Doctoral Thesis

The eXtreme Design approach

Author[s]:
Köbler, Adrian

Publication Date:
2001

Permanent Link:
https://doi.org/10.3929/ethz-a-004101569

Rights / License:
In Copyright - Non-Commercial Use Permitted
The eXtreme Design Approach

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZÜRICH
for the degree of
Doctor of Technical Sciences

presented by

Adrian Kobler
lic.oec.publ, dipl. Wirtschaftsinformatiker
born February 25, 1964
citizen of Zürich, Switzerland

accepted on the recommendation of

Prof. Dr. M.C. Norrie, examiner
Prof. Dr. S. Spaccapietra, co-examiner

2001
Abstract

Developing software is a complex task involving many technical and non-technical processes. To ensure high quality software systems, adequate software engineering methods must be applied which subdivide the software development process into meaningful phases.

Object-oriented software engineering techniques are suited to managing the complexity of software development and facilitate the decomposition of a software system into extensible and reusable components. This is a very important aspect since it must be taken into account that a software system must be adaptable to new requirements which might not be foreseeable while building it. Thus, it is crucial that the design of these components reflect the need for software evolution.

Software prototyping, component-based software construction and the use of design patterns are all means of coping with the complexity and extensibility of software systems. The main problem of designing software systems is to make appropriate decisions about the structure and interoperability of the various components. How should components be linked together? Which relationships between objects of the application domain need to be present? Which properties of an object, such as attributes and methods, must be specified? There are many more questions to be asked during the design process of software systems and components, and many attempts have been made to support this process by guidelines and methods.

In this thesis, we propose the eXtreme Design (XD) approach which can be regarded as a supplement to existing methods and models supporting the design process of software systems and components with respect to software evolution and persistent data management. XD is based on the XD meta model together with its algebra combining concepts of conceptual modelling and object-oriented software construction on a meta level.

This makes it possible that classification structures, relationships and object properties can be specified in such a way that they can evolve over time without having to change design and implementation specifications such as class and schema descriptions. Thus, also users can be involved in the evolution process of a software system since they can carry out changes, for example, by introducing new object properties without causing a redesign of the corresponding software components.

XD is especially tailored for designing and implementing those parts of a software system which manage persistent data. We present how XD increases the flexibility and adaptability of these parts of the system by showing how we have applied it in the fields of prototyping, component modelling and rapid information modelling. We illustrate how we used XD for prototyping a product information system capable of
managing product variants. Further, as an example for component modelling, we de-
scribe how XD facilitates the extension of the persistent object management system
OMS Java. Finally, we outline how XD supports the construction of information
spaces meeting the demands for various information needs.
Zusammenfassung


Objekt-orientierte Software-Engineering-Techniken sind sehr geeignet für die Bewältigung der Softwareentwicklungskomplexität und erleichtern die Zerlegung eines Softwaresystems in erweiterbare und wiederverwendbare Komponenten. Dies ist ein sehr wichtiger Aspekt, da in Erwägung gezogen werden muss, dass ein Softwaresystem anpassbar sein sollte an neue Anforderungen, die nicht vorhersehbar waren als das System entwickelt wurde. Daher ist es sehr entscheidend, dass das Design dieser Komponenten die Erfordernisse für Software-Evolution widerspiegelt.


Dies macht es möglich, dass Klassifikationsstrukturen, Objektbeziehungen und Objekteigenschaften in einer solchen Art spezifiziert werden können, dass sie sich mit der Zeit weiterentwickeln lassen, ohne dass Design- und Implementationsspezifikationen wie Klassen- und Schemabeschreibungen geändert werden müssen. Daher können auch Benutzer in den Entwicklungsprozess eines Softwaresystems miteinbezogen werden, da sie Änderungen vornehmen können – zum Beispiel um neue
Objekteigenschaften einzuführen – ohne ein Redesign der entsprechenden Softwarekomponenten zu verursachen.

Acknowledgements

While writing this thesis I realised how many people were supporting me during the last few years. Thank you for all the interesting discussions and your helpful assistance!

I enjoyed a very interesting time at ETH working on exciting projects together with people of the GLOBIS group to whom I owe a lot.

I also want to thank my co-examiner Prof. Spaccapietra very much for all his helpful advice and efforts to improve the quality of my thesis significantly.

I am deeply grateful that my supervisor Prof. Moira Norrie gave me the opportunity to work in her group. She supported me in every aspect of my work and helped me whenever possible to make this thesis become reality. Thank you very much!
## Contents

1 Introduction .......................... 1
  1.1 Software Development ................. 2
  1.2 eXtreme Design (XD) ................. 3
    1.2.1 Prototyping ...................... 4
    1.2.2 Component Modelling .......... 5
    1.2.3 Information Modelling .......... 6
  1.3 Contribution ........................ 7

2 Software Engineering and the Design Process 9
  2.1 Requirements Analysis: An Example .... 12
  2.2 Database Application Design .......... 15
    2.2.1 Database Design ................ 15
    2.2.2 Conceptual database design .... 16
    2.2.3 The Generic Object Model OM ... 19
    2.2.4 Data Model Mapping and Database System Implementation . 21
  2.3 Object-Oriented Design .............. 24
    2.3.1 Basic Object-Oriented Concepts ... 24
    2.3.2 The Object-Oriented Design Process .......... 27
  2.4 The eXtreme Design Approach .......... 32
3 eXtreme Design (XD)  

3.1 The eXtreme Design Meta Model  ........................................... 41
  3.1.1 The Generic Object Model OM ............................................. 43
  3.1.2 Classifying Objects ......................................................... 46
  3.1.3 Specifying Object Relationships ......................................... 49
  3.1.4 Defining Object Properties ............................................... 51

3.2 The XD Algebra ................................................................. 57
  3.2.1 Algebra Operations for Object Groups ................................... 59
  3.2.2 Algebra Operations over Meta Objects .................................. 60
  3.2.3 Algebra Operations for Object Relationships and Facts .............. 60
  3.2.4 Algebra Operations for Object Properties ................................ 61

4 Applying eXtreme Design ....................................................... 63
  4.1 eXtreme Design: an Example ................................................ 64
  4.2 The eXtreme Design Framework (XDF) ...................................... 69
  4.3 XD Meta Language (XDML) .................................................... 72

5 eXtreme Design for Product Data Management ............................... 76
  5.1 The Design Process ............................................................. 77
    5.1.1 Product Classification ................................................... 78
    5.1.2 Product Characterisation ................................................ 78
    5.1.3 Product Structuring ....................................................... 79
    5.1.4 Product Configuration .................................................... 79
    5.1.5 Design and Implementation Issues .................................... 86

6 The Persistent Object Management Framework OMS Java ................. 89
  6.1 OMS Java Multi-Tier Architecture ......................................... 90
  6.2 OMS Java Core System ....................................................... 91
    6.2.1 Basic Concepts ........................................................... 91
Chapter 1

Introduction

The eXtreme Design approach supports the process of software development. In particular, eXtreme Design (XD) facilitates the design process of software components and can be regarded as an integral part of software engineering. The IEEE Standard 610.12 defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and

(2) The study of approaches as in (1).

Not all people would fully agree with this definition. Some see the creation of software as “primarily an artistic endeavour; others consider it more mathematical, while others believe that process and method are key [PGL00].”

Probably, software engineering is a mixture of applying artistic as well as scientific procedures:

“The conception of a new structure can involve as much a leap of the imagination and as much synthesis of experience and knowledge as any artist is required to bring to his/her canvas or paper. Once the design is completed, it must be analyzed by the engineer as scientist in as rigorous an application of the scientific method as any scientist must make [Pet00].”

We claim that eXtreme Design is an approach suitable for managing the transformation of knowledge and experience into software design serving as a basis for application development, or being itself the final implementation of a software component.
This transformation process comprises mainly the following activities: Recognising entities in the application domain, specifying their characteristics, defining relationships among entities, and categorising them according to certain criteria.

"Categorization, the process by which distinct entities are treated as equivalent, is one of the most fundamental and pervasive cognitive activities. It is fundamental because categorization permits us to understand and make predictions about objects and events in our world. [WK99]"

By categorising information, people gain meaning of a specific domain of interest. This in itself is a prerequisite for being able to carry out deliberate actions and make decisions.

1.1 Software Development

Software development is a creative and innovative process. To ensure high quality software systems, adequate software engineering methods must be applied. "Methods, at their best, help people move to greater heights of creativity and innovation [CNM97]."

Most methods subdivide the software development process into several stages which are iteratively applied and which correspond to the tasks of analysis, design and implementation. For example, the method proposed in [DR99] includes stages for

- requirements analysis in which the real-world domain objects and the relationships among those are identified resulting in high-level class diagrams etc.
- analysis and preliminary design in which actions are specified and the diagrams are supplemented by objects that accomplish the stated scenario
- design in which the behaviour as well as the static model is detailed and verified
- implementation which includes developing, testing and delivering the software system.

Since software engineering is a creative process, there are no strict rules which can or must be carried out for solving a specific problem. But it is often the case that domain specific problems share common characteristics with problems of other domains described by the concept of design patterns. A design pattern is adaptable to solve a specific problem. "Design patterns, like most traditions, help establish a context for your solution to a design problem. Your imagination then takes over to move beyond the tradition to solve the problem. [Mul99]"
Further, a good software engineering method must take into consideration the fact that not all people involved in the software development process share the same experience, and that the “development of a software system is usually just a part of finding a solution to a larger problem. [CLF93]”

Hence, it is increasingly the case that it is not possible to use only one method for all stages of the software engineering process. Designers, for instance, typically commit to one software design method, but often find that, for example, its notation is not rich enough to express concepts of the application domain. So they attempt to borrow useful ideas and notations from other methods [Son97]. Combining methods on the one hand can cause inconsistencies in notation and terminology, but on the other hand are a necessity because existing methods do not cover all aspects of software development adequately.

For example, object-oriented software engineering methods such as UML [BRJ99] and OPEN [GHSY97] focus more on describing the structure of a software system and on defining the interaction between application objects, respectively, specifying the software development process, whereas methods for information engineering, such as those for conceptual modelling, emphasise “the semantics of the entities and relationships, including descriptions of connections and consistency constraints” [PM88].

Since modern applications tend to have a database component as an integral part, data management becomes more and more a crucial aspect of application systems and it is necessary that the development process considers aspects from information engineering as well as software engineering. The eXtreme Design approach supports the design process of such applications by combining concepts from software and information engineering in such a way that it is possible to specify concepts of the application domain on a high level of abstraction which facilitates not only the development of extensible and adaptable application components, but also motivates prototyping which can be regarded as “the process of quickly building and evaluating a series of concrete, executable models of selected aspects of a proposed system” [LG97].

### 1.2 eXtreme Design (XD)

The eXtreme Design (XD) approach supports the design process of applications and application components. XD supplements existing software engineering methods such as UML [PS99] in the area of domain modelling. Domain modelling is the task of discovering entities and objects of the application domain which represent real-world things and concepts [DR99].

XD combines object-oriented concepts with aspects of conceptual modelling on a higher-level of abstraction. XD is based on the XD meta model which is adaptable to
various application domains, and the XD algebra which offers a rich set of operations over the XD meta model.

XD supplements existing methods in three different areas: prototyping, component modelling and information modelling. We discuss each of these in turn.

1.2.1 Prototyping

Building complex applications and information systems is a task of iteratively analysing, designing and implementing concepts of the application domain. It is not always clear at the beginning of a development cycle which aspects and requirements of a software system are most crucial. Hence, to get more clues about user behaviour and application processing requirements, it is good practice to build a prototype. The goal of the prototyping activity is to reduce risk of failure and to discover critical properties of a software system before “making final design decisions that would be expensive or impossible to alter later in the life cycle [PLC91].”

Prototyping is a very iterative activity which comprises steps such as [PLC91]

- **Identifying objectives**
  The problem domain must be specified together with the criteria of success measurement which can usually be derived from requirements analysis. This can be done, for example, by using existing methods and their notations such as UML.

- **Identifying risks**
  Since by prototyping we should obtain insights about specific software components, it is not at all clear whether the problem domain definition carried out in the previous step is a good starting point. This risk must be identified and made explicit.

- **Implementing the prototype**
  If the risk analysis shows that the problem in hand is unclear, we focus on examining a variety of design options to which we refer as exploratory prototyping. On the other hand, by experimental prototyping we want to find out more technical aspects of software requirements from the user’s perspective, whereas by building an evolutionary prototype we place emphasis on the adaption of an application system to rapidly changing organisational constraints [LSHZ93].

- **Using the prototype**
  By using the prototype either within a test environment or, if it is an evolutionary prototype, as part of the target system, we gather more detailed information about system requirements. It is very important during this stage that changes due to new requirements and user feedback can be incorporated into the prototype very easily.
• *Evaluating results and iteratively refining the prototype*

The results of using the prototype have to be evaluated with regard to the problem domain definition. This should help to decide whether the crucial system requirements have been identified, or whether it is necessary to reformulate the problem domain and redesign the prototype.

We argue that the eXtreme Design approach is very suitable for prototyping because the underlying meta model and algebra make it possible to define concepts of the application domain on a very high level of abstraction, i.e. new concepts can be introduced seamlessly and existing ones can be changed easily. Using XD results in a high-fidelity prototype which can be characterised as being functionally complete, fully interactive, user-driven, defining the navigational scheme, and serving as a living specification [RSI96].

1.2.2 Component Modelling

A very important aspect of an application system is its openness in the area of topology (distributed systems), platform (heterogeneous hardware and software) and evolution (rapidly changing requirements) [Sch99]. *Component-based software development* addresses these issues by separating the components of an application system from the specification of their composition and cooperation. Hence, a component-based application system consists of components and scripts which define the composition and interaction of components [Sch99].

Component-based software development requires a software-component model comprising four essential characteristics [HS00]:

- **Component**
  A component can be regarded as a self-contained software construct serving a well defined purpose. It specifies a run-time interface and can be autonomously deployed.

- **Component socket**
  A component socket provides a well-defined run-time interface to a supporting infrastructure into which the component will fit.

- **Component development**
  The main focus of component development lies on building components for composition and collaboration with other components.

- **Component deployment**
  There must be appropriate tools for deploying components and their corresponding component sockets.
A good example for a component model and component architecture is *Enterprise JavaBeans* (EJB) providing complex middleware features so that a developer can focus on "writing the applications that solve real-world problems, rather than on all the overhead that goes with distributed server-side systems [Rom99]."

There are several dimensions of component technology to be taken into account [Dog98]:

- **Component granularity**
  Component granularity ranges from integers, stacks etc. to large-scale application components.

- **Technology targets**
  Components can be used for graphical user interfaces, databases, communication infrastructure, interoperability frameworks, etc.

- **Development life-cycle phases**
  Components can be the result of different development life-cycle phases such as reusable analysis and design models. For components, we can distinguish the phases *design-time, compose-time* and *runtime* [BW98].

Using the eXtreme Design approach for component construction makes it possible to specify components which are highly adaptable even at runtime because properties of components can be added, removed and changed without having to redeploy a component. Further, the XD meta model neatly fits into existing component environments since its algebra supports the specification of concepts on a very high level of abstraction making them independent of a particular component model.

### 1.2.3 Information Modelling

Information modelling can be understood as the activity of designing *information spaces* representing specific domains of interest. To gain value from an information space it is necessary to be able to make out entities within the information space, to classify them according to user defined criteria, to create relationships between them and to characterise them by specifying entity properties.

Information modelling can be carried out by using existing data models, for example, for defining a conceptual schema of the information space, or by applying object-oriented design techniques. In both cases we need to identify entities of the information domain, their relationships, classification structures and properties in order to be able to implement a specific application.

But sometimes an information space is semi-structured, dynamically changing, or cannot easily be disentangled from other information spaces. Thus, an application supporting information modelling for such types of information spaces should make it
possible to add, update and remove categories, relationships and properties without having to change schemas or application components.

We claim that the eXtreme Design approach is especially suitable for building application systems for information modelling activities because of the flexibility of the XD meta model with regard to classification, specifying relationships, and structuring of information domain entities.

1.3 Contribution

The design process is a very crucial phase of software development. Well-designed software components reduce development and maintenance costs, and are better adaptable to new requirements. Designing software is a very creative and complex process which should be supported by adequate design methods. Since it is seldom the case that a given design method is suitable for all parts of a software system, it is important that a designer can combine design approaches where needed. Especially if we look at database-oriented applications, i.e. applications that comprise a database component, a design method must be capable of supporting software engineering as well as information engineering tasks.

The contribution of this thesis is to provide a design approach which can be used in conjunction with other design methods for developing highly adaptable and database-oriented software components as well as for supporting the activity of rapid information modelling and prototyping. This is achieved by specifying a meta model together with its algebra based on ideas from conceptual modelling, object-oriented software construction and meta modelling. This makes it possible to define concepts of the application domain on a very high level of abstraction so that, for example, creating or modifying classification structures, object relationships and object properties can be carried out without having to change schema information or type specifications. And since the meta model itself is defined in terms of the generic object model \( OM \) [Nor93, Nor95], it can be easily extended and adapted to new requirements.

In chapter 2, we give an overview of software engineering and information engineering concepts on which the eXtreme Design approach is based. The approach itself, the meta model and its algebra are presented in chapter 3.

We implemented two frameworks supporting the eXtreme Design approach. One uses the object management system \( OMS \) \( Java \) as its core. OMS Java supports the generic object model OM and is part of the OMS Database Development Suite [KNW98]. The meta model is in this case defined as an OM schema and the algebra operations are implemented using the OM algebra provided by the OMS Java application programming interface. Thus, it is very straightforward to extend the meta model by changing the schema definition. The other framework implements the meta model and its algebra directly on top of a storage management compo-
nent making it very efficient but less adaptable. These frameworks can be used for applying the eXtreme Design approach and are described in chapter 4.

Chapter 5, 6 and 7 then go on to describe how the eXtreme Design approach has been used to implement a variety of applications. These applications not only validate the approach, but also show its flexibility and extensibility.

In chapter 5 we illustrate how we have used the eXtreme Design framework for prototyping a product information system. The main goal was to develop a system which can be used for classifying products and for defining their structure. Further, the system supports the process of product configuration.

In chapter 6 we show how we applied eXtreme Design for implementing components for the object management system OMS Java as an example for component modelling. OMS Java is an integral part of the OMS Database Development Suite and as such serves as a development platform. We built the OMS Java system with extensibility in mind. It is possible, for example, to extend the core object model OM with new semantics such as temporal information and to exchange or extend the algebra, constraint and language component. This was achieved by using various design patterns [GHJV95] for system design. Although design patterns help to make a system adaptable, it is not possible to foresee all future requirements which often causes parts of the system to be changed or redesigned. For example, introducing new data structures for bulk information processing requires the storage management component to be extended. By applying the eXtreme Design approach, we were able to form the storage management component of OMS Java in such a way that it is possible to add new data structures without having to carry out changes to the component.

In chapter 7 we discuss rapid information modelling (RIM). We describe the results of a supervised project showing how it is possible to extend the eXtreme Design approach with additional concepts. Further, we give an example of how to use the XD framework for implementing RIM applications.

Concluding remarks and a discussion of future work are given in chapter 8.
Chapter 2

Software Engineering and the Design Process

Developing software is a process involving many technical and non-technical issues such as requirements analysis, design and implementation. The resulting software system should provide the required services, be maintainable, reliable, and efficient [Som89]. Other important aspects of software quality are correctness, extendibility, reusability, and compatibility [Mey88]. To achieve this, appropriate software engineering techniques and development paradigms must be applied supporting tasks such as requirements analysis and definition, system and software design, implementation and testing, and operation and maintenance [Som89].

For example, software prototyping addresses these issues by iteratively building a working prototype of the software system which is intentionally incomplete at the beginning, and by providing mechanisms for users to try it out and give feedback. Further, a software prototype should be easily extensible and modifiable [GLZ99]. By focusing on rapid user feedback, simplicity, incremental changes and quality work [Bec00], it is possible to evolve a prototype into the final system.

Another very important aspect of the software life cycle is software evolution which can be regarded as the phase in which we “adapt the application to the ever-changing user requirements and operating environment [BR00]”. Thus, it is crucial to build software systems and components with the evolution phase in mind, so that extra capabilities can be added to a system keeping changes minimal and not corrupting system invariants but rather enhancing them.

This can be achieved by specifying concepts of the application domain on a high level of abstraction, for example, defining roles for application domain entities or objects. “The purpose of a role is to model different ‘active’ (application-specific) representation facets for the same object in terms of both structure and behavior [PK00].” Hence, new requirements can be introduced by adding new roles to an existing software component which means that a particular object of the application
domain can evolve by gaining additional properties such as attributes and methods through specific roles.

Software evolution demands software systems which are adaptable and reusable. This is a fundamental aim of object-oriented software construction in which the issue of design reuse is addressed by object-oriented techniques such as subclassing, frameworks and design patterns. For instance, design patterns help recognising forms of evolution for application domain objects by identifying aspects of a system that may vary [SPL96] and therefore improve the structure and extensibility of software systems as well as facilitate the development of reusable software components.

A component or subject in this context can be specified as being a “collection of classes, defining a particular view of a domain and/or providing a coherent set of functionality” [OT99]. By integrating classes into separate subjects which then are used for building a software system, we increase the flexibility of a system for future extensions caused by new requirements.

What is also important to take into consideration is the evolution of persistent data stored, for example, in a database management system (DBMS). If we change, for instance, properties of application objects, this needs to be reflected in the persistent storage. In the case of a DBMS, also the schema describing the structure of data in the database needs to be adapted which we call schema evolution. Basically, we can distinguish the following types of schema changes for an object-oriented system [BCG+87]: changes to object properties such as attributes and methods, changes to the class or type hierarchy and changes to class or type definitions.

In addition, we also want to be able to re-use existing data coming from various data sources. This might be supported on different levels of integration [PS00]: At the lowest level, one system can obtain data from another in a typical client/server manner such as by using gateways. Another way is to enable the user to access and manipulate data from several sources implying that the user is responsible for consistency across database boundaries. At a higher level, a global system provides the needed level of integration of data sources including tasks such as unifying the description of the input schemas as well as supporting integrated access to existing data.

All these issues - software evolution, extensibility, persistent data, data source integration - have a major impact on the design process of software systems. The eXtreme Design (XD) approach facilitates the design process with respect to software evolution and persistent data management. In contrast to other software engineering approaches, it is possible to carry out changes in classification structures and object properties without having to change class descriptions or schema information since the concepts of the application domain are specified on a meta level using the XD meta model together with its algebra. Applying XD improves the design of those software components which manage persistent data and which need to be adaptable to future requirements in terms of classification structures, object relationships and object properties. Because of the flexibility of the XD meta model
and its algebra, XD is also suitable for specifying the interoperability semantics for data source integration.

The goal of the eXtreme Design (XD) approach is to support the task of designing and implementing software applications and components and, as such, it can be part of an existing software engineering paradigm. XD can be regarded as a supplement to existing design methodologies in the area of object-oriented design facilitating the design process of software components in such a way that both users and developers are involved.

“In some ways ’design’ is the opposite of analysis. In analysis we break things down into standard and recognizable parts. In design we put things together to achieve a value and a purpose. [Bon00]”

In the context of software development, design can be defined as “the practice of taking a specification of externally observable behaviour and adding details needed for actual computer system implementation, including human interaction, task management, and data management details. [CY91b]”

In other words, “Design encompasses the disciplined approach we use to invent a solution for some problem, thus providing a path from requirements to implementation. [Boo91]”. This basically means that “Designing is creating a structure that organizes the logic in the system. [Bec00]”

Thus, before starting the design process, requirements collection and analysis has to be carried out. This includes, for example, the task of identifying and analysing the intended uses of a system which can be done by applying specific requirements specification techniques such as the use of diagrams for visualising information-processing requirements [EN94].

For example, use case diagrams document the behaviour and functionality of a system together with the main involved actor roles from the user’s point of view and are easy to understand without knowing the notation [PS99]. The concept of use cases was first introduced in [JCJO92] and is now part of the OMG Unified Modeling Language (UML) [Obj]. Use cases can help capturing the requirements by providing a structured way to identify the actors which represent the roles that someone might play, and by specifying what each actor needs from a system, i.e. which use cases have value for an actor, as well as other interactions they expect to have with the system.

The goal of the software design process is to transform and adapt the results of the analysis process into design specifications which are suitable as the basis for the implementation process. We typically define behavioural as well as structural aspects of a software system and represent the result using a modelling language such as UML. “The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system” [Obj]. UML consists of several languages such as use case diagrams, class and object
diagrams, and different forms of behavioural and implementation diagrams. Since different problem domains require different development processes, UML focuses on standardising the modelling language and not the process itself, thereby making it possible to use UML in the context of many domain specific process methods.

Object-oriented design (OOD) "lets system designers encapsulate data and behaviour in discrete objects that provide explicit interfaces to other objects" [MKMG97]. Further, a message-passing abstraction defines the communication channels and connections between objects of the application domain. Object-oriented concepts help to specify structural and behavioural aspects of a specific problem domain on a very high level of abstraction which facilitates the modelling of complex problem domains. Basic activities in OOD are, for example, refining the results of the analysis process, identifying important objects in the problem domain, identifying the operations on these objects, specifying object responsibilities, roles and relationships. The results of the OOD process usually are represented using a modelling language such as UML and serve as input for the implementation process.

The eXtreme Design (XD) approach supplements existing object-oriented design methods in such a way that the roles, relationships and properties of application domain objects identified during the object-oriented design process can be specified using the XD meta model together with its algebra which both support essential constructs necessary for object-oriented as well as database design. Especially for those parts of a system which manage persistent objects, i.e. objects that are stored in and retrieved from a non-volatile storage component such as a database management system, it is an advantage to use XD because the concepts of the application domain can be specified on a very high level of abstraction facilitating the design of classification structures and object properties.

This is achieved by combining conceptual modelling and object-oriented concepts. We therefore discuss these topics in section 2.2 and section 2.3 together with the main concepts of the software design process. To illustrate the various design methodologies, we introduce a simple example in the next section which can be regarded as the result of the analysis phase. The eXtreme Design approach itself is outlined in section 2.4.

2.1 Requirements Analysis: An Example

The first step in the software development process is to analyse the application domain and to specify the requirements. The results of this analysis phase are usually recorded in a set of documents and diagrams which define the important concepts of the application domain and which are basis for the design process. For the sake of simplicity, we give in this section only a short description of the requirements without specifying the corresponding diagrams.

Suppose a manufacturing company produces and sells furniture. The task is to
implement a new product information system providing ready access to product data for a variety of employee types such as project managers, designers and salesmen. Such information includes the types of products available, product characteristics and details of product structures including variants and configurations.

In our example, the company manufactures three different kinds of furniture which we refer to as product groups: chairs, lamps and beds. Since the company offers three different models of chairs, the product group “chair” comprises the product group “garden chair”, “living room chair”, and “kitchen chair”. Each product can be ordered in different variants. Such a variant is, for instance, the product “garden chair”. The term product covers single part products as well as composed products. “garden chair” is an example of a composed product, and is therefore composed of other products such as “chairleg” and “armrest”. For the sake of simplicity, our chairs consist of only three parts – the seat, armrests and legs – as shown in figure 2.1.

![Chair Diagram](image)

Figure 2.1: The main Parts of a Chair in our Example

Further, composed products and single part products are specified by characteristics such as colour, weight, and so on. Figure 2.2 gives an overview of product groups and products for the product group “chair”.

To fulfil all of these requirements, our product information system must at least support the following activities:

- **Product Classification**
  A convenient way of expressing the more conceptual aspects of products is to classify them according to certain criteria into different product groups.

- **Product Structuring**
  The result of this activity is usually a hierarchy of components. This hierarchy is described best with some sort of part_of relationship between products.

- **Product Characterisation**
  The characteristics of a product such as size, length and colour have to be identified together with their values.
Additionally, the system must be capable of defining rules, constraints, formulas and queries needed for specifying the overall product structure, and for supporting the product configuration process.
2.2 Database Application Design

A database application can be defined as an application that uses a database management system (DBMS) or a persistent storage engine (PSE) for managing application objects representing concepts of the application domain. Database applications typically share their persistent application objects with other applications and as such can be part of an information system “which includes all resources within the organization that are involved in the collection, management, use, and dissemination of information. [EN94]” Information in this context means a commodity, product, or thing which answers the questions when, where, who, and what [QD99]. In other words, information “is knowledge of ideas, facts and/or processes” [SW94]. Most information and software systems of today have a database as central part and comprise several database applications for accessing the database.

Database application development comprises activities such as requirements analysis, application design, implementation and testing. The overall goal is to produce a well engineered software system which is reliable, maintainable, efficient and user-friendly [Som89]. This can be achieved by applying appropriate software engineering techniques classified best according to the underlying development paradigm. For example, the well cited waterfall approach views the software process as a sequence of stages such as analysis, design and implementation, whereas prototyping emphasises more the iterative life-cycle aspect of software development resulting in a product-quality system.

Modern approaches also take into account the “unpredictable realm of increasingly complex systems” [Hig00]. For instance, the Adaptive Development Model subdivides the development life cycle into tasks such as speculating about the important dimensions of mission statements, collaborating manifested as interpersonal dynamics and collegiality, and learning as a means of gaining mastery through experience. “The overriding, powerful, indivisible, predominant benefit of the Adaptive Development Life Cycle is that it forces us to confront the mental models that are at the root of our self-delusion. It forces us to more realistically estimate our ability” [Hig00].

2.2.1 Database Design

The process of designing a database is just one of many activities in the development of database applications and information systems. The life cycle of an information system generally consists of activities such as feasibility study, collection and analysis of requirements, design, implementation, validation and testing, and operation and maintenance [AGPT00]. Database design is part of the design activity. Its main goal is to establish the structure and organisation of data, and it is closely intertwined and complementary to the task of designing database applications.

We further can subdivide the design process in several phases [EN94]:

15
• **Conceptual database design**
  The main goal of this phase is to specify the conceptual schema for the database in terms of a data model. "A data model is a set of concepts that can be used to describe the structure of and the operations on a database. [Nav92]" The conceptual schema should be independent from a specific DBMS and it is therefore recommended to use a high-level data model such as the entity relationship model, the object+relationship model $ERC+$ [SPS+95] or the generic object model $OM$ presented in section 2.2.3. A high-level data model makes it possible to model concepts of the application domain in a natural way by providing constructs for modelling objects, their relationships and roles on a high level of abstraction, i.e. independent of a specific implementation platform.

• **Data model mapping**
  In this phase we map the conceptual schema into the data model of the chosen DBMS (logical database design) which results in a conceptual schema of the chosen data model, for example a relational model.

• **Physical database design**
  This design represents the specification for the stored database in terms of physical storage structures.

We examine each of these stages in turn.

### 2.2.2 Conceptual database design

"Conceptual modeling is about organizing abstract ideas into concrete descriptions. [Emb98]" It takes place in the early phases of system development and addresses two complementary aspects: the conceptual product and the conceptual modeling process [RC92].

The conceptual product or conceptual schema describes the system to be built in terms of a high-level modelling language which is best done using a DBMS-independent data model in the case of conceptual database design. The schema should provide information representations expressing knowledge about objects and their associations to other objects, object behaviour, and object interactions. It is crucial that both users and developers have a good understanding of the conceptual schema. Hence, the chosen modelling language must provide a rich set of abstraction features supporting various types of views for data, relationships and constraints representation. It must possess an underlying formal foundation and must be simple enough so that non specialists can also understand and use it. This means that a person should be able to apply it for specifying concepts of the real world according to organisation principles naturally used by humans [SW94]:

16
• **Categorisation**
Humans tend to group things into categories or classes. Members of a category share some common aspects aside from individual aspects. Categorisation provides one means of controlling the level of detail which is required to describe something. It is a form of abstraction. For example, we can talk about lamps in general with one common aspect being that a lamp provides a means of lighting.

In data modelling, we use the notion of abstraction for obtaining categories of data. “Abstraction is the ability to hide detail and concentrate on general, common properties of a set of objects” and to “combine categories into more general categories” [TL82].

• **Grouping**
Grouping provides a means of partitioning a wide variety of things into smaller clumps of things that are relevant to some sub-topic within an overall topic. Grouping may be based on some categorisations but provides no abstraction. For instance, lamps in a warehouse could be grouped by size or be organised on the shelves by product number.

• **Shielding from details**
Description of things are often given in terms of other things, which are in turn described by yet other, more detailed, items. Hence, detail is only introduced when necessary. For example, if we talk about the armrests and legs of a chair, we assume that a listener knows about concepts of armrests and legs. Only if someone does not understand a concept, we explain it in more details in terms of other concepts.

• **Aggregations and ordering**
The concept that something is composed of like things is called an aggregation which also may be sequenced. For example, a book may consist of a table of context, a number of chapters and sections in which case the ordering of these components is important.

In the context of conceptual modelling, we can define aggregation as “treating a collection of concepts as a single concept” [BMW84]. For example, we could represent the concept of a chair as an object being the aggregation of its constituent objects such as chair seat, legs and armrests.

A variety of data models have been proposed to address these issues by “providing increased expressiveness to the modeler and incorporating a richer set of semantics into the database” [PM88]. Such data models usually are categorised as semantic data models. The generic object model OM presented in section 2.2.3 is a good example for a conceptually rich data model fulfilling all of these requirements.

The conceptual schema is produced during the conceptual modelling process. This process includes four main tasks which repeatedly take place. A description of the system is specified in the knowledge acquisition task, whereas a formal refinement
and structuring of the acquired knowledge is carried out in the conceptualisation task. Further, the consistency of the conceptual schema has to be checked in the validation task, and the problem of schema evolution has to be addressed in the evolution management task [RC92]. These tasks can be carried out, for example, by applying the following steps [AGPT00]:

- **Analysis of requirements and knowledge acquisition**
  This step includes activities such as specifying a glossary of terms, analysis of the requirements, elimination of ambiguities and classification of requirements.
  "To classify is to organize, arrange, or group things according to family, class, or category. [Mod92]"

- **Basic step**
  The most relevant concepts are identified and represented in a skeleton schema.

- **Decomposition step**
  The requirements are decomposed referring to the concepts present in the skeleton schema. This should be done whenever appropriate and necessary. For instance, suppose that the relevant concepts in the skeleton schema for the product information system of the example described in section 2.1 turn out to be products, product groups and the companies manufacturing products. For addressing the requirement of supporting product classification, we could decompose this requirement into the requirements “classification according to product type” and “classification according to area of use”, and we then could create subschemas for these requirements referring to the concepts “product” and “product group” specified in the skeleton schema.

- **Iterative step**
  Refinement of the concepts in the schema based on the requirements together with creating new ones which specify not yet represented requirements. This step should be carried out for all schemas until every specification is represented. For example, for fulfilling the requirement of representing the structure of products, we could refine the concept “product” by the two new concepts “composed products” and “assemblies” expressing the fact that some products are composed of others, whereas some are not, such as a screw.

- **Integration step**
  Integration of the various subschemas into a general schema according to the skeleton schema.

- **Quality analysis**
  Verification of the correctness, completeness, minimality and readability of the general schema.

Another important topic for database design is the specification of known transactions in a DBMS-independent way called transaction design [EN94]. There are
three main types of transactions to be taken into consideration: retrieval transactions, update transactions and mixed transactions, i.e. transactions that do some retrieval and some update. To define such transactions, one can identify the input data, output data and the internal functional flow of control which can be described using specification techniques such as TAXIS [MBW80] and GALILEO [ACO85]. Transaction design is just as important as schema design and should therefore be carried out in parallel.

In the next section, we introduce the generic object model OM which is a simple but powerful object data model, and which is the basis of the eXtreme Design meta model. OM is not only suitable for conceptual schema design, but also for the whole application development process. We also show how OM can be used for creating a conceptual schema for the example presented in section 2.1.

2.2.3 The Generic Object Model OM

The generic object model OM [Nor95, Nor93] is a good example for a semantic data model that can be used for all stages of database development [KNW98]. "The goal of semantic data models has been to represent the semantics of the real world as closely as possible. [SPS+95]" This is achieved through a two-level structure of typing and classification in which collections define object roles and associations and types define object representations as shown in figure 2.3.

![Figure 2.3: Classification and types](image)

OM specifies various collection constructs and structural constraints (see figure 2.4), and the representation and processing of associations between entities and objects, respectively, is supported by a special form of binary collection construct (see figure 2.5). Support for role modelling is achieved through subcollection constraints together with constraints over subcollections. Further, there is a general set of operations over collections and associations in terms of the OM algebra. Most semantic data models such as extended entity relationship models put the emphasis on data description and lack an algebra as well as concepts for describing the behavioural aspects making it necessary to translate the conceptual representation into a lower level target model such as the relational model [SPS+95]. This translation step is not necessary using semantic data models which provide concepts for specifying also the behavioural aspects. In OM, this is achieved through the OM algebra together with
the distinction between classification and typing. Another good example for such a data model is ERC+ which extends the entity relationship model with constructs for representing complex objects, with object-oriented features and with Petri net based process descriptions for representing behavioural aspects [SPS+95].

![Diagram](image-url)

**Figure 2.4: Classification structures**

![Diagram](image-url)

**Figure 2.5: Associations**

Specification of objects and associations is done in the stage of *data model mapping* (see section 2.2.4) by the process of *typing*. The type level in OM is defined by the eventual implementation platform and the separation of the classification and type levels enables the stages of design and prototyping to be completed independently of implementation decisions.

Figure 2.6 illustrates the *skeleton schema* for our example introduced in section 2.1. This is the result of the basic step carried out during the conceptual modelling process. The two main collections *Products* and *Companies* are related by the association *manufactured by*. These express the fact that each product sold by the furniture company is manufactured by one or more companies. The three subcollections *Chairs*, *Lamps* and *Beds* make it possible to classify products according to their nature. Subcollections *Household* and *Office* classify products according to where they will be used. The partition constraint demands that each product is classified in exactly one of the three subcollections, whereas the cover constraint requires all products to be classified whether they are designated to be used in a
household or an office or both. The collections are depicted as shaded boxes, with the name of the collection in the unshaded region and the name of its member type in the shaded region. Collection Products thus has a member type product.

![Example Schema in OM model](image)

Figure 2.6: Example Schema in OM model

It would now be necessary to continue with the decomposition, iterative, integration and quality analysis steps together with transaction design before proceeding with data model mapping, but we omit these steps for the sake of simplicity.

### 2.2.4 Data Model Mapping and Database System Implementation

In the data model mapping or logical database design phase, we map the conceptual schema to the data model of the selected DBMS. This is best done in terms of data definition language (DDL) statements of the target DBMS.

For example, in the case of OM we could choose the prototyping system OMS Pro [Wür00] as our target DBMS and map the OM schema to OMS Pro DDL statements:

```sql

type company
(
    name: string;
    locations: string;
);

type product
(
    name: string;
    price: real;
);

type chair subtype of product
```
type lamp subtype of product
(type ...

type bed subtype of product
(type ...

collection Companies: set of company;
collection Products: set of product;
collection Household: set of product;
collection Office: set of product;
collection Chairs: set of chair;
collection Lamps: set of lamp;
collection Beds: set of bed;
collection manufactured_by: set of (product, company);

constraint Household subcollection of Products;
constraint Office subcollection of Products;
constraint Chairs subcollection of Products;
constraint Lamps subcollection of Products;
constraint Beds subcollection of Products;

constraint (Household and Office) cover Products;
constraint (Chairs and Lamps and Beds) partition Products;

constraint manufactured_by association
from Products (1:* to Companies (1:*);

Some DBMSs provide mechanisms for specifying storage structures and access paths such as indexing and clustering for optimising performance which is called the physical database design process [EN94]. These specifications can either be part of the DDL definition or be defined separately by the storage definition language (SDL).

Finally, the database system has to be implemented which comprises two main steps:

- Database creation and population
  An empty database has to be created by DDL and SDL statements. In the case of OMS Pro, it could then be populated with data using the associated data manipulation language (DML).

create object chair1;

dress chair1 as chair values (name = "Chair 1";
price = 34.50;
- Database application development
  A database application typically supports creating, updating, deleting and displaying database objects and is developed according to a software engineering paradigm such as the object-oriented method described in the next section. Mainly, the development process consists of three steps [Kro95]:

  - **Requirements phase**
    User interviews, specification of data flow, of processes operating on data flows, and of application groups.

  - **Design phase**
    Design of application logic, forms and reports.

  - **Implementation phase**
    Building applications using DBMS frameworks, code generation systems and programming environments.

Finally, it is also necessary to implement all database transactions either within database applications or as triggers, stored procedures or methods within the database itself.

In the next section, we describe object-oriented concepts together with the object-oriented design process both of which are aspects of the eXtreme Design approach.
2.3 Object-Oriented Design

"The purpose of design is to bring together different things in order to deliver value. [Bon00]"

In the case of building a software system, 'different things' points to the requirements as a result of the analysis process, and 'bringing together different things in order to deliver value' certainly addresses the issues of focusing on the elements of a system that are most stable: the objects [WEK90]. It is therefore very important to have a good idea about what an object represents and what underlying concepts are to be taken into consideration while designing software. The object-oriented design process basically includes steps such as identifying objects in the application domain, defining concepts or classes representing sets of object properties [PW97], hierarchically organising objects and classes, reusing existing classes, and constructing application frameworks from class libraries [WEK90].

"Frameworks are a practical way to express reusable design. [Joh97b]" Ideally, frameworks provide components which can be easily combined to create a new system. The emphasis is on design reuse which is also achieved by using design patterns. A pattern “presents a template that is general in nature and can be applied to the construction of software over multiple projects. [BIT99]” The essence is to define solutions addressing specific problems. Further, design patterns capture nontrivial relationships and interoperations, but do not have to specify a closed solution [GL98]. In summary, a framework can be defined as a reuse technique based on components and design patterns. In this context, the term component means “a self-contained software construct that has a defined use” and “is built for composition and collaboration with other components” [HS00].

Hence, we can state that the object-oriented design process is heavily based on reusing existing solutions and is closely inter-linked with the analysis phase [KM90]. Whereas, in analysis, the problem domain objects are identified, in design, additional objects are specified which satisfy a specific computer-based solution. Further, the object-oriented design paradigm takes a modelling point of view by developing a model of an application domain specifying objects and object behaviour together with relationships between objects.

In the next sections we first describe some basic concepts of object-oriented technology. We then present the main ideas behind object-oriented design and show how a particular design methodology can be applied to our example.

2.3.1 Basic Object-Oriented Concepts

Object-oriented concepts were introduced in Simula which has its origins in simulation and was designed as both a description language and a programming language
The key concept in Simula is the classification of objects which share certain properties into classes and subclasses. Object-oriented programming achieved wide recognition in terms of general programming and system paradigm through the programming language Smalltalk [Kay93]. In a Smalltalk environment, an object represents a component of the Smalltalk software system. For instance, numbers, strings, file directories, programs and text editors are treated as objects which can receive a request in terms of a message specifying which operations should be carried out [GR89]. This is also called communication by message passing, thus encapsulating or hiding the object’s realisation, i.e. its data structures as well as the operation implementations [Nie89]. With this definition, we could state that any programming language that support encapsulation is object-oriented. Is Ada or Oberon an object-oriented language?

Classifying programming languages can be done according to various criteria such as whether a language is data-oriented or process-oriented, whether it supports strong typing and abstract data types, whether a language has been designed for specific application domains only, and so on. A very convenient way of looking at programming languages has been proposed in [Rec90]: "... contemporary programming languages are compared and classified according to their ‘thought model’, i.e., the distinguishing features of a family of programming languages that influence the way programmers think while writing or reading programs in this language family". This is also called the paradigm-oriented viewpoint for classifying programming languages.

Using this viewpoint for distinguishing classes of programming languages leads to the problem of finding appropriate classification characteristics. For instance, we may choose the characteristics imperative and non-imperative languages. Or, we may distinguish value-oriented and object-oriented languages, i.e. whether we take a "mathematical view" or "result-oriented view" of programming.

Non-imperative languages replace the state transformation view by other concepts. Functional languages express computations as mathematical functions which are timeless and containerless, hence a side-effect free concept for computing unknown values from known ones. Logical languages introduce the idea of logical deduction as a form of computation which is expressed in terms of logical predicates. A problem is defined as a set of rules (declarative programming) and during evaluation of predicates, logical variables are bound to values and unified with expressions.

Central to imperative programming languages is the idea of variables as containers of values and the transformation of these values. A variable is referenced by a name in the program source. During program execution it has a fixed address and contains a value which may be changed by assignment. Algorithm-oriented languages are based on the concept of transforming input data into output data focusing on computation of data whereas in object-oriented languages data and algorithms are packaged as objects using concepts such as data encapsulation and abstract data types. An abstract data type (ADT) is an encapsulation of a data
structure along with a collection of related operations on this encapsulated structure shielding details of its implementation. An abstract data type represents sets or classes of similar objects. A language supporting ADTs must satisfy the following [PCKW89]:

- **Object classes**  
  An entity or object must be an element of an ADT which is the object’s class.

- **Information Hiding**  
  An object can be accessed and modified only through the external interface methods defined for its ADT.

- **Completeness**  
  The operations of an ADT correctly and fully define the behaviour of the ADT as intended by the developer.

The object-oriented paradigm extends the idea of abstract data types by concepts such as object types, type hierarchies and inheritance. Hence, it is based on encapsulating data and related code into a single unit [SKS97], and is characterised by the following concepts [GLZ99, Set96]:

- **Object**: a collection of data and operations

- **Type or Class**: specifies a set of operationally similar objects sharing the same implementation (structure and operations)

- **Extent**: groups the set of created objects of a given type or class

- **Inheritance**: a class may inherit properties from superclasses and may have its properties inherited by subclasses

- **Instance**: a technical term for an object of a class

- **Method**: an object property implementing an operation in a procedure body

- **Message**: a procedure call; request to execute a method

An object has associated with it a set of variables containing the data for the object, a set of messages to which the object responds, and a set of methods each of which is a body of code to implement a message [SKS97].

Furthermore, many object-based environments support inheritance, strong typing and polymorphism. Inheritance makes it possible that a class may inherit properties such as variables and methods from superclasses, whereas strong typing means that "type compatibility of all expressions representing values can be determined from the static program representation at compile time. [Weg87]"
Polymorphism means “the ability to take more than one form” [KM90], i.e. a polymorphic reference can refer to instances of more than one class over time or, in other words, “the implementation of the operation depends on the object to which it is applied” [Lar95]. Two major kinds of polymorphism can be distinguished: universal and ad-hoc polymorphism [CW85]. Universal or “true” polymorphism includes parametric polymorphism obtained whenever an operation works uniformly on a range of types which usually exhibit some common structure, and inclusion polymorphism for modelling subtypes and inheritance which means that “an object can be viewed as belonging to many different classes which need not be disjoint, i.e. there may be inclusion of classes” [CW85]. Ad-hoc or “apparent” polymorphism addressed the situation when “a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type” [CW85]. This is achieved either by using the same name for different methods called overloading, or by the semantic operation coercion needed to convert a method parameter to the expected type.

Other desirable features of object-based environments include, among others, the following [Lar95]:

- **Object identity**
  Each object must have a unique identifier.

- **Complex objects**
  Basic objects can be combined into complex objects.

- **Extensibility**
  New classes can be added to the system-supplied ones.

- **Persistence**
  Object states must remain available after the application termination.

- **Concurrency**
  An object state can be accessed by multiple objects at the same time.

Most object-oriented languages and environments such as Smalltalk, C++ and Java support these features with the exception of persistence. Concepts of persistence are usually not part of such languages. Persistence is in this case achieved through mechanisms such as serialising objects into files or using interfaces to database management systems.

### 2.3.2 The Object-Oriented Design Process

Designing software is about value creation. There are values which the software design must deliver according to the requirements, and there are values which make a design acceptable or practical [Bon00]. What are the specific benefits of a design?
Does a design address the significant key values? Is it simple enough? What about the compatibility with other software components? Is it "doable" within a certain time and cost frame?

These and other questions about the value of a design are very important to guarantee software quality. Aspects of software quality are, for example, correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, verifiability, integrity, and ease of use [Mey88].

Good design techniques not only lead the design process towards these quality aspects, but also encourage and facilitate design thinking which is concerned with the "flow" characteristics of the design process such as "how to find new ideas and concepts", "how to modify or change existing results", "how to improve the value of a design", and "how to combine existing designs" [Bon00].

To achieve this, a design methodology should guide the various activities in all phases during the design process. Further, it should define the foundation of concepts and techniques, should specify a method for applying the concepts, should allow the method to scale up, and should provide supporting tools [JCJO92].

The methodological framework for object-oriented systems development proposed in [HSE90] is composed of seven steps:

- **Object-oriented system requirements specification**
  At this stage a high-level analysis of the system in terms of objects and their services is performed.

- **Identification of objects and services**
  The activities of step one and two are similar to the five activities proposed in [CY91a]: finding classes and objects, identifying structures, identifying subjects, defining attributes and services. The result of these activities is usually presented in several diagrams.

- **Establishing interactions between objects in terms of services required and services rendered**
  At this stage, the information flow between objects is described in terms of diagrams.

- **Merging the analysis stage into the design stage**
  Details of objects are specified taking into account the identification of reusable design components.

- **Bottom-up concerns**
  In parallel to the previous step, bottom-up concerns deals with the construction of objects from library or framework objects.

- **Specification of hierarchical inheritance relationships**
  Reevaluation of the total set of classes iteratively requires a restructuring of the class inheritance tree.
• *Aggregation and/or generalisation of classes*
  
  Further, the resulting class tree of the previous step may require reconsideration of the structures and diagrams describing the whole system.

The last few steps are covered also by the design method proposed in [CY91b]. Here the results of analysis are the basis for the design of the four main system components *Problem Domain Component, Human Interaction Component, Task Management Component* and *Data Management Component*. Also the approach presented in [Boo91] emphasises an incremental and iterative process of tasks such as identifying classes and objects, identifying the semantics of classes and objects, identifying the relationships among classes and objects, and implementing classes and objects.

All the experiences made during the last few years with the various object-oriented development methods led to the idea of unifying and standardising the most popular approaches, resulting in the *unified modeling language* (UML) adopted by the standardisation body of the *Object Management Group* (OMG) [PS99, Lau01].

UML is very expressive and can therefore be used with most of the object-oriented development methodologies or processes such as *Objectory* (the Object Factory for Software Development) [JCJO92]. The framework of Objectory is the design technique *design with building blocks* which views a system as a number of connected blocks, each of which represents a system service. After having specified all required building blocks, they are designed using a top-down approach [JCJO92]. According to the UML concept, such a process should be *use case driven*, i.e. the behaviour of a system is documented from the point of view of anything external to the system and interacting with it. Use case modelling facilitates the capturing of requirements, planning iterations of development and validating systems. Further, use case diagrams should be easy to understand [PS99].

Describing a design of a software system is best done using a diagram-based notation since that supports the way people think naturally about systems. Further, since it is not useful to show all aspects of a design in a single diagram, there must be a way to express specific models on several axis [PS99]:

- The *use case* model describes the required system from the users' points of view.
- A *static* model describes the elements of the system and their relationships.
- A *dynamic* model describes the behaviour of the system over time.

We further can distinguish the following views as proposed in [Kru95]:

- *Logical* view
  
  This view should help to check whether the functional requirements are met by looking at which parts belong notionally together.
• **Process view**
  Here, the focus is on the non-functional requirements such as performance and availability.

• **Development view**
  This view helps to manage a project by looking at issues such as which parts should be developed by whom, or what parts can be reused.

• **Physical view**
  This is a more concrete view than the process view analysing which non-functional requirements are met by what kind of resource (e.g. a computer).

A modelling language should make it possible to depict the same model by various diagrams, and should take care of the consistency of models and diagrams. "The diagram is not the design: the diagram is a representation of (part of) a model of the design, which captures an aspect of the design in a form which can be discussed. [PS99]"

The modelling language **UML** supports the development process by defining *model elements* representing fundamental modelling concepts and semantics, by specifying a *notation* for visual rendering of model elements, and by providing *guidelines* outlining idioms of usage within the trade. In terms of the views of a model, UML defines graphical diagrams for use case, class, behaviour and implementation models [Obj].

Figure 2.7 gives an example of a UML class diagram for our example. The *generalisation* relationship constructs have been used to model the fact that a product can either be a chair, a lamp or a bed, and the *association* construct denotes the link between a *product* and a *company* object.

Since there are usually no constructs supporting role modelling in object-oriented languages, it is not possible to directly model the fact that a product is one for a household, for an office, or for both. There are many possibilities to express roles, for example, by defining subclasses and associations. In our example, role modelling has been achieved by the association between a *product* and *product kind* object which can be specialised into a *Household* or *Office* object.
Figure 2.7: UML class diagram example
2.4 The eXtreme Design Approach

The goals of the design process as part of software engineering and information engineering is to transform the results from requirements analysis into specifications suitable as a basis for the implementation phase. This is an important and complex task and demands a clear understanding of all involved components of a software system as well as for good design methods at hand. And since most software applications and components have to manage persistent data which is usually stored in a database or files, it is important that the chosen design method also supports data modelling.

Object-oriented design methods focus mainly on the structure of systems and their components as well as on the interaction between the various components and objects together with their behaviour. Database design methods emphasise the semantic aspects of application objects together with their relationships and integrity constraints. Both approaches use similar concepts such as categorisation, abstraction and aggregation making it possible to specify software systems and components which are exchangeable and extensible. Further, the use of design patterns [GHJV95] for software construction and data model patterns for data modelling such as the ones proposed in [Hay96], increase the quality of software systems in terms of flexibility and extensibility.

The eXtreme Design (XD) approach can be applied in conjunction with existing object-oriented design methods and is tailored especially to be used for designing those parts of a software system or component which manage persistent data, for example, using a database management system. Hence, the XD meta model and its algebra are based on concepts from conceptual as well as object-oriented modelling. From conceptual modelling, XD incorporates the idea of role modelling, i.e. the task of categorising and grouping objects and building relationships between object groups, whereas from object-oriented modelling XD supports the concepts of object identity, object aggregation, information hiding, extensibility and a special form of typing.

Since in the XD meta model the classifications structures, relationships and object properties such as attributes and methods are not fixed, they can be changed and extended using the XD algebra without having to change, for instance, schema information or class definitions. This improves the flexibility of the design process especially in the case where it is not clear whether, for example, the chosen classification and object structures represent the application domain in an optimal way in terms of future requirements.

For instance, in the example shown in figure 2.7 the design specifies that products can be classified as being household or office furniture. Implementing this design specification results, in the case of an object-oriented programming environment, in class descriptions for each concept defining attributes and methods.
Suppose that after using the system for a while we recognise that we need more classification concepts as well as need to extend the existing ones by new properties. This makes it necessary to add new class descriptions as well as changing existing ones which must be carried out by a developer.

With the XD approach we can implement the classification concepts in such a way that it is possible to add new concepts or change existing ones without having to change class descriptions. This also could be done by a user of the system. So, for example, a salesman could add extra attributes and create additional classification structures without causing changes in the implementation. Further, we can analyse the changes made by users which might have impact on the design. If many users, for instance, add the same classification concepts, this might indicate that we should change the design and implementation and add these concepts. In this way, also users can be involved in the design process of a software component.

This is an addition to existing design methods and models with which design specifications for classification structures, relationships and object properties are expressed using a modelling language such as UML [Obj]. These specifications are then used as input for implementation. New requirements most often cause changes in the design specifications as well as implementation, whereas in the case of the XD meta model we can use the XD algebra which specifies operations on the meta model itself for making changes in design and implementation of a specific software component.

For example, the design methods proposed in [Boo91, RBP+91, CY91b] offer a rich set of concepts for specifying the various static and dynamic aspects of a software system. There is the notion of a class which defines the properties of objects such as attributes and methods. With aggregation and inheritance constructs we can express complex class and object compositions, and there are also association constructs for defining object relationships. With these methods we have to change the design specifications and implementation if we want, for example, to introduce new object properties or additional relationships.

Other methods focus especially on role modelling the essence of which is the fact that we always consider objects in specific contexts which specify their roles and properties. Central to the method called The Object Oriented Role Analysis Methodology (OORAM) [RWL96], for example, is the concept of role which is the basic object abstraction. In OORAM, the role model describes the structure of cooperating objects along with their properties. Each role is characterised by attributes and actions and it is possible that an object is associated to more than one role.

Another example is the generic object model OM described in section 2.2.3 which supports role modelling through the concept of collections and associations. The member type of a collection determines the properties of its members. Additionally, OM provides an algebra which can be used for evaluating operations on collections and associations.

Also EROOS which is an entity-relationship based object-oriented development
method [ER095, BLSR92] uses the concept of class as a means of categorising objects of the application domain. Classes in EROOS define a set of objects with similar characteristics as in the case of entity types in Entity-Relationship modelling. Classes can be dressed with all kinds of characteristics such as actions, relations, attributes and inheritance structures. For example, the following example specifies the class ACCOUNT with two actions which are invoked when creating, respectively, destroying an object:

```plaintext
class ACCOUNT is
    constructors
        open
    destructors
        close
end ACCOUNT
```

We can now, for example, dress this class with the attribute BALANCE. We first define the attribute by the following statements:

```plaintext
attribute BALANCE is
    decorates ACCOUNT
    representation AMOUNT
end BALANCE
```

Then, we relate this attribute to the ACCOUNT class and define actions for this attribute by extending the class definition:

```plaintext
class ACCOUNT
    decorated by BALANCE is
        ...
    mutators
        deposit(sum: AMOUNT)
            effect: new->BALANCE = old->BALANCE + sum
        withdraw(sum: AMOUNT)
            effect: new->BALANCE = old->BALANCE - sum
        ...
end ACCOUNT
```

Design approaches can further be distinguished based on whether the approach is model-driven or method-driven. Method-driven approaches define a fixed sequence of steps to follow, whereas model-driven techniques provide a set of fundamental concepts with which to model a particular system. The Business Object Notation BON [PO99] can be regarded as an example for a method-driven approach. BON defines a recommended process which is representative of many earlier object-oriented
methods. Basic steps in BON are "delineate the system border", "list candidate classes", "select classes and groups", "define classes", "sketch system behaviour", "define public features" and "refine system". The fundamental specification construct in BON is the class. A BON class represents both a module and a type, i.e. it is a possibly partial implementation of an abstract data type. A class is specified by its name, an optional class invariant, and a collection of features which are either queries or commands. A query returns a value and does not change the system state, whereas a command does change system state. There is no separate notion of attribute which is considered as a query returning the value of some hidden state information. Specifications in BON can be defined using a textual or graphical notation which are entirely equivalent. The following class definition is an example for a textual specification:

class CITIZEN feature
    name, sex, age: VALUE
    spouse: CITIZEN
    children, parents: SET[CITIZEN]

    single: BOOLEAN
        ensure Result<->(spouse=Void) end

    divorce
        require not single
        ensure single and (old spouse).single end
end

invariant
...
end -- CITIZEN

Another example for a method-driven approach is the Fusion method [CBA94]. Fusion comprises a complete method providing a direct route from requirements definition through to programming language implementation. The result of the design process is specified by object interaction graphs, visibility graphs, class descriptions and inheritance graphs. The interaction graphs show how functionality is distributed across the objects in a system. The visibility graph describes the structure of an object-oriented system, whereas class descriptions specify the internal state and external interface of a class which includes descriptions for methods, attributes and inheritance information. The inheritance graph depicts the inheritance structures of classes. Although Fusion defines a comprehensive set of notations for specifying analysis and design results, the method has been extended to introduce UML as its notation.

Some approaches have been specifically developed for a certain application domain. For example, TROLL [JSHS96] is a language particularly suited to information systems development. A basic concept in TROLL is the object which is specified as
possible sequences of events representing basic state transitions, and attributes as observable properties changed by events. Thus, objects are regarded as observable processes. TROLL knows the concept of templates which specify, for example, data types, attributes, events and constraints and can be reused by giving them a name. A class in TROLL is a container for instances of the corresponding class type which consists of a template and a data type for the possible identifiers of instances. The following definition is an example for a class type specification:

```plaintext
object class Account
    identification
        data types nat;
        No: nat;
    template BankAccount
end object class Account;
```

Modern approaches for software development should not be restricted to software construction methodologies [HS99]. They must take into account also the cultural aspects of people and organisations as well as the available technology and tools. Third generation methods address these issues by providing a process framework “which defines the overall architecture of the process without constraining the user to specific detail” [HS99]. An example for such an approach is OPEN: Public-domain Object-oriented Process, Environment and Notation [GHSY97, HS97]. OPEN is a flexible framework which can be used for creating an individually tailored method. OPEN focuses on the interaction between producers, the operations performed by producers called work units and the work products. A producer can be anything that produces versions of work products. A work product is a significant thing of value developed during the performance of work units.

Work products in OPEN are described using languages such as natural languages, modelling languages and implementation languages. Although UML [Obj, HSI00] can be used in many cases for expressing OPEN models, it is recommended to apply the Open Modeling Language OML [FHSG98] for specifying OPEN models. OML is based on object-oriented concepts such as objects, classes and types, and emphasises responsibility-driven design, i.e. specifying first responsibilities of objects, classes and types rather than early identification of properties.

In summary, we can state that modern software engineering methods such as OPEN and models such as UML are very suitable for the software development process of a variety of application domains. Further, there is a trend towards specifying the various models in terms of meta models which describe the concepts of a model on a higher level of abstraction making a model more extensible and adaptable for future requirements. For instance, the meta models for UML and OML “offer a full specification needed for creating a modelling language appropriate for OOAD” [HS98].
An important aspect of most object-oriented approaches for software construction is the fact that these approaches provide techniques and notations for categorising application domain objects, for defining relationships among objects and for specifying object properties. Introducing new requirements to a software system often make it necessary to change and adapt the corresponding design specifications together with their implementations.

Hence, by applying the eXtreme Design approach, we can extend and improve an existing design approach because concepts of the application domain are defined on a meta level through the XD meta model and its algebra. Hence, this makes it possible, for example, to change classification structures, relationships and object properties by evaluating the appropriate XD algebra operations. Thus, no changes of design specifications are required, but it is possible to derive design specifications by analysing the content of an XD database using the XD meta model and its algebra.

A meta model can be defined as a model which can be used for specifying constructs and concepts of other object models. Hence, we define the task of metamodeling as “simply creating a model at a higher level of abstraction than the thing being modelled” [HSB98]. For example, the concept of a type or class can be represented as a meta object using the XD meta model.

A meta model typically comprises the same elements that define any data model: data constructs, links and constraints [PR95]. Further, meta models are tailored to meet a specific purpose. For instance, the meta model proposed in [PR95] is especially useful for capturing and describing the semantics of a variety of data models, whereas Telos [MBJK90] is an example for a meta model suitable for the development of information systems.

The XD meta model and its algebra can be considered as a general purpose meta model combining conceptual modelling and object-oriented concepts on a meta level which facilitates the development of highly adaptable software components and information systems. We present the XD meta model and its algebra in more detail in chapter 3. In chapter 4 we describe two frameworks which can be used for applying the XD approach.

Basically, the XD approach involves the following steps:

- **Analysis of the problem domain and system requirements**

- **Transformation of analysis results into design specifications**
  This step can be carried out by using an existing object-oriented method or model such as BON [PO99], UML [Obj] or OM [Nor95, Nor93].

- **Identification of those parts of design to which the XD approach should be applied.**
  Good candidates are parts in which role modelling play a central role, which
must be highly adaptable to future requirements, and which manage persistent data.

- **Implementation of design specifications**
  All XD parts are implemented using a framework which supports the XD meta model and its algebra together with an implementation environment such as Java. All other parts can be implemented using the various libraries and frameworks of the target implementation environment.

- **Applying XD during system operation**
  Changes and extensions to those parts of the system which have been implemented using the XD meta model can be carried out by user’s and developers while the system is running.

- **Analysing changes**
  To improve system design, we can analyse the changes made in the XD parts. This might provide information for changes in the design specifications. For instance, if a lot of users introduce similar new classification structures or object properties, this might indicate that the corresponding design specifications should be adapted.

- **Redesign**
  Incorporating new requirements sometimes requires changes to parts of a system which cause in turn major changes to other parts. In this case, it is good practice to go through a redesign process for those parts. Further, by building prototypes for specific parts, we can gain better information about how to specify the new design. An example of using the XD approach for prototyping is given in chapter 5.

To cope with the increasing complexity of software systems, component-based software engineering (CBSE) techniques have emerged as a key for managing the development of complex software systems.

“A software component is a non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces [BW98].”

Component-based software development “shifts the focus from new software development to the integration of existing components to perform new tasks” [Hop00]. The ability to *reuse* existing components to create more complex systems as well as to *evolve* a system by changing its components with little or no effect on the remaining components are crucial aspects of CBSE. Hence, CBSE “must address both the development of reusable components and the development of applications using reusable components” [Kan99].

We have gained experience using the XD approach for the development of reusable components with the implementation of the object management framework *OMS Java*
which is presented in chapter 6. This system is highly adaptable so that, for instance, it is possible to extend the core object model $OM$ with new semantics such as temporal information. This was achieved on the one hand because of the extensibility of the object model $OM$ itself, and on the other hand by using various design patterns for system design.

So, various aspects of the system – such as the algebra, constraint and languages – can be extended by exchanging and extending the corresponding components without having to change the core system components such as the storage manager. But, for some changes, the core system also needs to be adapted or redesigned. For instance, in the case of introducing new data structures for bulk information processing, the storage manager needs to be changed. We implemented these data structures using the XD meta model and its algebra showing that the XD approach can also be applied for developing highly adaptable components. This is described in section 6.3.

Another form of component-oriented software construction is presented in chapter 7. Here we show how the XD approach facilitates the development of components supporting rapid information modelling which can be regarded as central for creating user specified information spaces.
Chapter 3

eXtreme Design (XD)

The eXtreme Design (XD) approach supplements existing software engineering techniques in the area of prototyping, component modelling and information modelling. XD facilitates the development of highly extensible and adaptable software components and information systems by combining aspects of conceptual modelling and concepts of object-oriented software construction on a high level of abstraction.

XD is based on the XD meta model and its algebra which is detailed in the following sections. From conceptual modelling, the XD meta model supports the idea of expressing the semantics of application objects by making it possible to specify their roles and relationships. From the object-oriented paradigm, the XD meta model is based on the concept of defining application objects and their interactions in terms of object properties, i.e. characteristics of objects are represented as attributes and their values, and their behaviour as methods.

The XD meta model specifies these concepts on a meta level which means that they are represented themselves by the XD meta model. For example, role modelling is usually supported by conceptual models through the notion of entities, collections of entities and associations between them. So, for instance, for developing a database application, we typically define the roles of application objects in diagrams which then are mapped to schemas and implemented on a specific implementation platform such as a database management system.

This implies that, whenever we change roles or introduce new ones, we also have to change the schema definition and corresponding application components. Since the role modelling concepts are specified in the XD meta model on a meta level, it is not necessary to create a schema for representing roles of application objects. Thus, changes and extensions to a specific conceptual model can be carried out without having to change schema information.

In the case of object properties, which are also represented on a meta level in the XD meta model, we do not need to define them in terms of type definitions which
makes it possible to introduce new properties or change existing ones without the necessity of changing type information.

Hence, we claim that by combining conceptual modelling aspects and object-oriented concepts on a meta level, the eXtreme Design approach is especially suitable for developing extensible and adaptable software components and information systems.

Furthermore, because of the fact that the XD meta model can be used for specifying and implementing concepts of the application domain on a very high level of abstraction, the eXtreme Design approach facilitates rapid information modelling (RIM).

We define rapid information modelling as the activity of classifying and structuring information about a specific domain. The word rapid claims that it should be possible to carry out this activity in such a way that it is not necessary to define schema and type definitions for concepts of the application domain.

In this context, “Concepts are used to interpret our current experience by classifying it as being of a particular kind, and hence relating it to prior knowledge. [WK99]” Knowledge expresses the fact that data about a subject is claimed to be true, i.e. correct. “Data corresponds to stating something (be it true or not) while information corresponds to speech acts which convey intentions. [HKL95]” Finally – to close the circle – we understand information as denoting the process of gaining knowledge and eliminating uncertainty.

In the next section we introduce the meta model on which the eXtreme Design approach is based. Section 3.2 presents the XD algebra which is necessary to support the XD approach, and chapter 4 outlines the XD approach by means of an example.

### 3.1 The eXtreme Design Meta Model

The XD approach is based on the XD meta model shown in figure 3.1 as well as the XD algebra described in the next section.

The meta model itself is defined in terms of the OM model making it possible to easily customise the XD meta model by adding new collections and associations. It is also possible, for example, to easily change the collections and associations in the XD meta model as being bags instead of sets.

The XD meta model is an extended version of the model presented in [KN98, KN97], and consists of three main parts supporting the activities of role modelling, of specifying object hierarchies and relationships, and of defining object properties such as attributes and methods.

The set of these object properties is not fixed, thereby enabling properties to be added and removed at runtime. Because of this characteristic, an object in our
Figure 3.1: The eXtreme Design Meta Model
model can be regarded as representing an object in the context of the object-oriented paradigm on a higher level of abstraction. We call it therefore a meta object. A meta object does have a unique object identifier, and its state can be altered by sending messages. The state of an object corresponds to all data it encapsulates at a specific time, whereas the behaviour of an object denotes the way an object acts and reacts to state changes and messages [PS99].

In the next section, we introduce those concepts of the generic object model OM which are necessary for specifying the XD meta model and its algebra.

3.1.1 The Generic Object Model OM

Since the XD meta model and its algebra is defined in terms of the OM model [Nor95, Nor93], we describe in this section those concepts of OM which are needed for specifying the XD meta model.

In OM, entities of the application domain are represented as objects. An object is determined by its unique object identifier generated at creation time of the object. A basic concept in OM is the notion of collection. A collection has an associated member type and is also an object. There are two kinds of collections: unary collections represent object roles and have unary values as elements, whereas binary collections or associations represent relationships between objects and have pair values as elements. Collection Objects in figure 3.1 is an example for a unary collection, and the association refer to between collection Objects and ValueEntities is an example for a binary collection.

Let OBJID be the domain of all object identifiers and COLL ⊆ OBJID be the set of all collections. It follows that for a given collection C ∈ COLL ⇒ C ∈ OBJID. The extension of C, i.e. the group of member objects, is denoted as ext(C). The collections used for defining the XD meta model are set collections, i.e. the elements of the collections are distinct and unordered. Further, a collection represents a grouping of values of a common type called the member type of a collection. Hence, for a given collection C and a given type T it follows that if member_type(C) = T then x ∈ ext(C) must be an instance of type T, written as x : T. Further, assume that there exists a membership relation ∈_{set} which is a criterion for determining whether any given value is a member of a given set, then we can define the membership relation ∈ on set collections in terms of the relation ∈_{set}. Let C be a collection with membertype(C) = T and x : T, then

\[ x ∈ C ⇔ x ∈_{set} ext(C) \]  \hspace{1cm} (3.1)

Let BINCOLL ⊆ COLL be the set of all associations. If C ∈ BINCOLL, there exists C_1, C_2 ∈ COLL such that source(C) = C_1 and target(C) = C_2 with
\[ ext(C) = \{(x, y) \mid x \in ext(C_1) \land y \in ext(C_2)\} \]  

There is a conceptual dependency between the collection ObjectGroups and Objects in the XD meta model specified by a isa relationship which is defined for two collections \( C_1 \) and \( C_2 \) as follows:

\[ isa \quad C_1 \preceq C_2 \implies ext(C_1) \subseteq ext(C_2) \]  

Thus, \( C_1 \preceq C_2 \) means that if \( x \in C_1 \) then also \( x \in C_2 \).

OM defines a set of algebra operations which can be applied to collections and associations. A full list of these operations is defined in [Nor93]. For specifying the XD meta model and its algebra, we need the following subset of OM algebra operations:

- **Selection**

\[ \% : (coll[t], t \rightarrow bool) \rightarrow coll[t] \]  

The selection operation on a given collection \( C \) returns a collection containing those elements of \( C \) which satisfy a predicate \( P \) as represented by a boolean function that maps an element of \( C \) to the Boolean value \text{true} or \text{false}. Let \( C_1 \) be a collection with \( ext(C_1) = S_1 \). If \( C = C_1\%P \) then \( ext(C) = S_1\%setP \)

\[ S\%setP = \{x \mid x \in set S \land P(x) = \text{true}\} \]

- **Cardinality**

\[ \# : coll[t] \rightarrow int \]  

The cardinality operation returns the number of elements of a collection.

- **Domain**

\[ dom : coll[(t_1, t_2)] \rightarrow coll[t_1] \]  

The domain operator returns a collection of all the values that appear as the first element of a pair belonging to a given binary collection \( C \). Let \( C \) be a binary collection with \( ext(C) = S \) then \( ext(dom C) = dom_{set} S \) where

\[ dom_{set} S = \{x \mid \exists y.(x, y) \in set S\} \]
• **Range**

\[
\text{rng} : \text{coll}[(t_1, t_2)] \rightarrow \text{coll}[t_2]
\]  

The range operator returns a collection of all the values that appear as the second element of a pair belonging to a given binary collection \(C\). Let \(C\) be a binary collection with \(\text{ext}(C) = S\) then \(\text{ext}(\text{rng} \ C) = \text{rng}_{set} \ S\) where

\[
\text{rng}_{set} \ S = \{y \mid \exists x. (x, y) \in_{set} S\}
\]

• **Domain Restriction**

\[
dr : (\text{coll}[(t_1, t_2)], \text{coll}[t_3]) \rightarrow \text{coll}[(t_1, t_2)]
\]  

The domain restriction operator takes a binary collection \(C\) and a collection \(C'\) and returns a binary collection containing all those pairs of \(C\) with first value in \(C'\). Let \(C\) and \(C'\) be collections with \(C\) a binary collection and \(\text{ext}(C) = S\). Then \(\text{ext}(C \ dr \ C') = S \ dr_{set} \ C'\) where

\[
S \ dr_{set} \ C' = \{(x, y) \mid (x, y) \in_{set} S \land x \in C'\}
\]

• **Range Restriction**

\[
rr : (\text{coll}[(t_1, t_2)], \text{coll}[t_3]) \rightarrow \text{coll}[(t_1, t_2)]
\]  

The range restriction operator takes a binary collection \(C\) and a collection \(C'\) and returns a binary collection containing all those pairs of \(C\) with second value in \(C'\). Let \(C\) and \(C'\) be collections with \(C\) a binary collection and \(\text{ext}(C) = S\). Then \(\text{ext}(C \ rr \ C') = S \ rr_{set} \ C'\) where

\[
S \ rr_{set} \ C' = \{(x, y) \mid (x, y) \in_{set} S \land y \in C'\}
\]

• **Inverse**

\[
\text{inv} : \text{coll}[(t_1, t_2)] \rightarrow \text{coll}[(t_2, t_1)]
\]  

The inverse operator returns a binary collection of pairs formed by swapping the elements of pairs in a given binary collection. Let \(C\) be a binary collection with \(\text{ext}(C) = S\). Then \(\text{ext}(\text{inv} \ C) = \text{inv}_{set} \ S\) where

\[
\text{inv}_{set} \ S = \{(y, x) \mid (x, y) \in_{set} S\}
\]

In the next sections, we detail the three parts of the meta model and point out how it can be used to support the various activities.
3.1.2 Classifying Objects

Classifying objects involves forming concepts or categories to abstract common characteristics of objects, and assigning new objects in these categories [PW97]. An object of the application domain represents a cognitive symbol designating the perceived existence of a thing in the world. The task of classifying objects is comparable to the task of specifying the roles objects play in the application domain, called role modelling. This should not be mixed up with the concept of classes in the object-oriented paradigm. There, classes “specify a context of interpretation for messages received by an object” [Per90], and hence specify the properties of objects such as attributes and methods.

Figure 3.2 shows the part of the XD meta model supporting role modelling.

![Figure 3.2: Classification and Role Modelling](image)

Objects and object groups are modelled as entities having a unique object identifier. Objects represent entities in the application domain, whereas object groups represent categories and concepts. Objects can be classified by being members of object groups denoted by the association are member of between collections Objects and Object Groups. Further, since object groups are a specialisation of objects, object groups can be members of other groups, making it possible to define complex classification hierarchies. For instance, in the example schema shown in figure 2.6, there is a classification hierarchy between the collections Chairs, Lamps and Beds and the collection Products, in this case a subcollection relationship. These hierarchy is represented as are member of associations between object groups in the XD meta model.

If we want to stress that only object groups can be members of another group, as is the case, for example, in specifying subcollection relationships, then we use the are member of association between object groups for creating such relationships.

The subcollection relationship between Object Groups and Objects also implies that an object group has the same characteristics as an object, i.e. can have object
properties and can be related to other objects. In addition, every object can become an object group. An object gains the role of an object group by creating an are_member_of association to it.

In the following example, we show how to create the classification structure for the example presented in section 2.1. First, we create object groups using the data manipulation language (DML) of the XD system:

create object Products;
create object Household;
create object Office;
create object Chairs;
create object Lamps;
create object Beds;

Note that there is only one DML statement for creating objects as well as object groups since object groups are just a specialisation of objects, i.e. every object can become an object group by gaining the object group role whenever an are_member_of association is created.

Next, we create all product objects and insert them into product groups by corresponding DML statements:

create object product1;
create object product2;
create object product3;
create object product4;

insert into Products: [product1, product2, product3, product4];
insert into Chairs: [product2, product3];
insert into Lamps: [product4];
insert into Beds: [product1];
insert into Household: [product2, product3, product4];
insert into Office: [product1, product3];

As can be seen by this example, it is not necessary to define schema information for objects and object groups using the XD meta model. Hence, it is very easy to introduce, for example, a new object group and populate it with members by just creating a new meta object:

create object Tables;

And then creating corresponding product objects and inserting them into the newly created object group:
create object table1;
create object table2;
insert into Table: [table1, table2];

If we use a system supporting a conceptual model such as OM, we have to change
schema information for achieving the same result. In the schema, we must create a
new type `table` together with the collection `Tables` using the corresponding data
definition language:

type table
  (  
    name: string;
    price: real;
  );

collection Tables: set of table;

To populate the database, we can use the data manipulation language:

create object table1;
dress table1 as table values (  
  name = "Table 1";
  price = 78.00;
);  

insert table1 into collection Tables;

And, if we use a system such as *OMS Java* which supports the OM model in the
Java environment (see chapter 6), we also have to provide a Java class definition for
type `table`:

```java
public class Table extends OMSInstance {  
  private String name;
  private double price;

  public synchronized void setName(String name) {  
    this.name = name;
  }

  public synchronized String getName() {  
    return name;
  }

  ...
}
```
As we see by this example, using the XD meta model for classifying application objects not only makes it possible to build complex classification hierarchies, but also simplifies the evolution of an existing conceptual schema.

In chapter 4, we present two possibilities of how to implement a system supporting the XD meta model and its algebra. Such a system must provide in its application programming interface (API) a set of operations which are needed to manage meta objects and which are the basis for the XD algebra. For supporting classification as described in this section, such a system must implement at least the operations listed in table 3.1 and table 3.2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE OBJECT</td>
<td>Creates a new object</td>
</tr>
<tr>
<td>MODIFY OBJECT</td>
<td>Modifies the name of an object</td>
</tr>
<tr>
<td>DELETE OBJECT</td>
<td>Deletes an object</td>
</tr>
<tr>
<td>RETRIEVE OBJECT</td>
<td>Retrieves an object by its OID or name</td>
</tr>
</tbody>
</table>

Table 3.1: XD API Operations for Managing XD Meta Objects

<table>
<thead>
<tr>
<th>Operation</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO GROUP</td>
<td>Inserts an object into a group</td>
</tr>
<tr>
<td>REMOVE FROM GROUP</td>
<td>Removes an object from a group</td>
</tr>
<tr>
<td>MEMBERS</td>
<td>Returns all members of a group</td>
</tr>
<tr>
<td>IS_MEMBER</td>
<td>Tests whether an object is member of a group</td>
</tr>
</tbody>
</table>

Table 3.2: XD API Operations for Managing Group Objects

3.1.3 Specifying Object Relationships

There are two important kinds of object relationships an object model must be capable of expressing: the traditional database relationships (one-to-many and many-to-many) and the “is_a” relationship implied by inheritance as explained in [Har00]. The is_a relationship specifies an inheritance hierarchy where objects positioned lower in the hierarchy either represent specialisations of the ones on higher levels in which case it is named a generalisation-specialisation relationship, or expand the object characteristics (“extends” relationship). Further, we distinguish the whole-part relationship denoting the fact that an object can be composed of other objects. This can be expressed using many-to-many relationships.

The XD model supports these kinds of relationships by the association refer_to between meta objects as shown in figure 3.3.

For example, to create an association between a product and a company object, we can use the association statement of the data manipulation language (DML):
create object product1;
create object company1;

association(product1, company1);

The specified_by association between refer_to association and Objects collection makes it possible to further characterise the refer_to association by assigning facts. Following the definition of a fact in the context of logic programming, a fact is a "special case of a Horn-clause with an empty rule body, i.e., a clause without premises" [CS98].

For instance, a fact can be used to assign attributes and values to refer_to associations in our example. Suppose, we want to keep track of how many items of a certain product have been manufactured by a specific company. First, we create a fact holding the number:

create object fact1;
assert(fact1, number, 3);

Then, we assign this object to the refer_to association:

fact((product1, company1), fact1);

A fact itself is also a meta object and can therefore have methods associated to it. In this case we call such an object not a fact, but a rule in analogy to Horn-clauses, and the methods correspond to rule bodies.

In Prolog, for instance, clauses without a body are called unit clauses or facts, whereas rules are non-unit clauses. Clauses are specified in terms of terms, atoms and queries [Apt97]:

Figure 3.3: Object Relationship
• **Terms** are defined as follows:
  
  – a variable or constant is a term.
  – if $f$ is an $n$-ary function symbol and $t_1, ..., t_n$ are terms then $f(t_1, ..., t_n)$ is a term.

• if $p$ is an $n$-ary relation symbol and $t_1, ..., t_n$ are terms then $p(t_1, ..., t_n)$ is an atom.

• a **query** is a finite sequence of atoms.

• a **clause** is a construct of the form $H ← B$, where $H$ is an atom and $B$ is a query; $H$ is called its **head** and $B$ its **body**.

Thus, we could represent fact1 of our example as unit clause in Prolog:

```prolog
fact1((number, 3)).
```

A detailed example of how to use facts and rules is given in chapter 5.

For supporting the activity of specifying relationships, at least the operations listed in table 3.3 must be provided by the API of a system implementing the XD meta model.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW ASSOCIATION</td>
<td>Creates an association between two objects</td>
</tr>
<tr>
<td>DELETE ASSOCIATION</td>
<td>Dissolves an association between two objects</td>
</tr>
<tr>
<td>ASSOCIATED OBJECTS</td>
<td>Returns all associated objects of an object</td>
</tr>
<tr>
<td>NEW FACT</td>
<td>Creates a relation between an object relation and an object</td>
</tr>
<tr>
<td>DELETE FACT</td>
<td>Dissolves a relation between an object relation and an object</td>
</tr>
<tr>
<td>RELATED FACTS</td>
<td>Returns all related facts</td>
</tr>
</tbody>
</table>

Table 3.3: XD API Operations for Managing Associations and Facts

### 3.1.4 Defining Object Properties

Properties of meta objects such as methods, attributes and their values can be specified by the part of the XD meta model shown in Figure 3.4.

**Defining attributes, values and types**

Object data is expressed on two levels. By the association are_specified_by between the collection **Objects** and **Attributes**, it is possible to assign attribute
Figure 3.4: Object Properties
objects to meta objects. An attribute object serves as a placeholder for grouping value entities in a similar way as group objects are used for classifying meta objects.

Value objects are related to meta objects by the association refer_to between collection Objects and Value Entities. Value objects can be grouped by the association are_defined_by between collection Attributes and Value Entities. It is possible that meta objects have associated value objects which are not related to attribute objects.

The following example creates an attribute and value object by means of DML statements:

```
create object product1;
assert(product1, name, chair1);
```

The first statement creates a new meta object. The assert statement creates an attribute object name, associates it to the meta object product1, creates a value object chair1 and associates it to the attribute as well as to the meta object.

To create such an object, for example using the programming language Java, we have to provide a class definition:

```
public class Product {
    private String name;

    public Product() {
    }

    public synchronized void setName(String name) {
        this.name = name;
    }

    public synchronized String getName() {
        return name;
    }
}
```

Then, we can create and initialise instances of this class by the following Java statements:

```
Product product = new Product();
product.setName("chair1");
```

Specifying, for instance, a new attribute weight for the product object product1, can be done by the following DML statement in the case of the XD meta model:
assert(product1, weight, 23);

To do that in Java, we need to change the class definition by adding the attribute weight and its access methods:

```java
private double weight;

public synchronized double getWeight() {
    return weight;
}

public synchronized void setWeight(double weight) {
    this.weight = weight;
}
```

A value object in the XD meta model can be accessed either through the corresponding meta object or through the attribute object. Multi-valued attributes can be created in two ways. One way is to group together several value objects by asserting them to the same attribute object as shown by the following example in which the two value objects address1 and address2 are related to the email attribute object.

```java
create object company1;
assert(company1, email, address1);
assert(company1, email, address2);
```

In Java, we could specify an attribute being of type Vector which can be used to manage a list of objects, e.g.:

```java
import java.util.Vector;

public class Company {
    private String name;
    private Vector emailAddresses = new Vector();

    public synchronized void addEmailAddress(String address) {
        emailAddresses.addElement(address);
    }

    ...
}
```

The other way is to create a value set by linking value objects through the association refer_to between value entities. This corresponds to the case when an attribute is
regarded as a property function returning a single value which might also be a set or bag of values.

In the XD meta model, the properties of a meta object are not fixed, implying that there is no notation of type or class in the sense of strong typing. A type in the XD meta model denotes a view to object properties and is itself a meta object. By applying the dress operator on meta objects, they gain a specific type view.

For example, the following DML statements create the meta object companyType and the attribute name the value of which is java.lang.String.

```java
create object companyType;
assert(companyType, name, "java.lang.String");
```

This object is used as a type object defining the attribute name of scalar type java.lang.String. A scalar type, also named base type, primitive type or unstructured type, is a type “whose elements consist of indivisible entities” [Sta95]. For instance, integer, string, boolean, or real are scalar types. The companyType object is an example for a compound or structured type.

Now, we “dress” the meta object company1 with this type by the following DML statements:

```java
create object company1;
dress company1 as companyType values
(    name = "Company 1";
);
```

Our approach of supporting types or classes by the concepts of type views is very similar to the one used for the OMS Pro rapid prototyping system [Wür00]. In OMS Pro, objects are represented by type units and information units. A type unit is a description of the contextual representation of an information unit which corresponds to a type view object. An information unit is an atomic element of object data represented in OMS Pro as the triple (O,T,V). O denotes an unique object identifier which is in our case a meta object. T is a type descriptor corresponding to an attribute object, and V specifies a set of value objects.

**Defining methods**

Method objects are related to meta objects by the association have between collection Objects and Methods. Depending on the implementation platform on which the meta model is running, methods can be specified using a programming language such as Java, or using the various XD languages such as the data manipulation language (DML) or the XD meta language (XDMQL). These languages are evaluated at runtime. The following method implementation is an example for a method which
creates new company objects. It is written in terms of XDML statements. XDML is based on Oberon-0 [Wir96] and provides additional operations for the meta model such as for creating relationships between meta objects or specifying object properties. We chose Oberon-0 because it is simple to learn and it offers a minimal set of necessary programming language constructs.

```
MODULE addCompany;

SIGNATURE addCompany(): BOOLEAN;

VAR
  company, companies, companyType: OBJECT;
  labels, values, results, indexes: LIST;
  ok: BOOLEAN;
  name: STRING;

BEGIN
  companyType.retrieve("companyType");
  companies.retrieve("Companies");

  // all statements for user interaction are omitted for
  // the sake of simplicity

  // the variable 'name' contains the value of the company
  // name which has been entered by the user
  writeln("adding company ", name, ");

  company.new(name);
  company.dress(companyType);
  company.assert("name", name);
  company.retrieve("Companies");
  companies.insert(company);

  RETURN TRUE;
END addCompany.
```

The `dress` operation has the same meaning as the DML `dress` operator. The XDML statement

```
companyType.retrieve("companyType");
```

retrieves the type object from the database and binds it to the variable `companyType`. And by

```
company.dress(companyType);
```
the meta object bound to the variable `company` is “dressed” by the type object `companyType`. That means that all further operations on the `company` object such as `assert` are evaluated as if it would be of type `companyType`. Hence, the statement

```java
company.assert("name", name);
```

checks whether there is an attribute `name` defined by the `companyType` object, and if so creates a value object, in our case of type `java.lang.String`, and assigns the value bound to the `name` variable to it. This value object is finally associated to the `name` attribute object as well as to the `company` meta object.

**Operations for managing object properties**

A system supporting the XD meta model must at least provide implementations for the operations listed in table 3.4 which are needed for managing object properties.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE METHOD</td>
<td>Creates a method object</td>
</tr>
<tr>
<td>MODIFY METHOD</td>
<td>Modifies the name or source code of a method object</td>
</tr>
<tr>
<td>DELETE METHOD</td>
<td>Deletes a method object</td>
</tr>
<tr>
<td>ASSIGN METHOD</td>
<td>Assigns a method object to an object</td>
</tr>
<tr>
<td>EVALUATE METHOD</td>
<td>Evaluates a method of a method object</td>
</tr>
<tr>
<td>CREATE ATTR</td>
<td>Creates an attribute object</td>
</tr>
<tr>
<td>MODIFY ATTR</td>
<td>Modifies the name of an attribute object</td>
</tr>
<tr>
<td>DELETE ATTR</td>
<td>Deletes an attribute object</td>
</tr>
<tr>
<td>ASSIGN ATTR</td>
<td>Assigns an attribute object to an object</td>
</tr>
<tr>
<td>ASSERT</td>
<td>Asserts a value object to an object and attribute</td>
</tr>
<tr>
<td>RETRACT</td>
<td>Retracts a value object from an object and attribute</td>
</tr>
<tr>
<td>MATCH</td>
<td>Returns all value objects related to an object and attribute</td>
</tr>
</tbody>
</table>

Table 3.4: XD API Operations for Managing Object Properties

### 3.2 The XD Algebra

In this section we describe a minimal set of operations for the XD meta model — called the *XD Algebra* (XDA). These operations must be implemented by any system supporting the meta model. The XD Algebra consists of algebraic operations for all three parts of the XD meta model. These are used, for example, by the XD languages such as the data manipulation language, the query language and the XD meta language. There are two groups of algebraic operations. One group includes set
operations from mathematical set theory such as union, intersection and difference, the other group covers operations specifically defined for the XD meta model such as the ones listed in table 3.1, 3.2 and 3.3.

Basically, the XD algebra provides similar operations as the ones defined for the generic object model OM which extends the notion of the relational data model algebra. The relational algebra operators are [GMUW00]:

- **Union, intersection and difference**
  On sets, these are the usual set operators. On bags, there are some differences in terms of how to treat tuples that appear more than once in a relation. In the case of the XD meta model, these operations can be applied on object groups, on relationships as well as on object properties.

- **Selection**
  This operator selects tuple elements from a relation according to some condition or predicate. Tuple elements of a relation correspond to object properties in the XD model.

- **Projection**
  This operator produces a new relation from an old one by choosing some of the tuple elements. In the case of the XD meta model a meta object is created choosing object properties from another meta object.

- **Product**
  This operator is the set-theoretic Cartesian product (cross-product) constructing tuples by pairing the tuples of two relations in all possible ways. Translated to the XD meta model, this means that the cross-product between two object groups creates relationships between the members of the first and second group in all possible ways.

- **Join**
  There are a number of useful join operators which combine related tuples from two relations into a single tuple. These operators are built from a product followed by a selection and projection.

Note that in the case of the XD meta model, the results of these operators are always meta objects having a unique object identifier. This contrasts with approaches proposed for object-oriented data management such as the object data standard ODMG 3.0 [CBB+00] where, for instance, the result of a select operation can also be an object without an identity. Thus, there is a distinction between objects having an object identifier and literals the identity of which equals their value. The following statement creates a structure with two fields as an example for an object without identity:

```plaintext
struct(a: 10, b: "Test")
```
Since all objects in the meta model have an identity by default, it is not necessary to distinguish objects with and without identity which keeps the implementation of algebra operators orthogonal and simple.

For the implementation of algebra operations it is necessary that a system supporting the XD meta model provides basic operations for managing meta objects, group objects, relations, facts, attribute, value and method objects. We refer to these operations as *XD API operations* emphasising that these operations should be part of the application programming interface (API) of a XD system. Table 3.1 lists XD API operations for managing meta objects.

### 3.2.1 Algebra Operations for Object Groups

The XD Algebra includes the set operations *union*, *intersection* and *difference* over object groups as well as the *selection* operation on object groups. These operations are based on the XD API operations shown in table 3.2.

The result of a *union* operation applied on two group objects is a group object containing all their members.

\[
\cup : (\text{GroupObject}, \text{GroupObject}) \to \text{GroupObject}
\]

\[
G_1 \cup_{\text{group}} G_2 = \{x \mid x \in G_1 \lor x \in G_2\} \tag{3.11}
\]

An *intersection* operation creates a group object containing members which are in both operand group objects.

\[
\cap : (\text{GroupObject}, \text{GroupObject}) \to \text{GroupObject}
\]

\[
G_1 \cap_{\text{group}} G_2 = \{x \mid x \in G_1 \land x \in G_2\} \tag{3.12}
\]

The *difference* operation creates a group object containing all the members of the first operand group object which are not member of the second operand group object.

\[
- : (\text{GroupObject}, \text{GroupObject}) \to \text{GroupObject}
\]

\[
G_1 -_{\text{group}} G_2 = \{x \mid x \in G_1 \land x \notin G_2\} \tag{3.13}
\]

The *selection* operation selects members of an object group satisfying a given predicate *Pred*, i.e. only those members are selected which return "true" when the predicate is applied.

\[
\% : (\text{GroupObject}, \text{Pred} \to \text{boolean}) \to \text{GroupObject}
\]

\[
G \%_{\text{group}} \text{Pred} = \{x \mid x \in G \land \text{Pred}(x) = \text{true}\} \tag{3.14}
\]
3.2.2 Algebra Operations over Meta Objects

The set operations union, intersection and difference over a meta object \( O \) comprises operations over the relationships, facts and properties of \( O \) denoted as \( \text{relations}(O) \), \( \text{facts}(O) \), \( \text{attrs}(O) \) and \( \text{methods}(O) \).

If the operator \(*\) is one of the set operators \( \cup \), \( \cap \) or \( -\), and \( O_1 \) and \( O_2 \) are meta objects then we define \(*\) as

\[
* : (\text{MetaObject}, \text{MetaObject}) \rightarrow \text{MetaObject}
\]

such that if \( O = O_1 * O_2 \) then

\[
\begin{align*}
\text{relations}(O) &= O_1 *_{rel} O_2 \\
\text{facts}(O) &= O_1 *_{\text{fact}} O_2 \\
\text{attrs}(O) &= O_1 *_{\text{attr}} O_2 \\
\text{methods}(O) &= O_1 *_{\text{method}} O_2
\end{align*}
\]

\(*_{\text{rel}}, *_{\text{fact}}, *_{\text{attr}} \) and \(*_{\text{method}} \) are defined in the following sections.

3.2.3 Algebra Operations for Object Relationships and Facts

The algebra operations for relationships and facts are the set operations union, intersection, difference and product which are in this case based on the XD API operations listed in table 3.3.

A union operation between two meta objects results in a meta object which contains the relationships, respectively, the facts of both objects. In the following definitions, the result of the operator \( \text{relations}(\text{object}) \) is the set of objects related to \text{object}, and the operator \( \text{facts}(\text{object}) \) returns all facts of all object relationships.

\[
\begin{align*}
O_1 \cup_{rel} O_2 &= \{ x | x \in \text{relations}(O_1) \lor x \in \text{relations}(O_2) \} \\
O_1 \cup_{\text{fact}} O_2 &= \{ x | x \in \text{facts}(O_1) \lor x \in \text{facts}(O_2) \}
\end{align*}
\]

The intersection and difference operations over relationships and facts are defined accordingly:

\[
\begin{align*}
O_1 \cap_{rel} O_2 &= \{ x | x \in \text{relations}(O_1) \land x \in \text{relations}(O_2) \} \\
O_1 \cap_{\text{fact}} O_2 &= \{ x | x \in \text{facts}(O_1) \land x \in \text{facts}(O_2) \}
\end{align*}
\]
\[ O_1 -_{rel} O_2 = \{ x \mid x \in \text{relations}(O_1) \land x \notin \text{relations}(O_2) \} \]  
(3.23)

\[ O_1 -_{fact} O_2 = \{ x \mid x \in \text{facts}(O_1) \land x \notin \text{facts}(O_2) \} \]  
(3.24)

The product operation applied to two object groups creates relationships between members of the first and second group in all possible ways.

\[ x : \text{GroupObject}, \text{GroupObject} \rightarrow \text{GroupObject} \]

\[ G_1 \times_{rel} G_2 = \{ [x,y] \mid \exists x,y \in G_1 \land y \in G_2 \} \]  
(3.25)

### 3.2.4 Algebra Operations for Object Properties

The algebra operations for object properties cover the set operations union, intersection and difference as well as the projection operation. These operations are based on the XD API operations listed in table 3.4.

There are two union operations on object properties. One builds the union of all attribute and value objects of two meta objects, the other one results in the union of all method objects. In the following definitions, \text{attrs(object)} refers to all attribute and value objects of object, and \text{methods(object)} to all method objects.

\[ O_1 \cup_{attr} O_2 = \{ x \mid x \in \text{attrs}(O_1) \lor x \in \text{attrs}(O_2) \} \]  
(3.26)

\[ O_1 \cup_{method} O_2 = \{ x \mid x \in \text{methods}(O_1) \lor x \in \text{methods}(O_2) \} \]  
(3.27)

The intersection and difference operations are defined accordingly:

\[ O_1 \cap_{attr} O_2 = \{ x \mid x \in \text{attrs}(O_1) \land x \in \text{attrs}(O_2) \} \]  
(3.28)

\[ O_1 \cap_{method} O_2 = \{ x \mid x \in \text{methods}(O_1) \land x \in \text{methods}(O_2) \} \]  
(3.29)

\[ O_1 -_{attr} O_2 = \{ x \mid x \in \text{attrs}(O_1) \land x \notin \text{attrs}(O_2) \} \]  
(3.30)

\[ O_1 -_{method} O_2 = \{ x \mid x \in \text{methods}(O_1) \land x \notin \text{methods}(O_2) \} \]  
(3.31)
The *projection* operation takes as arguments a meta object and a set of property names \((PN_{set})\) referring to attribute and method objects, and it creates a new meta object with only those properties matching the names in the set.

\[
\triangleright: (O_1, PN_{set}) \rightarrow O_2
\]

\[
v \triangleright O_2 = v
\]  

(3.32)

In the next chapter, we present two frameworks which support the XD meta model and its algebra, and describe how to apply the XD approach using these frameworks.
Chapter 4

Applying eXtreme Design

The eXtreme Design Approach (XDA) can be applied at three different levels as shown in figure 4.1. These levels correspond to the degree of abstraction in terms of application design and implementation.

![Diagram](image)

Figure 4.1: Three Application Levels of eXtreme Design

The top level labelled “XD Meta Model” represents all application systems and frameworks supporting the XD meta model together with its algebra operations. We implemented two such systems which can be used for the development of eXtreme Design applications. We refer to these systems as XD systems and present them in section 4.2.
On the second level, the XD meta model is used for the specification and implementation of meta models for specific areas. For instance, in chapter 5 we present a meta model for product information systems which is an example for an application meta model. We show by this example that the XD meta model can be used to specify a meta model for a specific application domain such as product data management, and that the XD system can serve as a prototyping platform for implementing a prototype of the application system. In this case, the application meta model together with the prototype can be regarded as the specification for the application design and implementation which means that the application model and the prototype serve as basis for designing and implementing the final application system. This activity is denoted by the arrow from "Application Meta Model" to "Application Model" in figure 4.1.

On the other hand, we can use the XD meta model for directly designing and implementing parts of an application system which is represented by the arrow from "XD Meta Model" to the third level, the level of application systems named "Application Model" in figure 4.1. We show in chapter 6 how we have applied eXtreme Design for extending the storage management component of the object management system OMS Java as an example for component modelling.

The eXtreme Design approach is further suitable for carrying out rapid information modelling (RIM) which is the activity of defining the information model of a specific application domain. The arrow from "XD Meta Model" to "Information Model" in figure 4.1 represents this activity. Chapter 7 illustrates how eXtreme Design supports rapid information modelling.

In the next section, we give an example of how to apply the eXtreme Design approach. A description of the eXtreme Design framework can be found in section 4.2.

### 4.1 eXtreme Design: an Example

In this section we explain the eXtreme Design approach by means of an example. Suppose, a book store company intends to sell books on the Internet. During the requirements analysis it becomes clear that the existing information system of the company cannot be adapted to the new requirements. Before building a complete new system, the company decides to first implement a prototype system for evaluation purposes.

After having specified the main requirements the system must fulfil, we create a conceptual schema for defining the main concepts of our example application which is depicted in the schema of figure 4.2.

As a next step, we map the conceptual schema to the XD meta model. For each concept in the schema, we create a meta object. So, for the concept Customers, Orders and Books which represent classification objects or collections, we create the
The following objects in terms of data manipulation language (DML) statements:

```java
create object Customers;
create object Orders;
create object Books;
```

Further, we specify type views which define properties of application objects in our system. Note that this step is not necessary because properties of meta objects can be added or changed without having to provide type view definitions. Since in our example we want to ensure that certain properties are always present, we define the following type views corresponding to application objects representing customers, orders and books:

```java
create object customerType;
assert(customerType, name, "java.lang.String");
create object orderType;
assert(orderType, date, "java.util.Date");
assert(orderType, quantity, "java.lang.Integer");
create object bookType;
assert(bookType, ISBN, "java.lang.String");
assert(bookType, title, "java.lang.String");
assert(bookType, author, "java.lang.String");
```

We are now ready for developing the prototype. For this, we can use the eXtreme Design framework described in the next section for implementing the core of our application, and Java Servlets [Mos98] for realising the connection between our application and the web server.
There are two alternatives for the application to use the XD framework for managing application objects. One way is to use the application programming interface (API) provided by the XD framework. The other way is to use the XD data manipulation language (DML) and XD meta language (XDML). The advantage of this is that these languages are evaluated at runtime which is especially useful in the prototyping phase where changes are often necessary.

For populating our database, we create some customer and book objects, assign values to them and insert them in the appropriate collections by the following DML statements:

```plaintext
create object customer1;
dress customer1 as customerType values
  (name = "Customer 1";
);

create object customer2;
dress customer2 as customerType values
  (name = "Customer 2";
);

insert into Customers: [customer1, customer2];

create object book1;
dress book1 as bookType values
  (ISBN = "123",
   title = "Book 1",
   author = "author 1";
);

create object book2;
dress book2 as bookType values
  (ISBN = "34-56",
   title = "Book 2",
   author = "author 2"
);

insert into Books: [book1, book2];
```

Adding, for example, a new order to the database, involves the following steps: first we have to create the order object, specify its values and insert it into collection orders. This can either be done by DML statements such as
create object order1;
dress order1 as orderType values
(
    date = "11.02.2000";
    quantity = "2";
);

insert into Orders: [order1];

or within a XDML method:

MODULE addOrder;

SIGNATURE addOrder(): BOOLEAN;

VAR
...
    order, orderType, orders: OBJECT;
    orderDate; DATE;
    quantity: INTEGER;
...

BEGIN

    // step 1: select customer
    ... 

    // step 2: select book
    ... 

    // step 3: create order
    ...

    orderDate = ...;
    quantity = ...;

    orderType.retrieve("orderType");
    order.new();
    order.dress(orderType);
    order.assert("date", orderDate);
    order.assert("quantity", quantity);
orders.retrieve("Orders");
orders.insert(order);
...

RETURN TRUE;
END addOrder.

The XD meta language is discussed in more detail in section 4.3 and chapter 5. The syntax specification of the meta language is given in appendix B and the one of the data manipulation language in appendix A.

Finally, we create the necessary associations between the customer object and the selected book using the corresponding DML or XDML statements:

association(customer1, order1);
association(order1, book1);

or

customer-> := order; // 'customer' refers to the customer object
order-> := book; // 'book' refers to the book object

If at some point while using the system it becomes evident that, for instance, the customer objects need to have additional properties, we can either add new properties to an existing customer object, for example, by the following DML statements:

assert(customer1, address, "address 1");
assert(customer1, phone, "123-456");

or we can change the type view of customer objects:

assert(customerType, address, "java.lang.String");
assert(customerType, phone, "java.lang.String");

in which case we can define the value by the dress operator:

dress customer1 as customerType values
    (    address = "address 1";
    phone = "123-456";
);
The first case is an example for the fact that it is possible to add new properties to a specific application object without having to change, for instance, type information which also means that only this application object “gains” the new properties. In the second case, we change the type view so that all application objects seen through, or “dressed” by, this type view carry the new properties.

Also changes in the conceptual schema can be made without having to change schema information. For example, introducing a new collection UrgentOrders requires only the creation of a new meta object representing this collection:

```
create object UrgentOrders;
```

So, we could then insert order objects into this collection by:

```
insert into UrgentOrders: [order1];
```

The prototype system could either evolve into the final application system, or could serve as a specification for the implementation phase, in which case one could derive schema information and class definitions by analysing the meta objects as proposed in [Ign00].

In the next section, we present two frameworks supporting the XD meta model and its algebra.

### 4.2 The eXtreme Design Framework (XDF)

There are mainly two possibilities for implementing a system for the XD meta model and its algebra operations. On one hand, since the XD meta model itself is defined in terms of OM, we can build such a system on top of an object management system which supports the OM object model such as OMS Java. OMS Java is described in chapter 6 in more detail. In this way, the meta model is simply defined as an OM schema, and the algebra operations can be implemented, for example, using the API and the algebra operations of the OM system. An advantage of this approach is that extensions to the meta model can be made by just changing the schema definition which facilitates adapting the meta model and its operations to a specific application domain. Hence, such a system is very suitable for prototyping.

We implemented such a system on top of OMS Java. The XD meta model is in this case defined in terms of an OM schema. The following data definition language (DDL) statements specify all necessary types:

```
type metaObject
type metaGroup subtype of metaObject
{
    name: string;
};

type metaAttribute
{
    name: string;
};

type metaValue
{
    value: javaObject;
};

type metaMethod
{
    name: string;
    script: text;
    evaluate : ( oid: string ) -> ( );
};

These types are implemented as OMS Java objects, i.e. as Java classes fulfilling the OMS Java interface OMInstance (see chapter 6). Further, all collections of the XD meta model are specified as

collection ObjectGroups: set of metaGroup;
collection Objects: set of metaObject;
collection Methods: set of metaMethod;
collection Attributes: set of metaAttribute;
collection Values: set of metaValue;

and the associations as

collection Groups_member_of_Groups:
    set of (metaGroup, metaGroup);
constraint Groups_member_of_Group
    association from ObjectGroups (0:* ) to ObjectGroups (0: *);

collection Objects_member_of_Groups:
    set of (metaObject, metaGroup);
constraint Objects_member_of_Groups
association from Objects (0:*) to ObjectGroups (0:*)

collection Objects_refer_to_Objects:
    set of (metaObject, metaObject);
constraint Objects_refer_to_Objects
    association from Objects (0:*) to Objects (0:*)

All operations for managing XD meta objects are specified in the class XDSystem which serves as an application programming interface (API) for client applications.

Another approach for implementing a system supporting the XD meta model is to create a system comprising all data structures necessary for managing meta objects and their relationships as well as group, attribute and value objects. With this approach, we can build an XD system which is optimised for a specific implementation platform.

We have implemented such a system for the Java environment, called the eXtreme Design Framework (XDF). The framework consists of the client application components XD Client and XD Client Object, and of the server components XD Meta Object, XD Workspace and Database Connection as depicted in figure 4.3.

![Diagram](image)

Figure 4.3: The eXtreme Design Framework Architecture
The XD workspace resides on the server side and is responsible for managing all XD objects. The workspace is linked to the XD database connection component which is capable of making XD objects persistent. The connection component itself uses a database management system (DBMS) as storage manager. We have implemented links to a relational DBMSs as well as to object-oriented ones. Hence, by decoupling the storage manager from the workspace, the XD framework becomes independent of the underlying DBMS. So the DBMS can be exchanged without having to carry out changes in the workspace or client applications.

The client application accesses XD objects through the XD meta object component to which it is remotely linked by the remote method invocation mechanism (RMI) supported by the Java environment. This component provides methods for remotely managing XD objects, and thus can be regarded as a remote interface to the XD workspace.

Another possibility for a client application to connect to an XD workspace is through the proxy classes XD client and XD client object. A proxy class “provides a surrogate or placeholder for another object to control access to it” [GHJV95]. The XD client represents the part of the XD meta object component which provides functions for creating meta objects and for managing object groups and object relationships. In addition, the XD client serves as an interface for the various languages such as the XD data manipulation language (DML) and the XD meta language (XDML). The XD client object is a placeholder for a concrete meta object of the XD workspace and offers functions, for example, for accessing attribute and value objects as well as for evaluating meta object methods. The advantage of using proxy classes for accessing the XD workspace and encapsulating XD meta objects is that from the client application’s perspective a XD meta object is like a normal Java instance.

In the next section, we discuss the XD meta language and its interpreter.

4.3 XD Meta Language (XDML)

The eXtreme Design Meta Language is based on Oberon-O [Wir96] which is a subset of the programming language Oberon [RW92]. Oberon-O comprises a minimal set of necessary programming language features such as value assignment, statement sequences, if and while constructs as well as the concept of procedures. Hence, the language is simple to learn and is expressive enough for specifying application processes. Further, the simplicity of the language keeps the compiler small and efficient which is important with respect to runtime compilation and evaluation.

We extended Oberon-O with new base types and with operations for the XD algebra and for user interaction. For example, we introduced the notion of type LIST representing a list of objects and with type OBJECT representing an XD meta object. Further, we introduced operations on XD meta objects. This was achieved in two ways. For some operations we changed the syntax of the language. For instance,
the \( \rightarrow \) operator refers to associations between meta objects and the \( \rightarrow \rightarrow \) operator can be used for managing facts.

\[
\text{MODULE test; }
\]
\[
\text{VAR}
\text{    object1, object2, fact: OBJECT;}
\text{    associatedObjects, facts: LIST;}
\text{BEGIN}
\text{    ...}
\text{    // the next statement creates an association between}
\text{    // object1 and object2}
\text{    object1\rightarrow := object2;}
\text{    ...}
\text{    // the next statement returns a list of all objects}
\text{    // associated to object1}
\text{    associatedObjects := object1\rightarrow;}
\text{    ...}
\text{    // the next statement associates a fact object}
\text{    // to the association between object1 and object2}
\text{    object1\rightarrow\rightarrow object2 := fact;}
\text{    ...}
\text{    // the next statement returns a list of facts associated}
\text{    // to the association between object1 and object2}
\text{    facts := object1\rightarrow\rightarrow object2;}
\text{END test.}
\]

The other operators are realised in terms of method calls. We extended the language syntax in such a way that it is possible to invoke a method associated with an object. For example, the statement

\[
\text{object.setName("object name");}
\]

calls the method \textbf{setName} associated with the meta object referred to by the variable \textbf{object} giving "object name" as input parameter. So, the operators are specified as predefined methods which belong to every object of type \textbf{OBJECT}. For example, to assert a new value to an object, we can use the \textbf{assert} method:
object.assert("phone", "123-456");

XD meta language methods are supposed to be evaluated at runtime, i.e. it is possible to change methods while the XD system is running. The system automatically compiles methods before evaluating them. “Compiling” means creating intermediary statements which can be interpreted by an interpreter.

In our system, these statements are instances of special Java classes providing an execute method which is called by the interpreter at runtime. Hence, for any given XD meta language method, the compiler creates a list of corresponding Java instances. This list is then given to the interpreter for evaluation.

For example, for the boolean operations “not”, “and” and “or”, the compiler creates an instance of the Java class BooleanStatement which is capable of evaluating boolean operations. The input parameters for statements are pushed by the compiler on the parameter stack of the interpreter and can be obtained by the method popParam() provided by the interpreter. For instance, the execute method of the BooleanStatement class looks as follows:

```java
public synchronized boolean execute(Interpreter interpreter) {
    boolean result = false;

    // the method 'popParam()' retrieves parameters
    // for the statement from the parameter stack
    ObjectDesc p1 = interpreter.popParam();
    if (!isValidParameter(p1)) return false;
    if (p1.getType() != ObjectDesc.BOOLEAN) {
        message = "parameter must be of type BOOLEAN";
    }

    // the variable 'statement' which is set by the compiler
    // specifies what kind of boolean statement should
    // be evaluated
    if (statement.equals("not")) {
        result = !((Boolean)p1.getValue()).booleanValue();
    } else {
        ObjectDesc p2 = interpreter.popParam();
        if (!isValidParameter(p2)) return false;
        if (p2.getType() != ObjectDesc.BOOLEAN) {
            message = "parameter must be of type BOOLEAN";
            return false;
        }

        if (statement.equals("and")) {
            result = 
                (((Boolean)p1.getValue()).booleanValue())
```
&& ((Boolean)p2.getValue()).booleanValue();
}
}
}

// the 'pushParam()' method is used for
// pushing results on the parameter stack
interpreter.pushParam
    (new ObjectDescImpl(statement, ObjectDesc.CONST,
                         ObjectDesc.BOOLEAN,
                         new Boolean(result)));
    return true;
}

Since the XD meta language methods are translated into statements specified as Java classes, it is possible to extend the meta language by changing these classes. For example, all operators for XD meta objects are defined in the Java class ObjectStatement. So, we can, for instance, introduce a new operator cardinality which, applied to an object of type OBJECT, returns the number of members of that object. This new operator is specified in an extension of class ObjectStatement. Further, we need to change the factory of the compiler in such a way that, whenever the compiler generates a statement for meta object operators, it creates an instance of our extended ObjectStatement class. Hence, the meta language statement

size = object.cardinality();

is then evaluated by an instance of the extended version of the ObjectStatement class.

The syntax of the XD meta language is described in section B.1, whereas an overview of predefined operators can be found in section B.2.

In the following chapters, we present the various aspects of the eXtreme Design approach by means of examples.
Chapter 5

eXtreme Design for Product Data Management

In this chapter we give an example of how the eXtreme Design Approach supports the design process of application systems. Our example is based on the requirements analysis presented in section 2.1. According to these requirements, we have to implement a product information system which makes it possible for various types of users to classify and specify the structure and characteristics of products. Further, the process of product configuration as well as the management of product variants has to be supported.

These are the typical requirements of product data management (PDM). Especially product customisation in terms of the characteristics and variants of products is an important aspect of PDM. We investigated the problem of managing product configurations [KN97, KN98] and implemented a prototype supporting the product configuration process. The prototype is based on the eXtreme Design framework and uses the eXtreme Design meta language (XDML) for controlling the configuration process which is described in section 5.1.4 in more detail.

Looking at proposals [RNS93, Sch96, Bjö95] and solutions [CAD, SAP] for product data management, we can distinguish two main approaches for managing product variants. One approach is to store all possible variants of product parts in maximum parts lists. This makes the product configuration task easier to manage, but it is not efficient in terms of storage resources. Another approach is to use a place holder to indicate positions in a product structure where variants can be chosen. Such a place holder thus represents all parts that could fit into a specific position making it easier to change or extend product structures.

Suppose that for our furniture company it is not obvious at this stage of evaluation whether to invest in an existing application system or to develop a new one. So we assume that they decide to build a prototype system which should provide more information about which aspects of product data management are most crucial.
for the various types of users. It should therefore be possible that all users are able, for example, to add new product properties without having to change the implementation of the prototype.

5.1 The Design Process

It is good practice to start the design process by creating a conceptual schema of the application domain, i.e., by mapping the results of the requirements analysis into diagrams.

Figure 5.1 gives the example introduced in section 2.1 for a conceptual schema describing the relationship between products and companies as well as the various possibilities to classify products.

![Diagram of a conceptual schema with entities like Household, Office, Products, Companies, and relationships such as cover, manufactured_by, and partition.]

To implement such a schema, we also have to specify the types of all entities. Further, the schema must be extended to express the fact that products are composed of other products. So, already in an early stage of the design phase, we are forced to make design decisions having a major impact on the flexibility and extensibility of the implementation. How can the system be extended by new classification structures? How should product variants be supported? Should the place holder concept be used?

The XD approach addresses these questions on a higher level of abstraction since, by using the XD framework, we can build a prototype system which needs not be changed whenever new concepts are introduced. Changes are rather made, for example, by adding new object properties such as attributes and methods at runtime. After having used the prototype system for a while, it becomes possible by analysing classification structures, object properties and relationships to derive a detailed conceptual schema together with descriptions of processes. These results can then serve
as the basis for further analysis and design steps. In chapter 7 we give an example of schema generation.

5.1.1 Product Classification

Classifying products into different product groups reflects the conceptual aspects of products. For example, in figure 5.1 we distinguish two concepts: the product category and the area of utilisation. For creating product groups, we can use the data manipulation language (DML) of the XD framework:

```DML
create object Products;  // this group contains all products
create object Household;
create object Office;
create object Chairs;
create object Lamps;
create object Beds;

It is now possible to classify products by inserting them into corresponding object groups, e.g.:

create object chair1;
create object chair2;
create object lamp1;

insert into Products:  [chair1, chair2, lamp1];
insert into Chairs:     [chair1, chair2];
insert into Lamps:      [lamp1];
insert into Household:  [chair1, lamp2];
insert into Office:     [chair2];
```

5.1.2 Product Characterisation

The characteristics of products such as colour, weight or description are best modelled as attributes of product objects. In the following example, we create first a product object and then assign some characteristics to it. Note that the characteristic colour is specified by more than one value which is an example for a multi-valued attribute.

```DML
create object lamp1
assert(lamp1, description, "Office Lamp No. 235");
assert(lamp1, weight, "1.3 kg");
assert(lamp1, colour, "blue");
assert(lamp1, colour, "yellow");
assert(lamp1, colour, "red");
```
5.1.3 Product Structuring

Products are either single part products or assemblies such as a screw, or they are composed of other products. Using the XD metamodel, we express the structure of a product by creating relationships between product objects. The following example creates a garden chair object which is composed of chair legs and armrests. An armrest itself is composed of upholsteries and screws.

create object gardenchair2;
create object chairleg2;
create object armrest2;
create object upholstery2;
create object screw;

association(gardenchair2, chairleg2);
association(gardenchair2, armrest2);

association(armrest2, upholstery2);
association(armrest2, screw);

As a next step, we have to specify all constraints for these relationships. For instance, the product gardenchair2 might be ordered with or without armrests. We model this by assigning a constraint or fact object to the association between the product gardenchair2 and armrest2:

create object fact1;
assert(fact1, no, 2);
assert(fact1, mandatory, false);

fact((gardenchair2, armrest2), fact1);

Two attributes define the fact1 object. The no attribute specifies the number of armrests and the mandatory attribute states that armrests are an optional component of gardenchair2. A fact in the XD environment can be compared to a fact in logic programming as described in section 3.1.3. In Prolog, for instance, clauses without a body are called unit clauses or facts, whereas rules are non-unit clauses.

5.1.4 Product Configuration

In this section we describe how the configuration process for products can be supported using the XD framework. What is necessary is to define various methods for the following tasks.
• **Product selection**
  First, the user has to select the product which should be configured.

• **Choosing product characteristics**
  Next, the user has to choose values in the case of multi-valued characteristics. For example, if the characteristic `colour` is specified by the values `red` and `blue` then the user must choose one.

• **Configuration of assemblies**
  Finally, all assemblies of the product have to be configured taking into account the corresponding constraints. For instance, in our example there is a constraint for each assembly specifying whether an assembly is mandatory or not.

There are two possibilities for implementing methods. A developer can either use the API of the XD framework or the XD meta language. In the first case, methods are written entirely in the programming language of the XD framework which is in our case the language *Java*. This implies that all facilities and services available for the Java environment can be used for implementing the various methods. On the other hand, it is also necessary to use a compiler for transforming the source code of a method to executable form. Hence, it is not easy to change a method during runtime. This approach is presented in the next chapter.

Using the XD meta language for implementing prototype systems – in our case for modelling the configuration process – is more convenient since this language is interpreted at runtime and methods can therefore be changed without the need of recompilation. The XD meta language (XDML) is based on Oberon-O [Wir96] and extends it with operations for the XD algebra as well as for user interaction (see section 4.3 and appendix B).

Every XDML method is subdivided into three main parts: a header specifying the method name and input/output parameters, a *VAR* part defining all variables, and a body containing the method implementation.

So, for our example we need a method which is evaluated when starting the configuration process. This method is called `startConfig` and returns `TRUE` if the configuration process was successful, and `FALSE` otherwise.

```plaintext
MODULE startConfig;

SIGNATURE startConfig(): BOOLEAN;

VAR
  selectedProduct, configuredProduct: OBJECT;
...

BEGIN
```
The \texttt{SIGNATURE} statement defines the input parameters and the return type of the method which, in our case, is of type \texttt{BOOLEAN}. There are base types such as \texttt{INTEGER}, \texttt{REAL} and \texttt{STRING}, and there are the complex types \texttt{OBJECT} and \texttt{LIST}. Type \texttt{OBJECT} represents an XD meta object, whereas an instance of type \texttt{LIST} can hold a list of base type or complex type instances.

The body of the \texttt{startConfig} method contains statements for asking the user to select a product, storing the selected product in the variable \texttt{selectedProduct}, and for starting the configuration process for that product the result of which is held by the variable \texttt{configuredProduct}.

```
...  
configuredProduct := selectedProduct.config(selectedProduct);
IF configuredProduct INSTANCE OF NULL THEN
  RETURN FALSE;
ELSE
  WRITELN("configured product = " + STRING(configuredProduct));
  WRITELN("alias = " + configuredProduct.getAlias());
END;
RETURN TRUE;
...
```

Starting the configuration process for a specific product is done by invoking the method \texttt{config} giving the selected product as parameter. This method then returns either an instance referring to the configured version of the product, or \texttt{NULL} if the configuration process failed.

\textbf{MODULE} \texttt{config};

\textbf{SIGNATURE} \texttt{config(product: OBJECT): OBJECT;}

\textbf{VAR}

\texttt{index1, index2: INTEGER;}
\texttt{ok: BOOLEAN;}
\texttt{labels, widgetValues, results: LIST;}
\texttt{newProduct, newAssembly: OBJECT;}
\texttt{products, characteristics, values: LIST;}

81
characteristic: STRING;
assemblies, facts: LIST;

BEGIN
...

RETURN newProduct;

END config.

First, we have to test whether the parameter product is valid, and if so we create a new product object with the same name as product. This new product object represents the configured version of product:

IF product INSTANCE OF NULL THEN
  WRITELN("product not specified");
  RETURN NULL;
ELSE
  newProduct.new(product.getAlias());
  newProduct.assert("name", product.retrieveValue("name"));
END;

Then, we iterate over all characteristics of a product assigning them to the new product. If it is a multi-valued characteristic then we have to ask the user to choose a value.

WRITELN("\nCharacteristics:");

// the next statement stores in the variable 'values'
// the names of all attribute objects
// associated to a meta object and stores them into the list
// 'characteristics'
characteristics := product.getAttributes();

index1 := 0;
WHILE index1 < characteristics.size() DO
  characteristic := characteristics[index1];
  WRITE(characteristic + " = ");

  // the next statement returns a list of value objects
  // associated to the attribute referred by the variable
  // 'characteristic'
  product.match(characteristic, values);

  // if there is only one value object then assign it
  // to the new product object
IF values.size() = 1 THEN
   WRITELN(STRING(values[0]));
   newProduct.assert(characteristic, values[0]);

   // otherwise ask the user to select a value;
   // the index '-1' indicates that an element has to be
   // appended to the end of a list
ELSIF values.size() > 1 THEN
   labels.clear(); widgetValues.clear(); results.clear();
   labels[-1] := characteristic;
   widgetValues[-1] := values;
   ok := FORM(product.getAlias(), labels, widgetValues, results);
   IF NOT ok THEN RETURN NULL END;
   IF results.size() > 0 THEN
      WRITELN(STRING(results[0]));
      newProduct.assert(characteristic, results[0]);
   ELSE
      WRITELN("no value selected");
   END;
END;

Next, we start the configuration process for all assemblies which fulfil, in our example, the mandatory constraint.

WRITELN("\nAssemblies:");

   // 'product->' returns a list of all objects related to the
   // object referred by the 'product' variable
   assemblies := product->;

   index1 := 0;
   WHILE index1 < assemblies.size() DO
      WRITELN(assemblies[index1].getAlias());
      // the '->->' operator returns a list of all facts related to
      // the association between two objects
      facts := product->->assemblies[index1];

      index2 := 0;
      WHILE index2 < facts.size() DO
         // the method 'mandatoryConstraint' returns TRUE if the
         // assembly is either mandatory or chosen by the user;
         // parameters are the current assembly and fact
         IF product.mandatoryConstraint(assemblies[index1],
facts[index2]) THEN

    // the next statement starts the configuration process
    // of the assembly
    newAssembly := product.config( assemblies[index1]);

    IF NOT newAssembly INSTANCE OF NULL THEN

        // the '-> :=' statement creates a new relationship
        // between two meta objects
        newProduct-> := newAssembly;

    END;

    END;
    index2 := index2 + 1;
END;

END;

WRITELN("---------------------");
index1 := index1 + 1;
END;

Finally, we have to provide an implementation for the mandatoryConstraint method which takes an assembly and a fact object as input parameters and returns TRUE if the assembly is mandatory or selected by the user.

MODULE mandatoryConstraint;

SIGNATURE mandatoryConstraint(assembly: OBJECT;
    fact: OBJECT): BOOLEAN;

VAR
    mandatory: BOOLEAN;

BEGIN

    // the next statement retrieves the value of the
    // attribute 'mandatory' associated to the 'fact' object;
    // this value is 'TRUE' if the assembly is mandatory
    mandatory := fact.retrieveValue("mandatory");

    IF NOT mandatory THEN
        mandatory :=
            CONFIRM("add assembly '" + assembly.getAlias());
    END;

    RETURN mandatory;

END mandatoryConstraint.
The mandatoryConstraint method is an example for a method returning a boolean value. Such methods are called *XD rules* expressing the similarity to *rules* and *predicates* in logic programming.

The following example illustrates the process of product configuration based on the XDML methods described in this section. The process is started by evaluating the method startConfig which prompts the user to select a product from a list of available products as shown in figure 5.2.

![Product Configuration](image)

**Figure 5.2: List of Products**

After the user selected a product, the configuration process starts. In our example, assemblies might be mandatory or not which is expressed by mandatory constraints. If an assembly is not mandatory then the dialog box shown in figure 5.3 is presented to the user asking whether the assembly should be part of the configured product or not.

![Dialog Box for Assembly Selection](image)

**Figure 5.3: Dialog Box for Assembly Selection**

The output as well as the state of the evaluation process is presented in a special window as depicted in figure 5.4. The first text box shows the output of statements such as WRITELN, and the second informs about method evaluation.

The output `configuration object = XD_261` means that the object identifier of the newly configured product is XD_261. A user can access this object, for example, through the browser tool which is shown in figure 5.5.

In the list of figure 5.6, we see all assemblies of product XD_261. By double-clicking on a list item, the user can inspect the characteristics of the corresponding assembly
5.1.5 Design and Implementation Issues

The idea of building a prototype system in the context of our example was to find out more about the crucial aspects of product data management for our company. Having a running prototype system makes it possible not only to rapidly adapt system services to user's requirements, but also to analyse and reuse the information stored in the system.

For instance, the XD framework is capable of generating a schema in terms of data definition language (DDL) statements which can be imported by a data management system such as OMS Pro or OMS Java.

as shown in figure 5.7.
It is not always necessary that the final system need be implemented on another platform. Rather, the existing prototype could become part of it.

For implementing the prototype, the XD framework provides various tools, for example, for developing XDML methods. The parser tool shown in figure 5.8 can be used for compiling methods. The XDML compiler generates statements which can be evaluated by the XDML interpreter. It is possible to use the interpreter tool depicted in figure 5.9 to control the evaluation of compiled XDML methods. For example, we can use the Step button for step by step processing. All statements of the compiled XDML method are listed in the top part of the interpreter tool. For instance, 0: getParam means that the statement getParam, which takes the input parameters for the method from the parameter stack, is evaluated as the first statement (see appendix B).
Figure 5.8: Parser Tool for XD Meta Language

Figure 5.9: Interpreter Tool for Evaluating Methods
Chapter 6

The Persistent Object Management Framework OMS Java

In this chapter, we present the persistent object management framework \textit{OMS Java} and show how we can use the eXtreme Design approach for extending the system with new bulk data structures.

“Frameworks are a practical way to express reusable designs” [Joh97a] and are therefore meant to be a flexible abstraction of a specific application domain facilitating the development of applications and application components. For instance, MET++ [Ack96] is a good example for a well designed application framework for multimedia software development.

However, most application frameworks tend to be very complex and are therefore hard to learn and typically offer little or no support for making application objects persistent. Hence, a major goal of the OMS Java project was to address these two problems by designing a framework which is not only easy to use, but also offers various and transparent options for persistent object management.

OMS Java supports the generic object model \textit{OM} and is part of the OMS Database Development Suite [KNW98]. We built the OMS Java system with extensibility in mind making it possible to adapt the system for specific application domains. Hence, the OMS Java framework can be used to develop advanced application systems such as Geographical Information Systems (GIS) through a combination of model and architecture extensibility [KN00b].

\textit{Model extensibility} makes it possible to extend the core object model and its associated languages by new constructs for supporting application specific models such as temporal and/or spatial ones [SKN98]. \textit{Architecture extensibility} is achieved through support for exchangeable storage components and the incorporation of new bulk data structures [KN99, KN00a].
In section 6.1 we describe the architecture of the OMS Java framework. In section 6.2 we give an overview of the main concepts used to support model extensibility. Section 6.3 outlines the OMS Java storage management component, and section 6.4 shows how XD has been applied to introduce new bulk data structures into the storage management component.

### 6.1 OMS Java Multi-Tier Architecture

OMS Java (Object Model System for Java) can be considered both as a multi-tier object management system and as an object-oriented application framework for the Java environment.

OMS Java consists of the two main components *OMS Java workspace* and *OMS Java server* as shown in figure 6.1. The workspace serves as the framework for client applications by providing functions for managing application objects as well as OMS Java objects as explained in section 6.2.1. The workspace can either be part of a client application or be a middleware component between a client application and an OMS Java server. We use the term *middleware component* to specify a software component which resides between the client and the server and which supports client/server interactions through multiple communication and data access protocols [Ber96].

![Figure 6.1: OMS Java Multi-Tier Architecture](image)

One or more OMS Java workspaces can be connected to an OMS Java server using the Java Remote Method Invocation Mechanism (Java RMI) for inter-component communication [Dow98]. The server manages all persistent objects of the workspaces and is itself linked to one or more database management systems which are used as storage managers for these objects. This means that the server maps the workspace objects to DBMS specific database objects. The DBMS can be an ODMG-compliant
[CBB+00] object-oriented DBMS (OODBMS), a relational DBMS (RDBMS) connected to the server by JDBC, the standard Java data access interface to RDBMSs [Ree97], or any other DBMS providing a Java interface. Details of this mapping mechanism are discussed in section 6.3.

The server further delegates tasks such as transaction management or data protection to these DBMSs. Regarding security, most existing DBMSs offer mechanisms for identifying and verifying users (identification and authentication) and for access control to data (authorisation) [Lar95]. However, to achieve a secure system, it is essential that all network connections between the various components of a multi-tier database system such as OMS Java are also made secure. For example, mechanisms are required to ensure that the link between a client application and the OMS Java workspace, and between the workspace and the OMS Java server are secure. For this purpose we have implemented our own security framework [Ost99].

6.2 OMS Java Core System

OMS Java supports the generic object model OM which specifies data structures for managing collections of objects and associations together with a rich set of algebra operations as well as integrity constraints. Further, OMS Java provides the same languages as the OMS Pro rapid prototyping system [Wür00]. In particular, these are the data definition language (DDL) for specifying data structures and integrity constraints, the data manipulation language (DML) for update operations, and the query language AQL. In OMS Java, components such as the algebra and the various languages can be exchanged or extended as illustrated in figure 6.2.

The Core System provides functions for managing OM Objects, OM Collections and OM Constraints. The Configuration Component consists of all those parts of the system that are exchangeable and extensible. For instance, we have extended OMS Java to support the temporal object model TOM [Ste98] and this included the specification and implementation of extensions for the object identifier, the algebra, the various integrity constraints and languages.

6.2.1 Basic Concepts

As has been described in section 2.2.3, the OM model supports object role modelling through separation of typing and classification. This is an important requirement for model extensibility and implies that more than one type representation can be associated with an object of the application domain. Figure 6.3 gives an example of such an application object.

Depending on the context, the object with identity 123 is accessed through type person, student or employee. For instance, if the object is accessed through the collection Students then it is viewed in terms of its student role and hence as being of
Figure 6.2: Extensibility of OMS Java

Figure 6.3: Example of an application object
type student. In our example, student and employee are subtypes of type person. Since type systems of existing object-oriented programming languages such as Java or C++ do not allow an object to be associated to more than one subtype, we build application objects out of objects at the implementation level as is shown in figure 6.4.

![Figure 6.4: OM Objects](image)

An OM Object represents an application object at the conceptual level and has a unique identity during its whole lifetime. In the OM model, most algebra operations and constraints are evaluated only on the identity of an OM Object (see e.g. [Ste98]). For supporting model extensibility, it must be possible to extend the object identifier with additional data such as lifespans, which is interpreted by algebra operations, constraints, etc. We achieve this by modelling the identifier of an OM Object as a separate class ObjectID. Extended object identifiers can then be implemented as subtypes of this class. This is in contrast to those approaches which support specialist databases such as temporal databases by simply adding attributes (e.g. to store lifespans) to types.

An OM Object can reference any number of OM Instances. In our example, the OM Object 123 references instances of types person, student and employee. The system automatically associates the correct instance to the OM Object when accessing it through a specific context, i.e. through a collection. Each OM Instance is in turn specified by exactly one OM Type. An OM Type instance holds meta information needed by the system for evaluating algebra operations and constraints.

### 6.2.2 Core System

The core system manages all application objects, OM Objects, and all objects of the core data model OM such as OM Collections and OM Constraints. A client
application accesses OM Objects through the workspace by using its application programming interface (API). An OM Object can refer to one or more OM Instances of the following categories:

- **OM Collections** (including associations)
  An OM Object can be classified according to different criteria by inserting it into collections. Removing an object from a collection and inserting it into another one simply changes its role. So, if an OM object is stored in a collection then it gains automatically the member type of that collection.

- **OM Constraints**
  OM Constraints comprise the various structural constraints such as subcollection, cover, disjoint and intersection between collections, as well as the cardinality constraints for associations.

- **OM Triggers**
  OM Triggers define functions which are evaluated whenever the corresponding trigger condition holds.

- **OM SimpleInstances**
  This category represents the various types of application domain objects which can be associated to OM Objects. For example, the following DDL statement defines the type `person`:

  ```java
type person
  (
    name: String;
  );
```

  The following Java class corresponds to type `person`:

  ```java
public class Person extends OMSInstance {

  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

An instance of class `Person` represents a specific application object and can be created, for example, by the following DML statements:
create object p1;
dress p1 as person values
(
    name = "Person 1"
);

• **OM Types**
  OM Types specify meta information about the various OM Instances.

In OMS Java, these five categories are modelled as subcollections of collection **OM Instances** as depicted in figure 6.5.

![OM Instances Diagram](image)

Figure 6.5: OM Instances

Every OM Instance category provides a specific set of functions which can be used by the client application. For instance, an OM Collection supports all of the algebra operations by functions such as \texttt{union}, \texttt{flatten}, \texttt{map}, etc.

### 6.2.3 The Link between the Core System and the Configuration Component

The core system and the configuration component are linked together using the design pattern \textit{Factory} as depicted in figure 6.6. Design patterns provide solutions to problems which occur repeatedly [GHJV95]. All components of the system have been designed and implemented using the appropriate design pattern so that adapting the system to future requirements should be easily possible.
Figure 6.6: The link between the core system and the configuration component

The Factory design pattern is the main construct for system parameterisation and configuration. It defines a set of abstract parameters needed at run-time by an application or, in our case, the core system. Such parameters include, among others, the algebra used for operations over collections, the object identifier generator, the query language and data definition language parsers and the various data structures such as hash tables.

For example, when a client application creates a new collection using the API of the workspace, the core system first creates an OM Collection instance. It then calls a method of the factory to get an instance of the algebra as well as of a container. Such a container is used for storing the object identifiers of the members of a collection. By decoupling the data structure for storing the members from the OM Collection class which is part of the core system, it is possible to exchange this data structure without having to change the OM Collection class. Finally, the OM Collection instance is initialised with the algebra and container instance and returned to the client application. Figure 6.7 shows the link between an OM Collection and its configuration components.

All algebra operations are implemented in the Algebra class. They can be invoked through the OM Collection class. Hence, when a client application wants to perform an algebra operation such as union then it calls the method union of the OM Collection instance. This call is redirected to the union method of the algebra instance with which the OM Collection instance has been initialised. Since the Algebra class is part of the configuration component, the Algebra class might be exchanged.
without affecting the implementation of the OM Collection class, Container class or client application. Additionally, this approach makes it possible not only to override the algebra operations of the core model but also to provide new ones.

6.3 OMS Java Storage Management Component

The OMS Java storage management component is responsible for making those application objects persistent which survive the lifetime of an application process.

Looking at object-oriented frameworks and pure object-oriented applications, i.e. applications developed entirely using an object-oriented language environment such as Java [KA96], we can distinguish three requirements for persistent object management. First, application objects typically refer to many other application objects making it necessary to efficiently manage large collections of, often small, objects and complex object hierarchies. This is especially true in the case of distributed applications where objects are stored in different locations. For example, the OMS Java workspace alone consists of about 2500 persistent Java instances. To achieve efficient object management, our storage management component is based on only a few basic data structures such as hash tables and vectors.

Further, often large numbers of objects need to be processed together. This kind of bulk information processing is also traditionally performed by a DBMS [KDM97]. Finally, not all objects and object attributes need to be made persistent. There are volatile application objects such as graphical user interface components or temporarily created objects only needed during the lifetime of an application process.

Many approaches have been made to persistent object management and basically they address these requirements in one of two ways. Either application objects are mapped to database objects or tables within the application using, for example,
query language interfaces to DBMSs such as JDBC [Ree97], or the application objects can be stored in the database directly in which case no mapping is involved or is performed automatically by the persistent object management component.

For instance, the ODMG standard [CBB+00] defines interfaces and language bindings to object-oriented DBMSs such as ObjectStore (www.odi.com) and Objectivity/DB (www.objy.com) whereas JDBC [Ree97] allows Java applications to connect to relational DBMSs such as Oracle (www.oracle.com) for evaluating SQL statements. Sun's Java Blend product [BS98] is an example for supporting the runtime mapping of Java objects to the data model of a DBMS, and there are even proposals for extending the Java virtual machine [PAD*97].

Our object management framework **OMS Java** (Object Model System for Java) combines several of these approaches for achieving persistence. Additionally, we have designed the storage management component in such a way that it is possible to use various relational or object-oriented DBMSs for storage of the application objects. Hence, an application developer can use our framework for designing and implementing applications without having to deal with implementation aspects of storage management. We have investigated two approaches for building the storage management component of OMS Java: The **Classloader approach** presented in the next section, and the **Object Mapping** approach described in section 6.3.2.

### 6.3.1 Classloader Approach

With the Classloader approach [Est99], all Java classes are postprocessed at runtime to make them persistent-capable. Whenever a class is loaded into the Java virtual machine, it will be determined whether the instances of a class need to be persistent and if so the class will be postprocessed. Postprocessing classes adds DBMS specific parts to the byte code of a class as is shown in Figure 6.8.

```
original class                      persistence-capable class

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>post-processor</td>
</tr>
</tbody>
</table>
```

Figure 6.8: Postprocessor adding persistence-capability

Classes are only postprocessed once, and special proxy classes are used for accessing data structures such as hashtables and vectors. A proxy class "provides a surrogate or placeholder for another object to control access to it" [GHJV95]. Although postprocessing classes at runtime is straightforward and fast, there are two major disadvantages: First, not all DBMSs provide a mechanism for postprocessing Java classes which can be invoked by the classloader at runtime. Second, even if there is
a postprocessor mechanism available, it depends on the DBMS which Java classes can be made persistent. For instance, ObjectStore PSE Pro for Java does support postprocessing but not all Java system classes can be made persistent. To circumvent these problems, we have implemented another storage management framework based on the Object Mapping approach which we present in the following section.

6.3.2 Object Mapping Approach

The storage management component of OMS Java which is based on the object mapping approach is divided into two main parts connected together using the Java Remote Method Invocation mechanism (Java RMI) [Dow98] as shown in figure 6.9.

![Object Mapping Approach Diagram](image)

Figure 6.9: OMS Java Storage Management Component

The OMOManager resides on the client-side and is responsible for managing all application objects. Whenever an object changes its state, it notifies the object manager. The changes are then propagated over the network to the RMIOManager. Similarly, application objects are retrieved from the database through the RMIOManager. In both cases, the state of application objects are copied to/from state containers which can be regarded as snapshots of object data. Only these state containers are actually stored in the database. There are two types of state containers: one for representing the object identifier (OID) of an application object and one for holding attribute values as shown in figure 6.10.
Thus, every application object on the client-side is represented by one or more state container objects on the server-side. Two categories of attribute values can be stored in a state container object: base type values and object references. Base type values of a Java object such as `int` and `float`, together with their corresponding Java wrapper classes such as `Integer` and `Float`, can be directly copied to/from state containers. Object references need special treatment since references to application objects are typically represented by the memory address of the referred object which is only valid during the runtime of an application. To solve this problem, we create our own unique persistent OIDs, as is usual in persistent object systems [EN94], and keep track of which OID belongs to which state container making it possible to map object references to the corresponding persistent OIDs and vice versa.

Suppose, for example, a client application calls the `setName` method of the following Java class which causes the value of the attribute `name` to be changed – in our case to the value 'test'.

```java
public class DemoObj extends OMSInstance {

    String name;

    public void setName(String name) {
        this.name = name;
    }
}
```
OMWorkspace.notify(this, "name", name);
}

By invoking the notify method of the OMS Java workspace, the OMOBJECTMANAGER is notified about the value change giving the reference to the Java instance, the name of the attribute and the reference to the value as parameters. This is depicted as (1) in figure 6.11.

Figure 6.11: Updating an Attribute Value

The OMOBJECTMANAGER then retrieves through the RMIOBJECTMANAGER a reference to the RMIOBJECT and calls its updateObject method giving the OID of the object, the name of the attribute as well as value and value type as parameters (2).

The RMIOBJECT retrieves the state container object corresponding to the client object from the database using the OID as key, and calls the method updateAttribute of the state container object passing the name of the attribute as well as value and value type as parameters (3). The name of the attribute is then used to retrieve the state container object representing this attribute from the database. The value of this state container object is finally updated by calling the setValue method (4).

There are two other aspects to be taken into account when designing a client/server persistent object management framework: data structures for managing collections of objects and the number of simultaneously opened remote connections.

As mentioned before, it is usual for object-oriented applications to frequently perform operations on collections of objects. Since these collections can contain large numbers of object references, it is not a good idea to retrieve whole collections over
the network. The OMS Java workspace therefore uses the special proxy classes DBMap, DBVector and DBEnumeration for accessing collections of objects. In our case, the proxy classes refer to corresponding remote objects via Java RMI. Thus, all bulk information processing is done on the server-side. The OMS Java framework only uses maps and vectors for managing collections of objects, but it can be extended with new data structures as presented in section 6.4. A map is a data structure which maps unique keys to values and a vector is used for managing variable lists of objects.

Further, to keep network traffic efficient, only those remote connections shown in Figure 6.9 are open between a client and the server at the same time. Server objects can be accessed by their unique OIDs using one of these connections. For instance, suppose a client application wants to retrieve an object stored in a vector at position 5. Let us further assume that the OID of this vector is 134 and that there already exists a proxy object for this vector on the client-side. Thus, the OID of the proxy object which is in our example an instance of type DBVector will also be 134. Now, as illustrated in Figure 6.12, the client application invokes the method elementAt(5) of the proxy object DBVector (1) which in turn calls the method elementAt(5, 134) of the remote object RMIVector giving the position and the OID of the vector as parameters (2). The RMIVector object retrieves first the vector object from the database using its OID and then calls the elementAt(5) method of this vector object (3) which returns the OID of the element at position 5 to the RMIVector object (4). This OID is passed to the DBVector object (5) and used to retrieve the object by calling the OMOBJECTManager (6) which gets the object either from the object cache or from the database and returns it to the DBVector object (7). Finally, the object is forwarded to the client application (8).

![Diagram of retrieving an object from a vector](image)

Figure 6.12: Retrieving an Object from a Vector
Integrating a new DBMS as a storage platform requires that all six interface classes shown in Figure 6.9 be implemented using the application programming interface (API) of that DBMS. In most cases, a small number of additional classes need to be provided. We have integrated various relational and object-oriented DBMSs [KN99] and our experiences show that, typically, a total of about ten DBMS specific classes have to be developed for each integration. For example, Table 6.1 lists all classes we had to implement to integrate ObjectStore PSE Pro (www.odl.com) which is an ODMG-compliant persistent storage engine for the Java environment.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMIOBJECTMANAGER</td>
<td>PSEOBJECTMANAGER</td>
<td>Managing remote objects and connection to DBMS</td>
</tr>
<tr>
<td>RMIOBJECT</td>
<td>PSEOBJECT</td>
<td>Managing state container objects</td>
</tr>
<tr>
<td>RMIMAP</td>
<td>PSEMAP</td>
<td>Managing remote maps</td>
</tr>
<tr>
<td>RMIVECTOR</td>
<td>PSEVECTOR</td>
<td>Managing remote vectors</td>
</tr>
<tr>
<td>RMIENUMERATION</td>
<td>PSEEENUMERATION</td>
<td>Remote iterator for maps and vectors</td>
</tr>
<tr>
<td><strong>All of the following classes were adapted for ObjectStore PSE Pro</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMIATTRIBUTE</td>
<td>PSEATTRIBUTE</td>
<td>State container representing the OID of an application object</td>
</tr>
<tr>
<td></td>
<td>MapImpl</td>
<td>Persistent map</td>
</tr>
<tr>
<td></td>
<td>VectorImpl</td>
<td>Persistent vector</td>
</tr>
</tbody>
</table>

Table 6.1: Integration of ObjectStore PSE Pro

6.4 eXtreme Design for Extending the Storage Management Component

The OMS Java framework offers a rich set of complex data structures for composing application objects such as OM Collections and OM Associations and it is therefore usually not necessary to add new data structures. However, in the case of specialist application domains, it is possible to extend the framework with new data structures and this involves two main steps. The first step is to implement a corresponding proxy class for each new data structure. This proxy class can be used by the client application for remote access to data structure instances which are managed only on the server-side as mentioned in the previous section. The second step is to provide an implementation for the data structure. This can be done using one of two different approaches. The simplest way is to implement the data structure using the API of the target DBMS as well as to provide a remote object class which can be used by the proxy class to access the data structure.
We have applied this approach for extending the storage management component of OMS Java with data structures vector and map which are used by the OMS Java framework. The corresponding proxy classes are in this case DBVector and DBMap which are linked to remote objects of class RMIVector and RMIMap via Java RMI as shown in Figure 6.9.

![Integration of Map Data Structure](image)

The following example sketches the implementations necessary for supporting a map data structure for the DBMS ObjectStore PSE Pro. First, we had to implement the proxy class DBMap which can be used by the client application to access the map as shown in Figure 6.13. Next, we developed the class PSEMap which implements the interface RMIMap. The PSEMap class must be capable of managing map instances stored in the ObjectStore PSE Pro DBMS. Finally, we used the API of ObjectStore PSE Pro to implement our PSE version of a map using the persistent hashtable OSHashtable as indicated:

```java
class MapImpl
{
    protected OSHashtable table; // reference to ObjectStore PSE hashtable
    ...

    public Enumeration elements()
    {
        return table.elements();
    }

    public void put(Object key, Object value)
    {
        table.put(key, value);
    }
    ...
}
```

Access to elements of a map through a DBMap proxy instance is given in the same way as illustrated by the example shown in figure 6.12. So, an object of the client
application can be linked to a map instance on the server-side by a reference to a proxy instance representing this map instance.

The disadvantage of using the API of a DBMS for implementing data structures is that it becomes necessary to provide implementations for every DBMS. For example, if you want to introduce a new data structure and your OMS Java server supports links to the DBMSs ObjectStore, Objectivity/DB and Oracle, then you have to implement the data structure three times using the corresponding API or, in the case of Oracle, SQL statements.

To avoid this, we can use the XD approach for specifying bulk data structures as abstract data types. An abstract data type can be regarded as a data structure which is described in terms of the operations performed, rather than in terms of details of implementation [Sed88].

Applying the XD approach for implementing bulk data structures involves mainly two steps: defining the data structure together with its operations which is described in the next section, and implementing these operations using the API of the XD framework which can be carried out in two ways. In Section 6.4.2 we illustrate how to implement the operations in such a way that they are evaluated on the client-side, whereas the approach taken in section 6.4.3 executes operations on the server-side.

### 6.4.1 Defining Bulk Data Structures

A data structure can best be modelled using an existing notation such as OM or UML [PS99]. For instance, figure 6.14 gives an example of a definition for a map data structure in terms of OM.

The collection Maps can be regarded as the main concept of the map specification. It is good practice to use meta objects for representing main concepts as indicated by the arrow from Maps to Meta Object. For all other concepts we may use a combination of attribute and value objects which are related to the meta object.

Figure 6.15 gives an example for a map representation. Suppose that we created a map with object identifier (OID) “123” and two key/value pairs: the key “Persons” maps to the value “178” and the key “Students” maps to the value “540”. So, the map is represented by a meta object with the same OID “123”. This meta object is specified by one attribute object “keys” according to the specification of figure 6.14. This attribute is defined by two value objects: one for the key “Persons” and one for the key “Students”. The value object “Persons” refers to another value object representing the value “178”, and the value object “Students” refers to the value object “540”. We can retrieve, for instance, the value associated with the key “Students” using the match operator:

```java
XDIterator enum =
    metaObject.match("123", "keys", "Students");
```
Figure 6.14: Specification of a Map
The match operator takes as input parameters the OID of the meta object, the attribute name and a value. It retrieves first all value objects associated with the attribute "keys", selects then the value object which is equal to the value input parameter "Students" and returns finally the value object which is referred by the "Students" value object. Another example of the match operator is given in the next section.

![Diagram of a Map]

Figure 6.15: Example of a Map

The operations of the data structure can be specified, for example, by descriptions of method signatures as shown in Table 6.2. Since our target system evaluates methods written in Java, we define these signatures also in terms of Java method signatures.

6.4.2 Implementing Operations for Client-Side Processing

By client-side processing we mean that operations over bulk data structures are evaluated on the client-side. The advantage of this approach is that the evaluation process does not overstrain the server. On the other hand, more data has to be retrieved over the network than is the case with server-side processing.

For the implementation of the map bulk data structure specified in the previous section, we use the API of the XD framework which can be accessed through a remote object of type XDMetaObject. Only one such remote object exists on the server-side, thereby keeping network traffic efficient. All operations for managing meta objects are supported by corresponding XDMetaObject methods taking the OID of the meta object as first parameter. This OID is needed to retrieve the meta object from the database. Hence, a client application must initialise a bulk data structure implementation class with the OID of the corresponding meta object.
Table 6.2: Method Signatures of a Map

<table>
<thead>
<tr>
<th>Signature</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear()</td>
<td>Removes all elements in a map</td>
</tr>
<tr>
<td>boolean containsKey(Object key)</td>
<td>Returns true if parameter 'key' is a key of this map</td>
</tr>
<tr>
<td>Enumeration elements()</td>
<td>Returns an enumeration of all elements</td>
</tr>
<tr>
<td>Object get(Object key)</td>
<td>Returns the element associated with 'key'</td>
</tr>
<tr>
<td>Enumeration keys()</td>
<td>Returns an enumeration of all keys</td>
</tr>
<tr>
<td>void put(Object key, Object element)</td>
<td>Associates 'key' to 'element'</td>
</tr>
<tr>
<td>void remove(Object key)</td>
<td>Removes the element associated with 'key'</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of elements stored in the map</td>
</tr>
</tbody>
</table>

and with a reference to the XDMetaObject. This can be done, for example, in the constructor of the class:

```java
public class MapImpl {
    protected XDMetaObject metaObject;
    protected String OID;

    public MapImpl(XDMetaObject metaObject, String OID, boolean create) {
        this.metaObject = metaObject;
        this.OID = OID;
        if (create) createMetaObject();
    }
    ...
}
```

The create parameter indicates whether a new meta object should be created on the server-side. In our example, calling the method createMetaObject creates a new meta object representing a map together with the attribute keys.

```java
public void createMetaObject() {
    // creating a new meta object
    metaObject.createObject(OID);
    // creating attribute 'keys'
}
metaObject.createAttribute(OID, "keys");
}

The following examples illustrate implementations of some MapImpl methods:

class MapImpl {
 ...

  public boolean containsKey(Object key) {
    return (get(key) != null);
  }

  public Object get(Object key) {
    /*
     * the 'match' operator returns an enumeration of all values
     * matching a specific attribute/value entity pair.
     * In our case the attribute is "keys" and the value entity
     * is specified by the 'key' parameter.
     * Hence, the enumeration covers all those value entities which
     * are related to the 'key' parameter by the association 'refer_to'
     * between value entities. And since there is at most one
     * key/element pair in the map, this enumeration contains either
     * zero or one value entity.
     */
    XDIterator enum = metaObject.match(OID, "keys", key);
    if (enum.hasMoreElements())
      return enum.nextElement();
    else
      return null;
  }

  public void put(Object key, Object value) {
    /*
     * removing all value entities which are equal to parameter 'key'
     * and which are associated with attribute 'keys';
     * thus, old key/element pairs are removed from the map
     */
    metaObject.retract(OID, "keys", key);

    // creating a new key/element pair
    metaObject.assert(OID, "keys", key, value);
  }

  ...
}
6.4.3 Implementing Operations for Server-Side Processing

If it is for some reason not possible to perform bulk information processing on the client-side, for example if the client application has only few resources at its disposal, then we have to use proxy classes for accessing and evaluating operations on the server-side. The operations themselves are defined as methods of the meta object representing the corresponding bulk data structure on the server-side. Figure 6.16 illustrates this situation for our map example.

![Integration of Map Data Structure Using the Meta Model](image)

The first step is therefore to implement proxy classes for each bulk data structure represented by a meta object on the server-side. Such a proxy class must provide methods for all operations over a specific bulk data structure, and can be used by client applications for evaluating those operations.

Thus, by invoking a method of a proxy class, the corresponding method on the server-side is evaluated. A proxy class accesses the meta object and its method object through a remote object of type XDMetaObject giving the OID of the meta object as parameter. This OID is needed to retrieve the meta object from the database. Hence, a client application must initialise a proxy class with the OID of the corresponding meta object and with a reference to the XDMetaObject. This can be done, for example, in the constructor of a proxy class:

```java
public class MapProxy
{
    protected XDMetaObject metaObject;
    protected String OID;

    public MapProxy(XDMetaObject metaObject, String OID, boolean create)
    {
```
The create parameter indicates whether a new meta object should be created on the server-side. In our example, calling the method createMetaObject creates a new meta object representing a map together with its attributes and methods.

```java
public void createMetaObject()
{
    // creating a new meta object
    metaObject.createObject(OID);

    /*
     * creating an association between the meta object and
     * the method object which implements 'createTable' and which is
     * specified in the class 'MapMethods'
     */
    metaObject.createMethod(OID, "createTable", "MapMethods");

    /*
     * The following statements show how to evaluate a method of
     * a meta object:
     * First, an array of all method parameters has to be created
     * which then is passed to 'execMethod' together with the OID
     * of the meta object and the method name.
     * 'createTable' is a method evaluated on the server-side creating
     * all necessary attributes and methods for a map data structure
     */
    Object[] args = new Object[0];
    metaObject.execMethod(OID, "createTable", args);
}
```

The following examples of map methods of the map proxy class illustrate how methods of a meta object can be remotely invoked. As in the case of the createMetaObject method, the OID of the meta object, the method name and an array of method parameters are passed to the execMethod of the XDMetaObject which then evaluates the method on the server-side.

```java
// true, if 'key' is a key of the map
public boolean containsKey(Object key) {
    
```
Evaluation of a meta object method is done the following way on the server-side: First, the method object associated with the meta object is retrieved from the method cache. If it is not in the cache then a Java instance representing the method object must be loaded into the cache using information stored in the database. Suppose, for instance, that all operations of our example have been implemented by methods of the Java class MapMethods. Instances of this class are associated to all meta objects representing a map data structure. So, if the method instance is not yet loaded into the cache, this name is used for creating the instance by the statement Class.forName("MapMethods").newInstance(). Finally, the method is invoked by its name using the reflection API of Java [Sun].

The following example illustrates implementations of some methods of class MapMethods. Note that the createTable() method creates all attributes and methods needed for implementing a map data structure. createTable() is called every time when the client application creates a new map through the map proxy class.

public class MapMethods
{
    String OID;
    XDMetaObject metaObject; // reference to the meta object
    // associated with this method object

    public void createTable()
    {
        // creating attribute 'keys' of the map
        metaObject.createAttribute(OID, "keys");
        */
creating all operations of the map; the parameter 'MapMethods' is used for creating an instance of this class when loaded into the cache
*/
metaObject.createMethod(OID, "clear", "MapMethods");
metaObject.createMethod(OID, "containsKey", "MapMethods");
metaObject.createMethod(OID, "elements", "MapMethods");
metaObject.createMethod(OID, "get", "MapMethods");
metaObject.createMethod(OID, "keys", "MapMethods");
metaObject.createMethod(OID, "put", "MapMethods");
metaObject.createMethod(OID, "remove", "MapMethods");
metaObject.createMethod(OID, "size", "MapMethods");
}

public boolean containsKey(Object key)
{
    return (get(key) != null);
}

public Object get(Object key)
{
    /*
     * the next statement returns an iterator to all values associated with
     * attribute 'keys' matching parameter 'key';
     * this value object represents the key of a key/element pair in the map;
     */
    XDIterator enum = metaObject.match(OID, "keys", key);

    /*
     * if there is a key object then it is returned; otherwise 'null' is returned
     */
    if (enum.hasMoreElements())
        return enum.nextElement();
    else
        return null;
}

public void put(Object key, Object value)
{
    /*
     * removing all value objects which are equal to parameter 'key'
     * and which are associated with attribute 'keys';
     * hence, old key/element pairs are removed from the map
     */
    metaObject.retract(OID, "keys", key);
6.4.4 Performance Considerations

One characteristic of object-oriented frameworks and object management systems is that there are a lot of small objects referring to one another, and a lot of special data structures such as maps for managing collections of objects. For example, all objects of the OMS Java workspace are mapped to 2586 database objects of which 108 are maps, 279 are lists, 679 state container objects representing Java instances and 1520 state container objects representing attribute values or references to other objects.

We made measurements for five operations: creating a new OMS Java workspace, loading an example schema, importing data, opening a workspace and retrieving an OM object which, in our example, consists of 12 lists, 5 maps, 38 state container objects representing Java instances and 85 state container objects representing attribute values or references to other objects. Table 6.3 lists the number of created database objects for the first three operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Maps</th>
<th>Lists</th>
<th>Object Containers</th>
<th>Attribute Containers</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>creating workspace</td>
<td>108</td>
<td>279</td>
<td>679</td>
<td>1520</td>
<td>2586</td>
</tr>
<tr>
<td>loading schema</td>
<td>79</td>
<td>199</td>
<td>499</td>
<td>2679</td>
<td>1936</td>
</tr>
<tr>
<td>importing data</td>
<td>269</td>
<td>638</td>
<td>2014</td>
<td>4690</td>
<td>7611</td>
</tr>
</tbody>
</table>

Table 6.3: Number of created database objects

We compared the classloader approach with the object mapping approach. The results are shown in table 6.4. In the column labelled “Classloader” are the results for the classloader approach. “PSE” denotes the results for mapping Java instances to the persistent object engine Objectstore PSE Pro for Java (www.odi.com) as an example for an object-oriented system. This system was also used with the classloader approach. “JDBC” denotes mapping results to a relational DBMS using JDBC, in our case to Oracle 8 (www.oracle.com). The numbers indicate how much longer or slower an operation was carried out compared to the classloader approach. For example, “2.0” means that this operation was slower by a factor of 2.

The results indicate two aspects. One aspect is that mapping Java instances to state containers using an object-oriented DBMS is only about 1.5 times slower than directly storing Java instances into the database. Another aspect is that mapping Java instances to relational DBMSs needs much more time when creating objects, but is not that much slower when retrieving objects compared to object-oriented
DBMSs. This is because we used indices on certain attributes such as the object identifier whereby making object retrieval about two times faster than without indices. Note that some performance loss is due to the fact that Objectstore PSE Pro for Java is a single-user storage engine optimised for the Java environment, whereas Oracle is a multi-user DBMS optimised for a large amount of data. Additionally, the JDBC measurements also include network traffic since there is a network connection between the JDBC driver and the Oracle DBMS. To optimise this connection, the framework uses prepared statements and minimises the amount of opened cursors.

Further, our experiences indicate that it is very important to have a good cache strategy on the client-side as well as on the server-side and that the DBMS specific classes must be optimised carefully. It is also crucial that, in a multi-tier architecture, the number of references to remote objects should be kept minimal for performance reasons. Since there are only a few remote connections open at the same time between a client and a server in our framework, performance loss due to network traffic can be kept minimal. Operation performance over the network is about 3.5 times slower.

The OMS Java framework uses maps and vectors for bulk information processing. We implemented these data structures using the XD meta model and changed the storage management component of OMS Java to use these implementations. Although OMS Java makes heavy use of maps and vectors, the version with the XD implementations of maps and vectors is only about 2.5 times slower compared with the one using data structures provided by Objectstore PSE Pro.

Overall we can state that the object mapping approach is suitable for persistent object management in terms of performance and flexibility, and also that a relational DBMS can be used for managing Java instances quite efficiently. Further, the XD meta model is very useful for implementing data structures for bulk information processing.
Chapter 7

Rapid Information Modelling

The term *information* denotes the transmission of signals in the perspective of the engineering domain, but mostly we interpret information as "something that has meaning — that is, it must make sense or have significance [McA95]." Hence, we understand *information modelling* as the activity of classifying and structuring information about a specific domain on a conceptual level, i.e. the organisation of information is such that people are able to gain value from their point of interest.

But what does *their point of interest* mean in this context? "Typically, the hard or insoluble problems occur at the conceptual level." People simply disagree on how data should be modelled even when they are using the same data model [ABS00]. So, the question is how to cope with the different information needs of people for the same information domain or information space.

Rapid information modelling (RIM) supports the construction of information spaces on the conceptual level for various information needs without the necessity of defining a data model for information organisation, hence the term rapid. This is achieved by using the XD approach for classifying and structuring information, and by providing additional operations to which we refer collectively as the RIM algebra.

In section 7.1, we introduce the concept of *information spaces*. In section 7.2, we present a framework which can be used for implementing RIM applications, i.e. applications managing information spaces, and an example of an RIM application is explained in section 7.3.

7.1 Information Spaces

Information spaces can be defined as environments into which users enter to find answers to questions about a specific topic, to browse large collections of information, and to update and reorganise information. Thus, to be of value, an information space must not only provide ready access to information, but also be well organised and
comfortably navigable. "A well-designed information space is one that (like any well-designed environment) is consistent, predictable, and transparent [McA95]."

An information space can also be regarded as a group memory based on a common repository of on-line accessible and structured information which provides value to a group of people. "A group memory must be appropriate to its context — the heterogeneity, stability, computer sophistication, goals, and social environment of its users [BJO+93]."

There are many ways to organise and structure information in an information space. For example, we can organise information according to user defined categories, location (e.g. maps), continuum (e.g. from largest to smallest), and chronology. "Ways of organizing information are also ways of understanding" [McA95].

Rapid Information Modelling (RIM) addresses all of these issues by being a framework of thought from the user's point of view. An RIM application manages information spaces for users facilitating the organisation and navigation of information. A user can conveniently represent concepts of a specific domain as information objects, create categories for classifying such information objects, build relationships between them, and add properties to them. All these activities can be carried out without the need for a user to know about information or data modelling. It is also not necessary to change the RIM application, for example, when new categories or properties are introduced.

A developer can use the API of the RIM framework for implementing sophisticated RIM applications very easily. Further, it is possible to use a variety of database management systems (DBMS) for storing data since the RIM framework is based on the XD framework which is independent of a particular DBMS.

7.2 RIM Framework

Classification of user domain objects requires not only the ability to group objects by inserting them into named object groups, but also the possibility to perform operations over object groups. For example, if we want to find out which objects are in object group Household and Office, we perform a union operation over these two groups the result of which is returned in a new object group. Classification is an important activity of rapid information modelling. Further, RIM requires that relationships between objects as well as object properties can be specified on an "ad hoc" basis, i.e. by being able to add and change relationships and properties without having to change schema information or type specifications.

Since the XD meta model defines concepts supporting these requirements on a meta level, and since the XD algebra provides a rich set of operations over object groups, object relationships and object properties as presented in section 3.2, we claim that the XD meta model and its algebra is a good basis for implementing RIM
applications.

In the context of the XD meta model, types are specified in terms of type views as described in section 3.1.4. As the name suggests, a type view defines which object properties are important in a specific application context. For example, if we want to express the fact that in our application context a book object is characterised by the properties ISBN, title and author, we can create a type view bookType defining these properties and use it, for instance, for creating meta objects representing books.

A type view is not the same as a type, since it is not a condition that an object “dressed by” a type view must have all properties of that view, whereas an object belonging to a type must at least have the properties of that type. Further, type views are not taken into consideration while evaluating XD algebra operations.

The XD meta model and its algebra can easily be extended with new concepts. For example, the extensions proposed in [Ign00] introduce the concept of heterogeneous collections and the notion of typing. A heterogeneous collection is an object group containing objects sharing some common features, but not all object properties are the same. For instance, the resulting collection of a union operation between collections Chairs and Lamps contains members representing chair and lamp objects. Common properties of these two object categories might be product name and product no, but otherwise an object possesses either chair or lamp characteristics.

Thus, additional concepts are required such as object types denoting common properties of objects and an algebra for heterogeneous collections. Such an algebra is proposed in [Ign00] called the RIM algebra which essentially extends the OM algebra by taking into account whether the operations have to be evaluated with or without type checking.

For example, suppose the object group Household contains the objects chair1 and lamp1, and the group Office the objects chair2 and lamp2. The result of a union operation over these two groups without type checking is a group containing the objects chair1, lamp1, chair2 and lamp2 as its members. Performing the same operation with type checking means that only those members are taken into consideration which are of a specific type. So, Household union(chairType) Office returns a group containing only the members chair1 and chair2 assuming that those objects are of type chairType, i.e. have properties specified by the type view chairType.

The RIM algebra is supported by the RIM framework which also provides functions for managing collections, binary collections and types [Ign00]. This framework is based on the XD framework and can be used for developing RIM applications.

An RIM application typically consists of two components: a user interaction component and an application logic component. In the case of an RIM application implemented using web technologies, the user interaction component might be spec-
ified by HTML pages so that it is possible to use the application through any web browser. In this scenario, the application logic component resides on the server-side, for instance, defined as Java Servlets which use the RIM framework for performing rapid information modelling. A servlet mainly “accepts requests from a client (via the Web server), performs some action, and returns the results” [Mos98].

In summary, we regard an RIM application as an application managing information spaces through the RIM framework or XD framework and providing user interaction through a standard web browser. We give an example of such an application in the next section. This example uses the XD framework and illustrates how to use servlets for implementing the application logic.

### 7.3 RIM Application Example

In this section, we present a simple application as an example for an RIM application. The application provides an interface for entering and evaluating query and DML statements. The logic of the application is defined within a Java servlet [Mos98] which uses the XD system for managing application objects. The servlet is connected to the XD system by Java RMI [Dow98].

The user interaction component for our example consists of only one file containing the following HTML specifications:

```html
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
  <title>XD Servlet</title>
</head>

<body>
  <h1>XD Servlet</h1>

  <form action="/servlet/XDServlet"
    enctype="x-www-form-encoded" method="POST">
    <b>1. Query or DML statement: </b>
    <textarea name="query" cols=50 rows=10></textarea>
    </form>
    <br>
    <b>2. Submit query or reset form: </b>
    <input name="ql" type="submit" value="Query">
    <input name="dml" type="submit" value="DML">
    <input name="clear" type="reset" value="Clear this Form">
    <input name="serverURL" type="hidden" value="/host/MLS_SERVER">

</body>
</html>
```
Loading this file by a web browser, creates the user interface shown in figure 7.1. The value of serverURL in the example specifies the name with which the XD system is registered at the Java RMI registry service. By this name, the servlet can establish a connection to the XD system.

Figure 7.1: Interaction Component for the Example

In the text box, a user can enter either query or DML statements and let them be evaluated by activating the appropriate button. The query members( Customers ), for example, retrieves all objects which are members of the object group Customers. For the book store example introduced in section 4.1, this query produces the list shown in figure 7.2.

Figure 7.2: Result of Query members( Customers )

Activation of the Query button causes the servlet method service to be invoked by the web server. This method establishes first a connection to the XD system:

```java
public void service(HttpServletRequest req,
```
HttpServletResponse res)
throws IOException
{
...

res.setContentType("text/html");
out = res.getOutputStream();
out.println("<HEAD><TITLE> XDServlet output " +
"</TITLE></HEAD><BODY>");

try {
...

// retrieving a remote reference to the XD server
server = (XDServer)Naming.lookup(serverURL);

// retrieving a remote reference to the XD system
metaObject = server.metaObject(null);
...

} catch (Exception e) {
out.println("<h1> Error while evaluating XD query </h1>");
out.println(convertNL(e.toString()));
}

out.println("</BODY>");
}

If the user activates the DML button then the servlet method dml is called which forwards the DML statements to be evaluated by the XD system:

public void dml(String dmlStatements) throws IOException
{
try {

XDResult xdRes = server.dml(dmlStatements, true);

if (!xdRes.successful()) {
out.println
("<h1> Error while evaluating XD query </h1>");
out.println
(convertNL(xdRes.msg()));
} else
out.println("DML: done.");
} catch (Exception e) {
out.println("<h1> Error while evaluating XD query </h1>");
}
On the other hand, activating the Query button calls the servlet method `query` which first invokes the corresponding method of the XD system:

```java
public void query(String query) throws IOException {
    try {
        XDResult xdRes = server.query(query, true);
        if (!xdRes.successful()) {
            out.println
                ("<h1> Error while evaluating XD query </h1>");
            out.println(convertNL(xdRes.msg()));
            return;
        }
    } catch (Exception e) {
        out.println("<h1> Error while evaluating XD query </h1>");
        out.println(convertNL(e.toString()));
    }
}
```

The result of the query is then processed by iterating over the members of the returned result group:

```java
XDIterator elms = (XDIterator)result;
// the iterator types OIDS and MEMBERS indicate
// that the elements returned by the iterator
// represent object identifiers
if (((elms enumType() == XDIterator.OIDS) ||
    (elms enumType() == XDIterator.MEMBERS)) {
    createOIDTable(elms, null);
} else {
    while (elms.hasMoreElements()) {
        out.println(elms.nextElement().toString());
        out.println("<BR>");
    }
```

122
The `createOIDTable` method creates the HTML page which is returned to the www browser and presented to the user as shown in figure 7.2:

```java
public void createOIDTable(XDIterator oids, String title)
    throws IOException
{
    try {
        out.println("<TABLE BORDER>");
        if (title != null)
            out.println("<CAPTION> " + title + " </CAPTION>");
        out.println("<TR> <th> OID </th> <th> Alias </th> </TR>");
        while (oids.hasMoreElements()) {
            String OID = (String)oids.nextElement();
            String alias = metaObject.retrieveAlias(OID);
            if (alias == null) alias = "";
            out.println("<TR> <td> <A HREF="" + servletURL + "?oid=" +
                        OID + "&serverURL=" + serverURL + "&servletURL=" +
                        servletURL + "" + OID + "</A>" + " </td> " +
                        "<td> " + alias + " </td> </TR>");
        }
        out.println("</TABLE>");
    } catch (Exception e) {
        out.println("<h1> Error while evaluating XD query </h1>");
        out.println(convertNL(e.toString()));
    }
}
```

The user can click on an object identifier which causes the servlet method `showObject` to be evaluated:

```java
public void showObject(String OID) throws IOException
{
    if (OID == null) return;
    try {
        out.println("<h1> Object " + OID + " </h1>");
    }
}
```
String alias = metaObject.retrieveAlias(OID);
if (alias != null)
    out.println("Alias = " + alias + " <BR>");

createOIDTable(metaObject.members(OID), "Members");
createOIDTable(metaObject.associations(OID), "Associations");

XDIterator attributes = metaObject.attributes(OID);
if (!attributes.hasMoreElements()) return;

out.println("<TABLE BORDER>");
out.println("<TR> <th> Attribute </th> <th> Value(s) </th> </TR>");

while (attributes.hasMoreElements()) {
    String attrName = (String)attributes.nextElement();
    boolean firstVal = true;
    out.println("<TR> <td> " + attrName + " </td> <td> ");

    for (XDIterator values = metaObject.match(OID, attrName);
        values.hasMoreElements(); ) {

        Object value = values.nextElement();

        if (firstVal) {
            firstVal = false;
            out.println(value.toString());
        } else {
            out.println(" <BR> " + value.toString());
        }
    }
    out.println(" </td> </TR> ");
}

out.println("</TABLE>");
catch (Exception e) {
    out.println("<h1> Error while evaluating XD query </h1>");
    out.println(convertNL(e.toString()));
}

So, if the user clicks on the object identifier XD_50 referring to the object customer1, the servlet retrieves the corresponding object and returns a HTML page as shown in figure 7.3.

In our example, customer1 placed three orders. The user can get more information about an order by clicking on the corresponding object identifier. For instance, clicking on XD_102 retrieves the order object shown in figure 7.4, and clicking on
XD_102 retrieves the book object which has been ordered by customer1 as shown in figure 7.5.

A user can create new objects by entering DML statements and activating the DML button on the XD Servlet main page. For instance, creating a new book object can be carried out by the DML statements shown in figure 7.6 the result of which is shown in figure 7.7.

With this simple example we wanted to illustrate how easy it is to build an RIM application using the XD system API for accessing and managing application objects.

In our example, the user must enter DML statements for creating and updating
Figure 7.6: Creating a new Book Object

Figure 7.7: Object book3
objects. An improvement would be to provide a special input form for managing objects such as depicted in figure 7.8. The user can change attribute values simply by entering new values in the corresponding text field. New attributes and their values can be created by specifying them as attribute = value statements in the text box. Activating the update button makes changes permanent.

![EDIT

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
</tr>
<tr>
<td>Author</td>
</tr>
<tr>
<td>ISBN</td>
</tr>
</tbody>
</table>

| price = 12.6 |

2. Action: Update

Figure 7.8: Input Form for Managing Objects

Another aspect shown with this example is the fact that the XD system can be accessed remotely through Java RMI making it possible that more than one application can be connected to the same XD system. Hence, RIM applications can concurrently access application objects managed by the XD system.

Finally, we claim that the RIM framework together with the flexibility of the XD meta model and XD system build a powerful basis for developing applications supporting users shaping their own information spaces as well as for analysing these information spaces in terms of classification, relationships and structure of domain specific objects.
Chapter 8

Conclusions

Building and maintaining software systems and components is a complex task demanding a clear understanding of all involved processes. Software engineering techniques address this by specifying guidelines for managing the software life cycle. The emphasis is on providing guidelines since it has become evident that it is not possible to define strict rules for developing and deploying software systems. Rather, it is more and more the practice to combine a variety of software engineering techniques to achieve a given goal.

In this thesis, we focused on the design process of software systems and components. The proposed eXtreme Design approach combines aspects of conceptual modelling and object-oriented concepts on a high level of abstraction. This makes it possible to use eXtreme Design in the area of prototyping, component construction and information modelling especially in cases where it is necessary to build highly adaptable software systems in terms of runtime environments and future requirements.

The eXtreme Design meta model together with its algebra complement existing software design and modelling techniques in a way that facilitates not only the design process but also development.

For example, in the area of object-oriented software construction there is a trend towards applying design patterns [GHJV95] together with standardised modelling techniques such as UML [PS99] for coping with the increasing complexity of software systems. The focus is on developing reusable and extensible software components which are building blocks for applications and other components. Component development involves activities such as [JTM00]

- Design of reusable software components, based on the generalisation of more specific ones,
- Validation and assembly of components that might come from many different sources, and
Management and maintenance of the components.

Although existing software engineering techniques are suitable for designing and developing component-oriented systems, it is not always possible to foresee all future requirements and adaptation needs. Even by using design patterns and existing software components it is often the case that a software system needs to be redesigned and refactored. Refactoring can be regarded as restructuring software without changing its observable behaviour, i.e. changing “the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” [Fow99].

Redesigning existing software in most cases makes it necessary to change source code, schema specifications and interfaces which is expensive in terms of time, costs and quality assurance. Using eXtreme Design and its meta model for developing critical components or for extending existing ones, leads to more extensible and maintainable systems since classification structures, object relationships and properties of application objects can evolve over time without having to redefine, for example, schema information.

Another important aspect of eXtreme Design is supporting rapid information modelling. Information systems of today have to be very adaptable to new information needs for a variety of users. Further, an information system must be capable of integrating existing data sources. The eXtreme Design meta model and its algebra are a good basis for developing those components of an information system which need to be extensible and changeable without the cost of schema evolution and application redesign.

8.1 The eXtreme Design Approach

In this thesis we have presented the eXtreme Design Approach (XD) and have shown that XD can be applied for three different design activities: Prototyping, component construction and information modelling. Since the XD meta model and its algebra combine aspects of conceptual modelling and object-oriented concepts on a meta level, it becomes possible to specify properties of application objects, classification structures as well as object relationships on a high level of abstraction.

Thus, application and information system components based on the XD meta model and its algebra are very extensible and adaptable to future requirements, and changing efforts are kept minimal because of the fact that classification structures, object relationships and properties can evolve over time without having to change schema information or existing application components.

Further, since the eXtreme Design meta model and its algebra is based on the generic object model OM, the model can be extended and adapted to specific application requirements very easily. In addition, an application developer can use both, the
XD meta model and OM model together with their algebras for developing highly adaptable software components.

8.2 Future Work

As a result of using the eXtreme Design approach for developing the software components presented in this thesis, it became evident that the following topics are a good starting point for further work on the approach and its meta model:

- **Method integration**
  Although the eXtreme Design approach is a supplement to existing software engineering techniques and it is therefore possible to use those methods and notations for the design process, it might help to define guidelines of how to apply eXtreme design in the context of a specific design method, or even to integrate eXtreme design into an existing one.

- **Model extensions**
  Especially in the context of rapid information modelling it is important that the model supports different kinds of object groups such as sets and bags, and that the algebra takes into account the notion of types as proposed in [Ign00]. Further, the meaning of concepts such as classes, inheritance and polymorphism need to be specified for the XD meta model which is necessary, for example, in the context of method evaluation and also when deriving schema and type information.

- **Languages**
  The eXtreme Design meta language is currently based on Oberon-0 which is simple to learn and not hard to integrate into other language and runtime environments such as Java. The next step in this direction would be to investigate other languages as basis for the meta language.

  Other open issues are to define a query language based on the eXtreme Design algebra, and to specify a data definition language which can be used for defining types, object groups and associations as is the case of the OM data definition language.

- **Framework**
  The framework could be extended in various directions. For example, the application programming interface of components used by client applications such as the workspace or meta object interfaces could be made compliant to existing component frameworks such as Enterprise JavaBeans. This way, it would be very easy to integrate meta objects into existing applications.

  Another important topic is data integration. The storage engine of the framework could be extended in such a way that it would be possible to obtain
and integrate data from other data sources. So a client application could, for instance, use data from a OMS Java database together with data of the meta model.

Further, the framework should provide tools for data analysis making it possible, for example, to create data definition and data manipulation language statements for other database management systems, or to analyse the relationships between meta objects.

Last but not least, it would be very interesting to use the eXtreme Design meta model for defining and implementing other data models such as the generic object model OM or the relational data model. If this turns out to be possible then it should also be possible to create a database management system which supports more than one data model.
Appendix A

Data Manipulation Language (DML)

DML = CreateStmt | DressStmt | StripStmt | DeleteStmt
    InsertStmt | RemoveStmt | AssociationStmt | FactStmt | AssertStmt | RetractStmt.

COMMENT = "//" <comment_until_eol> .

CreateStmt = "create" ["object"] ObjectAlias ";" .

DressStmt = "dress" ["object"] ObjectAlias "as" TypeName
            ["values" "(" {AttrAssign} ")"] ";" .

AttrAssign = AttrName "=" AttrVal ";" .

AttrVal = UniValue | MultiValue .

StripStmt = "strip" ["object"] ObjectAlias ["to" | "off"]
            TypeName ";" .

DeleteStmt = "delete" (["object"] | "group" ObjectAlias) |
            ("association"
            "(" ObjectAlias "," ObjectAlias ")") |
            ("fact"
            "(" ObjectAlias "," ObjectAlias ")"
            "," ObjectAlias ")") ";" .

UniValue = UniUnary | UniBinary .
UniUnary = BaseValue | ["object"] ObjectAlias .
UniBinary = "(" UniValue "," UniValue ")" .
MultiValue = [" [UniValue {"," UniValue} "]" .
ObjectAlias = StringO | """" String """" .
Basevalue = Integer | Boolean | ...
| StringO | """" String """" .
TypeName = StringO | """" String """" .
AttrName = StringO | """" String """" .
StringO = Alpha {AlphaNum} .
AlphaNum = Alpha | Number .
Alpha = "a"-"Z" | "_" .
Number = "0"-"9" .

InsertStmt = "insert" "into" ObjectAlias ":" MultiValue .
RemoveStmt = "remove" "from" ObjectAlias ":" MultiValue .
AssociationStmt = "association" "(" ObjectAlias "," ObjectAlias ")" .
FactStmt = "fact" "(" ObjectAlias "," ObjectAlias ")" 
"(" ObjectAlias ")" .
AssertStmt = "assert" "(" ObjectAlias "," AttrName "," MultiValue 
[ "," MultiValue ] ")" .
RetractStmt = "retract" "(" ObjectAlias [ "," AttrName 
Appendix B

XD Meta Language (XDML)

B.1 XD Meta Language Syntax

module = "MODULE" ident ";" declarations
[ "BEGIN" StatementSequence ] "END" ident ".".

declarations = [ "SIGNATURE" { ProcedureDeclaration } ]
[ "CONST" { ident "=" expression ";" } ]
[ "VAR" { IdentList ":=" type ";" } ].

ProcedureDeclaration = ident "(" [ FPSection { ";" FPSection } ] ")"
[ ":=" type ] .

FPSection = [ "VAR" ] IdentList ":=" type .

ident = letter { letter | digit } .

IdentList = ident { "," ident } .

type = BaseType | "LIST" | "OBJECT" | "NULL" .

BaseType = "INTEGER" | "REAL" | "BOOLEAN" | "STRING" | "DATE" .

StatementSequence = statement { ";" statement } .

statement = [ assignement | IfStatement | WhileStatement |

assignement = ident selector ":=" expression .

selector = [ ( "." ident | ( "[" expression "]" ) ) ] |
IfStatement = "IF" expression "THEN" StatementSequence
{ "ELSIF" expression "THEN" StatementSequence }
[ "ELSE" StatementSequence ] "END" .

WhileStatement = "WHILE" expression "DO" StatementSequence "END" .

GuiStatement = ( "ALERT" | "WRITE" | "WRITELN" | "MESSAGE" )
"(" expression ")" .

ReturnStatement = "RETURN" [ expression ] .

ProcedureCall = ident selector ActualParameters .

ActualParameters = "(" [ expression { "," expression } ] ")" .

expression = SimpleExpression
[ ( "=" | "+" | "-=" | "<" | "<=" | ">" | ">=\" ) SimpleExpression ] .

SimpleExpression = [ "+-" ] term { ( "+-" | "=*" | "MOD" | "&" ) term } .

term = factor { ( "+" | "/" | "MOD" | "+" ) factor } .

factor = boolean | string | number | conversion | GuiDialog |
"(" expression ")" | "+" factor |
ident selector [ ("INSTANCE OF" type ) | ActualParameters ] .

boolean = "TRUE" | "FALSE" .
string = {LetterDigit} .
LetterDigit = Letter | Digit .
Letter = "a"-"z" | "+" .
Digit = "0"-"9" .

number = integer | real .
integer = digit { digit } .
real = integer [ "." integer ] .

conversion = BaseType "(" expression ")" .

GuiDialog = ( ( "CONFIRM" | "ASK" ) "(" expression ")" ) |
( "FORM" ActualParameters ) .
B.2 XD Meta Language Operators

The XD operators presented in chapter 3 are specified as predefined methods in the XD meta language which can be invoked on an object of type OBJECT. For example, the assert operator can be evaluated as

```java
object.assert("phone", "34-78");
```

Table B.1 lists all predefined methods of the XD meta language.
<table>
<thead>
<tr>
<th>Method, Input Parameter(s)</th>
<th>Description, Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAlias(): STRING</td>
<td>alias of object: STRING</td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>getAttributes(): LIST</td>
<td>list of all attribute names: LIST</td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>new(alias)</td>
<td>creates a new meta object</td>
</tr>
<tr>
<td>alias of object: STRING</td>
<td></td>
</tr>
<tr>
<td>new(object)</td>
<td>creates a new meta object</td>
</tr>
<tr>
<td>object to be cloned: OBJECT</td>
<td>by cloning an existing one</td>
</tr>
<tr>
<td>retrieve(alias): OBJECT</td>
<td>object related to 'alias': OBJECT</td>
</tr>
<tr>
<td>alias of object: STRING</td>
<td></td>
</tr>
<tr>
<td>assert(attributeName, value)</td>
<td>asserts a new attribute/value pair to the meta object</td>
</tr>
<tr>
<td>name of attribute: STRING</td>
<td></td>
</tr>
<tr>
<td>value: any type</td>
<td></td>
</tr>
<tr>
<td>retract(attributeName)</td>
<td>deletes all values associated to attribute 'attributeName'</td>
</tr>
<tr>
<td>name of attribute: STRING</td>
<td></td>
</tr>
<tr>
<td>value: any type</td>
<td></td>
</tr>
<tr>
<td>retract(attributeName, value)</td>
<td>deletes the value 'value' of attribute 'attributeName'</td>
</tr>
<tr>
<td>name of attribute: STRING</td>
<td></td>
</tr>
<tr>
<td>value: any type</td>
<td></td>
</tr>
<tr>
<td>match(attributeName, values): BOOLEAN</td>
<td>matches all values associated to attribute 'attributeName' and returns them in the LIST 'values'; return value: TRUE if the operation was successful</td>
</tr>
<tr>
<td>name of attribute: STRING</td>
<td></td>
</tr>
<tr>
<td>value: any type</td>
<td></td>
</tr>
<tr>
<td>retrieveValue(attributeName): any type</td>
<td>returns the first value associated to attribute 'attributeName'</td>
</tr>
<tr>
<td>name of attribute: STRING</td>
<td></td>
</tr>
<tr>
<td>dress(type)</td>
<td>'dresses' the object with 'type'</td>
</tr>
<tr>
<td>type: OBJECT or STRING</td>
<td></td>
</tr>
<tr>
<td>insert(object)</td>
<td>inserts object 'object'</td>
</tr>
<tr>
<td>object: OBJECT or STRING</td>
<td></td>
</tr>
<tr>
<td>remove(object)</td>
<td>removes object 'object'</td>
</tr>
<tr>
<td>object: OBJECT or STRING</td>
<td></td>
</tr>
<tr>
<td>isMember(object): BOOLEAN</td>
<td>returns TRUE if object 'object' is a member</td>
</tr>
<tr>
<td>object: OBJECT or STRING</td>
<td></td>
</tr>
<tr>
<td>deleteGroup()</td>
<td>removes all members of an object group</td>
</tr>
<tr>
<td>members(): LIST</td>
<td>returns all members of an object group</td>
</tr>
</tbody>
</table>

Table B.1: Predefined XD Operators
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


A. Kobler and M. C. Norrie. OMS Java: An Open, Extensible Architecture for Advanced Application Systems such as GIS. In International Workshop on Emerging Technologies for GEO-Based Applications, Ascona, Switzerland, May 2000.


144


Curriculum Vitae

Name: Adrian Kurt Michael Kobler
Birthdate: February 25, 1964
Birthplace: Zürich, Switzerland
Citizenship: Zürich and Oberriet SG, Switzerland

1971 – 1972 Elementary School in Zürich, Switzerland
1972 – 1977 Elementary School in Küsnacht, Switzerland
1977 – 1983 High School in Zürich, Switzerland
1983 – 1984 High School in Oberägeri ZG, Switzerland
1984 Matura Type B
1984 – 1988 Study of Computer Science at the Swiss Federal Institute of Technology (ETH) in Zürich
1986 1. Pre-Diploma (Vordiplom) in Computer Science
1998 – 1993 Study of Computer Science and Economics at the University of Zürich
1993 Diploma (lic.oec.pupil, Dipl. Wirtschaftsinformatiker) in Computer Science and Economics, University of Zürich
1993 – 1994 Research Assistant at the IBM Research Laboratory, Rüschlikon, Switzerland
1994 – 1996 Kobler Information Engineering, Küsnacht, Switzerland
1996 – 2001 Research Assistant in the research group for Global Information Systems at ETH Zürich (Prof. Dr. Moira C. Norrie)