



Technical University of Denmark  
Department of Informatics and Mathematical Modelling

# Unified approach to management of distributed personal information

Master Thesis (35 ECTS)

Author:  
Piotr Szymański  
s053750

Supervisors:  
Jakob Eg Larsen  
Michael Kai Petersen

Kongens Lyngby 2010  
IMM-MSC-2010-1



## *Abstract*

---

Personal information is highly distributed between different data formats, applications, Internet services, computers and devices. It is difficult to get an overview of what one already knows. Popular personal information management (PIM) approaches usually provide yet another isolated place where the data is kept. On the other hand, experimental approaches that allow for accessing distributed information often do not provide sufficient means for integrating it.

The aim of this project is to develop a better understanding of how personal information management can be improved by integrating information, organizing it through linking, tagging and assigning meaning, and by supporting an individual way of working with it. Analysis of problems and scenarios has led to the design of a model for managing diverse types of information from various sources in a unified way. A prototype has been implemented based on this model.

The proposed model has been evaluated on the basis of the capability to realize certain scenarios, the functionality it provides and the potential to integrate with new technologies. It was found that the model eliminates many PIM problems and provides the user with a central place to store, organize and find their information.



## *Preface*

---

This thesis was prepared at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) in partial fulfillment of the requirements for acquiring the degree of Master of Science in Engineering (MSc).

This Master thesis is the result of work carried out in the period from July 2009 to January 2010 (including a summer break), with a workload equivalent to 35 ECTS credits.

Lyngby, January 2010

Piotr Szymański



## *Acknowledgments*

---

I would like to thank all the people who have assisted me during the period of writing this thesis.

I would like to thank my supervisors – Associate Professor Jakob Eg Larsen and PhD Student Michael Kai Petersen – for their time and support for this project. They provided me with valuable ideas, discussions and feedback related to this subject and the work I have been doing.

I would also like to thank my girlfriend Jola for her support, ideas and patience in listening to my loosely structured thoughts on the problems involved in this thesis.





# Table of Contents

---

1.	Introduction .....	1
1.1.	Motivation.....	1
1.2.	Problem definition .....	2
1.3.	Contributions.....	2
1.4.	Report structure .....	2
2.	Analysis.....	4
2.1.	Personal information.....	4
2.1.1.	Definition.....	4
2.1.2.	Characteristics .....	5
2.1.3.	Sources .....	6
2.1.4.	Metadata .....	7
2.1.5.	Related problem domains .....	8
2.2.	Commercial management approaches .....	10
2.2.1.	Microsoft Outlook .....	10
2.2.2.	Microsoft OneNote .....	10
2.2.3.	Ecco Pro .....	11
2.2.4.	Summary .....	12
2.3.	Combining distributed information .....	13
2.3.1.	Applications and concepts .....	14
2.3.2.	Duplicated and fragmented information .....	15
2.3.3.	Applications and data.....	16
2.3.4.	Linking and annotating.....	17
2.3.5.	Technical requirements.....	17
2.4.	Representation and storage.....	19
2.4.1.	Files and directories .....	19
2.4.2.	Relational model .....	19
2.4.3.	Object model .....	20
2.4.4.	Associative model.....	21
2.4.5.	RDF model .....	23
2.4.6.	Summary .....	27

2.5.	Summary .....	29
3.	Use cases .....	30
3.1.	Working on a project for a customer .....	30
3.2.	Semantic notebook .....	32
3.3.	Inviting people.....	35
3.4.	Summary .....	36
4.	Design .....	40
4.1.	Data model .....	40
4.1.1.	Application.....	40
4.1.2.	Physical .....	43
4.1.3.	Ontology .....	47
4.2.	Presentation layer .....	52
4.2.1.	Built-in views .....	54
4.2.2.	Custom views.....	59
4.2.3.	View selection .....	61
4.2.4.	Context menu .....	61
4.2.5.	View editor .....	63
4.2.6.	Ontology editor .....	63
4.2.7.	Internationalization .....	64
4.3.	Finding and managing information .....	65
4.3.1.	Searching for objects .....	65
4.3.2.	Related information .....	66
4.3.3.	Unstructured information .....	67
4.3.4.	Summarizing information.....	69
4.3.5.	Tagging information .....	71
4.4.	External information .....	72
4.4.1.	Selecting information .....	72
4.4.2.	Importing information.....	73
4.4.3.	Synchronizing information .....	74
4.4.4.	Exporting information .....	74
4.5.	Synchronization .....	74
4.6.	Sharing information.....	75
4.7.	Summary .....	76
5.	Implementation.....	77

5.1.	Three-tier design .....	77
5.1.1.	System components .....	77
5.1.2.	Third-party components .....	78
5.2.	Exchange of information .....	80
5.2.1.	Retrieving objects.....	80
5.2.2.	Querying for objects.....	81
5.2.3.	Committing modifications.....	82
5.3.	User interface .....	83
5.4.	Limitations.....	85
5.5.	Summary .....	86
6.	Evaluation.....	87
6.1.	Experimental difficulties.....	87
6.2.	Scenarios .....	88
6.2.1.	Working on a project for a customer .....	88
6.2.2.	Semantic notebook .....	90
6.2.3.	Inviting people.....	92
6.3.	Functionality.....	92
6.3.1.	Integrating information.....	92
6.3.2.	Structured and unstructured information .....	94
6.3.3.	Linking and finding information .....	94
6.4.	Potential .....	96
6.4.1.	Personal information domains.....	96
6.4.2.	Social communication .....	96
6.4.3.	Google Wave .....	97
6.5.	Summary .....	98
7.	Discussion.....	99
7.1.	Alternative design decisions.....	99
7.1.1.	Client-server interoperability .....	99
7.1.2.	Physical data model.....	100
7.1.3.	Custom views as RDF.....	100
7.1.4.	Data persistence.....	101
7.1.5.	Partial loading of data .....	101
7.2.	Third party components.....	102
7.2.1.	Sesame .....	102

7.2.2.	Hessian .....	103
7.2.3.	Aperture .....	103
8.	Related works.....	104
8.1.	Haystack .....	104
8.2.	Chandler .....	104
8.3.	NEPOMUK.....	105
8.4.	Summary .....	106
9.	Conclusion .....	109
10.	References.....	112
	Appendix A .....	116

## List of Figures

---

Figure 1: Ecco Pro's calendar view .....	12
Figure 2: A diagram of relationships between information sources.....	14
Figure 3: Table in the relational model .....	19
Figure 4: Items and links in tabular form. ....	22
Figure 5: Sample RDF graph .....	24
Figure 6: Conceptual illustration of the UI for the “Working on a project...” scenario .....	32
Figure 7: Conceptual illustration of UI for the “Semantic notebook” scenario .....	34
Figure 8: Objects and values .....	40
Figure 9: Persistent objects hierarchy.....	41
Figure 10: Classes for storing primitive values.....	41
Figure 11: Objects, properties and types .....	42
Figure 12: Type and property hierarchy.....	42
Figure 13: Equivalence and similarity for objects, types and properties .....	42
Figure 14: Diagram of all relationships owned by Properties and Types.....	46
Figure 15: UML class diagram for the Person type .....	48
Figure 16: UML class diagram for the Picture type .....	49
Figure 17: UML class diagram for the NotebookPage type .....	50
Figure 18: UML class diagram for the Location type.....	50
Figure 19: UML class diagram for the Photo Album type .....	50
Figure 20: UML class diagram for title properties.....	52
Figure 21: Equivalence relationship between title properties.....	52
Figure 22: UML class diagram for hierarchy of name properties.....	52
Figure 23: UML class diagram for View classes.....	53
Figure 24: View properties .....	53
Figure 25: Diagram of the OutlineView.....	55
Figure 26: Diagram of a DesktopView.....	55
Figure 27: UML class diagram for TypedOutlineView items .....	57
Figure 28: Diagram of a TimelineView .....	59
Figure 29: Information fragments data model.....	68
Figure 30: Example property hierarchy and values.....	70
Figure 31: Relations between picture properties. ....	70
Figure 32: Tags in RDF .....	71
Figure 33: Architecture diagram of the PIM system .....	78
Figure 34: Sequence diagram of requesting data from the server .....	80
Figure 35: Sequence diagram for the data commit process .....	82
Figure 36: Main window of client application.....	83
Figure 37: Client UI - context menu .....	84
Figure 38: Realization of the "Working on a project..." scenario using the prototype.....	89
Figure 39: Realization of the “Semantic notebook” scenario using the prototype .....	91
Figure 40: Client UI - a list of Person objects .....	116
Figure 41: Client UI - editing a note .....	117

## *List of Tables*

---

Table 1: Summary of differences between commercial approaches.....	13
Table 2: Summary of differences between data models .....	28
Table 3: Mapping of Type and Property relationships to RDF .....	46
Table 4: A list of pre-defined custom views .....	61
Table 5: Summary of context menu actions.....	62
Table 6: Summary of differences between related PIM projects .....	107

## 1. Introduction

### 1.1. Motivation

In today's world we are bombarded by digital information from all sides. We receive and share information in the form of e-mails, instant messages, rich text, pictures, documents, music files, etc. This happens all the time, whether we are at home, at school, at work, out on the town or on holidays. We use a multitude of applications which enable us to work with this information. We keep that information on different computers, mobile devices and on the Internet. As a result, the information that we are interested in is highly distributed between formats, applications, devices and locations.

The most important problems arising from the distributed nature of personal information are described below.

Due to the significant number of different applications and devices that are used to handle personal information, this information is highly fragmented (Karger, et al., 2006) and thus difficult to work with. The pieces of information that a person needs to make a decision might be scattered between different places: an email application, a social networking site, an address book on a mobile phone, etc. Sometimes we know that we are in possession of some information, but we abandon hope to find it as it requires too much effort (Teevan, et al., 2006). *At different times we may forget to use information even when (or sometimes because) we have taken pains to keep it somewhere in our lives* (Jones, 2005).

People try to unify fragmented information on their own. The most typical way is to use copy and paste functionality to copy data from one place to another. However, this leads to the duplication of information (Karger, et al., 2006). Duplicated information may quickly become inconsistent if the user updates it in one place, but forgets to do it in the other. It is also quite cumbersome to actually perform such updates. This often leads to a dilemma for users – which place should I use for putting new information?

Another problem arising from the use of different systems is that it is often not possible to link or reference information stored somewhere else (Jones, 2005). A picture stored in a disk file might be associated with a person in an address book, but it is not usually possible to link those two. Studies indicate that following links is the preferred method for finding information, rather than jumping directly to the target (Alvarado, et al., 2003).

Users are also limited on the amount and type of information that applications allow them to store. An application designed for managing pictures and galleries will not usually let the user keep other data that one would associate with them – people, places, events, etc. – or will only allow it to be stored in an unstructured form (i.e. a big text box).

Unstructured data is one that lacks semantics. It is only suitable for consumption by a human being who can understand the information it contains. *Machines cannot easily process this information*, which reduces the value that a computer system can add to it. *For this reason, information systems store data in a structured format wherever possible* (Dittrich, et al., 2005). Unfortunately, a lot of information that people produce is unstructured – e-mail messages,

documents, pictures, audio and video recordings – and poses problems for its automated management.

Structured information, on the other hand, can be queried and viewed in different ways. However, it is currently impossible to perform queries that would combine data from different sources (Dittrich, et al., 2005). Such queries can reduce the amount of manual work that a user would have to perform in order to answer a question like: What are the phone numbers of the people that I am supposed to have a meeting with today? Existing applications also limit the user with respect to the ways the information they contain can be viewed. They often provide just one or a few different ways of presenting it. However, there are unlimited possibilities in which different types of information can be visualized, especially if one takes into account related information.

### 1.2. Problem definition

This thesis proposes a model which allows for unifying distributed information. Unification is achieved by loosening the boundaries between different types of information and different sources. The user should be able to interact with information in a similar way irrespective of what it represents or where it comes from.

The thesis will try to find out how to improve personal information management through:

- integration of various sources of information, such as social networks, email, calendars, documents, images, music files, etc.,
- allowing the user to store different kinds of new information,
- combining, linking, assigning meaning, tags or other metadata to information,
- organizing information into meaningful groups,
- presenting information in a way that is useful to the user, and
- sharing information with others.

During this project a prototype has been constructed with the intention of providing more understanding and insight into these problems. The prototype is also used as a basis for validating the model. However, it is not the aim of the prototype to conduct user experiments with.

### 1.3. Contributions

As a result of this thesis, the following main contributions were made:

- A model enabling the unification of information available in different sources has been designed.
- A method of reducing information fragmentation and duplication through defining equivalent objects has been described.
- An approach to improving unstructured information management by extracting meaningful fragments has been proposed.

### 1.4. Report structure

This report is divided into the following sections:



*Analysis* examines the theoretical problems involved in creating a personal information management system that would realize the goals described earlier and proposes solutions to these problems. *Use cases* investigates a few scenarios to see how PIM processes can be improved. *Design* describes the essential elements for the PIM system and how they function together. *Implementation* discusses some of the design decisions that were made while creating the prototype, but which are not essential to the model of the system itself. *Evaluation* assesses the suitability of the proposed model for solving the problems outlined above. *Discussion* mentions technical problems that were encountered during implementation and alternative design decisions that could have been made. *Related works* compares the model proposed in this thesis to other academic approaches for solving similar problems. The last section, *Conclusion*, summarizes the project and its findings.

## 2. Analysis

This section starts by discussing what personal information is, what are its characteristic features and where can it be found. It continues with a description of three commercial applications for personal information management. Next, an analysis is performed of how personal information kept in different places can be combined together and what problems are associated with this. Finally, a comparison of the suitability of various data models for storing personal information is performed.

### 2.1. Personal information

#### 2.1.1. Definition

Personal information, in popular understanding, is a term whose scope varies significantly from person to person. For some, it represents any information that can be used to identify an individual, such as a name or some government personal ID number. This kind of information is usually associated with the problem of privacy and identity theft (Schoen, 2009). For others, it includes information that one might keep in a paper organizer, such as a phonebook, a private calendar or a to-do list (Teevan, et al., 2006 p. 42). Some make a clear distinction between private and corporate or business-related information, and do not consider the latter “personal” anymore. With the advent of web-based social networks, many people started to regard information exchanged through them, such as photos, videos, links and comments, as personal, as it relates directly to themselves or to their interests (Boyd, et al., 2007).

In fact, any kind of information that is somehow related to a person can be regarded as personal information. This is described by Jones in (Jones, 2008), where he defines six (sometimes overlapping) classes of information based on their relationship to a person, or “me”:

1. Information that is controlled or owned by me.
2. Information that is about me.
3. Information that is directed to me.
4. Information that is sent, posted or provided by me.
5. Information that has been already experienced by me.
6. Information that is relevant or useful to me.

In this project, I am particularly interested in the last class of information. This class can include:

- private or public information,
- information that is directly related to a person (such as a picture of someone), or related through the interests of a person,
- information collected for one’s own private needs, or for professional reasons,
- information that is accessed from home, school or a work environment.

For the purpose of this paper, I will define personal information as any information that is of importance to a person and which the person is interested in keeping track of, which is similar to the definition given in (Larsen, 2005 p. 18).

### 2.1.2. Characteristics

As personal information is such a broad term, it is natural that it includes types of information that differ substantially from one another. One can distinguish several dimensions along which these types can be classified:

#### *Frequency of use*

According to (Cole, 1982), information can be classified into the following categories based on the frequency with which the user refers to that information:

- Ephemeral (also called “action information”) - information which is used now or will be used in the near future; it has a short life time.
- Working – information which is frequently used, such as personal work files.
- Archived – infrequently used information.

In a physical environment, ephemeral information might be stored on stick-it notes placed on a user’s desk, where it is immediately available. Working information could be placed in a document tray so that it could be retrieved quickly. On the other hand, archived information would be put in binders located on a shelf or in a separate storage room, as it is only needed from time to time. The same organization principle should be used in a personal information management system.

#### *Internal structure*

The degree of internal structure determines whether a piece of information stored on a computer is organized into a form suitable for processing by a program. Although personal information is intended for consumption by a human user, it is beneficial when a computer algorithm can be used for its processing. This enables executing structured queries against that information, sorting it according to some property, and so on. *For this reason, information systems store data in a structured format wherever possible.* (Dittrich, et al., 2005)

One can distinguish the following categories of internal structure:

- Unstructured – information that has no internal structure, such as a clear-text note. It is difficult to automatically extract meaningful information from such data.
- Loosely structured – information that is generally unstructured, but which contains embedded markers that allow for processing of certain pieces of it. An example of this kind of information could be an HTML document using microformats (Micro09) – special markup elements that identify specific kinds of data, such as people, events or geographical locations, within the otherwise clear-text.
- Formally structured – information that is stored in a well-defined form, such as data in a relational database. Every piece of information has assigned metadata specifying its meaning – such as a field name or data type.

#### *Topology*

As personal information covers a broad range of subjects, the arrangement of relations between pieces of information can also differ substantially. For example, people in an corporate organizational structure can be arranged in a hierarchy, while the relations between friends in a

social network would rather resemble a mesh, where a person could be linked to any other. Not all topologies are suitable for all kinds of information.

### *Type of medium*

Personal information can include not only text-based data, but also images, audio and video.

### *Confidentiality*

Information in a PIM system can be classified as private or public. This classification is mostly unrelated to the type of information – for example, one appointment in a calendar can be considered by the user as private, while another can be freely shared with friends or colleagues.

### *Source*

Information can be available locally on a user's personal computer, it can be stored on a computer available through a network, or on a mobile device. It can also be available from an Internet web service, such as Facebook<sup>1</sup>. The problem of multiple sources is discussed in more depth in the following section.

#### **2.1.3. Sources**

Personal information can be found mostly anywhere. We keep it on our computers, mobile devices and, in the recent years, increasingly on the Internet. The World-Wide-Web is evolving from non-interactive websites to something that Tim O'Reilly called "Web 2.0". This concept is defined in terms of popular practices and certain principles, such as blogging, wikis, participation, tagging and user contributions over personal webpages, directories and content management systems (O'Reilly, 2005). The Web today is full of services that focus on social interaction and collaboration in different contexts, for example:

- **Blogs:** Blogger, WordPress, Blogspot
- **Communication:** Google Talk, Twitter<sup>2</sup>, Jabber, Google Wave
- **Social networking:** Facebook, Orkut, LinkedIn, MySpace, Meetup.com
- **Applications:** Gmail, Google Docs, Google Calendar, Office Live
- **Wikis:** Wikipedia, Wikia
- **Social news:** Reddit, Digg
- **Photo sharing:** Flickr, Picasa, Photobucket
- **Video sharing:** Youtube
- **Music sharing:** Last.fm, Pandora, Grooveshark
- **Virtual worlds:** Second Life, Lively

As the popularity of those services increases, people use them to store and manage more of their important personal information. In the past, we would store photos from trips in a photo album on a shelf, or, since the advent of digital cameras, burn them on a CD and maybe publish to a private web site for others to see; now we just upload them to Flickr or Picasa. Many people don't use an e-mail client anymore – all of their mail is kept on Gmail, Hotmail, or some other on-line mail service. Remembering about friends' birthdays? This is something that social networks take care of.

---

<sup>1</sup> <http://www.facebook.com/>

<sup>2</sup> <http://www.twitter.com/>

The problem with those services today is that they resemble data silos – separate databases that are in no way connected to each other. (Breslin, et al., 2008 p. 13) It is difficult or impossible to access or refer to information that is stored in one site from another site. In order to get an overview of all the people a person knows, it would be necessary to visit all social networking sites the person is a member of. Some of the friends and colleagues might only be present on one of those sites, while others could be a member of a few of them. As such, the user's personal information is scattered across different websites, with some of it being duplicated in many places.

But even on a single computer, personal information is distributed among different applications. We use one program for managing email, another for writing documents, and yet another for planning and calendaring. It is often the case that information in one application relates in some way to what is stored in another program. For example, a picture in an image gallery application might depict a person whose description is kept in an address book application. However it is usually not possible to create a link between these two information objects, so that the picture could be retrieved from the address book entry and vice versa. In other cases, information is duplicated or fragmented across applications: a calendar application often has an address book of its own, and so does an e-mail client. It is therefore necessary to look in many places to get a complete overview about a particular subject. This can also introduce other problems: *"We may change data in one place (perhaps a new married name in the address book) and fail to change it elsewhere, leading to inconsistency that makes it even more difficult to find information"* (Karger, et al., 2006). This situation also often leads to a dilemma for the user: which application should I use to store my information so that it will be easily accessible later?

#### 2.1.4. Metadata

Finding the desired information that was once entered into a system is not easy. Larsen described that there is an inherent dilemma to the problem of storing and retrieving information: the more effort is spent on properly filing information, the easier it is to later retrieve it. Not all users are willing to invest the time to file the information properly, especially since it is often difficult to identify in advance which information will be needed in the future (Larsen, 2005 pp. 43-45).

However, using metadata can help in information retrieval without requiring too much initial effort from the user. Metadata is extra information describing a piece of information stored in a system. It could be the title of a song, the date when a picture was taken, or a person that is an author of a document. Personal information is closely related to the user, meaning that the user already possesses some knowledge about the piece of information they are looking for. Metadata can therefore help in finding it.

Some metadata, such as types and links, is already kept as part of existing software systems because it is an essential part of how they function. Other metadata, such as the one comprising context, is usually available even though it is not normally recorded by a system. And yet another type of metadata, such as tags, is not assigned automatically by a computer program, but can be quickly and easily stored manually by the user.

### Tags

Tags are single words that somehow describe a piece of information. It should be easy for the user to come up with relevant tags just by thinking about what they associate the information with in their minds. Every piece of information can be tagged with one or more words. In this way, retrieving the information in the future would require the user to think about the possible associations and then browsing through the information items that match those tags.

### Context

Context is a property that applies especially to information gathered using a mobile device. In essence, it is the situation and the surroundings in which the user is in along with their mobile device, including the state of the user themselves. According to Dey, *“context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves”* (Dey, 2000).

Context can include such information as the current time, geographical position, available wireless networks, weather and temperature, movement velocity and acceleration, etc. A sophisticated mobile device could be able to use those parameters to determine the environment in which a user is in (driving a car, attending a university lecture, at work, outdoors – street name, etc.) This information can be attached as metadata to objects created on a device at a given time (photos, notes, messages, etc.) Finding information that is annotated with such metadata would be much simplified for the user, who would just need to answer questions such as: What were I doing? Where were I?

### Links

Information stored within an existing software system is already interlinked with other pieces of information. For example, an email client links messages to contacts (sender, recipients), to other messages (threads), and to folders. Most software is limited in the types of links and the types of information that can be linked together – e.g. it is usually not possible in an email client to link a message to an arbitrary contact, only senders and recipients are permitted. Nevertheless, reusing information contained in existing links can help the user in finding what they are looking for by following links from one item to another.

### Types

Another feature of information stored in existing systems is that it represents some particular thing. A piece of data might be a picture, an email, a document or can represent a person, an organization, or a place. Each piece of data usually has an associated type, which tells the application how should it be treated. A set of types in an application can form a hierarchy – an email could be a kind of document, a person and an organization could be treated as a contact. This hierarchy of types can then be leveraged when searching for information.

#### 2.1.5. Related problem domains

According to Thomas and Jones, personal information management (PIM) is concerned with *“the methods by which individuals handle, categorize and retrieve information”* (Jones, et al., 1997). However this statement does not reflect the broad set of activities and problem domains that are involved in managing one’s personal information. Some of them are listed below:

### *Information management*

Information management deals with problems associated with structuring information, its storage and retrieval. It is related to the following problems:

- choice of database models and systems,
- integration and interoperability between systems,
- defining taxonomies, ontologies and systems for classifying information,
- using machine learning and data mining to improve the quality of information.

### *Content management*

User's personal information is composed of various forms of data – documents, movies, pictures, contact information, web bookmarks, etc. A personal information management system must therefore, in some ways, resemble a Content Management System (CMS) such as the ones used for publishing information on web sites. Such a system aids in storing, classifying and retrieving data. Often these systems operate on the principle of the separation of content from its graphical presentation. They let the user define a custom data model and a set of templates which can be used to visualize the data corresponding to that model.

### *Time management*

Time management includes the activities associated with keeping appointments and dividing time between different tasks that need to be performed. It involves the problems of effective reminding about upcoming events, prioritizing tasks so that a person is not overwhelmed by the amount of work that needs to be done, and at the same time making sure they are accomplished before deadline.

### *Contacts management*

Contacts management involves storing information about people and organizations and the ways they can be contacted.

### *Communication*

This problem involves using different channels of communication (such as email, instant messages, phone calls, text messages, or even status updates on social networks) and managing information exchanged through these channels.

### *Psychology*

The differences between individual human beings in their approach to organizing information have a profound impact on the PIM domain. Research by Malone (Malone, 1983) identified two distinct styles in which people organized their offices – a “neat” style, where the flow and location of documents is structured (e.g. from an in-basket to an out-basket, optionally through a “hold” tray), and a “messy” style, where documents are stacked into many piles with the most recently used located on top. Obviously, any PIM system will have to take such individual differences into account.

But research into human psychology is important also in other areas, such as *problem-solving, decision-making, and categorization*. For example, work on project such as “*plan my wedding*” can be viewed as an act of problem-solving and folders created to hold supporting information may sometimes resemble a partial problem decomposition (Jones, 2005 p. 20)

### *Group information management*

Personal information management involves communication and cooperation with other people. The term *group information management* (GIM) is used by Erickson to describe the problem of *how personal information is shared with a group, emphasizing the norms that underlie that sharing, as well as the ways participants negotiate these norms in response to the tensions that sharing inevitably produces* (Erickson, 2006).

## **2.2. Commercial management approaches**

The following section describes three applications that have been on the market for a significant amount of time and represent different approaches to managing personal information.

### **2.2.1. Microsoft Outlook**

Microsoft Outlook is one of the most popular applications in the class of personal information managers offering the calendar-contacts-todo's functionality. This group also includes Lotus Organizer, Novell Evolution, and others.

Microsoft Outlook can be used to manage e-mail messages, contacts, calendar events, tasks (to-do items) and short textual notes. These items can be placed into folders, such as Inbox, Address book, or Calendar. The application provides the user with a very typical interface for working with these kinds of information, and it therefore feels familiar to new users. As such, it is an example of a desktop approach (Larsen, 2005 pp. 82-84).

However, the structure of data that can be stored in Outlook is very rigid and inflexible. Organizing items into folders means that any item can only be stored in a single folder. The user is also limited to working with only those information types that the application supports, such as contacts and calendar events. It is not possible to manage any other information. Moreover, it is difficult to quickly store any additional information in a structured way. Most of the time users have to rely on a generic "notes" or "description" field for storing such information. Similarly, it is not easy to organize information by linking it together. For example, putting a link to a to-do item in a calendar event does not automatically create a reciprocal link, and specifying an additional date for a contact (such as another type of anniversary) does not make it appear in the calendar. Finally, linking external information can be problematic, as the application will try to embed files instead of linking to them.

### **2.2.2. Microsoft OneNote**

Microsoft OneNote (Microsoft) is a note-taking application that is part of Microsoft Office and was first released in 2003. It uses the metaphor of a tabbed notebook for its user interface. This metaphor also dictates how the data is organized: all user-created content in OneNote is placed on pages, which are organized into sections and notebooks. Pages can be used for writing text, creating tables and bulleted lists, placing files, images and screenshots from other applications and drawing with simple vector graphics.

OneNote makes it easy to create a note in a similar way as one would jot something down on a piece of paper. For example, one can use OneNote to take a screenshot of a different application, draw arrows pointing to various places in the screenshot and add some explanatory text; or to paste an excerpt from a web page and use "electronic marker pens" to highlight the important



fragments. As these kinds of tasks can be accomplished relatively quickly, it makes the application useful for people making a lot of notes.

A notebook page is the equivalent of a piece of paper, however the individual elements of the page are still separate. It is possible to move them around on the page, or to drag them over to another page. Elements can be marked with customizable tags (which actually do not resemble tags in the sense they were described in Section 2.1.4, but rather categories), such as “*idea*”, “*important*”, “*note to self*”, which can help in finding them later through the *Search* feature.

The application suggests two strategies for creating new notes: either use the so-called *Unfiled Notes* section for creating a draft note and later deciding where to put it, or to create a new page in an already existing section. Either way, moving pages from one section to another or reorganizing entire sections can be done by dragging.

Microsoft OneNote provides the user with good functionality for storing different types of information, and has some support for incorporating information from different sources (such as local files and directories, but also web content). The main drawback of the application is that it only supports unstructured information, which means that it is not possible to automatically extract and reuse certain pieces of it. It is also not possible to view this information in any other way than as notebook pages. The information stored in OneNote lacks meaning to a computer.

### 2.2.3. Ecco Pro

Ecco Pro is a personal information manager that was originally developed in 1993 by Arabesque Software and later purchased by NetManage, Inc. The application has been discontinued in 1997 and later released as freeware. It has managed to acquire a devoted group of users who try to create patches and extensions to the software, even though the source code has never been released. (Dohmann)

Ecco Pro is an application that offers a great deal of flexibility to the user in terms of data storage. By default, it aids in managing calendar events, to-do items and contacts (called *phonebook* entries), but it can be also used to store many kinds of other information and to view it in different ways.

All information stored in Ecco can be regarded as being placed in a single big table. The table contains *boolean* columns specifying whether an item is an appointment, a to-do item, a phonebook entry or if it belongs to some other, possibly user-defined, category (called *folder* in Ecco terminology). The table also contains columns for every possible type of value that can be associated with an item. New columns may be created by the user, if needed. For example, appointments have start dates and end dates, to-do items have due dates and phonebook entries hold telephones, addresses, names and birthdays. As all the columns belong to the same table, it is possible for a single item to be associated with many folders. Effectively, Ecco Pro’s data model can be considered an attribute-oriented approach, as described in (Larsen, 2005 pp. 92-96).

The information in Ecco is presented to the user through *views*, which are user interface elements specialized for displaying certain kinds of information. There are three types of built-in views available in Ecco: calendar, phonebook and notepad.

The calendar view employs the metaphor of a personal organizer (see Figure 1) to give the user an overview of upcoming appointments and other important events. The user is presented with a day planner on which the duration of appointments occurring on a given date is marked. A list of *ticklers* is also shown. It typically includes to-do items which have been defined and await completion, but can also display important occasions (such as Mother's day, holidays, etc.) and birthdays of contacts from the phone book. In general, the *ticklers* list contains a list of all items which have dates that coincide with the current day. In this respect, data in Ecco is unified – the application treats all types of items in the same way. The flexibility of this approach is well illustrated by this example: it is enough to define an *Anniversary* field of type *date* for phonebook entries and assign a value for a particular entry for it to be displayed in the *ticklers* list.

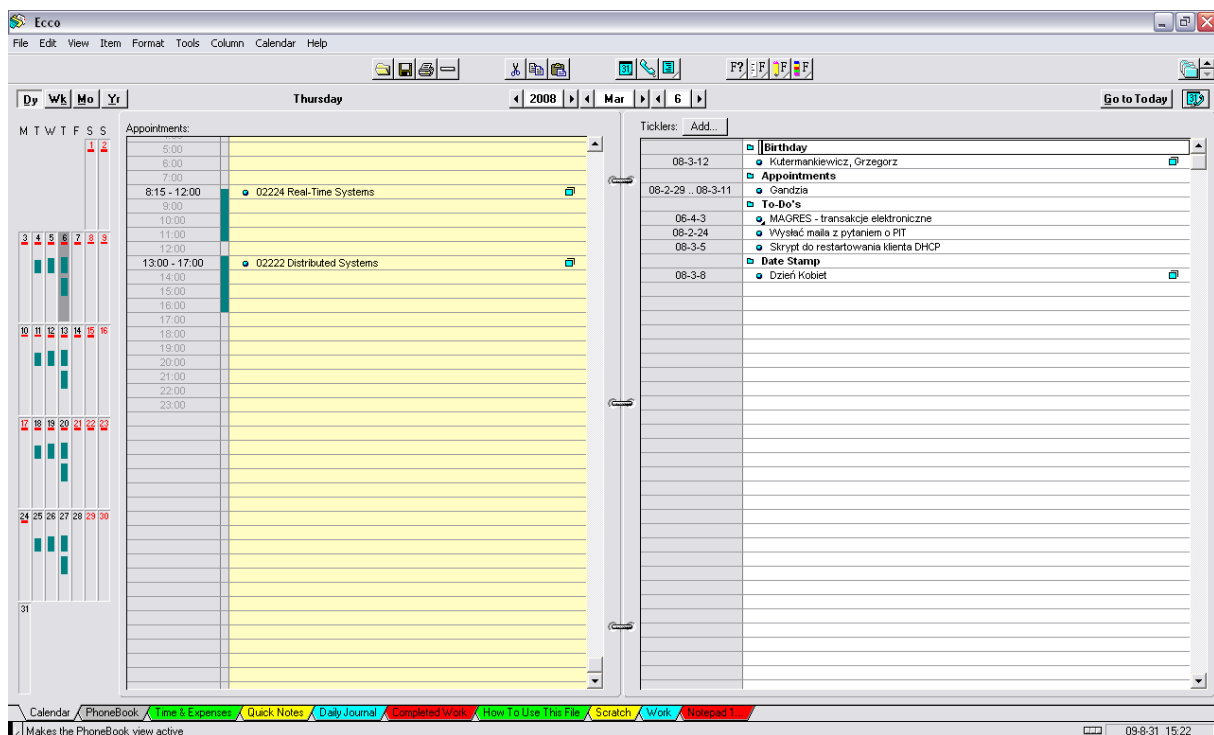


Figure 1: Ecco Pro's calendar view

Ecco gives the user the possibility of defining their own data structures for storing information that was not foreseen by the program authors. Its main drawback, apart from the dated user interface and discontinued development, is the very limited possibility of linking and including external information.

#### 2.2.4. Summary

The three applications described above represent three different approaches to managing personal information. The table below provides a summary of these approaches:

	Microsoft Outlook	Microsoft OneNote	Ecco Pro
Data organization	Desktop approach	Tabbed notebook	Attribute oriented
Internal structure	Structured	Unstructured	Structured
Multiple views	Yes (limited)	No	Yes
Linking	No	Yes	No
Annotating	Categories	Categories	Additional folders
Unification of data	No	n/a	Yes

**Table 1: Summary of differences between commercial approaches**

A lot of research into flexible data models and organization schemes is being conducted, as exemplified by the projects described in Section 7.1 (Related works). It is driven by the premise of giving the user the power to organize their data in the way they want to. Only Ecco Pro's data model supports defining new types of data and modifying existing ones. This is also the only model that allows for different types of information to be treated in a similar way by the application. The user can view different types of items on a single list and use different UI's offered by the program to interact with that data. In comparison, Microsoft Outlook lets some data to be viewed in different ways (e.g. calendar entries in a calendar view or as a list), but it does not allow mixing of types.

On the other hand, Microsoft OneNote shows that flexibility can be also achieved by simply allowing only unstructured information, which the user can organize through bulleted outlines, notes and pages. Unfortunately, such an approach eliminates the possibility of viewing the information in different ways – it is all flat text. It is also impossible to talk about unification of data in this context. All information is already treated by the application in the same way – the only way it can be.

All three approaches are lacking in terms of grouping and annotating information. In Outlook, items can be assigned to one or more categories. In OneNote, notes can be annotated by using “tags”, which are functionally identical to Outlook's categories (i.e. one must create them beforehand, and not type them when needed). Ecco Pro lets the user put an item into multiple folders. Only OneNote supports some form of linking – it is possible to create a link to a text fragment or to a whole page or section. However, it would be difficult in all of these applications to fulfill the individual needs of different users.

### **2.3. Combining distributed information**

It was mentioned earlier that personal information tends to be highly distributed among different applications, computers, mobile devices and web services. However pieces of information stored in different places are usually related to each other. They can represent similar types of information, and can either extend or duplicate the user's knowledge about some things. Let us analyze some possible relationships between an e-mail client, a calendar application, local disk files and a social networking website (such as Facebook):

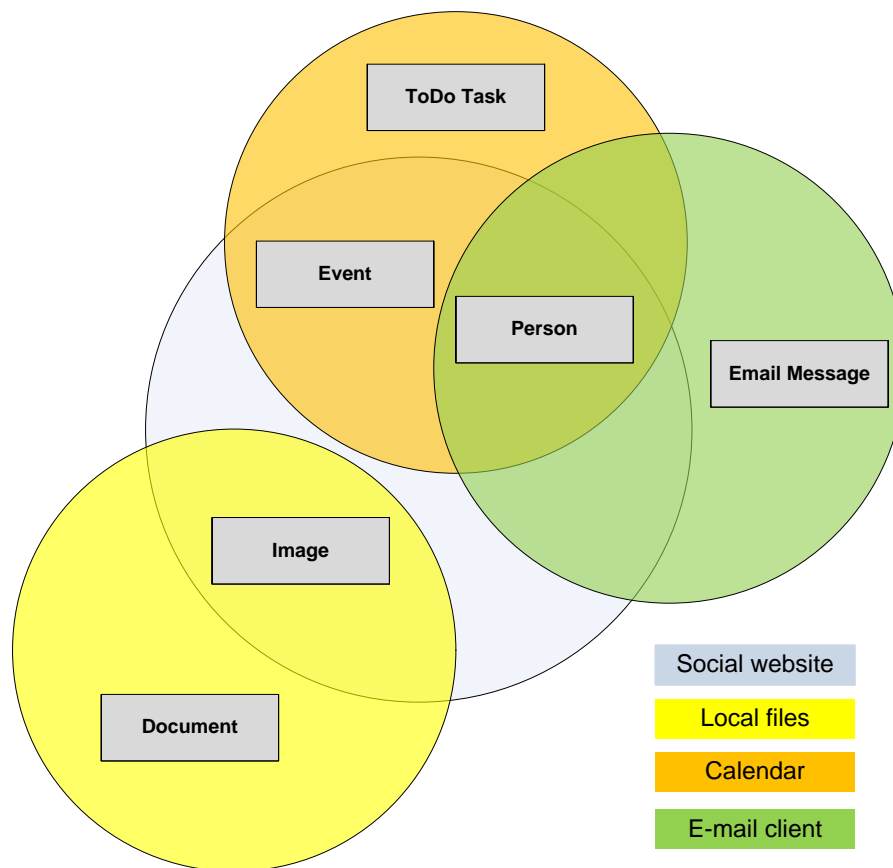


Figure 2: A diagram of relationships between information sources

An email client basically handles two types of information: email messages and contacts in the address book. A contact usually represents a person, which is also the central type of information for social networks. Social websites typically let people share messages, pictures and events between their friends. Events and persons are also central to calendar applications, where an event involving some specific persons can be called an appointment. Images can be shared through a social website, but they are also found as files on the user's computer.

### 2.3.1. Applications and concepts

The example applications from the previous section make use of concepts that represent things which exist in the real world, such as persons, appointments, places or projects. These are concepts that a user would mentally associate with information available through an application, even if the application assigns a very specialized meaning to them, such as contact, calendar event, geographical coordinates, etc.

Many applications are capable of handling information that represents the same concepts. For example, both an email client and a calendar use the concept of a person, either as representation of a sender/recipient of an email, or as an entity that can be associated with appointments. The *person* concept also plays a key role in social networks.

Other applications process information that is related to certain concepts, but the applications themselves have no special means of handling them as separate entities. Unlike an email client, where persons are presented as contacts, an image viewer might have no way of specifying that a picture depicts a person. The user could write the persons' names in a generic description field

for the picture, but this would make it impossible for a computer program to process this information as it would be unstructured.

Some concepts are not directly related to the information that a given application is concerned with. In the user's mental model, things relate to each other in complex ways. These relationships are usually not captured by applications, which only have a limited view of the "big picture". A spreadsheet document might be a part of a project involving many people who exchange email messages and have meetings with each other. In that example, the concept of a project is external to all the applications involved: an email client deals with emails, an office productivity suite allows for editing documents, a calendar application can be used for planning appointments. The applications can store some of the user's knowledge, while other parts need to be remembered by the user themselves. Moreover, one might imagine that a user would like to store more information than these applications allow for – write notes about a meeting, mark things as "to-do", etc.

### 2.3.2. Duplicated and fragmented information

The *concepts* described in the previous section can be seen as a high-level representation of information that is stored in various sources – a representation that is meaningful for the user. In a computer system, information is usually stored in the form of data accessible by applications. Applications are designed to fulfill a certain purpose – handle email, organize pictures, play music – while interoperability and exchange of data with other applications are usually not the key functions. As such, it is not easy to reuse information that has been stored using another application. Information relating to the same real world object might therefore be spread out across a number of applications, where each of them holds only a portion that is relevant to the application's function. On the other hand, the same information might be stored in many applications, however not necessarily in the same form.

Karger and Jones describe some existing approaches to dealing with this problem in (Karger, et al., 2006). In the simplest case, the user can get an overview of all information about an object by opening many applications and viewing them in windows, side by side. One can also copy-and-paste textual information from one application to another. In those two cases no permanent links between the information objects are established and the user would have to manually repeat the operation every time. More advanced approaches involve grouping similar objects based on metadata (ID3 tags, file metadata), as implemented in Google Desktop and Yahoo Desktop.

They also describe the Universal Labeler (Jones, et al., 2005) prototype, which allows the user to copy information from various application into the Labeler, where it can be organized as the user sees fit, while keeping links back to the original applications.

In order to solve the problem of duplicated and fragmented information, objects in different applications which represent the same real world object should be represented as a single entity. This single entity would replicate the user's mental model of such information. Creating such an entity would involve:

- Identifying application objects that represent the same real world objects and merging them together, creating a union of information available through different sources.
- Identifying and removing duplicate information.

However, it is not enough to just create such a combined representation. Any changes to this new object need to be propagated back to the original applications, and vice versa.

In practice, even merging information from two objects is a complex task. In particular:

- Merging objects requires duplicate information that would allow them to be identified as a single entity. For example, a person's name should appear in both objects.
- The fact that two strings are equal doesn't imply that they represent the same piece of information. For example, two people might have the same date of birth. This suggests that only certain types of information should be used for comparison.
- Unrelated similarities might occur even among types of information that qualify for comparison. For example, two different people might have the same name, even though names in general are used for identification. On the other hand, matching personal identification numbers (such as social security numbers, etc.) or even email addresses could be a strong indicator of a correct match.
- Duplicate information may be represented in different forms. For example, a name of an organization might be written as "DTU" or "Danmarks Tekniske Universitet".

An automatic process that gradually resolves simple matches between objects and uses them to infer more complex ones is described in (Dong, et al., 2005). However, any automatic process can introduce errors. The user would have to be able to review the automatic changes and optionally correct or undo them.

### 2.3.3. Applications and data

As has been shown in the previous sections, many applications make use of the same concepts. However, are the representations of concepts in different applications compatible with each other? For example, will a person always be represented in all applications by a name, which is a text string, a birthday (a date), and a photo (an image). When one thinks of a person, we naturally associate these kinds of properties with that concept. Similarly, a photo album will, in general, be a collection of pictures, a calendar event will have a title and a date, an email message will have a sender, recipient, subject and a body, and so on. It is therefore reasonable to assume that different applications will have a similar representation of the same concept, because this is what the user expects. Some observations follow:

- Some concepts will always have a certain base set of properties (e.g. name, sender, collection of pictures, etc.) A specific application, however, might be able to process a richer set of properties.
- Some properties will always have a fixed data type (a name is always a text string, a date of birth is a date, an email sender could be a person, organization or possibly a computer program).
- Some properties might contain various types of data (an email body could be text, a picture or an attachment).

As such, it is possible to construct a common representation of a concept shared by many applications and populate it with data from those applications. This common representation will, however, have to be the lowest common denominator of the information contained in various sources. This is because the different applications, and the PIM application itself, won't be able to

handle specialized types of content. For example, a picture gallery might contain a vector-based drawing which will have to be converted to a bitmap image (losing some of the inherent information), so that it can be displayed in an application that has no notion of vector graphics.

This problem is mentioned by Karger and Jones: *“Using a common denominator is, however, in tension with each application’s need for rich, specialized representations of its content. Rich representations let applications deliver powerful domain-specific operations.”* (Karger, et al., 2006). It is important that a system for managing personal information would not “reinvent the wheel”. It should not provide functionality that already exists in other applications. Instead, it should direct the user to a suitable application where a given task can be completed. The PIM system should be limited to combining, linking and viewing information in ways that the original applications do not support. Other reasons for limiting the functionality include:

- Existing applications are designed to perform certain functions and they can do them well.
- People are used to their applications. They won’t abandon their e-mail client just so that they can annotate emails with extra information.
- Rewriting existing applications to add a information management features is a waste of time.

#### 2.3.4. Linking and annotating

As stated by Vannevar Bush, the human mind operates by association of thoughts. Every object in a person’s mind is always linked to other objects and concepts (Bush, 1945). A system for managing personal information should therefore make it possible for the user to link information in various, non-predefined ways, as this would be a natural process for the user. In the context of distributed personal information, this means that it should be possible to link information originating from different sources as if it were stored in a single place. For example, a user might decide to link an MP3 file with a picture on Picasa and a friend on Facebook, because the song reminds him of an evening they spent together. Such an association in a person’s mind can bring back memories, and a similar approach can be implemented in an application: when viewing the details of a particular piece of information, the user might be given the opportunity to explore the relations it has to other information. By following links from one piece of information to another, the user might be able to find what he or she is looking for. Studies indicate that users prefer this behavior – i.e. navigating in small steps – than jumping directly to the target (Alvarado, et al., 2003). Moreover, most users will be familiar with the way that such a system works – it is a typical method of retrieving information in a hypertext system, such as the World Wide Web.

#### 2.3.5. Technical requirements

In order to combine distributed information there are a few requirements for a system that would make this possible.

First of all, there needs to exist a common way to address objects from different sources. *“As with grouping and annotation, linking requires only a shared namespace with which to name the linked objects and a common syntax for describing the relationship between them.”* (Karger, et al., 2006). If one looks at a computer system as a whole, there are multiple naming schemes in use by applications, operating systems, networks and services. Some of them are clearly visible,

such as the file and directory hierarchy of a filesystem, the Universal Naming Convention (UNC) for accessing resources in Microsoft Windows networks (Microsoft), or the Uniform Resource Locator (URL) for addressing resources accessible over the Internet (Berners-Lee, et al., 2005). Others are application-specific and not easily accessible from the outside: *“different applications insist on managing collections of their own information in their own “internal namespaces,” files go into file folders, email messages into email folders, Web references into bookmark folders accessed through Web browsers, and address book entries into address book folders.”* (Karger, et al., 2006).

As such, there is no single “universal” namespace that would be understood by all components of a computer system. A new namespace could be created, or an existing namespace could be reused for this purpose. The best candidate for this is the URI namespace used on the World-Wide-Web, not only due to the sheer number of resources that are already accessible using this scheme, but also because it is possible to construct globally-unique names (Berners-Lee, et al., 2005 p. 20).

Secondly, a way to list, read, write and delete objects from different sources is needed. This requires a separate data interface layer for each application – one which would either use the application’s API, if available, or operate on the application’s data files directly. The data interface would have to implement the following functions:

- Conversion between the PIM system namespace and the application’s internal namespace for addressing objects. This is needed so that modifications to an object done in the PIM system can be propagated back to the owning application, and vice versa.
- Discovery of new, modified and deleted objects. As the external application has no knowledge of the existence of the PIM system, it cannot notify it that a change has occurred. The data interface layer has to keep track of all objects and scan for modifications. This kind of approach is implemented in the Aperture framework (Ape09).

Finally, as the PIM system is intended to help manage information and not replace application-specific functionality, a way to quickly open an external application for editing a specified object from within the PIM system is needed. In case of most web applications, the PIM system just needs to direct the user to a specially crafted URL - no special support from the web application is usually needed. On the other hand, desktop applications need to have built-in support for this kind of functionality. For example, an e-mail client might accept an identifier of an e-mail to be opened as a command-line argument.

Moreover, it should be possible to open the PIM system from within a third-party application to manage a specific object. This, however, might require modifications to the application itself. In case of web applications, a browser plugin might be constructed that offers to open the PIM system when it detects that a user is viewing a certain object - for example, by analyzing the active URL. Some desktop applications also support a plugin architecture, which would make a similar solution possible. In case of other closed-source applications, it might not be possible to implement this kind of functionality at all, as only the application’s author could introduce such modifications.



## 2.4. Representation and storage

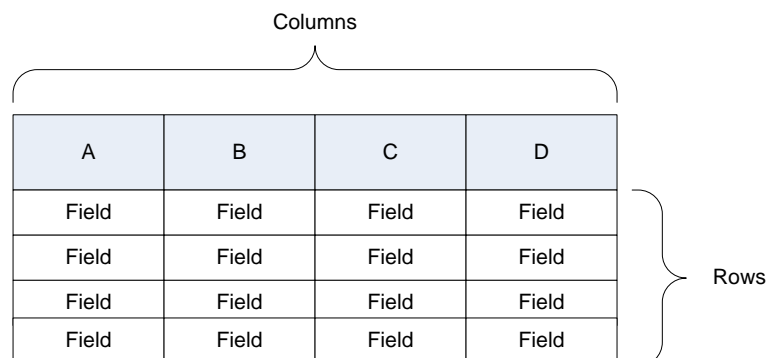
Personal information comes in many shapes and sizes. It can include primitive data such as plain text, numbers and dates, more complex forms such as e-mails, phone book entries and rich text (e.g. HTML or word processor documents). It can also include multimedia content like images, audio and video. Some data may be stored locally on a computer, while other may be accessible over a network. Combining such a wide range of data types in a single system requires a choice of a suitable data model.

### 2.4.1. Files and directories

In the simplest case, one can use the computer's file system to directly store personal information: files would store data about some subject, while directories would be used to group them together. Although simple to implement, this approach has many drawbacks. First of all, standard text files are flat – they don't have any internal structure which would allow for storing complex information. The user may store information about person A and person B in completely different ways. As such, text files are usable only by humans – it is difficult for a computer program to process their contents, and what follows, to perform more complex queries than a simple full-text search. Moreover, the hierarchical way of organizing files into directories is only suitable for certain types of information - those which naturally possess a tree structure. For others it will cause difficulties for the user to choose one of equivalently valid dimensions as the one that should be used for grouping. Finally, a file system has no means of referencing information located on the Internet, which means that it cannot be (e.g.) grouped into directories.

### 2.4.2. Relational model

The relational model is the most widespread model for storing information in use today. It has become nearly synonymous with the notion of a database. It was first proposed by E. F. Codd of the IBM Research Laboratory in San Jose in 1970 and is based around the concept of a *relation* (usually called a *table*), which holds an unordered set of *tuples* (*rows*). Each tuple in a relation has exactly the same structure and is composed of *attribute* (or *column*) name-value pairs. In essence, a relation in the relational model describes a table composed of rows and columns. Each column has a name and an associated data type (such as an integer, string, date, etc.), while each row holds values for those columns (see Figure 3).



A	B	C	D
Field	Field	Field	Field
Field	Field	Field	Field
Field	Field	Field	Field
Field	Field	Field	Field

Figure 3: Table in the relational model

The approach for storing data in this model is to create a separate table for each type of information to be stored. For example, information about people is stored in a Persons table,

while descriptions of photos could be put in a Pictures table. In order to link photos to the people that are depicted in them, a third linking table would be created.

Storing personal information in a relational database provides some significant advantages over the use of plain text files. The most important one is structure: every piece of information is stored in a field, which has a corresponding name and data type, belongs to a row and to a table. Because of that, it is possible to process the information using a computer algorithm. Moreover, the relational model specifies a language, called the Structured Query Language (SQL), which makes it possible to query the database about certain pieces of information – e.g. list the names of persons that live in Copenhagen and are 25 years old. Another advantage is the possibility of linking related information together through the use of foreign keys, which are references between rows in (usually different) tables. Finally, the popularity of the relational model translates into a wide availability of efficient and mature tools supporting it.

Despite its advantages, the relational model has some considerable drawbacks in the context of storing personal information. They originate mainly from its rigid structure:

- The structure of personal information evolves over time, while the relational model requires that it remains relatively fixed. For example, if a user decides to change the way they describe pictures, then the structure of the Pictures table needs to be modified, which in turn requires that all previously described pictures to be adapted to the new format. A single table cannot have two separate structures.
- It is difficult to add information to some rows in a table and not to others. For example, to store a bank account number in connection with just a single person will require the addition of a new column to a table, which is a cumbersome solution as it will only be used once.

### 2.4.3. Object model

The object model is an approach to organizing information that is used in the object oriented programming (OOP) paradigm, which became popular in the 1990s with the advent of such languages as C++ and Java. Most of the unique features of this model, such as encapsulation, abstraction and polymorphism, are of particular interest to programmers, as they make it easier to understand and manage the structure and relationships within the program, and also provide means for reuse of existing functionality, which results in less duplicated code.

An *object* is a data structure containing data-carrying fields and methods (program logic) that operate on this data. In principle, the data contained in an object should only be exposed to the outside world through the object's methods. In this way, the object is responsible for maintaining the integrity of its data.

The structure and functionality of an object is determined by its *class*. It specifies the fields that comprise an object along with their data types, and the methods that define the actions that an object can perform. An object that belongs to a certain class is called an *instance* of that class. Classes can be derived from one another through *inheritance*. When a child class (subclass) inherits from its parent (superclass), it adopts the fields and methods of its parent. For example, *Garfield* is an instance of *Cat*, which is a subclass of *Animal*.

In some respects, the object model is similar to the relational model. Instead of creating a table, one can create a class. Objects can be thought of as just rows in a table. But the object model provides some useful features in terms of personal information management:

- Inheritance allows for the creation of more general or more specialized types to store information as needed. For example, a *Document* class may specify a *Title* and *Author* fields, while a subclass called *BlogEntry* would add an *Url* field.
- Multiple inheritance, a more robust form of inheritance that is not so widely supported, allows a single class to inherit from multiple other classes at the same time. It makes it possible to combine many simple classes to form a complex one. For example, a *Picture* class might inherit from a *Document* class (*Title*, *Author*), an *InternetResource* class (*Url*) and a *Commentable* class (a class of objects that can store textual comments about themselves).
- Linking objects together is as simple as specifying that a class should have a field whose type is another class. A *Person* might have a *Photo* field of type *Picture*.

Still some of the drawbacks of the relational model apply also to this one. The object model also introduces a few problems of its own:

- The object model provides no generic way of querying data. As the data within an object is only exposed through its methods, there is no way to ask specific questions about it. Some object-oriented databases support a query language, but no dominant standard, similar to SQL, exists.
- The popularity of the object oriented programming did not translate into the popularity of object-oriented databases. In practice, most computer software uses a relational database as a way of storing object oriented data. Such a solution requires an additional layer which translates the object data into relational form, and vice versa, leading to problems collectively described as the object-relational impedance mismatch.

#### 2.4.4. Associative model

The associative model of data, as described by Williams in (Williams, 2000), is substantially different from the record-based models such as the ones described above. In this model, all information is stored in the form of *items* (or “entities”) and *links* (or “associations”). An item is anything that has a *discrete, independent existence*, while a link is a thing *whose existence depends on one or more other things* (Williams, 2000 p. 84). In practice, an item can represent a person, a book, a geographical location. A link will usually be a property linking two items, such as “name” (a property of a person), “location” (an association between a physical object and a geographical location), etc. Strictly speaking, an item is composed of a unique identifier, a name and a type. A link contains its own unique identifier and three other identifiers: of a source, verb and target, all of which can point to either an item or another link.

For example, storing the following information:

*John meets with Mark in the student pub on Monday at 12:00.*

would involve the following items and links:

*John **meets with** Mark*  
*... **in** the student pub*  
*... **on** Monday*  
*... **at** 12:00.*

In the above notation, the first link has the verb “*meets with*”, which connects the source and target “*John*” and “*Mark*”, respectively. In this case, the source, verb and target are all items. The second link has the first link as the source and uses the verb “*in*” and the target “*the student pub*”, and so on.

The same information can be presented along with the unique identifiers in this way:

Items		Links			
Identifier	Name	Identifier	Source	Verb	Target
I1	John	L1	I1	I2	I3
I2	meets with	L2	L1	I4	I5
I3	Mark	L3	L2	I6	I7
I4	in	L4	L3	I8	I9
I5	the student pub				
I6	on				
I7	Monday				
I8	at				
I9	12:00				

Figure 4: Items and links in tabular form.

The above example shows that with the associative model it is possible to store information without specifying any predefined structure, such as a database schema. In this model, a type system is a feature that can be used whenever necessary, however it is not required. The type system makes it possible to group entities that have similar associations together and to specify which associations should we expect from a particular entity, along with some more additional properties:

*John **is a** Person*  
*Person **has** (**first name** String)*  
*Person **has** (**birthday** Date)*  
*Person **has** (**parent** Parent) **cardinality** 2*

In the above example, we state that any entity of type *Person* (such as *John*) has *first name*, *birthday* and *parent* associations. We also specify the types of their targets. Moreover, we are saying that each *Person* must have exactly two *Parents*.

Types can form hierarchies, where each type can be a subtype of one or more supertypes - similar to the object model’s multiple inheritance. A subtype inherits all the associations of its supertypes and their supertypes.

*Parent **is subtype** Person*

As can be seen from the examples above, the schema is defined in terms of the same items and links as ordinary data. This makes it possible to manipulate the schema in the same way as normal data, and even combine it with normal data. From the software developers point of view, it reduces the amount of work that is required to create an application with a user-modifiable schema, as the same code and user interface elements that are used for interacting with normal data can be used for working with the schema.

People accustomed to the relational model might find this way of storing information odd, but Williams argues that it resembles reality more closely than what can be accomplished in other models. For example, the typical way to represent a customer in an application would be to create a separate *Customer* table, in which every row would have an independent existence and would be related to other information through foreign keys. However, a customer is not an independent entity – it is just a role that one associates with a person or company in some particular context; someone else might consider the same person to be a supplier or an employee, etc. (Williams, 2000 pp. 87-90)

The associative model is a powerful model that is well suited for storing personal information. One of its main advantages is the ease with which links can be created. The ability to create links between any two items is inherent to the model and is not a consequence of some particular database design, as is the case with the relational model. Here the user can freely link items without being restricted by the possible cases that were foreseen (or not) by the designer of the data model. Moreover, in the associative model the subject of a link is not restricted to just items, but can include other links, giving the user even more freedom. Such freedom is important from the personal information management point of view, as in the human mind associations between thoughts can be formed in complex ways (Bush, 1945).

The primary drawback of the associative model is its poor adoption. It is difficult to find any implementations of an associative database except for Sentences (Lazysoft), developed by LazySoft - the company associated with Williams.

#### 2.4.5. RDF model

The Resource Description Framework (RDF) is a language designed for expressing information about resources that can be identified on the World Wide Web. It is part of World Wide Web Consortium's (W3C) vision of the *Semantic Web*, in which computers will be able to understand the contents of the data put on the web in order to perform the more tedious tasks which now need to be handled manually by humans (World Wide Web Consortium, 2001). Even though this goal is still far from being realized, the activities of W3C in this area have resulted in the creation of a few technologies useful for information management purposes.

Information in RDF is expressed in the form of statements. Each statement is a *triple* composed of a subject, a predicate and an object, similar to a natural language. For example, in natural language the following triple:

<a href="http://www.dtu.dk/">http://www.dtu.dk/</a>	was created by	Technical University of Denmark .
(subject)	(predicate)	(object)

states that the creator of the given webpage is the Technical University of Denmark. However, for this statement to be understandable by a computer program, the natural language elements

need to be replaced by machine-processable identifiers. In RDF, resources about which statements are made are identified using Uniform Resource Identifiers (URIs). As anyone can construct a globally unique URI<sup>3</sup>, it is easy to provide unique names for things that are both available on the World Wide Web and real world objects, such as people. To express the previous triple in machine-readable form, we could say:

```
<http://www.dtu.dk/>    <http://purl.org/dc/elements/1.1/creator>
    "Technical University of Denmark".
```

This triple refers to a special creator URI that is part of the Dublin Core Metadata Initiative (DCMI) terms (DCMI) – a standardized vocabulary for expressing certain properties of documents.

All RDF statements can be presented in an equivalent form as graphs. The graph in Figure 5 illustrates a set of three triples, which state that <http://www.dtu.dk/> has a creator whose name is “Technical University of Denmark” and which is based near “Kongens Lyngby, Denmark”:

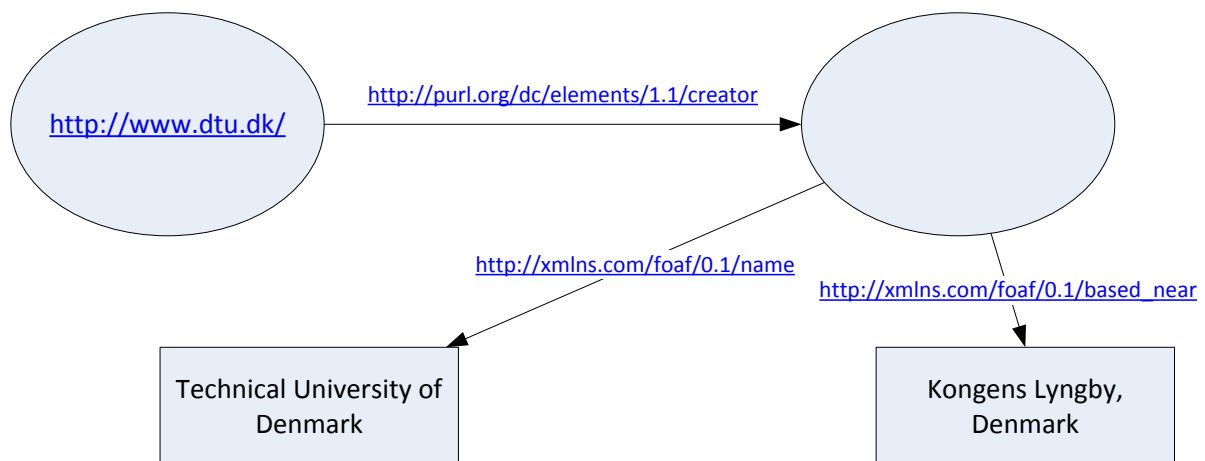


Figure 5: Sample RDF graph

### Schemas and ontologies

RDF by itself can be used to express simple statements about resources, similar to those presented above. However, much of the power of RDF comes from the ability to create and use *ontologies*. An ontology is a standardized vocabulary of terms and logical relationships within a knowledge domain.

A standardized vocabulary allows different parties to agree on the same language to use when exchanging data. As we have seen above, an ontology for describing documents (DCMI) lets us specify that some object is a creator of a document, with the meaning of the term “creator” being exactly defined in the ontology as “*An entity primarily responsible for making the resource*”. Figure 5 makes use of another ontology, called Friend-of-a-Friend (FOAF), which lets us express information about names and locations of objects in a way that will be understood by others. In terms of personal information management, having a standardized vocabulary means that data from third-party applications and web services can be easily accessed when it is exposed in RDF form.

<sup>3</sup> A unique URI can be constructed by registering a domain name and using it as the base for the address.

The ability to define logical relationships makes it possible to capture some of the knowledge associated with a domain so that it can be applied automatically by a computer program. The types of logical rules that one may define depend on the ontology language that is used. One of such languages is RDF Schema. It provides a vocabulary for describing classes of resources, properties and relations between them in a way that resembles the object model of data (i.e. with inheritance). With RDF Schema one can state that, for example, a *Dog* is a subclass of *Animal* and it has a property *Fur color*.

Another set of languages for authoring ontologies is called the Web Ontology Language (OWL). These languages are more expressive than RDF Schema and allow for specifying such aspects as transitive, symmetric or inverse properties, the cardinality of relationships, equivalence of classes and properties and identity of individuals. For example, OWL makes it possible to define a property “is a friend of” to be symmetric, which means that if *John is a friend of Peter*, then, automatically, *Peter is a friend of John*.

Today, according to the Swoogle home page, there exist thousands of ontologies for describing different things. They are tracked and indexed by specialized websites and search engines, such as:

- SchemaWeb (<http://www.schemaweb.info/default.aspx>) ,
- DAML Ontology Library (<http://www.daml.org/ontologies/>), and
- Swoogle (<http://swoogle.umbc.edu/>).

Some interesting ontologies include:

- Friend of a Friend (FOAF) – a vocabulary for describing people and the links between them (<http://www.foaf-project.org/>),
- Dublin Core Metadata Terms – a vocabulary for describing digital content (images, videos, sounds, documents, text, etc.),
- Personal Information Model (PIMO) – NEPOMUK project’s vocabulary for expressing personal information of individuals.

### *Serialization, stores and querying*

RDF triples can be stored in many different formats. The most popular of these is RDF/XML, which is simply an XML syntax for RDF data. In this paper, however, most RDF examples are given in the Turtle (Terse RDF Triple Language) form, which is more compact. In Turtle, every statement is written as a subject, predicate and object, separated by spaces and terminated by a full-stop (“.”). A subject and predicate can be either a full URI enclosed in angle brackets (e.g. `<http://www.dtu.dk>`) or a namespace-qualified node (e.g. `dc:Title`). An object can be written in the previous two forms, or as a literal enclosed in quotation marks (e.g. `“Technical University of Denmark”`). For example:

```
<http://www.dtu.dk>      dc:Title      “Technical University of Denmark” .
```

Serialization formats are mostly used for the exchange of RDF data between different systems. Internally, triples are usually stored in databases called *triple stores*. Some of the popular triple stores available today include Jena<sup>4</sup>, Virtuoso<sup>5</sup> and Sesame (Aduna).

Data in a triple store can be queried in a similar way as one executes queries on a relational database. W3C has created a query language, called SPARQL (World Wide Web Consortium, 2008), to standardize the format of the queries, and a protocol for issuing SPARQL queries against triple stores and receiving results. A sample query for getting a list of books and their titles in this language looks as follows:

```
SELECT ?book, ?title
WHERE
{
  ?book <http://purl.org/dc/elements/1.1/title> ?title .
}
```

Having a standardized query language means that it is possible to choose any triple store and it will be compatible with our application. But it also means that public databases which expose their data in RDF can be queried in a unified manner.

Most triple stores also provide some support for reasoning and inferencing based on the logical rules defined in an ontology (such as RDF Schema or OWL) and the accumulated data. In short, a triple store can automatically create new statements based on existing ones. For example:

- Inferring class hierarchies: If *Toyota* is a *rdfs:subClassOf* *Car* and *Car* is a *rdfs:subClassOf* *Vehicle*, then *Toyota* is also a *rdfs:subClassOf* of *Vehicle*. This transitivity rule is generalized in OWL so that it can apply to any property through the use of the *owl:TransitiveProperty* class. Thus just by specifying that a property is transitive, the inference engine can deduce new relationships.
- Inferring class membership: If **A** has property **P** and property **P**'s domain is class **C**, then **A** is an instance of **C**.

Inferencing can be an extremely valuable feature for personal information management. In this domain there are a lot of simple rules that apply to relationships between people. For example:

- Friendship – a symmetric property – if John is a friend of Peter, then Peter is a friend of John.
- Child/parent – inverse properties – if Mary is a parent of Peter, then Peter is a child of Mary.
- Ancestor – a transitive property – if Mark is an ancestor of Mary, and Mary is an ancestor of Peter, then Mark is an ancestor of Peter.

Similar relationships can apply to other objects. Consider the properties of a Picture:

<sup>4</sup> <http://jena.sourceforge.net/>

<sup>5</sup> <http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSTriple>



- Depiction – If a picture depicts an object (e.g. a person), then, obviously, that object is depicted in the picture. Having such an inverse property makes it easy to access all pictures which are related to a particular object.
- Location – The location where a picture is taken can be represented by an object naming the geographical location, such as Copenhagen or Odense. A named geographical location can be a part of a larger one, such as Denmark or Europe, and can be comprised of a set of smaller ones. By defining the relationships between locations as a transitive property (in a way similar to the Ancestor property above), searching for pictures taken in Denmark would return those taken in Copenhagen and Odense, too.

### Availability of RDF data

RDF is a language for stating facts. All statements expressed in RDF are made with a reference to a particular vocabulary of terms. Thus saying that some data is available in RDF means that there both exists a vocabulary that specifies the meaning of the terms used to describe the data, and that the data has been represented in the form of statements using that vocabulary.

This has an important consequence: an application that has no prior knowledge of the domain associated with the data, can, to a certain extent, manage and process the data in a way that is useful for the user.

Let's consider metadata about a picture from a digital camera expressed in RDF. The metadata will have to be based on some standardized ontology – probably EXIF<sup>6</sup>. By combining the ontology and the metadata, an application with no knowledge about digital photos can display to its user a human-readable description of the photo, containing possibly such information as camera type, aperture settings, lighting conditions, etc. Moreover, by utilizing the relationships between the photo ontology and some ontology that the application is designed to support, it can infer that the pictures can be handled in some more complex way. For example, geographical coordinates embedded in the pictures' metadata could be used to display them on a map.

Existing data can be converted to RDF through the use of “RDFizers” or RDF extractors. These programs extract data from some compatible source and express it in the form of triples in such a way as to closely follow the semantics specified in the ontology. RDF extractors can be found for many file formats and websites, including JPEG pictures, Microsoft Office documents, Flickr, Facebook, etc.

There is also a substantial amount of data that is already in RDF form available on the Web. One of the larger sources includes DBpedia, which is *a community effort to extract structured information from Wikipedia and to make this information available on the Web*. It contains descriptions of over 2.9 million things (DBped09).

#### 2.4.6. Summary

The choice of the appropriate data model for storing personal information is largely dependent on the flexibility of the model. Personal information encompasses many types of data and forms complex relationships. From the above comparison it is clear that the relational and object

---

<sup>6</sup> <http://www.exif.org/>

models are too rigid for this purpose: the model has to adapt to the user's needs, and this is not possible when the schema has to be defined up-front and hardcoded in the application.

The choice between the associative model and RDF is also straightforward and in favor of the latter, but this is due to the large ecosystem of technologies and software that accompany the core RDF specification, and not drawbacks of the associative model itself. In fact, both models are associative and they represent data in similar ways. The associative model has even certain advantages as it is more expressive: in RDF it is not possible to directly use a statement as a subject of another statement, and a workaround in the form of statement reification is needed.

The most important advantages of RDF are:

- **serialization** – it is easy to exchange RDF data between different systems due to many standardized serialization formats (XML, Turtle, etc.); this is an important factor in the context of obtaining RDF data from external sources and synchronizing user's personal information between different computers,
- **ontologies** – the ability to define a common vocabulary of terms and to specify logical relationships within a knowledge domain;
- **availability of data and tools** – RDF extractors make it possible to access data stored within files, programs and websites; triple stores have support for automated reasoning.

The following table summarizes the differences between the aforementioned models:

	Files and directories	Relational model	Object model	Associative model	RDF
Unstructured data	yes	yes	yes	yes	yes
Structured data	no	yes	yes	yes	yes
Topology	hierarchical	any	any	any	any
Adding extra information	no	no	no	yes	yes
Schema changes <sup>7</sup>	n/a	difficult	difficult	easy	easy
Reification	no	no	no	yes	yes <sup>8</sup>
Querying	no	yes	no	yes	yes
Tool availability and popularity	good	good	good	bad	average

Table 2: Summary of differences between data models

- *Unstructured data* – Does the model support unstructured data as a first-class citizen?
- *Structured data* – Does the model support structured data as a first-class citizen?
- *Topology* – List of supported data topologies (see Section 2.1.2)
- *Adding extra information* – Does the model support adding additional structured information that was not foreseen when the data model was created? (The relational and object models would require a schema change for this to be possible, while in the Associative model and RDF it is a matter of creating another association.)

<sup>7</sup> See description below the table for an explanation of the terms “difficult” and “easy”.

<sup>8</sup> Reification in RDF is only supported indirectly by creating an additional set of statements that describe the original one.

- *Schema changes* – The level of complexity for introducing changes to the data model of a running system. *Difficult* indicates that the change might require a programmer/developer to implement, while *easy* means that it could be realized by a user.
- *Reification* – Does the model have the ability to treat existing data and metadata as first-class citizens, i.e. entities which can be annotated with more data?
- *Querying* – Does the model and its implementations provide support for executing structured queries against the data?
- *Tool availability and popularity* – The number, quality and maturity of tools that support this data model. *Good* – support for this model is included with every major software development tool; *Average* – at least 5 different vendors or groups have released tools supporting this model; *Bad* – there is less than 5 independent implementations of tools supporting this model.

## 2.5. Summary

An analysis of the problems involved in managing distributed information was performed. Personal information can mean different things to different people, but the owner of that information has at least partial memory of what it is. Other characteristics of this type of information have also been discussed, such as the metadata that can be associated with it. It has been shown that commercial approaches to personal information management lack the capability to unify and integrate distributed information. An examination of how information is distributed between different sources has been performed and ideas for unifying it have been described: combining information representing similar concepts, using links to connect information objects in a similar way as the mind associates thoughts, and using a single namespace to access information located in different places. An overview of the suitability of different data models for keeping personal information was given. The RDF model was chosen as the one that has the most advantages for this purpose – the flexibility of this model being an important factor.

### 3. Use cases

This section presents a few use cases that investigate how personal information management can be improved by using different techniques. A hypothetical PIM application is discussed that can interact with information stored in different places. Certain features of this application are presented that try to solve specific PIM problems. Afterwards, a short summary of the features of a system that would allow for the realization of these scenarios is given.

#### 3.1. Working on a project for a customer

This scenario explores the possibility of using a PIM application as a standalone desktop application that integrates with Gmail.

##### *Problem*

John receives emails from a customer regarding a web project he is doing for them. The project is in its finalization phase and the emails they send him concern various problems they found with the website, things they want to have changed, rechecked, improved, etc. John gets a lot of these kind of emails and it is difficult for him to memorize all the problems that were reported and all the fixes and changes he has implemented. The emails are often sent by different persons in the customer's organization and may concern similar or different issues. One email usually lists several issues that need to be considered.

John is using Gmail for managing his email and he is tagging the messages with the customer's name to keep track of them. When a new email arrives, this is what John does to handle it:

1. He reads the email and identifies the issues that are discussed.
2. He checks if each issue is a new one or if this is a problem he has already worked on.  
This step usually requires going through many received emails (as it is often the case that the same or different person has already raised this problem), checking John's replies to those emails (it is possible that this problem has already been solved) and checking his notes on this problem (they can be located in different places as it is not possible to attach notes to emails directly).
3. He investigates the issue and solves the problem **OR** creates a to-do item in his calendar program that this problem needs to be solved later. He also tags the message with a to-do tag.
4. Finally, he sends a reply stating that the problem has been solved or that it will be solved later.

##### *Analysis*

Usually, if the problem cannot be solved immediately, managing the information associated with the problem becomes difficult. This is because:

- One has to remember that a certain problem exists and that it needs to be solved.  
This requires creating a reminder in some calendar software.
- One has to keep the customer's email that initially raised the problem.  
This email contains the description of the problem.
- One has to make notes related to the analysis of the problem.  
Without the notes, coming back to this problem later might require reanalyzing it.

Furthermore, the notes need to be stored somewhere – usually in an application that is external to the email client.

- One has to contact other people to discuss this problem.  
This discussion produces more emails or other forms of communication (message log in an Instant Messaging application, notes from a phone conversation, etc.)
- One might produce some files associated with the problem.  
They should be listed in the problem notes for later reference.

John's problem resembles a software bug fixing process for which specialized issue tracking software (such as Trac (Trac09), Bugzilla (Bug09), etc.) exists. However, this problem itself is quite general and can occur in many situations.

### *Solution*

John's workflow could be improved by integrating information from different applications in a single place. This could be accomplished in the following way:

When a new e-mail arrives, John opens the PIM application and finds the corresponding email by viewing the list of all emails sorted in descending chronological order. He opens the email and activates the *issues* view in split-screen mode – the email contents is displayed on the left, while the list of issues on the right. John can now mark text in the email, right click on it and select *Create Issue*. The application will ask to type a short title. A new *Issue* object is created and is displayed in the right part of the screen. The text in the email functions as a hyperlink to the *Issue* object and vice versa. The issues are displayed as bulleted list. They can be reordered and nested, allowing for one issue to be composed of several smaller ones. John can also quickly add some initial notes regarding each issue. He can also add tags to each issue that will help in finding it later on.

After having marked all issues in the email, John needs to find out if the new issues are related to any existing ones. Normally, this would require him to go through all emails from the customer. With the PIM application, he can find similar issues based on matching tags. Before he can do this, he needs to link the email to the website project by finding the project object in the PIM application and dragging the email onto it - all the issues are automatically linked to the project. If John finds a similar issue to the one reported in the email, he can merge the two issues together. The contents of both issues is combined and they are treated as one.

When going through the list of issues from the email, John can decide whether he wants to solve them now or later. For the postponed issues, he can quickly create a reminder by right clicking and selecting *Add To-Do*. A list of all things marked as *To-Do* is available in a different view of the application. In this way John can be certain that he will not forget about the customer's request.

While John is working to fix a particular problem, he can link any associated resources just by dragging them onto the issue. This will let him keep all the important files and documents in one place for later reference.

## Website problems continued

- Email from [Paul Black](#) received on 22 apr 2009.

- **Images are not transferred to the website**

- **Description:** We've noticed that some of the announcements that are published on our website do not have any images, even though in our system the images are clearly there. ...

- **Status:** **postponed**

- To-Dos: [1](#) | Files: [5](#) | Tags: [image](#), [transfer](#)

- **Navigation does not work**

- **Description:**

```
when I click on 'About us' it is not possible to go back. What's
interesting, this only happens from time to time. I haven't been able
to pinpoint the pattern yet.
```

- **Status:** **fixed**

- Notes: [1](#) | Files: [2](#) | Tags: [navigation](#), [back](#)

Figure 6: Conceptual illustration of the UI for the “Working on a project...” scenario

The user interface mockup that could help the user accomplish the above tasks is presented in Figure 6.

### Remarks

The presented solution involves taking a simple “flat-text” email and annotating it with metadata to create higher-order information that is useful to the user. The contents of the email acquires meaning and becomes linked to other objects in the system – projects, existing issues, notes and to-do items. This makes it possible to view that contents in different ways, one of which is the *issues* view.

It should be noted that the application does not try to automatically extract *Issue* objects from email. Only the user is capable of understanding and deciding what parts of the email constitute an Issue. It is then important to make sure that the process of extracting issues is quick and efficient, otherwise the user could find it too cumbersome to use.

## 3.2. Semantic notebook

This scenario explores the idea of a desktop PIM application that lets the user combine information from different sources in a visual way, but at the same time retaining its meaning. It has been inspired by the user interface of Microsoft OneNote and attempts to use it as a base for new features – ones that would combine the flexibility of the visual organization of information with the benefits of keeping structured information.

### Story

Michael is talking to a friend using Facebook's built-in chat program. The friend invites Michael to a party:

*"I'm organizing a BBQ at my place. My address is: Lillevej 12, 1 th, Lyngby.  
We are starting at around 7:00 PM this Saturday. You can bring some beer if  
you want."*

Michael starts the PIM notebook and pastes this text onto a blank page. He uses a sidebar to find the *Person* object for his friend and drags it onto the page – a small image of his friend appears. Michael wants to save the address given in the conversation as his friend's home address for later reference – he selects it with the mouse and drags and drops it over the friend's image. A dialog box appears asking for the type of link between those two objects and Michael selects *Address*. The address is now stored in the system and will appear in the *Person* view for his friend.

Next, Michael right clicks on an empty region of the page – a popup menu appears letting him choose *Add Social Event*. By doing this, some additional fields appear on the page, including *Date*, *Time*, *Location*. As Michael fills out the date and time fields, a link to this page appears in his calendar view. He will now be reminded that such an event is taking place whenever he opens the PIM application.

But Michael also wants to remember that he needs to buy some beer for the party. He selects the "You can bring some beer" text and right-clicks on it, choosing *Add To-Do* from the popup menu. A dialog box appears asking him to provide the *Due date* for this task. Michael accepts and a new to-do task is created with reference to the selected text.

Later on during the conversation, his friend tells him about some other people whom Michael knows and which are coming to the party. Once again Michael finds the associated *Person* objects and drags them to the page. However, he wants to distinguish the person organizing the party from the attendees. He right clicks on the page and selects *Add field* from the popup menu. A dialog box appears asking for the type of field to be added to the page, and Michael chooses *Group*. He types in "Attendees" as the field's name and drags and drops the person images onto the field's value box.

Furthermore, Michael might decide to associate some tags with the notebook page he has created. Tags will aid him in quickly locating similar information in the future.

The result of Michael's work could look similar to the mockup in Figure 7.

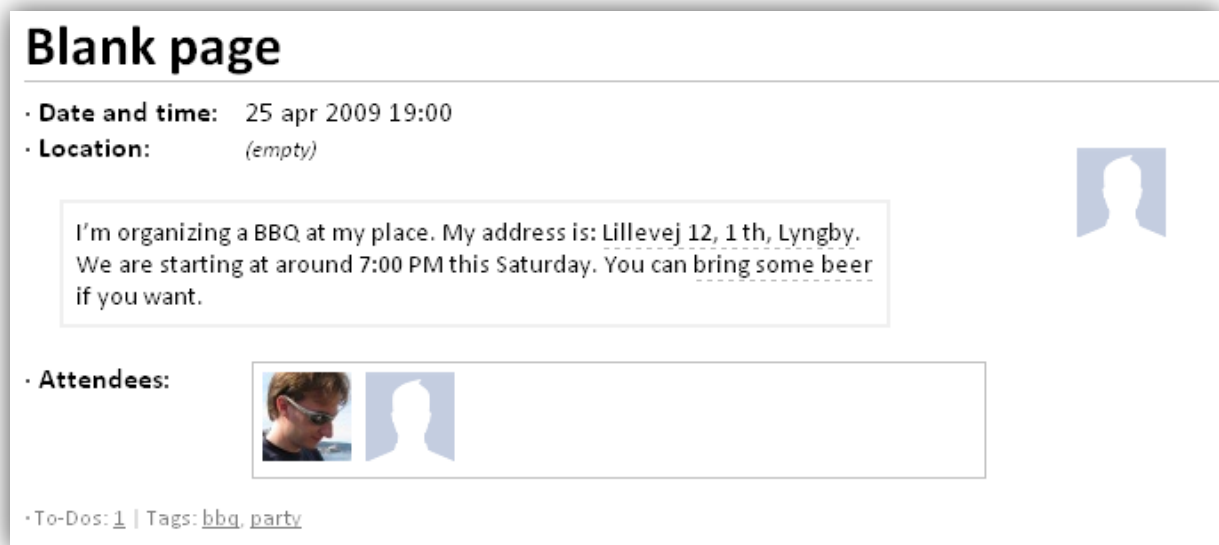


Figure 7: Conceptual illustration of UI for the “Semantic notebook” scenario

After the event has already taken place, Michael might decide to post the pictures that he and his friends have taken during the party. Michael has his pictures still on his local disk, while his friends have uploaded theirs to Picasa and Facebook. Michael would like to combine all of them into a single album. To do this, he right-clicks on one of his friend’s icons from the Attendees list and selects Explore. He sets the filter to list only objects of type Image gallery and in this way finds all albums related to his friend – including the ones published on Facebook and Picasa. He sorts the albums in descending chronological order and quickly finds the one he is interested in. The external albums and images accessible through the PIM application contain all the metadata that Facebook or Picasa stores, such as titles, tags and information about the people depicted in the pictures.

Michael can create a new subpage of the page containing the party invitation. He chooses an Image gallery view for this new page, which contains fields that are commonly used for such a task. He can now drag images from his friend’s album and from his local folder into the newly created album. He can also group them together, associate them with other objects (such as persons that are depicted in those pictures), tag and annotate them. Finally, when his album is ready he can publish it on a web page for others to see.

### Remarks

Organizing personal information as it is being collected, in an “on-the-fly” manner, can be difficult using typical approaches that are found in popular software, such as Microsoft Outlook. This is because such software requires the user to classify the information up-front in some way – either as a contact, calendar event, to-do item or a text-only note. However normally personal information is more complex than that – it contains a combination of different kinds of concepts.

The approach described in the above story lets the user quickly collect all information regarding a particular subject in a single place and then supplement it with semantic metadata. The user has effectively created a note, that holds all the important information about an event, and with the individual parts of the note being integrated into the system – the event appears in the calendar,



the user will be reminded about the to-do items, and perhaps the note can also be accessed while viewing information about the people involved in the event.

By linking new information with existing one, the user can tie loose facts (such as the home address of a person in the message) with the structured information that is already in the system (the corresponding *Person* object) and thus extending what they know. However, linking information in this way also serves the purpose of categorizing information, which can help in finding this information later from related objects. In order to aid less structured searches, simple text tags can be attached to objects.

### 3.3. Inviting people

Sue is inviting her friends and colleagues from work to a party. She needs to either send out invitations via email, or call people if she doesn't have their email address.

The process of finding contact information is actually quite problematic for her. She is inviting old friends from school and some of the e-mail addresses she has in her e-mail client might already be out of date. She often has more than one email for a person and it is hard to decide which one might be the current one. Some of the friends have Facebook accounts and this can serve as a good source for up-to-date contact information. Colleagues from work are a similar problem. As it is holiday season, not all people will check their company e-mail. It might be necessary to try their private e-mails or just call them on the phone.

Sue also faces a problem in keeping track of her efforts. She wants to know who was already contacted, did they respond, will they come, will they bring something, etc.

Fortunately, Sue uses the PIM application to manage her information. She creates a new note and adds the following fields to it:

- Pending – for a list of people that she intends to invite, but who have not yet been contacted,
- Invited – for a list of people that she e-mailed, but who have not yet responded,
- Will come – for a list of people that have accepted the invitation.

She can now add people to the “Pending” list by dragging them from the list of all the people she knows. By clicking the person in the “Pending” list, she can quickly view the details about that person – including their email addresses and phone numbers. This information was either imported from her e-mail client, from Facebook or was typed in manually by Sue. By right clicking on an email or phone number she can see its origin. This helps her in choosing the correct one to use. From the details view of a person she can also quickly go to a list of all email messages sent by the email addresses associated with that person. In this way, she can see which email was used most recently.

Sue sends out invitations and moves people from “Pending” to the “Invited list”. When someone confirms that he or she is coming to the party or not, Sue can move them to the “Will come” list or remove them, respectively. If no reply has been received from a person, Sue can try one of the other email addresses listed in their profile.

For tracking other information, Sue can add more fields to the note. For example:

- Will bring something – for a list of people that offered to bring some food or drinks for the party,
- Will be late.

Sue can add people to these groups and annotate those associations with a note giving the details (e.g. the type of food they will bring, when they will arrive, etc.) By annotating the association (i.e. the relationship between the note about the party, the “Will bring something” property and the person), the additional information will only be visible in the context of the note about the party. If the person itself would be annotated, then the annotation would be visible for all uses of that Person object - which does not make sense in this case, as the additional note applies only in the context of the party.

The application also allows Sue to view the information gathered in the party note in a different way. For example, she can right click on the “Will bring something” list and choose to open it in a separate window with a different view. A table view can be used, which, after selecting the proper fields to display, will show the name of the person and the contents of the associated annotation. In this way, Sue will have a summary of what each person will bring to the party.

### 3.4. Summary

A list of features of a personal information management system capable of realizing the scenarios described in the previous section is given below.

#### *Type system*

Information should be organized into types and objects. Types will represent concepts as described in previous sections. Objects will represent real-world entities or things that the user considers to have their own, distinct identity. For example, information about a particular email or a particular person will be represented as an object.

The actual data will be stored in objects in the form of values associated with properties. Each property might have multiple values. In this way, a property such as “name” could contain values for the multiple names that a person might have.

The properties that each type may have are intended to serve as suggestions and hints. The user is neither forced to supply values for all these properties, nor is he or she restricted from adding new properties that may be needed.

#### *Multiple ways to view information*

Most applications provide only a limited amount of choices for the user when providing ways in which information that they contain can be viewed. The PIM application should offer the ability to view the same information in different ways. For example, a view could be designed to display only the most significant information about the object, while filtering out the rest. Moreover, it should be possible to use different views for browsing collections of objects. For example, sometimes it is useful to view a list of emails in a way that it is normally presented in an email client, but it might also be useful to view emails as a set of thumbnails grouped by the project they belong to.

### *Annotating objects*

Annotating an object is a process when one creates a new object, such as a note or a to-do item, which references some existing object. Example use cases include receiving an e-mail message and creating a note about it. This note, or some fragment of it, could be, in turn, annotated with a to-do item that would appear in a list of things that need to be done.

A special case of annotating objects is tagging. When an object is annotated with a tag, either a new tag is created or an existing tag is found. The name of a tag is its only identity.

When a user annotates an object, the two objects become linked. This means that it should be possible to access all the annotations of an object from the object itself, and vice versa.

### *Linking and grouping objects*

A link denotes that two information objects are related in some way. It should be possible to create named links, which describe the type of relation (e.g. one person knows another, this picture depicts a given location), and simple links, which indicate that some relationship exists without specifying it.

Grouping objects is intended for keeping objects that are somehow related in a single place. Grouping objects is realized through linking. Multiple objects (or collections) can be linked to a single other object, forming a collection of objects. If a named link is used, the group also carries some meaning (e.g. a group of people that a given person knows).

### *Object equivalence*

Object equivalence lets the user state that two information objects represent the same real world object.

Treating two distinct objects as one has two principal use cases.

First, it lets the user combine objects from different sources that represent the same entity and concept. Due to the distributed nature of personal information, data about a particular thing is stored in many different places. For example, information about a single person could be stored on Facebook and in the address book of an e-mail client. By combining these two objects, the user would obtain a union of the information from Facebook and the e-mail client which would be treated as a single entity.

Secondly, equivalence can be used for merging objects that contain different aspects of an entity. For example, information about a holiday trip could be stored as a calendar entry in Microsoft Outlook (date and duration of the trip), as a photo album on Picasa (pictures from the trip), as a set of notes in Microsoft OneNote (a list of things to pack), and so on. Combining that information into a single object leaves the user with a single place that can be accessed to find all the information.

When two objects are merged, it is important for properties of these objects that carry the same information to be merged as well. For example, a Facebook profile and an Address Book contact will both contain a name and an e-mail address of a person, although the properties associated with those values might be called differently on Facebook and in the Address Book. By defining

the corresponding properties as similar, the merged object will have only one property stating the name and one showing the e-mail address instead of two.

### *Accessing distributed information*

The user should be able to enter new information into the PIM system and combine it with existing information that is stored in external sources. Both types of information should be treated by the system in the same way – there shouldn't be any difference for the user related to the type of information they are working with.

### *Extracting unstructured information*

Some information exists in an unstructured form, as described in Section 2.1.2. Plain text, for example, might contain references to existing objects. The user might want to mark that a certain text fragment refers to an existing object (e.g. a person), so that a link between the text and an object could be established, which would aid in later retrieval.

Text might also contain fragments that can be treated as property values for objects. As described in the Semantic notebook use case, a text fragment can be extracted and assigned as an address of a person. Similarly, a link between the original text and the person object will be created.

Finally, the user might want to annotate certain text fragments with notes or to-do items.

A similar approach can be used for pictures, where different regions of a picture can reference different objects (such as people, places, etc.)

### *Sharing information*

Sharing personal information is an important part of the interaction between humans. We like to exchange experiences, discuss our interests, and comment on the activities of others. In fact, these kinds of activities form the basis of most social networks, including Facebook and MySpace. Because sharing personal information is such a popular activity, social networks *have attracted millions of users, many of whom have integrated these sites into their daily practices* (Boyd, et al., 2007).

The PIM system is designed primarily as a central place where the user can integrate all of their personal information. However, it can also be an important tool for publishing this information for others to see. The user can, for example, combine pictures from different sources into a Photo album, and then publish it on Facebook. This kind of task is simplified by the system due to its mechanisms for accessing and combining distributed information.

One of the problems involved in sharing information is defining the scope of the data that should be published. In a system where all information is stored in the form of associations, this can be particularly difficult. In the Photo album example above, the album is linked to pictures, the pictures can be linked to places and persons, which in turn link to other pictures, notes, other persons and so on. If the application were to follow and publish all links, it might end up publishing the user's whole repository.

Another problem involves privacy considerations. Even if a delimited set of information to be published would be defined, it might be the case that it should not be available for viewing to just

anyone. Studies indicate (Boyd, et al., 2007) that even though most social networks provide some form of privacy controls, allowing the user to specify who gets to see what, it is not enough to effectively protect the privacy of users.

### *Intuitive user interface*

Interacting with the PIM system should be natural for the user. Therefore the design of the user interface should follow these principles:

- Right-clicking on an element displayed on screen should invoke a context menu with actions relevant for that element. For example, right clicking on an object that can be viewed in different ways should present a list of views that can be used in conjunction with the object.
- Linking objects should be possible by just dragging one object over another.
- Adding objects to a collection should be possible by dropping it on a collection.
- Adding objects from external sources should be possible by dragging the object from a third-party application window to the PIM system window.

## 4. Design

This section presents the design of a system for managing distributed personal information. It starts by presenting the data model used by this system – what types of entities are visible to the user (the ontology), how are they realized in the application (the application model) and how are they persisted in a database (the physical model). Then, the components of the user interface are described – with views being the core elements in this area. Next, an explanation of how the system enables a user to find, manage and access distributed information is given. The chapter ends with a description of how multiple devices can be used to access user’s personal information and how this information can be shared with others.

### 4.1. Data model

The PIM system’s data model is divided into three layers. The physical model describes how data is organized for persistent storage using a database management system. The application data model, unlike a typical logical data model, does not correspond to the domain model for the application’s domain, which is personal information management. This domain-specific information is captured by the ontological model, which is constructed on top of the application model.

#### 4.1.1. Application

The application data model follows from earlier considerations about concepts and types. Despite being designed for storing personal information, it is still quite a generic model and it is not tied to this particular domain.

#### *Objects and values*

All user information is organized into Objects. An Object is a set of values associated through Properties and organized into Collections. It represents a distinct entity, such as a particular person. For example, an Object representing a person called John Timothy Smith might contain a Property *given name* associated with a Collection holding the values “John” and “Timothy”.

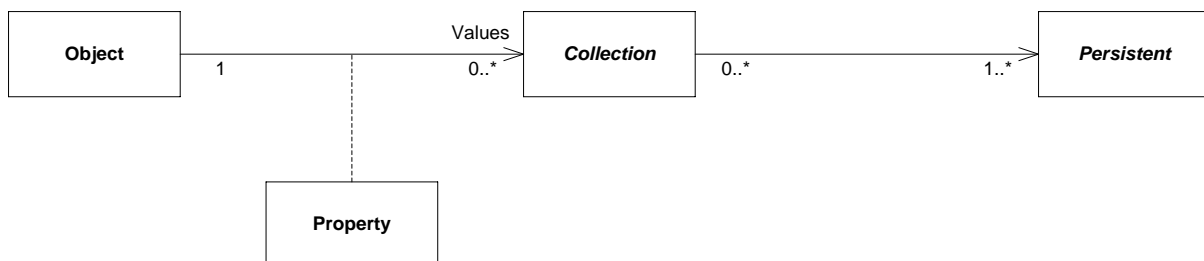


Figure 8: Objects and values

Values in a Collection can be stored either as an unordered set (a *Bag*), or an ordered list (a *List*). A Bag is used when the order of the values has no relevance or makes no sense – for example, a set of people that a given person knows. A List can be used for storing elements whose order is important, such as a list of names of a person. The implementation in this project will be limited to just the Bag collection.

A Collection can hold primitive values, references to other Objects or to other Collections. All these entities are subclasses of the Persistent class:

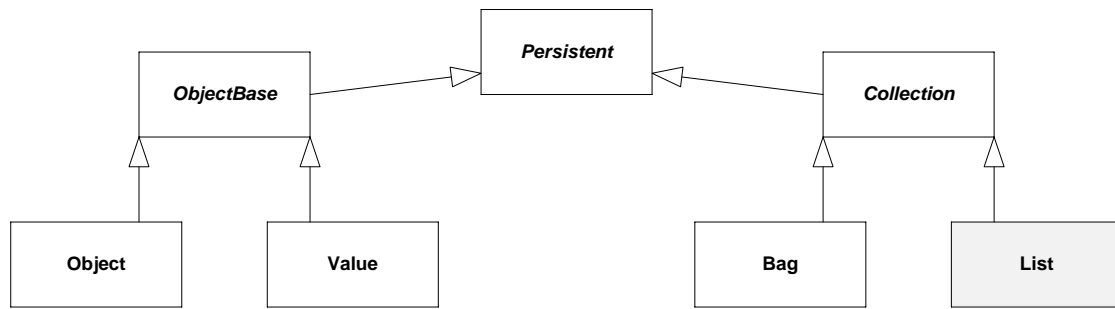


Figure 9: Persistent objects hierarchy

While an Object represents a complex container, simple values are stored through Value classes. Each Value contains a Datatype field informing about the type of value stored, such as a text, integer, date, etc.

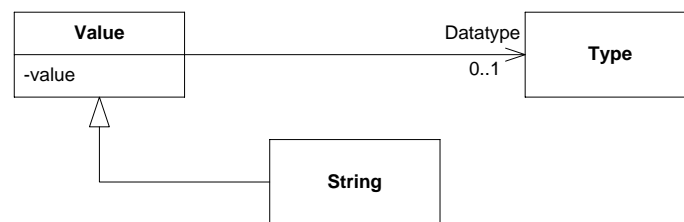


Figure 10: Classes for storing primitive values

In order to support internationalization, text strings are stored through a specialized subclass called String, which holds a version of a string in multiple languages. Basically, a String is a mapping between an ISO language identifier (such as “en” or “da”) and a text string.

### Type and property hierarchy

Type and Property classes are separate from the hierarchy of Persistent objects (see Figure 9) and are used to define the domain model for the PIM system.

A Property is a name assigned to describe a particular type of association between an Object and a value (a value can be any Persistent object, such as another Object, or a simple Value), and a suggested Type for that value (the Range of the Property). For example, a Property called *birthday* could suggest the Type *date* (its Range) for all new values assigned using it. Moreover, a Property can have two other characteristics:

- A set of inverse Properties. If Property A is an inverse of property B, then whenever object  $O_2$  is assigned to object  $O_1$  as a value for A, then the object  $O_1$  is assigned to object  $O_2$  as a value for B.
- A symmetric property bit. If Property A is marked as symmetric, then whenever object  $O_2$  is assigned to object  $O_1$  as a value for A, then the object  $O_1$  is also assigned to object  $O_2$  as a value for A.

A Type is an unordered set of Properties. It is important to note that the Properties of a Type serve only as guidelines for the possible associations that an Object belonging to that Type might have – it is neither required that any of these Properties have assigned values, nor the Object is restricted to having only those Properties that are contained in the Types that it belongs to.

Figure 11 presents these relationships in the form of an UML class diagram.

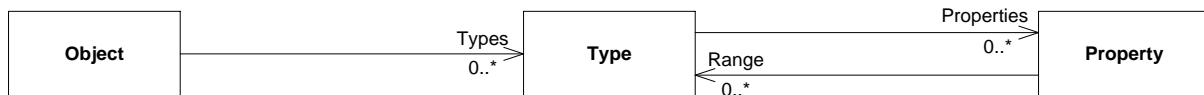


Figure 11: Objects, properties and types

Apart from the above characteristics, Properties and Types are similar to normal Objects. In fact, they could be modeled as subclasses of the Object class. However, in most object-oriented programming languages, including the one used for implementation in this project, it is not possible to dynamically modify an existing object to make it an instance of a specified class. Furthermore, a single object cannot extend two different classes, while the underlying physical model allows for an Object to be a Type and a Property at the same time. Instead, Properties and Types have been modeled as classes that depend on the existence of an “owner” Object.

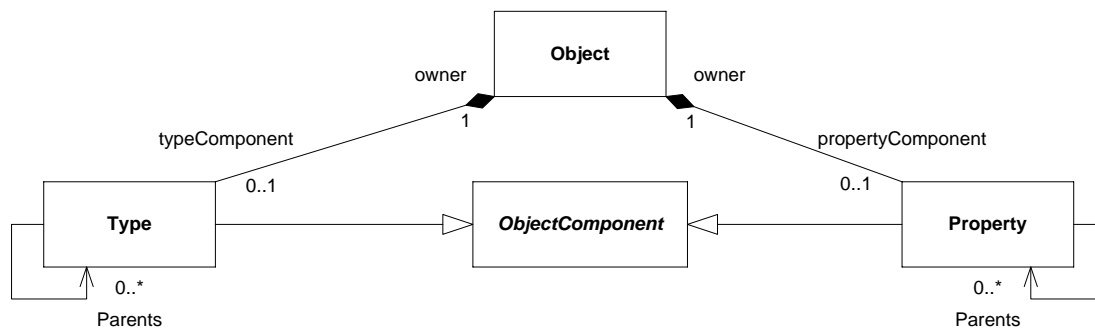


Figure 12: Type and property hierarchy

Both Types and Properties can form a hierarchy by having parent Types and Properties, respectively. In general, a Type or Property inherits certain characteristics from its parent. For a Type, the effective set of Properties is a union of its own Properties and those associated with its parents. A Property, on the other hand, inherits the Range – a set of suggested Types for a new value.

### Equivalence and similarity

Equivalence and similarity are used for indicating that certain objects are closely related to each other.

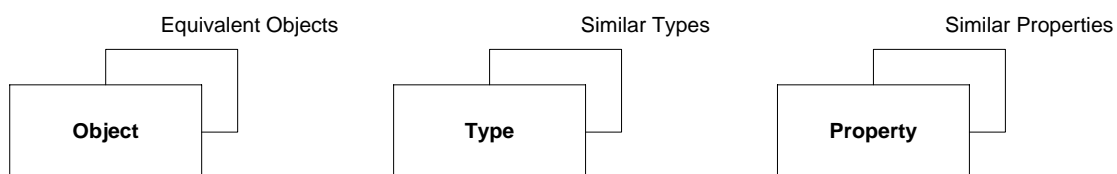


Figure 13: Equivalence and similarity for objects, types and properties

Equivalence states that two or more Objects are actually the same Object – the URIs assigned to them can be treated as synonyms. When two objects are marked as being equivalent, accessing



any of them will result in the same values. This is accomplished by merging these objects together. Merging Objects A and B involves:

- Replacing the set of Types of A and B with a union of sets of Types of A and B.
- Replacing the mapping between Properties and Collections for A and B with a new mapping that contains the union of Properties of both A and B mapped to:
  - The original Collection, if the Property was present only in A or B.
  - A Bag containing a union of values from Collections in A and B, if both A and B had values for that Property and both Collections were of type Bag. Note that a union of sets implies that duplicate values are eliminated.
  - A List containing values from Collection in A appended to the end of the Collection in B, if both A and B had values for that Property and both Collections were of type List.
  - A List containing values from Collection in A appended in random order to the end of the Collection in B, if both A and B had values for that Property and the Collection in A was of type Bag, while the Collection in B was of type List.

The individual objects A and B retain their original values in the underlying RDF model. When a new value is added to a merged object, the URI that will be used for storing this value is selected at random. This means that it is possible to separate objects that have been merged (i.e. un-do the merge), but any values added since the objects have been merged might be distributed between all the original objects.

Similarity, on the other hand, hints about a close tie between objects but the objects themselves maintain their separate existence. It can only be established for Types and Properties.

When two Types A and B are similar, the application will:

- Offer the same presentation logic for A as for B.

When two Properties A and B are similar, the application will:

- Treat parents of A as also being parents of B.
- Treat values assigned via property A as also assigned via property B.

Consider two types: *Contact* and *AddressbookEntry*. They might be associated with data imported from two different applications, e.g. a calendar and an e-mail client. They conceptually represent the same kind of information, therefore the PIM system should treat them in a similar way. These types will also use different properties for describing similar information. Defining those properties as similar will let the system know which pieces of data can be grouped together. Moreover, duplicate data for the same property can be removed.

#### 4.1.2. Physical

The application data model is physically stored in an RDF data store. As all data in RDF is stored in the form of triples (see Section 2.4.5), a conversion between the application model's objects and RDF statements is necessary.

Storing objects in an RDF store requires that all such objects – Objects, Collections, Values, Types and Properties – be uniquely identifiable. Because Types and Properties are tied to an instance of an Object, effectively only Object, Collections and Values (i.e. Persistent objects) have a need for a unique identifier – a Uniform Resource Identifier. Objects originating from external sources may already have an identifier assigned, as it was created when the object was imported (for example, an URL will be used for images imported from the Web). Other objects will be assigned an URI by the system when it is first needed. The URI is generated from a user-defined string combined with a number from a number sequence.

The descriptions below specify a bidirectional mapping between RDF and the application model. This means that it is both possible to serialize application objects to RDF and to deserialize RDF into application objects based on the specification below.

### Objects

Serialization of an Object to the RDF format is done by creating statements corresponding to all the Properties and values associated with that Object. As a Property is associated with values through a Collection, this process depends on the type of Collection. Bags are normally serialized in association with an Object – they are “attached” to the Object. For each element in a Bag, a statement corresponding to the following pattern is created:

Statement pattern	Example
<b>O</b> <b>P</b> <b>V</b> .	<i>ex:John</i> <i>foaf:givenname</i> “John” . <i>ex:John</i> <i>foaf:givenname</i> “Thomas” .

Note: **O** represents the URI of the Object, **P** represents the URI of the Property and **V** represents the value. As a value of a property could be another Persistent object, a detailed description is included in the next section.

When a Bag is serialized in the above way (“attached”), it’s existence is tied to the one of the Object – when the Object is deleted, so is the Bag. Moreover, the Bag cannot be referenced directly, as it has no associated URI.

Bags can also be serialized as separate objects, which is always the case for List collections. When a Collection is serialized in this way, the value V in the pattern above is the URI of the Collection. This, however, introduces an ambiguity for deserialization: an attached Bag holding a List results in the same statement as a Property associated directly with a List. In order to solve this problem, the following rule is used: If an Object’s property is associated with only one value and that value is a List, then the property is considered to be associated with a List.

The process of serializing elements of a Collection is described in the next section.

An Object can also be associated with a set of Types that the Object is an instance of. This is expressed in RDF using the *rdf:type* property, which is used to state that a resource is an instance of a class (World Wide Web Consortium, 2004/s). For each Type (with an URI **T**), the following RDF statement is constructed:

Statement pattern	Example
<b>O</b> <i>rdf:type</i> <b>T</b> .	<i>ex:John</i> <i>rdf:type</i> <i>foaf:Person</i> .

Furthermore, an Object can be marked as being equivalent to another Object. This fact is expressed using the *owl:sameAs* property, which states that the things identified by two separate URIs are actually the same thing – they have the same real world identity. In essence, this construct expresses that two URIs are synonyms (Bechhofer, et al., 2004):

Statement pattern	Example
$O_1$ owl:sameAs $O_2$ .	ex:John owl:sameAs ex:Johnny .

### Collections

The previous section described how Collections are serialized in connection with an Object. In this section a description of persisting the elements of the Collection is given.

Serializing a Collection involves creating a statement that describes the type of the Collection and then a statement for each element. A Bag uses the following pattern:

Statement pattern	Example
$C$ rdf:type rdf:Bag.	ex:ld#1 rdf:type rdf:Bag .
$C$ rdf:li $V$ .	ex:ld#1 rdf:li "John" .
	ex:ld#1 rdf:li "Thomas" .

(Note: **C** represents the URI of the Collection. *rdf:li* is a property that enumerates elements in a container, and *rdf:Bag* is the class of unordered containers (World Wide Web Consortium, 2004/s).)

A List uses the pattern below:

Statement pattern	Example
$C$ rdf:type rdf:Seq.	ex:ld#1 rdf:type rdf:Seq .
$C$ rdf:_nn $V$ .	ex:ld#1 rdf:_1 "John" .
	ex:ld#1 rdf:_2 "Thomas" .

(Note: *rdf:\_nn* is a pattern for list properties that enumerate elements in a container, and *rdf:Seq* is the class of RDF 'Sequence' containers (World Wide Web Consortium, 2004/s).)

The values of a Collection (represented by **V** in the above patterns) are stored depending on their type:

- For Objects and Collections, **V** represents the URI of the object. This is also the case for Values which are not attached.
- For attached Values, **V** represents the literal value (see the next section).

### Values

The serialization of a Value usually involves creating only the *object* part of an RDF statement.

A Value is serialized as a typed literal which is a combination of a string and a datatype specification. Different types of data are converted to a string form in accordance with the XML Schema specification (World Wide Web Consortium, 2004/d). For example, a Value holding a date "December 10, 2009" would be serialized as: 2009-12-10T00:00:00Z^^xs:date.

A special kind of Value is the String class, which is used for representing internationalized strings. Each language version of a string is serialized as a separate plain literal value in RDF, which is a combination of a string and an optional language tag (World Wide Web Consortium, 2004/s). This

means that a single String object can result in multiple statements – one for each language version. Again, this can introduce an ambiguity for deserialization: a Bag of independent Strings will have the same RDF representation as a single String. The problem is solved by treating all plain literals with a language tag having the same RDF subject and predicate as being part of the same String. For example, the statements below would result in two separate String objects:

```
ex:Object      ex:Property1  "My name is Peter"@en .
ex:Object      ex:Property1  "Jeg hedder Peter"@da .
ex:Object      ex:Property2  "RDF is nice"@en .
```

Values can also be persisted as standalone objects. This is required when the Value needs to be referenced using an URI. In this case, the *rdf:value* property is used. For example:

```
ex:SomeValue   rdf:value     "This is a value" .
```

### Types and Properties

Types and Properties are serialized as statements having the Object associated with a Type or Property as their subject. Figure 14 summarizes all relationships that are defined on the Type and Property classes and which need to be serialized.

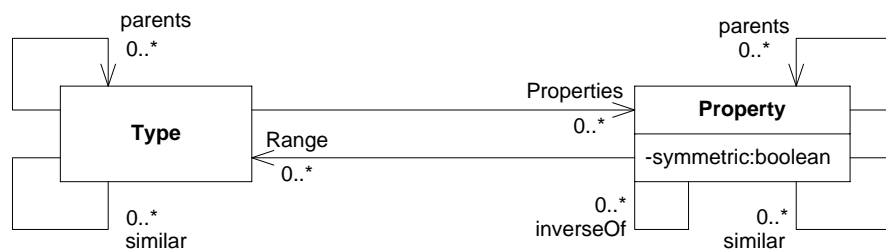


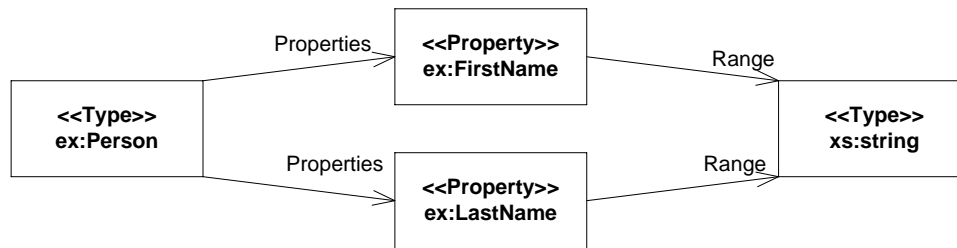
Figure 14: Diagram of all relationships owned by Properties and Types

The table below describes how the above relationships are mapped to RDF statements. *Source* and *Target URIs* denote the URIs of the elements at the source and target of a directed relationship, respectively. For bidirectional relationships, two sets of statements are constructed: one using the first end of the relationship as the source, and the second using the other as the source. One-to-many relationships are serialized by creating statements having the same subject and predicate but different object for each relationship end.

Relationship or property	Subject	Predicate	Object
parents	Source URI	<i>rdfs:subClassOf</i> (Type) <i>rdfs:subPropertyOf</i> (Property)	Target URI(s)
properties	Target URI(s)	<i>rdfs:domain</i>	Source URI
range	Source URI	<i>rdfs:range</i>	Target URI(s)
similar	Source URI(s)	<i>owl:equivalentClass</i> (Type) <i>owl:equivalentProperty</i> (Property)	Target URI(s)
inverseOf	Source URI(s)	<i>owl:inverseOf</i>	Target URI(s)
symmetric	Source URI	<i>rdf:type</i>	<i>owl:SymmetricProperty</i>

Table 3: Mapping of Type and Property relationships to RDF

For example, the following object graph:



would be serialized as:

```

ex:FirstName    rdfs:domain    ex:Person .
ex:LastName     rdfs:domain    ex:Person .
ex:FirstName     rdfs:range    xs:string .
ex:LastName      rdfs:range    xs:string .
  
```

### Equivalence

Equivalence between objects and similarity between types is expressed in RDF through three OWL constructs with slightly different meanings. The first one, *owl:equivalentClass*, applies to classes of things and states that the sets of instances of both classes are equal but the classes themselves have a different intentional meaning. In other words, the classes represent different concepts but when one object is an instance of the first class, it is also an instance of the other class. For example:

```

ex:US_President    owl:equivalentClass    ex:PrincipalResidentOfWhiteHouse .
  
```

The second construct, *owl:equivalentProperty*, resembles the previous one except that it operates on properties instead of classes. Equivalent properties have the same values, but do not imply the same meaning. This means that if property *P1* of object *O* has value *V* and *P1* is an equivalent property of *P2*, then property *P2* of object *O* will also have value *V*.

The above two OWL properties are used for serializing the Type and Property similarity, respectively.

The last OWL construct, *owl:sameAs*, states that the things identified by two separate URIs are actually the same thing – they have the same real world identity. In essence, this construct expresses that two URIs are synonyms. For example:

```

ex:BillGates        owl:sameAs        ex:WilliamHenryGates .
  
```

This construct is used for serializing the equivalence relationship between Objects.

#### 4.1.3. Ontology

The ontology described below defines the actual domain model for the personal information management system.

## Basic types

### Person

A type for describing a person. This class inherits most of its properties from *foaf:Person*, which includes such properties as names (family name, first name, given names), gender, email address, interests and a list of other people this person knows.

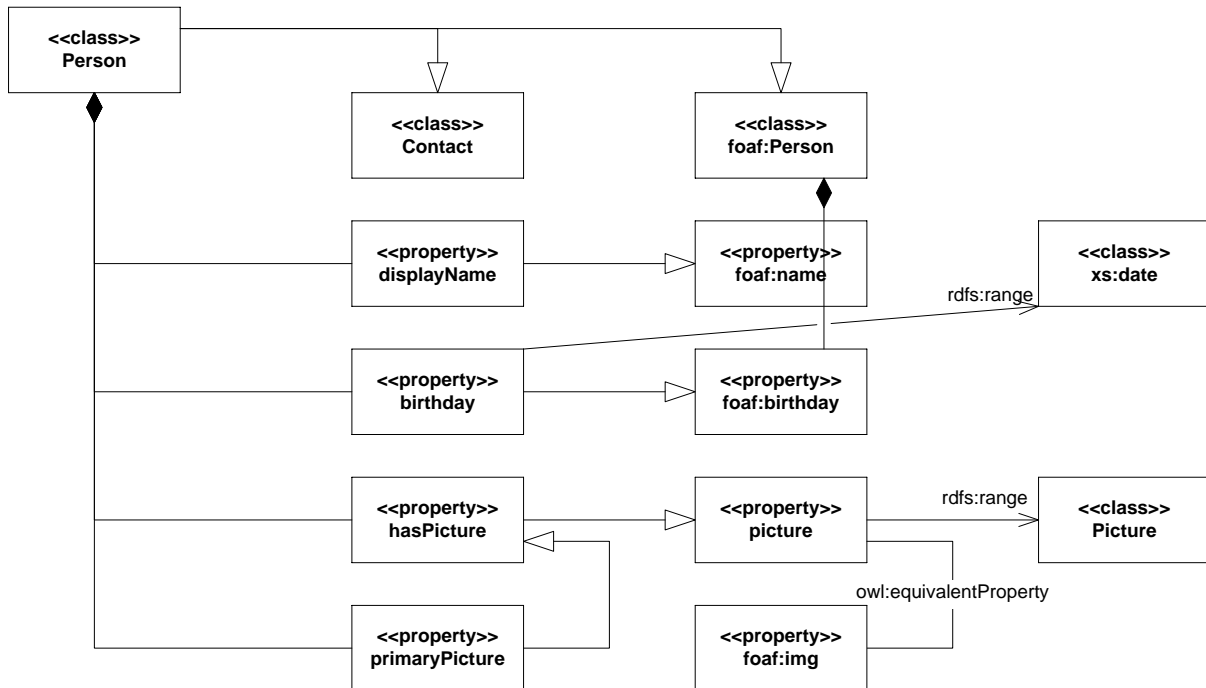


Figure 15: UML class diagram for the Person type

Some of the properties that are already defined in *foaf:Person* have been specialized in this class. These include:

- *displayName* is a property for holding the full name (i.e. given names plus surname) of a person, formatted in a way that should be used for displaying by the application. The *foaf:name* property, which it inherits from, is a generic name for any type of thing.
- *birthday* is a specialized version of *foaf:birthday* that specifies its range to be a date value.
- *hasPicture* is a subproperty of the *picture* property that associates a *Picture* to a *Person* – listing all pictures that depict this *Person*. Note that *picture* has an inverse property called *pictureOf*, which associates the *Picture* with the thing that it depicts.
- *primaryPicture* specifies the primary image for a person – i.e. a picture that is used when a single image representing the person needs to be shown. As this is a subproperty of *hasPicture*, all pictures associated via *primaryPicture* are also considered to be associated via *hasPicture*.

### Picture

A type for representing an image. This class is an equivalent of *foaf:Image*, which is used for describing pictures in the FOAF ontology, and *nexif:Photo*, which associates EXIF attributes with

images. By having these two classes defined as equivalent ones, the application will treat them in the same way as it handles *Pictures*.

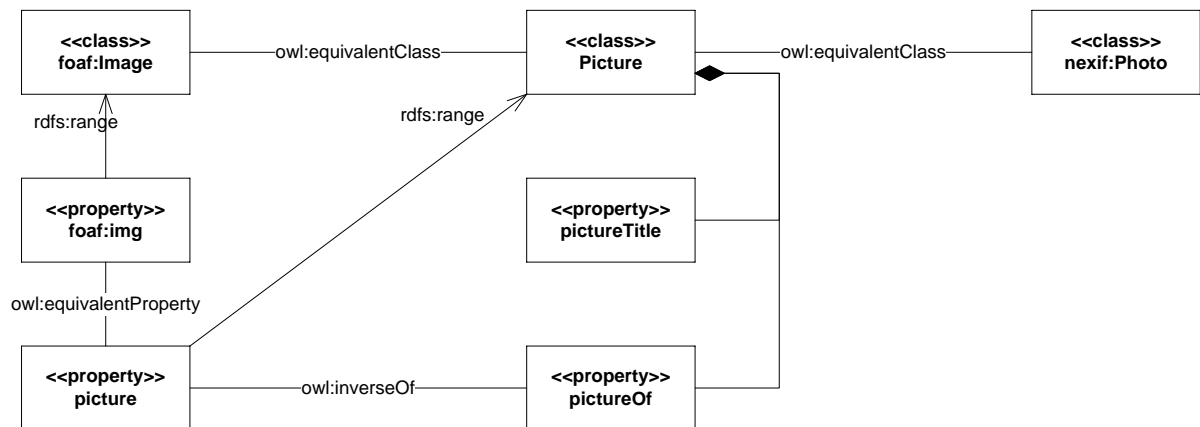


Figure 16: UML class diagram for the *Picture* type

A *Picture* can be associated with other objects through the *pictureOf* property, which states that the picture depicts another object. Whenever such a statement is created, an inverse statement, saying that a given object is depicted in the picture, is automatically created due to the inverse *picture* property.

A *Picture* type is automatically associated with all images imported into the application. For supported image types, such as JPEG files, the application will try to extract additional metadata from the image file, such as when the picture was taken and what camera settings were used.

### To-Do Task

A To-Do Task is a type for keeping track of things that the user needs to do. The model for this type is very simple – it is composed of a title, a description, a due date and time and a completion status (Pending, Started, Completed).

To-Do Tasks can be created as standalone items, but they are mainly intended to be used for annotating existing objects. In this way, the to-do item will serve as a reminder for another object which is directly related to the task that needs to be completed.

### Notebook Page

A Notebook Page is a type intended to realize the metaphor of a page in a notebook, where one can write, draw, attach pictures, newspaper clippings, and so on. The electronic Notebook Page is a blank area where the user can enter text, put pictures, URLs, references to other objects, etc. In this way different types of information can be combined as needed to form a single note. The elements on the Page can be positioned and resized as the user desires.

Figure 17 shows the involved types and properties:

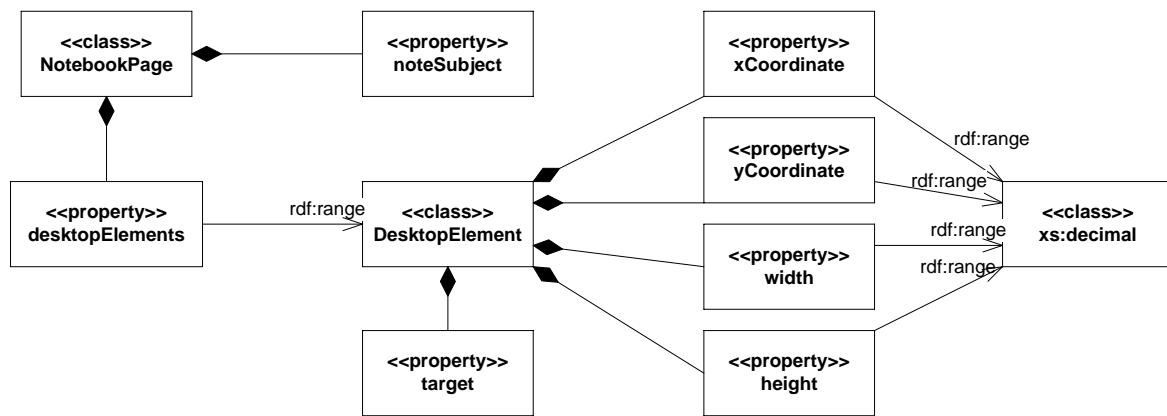


Figure 17: UML class diagram for the NotebookPage type

### Location

A type for representing a geographical location. Objects of this type can form hierarchies describing which locations are a part of another bigger location. Such a hierarchy can be used for finding associated objects. Consider a set of pictures associated with the location *Copenhagen*. As *Copenhagen* is a part of *Denmark*, displaying the pictures for Denmark might also include those for Copenhagen.

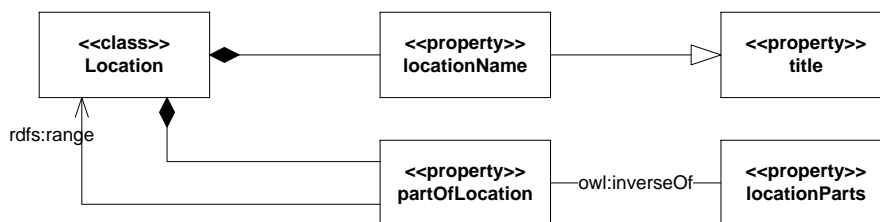


Figure 18: UML class diagram for the Location type

### Photo Album

A photo album is a collection of pictures. It can have a title and an associated location, which describes where the pictures were taken.

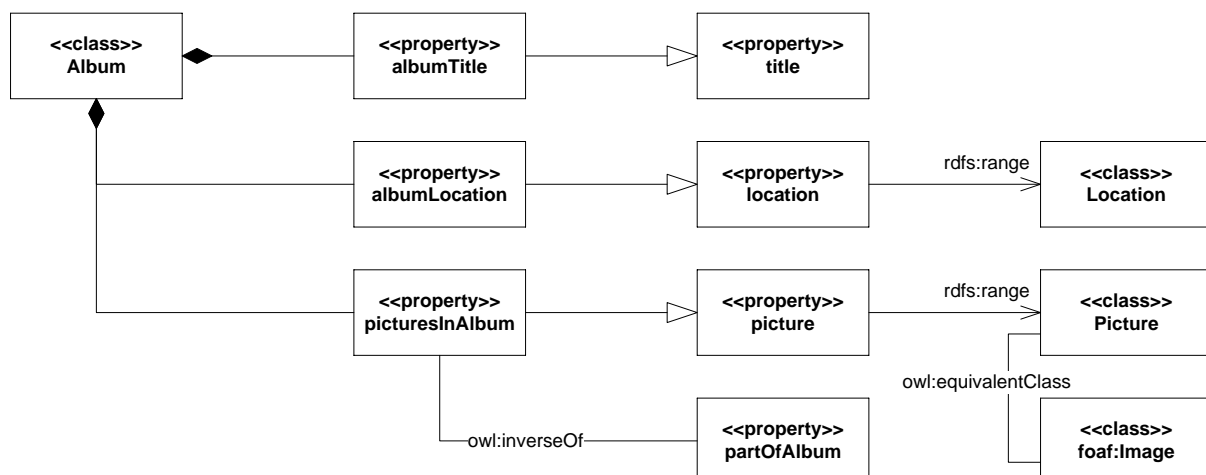


Figure 19: UML class diagram for the Photo Album type



An album is intended to group pictures that share something in common. A user might create an album for grouping pictures from a single trip, but could also use it for storing pictures of favorite places, etc.

### Trip

A Trip type is intended for grouping information related to a travelling. It contains a title, start and end time and a set of participants. Additionally, a Trip can be divided into a set of smaller Trip Fragments, which describe the individual places visited during the trip.

### Foreign types

The PIM system relies on several additional ontologies for working with external information. These are:

- Dublin Core (DCMI) – defines terms (e.g. title, creator, publisher), for describing resources typically found on the Web.
- Friend-of-a-Friend (Brickley, et al., 2007) – allows for describing persons, relations between them and to other objects.
- NEPOMUK Message Ontology<sup>9</sup> (NMO) - defines types for describing emails and instant messages; it is used by the Aperture (Ape09) framework for importing messages.
- NEPOMUK EXIF Ontology<sup>10</sup> (NEXIF) - an adaptation of the EXIF standard for describing metadata associated with digital cameras and photography as an RDF ontology; it is used by the Aperture framework for importing pictures.
- NEPOMUK Contact Ontology<sup>11</sup> (NCO) - defines types for describing contact information; it is also used by the Aperture framework.

Only some of the types and properties were explicitly integrated with the PIM system. That is, some of them were manually defined to be parents or children of, or that they are similar to the types and properties described in the previous section. For example, NCO's Contact is defined as a parent of the Person type. Some foreign types (such as NMO's Email and Email Folder) had custom views created for them so that the user can easily work with the data they describe. However, even if a type or property was not explicitly incorporated into the PIM system, the user should still be able to work with it if it has associations to other well-known types and properties, such as the ones defined in the Dublin Core or RDF Schema ontologies.

### Important relationships

Some of the relationships described above require a more in-depth look.

A common functionality for the PIM application is to display a name or title for an object, so that it can be easily identified by the user. For example, when a To-Do task is opened in a new window, the window title should contain the name of the To-Do task. This is achieved by defining all the individual title properties of various types as subproperties of a single *title* property (see Figure 20). The system can then exploit this relationship to find a correct value to display.

<sup>9</sup> <http://www.semanticdesktop.org/ontologies/2007/03/22/nmo/>

<sup>10</sup> <http://www.semanticdesktop.org/ontologies/2007/05/10/nexif/>

<sup>11</sup> <http://www.semanticdesktop.org/ontologies/2007/03/22/nco/>

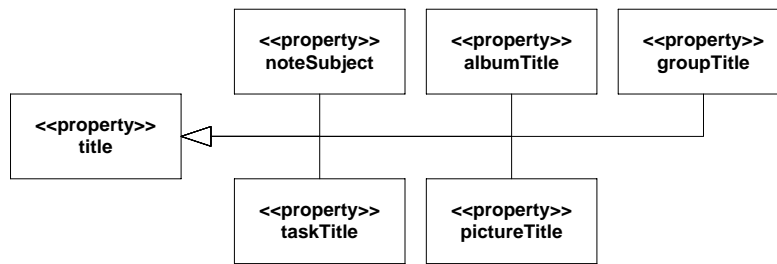


Figure 20: UML class diagram for title properties

However objects that were imported from external sources (such as from Facebook or Gmail) use foreign ontologies that have no notion of the title property used by this system. Two common properties that serve a similar purpose in other ontologies are the label property defined by RDF Schema (World Wide Web Consortium, 2004/s) and the title property defined by Dublin Core (DCMI). To properly handle title values for objects using these properties, an equivalence relation is defined between them and the title property used in the PIM system:



Figure 21: Equivalence relationship between title properties

A special case of the problem of finding the correct value to display as a title of an object is related to the *Person* type. A person has a first name, given names, surname, nickname, etc. Not all of these properties need to be assigned for every person, but the application should always display the most relevant value available. This is accomplished by properly defining the hierarchy of the *Person*'s name properties:

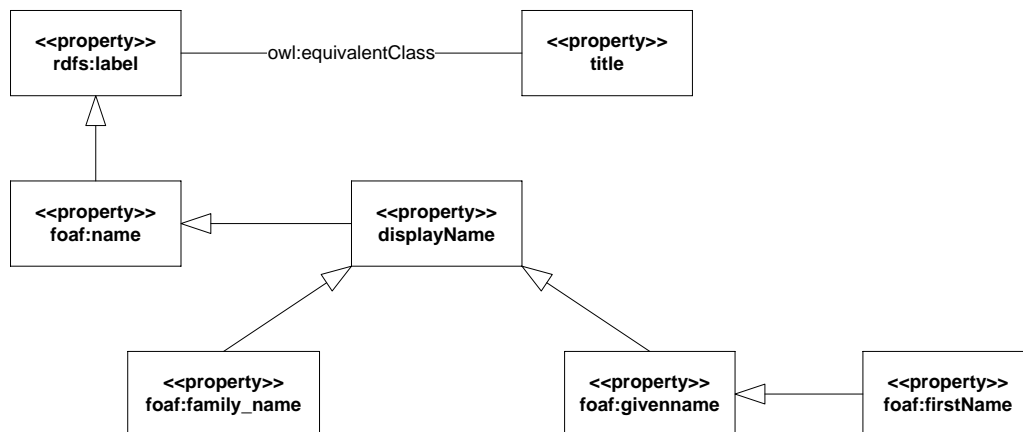


Figure 22: UML class diagram for hierarchy of name properties

When a value for the title is needed, the application will try to find it by starting at the *title* property and going down the hierarchy until a value is found. A detailed description of the algorithm will be given in Section 4.3.4.

## 4.2. Presentation layer

The primary element of the user interface for the PIM application is the View. A View is a GUI control for displaying information contained in an Object or a Collection. It implements the logic

necessary to present a certain type of information in a predefined way. A single type of information can be thus presented in many ways by choosing different views. The PIM system makes a general distinction between the following types of views:

- A *normal* view is used for displaying objects to the user when a detailed and complete set of information is needed. Normal view are usually displayed in a separate window or tab, and occupy a significant portion of the screen.
- A *façade* view is used for displaying an object for identification purposes. Façade Views are commonly used when many Objects are displayed as part of a Collection. They present only the most important information about an object – e.g. a picture and a name of a person.
- A *new object* view is a variant of a normal view that is used for editing a new object just after it has been created. It can present a special set of fields which should be initially filled by the user.

The main window of the application displays views in the form of tabs – whenever an Object or Collection is opened by the user, a new tab is created with an appropriate view for the type of object to display. Views are also embedded in other views. In this way, more complex views can be constructed from simpler ones.

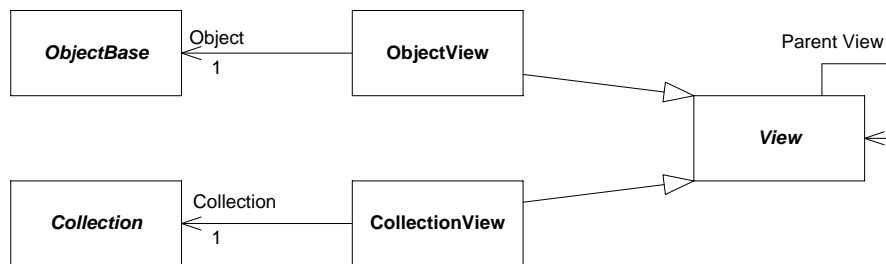


Figure 23: UML class diagram for View classes.

The simplest types of views are *value* views, which are used for displaying a single primitive value, such as a text string, a number, date and time, an image, etc. They are combined together to form more complex built-in views. In turn, built-in views can be combined by the user to form custom views.

Nesting simpler views within more complex ones requires some cooperation between them in order to give the user a unified experience. This is accomplished by propagating certain properties down the view hierarchy - whenever a property of a view changes, it is propagated to all child views that have a default value for that property set. The properties that use this mechanism are listed below:

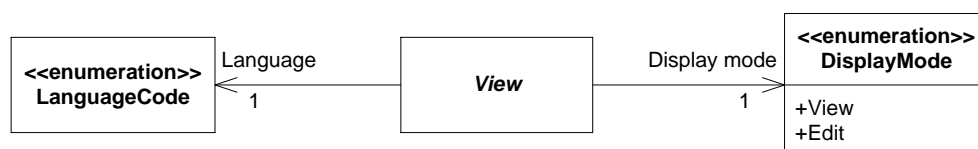


Figure 24: View properties

- Language specifies the language used for displaying messages. This affects the selection of Property titles and String values.
- DisplayMode determines whether a view should only display information or also allow for them to be modified.

Apart from the above propagating properties, a View might define other user-configurable properties that affect how information is displayed. These properties can be modified by the user for a view being shown on screen (the settings are effective as long as the view is displayed and are not saved), or for the whole class of views of a certain type (using the View editor described in Section 4.2.5).

Other elements of the user interface include controls for editing types and properties, displaying details about an object's types and relations and searching for objects.

#### 4.2.1. Built-in views

The application defines several types of built-in views for interacting with typical forms of data. The visual layout and behavior logic of these views is hard-coded in the application and cannot be modified by the user.

##### Value views

Value views enable interaction with basic forms of data – they are used in conjunction with Value objects. They either display read-only information or allow it to be modified, depending on the setting of their DisplayMode property (see Figure 23). The following value views are defined:

- **TextView** – A view for displaying and modifying plain text strings contained in Value and String objects. For the latter, the view displays a language version of the string that matches the Language property of the view.
- **DateEditorView** – A view for displaying and modifying date and time values. This view supports Value objects with a datatype of *xs:date* and *xs:dateTime*.
- **NumberEditorView** – A view for displaying and modifying numbers corresponding to the numeric types defined in XML Schema (World Wide Web Consortium, 2004/d).
- **ComboSelectionView** – A view that displays a single value which can be selected using a drop-down list from a set of values obtained from a Collection. This view requires that the Range of the property for which a value is edited would be a Bag holding a list of values, in addition to being a Type.
- **RichTextView** – A view for displaying and editing rich text strings contained in Value objects of type *HtmlText* (i.e. rich text is serialized to a HTML form). The view supports basic text formatting operations, such as changing the font type, size, style and color.
- **ImageView** – A view which treats the URI of an object as a Web URL and uses it to display an image retrieved from that address.

##### Outline view

The Outline view is the simplest type of a view for displaying Objects. It simply lists all properties defined for an Object and their values. As each property is associated with multiple values through a Collection, OutlineView uses other CollectionViews and ObjectViews to render the actual values. Displaying an Object in an OutlineView thus involves the following steps and components for each property contained in the Object:

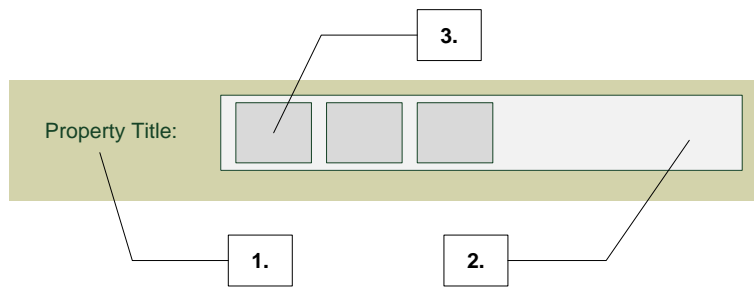


Figure 25: Diagram of the OutlineView

1. OutlineView displays the title of the property.
2. OutlineView chooses a CollectionView for displaying the Collection associated with a property. Usually the SimpleCollectionView is used.
3. The CollectionView displays the elements of the Collection using façade ObjectViews.

A special feature of the OutlineView is the possibility of quickly adding new properties (i.e. extra information) to an Object. By right-clicking on the view, the user can invoke a context menu from which the “Add new property” item can be selected. This displays a drop-down box where a property already defined in the system can be selected or a completely new property can be quickly created. For example, the user can create a new property “Bank account” to store the account information for a person. Once this property is created, it can be reused for other Objects.

### Desktop view

DesktopView is used for displaying Objects of type Notebook Page. A Notebook Page represents a page on which other objects can be placed. An Object of this type contains a list of references to other Objects together with coordinates and dimensions for displaying them on the screen.

The view is composed of the following components:

(see Figure 41 in Appendix A for an example of how this view is realized in the prototype)

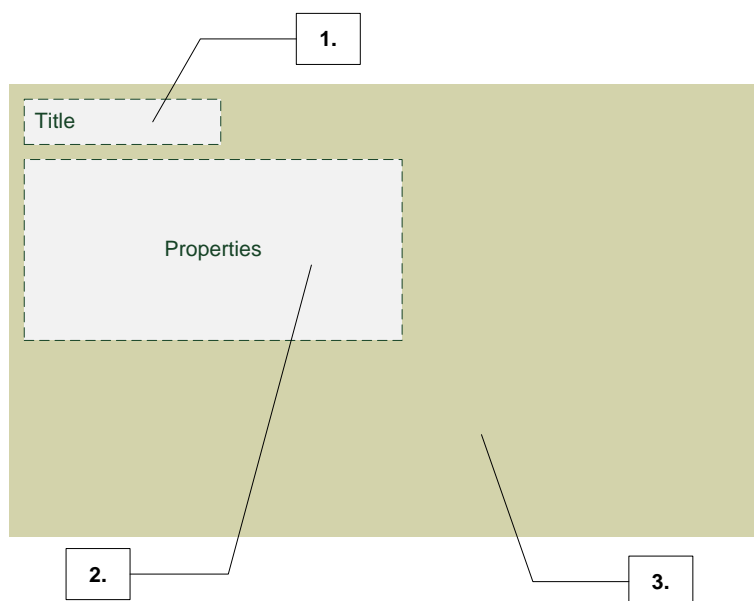


Figure 26: Diagram of a DesktopView

1. A box showing the title of the Notebook Page (the *noteSubject* property).
2. An OutlineView for displaying additional properties associated with the Notebook Page.

The rest of the view (3.) is an area where Objects can be placed. Each Object is rendered using a default ObjectView selected for that type of Object. The ObjectView is placed inside a border that can be resized and dragged by using the mouse, which results in the ObjectView being resized and moved around the Notebook Page, respectively.

By clicking on an empty area of the DesktopView, the user can create a new *HtmlText* object that will be rendered using the RichTextView. In essence, the user can add additional rich text notes to the Page just by clicking on it.

Similarly to the OutlineView, the user can right click on the list of properties (2.) and choose *Add property* to add a custom property to the note.

This view defines the following user-configurable properties:

DisplayProperties	Determines if a list of properties assigned to the Object be displayed in the View, i.e. should the box labeled (2.) in Figure 26 be visible.
DisplayTitle	Determines if title of the Notebook Page should be visible (1.)

### *Façade view*

FacadeView is the view that is used by default to display objects in *façade* mode, if no specialized view is available for that object type. The object is displayed as a button containing the title of the object (obtained through the *title* property, or the URI of the object if no title has been set) and a list of types of the object. Clicking on the button opens the object in a new tab using the default view – this will usually display the details of the object.

### *Xaml view and Typed Outline view*

XamlView and TypedOutlineView are two classes for displaying user-defined views. The former uses an Extensible Application Markup Language (XAML) file (Microsoft) as a source for the view definition. A view defined in a XAML file can contain any Windows Presentation Foundation (WPF) elements (Microsoft), such as buttons, labels, text boxes, tabs, etc. Additionally, it can contain other views. A description of how views are defined in XAML will be given in Section 4.2.2.

One of the views that is typically used in conjunction with the XamlView is the TypedOutlineView. This view displays a list of properties and values in a way similar to the normal OutlineView, which was described earlier. While an OutlineView displays all the properties associated with an Object, when using a TypedOutlineView the user can specify which values should be displayed on screen.

Items that are to be displayed by this view are describes using the Item class:

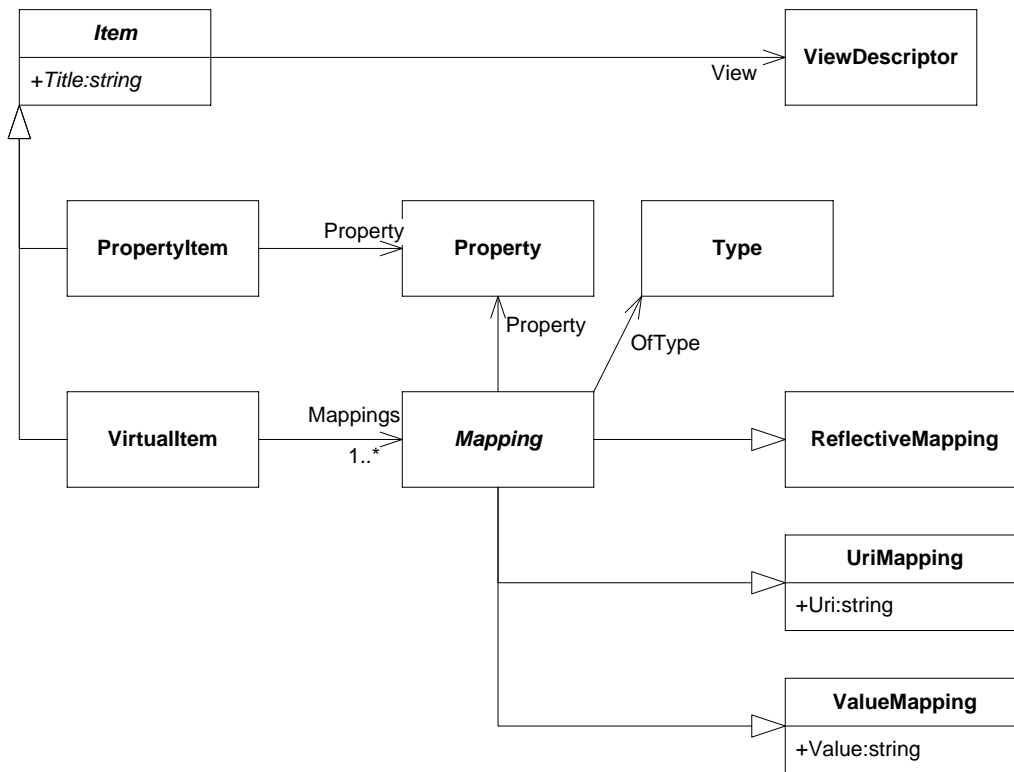


Figure 27: UML class diagram for TypedOutlineView items

An Item defines the title to be displayed on screen and the view that should be used for displaying the values. The view can be either an ObjectView – then only the first element of the underlying collection will be displayed, or a CollectionView, which will display the whole collection. The ViewDescriptor class holds a name of the View class and contains logic to instantiate it.

An Item can be either a PropertyItem, which displays values associated with a Property, or a VirtualItem, which holds a simple query that will return a set of values for display. The query is defined in terms of one or more Mappings, which are combined using the “and” logical operator:

- A ValueMapping retrieves all objects which have the given Value assigned using the given Property. The datatype of the Value can also be specified. An example query constructed using this mapping would be: “show all objects which have an Age equal to 24”.
- A UriMapping retrieves all objects which reference an Object identified by the given URI assigned as a value of the specified Property. In addition, the type of the returned objects can be restricted to the specified one (OfType). An example query constructed using this mapping would be: “show all objects of type Picture which reference a Person identified by the URI ex:JohnDoe”.
- A ReflectiveMapping is equivalent to UriMapping except that the URI of the referenced object is automatically set to the URI of the object currently displayed in the TypedOutlineView. For example, a custom Person view defined as a TypedOutlineView could use this mapping to show all pictures that reference the currently displayed person.

Furthermore, this view defines the following user-configurable properties:

Orientation	Determines whether the items should be displayed each on a separate line (Vertical) or all on the same line (Horizontal).
-------------	---

### *Simple Collection view*

SimpleCollectionView displays a list of objects belonging to a collection. The objects are rendered one by one using a default façade view for that type of object.

The list of objects can be displayed in one of the three modes: horizontal (all elements are displayed on a single line), vertical (each element is displayed in a separate line) or wrap (elements are displayed in a line, but overflow to following lines). If the collection being displayed is a Bag, then the elements are shown in random order. On the other hand, if the collection is a List, the elements are displayed accordingly to their position in the List. Moreover, the elements can be reordered by dragging them with the mouse.

SimpleCollectionView can also display elements grouped by a certain property value. The user selects a property that will be used for grouping from among the properties of the Objects present in the collection. Only elements of type Object having a value for that property are displayed in the view when the grouping feature is active. If an Object has multiple values for that property, it is displayed in more than one group.

See Figure 40 in Appendix A for an example of how this view is realized in the prototype.

This view defines the following user-configurable properties:

LayoutMode	Horizontal, Vertical or Wrap.
GroupBy	The property used for grouping.

### *Alternative values Collection view*

AltCollectionView is a view that displays just a single value from a collection at a given time. The user can switch the currently displayed value by clicking a button, or can use a drop down list to directly select a value.

This view is intended to be used when there typically is just a single value for a property, but additional values are possible. For example, most people will have a single mobile phone number, but some might have more.

### *List view*

ListView displays Objects belonging to a collection in the form of a table, where each column corresponds to a specific property. The header of a column can be clicked to initiate sorting by that property.

The set of properties that will be displayed in the table can be configured by the user. By default, the view will use all the properties of the type being the range of the property associated with the view's collection. For example, if the ListView is embedded in another view to display the values of the "pictures" property (which contains elements of type "Picture"), then it will use the properties of Picture as columns.



### Timeline view

The TimelineView is a view for displaying Objects belonging to a collection grouped by date. The user can select a specific property of type `xs:dateTime` which will be used for comparison, or the system can use the first property of that type which it finds assigned to an object. If an Object does not have a suitable property, it is not displayed at all in this view. If an Object has multiple values for that property, it is displayed multiple times in the view under different dates.

The view is composed of three parts:

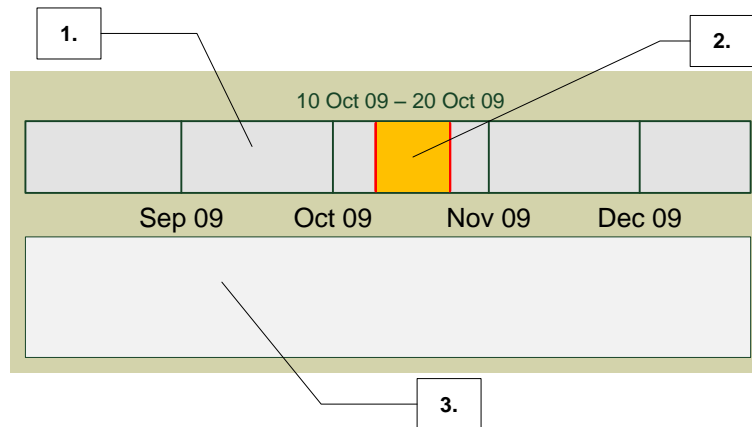


Figure 28: Diagram of a TimelineView

1. A draggable timeline.
2. A period indicator – the user can drag the period boundaries to change the length of the period.
3. A SimpleCollectionView for displaying a set of objects that belong to this period.

This view defines the following user-configurable properties:

GroupBy	The property used for grouping.
PeriodLength	The length of the period for classifying objects into a group.

#### 4.2.2. Custom views

Custom views are views which can be created and modified by the user. Defining a custom view involves defining the graphical layout and contents of the view and assigning it to one or more types for which this view is suitable. The prototype supports only custom ObjectViews, i.e. it is not possible to define a custom view for a Collection.

The contents of a view is defined using XAML (Microsoft), an XML-based language for declaratively defining the elements of the user interface. Defining a user interface in XAML is somewhat similar to how web pages are defined using HTML. The prototype design uses the XAML concept of attached properties<sup>12</sup> to assign which user interface elements defined in the view should display particular Object properties.

The following attached properties can be used:

<sup>12</sup> An attached property is simply a named value assigned to an existing object. The object itself doesn't need to have any knowledge or support for that value.

- **Property** – specifies the URI of the property that is to be used as a source of values for an element;
- **VirtualCollection** – specifies a Collection created by executing a user-defined query as the source of values for an element;
- **InheritanceDirection** – specifies how should values from ancestor and descendant properties be used when no value for the given property is assigned to the Object; used in conjunction with the “Property” attached property.

The above attached properties can be assigned to standard WPF elements, such as text boxes, labels, buttons, etc., or to ObjectViews and CollectionViews. When a custom view is used to display an Object, the Object itself is assigned to the topmost element in the view. If that element has no attached property assigned, then the child elements are inspected. This operation continues all the way to the leaf nodes of the element tree. If an element with a Property or VirtualCollection attached property is found, then these properties are used for retrieving the values to be displayed.

For example, the façade view for a Person type is defined in the following way:

```
<v:XamlView ...>
  <StackPanel>
    <c:ImageView v:XamlView.Property="pim:primaryPicture"
v:XamlView.InheritanceDirection="Both" IsFacadeMode="True"/>
    <Label v:XamlView.Property="foaf:name"
v:XamlView.InheritanceDirection="Both" Content="{Binding /Value}"/>
  </StackPanel>
</v:XamlView>
```

It displays a StackPanel (a standard WPF element for displaying child elements beside each other), which contains an ImageView (a built-in ObjectView) that will display an image obtained from a property called *pim:primaryPicture*, and a Label (another standard WPF element) that will display the name of the person.

The prototype defines several custom views for handling types defined in the ontology:

Type	View	Description
<b>Email</b>	EmailView	Displays a single e-mail message: a sender, recipients, subject and message body.
<b>Location</b>	LocationView	Displays details of a geographical location: name, parent and child locations.
<b>Mail Folder</b>	MailFolderView	Displays a list of messages in an e-mail folder in a way similar to traditional e-mail clients.
<b>Person</b>	PersonView	Displays details about a person, including names, birthday, picture, e-mail address and homepage. Additional details are available through a series of tabs: people this person knows, pictures, notes and trips that are related to the person.
	PersonNewView	A view listing only the most important properties to fill in when creating a new Person object: first name, surname, nick name and date of birth.
	PersonFacadeView	A façade view for Person. Displays an image of a person together with their name.

<b>Photo Album</b>	PhotoAlbumView	Displays a description of the photo album together with a gallery of images belonging to that album.
<b>Picture</b>	PictureView	A detailed view for a Picture, which, apart from the image itself, displays associated objects ( <i>pictureOf</i> property), a title and description, and a set of EXIF image properties (such as camera maker, settings when the picture was taken, etc.)
<b>ToDo Task</b>	ToDoView	Displays the title, due date, completion status and description of a to-do task.
<b>Trip</b>	TripView	Displays a description of a trip, including a title, participants and visited locations.

Table 4: A list of pre-defined custom views

### 4.2.3. View selection

A complex view displays information to the user by relying on simpler views to render their corresponding parts of the data contained in an object or a collection. For example, a Collection view might display objects in the form of a list, but the way each individual object in the list is rendered is left to an automatically chosen façade view.

The system maintains a list of object views that are suitable for a given type, similar to the one given in Table 4 above. Each type can be mapped to zero or more views. For each such mapping, the type of the view is indicated (i.e. normal, façade or new object, as described in the beginning of this section). One of the views is marked as the preferred view – a view that will be automatically selected, when needed. The remaining views are offered as suggestions when the user wants to select a different view by using the context menu.

A complex view can request a view that will be suitable for displaying a property, a collection or an object or value. These cases are handled in the following way:

- A property – The range of the property is inspected. If the range indicates that this property stores literal values (e.g. strings), then an Alternative values collection view is selected. Otherwise, a Simple Collection view is selected.
- A collection – A Simple Collection view is always selected.
- An object or value - The mapping between types and views is consulted to find a preferred view for one of the types of the object or value. The parent types are inspected next if no view is found. Additionally, the preferred view must match the requested view-type (normal/façade/new-object). Finally, if no view can be found in this way, a default view is selected: an Outline view for the normal view-type, a Façade view for façade type and a Typed Outline view for new-object.

### 4.2.4. Context menu

Any area of a view can be right-clicked to invoke a context menu with actions relevant to the clicked element. Different actions are displayed dependant on the type of the element and whether it is part of a collection. Standard actions include:

Applies to:	Title	Description
Objects	Add Annotation...	Creates a new annotation for the selected object. The user is presented with a list of possible annotations (Note or To-Do).

Objects	Open in Application...	This option is only available if the object was imported from an external source. It will try to open the object in the application that the object originated from.
Collections	Add Element...	Creates a new object of the selected type and adds it to the collection.
Objects and Collections	Open With...	Opens the object or collection using a chosen view. The user is presented with a list of views suitable for the object's types.
Properties	Edit Property	Invokes the Ontology Editor on the selected property.
Objects and Values	Edit Type...	Invokes the Ontology Editor on the selected type. The user can choose from the types associated with the Object or Value clicked.
Objects and Values	Show Details	Displays the Object Details form with information about the clicked object.
Objects and Values	Delete	Marks the Object or Value for deletion. It will be removed from the system.
Persistent <sup>13</sup>	Edit / View	Switches between View and Edit mode for the selected view.
Persistent	Replace View...	Replaces an existing view with another one chosen from a list.
Persistent in Collection	Remove Element	Removes an element from a collection.
Any	Set Language...	Changes the language used to display the selected element.

Table 5: Summary of context menu actions

In addition to the actions listed above, views can define their own additional actions that apply to the elements they are displaying. For example, the Outline View provides an “Add Property” context menu action, which allows for adding new properties to an object.

Some of the actions listed above require further discussion:

The “Open in Application...” action lets the user quickly open the external source from which an object was imported. As described in Section 2.3.3, the PIM system is intended to help the user organize, retrieve and keep track of different kinds of information. It is not intended to replace the functionality offered by specialized applications. That is why being able to quickly open the original source is an important function. If an object was merged with other objects that were also imported from external sources, the context menu will let the user choose which application should be invoked.

The “Replace View...” action is intended to let the user temporarily modify how information is displayed on the screen. When a view is displayed, child views are selected automatically based on the view's settings or by choosing the most appropriate view for the given object or collection, as described in the previous section. However, the user can also choose another view to replace the current one. This action is similar to “Open With...”, except that instead of opening the view in a new tab or window, it replaces it in-place of an existing view.

<sup>13</sup> Persistent objects include Objects, Values and Collections.

The potential use cases of this function are as follows. First, it lets the user view the same object in different ways. This could be used for switching between views that focus on different parts of the information contained in an object. Secondly, it allows collections of objects to be viewed in different ways. Suppose a Photo Album view displays a list of pictures as a set of thumbnails. However, the user might be interested in viewing those images grouped by time (e.g. using the Timeline view), or to view the title and description of each picture in the form of a table (the List view).

#### 4.2.5. View editor

The view editor is a simple graphical editor for defining custom views. It allows the user to create specialized views for new or existing types of information. As described in (Huynh, et al., 2002), the *“user interface changes performed by the user are high-level: they are to programmers’ user interface work as interior design is to carpentry. In other words, customizing the user interface is akin to editing a word processing document or manipulating a spreadsheet.”*

The view editor functions in the following way. A user is presented with a blank form (when creating a new view) onto which UI elements can be placed. The following elements should be available:

- A Label – for displaying read-only text.
- A Textbox – for displaying modifiable text.
- A Grid – An element container that has a customizable number of rows and columns; each cell can contain one other element, including another grid. The width and height of rows and columns can be modified.
- Any built-in or user-defined View.

The UI elements are sized automatically to fill the full space available in the parent container. Thus all layout and element sizing is accomplished using the Grid element.

Each element can be edited in order to set its user-configurable properties. These will include:

- The Property, VirtualCollection and InheritanceDirection attached properties as described in the Custom views section.
- A static text to display in the Label element (the Label can also display values obtained using the attached properties).
- Any of the built-in view-specific user-configurable properties, such as the LayoutMode and GroupBy properties for SimpleCollectionView.

The view editor will load and save view definitions in XAML format.

#### 4.2.6. Ontology editor

The ontology editor allows the user to modify the types and properties defined in the system. It is composed of two parts:

The type editor allows the user to create, edit and delete types. Apart from setting the title and description for a type, the set of properties for that type can be defined. When displaying a list of properties, the editor indicates which properties are defined specifically for the edited type and which are inherited from the type’s parents. Moreover, the user can inspect the location of the

edited type in the type hierarchy: the editor displays a list of ancestors, parents, children and descendants. The set of parents and children can be modified. Finally, the type editor lets the user define a set of similar types.

Similarly, the property editor allows the user to create, edit and delete properties. The user can set a title and description for a property, as well as suggested types for objects created as values of this property (the range of the property). The user can also inspect and modify the property's location in the property hierarchy: its ancestors, parents, children and descendants. The editor also allows for specifying special relationships to other properties: the set of inverse and similar properties, and whether this property is a symmetric one. Finally, the user can browse a list of types that make use of this property.

There are three major use cases for the ontology editor:

First, the user can define new types and properties for objects as he or she see fits. As the PIM system is intended for personal use, it is encouraged to define new types and properties when the existing ones do not fulfill the user's needs.

Secondly, the user can extend and modify existing types. If an existing type does not have a property that is commonly used by the user, it is easy to define it (the view editor can then be used to add this property permanently to an existing view). Additionally, the user can create new, specialized types that inherit some of the properties from existing ones.

Finally, the ontology editor can be used to relate new ontologies to the ones already defined in the system. This is a useful feature when working with data from external sources. If a foreign ontology has types representing the same concepts as one of the user's ontologies, the types and properties from one ontology can be marked as similar to matching types and properties from the other. In this way, the PIM system will treat objects and values imported from a new external source as it treats other objects and values representing the same concepts. For example, if a user would import data from a database about movies, they could relate the type Actor to an existing type Person. In this way, the system would use the same view for Actor objects as it does for Person objects.

#### **4.2.7. Internationalization**

The system provides lightweight support for internationalization. It enables the user to interact with the application in a chosen language and to use multiple languages when storing data. For example, when describing a photo that is to be published for others to see (more on this in Sharing information later on), a user might enter text both in English and in Polish so that different users can view it in their preferred language.

The underlying database model – RDF – allows for literal values to be stored with an associated language tag. This functionality is handled at application level by the String class. Most text strings presented by the user interface already originate from RDF data. These are type names, property titles and user-data values. In this way, the language used by the user interface can be customized.

The application runs with a global default language. Each view uses this language to select appropriate strings for display. The view's language can be changed to a different one using the context menu. Such a change will affect all descendant views as well.

Unfortunately, not all text strings can be localized. In particular, text used in custom views can only be entered in a single language.

### 4.3. Finding and managing information

One of the most important functions of an information management system is to allow the user to quickly find relevant information that he or she needs. This section describes how this functionality is realized in the prototype.

#### 4.3.1. Searching for objects

The most typical way for searching for information is a function that lets the user enter keywords that will be used as the basis for a full-text search. Such a function can be found in most applications – it is suitable for both structured and unstructured information. On the other hand, structured information, such as RDF data, can be searched by creating arbitrarily complex queries in one of the available languages (e.g. SPARQL). Entering a few keywords for a full-text search is easy, but often produces inaccurate results. Creating structured queries is more time consuming and requires the knowledge of the query language, but may find exactly what the user is looking for. Therefore a balanced solution is needed.

The prototype combines these two approaches by giving the user a possibility of defining queries in simple and advanced mode.

In simple mode, the user can:

- choose whether they are searching for an object, a type or a property;
- enter a few keywords that will be used as a basis for a full-text search on all primitive values and URIs associated with an object;
- for objects - choose a type that the object must be an instance of;
- for types/properties – choose a type/property as a parent for what is being searched.

For example, in order to find a person named “John Doe”, the user might choose an object search, enter “John Doe” in the keywords box and put “Person” in the type box.

In advanced mode, the user can refine the query by specifying a list of constraints that the objects must satisfy. The following predicates can be used when defining constraints when searching for objects:

- is instance of **T** – List all objects that are an instance of type **T**.
- has value **V** [via **P**] – List all objects that have a value **V**. The value can be either a primitive value (e.g. a string or date), or an object. Optionally, a property **P** can be specified that this value must be assigned to.
- has value **V** [max-distance **D**] – List all objects that have a value **V** associated with them through no more than **D** intermediate objects.
- is referenced by **O** [via **P**] – List all objects that object **O** has a reference to. Optionally, a property **P** can be specified through which the searched objects must be referenced.

When searching for types and properties, the following predicates can be used in addition to the ones described above:

- is subtype of **X** – The type or property must be a subtype/subproperty of **X**.
- belongs to **T** – The property must belong to type **T**.

The advanced search UI feature presents the constraints as a list where items can be added and removed. A predicate is selected from a drop down box, a plain text value can be entered from keyboard, while a type or property can be chosen using a text box with autocompletion by typing its title or URI. An object can be selected by dragging it from some other location and dropping it on the constraint value field.

A query constructed in the above manner (either in simple or advanced mode) is executed through a special Collection class – a `VirtualCollection`. This has the following advantages:

- Any `CollectionView` can be used to display the results. They can be presented in a form of a table, a list, a timeline, and so on. Depending on the view, different aspects of the listed objects can be emphasized.
- Elements can be added and removed from the `VirtualCollection`, which results in the objects themselves being modified to match the constraints defined in the query. For example, when results of a search for all objects of type `Person` which have a value “James” are displayed and a new object is added to that collection, it will automatically be assigned the value “James” (through a special *rdf:value* property, if no *via* property was specified) and the `Person` type.

#### 4.3.2. Related information

Another important way of finding relevant information is to view all information that is related to a particular object, so that the user can explore and follow links from one object to another (as explained in Section 2.3.4). This is accomplished through the Object Details form.

The Object Details form displays all objects that reference the selected object, grouped by their type. For example, viewing the details of a person could show all pictures, notes, to-do items, etc. that are associated with that person. The user can click on any of the displayed objects to open it, or right click and select the “Show Details” menu item to view the associations of that object. The Object Details form is displayed in a special side pane of the main application window. The side pane tracks the history of all the forms that were displayed in it, offering *back* and *forward* capabilities similar to a web browser. In this way, the user can go back to the details of a previous object if the link he or she followed is not the desired one.

Moreover, as the list of associated objects is displayed as a standard `SimpleCollectionView`, the user can use the context menu to replace it with any other view that is needed. For example, instead of displaying objects grouped by their type, the user can invoke a time-line view.

Furthermore, the Object Details form can be used for viewing and modifying tags associated with a particular object.



### Annotating objects

Annotating objects involves creating a new object that references an existing one. There is a special set of Types that can be used for creating annotations. In this prototype, only two types of annotations are supported: a note (the *NotebookPage* type) and a todo item (the *Todo* type).

When an annotation is created, the two objects become linked by the *pim:isReferencedBy* property, making it easy to find one object from another using the Object Details form.

### Linking objects

Linking objects is used to either express that the two objects are related in some unspecified way (but meaningful to the user), or to associate one object to another through a selected property.

Linking objects is accomplished through drag-n-drop. The user can drag an object from one view onto another object or a collection of objects. When an object is dropped on a collection, the object is added as a member of that collection. However, when an object is dropped on another object, a popup window appears asking for the type of association to establish between these two objects. The following choices are available:

- A “generic” link. This indicates that the user does not want to specify the type of association between the objects. In practice, a special property *pim:isReferencedBy* is used to create this link.
- A specific property. The user can select a property that is to be used to link these two objects. The application automatically suggests a property based on the types of objects that are being linked. For example, a “knows” property is suggested when linking two Person objects, and a “picture of” property when a Picture is dropped on a Person.

In addition to defining a link, the user can also indicate that two objects are equivalent by dragging one onto the other. This will merge the information contained in these objects into a single object. Any reference pointing to one of the original objects will now point to the newly created merged object.

#### 4.3.3. Unstructured information

Not all information stored in the PIM system is available as separate Objects. For example, the contents of an email message is just a text string, although it might contain fragments that could reference other Objects. In a similar way a picture is just a stream of binary data, but might depict persons that are represented in the PIM system as separate Objects.

The application therefore provides support for marking fragments of unstructured information so that they could be treated as separate Objects, allowing them to be used as property values, linked to existing Objects or just annotated.

The following data model is used for expressing information about fragments:

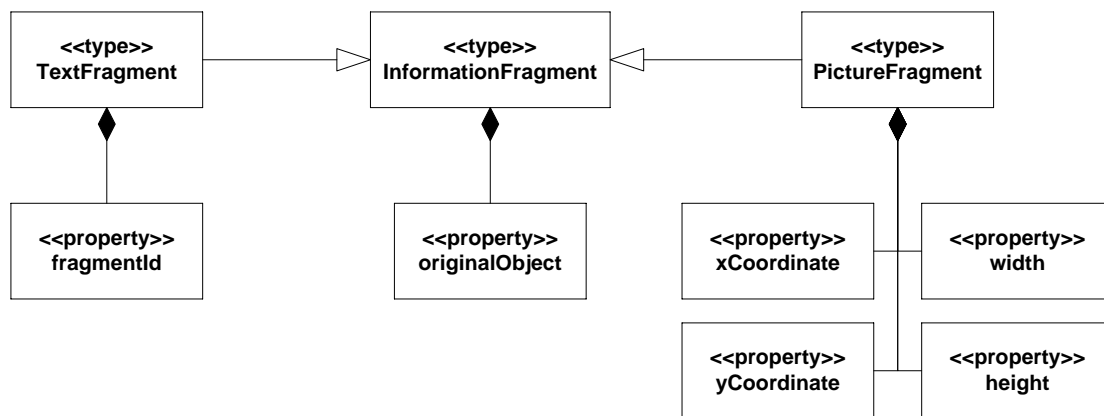


Figure 29: Information fragments data model

The *InformationFragment* type holds a reference to the object holding unstructured information. In case of a text fragment, the associated Value object will have to be stored in a form identifiable by an URI, so that it can be referenced (see Section 4.1.2).

When referencing objects in a picture, the user can move and resize a rectangle indicating the picture fragment that shows the relevant information. The coordinates of the rectangle are then stored in an instance of the *PictureFragment* type.

When referencing text, the user simply selects a particular text fragment. The original text stored in the Value object is then converted to an XHTML form corresponding to the *pim:HtmlText* type (if not already in that form). This is needed so that an XHTML tag can be inserted into the text, marking the selected fragment. The visual form of the text presented to the user does not change even if the conversion took place. The XHTML tag contains an attribute with a value that uniquely identifies the fragment within the scope of the Value object. This unique ID is then stored in the *TextFragment* object as the *fragmentId*.

For example, consider the following text:

*Let's meet tomorrow for a drink.*

Suppose the user marks the fragment “meet tomorrow” in order to annotate it with a To-do task. As the original text is in a plain-text format, it will be converted to XHTML and a special tag will be added to mark the fragment (indentation added for clarity):

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:x="http://szyman.magres.net/2009/pim-html">
  <body>
    Let's <x:ref="ra82daex">meet tomorrow</x:ref> for a drink.
  </body>
</html>

```

The *TextFragment* and *PictureFragment* objects are standalone objects which can be treated as any other object – i.e. they can be linked and annotated. The visual presentation of these objects to the user is accomplished through special views (*TextFragmentView* and *PictureFragmentView*), which display only the marked text fragment and the marked image fragment, respectively. In case of the *TextFragment* object, modification of the text will result in

the fragment in the original text being modified, ensuring consistency. Moreover, as the *InformationFragment* objects contain a reference to the underlying unstructured information object, invoking the Object Details form on them will let the user follow a link to the source of the extracted fragment.

Extracting information fragments allows for the following use cases:

- Marking text in an email and annotating it with a to-do. The annotated text can automatically serve as the title of the to-do item.
- The user receives an email with an address of a friend. The fragment containing the address can be extracted and used as the value of an “Address” property for that friend’s Person object. A reference to the original email is maintained, so the origin of that information can be traced back later, if needed.
- The user finds a photo in a Picasa album, showing some of his friends. He may mark their faces as picture fragments, which will result in this photo being automatically linked to the respective persons.
- The user might also want to use one of the faces from the aforementioned photo as an emblem for a Person object. He just marks it and drags it onto the Primary picture property in the Person view.

#### 4.3.4. Summarizing information

An information object (e.g. describing a person) can contain a lot of information. Usually it is not desirable to display all this information at once, as this will lead to a cluttered view. Sometimes not all information about an object is available – for example, an email address might be available but not the name of the associated person. In order to display relevant information to the user, the PIM prototype can group information together and use substitute information when the exact one is not available.

This function is accomplished by the *PropertyViewHelper* class which uses the parent-child and equivalence relationships between properties when selecting values to be displayed in a view. For each property displayed by the view, the set of values for that property contains the values belonging to itself and its *related* properties, but not including values that belong to properties already displayed by the view.

The set of properties that qualify as *related* depends on the *InheritanceDirection* setting requested by the view. The following modes are possible:

- Descendants – Related properties include all equivalent properties and all descendant properties.
- Both – Related properties include all of the above and all ancestor properties.
- BothWithTree – Related properties include all of the above and all descendants of the ancestor properties.

In cases when only a single value for a property is needed, the following precedence rules apply:

- Descendants – First try the original property, then its children, then their children, and so on.

- Both – As with Descendants, and then the parents, then their parents, and so on.
- BothWithTree – As with Descendants, and then the parents, the descendants of parents, then the parents of the original property's parents, and so on.

The different use cases for the above algorithm are described below.

### Case 1

Let's consider a Person object with values assigned as shown in Figure 30, and let the relations between properties be also defined as shown.

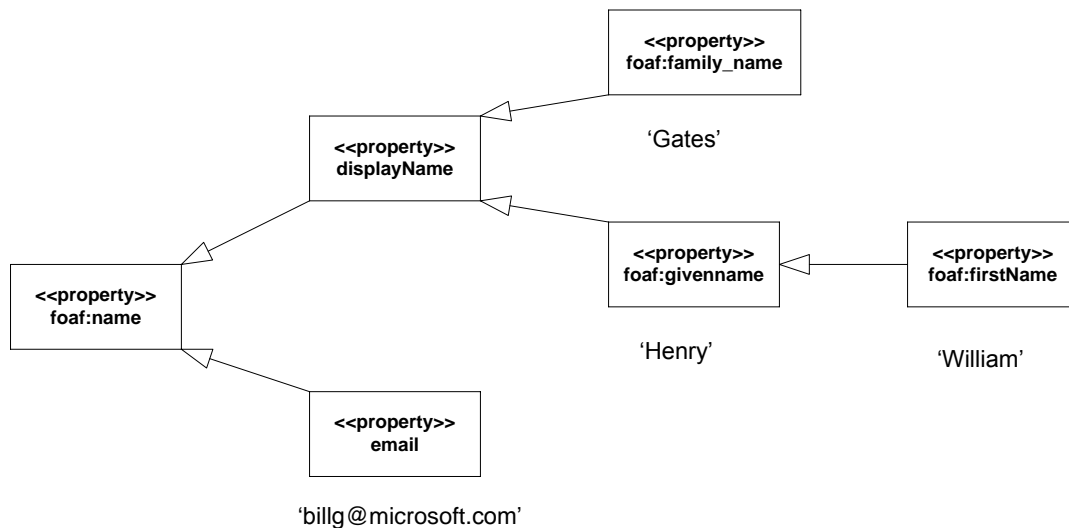


Figure 30: Example property hierarchy and values

A View displaying only the *family\_name* and *givenname* properties, without the use of the Helper, would only show the values “Gates” and “Henry”, respectively. This information is a little bit misleading, as *firstName* is actually a type of *givenname*. When displaying these properties in Descendant mode, the view will show “Gates” as the family name, and “Henry” and “William” as the given names. If all three properties were displayed by the view, the grouping of “Henry” and “William” together would not occur even with the Descendants mode active.

### Case 2

Let's consider the same Person object but in relation to displaying pictures. A person might have many pictures, and one of those pictures could be set as the primary picture (a picture that is to be displayed as an emblem for that person). The relationship between these two properties is the following:



Figure 31: Relations between picture properties.

Displaying these two properties can lead to a few interesting cases:

- When an emblem for a person is needed (e.g. when displaying the Person object in façade mode), then first the *primaryPicture* should be displayed, but if no picture is

defined as primary, then it is desirable to display one of the other pictures (as opposed to showing no picture at all). This is accomplished by displaying the *primaryPicture* property in Both mode.

- When displaying the Person object in a view that lets the user choose the primary picture, it is not desired to show all the other pictures as values for that property. On the other hand, any descendant properties of *primaryPicture* should be displayed. This is accomplished by showing the *primaryPicture* property using the Descendants mode.
- When displaying a list of all pictures for that person, it is desirable to include the primary picture in this list as well. This can be accomplished by displaying the *picture* property in Descendants mode.

### Case 3

Let's consider the same Person object but displayed using a façade view. Suppose the façade view for a Person type shows, apart from the primary picture, also a *displayName* for that person in BothWithTree mode. The PIM system will then try to find a single name to display by searching through the properties in this order:

1. *displayName*
2. *family\_name* or *givenname*
3. *firstName*
4. *name*
5. *email*

Even though *email* is not an ancestor of *displayName*, it is checked in BothWithTree mode as this can be potentially useful substitute information when no name is available.

#### 4.3.5. Tagging information

Tagging provides a quick and easy way of classifying information in a way that makes sense to a user. As such, tags are not directly supported in RDF. However, they can be easily introduced in the following way.

Each tag is an RDF resource belonging to the class *Tag* (see Figure 32). A tag is composed of just its name. The name is encoded in RDF by prefixing it with a specific namespace. Therefore the name itself must conform to the rules of creating RDF URIs (World Wide Web Consortium, 2004/c).

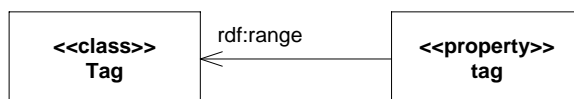


Figure 32: Tags in RDF

Tags are associated with Objects and Values using the *tag* property. For example, the following statements would be used to express that the specified picture is tagged with the words “biking” and “holidays”:

<code>&lt;file:///c:/Pictures/img_990123.jpg&gt;</code>	<code>pim:tag</code>	<code>tag:biking, tag:holidays .</code>
<code>tag:biking</code>	<code>rdf:type</code>	<code>pim:Tag .</code>
<code>tag:holidays</code>	<code>rdf:type</code>	<code>pim:Tag .</code>

Although tags in RDF are serialized in the same manner as Objects, they are not treated as Objects by the application.

Tags can be added to Objects and Values through the Object Details form. The user can search for tagged objects by typing tags as keywords into the Simple search box.

#### **4.4. External information**

In order to manage distributed information, the PIM system must be able to access information that is stored in external sources. Ideally, accessing external information through the system should work as if the original source of information was used. This means that any changes to the information on either end would be instantly visible on the other. For example, if an email arrives in an email client, it is also visible in the PIM application. If a friend on Facebook modifies his profile, those changes are reflected in the corresponding Person object right away. And if the user modifies a To-Do Task object, the change would be visible in Microsoft Outlook's To-Do list.

In theory, this could be achieved by directly accessing and manipulating data from external sources using some API in real time. The RDF store would then hold information about additional property values and links between objects from different sources – data that cannot be stored as part of the original information sources themselves. However, this would make querying information more complicated, as any query would have to operate on many disparate sources at the same time.

Furthermore, such a setup would require an API that supports reading and writing any information available in an information source, querying that information using a structured query language, and it would have to be efficient enough to support real time operation. Most existing software does not fulfill these requirements. For example:

- Microsoft Outlook does support reading and writing information, but querying tends to be slow (Cruysberghs, 2007),
- For Mozilla Thunderbird it is difficult to find information about any documented API, it is necessary to parse its mail storage files manually (Rea10),
- Facebook does have an API for reading and limited querying (FQL), but the latency of using a remote web service might be too big for practical applications.

It is thus necessary to copy information from the original sources into the RDF repository and then keep it synchronized. This problem involves choosing what external source of information to use and which information from it is desired, then reading it in order to create an object representation, and finally updating the original source with any changes to that information.

##### **4.4.1. Selecting information**

In order to work with external information, the user needs to choose which information is of interest to them and which should be incorporated into the information repository of the PIM system. Two approaches can be used here.

The user might choose a specific information object from an external source. This information object would be imported with only the most essential related data. For example, a user received an email and wants to annotate it with a To-Do Task. The user clicks on the email in the email

client and drags it to the PIM application window. The email is imported, along with information about the associated contacts (sender and other recipients).

The user might also decide to import all information available in a particular source. This will make it possible to use the searching and querying facilities built into the PIM system to find objects that are of interest.

Both approaches are valid, and it is up to the user to decide which will be useful in a particular situation. Some information sources might not support one approach or the other. Other practical considerations might apply:

- Importing a whole dataset takes considerably more time than importing a single item. The RDF repository used for storing this information will also grow in size. The user's information is effectively duplicated in the original source and the PIM system. This might not be desired, if the user does not plan to manage that information through the system.
- Importing information involves identifying duplicates and reconciling references, as described in Section 2.3.2. This automated process is not perfect. It might leave the user with a lot of objects that need correcting. The user's information repository will therefore end up in an state where the information is not well-organized. This might not be desired by the user.

#### 4.4.2. Importing information

Each source of information requires its own dedicated module that will read and write data to it. A module requires an ontology – a set of types and properties that will be used to map the data model used by the source onto RDF. The mapping should be an injective one, so that transformations from both the external domain to RDF and from RDF to the external domain are possible.

A module must also choose a naming scheme that will uniquely identify all objects imported from the external source. It must be possible to retrieve an object from the external source based solely on this scheme. If the source already has some inherent scheme for identifying objects, it can be used as a basis for an URI-based scheme, as required by RDF.

For each object in the external source, the module can then choose a suitable type, create an object based on that type, assign it an URI based on the naming scheme and set values for the object's properties. Primitive values need to be converted to a format supported by RDF, while references to other objects in the external source need to be corrected to point to the imported objects.

After this process is completed, an RDF representation of the current state of data is obtained. It could now be processed further to identify objects that already exist in the repository and merge the external information with them. For example, when importing an email contact with a certain email address, the RDF repository could be checked for Person objects having this email address. The email contact could then be marked as being equivalent to the Person object. However, this process is not intended to be performed by the prototype implementation. It is up to the user to manually reconcile any references.

#### 4.4.3. Synchronizing information

The module for accessing information from external sources needs to also take into account that both the information in the external source and the one in the PIM repository can change. These changes need to be propagated from one place to the other. The following strategy can be employed for this purpose:

Whenever an object originating from an external source is modified in the PIM system, the corresponding module is invoked to propagate the change to the external source. As any property can be associated with multiple values, which is not always the case with data stored in external systems (e.g. in Mozilla Thunderbird a contact can have only one given name and one surname), the module will have to select a single value that will be exported.

The module is also invoked periodically to scan for changes in the external source. This involves running the import process as described in the previous section, but comparing the resulting RDF data with the one stored in the repository. The following rules are then applied:

- Statements that exists in the external source but not in the repository are added to the repository.
- Statements that exist in the repository but not in the external source are deleted from the repository.
- Statements that exist in both places are omitted.

Instead of running the module periodically, a plugin for the external source could be developed that would monitor changes to that source and apply them to the repository.

#### 4.4.4. Exporting information

When information is exported from the PIM system to external sources as part of the synchronization described above, some special cases arise that are worth considering.

- New objects – An object can be exported to a source even if it does not originate from that source. For example, a To-Do Task created in the PIM system could be exported to Microsoft Outlook. In order for such export to be possible, the To-Do Task needs to be assigned an URI corresponding to an object in Outlook.
- Local objects – When exporting an object identifying a resource on the local disk (e.g. a locally stored image) to a source that is not located locally (e.g. Facebook), the resource might need to be uploaded to a remote server.
- Merged objects – If two objects originating from the same source are merged (i.e. defined as being equivalent), the module associated with that source might also merge them in that source (i.e. delete one object and leave the other one). However, this depends on the capabilities of a particular source – the remaining object must be able to hold the information previously contained in the deleted one. For example, a Thunderbird contact can hold up to two e-mail addresses. Therefore it wouldn't be possible to merge more than two Thunderbird contacts together.

### 4.5. Synchronization

People access and manage personal information from many locations: at home, at work, while riding a bus, or on holidays in the wilderness. Naturally, it is not always possible to have an on-



line connection to a single central repository of personal information. This requires that a copy of some of the user's information be available on a laptop or a mobile device that the user can take with them.

However, having independent copies of the user's RDF repository can lead to inconsistencies when one copy is modified while the other is not. Fortunately, it is easy to synchronize data between different repositories. All data in a RDF is stored in the form of triples. The repository supports only two types of write operations: add or remove statement. All modifications of existing values, such as changing the name of a person, involve removing the old triple and adding a new one. In order to track the changes to the repository, it is sufficient to log all such additions and removals. Such a change log can then be used to apply those changes to a different repository. Modification timestamps can be used as an automatic means of resolving potential conflicts.

The functionality of synchronizing data opens the possibility for more customizable ways of accessing and keeping user's personal information. For example:

- Central repository on a personal computer. The user can access the repository locally, connect to it remotely when Internet access is available, or synchronize data with mobile devices for off-line access. This however forces the user to keep his personal computer powered on and on-line for remote access to be possible.
- Central repository on a server. The user might decide to host the central repository on an Internet server and use other devices for connecting to it.
- No central repository. The user might treat all copies of their data repository on an equal basis and synchronize them when needed. This, however, will make some of the data unavailable until all repositories have been synchronized.

#### 4.6. Sharing information

Sharing information that is available through the PIM system can be accomplished in number of ways.

First, information can be already shared through the system's functionality of synchronizing data with external sources. One can use pictures from a disk folder to create a photo album and have them exported to a social networking site.

Secondly, data can be published using standard formats associated with some of the ontologies used in the system. For example, calendar information can be published using the iCalendar<sup>14</sup> format and contact information using the vCard<sup>15</sup> standard.

Furthermore, a dedicated web application can be created that would allow for publishing selected resources for others to see. The web application can implement some of the same views as were described earlier, but access to information would be read-only. Information presented through these views can be annotated using microformats (Micro09), so that the underlying semantic information can be exposed for other applications to process.

<sup>14</sup> <http://tools.ietf.org/html/rfc5545>

<sup>15</sup> <http://www.imc.org/pdi/vcardoverview.html>

The main PIM application must permit the user to choose which information is to be published through the web application. This can be accomplished on a per-object basis. Each object can be marked with a “publish” flag. By default, this will result in all the properties of the object that hold literal values to be published as well. Any linked objects will not be published automatically. The user can review the list of object’s properties and explicitly choose which ones will be published. Publishing a property results in all the values associated with that property (including other objects) being marked with the “publish” flag.

#### **4.7. Summary**

This section has described how a generic application data model could be constructed so that it would enable the user to define custom data structures on top of it. An overview of the types and properties initially defined for testing its use for personal information management has been given. The design also explains how the application’s model maps to an RDF database. Furthermore, the core components of the user interface, such as views and the context menu, which let the user interact with the data and modify the model have been described. The application’s facilities for filing and retrieving information were explained. Then, a method for incorporating external information into the system was described. The section concludes by discussing issues related to synchronizing personal information between different installations of the system and sharing information with others.

## 5. Implementation

This section presents how the system has been partitioned into major components and what functions do they realize. It also discusses how the exchange of information between these components is implemented. Then an overview of the user interface of the client application is given. It concludes by providing an outline of the functionality that was included in the prototype.

### 5.1. Three-tier design

The prototype for the PIM system is composed of three major components:

- The client – an application that is directly operated by the user.
- The server – an application that acts as a proxy for interacting with the RDF repository and external sources of information.
- The RDF Store – the triple store where all user data is kept.

The decision to create a client application that is separate from the server is not actually a requirement of the model, but was dictated by the following considerations:

First of all, the implemented client prototype could be one of many possible client applications. A web-based client or a client for mobile devices should also be possible to implement. Therefore the Server should contain the logic that would be independent of the particular client implementation.

Secondly, some of the tasks that the PIM system might perform could be long-running (such as importing data) or require constant monitoring of external resources (e.g. in order to check if new data is available, etc.) It is therefore reasonable to have a separate background process that would be tasked with those functions.

Finally, such a division was dictated by the choice of implementation technologies. Most of the RDF related technologies available today, such as triple stores and RDF extractors, are developed in Java. This suggests that the whole system should be developed in Java, in order to minimize the effort required to exchange information between Java and a different platform. However, Java does not offer the best tools with respect to user interface development. In this respect, Microsoft's Windows Presentation Foundation (Microsoft) proved to have a lot of features whose presence considerably simplifies the development of the user interface of the client. These include:

- the ability to nest user interface controls within other user interface controls,
- databinding – using an object as a source of data for a UI control, and
- XAML – the ability to define user interface elements in XML.

#### 5.1.1. System components

The architecture of the PIM system is given in the figure below:

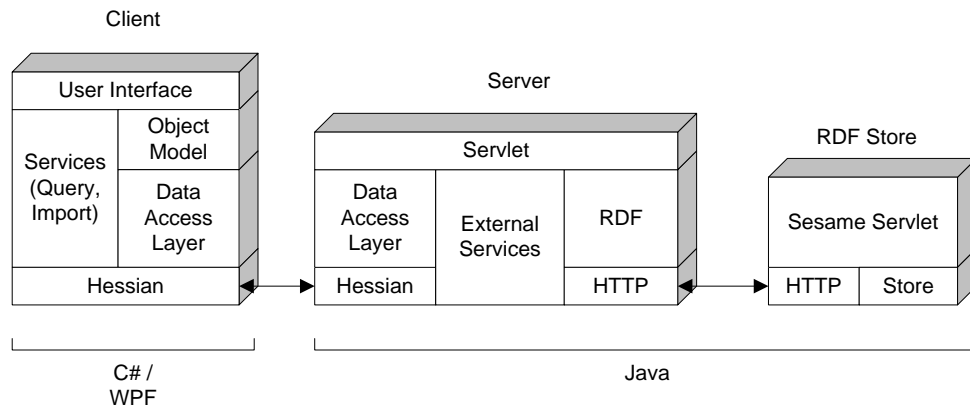


Figure 33: Architecture diagram of the PIM system

The user interacts with the system through the client. The client communicates with the server to retrieve, save and query data, and to invoke other operations, such as importing data from external sources. The server in turn communicates with the RDF Store, where all data is ultimately kept.

The client is implemented as a .NET 3.5 application:

- User Interface – responsible for interacting with the user as described in Section 4.2, implemented using Windows Presentation Foundation,
- Object Model – the object model on which the domain model is based, as described in Section 4.1.1,
- Data Access Layer – used for exchanging data between the Object Model and the Server,
- Services – an interface for invoking structured queries on the Server and to control services for importing data from external sources,
- Hessian (Caucho Technology) – a binary communication protocol available for many platforms, including Java and .NET.

The server is implemented in Java 5 as a servlet running under Apache Tomcat:

- Data Access layer – used for exchanging data between the Client and the RDF store, it implements the serialization and deserialization routines described in Section 4.1.2,
- External Services - used for importing data from external sources – this feature is accomplished through the use of the Aperture framework (Ape09),
- RDF – a HTTP interface to the triple store used in the system, called Sesame (Aduna).

### 5.1.2. Third-party components

This section describes the third-party applications, libraries and classes that were used in the implementation of the prototype. All of these pieces of software are either available under an open-source license or the author has expressed permission on their website for the use of this software.

#### Sesame

*Sesame is an open source RDF framework with support for RDF Schema inferencing and querying (Aduna). It provides a RDF repository which can be integrated in Java applications or run as a*

standalone servlet, a Java API for interacting with the repository, a powerful RDF query language called SeRQL, and an web console for interacting with repositories.

### *Hessian*

Hessian (Caucho Technology) is a protocol intended as a light-weight replacement for SOAP (World Wide Web Consortium, 2007). It was chosen for this project because an efficient way of transferring data between the client and a server was needed. Its main advantage over SOAP comes from the fact that the protocol is binary, as opposed to an XML encoding, which results in less data to transfer.

### *Aperture*

Aperture (Ape09) is a framework for extracting RDF data from various information sources, including file systems (Microsoft Office documents, iCal calendars, JPEG images, PDF documents, etc.), mail boxes (Thunderbird mail folders and address books, Microsoft Outlook objects, IMAP servers) and websites (Flickr, Delicious).

It is used as the base of the External Services component (Figure 33) of the server. Effectively all data that the prototype can automatically import from external sources is processed by this framework.

### *Facebook FOAF Generator*

As Facebook is an important source of information about a person's friends and colleagues, it was important for the prototype to be able to access the data located there. Unfortunately, the Aperture framework did not support extracting data from Facebook. However, Matthew Rowe has created a Facebook Foaf Generator (Rowe) application, which runs as any other Facebook application and lets the user download an RDF file containing information about their friends expressed using the Friend-of-a-Friend ontology. This file can then be imported into the repository and the information it contains can be accessed using the client.

### *WPF libraries*

This is a list of classes and libraries that were used in the implementation of the client for specific GUI functionality:

- Avalon Controls Library<sup>16</sup> is a set of WPF controls that were used in the client. In particular, the following controls were used as part of the view for displaying date and time values: DateTimePicker, TimePicker, DatePicker.
- DragCanvas class<sup>17</sup> – A canvas that allows for dragging of the elements it contains. It was modified for use as part of the view for displaying Notebook Pages.
- AutoCompleteTextBox class<sup>18</sup> – A text box with an autocomplete feature. It was modified for use as part of a control for choosing properties and types.
- CloseableTabItem class<sup>19</sup> – A tab control with a closing button. Used as part of the client application's main window.

<sup>16</sup> <http://www.codeplex.com/AvalonControlsLib>

<sup>17</sup> <http://www.codeproject.com/KB/WPF/DraggingElementsInCanvas.aspx>

<sup>18</sup> <http://www.codeproject.com/KB/WPF/WPFAutoCompleteTextbox.aspx>

<sup>19</sup> <http://geekswithblogs.net/kobush/archive/2007/04/08/CloseableTabItem.aspx>

- WpfRichText project<sup>20</sup> – A control for editing rich text, containing standard text-formatting buttons (copy, paste, bold, italic, etc.) Used as a part of the RichTextView.

## 5.2. Exchange of information

Dividing the system into a client and server parts, while providing certain benefits, also introduces additional complexity associated with transferring and synchronizing data between the tiers. The prototype implementation uses the following approach to solve this problem.

### 5.2.1. Retrieving objects

The client maintains its own set of in-memory Persistent objects (see Section 4.1.1) which are independent from the server. At any given time there exists at most one instance of any object, as identified by its URI. This means that (e.g.) many views will operate on the same instance of an object at the same time.

Data can be created, modified and deleted on the client without any interaction with the server – all using the client's in-memory state of the objects. If data from the repository is needed, it can be retrieved via the server. It is possible to retrieve a full set of data associated with an object, only a specific part of an object (e.g. certain properties) or only a list of object references (without any data).

In order to retrieve a full object, the client sends a request to the server for that particular object as identified by its URI. See Figure 34:

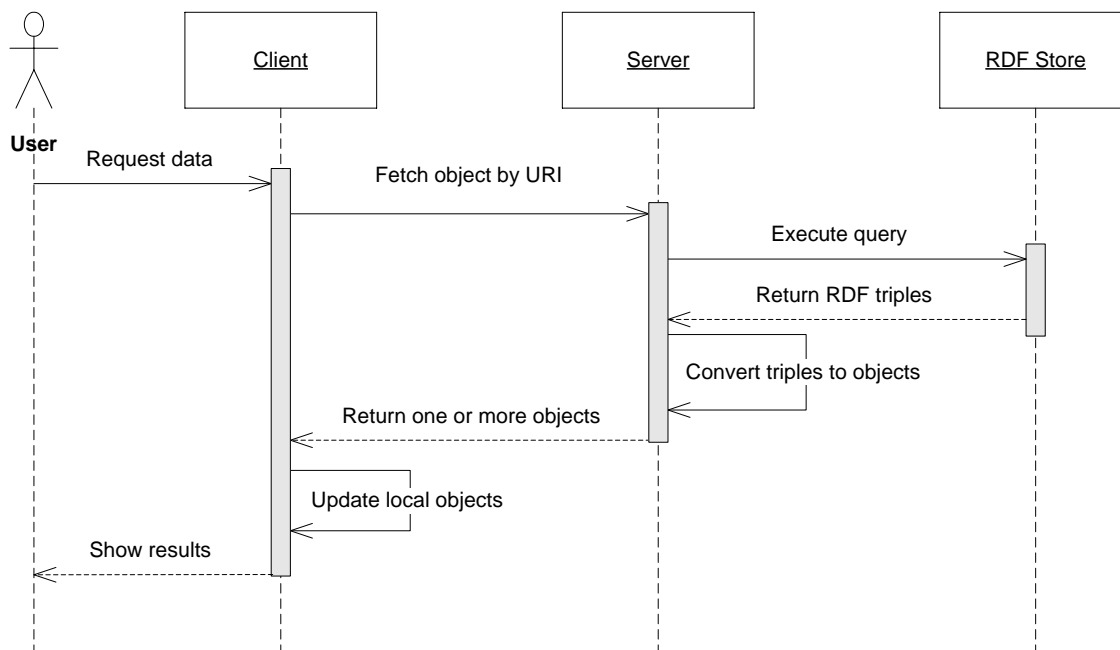


Figure 34: Sequence diagram of requesting data from the server

The server, in turn, executes a query on the RDF Store to fetch all triples whose subject is equal to the requested URI. Using the resulting triples, the server constructs an object representation using the rules described in Section 4.1.2. Such an object is then transmitted to the client.

<sup>20</sup> <http://www.codeproject.com/KB/WPF/wpf-richtexteditor.aspx>

The client receives the object and needs to incorporate the data it contains into its own in-memory set of objects. This step is needed due to the following reasons. First of all, the requested object might already exist on the client side and contain some data – it might have been retrieved earlier with only partial data. Secondly, the object obtained from the server contains references to other objects and these references need to be updated to point to objects in the client's memory. Converting a server object to a client one involves going through all the properties of the object, checking if the local copy already contains such data, and if not, copying it while correcting any references.

### 5.2.2. Querying for objects

Apart from fetching a single object from the server, the client can also execute structured queries on the server. A special Query class has been created, which serves as an object representation of the query, making it independent of the particular query language (such as SeRQL or SPARQL) which might be used by the repository. A query contains a list of expressions joined by the logical AND or OR operators. Each expression can introduce constraints on the subject, predicate or object part of RDF statements through the use of comparisons (e.g. LIKE or EQUALS). For example, the following query fetches all *Trip* objects that have a *location* property whose value is the string "Norway" in English or "Norwegia" in Polish:

```
Query q = new Query();
q.add("pim", Repository.PIM);
q.where("rdf:type", "pim:Trip");
Expression expr = q.where("pim:location").orExpr();
expr.andExpr().label().like("Norway").language().like("en*");
expr.andExpr().label().like("Norwegia").language().like("pl*");
```

Such an object representation would be converted to the following SeRQL query by the server:

```
SELECT DISTINCT QName
FROM      {QName} rdf:type {rdfs:Resource},
          {QName} rdf:type {pim:Trip},
          {QName} pim:location {q0}
WHERE     ((label(q0) LIKE "Norway") AND (lang(q0) LIKE "en*")) OR
          ((label(q0) LIKE "Norwegia") AND (lang(q0) LIKE "pl*"))
USING NAMESPACE
      rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
      rdfs = <http://www.w3.org/2000/01/rdf-schema#>,
      pim = <http://szyman.magres.net/pim/>
```

One thing to note about the above query is that it will return a list of URIs of objects that match the specified constraints (we are SELECTing only the *QName* variable, which is the subject of all statements). It is possible to add more variables to the SELECT part, which will result in additional information being retrieved from the repository. In this way, only part of the information contained in objects can be retrieved.

Retrieving only a partial set of information is important when displaying a list of many objects. Completely loading all the objects on the list is usually not necessary, as only part of the data will be displayed to the user, and is also time consuming.

### 5.2.3. Committing modifications

In order to propagate the changes to the in-memory data model back to the repository, the client uses the Unit of Work pattern (Fowler, 2002 p. 184). The client maintains a list of objects that have been modified – each object is assigned one of the following state values:

- Clean – the object is at least partially retrieved from the repository and no changes have been made to it,
- New – the object does not exist in the repository,
- Modified – the object has been modified,
- Deleted – the object has been marked for removal from the repository,
- DeletedNew – the object does not exist in the repository and has been marked for deletion.

Committing data to the repository is done periodically by the client. The user can modify a certain set of objects and after some time a separate thread will pick them up and send the modifications to the server. This process is outlined in Figure 35:

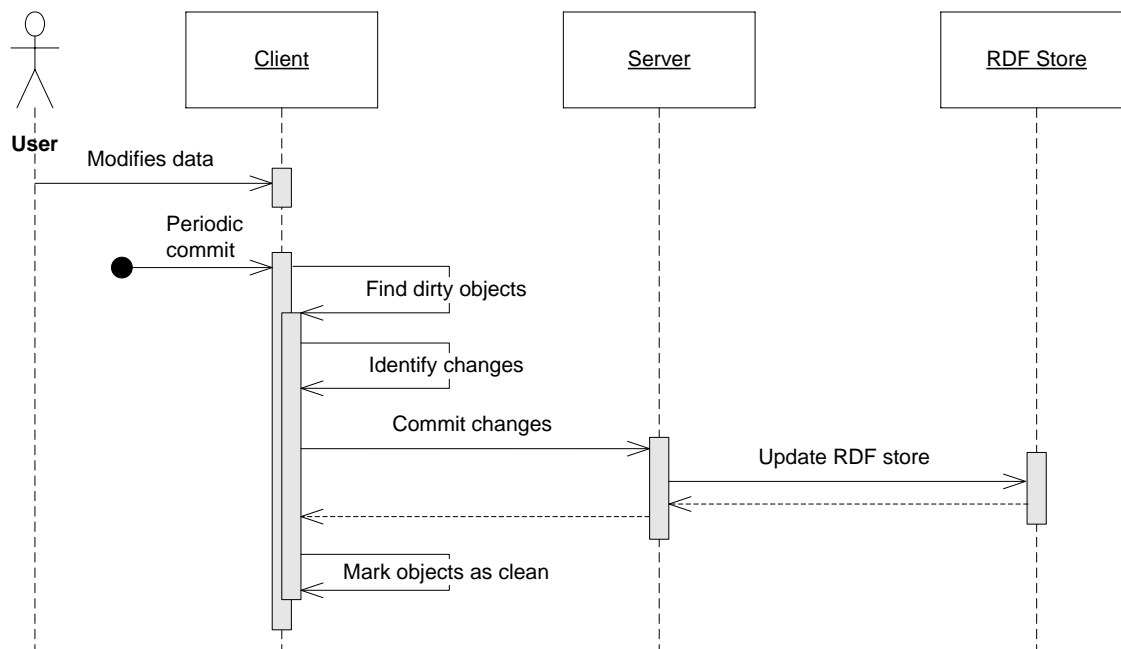


Figure 35: Sequence diagram for the data commit process

When the commit process starts, the client identifies objects which have pending changes (i.e. which are in the Modified or Deleted state) and creates a set of Data Transfer Objects (Fowler, 2002 p. 401) corresponding to the original objects, but carrying only the original and new values for the modified fields. For example, consider a `Value` object that originally contained the text “hello”. The user modified that text to “hello world”. The resulting Data Transfer Object (of type `TransferValue`) will contain both “hello” and “hello world”. Such an approach is needed as in order to replace an existing value stored in the RDF repository, the old value has to be removed and a new value needs to be added. This can only be accomplished if both the old and new values are known. This is in contrast to standard relational databases, where replacing a value involves just specifying a new value for a given field.



As a further consequence of how the repository operates, it is necessary for the client to store all the original values corresponding to all the values that were modified. Most of the user information contained in the objects is stored in lists or sets – for example, an Object contains a set of Properties, which in turn contain a set or list of values. The Data Transfer Objects for lists and sets must contain the indexes and/or values of the elements that were added or removed. Effectively, the contents of the Data Transfer Objects can be compared to a result of a difference algorithm of two pieces of text.

Once the Data Transfer Objects have been created, they are sent to the server. The server just needs to translate them into the addition and removal of appropriate RDF statements. After the changes have been committed, the client can mark the modified objects as Clean.

### 5.3. User interface

The user interface of the client application consists of a single main window (see Figure 36). The window is divided into three parts.

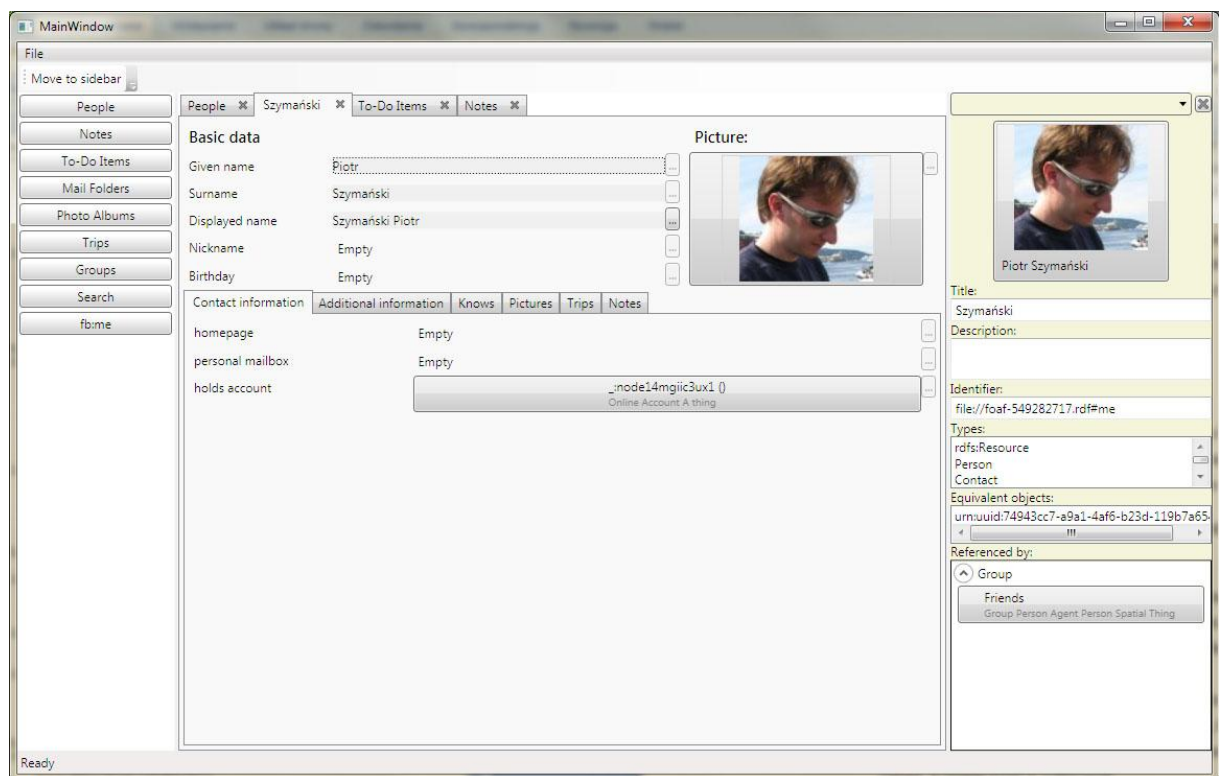


Figure 36: Main window of client application

The left pane provides a set of buttons that open predefined locations in the application. They include links to views listing all objects of a specific type (Persons, Notes, To-Do Items, etc.) and to a generic search function. The right pane (called “the sidebar”) usually contains the Object Details form, which displays some overview information about a chosen object.

The middle pane is the place where all (object- and collection-) views are being displayed. When a user clicks on an object displayed through a façade view (e.g. as one of the elements in a collection), it will result in the object being opened in a new tab. Multiple tabs can be open at the same time allowing the user to work with different objects and collections simultaneously.

Moreover, the user can click the “Move to sidebar” button to move the active tab to the right pane (see Figure 41 in Appendix A). In this way, it is easy to drag-n-drop objects between different collections as two different views can be active side by side.

Figure 36 illustrates how a view assembled from other views looks like. The active tab in the middle pane displays the custom view for the Person type. This view is split into three regions: Basic data, Picture and set of tabs containing more detailed information.

The Basic data section lists properties that provide an overall description of the person (e.g. given names, surname, etc.) Each property is associated with a collection of values. Therefore a collection view is used to display those values. In the illustrated case, the Alternative values collection view is used. It displays a single value from the collection at a time, but the user can click on the button (“...”) next to the value to switch to another one. Values can be added and removed from the collection using the context menu. The *Given name*, *Surname* and *Displayed name* properties hold string values. Therefore, the collection view uses the text editor view (see Section 4.2.1) to render them.

The Picture section also uses the same Alternative values collection view to display the primary picture associated with a person. However, instead of a text editor view, an image view is used. To add more pictures to this collection, the user can just drag it from a disk folder or from a web browser onto this collection.

The right pane shown in Figure 36 contains the details for the selected Person object. In particular, it displays all other objects that reference this one. Here we can see that this person belongs to a group called “Friends”. The referencing objects are displayed using standard façade views. This means that it is possible to interact with them as with any other object displayed in the application. In particular, one can invoke a context menu on these objects.

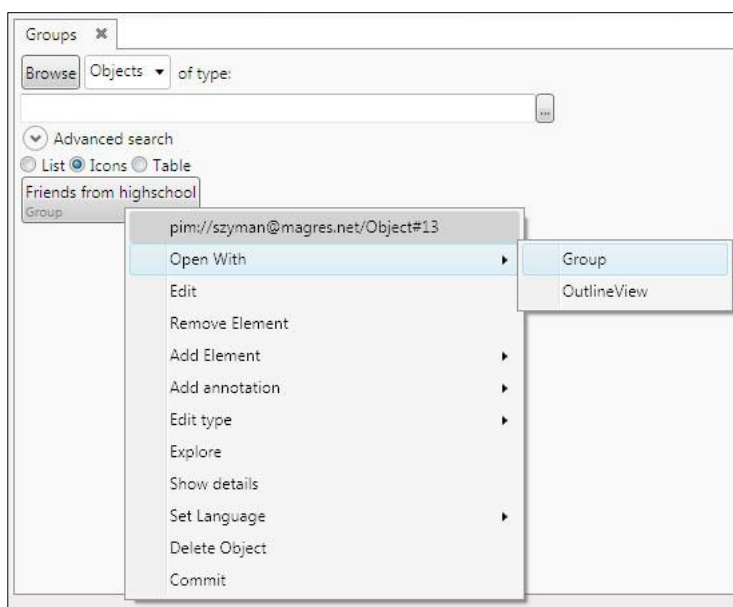


Figure 37: Client UI - context menu

The context menu can be invoked by right clicking on any view in the application. Figure 37 shows the actions that can be executed when clicking on a group of elements.

More illustrations of the prototype's user interface can be found in Appendix A.

## 5.4. Limitations

Most of the functionality described in the Design section was implemented in the prototype. Some designed functionality was not implemented due to time constraints. In some cases, there exists similar functionality to the one that was left out which can be used as a basis for evaluation.

The following functions were **not** implemented in the prototype:

- Data model (Section 4.1)
  - Only the Bag collection was implemented, and not the List collection. As a result, all functionality related to the List collection has been left out of the implementation. The impact of this is minimal, as the only difference between a Bag and a List is the ordering of elements.
  - The Range property of a Property is limited to a single Type, while the design requires that it contain a set of Types. Again, the impact of this is minimal. Range serves as a hint for the type of values that can be associated with a property. In most cases, a single type is sufficient.
  - Value objects (i.e. literal values such as strings, dates and numbers) can only be persisted as an *attached* value. This means that it is not possible to annotate and create links to values as one can annotate and link to normal objects.
- Presentation layer (Section 4.2)
  - The NumberEditorView was not implemented. Numbers can still be edited as normal text strings.
  - SimpleCollectionView does not support grouping objects.
  - Timeline view was not implemented.
  - It is not possible to open the original application from which an information object originated.
  - It is not possible to open a collection in a separate view. A similar function works for objects.
  - The “Replace View...” action of the context menu was not implemented.
  - The editor for custom views was not implemented. It is still possible to create custom views by manually editing XAML files.
- Finding and managing information (Section 4.3)
  - Searching for objects associated to a value through not more than N intermediate objects.
  - Extracting unstructured information from images is not implemented. It does however work for text strings.
  - Tagging information.
- External information (Section 4.4)
  - The prototype is limited to importing information from sources supported by the Aperture framework.
  - Synchronizing information with external sources has not been implemented.
- Synchronization of information between multiple installations of the system (Section 4.5) has not been implemented.

- Sharing information, as described in Section 4.6, has not been implemented.

The prototype implementation also requires work in terms of stabilization of the existing features. Some problems that were discovered include (they are discussed later in the Evaluation and Discussion chapters):

- Loss of user data related to concurrency issues.
- RDF triple store is slow when updating data.
- Insufficient inferencing support from the RDF repository

## 5.5. Summary

This chapter covered the details about the prototype implementation. The division of the system into separate tiers was discussed together with the rationale for choosing certain implementation technologies. An outline of the third-party components involved in the system was given. Then, the process of exchanging information between the tiers was explained, which included how objects are retrieved, queried and how modifications are saved to the RDF repository. Afterwards, an overview of the client's user interface was presented. Finally, the status of the implementation, including a list of features that have not been included in the prototype, was given.

## 6. Evaluation

This section starts by explaining what are the inherent problems in evaluation personal information management systems and then tries to assess the model and its implementation by examining the ability to realize scenarios, by discussing the benefits and drawbacks of certain functionality and by investigating the potential for integration with new technologies.

### 6.1. Experimental difficulties

Evaluating the proposed model and its prototype implementation is a difficult task. Such an evaluation can be performed through user experiments – observing how real people solve their information management problems with the help of this system. However, there are a number of problems associated with this approach that have led me to consider a different one. These problems are described below.

First of all, the prototype does not implement all the functionality described in the Design section. Due to time constraints, it was not possible to create such an extensive prototype. It is therefore not possible to fully evaluate the model based on the prototype itself. Some of the features that were left out often have corresponding similar features that were implemented, but not all of them. For example, while annotating picture fragments can be evaluated on the basis of annotating text fragments, there is no similar feature to the view editor.

Secondly, the prototype needs more work in terms of stabilization, fixing deficiencies in the implementation and in third party components before it would be suitable for user testing. For example, it was discovered that the RDF repository becomes extremely slow when updating data after a certain number of triples have been entered into the system. This problem is related to the implementation of that 3<sup>rd</sup> party component. Even though the updates are performed in a background thread and do not directly lock up the system, the problem manifests itself in different, perhaps even more serious way. While a chunk of data is waiting to be written to the repository, any data it contains will not be returned when a user performs a query. In turn, it might seem to the user that the information he or she just entered into the system has disappeared.

Moreover, the user interface of the prototype is quite primitive. Only functions that were directly related to the model of the system were implemented, but not necessarily in a user-friendly way. For example, the prototype does allow the user to formulate a query by specifying a list of constraints, but one might need to work with URI addresses for choosing objects. This can be difficult to understand for the user. A polished user interface could offer a drag-n-drop functionality instead.

Due to the above reasons, it would be difficult to conduct a user test with real people that would evaluate the underlying model. The test results would be skewed by stability problems and user interface deficiencies.

Additionally, there are some issues related to testing personal information management systems in general. In particular, it is difficult to test these systems in an artificial environment, such as a laboratory experiment. One of the reasons is that PIM happens over time. *The effectiveness of an action to file information, for example, can't be assessed without also looking at later efforts to retrieve this information* (Jones, 2005 p. 53). But by forcing the user to file information and then

retrieve it half an hour later we cannot really test this effectiveness. This is because *humans have memory or partial memory of all personal information that is kept* (Larsen, 2005 p. 22).

Sometimes a significant amount of time needs to pass before it will be possible to examine if the system makes it easier to retrieve information based on that partial memory.

Another such problem is related to the fact that it is difficult to use simulated personal information instead of real one. Being personal, the user has at least partial memory of that information. However, the user might be not willing to provide their own information due to privacy concerns. On the other hand, using simulated information will eliminate memory - one of the essential aspects of personal information management (Larsen, 2005 p. 183).

Based on the above considerations, I will try to evaluate the model and the prototype without conducting user tests. Instead, I will assess its support for realizing the scenarios described in Section 3, try to identify the benefits and drawbacks of specific functionality and consider the model's capabilities to interact with new technologies and approaches to keep and share information.

## 6.2. Scenarios

### 6.2.1. Working on a project for a customer

This scenario describes the difficulty of finding information that is scattered across multiple places and entities. Although the prototype implementation does not fully realize this scenario (due to the effort required to implement the needed user-interface functionality), it does give the tools necessary to effectively manage this information in a similar way. In fact, this also proves the flexibility of this model. With more coding effort, a specialized view could have been created that would let the user manipulate the information more effectively in a visual way. However, the underlying structure would not change. The user is free to create new types and properties, and also new views (though these need to be composed of existing ones).

The prototype supports importing e-mail messages from an IMAP server, which makes it possible to connect to email accounts operated by many providers, including Gmail. The external data is converted into e-mail message, e-mail folder and contact types, as described in the Foreign types section. The user can thus browse through an e-mail folder in the PIM system to find the latest messages, which resembles how such an action is performed in a normal e-mail application.

When a desired e-mail is found, the user can annotate (e.g. with a To-Do task) its text to mark tasks that are requested by the customer. By doing this, the user identifies the relevant portions of the email, which require some additional action to be taken, while the rest of the email can be ignored. This reduces the amount of work required when the user has to revisit this email message in the future – all important details have already been highlighted. Moreover, they can then be viewed as entities that are separate from the email – e.g. a list using the Object Details form. By extracting unstructured information, the user reifies it, allowing it to be treated in the same way as any other information in the system. On the basis of that I argue that it provides important value to the user.

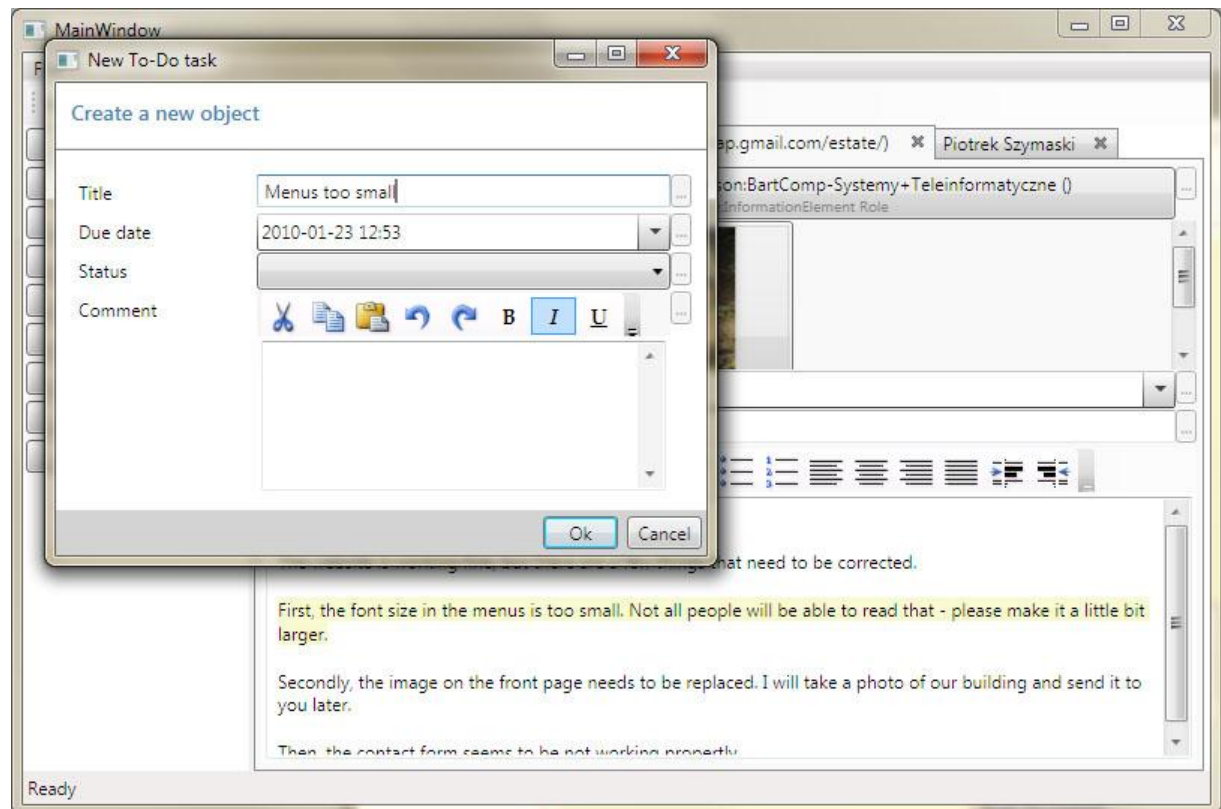


Figure 38: Realization of the "Working on a project..." scenario using the prototype

Each task marked in an e-mail message can be, in turn, annotated with a note. The note can function as a central place where all information related to the task is stored. The user can drag disk files, documents, other emails, web links, etc. onto the note as needed. They can also enter their own comments as part of the note. Using this approach has several benefits:

- While a To-Do task is pending, this information is classified as ephemeral or working. It can be reached from the list of pending to-do tasks. But once it is completed, it does not disappear. It can be retrieved later by following links.
- It is easy to find where the notes and resources associated with a problem being worked on are kept. The user can start with the e-mail message and follow links to find the desired information. In fact, if the user can remember just a single object involved in the note, it can be used as a starting point to find the note itself.
- The context menu for external objects allows the original application to be invoked. The user can quickly reach those objects from the PIM application, allowing for more specialized functionality to be contained in the original applications and not duplicated in the PIM system. Unfortunately, it is not easy to implement the reverse functionality.

The scenario also describes how similar issues that appear in multiple emails could be merged into a single one. As the prototype implementation does not support tags, the user would be limited to manual searching for locating those similar items. Once found, however, they can be easily merged.

Such merging of issues significantly reduces the user's amount of work. Without such an ability, the user could either leave the two issues as separate entities or merge them manually. The first



solution leaves the user with two or more objects that represent the same thing. Information will be scattered between them and finding something will require going through all of them. The second solution would require the user to manually copy information from one issue to another, replace all existing references to issue A with issue B and then delete one of them.

### 6.2.2. Semantic notebook

This scenario describes how small fragments of information can be linked and annotated to keep the user organized. It can be almost fully realized using the implemented prototype.

The user can create a Notebook Page object and place some text onto it. Other objects, such as Persons, can be also placed in the note as well. Just by placing objects in the note, all these elements become linked with each other through their association with the Notebook Page. They are grouped together not only visually, but also semantically. This means that the user can view the relationships of one of these objects to discover the remaining ones. In this way, the problem of filing a note in the appropriate folder or some other container is partially solved – the user does not have to put the note in any particular place. It can be always found later by following associations – if the user remembers just one of the objects that were contained in the note (e.g. a particular person), they can use it as a starting point either for traversing links from one object to another or for doing a search and specifying this object as one of the criteria. This approach is in line with how the human mind operates (Bush, 1945) in terms of associating information and makes use of the fact that a person has partial memory of that information (Larsen, 2005 p. 183).

Furthermore, the information contained in the note is not locked in its standard graphical representation (DesktopView), as is the case with (e.g.) Microsoft OneNote. It can be also viewed as a container that holds some objects, without regard for their X and Y screen coordinates. Any CollectionView can be used to browse these objects, increasing the chance that one of the available views will match what the particular user is looking for.

Continuing with the scenario, the user selects a fragment of text that was pasted to the note and annotates it with a To-Do Task. A To-Do Task created in this way will appear in a list together with other tasks, but is also linked to the text fragment and, indirectly, to the note. If a reminding functionality was implemented in the prototype, by opening a pop-up window containing information about a pending task, the user could go directly to the note. Compare how such a use case could be realized with one of the traditional approaches – e.g. in Microsoft Outlook: It is not only impossible to create a link to a fragment of text, but even linking to the whole note is not supported. Instead, a user might create a to-do task and put a *copy* of the whole note in it. This means that now two separate notes exist, which happen to initially have the same content, but problems will quickly arise once the user decides to modify one of them. By using the approach employed in the prototype, such inconsistencies are avoided.

The scenario also describes how a picture gallery containing images from different sources can be created – the user finds their friends' albums on Picasa or Facebook and drags images from there into his or hers own album, combining them with the user's own pictures. The prototype supports pictures imported from the Web and from a local disk. This permits the user to freely combine these images into a Photo Album, without paying special attention to their original source – whether an image is dragged from a disk folder or from a browser window, the application treats it in the same way. In this way, the strict boundaries between applications are



loosened up – the information available through a computer can be accessed in a more unified way.

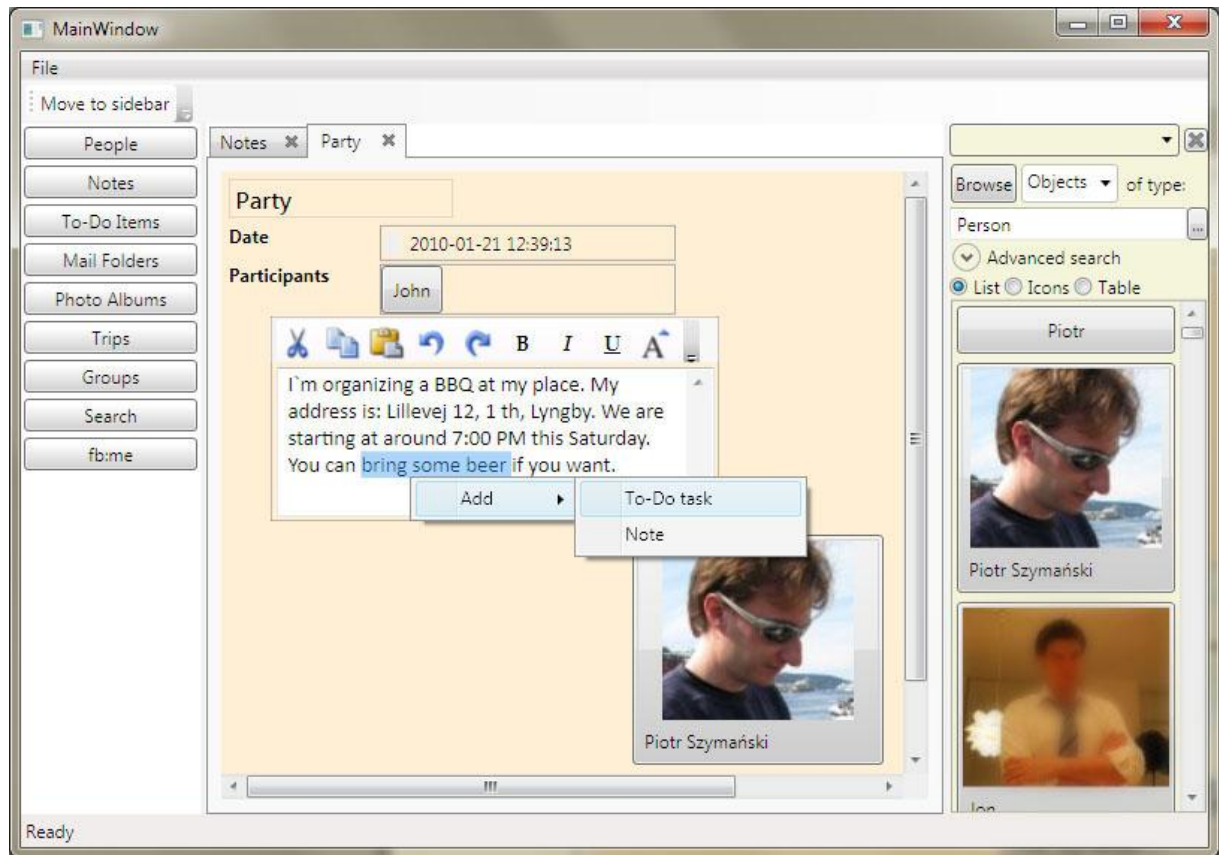


Figure 39: Realization of the “Semantic notebook” scenario using the prototype

The user might also combine the albums in an even quicker way. If the albums from Picasa and Facebook were imported into the PIM system (which is not implemented in the prototype), the user could simply drag one album onto the other and specify that they are equivalent. This would result in the system merging the contents of those albums together and instantly creating a single one holding all the pictures.

In addition to creating a note about a current event – a form of ephemeral information – the scenario also shows how the user’s long-term database can be updated in the process. In a similar way as fragments of text can be annotated and linked, the user could mark portions of images and link them to existing persons. These photos will then appear when viewing the associated Person objects. Moreover, the scenario explains how a fragment of text representing a person’s address can be associated with a person by dragging (this functionality was not implemented for text fragments, but does work for other types of objects). The address will be visible in the person’s profile from that point on. By reducing the amount of effort required to classify information, thanks to operations such as dragging and dropping, the user will be able to find this information in the future.

### 6.2.3. Inviting people

This scenario describes how combining contact information from different sources helps in finding the most up-to-date one, and how the process of inviting people can be improved. The scenario can be mostly realized using the implemented prototype.

The user starts by creating a Notebook Page and adding custom properties to it. This function gives the user the possibility to structure information in a way that is needed to fulfill a task. Even though the Notebook Page type does not contain these properties, the user is not limited or restricted by the properties that a given type provides. In this way, additional user-specific information can be stored in a structured way, as opposed to putting it in a big textual “Description” field which is often the case with other applications. Such a design gives the user a lot of flexibility.

The scenario then describes how the contact information from external sources can be used to find the best way to contact a person. In this respect, the system combines information distributed among various sources and gives the user a central place where more information can be stored. Thus it is possible to quickly get an overview of what is known.

One of the functions that cannot be realized with the current design is annotating an association (i.e. a complete statement, composed of a subject, a predicate and an object). It is only possible to annotate a particular object, and not the object in the context where it was used. This limitation comes from the fact that there is no support for reification of statements, i.e. it is not possible to treat statements as other objects in the system. During testing of the prototype this was not identified as a very serious drawback, however as the scenario shows, there are some use cases that require it. It should not be difficult to implement such a function in the future.

## 6.3. Functionality

### 6.3.1. Integrating information

One of the problems that are a consequence of the distributed nature of personal information is information fragmentation and duplication. Different and unrelated formats, applications and Internet services that are used for keeping information result in the information being scattered between multiple places.

With existing applications, storing information is associated with certain risks. First, most applications appear as data silos. They enable the user to work with some specific kind of information, but do not interact with the outside world. Information in these applications might be locked in. It might be necessary to duplicate it, if it needs to be reused in another application, and this quickly leads to inconsistencies when the user updates information in one place but does not do it in others. Secondly, with every new application that one uses, the old problems and dilemmas for the user are getting worse - Where did I put my old information? Where to put new information?

The system proposed in this paper eliminates these problems by functioning as *the* central place where all the user’s personal information about a particular subject can be accessed and modified. It is not *yet another* program for storing personal information. Its distinguishing feature is the capability to import and synchronize information with other applications. Although the

prototype implementation is not capable of exporting information back to the original sources, being able to read from some existing sources already provides great value. Information fragmentation is reduced by being able to access existing data, and duplication can be eliminated by using the equivalent objects feature.

Elimination of duplicate information can only work if it can be correctly identified so that only the redundant one is removed. The model does not use any sophisticated methods for this purpose, such as the ones described by (Dong, et al., 2005). Instead it proposes a simpler approach – the user manually identifies entities that represent the same thing, while the system uses the relationships between different ontologies (i.e. type and property similarity) to merge them together. It has to be noted that this approach has some limitations: it requires manual work and only the first level of data associated with an object is merged. Consider two person objects ( $P_1$ ,  $P_2$ ) and two document objects ( $D_1$ ,  $D_2$ ).  $P_1$  is the author of  $D_1$ , and  $P_2$  is the author of  $D_2$ . Assuming that  $P_1$  and  $P_2$  are the same, and so are  $D_1$  and  $D_2$ , merging  $P_1$  and  $P_2$  will not automatically merge  $D_1$  and  $D_2$ . The user will have to first merge the persons, then merge the documents. However, when working with real data (such as Person objects imported from the Thunderbird Address book and from Facebook), most data existed in text form which was merged automatically and the described limitation was not a problem.

The usefulness of specifying equivalent objects was quickly verified when working with real data. The prototype has been tested with external information coming from IMAP servers (e-mail messages, contacts), Facebook (profiles of friends), Mozilla Thunderbird's Address Book (profiles of e-mail contacts), and images (JPEG) and documents (Microsoft Word, text files) coming from both the local disk and the Web. In particular, after importing Thunderbird contacts it became apparent that there exist many Contact objects with different e-mail addresses, but representing the same person (this is how Thunderbird's Address Book works – each email address is usually stored as a separate contact). Merging these objects and the Facebook profile for that person together resulted in that information being represented by a single entity. All information related to a person was then available in a single place, and not scattered between multiple entities.

The usefulness of the prototype is limited by the fact that it is not capable of accessing all of the information that the user might be interested in. For each source of information there must exist a dedicated module that will link it to the PIM system. This is a serious drawback of the system, but one that cannot be easily overcome. It would require all applications to agree on a single standard format for exchanging information, which is unrealistic due to many reasons, ranging from technical to political. Thus the user will have to content his or herself with the limited set of options that are available. Fortunately, many applications do support some form of external interoperability. It might be therefore possible, maybe indirectly, to import this information into the PIM system. Moreover, information storage in the system is “additive” – one can always add new information and merge it with existing one. The user does not have to fear that by storing some information in the PIM system and some in an unsupported application will prevent them from combining it in the future. This is also one of the purposes of defining equivalent objects – to merge together different pieces of information that all describe the same entity.

### 6.3.2. Structured and unstructured information

One of the problems with existing applications is that they restrict the user regarding the amount and type of information that they can keep in a structured way. For example, Mozilla Thunderbird's Address Book has a fixed set of fields for storing information about contacts. If one would want to store some additional information, such as a list of bank account numbers, the only option is to use a generic "notes" field.

The proposed system eliminates this restriction by allowing the user to store any amount of additional information and of any type in a structured way. An existing information object (such as a contact from Thunderbird's Address Book) can be extended with additional properties. If a desired property does not exist, the user can create a new one and specify the type of data that it should hold (text, numbers, dates, images, etc.) And every property in the system can have multiple values.

The system also provides support for describing the contents of unstructured data. A lot of information that people work with on their daily basis exists in an unstructured form (i.e. it is only intended by human, and not computer, consumption): e-mails, documents, pictures, audio and video files, etc. This information cannot be queried in a robust way and it can only be viewed as a whole – the internal structure, if there is any, is not visible to the computer.

The model proposes a method for marking fragments of text and images and extracting those fragments as standalone entities. In this way, any underlying structure within a piece of data can be identified by the user. These pieces can then be linked, annotated, queried, etc. in similar ways as "normal" structured information. In this way, the user can manage their information more thoroughly.

### 6.3.3. Linking and finding information

Linking and finding information are closely related. People associate some things with others in their minds and through those associations they can recall them (Bush, 1945). It is therefore an important feature of any information management system to be able to link information. However, it is not easy to link information available through the computer. Every application uses a different filing scheme, and there is no common way of addressing and accessing different pieces of information.

The proposed system provides this functionality. It can address and access information from different sources. It allows links to be created between any two information objects, regardless of their type and origin.

In general, the prototype has experimented with finding information in the following ways:

- *Listing all objects of a particular type.*

The prototype UI provided links to lists of objects belonging to some predefined types (including Person, Notebook Page, To-Do Item, Mail Folder). This method proved to be the quickest one, provided that the number of objects was rather small. With increasing amounts of information, it is often required to filter the results by specifying some additional criteria. E.g. after importing friends from Facebook, it was necessary to filter the list of people by name in order to quickly find the desired person.

- *Following links between objects.*

The user can start at one object and then view a list of all objects that reference the selected one, together with the ones that this object references. Thus the same principle as with recalling thoughts by association can be employed for finding information stored on the computer. (Alvarado, et al., 2003) suggests that this is the preferred method of finding information for many users. However, once the number of links is large, this process also becomes more time consuming. This effect can be counteracted by filtering the list of linked objects according to some criteria.

A special case of following links is finding an annotated object. In this case the annotation (e.g. a note or a to-do task) usually has only one link leading to the object it annotates. Therefore this object can be retrieved very quickly.

- *Searching for objects based on some criteria.*

The user can specify search criteria by describing the characteristics of the object they are looking for. It was, however, observed that it is not always easy to specify a type of an object being searched for or a property that some value might be assigned to. This is due to the large number of types and properties that originate from the ontologies used in the system.

It is worth noting that all of the above methods allow for seamlessly crossing application and information type boundaries. As all information managed by the system is treated in a similar way, any type of information can be retrieved by all of the above methods. Furthermore, structured queries can be created that operate on information that was previously split between different sources. This solves the problem of combined queries on distributed information as described by (Dittrich, et al., 2005).

The ability to link, tag, annotate and supplement existing information reduces the effort needed to file a piece of information so that it can be found later. The user is not forced to decide on an elaborate filing scheme beforehand. They may just create an information object and find it later by searching for some aspect of that information which the user remembers – e.g. associated objects, a type, some text that was written, etc.

It was observed that it is not always easy to find information that was imported into the system in large amounts (as opposed to dragging a particular item into the PIM window). This problem was encountered when the complete set of messages or contacts were imported from an IMAP server or from Mozilla Thunderbird. The original application provides a familiar interface for finding information. For example, messages are found in folders, and contacts are listed in the address book. The prototype does provide a similar view for folders and messages, but not for contacts. As contacts do not necessarily represent persons (e.g. a contact could be an organization), they are also not shown in the list of persons. A full list of contacts can only be obtained by performing a search for all objects of the Contact type. However, this requires the knowledge of the type name, which does not always have to be straightforward. As the list of types increases, it might get increasingly difficult to find information that is not yet linked to other user's information.

## 6.4. Potential

### 6.4.1. Personal information domains

Personal information management covers a broad set of domains that all have their specific activities and needs related to keeping, organizing and retrieving information. Calendaring requires reminding about upcoming meetings and events and communicating to settle on a suitable time and date. Task management involves prioritizing to-do's while keeping track of the dependencies between them. Contact management requires the ability to quickly contact a person using different means of communication – an email, an instant message or a voice call. All of these different domains have a thing in common – they form a part of a person's body of information. The information managed in one domain is not isolated and independent, but it is shared with other domains.

Due to these considerations, a unified approach to personal information management, such as the one proposed by the model described in this thesis, is beneficial to the user.

The data model used in this system (see Section 4.1.1) is designed as a generic data model that can support any domain, with personal information management being just one of the possible applications. The client application, in turn, is partly designed as a generic information editor. The user can adapt the system to support the specific type of information they need to store. Whether it is calendaring, task management or making a library of favorite movies, the system just needs a set of types and views in order to support that activity. Some specialized activities, such as sending an email or reminding, can be achieved by defining additional views programmatically. In this way the capabilities of the system can be expanded even further.

### 6.4.2. Social communication

The rise of social networks and other services that focus on social interaction has led to the emergence of new forms of communication. People using Facebook can communicate by posting a "status message" for their friends to see. They can also post pictures, videos and links. Whenever they add a new person to their social network, this information becomes visible to others. All these activities can motivate their friends to comment on them or indicate that they "like" them. Users of Twitter communicate by posting short messages. However, these messages are not organized in any particular way. Unlike Facebook, it is not directly possible to comment on what someone else has said. Instead, people use various tags (e.g. "#iphone", "#BBC", "@username") to indicate what they are referring to.

This form of communication is very spontaneous. If a person finds something interesting, they may comment on it. Unlike e-mail, these messages are usually very short. They may also be scattered between various social networking sites that the person uses – there is no centralized place, like an Inbox folder, where they reside. However, in a similar way to email, these messages are static – once created, their contents does not change.

The model proposed in this thesis can be used for managing this kind of communication. In the case of Facebook, the system might keep track of all items that a user has "liked" or commented on. Such an item could be represented by an object of a corresponding type (e.g. StatusUpdate, Comment, Link, Picture, etc.) Various metadata would be imported together with the item –

posting date, associated user, etc. It would then be easy to find such an item by performing a search and specifying the things that a person can remember about it.

Managing Twitter feeds could be accomplished by extracting the tags that each message carries as separate objects – an automated version of the functionality for marking unstructured information. The user could then find messages either by doing a keyword search or by following links.

One of the benefits of using the PIM system for working with this type of communication is the possibility of unifying it. The user could group together messages from different social networks into threads or topics that are meaningful to him or her. If a topic involves other resources, such as websites, pictures or videos, they can also be linked to that conversation.

### 6.4.3. Google Wave

Google Wave<sup>21</sup> is an application being a mix of communication, collaboration and personal information management systems. A user can create a wave, which resembles a collection of documents. Each such document can contain rich text, photos, videos, maps, etc. A place in a document can be a starting point of a new one. Waves can be shared between users and modified by many people at the same time.

This kind of functionality enables many different applications. Google Wave can be used in a similar way as email, where a wave contains a series of messages formed as a thread. Because the modifications are visible in real time, it can be used as an instant messaging system. Due to its support for rich content, it can be used as a collaborative document editing system, where users can directly comment on different fragments of text. Various extensions (such as a reservation confirmation gadget and a map annotation tool) make it well suited for sending out invitations.

An interesting combination would be an integration of the system described in this paper with Google Wave. The latter is a good communication and collaboration tool, but lacks support for storing different types of structured information. The former enables the user to store and retrieve different kinds of information, but does not (per design decision) have such extensive sharing capabilities.

Wave can function as a frontend for different kinds of data managed by the PIM system. For example, in the Inviting people scenario, Sue could publish the invitations as a wave. People could indicate whether they are coming or not or add some comments, and this information would be automatically visible in the PIM system. From there, Sue can annotate the comments with to-do's, if they require some attention on her part.

In general, different types of data from the PIM system could be exposed to outside users through Google Wave, possibly through the use of some specialized extensions (such as the reservation gadget). The PIM system will collect this data and let the user view and work with it in different ways.

---

<sup>21</sup> <http://wave.google.com/>

Integration of these systems should be possible due to the flexibility of the model proposed in this thesis. For example, in a similar way as a wave is composed of different types of content (text, videos, sub-waves, etc.), a Notebook Page can contain different types of objects (text, pictures, other Pages). The main difficulty would come from the dynamic nature of the wave. A method for synchronizing data would have to be developed.

## 6.5. Summary

The evaluation has shown that:

- The proposed model is flexible, as it supports alternative means to realize scenarios, can support different types of activities and information, and can be integrated with new ways of communicating and collaborating.
- Combining information from different sources enables it to be linked and queried without regard for application and information type boundaries.
- The system supports an individual way of working with information. The user can structure it as is needed to accomplish their task.
- This system gives the user more freedom in working with their information. The user is not limited by the data types, structure of information and ways of viewing it that a particular application may support. Additional information can be always stored in a structured way.
- Extracting unstructured information lets previously indivisible bodies of data be decomposed into their constituent parts. This enables them to be treated as any other information in the system and makes them easier to find and manage.
- The gap between this system and other applications is reduced by allowing the user to open an object in the application it originated from.
- The effort to file information so that it can be retrieved later is reduced when using this model. Any part of a piece of information that the user can remember can be used as a starting point to find it.
- Merging objects reduces the effort required to find information by eliminating information fragmentation. It also keeps the user's body of personal information consistent and organized by removing duplicated information.
- The system enables the user to work with information stored in different places in a similar way (e.g. combining pictures from disk and from the web).
- Annotating objects provides a quick way to retrieve them later. Due to integration of information from many sources, any kind of information can be annotated.
- The separation of information from its visual representation lets it be viewed in ways that are useful to the user.



## 7. Discussion

This chapter explains some of the design decisions that have been made while working on this project and proposes some alternative solutions that could have been chosen. It also describes the technical problems that have been encountered while implementing the prototype.

### 7.1. Alternative design decisions

#### 7.1.1. Client-server interoperability

It has been stated earlier that the primary reasons for creating a separate client and server parts of the system were to allow for the possibility of having different types of clients (web client, mobile client, etc.), to have a background process where long-running tasks could be executed and due to the choice of implementation technologies.

These are all valid reasons. In order to realize them, the initial approach was to implement the object model (Section 4.1.1) and associated logic on the server side, and create a “dumb” client that would only exchange data with the server and display it to the user. In this way, the logic on the server side could be reused by different clients. However, as the design progressed it was found that the client would have to contact the server many times for even the simplest tasks – such as finding a property value to display. As this would result in much overhead for remote method calls, most of the logic was moved to the client tier and the server is responsible only for reading and writing data to the RDF repository.

In general, it was found that the application object model was the single most difficult part to realize. The complexity of a data model of a normal application (e.g. backed by a relational database) depends on how precisely the details of a given domain are reflected in the design. For example, an address book application might provide a fixed set of fields to the user and is not concerned about all the other possibilities in which a person might be described and associated with other things. In this system, the data model hard coded in the application is used to describe another, different data model residing on top of it. This means that the data model needs to be robust enough to support different ways in which a user could structure the data. The functionality associated with types, objects, inheritance, type checking, collections, normally offered by a programming language has to be implemented as part of the application.

Another problem that was observed is that the communication between server and client (which is realized through the exchange of data transfer objects) results in a complex implementation that is error prone. Each type of Persistent object requires two sets of data transfer objects – for sending data to the client (these transfer objects resemble “normal” Persistent objects) and for committing changes (these carry the old and new values for each field).

Several solutions to these problems are possible. One of them is to create a server which does not function as a proxy between the client and the triplestore. The client would communicate directly with the repository, eliminating the need for the data transfer object layer. The server would still be required to execute background tasks, such as importing data, but would maintain its own connection to the RDF repository.

Another solution would be to refactor the data transfer layer into a more generic one and replace the plethora of objects with only a few. This would remove the problems associated with this

approach while keeping its benefits – the client is now designed in a way which makes it largely independent of the underlying physical data model. It can support any model that would satisfy the requirements of the application object model, but it does not have to be RDF. In principle, the associative model (described in Section 2.4.4) could be used as well, although this would introduce other problems. Another idea for a data model will be discussed later on.

Yet another solution, which was tested during this project, was to use an interoperability layer that would allow for executing Java code on the .NET virtual machine. Ja.NET<sup>22</sup> is an open-source implementation of the Java 5 SE SDK environment for .NET. In this scenario, the whole system would be implemented in .NET only, with the Java code being an integral part. The client and the server could be executed as one process, eliminating the need for a data transfer layer. They would also share the same codebase, which means that for less robust client implementations most of the logic could still be located on the server. However, Ja.NET is still in alpha development stage and there were compatibility problems with the libraries used.

### 7.1.2. Physical data model

Instead of using the Resource Description Framework as the physical model for the system, a similar model could have been constructed on top of a relational database. The rationale for creating such a model from scratch would be to include functionality which is not directly supported by RDF and has to be realized by certain workarounds. Some of these limitations are listed below:

- It is not possible to directly reference the object part of the statement if it is a literal value. Instead, a separate statement needs to be constructed
- It is not possible to reference a whole statement. The solution employed in RDF to overcome this (creating a separate resource describing the original statement) can be seen by itself as a “workaround” and can lead to inconsistencies if the original statement is modified and the reified one is not.
- It is not possible to attach some application-level metadata to statements. For example, storing such information as the date of creation or a deletion bit (indicating that a statement was removed without actually removing it), would help in synchronizing data between multiple installations of the system. It would also make it possible to provide a history of changes to the user – e.g. the user could view how their body of information evolved over time.

However, building such a model would require duplicating most of what RDF already does. It would be difficult to recreate some of the advanced functionality that RDF stores already implement, such as querying and reasoning. A better solution might therefore be to modify one of the existing open-source RDF triple stores and introduce the needed changes. Sesame is a potential candidate.

### 7.1.3. Custom views as RDF

The current design allows the user to define custom views, which are composed of other (custom or built-in) views. The definition of a view is stored as a XAML document.

---

<sup>22</sup> <http://www.janetdev.org/>

The use of XAML was mostly an outcome of the selected UI framework – the Windows Presentation Foundation – which uses XAML natively to describe user interface elements. From the implementation point of view, it was convenient to use the features already offered by the library – loading and saving user interface elements to XAML files.

However, an alternative choice was to use a custom ontology for describing views and persisting them in RDF. This is an approach employed by the Haystack project (Huynh, et al., 2002).

Both approaches give the opportunity to exchange custom view definitions between different users. Huynh, Karger and Quan given an example in which a new employee enters a company with a large intranet. He can make use of the various custom views that other employees have collected (or created) over time as his starting point for exploring the resources available as part of this intranet (Huynh, et al., 2002 p. 6).

The advantage of RDF over XAML in this area would be that user interface elements could be annotated in the same way as other data in the system. It would also be possible to query for views that have a particular feature.

#### 7.1.4. Data persistence

The implementation of the application object model, as described in Section 4.1.1, allows for only a single instance of any Persistent object to exist at a given time. Such an approach was taken to maintain consistency of the data across all views active in the application. For example, when one of an object's properties is modified in one view, the change will be automatically visible in another.<sup>23</sup>

What follows from that design decision is that it is difficult to define a transactional boundary that would encompass a certain set of modifications that a user has performed. It is possible for the user to start editing an object in one view, then switch to another view to modify a different object and then use a third view to link yet another object to the first one. All these changes take application-wide effect immediately.

For this reason, the client does not have any UI elements such as “Save” buttons. Whenever a change is made, a background thread will commit it to the RDF repository. The drawback of this approach is that there is no transactional integrity with respect to a single object. The data is saved in chunks, which contain changes done to all in-memory objects. If a problem occurs while saving a particular chunk, it is difficult to determine which changes were committed (as part of the previous chunks) and which not (i.e. are still waiting to be committed).

An alternate design could allow multiple instances of a single object. Then any view would operate on a copy of the object's data. Clicking the view's “Save” button would commit the changes to the repository and update the master in-memory copy of the object.

#### 7.1.5. Partial loading of data

In a personal information management system, each information object can be linked to many other ones. As such, the amount of data stored in association with an object can be quite large.

<sup>23</sup> Such functionality also requires a notification system that will inform all interested views that a particular property value was modified and that they need to refresh the data being displayed. This is accomplished using .NET's databinding functionality, among others.

Preventing all data contained in an object from being retrieved whenever that object is referenced is important from the performance and memory consumption points of view.

Consider, for example, that the user wants to browse a list of all the people he or she knows. Each person object could have many properties associated with it – names, addresses, telephone numbers, bank account numbers, e-mail addresses, etc. If an object is always retrieved in a fully populated state, then all that data will be loaded for every object in the list, even though it is not actually needed – the application will just to display a list of names.

The implementation of the object model (see Section 5.2.1) allows for objects to be partially populated with data from the repository. However, this feature is currently not used to realize a lazy loading pattern (Fowler, 2002 p. 200), due to other complexities involved:

First it has to be noted that it needs to be known up-front (i.e. before the data is requested from the RDF repository) what properties of an object are actually needed. In this way a query can be constructed that will fetch only those properties. An alternative approach would be to request the value of each property at the moment it is actually needed – this would result in multiple queries instead of one.

This list of properties can only be provided by the code responsible for requesting the data in the first place. In most cases, this will be the view that renders an object on screen. If multiple objects are displayed as part of a collection, each object will be rendered by its own façade view. The choice of the view usually depends on the type of object. Therefore the view rendering the collection must provide the types of objects that will be displayed, so that the corresponding façade views can be analyzed to extract the properties they need. Not all of that information is always available in advance.

Apart from the above problems, working with partially loaded objects is also complicated. In particular, it is difficult to determine if the fact that the given data is not present means that it does not exist at all or that it hasn't been loaded yet. The reason for this is that data is obtained as a result of executing queries, and queries (such as the ones constructed when the user is specifying search criteria) can be arbitrarily complex. It is not trivial to determine whether a query did not return a particular property value because it does not exist, or because it was excluded by the search criteria. The solution is to execute a query that is known to produce the desired result, and not rely on the data that might have been retrieved as part of user queries.

The design decision for prototype implementation was to always retrieve all data associated with the object, therefore sacrificing performance for simplicity.

## **7.2. Third party components**

### **7.2.1. Sesame**

Sesame is the RDF store used to hold all data. Although it is already a stable and mature project, it still lacks in certain areas that are of importance to the PIM system. The two primary problems are performance and inference support.

Sesame's native store is fast when it comes to retrieval of data, but as the number of triples increases, the removal of small sets of statements becomes very slow. Every modification of data

in the PIM system involves the removal of old statements and the addition of new ones. It has been observed that after importing profiles of just 50 Facebook friends, the update time for a few statements increased to a couple of seconds. Fortunately, this problem is supposedly solved by the latest version of Sesame.

Another problem involves the lack of support for OWL inferencing. This functionality is required so that the use of inverse and symmetric properties would result in the automatic generation of associated statements in the repository. Currently, this functionality is, to a limited extent, emulated by the prototype.

### 7.2.2. Hessian

The Hessian protocol was used as a binary replacement for XML webservices. However, a few problems were encountered when using it in this project:

- No support for generics. Even though both Java and C# do support generic collections (such as `List<T>`), the protocol did not map C# types to corresponding Java types. As a result, the Data Transfer Object classes could not use generic collections.
- Exception handling not working. Even though the protocol specification does support passing exceptions from the server to the client, this functionality did not work in the C# implementation that was used (called HessianC#<sup>24</sup>).
- Buggy implementation. The HessianC# implementation of the protocol proved to have concurrency issues when multiple requests were being executed at the same time. A fix was filed with the maintainers of that project.

### 7.2.3. Aperture

Aperture is the framework used for importing data from external sources. In Section 4.4.1 it was described that it should be both possible to import all information contained in a datasource and to select a single item that needs to be imported. Unfortunately, Aperture only supports the former mode.

This is a serious drawback if one considers real-world applications. For example, my Inbox folder in Mozilla Thunderbird contains almost 3000 messages. The size of the disk file which stores those messages is around 1 GB. Most of these messages are kept for archival purposes and do not need to be managed by the PIM system. However, due to Aperture's limitation, in order to manage only my newest messages I would be forced to import the whole Inbox folder.

---

<sup>24</sup> <http://www.hessiancsharp.org/>

## 8. Related works

### 8.1. Haystack

Haystack (Adar, et al., 1999) is a MIT research project started in 1997 with the aim to “*develop a tool that allows users to easily manage their documents, e-mail messages, appointments, tasks, and other information*” (Huynh, et al., 2002). There were four main goals to the project:

- Maximum flexibility to the user in organizing information.
- Treating different types of information in a similar way.
- Ease of manipulation and visualization of information.
- Delegation of tasks to automated agents.

As such, Haystack has many similarities to the system described in this paper. They both use RDF for storing and describing data, both make it possible to import data from external sources, provide the user with an ontology editor, and have a customizable user interface.

Haystack puts more focus on searching and retrieving information. For example, it observes how a user follows links from one information object to another to offer a similar traversal path as a hint in the future. It also uses various automated processes for extracting more information from the data already contained in Haystack. For example, it might associate a person with document by analyzing the document’s metadata and concluding that a person known to the user is an author of that document.

This paper, on the other hand, concentrates on working with information from different sources. The Haystack project doesn’t seem to be concerned with the implications of combining such information except that it can be searched, grouped and annotated. In particular, it does not have any mechanism for defining equivalence between information objects. Furthermore, the data sources used for importing information seem to be used in a read-only fashion, i.e. once the information is available in Haystack, any changes cannot be propagated back to the original sources.

Moreover, Haystack uses RDF for its data layer and RDF Schema for the ontology definition, but does not appear to use the benefits that come from defining a hierarchy of properties (see Section 4.1.3), such as displaying more specific values when the user requests a more general one. As the OWL specification (Bechhofer, et al., 2004) was first proposed in 2002 – when the project was near its end, it also doesn’t support complex relationships between properties, such as symmetry or inversion.

### 8.2. Chandler

Chandler (Open Source Applications Foundation) is an open-source information manager intended for personal use and small group collaboration. The primary types of information that can be managed include notes, e-mails, calendar entries, invitations, tasks and contacts. The primary problem that Chandler attempts to solve is to limit the amount of tasks a user has to keep track of at any single time. This is accomplished through triage – any information item can be classified as requiring immediate attention (“now”), being postponed to a future date (“later”) or marked as completed (“done”). Such a classification conforms to the frequency of use categories as described by Cole (Cole, 1982).

Chandler shares some similarities with the PIM system described in this paper. First of all, it tries to integrate different kinds of information in a single place. The vision for the project was that it should be possible to “*drag and drop emails, documents, tasks and events into Chandler from other apps*” (Open Source Applications Foundation p. Product Plan Process). However, it seems that the only type of information that can be imported into the 1.0 version of Chandler are emails (Chandler operates as a replacement for an email client with support for the IMAP protocol), tasks and calendar entries (from iCalendar files).

Secondly, Chandler uses a data model that allows treating different types of information in a similar way - all content is stored in instances of classes that have a common base type. Information items can be added to multiple collections, and the collections themselves are modeled as items. The data model also supports adding attributes (such as string, binary or integer values) to items. This provides some of the flexibility that is provided by RDF.

Despite the robust data model, the user interface doesn’t appear to allow the user to attach arbitrary values to information items – a function that is supported by the PIM system described in this paper. This is most prominently visible when one collects many types of information into a note created in Chandler: text, references to people, website URLs, etc. are all put into a single plain-text “content” field of a note item, where the meaning of those entities is lost and only obvious to the human user. In contrast, a note created in the system proposed here (see Section 4.1.3) can contain different types of elements which retain their semantic relationships.

Another drawback of Chandler is that it is designed to function as a replacement for an e-mail client. Being that e-mail plays an important role in personal information management - as a popular means of communication which often results in other follow-up activities, such as creating to-do tasks, notes, etc., this design decision might seem somewhat justified. However, writing an e-mail client from scratch is an enormous undertaking on its own. It is doubtful that Chandler will ever be able to compete with the functionality offered by (e.g.) Mozilla Thunderbird<sup>25</sup> or Microsoft Outlook. It forces prospective users to make a choice between having a bad e-mail client that does personal information management and a good e-mail client that doesn’t.

### 8.3. NEPOMUK

“NEPOMUK - The Social Semantic Desktop” was an European research project running from the beginning of 2006 to the end of 2008 whose goal was to develop methods, data structures, and a set of tools for building an environment for collaboration, personal data management, and organization of information created by persons and groups (NEP10). One of the outcomes of this project was the creation of the NEPOMUK (an acronym for Networked Environment for Personalized, Ontology-based Management of Unified Knowledge) framework, which helps in developing Semantic Desktop applications.

The concept of the Semantic Desktop comes from the idea of treating resources stored on a local computer as web resources and using technologies developed for the Semantic Web (such as RDF) to manage them - *every file on the desktop can be seen as a resource, as every email, photo, address book entry, and all other information we find on a typical PC* (Sauermann, 2005). In this

<sup>25</sup> <http://www.getthunderbird.com/>



view, systems such as Haystack or the one described in this paper qualify as Semantic Desktop implementations.

The NEPOMUK prototype provides the user with a client application for managing personal information. The user interface is divided into “perspectives”. Each perspective is composed of one or more windows or tabs. For example, the “PIMO Perspective” lets the user browse a hierarchy of classes and entries belonging to those classes. Clicking on a entry opens a tab showing its details. This is somewhat similar to the way finding, viewing and editing object works in the prototype described in this paper. However, it seems that the NEPOMUK client does not provide customized views for different classes of data – it always presents a list of properties and their values when editing class instances.

Another perspective is called “NepomukSimple”. It lets the user group resources into a collection, called a “pile”, and then use specialized views to display certain properties of those resources. For example, entries in the pile can be displayed on a map or as a timeline. This functionality is similar to creating a normal collection in the PIM prototype described here and using different `CollectionViews` for viewing it.

In contrast to this project and the Haystack project, NEPOMUK does not give the user the possibility of defining custom views for information. It does, however, provide the means for creating and modifying types.

Under the hood, NEPOMUK is a collection of a large number of components. It defines a set of ontologies for managing different types of information that might be stored on a personal computer. In comparison, this prototype only experimented with a few types. Both systems, however, make use of OWL, which allows for specifying e.g. inverse and symmetric properties.

NEPOMUK can import data from external sources using the Aperture framework, which is also used as part of the project described in this thesis. However, NEPOMUK’s design does not provide support for synchronizing data with the original application. External data imported into that system is treated as read-only.

The social and collaboration features of NEPOMUK include the ability to share metadata, exchange instant messages, and a function for distributed search and storage.

Some of the other components include:

- A Thunderbird plugin for tagging emails and adding them to piles.
- A tool for tagging and annotating webpages.
- DropBox – a tool for quickly classifying files downloaded from the Internet.
- Recommendation Services – provide recommendations on related resources.
- Text Analytics Services – provide algorithms for natural language processing of information to extract semantic relations.

## 8.4. Summary

The following table summarizes the differences between the systems described above and the one described in this paper.



	Haystack	Chandler	NEPOMUK	This system
Specialized views for objects	Yes	Yes	No	Yes
Specialized views for collections	(?)	No	Yes	Yes
Defining custom views	Yes	No	No	Yes
Modifying the type system	Yes	No	Yes	Yes
Adding extra information in a structured way <sup>26</sup>	No	No	Yes	Yes
Working with unstructured information	No	No	No	Yes
Tagging information	No	No	Yes	Yes*
Annotating information	No	No	No <sup>27</sup>	Yes
Linking information	Yes	No	Yes	Yes
Bidirectional relationships	No	No	Yes	Yes
Using external information	Yes	Yes	Yes	Yes
Synchronizing with external sources	No	Yes	No	Yes*
Removal of information fragmentation and duplication	Automatic analysis	No	Automatic analysis	Equivalent objects

\* This functionality is part of the design, but was not implemented in the prototype.

**Table 6: Summary of differences between related PIM projects**

The meaning of the items in the table is explained below:

- *Specialized views for objects* – Does the application provide different, specialized views for objects depending on their type?
- *Specialized views for collections* – Is it possible to view collections in more than one way (e.g. list view, timeline view, etc.), possibly depending on the type of objects in the collection or on the type of the collection itself?
- *Defining custom views* – Can the user extend the system by creating new views?
- *Modifying the type system* – Can the user extend the system by creating new data structures for storing information?
- *Adding extra information in a structured way* – Describes whether the application supports adding new values to existing entities as named properties.
- *Working with unstructured information* – Does the application provide means in which the user can annotate unstructured information, such as plain text and images? See Extracting unstructured information in Section 3.4
- *Tagging, annotating and linking information* – Indicates whether the application has support for adding tags, annotating existing information with notes or to-do's and for creating links between information objects that the user considers to be related.
- *Bidirectional relationships* – Does the data model make use of relationships (such as symmetric or inverse) which automatically create an association between the source and target entities and vice versa?

<sup>26</sup> Neither Haystack nor Chandler have support for this function even though their underlying data models do support it.

<sup>27</sup> Although NEPOMUK does have an “Annotate” feature, it realizes what is called by “Linking” and “Tagging” in this paper.

- *Using external information and Synchronizing with external sources* – Is the system capable of using information stored in external sources and propagating any changes back to those sources?
- *Removal of information fragmentation and duplication* - The method used by the system to eliminate duplicated information and to combine fragmented information. (*Automatic analysis* involves some automatic process, e.g. text analysis; *Equivalent objects* – See Section 2.3.2)

## 9. Conclusion

The aim of this project was to propose solutions and provide insight into the problems of managing distributed personal information in a unified way. Personal information in today's world is scattered between a large number of places. It is hard to get an overview of what we know due to the different boundaries that current technology imposes. Every application imposes its own way of organizing and working with information. A unified approach was therefore investigated as a potential solution to these problems.

The research for this project has started with the realization that existing popular solutions to personal information management are inadequate. An analysis of this problem has shown that "personal information" can generally include any kind of information that a person is interested in, while existing solutions restrict the user and even make the management problem worse by contributing to information fragmentation. An examination of how information is distributed between different sources has been performed and methods for unifying it have been described: combining information representing similar concepts, using links to connect information objects in a similar way as the mind associates thoughts, and using a single namespace to access information located in different places. An analysis of data models that could be used for storage of personal information revealed that the traditional approaches (e.g. the relational and object models) are not flexible enough for this purpose. A more appropriate model – the *Resource Description Framework* – supporting different ontologies, inferencing, extensibility and ease of linking information, was chosen instead.

The design of a prototype application was preceded by an investigation of scenarios demonstrating the benefits of unified distributed personal information management. The analysis of these scenarios has further revealed the features that a PIM system must have. These include flexibility in the amount and type of information that can be stored, the importance of treating all data in a similar way, the ability to add structure to unstructured information and later view that information in different ways.

Then, a system was designed that would solve some of the analyzed problems. It consists of a generic application data model, which enables the user to define custom types and properties on top of it. A mapping of this model onto RDF data has also been described. A number of data structures (including Person, Picture, To-Do Task, Notebook Page types) for storing different kinds of personal information have been defined on top of that model. The user interface consists of different kinds of views that enable the user to interact with data. The system includes the ability to compose views – complex views can be created from simpler ones. Additionally, methods for filing and organizing information through linking, merging, annotating and grouping have been described. Retrieval of information is facilitated by querying or following links between information objects. The requirements associated with reading and synchronizing external information have also been explained.

A three-tier system comprising a Windows client application, a Java server part and a (3<sup>rd</sup> party) RDF repository has been implemented according to the design. The implementation-specific details, alternative design decisions and technical problems have been discussed. It has been found that most of the logic needs to reside on the client tier, which makes it difficult to reuse implemented components if a client for a different platform were to be created. The RDF data

model is lacking in terms of attaching metadata to whole statements, which makes it difficult to provide synchronization and change tracking features. The selected approach for modifying data (all changes take place immediately) makes it difficult to perform transactional updates. It was also observed that a system like this needs the ability for lazy-loading of data. Some deficiencies related to third-party components have also been described. Due to the time constraints on this project, not all designed features were implemented in the prototype.

An evaluation of the model and prototype has been performed. Due to experimental difficulties, user tests have not been conducted. Instead, the model has been evaluated based on the realization of scenarios, examination of specific functionality and the potential for integration with new technologies. The following novel characteristics have been identified:

- The model provides a user with a central place to store, organize and retrieve their information. By synchronizing information with their original sources, the users body of personal information is always consistent. Furthermore, information fragmentation and duplication is reduced by merging equivalent objects. Even though this is a manual method, it can be used for resolving real world cases.
- The model also improves the way that unstructured information can be managed. By assigning meaning to unstructured information, the underlying structure can be exposed to the PIM system. This information can then be viewed in new ways, linked, annotated and can be retrieved more easily.

Other evaluation findings include:

- The model is flexible enough to support different ways to realize scenarios and does not restrict the user in terms of the amount and type of information that can be stored. Approaches to tighter integration with Facebook, Twitter and Google Wave have been positively evaluated, which also confirms this flexibility.
- Integration of different sources allows for information to be linked and queried without regard for application boundaries. It also removes the restrictions imposed by external applications in terms of storing and viewing data. The user is free to structure information in an individual way.
- The effort to file information was reduced considerably by increasing the amount of metadata that can be used to find it later.

Addressing the questions formulated in the problem definition for this thesis, it was found that:

- *Integration of sources* is the primary enabling factor that significantly increases the value of other functionality offered by the system. Accessing information from various sources has been considered in other works, such as Haystack (Huynh, et al., 2002) or NEPOMUK (NEP10), but the unique feature of this model are the provisions for synchronizing information with those sources. This makes it possible to work with information available through the computer system in a unified way - as if the boundaries imposed by the division of data into applications did not exist.
- *Storing different kinds of new information* enables the user to freely collect any additional information that might be worth keeping, without being limited by the type and amount of information that an existing application allows one to store.

- *Combining, linking, assigning meaning, tags or other metadata to information* has multiple uses. Merging equivalent objects helps reduce information fragmentation and duplication, which results in the user's body of information being better organized. Linking information allows the user to be explicit about the relations that exist between different entities. Together with tagging, it reduces the effort needed to file information so that it can be easily retrieved later. In addition, assigning meaning to unstructured information opens new ways in which it can be managed – once it is decomposed into individual items, these can be linked, annotated, grouped and viewed independently.
- *Organizing information into meaningful groups* can be realized as a function of linking. Information can be organized better if it can be a part of multiple groups at the same time, however this is not usually possible with popular existing applications.
- *Presenting information in a way that is useful to the user* can be accomplished by combining two different functions. The user can choose between different ways of visualizing information. They can also formulate queries that select specific information objects. Together this enables different properties of objects to be summarized in a single view.
- *Sharing information with others* is simplified by the system's ability to work with different sources. Selected parts of user's information can be published to social networks. New technologies, such as Google Wave, can be used as a public frontend to the user's body of information.

Due to the time constraints on this project, some ideas and functions are still waiting to be developed. There are a few dimensions along which this project could be continued:

- *More features.* Not all the views described in the design were implemented in this prototype. New ways of presenting information could be developed. Moreover, the proposed model tries to eliminate user dilemmas regarding the choice of an application by integrating information from many applications. Dilemmas concerning modification of existing information could be eliminated by introducing a change tracking feature.
- *Tighter integration.* Working with data stored in external sources is limited to just a few applications and in read-only mode. The prototype could be further developed in the direction of supporting more types of sources and, more importantly, allowing data to be synchronized with those sources. Furthermore, client applications for different platforms (mobile, web, etc.) could be developed, which would let the user access their personal information repository from any place.
- *More extensive prototype.* The prototype implemented during this project is not suitable for conducting user tests. However, it would be interesting to develop it further (in terms of stabilization, improving usability, etc.) and to see how this model is evaluated by real people.

## 10. References

*DCMI Metadata Terms*. [Online] [Cited: November 9, 2009.]

<http://dublincore.org/documents/dcmi-terms/>.

*FQL. Facebook Developer Wiki*. [Online] [Cited: January 11, 2010.]

<http://wiki.developers.facebook.com/index.php/FQL>.

*Bugzilla*. [Online] [Cited: May 4, 2009.] <http://www.bugzilla.org/>. (Bug09).

*Aperture framework*. [Online] [Cited: December 7, 2009.] <http://aperture.sourceforge.net/>.

(Ape09).

*DBpedia homepage*. [Online] [Cited: December 2, 2009.] <http://dbpedia.org/About>. (DBped09).

*Gnowsis*. [Online] [Cited: January 16, 2010.] <http://www.gnowsis.org/>. (Gno10).

*NEPOMUK - The Social Semantic Desktop*. [Online] [Cited: January 16, 2010.]

<http://nepomuk.semanticdesktop.org/>. (NEP10).

*Reading Thunderbird emails from a java application*. [Online] [Cited: January 11, 2010.]

<http://forums.mozillazine.org/viewtopic.php?p=2296839#2296839a>. (Rea10).

*The Trac Project*. [Online] [Cited: May 4, 2009.] <http://trac.edgewall.org/>. (Trac09).

*what-are-microformats. Microformats Wiki*. [Online] [Cited: October 14, 2009.]

<http://microformats.org/wiki/what-are-microformats>. (Micro09).

**Adar, Eytan, Karger, David and Stein, Lynn Andrea. 1999.** Haystack: per-user information environments. *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*. 1999, pp. 413-422. <http://doi.acm.org/10.1145/319950.323231>.

**Aduna. Sesame**. [Online] [Cited: January 06, 2010.] <http://www.openrdf.org/>.

**Alvarado, Christine, et al. 2003.** *Surviving the Information Explosion: How People Find Their Electronic Information*. 2003. MIT AI Memo AIM-2003-006.

**Bechhofer, Sean, et al. 2004.** OWL Web Ontology Language Reference. *W3C Recommendation*. February 10, 2004. <http://www.w3.org/TR/owl-ref/>.

**Berners-Lee, Tim, Fielding, Roy and Masinter, Larry. 2005.** *Uniform Resource Identifier (URI): Generic Syntax*. 2005. Request for Comments: 3986.

**Boyd, Danah M. and Ellison, Nicole B. 2007.** Social Network Sites: Definition, History, and Scholarship. *JOURNAL OF COMPUTER MEDIATED COMMUNICATION, ELECTRONIC EDITION*. 2007, Vol. 13, no. 1, pp. 210-230.

**Breslin, John and Decker, Stefan. 2007.** The Future of Social Networks on the Internet: The Need for Semantics. *IEEE Internet Computing*. Nov./Dec. 2007, Vol. 11, No. 6, pp. 86-90.

- Breslin, John G., Decker, Stefan and Bojars, Uldis. 2008.** The Future of Social Networks on the Internet: The Need for Semantics. *Presentation at the Semantic Technologies Conference 2008 in San Jose*. May 19, 2008. <http://url.ie/e46>.
- Brickley, Dan and Miller, Libby. 2007.** FOAF Vocabulary Specification 0.91. [Online] November 2, 2007. <http://xmlns.com/foaf/spec/20071002.html>.
- Bush, Vannevar. 1945.** As We May Think. *The Atlantic Monthly*. July 1945.
- Caucho Technology.** Hessian Binary Web Service Protocol. [Online] [Cited: January 6, 2010.] <http://hessian.caucho.com/>.
- Cole, I. 1982.** *Human aspects of office filing: Implications for the electronic office*. In Proc. of the Human Factors Society - 26th annual meeting. 1982. pp. 59-63.
- Cruysberghs, Stefan. 2007.** .NET - Querying Outlook and OneNote with LINQ. [Online] November 14, 2007. [Cited: January 11, 2010.] <http://scip.be/index.php?Page=ArticlesNET05&Lang=EN>.
- Dey, A.K. 2000.** *Providing Architectural Support for Building Context-Aware Applications*. College of Computing, Georgia Institute of Technology. 2000. Ph.D. thesis.
- Dittrich, Jens-Peter, et al. 2005.** iMeMex: escapes from the personal information jungle. *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. 2005, pp. 1306-1309.
- Dohmann, Friedhelm.** ECCO PRO still alive. [Online] [Cited: August 31, 2009.] <http://www.compusol.org/ecco/>.
- Dong, Xin and Halevy, Alon. 2005.** *A Platform for Personal Information Management and Integration*. In Proc. of CIDR. 2005.
- Erickson, Thomas. 2006.** From PIM to GIM: personal information management in group contexts. *Commun. ACM*. 2006, Vol. 49, no. 1, pp. 74-75. <http://doi.acm.org/globalproxy.cvt.dk/10.1145/1107458.1107495>.
- Fowler, Martin. 2002.** *Patterns of Enterprise Application Architecture*. s.l. : Addison-Wesley Professional, 2002.
- Hillmann, Diane. 2005.** Using Dublin Core. *Dublin Core Metadata Initiative*. [Online] November 7, 2005. [Cited: September 9, 2009.] <http://dublincore.org/documents/usageguide/>.
- Huynh, David, Karger, David and Quan, Dennis. 2002.** Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. *AAAI Technical Report WS-02-11*. 2002.
- Jones, S. R. and Thomas, P. J. 1997.** Empirical assessment of individuals' "personal information management systems". *Behaviour & Information Technology*. 1997, No. 16(3), pp. 158-160.
- Jones, William. 2008.** How is information personal? *In Proc. of PIM Workshop, SIGCHI*. April 5-6, 2008.

**Jones, William. 2005.** *Personal Information Management*. The Information School Technical Repository, University of Washington, Seattle. 2005. Technical Report.  
<http://hdl.handle.net/1773/2155>. IS-TR-2005-11-01.

**Jones, William, Munat, Charles F. and Bruce, Harry. 2005.** The Universal Labeler: Plan the Project and Let Your Information Follow. *Proceedings of the American Society for Information Science and Technology*. 2005, Vol. 42, No. 1.

**Karger, David R. and Jones, William. 2006.** Data unification in personal information management. *Commun. ACM*. 2006, Vol. 49, No. 1, pp. 77-82.

**Larsen, Jakob Eg. 2005.** *NEXUS. A Unified Approach to Personal Information Management in Interactive Systems*. Technical University of Denmark. 2005. Ph.D. Thesis, CICT Ph.D. Series No. 6.

**Lazysoft.** Lazysoft Technology: Sentences. [Online] [Cited: January 10, 2010.]  
[http://www.lazysoft.com/technology\\_sentences.htm](http://www.lazysoft.com/technology_sentences.htm).

**Lelli, Francesco, et al. 2008.** *NEPOMUK user guide*. 2008. Deliverable D6.7.A.  
<http://nepomuk.semanticdesktop.org/xwiki/bin/view/Main1/D6-7-A>.

**Malone, Thomas W. 1983.** How do people organize their desks?: Implications for the design of office information systems. *ACM Trans. Inf. Syst.* 1983, Vol. 1, no. 1, pp. 99-112.  
<http://doi.acm.org/10.1145/357423.357430>.

**Microsoft.** Naming Files, Paths, and Namespaces. *MSDN Library*. [Online] [Cited: December 4, 2009.] <http://msdn.microsoft.com/en-us/library/aa365247%28VS.85%29.aspx>.

**Microsoft.** OneNote Home Page. *Microsoft Office Online*. [Online] [Cited: September 4, 2009.]  
<http://office.microsoft.com/en-us/onenote/default.aspx>.

**Microsoft.** Windows Presentation Foundation. *MSDN Library*. [Online] [Cited: January 4, 2010.]  
<http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

**Microsoft.** XAML Overview. *MSDN Library*. [Online] [Cited: January 4, 2010.]  
<http://msdn.microsoft.com/en-us/library/ms752059.aspx>.

**Montalbano, Elizabeth. 2009.** Forrester: Microsoft Office in No Danger From Competitors. *PC World*. [Online] June 4, 2009.  
[http://www.pcworld.com/businesscenter/article/166123/forrester\\_microsoft\\_office\\_in\\_no\\_danger\\_from\\_competitors.html?tk=nl\\_dnx\\_h\\_crawl](http://www.pcworld.com/businesscenter/article/166123/forrester_microsoft_office_in_no_danger_from_competitors.html?tk=nl_dnx_h_crawl).

**Open Source Applications Foundation.** Chandler - The Note-to-Self Organizer. [Online] [Cited: December 29, 2009.] <http://chandlerproject.org/>.

**O'Reilly, Tim. 2005.** *What Is Web 2.0*. [Online] September 30, 2005. [Cited: November 9, 2009.]  
<http://oreilly.com/web2/archive/what-is-web-20.html>.

**Rowe, Matthew.** Facebook Foaf Generator. [Online] [Cited: January 8, 2010.]  
<http://www.dcs.shef.ac.uk/~mrowe/foafgenerator.html>.



**Sauermann, Leo. 2005.** The Gnowsis Semantic Desktop for Information Integration. *Proceedings of the IOA 2005 Workshop at the WM*. 2005. <http://www.dfki.uni-kl.de/~sauermann/papers/Sauermann2005a.pdf>.

**Schoen, Seth. 2009.** What Information is "Personally Identifiable"? *Electronic Frontier Foundation*. [Online] September 11, 2009. [Cited: December 2, 2009.] <http://www.eff.org/deeplinks/2009/09/what-information-personally-identifiable>.

**Szymański, Piotr. 2009.** *Requirements for distributed personal information management*. 2009. DTU Special course.

**Teevan, Jaime, Jones, William and Bederson, Benjamin B. 2006.** Personal Information Management. *Communications of the ACM*. 2006, Vol. 49, No. 1, pp. 40-43.

**Williams, Simon. 2000.** *The Associative Model of Data*. s.l. : Lazy Software Ltd, 2000.

**World Wide Web Consortium. 2004/s.** RDF Vocabulary Description Language 1.0: RDF Schema. [ed.] Dave Beckett. *W3C Recommendation*. February 10, 2004/s. <http://www.w3.org/TR/rdf-schema/>.

**World Wide Web Consortium. 2004/c.** Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C Recommendation*. 2004/c. <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.

**World Wide Web Consortium. 2001.** Semantic Web Activity Statement. *World Wide Web Consortium*. [Online] 2001. [Cited: December 2, 2009.] <http://www.w3.org/2001/sw/Activity.html>.

**World Wide Web Consortium. 2007.** SOAP Version 1.2 Part 0: Primer (Second Edition). *W3C Recommendation*. 2007. <http://www.w3.org/TR/2007/REC-soap12-part0-20070427/>.

**World Wide Web Consortium. 2008.** SPARQL Query Language for RDF. January 15, 2008. *W3C Recommendation*. <http://www.w3.org/TR/rdf-sparql-query/>.

**World Wide Web Consortium. 2004/d.** XML Schema Part 2: Datatypes Second Edition. *W3C Recommendation*. October 28, 2004/d. <http://www.w3.org/TR/xmlschema-2/>.

## Appendix A

Figure 40 shows how a Simple collection view is used to display a list of Person objects. Each Person object is rendered using a PersonFacadeView, which displays the primary picture and the display name of that person.

Note that some pictures and names were blurred to preserve privacy of their owners.

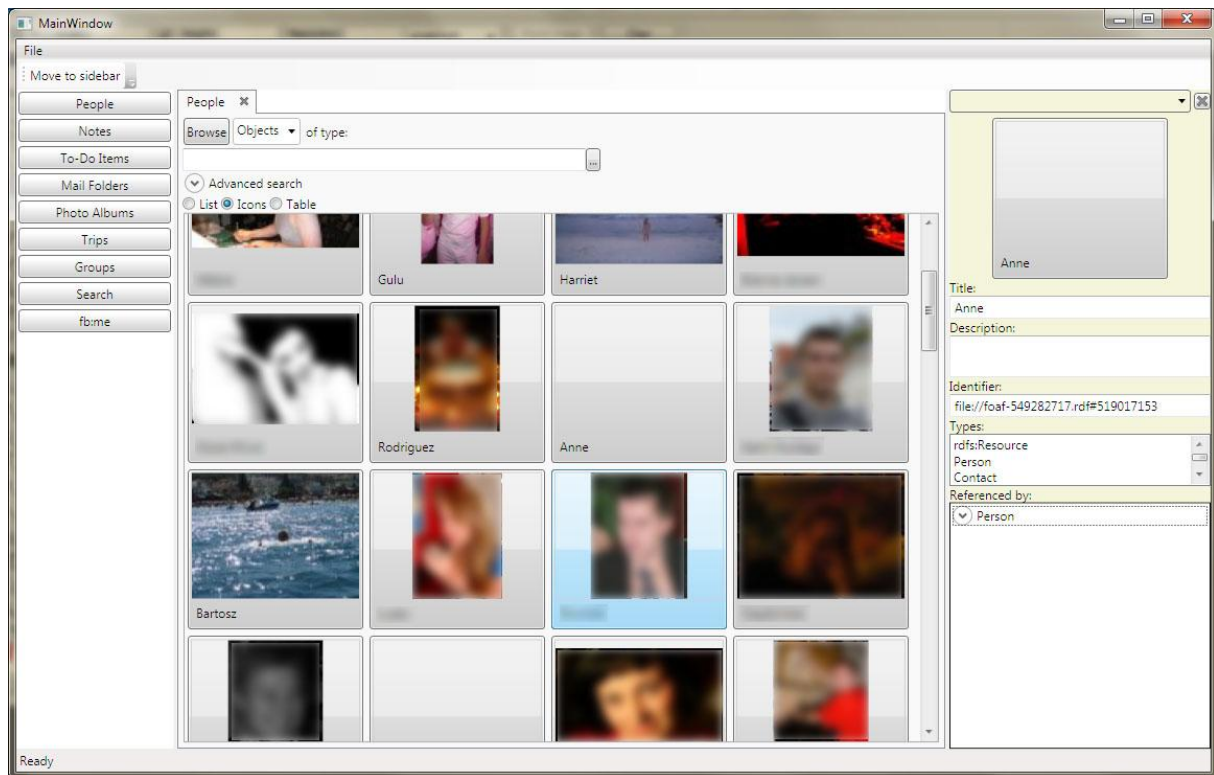


Figure 40: Client UI - a list of Person objects

Figure 41 shows how the sidebar can be used to operate the application in “split-screen” mode. In this mode the user can work with two views at the same time. It is therefore easy to drag elements from one view to another.

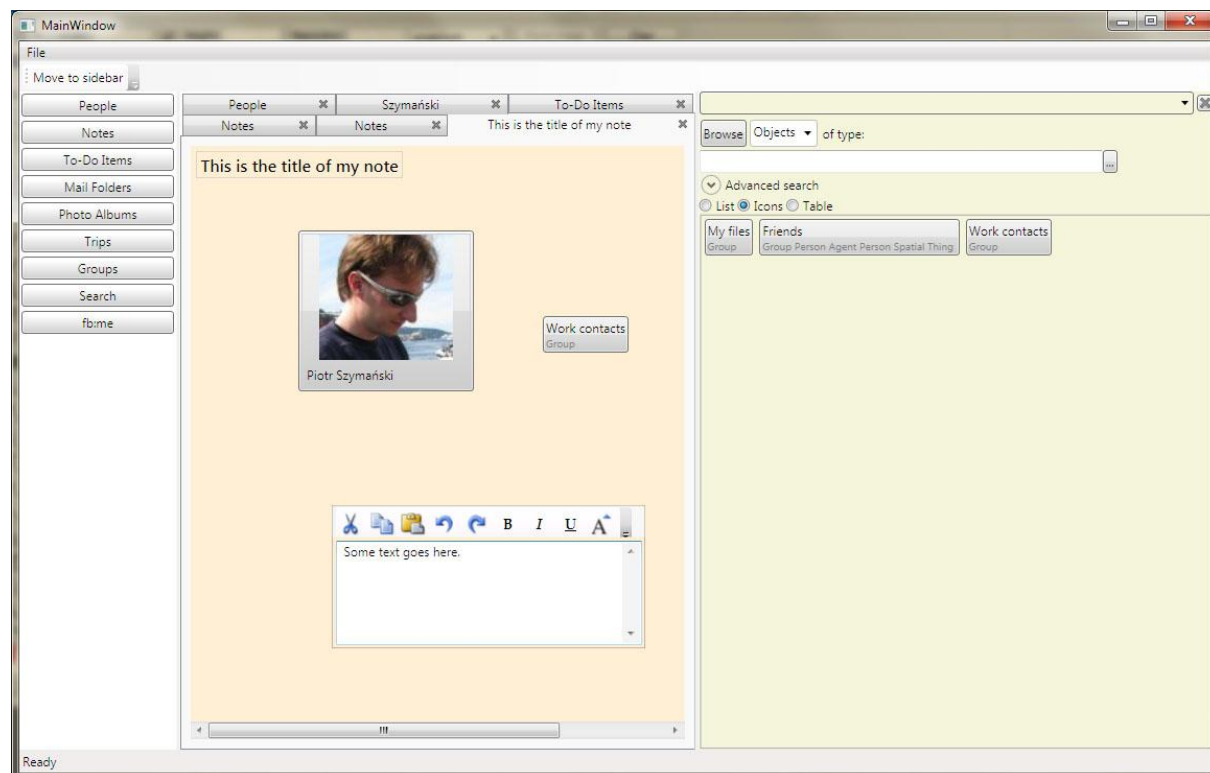


Figure 41: Client UI - editing a note