, we can use properties. The RBAF model allows a user to add properties to any file and folder (and also to other entities that are introduced in the next sections). Figure 4.3 shows how the properties are modeled in the RBAF model.

Returning back to the first scenario, the journalist Charles Fricker noticed that his data hierarchy started to become difficult to organize even with a little amount

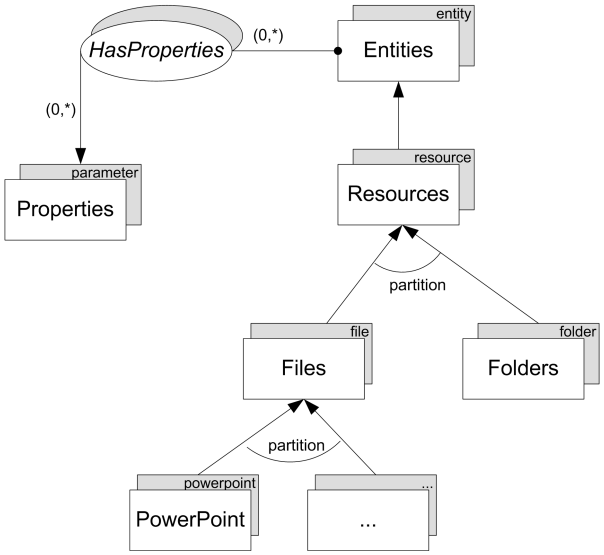


Figure 4.3: Properties in the RBAF model

of data. He can now annotate his files with properties and later use a search function in order to retrieve them more easily. We can compare this with *Youtube*[3] where a user can enter different tags as keywords when posting a video online. Those tags are very useful because when searching for a video, the entered keywords are used to look for a video which has video sequences that are described by tags [23]. This means that our search results will point more accurately to videos with content that we are interested in.

The same result can be obtained in the RSL-based associative filesystem, by using a search function that bases its search process on the properties of a file or folder. But are properties only useful for users? The answer is no since a user can add a property to a file with information from which computer it comes from (e.g. the computer in the living room), whereas an application can add a property to a file in order to identify it when the application opens the file. Even the RBAF system can take advantage of this, for example if a file in a temporary folder is older than 200 days and has not been read or modified during that period, then it should be deleted.

Properties can be used for other applications as well, as indicated in [4, 18, 22], where we can use properties of files to generate virtual folders, as a search result or for classifying the files depending on the search criteria. A virtual folder is a representation of a group of files that share some common properties. For example, searching for a file with the keyword 'dog' can be regrouped in a folder that contains all photos of a dog called Bobby, and in a folder with all documents about

[3]http://www.youtube.com

canine medicine if the user is for instance a veterinary. Figure 4.4 shows a possible graphical representation of virtual folders.

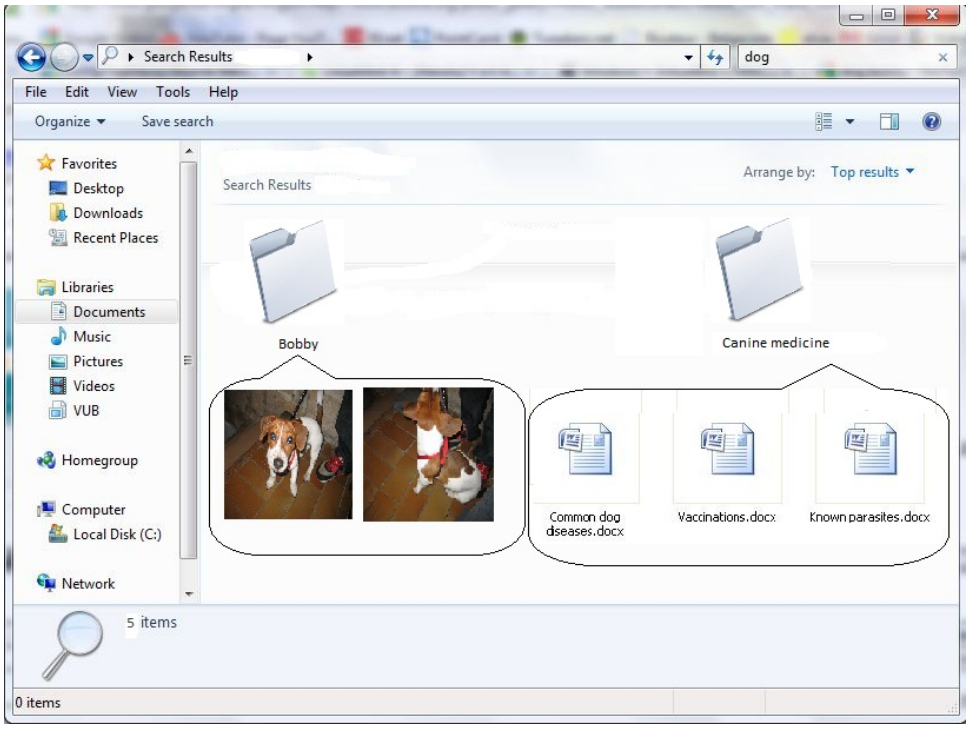


Figure 4.4: Virtual folders grouping the files whose properties match to the search criteria

Using properties is not the only feature in the RBAF system to retrieve and organize our data. In the next sections, we will introduce other concepts that can be helpful to relate and organize our data.

4.3 Semantic Links

In the previous section, we have seen a subtype of the `entity` type called `resource`, which has two subtypes: `file` and `folder`. The second subtype of the `entity` type is the `link` type. A link is a concept that logically connects entities, and is bi-directional. This means that we can see if an entity is the source or target of a link, and by which entity it is linked or what it is linked to. In contrast to one-way links, the fact that we, for example, know if an entity is being linked by another entity, can influence operations on a particular entity such as the decision to delete a file or not, depending on whether it is used elsewhere. Links are treated as first-class

[3]photo of dog taken from http://www.brolive.org/gallery/Photos_Skateboarders

objects [20], which means that they are not treated as simple metadata, but we can use them for other purposes as highlighted in the next sections. There are three different types of links. Therefore, the `Links` collection in the RBAF model is made abstract, and is meant to be extended. In this section, we first introduce the subclass called `SemanticLinks`. Figure 4.5 shows how links and semantic links are modeled in the RBAF model. Other types of links will follow in the next sections.

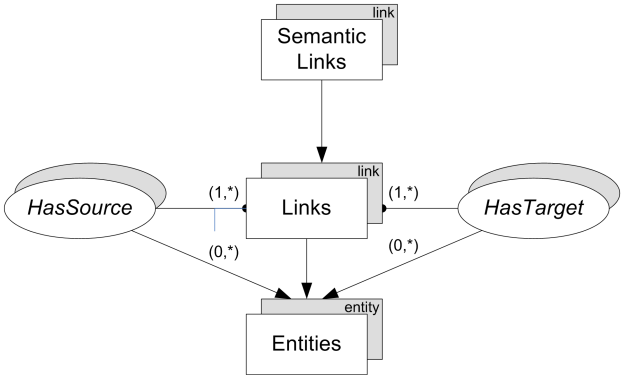


Figure 4.5: Semantic links in the RBAF model

The semantic link concept is used to semantically associate entities. We can, for instance, link any file or folder, including links themselves. With the word "semantic", we mean to relate any data on a high level. This data is probably not meant to fit together on the first view, but the relations form a new source of information and make a more flexible organization of files and folders possible.
Let us take two examples. We can semantically link a video of our friends of their vacation in China, to an e-mail that we have received earlier from them during their journey in China. The e-mail body describes events they have experienced, and the video shows those events they have filmed. While it may seem strange to "couple" a video together with an e-mail (i.e. when viewing the video we see that an e-mail is linked to it), it does make sense if we want to remember that some sequences of the video are described in the body of the e-mail. For a second example, as illustrated in Figure 4.6, suppose that a user has two different folders: one containing documents about mathematics and another about biology. They know that they need to consult documents about mathematics when reading biology documents in order to understand some calculations. Currently, there is no other way to organize files and folders but with folders. It does not seem logical to put the folder containing mathematics documents within the folder containing biology documents (because they are for instance two distinct courses), or the other way around. If they are not sure how to organize files in that situation, they can create a semantic link between the two folders. Without creating or moving folders, the semantic link "couples"

the two folders, and thus the folders are organized in a flexible and logical way. Semantic links can also be handy when one needs to associate their files with other people's similar files (i.e. "keep an eye") if they are working together on the same project (for example in an enterprise network), similarly as in [3].

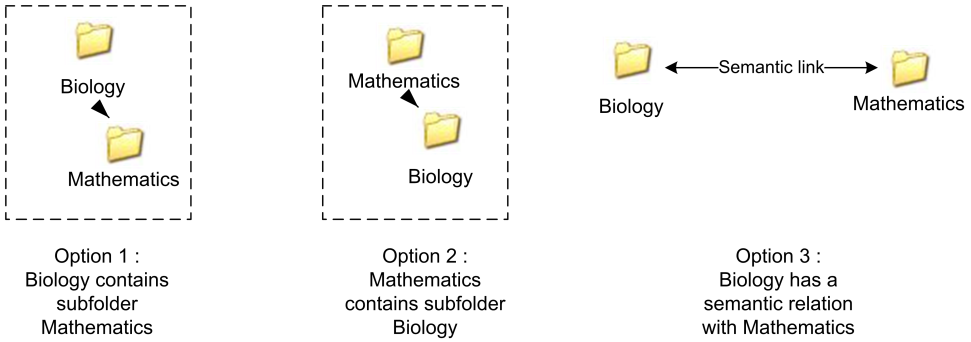


Figure 4.6: Instead of putting the Mathematics folder in the Biology folder the other way around, we can create a semantic link between the Biology and Mathematics folder

A semantic link has the purpose to link data in order to semantically unify, organize, and view it in different aspects.

4.4 Content Recycling

There are common problems present in most filesystems that pollute our data organization and waste some space on our disk drive. When we surf the Web, our web browser creates a lot of temporary files in a temporary folder. Most of these files are only required for a short time. There exist some applications such as *Ccleaner*[4] that analyze the folders containing those temporary files and delete them. Temporary files are not only created by applications, but also by users (such as pictures that have been downloaded, and copied and pasted in a document). Those files are not necessarily localized in a single folder, but probably in many other separated folders. After some time, we do not remember their presence anymore, and thus we leave them on our disk drive. For example, suppose that one has to write a document containing a lot of diagrams. They gradually create diagrams in an application such as Microsoft Visio 2007. If they draw thirty diagrams, they have to manually save every diagram as an image file (in order to use them in another application). After that, they have to copy and paste those images one by one in

[4]http://www.piriform.com/ccleaner

their document.
The first consequence of this way of working is that we waste space of our disk drive just to make images of diagrams, while they already exist in our Visio 2007 project file (i.e. `.vsd`). This project file takes much less space than all created images together. For example, a `.vsd` file containing thirty diagrams will more or less have a size of 416 KB. Grouping the thirty created images (in the PNG format) together, and knowing that each image has a size of 145 KB, will result with data with a total size of 4,24 MB! This number may seem innocuous for those thirty images, but we may create higher quality images other than diagrams (such as family photos), which each can have a size of 4 MB. The number of those high quality photos can be more than thirty.
The second consequence is that we are polluting our images folder with images that are not useful outside our document. In other words, our folder contains images that we do not even wish to see or organize.
Third, we have to copy and paste each image into our document, which means that we merge them with our document to extend its size. This is again wasted space on our disk drive space.

To clarify, we are not using our data efficiently. We create temporary, irrelevant or duplicate files that take space, and that are not easy to organize. We probably will never look for those files when we do not need them anymore. Thus, we just abandon them in our collection of files.
To solve this problem, we introduce the concept of *content recycling*. The purpose of content recycling is to grab the content of files, and reuse (parts of) that content in other files by linking to it (and not recreate that same content again or merge the content with any file). In this section, we show how the concept of content recycling works.

As we have seen earlier, the entity type has until now two subtypes: the `resource` and `link` type. The `entity` type has also a third subtype, which is the `selector` type. A selector is a concept to select parts of the content of a file. It is an abstract concept and should be extended with other concrete selector subclasses. Figure 4.7 shows how concrete selectors are modeled in the RBAF model.

The concrete selector subclasses are meant for specific file types (i.e. subtypes of the `file` type in the RBAF model), and are created and accessed via plug-ins, which are developed by developers and plugged-in to the RBAF system. The selector contains metadata about the selected part(s) of the file, such as the coordinates of the selected part of an image.

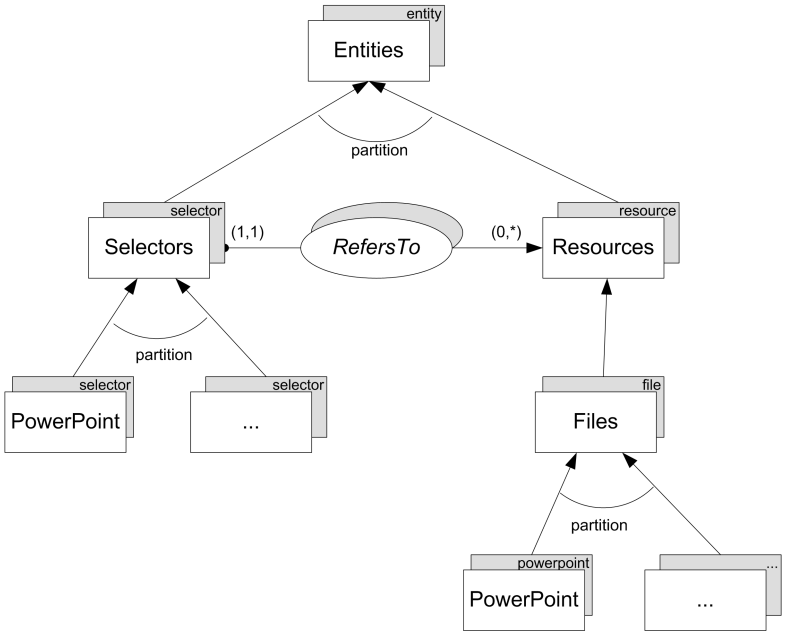


Figure 4.7: Concrete selectors associated with file types

The general architecture of how applications communicate with the RBAF system and plug-ins is shown in Figure 4.8.

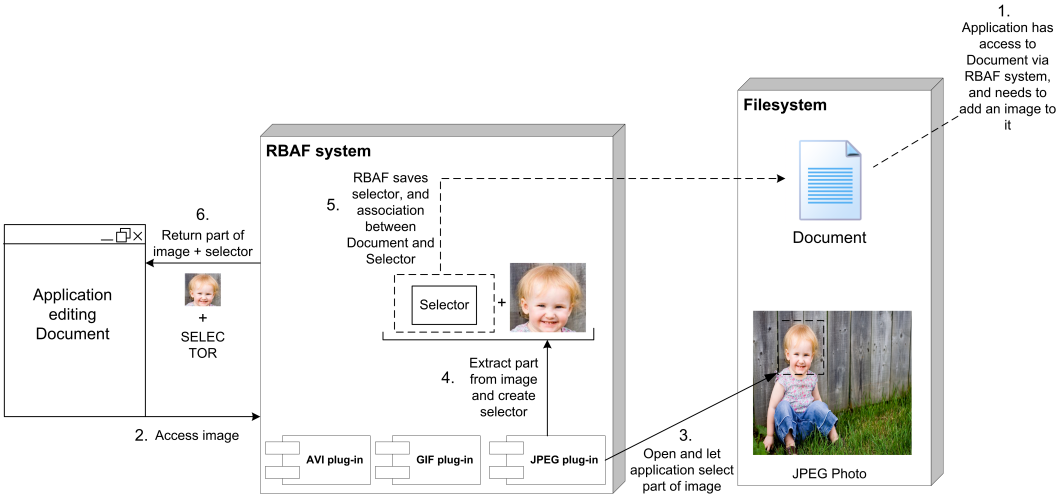


Figure 4.8: RBAF system architecure communicating with applications and plug-ins[5]

Applications always communicate with the RBAF system when they want to

[5]photo of child taken from http://www.mamaenzo.nl/images/redactioneel/Kinderen/plus 1 jaar/Kind_water_de_vijver.jpg

access (parts of) files. We will call external files that an application wants to access its content and to select parts of it (such as an image), *alpha files*, and files that applications are editing (such as a document), *beta files*. The RBAF system in its turn will communicate with the appropriate plug-in, which will provide the functionality to the application to select and access (parts of) a alpha file (for example, a GUI window shows the content of an image, and the user can use a tool such as a dotted square to select parts of it). Once the application has selected the part of the alpha file it needs, the plug-in will extract the part of the alpha file (as a stream of bytes), and create a selector for the alpha file. The RBAF system will save the selector and the association between the selector and the beta file located on the filesystem. Note that it does not merge the selector with the alpha file or beta, nor does it save the association in the filesystem: the RBAF system saves the selector, and an association (as a reference) between the selector and the beta file within the RBAF system itself. Next time, when the application opens the beta file again (still via the RBAF system), it will see that there is a selector associated with the beta file, and ask to the RBAF system, which in its turn will ask to the plug-in to read the selector in order to give access to the specific part of the alpha file the application selected the previous time (so the selector contains information about which parts of the file to select). The plug-in will then open the alpha file and return a stream of bytes of that part of the file to the application.

Separating the plug-ins from applications has the advantage to make both independent of each other. One might want to update an older plug-in with a newer one if a new file type has come out, or alternatively add a newer version of a plug-in and leave the older one if we need to work with older and newer file types. We will illustrate an instance of the working of the architecture with an example, also shown in Figure 4.8. Suppose that a developer has built one plug-in that can select parts of JPEG images, and another plug-in for GIF images. He also has created a plug-in to select parts of AVI videos (selecting parts of a video is for instance selecting time sequences). All three plug-ins have been plugged-in to the RBAF system. At a certain moment, a document editor application, once passed through the RBAF system, wants to get access to parts of a JPEG image. The RBAF communicates with the JPEG plug-in, which will let the application select a part of a JPEG image (for example with a dotted square). The plug-in extracts the part of the image and creates a selector. The RBAF system saves the selector and the association between the selector and the document. The application will receive a stream of bytes that are the part of the file, and render it in its own view. Next time, when the application opens the document, it will see that there is a reference to a selector. Therefore, it will do the same process again where the plug-in will

be asked to read the selector to select the part of the file the application needs to access and return the part of the image to render it (as a stream of bytes again). The application does not have to select the part of the image with the intervention of the user again) .

Until now, we talked about "accessing" parts of files and "holding a reference" between a selector and a file. But how does a file point to (parts of) another file and structure its content? And what information do we save about the association between the file and its selectors? We will answer these questions in the next subsections.

4.4.1 Structuring File's Content, and Metadata on Links

We have mentioned earlier that the `link` type has multiple subclasses. One of those subclasses we saw was the `SemanticLinks`. Now, another subclass of the `link` type is the `StructuralLinks`. The idea is to create links from alpha resources(which we call here to nominate source resources) that point to other beta resources (which we call here to nominate target resources) such as a paragraph in a document or images, structure those links and render the linked target content of the beta resources in the alpha resources. It is an abstract concept and is extended with two subclasses. One of those subclasses is the `StructuralFileLinks` shown in Figure 4.9.

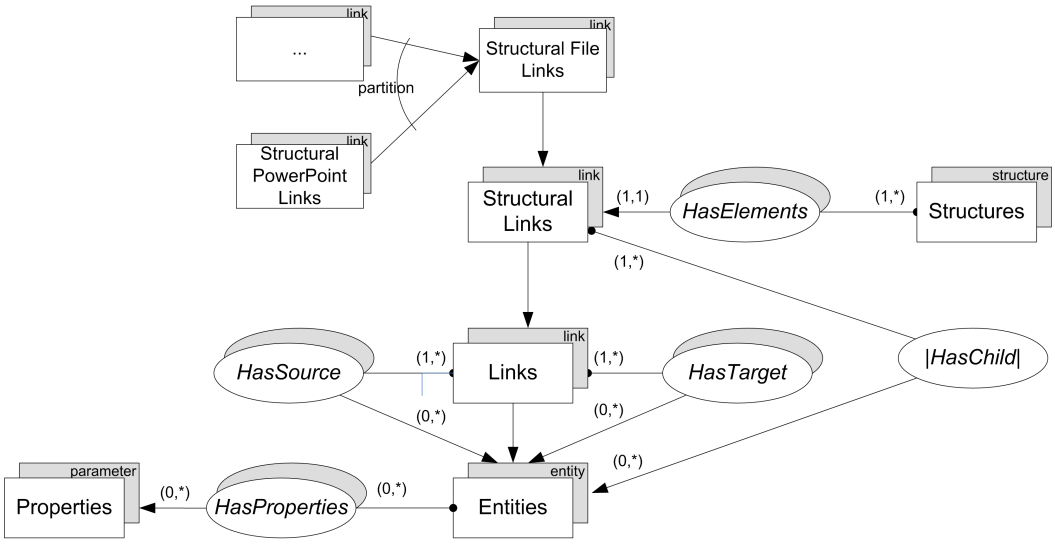


Figure 4.9: Structural filelink

A structural filelink associates a file with the content of another file in a structured way. Thus, those structural filelinks let other files to be composed with other

resources (e.g. images) by linking to them. The `StructuralFileLinks` class can be extended with other subclasses specialized for other file types such as PowerPoint presentations. In such a case, we call this class a `StructuralPowerPointLinks`. Creating specialized structural filelinks can be useful for example when we want to refine a search process to find only structural powerpointlinks. Another thought is, for example, to be able to classify structural powerpointlinks. We will refine the first example of this section to explain the interest of structural filelinks combined with selectors. Suppose that a biologist has to write a document about the lifecycle of frogs. Beside including diagrams that represent statistics, he also needs photos about frogs to include in its document. Therefore, he goes to a river multiple times to take photos of the evolution of the frogs in six stages: egg, embryo in egg, tadpole without mouth and eyes, tadpole with mouth and eyes, tadpole with back legs, tadpole with front arms, and finally lost of tail to become a full frog. After having taken enough photos, he starts to write his document. All diagrams (saved as images) and photos are saved in a folder on the disk drive. Meanwhile, he adds diagrams and photos of the frogs. But in a part of his document, he wants to have frog images that are zoomed versions of the original ones. So the biologist must edit the photos in an image editor, crop the part where only the frog is visible, zoom in and save it as a new image. Finally, the images are copied and pasted into the document.
From this example, we first see that the user has to open another application, edit the original image to make a new one, and copy and paste it into the document. Second, if he does that task of creating new images and edit them for other documents, he will end up with a lot of irrelevant images in his images folder. We can instead let the user open the image, select the part he wants, zoom in and link to that aspect of the image, which will be rendered in the document. Note that the part of the image will not be merged with the document, but referenced and finally rendered in the document. The user is not limited to link to parts of images, but also can link to the whole image and render it in the document. The link from the document to (parts of) the images is modeled via structural filelinks. An example of a WYSIWYG editor that edits a document which is structurally linked with images is shown in Figure 4.10.

Properties are added to the structural filelinks (such as if we are pointing to a selector) and those selectors contain information about where the image needs to be cropped in pixels and the zoom factor. When the document viewer will see that there is structural filelink to a selector, it will ask to the plug-in, via the RBAF system, to read it to get the part of the image zoomed in, and render the received part of the image in its turn. Figure 4.11 shows that the document has a

[6]photo of frog taken from http://www.ristohurme.com/insularum.htm

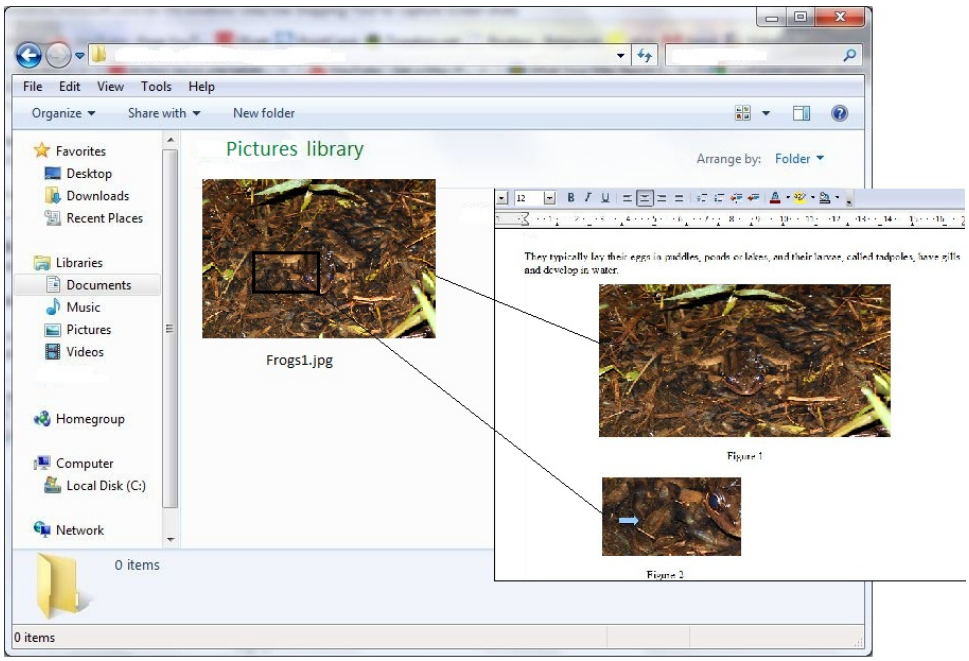


Figure 4.10: A WYSIWYG editor edits a document structurally linked to an entire image and to a part of the image[6]

structural filelink to the selector, which describes how to access parts of an image via a JPEG-plugin.

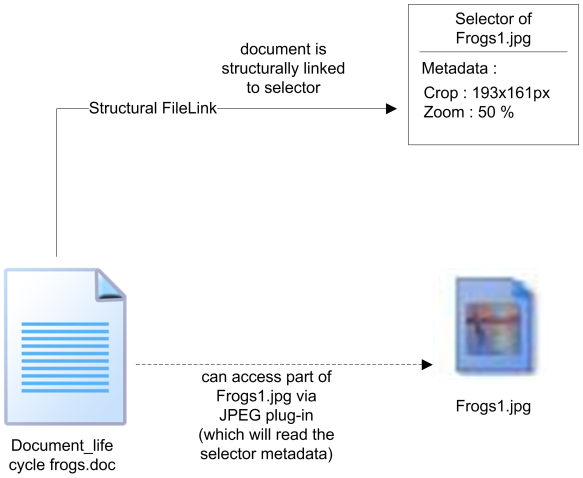


Figure 4.11: A document has a structural filelink to a selector. The document can access parts of an image via a plug-in, which will require the selector's metadata

The user sees the document, with its own content (e.g. text) and structural links to images, as one file. But what do we mean with "a file that is composed of the content of another file"? Figure 4.12 *illustrates* in UML (it does not strictly reflect

the structural filelinks in the RBAF model) the structural filelinks: any (part of a) the content of a file can be part of any other file.

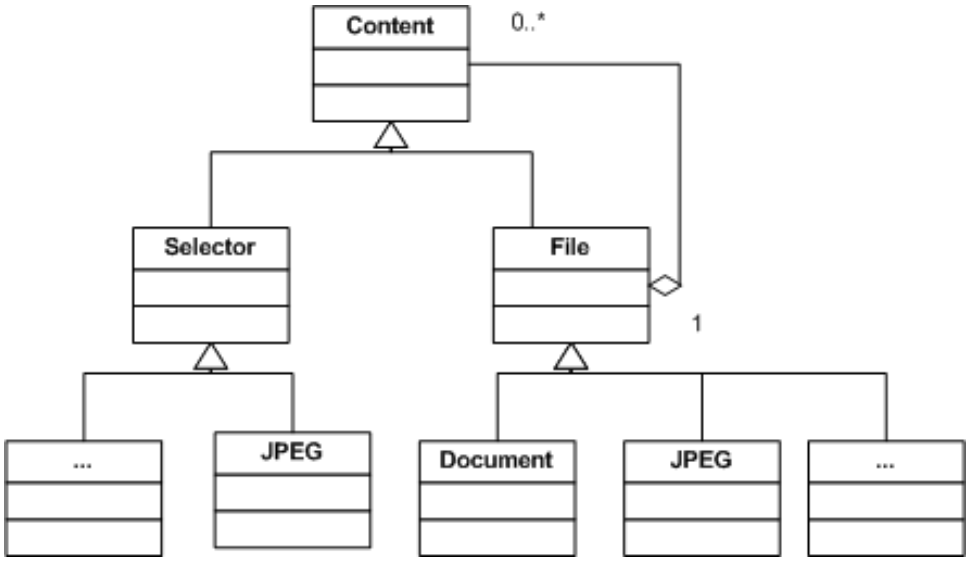


Figure 4.12: UML representation of structural filelinks

We can use structural filelinks not only to compose documents, but also to compose other types of files such as linking to parts of an image within an image itself, in order to enrich the image. The idea of structural filelinks is similar as in [17], but we use them also for other files than digital documents.
Using structural filelinks, a file can be flexible and change easily. Suppose that we work in a WYSIWYG document editor and decide to update an image A, we can then change the link from image A to image B, instead of deleting image A and copy and pasting image B. Returning to the scenario of Dr. John Kennon in Chapter 3, some of his PowerPoint presentations are composed of parts of other PowerPoint slides. Using structural filelinks, he does not have to copy and paste text from a slide to a presentation because its presentation is updated if the slide to which it is linked to, changes.
It is not excluded to compose files from the content of other files that is structurally composed of other contents (or selectors), as we have seen in the background chapter of the RSL model where a resource can link to a collection of structures.

4.4.2 Layering Selectors within Files

Consider the following situation: suppose that one has a document that contains text, and they are interested to link multiple parts of that document *to* another document (in other words, they want that some parts of the document are accessible to other documents). For example, they want to link a paragraph that contains ten

lines. First, they select line 1 to line 4 and link that selection to one document, and second they select line 3 to line 10 and link it as well in another document. How can two selectors overlap each other and how can one visually distinct between two overlapping selectors? We can solve that problem with the concept of *layers*. A layer is used to encapsulate a selector of a specific file. The selector linked by a file or by a part of a file, and is on a layer. The file knows which layers it contains and the order of the layers. Now that we achieved that a document can have multiple selectors linked from other files, suppose that one would like to show or hide some selectors in the document. For that purpose we can activate or deactivate certain selectors in order to obtain the exact information we want via the `ActiveLayers` collection. To illustrate the usefulness of using layers, suppose that we have written a document A. We remember that we have another document B that we find interesting, and thus we decide to link to a paragraph and select the first four lines. Then in another part of the document, we would like to refer to the same paragraph of the other document, but starting from line 3 to line 6. In the current document B, by using a WYSIWYG editor, we can render the differentiation of the selections, which would be the task of a plug-in to add metadata to the selector that is overlapping another, and the application shows two tabs on the right of the paragraph. Each tab would have a distinct color. When clicking on one of those two tabs, the corresponding selector will show up and show the text that is selected from the other document. Figure 4.13 shows in a first instance the above text addressed by a selector. The selector is visually rendered when the top tab is selected. Figure 4.14 shows in a second instance the text below, which is also selected by a selector and is visually rendered when the bottom tab is selected. The point is that overlapping selectors can be visually rendered.

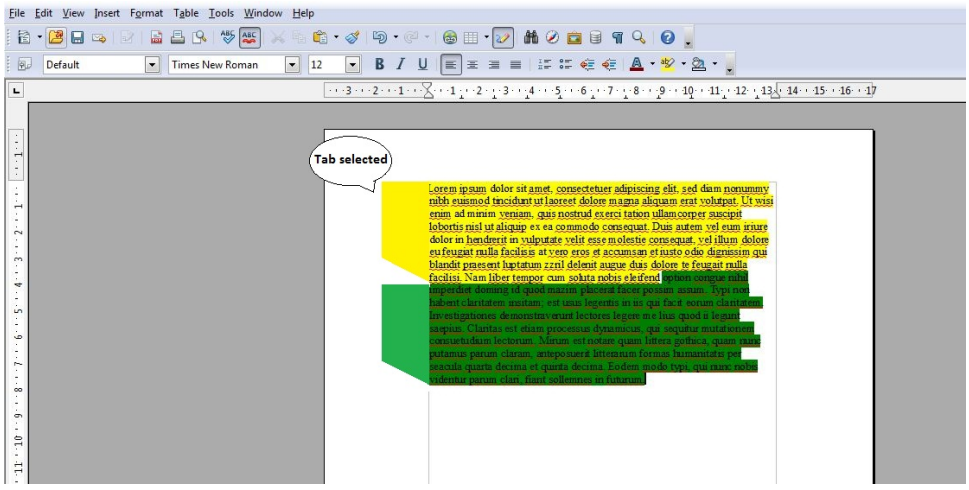


Figure 4.13: The above selector is selected and shows the text with the top tab

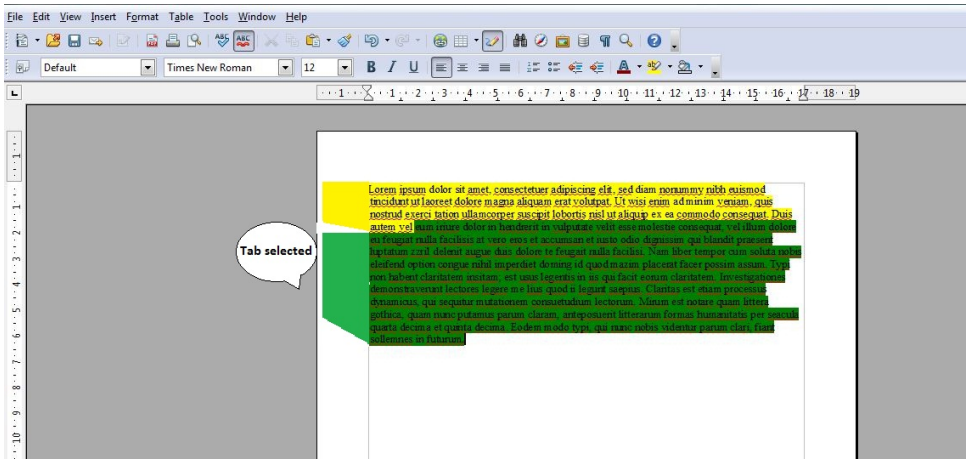


Figure 4.14: The bottom selector is selected and shows the text with the bottom tab

4.5 Multiple Classification

In the previous section, we have seen that a the `StructuralLinks` collection has a subclass called `StructuralFileLinks` and what it is used for. It also has a second subclass called `StructuralFolderLinks` shown in Figure 4.15.

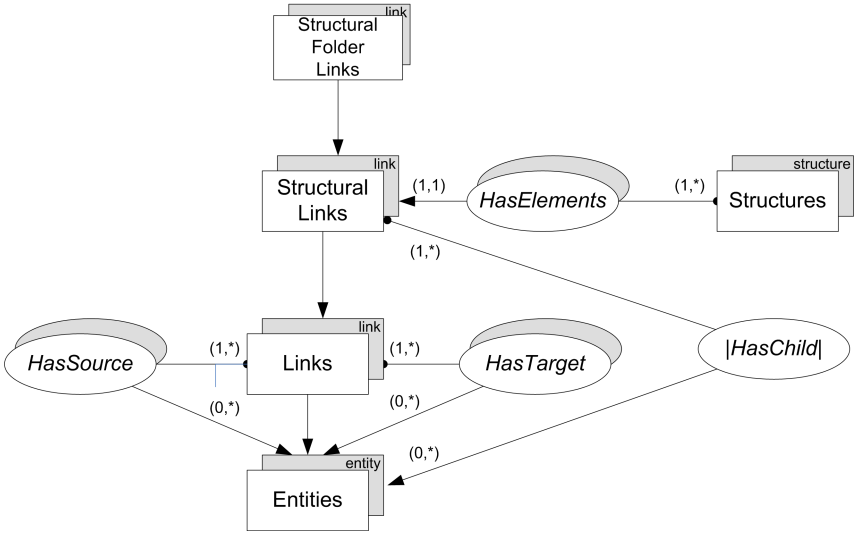


Figure 4.15: Structural folderlinks modeled in the RBAF model

This concept has the purpose to provide extra flexibility for folders, when organizing files and folders within folders. Currently, we are limited to save our files and folders in one folder at a time. As it is stated in [14], we can only localize files and folders within a unique path, which prevents us from classifying our files and folders in more than one way (i.e. in more than one folder). With structural folderlinks,

we can classify files and folders in multiple ways within different folders at the same time. We call that *multiple classification*, which is similar in [16, 7]. A folder can therefore point to the files and folders located in other folders, as if they owned those files and folders. The pointers are modeled with structural folderlinks. With multiple classification, we can avoid the problem of long filepaths (in other words, files or folders located in a deep folder hierarchy), and therefore not force the user to recursively create subfolders within folders. The user has also the possibility to reach their files or folders from different locations (from other folders) and classify them in a more flexible way. In the first scenario in Chapter 3, Charles Fricker can use multiple classification to create folders that will contain (i.e. point to) some of those same files, instead of moving files from one folder to the other each time a new (sub)folder is created. For example, instead of creating a folder Riccione and Coriano and put them as subfolders inside the folder Rimini, and moving some files to Riccione and other ones to Riccione, he can leave all his files in the folder Rimini, place the two folders Riccione and Coriano "on the same level" as the folder Rimini (i.e. they are not subfolders of Rimini, but next to it) and let them point to specific files that are located inside the folder Rimini, as shown in Figure 4.16 (the arrows are structural folderlinks). The advantage is that the user no longer has to browse a deep folder hierarchy, but can click on the corresponding folder (let us say Riccione) and immediately go to the photos he is looking for.

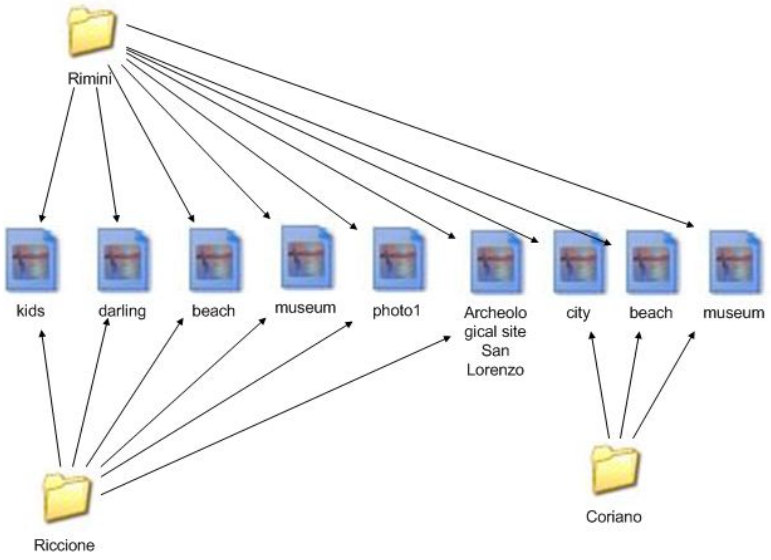


Figure 4.16: Multiple classification of pictures

4.6 Content Traveling

Until now we have seen that there are three types of links: semantic links and structural file- and folderlinks. The last subclass of the `Links` class is the `Navigational-Links` class shown in Figure 4.17.

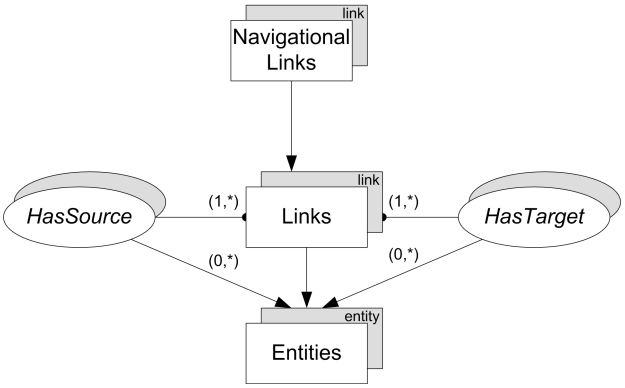


Figure 4.17: Navigation links

We can use a navigational link for any entity, for example to travel between the content of different files, from one file to the other and vice versa. A file can have multiple other navigational links that point to different files, which gives us the possibility to follow other paths. As we have seen in the background chapter about the memex, Bush said that a page can be linked with a second page. The second page can be in its turn linked with another page, and so on. The utility of navigational links is to follow a trail and navigate between entities (which can for example be files or selectors) when we are looking for information contained in the linked entities one after the other. We can compare a navigational link with a hyperlink from the Web. We can navigate from one page to another by clicking on the corresponding hyperlink. Websites do not necessarily deal with the same subject, but they can contain paragraphs in their webpages that have a certain relation. Navigating between different information and organization of data is called *content traveling*, as illustrated in Figure 4.18.

We can also create navigational links between different folders in order to follow a frequent path when we browse from a folder to a (sub)folder. Also the system can make use of content traveling to keep track of a navigation behavior of a user, and suggest that same behavior to the user later. For example, a student often browses within the same folders and subfolders to look after documents. When browsing the folders, the student is not aware that the system is tracking his navigational behavior. Next time, the system can show to the student his navigational behavior

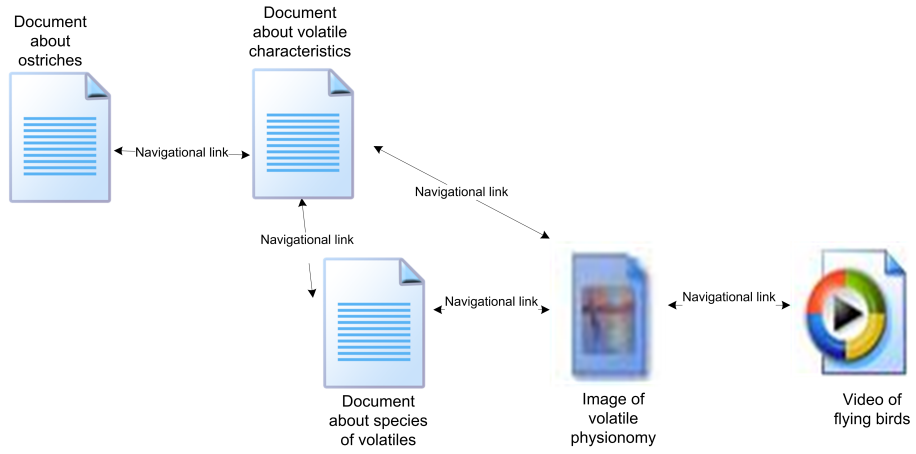


Figure 4.18: Illustration of navigational links between different files

and choose between different suggestions. The system will show to the user all the files or folders he was the most interested in, and let the student choose between the suggestions to immediately access what he desires. Figure 4.19 shows the order of the student browsing a folder and its subfolders. A possible graphical representation of the system's suggestions is shown in Figure 4.20.

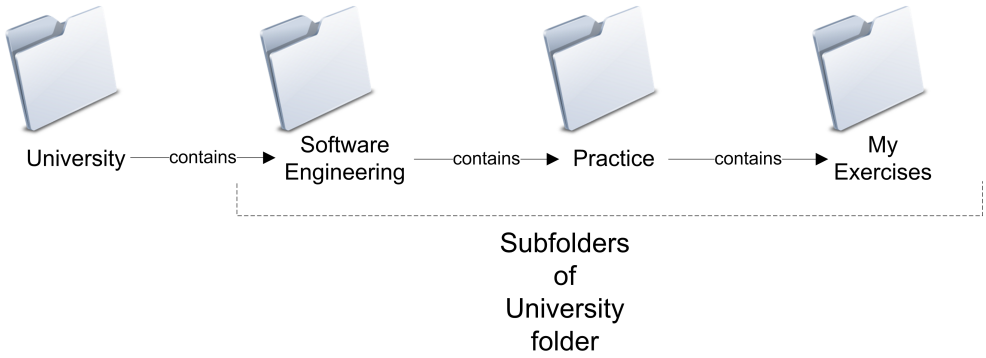


Figure 4.19: The student sequentially browses from folder to subfolder

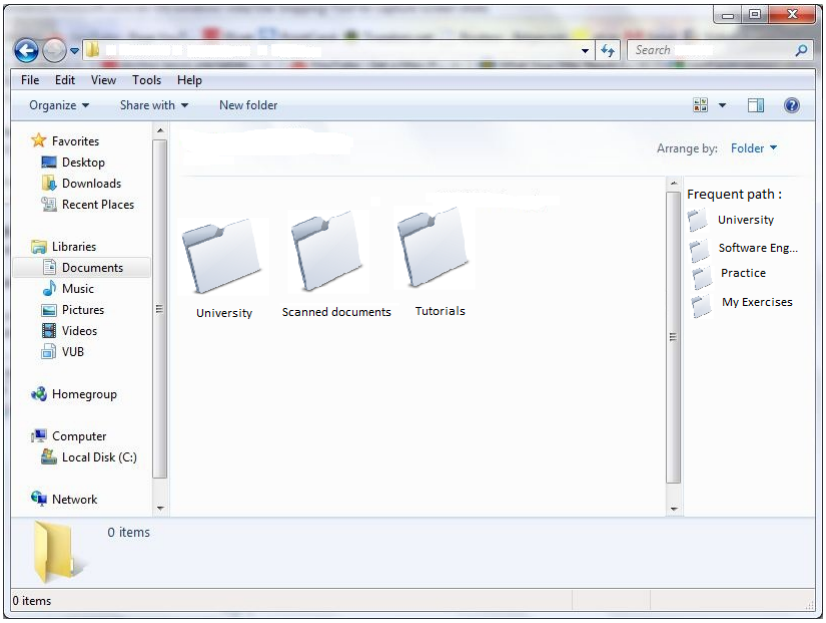


Figure 4.20: The system suggests folders (in the right corner) based on the students's previously tracked navigational behavior

4.7 Advanced Linking

Links in the RBAF system are very flexible, and can be used for different purposes such as structuring and organizing data. But we may think that one particular type of link, the semantic link, is a weak concept that does not bring much to organize our data.
We said earlier that we can link not only files and folders, but also links themselves. We can for instance use links to annotate any files, and create links whose source and targets are defined by other links. We will illustrate that with a more complex example. A first illustration is shown in Figure 4.21.

Suppose that a developer works with the Eclipse IDE for Java on a Java project called `AccountancyApplication`. This application has the purpose to manage the accountancy of a company. Recently, the developer has discovered by reading a book that there exist an API to compute interesting XML operations which can be used for the `AccountancyApplication` project. Therefore, they create a new Java project called `TestXmlAPI`, and import all the `.jar` archives that they downloaded from the Internet to that project and test the API. In the `TestXmlAPI` project, the developer often adds new Test-classes and does some other experiments because they gradually discover other functionality of the API that they can use in the `AccountancyApplication` project. Because the developer often uses the `AccountancyApplication` project with the `TestXmlAPI` project, they want to

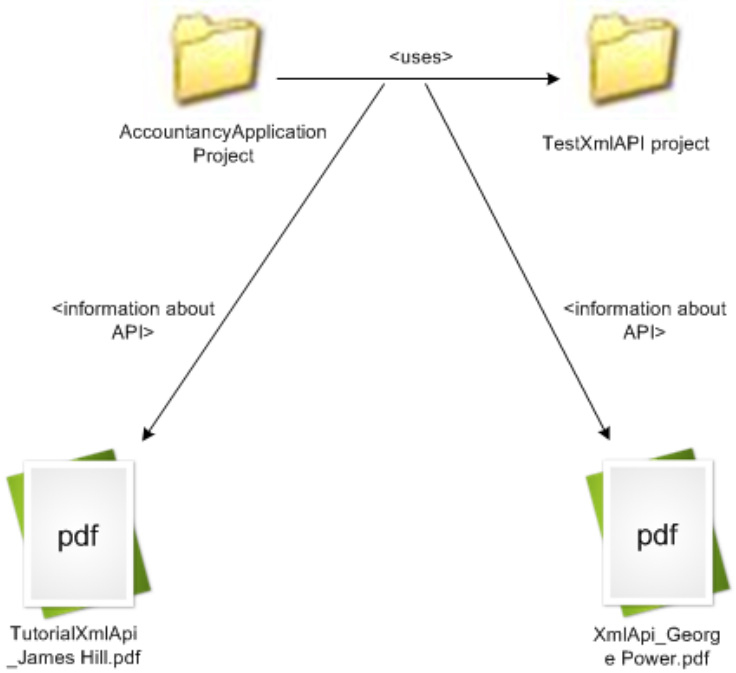


Figure 4.21: The two Java projects are semantically linked together. Another link has been attached to the previous link as an annotation

make clear that both projects are related, and therefore they semantically link them. The developer has also found two PDF documents on the Internet that are tutorials explaining other functionality of the API used in the `TestXmlAPI` project. So the developer decides to annotate the semantic link between the two projects with two other semantic links, each pointing to a PDF document. If the developer does not work on the `AccountancyApplication` project for a few weeks, and reopens the project again in the Eclipse IDE, the semantic link between the two projects will remind the developer that the `TestXmlAPI` project is used together with the `AccountancyApplication` project to test some functionality of the API. The semantic link between the two projects is also semantically linked to the two PDF documents that have been downloaded the previous time, and the developer will see that those documents are tutorials about the XML API. Also in the other way around, when browsing through their documents, they can see that the documents are linked to the semantic link between the two projects (because links are bi-directional). Thus, the developer knows why they have downloaded those PDF files. Figure 4.22 shows a possible graphical representation when a developer wants to view the semantic links between files and projects in the Eclipse IDE.

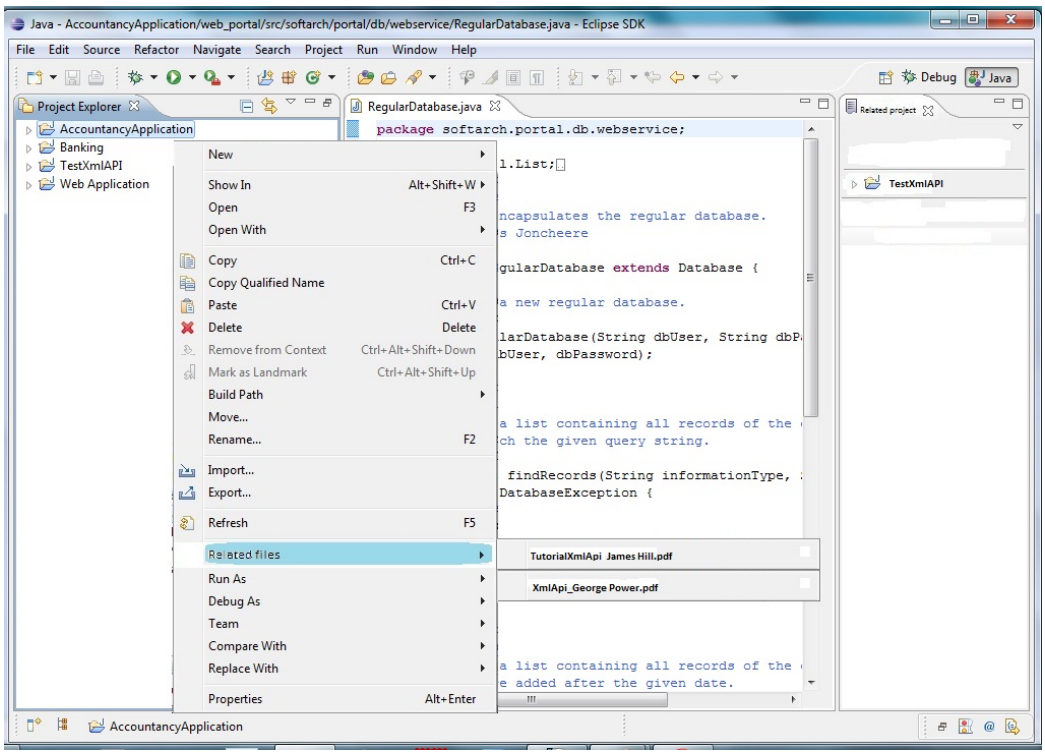


Figure 4.22: The two Java projects have been semantically linked together. Two other links have been attached to the previous link as annotations

4.8 Content Outsorting and Access Rights

With the help of properties, links and selectors, we can efficiently organize our data and enrich our data with other sources of information. However, in some situations we would like to include or exclude information from other sources. A file for example can have a certain association with other files with specific properties that are not relevant in some contexts. As we have seen in the background chapter when we talked about *context resolvers* from the RSL model, we now extend the `ContextResolvers` collection in the RBAF model with a subclass called `ContentOutsorters`, as shown in Figure 4.23.

This concept is used to adapt our data depending in which situation we are, or what we desire to adapt. A content outsorter is accessed by the RBAF system (or an application) to determine if an entity (such as a file or a link to a selector) should be visible or not. Suppose that we regularly want to view specific photos we have taken in Blankenberge, at the Belgian coast, and we only want to see photos where a friend called James is visible. Instead of searching those photos via a search functionality, or creating a folder to classify all the photos of our friend by selecting them one by one, we can create a content outsorter where the

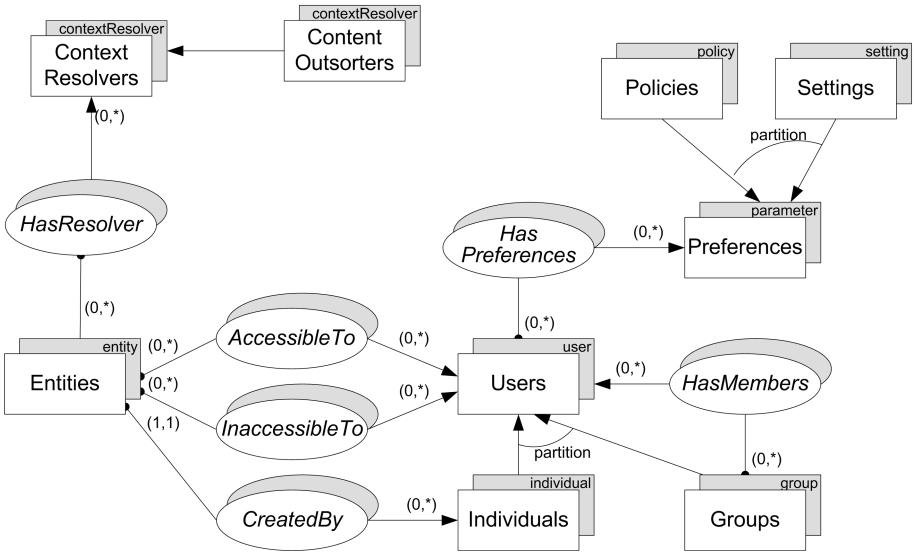


Figure 4.23: On the left side we see how the content outsorters are modeled. On the right side we see how access rights are modeled

view conditions can be based for example on properties that the images contain like *James* or *Blankenberge*. A possible way to implement content outsorters is to use an XML configuration file and defining for which (type of) file, together with its properties, the content outsorter should be applied on, and which properties should be excluded or included. Beside properties, we can also base the content outsorter on other criteria like structural filelinks to include or exclude. The user will be able to define their own criteria with an appropriate graphical interface to configure a content resolver. In Listing 4.1, we can see an example of such an XML configuration for a content outsorter.

Listing 4.1: contentOutsorterForJamesPhotos.xml

```
1  <contentoutsorter>
2  <name>Photos of James</name>
3  <description>Extract photos of James</description>
4  <resource>
5    <type>folder</type>
6    <name>My Images</name>
7  </resource>
8  <resource>
9    <type>image</type>
10 </resource>
11 <include>
12   <properties>
13     <property>user:vacation</property>
```

```
14        <property>user:blankenberge</property>
15        <property>user:James</property>
16      </properties>
17  </include>
18  </contentoutsorter>
```

In the XML file, we first give a name and a description to the content outsorter. Then we define on which group of files and which type of file we want to apply the content outsorter. In this case, we only want to view images located in the folder My Images. Next, we define the criteria (which in this case is based on properties) that the content outsorter must consider.
Once the content outsorter has been created, it is seen by the user who can click on it and view all the photos of our friend in Blankenberge, as shown in Figure 4.24.

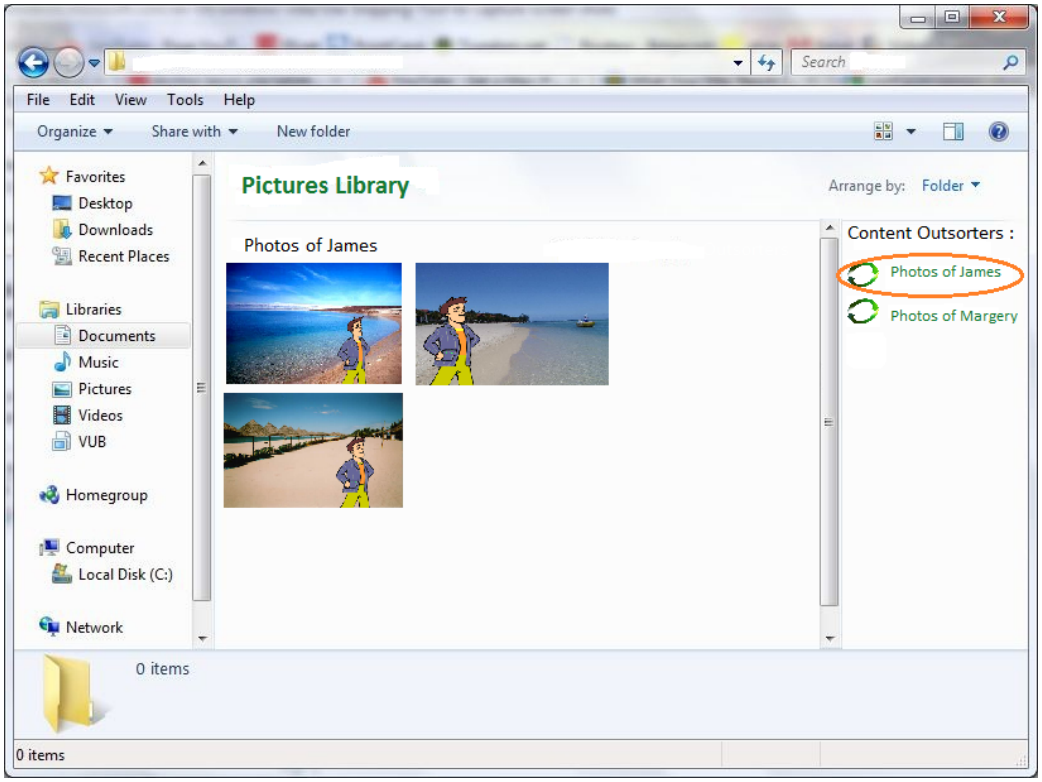


Figure 4.24: On the right-hand corner we see content outsorters associated with the My Images folder (which is located in the Pictures Library)

Beside adapting the presentation of our files, we can also give access rights to our files.

The RBAF model extends the *preferences* concept with two other concepts, as shown in Figure 4.23. The first subtype is the *policies* type. A policy is a rule

which defines the ownership and access rights to entities. The second subtype is the *settings* type. A setting defines a special requirement in which way and what is needed when an entity should be accessed by a user or an application. For example, a user might define a setting when they want to have access to files located in an external data storage if the local ones are not present on the current data storage. Applications on the other hand can load other defined settings in order to know what to do with a file.
We will illustrate with an example that it is possible to combine access rights with adaption of the environment depending the user, as shown in Figure 4.25.

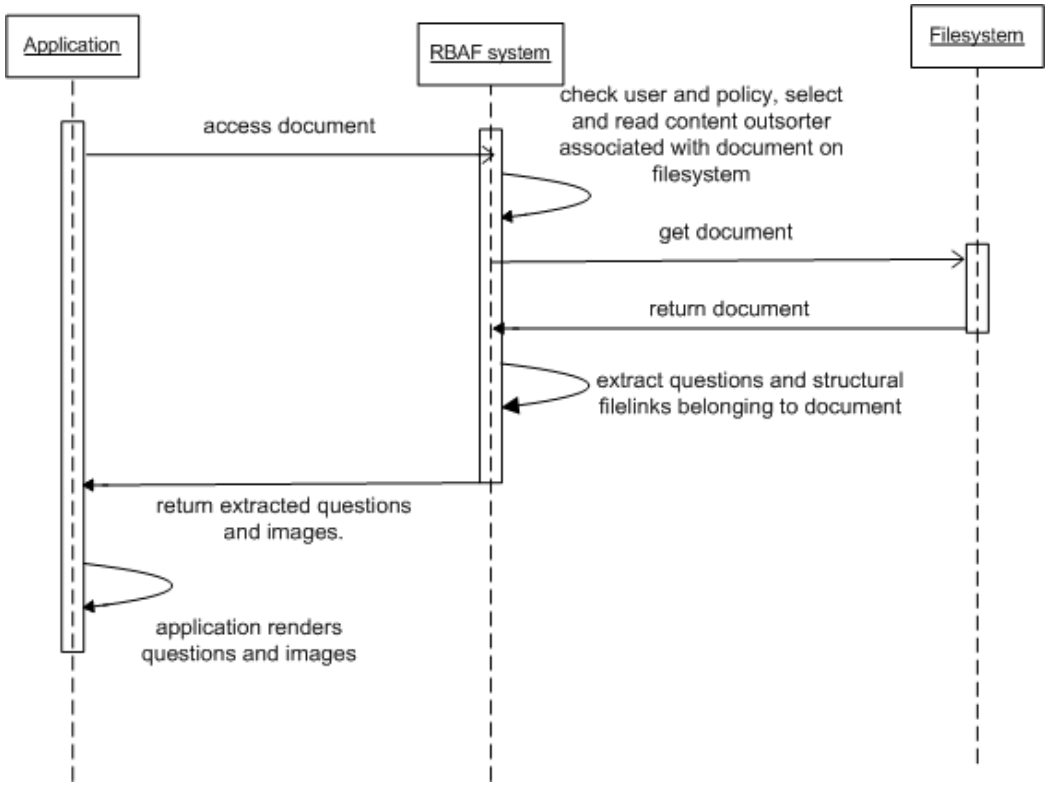


Figure 4.25: The document's content presentation will be influenced depending on who is the user, and the content outsorter for that user

Imagine that a doctor is doing a research about the variability of human's temper. He has invited eight groups of three people to complete a form on a computer, where each participant is going to access the computer in a quiet, closed room. Each group has to fill in a different form with some questions and images that are the same in every form, and other questions and images that are unique for each form. The doctor does not want to write eight different documents differing in their questions and images. So he writes one document with all the possible questions with the corresponding images that are being pointed to via structural filelinks.

The doctor creates eight content outsorters for the document, and each content outsorter defines which questions and images are included in the document. Each user receives beforehand an account with a password to login to the computer. A policy is applied for each user, which defines which content outsorters are accessible from the document, and setting defines where the document is located (in this case on the desktop). When the user is logged in, they will view only one document on the desktop. When they open it, the application to view the document will try to access the document. The RBAF system will first check which user is authenticated and then let the application read the authorized content outsorter in order to know which questions and structural filelinks to images should be viewed. When the user saves the document after they are finished, the application will save the results in a file only accessible by the doctor.
This example shows that we can restrict and adapt the structure of our data without creating additional files (i.e. creating documents with different questions and images), while having the control of what needs to be viewed and what not.

4.9 Summary of the RBAF System

A resource can be either a file or a folder. The `file` type in the RBAF model is extended with concrete file types, for example PowerPoint presentations (`.ppt`).

An entity can have properties, which is a key/value tuple that adds extra information. It can be used to describe an entity, or used by applications or the RBAF system as instructions to understand how to work with an entity.

We have introduced the concept of content recycling, which uses the concepts of selectors and structural filelinks.

A selector is used to select parts of entities in order to be accessed by other entities. For example, selecting a part of an image and refer to it in a document. The selector type is abstract and is extended with concrete selectors for specific file types. It is used by plug-ins to address parts of specific file types. Each selector is on a layer. A layer shows the distinction of overlapping selectors. For example, a selector which selects a piece of text can overlap another piece of text selected by another selector.

A link logically associates two entities. It can be a semantic link, structural link or navigational link.

A semantic link semantically relates two entities. It is useful to relate data in

order to form a new information source (for example we do not see two files as two distinct pieces of information, but as one piece of information), or to organize data in a flexible way.

A structural link associates an entity with another entity, making the first entity structurally composed of the second entity. It can be either a structural file- or folderlink.

A structural filelink is used to compose the file's content with external content. For example, we can associate a document with a photo, in order to make the photo structurally part of the document without merging the document with the photo.

Structural folderlinks are used to classify files or folders in multiple folders. They are used for the concept of multiple classification.

A navigational link associates two entities in order to navigate from one entity to the other. For example, consulting pages from different documents one after the other. Navigational links are used in the content traveling concept.

An entity can also have content outsorters. A content outsorter, which belongs to the `ContentOutsorters` subclass of the `ContextResolvers` class, is used to attach visibility criteria to entities. For example, we might want that a file should not be visible when we view it in certain folder.

Entities are owned and accessible by users. Each user can have preferences for entities, which can be policies or settings. We can combine content outsorters and access rights in order to adapt the presentation or environment depending the user, its policies and settings.

Chapter 5

Implementation

In this chapter, we discuss the implementation of the RBAF system in general. The RBAF system and two other applications have been implemented in Java. For details please have a look at the class diagrams in Appendix A. We first look in more detail at the architecture of the RBAF system. Next, we justify the usage of db4o as a database and the Prefuse toolkit and its API for building information visualization. Finally, we explain how the structure of PowerPoint files in the `.pptx` format looks like, which has enabled us to create a PowerPoint plug-in to manipulate `.pptx` files.

5.1 The RBAF Architecture

We have already discussed in Chapter 4 one aspect of the general functionality of the RBAF system on a high level. Now we are going to look at the organization of the architecture one level deeper. As shown in Figure 5.1, the RBAF system itself is divided in four main parts: *RbafFacade, Model, Metadatastorage* and *Plug-ins*. The RafFacade can be seen as an API, which is the entry point for applications when communicating with the RBAF system. The RbafFacade is also responsible to access files and folders in the filesystem. The Model is where the RBAF model is implemented, such as the definition of entities and links. The Metadatastorage is responsible to save and retrieve all objects and their relation (such as the semantic links between resources) in a db4o database. Finally, the Plug-in is where we add extra functionality in the form of stand-alone code to the RBAF system to access new types of resources, such as manipulating slides in PowerPoint files. The RbafFacade can access and manipulate new types of data via the plug-in's functionality.

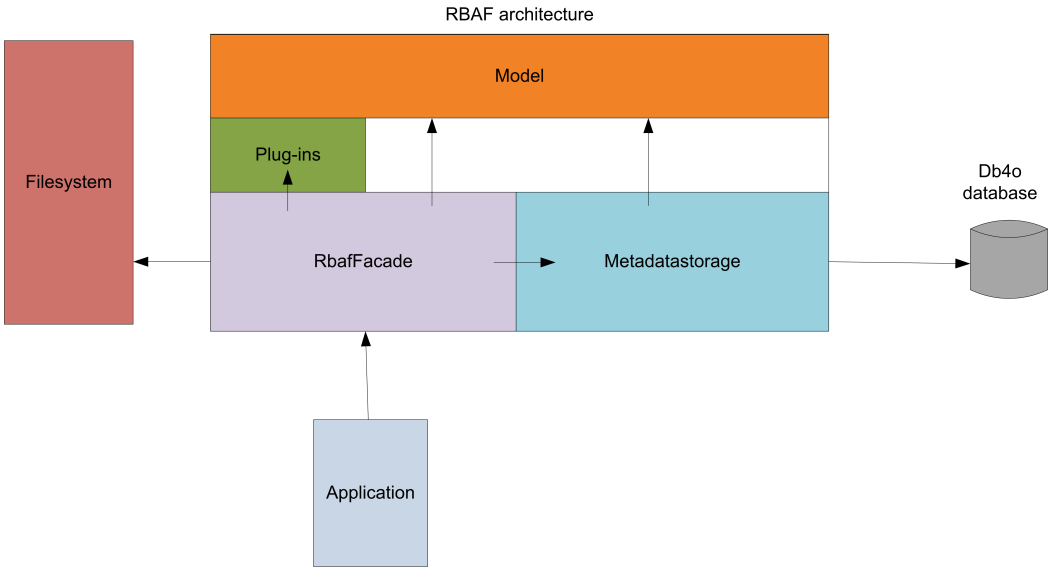


Figure 5.1: RBAF architecture

5.2 Db4o database

We have chosen to store all metadata (such as as resources, properties and semantic links) in a database called *db4o*[1]. Db4o (database for objects) is an embeddable open source object database for Java and .NET . There are a few advantages that influenced our decision to make use of it above a relational database:

- we do not have to create a database beforehand with tables, nor dealing with relations (primary and foreign keys) between tables. Db4o stores all objects in a file. The advantage is that if the implementation in the RBAF system changes, nothing has to be changed in the db4o database.
- it is portable: we can not only carry it with the actual application, but also transfer and use it in other applications, since the database is simply a file.
- we can use the native programming language (in this case Java) to query objects instead of using string-based queries such as SQL, which takes time to update manually.
- we do not have to care about how objects are stored (including their relation with other objects). When querying the object, related objects are automatically queried.
- indexing of objects is possible in order to speed up queries.

[1] http://www.db4o.com/

We can browse the objects saved in a db4o database with an Eclipse plug-in called *Object Manager Enterprise* (OME). OME offers functionality such as browsing and searching for classes and objects easily, and creating queries with drag and drop. A screenshot of OME in Eclipse is shown in Figure 5.2.

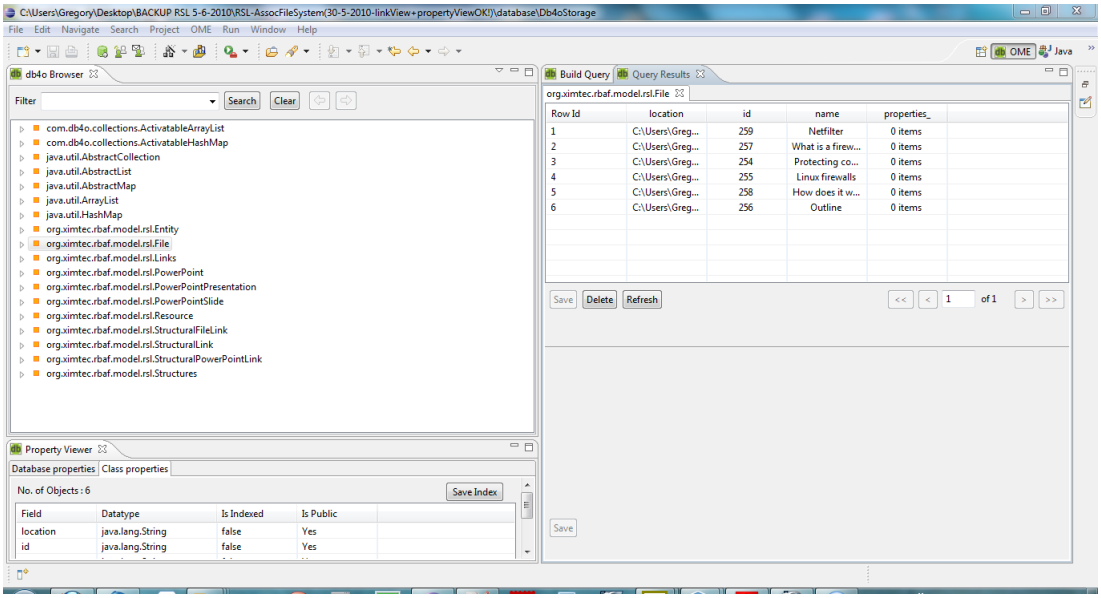


Figure 5.2: Object Manager Enterprise plug-in in Eclipse

5.2.1 How Objects are queried and stored in Db4o

Accessing the database does not require to create configuration files, or even to create the database itself. When we start to store and query objects in the database, the database itself will be automatically created. In order to be able to access the database, we must at least mention where it must be located. Listing 5.1 shows a method used to create and return a new ObjectContainer in order to access the database as an object.

Listing 5.1: openObjectContainer method

```
private static ObjectContainer openObjectContainer()
{
EmbeddedConfiguration embeddedConfiguration =
Db4oEmbedded.newConfiguration();

embeddedConfiguration.common().add(new
    TransparentPersistenceSupport());
```

```
return Db4oEmbedded.openFile(embeddedConfiguration, "database/
    Db4oStorage");

}
```

The last line of code enables us to open the database named Db4oStorage located in the database folder. The database file is automatically created if it does not exist.

Listing 5.2 shows an example of a method to store an object of type File in db4o.

Listing 5.2: Storing an object in db4o

```
ObjectContainer db=openObjectContainer();
try
{
   File newfile = new File(id,newfilename,newlocation);
   db.set(newfile);
   return (File) newfile;
}
finally
{
   db.close();
}
```

First we create an *ObjectContainer* object allowing us to access the database. Then, we create a new File object, and save it in the database with the method `set(object_to_save)`. Finally we close the database with the method `close()`.

Listing 5.3 shows an example of querying all File objects in the db4o database.

Listing 5.3: Querying an object in db4o

```
ObjectContainer db=openObjectContainer();

try
{
File file=new File(null,null,null);
ObjectSet result=db.get(file);
return result; //result is a datastructure of type List
}
finally
{
db.close();
}
```

The used way to query objects is *Query by Example* (QBE). This means that we create a prototypical object for db4o to use as an example. In the previous example code, the prototypical object is a new File object. Db4o will retrieve all objects of the given type that contain the same (non-default) field values as the candidate. Since we want to retrieve all File objects, we give null values in the constructor. The result will be handed as an *ObjectSet* instance.
Updating an object is done by querying the object we want from db4o, change its fields (for example via setters of the object) and save that object again in db4o with the `set(object_to_update)` method.
Deleting an object is also very easy: just query the object from the database and execute the method `delete(object_to_delete)`.
Just because db4o uses a file to store objects does not mean that queries are slow. We can speed up queries by telling to db4o that we want our objects to be indexed when we store them. This is done with one line of code:
`Db4o.configure().objectClass("org.ximtec.rbaf.model.rsl.File").indexed(true);`
This line of code means that we want to index objects of type File in the package `org.ximtec.rbaf.model.rsl`.

5.3 Prefuse Toolkit

In one of our applications, which is a specialized file explorer, we have developed a way to visualize semantic links between resources of the RBAF system. We have used a toolkit called *Prefuse* with its API[2] to help us to realize that. Prefuse is a user interface toolkit written in Java for building interactive information visualization applications. The API simplifies the processes of representing and efficiently handing data, mapping data to visual representations such as graphs, and interacting with the data. The efficiency to show data is one of our criteria which influenced our decision to use this API over others. Another criteria is that it offers the possibility to create different forms of visualizations of data. Figure 5.3 shows an example.

We can describe and hand in data (such as the definition of the edges and nodes) to the toolkit in an XML file, which will read it and generate a predefined visualization.

In the background chapter, we have seen that we can represent the human nervous system, or memory, with a sort of a graph, as one in Figure 2.10. We also wanted to use a graph to represent the semantic links as connections between

[2]http://prefuse.org/
[3]images taken from http://prefuse.org/gallery/

Figure 5.3: Example of a possible data visualization with Prefuse[3]

resources, just as the associated nodes explained in the Synapse-State theory. In our graph, the nodes represent the resources (file or folder), and the edges are the semantic links. When the user selects a resource in our file explorer application, they have the possibility to view all other resources semantically linked with it.

5.3.1 Integrating the Prefuse API to draw a Semantic Link Graph

We first need to create the XML file containing the definition of the nodes, edges and other data that influences the visualization of the graph. The `GraphMLReader` class will load the graph from the XML file and return an instance of a `Graph`. An example of such an XML file is shown in Listing 5.4.

Listing 5.4: semanticgraph.xml

```
<graphml>
    <graph edgedefault = "undirected">
        <key id = "id" for = "node" attr.name = "id"
        attr.type = "string"/>
        <key id = "name"  for = "node" attr.name = "name"
        attr.type = "string"/>
        <key  id = "type" for = "node" attr.name = "type"
        attr.type = "string"/>
        <key id = "location"  for = "node"  attr.name = "location"
        attr.type = "string"/>
        <key id = "linkname"  for = "edge" attr.name = "linkname"
        attr.type = "string"/>

        <node id = "286">
            <data key = "id">286</data>
            <data key = "name">Images2</data>
            <data key = "type">Folder</data>
            <data key = "location">
            C:\Users\Gregory\Pictures\Moi\SubFolder\Images2
```

```
20              </data>
21          </node>
22          <node id = "287">
23              <data key = "id">286</data>
24              <data key = "name">Images3</data>
25              <data key = "type">Folder</data>
26              <data key = "location">
27              C:\Users\Gregory\Pictures\Moi\SubFolder\Images3
28              </data>
29          </node>
30
31          <edge source = "286" target = "287">
32              <data key = "linkname">link1</data>
33          </edge>
34
35      </graph>
36  </graphml>
```

Line 2 in the `semanticgraph.xml` file means that we want an undirected graph. Starting from line 3 to 12, we declare our attributes we want to assign to nodes and edges. From line 14 to 29, we define the nodes with their attributes we want to appear in the graph. In this `semanticgraph.xml` example, we declare two nodes which represent folders in the RBAF system. Finally, from line 31 to 33 we define the connections between the nodes with edges (in this case we define one edge between the two nodes). Each edge will have to show its semantic linkname when drawn. Next, we need to create a `Visualization` instance, which constructs an abstraction of the graph, and add the graphdata to it.

In order to show labels (such as the name of the nodes and edges), we need to create a `LabelRenderer` instance to create the labels, and pass that instance to the `DefaultRendereFactory` to render the labels visually.

We can also assign colors to the nodes so that we can see the difference between types of nodes. In this case, we have to see the difference between nodes that represent folders and nodes that represent files. So we create a `ColorAction` instance and define the colors for the nodes and edges (we assign the default black color to the edges).

Then we create an `ActionList` instance that groups all the color assignment actions into a single executable unit.

We also need to create a separate `ActionList` providing an animated layout. We have created a class `SemanticLinkLayout` which extends the `ForceDirectedLayout` class to inherit the functionality to position the nodes and edges correctly when they are drawn. For example the nodes and edges must not overlap each other.

Finally, we have added existing interactive controls, such as the `DragControl`

class that permits to move the graph with the mouspointer. We only needed to create the `ClickResourceControl` class that extends the `FocusControl` class. `ClickResourceControl` listens to click events on nodes and shows the clicked resource that represents the node in our file explorer.

5.4 PowerPoint Files Manipulation in the .pptx Format

In the second application that we have developed, we needed to manipulate PowerPoint presentations. The idea behind is to be able to grab PowerPoint slides from any PowerPoint presentation and to generate a customized PowerPoint presentation that includes those slides. We have first been looking for existing API's to manipulate .ppt or .pptx files. The APIs we thought were the most promising were *POI-HSLF*[4] and *openxml4j*[5] (manipulates only `.pptx` files). Unfortunately, POI-HSLF still does not support `.pptx` files. During our tests, we have tried to manipulate `.ppt` files. But the API has limitations, such as the impossibility to copy a slide from one presentation to the other. The official website of openxml4j was during the period of the thesis not available. In other words no official source code or documentation was available.
We have realized that Microsoft Office's new file types `.pptx`, which are Office Open XML files (OOXML), are fully described in XML. Thus, we have decided to study the structure of such an OOXML file and write our own code to access and manipulate PowerPoint OOXML files.

5.4.1 What is OOXML?

Since the introduction of Microsoft Office 2007 suite, new file types have been introduced with extensions like `.docx` and `.pptx` (older versions have the extension `.doc` and `.ppt`). Those new file types are called *Office Open XML* (OOXML). One of the objectives of OOXML is to allow applications to store their documents in a form fully described in XML and then compress it in a Zip archive. The older file types such as .ppt are not described in OOXML, and are simply a stream of non-readable bytes. An advantage of OOXML is that we do not have to run an instance anymore of a Microsoft Office application in order to access and edit files. For example, instead of using macros written in Visual Basic, which requires an instance of Microsoft Office to run, we can create and manipulate OOXML files with any developed application or API that can manipulate XML documents and compress files in the Zip format. It offers many opportunities, such as creating a webservice that enables users to create and edit PowerPoint presentations.

[4]http://poi.apache.org/
[5]http://openxml4j.org/

Each OOXML file has a slightly different structure. In the next subsection, we explain how the general structure of `.pptx` files looks like.

5.4.2 The General Structure of PowerPoint OOXML Files

Once a `.pptx` file has been created (for example by using Microsoft Office), we can change the file extension from .pptx to .zip in order to have a compressed Zip archive. Figure 5.4 shows the structure of the zipped file.

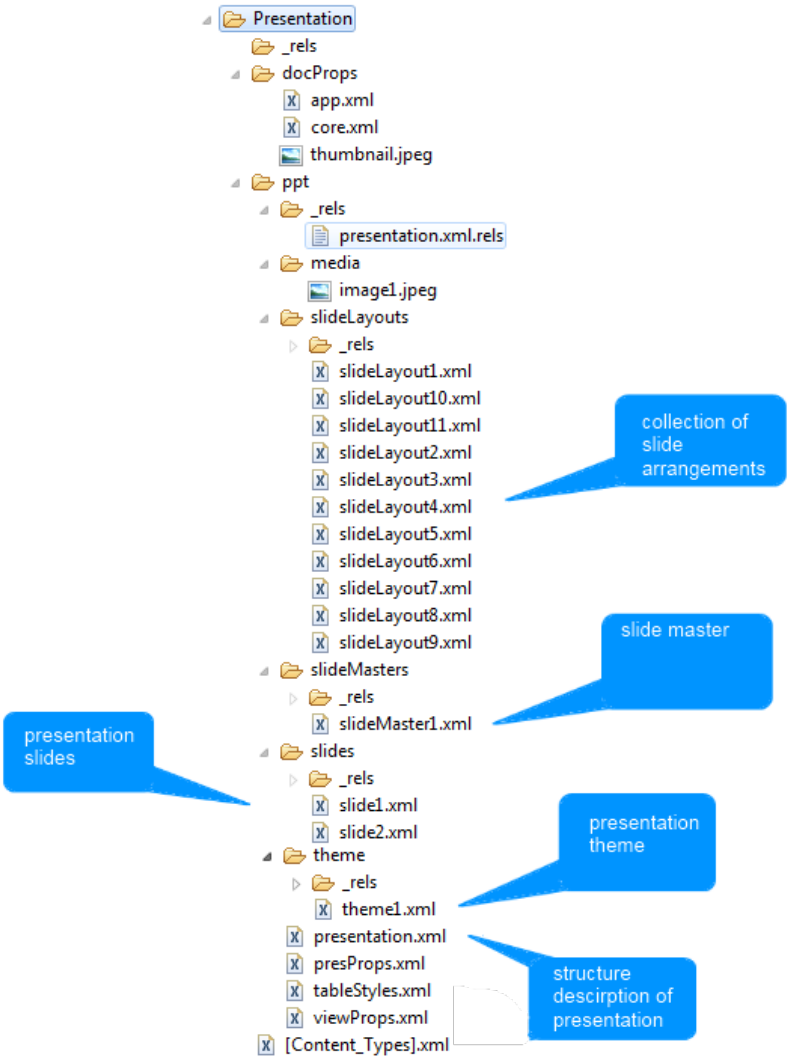


Figure 5.4: Internal structure of a `.pptx` file

We only describe some files and directories we needed to access and manipulate.

All slides are described in a file called `slide`*X*`.xml`, where the *X* behind slide is the number of the slide order in the PowerPoint presentation. Each XML file describes the layout and structure of the slide.
For each `slideX.xml`, a file called `slide`*X*`.xml.rels` is necessary. That file defines a relation between the slide and a template (described in `slideLayout`*X*`.xml`).
The `presentation.xml.rels` file defines a relationship between each slide and other necessary XML files (which, for example, contain properties). For each relationship, an *rId* is assigned in order to uniquely identify the slides and XML files.
The `presentation.xml` file defines the structure of the entire PowerPoint presentation.
The `[Content_types].xml` file , located in the root of the zip package, identifies every necessary type of XML file, such as slides, found within the PowerPoint presentation zip archive. Each XML file needs to have its type listed in the `[Content_types].xml` file. XML files need to have identifiable types so that the Microsoft Office 2007 knows how to use each XML file when rendering the PowerPoint presentation. The types also enable to understand the XML file's purpose and how to use them.
Finally, media files such as images are located in the folder *ppt/media*.

In the next chapter, we explain the functionality of two developed applications that work with the RBAF system.

Chapter 6

User Guide

In this chapter, we introduce two implemented applications that work with the RBAF system. The first application is called *Associative File Explorer*. The second one is called *PowerPoint Linker*.

6.1 The Associative File Explorer

6.1.1 Presentation of the User Interface

The Associative File Explorer is like a file explorer that we find in most operating systems, but with some extra functionality of the RBAF system. The user interface is divided in five parts as highlighted in Figure 6.1.

The first part contains two panels next to each other which we call the *main panels*. Each panel shows the resources forming part of a folder. We can single-click on a file or folder (which triggers operations that we describe later). Additionally, we can double-click on a folder in order to view the resources of that folder, just as we double-click on a folder in a any file explorer to see the files and folders within it. Above each main panel, there is a Go Back button that allows to view the resources of a previously visited folder. Next to each Go Back button is a field that shows the name of the folder we are currently browsing. Below each main panel, a panel shows the parent folders of a selected resource. It shows the parent folder of the selected resource, and the folders that are structurally folderlinked with the selected resource. We will see later how we can create a structural folderlink. We can double-click on a parent folder to view the containing resource in the main panel. The reason why two panels are present is to facilitate the navigation between multiple folders and to keep an eye on the content of two folders at the same time. For example, we might want to copy multiple files from one folder to another folder. The second part of the user interface contains a panel that lists the properties of a selected resource from one of the main panels. This panel that shows the properties

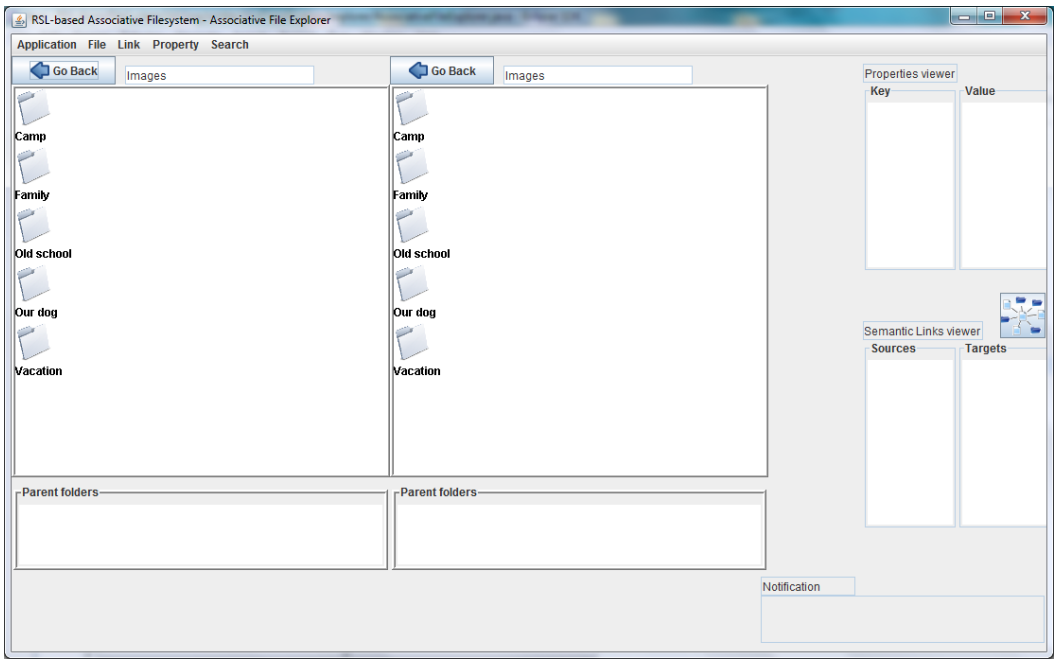


Figure 6.1: Associative File Explorer

is called *properties viewer*. The properties viewer is divided in two columns. The first column shows the *keys* of properties, while the other column shows the *values*. Each associated key and value is shown next to each other. The third part of the user interface is a panel that has a similar layout as the properties viewer and is called *semantic links viewer*. When a resource is selected, a list of all resources which are the source of semantic links to the selected resource are shown in the column *sources*. The other column shows the *target* resources of the semantic links of the selected resource. The semantic links viewer also has a button that shows a graph of all semantic links of a selected resource in a separate window. It is called the *semantic link graph viewer*. The functionality of the semantic link graph viewer will be explained later. The fourth part of the user interface is the menu of the application. What the menu items mean will also be clear later. The last part is a notification area on the bottom-right corner of the Associative File Explorer. This area shows any application feedback.

6.1.2 Files, Folders, and Properties

When we open the Associative File Explorer for the first time, we assume that there are no files or folders present on the disk drive. The Associative File Explorer gives use the possibility to create an empty file or folder on the disk drive. To create a file or folder, go to the menu and select `File` → `New File` or `New Folder`. We are

asked to give a name to the resource. We can also import existing files and folders by going to the menu and selecting `File` → `Import files and folders in RBAF system`. In this case we are asked to select the folder in which the resources we want to be imported are located. Once the importation is done, we will see the content of the imported folder in both main panels. When resources are created on the disk drive or imported, they are registered in a Db4o database in order to identify and keep track of them. We can add properties to any resource as much as we want. To add a property to a resource, select the resource of your choice, then in the menu, choose `Property` → `Add property to a resource`. We are finally asked to give a property key and value. Every time we select a resource, all the properties we have created for a resource will be shown in the properties viewer, as shown in Figure 6.2.

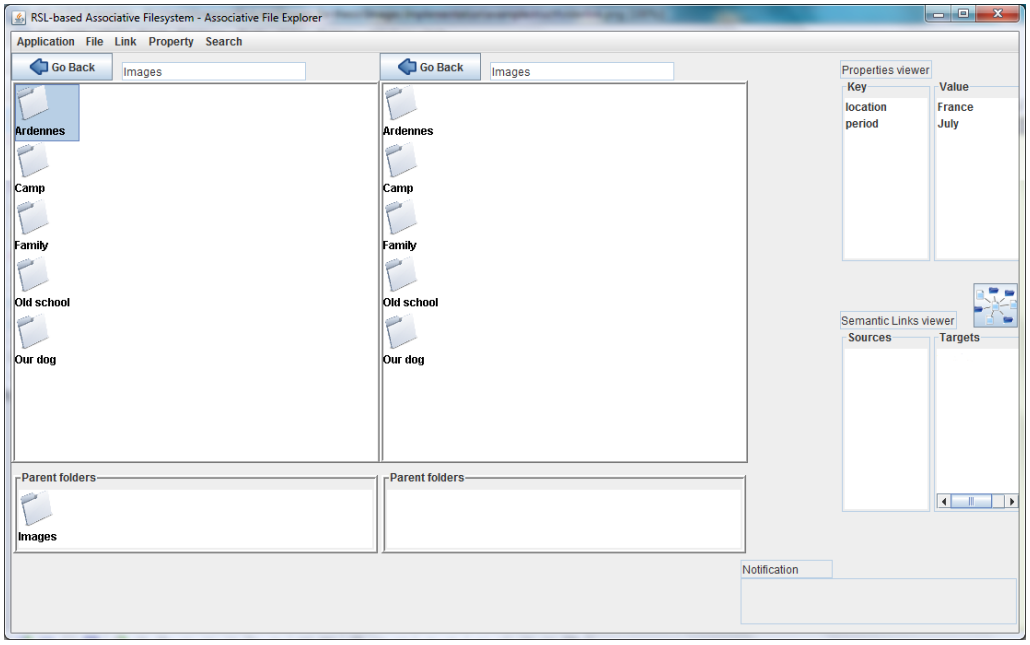


Figure 6.2: The properties viewer shows the properties of the selected resource

In the properties viewer, we can also select a property key or value, and view all resources that have the same property key or value within one of the main panels. The main panel that will show the resources depends which one got the focus for the last time. By default, it is the left main panel that shows the resources.

6.1.3 Semantic Links

The Associative File Explorer also allows to semantically link resources. In order to create such a semantic link, we first have to select in the menu `Link` → `Semantic link two resources`. Next, we have select a resource as a the source of the se-

mantic link, and then the second resource as the target of the semantic link. We are finally asked to give a name to the semantic link. To view all semantic links of a resource, select the desired resource in one of the main panels and an overview of all semantic links will be shown in the semantic links viewer. Figure 6.3 summarizes the procedure.

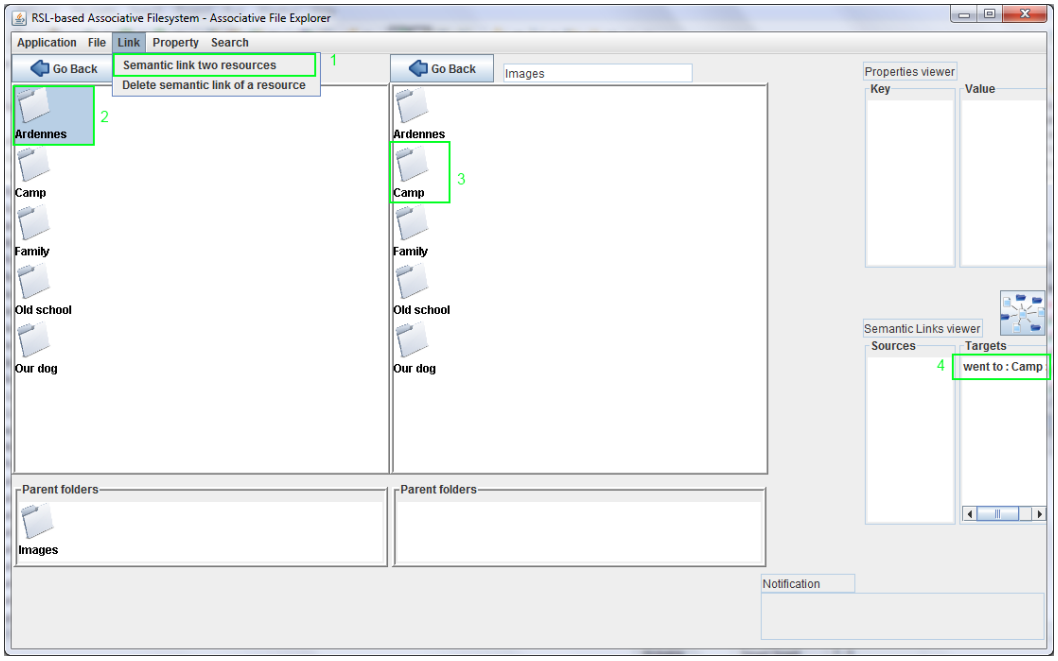


Figure 6.3: Summary of how to create a semantic link between two resources. The last step just shows the result after the creation

Each column of the semantic links viewer (source and target) shows the semantic linkname and the name of the resource separated by a ':'. When we click on the resource name in the semantic links viewer (source or target), the actual resource will be shown in one of the two main panels (depending which one got the focus for the last time). The button on the right side of the semantic links viewer enables us to view all semantic links (sources and targets) of the selected resource in the form of a graph (which we called semantic link graph viewer). Figure 6.4 shows the semantic link graph viewer.

The semantic link graph viewer is divided in two parts: the left side shows all nodes (which represent the resources) and edges (which represent the semantic links with their name). The right-hand side has two sliders. The bottom slider lengthens the edges between the nodes. The upper slider determines how fast the lengthening of the edge's length should be visible. The semantic link graph viewer is interactive and is to use in combination with the Associative File Explorer. When clicking on one of the nodes, the corresponding resource will be shown in one of the two main

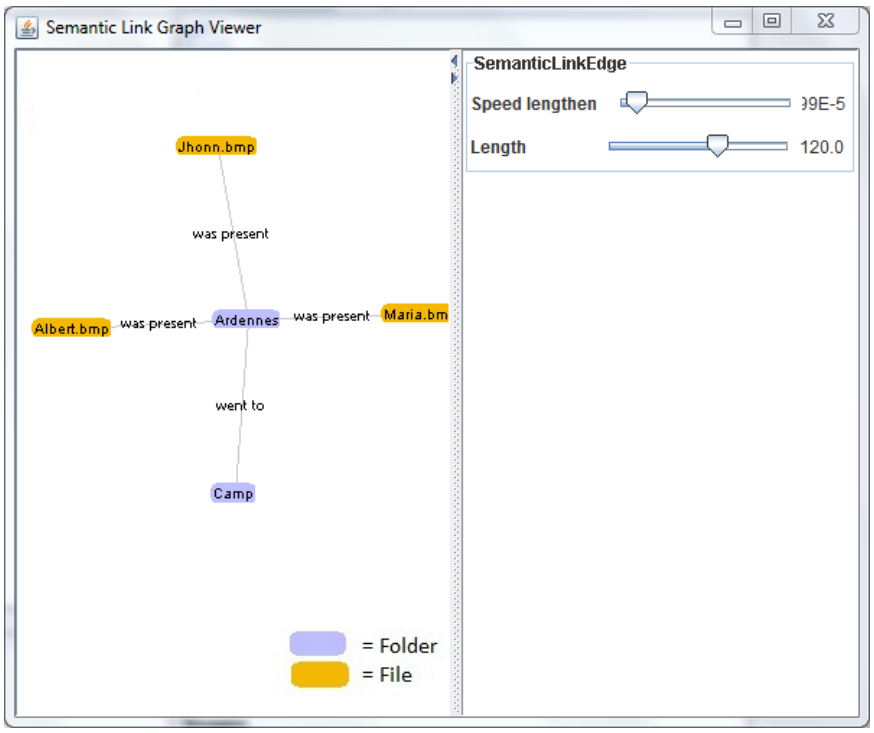


Figure 6.4: The Semantic Link Graph Viewer

panels. In that way, we can continue to perform further actions on that resource such as viewing the properties of it. The line separating the two sides can be moved to the right in order to hide the right side and leave more space for the graph itself.

6.1.4 Structural Folderlinks

We can classify any file or folder in multiple folders with structural folderlinks. With this feature, we do not have to physically copy and paste a file or folder into another folder, and thus data is not duplicated. Creating structural folderlinks is made transparent to the user by using the terms *copy* and *paste*, instead of saying *start creation of structural folderlink* and *accomplish creation of structural folderlink*. To create a structural folder link, first select the desired file or folder. Then go to the menu and select `File` → `Copy`. Navigate to any other desired folder where the "copied" file or folder should be "pasted", and select in the menu `File` → `Paste`. The previously selected file or folder will be classified in the folder we want it to be pasted. When clicking on the resource we have structurally folderlinked, we see the original parent folder (i.e. in the real folder where the resource is located) and all other folders that are structurally folderlinked with the resource. Figure 6.5 illustrates that. Note that a structural folderlink is not a shortcut, NTFS junction point, symbolic or hard link. The RBAF system treats a folder which is structurally

folderlinked with a resource as a parent of the resource, and thus the resource as a real resource. In the next subsection, the differences will be more clear. Refer to 2.2 to review the differences between shortcuts, NTFS junction points, symbolic and hard links.

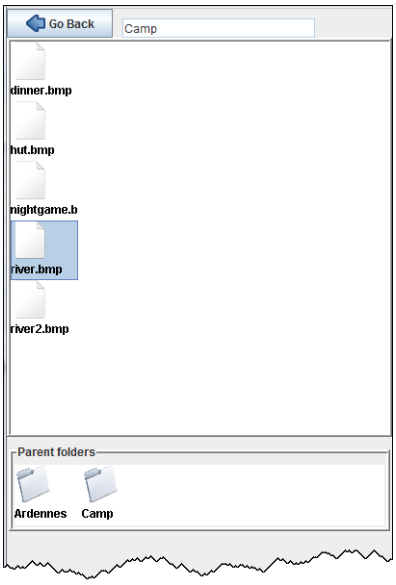


Figure 6.5: The selected file (which original parent folder is Camp) has been structurally folderlinked in the Ardennes folder. The parent folders panel shows the parent folders of the selected resource

6.1.5 Move and Delete operation

The Associative File Explorer allows us to move and delete resources, but we can only delete properties and semantic links. Moving a resource means changing the physical location of it. For example a file physically located in `C:\file.txt` can be moved to `C:\folder\file.txt`. Whether we are trying to move a resource that is structurally folderlinked or not, we will always move the resource from its physical location (we do not modify the structural folderlink of the resource). We can modify a structural folderlink by moving the target resource from folder to another folder. It is also possible to move the resource physically and leave all structural folderlinks to it unchanged. The deletion of a resource operation works differently in two situations: when we try to delete a resource structurally folderlinked, the structural folderlink itself will be deleted, and not the real resource. For example when we try to delete a file that is structurally folderlinked with a folder A within that folder, and is physically located in folder B, we will not see the file in folder A anymore, but it will still be available in folder B. However, suppose that in that situation we try

to delete the file in folder B, then we are at the point to delete not only physically the file, but also all structural folderlinks with other folders. Therefore, we are asked whether we are sure about this action. Otherwise we are advised to move the file somewhere else if it is not our intention to delete the real file, but to remove the classification of the file from folder B. We can perform the move and delete operation either by right-clicking on the resource and select the operation in the popup menu, or go to the menu and select `File` → `Move` or `Delete`. We can delete a property by right-clicking on the property key in the properties viewer and select the operation in the popup menu. Or, we can select the property key, go the menu and select `Property` → `Delete property of resource`. Finally, we can delete a semantic link by right-clicking on the source or target resource in the semantic links viewer, and select the operation in the popup menu. Alternatively, we can delete a semantic link by selecting the source or target resource in the semantic links viewer, go to the menu and select `Link` → `Delete semantic link of resource`.

6.1.6 Searching by Name, Property or Semantic Link

We have the possibility to search resources in three ways:

1. by name
2. by property: we can give in a property key or value
3. by semantic link: we can give in the name of the semantic link. The results are the sources and targets of all semantic links that have that name

To start a search process, go to the menu and select `Search` → `By name`, `By property` or `By semantic link`. The search result shows all resources that match the search criteria in one of the two main panels, depending which one got the focus for the last time.

6.2 The PowerPoint Linker

6.2.1 Presentation of the User Interface

The PowerPoint Linker allows to create structural powerpointlinks from a PowerPoint presentation to slides contained in other PowerPoint presentations. The user interface, shown in Figure 6.6, is divided in four parts. The first part consists of a panel that shows the PowerPoint presentation from which we want to create structural powerpointlinks to slides. The second part consists of another panel that shows slides that we can structurally powerpointlink to the current PowerPoint presentation. The third part consists of a menu, and the last part is a notification area which shows the application's feedback.

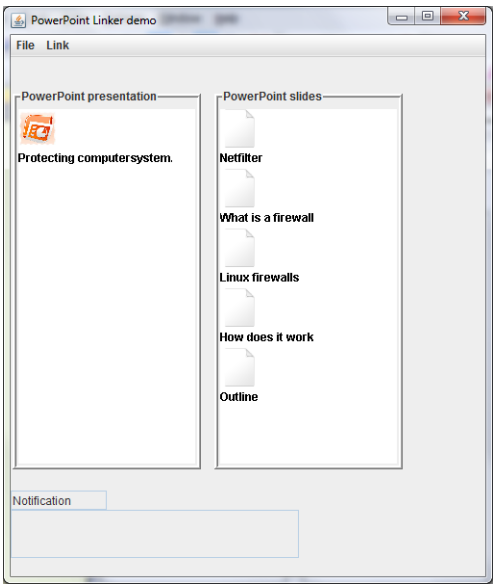


Figure 6.6: The PowerPoint Linker

6.2.2 Functionality of the PowerPoint Linker

To start to use the application, we need to import 1) the PowerPoint presentation to which we want to add structural powerpointlinks to PowerPoint slides, and 2) other PowerPoint presentations, which contain those slides. To do so, go to the menu and select `PowerPoint` → `Import presentation and slides`. We are asked to browse the hard disk drive and select the main PowerPoint presentation and other PowerPoint presentation(s) that contain the desired slides. When finished, we will see the imported presentation and slides. To create a structural powerpointlink from a presentation to a slide, go to the menu and select `Link` → `Create structural powerpointlink`. Then select the presentation to which we want to add a slide, and finally select the desired slide. When finished, the structural powerpointlinked slide will have a visual mark. We can delete a structural powerpointlink to a slide by selecting the desired slide, going to the menu and selecting `Link` → `Delete structural powerpointlink`. When satisfied with the result, we can generate the PowerPoint presentation with the structural powerpointlinked slides. Go to the menu and select `PowerPoint` → `Generate PowerPoint presentation`. The location of the generated PowerPoint presentation is defined in a configuration file called *PowerPoint.properties*.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we first have investigated how we can improve hierarchical filesystems in order to help users to better organize and retrieve their data, because the number of data that are created and stored on disk drives is constantly increasing.
We have introduced six concepts:

- multiple classification via folders
- semantically link data
- content recycling
- user-defined metadata
- content traveling
- content outsorting and access rights

We have shown that single classification of files via folders limits the flexibility of data organization. Thus, we have argued that introducing *multiple classification via folders* not only improves the flexibility of organization, but also reduces the complexity of folder hierarchies (i.e. deeply nested folder structures).
When we create files such as pictures and videos, it often happens that they are related to each other, as we have seen in the first scenario in Chapter 3. Using folders to group related files is not always possible. We have shown that creating *semantic links* between related data helps in two ways: to understand and view data as a unification, and to organize data in an alternative and flexible way that is impossible to achieve via folders. Semantic links are not only used for resources (i.e. files and folders), but can be used for other purposes, like semantically linking links (as annotation) with other entities as we have seen in 4.7.

We have further argued that with *content recycling*, we can reuse (parts of) of the content of other data in order to limit the creation of temporary data and replication of existing data. This is possible by creating *selectors* that select parts of the data we want to reuse, and create structural filelinks that point to them, and finally rendering the content of the selected data.
Furthermore, we postulated that creating user-defined metadata to describe data, called *properties*, has many advantages such as describing the content of a file and improving the search functionality in operating systems.
Moreover, we have shown that *content traveling* helps in defining the consultation order of the content of different sets of data, and that it has different applications. This is similar to hyperlinks with which we can jump to and consult the content of related webpages.
Last but not least, we have seen that *content outsorters* enable us to define arbitrary visibility criteria for data, such as creating a content outsorter to view only pictures of a particular person in a specific context. We can also combine content outsorters with *access rights* to establish policies of who can view what (content of) data in a particular context.

We have developed a prototype called RSL-based Associative Filesystem (RBAF) that has the purpose to provide services for the previously mentioned concepts. The RBAF system is based on the RSL-based Associative Filesystem model, which in its turn extends the Resource-Selector-Link model (RSL).

To demonstrate some functionality of the RBAF system, we have developed two applications that make use of the RBAF system. One application is called the *Associative File Explorer*, and the other one is called *PowerPoint Linker*.

7.2 Future work

Parts of the introduced RBAF functionality have been implemented at this moment including the following concepts:

- multiple classification via folders (structural folderlinks)
- semantics links
- user-defined metadata (properties)
- structural filelinks

The implementation of the non-mentioned concepts, and probably the improvement of the currently existing RBAF system may be part of future efforts. Finally,

other aspects need to be investigated: the RBAF system is currently supposed to work on a local filesystem. It would be nice to be able to move resources from one RBAF system to the other. Another interesting feature would be to not worry anymore about the exact location of resources (such as local disk drives and remote servers), which is also a vision we find in [5].

Appendix A

UML Diagrams

In this appendix, the UML class diagrams of currently implemented parts of the RBAF system are presented.

A.1 application.explorer package

Figure A.1: Classes of the Associative File Explorer (part 1)

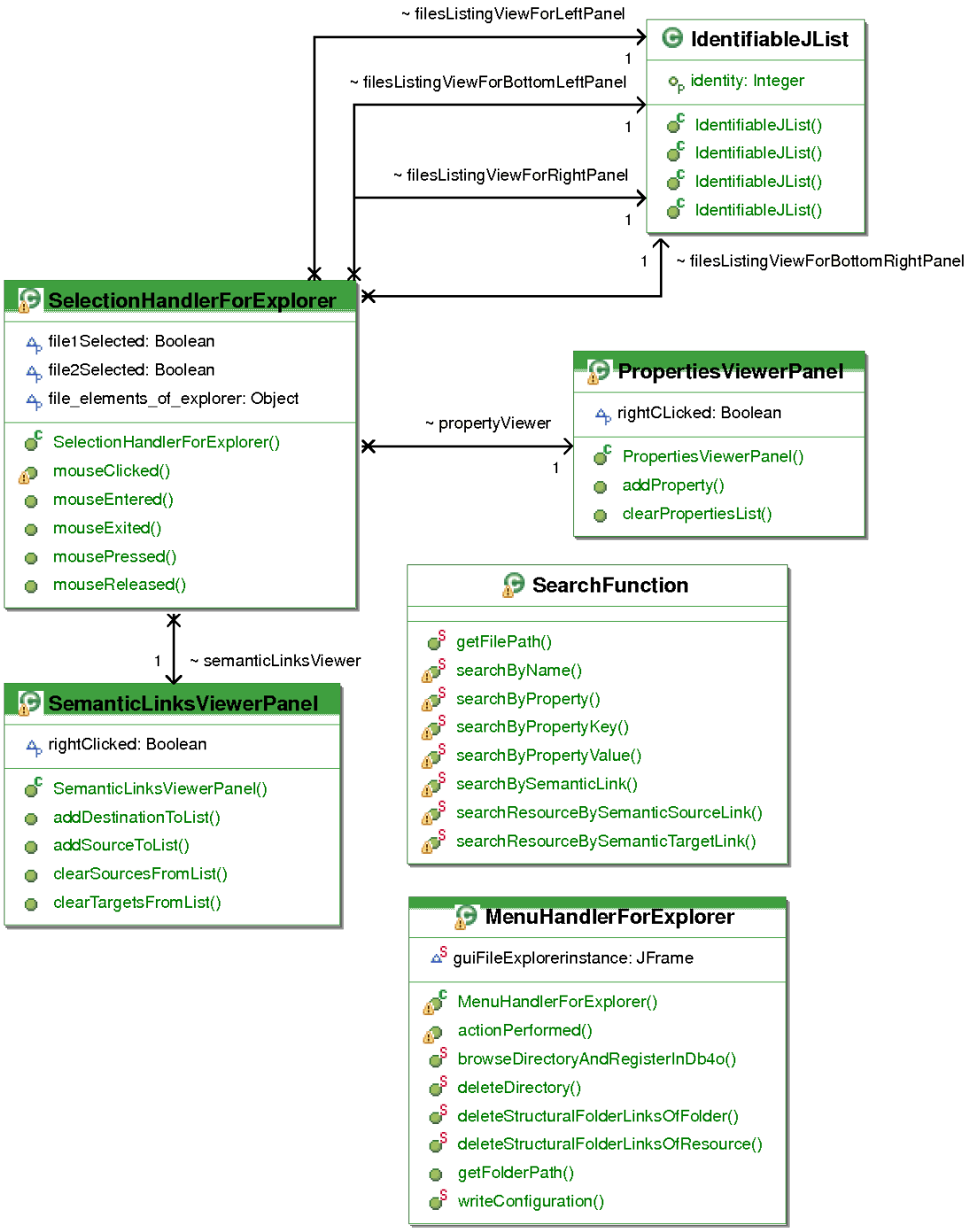


Figure A.2: Classes of the Associative File Explorer (part 2)

A.2 application.powerpointdemo package

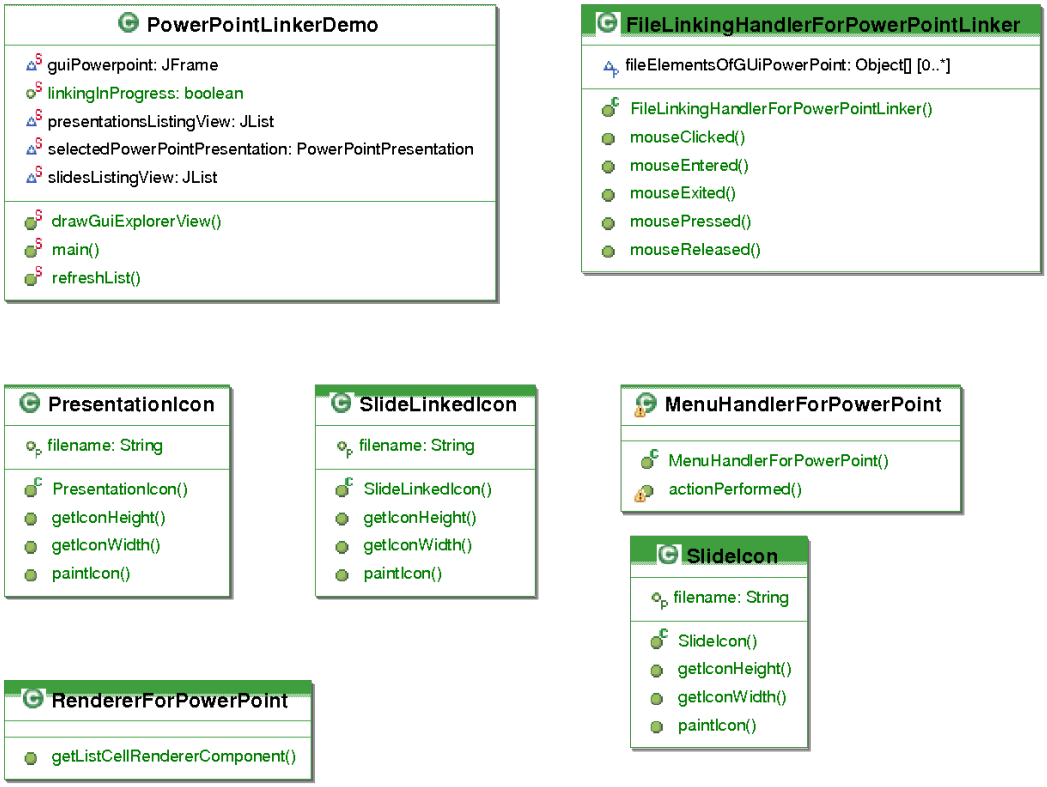


Figure A.3: Classes of the PowerPoint Linker

A.3 model.rsl package

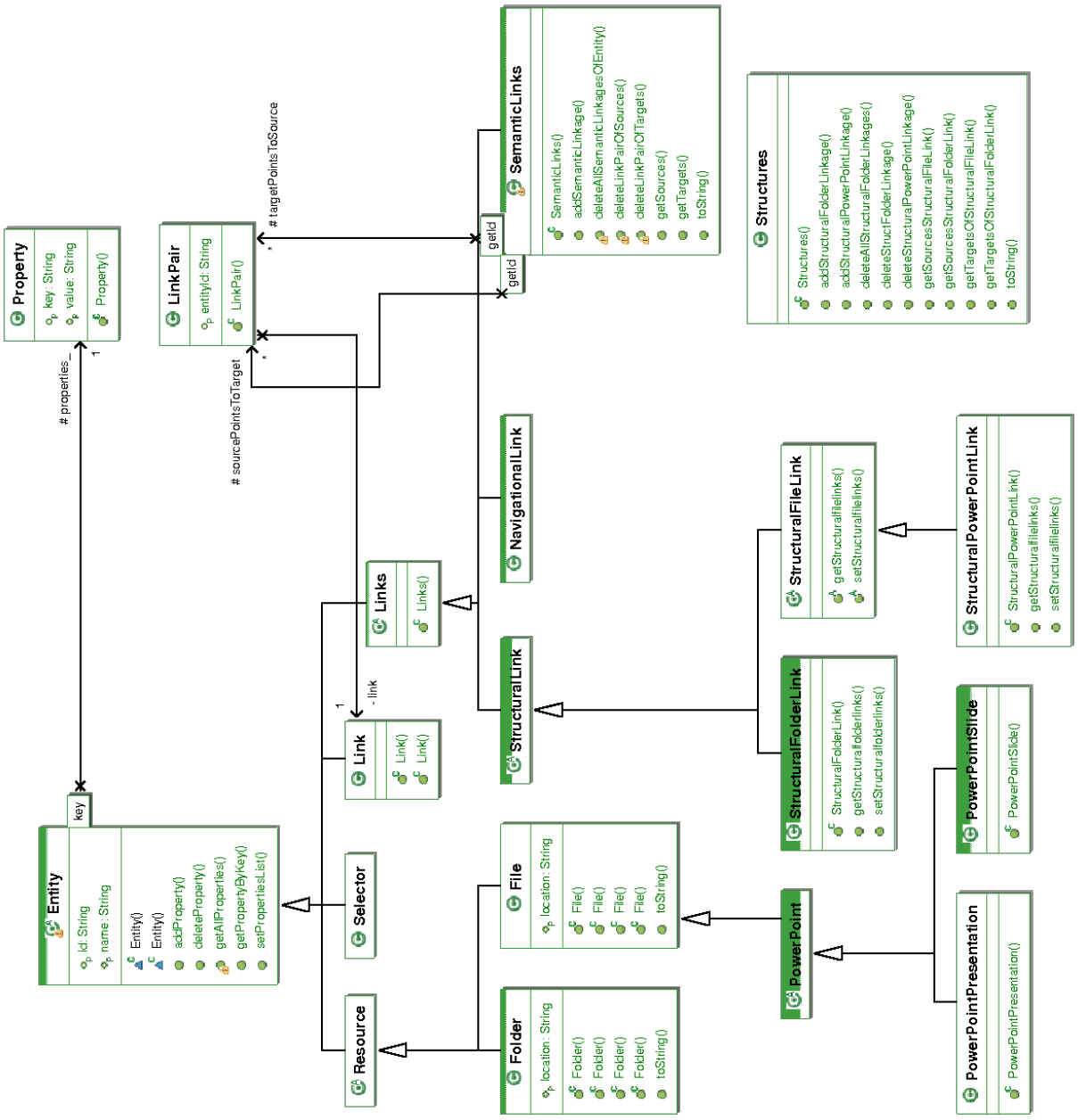


Figure A.4: Classes of the extended RSL model

A.4 facade package

Figure A.5: RbafFacade class

A.5 metadatastorage.db4o package

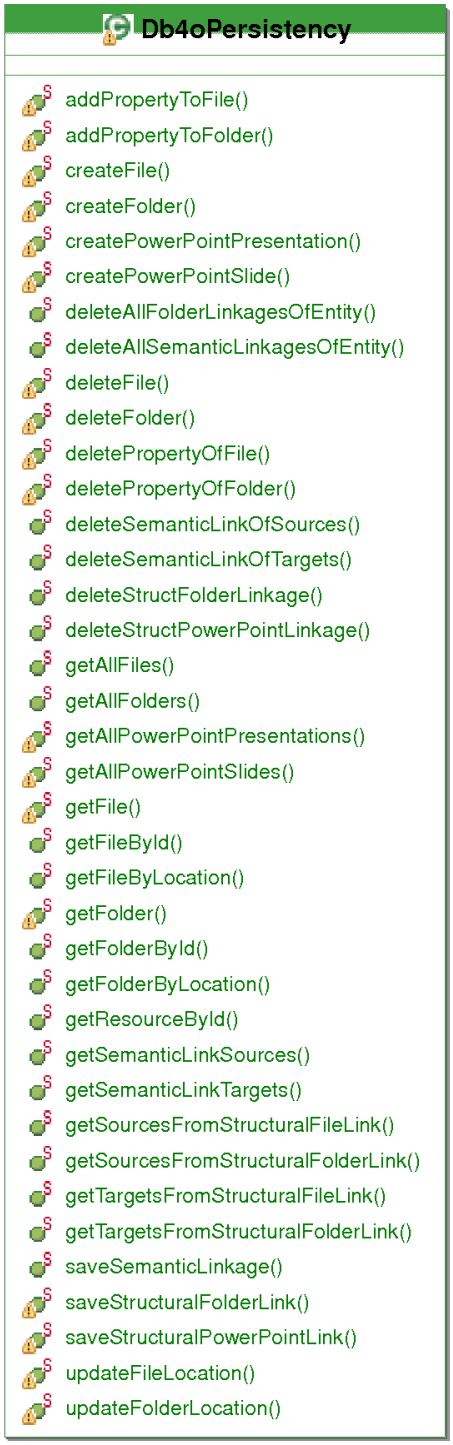


Figure A.6: Db4oPersistency class

Bibliography

[1] Douglas C. Engelbart : A Profile of His Work and Vision : Past, Present and Future, November 2008.

[2] Vannevar Bush. As We May Think. *Atlantic Monthly*, 176(1):101–108, July 1945.

[3] Anthony Collins. Tabletop File System and Personal Information Management in Pervasive Computing. Master's thesis, School of Information Technologies, University of Sydney, Australia, 2006.

[4] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon and James W. O'Toole, Jr. Semantic File Systems. In *Proceedings of SOSP 1991, 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA USA.

[5] Jens-Peter Dittrich, Lukas Blunschi, Markus Färber, Olivier René Girard, Shant Kirakos Karakashian and Marcos Antonio Vaz Salles. From Personal Desktops to Personal Dataspaces: A Report on Building the iMeMex Personal Dataspace Management System. In *Proceedings of BTW 2007, GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web*, Aachen, Germany, March 2007.

[6] Markus Majer. Database Driven Filesystems. Seminar Talk, January 2008.

[7] Gary Marsden and David E. Cairns. Improving the Usability of the Hierarchical File System. *South African Computer Journal*, pages 67–78, 2004.

[8] Peter Naur. Computing versus Human Thinking. Turing Award 2005 Lecture, Communications of the ACM, January 2007.

[9] Ted Nelson. Computer Lib/Dream Machines. Microsoft Press, 1987.

[10] Ted Nelson. Xanalogical Structure, Needed Now More than Ever: Parallel Documents, Deep Links to Content, Deep Versioning, and Deep Re-Use. *ACM Computer Surveys*, 31(4), December 1999.

[11] M.C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proceedings of ER 1993, 12th International Conference on the Entity-Relationship Approach*, pages 390–401, Arlington, USA, December 1993.

[12] Ravasio Pamela. *Personal Information Organisation: Studies on User-Appropriate Classification and Retrieval Strategies and their Implications for Information Management Systems Design.* PhD thesis, ETH Zurich, 2004.

[13] Jeremy Reimer. From BFS to ZFS: Past, Present, and Future of File Systems.

[14] Gabrio Rivera. *From File Pathnames To File Objects, An Approach to extending File System Functionality integrating Object-Oriented Database System Concepts.* PhD thesis, ETH Zurich, September 2001.

[15] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems are Dead. In *Proceedings of HOTOS XII, 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland.

[16] Beat Signer. GOMES - An Object-Oriented GUI for the Object Model Multi-User Extended Filesystem. Master's thesis, ETH Zurich, July 1999.

[17] Beat Signer. What is Wrong with Digital Documents? A Conceptual Model for Structural Cross-Media Content Composition and Reuse. In *Proceedings of ER 2010, 29th International Conference on Conceptual Modeling*, Vancouver, Canada, November 2010.

[18] Beat Signer and Moira C. Norrie. An Extensible Framework for Personal Cross-Media Information Management. In *Proceedings of EuroIMSA 2005, European Internet and Multimedia Systems and Applications*, Grindelwald, Switzerland, February 2005.

[19] Beat Signer and Moira C. Norrie. As We May Link : A General Metamodel for Hypermedia Systems. In *Proceedings of ER 2007, 26th International Conference on Conceptual Modeling*, pages 359–374, Auckland, New Zealand, November 2007.

[20] Beat Signer and Moira C. Norrie. An Architecture for Open Cross-Media Annotation Services. In *Proceedings of WISE 2009, 10th International Conference on Web Information Systems Engineering*, pages 387–400, Poznan, Poland, October 2009.

[21] Michael Solberg. Moving From Solaris to Red Hat Enterprise Linux. White Paper, 2009.

[22] Stephan Bloehdorn, Olaf Goerlitz, Simon Schenk and Max Voelkel. Tagfs - Tag Semantics for Hierarchical File Systems. In *Proceedings of I-KNOW 2006, 6th International Conference on Knowledge Management*, Graz, Austria.

[23] Cameron Dale Xu Cheng and Jiangchuan Liu. Understanding the Characteristics of Internet Short Video Sharing: YouTube as a Case Study, July 2007.