Exploiting Web 2.0 concepts for personal information management

Exploiting Web 2.0 Concepts for Personal Information Management

Master Thesis

Matthias Geel
<geelm@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Stefania Leone

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

1st April 2009

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

“Theory is when you know everything but nothing works. Practice is when everything works but you don’t know why. Our aim is to join theory and practice: Nothing works and we don’t know why.”

Unknown

Abstract

This thesis proposes an architecture that exploits concepts from Web 2.0 applications for personal information management. Sites such as Facebook provide an integrated portal-like solution to the management of all sorts of data through a very simple, intuitive style of interface. The interface is extensible and customisable in the sense that it is very simple for users to install other applications of interest and even to write their own applications. On the one hand, the architecture incorporates approaches that have already been proven useful in personal information management such as a uniform data model and semantic grouping. On the other hand, it introduces the concept of reusable database components to enable the development of a web-based, pluggable interface architecture for personal data management similar to that offered by many Web 2.0 applications. Such a pluggable interface architecture allows a user to use a set of different applications to manage the same data, or combinations of different data domains. A database component manages data of a specific domain and can be reused and extended by other components and applications. In order to tackle these challenges, this thesis has been divided into two parts. In a first step, the proposed architecture has been implemented as an extension to the database management system Avon. The resulting extension is available to all application developers that build upon Avon. In a second step, *MyApp Builder*, a web-based PIM application, has been developed that allows users to create their own *Personal Information Management* applications.

Contents

1 Introduction

A brief history of (personal) information management

Those who think personal information management did not start until recently are terribly mistaken. In the beginning, it was purely a task of the human mind due to lack of appropriate tools to record information. But very soon, people began to use dry to paint walls and chisels to scribble stone. Although these people surely did not think about information management, it was the first time ever information was *preserved*. The invention of parchment and paper, or rather their adoption in the western world during the Middle Ages, heralded a new era of information sharing but at that time it became evident that information *management* is necessary as well. Pages were bundled in folders, books were organised in bookshelves, and the most important source of personal information at the time, the classic mail, was locked up in escritoires. However, personal information management was still not a big issue because casual people could rarely read or write and hence there was still no other medium to transport information than one's own mind. The predominant areas at that time where information management was necessary were libraries and of course governments because piles of paper is the embodiment of bureaucracy. But after the industrialization, management efforts grew steadily as companies increased in size and more and more paper was produced as a result of daily business. Files, folders and cabinets were omnipresent not only in enterprises but also at home for personal information management. There was already more than enough material to organize: Bills, receipts, letters, news papers, magazines, publications, notes and those numerous official documents required by the governance. Nevertheless, the amount of information was bound by physical constraints. In 1945, Vannevar Bush [3] came up with an ingenious vision of a future where literally everything can be recorded and retrieved again. About forty years later, IBM and Apple presented their home computers. The revolutionary aspect of the first home computers, namely the Personal Computer from IBM and the Macintosh from Apple was not their technology but the fact that they were the first truly personal computers for everyone. And then the technological evolution accelerated - more processing power, more storage, more data, Internet, E-mail, the Web, we know the story. The temporarily last step in terms of web evolution has been coined Web 2.0 by Tim O'Reilly [14]. *Web 2.0* is not just another buzzword but serves as a synonym for a sociological phenomenon that reaches out to almost everyone. From passive to active, from consuming to producing, from retention to sharing, the users have become an essential part of the brave new Web. Whereas initially, only a few were responsible for creating and managing content, everyone can now actively participate and contribute. Today, users have become the web. They create, annotate, build, associate and upload their own data into zillions of social networks, blogs, web galleries and online applications. Although most of those web applications were created with sharing and collaboration in mind, users also have started to manage their personal data, which is not meant to be shared, with these applications. On Facebook, the world's largest social community, millions of people entered their very personal data, shared private pictures and sent confidential messages to their friends. Ignorant or just careless, a substantial amount of users literally hand over a significant part of one's privacy to a company that has probably as much commercial interests as any other company. This behaviour seemed to be unimaginable a couple of years ago, yet it is happening right now. The same with Flickr, the famous photo-sharing community. Although the company has no obligation to provide certain guarantees and can discontinue their services at any time or change to a different business model, people all over the world entrust Flickr their pictures. Many users have thus become dependent on such platforms. Nevertheless, Web 2.0 applications, especially social networks, have achieved what researchers are proclaiming for years. Data shall be managed in databases. However, this is not surprisingly because such applications have a number of advantages. Due to their very nature, they are accessible from almost everywhere and by everyone. In most cases, the applications are even free of charge and require very little previous knowledge. All what is needed is an Internet connection and a web browser. These advantages apparently match the user's expectations of a modern personal information management system and as a consequence must not be neglected in the design of similar systems.

1.1 Goals of this Thesis

The goal of this thesis is to harness some of the concepts that were responsible for the success of these Web 2.0 platforms introduced above and apply those techniques to the field of personal information management, abbreviated as PIM in the following.

The main deliverables of this project are as follow:

- The design and implementation of an infrastructure that supports a component-like architecture on the database layer with the goal of making application domain models extensible, reusable and ready for composition. The goal is to support the development of a web-based, pluggable interface architecture for personal data management similar to that offered by many Web 2.0 applications.

- The design and implementation of a web-based user interface that empowers ordinary users to design application domain models on their own and to manage personal data uniformly. Firstly, users shall be able to interactively design the application domain model for the personal information they want to manage. Based on that model, the system automatically generates a PIM application including a graphical user interface.

Secondly, users can create new data and edit existing objects with the aid of the generated applications. Finally, developers shall have the possibility to customize the visual representation of data types.

- As a proof-of-concept, an exemplary PIM application shall be implemented with the aid of the web interface and the underlying component database.

1.2 Requirements

While most design and architectural choices were left to the author of this thesis, there are some restrictions that apply to any implementation of the goals outlined in the previous section.

- *Java SE 6* will be used as the programming language.
- For all data, the database of choice is *OMS Avon*, a new Java-based implementation of the *OM model* that provides a semantic data management infrastructure on top of an existing storage provider. Avon can be used either with an in-memory storage provider for rapid development or a persistent storage implementation that has been developed on top of the open-source, object-oriented database *db4objects*.
- The infrastructure needed on the data management layer to introduce the notion of *database components* shall be implemented as an Avon module, which is the extension mechanism of the Avon database.
- The web application and client interface shall be implemented with a Rich Internet Application (RIA) framework and the latter shall be accessible with a web browser.

1.3 Thesis Outline

Chapter 2 sets the stage by introducing the reader with common topics and areas of research within the personal information management community. In addition, we motivate systems that support rich classification schemes and provide a uniform data model. Furthermore, some popular Web 2.0 applications are analysed and criteria for success will be identified. In chapter 3, we present the concepts and technologies that build the basis of this project. It includes the *OM* data model, the *Avon* database which implements this model and the extension mechanism of that database. As a preparatory step for the web technology evaluation, we also give a brief overview of the web technology evolution so far. The design and architecture of the *database components* infrastructure as well as the web application is explained in chapter 4. As already mentioned, a web technology evaluation has been performed in order to determine the implementing platform for the web application. Its outcome is disclosed in chapter 5. Implementation details of a component architecture for data management on the one hand and of an extensible, web-based user interface on the other hand are given in chapter 6. Afterwards, chapter 7 illustrates how the developed applications can be used to create PIM applications interactively. The last chapter concludes the result of this thesis and highlights some possible extensions to this project.

2 Related Work

This chapter highlights current trends in the field of PIM, as well as data management in general, and gives the reader an overview of related work in that area.

2.1 Data management

How to manage data, personal or general, effectively and efficiently has always been an area of great interest not only in research but also in business applications. Data management is a wide topic and spans over many active research areas. This section motivates classification structures by a comparison of *Typing vs. Classification* and discusses related work in the field of *Integration and Composition*.

2.1.1 Typing vs. Classification

The first question one has to ask when managing data is how to represent and structure it. Humans have been very creative about classifying things into categories and over the past decades, several scientific branches have come up with rich classification schemes and taxonomies, for example the naming schemes for species in biology. Classification is indeed quite natural and used heavily by the human mind to recognise similar concepts. Further evidence for the usefulness of classification as a mean to organise information can be found in [10] where collections are promoted as *flexible, essential Tools for Information Management*. Classification also includes quite philosophical aspects. If we attribute some properties to an object, do those properties become an integral part of that object and therefore contribute to its existence or is it rather a question of perspective, meaning that certain properties only exist if the object is being observed by another entity? As a consequence, any system that tries to model virtual or real entities should have some means to represent not only properties closely tied to those entities but also account for the fact that entities may be organised by semantic grouping, a concept that has been introduced in [13].

Example In gemmology, gemstones have been classified historically into two main categories: Precious stones and semi-precious stones. However, the categorization of gemstones into these classes has not been standardized or based on physical characteristics and has thus changed over time and even differs among cultures. Amethyst, for example, was regarded as a precious stone before the discovery of considerable occurrences in Brazil in the late 19th century. So, when a classification scheme for gemstones has to be modelled, it is reasonable to attribute measurable characteristics, such as hardness or market value (which of course can also vary over time but ideally represents the consolidated views of many), directly to gemstones. However, whether it is precious or semi-precious, depends heavily of the current context or the *perspective* of individuals. Such a categorization is better interpreted as a role or class membership. A gemstone *belongs* either to the class of precious or to the class of semi-precious stones rather than *possessing* the property preciousness. That is why it may make sense to regard types and classification structures as two distinct concepts that can be combined to enrich the expressiveness of a model.

Motivated by this example, we conclude that an information management system should expose a single unified concept of groups or collections to support classification. Collections help people navigate and retrieve information more easily and play an important role in structuring and organizing knowledge. It should be easy to create collection at any time, either as part of the design and model phase or even during operation of a system. For instance, collections can also be completely transient, for example as part of a query evaluation. To sum up, the idea of collections is to structure objects that represent data into (semantic) categories and also allow re-organising and re-classifying objects with absolute freedom.
Systems that try to model roles and classification with a single inheritance type system will sooner or later run into severe limitations. For example, consider the case where a university setting is to be modelled. We probably have a class to represent Students. Additionally, we want to model whether students are studying currently in the Bachelor or Master Curriculum. While a simple property to store the current curriculum might be sufficient at first sight, this approach exposes several problems. What if students can be in both curricula at the same time? What if they have to be in at least one curriculum at any time? Even worse for such systems, there can also be roles that are completely independent of each other. For example, a person can be a student but also a part-time employee in a company at the same time. Another important aspect of a classification system is that it should not only be hierarchical, but also permit structures that are orthogonal to each other. Assume a retail company that wants to manage its customer base. Apart from the distinction upon gender and maturity (for marketing purpose), customers should also be categorized according to their payment behaviour. Traditional approach is to have a single type-hierarchy that reflects the most important classification (for example maturity at the first level, gender at the second level) and then introduce properties for every other classification (payment behaviour) into the type at the corresponding hierarchy level. This technique exhibits several problems. What if the importance of a specific classification changes over time? What if two or more classifications are equally important? A more flexible classification model will be needed.
Classification or role modelling is of course also possible with a type model only, but this requires multiple inheritance and the object's ability to change its types freely. Which approach is more appropriate depends on the expressiveness of the classification model in place. If the classification model does not define new properties for each category, role modelling by types

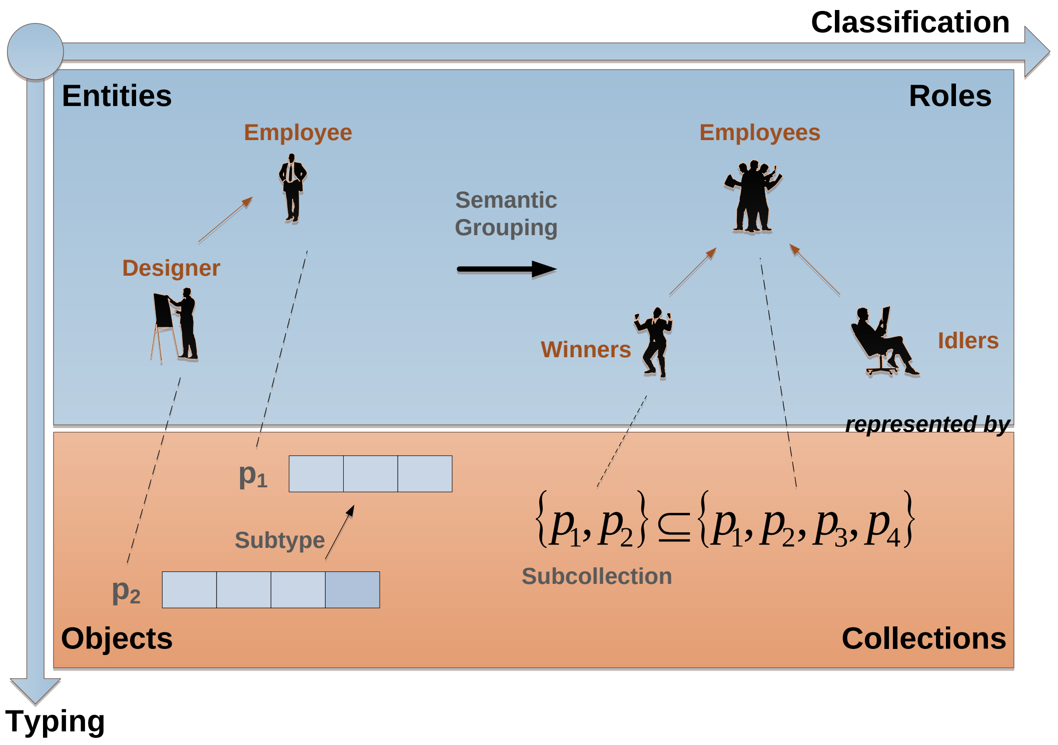


Figure 2.1: Typing vs. Classification

is inevitable when those roles are coupled with certain characteristics. Figure 2.1 is an adaptation of a figure from [12] and illustrates the differences between typing and classification. Common practices in Java (prior to the dawn of annotations) illustrate the developer's need for a classification mechanism and motivate the inclusion of such a system in a data model. Since Java does not offer any classification mechanism, programmers have started to use another concept for the same purpose: Interfaces. Instead of declaring any methods, only the interface declaration is given. Such interfaces are also called *tagging interfaces* and any implementing class is said to belong to the same group (or category as in classification) of classes tagged with that interface. Actually, they define roles or group membership. For example, the Serialization interface in Java denotes whether a class can be serialized or not. However, neither does it define any methods that have to be implemented nor does it enforce certain constraints. It is just a classification.

As a conclusion, a list of advantages of systems differentiating between typing and classification has been compiled:

- Better understanding of issues
- Reduces complexity of type graphs
- Aids rich classification structures
- Aids support for relationships between objects

In chapter 3, a data model is presented that includes a typing and classification layer and provides rich classification structures. The work of this thesis is based on that data model.

2.1.2 Integration and Composition

Integration describes the act of merging heterogeneous sources into a coherent and homogeneous form. Integration can happen at different levels.

Database Layer At the database layer, integration often involves consolidating data models and schemas. The data itself is then integrated into a central repository or database like in MyLifeBits [6] where everything is eventually stored within an relational database. Or it resides in its original place but is accessed through some intermediary integration layer, such as the Physical Data Independence Layer in iMeMeX [2]. Regardless of the approach chosen, the idea of any integration is to have either the data or metadata represented in a uniform data model or to provide uniform access to various data sources. The former idea became very popular with the advent of data warehouses and business intelligence. Usually, data warehouses are meant to collect and integrate a company's digital information into one central repository. They have been designed with schema consolidation and data integration in mind and thus offer a wide set of tools to extract data from other commercial database systems. Business intelligence tools retrieve and analyse that data in order to provide people in management positions with important figures and consolidated views. Such tools can then be used to support important business decisions. Data warehouses are especially well-suited to keep track of changes over time.

Application Layer Also known as enterprise application integration, the interoperability of applications is crucial in many environments. Ideally, applications possess a clean and public application programming interface (API) that allow them to be integrated into other systems. In reality, however, things are not that easy and the field of application integration ranges from sledgehammer approaches with screen scraping over standardized communication protocols to more elaborate systems such as the enterprise service bus that interconnects applications by messages.

Presentation Layer Integration at the UI level is a fairly new research area and has been explored by [16]. According to [5], creating user interfaces is a tedious and time-consuming task. That is the reason why the integration of already existing user interfaces is beneficial. In addition, integration at a lower level usually causes the layers above (most notably the presentation layer) to change even though the functionality requirements have not changed at all, resulting in a very bad cost-benefit ratio.

Integration is closely related to composition. However, a composition postulates that the components to be combined adhere to a common specification before they can be composed. Again, composition can occur at different levels and requires the presence of a component-based infrastructure.

This thesis focuses on the aspect of data (model) composition at the database layer and the construction of an integrated user interface for the newly created compositions. Integration, synchronisation and reconciliation of entities in a database system with external data sources have already been explored by a former student in his master thesis [15].

2.2 Personal Information Management

In contrast to traditional database systems that have become part of almost all companies' IT infrastructures, the success of databases on John Doe's personal computer is still awaited. On the other hand, conventional information retrieval techniques, such as keyword-based retrieval of information, have become widely accepted by the ever-growing Internet community and recently sneaked into everyone's desktop disguised as desktop search engines.
Interestingly enough, current state-of-the-art tools offer very little of what is being taught by researches to be "key ingredients" and yet they seem to make a large audience very happy. Microsoft Outlook for example has evolved very little in terms of its application model (as it manages basically the same data for almost a decade) and is not even extensible without considerable effort. Still it is widely used as one's favourite personal information management system. Admittedly, Outlook does its job perfectly but is it the answer to all our needs? The question remains whether ordinary users nowadays only manage limited set of information types because they have no choice or because they do not want to. Sadly enough, the most commonly used technique to manage personal data is still the good old file system based folder hierarchy. What is even worse is the fact that despite researchers' effort to come up with applications that allow the organisation of personal data in a more sophisticated way, people rather use Microsoft Outlook and just pack everything into e-mails, calendar entries and to-do items.
However, commercial tools solve only a fraction of all open problems and the lives of PIM researchers are very unlikely to become boring anytime soon. To illustrate this, a list has been compiled with some of the most challenging problems:

Personal data is heterogeneous. There is no application to rule them all. Domain-specific applications currently provide better and more sophisticated possibilities to deal with a certain kind of data or model respectively.

Not everyone is a database administrator. While it is definitely appropriate to have database administrators, domain experts, consultants and IT staff in a business environment, a personal information management system should be accessible by anyone, regardless of his or her technical background. Developer of new database systems not only have to provide interfaces which are more accessible to non-technical users but also have to rethink the way they present fundamental database concepts to ordinary user. Talking of nested queries, joins and relations will definitely not do the job.

Data gets fragmented. With the Web 2.0, data fragmentation has become an important issue. Even worse, electronic devices that produce personal data have proliferated. Notebooks, mobile phones, PDAs, netbooks, smart phones, and personal navigation systems contribute to the ever-growing digital haystack and searching the digital needle has become a labour-intensive task. With users publishing more and more content on the Web, personal data started to proliferate and one's own data is scattered across dozens of different web sites, local applications and even different devices. It is especially difficult to keep track of personal data within the Web because such data is being entered directly into an external system via the corresponding website, thus becoming actually a part of a different dataspace altogether.

Data duplication occurs. We worsen the situation even more by duplicating the very same data and spread it on the Web as we upload the same pictures to Flickr, Facebook and our personal blog, publish the same personal details in half a dozen social communities and may soon store the same files on Google-Storage, Amazon's Simple Storage Service and Microsoft's Skydrive. Those redundant pieces of data continue to live as separate entities and if we change one entity, the other (although semantically the same) does not change as well. As a direct result of the point above, it is not surprising that duplicates are omnipresent and unwanted redundancy as well as synchronisation between explicit copies remain open issues.

Dolce far niente. (or to be more honest, humans are lazy) Without a reason or proper motivation, people are reluctant to do anything. This is even truer when it comes to associating data manually or annotating them with additional information. As a result of this observation, successful systems either have to make sure that annotations or any other meta information can be created very easily without too much effort or, even better, create those automatically wherever possible.

Given that short list above, one can imagine how difficult it is for a single application to fulfil all these needs. Nevertheless, researchers have successfully tackled some of the problems mentioned but the tools that overcome both the technological challenges as well as the challenges in terms of usability are still to be born. For this is the hardest challenge of all - convincing users that a tool does what they were always looking for.
This thesis focuses on the concepts that are necessary to support not only an infrastructure for personal information management on the database layer, but also an extensible, pluggable web interface.

2.2.1 Hierarchical vs. Relational Data Organisation

From our experience with the physical world, we conclude that an item can only be in one place at the same time. In the digital world, this simple but yet fundamental observation no longer holds but nevertheless, we are accustomed to browse hierarchical file system where each file belongs to exactly one folder regardless of numerous options to overcome this nuisance such as symbolic links. And even if we have reached our goal after a long journey through the labyrinth of the file system, the next item we seek for, although semantically close to our current, might not even be along the same hierarchical path we have taken. So we have to restart and start our journey once again. But, as excellently stated by Vannevar Bush [3], "Human mind does not work that way. It operates by association."
Within the PIM community, there is a strong belief that modern personal information management tools should not only be able to cope with traditional data structures such as hierarchies and top-down categorization systems but should also handle more sophisticated approaches such as relations, tagging and associations. MyLifeBits [6] started as a project that relied almost completely on file and folder structures but as the amount of files grew steadily, that approach did not scale well so they shifted to a relational database. The reasons for this are

obvious. Databases offer much better query functionalities and can include arbitrary metadata. They have been designed with large quantities of data in mind and particularly relational databases have been tuned for performance for decades. File systems on the other hand are inherently hierarchical and offer limited support for additional metadata. Symbolic links and shortcuts may overcome some limitations but turned out to be very cumbersome in practice for most users. A short example illustrates why hierarchical structures are not sufficient.

Example After our return from a two-month journey through Europe, we upload a whole bunch of pictures from the digital camera to the computer. Since we visited quite a few cities, we might create folders for each city. Additionally, we would like to group pictures by the people depicted on them. So which attribute is more important and therefore at the top of our hierarchy? What if we later also want to group them by dates in order to see the destinations of the trip in chronological order?

We observe that data with orthogonal attributes does not fit into a hierarchical structure. Instead, classification structures shall be used where items can be members of different classes. Classification schemes are also necessary because flat systems will easily become incomprehensible as soon as the system scales in size.

2.2.2 Uniform data model

To provide linking and associations over the whole collection of personal data, that data has to comply with a common data model. There have been several proposals for such a data model and while their expressiveness is more or less the same, they differ greatly in terms of supporting tools and acceptance by developers.
Haystack [1] for example started with a proprietary data model that mirrored the metaphor of searching a needle in a haystack. “Needles” represented single pieces of information and “bales” were the equivalent to collections. Individual needles could be linked together by “ties”. Nowadays, the data model has been switched to RDF [8], the Resource Description Framework. RDF[1] is a formal specification to describe any resources that can be identified with a Uniform Resource Identifier and it was conceived by the W3C to annotate web resources with metadata. Its basic modelling elements are triples that consists of a “subject”, a “predicate” (property) and an “object”. Such a triple represents a directed, nevertheless binary, association between the subject and the object, given a specific predicate. Naturally, RDF structures can be viewed as a directed graph. RDF can be used without an enforced schema because triples can be added without any restrictions. That flexibility and the fact that it does not impose a hierarchical structure on the information to be managed are responsible for its success especially in the field of semantic web but also for the management of metadata.
Other approaches such as MyLifeBits are working with relational databases to represent all information. MyLifeBits tries to store everything from articles, books, music, pictures to phone calls, room conversations and even keystrokes and mouse clicks in a single relational database and uses SQL to manage this data.
On the other hand, there are always applications that implement their own proprietary data model. Popular ambassadors of such systems are iMeMeX and Semex [4]. iMeMeX intro-

[1]http://www.w3.org/RDF/

duces the notion of dataspace which is basically personal data grouped into a logical view. It therefore serves as an intermediate logical layer between applications and the data source layer. That additional layer then abstracts from underlying subsystems, from data formats and from devices. Data is therefore represented as a logical graph model independently of the actual location or its format.
Nevertheless, a couple of concepts are common to all uniform data models: Elementary building blocks that contain single pieces of data, composite structures that build more complex structures from simpler ones, the concept of containers or collections to group similar items and a method to link items together.

2.2.3 Database usability

When talking about PIM systems and data management in general, one must not forget an essential part of the whole ecosystem: the user. It were the search engines that allowed lay users to perform information retrieval on the Web by themselves and especially Google became known for its clean and simple interface that merely consisted of a single text field to enter keywords. The huge success of such systems indicates that usability is an important factor. But search engines on top of databases are not the answer to all needs. Sometimes, users want to formulate more complex and more sophisticated queries or do not want to be limited to retrieval only. However, according to [9], many DBMS interfaces that are available today exhibit the following problems:

Painful relations Joins between tables are not natural to ordinary users. They are difficult to understand and very complex due to normalization of relational database schemas. As a result of this normalization, information is scattered and it is difficult to trace data origin. Furthermore, it is tedious to follow foreign key constraints without any visualisation.

Painful options If users are confronted with too many options to choose from when they want to reach a specific database goal, they may get confused and the resulting interface can be counterintuitive. Additionally, options are mostly not ranked and so it happens that irrelevant options are as prominently placed as relevant ones, making it even harder for potential users to decide which option to choose. That is the reason why form-based interfaces are very successful; simplicity is the key.

Unexpected Pain In terms of unexpected behaviour, there are two different scenarios that can happen. *Unable to Query*: User frustration occurs if users cannot perform a specific query (either due to the lack of flexibility of the querying system or due to unexplained query restrictions) although they know that the data which supports the query exists. *Unexpected results*: If it is not visible where the results of the query come from, some results might be unexpected. The database or rather the user interface should explain why a certain result is on the result list.

Unseen Pain In most systems, the outcome of a query is only visible at the very end of the query formulation. This leaves users in a state of uncertainty while they are creating the query. Query feedback or intermediary results are very helpful for inexperienced users to see almost instantaneously what effect a specific input has. But this requires a "predictive capability" of the system on behalf of the user.

Birthing Problem Creating a database is cumbersome and getting the data into the newly created database even more, especially with an application that has not been tailored to the kind of information one wants to enter. Additionally, database systems may evolve over time because users usually do not have the full picture yet when they start to use a database. Appropriate tools are necessary that allow them to refactor the schema at any time.

Although database technology has made great process in the past decades, a large corpus of data is still managed outside of databases because of these issues. Some of these problems are going to be tackled as part of this thesis. In addition to the points mentioned above, personal information management systems have some more idiosyncrasies.

User-centric systems

One of the most fundamental differences between traditional information systems and personal information systems is that PIM tools have to adopt themselves to the user and not the other way round. While it might be appropriate for employees within a professional context to get in-house training in order to be able to use a specific information system, things just do not work like that for non-technical users in their private environments. Especially today, where the choice of tools is large and the time of users is small, only the ones that provide a reasonable benefit-cost ratio are taken into consideration. Matching ordinary users' expectations is either done by offering exactly that specific functionality they are looking for or by offering a flexible platform which allows ideally arbitrary customizations. Obviously, the latter one is much more appealing but flexibility always comes with a price, most notably increased complexity. So it is essential that PIM tools support the user mastering that complexity.

2.2.4 Web 2.0 and Personal Information Management

Despite the concerted efforts of researchers, hardly any tool for personal information management was successful in terms of reaching a larger audience. On the other hand, several web applications have successfully convinced ordinary users to entrust them with personal and even private data. Whether the extensively hyped buzzword *Web 2.0* refers to a change of web technologies, a sociological phenomenon, new business models or simply a new marketing gag, one aspect cannot be neglected. The self-perception of the internet user has changed drastically. Whereas initially, only a few were responsible for creating and managing content, everyone can now actively participate and contribute. Some prominent examples for *Web 2.0* applications are:

del.icio.us Bookmarks: Hyperlinks to web sites can be saved, annotated and shared with others. It also features recommendations, comments and browsing by associated tags.

Flickr Photos: Presumably the world's largest photo sharing community, Flickr is being known for one of the most accessible sites, featuring numerous tools and APIs to get pictures in and out of the system.

Facebook Friends: With over 150 million users, Facebook is the largest and fastest-growing social community site worldwide.

Xing Business: While Facebook was primarily designed for Students, Xing is the equivalent for the business world, focusing on features such as a marketplace for job offers or searching for members with specific qualifications.

Wikipedia Knowledge: An online encyclopaedia that promotes free information flow and whose content can be edited by anyone.

YouTube Videos: *The* online video portal, hosts millions of clips uploaded by Internet users and being held responsible for over 10% of global internet traffic.

Many web applications have specialised in managing information of a specific application-domain only, although there also exist systems such as Facebook that allow additional applications to be integrated via their extensive API in order to manage other data or provide additional functionality. Although most of those web applications were created with sharing and collaboration in mind, users also have started to manage their personal data, which is not meant to be shared, with these applications. Several applications therefore allow certain content to be marked as private which is then only visible by the owner of the data. More elaborate systems may provide the possibility to create entire groups or roles and to define fine-grained access rights for them.
What might be surprising at first sight, can be explained very easily once the advantages of such applications have been analysed and the ingredients of success have been identified.

Access Everywhere Data that resides on an online storage system can be accessed from everywhere and ideally, nothing more than a web browser is required on the client side.

Free of charge Basic membership is free and the core functionalities are available to everyone.

Accessible Usage of the system does not require any technical or specific knowledge.

Sharing Information can be shared with friends, groups or everyone.

These advantages apparently match the users' expectations of personal information management systems and as a consequence must not be neglected in the design of similar systems. While the sharing aspect has not been exploited in this thesis, the web-based access to (personal) information has.
Current trends in this area are the cooperation and integration of different web applications. This is possible because more and more web applications have started or are already offering an API to enable third-party developers to access the corresponding user network and data. Another trend is the movement towards a consolidated user management or login system, also known as "single-logon". Popular examples of that approach are Passport[2] by Microsoft and OpenID[3] which is also supported by Yahoo!, Google and Facebook.

[2]http://www.passport.net
[3]http://www.openid.net/

3
Background Information

This chapter introduces the Object Model (OM) and the Avon database system which are used as a model- and implementation base in this thesis. It will also introduce the reader with the technological and social evolution of the web and depicts how these changes have influenced the design of the web interface that has been developed as part of this thesis.

3.1 OM Data Model

In this section, the Object Model (OM) is introduced. The OM model is an extension to the well-known E/R model, one of the main modelling languages for relational databases. It is basically a symbiosis of concepts from the object-oriented world and the E/R model. Most importantly, it introduces the distinction between typing and classification, whose justification was given in section 2.1.1. Role modelling, however, can be accomplished by either the type layer or the classification layer. The choice of the appropriate layer depends on whether the role is associated with additional properties (role modelling by multiple types) or can be regarded as a semantic grouping (role modelling by classification).
The OM data model provides three main elements to model data: *Types*, *Collections* and *Associations*. The graphical notation of a simple OM model is given in figure 3.1.

3.1.1 Types and Objects

Types are the fundamental building blocks that describe how to represent entities within the OM data model. Distinctive features of the OM type model are *multiple inheritance*, allowing a type to have several supertypes and *multiple instantiation*, meaning that an object can have multiple types at the same time. Depending on the kind of type, they can have attributes, methods and triggers. The type of attributes can be any type that is part of the OM type model. One distinguishes between 4 different kinds of types: `Base types` reflect primitives from other data models and include strings, numeric types, Booleans and dates. `Object types` describe attributes and methods of objects and have an identity. `Structured types` on

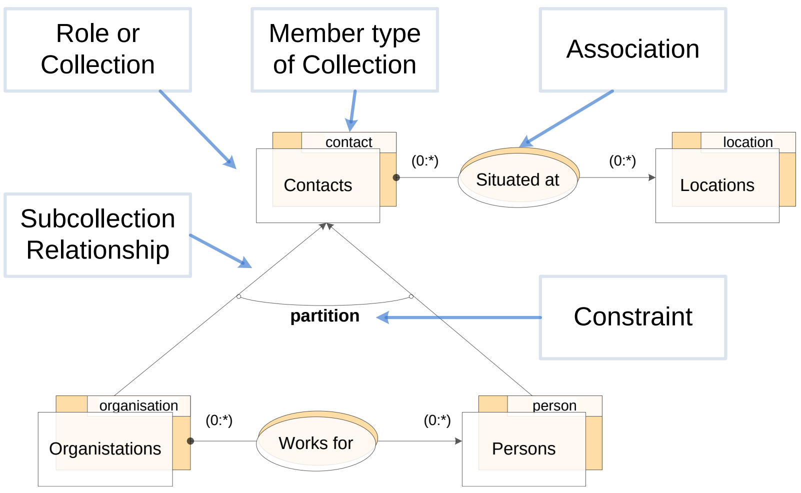


Figure 3.1: A simple OM model

the other hand, do not have an identity but define sets of attributes for structures similar to records in databases and structs in C. The last kind are `bulk types`, which introduce list- and array-like structures (collections) that support different collection behaviours, namely set, bag, sequence and ranking.
Unlike other object-oriented systems, newly created objects in the OM data model are not tied to a specific object type. In fact, any object is merely a bare individual with nothing more than an identity. An object in the context of a specific object type is called an *instance*. An instance of an object is a set of concrete attribute values that corresponds to the attributes defined by an object type.
Since an object can have multiple types, context becomes rather important when accessing attributes of that object. In fact, any access to attributes or methods always requires the object type to be specified that defines that attribute or method. In contrast to Java, no shadowing of field members occurs because in order to access a specific field or attribute respectively, the object type has to be specified in any case.

3.1.2 Collections

Collections are the building blocks of the classification layer of OM. They provide a mean to semantically group objects, but unlike types, they do not introduce new properties to those objects. All collections have to specify a member type. It serves as a constraint to the membership of objects. Only objects that exhibit an instantiation whose type is compatible with the required member type are allowed to be a part of that collection. If an object fails to comply with a membership constraint, it can simply acquire the necessary type because of multiple instantiation. A member type also defines the set of properties that are "visible" when objects are accessed in the context of the corresponding collection. This compensates the fact that collections by themselves cannot define new properties for their member objects.
Apart from the member type, collections also specify a collection behaviour that determines how objects are stored in that collection. `Sets` are unordered collections that do not allow duplicates. They correspond to Java classes that implement the *Set* interface. `Bags` are unordered collections that allow duplicates. They correspond to Java classes that directly

implement the *Collection* interface. `Rankings` are ordered collections that do not allow duplicates. They correspond to the Java class *LinkedHashSet*. `Sequences` are ordered collections that allow duplicates. They corresponds to Java classes that implement the *List* interface.
With collections, role modelling over time is straightforward. An object can acquire or lose a certain role simply by becoming a member of a collection, provided that any of its types is compatible with the required member type. When membership is not constraint at all, role modelling can be done entirely by classification.
Categorization schemes or classification structures are built by subcollection relationships and classification constraints over them. The subcollection relationship capability of the OM classification layer mirrors the expressiveness of the type layer and allows generalization as well as specialisation of collections. Classification constraints apply to a set of subcollection that have the same supercollection in common and restrict the membership of objects in the classification structure even further. For subcollections, these constraints are *partition*, *disjoint* and *cover*, for supercollections there exists only the *intersection* constraint.

3.1.3 Associations

Notwithstanding the recognition of their importance by many researchers, hardly any system or object-oriented language introduces associations as a first-order concept but resort to attribute mechanisms to represent relations between objects. In the OM data model, associations are a central part of the classification layer. They are defined over collections and thus have a domain and a range collection. Although that implies a direction of the association, they are bidirectional by nature.
Internally, associations are represented by binary collections. Each member of that collection consists of one object from the domain and one object from the range collection. Obviously, the same features that have been presented for collections will also work for binary collections. As with collections, no additional attributes can be defined for associations. However, there are cardinality constraints that can be enforced for the domain and range. Those constraints specify how many times the same object from the domain and range respectively can participate in an association. Similarly to the multiplicities in UML2, they are described as an interval with a lower and upper bound where a concrete upper bound can also be omitted, thus denoting an arbitrarily large number.
In OM, the arity of associations seems restricted to two. Nonetheless, N-ary associations are possible due to the fact that associations are collections themselves and hence are valid domain and range targets for new associations. By nesting associations, any arity is possible.

3.2 Avon

Avon is an implementation of OM written in Java that provides a semantic data management infrastructure on top of existing databases such as db4o, an object-oriented database, and other storage facilities. It is divided into three separate layers where each layer abstracts the implementation below and thus accounts for exchangeable implementations.
The top-most layer, the *interface layer*, provides an API to the concepts of the OM data model that have been described in section 3.1. Applications that are built on top of Avon and want to leverage the semantic data management features, should use this layer to interact with Avon.

The layer immediately below the interface layer is the *model layer*. It builds the core of the Avon database architecture and implements the OM metamodel, OML (Object Model Language) and utilities for database management in general. OML is the superordinate concept of a data definition language, a data manipulation language and a query language similar to SQL in the relational world, but adapted to the object-oriented OM data model. In addition, it is the model layer's responsibility to map concepts from the OM data model to the Java data model. Recently, the OM model layer has been extended by a module mechanism that allows new functionality to be introduced by extensions to the metamodel, the database language and to the Java classes responsible for managing the concepts described in the metamodel. Details thereof are given in the next section. The storage layer at the bottom is responsible for the persistence of the data. Currently, there are two implementations available for productivity. Firstly, an in-memory implementation that offers no long-term persistence and which is mainly suited for prototyping and rapid development. Secondly, a persistent implementation based on db4o. Db4o is an object-oriented database with a small memory-footprint and the capability to store any Java object transparently. There is no need for implementing certain interfaces or complying with certain requirements. The storage layer also implements the query engine needed by OML in the layer above. Other implementations that build on a different technology such as relational databases are possible but have not been implemented yet.

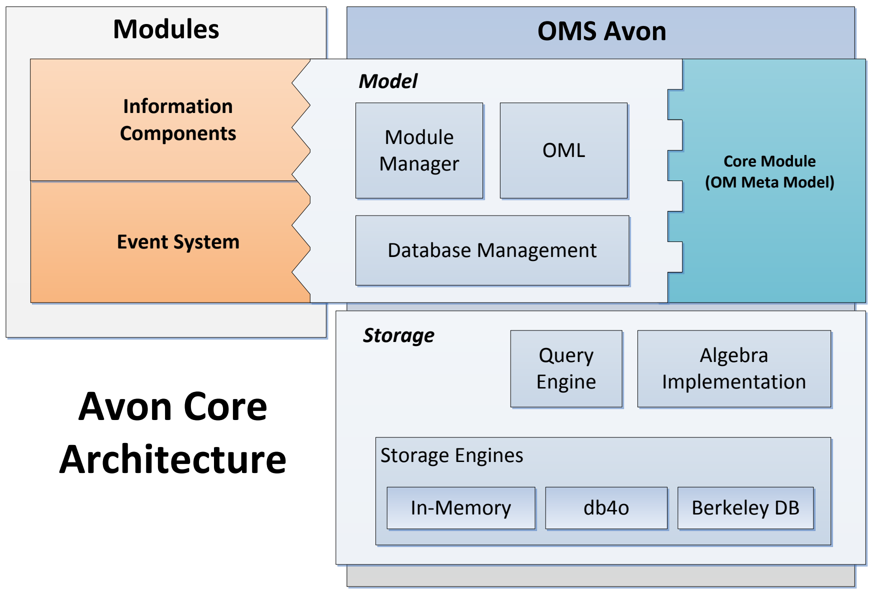


Figure 3.2: Avon Module Architecture

3.2.1 Avon Module Mechanism

The Avon module system came into existence as a result of traditional database systems becoming larger and larger in terms of functionality. Instead of having a "one size fits all" approach, orthogonal functionalities can be moved into separate modules that can be loaded and removed at runtime and thus, allowing Avon to adapt itself to various different environments. As a matter of fact, even the core functionality of Avon is a module, although it is

reasonable to automatically load it with the database system because most other modules have a strong dependency on the core module. Modules are extensions to the OM data model and therefore reside on the model layer of Avon together with the module manager. Figure 3.2 illustrates the Avon architecture including the extension mechanism.
Avon offers three extension points and it is up to the module developers whether they implement extensions for all of them. These extension points are: The OM metamodel, CRUD classes to manage instances of newly introduced concepts and new language constructs for the OML database language to integrate these concepts into the DDL, DML and query language. This module mechanism is of great importance to this thesis because the notion of reusable components on the database layer has been implemented as such a module.

3.3 (R)Evolution of Web Technology

Although the essence of Web 2.0 revolution was a sociological phenomenon, it was backed heavily by technological advances within the web community. Unlike many other areas where each group or organisation may develop their own solutions, it was inevitable to have organisations in place that work on standards and common practices in order to guarantee compatibility between different applications and clients (i.e. browsers) on a global scale. In addition, the W3 consortium, IETF (Internet Engineering Task Force) and IEEE (Institute of Electrical and Electronics Engineers) were able to establish well-known standards and specifications such as HTML, XML, XPath, RDF, XML Schema, TCP/IP or DNS for the benefit of interoperability between different implementations of web servers and browsers.
Nevertheless, it is important to analyse the different web technologies in order to understand some principles of today's web application frameworks. During the implementation phase of this thesis, several of those principles that have been successful in recent years were applied.

3.3.1 Static Web

As the reader may already know, the World Wide Web came into existence as a result of the academic world wanting to access information more easily on the Internet. Before the 90s, the most common way to access information remotely was to connect directly to a remote computer and browse its file system. It was Tim Berners-Lee, an employee at CERN, who came up with the idea of combining hypertext with the Internet. He developed initial drafts of what later became HTML and implemented the first ever web server at CERN. The design of HTML was motivated by his experience with the hypertext community. However, HTML was more concerned with the presentation of information directly embedded within the document than with the management of it. And while the notion of hyperlinks does indeed allow some kind of semantic linking, its primary purpose was to reference related documents for easier navigation.

3.3.2 Server-side scripting - Dynamic Web

Due the inconspicuous but very essential fact that every (HTML) document is retrieved by a separate request to a web server, it was only a matter of time until researchers started to create those web sites on the fly. There is and has been a whole panoply of different approaches, including *Server-side includes*, *CGI*, *Java Servlets*, *PHP* etc.

Today, web developers rarely use one of these technologies directly but rather make use of one of many available frameworks that are built on top of those technologies. Popular examples of such frameworks include *Struts*[1], *Ruby on Rails*[2], *Google Web Toolkit*[3] and the *Spring Web Framework*[4].
Research in this area of dynamic web site creation is primarily centred on (semi-)automatic generation of web sites from a database or any other structured data source.

3.3.3 Client-side scripting and Ajax

As the name implies, client-side scripting is a technique that allows scripts embedded into web pages to be executed on the client side within the client's web browser. Scripts can be embedded directly into HTML or included from a separate file. Client-side scripting also became widely known as part of the Dynamic HTML concept and DOM-Scripting.

> *"The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page."*

The main purpose of client-side scripting is to enrich the user's web site experience by providing additional interactivity or animations. Also very common is the binding of code to specific user-generated events such as clicking a button, a link or hovering over an area. This programming paradigm, also known as event-driven programming, has been very successful with regard to traditional client applications. That is why it got introduced for interactive web sites as well. However, for the sake of security, scripts have very limited access to the client's computer resources outside the web browser and are mostly intended to manipulate the content of the page currently displayed.
Today, the most common scripting language is JavaScript, originally developed by Brendan Eich of Netscape. Here, technological advances were primarily a result of the "browser-war" in the late '90s between Netscape and Microsoft. After the introduction of JavaScript into Netscape 2.0 in March 1996, it did not take long for Microsoft to come up with a supposed-to-be compatible dialect called JScript. And while the scripting language has been standardized since 1997 under the name ECMAScript, JavaScript (as implemented by Firefox) and JScript (as implemented by Internet Explorer) expose some subtle differences and may not even have the same set of properties. For creators of web page with client-side scripting, it is therefore necessary to either stick to standards-compliant code only or to adopt the scripts for several script engines.
In the beginning, JavaScript and other scripting languages were conceived as nothing more than gadgetry and indeed, their first applications were basically annoying news tickers, pop up windows and some more or less fancy animations. What was missing was a way to access additional data and to embed that data into the current web page. The famous XMLHttpRequest object, at first introduced as an ActiveXObject by Microsoft and later adopted by all major browser vendors, fulfilled this job. Invoked from a scripting language, it can be used

[1]http://struts.apache.org/
[2]http://rubyonrails.org/
[3]http://code.google.com/webtoolkit/
[4]http://www.springsource.org/webflow

to request additional data from the server side and handle that request asynchronously to integrate the resulting data into the web page by manipulating its DOM. This de facto standard became an integral part of many dynamic web applications and was later coined Ajax (Asynchronous JavaScript and XML). It is not limited to XML though and can be used to transfer any textual data, most notably JavaScript Object Notion (JSON), an alternative way of representing objects textually which is less verbose than XML. Yet so simple in terms of functionality, the possibilities were (almost) endless. With deferred loading of content and the ability to process user requests without having to reload the whole page, it changed the appearance and functionality of many web applications considerably. Ever since, the web has also become a platform for desktop-like applications.
As a matter of fact, the web interface that is part of this thesis is based on these techniques and also uses XML as the data exchange format to communicate with the server-side. Implementation-specific details can be found in chapter 6.

3.3.4 RIA Frameworks

Rich Internet Application frameworks claim to be the next step in web browser evolution. The term was originally created in the late 90s to contrast the limited functionalities browser offered at time with respect to media and user interaction. The preferred mean to overcome these limitations is to offer an additional application runtime which is installed on the user's computer in the form of a plugin, sandbox or virtual machine. Applications built with such a framework aim for a user experience usually offered by desktop applications. Characteristics of such applications may include: increased *performance* by local processing where applicable, support for various *audio and video formats* including streaming capabilities, browser-independent *look & feel* or access to additional *peripheral equipment* such as webcams or microphones.
Nowadays, there are four different plugin-based RIA frameworks worth mentioning. Flex by Adobe which is based on MXML and Action Script, Silverlight by Microsoft which is based on XAML and .Net, OpenLaszlo by Laszlo which is based on LZX and JavaScript and last but not least, JavaFX by Sun which is based on Java and a new declarative syntax. A thorough analysis of these frameworks has been performed in chapter 5.

4 Design and Architecture

This chapter presents the design patterns and approaches that have been chosen to accomplish the goals specified in section 1.1. We will start with an introduction to Web 2.0 principles and show how they have been adapted for PIM. In a second step, we propose a component architecture for data management that incorporates some of the observations we have made with traditional PIM tools, as well as modern and extensible Web 2.0 application used for the same purpose. We then present an extensible, web-based user interface for the creation of customized, pluggable applications to manage personal data.

4.1 Web 2.0 Principles for Personal Information Management

Motivated by the huge success of Web 2.0 applications such as Facebook[1] or Ning[2] where users can assemble their personal portals from a large variety of applications or components, we would like to apply these concepts to PIM in general. Throughout this chapter, Facebook is taken as an exemplary Web 2.0 application because on the one hand, it is one of the most prominent and well-known social networking communities and on the other hand, it was one of the first web sites of its kind which offered an API to extend the system with custom applications.

Facebook Application Facebook provides a complete platform to develop applications that extend the basic functionality of Facebook and build upon its user data. The Facebook Platform consists of several core components: An API to access its services, the Facebook Markup Language (FBML) to deeply integrate the applications with Facebook's ecosystem, the Facebook Query Language (FQL), an SQL-style interface to query Facebook data more easily than through the Facebook API and Facebook JavaScript (FBJS), Facebook's modified JavaScript language that on the one side allows developers to add JavaScript

[1]http://www.facebook.com
[2]http://www.ning.com

capabilities to their applications described in FBML and on the other side prevents them from performing malicious operations or compromising privacy settings. Another important point of Facebook applications is that they have to be hosted externally on separate servers. The API itself is exposed as a RESTful web service and a number of client libraries exist for various programming languages that abstract the HTTP interaction completely. First and foremost, the API allows accessing the profiles of the users, their friends, photos and events. However, most operations are read-only and it is impossible to change Facebook's core data set directly (for obvious reasons). With the FBML, Facebook offers several integration points to hook applications into all major UI elements, for example as a profile box on the left, as a bookmark symbol in the bookmark tab at bottom or as complete application tabs. FBML is a derivation of the HTML dialect, extended by Facebook-specific tags. The result is converted to HTML by Facebook and rendered on the client-side. So, although applications are hosted externally and all communication with Facebook happens via its API, they can be integrated seamlessly with the Facebook Platform and therefore become part of the users' Facebook experience.

However, Facebook applications are completely separate entities and while they of course share some common data, most applications do have their own databases to manage additional data. The only common denominator is the Facebook core data set which, in addition, cannot be edited by third-party applications due to security constraints. Therefore, it is not possible for any application to leverage the data of another application for its own purposes. The current situation is depicted on figure 4.1

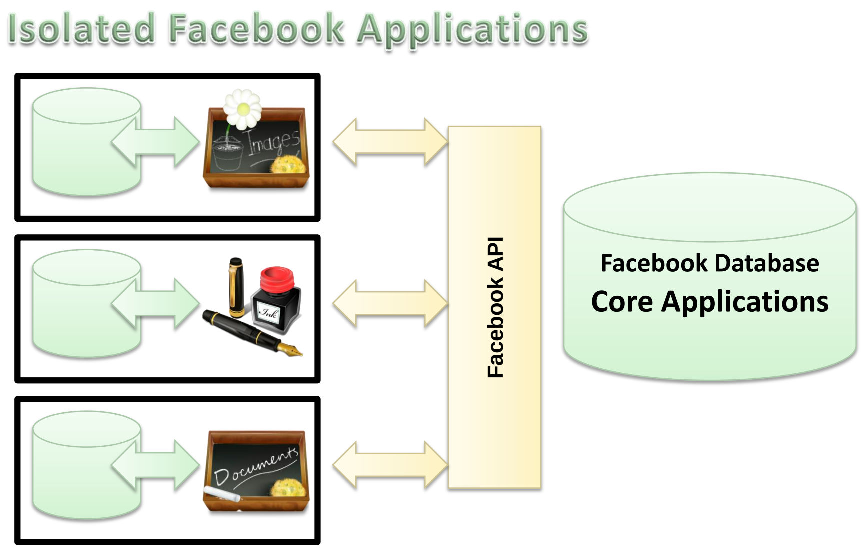


Figure 4.1: Current Situation

In addition, users do not have the possibility to customize *how* the data is actually managed. For example, in Facebook it is not possible (yet) to extend the kind of personal information that can be entered into one's profile or to define what data should be stored along with a picture. Although they try to cover as much as possible and even have fields for favourite music, movies, books et cetera, what if users also want to store they favourite food? Or in

terms of Facebook photos, may it not be desirable to display the date when the picture was taken as well?
In order to overcome these limitations, we aim for a system that on the one side supports the reuse and customisation of data and on the other side offers a web-based, pluggable interface architecture for personal data management similar to that offered by many Web 2.0 applications. The integral parts of the system and its overall architecture are depicted in figure 4.2. As can be seen in the picture, the web application follows a client-server architecture. That architecture can be divided into three distinct parts:

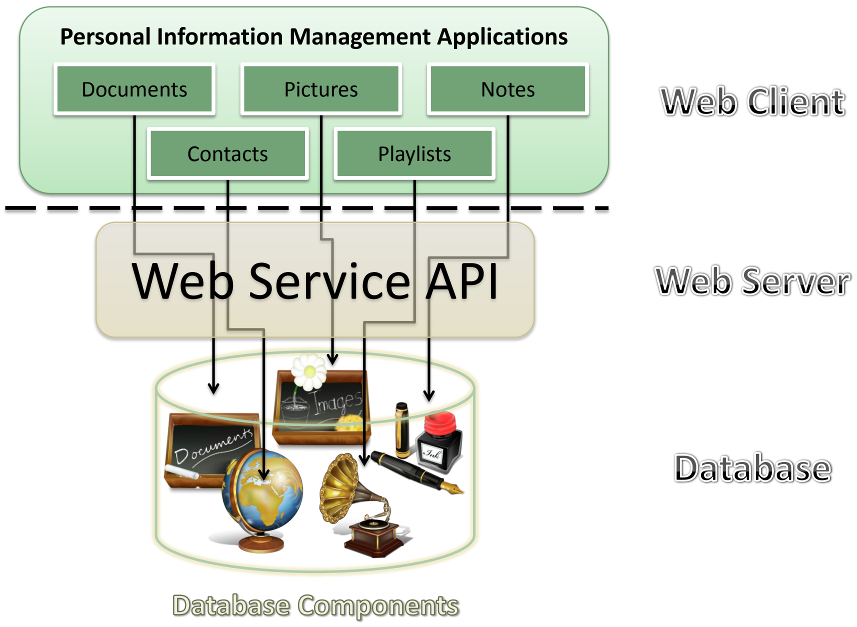


Figure 4.2: Database Components

- A component architecture that reflects the notion of components on the data management level. In particular, it should leverage the use of a single database for enhanced personal information management.
- A web service on top of the component architecture that maps concepts and operations from the database layer to a REST-style web service.
- On the user-side, a web interface that can be accessed from any browser and allows the users to create applications for PIM on their own. The resulting applications can then be used to manage personal data.

4.2 A Component Architecture for data management

Traditionally, PIM has been the realm of the desktop-based tool Microsoft Outlook. There are a few other tools such as Mozilla Thunderbird and Chandler, but none of them can match Outlook in terms of functionality. Interestingly, many of these tools evolved around e-mail as *the* digital personal information everybody wants to manage. In contrast to desktop applications, Web 2.0 applications for PIM are usually built around the idea of friends or contacts in general with the notable exception of Google's web-based application suite that includes PIM tools such as Google Mail, Google Calendar or Google Docs. The design of the component architecture which was built as part of this thesis is motivated by the following important observations and experiences that have been made with PIM tools, both traditional ones and Web 2.0 applications.

- Most applications manage data of a *specific* application domain. These applications are limited to a pre-defined set of concepts or types and are specifically geared to deal with those types of information such as contacts, pictures, organisations or books. However, the boundaries between their domains are not necessarily sharp and overlapping can occur which leads to the next observation.

- Several different applications may manage the *same* data. For example, an application that manages cities that users have visited may contain exactly the same cities that are also used by the application that provides geotagging for pictures and annotates them with city names. Despite that fact, each application manages its data separately. Even worse, applications may manage their data not in the same database management system or not in a database at all. This causes the same set of data (for example countries) to be recreated over and over again. Additionally, reusing the same data is complicated even if there is an intention to do so because customary DBMS are not tailored to data reuse. Instead, applications are usually coupled one-to-one with a single database or enforce their own proprietary data format. This results in a strong entanglement of applications, their data model and the data itself which prevents other applications from accessing that data in straightforward manner. For example, Outlook manages not only emails but also contacts, calendar entries and task lists. Imagine that you may want to tag persons from your Outlook address book on pictures managed by your favourite picture application. But Outlook stores contacts and other personal data in a format called Personal Storage Table (.pst). Unfortunately, the PST file format is proprietary and thus cannot be read directly by third-party applications. This makes the development of applications that try to leverage personal data managed with Outlook very difficult. While it is natural in a distributed environment that each application has its own data storage, in personal information management, the exact opposite is desirable. Ideally, all personal information is contained in one large database. There are a number of advantages of such a system. Firstly, integration happens at the database layer which means that the data can be accessed and queried uniformly. Secondly, data from different applications can be reused. Lastly, data can be associated and annotated independently of the application that created it.

- Not all applications have full access to shared data. In Facebook, third-party applications have access to the core data of Facebook but this access is restricted to read-only for various reasons. First of all, not all third-party applications can be trusted and therefore it is reasonable to have some kind of security mechanism in place to prevent malicious operations or data theft. From a business perspective, Facebook has a strong interest in keeping user profiles and other data for themselves in order to make money through advertisements and offers tailored to the users' particular needs. Not surprisingly, they are promoting their own platform by forbidding external applications to edit the Facebook core data in any form. Within an ideal PIM environment however, data sharing is highly preferable. Nevertheless it shall be possible for an application to specify which data is visible and can be reused by other applications.

Those observations led to the following requirements for a component architecture:

Encapsulation The system has to support the encapsulation of single application domain models. Since this system will use the OM data model (section 3.1) to define schemas, the system shall provide the possibility to encapsulate entire OM models. Until now, there is no metamodel construct to enclose such models. This means that if several applications build on top of the same database, there is no straightforward way to tell which schema elements belong to which application.

Reuse of existing data In personal information management, redundancy which is not synchronised and duplicates that are not linked to the original data are recurring evils. The good news is that they can be avoided to some extent when the data is shared by all applications that manage the user's personal data. The use of a single database for PIM purposes can be assumed because encapsulation has been advocated as a necessary requirement of the system. With that assumption in mind, the system shall provide the possibility to reuse the same data by combining or reusing existing components that are within the same database.

Modifiers The requirement for encapsulation of data models on the one hand and for data reuse on the other hand makes it necessary to specify clear interfaces between database components. Similarly to Java access modifiers, the system shall provide the possibility to define which parts of components can be reused by other components.

4.2.1 Information Components

In this section, we propose an architecture that fulfils the requirements described above. We introduce the idea of "Information Components" and present a metamodel that describes the parts of such a component on the database layer.

Definition 1 (*Information Components*)

Information Component is the name given to the idea of incorporating the concept of reusable components into the data management layer. Similarly to database schemata, an Information Component groups all model entities that belong to the same application domain. However, it is not another modelling technique but rather an additional abstraction on the data management layer to separate application domains that share the same information space, i.e. the same database. Information Components are comprised of a number of parts. These parts are model entities already introduced by the OM model, namely Collections, Association and possibly additional entities such as Constraints. The application domain modelled by a single Information Component can thus be described by a formal OM model.

We came to the conclusion that if we move the applications or rather their application models to the database layer and encapsulate them into "components", we will be able to combine data from different application domains much more easily. Figure 4.3 depicts the OM model

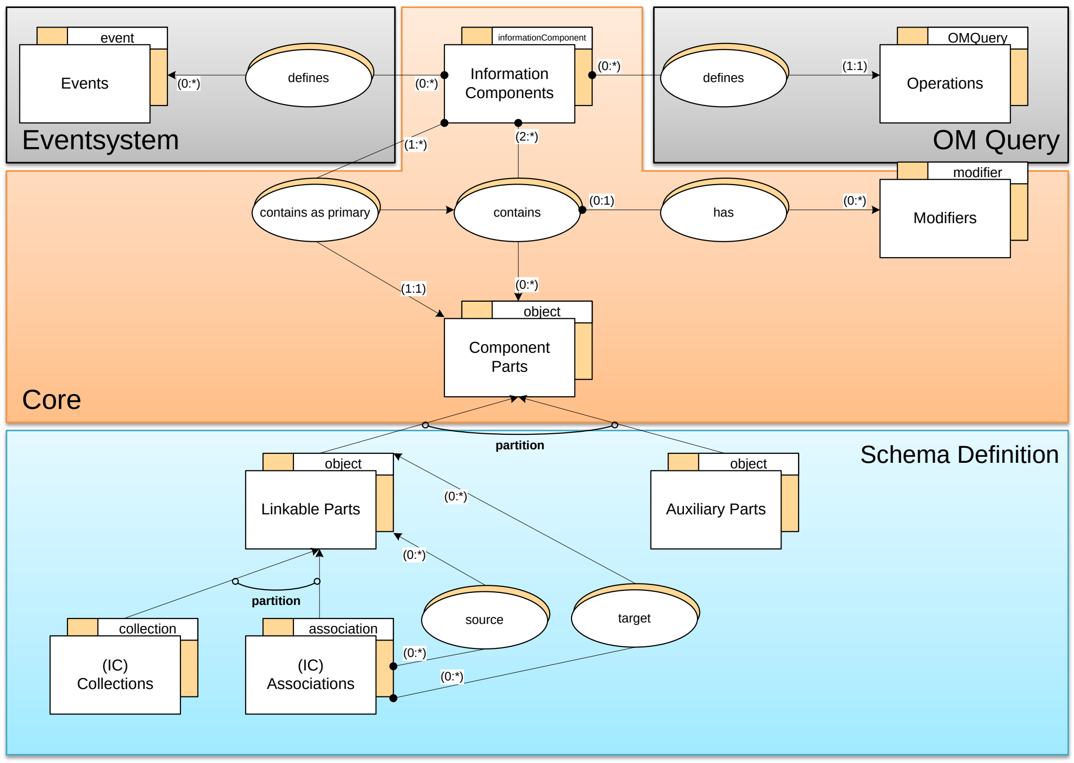


Figure 4.3: Information Components - Meta Model

of the Information Components. The core of this metamodel is basically very simple. An Information Components consists of several parts. Each component part is connected to exactly one Information Component through a `contains as primary` association. This Information Component is the "owner" of that component part. All parts have to be encapsulated by exactly one Information Component. Through the `contains as primary` association, it is always traceable which parts have been defined by which Information Component. Data reuse is modelled by the `contains` association. Whenever an Information Component reuses another component, the parts of the included component become also parts of the including component, this time however, membership is expressed with the `contains` association only. Of course, it is also possible to reuse only individual parts of other Information Components.
The benefit of this approach in contrast to a reuse-relationship between Information Components directly, is the finer granularity. Annotations to the member- and ownership of parts is now possible on a per-contains basis. For example, modifiers of a component part are associated directly with its `contains` association. This allows for the redefinition of modifiers whenever a part is being reused. These modifiers are used to express visibility constraints that define whether component parts can be reused again.
In the Schema Definition part of the model, one encounters already familiar concepts, namely `Collections` and `Associations`. They are modelled as subcollections of the more general notation `Linkable Parts`. This expresses the ability of associations to create a link not only between collections but also between associations in order to create nested

associations. Additionally, it becomes obvious that associations are only allowed between `Linkable Parts` of the same Information Component. This prevents the violation of the encapsulation principle enforced by Information Components. The `Auxiliary Parts` collection is meant to serve as an extension point for additional parts of an Information Component that might be included in the future, for example subcollection constraints.
It is important to note that the Information Component metamodel operates on the classification layer only. Type definitions are not part of an Information Component but are shared amongst all components in order to promote reusability. The `Events` part and the `Operations` part of the model are currently surrogates for possible extensions of this model. Suggestions include the Event system, developed by another student as part of his master thesis [11], to implement events and OML to implement operations.

4.3 An extensible, web-based user interface

One of the goals of this thesis is to empower users with the ability to create data management applications on their own. In section 4.2, an architecture has been proposed to add the necessary functionality to the data management layer. This section shows the design and approach taken to realise the idea of Information Components on the user interface layer.
During the analysis of Web 2.0 applications in section 2.2.4, it was stated that one of the main advantages of Internet applications over traditional desktop applications is that they can be accessed from almost everywhere. It is therefore reasonable to argue that a web-based client is also appropriate for PIM. As a consequence, it was already defined as part of the master project definition, that the user interface layer shall be implemented with any Rich Internet Application (RIA) framework. Several different web application frameworks have been evaluated and a plug-in based RIA architecture has been found to be the most appropriate, mainly because it offers the same responsiveness that users already expect from traditional desktop applications. A thorough evaluation of major plugin-based RIA frameworks can be found in chapter 5. The resulting web application has been coined *MyApp Builder* and is accessible with any web browser that has the Flash plugin installed.

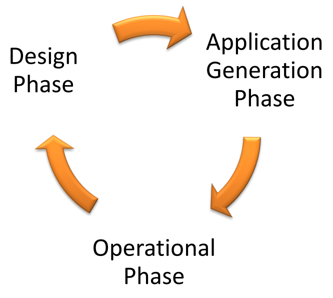


Figure 4.4: Work Flow

MyApp Builder takes a model-first approach which means that users develop the model and define types before any data is created. The UI incorporates that approach by providing an integrated interface for both the creation of Information Components as well as the management of data described by those components, their classification models and their type definitions. The general work flow can be best described with the workflow picture 4.4. Interaction with the application is usually divided into three distinct phases. A design phase followed by an application generation phase and concluded by an operational phase. At any time during operation of the system, users can start a new development cycle.

Design Phase During the design phase, users create and edit application models that describe the personal data they want to manage afterwards. This includes the definition of types and their properties to represent data, the specification of categorization schemes with the aid of collections and the creation of associations to express relationships between members of those collections. From [9], it is known that today's interfaces to databases expose several shortcomings in terms of usability. The design of *MyApp Builder* resembles traditional, graphical database management tools but tries to be more user-friendly, especially for non-technical users. While the notion of types and properties is comprehensible for most users, many still have problems to grasp the idea of joins in relational databases. Fortunately, the OM data model defines associations over collections as first-order entities. For this reason, it is only logical that the web interface presents the same concepts to the users, albeit with a limited set of possible operations. The system is therefore extensible with regard to new application domains for personal information. In addition, it is possible to include collections and associations from other applications and combine them to form a new application altogether. In a first step, for example, users may have created an application to manage personal contacts and another to manage their pictures. After having entered data into both applications, users may want to be able to indicate who is visible on which pictures. So they might want to take the contacts application and the picture application as their ingredients and blend them together by associating people with pictures. The result is then a new application that allows contacts to be tagged on pictures. The design phase is followed by the application generation phase whenever the users consider their application schemata to be ready for operation.

Application Generation Phase The application generation phase is the only phase that, once started, requires no explicit user interaction. In this phase, a UI is generated according to an application domain model created in the design phase. In contrast to the other two phases, this step runs entirely on the server-side and results in a graphical application that can once again be accessed by the user's web browser. This is a relatively novel approach and differs from the traditional way of generating UI dynamically. Normally, dynamic UIs are created at runtime either on the client-side or the server-side. On the client-side, this is usually done programmatically with the programming language a particular client framework offers. In the context of (dynamic) web sites, it is usually done on the server-side as part of the web site creation. With either approach, there are a few possible drawbacks. Usually, dynamic UIs or web sites are generated over and over again, even if the underlying model has not changed and thus the result remains the same. Additionally, it is often more tedious to create UI components programmatically than to describe them declaratively. This is why several web frameworks ship with a template engine that combines languages to describe web sites declaratively with the programming language of the web framework. These template engines mostly features markup languages that target HTML and are therefore suitable only for the presentation layer. However, most RIA frameworks offer a user interface markup language that provides markup code which is not limited to purely representational tasks but may also contain language elements that describe part of the application logic. Based on such a RIA framework, we propose a hybrid approach that tries to generate code for a UI markup language on the server-side but also tries to leverage the declarative language of that framework to improve interactivity and to include additional functionality on the client-side. We claim that such a system offers several advantages:

- Structures that do not change at runtime can be created statically on the server-side and do not need to be re-created as long as the domain model stays the same.

- Because many parts of the UI have already been created statically with information from the domain model, there is less need to send metadata about the domain model to the client during the operational phase.

- Wherever adaptability on the client-side is required at runtime, an appropriate declarative statement or a script block can be injected on the server-side already at design-time.

In order to overcome the inherent limitations of the automatically generated applications in terms of functionality and representation, we aim for a plugin system that allows customized representations to be registered with specific types from the domain model. The system is therefore extensible with respect to new representations for types.

Operational Phase The operational phase is the standard phase and describes the normal daily interaction with the system, after the users have successfully created one or more applications. It is the phase where users actually manage their personal data. Entities from the application domain can be created, edited and grouped into collections. Associations between them can be established and dissolved again.

5 Web Technology Evaluation

For the database layer, the choice of suitable technologies was limited and general conditions as defined in the task description had to be applied instead to implement the component architecture described in section 4.2. For the web interface however, the choice of an appropriate technology was left to the author of this thesis. This chapter starts with an analysis of the basic requirements for the proposed web interface architecture and then gives an overview of the web technologies that are currently available to realize such an architecture on the server-side as well as on the client-side. Several different technologies and their supporting frameworks have been analysed as part of this thesis and for each of them, a general introduction is given, followed by a breakdown of its properties into advantages and disadvantages which is then concluded by an assessment of their practicability for this particular system.

5.1 Requirements Analysis

Before web technologies have been selected for the evaluation process, a requirements analysis for the web application has been performed to identify the key features the web application has to offer. These requirements have been derived from the initial master project description and the proposed architecture in the last chapter. The basic functionality has been captured in the following scenario:

Scenario Within the web interface to be created, users shall have the possibility to create the domain model for the data they want to manage afterwards. For example, users might want to manage their personal contacts, so they define a `contact` type to have a forename, a surname, an address and an e-mail as its attributes. Then, users might want to have different categories or collections of contacts, for example “Friends” and “Business” contacts. After the users have completed this step, they shall be able to actually manage their contacts in a very simple way, i.e. create and edit contacts and add them to the `Friends` or `Business` categories. The user interface for those tasks has to be generated automatically and shall leverage existing UI components from the RIA framework whereas possible. Moreover, there shall be the possibility for power-users to develop their own UI components for certain types, for example they want to create a custom view for a `contact` that allows the user to send an e-mail to that contact directly from within the view. These custom views therefore contain not only information about the presentation of types but may also contain additional functions to define new control and business logic.

From this very simple scenario, three important requirements for a possible RIA framework have been concluded:

Communication with a Java-based back-end Eventually, all data (and metadata) is stored in and retrieved from a database. The RIA framework shall provide facilities that make communication and data retrieval with the server as easy as possible. Additionally, it has to support an exchange message format that is human-readable for debugging purposes. If possible, the necessity to create proxies or stubs to access a web service should be avoided.

Create UI dynamically In order to provide a UI to the users after they have created the Information Components, a corresponding UI has to be created automatically. Two distinct approaches have been identified to accomplish this task in a feasible way. Either UI components are created programmatically on the client-side with the programming language supported by the RIA framework or there is a way to generate and/or pre-compile UI components on the server-side. With the former it is necessary that metadata has to be sent from the server to the client whereas the latter requires some kind of preparatory step on the server-side after the Information Components have been created via the user interface and before the generated interface can be accessed. However, it has already been concluded in section 4.3, that a hybrid approach has several benefits. In any way, a possible RIA framework has to support at least one of these approaches.

Plugin system for different representations It should be possible to store complete components/containers in the database including their application logic. That would enable the web application to have some kind of plug-in/widget system, where users can load different views for their data. This is illustrated in the following, very simple example. The user might have created a very basic `Picture` entity. A picture entity defines only a name, a list of tags and an URI. The automatically generated UI might display the URI only as a string, whereas the user probably would like to display the image the URI points to. However, it is most likely very simple to create a PictureViewer component in the given RIA framework that actually displays that image given its URI. Ideally, this UI component should be embeddable into the user interface without having to manually include the component in the web application project and compile and deploy it again after every minor modification to one of these plug-ins/widgets.

5.2 Choice of RIA Framework

The Web has become a platform for all kinds of application. Gone are the days where web sites were nothing more than some simple HTML text documents interspersed with links and

pictures. Today's web applications already compete with conventional desktop applications and are becoming more and more popular.
Similarly to Ajax, the term *Rich Internet Applications* (RIAs) does not refer to a new technology but rather describes web applications that try to provide a desktop-like user-experience. Originally, RIAs always required some kind of browser extension to overcome the limitations of web browser at that time with respect to overall appearance of application interfaces, media support and limited user interaction. Nowadays, JavaScript has matured and has become a serious competitor to those plugin-based frameworks. This trend has also been recognised by all major browser vendors and in 2008, they have started to optimise their JavaScript execution engines. Just recently, Google has entered the browser market with their own development of a browser engine that has been explicitly tailored to JavaScript applications. That is not surprising if one considers the vast amount of JavaScript applications offered by Google itself. A second important development is HTML 5 which is currently developed by W3C. A third version of the working draft has been published on the 12th of February 2009. Among other things, HTML 5 introduces new tags to embed multimedia directly in documents and has a new canvas element that offers several methods to draw 2-dimensional objects. However, it is still a work in progress and browser support is limited to the most stable subset of it.
Current trends indicate that JavaScript is very likely to become the dominant technology for web applications within the next few years. But despite the current progress in that area and the performance gains of recently released JavaScript engines, JavaScript is still slower than most plugin-based frameworks, primarily because JavaScript is an interpreted language whereas other solutions are built on a browser extension that runs natively on the client. This is why a decision has been made in favour of a plugin-based RIA framework. Four different plugin-based RIA frameworks have been considered: Flex by Adobe, Silverlight by Microsoft, OpenLaszlo by Laszlo and JavaFX by Sun Microsystems.
It is important to note that only out-of-the-box or built-in features of each framework were considered while the evaluation was conducted. Therefore, it is possible that third-party libraries exist which implement missing functionality for the corresponding framework. However, evaluating the enormous amount of additional libraries that are available for each RIA platform was beyond the scope of this thesis.
Over and above, a small set of features that is usually closely related to web applications was excluded from this evaluation because they are irrelevant or not important enough for this project. These features fall into two categories:

Media capabilities In contrast to HTML and embedded JavaScript, plugin-based RIA frameworks support a wide variety of *media types*, for example different music formats (MP3, Ogg Vorbis, WMA etc.) and video codecs (Flash Video, H.264, WMV etc.). Most importantly, they offer facilities to *stream* those types from the server to the clients and thus allow them to start the playback of music and video files before the whole content has been downloaded. Closely tied to that subject are *content delivery services* offered by the company behind the corresponding RIA platform and which are intended to deliver high volumes of data to customers.

Platform-dependent facilities and services Every platform-dependent feature that only works with a specific RIA platform has not been considered for this evaluation.

This includes but is not limited to: *Platform specific message formats* (i.e. Adobe Message Format), *Platform-specific web services* (i.e. Adobe LifeCycles, OpenLaszlo Server, Windows Communication Foundation web services, ASP.NET), *Media conversion services* and support for *direct invocation* of remote objects (i.e. Java classes, C# assemblies).

The evaluation starts with an introduction to plugin-based RIA architectures. Subsequently, some of the mechanisms and concepts that are characteristical for such frameworks are presented. What follows is a detailed description of each of the four plugin-based RIA frameworks that have been analysed as part of this evaluation. The section is then concluded by a report of the evaluation process and the final decision.

5.2.1 Plugin-based RIA Architecture

As the name implies, such architectures require an additional piece of software to be installed on the client that extends the web browser's capabilities. Their size can range from very small packages with only a few kilobytes to complete application frameworks with over hundreds of megabytes. However, they are downloaded and installed only once with the exception of updates. With respect to RIA frameworks, plugins are required to execute applications written for the corresponding platform on the clients' machines. A general illustration of such an architecture is given in figure 5.1. It is an adaptation of an illustration from the Web Engineering lectures [7].

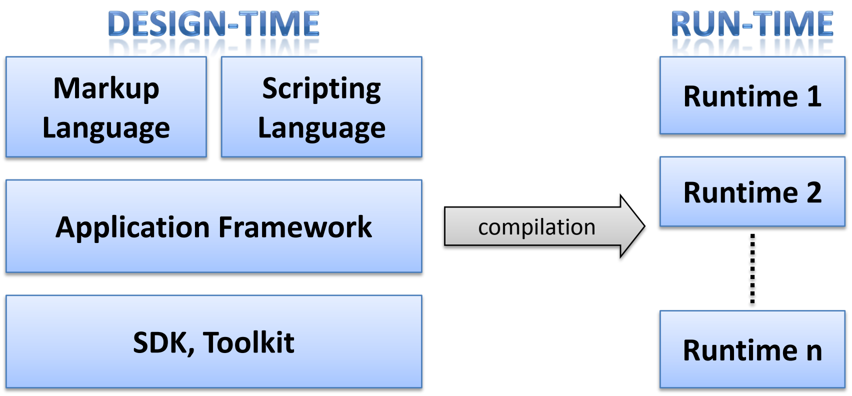


Figure 5.1: Rich Internet Application Architecture

5.2.2 Common Mechanisms

RIA frameworks differ from traditional, desktop-based application frameworks not only in terms of functionality but also in terms of programming concepts. This section will list some of the most distinctive concepts and features that RIA frameworks offer. They support rapid development of visually rich applications and help making some reoccurring programming tasks more manageable and easier to cope with. Of course, these design principles and features are not exclusive to RIA development, but can also be found in other frameworks. However, they are regarded as essential ingredients for any RIA framework and can therefore be found in all major systems, although with subtle differences.

Declarative Language

A declarative style focuses on the *what should be accomplished* rather than on the *how it is accomplished*. Instead of explicitly specifying each single step in a sequence of actions needed to reach a certain (computing) goal, the declarative programming paradigm emphasises on describing what the goal is, letting the system decide how to reach that goal. In most cases, they are closely related to a mathematical theory. Popular examples of such declarative programming approaches are functional programming and logic programming.
In the context of RIA framework, declarative languages are mainly used to arrange and describe the user interface, although they may also be used to describe part of the application logic declaratively. They are also known as user interface markup languages and some languages already exist for many years, albeit with various success. To some extent, such declarative languages can be compared with HTML which also describes the layout of web pages declaratively and it is up to the browser or the layout engine to decide *how* to render a specific HTML document.
Figure 5.2 shows an excerpt from a program written in OpenLaszlo and the resulting UI rendered by the Flash runtime on the client-side.

```
<canvas width="200">
    <window x="10" y="10"
        width="150" height="150" >
        <button>Hello World!</button>
    </window>
</canvas>
```

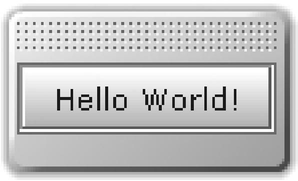


Figure 5.2: OpenLaszlo Example

As with programming languages, different declarative languages and syntaxes exist and each RIA framework has its own language with certain advantages and disadvantages. OpenLaszlo was one of the first publicly available frameworks that supported the development of web interfaces declaratively using their own XML-dialect LaszloXML (LZX) in conjunction with JavaScript. Second in line is Macromedia which introduced MXML as another XML-based interface markup language that works together with Action Script to develop RIAs for the Adobe Flash platform. The Extensible Application Markup Language (XAML) by Microsoft is part of their Windows Presentation Foundation, the graphical subsystem of their .NET Framework 3.0. It is also the declarative language for their RIA framework Silverlight. Surprisingly, the newest entry in the RIA framework market, JavaFX by Sun Microsystems, does not feature an XML-based syntax but a less verbose syntax called JavaFX Script that shows a significant resemblance to JavaScript Object Notation (JSON). A very simple application has been implemented in all frameworks in order to get some hands-on experience. This application calls a web service API from Flickr that merely echoes all parameters back in the response. Examples of the different declarative languages used in Flex, JavaFX, Silverlight and OpenLaszlo can be found in the Appendix.

Data binding

Data binding is a language feature that allows a variable or property to be *bound* to another variable. As a result, a close relationship is established between the source variable and the

corresponding target variable. Whenever the value of the source changes, an event is triggered that causes all bound variables to be updated as well. The whole update process takes place behind the scenes and no explicit registrations to update events is necessary. This greatly simplifies Model-View-Controller (MVC) interaction because it basically eliminates the need for the controller component to update the view after the model has changed. Developers are relieved from writing the same elementary boilerplate code again and again. Variables can also be bound to composite expressions that are re-evaluated whenever an element of the expression changes.
Data binding is used primarily for UI development to bind properties of UI components to data from the data model. It helps synchronising the view with the underlying model. Furthermore, data bindings and bound expressions are very well-suited to define constraints for and dependencies between UI components such as resizing behaviour, drag constraints and synchronized properties.

Binding direction Usually, data binding is directed from a source to a target. When the source variable changes, the target variable changes too but changes to the target variable are not propagated back to the source. This behaviour is appropriate in all cases where the target is read-only, for example if the target is simply a label that displays some text. For editable controls however, it is reasonable to reflect any changes back to the data model or the bound variable. With bi-directional binding, both source and target are synchronised in such a way, that changes are propagated immediately in any direction.

Binding participants Data binding does not have to be limited to plain variables only but may also allow any computable expression in the corresponding programming language as the bound "property". Expressions may also include functions as long as functions are a first-order concept of the current programming language. Binding to a function requires specifying when the bound function is going to be re-evaluated and hereby causing all variables that bind to this function to update as well. The two most common approaches are to bind that function to another event or to use bound variables as its parameters. With the first approach, the function is executed whenever the event it is bound to is triggered. The second approach forces the function to be evaluated whenever any of its bound parameters change. It is evident that functions and composite expressions can only occur in a uni-directional setting where they serve as data sources. Additionally, collections are also applicable as data sources in binding expressions, especially if they are bound to multi-object UI components such as lists, tables or combo boxes.

Evaluation of bindings By default, bindings are re-evaluated whenever a change occurs. Under certain circumstances, it is more suitable for bound expression to be evaluated only once. For example when components have to be initialised with properties that do not change over time but are unknown at compile-time. This reduces overhead and prevents properties from being changed accidentally.

It is important to note that not all concepts presented above are available in all RIA frameworks. However, not all features have been regarded as essential for the web interface application developed as part of this thesis, but where they have, any limitations and absence of them are mentioned in the subsections of each evaluated RIA framework.

Web service capabilities

Rich internet applications without the Internet would not make much sense so it is vital for any RIA framework to provide a rich set of facilities and mechanisms to retrieve and process data from various remote systems. All frameworks can handle XML Web services and some even support JSON directly. SOAP on the other hand requires an additional step to create stubs from a WSDL file at compilation time.

Vector-based drawing

When the Flash plugin was published back in 1996, it was primarily used for animations because of its extensive support for vector graphics. In contrast to raster graphics, vector graphic formats do not store a picture pixel by pixel but rather represents images with mathematical formulae that define primitives such as points, lines and curves. Due to that fact, vector graphics are scalable to any screen size and produce crisp images at any given desktop resolution which makes them perfect candidates for scalable, appealing user interfaces. However, they are not suited for photographic images because it may be infeasible if not impossible to derive a construction of the image by mathematical equations. The lack of drawing capabilities is indeed a deficiency of HTML. Not surprisingly, it is considered important enough to be included in the upcoming standard HTML 5. In the meantime, vector-based drawing is still the domain of plugin-based RIA frameworks and all of them offer methods to draw points, lines, circles, ellipses and polygons. Furthermore, several gradients and colouring schemes are available to add interesting fillings. Some even provide advanced special effects such as lightning, shadow or emboss.

5.2.3 Flex

Adobe Flex is one of the most popular Rich Internet Applications frameworks due to the success of its underlying Adobe Flash platform. The Flex SDK became open source in February 2008, but the Flash Player runtime as well as its primary developer tool, the Adobe Flex Builder, remained proprietary. Basically, Flex is a different approach to the Flash platform. Instead of a designer-oriented, animation-based design paradigm, a new approach has been created that better suits software engineers and traditional programmers. The graphical user interface is being composed by an XML dialect called MXML where components and elements are declaratively described. All application logic is accomplished with the aid of ActionScript 3, the standard script language used by the Flash Player. ActionScript in turn is an object-oriented script language that origins from ECMAScript.

Features Flex bundles with a comprehensive library of over 100 common components. Those components range from standard GUI elements such as list boxes, buttons and trees to more complex components such as data grids and layout managers. For most components, drag-and-drop is supported out-of-the-box. All animation and effect capabilities of the Flash platform are also available for Flex applications. Flex integrates with Adobe Creative Suite to allow designers to create assets in Photoshop, Illustrator, Fireworks etc. directly for Flex RIAs. With the Adobe AIR runtime environment, it is also possible to develop desktop applications with Flex that are able to access the local file system. Furthermore, applications have the possibility to work online as well as offline in case there is no Internet connection.

Drawbacks While the SDK is freely available, the recommended IDE (Adobe Flex Builder) is a commercial application. In addition, all runtimes are proprietary.

Plugin Applications that have been developed with Flex can be either deployed as Flash applications for the Adobe Flash Player runtime or as conventional desktop application with the Adobe AIR runtime environment. According to Adobe, the Flash Runtime is the most widely used browser runtime and has a user base of 99%[1].

5.2.4 JavaFX

JavaFX is the newest competitor in the relatively young field of RIA frameworks. It is Sun's second try to compete in the market of internet-based applications, after Java applets have failed due to severe security restrictions and the Java runtime being too bloated at that time. Starting as an interpreted scripting language for interactive web applications, JavaFX has now become a general purpose framework for developing (graphical) Java applications. It not only compiles to Java byte code but is also compatible with existing Java libraries. Therefore it is evident that JavaFX applications or applets require a Java runtime to be installed and are executed within a Java virtual machine.

Features JavaFX is different from many other frameworks with respect to its language syntax. Unlike most frameworks, JavaFX's syntax is not an XML language but fancies a rather peculiar, JSON-like syntax. Nevertheless, it is declarative as well and hierarchically organised. JavaFX can also be used to create traditional Java desktop applications more easily. The recommended IDE for JavaFX is Sun's NetBeans editor with the additional JavaFX plugin. It does not offer a visual editor yet but at least provides the developer with a live, interactive preview of the application being developed, similar to Silverlight development in Visual Studio.

Drawbacks There is no built-in XML query mechanism. Instead, XML has to be parsed "manually" and only a pull parser has been implemented yet. Furthermore, JavaFX offers limited support for common web technologies such as SOAP.

Plugin In order to run JavaFX applications, any recent Java Runtime is sufficient. That is why JavaFX runs basically on all platforms where a Java Virtual Machine is available. Additionally, JavaFX applications, even those intended to be run over the internet, can be moved to the desktop and will continue to run within the sandbox of the Java platform but independently of the browser.

[1]http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

5.2.5 Silverlight

Silverlight is Microsoft's answer to the growing market of RIAs. While its first version was merely a proof-of-concept and included only a very limited set of built-in controls, the recently released version 2.0 brings Microsoft back into competition. In fact, Silverlight 1.0 did not include any UI controls but relied on the native browser controls for input. Compatible data formats were limited to XML and JSON and the only supported scripting language was JavaScript running in the browser. However, with the dawn of Silverlight 2.0 (October 2008) it is now possible to use any .Net language to implement the application logic for Silverlight applications.

Features One of the most distinct features of Silverlight is its strict separation of presentation and content. Ideally, no code snippets should be contained in the XAML that describes the UI. Instead, the XAML file should only include hooks to event handlers that are effectively implemented in a separate class. In Microsoft terms, this is called code-behind and describes the fact that all code has been moved to a partial class. Apart from browser application, Silverlight can also be used to create applications for the Windows Vista Sidebar. Those applications are called gadgets and are also compatible with the upcoming Windows 7 operating system (although the Sidebar has been abolished there).

Drawbacks The required plugin/runtime is currently available only for Windows and Mac OSX. However, the Moonlight[2] project (a subproject of the Mono project) is working on a free software implementation for Linux-based operating systems.

Plugin Silverlight requires an additional browser plugin provided by Microsoft. In contrast to JavaFX which requires the full Java runtime, the plugin is relatively small in terms of size and has only a couple of megabytes.

5.2.6 OpenLaszlo

OpenLaszlo (formerly just Laszlo) started as commercial product to develop applications for the Flash runtime long before Adobe even started to develop Flex. When Flex was released in 2004, the company behind Laszlo was forced to make Laszlo OpenSource (named OpenLaszlo ever since) in order to be able to compete with Adobe.

Features OpenLaszlo features its own declarative language which is independent from the target runtime. Currently, SWF8, SWF9 (OpenLaszlo 4.2) and DHTML are supported. Any application logic that cannot be expressed declaratively has to be implemented with JavaScript which is directly embedded into the LZX file. LZX is also a prototype-based language. Any attribute or method that can be attached to a class definition, can also be attached to an instance of that class, thus OpenLaszlo is best suited for rapid prototyping where usually many objects with one-off behaviours exist. Another part of the OpenLaszlo platform is the OpenLaszlo Server. The OpenLaszlo Server is a Java Servlet that dynamically compiles LZX to Flash or DHTML when the first URL request for that particular LZX file arrives.

[2]http://www.mono-project.com/Moonlight

Drawbacks Because of the desired runtime independence, one usually only has the features at hand that are common to all supported runtimes. To leverage a particular runtime is cumbersome and prone to problems when the runtime is changed. OpenLaszlo is also the only platform where no visual editor or instant preview is available.

Plugin OpenLaszlo does not ship with its own runtime system but relies on third-party technologies instead. If the target runtime is Flash, Adobe's Flash Player browser plugin is required. For DHTML, any browser that executes JavaScript is suitable.

5.2.7 Evaluation process and decision

The general evaluation took place from the middle of October 2008 to the end of November 2008. At the beginning of December, several prototypes have been developed for OpenLaszlo and Silverlight, the two remaining frameworks after the general evaluation. At the end of December, a decision has been made in support of OpenLaszlo. The following section details the evaluation process and highlights the findings that led to this decision. Although the frameworks have been evaluated in parallel, they are discussed in consecutive order for reasons of clarity.
Flex is one of the most widely used RIA frameworks and the Flash plugin has a high a market penetration which makes it ideal for RIA development. On the one hand, Flex offers very mature and robust components that have been developed and improved for several years. On the other hand, an impressive IDE is available (Flex Builder) that allows user interfaces to be designed interactively in a drag & drop fashion. Unfortunately, Flex could not be much investigated into because its primary developer tool, the Flex Eclipse Plug-In, was available only under a commercial licence which is not acceptable for research project. It was only discovered later that there is indeed a way to apply directly at Adobe North America for a student license, but at that time it was already too late to re-include it in the evaluation process.
Quite some time was spent to explore JavaFX, which was released as a preview version at the end of July 2008 and seemed very promising. However, at that time, the API was still not finished. Most notably, two different design approaches for the creation of user interfaces existed in parallel and it was not sure, which methods and classes will make it into the final release. Furthermore, several libraries and their documentation were just incomplete. For example, there was no grid or table construct because it has been removed for the preview version and XML & JSON support was not complete either. At the beginning of December 2008, the final version has seen the light of day but again, it was too late to re-evaluate the product.
The situation was similar with Silverlight. During the period of the evaluation, Microsoft published the release candidate of Silverlight 2, so there was also a chance that applications developed for this release will not work with the final release unless their source code is adopted to the new version. One of the great characteristics of Silverlight 2.0 is that it implements the Common Language Runtime of the .NET Framework. This means that the application logic behind the user interface described in XAML can be written in any .NET programming language. With the Silverlight Tools for Visual Studio installed, Visual Studio can create Silverlight projects and thus offers code completion and UI preview for XAML as well as the statically typed language C#. Silverlight works very well with SOAP, because Visual Studio

can automatically generate classes and stubs based on a WSDL file. The extensive, and very well written documentation is another plus factor of Silverlight.
OpenLaszlo on the other hand was not that appealing at first. It is the only product which offers neither a visual GUI editor nor an instant preview feature for its markup language. Of all the frameworks, OpenLaszlo integrates best with Eclipse, the IDE used for the rest of this project. Not only is there a third-party Eclipse plugin available that provides syntax highlighting and rudimentary code completion, but the OpenLaszlo server architecture itself is in fact a Java Servlet that can be deployed to any servlet container. With the integration of such a web application server into Eclipse, development cycles became extremely short and any changes to the client's source code were reflected almost instantly on the web server accessed locally by a browser. Additionally, the data model of OpenLaszlo is very simple because all the data is represented both internally and externally as plain XML. Apart from the faster development cycle, OpenLaszlo features an instance-first approach that facilitates prototyping. Any element that can be part of a class definition can also be part of an instance of that class instead. Of course, this is also true for elements that define attributes and methods. Due to that circumstance, developers can add new methods and attributes directly to instances for one-of-a-kind behaviour. In case it turns out at a later stage during application development that an instance should become a blueprint for other instances, a promotion to a class is done within seconds.
At some point in time, a choice had to be made between Silverlight and OpenLaszlo. Silverlight seemed to be more appropriate where the web service interface was already fixed whereas OpenLaszlo seemed to be more flexible and more adapted for iterative development. The latter appeared to be a better choice for a research project where ideas are constantly re-evaluated so the decision was made in favour of OpenLaszlo. Besides, a vote for Silverlight was also made dependent on the availability of the second milestone of *eclipse4SL* (Eclipse Tools for Microsoft Silverlight). Eclipse4SL is an open-source project that aims to integrate the development of Silverlight applications with the Eclipse IDE. The second milestone was scheduled for December 2008 and promised to facilitate interoperability with Java web services. Unfortunately, it was released not before middle of December. At that time, OpenLaszlo has already been chosen as the implementing RIA framework.

5.3 Choice of Web Service Architecture

In order to connect the web interface on the client-side with the Avon database on the server-side through the Internet, the implementation of a web service is inevitable. This section presents three different back-end technologies that have been considered suitable to work on top of the Avon database layer and the Information Component module. In addition to the functionalities described in the requirements analysis above, a few functional and technical constraints limited the choice of an appropriate architecture. These are as follows:

- Java shall be chosen as the implementing language because the API of Avon and the Information Component module is available in Java only. This also means that the resulting web service shall be deployable to any JavaEE application server such as JBoss or GlassFish.
- As its transport protocol, the web service shall use HTTP. This guarantees interoperability with all available RIA frameworks.

- Maturity of the web service framework is highly preferable and wherever necessary, the decision shall be in favour of a stable, simpler framework instead of a more sophisticated (but potentially unstable) one.
- Integration with Maven, in terms of build cycle and repositories, is optional but preferable.

As part of the evaluation process of this thesis, three different web technologies have been analysed more closely: **Java Servlets** (part of the JavaSE and JavaEE standard library), **Struts 2**[3] (an open-source project by the Apache Software Foundation), and **Enunciate**[4] (open-source, developed by Ryan Heaton). For each framework, a description of its core functionality and its distinct features are given. This section is then concluded by a report of the evaluation and selection process.

5.3.1 Java Servlets

Java servlets are the most basic pieces of web technology that are available in the Java world. A single servlet is an instance of a Java class that implements the Java Servlet API. Similarly to the Common Gateway Interface (CGI) approach, HTTP requests are not served by the web server directly, but are handed over to a servlet for further processing. The servlet's response is than sent back to the requesting client. Dynamic websites can be generated that way and arbitrarily computations as well as database access on the server-side are possible. To that extent, Java servlets are similar to PHP and Perl. Concerning response messages, any data can be sent back to the client, though HTML and XML are the most common data formats.
Java Servlets require a specialised web server called *servlet container*. Such a servlet container embeds the Java runtime into a web server and is responsible for initialising all registered servlets, the mapping of URLs to the appropriate servlet and dispatching incoming requests. Additionally, the servlet container manages the whole life-cycle of a servlet. This includes initialisation at the start, request handling during ongoing operation and destruction at the end.
Generally, servlets are protocol-independent. However, there exists an abstract HttpServlet class, as part of the Java standard library that has been tailored especially to cope with HTTP-based interaction. The data flow of a web service that is built with a servlet that subclasses the HttpServlet class is very simple. Upon an URL request, the instance of the implementing servlet class is invoked by the servlet container and a request object together with a response object is passed to it. The request object may contain additional information such as HTTP headers, URL query string and cookies sent. The response object can be used to include arbitrary data in the response and to set the HTTP headers and other HTTP related settings.

Advantages Java servlets are very simple to set up and most IDEs offer rich support to create the necessary basic classes, configuration files and descriptors for deployment. Those descriptors are self-contained and instruct the servlet container how to deploy that particular servlet. There is no need to explicitly adopt the server when a new servlet has to be added and the servlet itself can be deployed to any servlet container that implements the Java servlet API. Another advantage that stems from their simplicity is that the control flow is straightforward

[3]http://struts.apache.org/2.x/
[4]http://enunciate.codehaus.org/

and easy to understand. To obtain a fully working HTTP servlet, only one single method has to be implemented.

Disadvantages One of the major disadvantages of Java servlets is that they contain both the view (presentation logic) and the controller (application) logic. There is no clean separation between those two layers which might reduce the readability and maintainability of the application's source code. Since servlets are written in Java, all typed text and messages that are included in the Java source code are bound to several limitations. First, any character that is also part of the Java programming language syntax has to be escaped. Second, it is difficult to visually layout multi-line text within the source code because newlines are not allowed and have to be escaped as well. Lastly, any change of a single character requires the whole class to be re-compiled.

Java servlets have been chosen for this evaluation because they are extremely simple to develop and deploy with Eclipse.

5.3.2 Struts

Struts is an open-source framework for web application development. It has been developed as an answer to the growing need of web application developer to have a clear separation of presentation and application logic. This goal is accomplished by enforcing a model-view-controller (MVC) approach that uses existing technologies for each part of the MVC pattern. Most commonly, JSP are used for presentation layer and JavaBeans are used for the data model. All parts (model, view and controller) are wired together according to a central configuration file. The replacement of a view by another thus requires only a single change within that configuration file.
In contrast to plain Java servlets, the control flow is a bit more complex but still straightforward. In fact, the controller part of Struts is actually a Java servlet which acts as an additional dispatcher for all URL request that are routed to this Struts servlet by the servlet container. This component, called filter dispatcher, first determines if so-called *interceptors* have to be executed first. Interceptors capture functionality that is common to many (possibly all) requests like input validation, user authentication or internationalisation of strings. Interceptors can be stacked and executed in a pre-defined order. After all interceptors have been processed, the controller invokes the *Action* class object that has been associated with the URL of the current request. That *Action* object is responsible for implementing the business logic. It will usually store and retrieve information from a database system. When the Action object has finished its work, it will delegate the control flow to a view that may further process the response and convert the data to a format that is suitable for the client which initiated the request.

Advantages Struts is very mature and widely used for all kinds of web applications. Being a top-level Apache project, it fulfils certain requirements standards of the Apache Software Foundation and can be considered as very stable and robust. It also integrates nicely with other application frameworks like Spring. Besides, Maven integration is easy and Struts is also maintained by the most important Maven repositories.

Disadvantages The framework has been primarily designed to ease the development of dynamic web sites and is built around interaction with web forms. It is less suitable for other application types like web services.

Struts has been chosen for this evaluation because the author of this thesis already had some experience with the framework and because most properties are stored in a single configuration file which is very convenient.

5.3.3 Enunciate

"Enunciate is an engine for creating, maintaining, and deploying your rich Web service API for the Java platform." It is intended to facilitate the development of web services that are going to be published to several different endpoints such as SOAP, REST, JSON etc. Enunciate is not a traditional web service framework but rather a J2EE web service deployment framework that uses several existing frameworks (Jersey for REST, XFire for SOAP etc.) to publish service interfaces defined in Java. A web service developer may start with the implementation of the service classes in Java. Afterwards, the service interfaces are annotated with metadata, using the annotation mechanism of the Java platform. The Enunciate engine extracts those metadata from the source code and generates all the necessary classes, descriptors and other resources. Everything is then packaged in a WAR file that can be deployed to any web application server.

Advantages Clearly, the amount of available endpoints is the most promising feature of Enunciate. Web services can be published via SOAP, REST (XML and JSON), GWT-RPC (Google Web Toolkit), and AMF (Action Message Format) for Flex. In addition to the web service endpoints, Enunciate can also generate a full HTML documentation of the service interfaces, code for consumers of the web service and auxiliary interface definition documents like WSDL and XML-Schema.

Disadvantages Because Enunciate is a framework that generates extra (Java) code from service interface descriptions, each minor change to the service interface requires all Enunciate steps to be performed again, thus slowing down the development process in early stages where changes to the interface can happen fairly often, especially in a research project. Additionally, it requires some effort to integrate Enunciate with the Maven build cycle and the Web Tools Platform (WTP) of Eclipse. Last but not least, Enunciate is a fairly new and unknown project. No large project is known so far that uses Enunciate and it has yet to be proven whether Enunciate is stable enough for mature projects.

Enunciate has been chosen for its ability to publish the same Java interfaces to several endpoints. This was extremely helpful during the evaluation of the RIA frameworks because the same sample web service could be accessed through various channels and with differing data formats without having to install and set up another framework for each distinct technology on the server-side.

5.3.4 Evaluation process and decision

The evaluation of a Java server architecture that supports communication over the Internet started at the beginning of December when the field of possible RIA frameworks has already been thinned out and only Silverlight and OpenLaszlo were left. Since both support plain XML as a means of communication, the choice of an appropriate web service architecture was not limited at all because even if XML is not directly supported, it can always be generated with the help of one of many XML libraries available for the Java platform.

The first technology that has been assessed was the Java Servlet API. Because they are part of the official specification and thus included in the standard library, servlets do not require any additional libraries and can be created very fast. Additionally, Eclipse has built-in support for Java servlets and provides a wizard that generates not only the necessary Java classes but also descriptor files which are needed for deployment. Another advantage of servlets is the simplicity of the control flow due to the fact that they model a simple request-response pattern. However, that simplicity is also one of the main drawbacks of servlets. They are often too primitive to build complex application with. Today, Java servlets are regarded as an enabling technology. This means that applications should not be built on top of Java servlets directly but rather use an additional framework that provides better abstractions of the underlying raw servlets. Most web frameworks enhance the functionality provided by the standard Java Servlet API and offer supplementary, common web functionality such as access restrictions, or transactions. As a result, the developer is released from writing the same boilerplate code over and over again.

Struts is such a framework that alleviates the developer from many recurring, cumbersome tasks. Furthermore, Struts is a well recognised framework with a large user base and has abundant examples and a good documentation available on their web site. In parallel, Enunciate has been evaluated to analyse to impact of a strict REST-style interface on the development of a web service. It was also used to test the SOAP capabilities of the Silverlight platform.

While Enunciate came closer to the definition of a web service framework than Struts, the higher development speed of the latter could not be neglected. After OpenLaszlo has been chosen as the RIA framework for the client application, the ability of Enunciate to publish to various endpoints became insignificant because OpenLaszlo supports only XML web services anyway. And since both the server and the client part of the web application are developed by the author of this thesis, automatically generated interface definition files and online documentation was not necessary either. In addition, the fact that Struts integrates very nicely with Spring was another plus factor. Therefore, the author of this thesis came to the conclusion that Struts will suffice his needs. More details of Struts and Spring can be found in the implementation chapter.

6 Implementation

In the following sections it is shown how the concepts and ideas presented in chapter 4 have been incorporated into a fully working software system. Firstly, implementation details are given for the Avon module called *Information Component Module* (ICModule) that introduces the notion of database components to the semantic database layer of Avon. Secondly, the architecture and implementation of *MyApp Builder*, a web application on top of the Information Component module, is presented. This chapter is concluded by a description of the user interface generation algorithm and other components that are relevant for this process.

6.1 Information Components Module

Based on the OM model depicted in figure 4.3 and explained in details in section 4.2.1, a new Avon module has been created that extends the Avon core database system by the notion of Information Components. Modules that extend the OM layer of Avon have to implement the *OMModule* interface. The class *ICModule* implements this interface for the current module. When the Avon database system is started and after the core module has been loaded, its *OMModuleManager* reads a specific property file from the classpath and loads all listed modules. The content of this property file for the ICModule is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>OM Module definition file.
    Lists the modules that are loaded.</comment>
    <entry key="ICModule">
        ch.ethz.globis.avon.icm.module.ICModule
    </entry>
</properties>
```

That *ICModule* class specified in this property file serves as an entry point to the initialisation of the ICModule. Since modules create new metamodel entities in the database, they have to be able to determine whether they have already been initialised before for that particular database or whether they currently act on a fresh database. In either case, they have to register their CRUD classes in the Avon database manager.
Governed by the guidelines of the Avon module mechanism (see section 3.2.1), the Information Component module consists of three extensions, namely a *Meta Model Extension*, the corresponding *CRUD classes* and the respective *Database Language Extension*. Because the OML database language engine has not been implemented yet for OMS Avon, no proposal was written for possible database language extensions. They are therefore left for future work. In the subsequent sections, the other two extensions are explained separately.

6.1.1 Meta Model Extension

Upon initialisation, the IC module creates, if necessary, the metamodel objects which implement the IC model. There exists two possible ways to instantiate the IC model. The *ICMetaModel* class creates the schema elements programmatically in Java with the aid of the CRUD classes from the Avon core module. A more elegant solution is available with the supplied schema definition file *icmSchema.oml*. It contains all necessary OML statements to create the schema elements with the OML database engine.

6.1.2 CRUD Classes

The metamodel concepts of the IC model are manipulated via corresponding CRUD classes. They implement basic data management operations such as create, read, update and delete (hence the name CRUD) for the various metamodel concepts as well as additional methods that provide more sophisticated operations. These CRUD class are used by the web application to manage Information Components. As part of the CRUD registration process, references to instances of them are stored in a HashMap of the *OMDatabase* instance which manages the access to the Avon database. To provide a more convenient access to those CRUD classes, a new class called *ICOrb* was created which possesses discrete methods to retrieve individual CRUD classes of the IC module. For each collection of the IC model, a separate CRUD class was created. However, there exists no CRUD classes for *Events* and *Operations* at the moment because the supporting technology has not been implemented yet. With regard to *Events*, an event system was being developed by another master student [11] while this thesis was still a work in progress. In terms of *Operations*, it was intended to model them as OML expressions which were not available either. The current set of CRUD classes is depicted on figure 6.1.

Class InformationComponentsCRUD

The class *InformationComponentsCRUD* is the heart of the IC module and controls most aspects of all Information Components. First of all, it provides methods to create, retrieve and delete new Information Components. Secondly, several methods are available to reuse component parts that were already defined in another Information Component. Their counterparts are the decoupling methods that remove parts from an Information Component without influencing other relationships of that part and of course without deleting them in the original

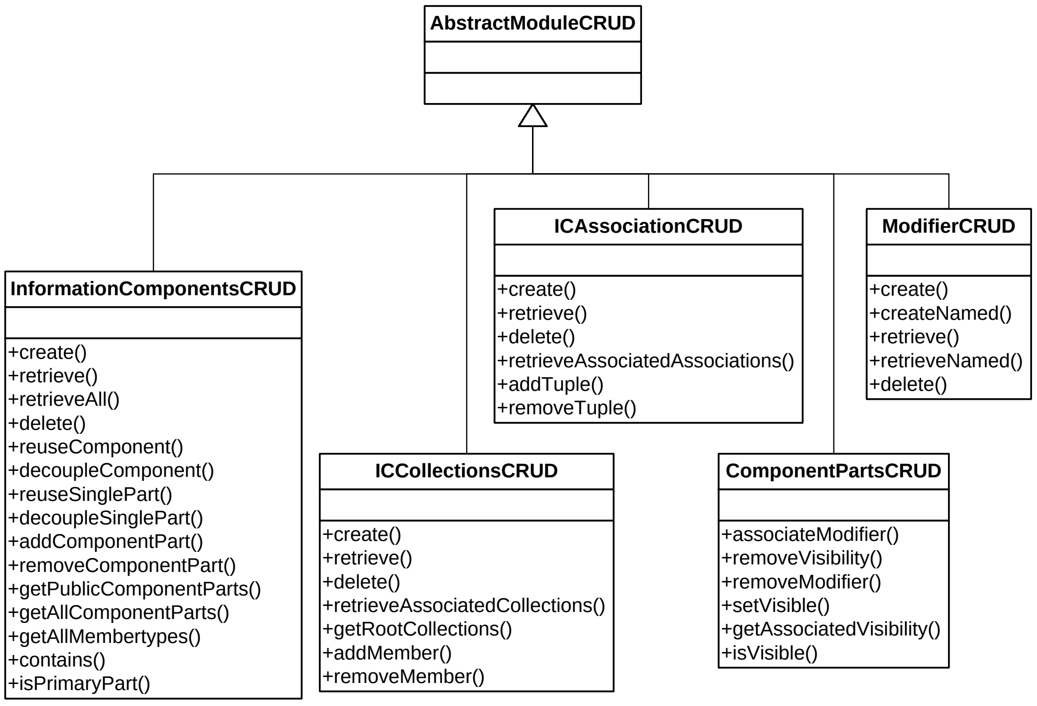


Figure 6.1: CRUD Classes

component. The class also contains methods to add and remove component parts directly. In order to add or remove them from the correct subcollections of the *ComponentParts* collection, the type of the *OMObject* passed as a parameter has to be determined first. This is the reason why in the case of collections and associations it is recommended to add and remove them via the ICCollectionsCRUD and ICAssociationsCRUD classes. Beside those fundamental methods, a few auxiliary methods are provided that reveal details about the content of an Information Component. This includes methods to retrieve all parts of a component or only the public ones. It also offers methods to check whether a part belongs to a given component and whether that part is a primary part (not a reused one) of that component.

Class ComponentPartsCRUD

The *ComponentPartsCRUD* class is responsible for the modifiers associated with parts of Information Components. As was explained in section 4.2.1, those modifiers are not associated with the parts directly but rather with the appropriate `contains` relationship. Therefore, each method of this class also requires a reference to the Information Component whose part, whether reused or not, is going to be manipulated. Apart from the general methods to add and remove arbitrary modifiers, there are some convenience methods to control and query the visibility of component parts directly.

Class ICCollectionsCRUD

The *ICCollectionsCRUD* class provides methods to create, retrieve and delete standard OM collections in the context of an Information Component. This guarantees that the collection will be associated with the given Information Component as a primary component part. Additionally, the methods take care that the collection becomes a member of all the necessary collections. It is the only recommended way of deleting a collection that has been part of an Information Component because the methods perform all the necessary checks first. There is also a method that returns a list of all collections that have no super-collections and are therefore root collections of the specified Information Component. Root collections are assumed to represent the core concepts modelled by a given Information Component and play an important role during the UI generation phase of the web application.

Class ICAssociationsCRUD

Just like the *ICCollectionsCRUD* class, the *ICAssociationsCRUD* class is a helper class too. It offers similar methods for associations and again makes sure everything is consistent when adding or deleting associations that belong to an Information Component. The usage of the standard OM CRUD classes to create or delete associations and collections that are tied to a specific Information Component is strongly discouraged.

Class ModifiersCRUD

The class *ModifiersCRUD* contains methods to manage modifiers than can be attached to `contains`-relationships of Information Components and their parts. Normally, modifier objects do not need a name but merely define a flag such as “public” or “private” which can also be used to retrieve them for reuse.

6.1.3 Application Programming Interface

Apart from the CRUD classes that define the official interface to access the concepts introduced by this module, a number of additional API classes have been added to increase the usability of this module for applications that build on top of it. Most notably, they follow an object-oriented approach and provide a class for each concept of the metamodel as can be seen in figure 6.2. However, they are still closely coupled with the base CRUD classes and all manipulation operations are immediately reflected in the Avon DBMS. In terms of functionality, there is no difference between these API classes and the CRUD classes of the IC module. Which set is more appropriate is only a matter of style and personal preference. In addition, the *ICRepository* class offers methods to easily retrieve instances of IC elements by name.

6.1.4 Reuse Semantic

When parts of an Information Component should be reused, one has to differentiate between associations and all other parts. While collections and auxiliary parts can be reused directly without further processing, things are a little bit more complex with respect to associations. Associations are always defined over collections and since associations are also collections, arbitrary nesting is possible. But associations do not contain any data besides the identity of the objects from the source and target collection involved in a relationship. As a result, associations cannot be considered as standalone entities and are always dependent on at least

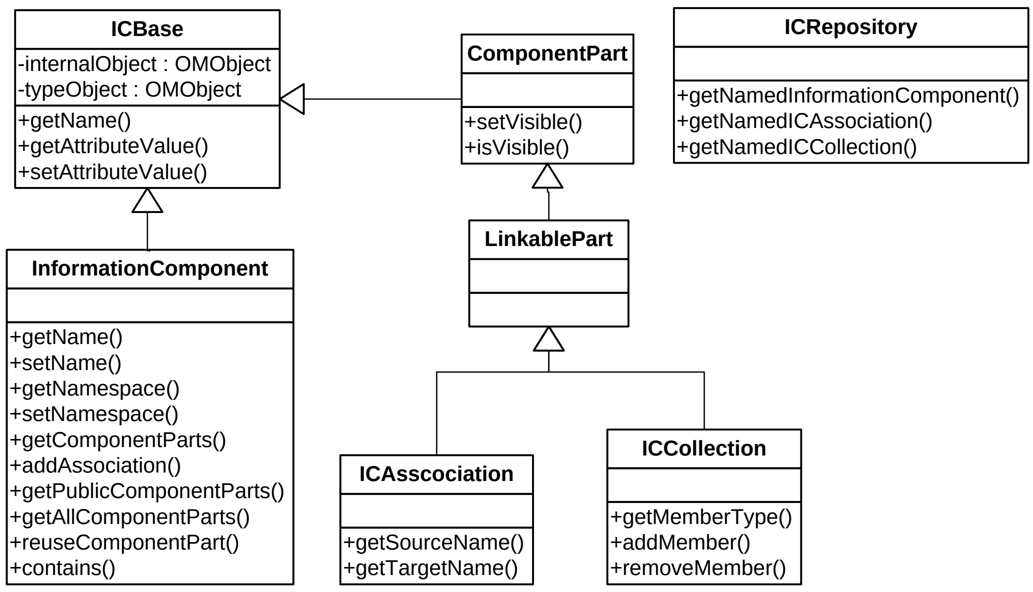


Figure 6.2: API Classes

one collection. This fact has to be reflected in the "add" and "reuse" mechanism. Therefore, their algorithms work as follow:
The component part is analysed first and its type is determined. If the part is a collection or was classified as an auxiliary part, it will be added to the `contains as primary` or `contains` association. If the part was identified as an association, the source and target of this association have to be added/reused first. Hence the algorithm calls itself recursively. As a side effect, this also ensures that nested associations are added/reused correctly.
However, there is no reverse algorithm when component parts are removed or decoupled. The reasons for this are simple. First of all, the source and target of an association, whether it is a collection or a nested association, can exist on their own. Additionally, the system cannot assume the intention of the client that initiated the removal or decoupling. It might well be possible that only the association itself should be removed to break a relation between two collections for whatever reason.

6.1.5 Encapsulation

Two predefined modifiers are created during the initialisation of the IC module, a "private" and a "public" modifier which are also known as *visibility* modifiers. They are used throughout the module to differentiate between public and private parts of Information Components. The purpose of these visibility constraints is to control reutilisation of component parts, though they are not enforced by the system and could be circumvented manually. By default, reused parts are declared as private and have to be set to public before they can be reused again.

6.2 Web Application

In this section, the web application's architecture will be presented from the point of view of the implementation side. Technical details of the libraries and frameworks used to build this web application are given. Furthermore, methodologies and solutions are illustrated that have been developed to tackle some of the design problems presented earlier.
The overall system architecture of the *MyApp Builder* application can be described as a N-tier architecture with 4 different layers, illustrated in figure 6.3. They are grouped into three development projects for better maintainability and to illustrate the fact that they are exchangeable as long as the interfaces between the layers do not change.

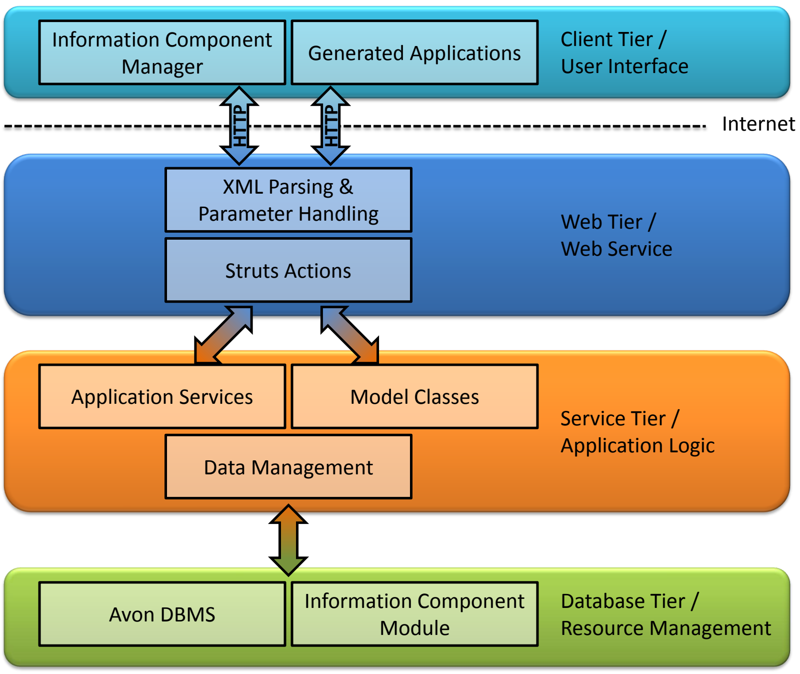

Information Component Manager
Generated Applications
Client Tier / User Interface
HTTP
HTTP
Internet
XML Parsing & Parameter Handling
Struts Actions
Web Tier / Web Service
Application Services
Model Classes
Data Management
Service Tier / Application Logic
Avon DBMS
Information Component Module
Database Tier / Resource Management

Figure 6.3: Web Application Architecture

The Information Components module is managed in *avon-infcom*, a project that belongs to the Avon database repository. The web application on the server-side, which includes the service and web tier, has been set up as a dynamic web site project in Eclipse called *avon-infcom-webservice*. Another project named *AvonPIM-OpenLaszlo* has been created for the OpenLaszlo web client.

6.2.1 Database Tier / Resource Management

MyApp Builder uses Avon as its database to store all its application data as well as the actual personal data. The Avon database is packaged to a *jar* file and embedded into the web application project including all its dependencies. In addition to the Information Components schema described by its OM model, *MyApp Builder* introduces a few more concepts to store metadata associated with the web interface. The extended OM model includes schema definitions to store:

User-defined Types User-defined types are type objects that adhere to a simplified structure and are stored in a separate collection. They include all types that have been created by users via the web client during the design phase of a new Information Component.

View Metadata There are two new types that store metadata that pertains to the representation of Information Components and collections in OpenLaszlo on the client-side. The first type `infComView` is meant to record when a specific Information Component has been updated by the user and when a user interface has been generated from it. Whenever these two dates differ, the user interface is not in sync with the appropriate application domain model defined by that Information Component. The second type `collectionView` is used to store information about the default attribute and the default representation of the member type of this collection. Both properties influence the way members of this collection are displayed to the user. The default attribute specifies which attribute from the member type shall be used whenever the UI has to represent members in a compact form, for example as entries in lists. Similarly, the default representation denominates the representation used to display members in an extended format which in most cases encompasses all attributes of the corresponding member type. With regard to the possible values, the value `Generated` is reserved and indicates that the default representation shall be automatically generated. A `collectionView` property object is not associated with a collection directly but instead associated with the `contains` association of the IC model. It is thereby possible to re-define these properties for every reused collection.

Labels The `label` type allows developers to add two properties to any *OMObject* that are important in the context of a user interface. Those two properties are *prettyname* and *description*. *Prettyname* is meant to represent a user-friendly, comprehensible name of an OMObject. It should primarily be used for OMObjects that describe types, collections and associations because they normally do not possess a pretty name beside their internal name. Similarly, the intention of the *description* property is to add a plain-text description that can be displayed as additional information about an OMObject to the end-user.

6.2.2 Service Tier / Application Logic

The service tier implements the main application logic of the web application. It has been decoupled from the web tier in order to leave the door open for different web service implementations.

Web application bootstrap

The *LocalAvon* class offers two methods that control the lifecycle of the Avon sub-system. The Spring container invokes *initDatabase* when the web application is started and *closeDatabase* when the system is exited. In the initialisation method, a new instance of the *OMDatabaseManager* class is created. With the aid of this *OMDatabaseManager*, new databases can be created and existing ones can be opened. For the development of the web application, the "In-Memory" implementation of the storage layer of Avon was used because it offers the fastest response times and processing performance. However, the "In-Memory" implementation provides no long-term persistence and is not suitable for production. In that case, the db4o

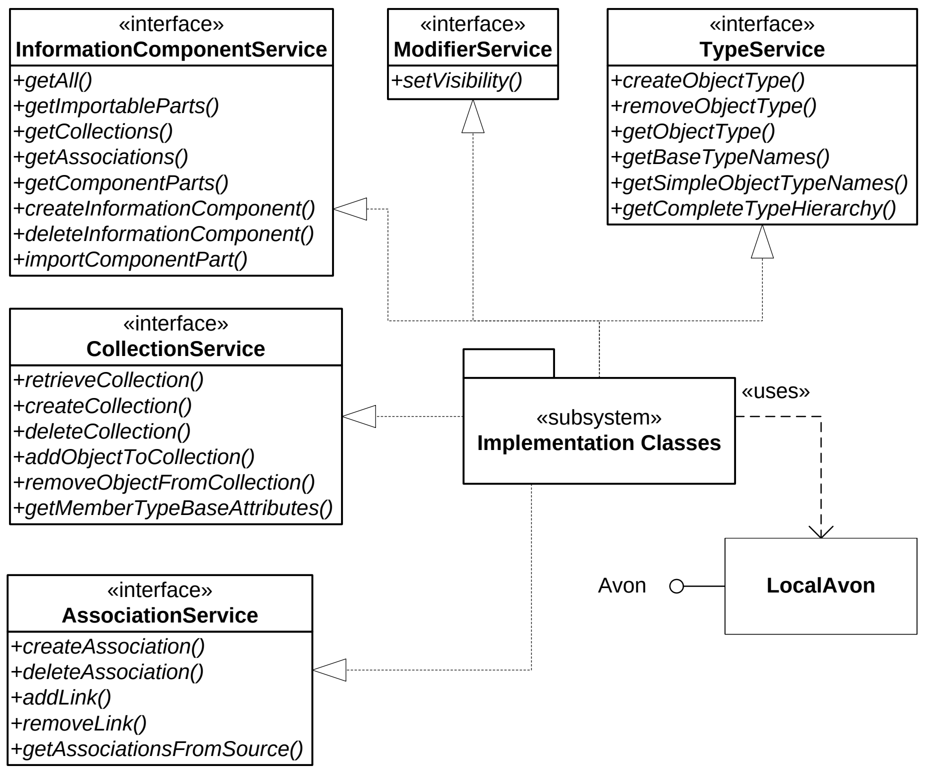


Figure 6.4: Metadata Services

implementation should be used instead which stores all data and metadata to a database file in the file system. As soon as an existing database is opened, the Avon DBMS takes over and initialises the core module as well as any other modules defined in the *ommodules.properties* file. For *MyApp Builder*, the IC module is the only necessary module. The initialisation process is finished by the generation of some sample data that was used to develop the web application. This step is required because the “In-Memory” database is lost after the system has shut down. As a side effect, the system always starts with a fresh database which greatly simplified the development process.

Service Classes

For each IC concept introduced in the Information Components module, a service class has been created on the web application layer. However, only the functionality needed by the web client has been implemented and therefore some OM concepts of the Avon OM layer do not have a corresponding service class. These service classes contain not only the application logic of *MyApp Builder* but also implement the data access layer where appropriate. They access the database via the CRUD classes offered by the Avon OM core module and the Information Components module.
It is obvious, that most service interfaces resemble the corresponding CRUD classes in the database layer. However, none of the internals of Avon and the OM model are visible from the outside and all interaction with the service tier is done over the service interfaces that together comprise the service API. All the service classes are located in the package

ch.ethz.avon.icm.webservice.services. Nevertheless, they can be divided into three categories. *Metadata Services* (figure 6.4) provide methods to deal with Information Components. *Data Services* (figure 6.5) are used to create and manipulate objects that contain user-generated data. The *Presentation Services* (figure 6.6) offer methods to manipulate the metaconcepts introduced particularly for this web application, see section 6.2.1.

InformationComponentsService The *InformationComponentsService* class provides methods to create and delete Information Components and to handle the import and exemption of component parts. In this respect, *getImportableParts()* returns a list of all Information Components and their component parts which are importable by a specific Information Component. The names of Information Components have to be unique because all methods reference them by name. Additional methods allow to retrieve all component parts, only collections or only associations that belong to the given Information Component. For each of these methods, it can be specified whether all parts or only the public ones should be returned. It is also possible to get a list of all Information Components currently present in the system. In contrast to the CRUD classes, no methods to add and remove arbitrary OMObjects to and from an Information Component are offered at the service level.

CollectionService The class *CollectionService* mirrors the functionality of the ICCollectionsCRUD class but includes methods to add and remove objects to and from a collection. Another method returns a list of all base attributes of the member type of the specified collection. This list will be used by the web client to create new members of that collection.

AssociationService The *AssociationService* class is the equivalent to the ICAssociations-CRUD class. In addition to the basic functionality, it offers methods to add and remove links between two objects in the context of a particular association. It has also a method to retrieve all associations from a specific Information Component that have the specified linkable part as their domain collection.

ModifierService The only purpose of the *ModiferService* class is to toggle the visibility of component parts in the context of a specific Information Component.

TypeService The class *TypeService* is responsible for all functionality necessary for the web application that is associated with the type layer of OMS Avon. Most importantly, it allows new object types to be created. The remaining methods have an informational character. They return the names of all base types or all simple types and they can create a graph of the complete type hierarchy. The *TypeService* also allows to enquire about a specific object type and returns attribute names and subtypes of that object type.

ObjectService The *ObjectService* class deals with the actual data objects that contain personal information. With the aid of this class, data objects can be created, retrieved, updated and deleted. For a given object, its attribute values can be fetched from the database. Together with a reference (by name) to an association, the *getLinks()* method compiles a list of all objects that are associated with the supplied object. In addition, there is a method to determine all collections an object is a member of.

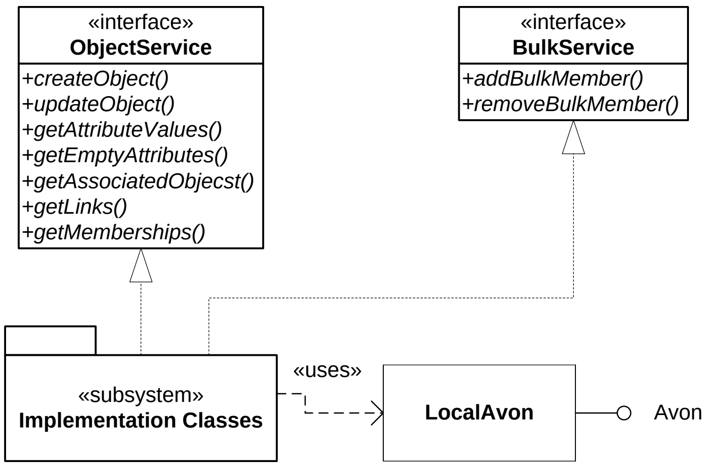


Figure 6.5: Data Services

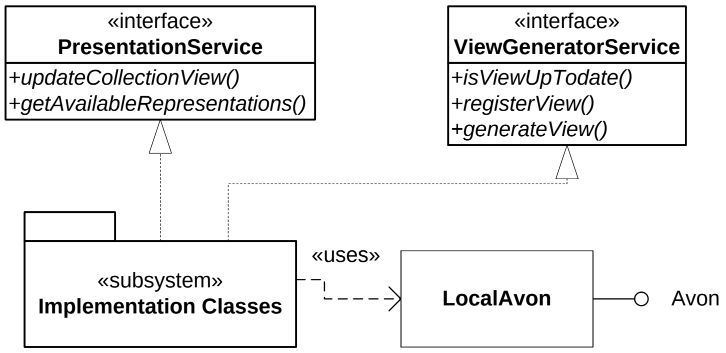


Figure 6.6: Presentation Services

BulkService The class *BulkService* is used to manage attributes of objects that are lists of base values. This is a simplification of the general notion of bulks in the OM model where bulks are not limited to base types only.

PresentationService The *PresentationService* class offers one method to enquire about the available representations for a specific, user-generated type. To other method is used to update the `collectionView` object that is associated with a collection an stores the default attribute and default representation of its member type.

ViewGeneratorService The class *ViewGeneratorService* is closely coupled with the View Generator architecture described in great details in section 6.3.3. Apart from the possibility to generate user interfaces for whole Information Components, the service also offers methods to query and manipulate properties of the user interface to be generated.

Model Classes

The model classes are used by the web application for two purposes. On the one hand, they are used internally to represent concepts from the database at runtime within the Java virtual machine. On to other hand, they are used to transfer data to the web client over a web service interface. All model classes are simple Java classes and follow the JavaBeans specification. They are solely meant to represent data and do not provide any additional functionalities. According to their belonging, they are split up into three Java packages. The *ch.ethz.globis.avon.icm.webservice.model* package contains model classes for collections, associations and Information Components. It also includes a class for data objects. All instantiations of user-defined types are data objects and they will contain the pieces of personal information the users enter into the system via the web client. Each data object has a list of attribute values that correspond to the attributes of all its types. Possible attribute values are typified by the different model classes in *ch.ethz.globis.avon.icm.webservice.model.attributes*. Last but not least, the package *ch.ethz.globis.avon.icm.webservice.model.meta* contains model classes to represent metadata such as attributes and object types.

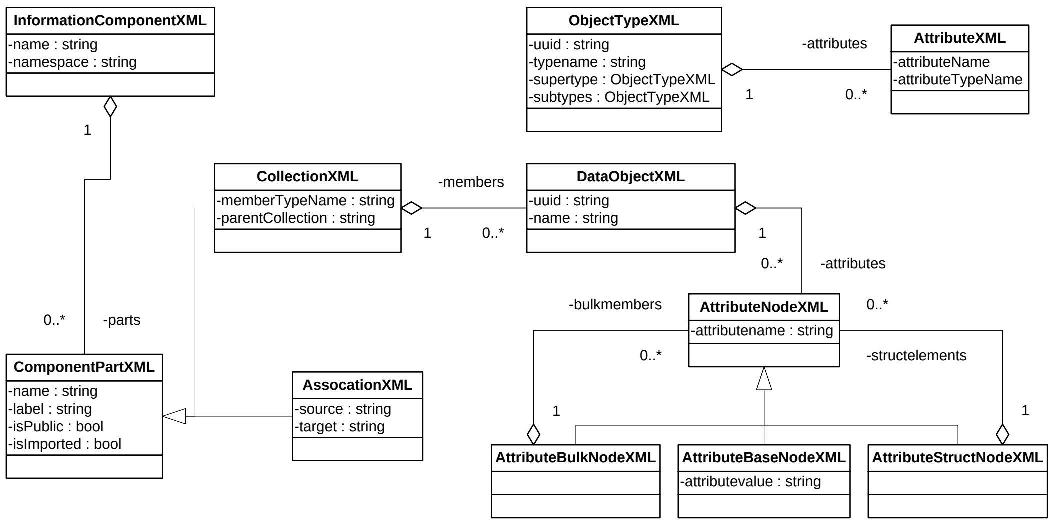


Figure 6.7: Model Classes

6.2.3 Web Tier / Web Service

The web service tier mediates data between the web application on the server-side and the web client. It also processes URL requests and serves response back to the clients. It was implemented using the Struts web application framework which has already been explained from a design perspective in chapter 4. All web service related classes can be found in the package *ch.ethz.globis.avon.icm.webservice.struts*. These classes are called *Action* classes and implement the actual web service interface. When it comes down to the implementation level, all Struts Action classes inherit from the *ActionSupport* class. Each class has to override the *execute()* method. This method is being called by the Struts framework if the corresponding URL has been requested by a client. The mapping of URL to Struts Action classes is managed in the configuration file *struts.xml*. To simplify development, several operations that belong to the same subject (collections, associations, objects etc.) have been multiplexed together in the same *Action* class. At the end of each request, all Action classes write their XML response, if there is any, to a Java field which is then embedded into a JavaServer Page (JSP) and processed by the web server before the final XML is returned to the client. Most of the XML responses are generated with XStream directly from instances of the model classes. XStream is a library that transforms Java classes to XML and vice versa. It is explained in great detail in section 6.2.5.

XMLWorkhorse Instances of *XStream* objects are configured programmatically. To avoid the same objects being created over and over again, the utility class *XMLWorkhorse* bundles the most important *XStream* objects that are used by the web tier to convert instances of the model classes to XML streams. A great characteristic of these converter objects is that they also work in the other direction. This means that an XML stream can also be converted to corresponding Java objects, provided the *XStream* object has be configured with the same properties used to convert the Java objects to XML in the first place.

Object identity Due to the fact that all parts of an Information Component have a name that needs to be unique, the very same name could also be used to identify parts of an Information Component when they are accessed over the web service interface. With respect to instances of the application domain model created by the user, individual objects are identified by a *Universally Unique Identifier* (UUID). The UUID is an internal identifier created by the Avon database system that identifies each distinct object uniquely. It is the only unique property an object has in the Avon DBMS system and it is therefore reasonable to use the same identifier to identify objects via the web service interface. Of course, such an identifier cannot be derived from any other property the object might have and thus it is only possible for the web client to acquire such identifiers by retrieving a collection of objects first.

6.2.4 Client Tier / User Interface

The OpenLaszlo architecture provides for two ways of developing and deploying OpenLaszlo applications. Either applications are developed as standalone, pre-compiled programs (called *SOLO* in OpenLaszlo) that can be deployed to any web server and operate independently or applications function in a *proxied* context. The latter requires the OpenLaszlo Server servlet to be deployed to the target web application server but allows additional functionalities to be used such as on-the-fly compilation. *Proxied* mode was necessary to develop *MyApp Builder*, because the automatically generated OpenLaszlo applications are created dynamically at runtime of the system. A web client has been written to create and manipulate Information Components in the database via a convenient web interface accessible by any browser that has the Flash runtime installed. That web client has been named *Information Component Manager* (ICM) internally for obvious reasons. It is part of the same project that also houses the OpenLaszlo Server servlets and is thus deployed together with the whole OpenLaszlo infrastructure.

6.2.5 Libraries used in the web application

JDOM

JDOM is an open-source Java library to programmatically manipulate and create XML documents which has been developed for over 4 years. It is a very light-weight and easy-to-use library that builds upon the *Simple API for XML* standard and incorporates a *Document Object Model* to represent XML structures in Java. For all kinds of buildings blocks in XML

(elements, attributes etc.), there exists an appropriate equivalent in JDOM and the hierarchical structure of an XML document is matched one-to-one with an internal tree structure, the DOM. JDOM documents can be built from XML files and XML files can be created from JDOM documents. However, JDOM is not meant to provide a mapping of Java classes to XML structures but rather allows developers to easily manipulate any XML structures by altering their structure directly via a straightforward and simple API. Due to that fact, it is especially well-suited for rapid development of XML applications.
JDOM was used in this project to implement the *ViewGenerator* described in section 6.3.3. The *ViewGenerator* class is responsible for creating LZX files based on user-defined Information Components.

Spring

Spring is an open-source and widely used application framework that facilities many common tasks found in J2EE application development. It consists of several modules that provide infrastructure services such as object-relational mapping, transaction management or even aspect-oriented programming. The most important concept of the Spring framework is called *Inversion of Control*. The idea behind this principle is that applications move the responsibility for startup, initialisation and configuration of objects to the framework. That part of the Spring framework is called IoC container and this container is responsible for the lifecycle of all objects. Spring makes the configuration of an application very easy by handling configuration in a simple XML file (*applicationContext.xml* in this project). All objects to be created by the container, also called *Managed Objects*, are defined in this configuration file. In this project, Spring is used to manage the services and Struts Action objects from the service and web tier. So for every such class, there exists an entry in the XML file that specifies how the class shall be instantiated. Additionally, it is possible to define whether the object is a Singleton or whether multiple instances of the same class can coexist. This eliminates the need for an extra mechanism to control Singletons. Furthermore, Spring tries to be as non-invasive as possible. Apart from some annotations to indicate which properties should be auto-wired by the Spring container, no further modification of the system was necessary. Upon startup of the application, the container then creates those objects, calls initialisation methods and wires objects together by establishing references between them. The latter is called *Dependency Injection* and was heavily used in this project. With dependency injection, dependencies between objects are managed by the container. Dependency injection happens either via the constructor or via JavaBeans setter methods based upon the names given to objects in the configuration file or upon their types at runtime. For example, each service class needs a reference to an object that implements the *Avon* interface in order to access the Avon database. Instead of writing initialisation methods for each and every service class to set up this reference, the Spring container takes care of everything and ensures that each object will have the references it needs at runtime. Because all objects are configured in a dedicated configuration file, changing for example the implementing class of the *Avon* interface only requires a change of the corresponding XML entry. Additionally, the IoC Container injects a dependency to the appropriate service into each Struts Action class of the web tier.

XStream

While JDOM provides a way to work with XML documents in a fashion familiar to Java programmers, it has not been designed to bind XML structures to concrete, application-specific Java classes. This is where XStream comes into the picture. XStream is a simple Java library to serialize objects to XML and vice versa. It provides similar functionalities as the *Java Architecture for XML Binding* (JAXB) although the latter has been optimised to work with XML schemata while the former is schema-less and has a much lower footprint. Therefore, no explicit mapping is required between Java classes and XML elements. Instead of an XML schema, XStream uses reflection to discover the internal structure of Java classes. As with JAXB, it can be used to marshal object graphs to XML files and to unmarshal XML files back to object graphs. Additionally, there is no need to modify any classes or to add annotations in order to make them compatible with XStream. However, conversion strategies are customizable to account for the developer's desire to modify the way types are represented in XML or to rename elements during the conversion process.
Especially its ease of use and the generation of clean, human-readable XML was decisive factor for the inclusion of XStream in *MyApp Builder*. By default, XStream uses fully qualified names to denominate XML elements that correspond to Java classes. Fortunately, it is very simple to configure aliases for any class or field. In order to centralize these configuration steps, the *XMLWorkhorse* class was created.

6.3 User Interface Generation

One of the distinct features of the *MyApp Builder* system is the generation of user interfaces based on Information Components created by the users. The corresponding algorithm implements the user interface generation phase described in section 4.3. On the one hand, it generates OpenLaszlo classes programmatically on the server-side. On the other hand, these OpenLaszlo classes are written in the declarative XML language of OpenLaszlo (LZX). It is thereby possible to leverage the use of LZX to enhance the user interfaces. For example, it is feasible to exploit the binding mechanism (described in section 5.2.2) for lists and combo boxes. Conventional dynamic web development frameworks generate the entries of such lists programmatically and their content is static after the web site has been served. Every generated UI is comprised of several views. The definition of a view follows the definition of views in OpenLaszlo.

Definition 2 (*View*)

Views are the most fundamental visible elements in OpenLaszlo. Therefore, any (visual) representation of a collection, an association or type is implemented as a view in OpenLaszlo. They function as rectangular containers which are strictly hierarchical and have only one parent, but possibly many children. Additionally, views can be associated with resources such as images, audio or video files to embed them into OpenLaszlo applications. By default, they are transparent.

Whenever we talk of views in the following sections, we always mean entities or classes that are displayable in OpenLaszlo. However, views are not restricted to simple UI components but can be arbitrarily complex. They can also contain methods, event handlers and attributes to embed additional application logic. Views can be defined declaratively in LZX and every generated UI consists of several such view elements.

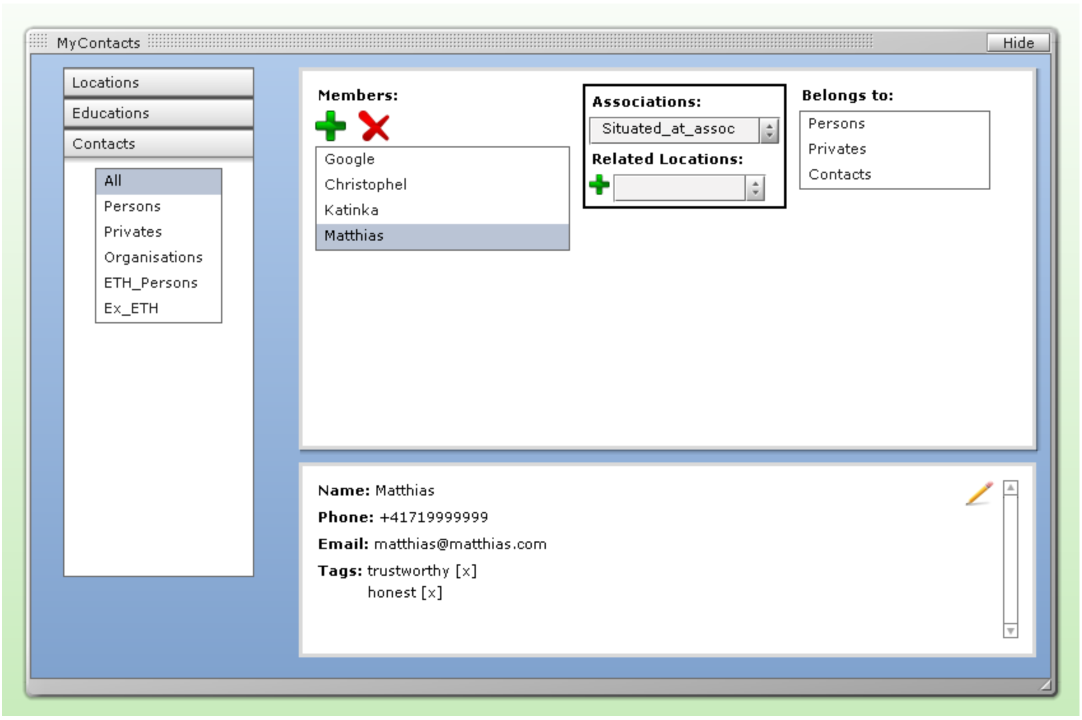
6.3.1 Design of the User Interface

The visual appearance of all generated user interfaces is depicted in figure 6.8. Each application has its own window which is separated into two sections, collection navigation and a master/details view. The representation of the categorization scheme can be found on the left where a tab slider houses a tab element for each root collection.

Definition 3 (*Root collection*)

> *A root collection is a collection of an Information Component that has no supercollection within the same Information Component. Therefore, a root collection represents a central category of that Information Component and should be treated accordingly.*

Every tab element contains a list of all subcollections of the corresponding root collection. Additionally, a "All" entry at the top is used to represent the root collection itself. The top view on the right-hand side will change according to the currently selected collection and features a member list and an area to manage associations defined over that particular collection. As soon as a member has been selected, the details view will display the properties of that member. It is also possible to edit these properties by clicking on the edit icon in the upper-right corner. A detailed explanation of the usage of this web application can be found in section 7.2 where everything is explained from the perspective of the end-user.



Name: Matthias
Phone: +41719999999
Email: matthias@matthias.com
Tags: trustworthy [x]
honest [x]

Figure 6.8: Generated User Interface

The generation of those user interfaces is accomplished with three classes. *ViewGeneratorServiceImpl*, *ViewGenerator* and *TemplateManager*. ViewGeneratorServiceImpl builds on top of ViewGenerator which in turn requires a reference to the TemplateManager. This is why they are explained in reverse order.

6.3.2 Template Manager

In order to facilitate creation of OpenLaszlo classes in LZX, a template manager has been implemented. Upon instantiation, it reads a pre-defined set of LZX files from the *templates* folder which is part of the classpath. Each file is then converted to a JDOM Document whose root element is stored in a field of the template manager. Whenever the view generator retrieves a template via one of the template manager's methods, a fresh copy of the corresponding template is returned. The template manager is meant to be a singleton and thus instantiated only once.
The automatically generated UI is comprised of three different elements and for each of them, a corresponding template exists. The main application class named after the Information Component is just a container for the remaining views and divides the space vertically into two panes. The categorization scheme (collections) will be shown in the left pane whereas information about particular collections and their members are shown in the right pane. There are also two templates used to build views for types.

6.3.3 View Generator

The view generator is one of the most important parts in the *MyApp Builder* system. It features several methods to generate LZX files. It uses JDOM to create and manipulate these files. There are two different view generation algorithms. One is used to generate a UI for the Information Component and its categorization scheme, the other is used to generate representations for user types.

Generation of Views for PIM Applications

The main algorithm to generate a PIM application and its user interface can be described as follows: It starts by retrieving the OpenLaszlo template for an Information Component and generates include statements for all member types. Include statements are similar to imports in Java with the distinction that the content of the included file will actually be part of the including file. In OpenLaszlo, each distinct application will eventually fit into one file and be compiled as a single unit. It is important to note, that the included files do not exist at this stage because they are generated later. If for any reason, the generation of a single file fails, the whole UI becomes invalid and the generation process has failed. There exists a mechanism in OpenLaszlo to defer the loading of included libraries but this technique has not been used for reasons of simplicity. This step also generates include statements for all custom views that have been registered for types of this Information Component. Details of the idea behind customized views for types can be found in section 6.3.5. In a second step, it gets all root collections for the current Information Component and generates a representation for them. For each root collection, the algorithm constructs a tab element for the tab slider, a master/details view and a state element. The master/details view is then wrapped by the state element. State elements are used in OpenLaszlo to retain subelements until a certain condition applies. They are the declarative counterpart to state-based programming in script languages. In the generated application, states are responsible to switch the master/details view when a different tab element has been selected by the user. Although there could have been a single master/details view shared by all root collections (because there are no significant differences between the individual master/details views), having separate views is more convenient for the user because they keep their state when the user switches back and forth between root collections. For each tab element, the algorithm creates a list of all sub collections of the current root collection. Because the categorization scheme does not change at runtime of the application, the list can be created statically. At first, a special entry is created for the root collection. This entry is named "All" to stress the fact that the root collection obviously contains all elements of its subcollections. Secondly, entries for the subcollections are created that are named after them. Every list entry contains an array as its entry value. That array consists of the name of the collection and the name of the view which shall be used to display members of the collection. Before this array can be built, the algorithm has to check whether a custom view has been stored for the member type of this particular collection. It does so by retrieving the *CollectionViewProperty* object that is associated with the currently processed IC and collection. If there is no such object or if the default representation has been set to `Generated`, the array will contain the plain name of the member type because this is also the name of the automatically generated view for that user type. In all other cases, the class name of the customized view is injected and this ensures that whenever this particular collection is browsed by the user, the associated custom view will be used to display the members of this collection to the user. The algorithm finishes by creating an OpenLaszlo window instance that holds the generated user interface. The resulting application can then be started from the Dashboard (see section 7.2), a single web site that provides access to all generated applications.

Generation of Views for Types

The second important algorithm creates representations for user types and can be described as follows: Firstly, the algorithm retrieves a new template for user-generated object types and creates header information. Subsequently, it starts to iterate through all attributes of the object type. For each attribute, an embedded view will be generated. The form of the embedded view depends on the type of the attribute. Currently, base types and bulk types are supported while object types and structured types are not. For base type attributes, two views are created effectively. One is static text only and meant to display the value of the base type attribute to the user. The other is a modifiable text field that can be used to change the value of this attribute. Again, OpenLaszlo's state elements are used to wrap both views. These states are bound to a value of the type view that indicates if the view is currently in edit mode or read-only mode respectively. Whether the type view is in edit mode or not can be controlled by an "Edit" button which is also generated by the algorithm. The matter is only slightly more complex with view for bulk values. Instead of two views, only one view is generated from a template. However, this view incorporates UI components to manipulate the member of the bulk directly. It is not dependent on the global "editable" state of the type view and members can be added and removed at any time.

6.3.4 View Generator Service

The view generator service orchestrates the whole user interface generation process. It has to be explicitly started by the client because only the users know when the design phase has finished as they decide what to include in an Information Component and which settings have to be changed before the design phase is over. Additionally, a path to the local OpenLaszlo server is necessary to indicate where the generated files have to be put. Of course, this is a server-side property unknown to the client. That is the reason why it has to be specified in the Spring configuration file of the web application. When invoked, the view generator performs the UI generation process by applying the subsequent steps:

1. Based on the path to the OpenLaszlo server installation, a file path is derived for the main class that will be generated for the user interface of the current Information Component. Additionally, a subfolder is created that stores all OpenLaszlo view classes for the types. To write XML data to those files, a *FileWriter* and a *XMLOutputter* instance are prepared. The latter requires a *Format* object that specifies how the XML derived from a JDOM representation will be formatted.

2. The main view class is then generated by invoking the appropriate method of a *View-Generator* instance. This method returns an XML document as a JDOM object tree. With the aid of the FileWriter and XMLOutputter objects, that document is written to the file system.

3. In the last step, a list of all member types of the collections that are contained in the current Information Component is compiled. The algorithm iterates through this list and commands the *ViewGenerator* instance to generate basic OpenLaszlo view representations for each type used. Again, the returned XML document is being manifested to the file system. What is more, for each distinct member type, the view generator service retrieves a list of all custom views that have been registered for that type. Subsequently, the appropriate OpenLaszlo file of every custom view found that way is copied to the location where the user interface will eventually reside. All type-related LZX files are put into a single subfolder named after the Information Component in question.

At this stage, the UI generation process can be considered as successful and the new OpenLaszlo application file is ready to be accessed by the users' web browsers via the *Dashboard*. The Dashboard is an OpenLaszlo application that provides access to all generated applications from a single page. It is only then that the LZX files are actually compiled for the first time.

6.3.5 Customized Views

Being talked about much in the section before, it is overdue to introduce the concept of customized views. While the automatically generated views for types may provide basic functionalities to edit and browse instances of that type, they are naturally not very appealing in terms of appearance and can almost never compete with manually created, customized views. In order to overcome these limitations, a facility has been implemented that allows developers familiar with OpenLaszlo to create their own views for user-defined types. These custom views are stored in a folder named *ViewRepository* which is part of the web service application. Within that folder, a single configuration file (*viewrepository.xml*) is responsible for the housekeeping. When a new custom view shall be added, two actions have to be undertaken. First, the LZX file with the custom view class has to be put into the ViewRepository folder. Secondly, a new section has to be added to the viewrepository.xml file. Because this file is read only once when the system starts, a restart of the server is required to incorporate the changes. The necessary XML section has the following simple structure:

```
<view name="Person With Photo">
    <filename>extendedperson.lzx</filename>
    <classname>ExtendedPerson</classname>
    <compatibletypes>
        <type name="person"/>
    </compatibletypes>
</view>
```

While *filename* and *classname* are self-explanatory, the *compatibletypes* element deserves a little more attention. In most cases, custom views have been created with a particular target type in mind. They are usually not compatible with other types, even if they belong to the same type hierarchy. However, due to OpenLaszlo's ability to hide certain UI elements in response to absent data, it might be possible that the view is also suitable for other types that possess only a subset of the attributes the original type had. In order to prevent incompatible matches, developers have to explicitly define the types their custom view is compatible with. The name of the type has to match the name given by the user in the *MyApp Builder* application.
A custom view has to follow several guidelines which guarantee that it can be successfully embedded into the generated user interfaces. Above all, they have to subclass *BasicDetailsView* which takes care of all the necessary communication with the server. The most straightforward approach is to take the generated view for the target type as a basis, implement the changes and save the resulting LZX file to the *ViewRepository* folder of the web application installation. Afterwards, the *viewrepository.xml* should be changed to include this new custom view.

7
Scenario

This chapter explains the usage of *MyApp Builder* by means of a straightforward scenario. In that scenario, the user starts with the design of simple *MyContacts* and *MyPictures* management applications. These two applications are then combined by the user to allow people from the *Contacts* application to be tagged on pictures from the *Pictures* application. The structure of this chapter follows the general workflow described in section 4.3.

7.1 MyApp Builder

During the design phase, the user works with the *MyApp Builder* application which is depicted in figure 7.1. On the left-hand side, the tool displays a list of available applications together with icons to create, edit and delete applications.

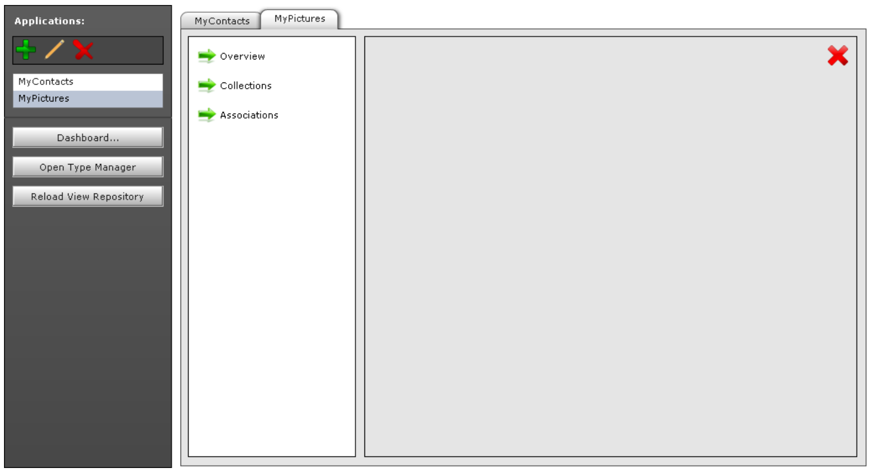
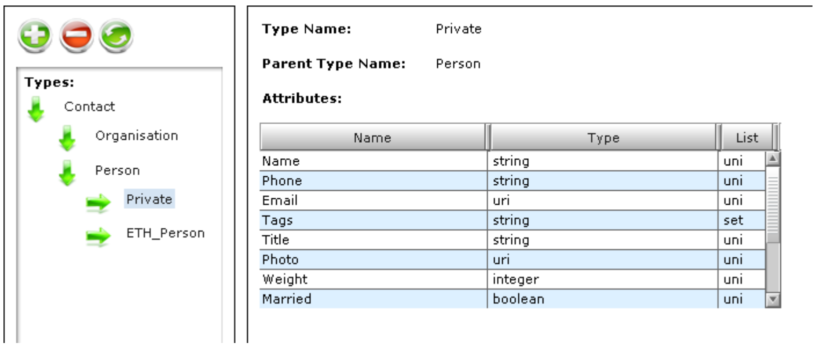


Figure 7.1: MyApp Builder

Now, let us build the *MyContacts* and the *MyPictures* application. We open the *Type Manager* by clicking on the corresponding button. The Type Manager is depicted in figure 7.2 and serves two purposes. Firstly, we can use it to browse the existing type hierarchy in order to explore the types which already exist in the system. Secondly, we can create new types and define what attributes they have.



Types:
- Contact
  - Organisation
  - Person
    - Private
    - ETH_Person

Type Name: Private

Parent Type Name: Person

Attributes:

Name	Type	List
Name	string	uni
Phone	string	uni
Email	uri	uni
Tags	string	set
Title	string	uni
Photo	uri	uni
Weight	integer	uni
Married	boolean	uni

Figure 7.2: Type Manager

We are going to create four types: three types to represent different kinds of contacts and one type for pictures. The types and their attributes can be seen in figure 7.3. Note that `Friend` and `Business` inherit from the general `Contact` type.

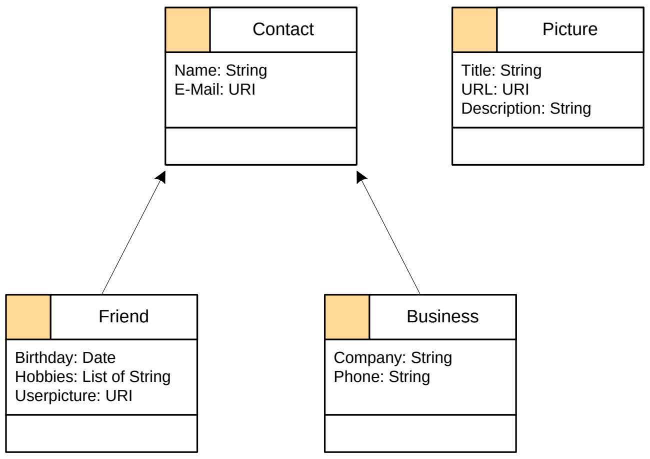
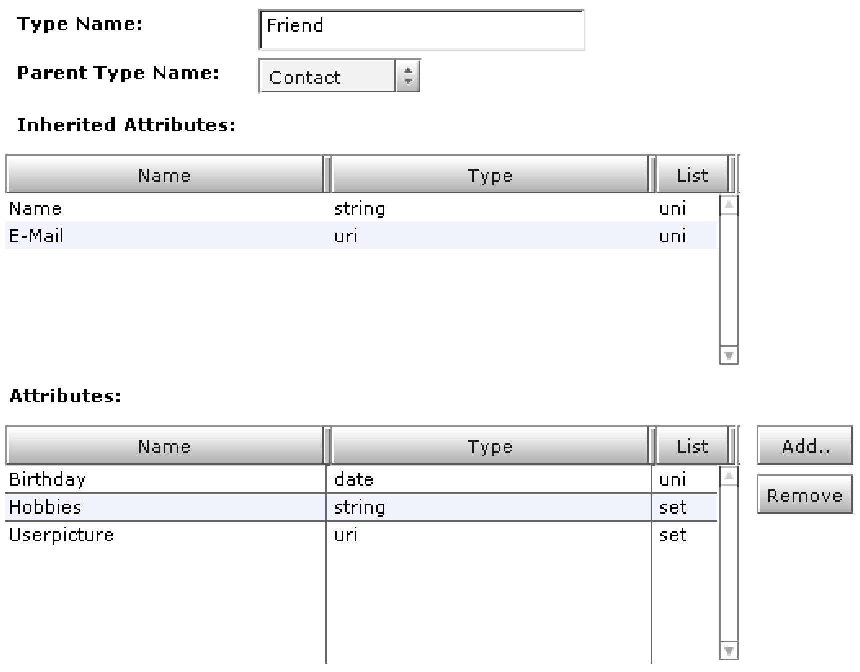


Figure 7.3: Type Hierarchy

Figure 7.4 shows the last step where we create the `Friend` type. Whenever we add an attribute, we can decide which base type the attribute has and whether it represents a single value or a list of values. The type manager can be closed with the appropriate button.
After all the types have been created, it is time to set up the *MyContacts* and *MyPictures* application. With the “Add” symbol on the navigation pane, we can create a new application and give it a name. We can then double-click on entries of the application list in order to open a new tab on the right-hand side that allows us to edit that particular application. Each tab features three menu items:



Type Name:	Friend
Parent Type Name:	Contact

Inherited Attributes:

Name	Type	List
Name	string	uni
E-Mail	uri	uni

Attributes:

Name	Type	List
Birthday	date	uni
Hobbies	string	set
Userpicture	uri	set

Figure 7.4: Create Friend Type

Overview The overview section informs the user about the status of the UI generation. If the application has been changed since the UI was generated the last time, an exclamation mark indicates that the application is currently not available during the operational phase. In that case, the user is advised to re-initialise the UI generation phase after the current design process has been finished.

Collections In this view, users can add, import and delete collections that comprise the categorization scheme of the application. For each collection to be created, the users enter a name, choose a member type and select a parent collection if appropriate. After the creation was successful, the users can further customize certain properties of a collection. They can choose a default attribute of the member type which will be displayed when all members of this collection are presented in a list. Additionally, one can decide whether a collection is public or not. Public collections can be reused by other applications. Last but not least, if there are any customized views available for the current member type, the user can select one of them or stick with the default setting "Generated".

Associations Here, users can create associations between the collections they have created so far. Associations indicate a relationship between the members of one collection and the members of another collection. For example, a `Works for` association can be created between a collection `People` and `Companies`. We will use associations to build our PictureTagging example in section 7.3.

In the course of our scenario, we create a collection for each type. Afterwards, we are ready to let the system generate the applications. We do so by clicking on the "Create User Interface" button in the *Overview* section.

7.2 The Dashboard

All applications are accessible via the *Dashboard* application. However, only applications whose UIs match the current application domain model can actually be loaded. This means that we first have to generate the UIs for the MyPictures and MyContacts application before we can start working with these applications. Afterwards, we can open the Dashboard. In figure 7.5 we see the *Dashboard* for the MyContacts and MyPictures application. A double-click on the application icon opens the application in a new window. From here on, we can start managing our personal data. Each tab on the left-hand side represents one core concept of our application. Core concepts are derived from the categorization scheme. Each collection that has no parent collection in the same application is regarded as such a core concept. All sub-collections are then specialisations of this core concept. These specialisations are available as soon as the user selects a tab. In the case of our MyContacts application, there is only the core concept *Contacts* and two specialisations, *Friends* and *Business*, available. If the user has selected a specialisation or decides to browse "All" members of the corresponding core concept, a master/details view will be shown on the right-hand side of the application. In the master view for *Contacts*, existing contacts can be browsed and new contacts can be added to the current collection. They can also be added to other collections by a simple drag & drop operation. For example, we can drag contact items from the member list and drop them over the target collection entry on the left-hand side. Furthermore, depending on the chosen collection, a number of associations are available that can also be edited directly. The details view displays all the attributes of the current contact. Single attribute values can be edited in-place by clicking on the edit icon in the upper-right corner. For attributes that represent lists of values, individual items can be added and removed at any time. We continue by adding a few contacts and pictures. The URL of pictures is given as a path to their online location.

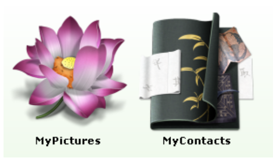
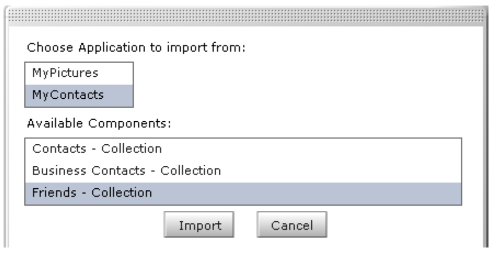


Figure 7.5: Dashboard

7.3 Composition of the PictureTagging Application

In this section, we demonstrate how the MyContacts and MyPictures applications can be reused and extended in order create a new application that offers the functionality of tagging people on photos. We start by creating a new application in *MyApp Builder* and call it *PictureTagging*. We then open this application and switch to the *Collections* view. Instead of creating new collections, we import the *Pictures* collection from the *MyPictures* application (figure 7.6) and the *Friends* collection from the *MyContacts* application. We deliberately chose the *Friends* collection since we do not want to tag business contacts on our private pictures anyway. In a next step, we open the *Associations* view and create the assocation `on this Picture` from the source *Pictures* to the target *Friends* (figure 7.7). That's it. We can now generate our new application and access it from the dashboard.
Figure 7.8 shows the result. The PictureTagging application has two core concepts and friends can be associated with pictures. We can still add new pictures or manipulate existing friends. Since we share the data between the applications, it does not matter where we edit details or



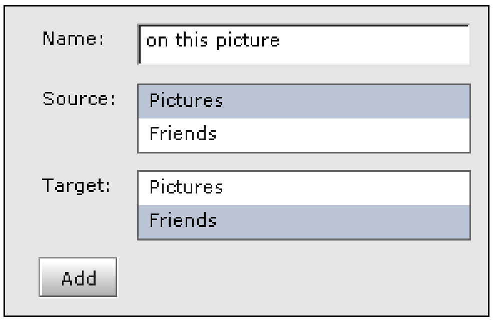
Choose Application to import from:

- MyPictures
- MyContacts

Available Components:

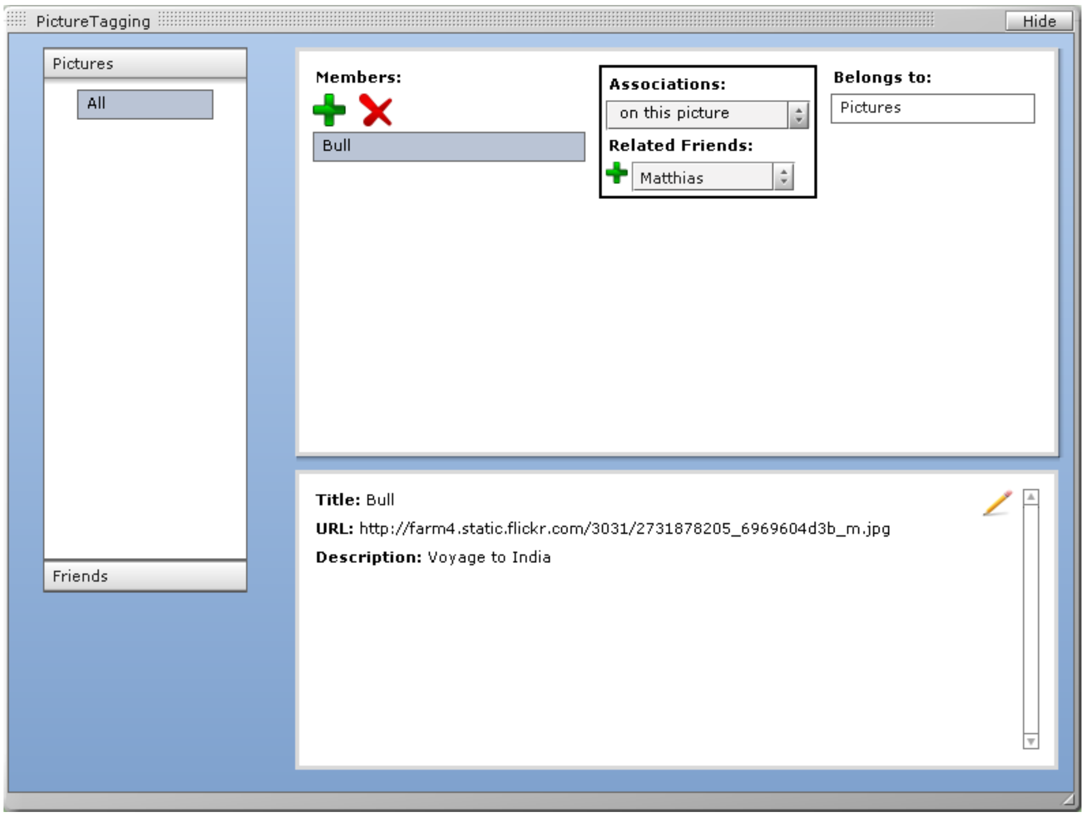
- Contacts - Collection
- Business Contacts - Collection
- Friends - Collection

Figure 7.6: Import Component Part



Name:	on this picture
Source:	Pictures Friends
Target:	Pictures Friends

Figure 7.7: Create Association



Members: Bull

Associations: on this picture

Related Friends: Matthias

Belongs to: Pictures

Title: Bull

URL: http://farm4.static.flickr.com/3031/2731878205_6969604d3b_m.jpg

Description: Voyage to India

Figure 7.8: Final PictureTagging Application

add new contacts.
Well, to be honest, the default representation for URLs is a plain string, which is not very appealing in terms of pictures. Instead of this URL, we would prefer to see the actual image the URL points to. Whenever the automatically generated representations of types do not fit, we are free to create a custom representation in OpenLaszlo and register it with the `Picture` type. How to do this is illustrated in section 6.3.5.

8 Conclusions

In this thesis, we strove for a way to improve personal information management (PIM) by incorporating concepts found in today's web applications. It has been shown that one central paradigm of these Web 2.0 applications is that they can be customised and extended by the end-users. Portals such as Facebook and Ning enable ordinary users to install and even develop their own applications for these platforms in order to extend the core functionality offered by them. The core idea behind our approach was to adopt those notions for PIM and thereby empowering users to become designers of their own PIM applications. For this reason, we combined traditional data management techniques with architectural patterns of Web 2.0 application.

First of all, an analysis of research results has been performed in the field of conventional data management in general and PIM in particular and several key ingredients of PIM systems were identified. Firstly, the use of a single, uniform data model is advised. Secondly, a classification or categorization system enables semantic grouping and role modelling which is helpful in data organisation. And lastly, associations and relations should be exposed as first-order concept to allow arbitrary relations between entities. Furthermore, evidence was collected to underpin the author's claim that a relational data organisation is superior to a hierarchical approach. After a discussion of the evident success of Web 2.0 applications that provide users with the ability to manage data from different application domains, it has been concluded that today's PIM solutions shall make a move towards the Web and provide a user interface similarly to Web 2.0 applications that is simple to use yet powerful enough to cater for the fact the users have different needs in terms of how they model and manage personal data. We also made the reader familiar with the Facebook Application platform and thereby motivated the introduction of a extensible, application-based system for personal databases. In addition, we revealed several advantages of a shared dataspace such as data reuse and combination of existing application domain models which is not possible with today's web applications.

As an immediate result from these studies, two systems have been presented that together facilitates the development of PIM applications, namely a component architecture for data

management and an extensible, web-based user interface. The first system is an extension of the object-oriented database system *OMS Avon* and has been implemented as a module that can be loaded at runtime. In section 4.2, we have defined the requirements for such a component architecture on the database layer. A module has then been developed that introduces the notion of *Information Components*. Information Components provide a mean to group OM models into distinct application domains and allow them to be reused, composed and extended. In a second step, we illustrated the approach which has been taken to make use of Information Components in order to realise the idea of a user interface to create and manipulate PIM applications via the Web. In section 5.1 we have defined the requirements for such a web application that builds on top of our Information Components module.
As a proof of concept, a scenario has been implemented with the aid of the MyApp Builder web application. In that scenario, the user starts with the design of simple *Contacts* and *Pictures* management applications. These two applications are then combined by the user to tag contacts from the *Contacts* application on pictures from the *Pictures* application.

8.1 Summary of Contributions

Information Components Module OMS Avon has been extended by incorporating the concept of reusable components into the data management layer. Similarly to database schemata, an Information Component groups all model entities that belong to the same application domain. This additional abstraction enables application developers to share the same information space, i.e. the same database because the different application domains are now clearly separated. However, they can specify which parts of the domain model encapsulated by an Information Component can be reused by other applications.

An intuitive user interface to create new PIM applications On top of the Information Components module, the *MyApp Builder* web application was developed to create new PIM applications interactively. It features a type manager to create types that define how entities and objects are represented in the PIM applications. Users can then create PIM applications based on collections and associations. We therefore promote categorization and relationships as the main building blocks to organise personal information.

A hybrid approach towards dynamic user interfaces We presented an algorithm that dynamically generates UIs based on the application domains defined by the users with *MyApp Builder*. However, the novelty was not the algorithm itself but the approach we have taken by combining a pre-compilation step on the server-side at design time with the expressiveness of OpenLaszlo's declarative UI markup language to provide interactivity on the client-side at runtime.

Injection of customized views As a direct result of the generation of OpenLaszlo code on the server, we were able to demonstrate how to streamline user-generated code snippets into the overall UI generation process. The current system supports the injection of custom visual representations for user-defined types.

8.2 Future Work

It is in their very nature that software systems are most likely never all-embracing. In fact, it is often desirable to have a system that offers several extension points and serves as a base for future work. In this section, several ideas are presented that are either complementary to the existing system or that are meant to upgrade the system itself to provide additional functionality. In terms of the user interface, this also includes suggestions to improve the user experience without necessarily adding any new application functionality. Of course, changes to the Information Components Module will probably influence the web interface and vice versa. However, the propositions have been divided into two parts according to their greater impact.

8.2.1 Information Components Module

- An Information Component contains collections and associations and may import them also from other Information Components. Types however are shared by all Information Components. Since collections are always linked with a member type, it could make sense to include them as parts of the corresponding Information Component. As with all component parts, they can then be reused by other Information Components and all modifiers such as visibilities are applicable to them as well.

- Information Components are intended to encapsulate entire application domain models in the database. It is therefore reasonable to think about a mechanism to share such models between databases. From the author's point of view, two approaches are feasible. Either Information Components are serialised to OML and imported on the other side or they can be shared with a framework that supports OM database objects directly, such as the information sharing platform [11]. In either case, it is necessary to resolve all dependencies not only within an Information Component (in order to include types and other objects as well) but also across Information Components as they can include parts from other components.

- Currently, the data managed by Information Components is completely isolated and limited to a single database. The IC module could be extended by a sharing and synchronisation engine. Such a synchronisation framework has already been implemented [15] as a module for OMS Avon and could therefore be used by the IC module directly. While the synchronisation framework has been designed to work on the level of collections, it could be adopted to support synchronisation for entire Information Components. The framework supports synchronisation with external data providers such as Facebook or Outlook. Synchronisation of data between the same ICs in different Avon databases is another issue that has to be addressed separately.

- While there is already a modifier to control whether parts can be reused, additional modifiers could be added in order to allow for a more fine-grained access control mechanism. For example, one could attach read and write permissions to reused part that specify whether the including Information Component can modify the parts (and their content) or not.

- In order to fully integrate a module into the Avon ecosystem, extensions to its database language are indispensable. Suggestions for all major concepts of the Information

Component metamodel should be elaborated and implemented as soon as the Avon OML engine is available.

8.2.2 MyApp Builder and Web Interface

- Since MyApp Builder has been designed with the casual user in mind, it is advisable that at some point a user study should be carried out. An evaluation of the current user interface could be performed and the comments and reactions from the audience could be used to further optimise the UI. It is also important to detect possible misunderstandings and provide additional information in areas that are not clear to the users.
- At the moment, MyApp Builder offers the following building blocks for the design of applications: collections, subcollections, associations and types. The web application could offer additional concepts from the OM model that can be used by users to model their application domains during the design phase. This includes but is not limited to *subcollection constraints*, *nested associations* and different *collection behaviours* (set, bag, sequence and ranking). From a user perspective however, different collection behaviours make only sense if a type defines a primary or unique key.
- The current type manager exposes only a subset of the typing layer. It should be extended to support more complex types such as structured types and nested types. Such an extension would also require thinking about visualisations for these types. Especially arbitrary nesting is a concept that is difficult to cope with in a traditional GUI and might lead to a navigation style which is closer to the (hyper-)link approach found in today's web applications where links may lead to a completely different web site altogether.
- As soon as the amount of data to be managed increases, there is a need for querying facilities. In that respect, OMS Avon supports a query language to retrieve filtered information from the database. The real challenge is to offer that functionality on the UI level in such a way, that ordinary user can utilise this query engine without a hassle and even get support in formulating complex queries. Another possible extension of the user interface would be an attribute filter that can be used to restrict the displayed members of a collection to those which fulfil the specified requirement.
- Within the user interface, a default attribute specifies the attribute to be displayed when members of a collection have to be presented in a compact form, for example as entries of lists. Instead of a single default attribute, more complex expressions could be supported to better define how information is presented to the user. Another idea would be to use grids instead of lists and to offer the possibility to customize them, i.e. add and remove columns. A grid would also allow to sort members of a collection according to an attribute.
- The dashboard could be extended with multi-user capabilities in order to provide a different set of applications for different users of the system. Additionally, full multi-user support could be implemented for the whole system. With such an environment in place, the same applications could work on different databases but would of course still comply with the same application domain model.

A
Framework Comparison

A.1 Flex

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">

    <mx:Script>
        <![CDATA[
            import mx.rpc.events.ResultEvent;
            import mx.rpc.events.FaultEvent;
            import mx.controls.Alert;

            [Bindable]
            private var response:String = "Result";

            public function handleXML(event:ResultEvent):void
            {
                response = event.result.message;
            }
        ]]>
    </mx:Script>

    <mx:HTTPService result="handleXML(event);" id="userRequest" ⤸
    resultFormat="e4x"
        url="http://www.flickr.com/services/rest/" useProxy="false">
        <mx:request xmlns="">
            <method>flickr.test.echo</method>
            <message>HelloWorld</message>
            <api_key>YOUR_API_KEY</api_key>
        </mx:request>
    </mx:HTTPService>

    <mx:Button x="10" y="10" label="Button" click="userRequest.send();"/>
    <mx:Label id="myLabel" x="10" y="40" text="{userRequest.lastResult.message}" ⤸
    width="65" height="19"/>
    <mx:Label x="10" y="70" text="{response}" width="65" height="19"/>

</mx:Application>
```

A.2 JavaFX

```
package flickrecho;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.ext.swing.SwingButton;
import javafx.ext.swing.SwingTextField;
import javafx.scene.layout.VBox;
import javafx.async.RemoteTextDocument;
import javafx.io.http.HttpRequest;
import javafx.data.pull.Event;
import javafx.data.pull.PullParser;
import javafx.ext.swing.SwingLabel;

var resultString = "Result" as String;
def request = HttpRequest {
    location: "http://www.flickr.com/services/rest/?method=flickr.test. ⤸
    echo&message=HelloWorld&api_key=YOUR_API_KEY";
    onInput: function(is: java.io.InputStream) {
        try {
            parser.input = is;
            parser.parse();
        } finally {
            is.close();
        }
    }
}

def parser = PullParser {
    onEvent: function(event: Event) {
        if (event.type == PullParser.END_ELEMENT) {
            if(event.qname.name == "result") {
                resultString = event.text as String;
            }
        }
    }
}

def resultLabel = SwingLabel {
    text: bind resultString
    width: 200
}

Stage {
    title: "FlickrEcho"
    width: 250
    height: 200
    scene: Scene {

        content: VBox{
            spacing: 10
            content:[SwingButton {
                    text: "Click Me"
                    action: function() {
                        request.enqueue();
                    }
                }, resultLabel]
        }
    }
}
```

A.3 Silverlight

A.3.1 XAML

```
<UserControl x:Class="FlickrEcho.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
        <Button Margin="35,47,199,0" VerticalAlignment="Top" Content="Button" ⤸
        Click="Button_Click"/>
        <TextBlock Name="myLabel" Margin="35,140,199,101" Text="Result" ⤸
        TextWrapping="Wrap"/>
    </Grid>
</UserControl>
```

A.3.2 C# Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Xml.Linq;

namespace FlickrEcho
{
    public partial class Page : UserControl
    {
        string url = "http://api.flickr.com/services/rest/?method=flickr.test.⤸
        echo&message=HelloWorld&api_key=YOUR_API_KEY";

        public Page()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            WebClient service = new WebClient();
            service.DownloadStringCompleted += new ⤸
            DownloadStringCompletedEventHandler(DownloadMessageCompleted);
            service.DownloadStringAsync(new Uri(url));

        }

        private void DownloadMessageCompleted(object sender, ⤸
        DownloadStringCompletedEventArgs e)
        {
            XDocument xml = XDocument.Parse(e.Result);
            string response = xml.Element("rsp").Element("message").Value;
            myLabel.Text = response;
        }
    }
}
```

A.4 OpenLaszlo

```
<canvas>
    <include href="../lps/components/lz/edittext.lzx" />
    <include href="../lps/components/lz/button.lzx" />
    <include href="../lps/components/utils/layouts/simplelayout.lzx" />

    <dataset name="dset" type="http"
        src="http://www.flickr.com/services/rest/"/>

    <window x="200" y="200" width="400" height="400" title="Flickr Echo" ⤸
    resizable="true">
        <simplelayout axis="y" spacing="10" inset="10"/>
        <button>
            Call Webservice
            <handler name="onclick">
                Debug.write("Yeah")
                dset.setQueryString({
                    method : 'flickr.test.echo',
                    message : 'HelloWorld',
                    api_key : 'YOUR_API_KEY'
                });
                Debug.write(dset.getQueryString());
                dset.doRequest();
            </handler>
        </button>
        <text datapath="dset:/rsp/message/text()" name="myLabel"/>
    </window>
</canvas>
```

B Graphic Resources

Several icons and graphics from third-party contributors have been used in the web application and as part of some illustrations of this thesis. All of them are free for non-commercial use but require attribution to the author(s).

B.1 VistaICO Toolbar Icons

The VistaICO Toolbar Icons set[1] is a pack that contains 50 quality icons (arrows, cut, copy, help, symbols etc.) in PNG format which can be used in applications, websites or other projects. It was created by VistaICO.com and has been released under the *Creative Commons Attribution 3.0 Unported* license. Icons from this pack have been used in the web application.

B.2 Yuuyake Icons

An Asian-themed pack of 51 application icons. It was created by Gary Leleu[2] and has been released under the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported* license. Icons from this pack have been used for the application icons of the dashboard application.

B.3 Old School Icons

A set of 82 icons featuring graphics for applications and folders in a old-school-style. It was created by Sebastien Durel[3] and has been released under the *Creative Commons Attribution-Noncommercial 3.0 Unported* license. Icons from this pack have been used for the illustrations in figure 4.1 and figure 4.2.

[1] http://www.vistaico.com/
[2] http://www.datenshi.1k.fr/
[3] http://babasse.deviantart.com/

List of Figures

Acknowledgements

Stefania Leone has deserved my uttermost gratitude for the great support she gave me during the creation of this thesis. She was the only person who had the "pleasure" to read through all my drafts, proposals, artefacts and alpha, beta, preview versions. If it ever happened that I lost sight of my goals, she was the one who showed me the right direction again. This thesis would never have been possible (and that long) without her weekly inputs and suggestions.

Special thanks go to Prof. Norrie for supporting my ambitions in this research topic.

I am extraordinarily grateful to my parents for their support and for enduring all the various moods I went through during the writing process, whether it was euphoria, desperation, happiness or sadness.

I would also like to thank my peers in the Lab for all the distractions which made life much more interesting. Christoph the First who was never lost for jokes, my antagonist Philip who dared to question the greater meaning of my thesis, Katinka who was my combatant in the never-ending fight against OpenLaszlo and Christoph the Second (but no less important), our db4o guru.

Bibliography

[1] Eytan Adar, David Karger, and Lynn Andrea Stein. Haystack: per-user information environments. In *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*, pages 413–422, New York, NY, USA, 1999. ACM.

[2] Lukas Blunschi, Jens peter Dittrich, Olivier René Girard, Shant Kirakos, Karakashian Marcos, and Antonio Vaz Salles. A dataspace odyssey: The imemex personal dataspace management system. In *In CIDR*, pages 114–119, 2007.

[3] Vannevar Bush. As we may think. *Atlantic Monthly*, 176(1):101–108, Juli 1945.

[4] Yuhan Cai, Xin Luna Dong, Alon Halevy, Jing Michelle Liu, and Jayant Madhavan. Personal information management with semex. In *In SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 921–923. ACM Press, 2005.

[5] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, and Regis Saint-paul. Understanding ui integration: A survey of problems, technologies, and opportunities. Technical report, IEEE Internet Computing. May/June, 2006.

[6] Jim Gemmell, Gordon Bell, and Roger Lueder. Mylifebits: a personal database for everything. *Commun. ACM*, 49(1):88–95, 2006.

[7] Michael Grossniklaus. Lecture in web engineering, March 2008.

[8] David Huynh. Haystack: A platform for creating, organizing and visualizing information using rdf. 2002.

[9] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24, New York, NY, USA, 2007. ACM.

[10] David R. Karger and Dennis Quan. Collections: flexible, essential tools for information management. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1159–1162, New York, NY, USA, 2004. ACM.

[11] Christoph Lins. Event-based information sharing. Master's thesis, ETH Zürich, 2009.

[12] M. C. Norrie. Distinguishing typing and classification in object data models. In *In Information Modelling and Knowledge Bases, volume VI, chapter 25. IOS*, 1995.

[13] Moira C. Norrie. An extended entity-relationship approach to data management in object-oriented systems. In *In 12th Intl. Conf. on Entity-Relationship Approach*, pages 390–401. Springer-Verlag, LNCS, 1993.

[14] Tim O'Reilly. What is web 2.0? design patterns and business models for the next generation of software. www.oreilly.com, September 2005.

[15] Martin Schnyder. Web 2.0 data management. Master's thesis, ETH Zürich, 2008.

[16] Jin Yu and Boualem Benatallah. A framework for rapid integration of presentation components. In *In the Proceedings of WWW'07*. ACM Press, 2007.